28/02/2024

# GRAPHS CLASS - 6

## 1. Number of Provinces (Leetcode-547)

PROBLEM STATEMENT:

There are **n** cities. Some of them are connected, while some are not. If city **a** is connected directly with city **b**, and city **b** is connected directly with city **c**, then city **a** is connected indirectly with city **c**.

A **province** is a group of directly or indirectly connected **cities** and no other cities outside of the group. You are given an **n x n** matrix **isConnected** where **isConnected[i][j] = 1** if the **ith** city and the **jth** city are directly connected, and **isConnected[i][j] = 0** otherwise.

DISCONNETED NODE

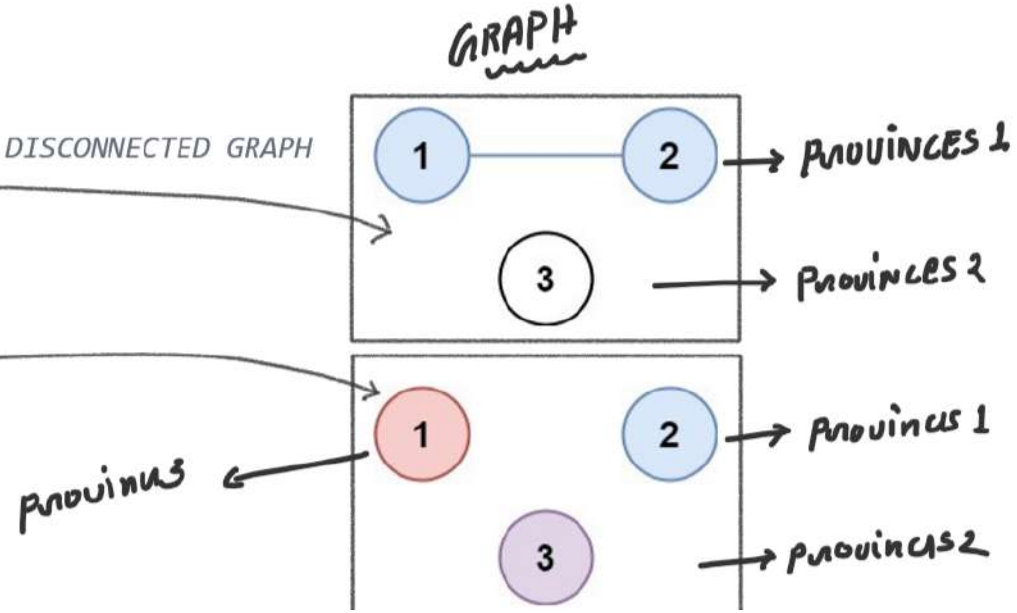
CONNECTED NODE

**Return the total number of provinces.**

ACTUAL ME **PROVINCES** KA MATLB HAI **COMPONENTS** FIND KARNA FROM DISCONNECTED GRAPH

**Example 1:**
Input: isConnected = [[1,1,0],[1,1,0],[0,0,1]]
Output: 2

**Example 2:**
Input: isConnected = [[1,0,0],[0,1,0],[0,0,1]]
Output: 3



Example 1:
Input: isConnected = [[1,1,0],[1,1,0],[0,0,1]]
Output: 2

jth

$$\begin{array}{c|ccc} \text{ith} & 0 & 1 & 2 \\ \hline 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 2 & 0 & 0 & 1 \end{array}$$

NXN

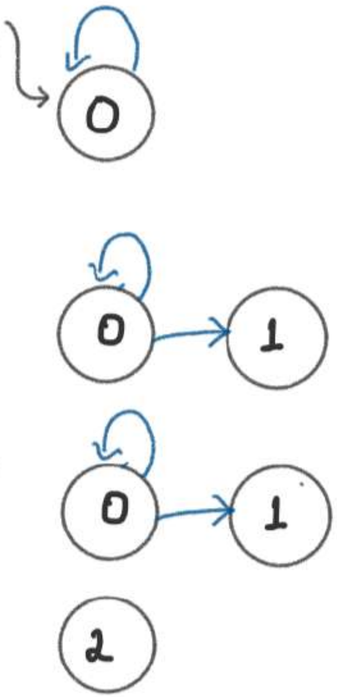
According to Given matrix

index = NODE

[0][0] = 1

[0][1] = 1

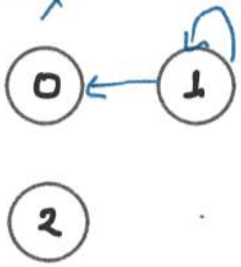
[0][2] = 0



[1][0] = 1

[1][1] = 1

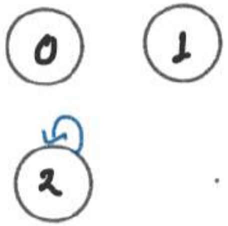
[1][2] = 0



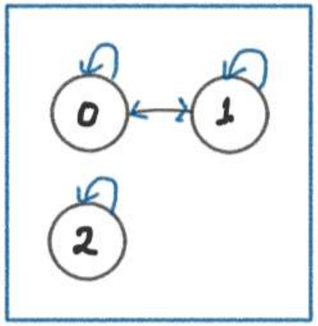
[2][0] = 0

[2][1] = 0

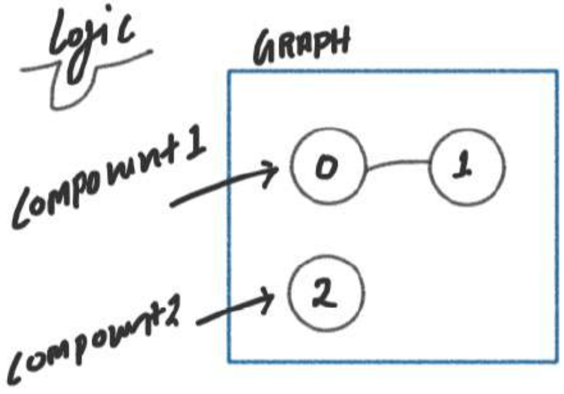
[2][2] = 1



Final Graph



## Logic



⇒ Find No. of provinces/components using DFS Algorithm

```
Count [0]
For( 0 ⟶ N-1 )
{
    if (!visited[i]) {
        ↳ DFS call
        ↳ count ++
    }
}
```

NbrNode Jth

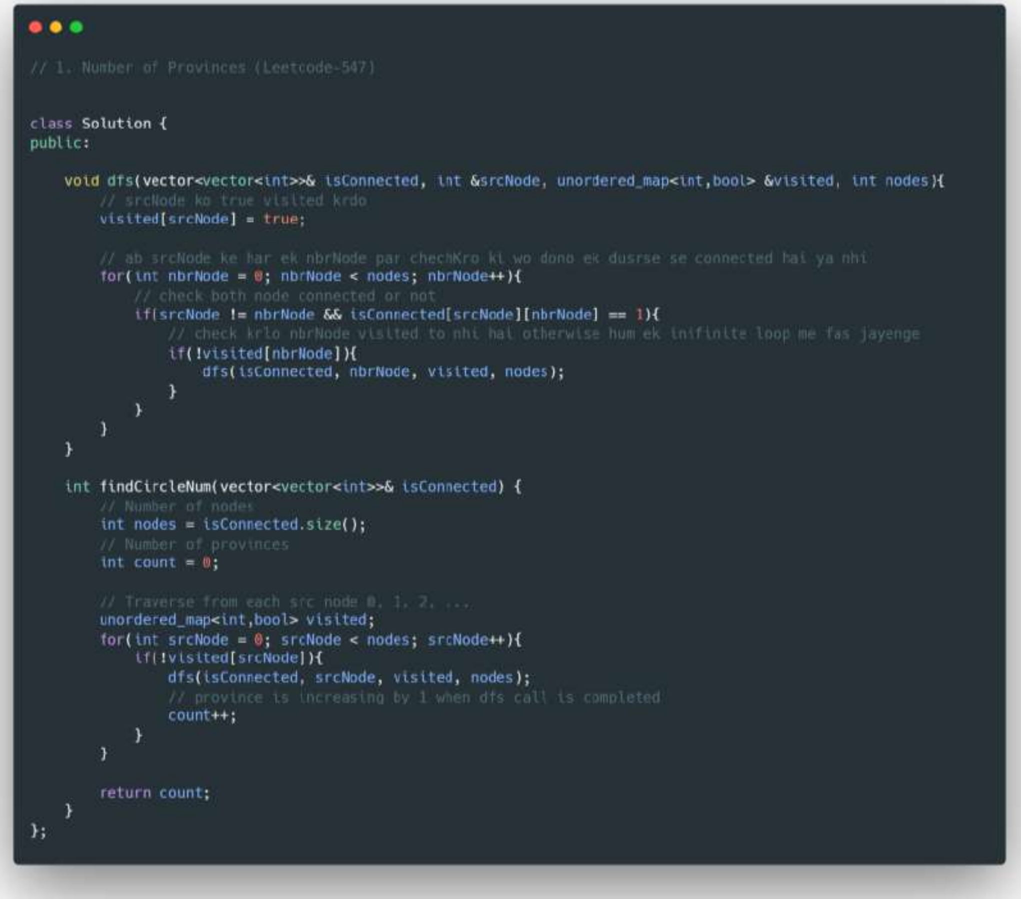
ith SRC NODE

$$\begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 2 & 0 & 0 & 1 \end{array}$$

