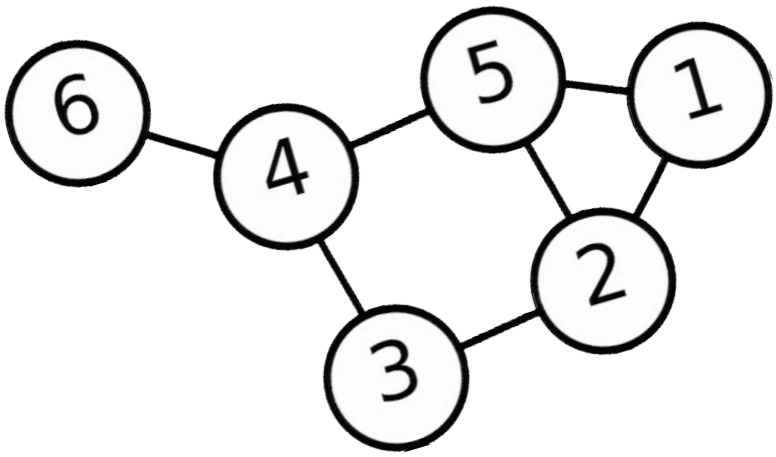
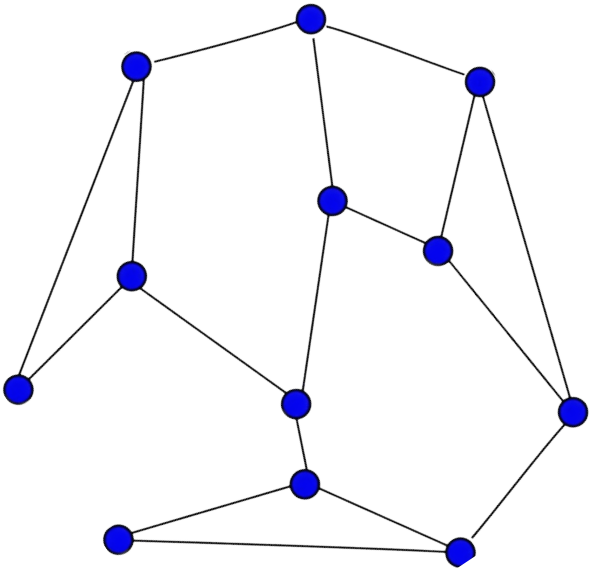


Graphs (Part-2)



- By Kapil Yadav

1. BFS/DFS Traversal Problems
2. Cycle Detection In Undirected and Directed Graph

https://www.linkedin.com/in/kapilyadav22/

Basic Traversal Questions:

Number of provinces
Flood fill
Number of islands
Max Area of island
Number of closed islands
Count Sub Islands
Number of Enclaves
Rotting Oranges
All Paths from source to target
Keys and Rooms
Surrounded Regions
https://leetcode.com/problems/clone-graph/
https://www.geeksforgeeks.org/detect-cycle-undirected-graph/
Find Eventual safe state
01 Matrix
Pacific Atlantic Water Flow
Shortest path with alternating colors
Reorder Routes to Make All Paths Lead to the City Zero
Shortest Path visiting all nodes
Shortest Bridge
As Far from Land as Possible
Jump Game 3

547. Number of Provinces

Medium 6.5K 270

Companies

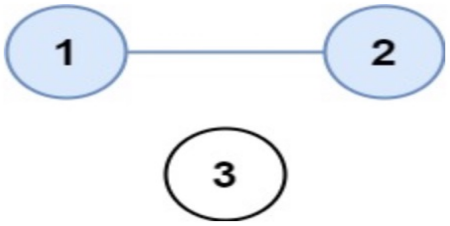
There are `n` cities. Some of them are connected, while some are not. If city `a` is connected directly with city `b`, and city `b` is connected directly with city `c`, then city `a` is connected indirectly with city `c`.

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 i^{th} city and the j^{th} city are directly connected, and `isConnected[i][j] = 0` otherwise.

Return *the total number of* ***provinces***.

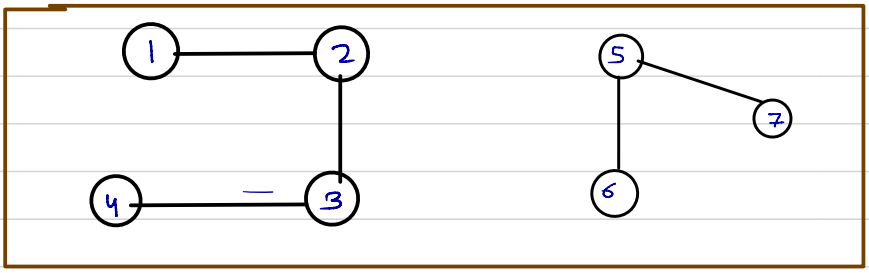
Example 1:




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

→ We need find the No. of Provinces, or we can say the number of connected components.

for example:-



→ In the above example, there are two components, (1-4) and (5-7)
→ If we run BFS/DFS on graph, we can find the no. of connected components.
→ Lets say we run a DFS on the graph, starting from 1, from 1 → 2 → 3 → 4, so to keep track of traversed nodes, we need a visited array.
→ Now from 4, there is no way to reach rest of the unvisited nodes (5, 6, 7), so we need to check every node and call DFS for unvisited nodes.
→ The number of times DFS will be called from findCircleNum

function will be the No. of provinces.

→ We can either create Adjacency List from given grid or can use that grid as Adjacency Matrix.

→ We will use the given grid as adj matrix.

→ Since, we need to count the number of provinces, numbering of nodes does not matter.

→ We will start numbering from 0 to N-1, since it will be easy for adj matrix indices.

Algorithm :-

→ Declare a visited array and initialize it with 0.

→ Declare a count variable and initialize it with 0.

→ Run a loop from 0 to V-1, where V is the No. of nodes.

→ Check if the Node is visited or not.

↳ if not, then increament the count and run DFS.

```cpp
#Number of Provinces
void dfs(int node,vector<bool>& vis, vector<vector<int>>& grid){
    vis[node]=1;
    for(int i=0; i<grid[node].size();i++){
      if(grid[node][i] && vis[i]==0){
      dfs(i,vis,grid);
      }
    }
}

int findCircleNum(vector<vector<int>>& isConnected) {
    int V = isConnected.size();
    vector<bool> vis(V,0);
    int cnt=0;
    for(int i=0;i<V;i++){
      if(vis[i]==0) {
        cnt++;
        dfs(i,vis,isConnected);
      }
    }
    return  cnt;
}
```

$TC = O(V \times V)$

$SC = O(V) + O(V)$

↳ visited ↳ recursive stack space

→ This can be solved using Adj list also, where $TC :- O(V) + O(V+2E)$.

733. Flood Fill

Hint

Easy 5.8K 563

Companies

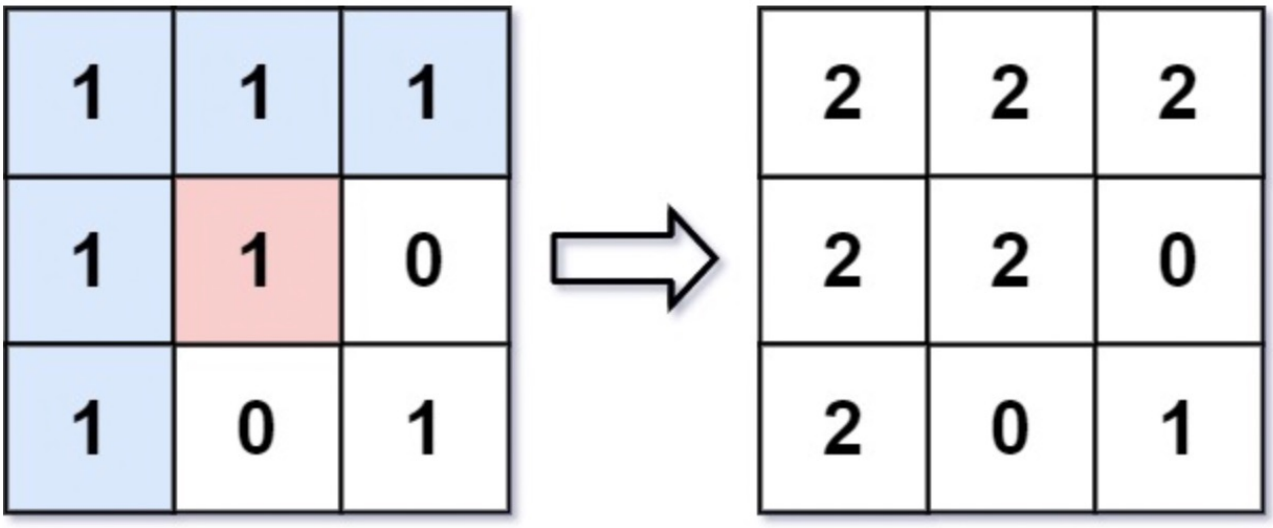
An image is represented by an `m x n` integer grid `image` where `image[i][j]` represents the pixel value of the image.

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:



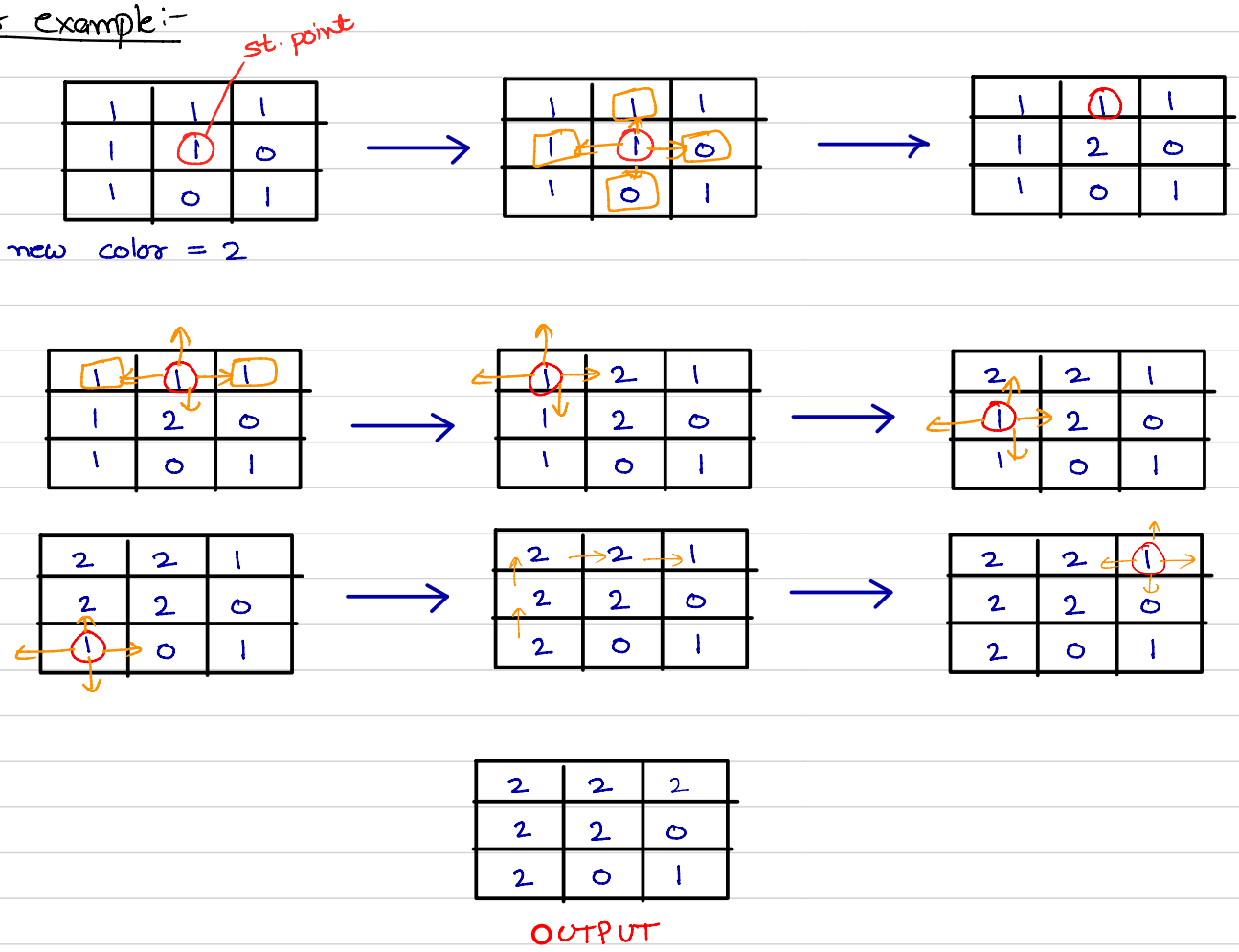

```
Input: image = [[1,1,1],[1,1,0],[1,0,1]], sr = 1, sc = 1, color = 2
Output: [[2,2,2],[2,2,0],[2,0,1]]
Explanation: From the center of the image with position (sr, sc) = (1, 1)
(i.e., the red pixel), all pixels connected by a path of the same color
as the starting pixel (i.e., the blue pixels) are colored with the new
color.
Note the bottom corner is not colored 2, because it is not 4-
directionally connected to the starting pixel.
```

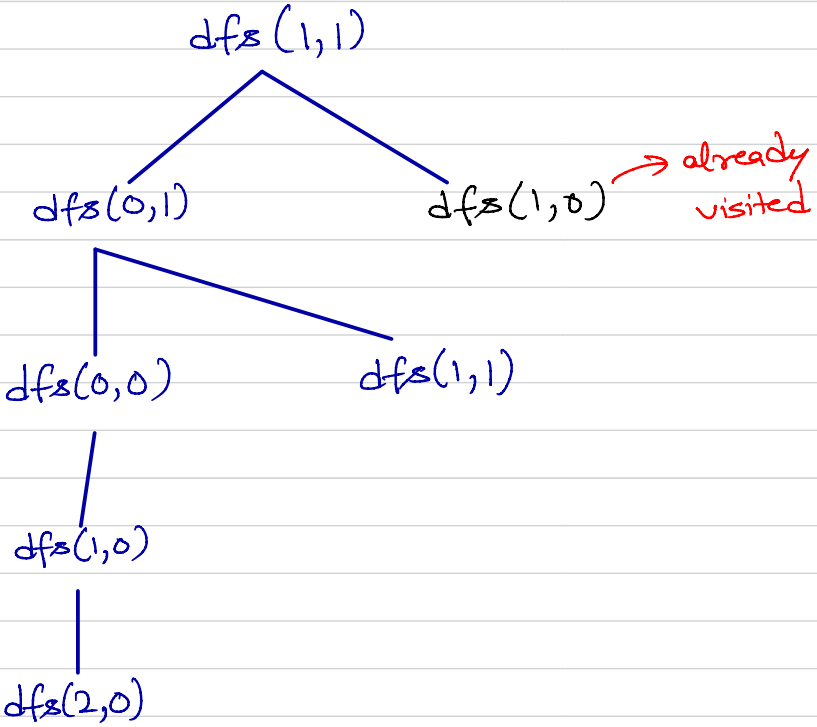
Example 2:

```
Input: image = [[0,0,0],[0,0,0]], sr = 0, sc = 0, color = 0
Output: [[0,0,0],[0,0,0]]
Explanation: The starting pixel is already colored 0, so no changes are
made to the image.
```

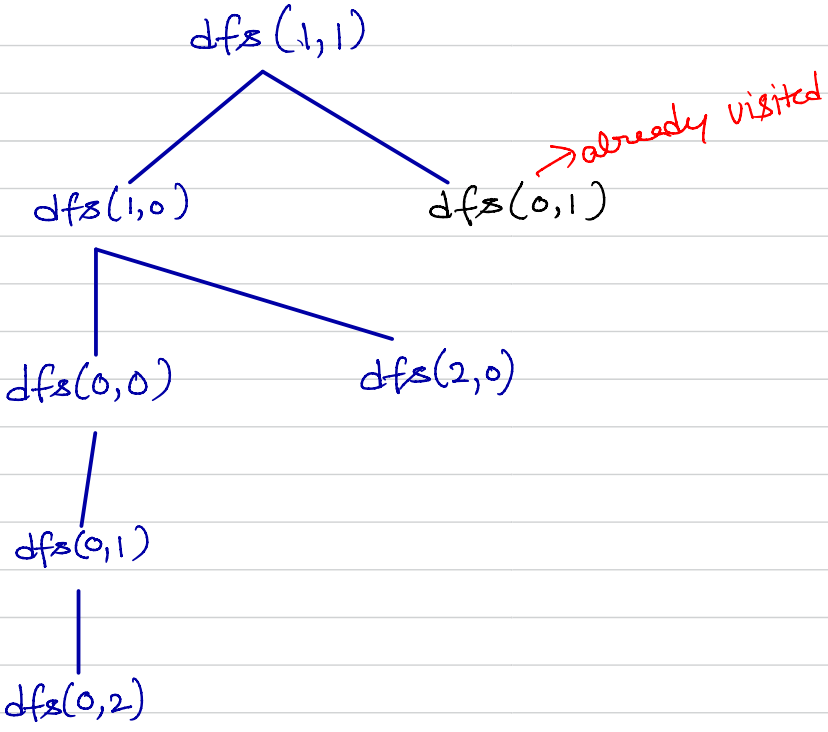
→ We have given the starting point in the form of row and column.
→ We need to change the colour of starting point and its connected neighbours having same colour as starting point to new color.

for example:-




dfs(1,1)
dfs(0,1)
dfs(1,0)
already visited
dfs(0,0)
dfs(1,1)
dfs(1,0)
dfs(2,0)

OR


dfs(1,1)
dfs(1,0)
dfs(0,1)
already visited
dfs(0,0)
dfs(2,0)
dfs(0,1)
dfs(0,2)

NOTE :- If the starting point color is same as new color, we don't need to do anything.
Given grid will be our final answer.
Example 2 in input.

→ In WC, we need to traverse every node, which will take O(N×M) time. and at every node, we are traversing for four neighbours, so O(N×M×4)

```
#Flood Fill
void dfs(vector<vector<int>>& image,int cr,int cc,int newcolor,int oldcolor){
    int m = image.size();
    int n= image[0].size();
    if(cr<0 || cr>=m || cc<0 || cc>=n || image[cr][cc]!=oldcolor){
            return;
        }
    image[cr][cc]= newcolor;
    dfs(image,cr+1,cc,newcolor,oldcolor);
    dfs(image,cr-1,cc,newcolor,oldcolor);
    dfs(image,cr,cc+1,newcolor,oldcolor);
    dfs(image,cr,cc-1,newcolor,oldcolor);
    return;
    }

vector<vector<int>> floodFill(vector<vector<int>>& image, int sr, int sc, int color) {
    if(image[sr][sc]==color) return image;
    dfs(image,sr,sc,color,image[sr][sc]);
    return image;
    }
```

TC = O(N×M) + O(N×M×4)
Calling dfs ↵
↳ for every dfs call, calling its 4 neighbours [so, in total 4×N×M times neighbours will be called]

≃ O(N×M)

SC :- O(N×M) + O(N×M)
↳ Recursive stack space ↳ visited array

→ There is a better way to call multiple recursive functions on a grid.
→ If we want to go 4 Directions, then take

row[4] = {-1, 0, 1, 0}
col[4] = {0, -1, 0, 1}

```cpp
#Flood Fill Using DFS
int rowdir[4] = {-1,0,1,0};
int coldir[4] = {0,-1,0,1};
void dfs(vector<vector<int>>& image,int cr,int cc,int newcolor,int oldcolor){
    int m = image.size();
    int n= image[0].size();
    if(cr<0 || cr>=m || cc<0 || cc>=n || image[cr][cc]!=oldcolor){
        return;
    }
    image[cr][cc]= newcolor;
    for(int i=0;i<4;i++){
        int nr = cr+rowdir[i];
        int nc = cc+coldir[i];
        dfs(image,nr,nc,newcolor,oldcolor);
    }
    return;
}

vector<vector<int>> floodFill(vector<vector<int>>& image, int sr, int sc, int color) {
  if(image[sr][sc]==color) return image;
  dfs(image,sr,sc,color,image[sr][sc]);
  return image;
}
```

Flood Fill using BFS :-

```cpp
#Flood Fill Using BFS
vector<vector<int>> floodFill(vector<vector<int>>& image, int sr, int sc, int newColor) {
  if(image[sr][sc]==newColor)
      return image;

    //using bfs;
    queue<pair<int,int>>pq;
    pq.push({sr,sc});

   int m = image.size();
   int n = image[0].size();
   int oldcolor = image[sr][sc];

    while(pq.size()>0)
    {
        int currRow=pq.front().first;
        int currCol=pq.front().second;

        pq.pop();

        if(currRow<0 || currRow>=m || currCol<0 || currCol>=n
           || image[currRow][currCol]!=oldcolor)
               continue;

      image[currRow][currCol]=newColor;
      pq.push({currRow-1,currCol});
      pq.push({currRow,currCol-1});
      pq.push({currRow+1,currCol});
      pq.push({currRow,currCol+1});
        }
    return image;
}
```

BFS approach also works in same way, its just check and updates neighbours first, then go to next level. i.e neighbours of neighbours.

```cpp
#Flood Fill Using BFS
vector<vector<int>> floodFill(vector<vector<int>>& image, int sr, int sc, int color)
    {
  if(image[sr][sc]==color) return image;
  int m = image.size();
  int n = image[0].size();

  int rowdir[4] = {-1,0,1,0};
  int coldir[4] = {0,-1,0,1};

  queue<pair<int,int>> q;
  q.push({sr,sc});
  int oldcolor = image[sr][sc];

  while(!q.empty()){
      int currrow = q.front().first;
      int currcol = q.front().second;
      q.pop();
      image[currrow][currcol] = color;

      for(int i=0;i<4;i++){
          int nr = currrow+rowdir[i];
          int nc = currcol+coldir[i];

          if(nr<0 || nr>=m || nc<0 || nc>=n || image[nr][nc]!=oldcolor)
              continue;

          q.push({nr,nc});

      }
  }
  return image;
}
```

