# 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.

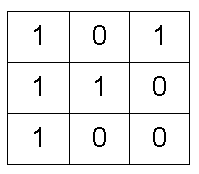
Same condition as above , the case where neighbour is not a paraent and visited already is the satisfactory condition for the cycle .

# Distance of Nearest Cell having 1

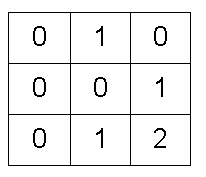
**Problem Statement**: Given a binary grid of N\*M. Find the distance of the nearest 1 in the grid for each cell.

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 ?

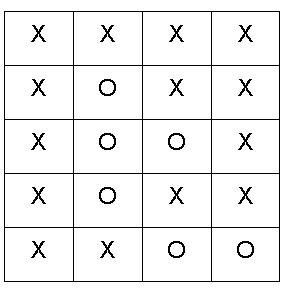
**Answer :** The idea is to cover only the steps which are at a distance of 1, which are the ones nothing but adjacent to that in 4 directions.

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

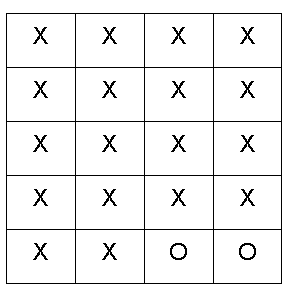
# Surrounded Regions | Replace O’s with X’s

**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’.

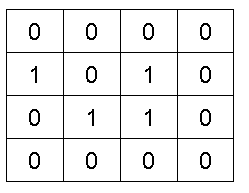
The intuition is that we start from boundary elements having ‘O’ and go through its neighboring Os in 4 directions and mark them as visited to avoid replacing them with ‘X’.

**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.

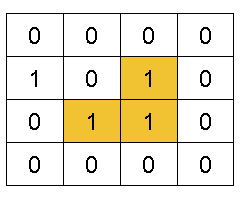
# Number of Enclaves

**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:

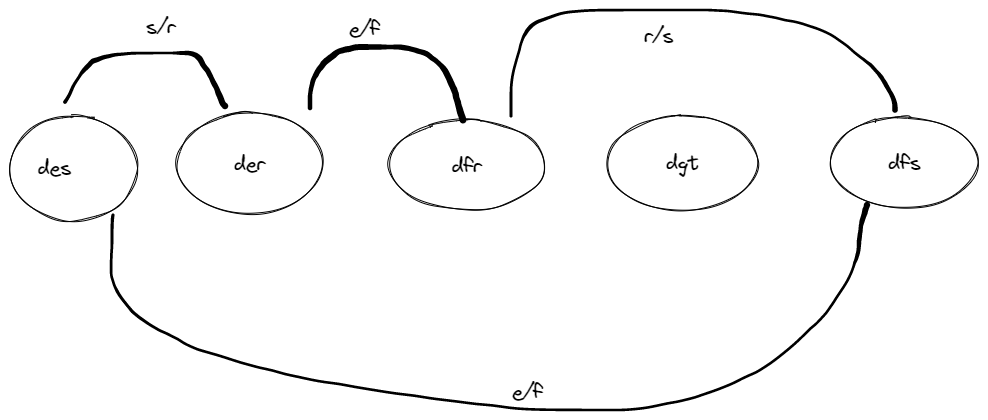
Input:

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

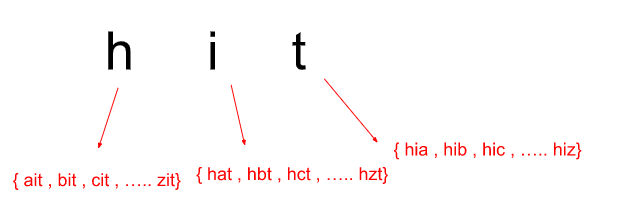
from "gedk" to "geek" is 2 i.e. "gedk" -> (replace ‘d’ by ‘e’) -> "geek"

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

Each of these strings are present in the wordList.