NxN

↳ NODES ⇒ N = 3

0 1 2

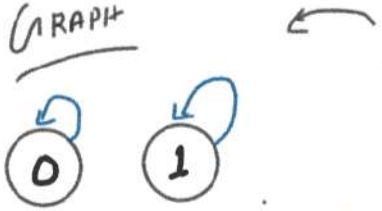


```
// 1. Number of Provinces (Leetcode-547)


class Solution {
public:

    void dfs(vector<vector<int>>& isConnected, int &srcNode, unordered_map<int,bool> &visited, int nodes){
        // srcNode ko true visited krdo
        visited[srcNode] = true;

        // ab srcNode ke har ek nbrNode par chechKro ki wo dono ek dusrse se connected hai ya nhi
        for(int nbrNode = 0; nbrNode < nodes; nbrNode++){
            // check both node connected or not
            if(srcNode != nbrNode && isConnected[srcNode][nbrNode] == 1){
                // check krlo nbrNode visited to nhi hai otherwise hum ek inifinite loop me fas jayenge
                if(!visited[nbrNode]){
                    dfs(isConnected, nbrNode, visited, nodes);
                }
            }
        }
    }

    int findCircleNum(vector<vector<int>>& isConnected) {
        // Number of nodes
        int nodes = isConnected.size();
        // Number of provinces
        int count = 0;

        // Traverse from each src node 0, 1, 2, ...
        unordered_map<int,bool> visited;
        for(int srcNode = 0; srcNode < nodes; srcNode++){
            if(!visited[srcNode]){
                dfs(isConnected, srcNode, visited, nodes);
                // province is increasing by 1 when dfs call is completed
                count++;
            }
        }

        return count;
    }
};
```

Why usd this condition?

if (srcNode != nbrNode)

GRAPH



$$\begin{array}{c|cc} & 0 & 1 \\ \hline 0 & 1 & 0 \\ 1 & 0 & 1 \end{array}$$

NXN

MATRIX

N = 2

nodes = 0, 1

when srcNode == nbrNode
$\quad\quad\quad\quad\quad\;\; 0 \quad\quad\quad\quad\;\; 0$

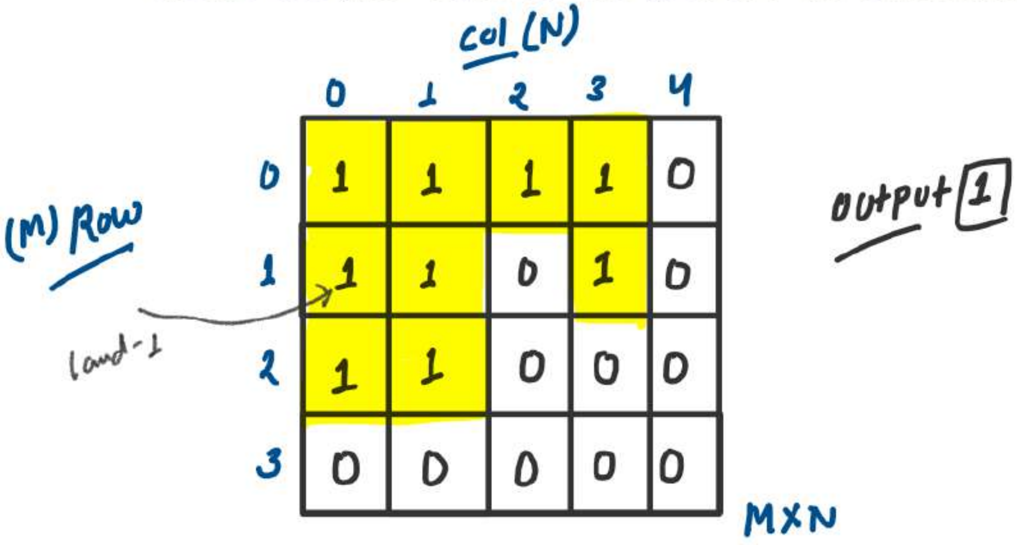
↳ iska mtlb hum DFS call one same node par dobara से nahi karna chahte hai. → DFS call only = 2 hogi in this case

## 2. Number of Islands (Leetcode-200)

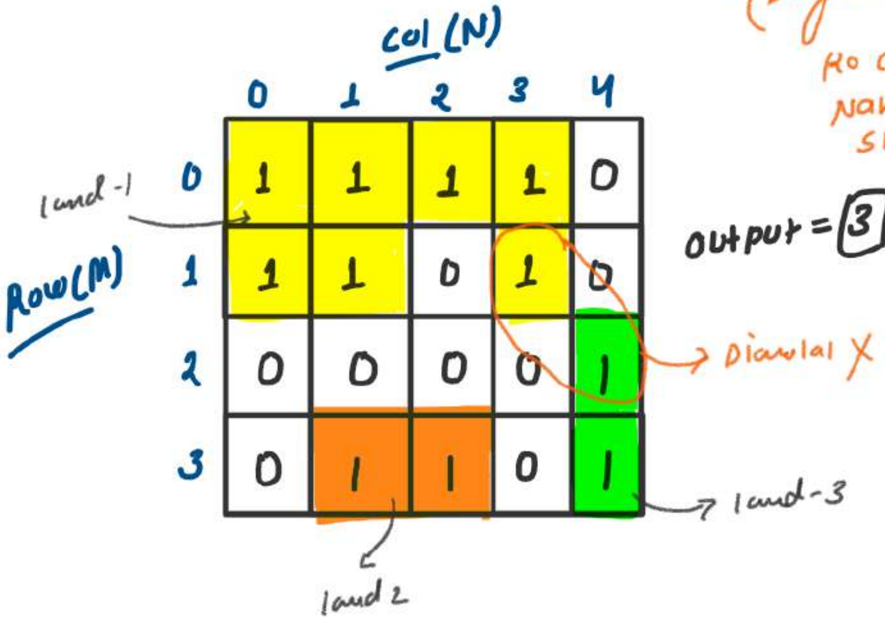
***Problem Statement:***
*Given an `m x n` 2D binary **grid** which represents a map of **'1's (land)** and **'0's (water)**, return the number of **islands**.*

***Important Line:***
*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.*

Mtlb (Diagonally lands Ko CONNECT Nahi kar sakta hun)



| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | 0 |
| 2 | 1 | 1 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 |



| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | 0 |
| 2 | 0 | 0 | 0 | 0 | 1 |
| 3 | 0 | 1 | 1 | 0 | 1 |

**Obsenbation**

0 → water

1 → land

[GRID] Row x Col

$\boxed{1} \leftrightarrow \boxed{1}$ Horizontal

and

$\boxed{1}$
↕
$\boxed{1}$ vertical

GRID

|   | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | 0 |
| 2 | 0 | 0 | 0 | 0 | 1 |
| 3 | 0 | 1 | 1 | 0 | 1 |

MxN

output = $\boxed{3}$

{ Yamha par vhi Ham No. of Components Hi find Kane Hai }

We Have 4 Move from Each Cell of Grid jamha par Hum stand Kar Rhe Hai

1 TopMove

2 Bottom Move

3 Right Move

4 Left Move

$$\begin{tikzcd}
 & f(i-1, j) & \\
f(i, j-1) & {[i][j]} \arrow[l] \arrow[r] \arrow[u] \arrow[d] & f(i, j+1) \\
 & (i+1, j) &
\end{tikzcd}$$

|    | T  | R | B | L  |
|----|----|---|---|----|
| dx | -1 | 0 | 1 | 0  |
| dy | 0  | 1 | 0 | -1 |