200. Number of Islands

Medium 17.7K 407

Companies

Given an `m x n` 2D binary grid `grid` which represents a 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
```

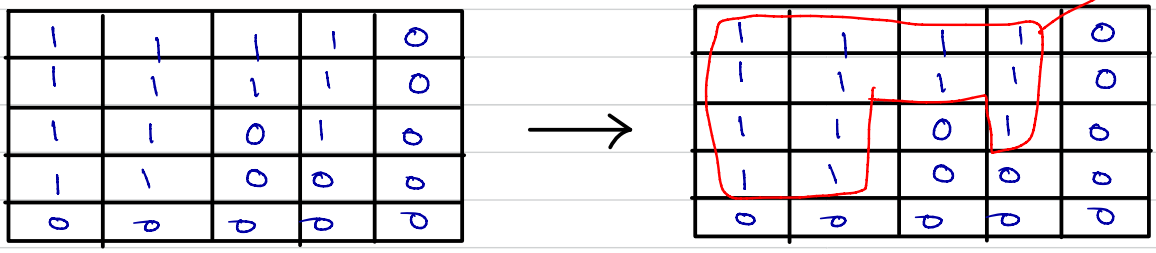
Example 2:

```
Input: grid = [
  ["1","1","0","0","0"],
  ["1","1","0","0","0"],
  ["0","0","1","0","0"],
  ["0","0","0","1","1"]
]
Output: 3
```

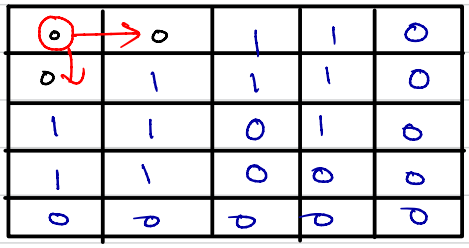
→ We need to count the number of islands.
→ We can use BFS/DFS to traverse through the grid.
→ Lets use BFS first.
→ We need a starting node to put in the queue, so start traversing from row=0, col=0 index, and push the row & column index, where we get our first land.
→ In above ex, (0,0) = "1", so push (0,0) in the queue and insert all the neighbours having value = "1".
→ We can either take another matrix for visited or make changes in our input matrix only.

→ We will traverse through every element in the matrix and check whether there is a Land or not.

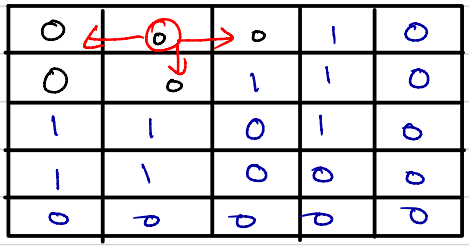
→ No. of islands will be equal to the number of times BFS called while traversing the nodes.



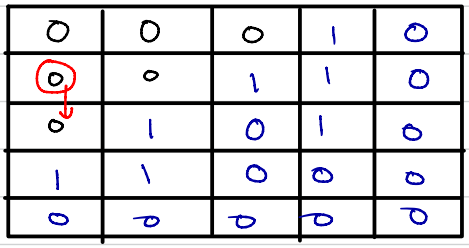
Input → Output = 1



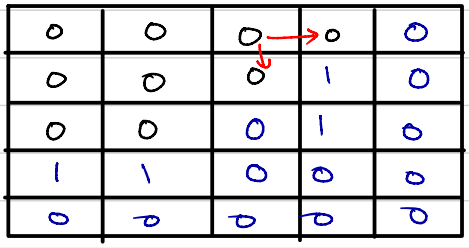
(1,0)
(0,1)
~~(0,0)~~



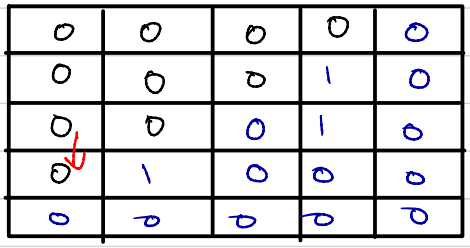
(1,1)
(0,2)
(1,0)
~~(0,1)~~



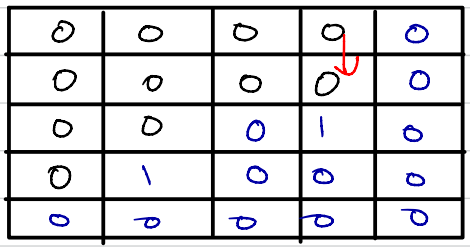
(2,0)
(1,1)
(0,2)
~~(1,0)~~

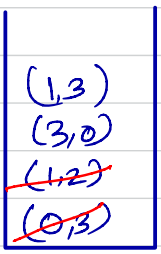


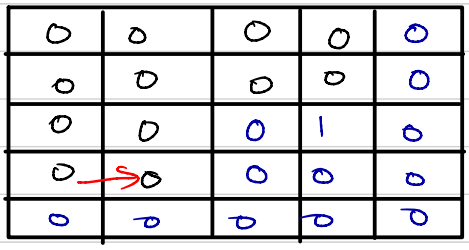
(1,2)
(0,3)
(2,0)
(1,1)
~~(0,2)~~

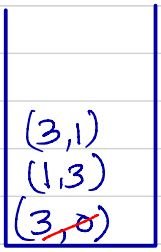


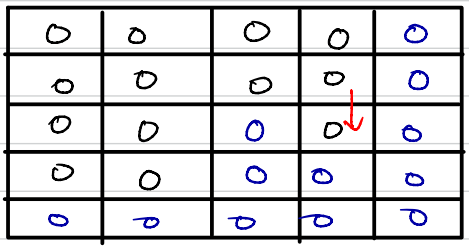
(3,0)
(1,2)
(0,3)
~~(2,0)~~
~~(1,1)~~

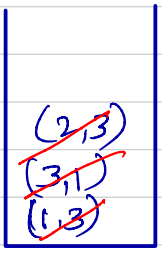












$TC = O(N \times M) + O(N \times M \times 4)$

calling dfs

↳ for every dfs/BFS call, calling its 4 neighbours [so, in total $4 \times N \times M$ times neighbours will be called]

$\simeq O(N \times M)$

$SC :- O(N \times M) + O(N \times M)$

↳ Recursive stack space (or queue space (BFS)) ↳ visited array

```cpp
//No of islands
int rowdir[4] = {-1,0,1,0};
int coldir[4] = {0,-1,0,1};

void bfs(int row,int col, vector<vector<char>>& grid){
    int m = grid.size();
    int n = grid[0].size();

    queue<pair<int,int>> q;
    q.push({row,col});
    grid[row][col] = '0';

    while(!q.empty()){
        int currrow = q.front().first;
        int currcol = q.front().second;
        q.pop();


        for(int i=0;i<4;i++){
        int nr= currrow + rowdir[i];
        int nc = currcol + coldir[i];

        if(nr>=0 && nr<m && nc>=0 && nc<n && grid[nr][nc]=='1'){
            grid[nr][nc]='0';
            q.push({nr,nc});
             }
    }
  }
    return;
}

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

    int rowdir[4] = {-1,0,1,0};
    int coldir[4] = {0,1,0,-1};
    for(int i =0;i<m;i++){
        for(int j=0;j<n;j++){
            if(grid[i][j]=='1')
            {   bfs(i,j,grid);
                count++;
            }
        }
        }
    return count;
}
```

```cpp
//No of islands
int rowdir[4] = {-1,0,1,0};
int coldir[4] = {0,-1,0,1};

void dfs(int currrow,int currcol, vector<vector<char>>& grid){
    int m = grid.size();
    int n = grid[0].size();
    if(currrow<0 || currrow>=m || currcol<0 ||currcol>=n || grid[currrow][currcol]=='0'){
        return ;
    }
    grid[currrow][currcol] = '0';
    for(int i=0;i<4;i++){
        int nr= currrow+rowdir[i];
        int nc = currcol+coldir[i];
        dfs(nr,nc,grid);
    }
    return;
}

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

    int rowdir[4] = {-1,0,1,0};
    int coldir[4] = {0,1,0,-1};
    for(int i =0;i<m;i++){
        for(int j=0;j<n;j++){
            if(grid[i][j]=='1')
            {   dfs(i,j,grid);
                count++;
            }
        }
        }
    return count;
}
```

695. Max Area of Island

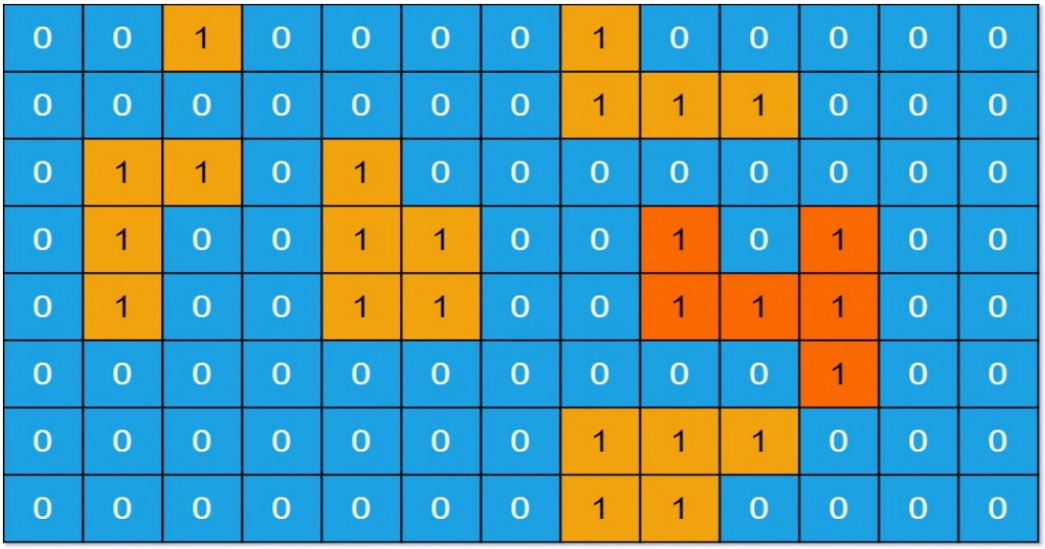
Medium 8.2K 180

You are given an `m x n` binary matrix `grid`. An island is a group of `1`'s (representing land) connected **4-directionally** (horizontal or vertical.) You may assume all four edges of the grid are surrounded by water.

The **area** of an island is the number of cells with a value `1` in the island.

Return *the maximum **area** of an island in* `grid`. If there is no island, return `0`.

Example 1:




```
Input: grid = [[0,0,1,0,0,0,0,1,0,0,0,0,0],[0,0,0,0,0,0,0,1,1,1,0,0,0],
[0,1,1,0,1,0,0,0,0,0,0,0,0],[0,1,0,0,1,1,0,0,1,0,1,0,0],
[0,1,0,0,1,1,0,0,1,1,1,0,0],[0,0,0,0,0,0,0,0,0,0,1,0,0],
[0,0,0,0,0,0,0,1,1,1,0,0,0],[0,0,0,0,0,0,0,1,1,0,0,0,0]]
Output: 6
Explanation: The answer is not 11, because the island must be connected
4-directionally.
```

→ This problem is same as Number of island. We just need to count the size of each island and compare whether it is maximum or not.

→ TC and space complexity will be same as Number of island.

```cpp
//Max Area of Island
int rowdir[4] = {-1,0,1,0};
int coldir[4]= {0,-1,0,1};

int dfs(int row,int col,vector<vector<int>>& grid){
  int m = grid.size();
  int n = grid[0].size();

  grid[row][col] =0;
  int currarea =1;

  for(int i=0;i<4;i++){
      int nr = row+rowdir[i];
      int nc = col+coldir[i];
      if(nr>=0 && nr<m  && nc>=0 && nc<n && grid[nr][nc]==1){
        currarea+=dfs(nr,nc,grid);
      }
  }
 return currarea;
}

int maxAreaOfIsland(vector<vector<int>>& 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]==1){
              int temp=dfs(i,j,grid);
              ans = max(ans,temp);
          }
      }
  }
  return ans;
}
```

```cpp
//Max Area of Island
int maxAreaOfIsland(vector<vector<int>>& image) {
  int m = image.size();
  int n = image[0].size();
  int ans =0;
  for(int currRow = 0;currRow<m;currRow++)
  {
      for(int currCol = 0;currCol<n;currCol++)
      {
          if(image[currRow][currCol]==1)
          {
             int temp =fillgraph(image,currRow,currCol,m,n);
              ans=max(ans,temp);
          }
          }
  }
  return ans;

}
int fillgraph(vector<vector<int>>& image, int currRow, int currCol,int m,int n)
 {
     if(currRow<0 || currRow>=m || currCol<0 || currCol>=n || image[currRow][currCol]==0)
            return 0;

     image[currRow][currCol]=0;

     //up move
     int a =  fillgraph(image,currRow-1,currCol, m,n);

     //right move
    int b =  fillgraph(image,currRow,currCol+1, m,n);

     //bottom move
    int c = fillgraph(image,currRow+1,currCol, m,n);

     //left move
    int d = fillgraph(image,currRow,currCol-1, m,n);

     return 1 + (a+b+c+d);
}
```

1254. Number of Closed Islands

Hint

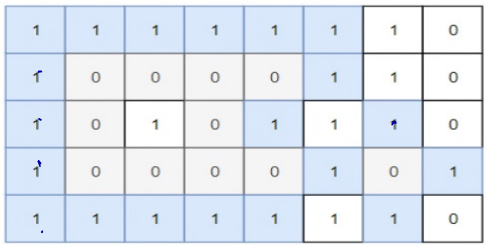
Medium 2.5K 57

Companies

Given a 2D `grid` consists of `0s` (land) and `1s` (water). An *island* is a maximal 4-directionally connected group of `0s` and a *closed island* is an island **totally** (all left, top, right, bottom) surrounded by `1s.`

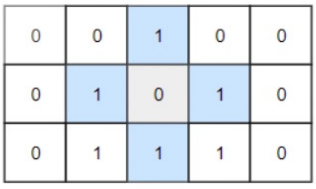
Return the number of *closed islands*.

Example 1:




```
Input: grid = [[1,1,1,1,1,1,1,0],[1,0,0,0,0,1,1,0],[1,0,1,0,1,1,1,0],
[1,0,0,0,0,1,0,1],[1,1,1,1,1,1,1,0]]
Output: 2
Explanation:
Islands in gray are closed because they are completely surrounded by
water (group of 1s).
```

Example 2:

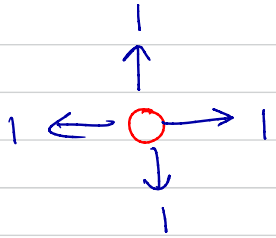



```
Input: grid = [[0,0,1,0,0],[0,1,0,1,0],[0,1,1,1,0]]
Output: 1
```

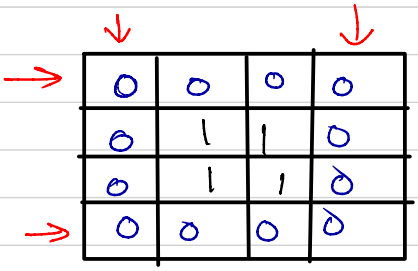
Example 3:

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

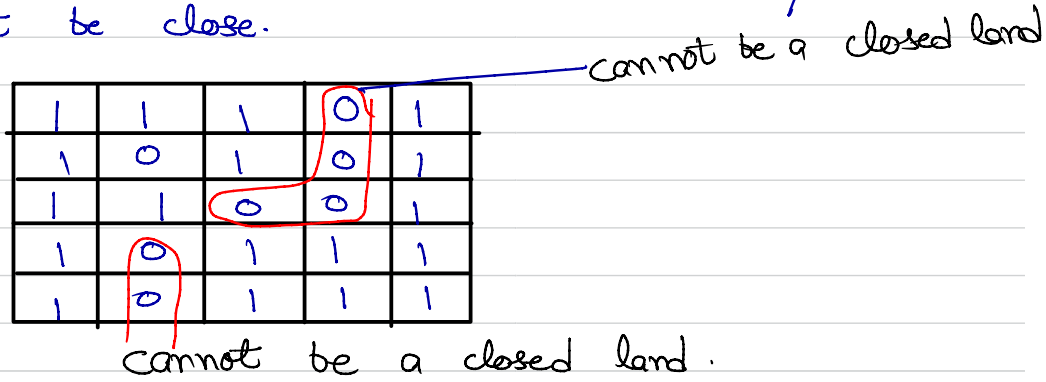
→ It is given that, closed land should be covered by water from all the four directions (up, down, left and right)



→ Let's start observing the pattern:-
→ The boundary land of the grid cannot be close.



→ The Land connected with the boundary land cannot be close.



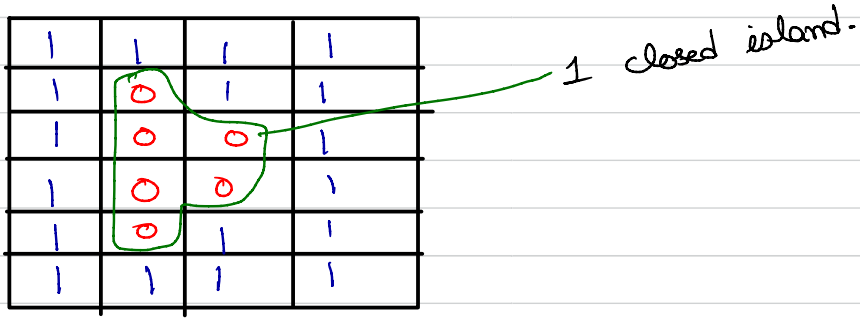
→ Other than above two conditions, rest land will be closed.
→ Basically, we are identifying open lands and convert them into water or we can simply mark them visited, so that we don't consider them in closed island.

→ So first check for boundaries and run a dfs/Bfs to mark all its neighbour lands to visited. (Because they cannot be closed).

→ We need to check first row, last row, first column and last column.

→ Now Traverse the grid again, and check if there is any land. If any, run a BFS/DFS and increase the count of closed island.
BFS/DFS will mark all the neighbouring land as visited

Because it will considered as 1 closed island.



TC = O(M) + O(N) + O(N × M × 4)

Calling dfs ↵

↳ for every dfs/BFS call, calling its 4 neighbours [so, in total 4×N×M times neighbours will be called]

≃ O(N × M)

SC :- O(N × M) + O(N × M)

↳ Recursive stack space or queue space (BFS)

↳ visited array

```cpp
//Number of Closed Islands
int rowdir[4] = {-1,0,1,0};
int coldir[4] = {0,-1,0,1};

void dfs(int row,int col,vector<vector<int>>& vis,vector<vector<int>>& grid){
 int m = grid.size();
 int n = grid[0].size();
 if(row<0 || row>=m || col<0 || col>=n || grid[row][col]==1
    || vis[row][col]==1){
     return; }

  vis[row][col]=1;
  for(int i=0;i<4;i++){
      int nr = row+rowdir[i];
      int nc = col+coldir[i];
      dfs(nr,nc,vis,grid);
  }
}

int closedIsland(vector<vector<int>>& grid) {
  int m = grid.size();
  int n = grid[0].size();
  int count =0;
  vector<vector<int>> vis(m,vector<int> (n,0));

  //traverse first col
  for(int i=0;i<m;i++){
    if(grid[i][0]==0 && !vis[i][0]){
      dfs(i,0,vis,grid);
    }
  }
  //traverse last col
  for(int i=0;i<m;i++) {
    if(grid[i][n-1]==0 && !vis[i][n-1]){
      dfs(i,n-1,vis,grid);
    }
  }
  //traverse first row
  for(int i=0;i<n;i++) {
    if(grid[0][i]==0 && !vis[0][i]){
        dfs(0,i,vis,grid);
    }
  }
  //traverse last row
  for(int i=0;i<n;i++){
    if(grid[m-1][i]==0 && !vis[m-1][i]){
        dfs(m-1,i,vis,grid);
    }
  }

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

1905. Count Sub Islands

Hint

Medium 1.4K 45

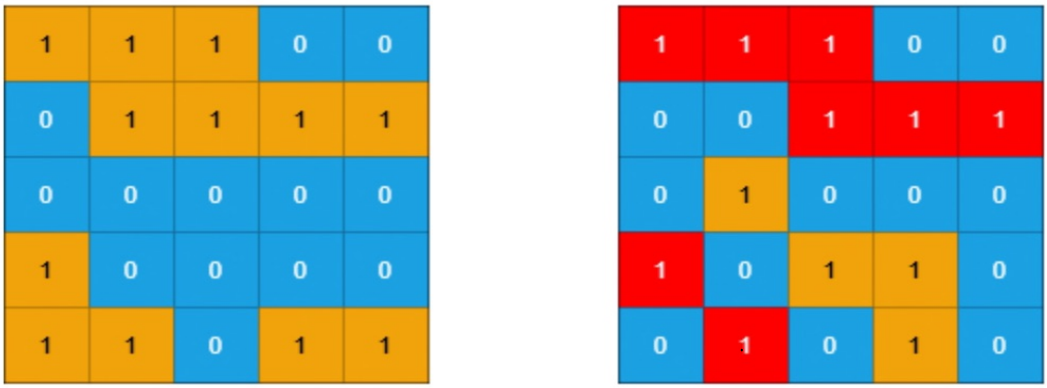
Companies

You are given two `m x n` binary matrices `grid1` and `grid2` containing only `0`'s (representing water) and `1`'s (representing land). An **island** is a group of `1`'s connected **4-directionally** (horizontal or vertical). Any cells outside of the grid are considered water cells.