**Approach :**

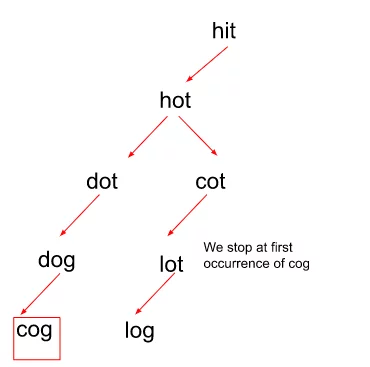
In Brute force, we just simply replace the startingWord character by character and then check whether the transformed word is present in the wordList. If a word is present in the wordList, we try replacing another character in that word by again following similar steps as above, in order to attain the 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.

# Word Ladder-II

Given two distinct words startWord and targetWord, and a list denoting wordList of unique words of equal lengths. Find all shortest transformation sequence(s) from startWord to targetWord. You can return them in any order possible.

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.
* Return an empty list if there is no such transformation sequence.

Examples:

Example 1:

Input:

startWord = "der", targetWord = "dfs",

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

Output:

[ [ “der”, “dfr”, “dfs” ], [ “der”, “des”, “dfs”] ]

Explanation:

The length of the smallest transformation sequence here is 3.

Following are the only two shortest ways to get to the targetWord from the startWord :

"der" -> ( replace ‘r’ by ‘s’ ) -> "des" -> ( replace ‘e’ by ‘f’ ) -> "dfs".

"der" -> ( replace ‘e’ by ‘f’ ) -> "dfr" -> ( replace ‘r’ by ‘s’ ) -> "dfs".

Example 2:

Input:

startWord = "gedk", targetWord= "geek"

wordList = {"geek", "gefk"}

Output:

[ [ “gedk”, “geek” ] ]

Explanation:

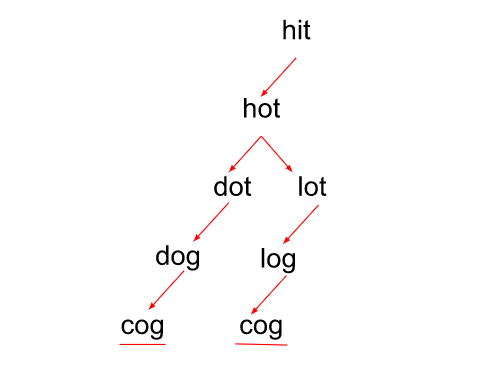
The length of the smallest transformation sequence here is 2.

Following is the only shortest way to get to the targetWord from the startWord :

"gedk" -> ( replace ‘d’ by ‘e’ ) -> "geek".

**Approach :**

Here we do not stop the traversal on the first occurrence of the targetWord, but rather continue it for as many occurrences of the word as possible as we need all the shortest possible sequences in order to reach the destination word.



Example : WordList = [“pat” , “bot” , “pot” , “poz” , “coz” ] , beginWord = “bat” , endWord = “coz”

The queue would look something like this.

{bat , bot , pot , poz , coz }

{bat , pat , pot , poz , coz }

{bat , bot , pot , poz }

{bat , pat , pot , poz }

{bat , bot , pot}

{bat , pat , pot}

{bat , bot}

{bat , pat}

{bat}

From {bat , pat} , last word is “pat” , if we mark the next transformed word “pot” as visited (removed from visited array ) , we would not see the next possible sequence

{bat , bot , pot} .

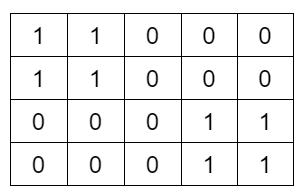
**Time Complexity :** From the above approach , it will be O(N \* M \* 26)

But , It cannot be predicted for this particular algorithm because there can be multiple sequences of transformation from startWord to targetWord depending upon the example

# Number of Distinct Islands

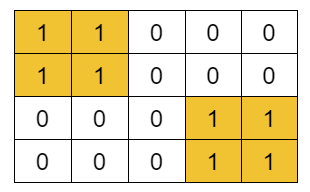
**Problem Statement:** Given a boolean 2D matrix grid of size N x M. You have to find the number of distinct islands where a group of connected 1s (horizontally or vertically) forms an island. Two islands are considered to be distinct if and only if one island is equal to another (not rotated or reflected).