Visiting Graph

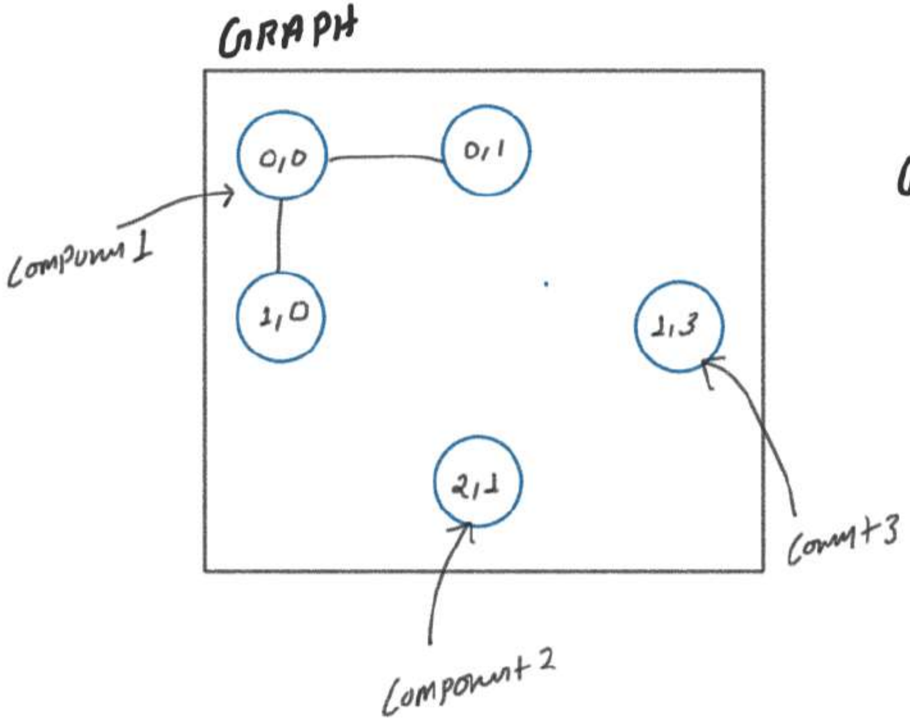
**NODE = CEll of GRID**

NODE 1 = [0][0]

NODE 2 = [0][1]

⋮

NODEN = [N-1][M-1]



| Grid | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 |
| 2 | 0 | 1 | 0 | 0 |

N x M

Total component / Islands = 3 Ans

solve using BFS

visited

|   | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | (0,0)<br>~~F~~ T | (0,1)<br>F | (0,2)<br>F | (0,3)<br>F |
| 1 | (1,0)<br>F | (1,1)<br>F | (1,2)<br>F | (1,3)<br>F |
| 2 | (2,0)<br>F | (2,1)<br>F | (2,2)<br>F | (2,3)<br>F |

initial state

N×M

Grid

|   | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | SRC<br>1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 |
| 2 | 0 | 1 | 0 | 0 |

N×M

Count

|   |
|---|
| 0 |

Queue

|   |   |
|---|---|
| (0,0) |   |

initial state

## Itenation 1

BSF call $\boxed{0,0}$

STEP1 get frontNode and pop it

(0,0)

STEP2 find safe move $(\underset{x}{T}, \underset{✓}{R}, \underset{✓}{B}, \underset{x}{L})$

$\boxed{R} \Rightarrow (0,1)$

$\boxed{B} \Rightarrow (1,0)$

STEP3 BFS call computed so count incremented by 1

Count $\boxed{0}$

visited

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | (0,0)<br>~~F~~ T | (0,1)<br>~~F~~ T | (0,2)<br>F | (0,3)<br>F |
| 1 | (1,0)<br>~~F~~ T | (1,2)<br>F | (1,2)<br>F | (1,3)<br>F |
| 2 | (2,0)<br>F | (2,1)<br>F | (2,2)<br>F | (2,3)<br>F |

N x M

Grid

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | SRC<br>1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 |
| 2 | 0 | 1 | 0 | 0 |

N x M

initial state

Queue

| | | | |
|---|---|---|---|
| ~~(0,0)~~ | (0,1) | (1,0) | |

## Iteration 1.1

BFS call (0,0)

STEP1 get frontNode and pop it

(0,1)

STEP2 find safe move (T, R, B, L)
x x x x

No safe node from (0,1)

STEP3 BFS call completed so count incremented by 1

| visited | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | (0,0)<br>~~F~~ T | (0,1)<br>~~F~~ T | (0,2)<br>F | (0,3)<br>F |
| 1 | (1,0)<br>~~F~~ T | (1,2)<br>F | (1,2)<br>F | (1,3)<br>F |
| 2 | (2,0)<br>F | (2,1)<br>F | (2,2)<br>F | (2,3)<br>F |

N x M

| Grid | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 |
| 2 | 0 | 1 | 0 | 0 |

N x M

Count 0

Queue

| | | |
|---|---|---|
| ~~(0,1)~~ | (1,0) | |

## Iteration No2

BFS call $\boxed{0,0}$

STEP1 get frontNode and pop it

(1,0)

STEP2 find safe move (T, R, B, L)
x x x x

No safe move from $\boxed{1,0}$

STEP3 BFS call completed so count incremented by 1

Now Queue is Empty so Incremented count by 1

Count $\boxed{\cancel{0}\,1}$

visited

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | (0,0)<br>~~F~~ T | (0,1)<br>~~F~~ T | (0,2)<br>F | (0,3)<br>F |
| 1 | (1,0)<br>~~F~~ T | (1,2)<br>F | (1,2)<br>F | (1,3)<br>F |
| 2 | (2,0)<br>F | (2,1)<br>F | (2,2)<br>F | (2,3)<br>F |

NxM

Grid

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 |
| 2 | 0 | 1 | 0 | 0 |

NxM

Queue ~~(1,0)~~ |

# $\boxed{\text{Iteration 2}}$

BFS call $\boxed{1,3}$

STEP1 get frontNode and pop it

(1,3)

STEP2 find safe move (T, R, B, L)
x x x x

No safe mode from $\boxed{1,3}$

STEP3 BFS call completed
so count incrementing 1

Now Queue is Empty
so Increment the
Count by 1

Count $\boxed{\cancel{1}\ 2}$

visited

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | (0,0)<br>~~F~~ T | (0,1)<br>~~F~~ T | (0,2)<br>F | (0,3)<br>F |
| 1 | (1,0)<br>~~F~~ T | (1,2)<br>F | (1,2)<br>F | (1,3)<br>~~F~~ T |
| 2 | (2,0)<br>F | (2,1)<br>F | (2,2)<br>F | (2,3)<br>F |

NxM

Grid

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 |
| 2 | 0 | 1 | 0 | 0 |

NxM

Queue | ~~(1,3)~~ \| |

# Iteration 3

BFS call [2,1]

STEP1 get frontNode and pop it
(2,1)

STEP2 find safe move (T, R, B, L)
x x x x

no safe move from [2,1]

STEP3 BFS call completed
so count incremented by 1

Now Queue is Empty
so Increment the
count by 1

Count ~~2~~ 3

| visited | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | (0,0)<br>~~F~~ T | (0,1)<br>~~F~~ T | (0,2)<br>F | (0,3)<br>F |
| 1 | (1,0)<br>~~F~~ T | (1,2)<br>F | (1,2)<br>F | (1,3)<br>~~F~~ T |
| 2 | (2,0)<br>F | (2,1)<br>~~F~~ T | (2,2)<br>F | (2,3)<br>F |

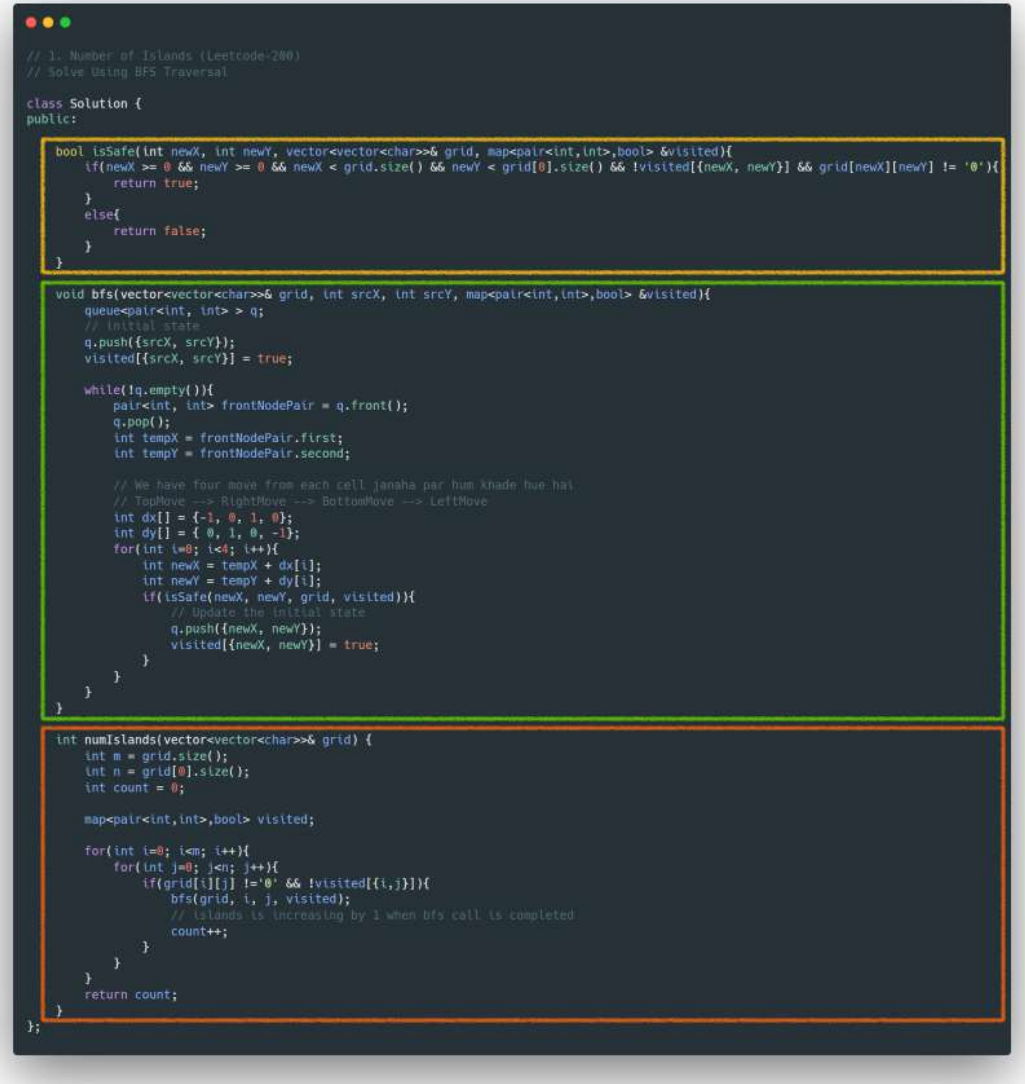
NxM

| Grid | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 |
| 2 | 0 | 1 | 0 | 0 |