An island in `grid2` is considered a **sub-island** if there is an island in `grid1` that contains **all** the cells that make up **this** island in `grid2`.

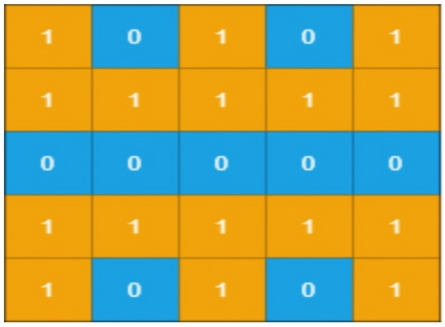
Return the ***number*** *of islands in* `grid2` *that are considered* ***sub-islands***.

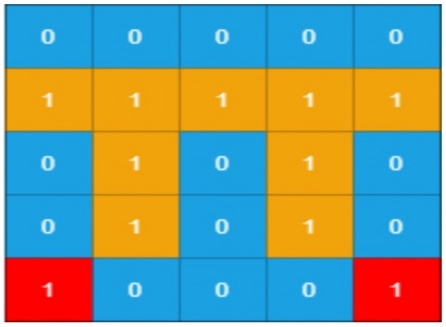
Example 1:




```
Input: grid1 = [[1,1,1,0,0],[0,1,1,1,1],[0,0,0,0,0],[1,0,0,0,0],
[1,1,0,1,1]], grid2 = [[1,1,1,0,0],[0,0,1,1,1],[0,1,0,0,0],[1,0,1,1,0],
[0,1,0,1,0]]
Output: 3
Explanation: In the picture above, the grid on the left is grid1 and the
grid on the right is grid2.
The 1s colored red in grid2 are those considered to be part of a sub-
island. There are three sub-islands.
```

Example 2:





Input: grid1 = [[1,0,1,0,1],[1,1,1,1,1],[0,0,0,0,0],[1,1,1,1,1],[1,0,1,0,1]], grid2 = [[0,0,0,0,0],[1,1,1,1,1],[0,1,0,1,0],[0,1,0,1,0],[1,0,0,0,1]]
Output: 2
Explanation: In the picture above, the grid on the left is grid1 and the grid on the right is grid2.
The 1s colored red in grid2 are those considered to be part of a sub-island. There are two sub-islands.

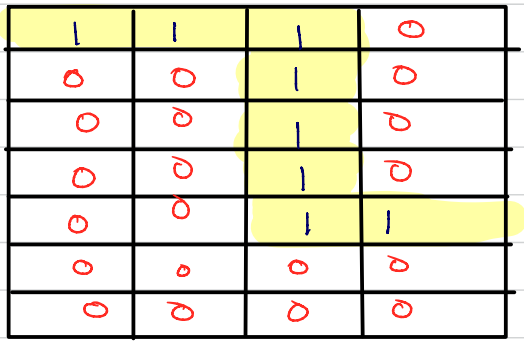
→ In this problem, we need to find the islands in grid2 are sub-islands of grid1 or not.

→ so we need to consider every island in grid2, and match them in grid1.

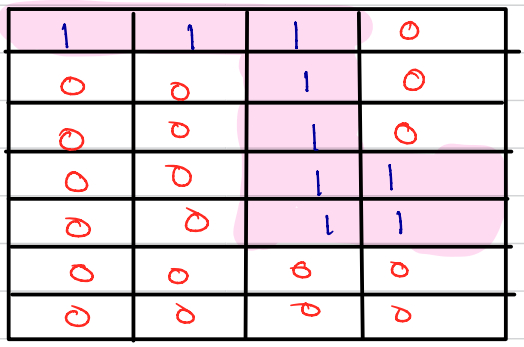
→ If the island in grid2 matches with the island in grid1, then the island will be consider as sub-island.

→ Even if it don't matches with grid1, traverse it completely and marked it visited, but it is not a sub-island of grid1.

→ We can use a boolean flag to return, whether it is sub-island or not.



grid1



grid2

NOTE :- grid2 is not sub-island of grid1, because we need to consider the complete island in grid2, not partially. But traverse the island in grid2 completely and mark flag = 0.

TC = $O(N \times M)$ + $O(N \times M \times 4)$
- $O(N \times M)$: calling dfs
- $O(N \times M \times 4)$: for every dfs/BFS call, calling its 4 neighbours [so, in total $4 \times N \times M$ times neighbours will be called]

$\simeq O(N \times M)$

SC :- $O(N \times M)$ + $O(N \times M)$
- $O(N \times M)$: Recursive stack space or queue space (BFS)
- $O(N \times M)$: visited array

```cpp
//Count Sub Islands
int rowdir[4] = {-1,0,1,0};
int coldir[4] = {0,-1,0,1};

void dfs(int row, int col, vector<vector<int>>& grid1, vector<vector<int>>&
grid2,vector<vector<int>>& vis,int& flag){
    int m = grid2.size();
    int n = grid2[0].size();
     if(grid1[row][col]!=1){
         flag=0;
     }
     vis[row][col]=1;
     for(int i=0;i<4;i++){
         int nr = row+rowdir[i];
         int nc = col+coldir[i];

         if(nr>=0 && nr<m && nc>=0 && nc<n && grid2[nr][nc]==1 && vis[nr][nc]==0){
             dfs(nr,nc,grid1,grid2,vis,flag);
         }
     }
        return;
     }

int countSubIslands(vector<vector<int>>& grid1, vector<vector<int>>& grid2) {
    int m = grid2.size();
    int n = grid2[0].size();
    int count =0;

    vector<vector<int>> vis(m, vector<int> (n,0));

     for(int i=0;i<m;i++){
         for(int j=0;j<n;j++){
             if(grid2[i][j]==1 && vis[i][j]==0){
                 int flag =1;
                 dfs(i,j,grid1,grid2,vis,flag);
                     count+=flag;
             }
         }
     }
   return count;
   }
```

1020. Number of Enclaves

Hint

Medium 1.8K 37

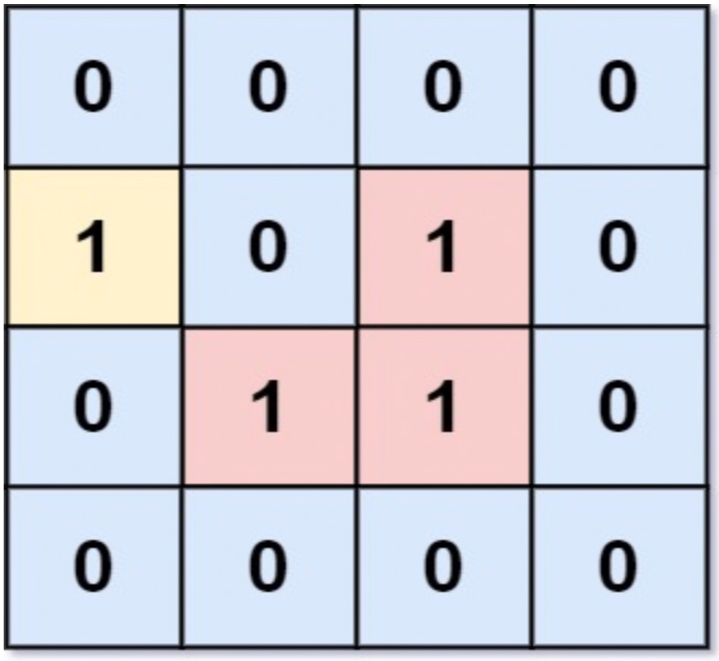
Companies

You are given an `m x n` 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`.

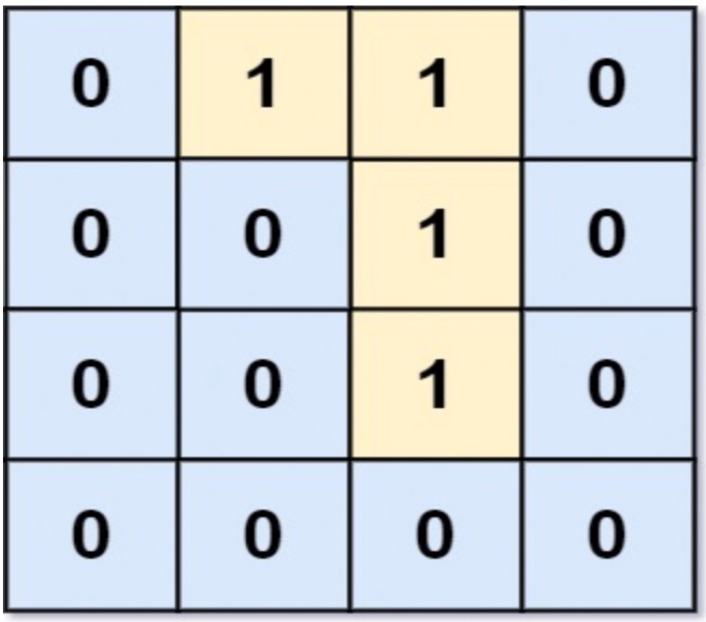
Return *the number of land cells in* `grid` *for which we cannot walk off the boundary of the grid in any number of* ***moves***.

Example 1:




```
Input: grid = [[0,0,0,0],[1,0,1,0],[0,1,1,0],[0,0,0,0]]
Output: 3
Explanation: There are three 1s that are enclosed by 0s, and one 1 that is not enclosed because its on the boundary.
```

Example 2:



Input: grid = [[0,1,1,0],[0,0,1,0],[0,0,1,0],[0,0,0,0]]
Output: 0
Explanation: All 1s are either on the boundary or can reach the boundary.

→ It is similar as number of closed islands.
→ But here 1 denotes to land and 0 denotes the water.

→ We need to find the closed enclave, so boundary connected lands cannot be closed enclave.

TC = O(M)+O(N) + O(N×M×4)
calling dfs ↵ ↳ for every dfs/BFS call, calling its 4 neighbours [so, in total 4×N×M times neighbours will be called]

≃ O(N×M)

SC :- O(N×M) + O(N×M)
↳ Recursive stack space or queue space (BFS) ↳ visited array

```cpp
//Number of Enclaves
int rowdir[4] = {-1,0,1,0};
int coldir[4] = {0,-1,0,1};
void dfs(int row,int col,vector<vector<int>>& vis,vector<vector<int>>& grid){
  int m = grid.size();
  int n = grid[0].size();

 if(row<0 || row>=m || col<0 || col>=n || grid[row][col]==0 || vis[row][col]==1)
    return;

    vis[row][col]=1;
    for(int i=0;i<4;i++){
        int nr = row+rowdir[i];
        int nc = col+coldir[i];
        dfs(nr,nc,vis,grid);
    }
}

int numEnclaves(vector<vector<int>>& grid) {
    int m = grid.size();
    int n = grid[0].size();
    int count =0;
    vector<vector<int>> vis(m,vector<int> (n,0));

    //travering the first row;
    for(int i=0;i<n;i++){
        if(grid[0][i]==1 && !vis[0][i]){
            dfs(0,i,vis,grid);
        }
    }

    //travering the last row;
    for(int i=0;i<n;i++){
        if(grid[m-1][i]==1 && !vis[m-1][i]){
            dfs(m-1,i,vis,grid);
        }
    }
    //traversing the first column;
    for(int i=0;i<m;i++){
        if(grid[i][0]==1 && !vis[i][0]){
            dfs(i,0,vis,grid);
        }
    }
    //traversing the last column;
    for(int i=0;i<m;i++){
        if(grid[i][n-1]==1 && !vis[i][n-1]){
            dfs(i,n-1,vis,grid);
        }
    }
  for(int i=0;i<m;i++){
    for(int j=0;j<n;j++){
       if(grid[i][j]==1 && !vis[i][j])
           count++;
      }
  }
    return count;
    }
```

```cpp
//Number of Enclaves Using BFS
int rowdir[4] = {-1,0,1,0};
int coldir[4] = {0,-1,0,1};

int numEnclaves(vector<vector<int>>& grid) {
  int m = grid.size();
  int n = grid[0].size();
  int count =0;
  vector<vector<int>> vis(m,vector<int> (n,0));
  queue<pair<int,int>> q;

  //travering the first row;
  for(int i=0;i<n;i++){
      if(grid[0][i]==1 && !vis[0][i]){
          vis[0][i]=1;
          q.push({0,i});
      }
  }

  //travering the last row;
  for(int i=0;i<n;i++){
      if(grid[m-1][i]==1 && !vis[m-1][i]){
          vis[m-1][i]=1;
          q.push({m-1,i});
      }
  }
  //traversing the first column;
  for(int i=0;i<m;i++){
      if(grid[i][0]==1 && !vis[i][0]){
          vis[i][0]=1;
          q.push({i,0});
      }
  }
  //traversing the last column;
  for(int i=0;i<m;i++){
      if(grid[i][n-1]==1 && !vis[i][n-1]){
           vis[i][n-1]=1;
          q.push({i,n-1});
      }
  }

  while(!q.empty()){
      int row = q.front().first;
      int col = q.front().second;
      q.pop();

      for(int i=0;i<4;i++){
      int nr = row+rowdir[i];
      int nc = col+coldir[i];
       if(nr<0 || nr>=m || nc<0 || nc>=n || grid[nr][nc]==0 || vis[nr][nc]==1)
       { continue;
       }
        vis[nr][nc]=1;
        q.push({nr,nc});
      }
  }

  for(int i=0;i<m;i++){
    for(int j=0;j<n;j++){
       if(grid[i][j]==1 && !vis[i][j])
           count++;
      }
  }
  return count;
}
```

994. Rotting Oranges

Medium 8.8K 319

Companies

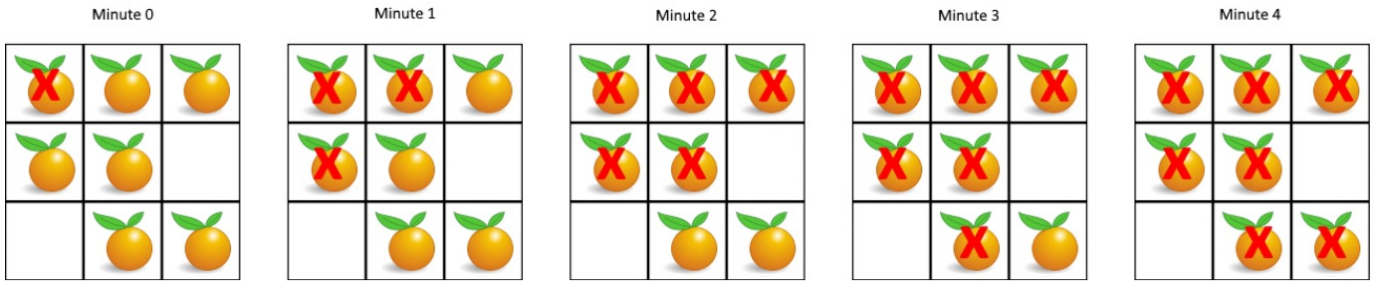
You are given an `m x n` `grid` where each cell can have one of three values:

- `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
```

Example 2:

```
Input: grid = [[2,1,1],[0,1,1],[1,0,1]]
Output: -1
Explanation: The orange in the bottom left corner (row 2, column 0) is
never rotten, because rotting only happens 4-directionally.
```

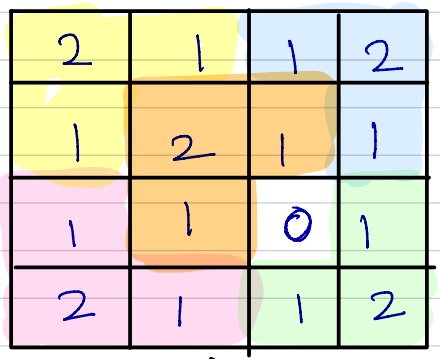
Example 3:

```
Input: grid = [[0,2]]
Output: 0
Explanation: Since there are already no fresh oranges at minute 0, the
answer is just 0.
```

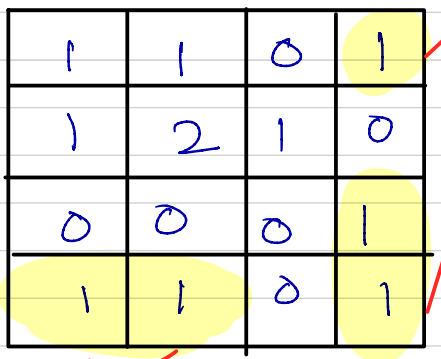
→ We have given the fresh oranges, rotten oranges and empty cell.

→ Rotten oranges rot the adjacent fresh oranges every single minute.

NOTE – a.) There can be more than one rotten orange initially.
b.) It is possible that not all fresh oranges are reachable by rotten oranges.



Case a.)



b.)

Unreachable

→ In case a, there are 4 rotten oranges, which can rot all other oranges in one minute.
So after 1 minute, there will be no fresh orange left.

→ In case b.), there are multiple fresh oranges which are unreachable from rotten oranges, so return -1.

→ Consider case a.) again, there are multiple rotten oranges, and each rotten orange impacting its neighbour orange at same time.

→ Here time is the constraint, we need to do it in minimum time.

→ So in these types of problems, when we need to find shortest path, minimum time, minimum cost etc, just try to think about BFS.
(Where each edge weight is same).

→ Here each move takes one minute, so we should use BFS.

→ Because we can consider all rotten oranges at one level on time $t=0$.

→ At next level, t=1, further oranges will be rotten and so on, untill all the reachable fresh oranges will be rotten.

→ At the end, traverse again the grid to check, whether all the fresh oranges get rotten or not.

Algorithm :-

1. Take a queue to store row no, col no and time stamp of a rotten orange.
2. Push all rotten oranges in queue initially.
3. Untill queue does not get empty :-
   - → Pop queue
   - → check fresh oranges in neighbour cells. (unvisited)
   - → push the row no, col no, t+1 in queue
   - → Mark neighbours visited.
4. Traverse grid to check if any fresh orange exists :-
   - → if yes, return -1.
   - → if no, return max timestamp.

$TC = O(N \times M) + O(N \times M \times 4)$

Calling BFS ↵

↳ for every BFS call, calling its 4 neighbours [so, in total $4 \times N \times M$ times neighbours will be called]

$\simeq O(N \times M)$

$SC :- O(N \times M) + O(N \times M)$

↳ queue space ↳ visited array

```cpp
//Rotting Oranges
int orangesRotting(vector<vector<int>>& grid) {
  int m = grid.size();
  int n = grid[0].size();
  vector<vector<int>> vis(m,vector<int> (n,0));
  queue<pair<pair<int,int>,int>> q;

  for(int i=0;i<m;i++){
      for(int j=0;j<n;j++){
          if(grid[i][j]==2){
            q.push({{i,j},0});
            vis[i][j]=2;
          }
      }
  }
  int maxt=0;
  int rowdir[] ={-1,0,1,0};
  int coldir[] ={0,-1,0,1};

  while(!q.empty()){
    int r = q.front().first.first;
    int c = q.front().first.second;
    int t = q.front().second;
    q.pop();
    maxt = max(maxt,t);

    for(int i=0;i<4;i++){
      int nr = r+rowdir[i];
      int nc = c+coldir[i];
      if(nr>=0 && nr<m && nc>=0 && nc<n && grid[nr][nc]==1 && vis[nr][nc]==0) {
          q.push({{nr,nc},t+1});
          vis[nr][nc]=2;
      }
    }
}
  for(int i=0;i<m;i++){
      for(int j=0;j<n;j++){
          if(grid[i][j]==1 && vis[i][j]==0){
            return -1;
          }
      }
  }
 return maxt;
}
```