**Input:**



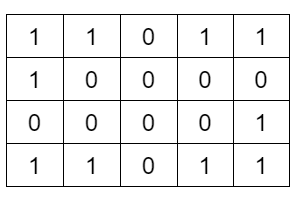
**Output:** 1

**Explanation:** Island at the top left corner is the same as the island at the bottom right corner.

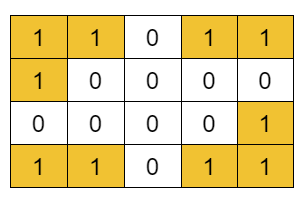


Example 2:

Input:



Output: 3



**Explanation:** Island at the top right corner is the same as the island at the bottom left corner , but the islands at top left and bottom right are distinct , because it is a rotation of the above one or viceversa.

**Approach :**

Depending on the shape of the island formed, we count the number of islands.

The question arises how to store these shapes?

We can store the shapes in a set data structure, then the set will return unique islands. We can store the coordinates in a vector or a list.

But how to figure out if the coordinates stored in the set data structure are identical? We can call one of the starting points a base, and subtract the base coordinates from the land’s coordinates **(Cell Coordinates – Base coordinates).**

Make sure to follow a particular traversal and a particular order pattern, so that list ordering remains the same for every cell.(This will be taken care automatically because either for BFS or DFS we are going to follow a pattern of the traversal which is consistent between the function calls )

**Time Complexity:** O(N x M x log(N x M)) , For the worst case, assuming all the pieces as land, the BFS function will be called for (N x M) nodes.

Set datastructure at max will store the complete grid, so it takes log(N x M) time.

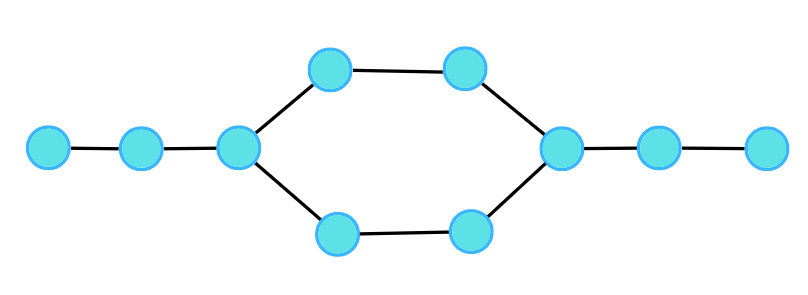
# Bipartite Graph | DFS Implementation

**Problem Statement:** Given an adjacency list of a graph adj of V no. of vertices having 0 based index. Check whether the graph is bipartite or not.

If we are able to colour a graph with two colours such that no adjacent nodes have the same colour, it is called a bipartite graph.

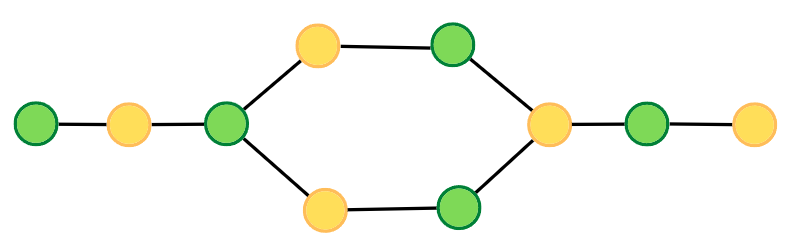
**Example 1:**

**Input:**



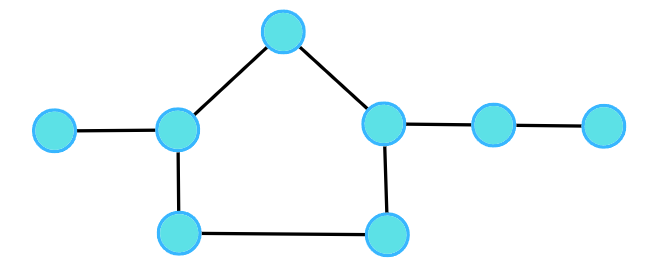
**Output:**1

**Explanation:**

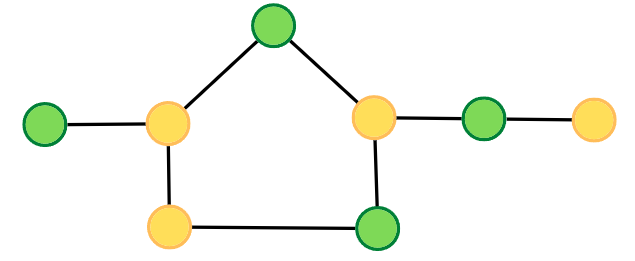
****

**Example 2:**

**Input:**



**Output:**0

****

**Approach :**

A bipartite graph is a graph which can be coloured using 2 colours such that no adjacent nodes have the same colour. Any linear graph with no cycle is always a bipartite graph. With a cycle, any graph with an even cycle length can also be a bipartite graph. So, any graph with an odd cycle length can never be a bipartite graph.