NxM

Queue ~~(2,1)~~

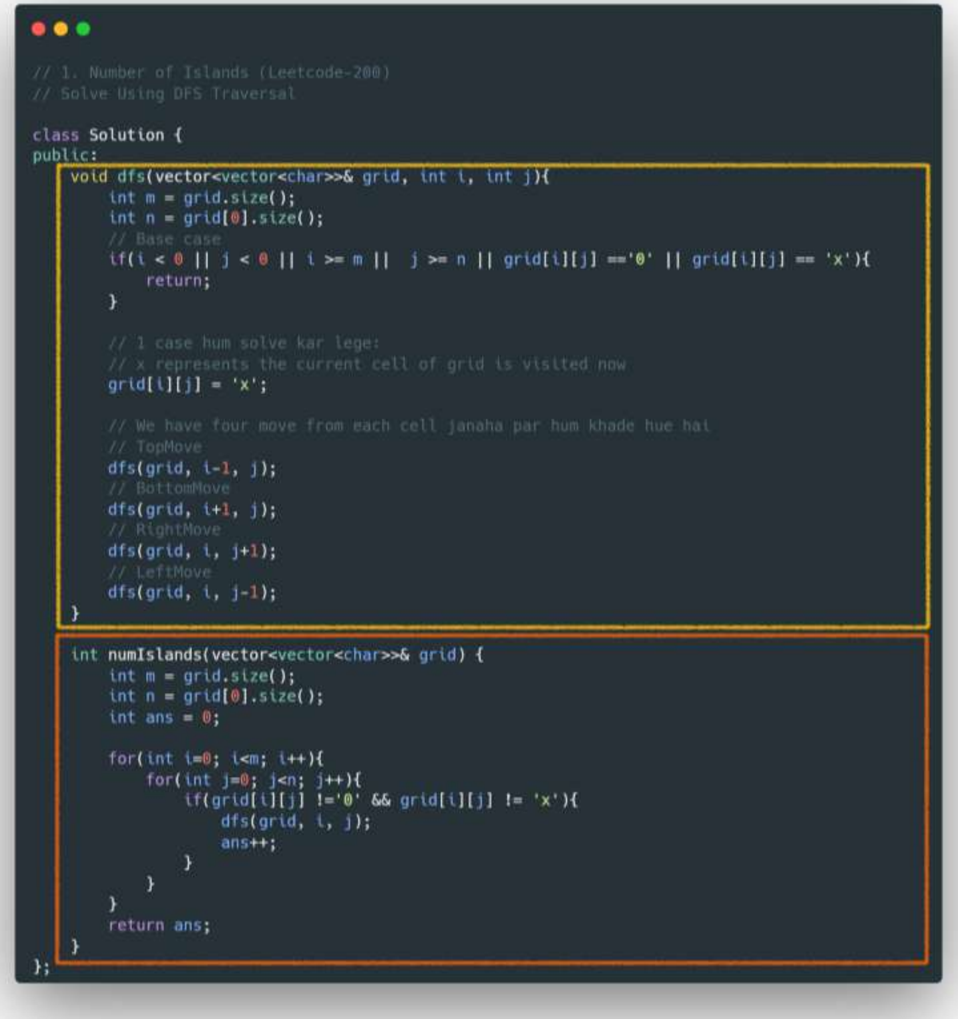
Output = 3



```
// 1. Number of Islands (Leetcode-200)
// Solve Using BFS Traversal

class Solution {
public:
    bool isSafe(int newX, int newY, vector<vector<char>>& grid, map<pair<int,int>,bool> &visited){
        if(newX >= 0 && newY >= 0 && newX < grid.size() && newY < grid[0].size() && !visited[{newX, newY}] && grid[newX][newY] != '0'){
            return true;
        }
        else{
            return false;
        }
    }

    void bfs(vector<vector<char>>& grid, int srcX, int srcY, map<pair<int,int>,bool> &visited){
        queue<pair<int, int> > q;
        // initial state
        q.push({srcX, srcY});
        visited[{srcX, srcY}] = true;

        while(!q.empty()){
            pair<int, int> frontNodePair = q.front();
            q.pop();
            int tempX = frontNodePair.first;
            int tempY = frontNodePair.second;

            // We have four move from each cell janaha par hum khade hue hai
            // TopMove --> RightMove --> BottomMove --> LeftMove
            int dx[] = {-1, 0, 1, 0};
            int dy[] = { 0, 1, 0, -1};
            for(int i=0; i<4; i++){
                int newX = tempX + dx[i];
                int newY = tempY + dy[i];
                if(isSafe(newX, newY, grid, visited)){
                    // Update the initial state
                    q.push({newX, newY});
                    visited[{newX, newY}] = true;
                }
            }
        }
    }

    int numIslands(vector<vector<char>>& grid) {
        int m = grid.size();
        int n = grid[0].size();
        int count = 0;

        map<pair<int,int>,bool> visited;

        for(int i=0; i<m; i++){
            for(int j=0; j<n; j++){
                if(grid[i][j] !='0' && !visited[{i,j}]){
                    bfs(grid, i, j, visited);
                    // islands is increasing by 1 when bfs call is completed
                    count++;
                }
            }
        }
        return count;
    }
};
```

Time and space compluxity

= ?



```cpp
// 1. Number of Islands (Leetcode-200)
// Solve Using DFS Traversal

class Solution {
public:
    void dfs(vector<vector<char>>& grid, int i, int j){
        int m = grid.size();
        int n = grid[0].size();
        // Base case
        if(i < 0 || j < 0 || i >= m ||  j >= n || grid[i][j] =='0' || grid[i][j] == 'x'){
            return;
        }

        // 1 case hum solve kar lege:
        // x represents the current cell of grid is visited now
        grid[i][j] = 'x';

        // We have four move from each cell janaha par hum khade hue hai
        // TopMove
        dfs(grid, i-1, j);
        // BottomMove
        dfs(grid, i+1, j);
        // RightMove
        dfs(grid, i, j+1);
        // LeftMove
        dfs(grid, i, j-1);
    }

    int numIslands(vector<vector<char>>& grid) {
        int m = grid.size();
        int n = grid[0].size();
        int ans = 0;

        for(int i=0; i<m; i++){
            for(int j=0; j<n; j++){
                if(grid[i][j] !='0' && grid[i][j] != 'x'){
                    dfs(grid, i, j);
                    ans++;
                }
            }
        }
        return ans;
    }
};
```

Time and space complexity =

?

## 3. Flood Fill (Leetcode-733)

**Problem Statement:**
An image is represented by an **m x n** integer grid **image** where **image[i][j]** represents the pixel value of the image.

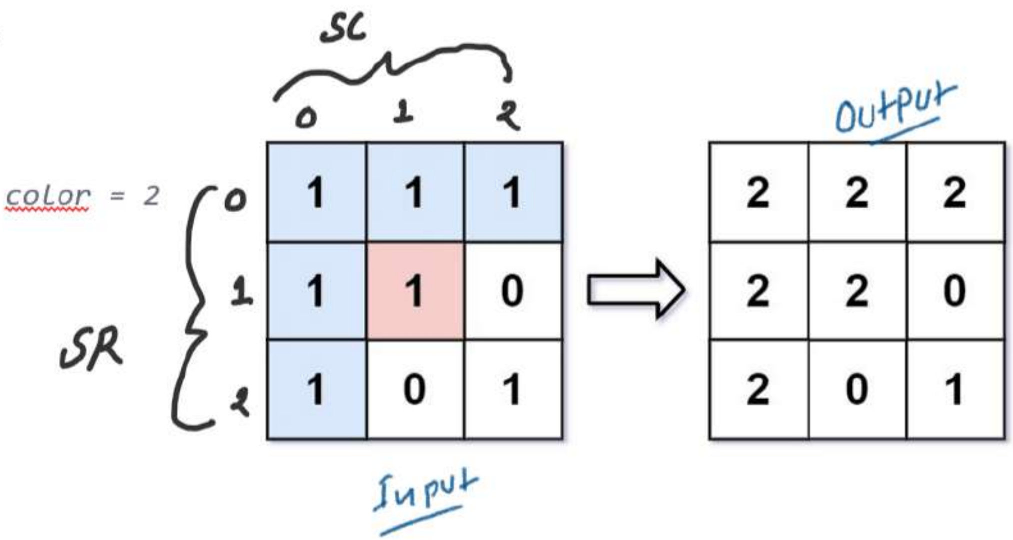
You are also given three integers sr, sc, and color. You should perform a **flood fill** on the image starting from the pixel **image[sr][sc]**.