```cpp
//Rotting Oranges
int orangesRotting(vector<vector<int>>& grid) {
  int m = grid.size();
  int n = grid[0].size();
  vector<vector<int>> vis(m,vector<int> (n,0));
  queue<pair<pair<int,int>,int>> q;

  int ones = 0;
  for(int i=0;i<m;i++){
    for(int j=0;j<n;j++){
        if(grid[i][j]==2){
          q.push({{i,j},0});
          vis[i][j]=2;
        }
        else if(grid[i][j]==1) ones++;
    }
  }
  int maxt=0;
  int rowdir[] ={-1,0,1,0};
  int coldir[] ={0,-1,0,1};

  int onesvisited =0;
  while(!q.empty()){
    int r = q.front().first.first;
    int c = q.front().first.second;
    int t = q.front().second;
    q.pop();
    maxt = max(maxt,t);

    for(int i=0;i<4;i++){
        int nr = r+rowdir[i];
        int nc = c+coldir[i];
        if(nr>=0 && nr<m && nc>=0 && nc<n && grid[nr][nc]==1 && vis[nr][nc]==0) {
           q.push({{nr,nc},t+1});
           vis[nr][nc]=2;
           onesvisited++; }
        }
  }

  if(ones!=onesvisited){
  return -1;
  }

  return maxt;
  }
```

797. All Paths From Source to Target

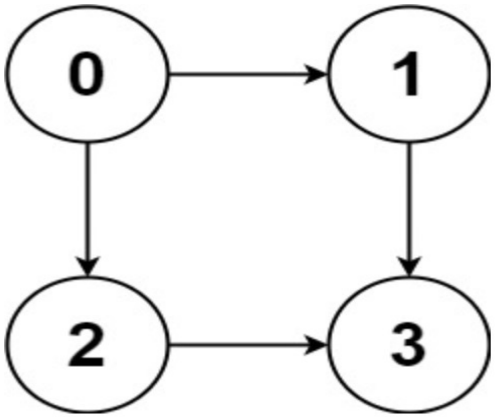
Medium 5.2K 123

Companies

Given a directed acyclic graph (**DAG**) of `n` nodes labeled from `0` to `n - 1`, find all possible paths from node `0` to node `n - 1` and return them in **any order**.

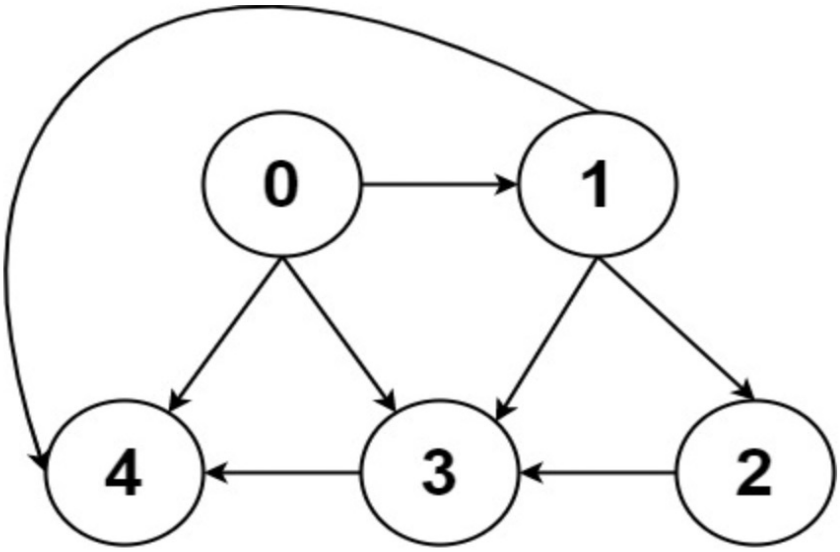
The graph is given as follows: `graph[i]` is a list of all nodes you can visit from node `i` (i.e., there is a directed edge from node `i` to node `graph[i][j]`).

Example 1:



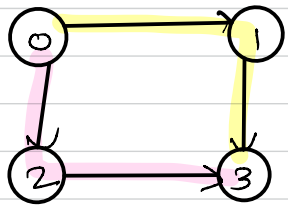

```
Input: graph = [[1,2],[3],[3],[]]
Output: [[0,1,3],[0,2,3]]
Explanation: There are two paths: 0 -> 1 -> 3 and 0 -> 2 -> 3.
```

Example 2:




```
Input: graph = [[4,3,1],[3,2,4],[3],[4],[]]
Output: [[0,4],[0,3,4],[0,1,3,4],[0,1,2,3,4],[0,1,4]]
```

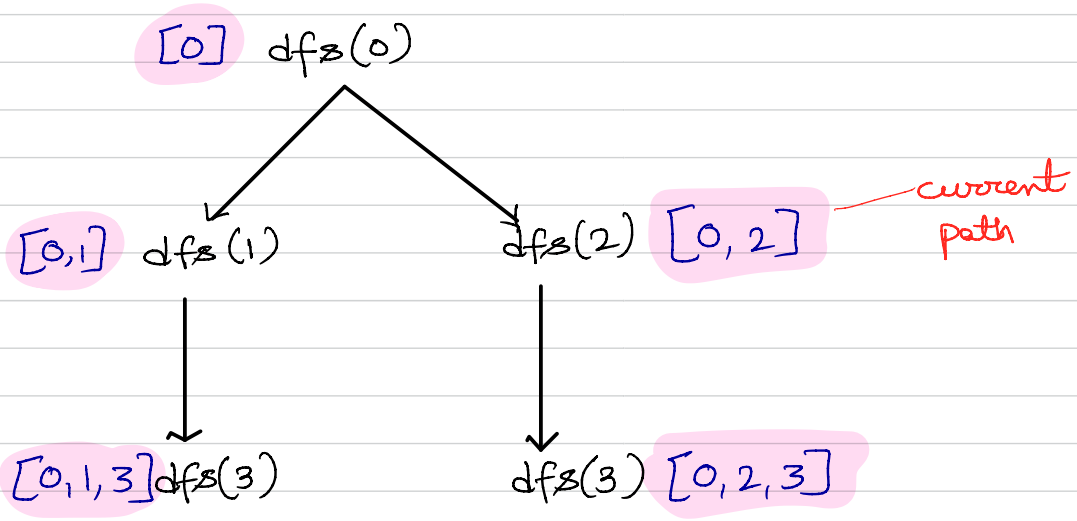
→ We need to find all paths from source to destination.



→ possible paths can be from 0 → 1 → 3 and 0 → 2 → 3.

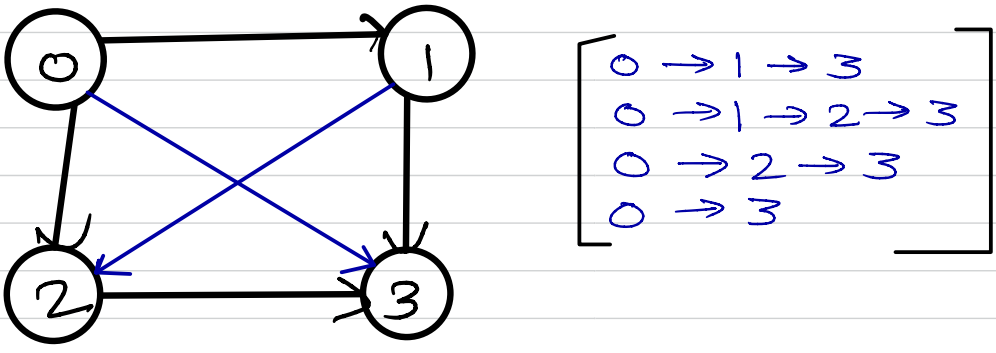
→ We need to go to depth untill we reach to target, so we will use dfs.

→ While going to depth, add current node to our current path, while coming back, remove current node to find other possible paths.



→ If our current node is target node, add current path in answer.

→ There can be atmax 2^N paths in DAG.



```cpp
//All Paths From Source to Target
vector<vector<int>> allPathsSourceTarget(vector<vector<int>>& graph) {
        vector<vector<int>> ans;
        int n = graph.size();
        vector<bool> visited(n);
        vector<int> currentpath;
        sourcetotarget(graph,0,visited,n,currentpath,ans);
        return ans;
    }
void sourcetotarget(vector<vector<int>>& graph,int currentindex,vector<bool>&visited,int n,vector<int> currentpath,vector<vector<int>>& ans)
 {
     if(currentindex==n-1)
     {    currentpath.push_back(currentindex);
          ans.push_back(currentpath);
          currentpath.pop_back();
          return;
     }
     if(visited[currentindex]==true)
         return;

     visited[currentindex]=true;
     currentpath.push_back(currentindex);

     for(int neighbour : graph[currentindex])
      sourcetotarget(graph,neighbour,visited,n,currentpath,ans);

      visited[currentindex] = false;
      currentpath.pop_back();
      return;
 }
```

TC = exponential.

= $O(2^N)$, where N is the number of nodes.

SC = $O(2^N)$, to store all the paths.

NOTE :- In Actual, the total possible paths will be less than $O(2^N)$, but 2^N is the upper limit.

841. Keys and Rooms

Medium 3.8K 195

Companies

There are `n` rooms labeled from `0` to `n - 1` and all the rooms are locked except for room `0`. Your goal is to visit all the rooms. However, you cannot enter a locked room without having its key.

When you visit a room, you may find a set of **distinct keys** in it. Each key has a number on it, denoting which room it unlocks, and you can take all of them with you to unlock the other rooms.

Given an array `rooms` where `rooms[i]` is the set of keys that you can obtain if you visited room `i`, return `true` *if you can visit **all** the rooms, or* `false` *otherwise.*

Example 1:

```
Input: rooms = [[1],[2],[3],[]]
Output: true
Explanation:
We visit room 0 and pick up key 1.
We then visit room 1 and pick up key 2.
We then visit room 2 and pick up key 3.
We then visit room 3.
Since we were able to visit every room, we return true.
```

Example 2:

```
Input: rooms = [[1,3],[3,0,1],[2],[0]]
Output: false
Explanation: We can not enter room number 2 since the only key that
unlocks it is in that room.
```

→ We will enter room 0, from room 0 we can visit rooms whose keys are these in room 0.

→ So, keep entering the rooms untill we don't get any key in any room or all the keys are used.

ex-2 → 0 → 1 → 3 } → We cannot enter room 2.
0 → 3

→ So, to keep track of entered room, take a visited array.
→ At the end, traverse the visited array,
if all rooms are visited return true
else false.

```cpp
//Keys and Rooms
void dfs(int ind, vector<vector<int>>& rooms,vector<int>& vis){
    if(vis[ind]) return;
    vis[ind]=1;
    for(auto it : rooms[ind]){
        if(!vis[it]){
            dfs(it,rooms,vis);
        }
    }
}

bool canVisitAllRooms(vector<vector<int>>& rooms) {
  int m = rooms.size();
  vector<int> vis(m,0);
  dfs(0,rooms,vis);

  for(int i=0;i<m;i++){
      if(!vis[i])
          return false;
  }
  return true;
}
```

TC = O(N+E), where N is number of rooms and E is number of keys.

SC = O(N) + O(N)
- O(N): visited
- O(N): recursive stack space

→ We can use BFS as well.

130. Surrounded Regions

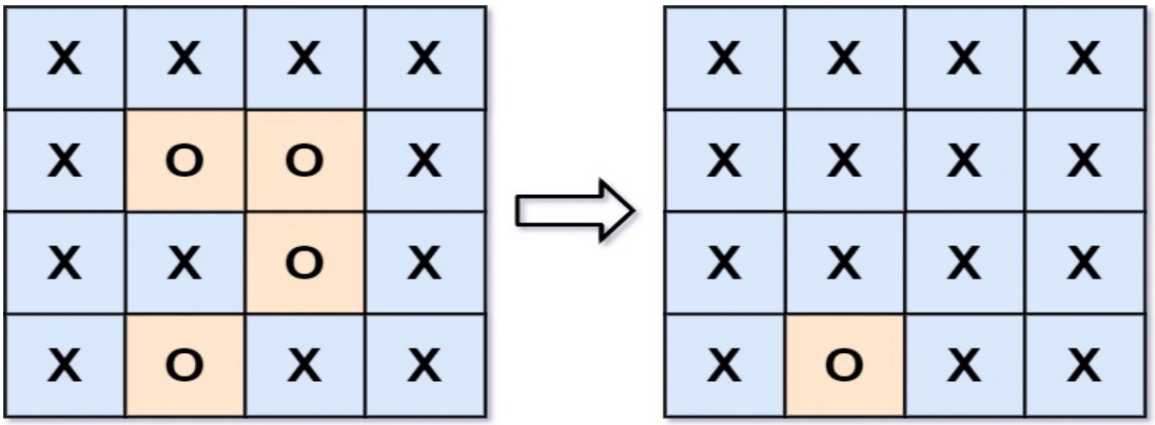
Medium 6K 1.4K

Companies

Given an `m x n` matrix `board` containing `'X'` and `'O'`, *capture all regions that are 4-directionally surrounded by* `'X'`.

A region is **captured** by flipping all `'O'`s into `'X'`s in that surrounded region.

Example 1:




```
Input: board = [["X","X","X","X"],["X","O","O","X"],["X","X","O","X"],
["X","O","X","X"]]
Output: [["X","X","X","X"],["X","X","X","X"],["X","X","X","X"],
["X","O","X","X"]]
Explanation: Notice that an 'O' should not be flipped if:
- It is on the border, or
- It is adjacent to an 'O' that should not be flipped.
The bottom 'O' is on the border, so it is not flipped.
The other three 'O' form a surrounded region, so they are flipped.
```

Example 2:

```
Input: board = [["X"]]
Output: [["X"]]
```

→ It is similar to Number of closed islands and No. of enclaves.

→ Here we don't need to count the numbers, but we need to replace surrounded region with X.

→ Start from boundary zeroes and mark them visited.
→ Traverse the matrix again and replace all unvisited 'O' with X.
→ We can use BFS/DFS to traverse the nodes.

TC = O(M) + O(N) + O(N×M×4)
calling dfs ↵ ↳ for every dfs/BFS call, calling its 4 neighbours [so, in total 4×N×M times neighbours will be called]

≃ O(N×M)

SC :- O(N×M) + O(N×M)
↳ Recursive stack space or queue space (BFS) ↳ visited array

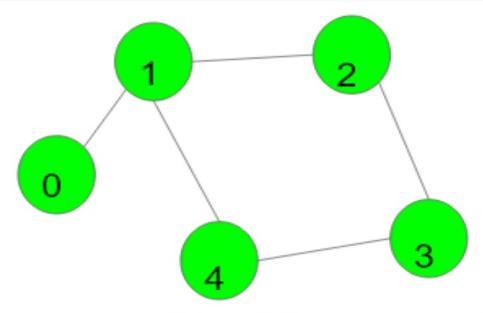
```cpp
//Surrounded Regions
int rowdir[4] = {-1,0,1,0};
int coldir[4] = {0,-1,0,1};

void dfs(int row,int col,vector<vector<int>>& vis,vector<vector<char>>& grid){
    int m = grid.size();
    int n = grid[0].size();
    if(row<0 || row>=m || col<0 || col>=n || grid[row][col]=='X'
       || vis[row][col]==1){
        return;
    }

    vis[row][col]=1;
    for(int i=0;i<4;i++){
        int nr = row+rowdir[i];
        int nc = col+coldir[i];
        dfs(nr,nc,vis,grid);
   }
}
```

```cpp
//Surrounded Regions


void solve(vector<vector<char>>& board) {
    int m = board.size();
    int n = board[0].size();
    vector<vector<int>> vis(m, vector<int> (n,0));

    //traversing first column
    for(int i=0;i<m;i++){
       if(board[i][0]=='O' && !vis[i][0]){
           dfs(i,0,vis,board);
       }
    }
    //traversing last column
    for(int i=0;i<m;i++){
       if(board[i][n-1]=='O' && !vis[i][n-1]){
           dfs(i,n-1,vis,board);
       }
    }
    //traversing first row
    for(int i=0;i<n;i++){
       if(board[0][i]=='O' && !vis[0][i]){
           dfs(0,i,vis,board);
       }
    }
    //traversing last row
     for(int i=0;i<n;i++){
       if(board[m-1][i]=='O' && !vis[m-1][i]){
           dfs(m-1,i,vis,board);
       }
    }
    for(int currrow=0;currrow<m;currrow++){
        for(int currcol=0;currcol<n;currcol++){
            if(board[currrow][currcol]=='O' && !vis[currrow][currcol]){
                board[currrow][currcol]='X';
            }
        }
    }
}
```

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

Example 1:

Input:
V = 5, E = 5
adj = {{1}, {0, 2, 4}, {1, 3}, {2, 4}, {1, 3}}
Output: 1
Explanation:



1->2->3->4->1 is a cycle.

→ Given a graph having vertex v from 0 to V-1. We need to find whether the graph have cycle or not.

NOTE:- There can be multiple connected components.

→ We can use BFS/DFS to solve the problem.

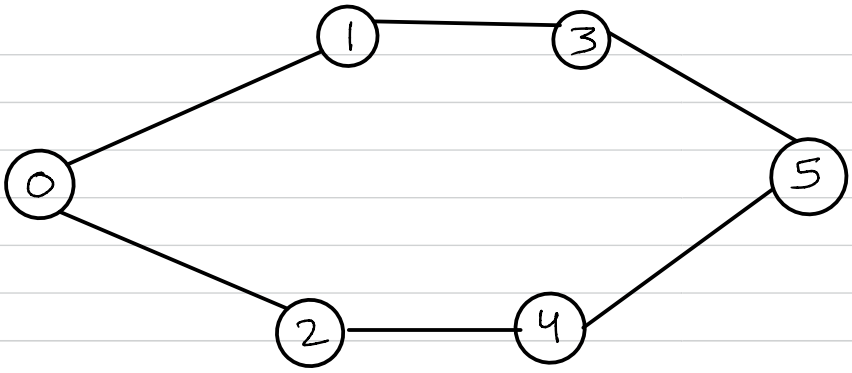
1. Find the cycle using BFS.

→ The idea is to traverse nodes in BFS manner and since it is undirected graph, so Adjacency list will contain same edge two times, from A→B and from B→A.

→ So we need to keep record of parent node. If a node is previously visited, we need to check whether its is parent node or not.

→ If a node is visited, but not a parent of current node, its means, the current node is already visited by other node, so it forms a cycle.

Ex –



Adjacency list of graph

0 → {1, 2}

1 → {0, 3}

2 → {0, 4}

3 → {1, 5}

4 → {2, 5}

5 → {3, 4}

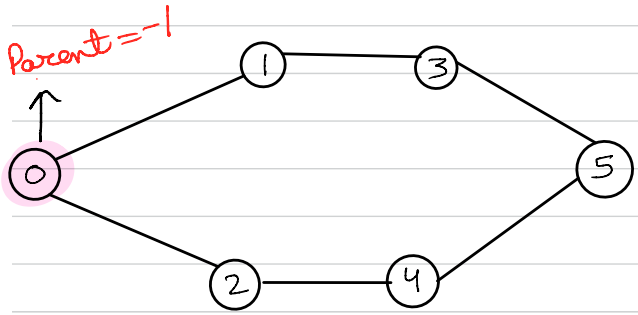
→ Start with 0, parent of 0 is null, so take it as -1.

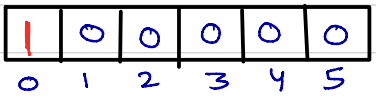
Algorithm :-

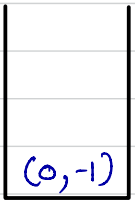
→ Take a visited array of size V. (0 to V-1).
→ Take a queue of pair (node, parent).
→ Push source node and parent in queue, mark it visited.
⇒ while queue does not empty
  → Traverse the neighbours of current node.
  → If neighbours are not visited, push it in queue,
  → if visited and is not parent of current node, then it contains cycle, return True.

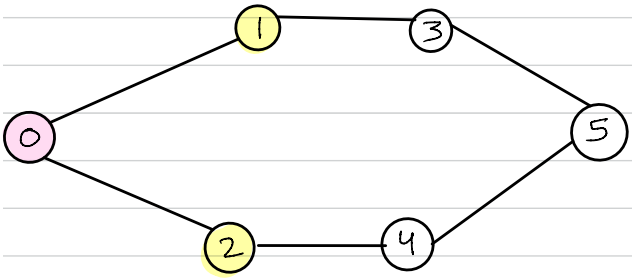
→ Return false, it does not contain cycle.

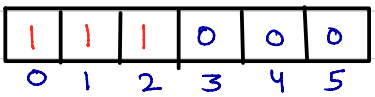
→ denotes current node
→ denotes unvisited neighbouring nodes.
→ denotes already visited nodes.

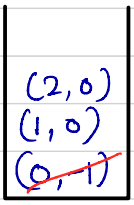

Parent = -1
0
1
3
5
2
4

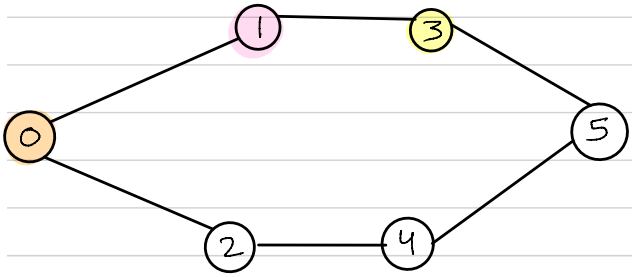

1 0 0 0 0 0
0 1 2 3 4 5


(0,-1)

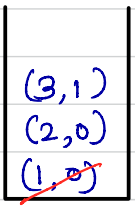

0
1
3
5
2
4

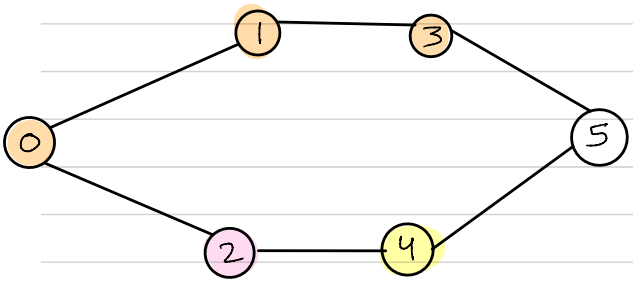

1 1 1 0 0 0
0 1 2 3 4 5


(2,0)
(1,0)
(0,-1)

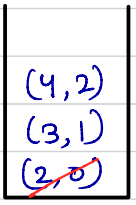

0
1
3
5
2
4

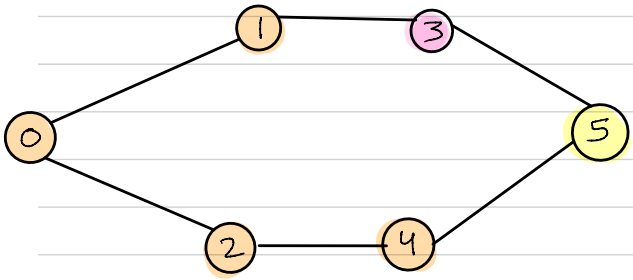

1 1 1 1 0 0
0 1 2 3 4 5


(3,1)
(2,0)
(1,0)

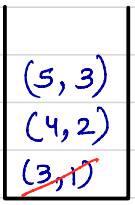

0
1
3
5
2
4

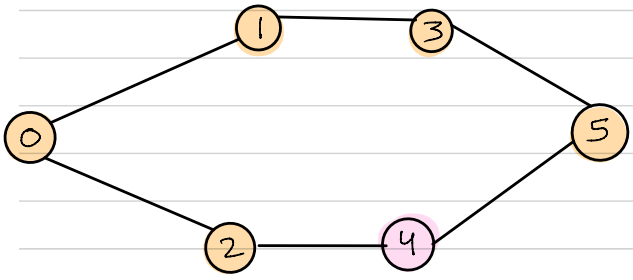

1 1 1 1 1 0
0 1 2 3 4 5


(4,2)
(3,1)
(2,0)

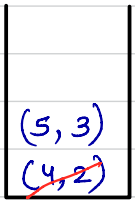

0
1
3
5
2
4


1 1 1 1 1 1
0 1 2 3 4 5


(5,3)
(4,2)
(3,1)


0
1
3
5
2
4


1 1 1 1 1 1
0 1 2 3 4 5
already visited


(5,3)
(4,2)