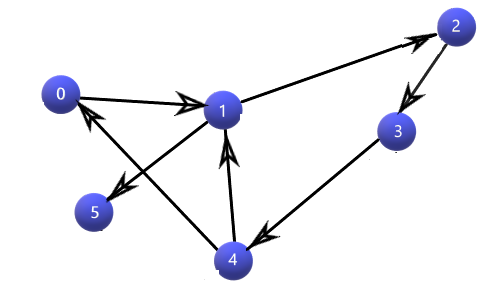
The intuition is the brute force of filling colours using any traversal technique, just make sure no two adjacent nodes have the same colour. If at any moment of traversal, we find the adjacent nodes to have the same colour, it means that there is an odd cycle, or it cannot be a bipartite graph.

Instead of the visited array we can maintain a color array with all initialized to -1 indicating that all are unvisted at the first , we can fill the alternative colors 0 or 1 using 1-col .

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

# Detect a Cycle in Directed Graph using DFS

**Problem Statement:** Given a Directed Graph with V vertices and E edges, check whether it contains any cycle or not.

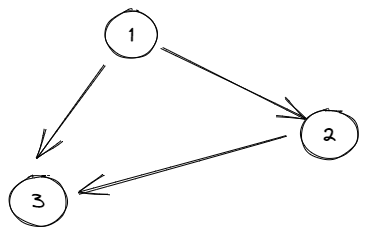


Here we can see that there are 2 cycles exist 0 -> 4 -> 0 and 1-> 4 -> 1

**Question :** Why does the normal DFS approach for directed graph does not work in the case of undirected ?

**Answer :**

Let us take the below scenario dfs(1 , -1 ) wil be the first call , -1 represents the parent , it will be marking both the vertices 2 , and 3 are visited has neighbours {2 , 3 } first 2 will be explored and when we come back to exploring the 3 we will have it already visited and also it is not a parent 3 != -1 , we naturally return true incase of undirected for this scenario.but we should not here , as it’s direction is different and there is no edge from 3 to 1.



**Approach :** To find cycle in a directed graph we can use the Depth First Traversal (DFS) technique. It is based on the idea that there is a cycle in a graph only if there is a **back edge** [i.e., a node points to one of its ancestors] present in the graph.

To detect a back edge, we need to keep track of the nodes visited till now and the nodes that are in the current recursion stack [i.e., the current path that we are visiting]. If during recursion, we reach a node that is already in the recursion stack, there is a cycle present in the graph.

**While returning from the recursion call, unmark the current node from the recursion stack, to represent that the current node is no longer a part of the path being traced.**

That means after having explored all the possible nodes in depth for a current node consider marking as unvisited , so that here in the above example 3 will be marked as unvisited and therefore , from 1 again we are exploring would have this in the path again freshly.

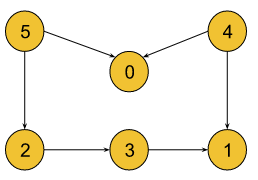
**Time Complexity :**

O(V + E), since in its whole, it is a DFS implementation, V – vertices; E – edges;

# Topological Sort Algorithm

**Problem Statement:** Given a Directed Acyclic Graph (DAG) with V vertices and E edges, Find any Topological Sorting of that Graph.

Note: In topological sorting, node u will always appear before node v if there is a directed edge from node u towards node v(u -> v).



Output: 5, 4, 2, 3, 1, 0

Explanation: A graph may have multiple topological sortings.

The result is one of them. The necessary conditions

for the ordering are:

* According to edge 5 -> 0, node 5 must appear before node 0 in the ordering.
* According to edge 4 -> 0, node 4 must appear before node 0 in the ordering.
* According to edge 5 -> 2, node 5 must appear before node 2 in the ordering.
* According to edge 2 -> 3, node 2 must appear before node 3 in the ordering.
* According to edge 3 -> 1, node 3 must appear before node 1 in the ordering.
* According to edge 4 -> 1, node 4 must appear before node 1 in the ordering.