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

Return the modified image after performing the flood fill.

**Example 1:**
Input: image = [[1,1,1],[1,1,0],[1,0,1]], sr = 1, sc = 1, color = 2
Output: [[2,2,2],[2,2,0],[2,0,1]]

- oldColour = image[1][1]
  = 1
- newColour = 2



|  | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 |
| 2 | 1 | 0 | 1 |

|  |  |  |
|---|---|---|
| 2 | 2 | 2 |
| 2 | 2 | 0 |
| 2 | 0 | 1 |

## Solve using DFS Algo

**Example 1:**

Input: image = [[1,1,1],[1,1,0],[1,0,1]], sr = 1, sc = 1, color = 2
Output: [[2,2,2],[2,2,0],[2,0,1]]

| Image | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 1 | 1 | 1 |
| 1 | 1 | SRC<br>1 | 0 |
| 2 | 1 | 0 | 1 |

SRC Node = Image[1][1]

Old Color = 1

New Color = 2

| Ans | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 2 | 2 | 2 |
| 1 | 2 | 2 | 0 |
| 2 | 2 | 0 | 1 |

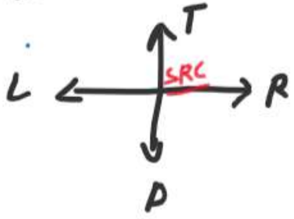
Output

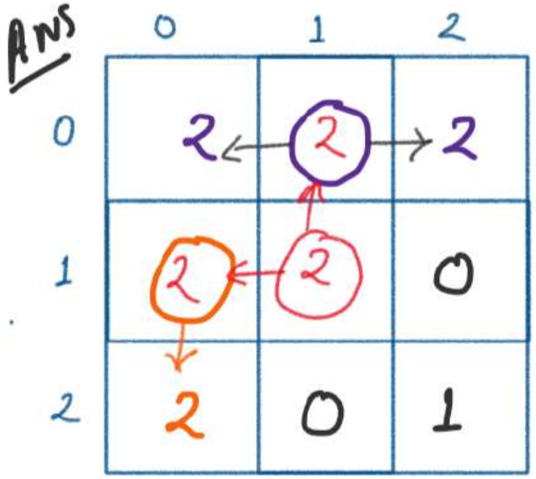
# DRY RUN

| key | value |
|---|---|
| 0,0 | ~~F~~ T |
| 0,1 | ~~F~~ T |
| 0,2 | ~~F~~ T |
| 1,0 | ~~F~~ T |
| 1,1 | ~~F~~ T |
| 1,2 | F |
| 2,0 | ~~F~~ T |
| 2,1 | F |
| 2,2 | F |

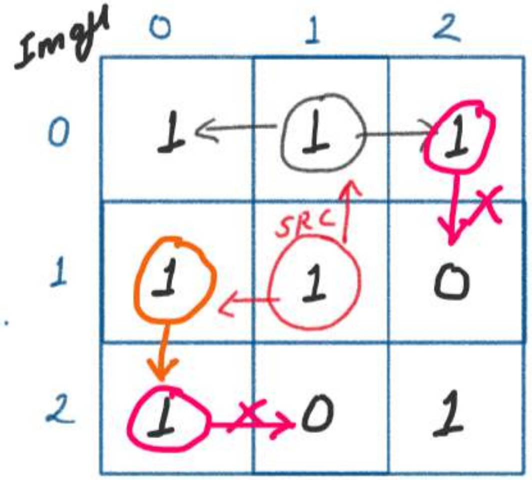
visited

we have 4-move





| | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 2 | 2 | 2 |
| 1 | 2 | 2 | 0 |
| 2 | 2 | 0 | 1 |

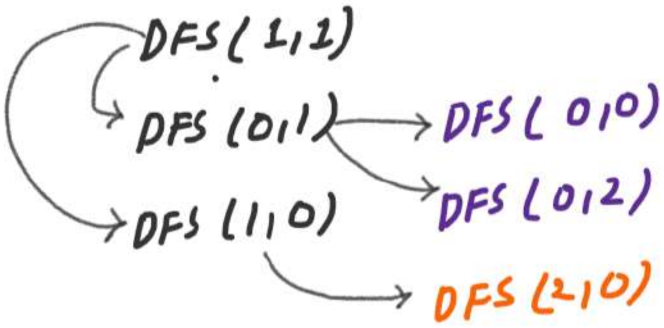


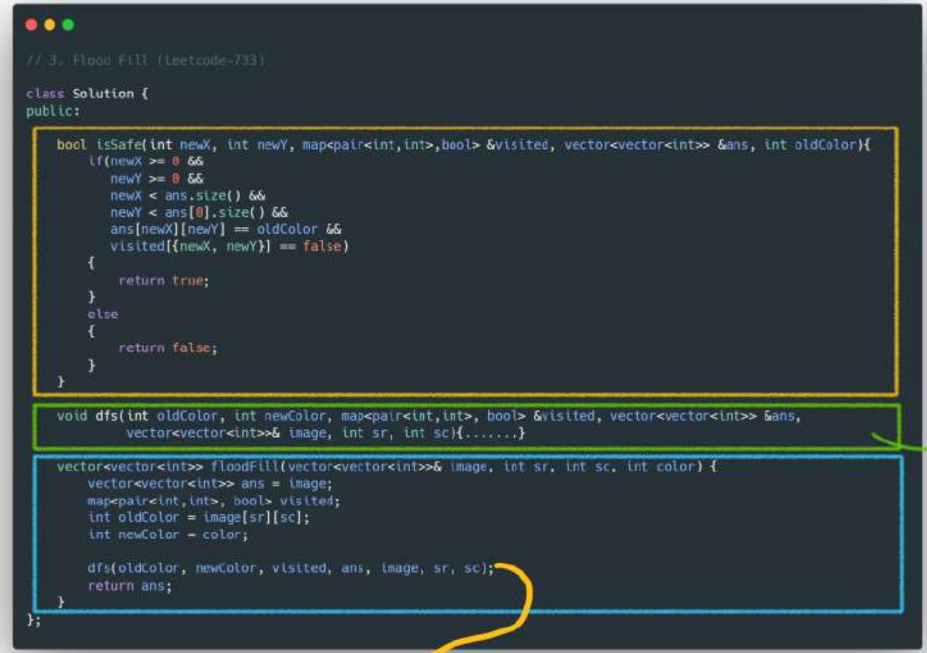
| | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 |
| 2 | 1 | 0 | 1 |

SRC Node = Image[1][1]