```cpp
//Function to detect cycle in an undirected graph using BFS
bool Checkcyclebfs(int src,vector<int> adj[],vector<bool>& visited){
        queue<pair<int,int>> q;
        q.push({src,-1});
        visited[src] = 1;

        while(!q.empty()){
            int currele = q.front().first;
            int parent = q.front().second;
            q.pop();

            for(auto adjacent : adj[currele]){
                if(!visited[adjacent]){
                    visited[adjacent]=1;
                    q.push({adjacent,currele});
                }
                else if(visited[adjacent] && parent!=adjacent)
                    return true;
            }
        }
     return false;
    }

    bool isCycle(int V, vector<int> adj[]) {
        vector<bool> visited(V,0);
       for(int i=0;i<V;i++){
           if(!visited[i]){
               if(Checkcyclebfs(i,adj,visited)==1) return true;
           }
           }
       return false;
    }
```

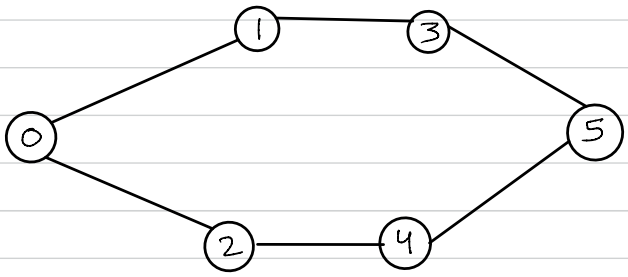
$TC = O(V + 2E) + O(V)$

$SC = O(V) + O(V)$

$O(V)$ → queue

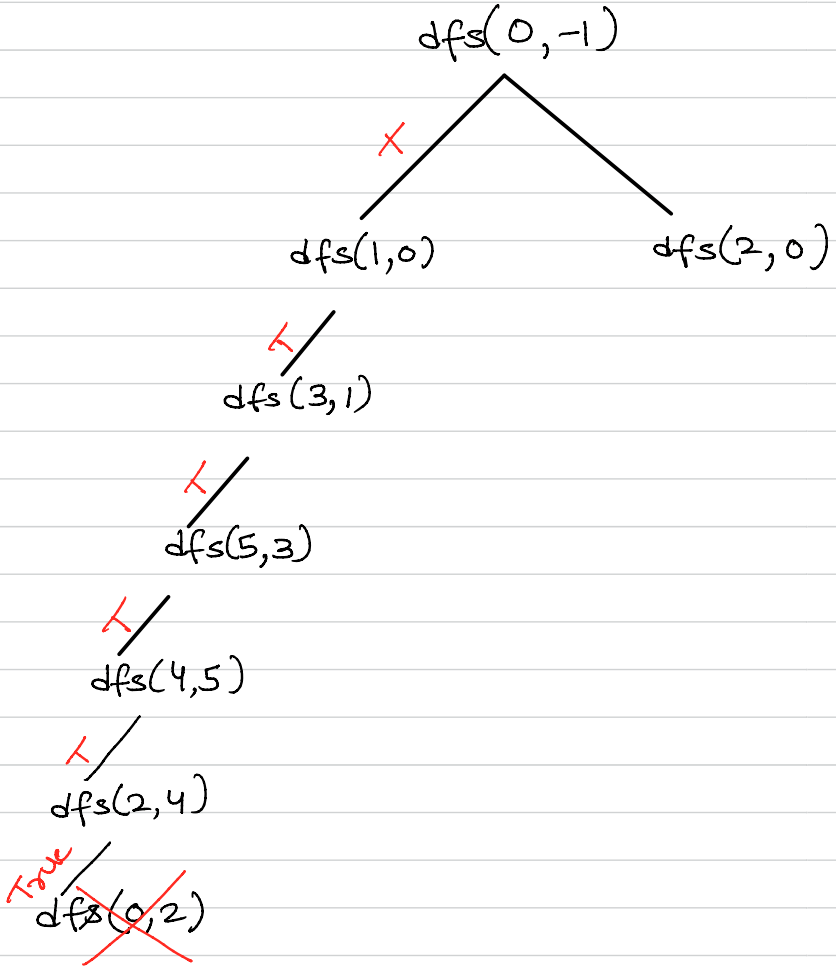
$O(V)$ → visited

Detect cycle in Undirected graph using DFS :-



Adjacency list of graph

0 → {1, 2}
1 → {0, 3}
2 → {0, 4}
3 → {1, 5}
4 → {2, 5}
5 → {3, 4}



```cpp
// Function to detect cycle in an undirected graph.
bool Checkcycledfs(int src,vector<int> adj[],int parent,vector<bool> visited){
    visited[src]=1;

    for(auto adjacent : adj[src]){
        if(!visited[adjacent]){
            if(Checkcycledfs(adjacent,adj,src,visited)==true)
            return true;
        }
        else if(adjacent!=parent)
                return true;
    }
return false;
}


bool isCycle(int V, vector<int> adj[]) {
    vector<bool> visited(V,0);
    int parent = -1;
    for(int i=0;i<V;i++){
       if(!visited[i]){
           if(Checkcycledfs(i,adj,parent,visited)==true)
             return true;
       }
       }
   return false;
}
```

$TC = O(V) + O(V+2E)$

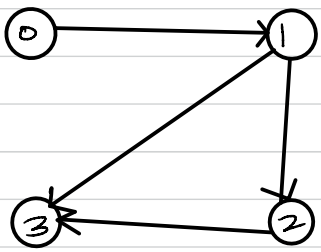
$SC = O(V) + O(V)$

↓ recursive stack space

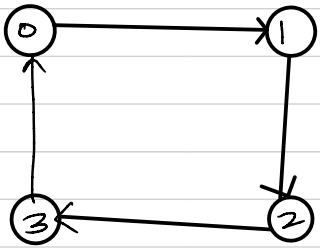
↳ visited

DETECT A CYCLE IN A DIRECTED GRAPH

Given a Directed Graph with V vertices (0 to V-1), and E edges, check whether it contains any cycle or not.



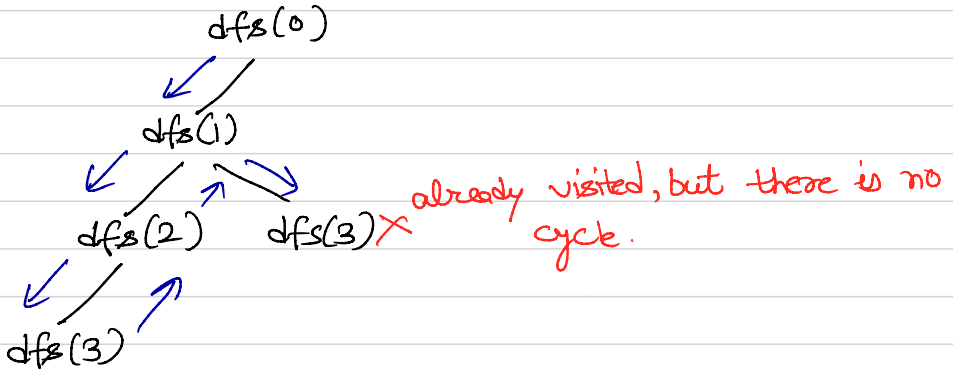
Ex - 1



Ex - 2

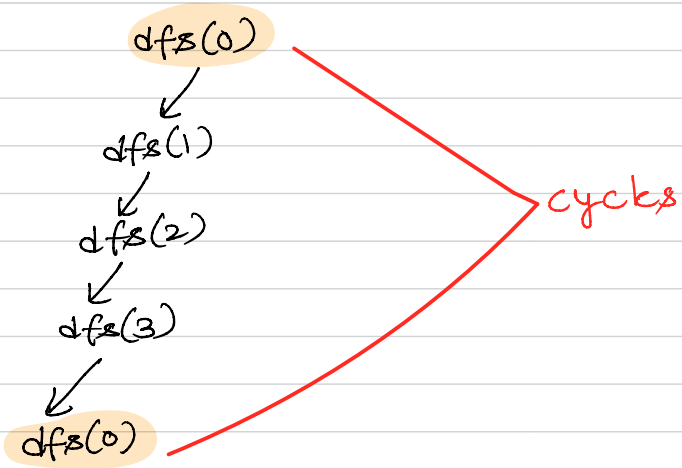
→ We have already done cycle detection in undirected graph using DFS. But the same algorithm will not work here.

→ Let's understand using ex-1.



→ In directed graph, direction also matters, so above algo will not work.

→ Consider ex-2,



→ so on the same path, if the node is visited again, then only its contains cycle in directed graph.

→ Along with visited array, we need a pathvisited array to find whether the visited node is on current path or not.

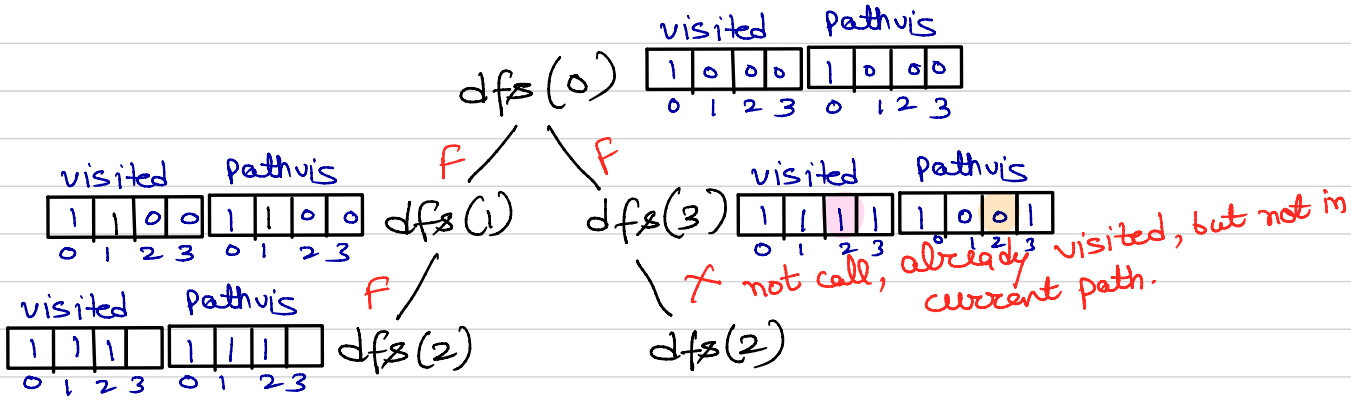
Ex –

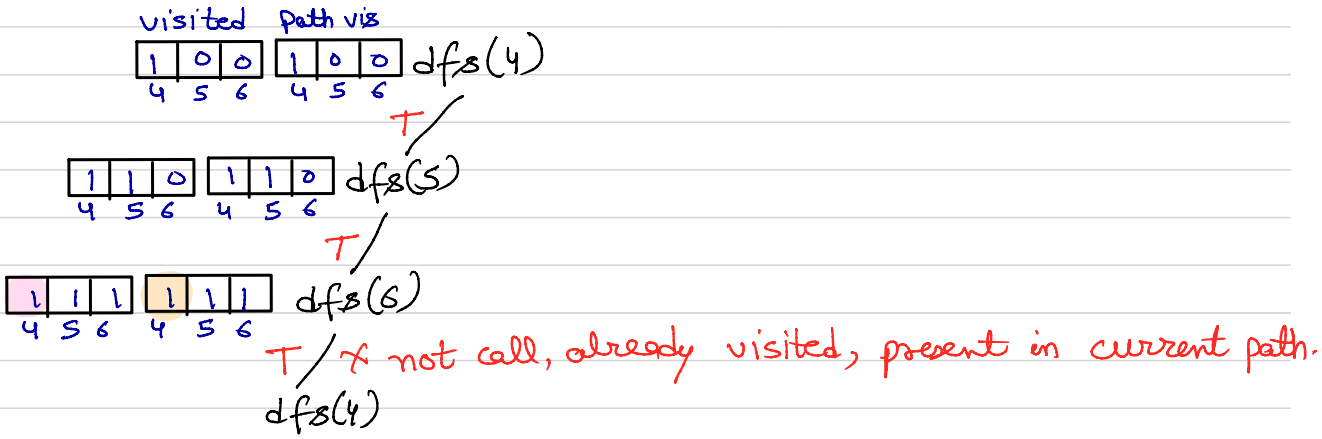
visited

0	0	0	0	0	0	0
0	1	2	3	4	5	6

pathvis

0	0	0	0	0	0	0
0	1	2	3	4	5	6





```cpp
class Solution {
  public:
    // Function to detect cycle in a directed graph.
    bool dfs(int node,vector<int>& vis, vector<int>&
pathvis,vector<int> adj[]){
        pathvis[node]=1;
        vis[node]=1;

        for(auto adjacent : adj[node]){
            if(!vis[adjacent]){
                if(dfs(adjacent,vis,pathvis,adj)==true)
                    return true;
            }
            else if(pathvis[adjacent]==1) return true;
        }
        pathvis[node]=0;
        return false;
    }

    bool isCyclic(int V, vector<int> adj[]) {
        vector<int> pathvis(V,0);
        vector<int> vis(V,0);
        for(int i=0;i<V;i++){
            if(!vis[i]){
                if(dfs(i,vis,pathvis,adj)==true){
                    return true;
                }
            }
        }
        return false;
    }
};
```

$TC = O(V+E)$

$SC = O(2V)$.

```cpp
// Function to detect cycle in a directed graph optimizing SC
bool dfs(int node,vector<int>& vis,vector<int> adj[]){
        vis[node]=2;

        for(auto adjacent : adj[node]){
            if(!vis[adjacent]){
                if(dfs(adjacent,vis,adj)==true)
                    return true;
            }
            else if(vis[adjacent]==2) return true;
        }
        vis[node]=1;
        return false;
    }

    bool isCyclic(int V, vector<int> adj[]) {
        vector<int> vis(V,0);
        for(int i=0;i<V;i++){
            if(!vis[i]){
                if(dfs(i,vis,adj)==true){
                    return true;
                }
            }
        }
        return false;
    }
```

802. Find Eventual Safe States

Medium 3K 342

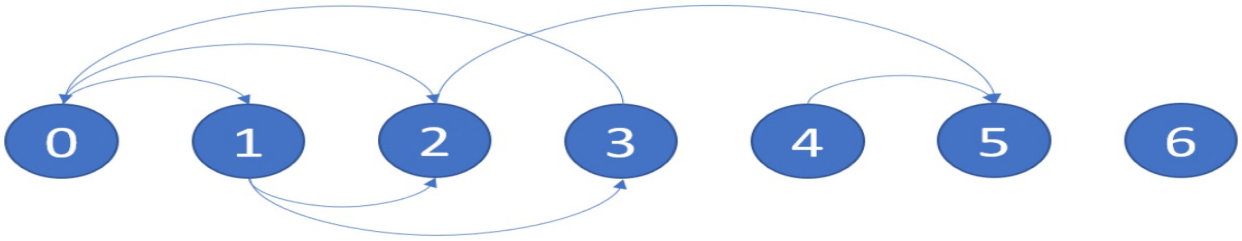
Companies

There is a directed graph of `n` nodes with each node labeled from `0` to `n - 1`. The graph is represented by a **0-indexed** 2D integer array `graph` where `graph[i]` is an integer array of nodes adjacent to node `i`, meaning there is an edge from node `i` to each node in `graph[i]`.

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** (or another safe node).

Return *an array containing all the **safe nodes** of the graph*. The answer should be sorted in **ascending** order.

Example 1:



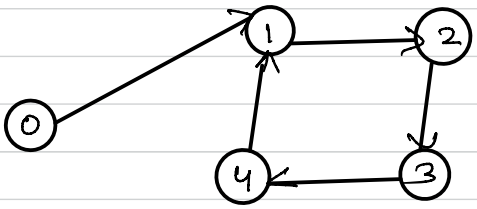

```
Input: graph = [[1,2],[2,3],[5],[0],[5],[],[]]
Output: [2,4,5,6]
Explanation: The given graph is shown above.
Nodes 5 and 6 are terminal nodes as there are no outgoing edges from
either of them.
Every path starting at nodes 2, 4, 5, and 6 all lead to either node 5 or
6.
```

Example 2:

```
Input: graph = [[1,2,3,4],[1,2],[3,4],[0,4],[]]
Output: [4]
Explanation:
Only node 4 is a terminal node, and every path starting at node 4 leads
to node 4.
```

→ Terminal nodes are nodes having outdegree = 0.
→ Safe nodes are nodes, from every path leads to a terminal node.
→ We have done cycle detection in directed graph using dfs in above problem. we will use the same in this problem.

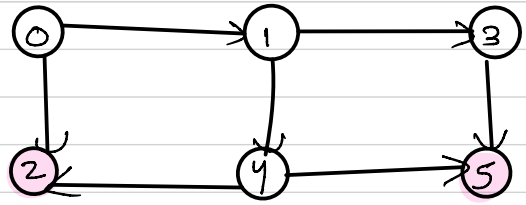
→ Any node which is a part of cycle will never be a safe node.
→ Any node which is connected to a cycle will also not a safe node.



→ 1, 2, 3, 4 contains cycle, but 0 is also connected with the cycle. So, 0, 1, 2, 3, 4 cannot be safe node.

→ If we reach to the nodes, which doesn't contain cycle or does not follow a path containing cycle, we can say that node a safe node.

ex –

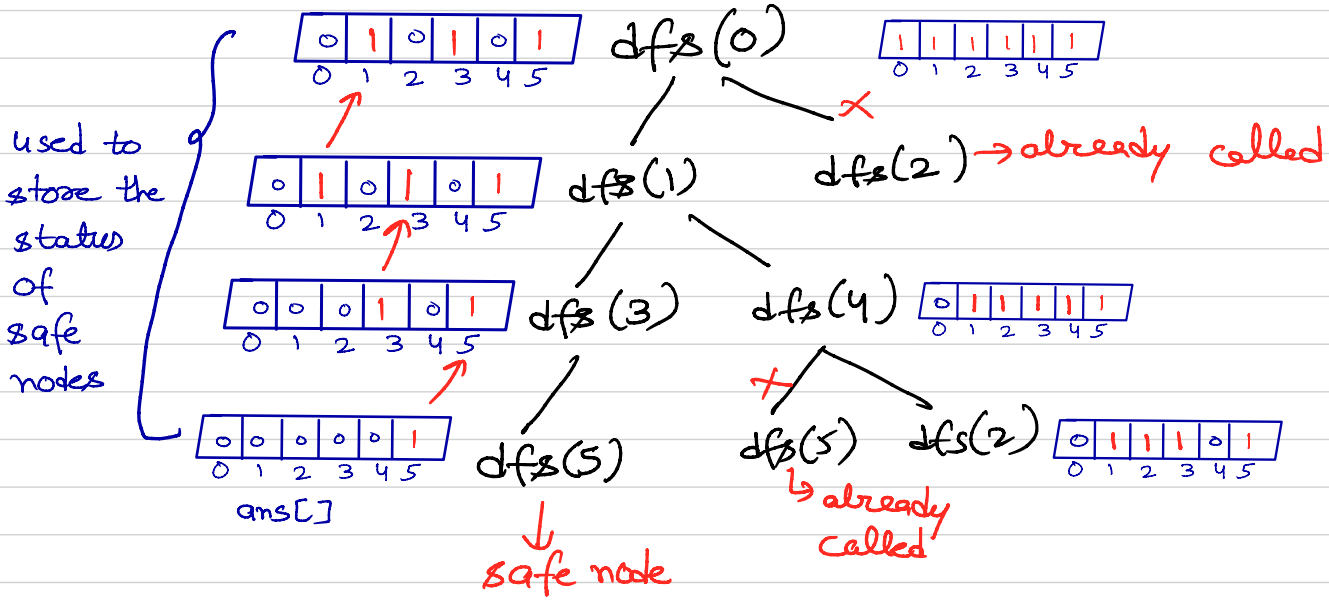


→ 2 and 5 are terminal nodes, there is no cycle in the graph and every path is either ends at 2 or 5.

$0 \to 1 \to 3 \to 5$
$0 \to 1 \to 4 \to 5$
$0 \to 1 \to 4 \to 2$
$0 \to 2$

} all paths reaching terminal nodes.

→ we need an array to store the safenodes after every dfs call. We name it as ans