The above result satisfies all the necessary conditions.

[4, 5, 2, 3, 1, 0] is also one such topological sorting

that satisfies all the conditions.

**Question :** why the normal DFS call does not work ?

In the normal DFS call what ever that comes first or the starting node here 0 , we are pushing from there , but if we see here there are two nodes coming into the 0 , and no node coming out from 0.

So , we would need to first let all the childs travsered and then push the parent , in that cases if we just reverse the order of the visit we will be able to get the correct order.

**Approach :** We will be solving it using the DFS traversal technique. 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.

Since we are inserting the nodes into the stack after the completion of the traversal, we are making sure, there will be no one who appears afterward but may come before in the ordering as everyone during the traversal would have been inserted into the stack.

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

# Kahn’s Algorithm | Topological Sort Algorithm | BFS

**Problem Statement:** Given a Directed Acyclic Graph (DAG) with V vertices and E edges, Find any Topological Sorting of that Graph.

**Approach :**

First, we will calculate the indegree of each node and store it in the indegree array.

Initially, there will be always at least a single node whose indegree is 0 (why ? as it is the DAG due to its acyclic nature , there should be atleast one node whose indegree is 0 and also if we could remove that node it would definitely make the indegree of another node to 0, so it guaranteed that every level we would find a node with indegree 0 to proceed for the BFS ) . So, we will push the node(s) with indegree 0 into the queue.

we will pop a node from the queue including the node in our answer array, and for all its adjacent nodes, we will decrease the indegree of that node by one.

After that, if for any node the indegree becomes 0, we will push that node again into the queue.

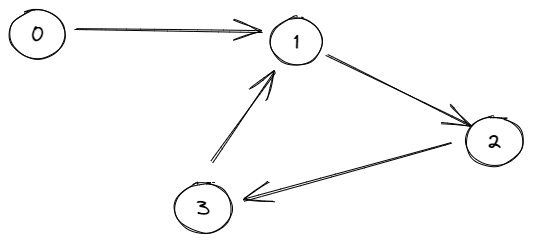
**Time Complexity** : O(V+E)

# Cycle Detection in Directed Graph (BFS)

**Problem Statement:** Given a Directed Graph with V vertices and E edges, check whether it contains any cycle or not.

**Approach :** We shall start with the assumption that cycle does not exists in the graph (DAG) and will start applying the Khanns algorithm , we shall at somepoint come to the situation of queue becoming empty and there are no 0 indegree nodes , in essence we will be not able to cover all the nodes.

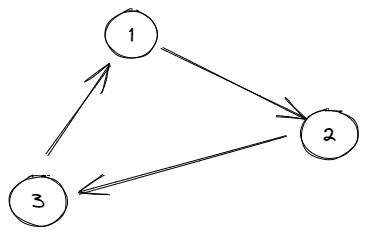
Example 1 :



Here at first only 0 will have indegree of 0 and we shall push to the queue and after popping that out and reducing the indegree of neighbour node 1 , we shall still see that 1 has an indegree of 1 because of the back edge from 3 coming into 1.

So we can not proceed futher with the queue and return cycle found.

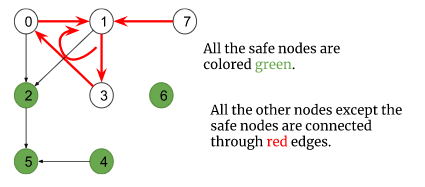
Similarily , if we take the below example , we shall see that at the beginning itself the indegree is none of the nodes is 0 , so can not proceed further and return cycle found.



**Time Complexity** : O(V+E) same as above.

# Find Eventual Safe States – DFS

**Problem Statement:** A directed graph of V vertices and E edges is given in the form of an adjacency list adj. Each node of the graph is labeled with a distinct integer in the range 0 to V – 1. A node is a terminal node if there are no outgoing edges. A node is a safe node if every possible path starting from that node leads to a terminal node. You have to return an array containing all the safe nodes of the graph. The answer should be sorted in ascending order.