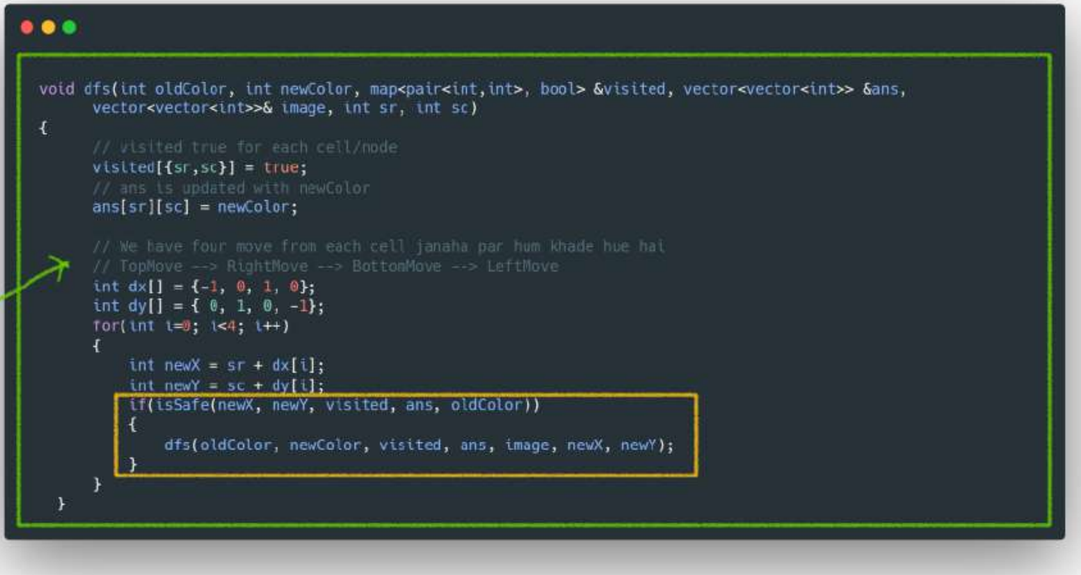
Old Color = 1

New Color = 2





```
// 3. Flood Fill (Leetcode-733)

class Solution {
public:
    bool isSafe(int newX, int newY, map<pair<int,int>,bool> &visited, vector<vector<int>> &ans, int oldColor){
        if(newX >= 0 &&
           newY >= 0 &&
           newX < ans.size() &&
           newY < ans[0].size() &&
           ans[newX][newY] == oldColor &&
           visited[{newX, newY}] == false)
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    void dfs(int oldColor, int newColor, map<pair<int,int>, bool> &visited, vector<vector<int>> &ans,
             vector<vector<int>>& image, int sr, int sc){.......}

    vector<vector<int>> floodFill(vector<vector<int>>& image, int sr, int sc, int color) {
        vector<vector<int>> ans = image;
        map<pair<int,int>, bool> visited;
        int oldColor = image[sr][sc];
        int newColor = color;

        dfs(oldColor, newColor, visited, ans, image, sr, sc);
        return ans;
    }
};
```



```
void dfs(int oldColor, int newColor, map<pair<int,int>, bool> &visited, vector<vector<int>> &ans,
      vector<vector<int>>& image, int sr, int sc)
{
      // visited true for each cell/node
      visited[{sr,sc}] = true;
      // ans is updated with newColor
      ans[sr][sc] = newColor;

      // We have four move from each cell janaha par hum khade hue hai
      // TopMove --> RightMove --> BottomMove --> LeftMove
      int dx[] = {-1, 0, 1, 0};
      int dy[] = { 0, 1, 0, -1};
      for(int i=0; i<4; i++)
      {
          int newX = sr + dx[i];
          int newY = sc + dy[i];
          if(isSafe(newX, newY, visited, ans, oldColor))
          {
              dfs(oldColor, newColor, visited, ans, image, newX, newY);
          }
      }
  }
```

Why single call of DFS ?

↳ Because of SRC Node ke according HAME newColor fill Karna Hai.

# 4. Rotting Oranges (Leetcode-994)

***PROBLEM STATEMENT:***

*You are given an **m x n** grid where each cell can have one of three values:*

- ***0** representing an **empty** cell,*
- ***1** representing a **fresh** orange, or*
- ***2** representing a **rotten** orange.*

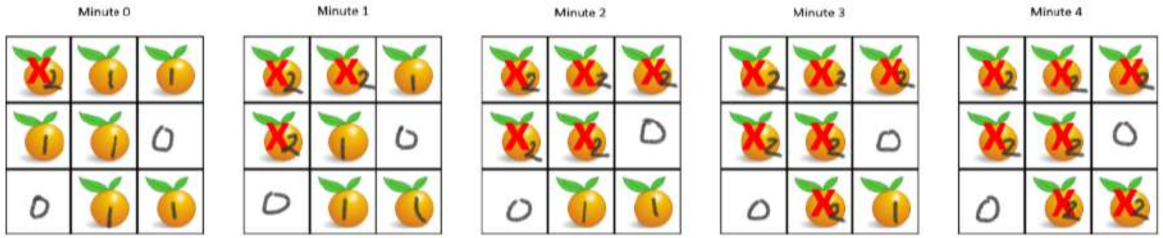
*Every minute, any fresh orange that is 4-directionally adjacent to a rotten orange becomes rotten.*

*Return the **minimum number of minutes** that must elapse until **no cell has a fresh orange**. If this is impossible, return **-1**.*

***Example 1:***

*Input: grid = [[2,1,1],[1,1,0],[0,1,1]]*

*Output: 4*

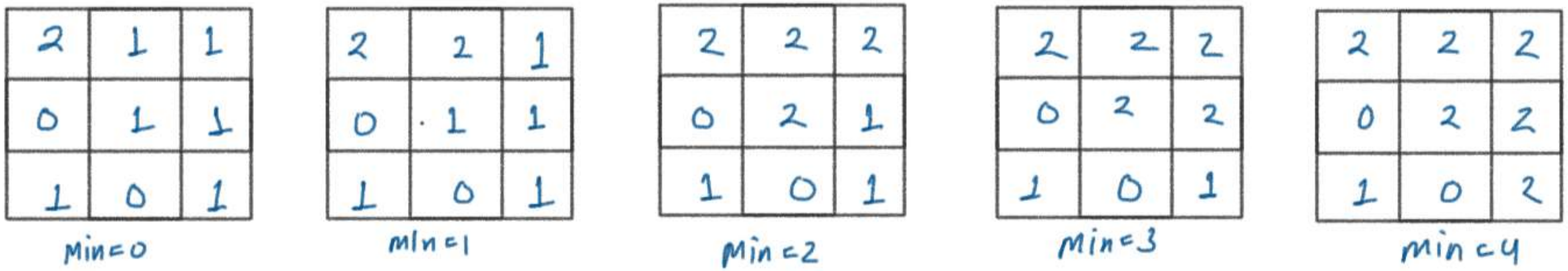


***no cell has a fresh orange***

**Example 2:**

*Input: grid = [[2,1,1],[0,1,1],[1,0,1]]*

*Output: -1*



| | | |
|---|---|---|
| 2 | 1 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |

Min=0

| | | |
|---|---|---|
| 2 | 2 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |

mIn=1

| | | |
|---|---|---|
| 2 | 2 | 2 |
| 0 | 2 | 1 |
| 1 | 0 | 1 |

Min=2

| | | |
|---|---|---|
| 2 | 2 | 2 |
| 0 | 2 | 2 |
| 1 | 0 | 1 |

Min=3

| | | |
|---|---|---|
| 2 | 2 | 2 |
| 0 | 2 | 2 |
| 1 | 0 | 2 |

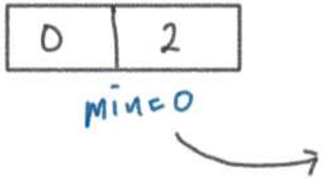
min=4

***One cell has a fresh orange so return -1***

**Example 3:**

*Input: grid = [[0,2]]*

*Output: 0*



| | |
|---|---|
| 0 | 2 |

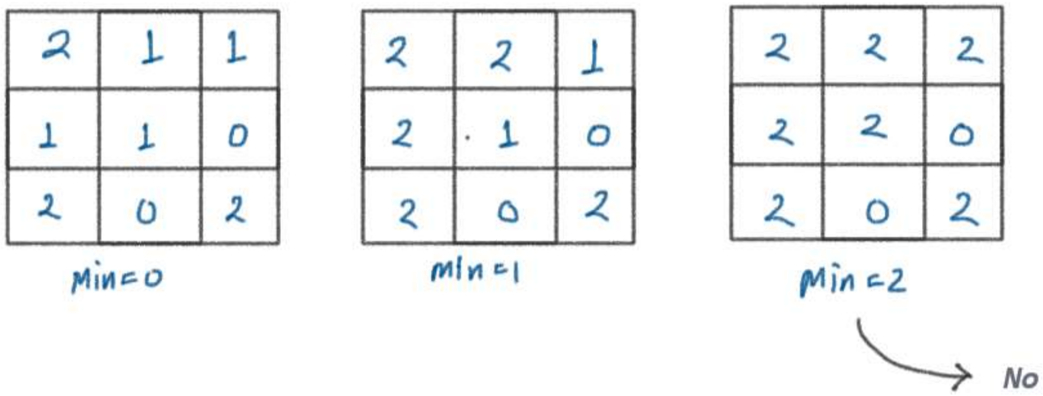
Min=0

***No cell has a fresh orange***

**Example 4:**

*Input: grid = [[2,1,1],[1,1,0],[2,0,2]]*

*Output: 2*



| | | |
|---|---|---|
| 2 | 1 | 1 |
| 1 | 1 | 0 |
| 2 | 0 | 2 |

Min=0

| | | |
|---|---|---|
| 2 | 2 | 1 |
| 2 | 1 | 0 |
| 2 | 0 | 2 |

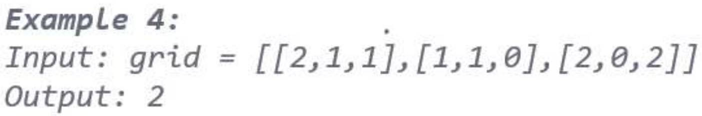
min=1

| | | |
|---|---|---|
| 2 | 2 | 2 |
| 2 | 2 | 0 |
| 2 | 0 | 2 |

Min=2

*No cell has a fresh orange*

solve using BFS



**Example 4:**

Input: grid = [[2,1,1],[1,1,0],[2,0,2]]