```cpp
//Find Eventual Safe States Using DFS
bool dfs(int node,vector<vector<int>>& graph,vector<int>& vis,vector<int>&
currpath,vector<int>& ans){
    vis[node] = 1;
    currpath[node]=1;

    for(auto it: graph[node]){
        if(!vis[it]){
            if(dfs(it,graph,vis,currpath,ans)==true)
            {   ans[node]=0;
                return true;
            }
        }
        else if(currpath[it]) return true;
    }
    ans[node]=1;
    currpath[node]=0;
    return false;
}

vector<int> eventualSafeNodes(vector<vector<int>>& graph) {
    int m = graph.size();
    vector<int> vis(m,0);
    vector<int> currpath(m,0);
    vector<int> ans(m,0);
    vector<int> safenodes;
    for(int i=0;i<m;i++){
        if(!vis[i]){
            dfs(i,graph,vis,currpath,ans);
        }
    }
    for(int i=0;i<m;i++){
        if(ans[i]==1) safenodes.push_back(i);
    }
 return safenodes;
}
```

$TC = O(V+E)$

$SC = O(4V)$

542. 01 Matrix

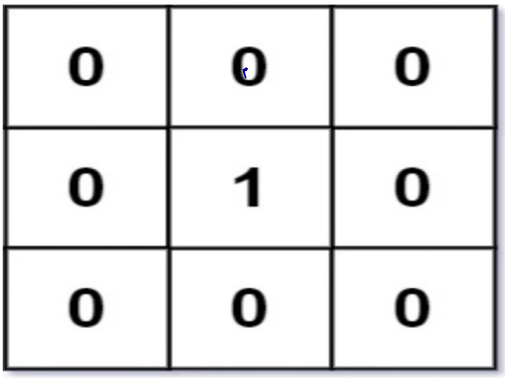
Medium 5.9K 292

Companies

Given an `m x n` binary matrix `mat`, return *the distance of the nearest* `0` *for each cell.*

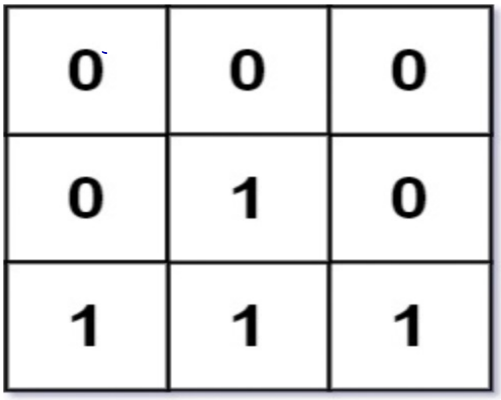
The distance between two adjacent cells is `1`.

Example 1:




```
Input: mat = [[0,0,0],[0,1,0],[0,0,0]]
Output: [[0,0,0],[0,1,0],[0,0,0]]
```

Example 2:




```
Input: mat = [[0,0,0],[0,1,0],[1,1,1]]
Output: [[0,0,0],[0,1,0],[1,2,1]]
```

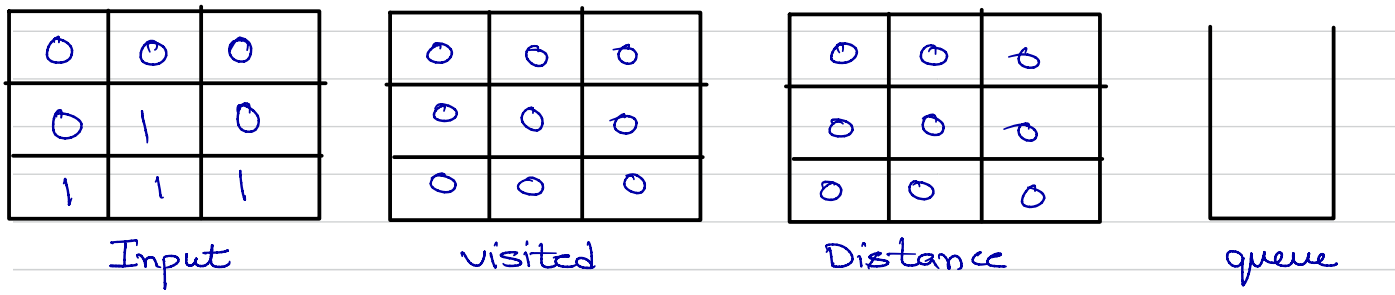
→ If we carefully observe, it's a shortest distance finding problem, where we need to find nearest 0 from each cell.

→ We need to find nearest 0, so it become easy, if we find nearest 1 from 0, because distance from 0 to 0 will be 0.

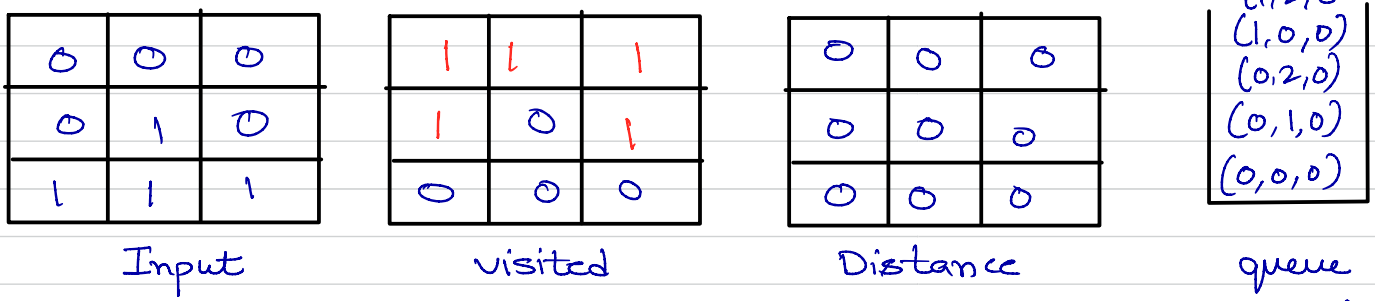
NOTE:- There are multiple 0's in the grid, so we need to consider all zeroes to calculate minimum distance.

→ The better approach will be BFS, because it will check level wise level. first all zeroes with 0 distance, then all 1's having one distance, and so on.

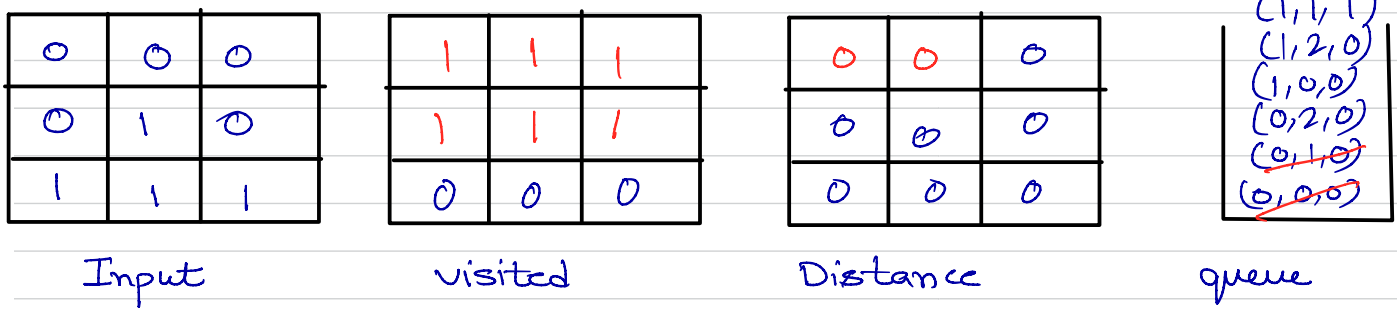
Algorithm:
1. Take a (m × n) distance array.
2. Take a (m×n) visited array.
3. Take a queue storing row no, col no and distance
4. store all the entries having 0 in the queue. Mark them visited
5. while queue does not empty :-
   → pop current element
   → update dist array.
   → check if the neighbours are unvisited,
   if yes, push the row no, col no and distance+1 i.e & mark it visited.



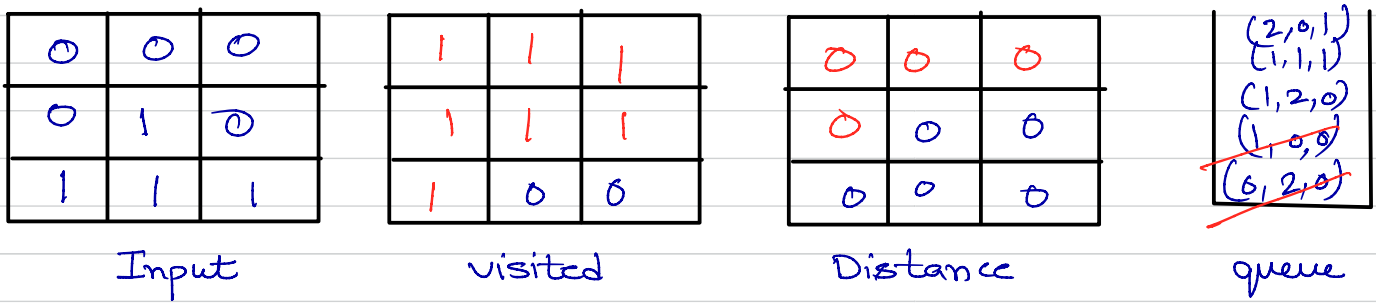
Input visited Distance queue



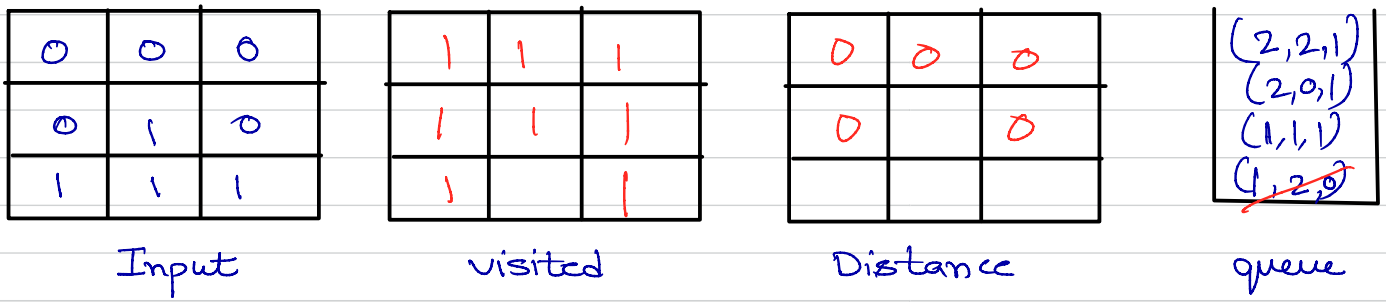
Input visited Distance queue

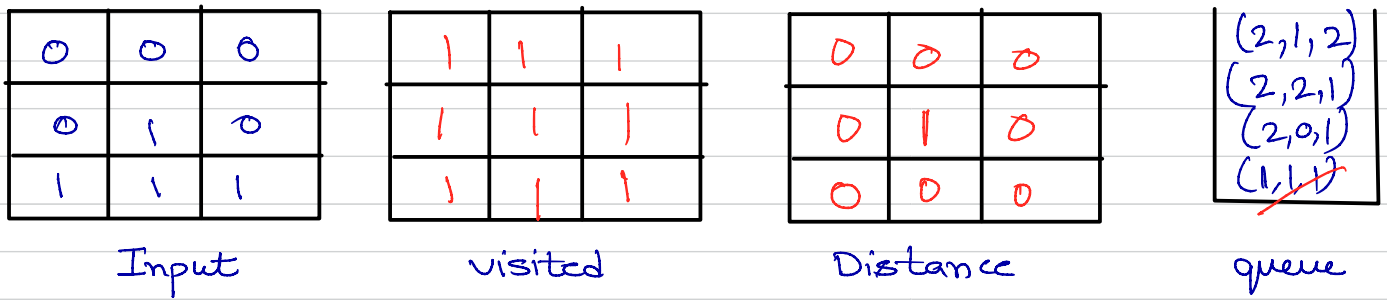


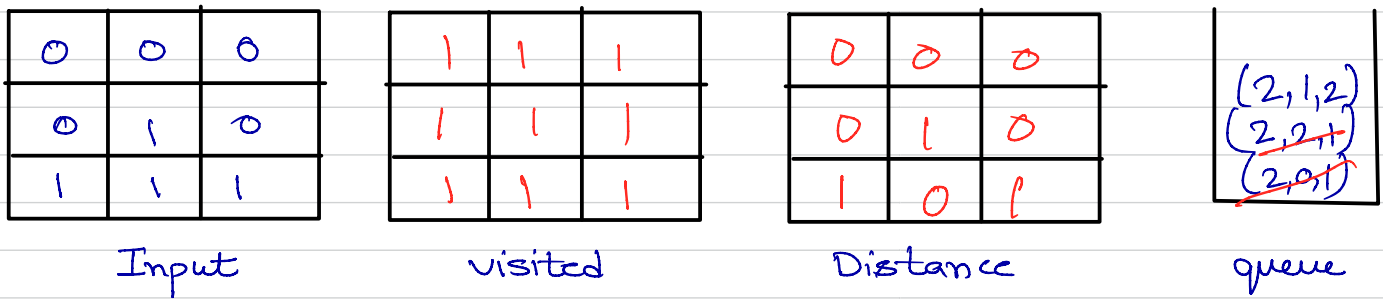
Input visited Distance queue

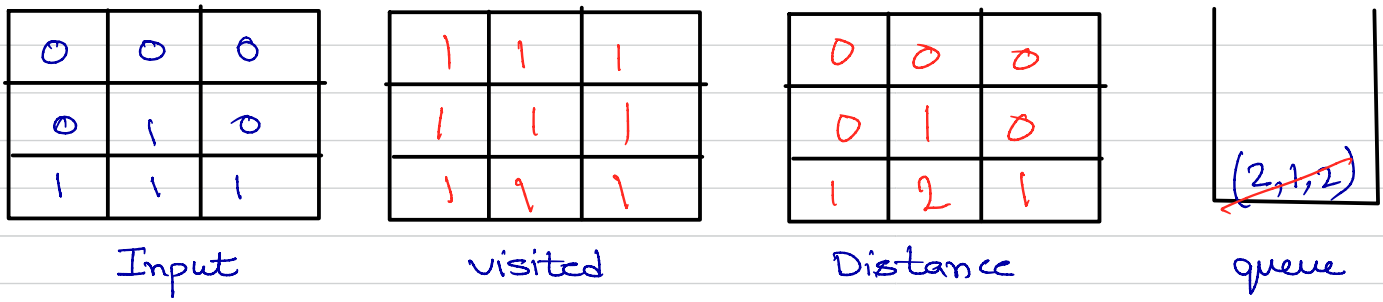


Input visited Distance queue









$$TC = O(M \times N) + O(M \times N \times 4)$$

$$SC = \underbrace{O(M \times N)}_{\text{distance}} + \underbrace{O(M \times N)}_{\text{visited}}$$

```cpp
// 01 Matrix
vector<vector<int>> updateMatrix(vector<vector<int>>& mat) {
  int m = mat.size();
  int n = mat[0].size();

  vector<vector<int>> visited(m, vector<int> (n,0));
  vector<vector<int>> dist(m, vector<int> (n,0));

  queue<pair<pair<int,int>,int>> q;
  for(int i=0;i<m;i++){
      for(int j=0;j<n;j++){
          if(mat[i][j]==0){
              q.push({{i, j},0});
              visited[i][j]=1;
          }
      }
  }

   int rowdir[4] = {-1,0,1,0};
   int coldir[4] = {0,-1,0,1};
    while(!q.empty()){
    int row = q.front().first.first;
    int col = q.front().first.second;
    int count=q.front().second;
    q.pop();
    dist[row][col] = count;

    for(int i=0;i<4;i++){
        int nr = row + rowdir[i];
        int nc = col + coldir[i];
     if(nr>=0 && nr<m && nc>=0 && nc<n && visited[nr][nc]==0){
         visited[nr][nc]=1;
         q.push({{nr,nc},count+1});
     }
    }
}

return dist;
}
```

417. Pacific Atlantic Water Flow

Medium 5.8K 1.1K

Companies

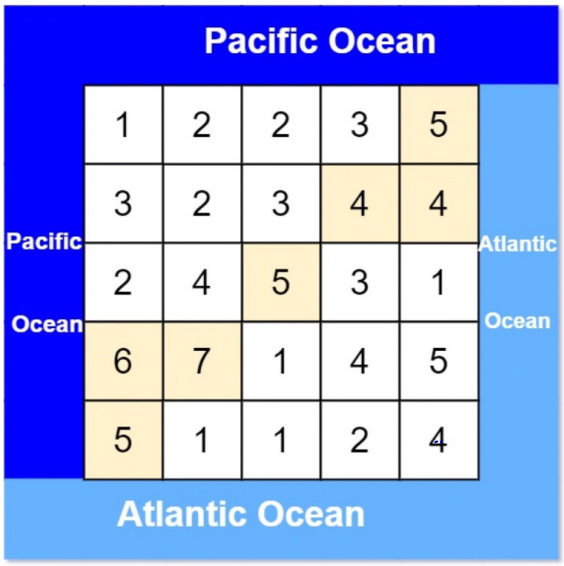
There is an `m x n` rectangular island that borders both the **Pacific Ocean** and **Atlantic Ocean**. The **Pacific Ocean** touches the island's left and top edges, and the **Atlantic Ocean** touches the island's right and bottom edges.

The island is partitioned into a grid of square cells. You are given an `m x n` integer matrix `heights` where `heights[r][c]` represents the **height above sea level** of the cell at coordinate `(r, c)`.

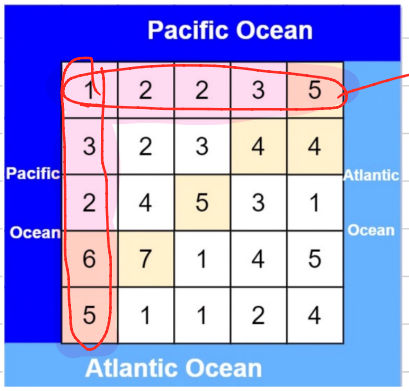
The island receives a lot of rain, and the rain water can flow to neighboring cells directly north, south, east, and west if the neighboring cell's height is **less than or equal to** the current cell's height. Water can flow from any cell adjacent to an ocean into the ocean.

Return *a **2D list** of grid coordinates* `result` *where* `result[i] = [$r_i$, $c_i$]` *denotes that rain water can flow from cell* `($r_i$, $c_i$)` *to **both** the Pacific and Atlantic oceans.*

Example 1:

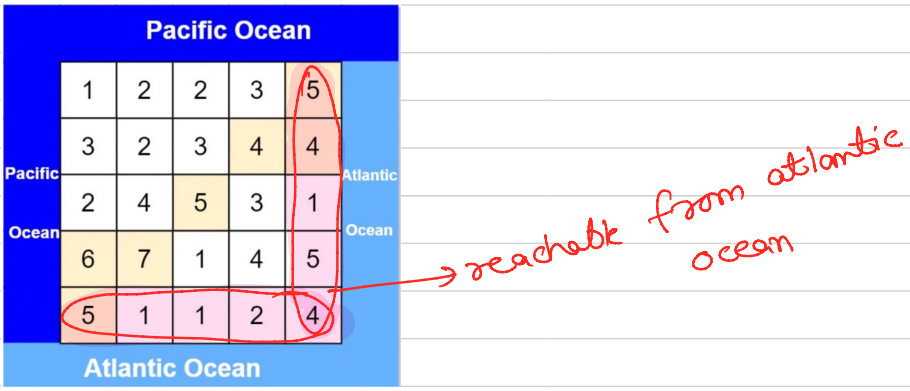


→ Pacific ocean is connected with top and left edges. So, row = 0 cells and column = 0 cells are reachable from pacific ocean.

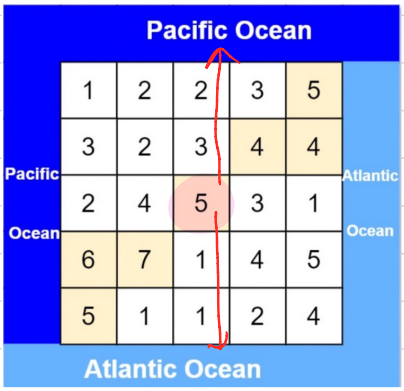


connected with pacific ocean

→ Atlantic Ocean is connected with right and down edges, so last row and last column cells are reachable from atlantic ocean.



→ Now, we need to find the cells, from which water can flow to both the oceans.
But the condition is the water can flow from higher cell to lower cell.
for ex -



from 5, water can flow in both the oceans.

→ Now the idea is to find whether the water will flow from particular cell or not individually using Two - 2d grids.

→ One 2-d grid is required to store the answer of pacific ocean and another 2-d grid is required to store the answer of atlantic ocean.

→ Atlast we will traverse and compare each cell of both the grids, if water can flow from both the grids, we will store the row no & column no in a 2d vector.

Now, How to find whether water can flow from particular cell or not?

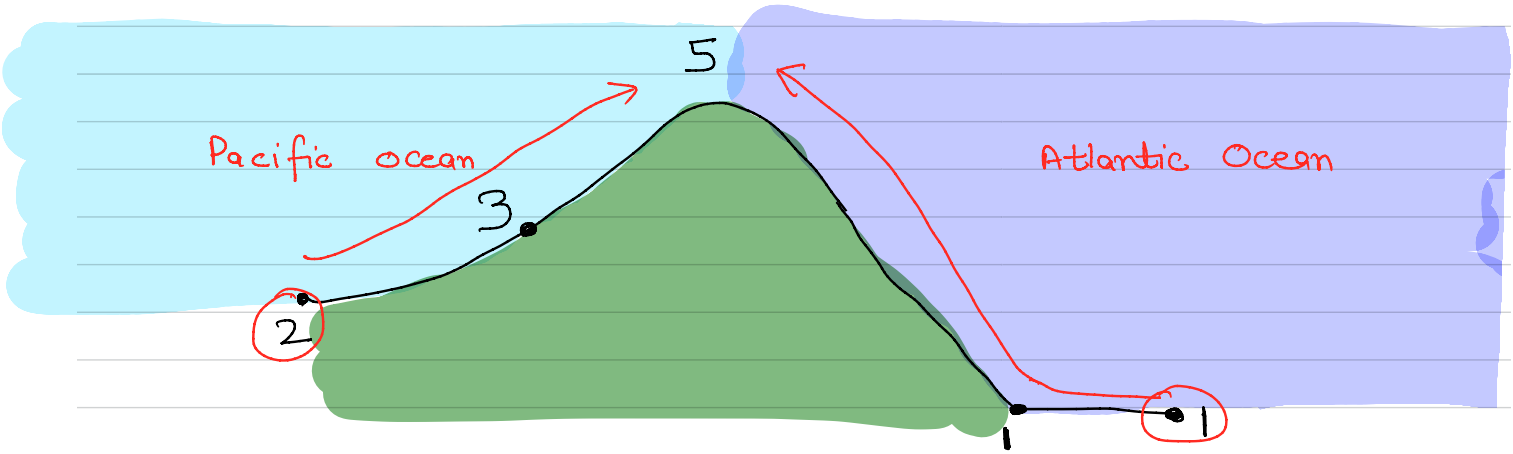
→ We can start with the edges cell, for pacific ocean start with first row & first column cells. Water can flow from these cells.

→ Now, check whether the water can flow from their neighbour cells or not.

→ Take a queue & store all the left & top edges cell in it and start traversing in BFS manner.

→ BFS will traverse for all the cells from where water can come to current cell.

→ We are going from lower cell value to upper cell value, means



→ Either we can go from ocean to cells having higher value or cells to ocean by going through lower value.

→ Our idea is to start from edge cells and reach to the cells having value equal to or greater than current cell.

1	1	1	1	1
1	1	1	1	1
1	1	1	0	0
1	1	0	0	0
1	0	0	0	0

Water flow towards pacific ocean

0	0	0	0	1
0	0	0	1	1
0	0	1	1	1
1	1	1	1	1
1	1	1	1	1

Water flow towards atlantic ocean

0	0	0	0	1
0	0	0	1	1
0	0	1	0	0
1	1	0	0	0
1	0	0	0	0