**Approach :**

A terminal node is a node without any outgoing edges(i.e outdegree = 0). Now a node is considered to be a safe node if all possible paths starting from it lead to a terminal node. Here we need to find out all safe nodes and return them sorted in ascending order.

If we closely observe, all possible paths starting from a node are going to end at some terminal node unless there exists a cycle and the paths return back to themselves.

Though node 2 is connected to the cycle, the edge is directed outwards the cycle and all the paths starting from it lead to the terminal node 5. So, it is a safe node and the rest of the nodes (4, 5, 6) are safe nodes as well.

The intuition is to figure out the nodes which are either a part of a cycle or incoming to the cycle(node 7). We can do this easily using the cycle detection technique that was used previously to detect cycles in a directed graph (using DFS).

If there are no further nodes to visit, we will mark the node as safe ( when we unmark the onpath array in above approach ) .

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

# Find Eventual Safe States – BFS

**Problem Statement:** A directed graph of V vertices and E edges is given in the form of an adjacency list adj. Each node of the graph is labeled with a distinct integer in the range 0 to V – 1. A node is a terminal node if there are no outgoing edges. A node is a safe node if every possible path starting from that node leads to a terminal node. You have to return an array containing all the safe nodes of the graph. The answer should be sorted in ascending order.

**Approach :** The node with outdegree 0 is considered to be a terminal node but the topological sort algorithm deals with the indegrees of the nodes. So, to use the topological sort algorithm, we will reverse every edge of the graph. Now, the nodes with indegree 0 become the terminal nodes. After this step, we will just follow the topological sort algorithm as it is.

Why is it necessarily work ? The outdegrees of a given graph will match with the indegrees of its reversed graph.

**Time Complexity:** O(V+E)+O(N\*logN)

# Course Schedule I

**Problem Statement :** There are a total of N tasks, labeled from 0 to N-1. Some tasks may have prerequisites, for example, to do task 0 you have to first complete task 1, which is expressed as a pair: [0, 1]

Given the total number of tasks N and a list of prerequisite pairs P, find if it is possible to finish all tasks.

**Approach :** If we can think of each task as a node and every pair as a directed edge between those two nodes, the whole problem becomes a graph problem.

Simply , we need to identify a graph as a directed acyclic graph.

**TimeComplexity :** O(V+E)

# Course Schedule II

There are a total of n tasks you have to pick, labeled from 0 to n-1. Some tasks may have prerequisites tasks, for example, to pick task 0 you have to first finish tasks 1, which is expressed as a pair: [0, 1]

Given the total number of n tasks and a list of prerequisite pairs of size m. Find the order of tasks you should pick to finish all tasks.

Note: There may be multiple correct orders, you need to return one of them. If it is impossible to finish all tasks, return an empty array.

Approach : Same as above.

**TimeComplexity :** O(V+E)

# Alien Dictionary

**Problem Statement:** Given a sorted dictionary of an alien language having N words and k starting alphabets of a standard dictionary. Find the order of characters in the alien language.

Note: Many orders may be possible for a particular test case, thus you may return any valid order.

Example

Input: N = 5, K = 4

dict = {"baa","abcd","abca","cab","cad"}

Output: b d a c

Explanation:

We will analyze every consecutive pair to find out the order of the characters.

The pair “baa” and “abcd” suggests ‘b’ appears before ‘a’ in the alien dictionary.

The pair “abcd” and “abca” suggests ‘d’ appears before ‘a’ in the alien dictionary.

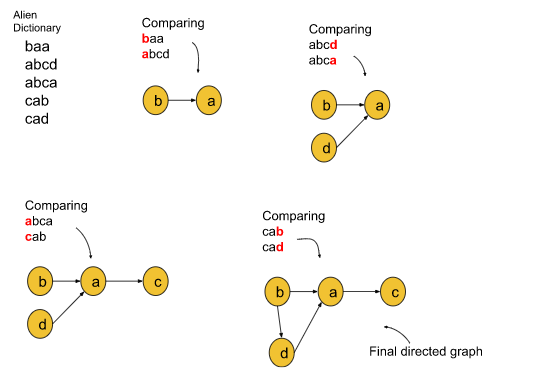
The pair “abca” and “cab” suggests ‘a’ appears before ‘c’ in the alien dictionary.