Output: 2

⇛ initial state

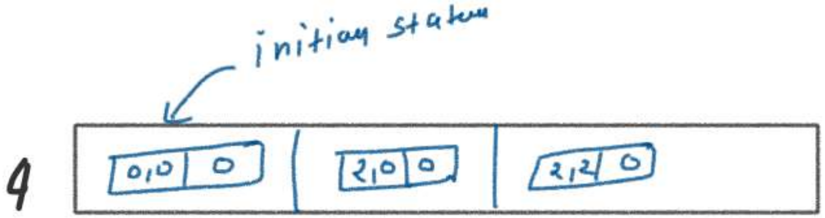
q.push ( coordinates, Time)

when grid[x][y] == 2

grid

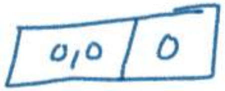
|   |   |   |   |
|---|---|---|---|
| 0 | 2 | 1 | 1 |
| 1 | 1 | 1 | 0 |
| 2 | 2 | 0 | 2 |
|   | 0 | 1 | 2 |

Min=0

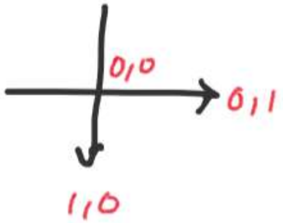


# Iteration 1

**STEP1** Get frontNode and pop it

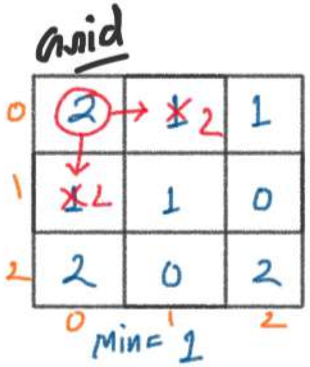


**STEP2** we Have four from each node
so go to possible move to mark rotten orange



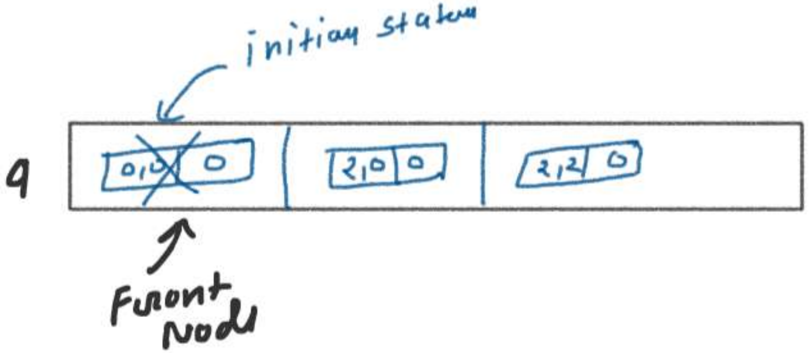
and push rotten orange into queue

Grid



|   | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 2 | ~~1~~ 2 | 1 |
| 1 | ~~1~~ 2 | 1 | 0 |
| 2 | 2 | 0 | 2 |

Min= 1

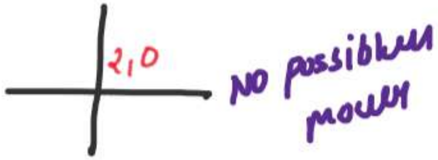


## Iteration 2

**STEP1** Get frontNode and pop it

| | |
|---|---|
| 2,0 | 0 |

**STEP2** We Have four from each node
so go to possible move to mark rotten orange



and push rotten orange into queue

Grid

| | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 2 | 2 | 1 |
| 1 | 2 | 1 | 0 |
| 2 | 2 | 0 | 2 |

Min= 1

initial state

| q | | | | |
|---|---|---|---|---|
| | ~~2,0~~ \| 0 | 2,2 \| 0 | 0,1 \| 1 | 1,0 \| 1 |

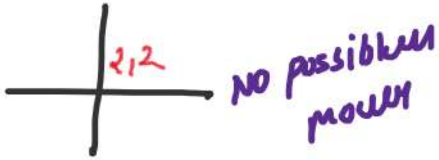
Front Node

# Iteration 3

STEP1 Get frontNode and pop it

| | |
|---|---|
| 2,2 | 0 |

STEP2 we Have four from each node
so go to possible move to make rotten orange

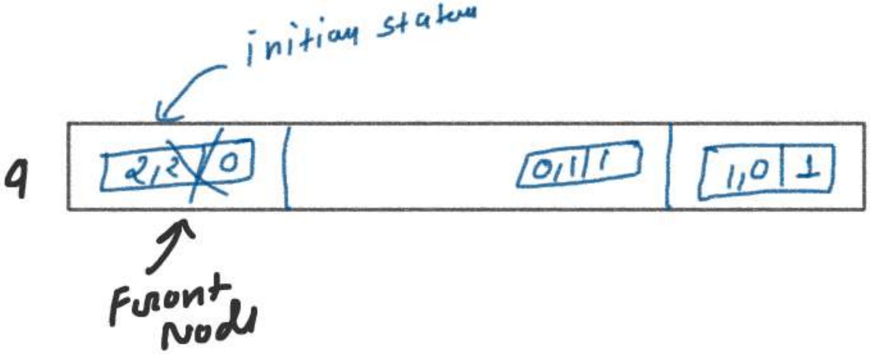


and push rotten orange into queue

Grid

| | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 2 | 2 | 1 |
| 1 | 2 | 1 | 0 |
| 2 | 2 | 0 | 2 |

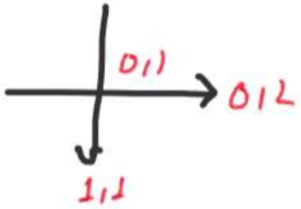
Min=1



# Iteration 4

**STEP1** Get frontNode and pop it

**STEP2** We have four from each node
so go to possible move to mark rotten orange

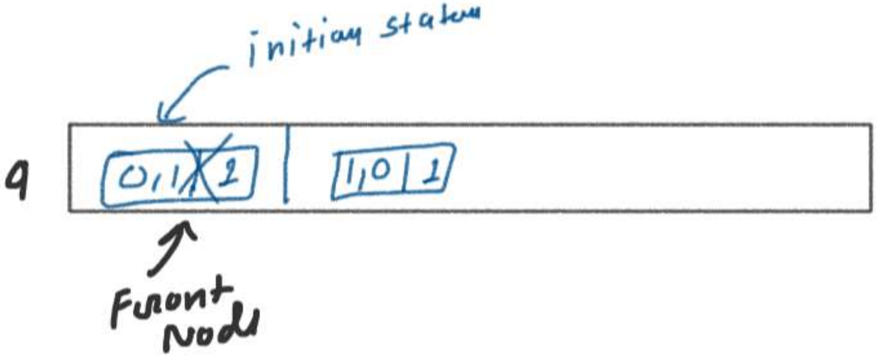


and push rotten orange into queue

Grid

|   | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 2 | 2 | ~~1~~ 2 |
| 1 | 2 | ~~1~~ 2 | 0 |
| 2 | 2 | 0 | 2 |

Min = 2

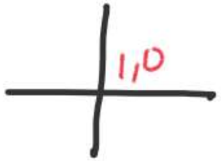


# Iteration 5

**STEP1** Get frontNode and pop it

| 1,0 | 1 |
|---|---|

**STEP2** we Have four from each node
so go to possible move to make rotten orange



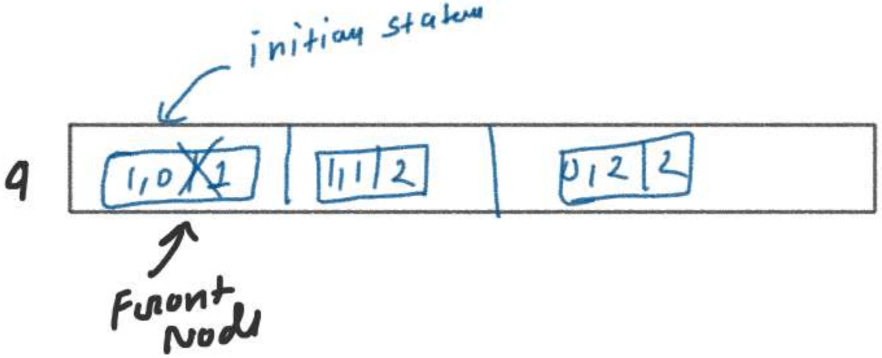
No possible move

and push rotten orange into queue

Grid

| | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 2 | 2 | 2 |
| 1 | 2 | 2 | 0 |
| 2 | 2 | 0 | 2 |