```cpp
//Pacific Atlantic Water Flow
void bfs(queue<pair<int,int>>& q,vector<vector<int>>&
ocean,vector<vector<int>>& heights){
  int m = heights.size();
  int n = heights[0].size();

  int rowdir[4] = {-1,0,1,0};
  int coldir[4] = {0,-1,0,1};

  while(!q.empty()) {
      int currrow = q.front().first;
      int currcol = q.front().second;
      q.pop();

      for(int i=0;i<4;i++){
          int nr = currrow+rowdir[i];
          int nc = currcol+coldir[i];
          if(nr>=0 && nr<m && nc>=0 && nc<n && ocean[nr][nc]==-1 &&
heights[nr][nc]>=heights[currrow][currcol]){
              q.push({nr,nc});
              ocean[nr][nc]=1;
          }
      }

  }
 return ;
  }
```

```cpp
//Pacific Atlantic Water Flow
vector<vector<int>> pacificAtlantic(vector<vector<int>>& heights) {
  int m = heights.size();
  int n = heights[0].size();
  queue<pair<int,int>> q;
  vector<vector<int>> pacific(m, vector<int> (n,-1));
  vector<vector<int>> atlantic(m, vector<int> (n,-1));

  //traverse first row for pacific ocean
  for(int i=0;i<n;i++){
      pacific[0][i]=1;
       q.push({0,i});
  }
  //traverse first col for pacific ocean
  for(int i=1;i<m;i++){
      pacific[i][0]=1;
      q.push({i,0});
  }
  bfs(q,pacific,heights);

//traverse last row for atlantic ocean
  for(int i=0;i<n;i++){
      atlantic[m-1][i]=1;
       q.push({m-1,i});
  }
  //traverse last col for atlantic ocean
  for(int i=0;i<m-1;i++){
      atlantic[i][n-1]=1;
      q.push({i,n-1});
  }
  bfs(q,atlantic,heights);

  vector<vector<int>> ans;
  for(int i=0;i<m;i++){
      for(int j=0;j<n;j++){
          if(atlantic[i][j]==1 && pacific[i][j]==1){
              ans.push_back({i,j});
          }
      }
  }
return ans;
}
```

1129. Shortest Path with Alternating Colors

Hint

Medium 1.6K 80

Companies

You are given an integer `n`, the number of nodes in a directed graph where the nodes are labeled from `0` to `n - 1`. Each edge is red or blue in this graph, and there could be self-edges and parallel edges.

You are given two arrays `redEdges` and `blueEdges` where:

- `redEdges[i] = [a_i, b_i]` indicates that there is a directed red edge from node `a_i` to node `b_i` in the graph, and
- `blueEdges[j] = [u_j, v_j]` indicates that there is a directed blue edge from node `u_j` to node `v_j` in the graph.

Return an array `answer` of length `n`, where each `answer[x]` is the length of the shortest path from node `0` to node `x` such that the edge colors alternate along the path, or `-1` if such a path does not exist.

Example 1:

```
Input: n = 3, redEdges = [[0,1],[1,2]], blueEdges = []
Output: [0,1,-1]
```

Example 2:

```
Input: n = 3, redEdges = [[0,1]], blueEdges = [[2,1]]
Output: [0,1,-1]
```

→ There are red and blue edges on graph.

→ There can be both red and blue edges on graph.

→ We need to find the shortest distance from 0 to every vertex, but the condition is the edges color should be in alternate order, means either Red Blue Red Blue Red, or Blue Red Blue Red Blue.

→ We have given red and Blue edges in separate vectors.

→ We can create a combined adjacency list containing edges from red and blue edges array.

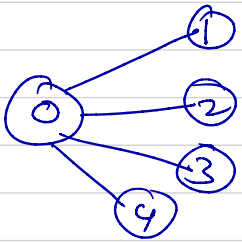
→ But wait, if we combine Red & blue edges array, then how we will identify, which one is connected to red edge & which one is connected to Blue.

→ We can store the edges in a form of pair containing the vertex and its color, lets denote red as −1 and Blue edge as 1.

→ So if a edge from 0 to 1 is red edge, we can store it like 0 → {1, -1}

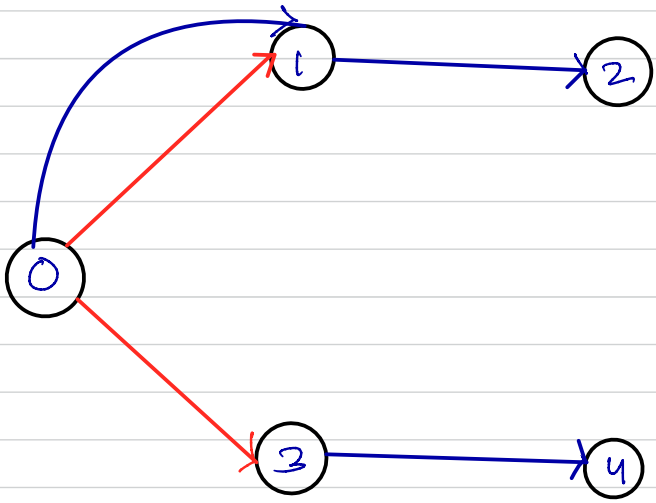
→ Now starting from 0, run BFS and check for alternate color, maintain a answer vector to store shortest path.

→ Take a visited array of size V. One node can be visited V times, because there can be at max V different paths like :-



here, 0 will visit 4 times.

ex →



Adjacency list

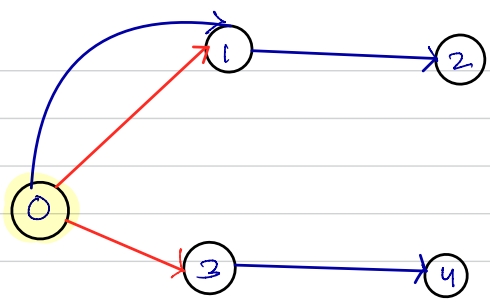
0 → { {1, -1} {1, 1} {3, -1}

1 → { {2, 1} }

2 → { }

3 → { {4, 1} }

4 → { }

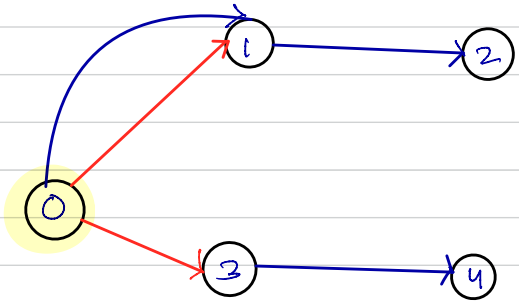


INT MAX

ans

INT MAX	INT MAX	INT MAX	INT MAX	INT MAX
0	1	2	3	4

(0, 0) → Level-1

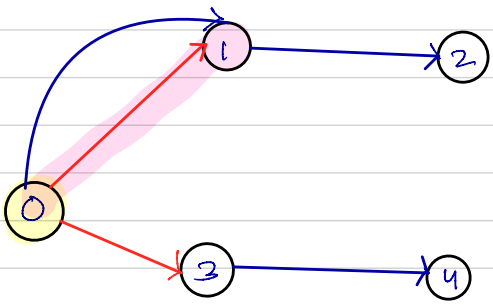


ans

0	INT MAX	INT MAX	INT MAX	INT MAX
0	1	2	3	4

(3, -1)
(1, 1)
(1, -1)
(0, 0)

} Level-2



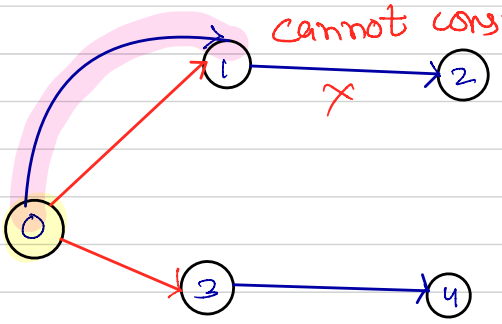
ans

0	1	INT MAX	INT MAX	INT MAX
0	1	2	3	4

(2, 1) → Level-3
(3, -1)
(1, 1)
(1, -1)

} Level-2

cannot consider 1-2

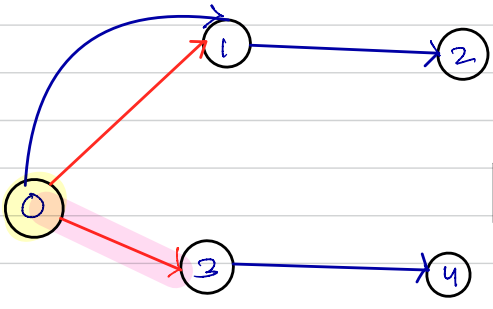


ans

0	1	INT MAX	INT MAX	INT MAX
0	1	2	3	4

(2, 1) → Level-3
(3, -1)
(1, 1)

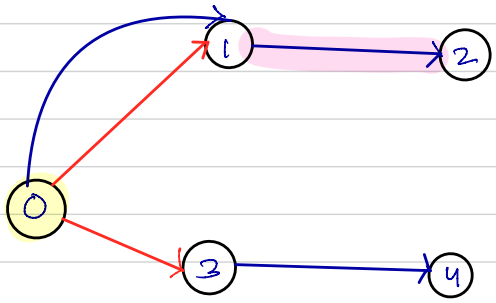
} Level-2



ans

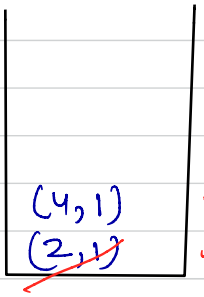
0	1	INT MAX	1	INT MAX
0	1	2	3	4

(4, 1)
(2, 1)
(3, -1)

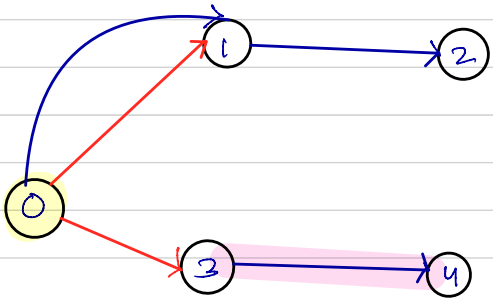


ans

0	1	2	1	INT MAX
0	1	2	3	4

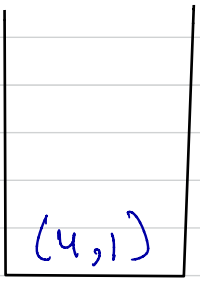


Level - 3



ans

0	1	2	1	2
0	1	2	3	4



Level - 3

```cpp
//Shortest Path with Alternating Colors
vector<int> shortestAlternatingPaths(int n, vector<vector<int>>& redEdges,
vector<vector<int>>& blueEdges) {

  vector<vector<pair<int,int>>> adj;
  adj.resize(n);
  int rs = redEdges.size();
  int bs = blueEdges.size();
  //representing rededges from -1 and  bluedges from 1;
  for(int i=0;i<rs;i++){
      adj[redEdges[i][0]].push_back({redEdges[i][1],-1});
  }
   for(int i=0;i<bs;i++){
      adj[blueEdges[i][0]].push_back({blueEdges[i][1],1});
  }

  vector<int> visited(n,0);
  vector<int> ans(n,INT_MAX);
  queue<pair<int,int>> q;
  int count =0;

  q.push({0,0});
  visited[0]+=1;
```

```cpp
while(!q.empty()){
  int qs = q.size();

  for(int i=0;i<qs;i++){
      int node = q.front().first;
      int color = q.front().second;
      q.pop();

      ans[node]=min(ans[node],count);

      for(int i=0;i<adj[node].size();i++)
      {
       if(visited[adj[node][i].first]<=n && adj[node][i].second!=color){
           visited[adj[node][i].first]+=1;
           q.push({adj[node][i].first,adj[node][i].second});
         }
        }

}
  count++;

}
for(int &el : ans) {
    if(el == INT_MAX) {
        el = -1;
    }
}
       return ans;
      }

```

1466. Reorder Routes to Make All Paths Lead to the City Zero

Hint

Medium 1.9K 51

Companies

There are `n` cities numbered from `0` to `n - 1` and `n - 1` roads such that there is only one way to travel between two different cities (this network form a tree). Last year, The ministry of transport decided to orient the roads in one direction because they are too narrow.

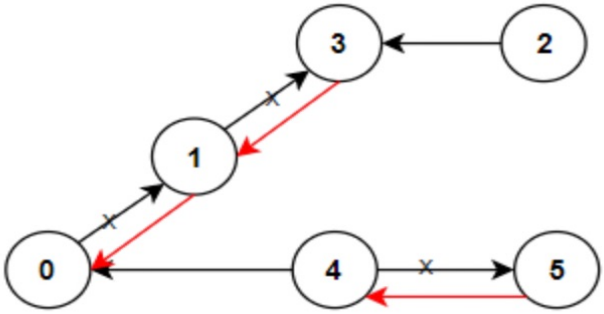
Roads are represented by `connections` where `connections[i] = [a_i, b_i]` represents a road from city `a_i` to city `b_i`.

This year, there will be a big event in the capital (city `0`), and many people want to travel to this city.

Your task consists of reorienting some roads such that each city can visit the city `0`. Return the **minimum** number of edges changed.

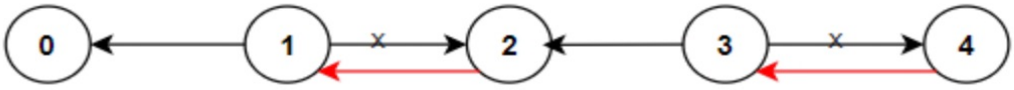
It's **guaranteed** that each city can reach city `0` after reorder.

Example 1:




```
Input: n = 6, connections = [[0,1],[1,3],[2,3],[4,0],[4,5]]
Output: 3
Explanation: Change the direction of edges show in red such that each
node can reach the node 0 (capital).
```

Example 2:




```
Input: n = 5, connections = [[1,0],[1,2],[3,2],[3,4]]
Output: 2
Explanation: Change the direction of edges show in red such that each
node can reach the node 0 (capital).
```

→ Since, we need to find minimum no. of edge changes, the approach comes in my mind is using BFS.

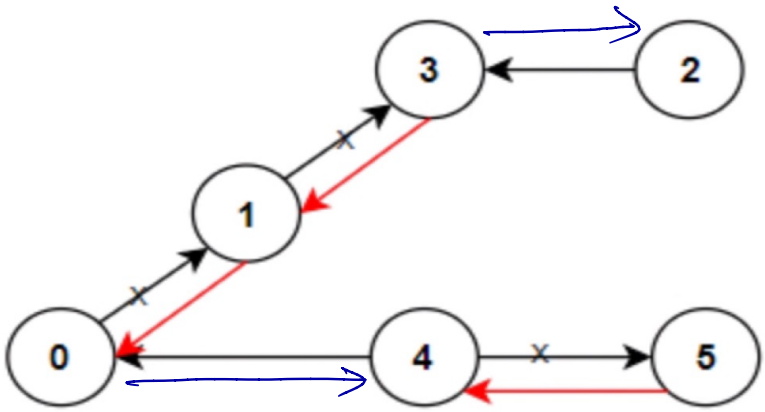
→ We can create a Bidirectional graph and the edges which are going away from 0 needs to change.

→ We will take an adjacency list containing bidirectional graph.
→ To compare which edges needs to change, we need original graph edges, so store original graph edges into a set.

→ Start traversing from node 0, and if the edges present from 0 to its neighbouring nodes, we need to change them, so count them.

→ Basically, we are originating BFS from 0, so we going away from 0, and if we are getting edges, which are also moving away from 0, we need to reverse them.

→ In ex-1, 4→5 is moving away from 0, so we need to reverse it.



Adjacency list

0 → {1, 4}

1 → {0, 3}

2 → {3}

3 → {1, 2}

4 → {5, 0}

5 → {4}

→ Store the original graph edges in a set:-

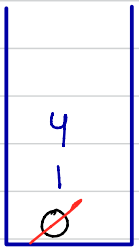
{ (0,1), (1, 3), (2,3), (4,5), (4,0) }

→ Take a queue & push 0

→ {0,1} present in set, so need to reverse it.

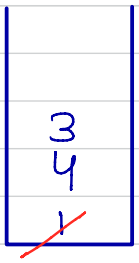
→ {0,4} not present in set, so no need to reverse it.

(1)



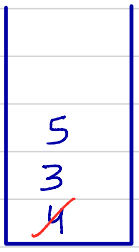
→ {1,3} present in set, so need to reverse it.

(2)



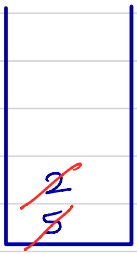
→ {4,5} present in set, so need to reverse it.

(3)



→ {3,2} not present in set, so no need to reverse it.

→ all neighbours visited.