The pair “cab” and “cad” suggests ‘b’ appears before ‘d’ in the alien dictionary.

So, [‘b’, ‘d’, ‘a’, ‘c’] is a valid ordering.

**Approach :**

we need not check every pair of words rather we will just check the consecutive pair of words in the dictionary. Comparing each pair of consecutive words in the dictionary, we can construct a directed graph like the following:



Now, we have successfully reduced the problem to a known directed graph problem. If we look at the problem from the graph point of view, we just need to find out the linear ordering of the nodes of the directed graph. And we can do this easily using the topological sort algorithm.

To further simplify the problem, we will denote the alphabet with numbers like: ‘a’ with 0, ‘b’ with 1, ‘c’ with 2, and so on.

**The follow-up question for the interview:**

If every character matches and the largest word appears before the shortest word: For example, if “abcd” appears before “abc”, we can say the ordering is not possible.

If there exists a cyclic dependency between the characters: For example, in the dictionary: dict: {“abc”, “bat”, “ade”} there exists a cyclic dependency between ‘a’ and ‘b’ because the dictionary states ‘a’ < ‘b’ < ‘a’, which is not possible.

**Time Complexity:** O(N\*len)+O(K+E), where N is the number of words in the dictionary, ‘len’ is the length up to the index where the first inequality occurs, K = no. of nodes, and E = no. of edges.

# Shortest Path in Undirected Graph with unit distance

Given an Undirected Graph having unit weight, find the shortest path from the source to all other nodes in this graph. In this problem statement, we have assumed the source vertex to be ‘0’. If a vertex is unreachable from the source node, then return -1 for that vertex.

**Approach :** We will calculate the shortest path in an undirected graph having unit weights by using the Breadth First Search. BFS is a traversal technique where we visit the nodes level-wise, i.e., it visits the same level nodes simultaneously, and then moves to the next level.

we create a dist array of size N initialized with a very large number which can never be the answer to indicate that initially, all the nodes are untraversed.

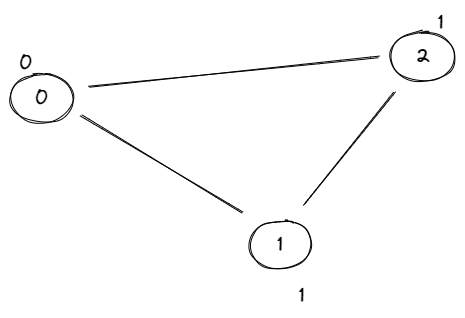
In every iteration, pick up the front() node in the queue , and then traverse for its adjacent nodes. For every adjacent node, we will relax the distance to the adjacent node if (dist[node] + 1 < dist[adjNode] ).

dist[node] : Distance of current node from the src .

dist[adjNode] : Distance of adjNode from the src .

we shall check if this condition relaxes the distance , if so we shall push the updated distance to the queue which will be used further by its adjacent nodes.

So , actually we are not using the visited array.



We can see that in the above graph , to reach 2 node , we shall have two paths from 0 node (0 -> 2 , 0 ->1 -> 2 ) , but the latter takes 2 steps as updated in the top of the nodes represents the corresponding distances.

**Here we don’t need to store distance information to the queue , because it can be found out from the distance array directly.**

Time Complexity : O(V + E ) ,as it is pure BFS algo.

# Shortest Path in Directed Acyclic Graph Topological Sort

Given a DAG, find the shortest path from the source to all other nodes in this DAG. In this problem statement, we have assumed the source vertex to be ‘0’. You will be given the weighted edges of the graph.

**Intuition:** Finding the shortest path to a vertex is easy if you already know the shortest paths to all the vertices that can precede it. Processing the vertices in topological order ensures that by the time you get to a vertex, you’ve already processed all the vertices that can precede it.

In this approach, we traverse the nodes sequentially according to their reachability from the source.

Dijkstra’s algorithm is necessary for graphs that can contain cycles because they can’t be topologically sorted. In other cases, the topological sort would work fine as we start from the first node, and then move on to the others in a directed manner.

**Time Complexity:** O(N+M) {for the topological sort} + O(N+M) {for relaxation of vertices, each node and its adjacent nodes get traversed}

# Print Shortest Path – Dijkstra’s Algorithm