Min = 2

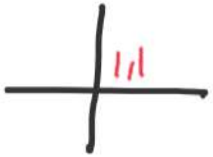


## Iteration 6

**STEP1** Get frontNode and pop it

| 1,1 | 2 |
|---|---|

**STEP2** We have four from each node so go to possible move to make rotten orange



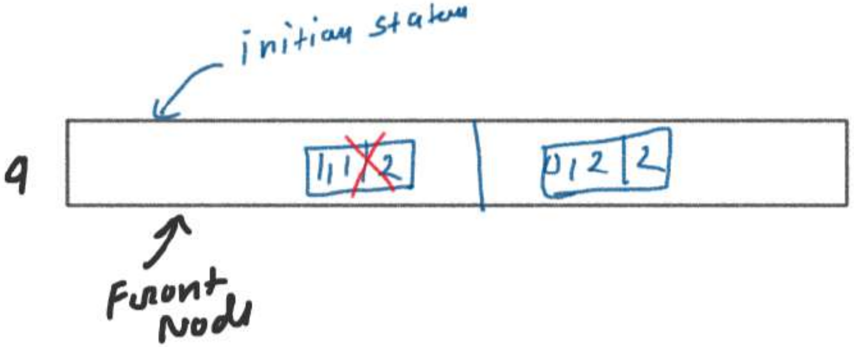
No possible move

and push rotten orange into queue

Grid

|   | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 2 | 2 | 2 |
| 1 | 2 | 2 | 0 |
| 2 | 2 | 0 | 2 |

Min = 2

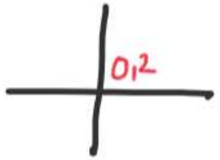


# Iteration 7

**STEP1** Get frontNode and pop it

| 0,2 | 2 |
|---|---|

**STEP2** we Have four from each node so go to possible move to make rotten orange



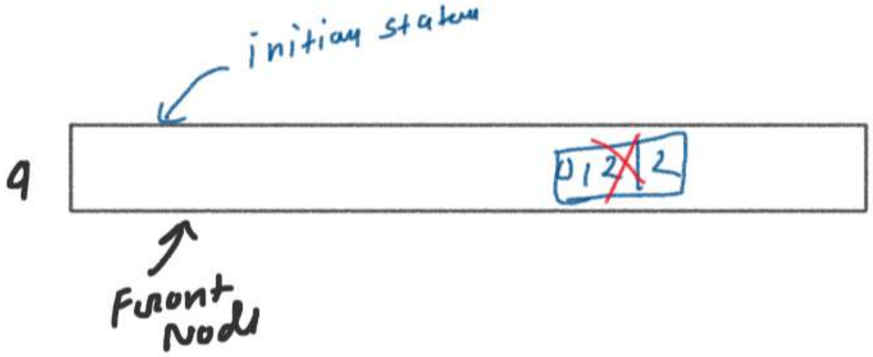
No possible move

and push rotten orange into queue

Grid

| | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 2 | 2 | 2 |
| 1 | 2 | 2 | 0 |
| 2 | 2 | 0 | 2 |

Min=2



# Iteration 8

grid

|   | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 2 | 2 | 2 |
| 1 | 2 | 2 | 0 |
| 2 | 2 | 0 | 2 |

Min= 2

STEP3 jab queue empty Ho jayega it means Hamne all possible oranges ko Rotten kar diya Hai ⇒

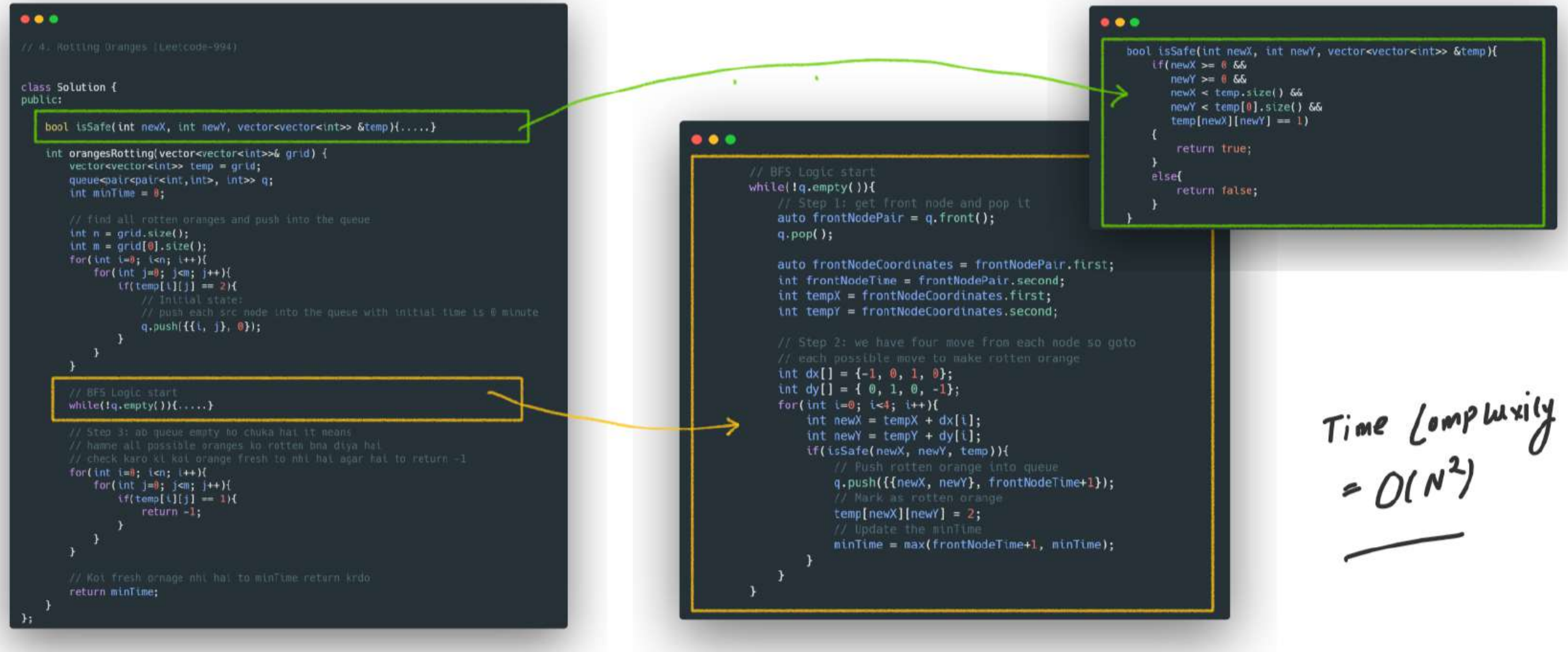
fresh-Oranges == 1

↳ return $\boxed{-1}$

fresh-Oranges != 1

↳ return $\boxed{\text{Time}}$

q | EMPTY



```cpp
// 4. Rotting Oranges (Leetcode-994)


class Solution {
public:

    bool isSafe(int newX, int newY, vector<vector<int>> &temp){.....}

    int orangesRotting(vector<vector<int>>& grid) {
        vector<vector<int>> temp = grid;
        queue<pair<pair<int,int>, int>> q;
        int minTime = 0;

        // find all rotten oranges and push into the queue
        int n = grid.size();
        int m = grid[0].size();
        for(int i=0; i<n; i++){
            for(int j=0; j<m; j++){
                if(temp[i][j] == 2){
                    // Initial state:
                    // push each src node into the queue with initial time is 0 minute
                    q.push({{i, j}, 0});
                }
            }
        }

        // BFS Logic start
        while(!q.empty()){.....}

        // Step 3: ab queue empty ho chuka hai it means
        // hamne all possible oranges ko rotten bna diya hai
        // check karo ki koi orange fresh to nhi hai agar hai to return -1
        for(int i=0; i<n; i++){
            for(int j=0; j<m; j++){
                if(temp[i][j] == 1){
                    return -1;
                }
            }
        }

        // Koi fresh ornage nhi hai to minTime return krdo
        return minTime;
    }
};
```

```cpp
bool isSafe(int newX, int newY, vector<vector<int>> &temp){
    if(newX >= 0 &&
       newY >= 0 &&
       newX < temp.size() &&
       newY < temp[0].size() &&
       temp[newX][newY] == 1)
    {
        return true;
    }
    else{
        return false;
    }
}
```

```cpp
// BFS Logic start
while(!q.empty()){
    // Step 1: get front node and pop it
    auto frontNodePair = q.front();
    q.pop();

    auto frontNodeCoordinates = frontNodePair.first;
    int frontNodeTime = frontNodePair.second;
    int tempX = frontNodeCoordinates.first;
    int tempY = frontNodeCoordinates.second;

    // Step 2: we have four move from each node so goto
    // each possible move to make rotten orange
    int dx[] = {-1, 0, 1, 0};
    int dy[] = { 0, 1, 0, -1};
    for(int i=0; i<4; i++){
        int newX = tempX + dx[i];
        int newY = tempY + dy[i];
        if(isSafe(newX, newY, temp)){
            // Push rotten orange into queue
            q.push({{newX, newY}, frontNodeTime+1});
            // Mark as rotten orange
            temp[newX][newY] = 2;
            // Update the minTime
            minTime = max(frontNodeTime+1, minTime);
        }
    }
}
```

Time Complexity

$= O(N^2)$