```cpp
//1466. Reorder Routes to Make All Paths Lead to the City Zero
int minReorder(int n, vector<vector<int>>& connections) {
  vector<vector<int>> adj(n);
  int cs = connections.size();
  set<pair<int,int>> s;

  for(int i=0;i<cs;i++){
      adj[connections[i][0]].push_back(connections[i][1]);
      adj[connections[i][1]].push_back(connections[i][0]);
      s.insert({connections[i][0],connections[i][1]});
  }

  vector<int> visited(n,0);

  queue<int> q;
  q.push(0);
  int total = 0;

  while(!q.empty()){
      int curr = q.front();
      q.pop();
      if(!visited[curr]){
          visited[curr]=1;

        for(auto it: adj[curr]){
         q.push(it);
          //we are here checking for the edge from 0 to its adjacent,
          if found we will reverse it and count it.
          if(!visited[it] && s.find({curr,it})!=s.end()){
          total++;
          }
        }
     }
  }
  return total;
}
```

$TC = O(V+2E)$

934. Shortest Bridge

Medium 3.3K 146

Companies

You are given an `n x n` binary matrix `grid` where `1` represents land and `0` represents water.

An **island** is a 4-directionally connected group of `1`'s not connected to any other `1`'s. There are **exactly two islands** in `grid`.

You may change `0`'s to `1`'s to connect the two islands to form **one island**.

Return *the smallest number of* `0`*'s you must flip to connect the two islands.*

Example 1:

```
Input: grid = [[0,1],[1,0]]
Output: 1
```

Example 2:

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

Example 3:

```
Input: grid = [[1,1,1,1,1],[1,0,0,0,1],[1,0,1,0,1],[1,0,0,0,1],
[1,1,1,1,1]]
Output: 1
```

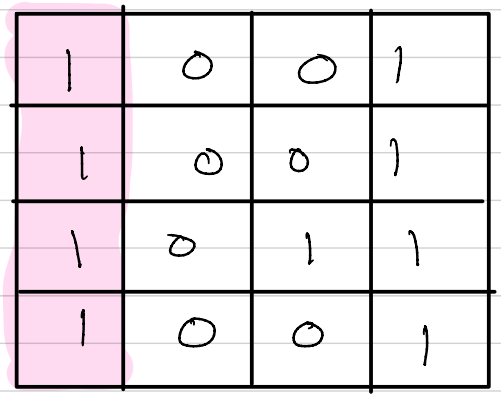
→ There are two islands, both are connecting with water. We need to find the minimum water cells to be reverse to make them a single island.

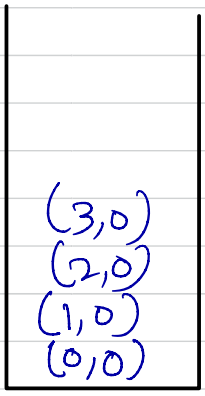
→ The idea is to visits one of the island completely, mark all the cells of this island visited.

→ And from that island, visit the next island and the shortest path to reach next island will be the answer.

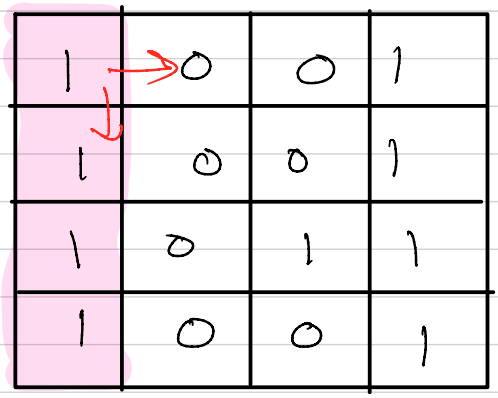
→ To store all the cells of an island, use queue.

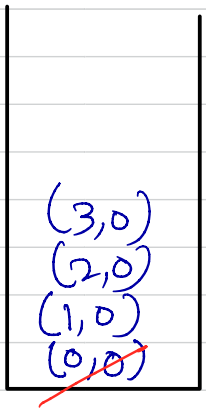
→ And run a BFS from the cells of queue, at first level, we need to change 0 cells, at second level we need to change 1 cell and so on.

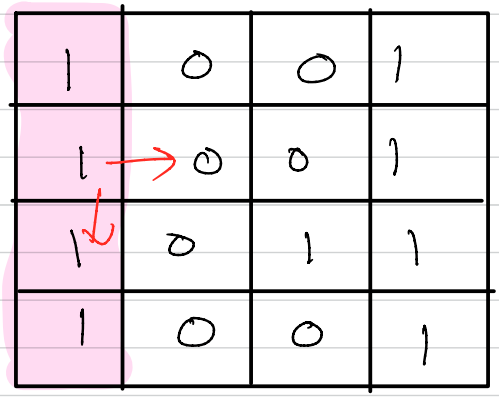




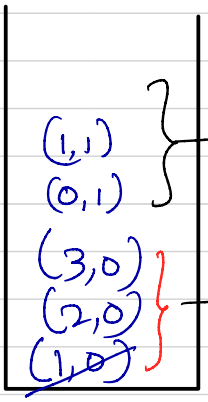
→ Now run BFS and find nearest unvisited one (2nd island).

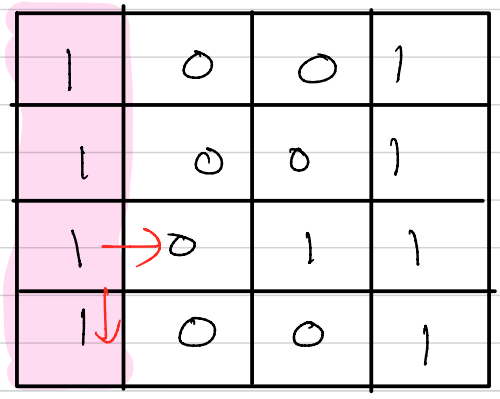




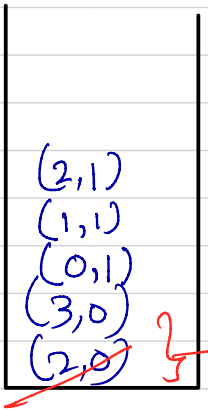


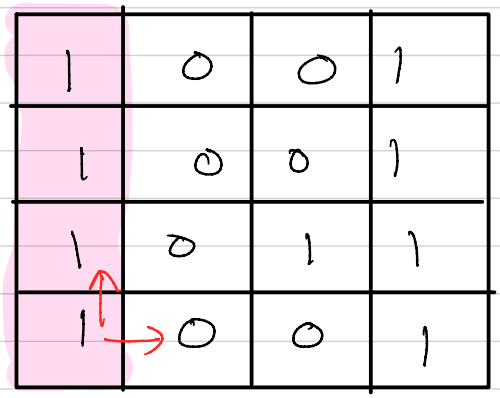
count=0





count=0

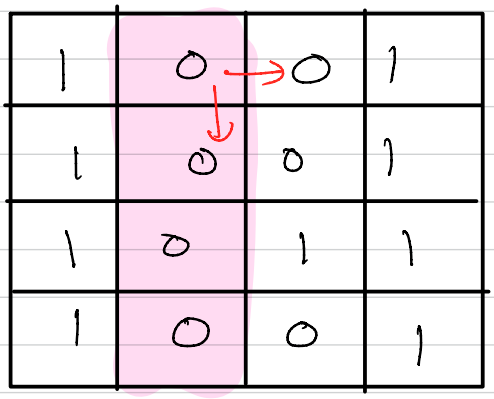




count=0

(3,1)
(2,1)
(1,1)
(0,1)
~~(3,0)~~

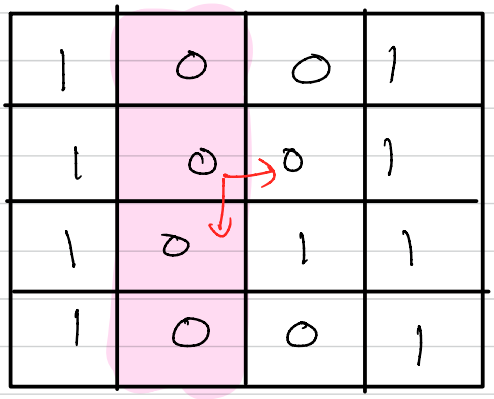
level 2



count=1

(0,2)
(3,1)
(2,1)
(1,1)
~~(0,1)~~

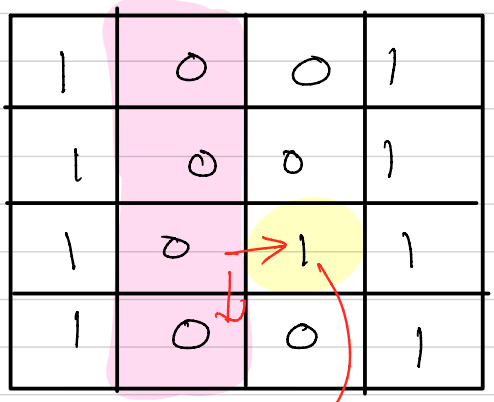
level 2



count=1

(1,2)
(0,2)
(3,1)
(2,1)
~~(1,1)~~

level 2



count=1

(1,2)
(0,2)
(3,1)
(2,1)

level 2

level=2, count=1

```cpp
//Shortest Bridge

int rowdir[4] = {-1,0,1,0};
int coldir[4] = {0,-1,0,1};

void dfs(int row,int col,vector<vector<int>>& grid,vector<vector<int>>&
vis,queue<pair<int,int>>& q){
    int m = grid.size();
    int n = grid[0].size();
    if(row<0 || row>=m || col<0 || col>=n || grid[row][col]==0 || vis[row][col]==1){
                return;
            }

     q.push({row,col});
     vis[row][col]=1;

    for(int i=0;i<4;i++){
            int nr = row+rowdir[i];
            int nc = col+coldir[i];
            dfs(nr,nc,grid,vis,q);
            }
    return ;
}
```

$$TC = O(M*N) + O(M \times N \times 4)$$

```cpp
//Shortest Bridge

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

  vector<vector<int>> vis(m, vector<int> (n,0));
  queue<pair<int,int>> q;
  bool flag = 0;
  for(int i=0;i<m;i++){
      for(int j=0;j<n;j++){
          if(grid[i][j]==1 && !flag){
              dfs(i,j,grid,vis,q);
              flag=1;
          }
      }
  }

  int count=0;
  while(!q.empty()){
      int qsize = q.size();

      for(int i=0;i<qsize;i++){
           int row = q.front().first;
      int col = q.front().second;
      q.pop();


      for(int i=0;i<4;i++){
          int nr = row+rowdir[i];
          int nc = col+coldir[i];
          if(nr<0 || nr>=m || nc<0 || nc>=n || vis[nr][nc]==1){
             continue;
          }
          if(grid[nr][nc]==1)
              return count;
           {
           q.push({nr,nc});
           vis[nr][nc]=1;
          }
      }
      }
     count++;
  }
  return -1;
}
```

1162. As Far from Land as Possible

Hint

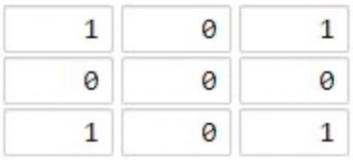
Medium 2.3K 69

Companies

Given an `n x n` `grid` containing only values `0` and `1`, where `0` represents water and `1` represents land, find a water cell such that its distance to the nearest land cell is maximized, and return the distance. If no land or water exists in the grid, return `-1`.

The distance used in this problem is the Manhattan distance: the distance between two cells `(x0, y0)` and `(x1, y1)` is `|x0 - x1| + |y0 - y1|`.

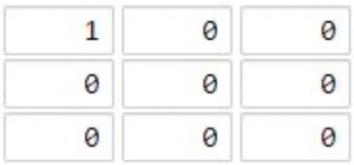
Example 1:



```
1 0 1
0 0 0
1 0 1
```

Input: grid = [[1,0,1],[0,0,0],[1,0,1]]
Output: 2
Explanation: The cell (1, 1) is as far as possible from all the land with distance 2.

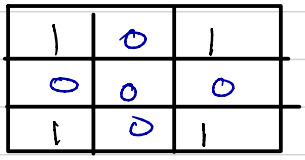
Example 2:



```
1 0 0
0 0 0
0 0 0
```

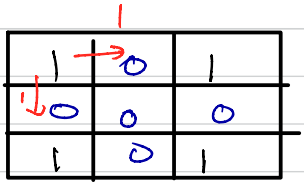
Input: grid = [[1,0,0],[0,0,0],[0,0,0]]
Output: 4
Explanation: The cell (2, 2) is as far as possible from all the land with distance 4.

→ It is similar to 01 matrix.
→ We will use Multisource BFS.
→ We will store row, col, distance in queue of cells having value 1.



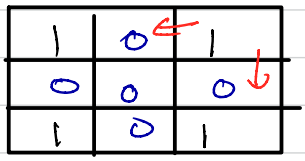
visited

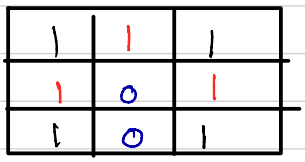
(2,2,0)
(2,0,0)
(0,2,0)
(0,0,0)



visited

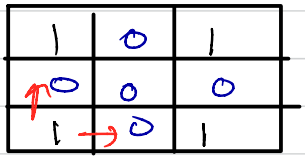
(1,0,1
(0,1,1)
(2,2,0)
(2,0,0)
(0,2,0)
~~(0,0,0)~~

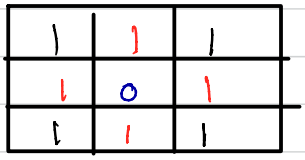




visited

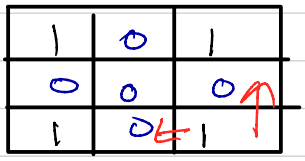
(1,2,1)
(1,0,1
(0,1,1)
(2,2,0)
(2,0,0)
~~(0,2,0)~~

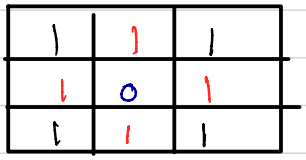




visited

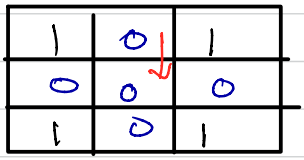
(2,1,1)
(1,2,1)
(1,0,1
(0,1,1)
(2,2,0)
~~(2,0,0)~~

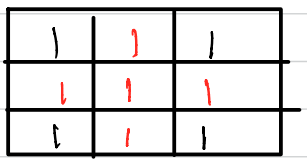




visited

(2,1,1)
(1,2,1)
(1,0,1
(0,1,1)
~~(2,2,0)~~





visited

(1,1,2)
(2,1,1)
(1,2,1)
(1,0,1)
~~(0,1,1)~~

```cpp
//As Far from Land as Possible

int maxDistance(vector<vector<int>>& grid) {
  int m= grid.size();
  int n = grid[0].size();
  int maxdis = 0;
  vector<vector<int>> vis(m, vector<int> (n,0));
  queue<pair<pair<int,int>,int>> q;

  for(int i=0;i<m;i++){
      for(int j=0;j<n;j++){
          if(grid[i][j]==1 && vis[i][j]==0){
              q.push({{i,j},0});
              vis[i][j]=1;
          }
      }
  }
  int rowdir[4]= {-1,0,1,0};
  int coldir[4] = {0,-1,0,1};

  while(!q.empty()){
      int row = q.front().first.first;
      int col = q.front().first.second;
      int count = q.front().second;
      q.pop();


      for(int i=0;i<4;i++){
        int nr = row+rowdir[i];
        int nc = col+coldir[i];

        if(nr>=0 && nr<m && nc>=0 && nc<n && vis[nr][nc]==0){
          vis[nr][nc]=1;
          q.push({{nr,nc},count+1});
        }
        maxdis = max(maxdis,count);
      }
 }
return maxdis==0? -1 : maxdis;
}
```

$TC = O(M \times N) + O(4 \times M \times N)$

$SC = O(M \times N) + O(M \times N)$

1306. Jump Game III

Hint

Medium 3.3K 77

Companies

Given an array of non-negative integers `arr`, you are initially positioned at `start` index of the array. When you are at index `i`, you can jump to `i + arr[i]` or `i - arr[i]`, check if you can reach to **any** index with value 0.

Notice that you can not jump outside of the array at any time.

Example 1:

```
Input: arr = [4,2,3,0,3,1,2], start = 5
Output: true
Explanation:
All possible ways to reach at index 3 with value 0 are:
index 5 -> index 4 -> index 1 -> index 3
index 5 -> index 6 -> index 4 -> index 1 -> index 3
```

Example 2:

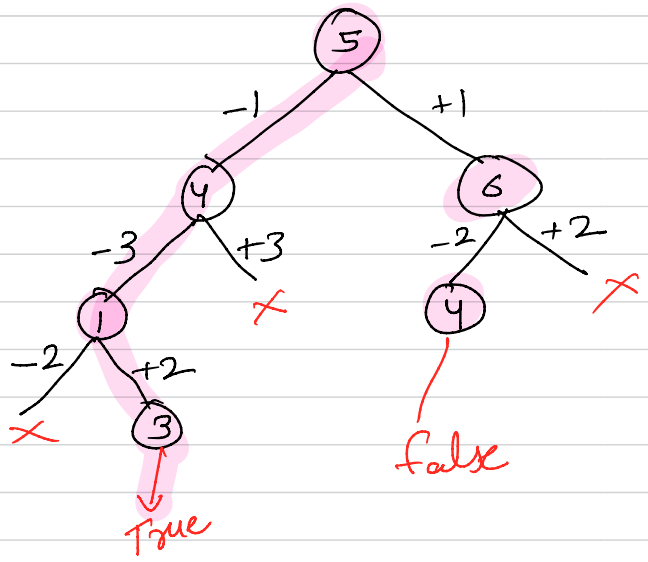
```
Input: arr = [4,2,3,0,3,1,2], start = 0
Output: true
Explanation:
One possible way to reach at index 3 with value 0 is:
index 0 -> index 4 -> index 1 -> index 3
```

→ We can solve this problem using recursion as well as using BFS.

1) Using recursion :-

Ex-1 : {4, 2, 3, 0, 3, 1, 2}

↳ start



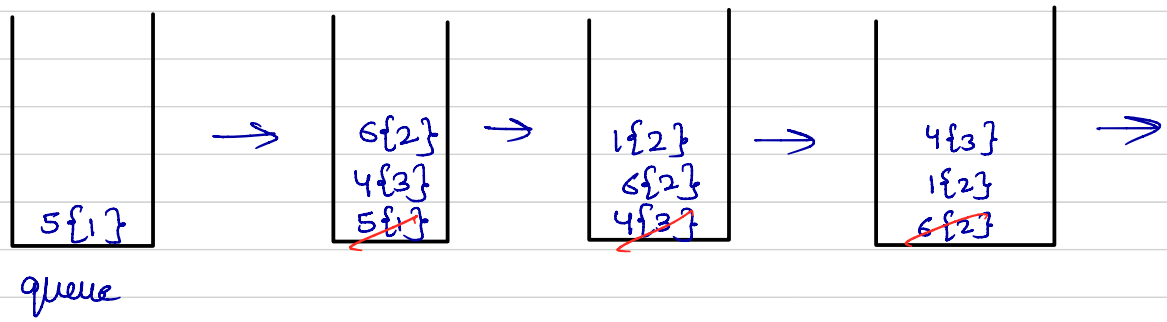
```cpp
//Jump Game III

bool canReach(vector<int>& arr, int start) {
    int m = arr.size();
  return jumpgame(arr,start,m);
}

bool jumpgame(vector<int>& arr, int currindex, int m)
{
  if(currindex>=m || currindex<0 || arr[currindex]== -1)
      return false;

  if(arr[currindex]==0)
      return true;

  int currentmoves = arr[currindex];
  arr[currindex] = -1;

 bool plus = jumpgame(arr,currindex+currentmoves,m);
 bool minus = jumpgame(arr,currindex-currentmoves,m);
 return plus || minus;
}
```

⇒ We are doing arr[currindex] = -1, just to mark non-zeroes values index as visited.

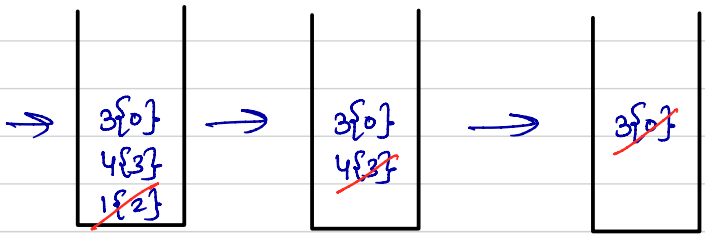
$$TC = O(N)$$
$$SC = O(N)$$

Using BFS :-

→ We don't need to find all paths reaching index with value 0. We only need to find one path, so we will keep storing the values in queue, untill we reach till the index having 0.



queue



```cpp
//Jump Game III

bool canReach(vector<int>& arr, int start) {
  int n = arr.size();
  vector<int> visited(n,0);
  queue<int> q;
  q.push(start);
  visited[start]=1;
  while(!q.empty()){
      int currind = q.front();
      q.pop();

      if(arr[currind]==0) return true;
      int incind = currind+arr[currind];
      int decind = currind-arr[currind];
      if(incind<n && !visited[incind]){
          visited[incind]=1;
          q.push(incind);
      }
      if(decind>=0 && !visited[decind]){
          visited[decind]=1;
          q.push(decind);
      }

  }
  return false;
    }
```

$TC = O(N)$

$SC = O(N) + O(N)$

TRY YOURSELF

133. Clone Graph

Medium 7K 2.9K

Companies

Given a reference of a node in a **connected** undirected graph.

Return a **deep copy** (clone) of the graph.

Each node in the graph contains a value (`int`) and a list (`List[Node]`) of its neighbors.

```
class Node {
    public int val;
    public List<Node> neighbors;
}
```

Test case format:

For simplicity, each node's value is the same as the node's index (1-indexed). For example, the first node with `val == 1`, the second node with `val == 2`, and so on. The graph is represented in the test case using an adjacency list.

An adjacency list is a collection of unordered **lists** used to represent a finite graph. Each list describes the set of neighbors of a node in the graph.

The given node will always be the first node with `val = 1`. You must return the **copy of the given node** as a reference to the cloned graph.

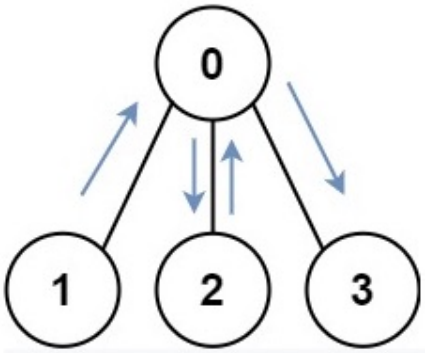
847. Shortest Path Visiting All Nodes

Hard 3043 137 Add to List Share

You have an undirected, connected graph of `n` nodes labeled from `0` to `n - 1`. You are given an array `graph` where `graph[i]` is a list of all the nodes connected with node `i` by an edge.

Return *the length of the shortest path that visits every node*. You may start and stop at any node, you may revisit nodes multiple times, and you may reuse edges.

Example 1:




```
Input: graph = [[1,2,3],[0],[0],[0]]
Output: 4
Explanation: One possible path is [1,0,2,0,3]
```

```cpp
//Shortest Path Visiting All Nodes

int shortestPathLength(vector<vector<int>>& graph) {
  int n = graph.size();
  int all = (1<<n)-1; //11111

  queue<pair<int,pair<int,int>>> q; //node,{dist,mask}
  set<pair<int,int>> vis;  //{node,mask}

  for(int i=0;i<n;i++){
      int mask = (1<<i);
      q.push({i,{0,mask}});
      vis.insert({i,mask});
      }
  while(!q.empty()){
      int node = q.front().first;
      int currdis = q.front().second.first;
      int mask = q.front().second.second;
      q.pop();
      for(auto neighbour : graph[node]){
          int newmask = (mask | (1<<neighbour));

          if(newmask==all){
              return currdis+1;
          }
          else if(vis.count({neighbour,newmask})){
              continue;
          }
          else {
              q.push({neighbour,{currdis+1,newmask}});
              vis.insert({neighbour,newmask});
          }
      }
  }
  return 0;
}
```

THANKYOU

https://www.linkedin.com/in/kapilyadav22/