Graphs
Lesson 3

Find the number of islands

Given a grid consisting of '0's(Water) and '1's(Land). Find the number of islands.

Note: An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically or diagonally i.e., in all 8 directions.

Example 1:

Input:
grid = {{0,1},{1,0},{1,1},{1,0}}

Output:
1

Explanation:
The grid is-
0 1
1 0
1 1
1 0
All lands are connected.

Expected Time Complexity: O(n*m)

Expected Space Complexity: O(n*m)

Constraints:

$1 \leq n, m \leq 500$

```
static int rowNbr[]={1 , -1 , 0 ,  0 , 1 , -1 , -1 ,  1};
static int colNbr[]={0 ,  0 , 1 , -1 , 1 , -1 ,  1 , -1};
void DFS(vector<vector<char>>& grid , int row , int col ,int R , int C)
   {
       grid[row][col]='2';
       for(int i=0;i<8;i++)
       {
           if(row+rowNbr[i]>=0 && row+rowNbr[i]<R &&
               col+colNbr[i]>=0 && col+colNbr[i]<C &&
               grid [row+rowNbr[i]] [col+colNbr[i]] =='1')
           {
               DFS(grid , row+rowNbr[i] , col+colNbr[i], R ,C);
           }
       }
   }
   int numIslands(vector<vector<char>>& grid)
   {
       int R=grid.size(),C=grid[0].size();
       int count=0;
       for(int i=0;i<R;i++)
       {
           for(int j=0;j<C;j++)
           {
               if(grid[i][j]=='1')
               {
                   DFS(grid,i,j,R,C);
                   count++;
               }
           }
       }
       return count;
       }
```

Find whether path exist

Given a grid of size n*n filled with 0, 1, 2, 3. Check whether there is a path possible from the source to destination. You can traverse up, down, right and left.

The description of cells is as follows:

- A value of cell 1 means Source.
- The value of cell 2 means Destination.
- A value of cell 3 means Blank cell.
- A value of cell 0 means Wall.

Note: There is only a single source and a single destination.

Example 1:

Input: grid = {{3,0,3,0,0},{3,0,0,0,3}
,{3,3,3,3,3},{0,2,3,0,0},{3,0,0,1,3}}
Output: 0
Explanation: The grid is-
3 0 3 0 0
3 0 0 0 3
3 3 3 3 3
0 2 3 0 0
3 0 0 1 3
There is no path to reach at (3,1) i,e at destination from (4,3) i,e source.

Example 2:

Input: grid = {{1,3},{3,2}}
Output: 1
Explanation: The grid is-
1 3
3 2
There is a path from (0,0) i,e source to (1,1) i,e destination.

Expected Time Complexity: O(n^2)

Expected Auxiliary Space: O(n^2)

Constraints:

1 ≤ n ≤ 500

```
bool DFS(int x,int y , vector<vector<int>>& grid , int n )
 {
  if(x < 0 || y < 0 || x >= n || y >= n || grid[x][y] == 0)
   return false;
  if(grid[x][y]==2)
   return true;
  grid[x][y]=0;
  bool found = false;
  int xs[4]={0,0,1,-1};
  int ys[4]={1,-1,0,0};

  for(int i=0;i<4;i++)
  {
   found = found || DFS(x+xs[i] , y+ys[i] , grid , n);
  }
  return found;
 }
 //Function to find whether a path exists from the source to destination.
 bool is_Possible(vector<vector<int>>& grid)
 {
  int n=grid.size();

  int x,y;
  for(int i=0;i<n;i++)
  {
   for(int j=0;j<n;j++)
```

```
        {
            if(grid[i][j]==1)
                {
                    x=i,y=j;
                    break;
                }
        }
    }

    return DFS(x,y,grid,n);
    //code here
}
```

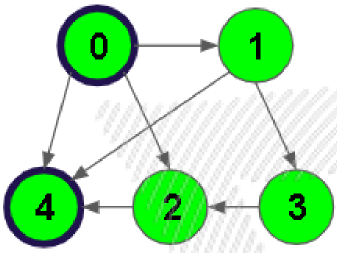
Possible paths between 2 vertices

Given a Directed Graph. Count the total number of ways or paths that exist between two vertices in the directed graph. These paths don't contain any cycle.

Note: Graph doesn't contain multiple edges, self loop and cycles and the two vertices(source and destination) are denoted in the example.

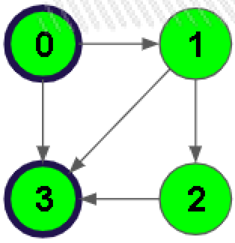
Example 1:

Input:



Output: 4
0 -> 4
0 -> 1 -> 4
0 -> 2 -> 4
0 -> 1 -> 3 -> 2 -> 4
Example 2:

Input:



Output: 3
0 -> 3
0 -> 1 -> 3
0 -> 1 -> 2 -> 3

Expected Time Complexity: O(V!)

Expected Auxiliary Space: O(V)

Constraints:

1 ≤ V, E ≤ 100

1 ≤ source, destination ≤ V

```
int path(int V, vector<int> adj[], int src, int dest)
    {
        if(src==dest)
            return 1;
        int val=0;
        for(int i=0;i<adj[src].size();i++)
        {
            val+=path(V,adj,adj[src][i],dest);
        }
        return val;
    }
    //Function to count paths between two vertices in a directed graph.
            int countPaths(int V, vector<int> adj[], int source, int destination)
            {
                return path(V,adj,source,destination);
                // Code here
            }
```

Unit Area of largest region of 1's

Given a grid of dimension nxm containing 0s and 1s. Find the unit area of the largest region of 1s.

Region of 1's is a group of 1's connected 8-directionally (horizontally, vertically, diagonally).

Example 1:

Input: grid = {{1,1,1,0},{0,0,1,0},{0,0,0,1}}
Output: 5
Explanation: The grid is-
1 1 1 0
0 0 1 0
0 0 0 1
The largest region of 1's is colored
in orange.

Example 2:

Input: grid = {{0,1}}
Output: 1
Explanation: The grid is-
0 1
The largest region of 1's is colored in
orange.

Expected Time Complexity: O(n*m)

Expected Auxiliary Space: O(n*m)

Constraints:

1 ≤ n, m ≤ 500

```
static int rowNbr[]={1 , -1 , 0 ,  0 , 1 , -1 , -1 ,  1};
    static int colNbr[]={0 ,  0 , 1 , -1 , 1 , -1 ,  1 , -1};

void ones(int row,intcol , vector<vector<int>>& grid , vector<vector<int>>&visited , int&cnt,int R , int C)
  {
    if(visited[row][col])
      return;
    visited[row][col]=true;


    for(int i=0;i<8;i++)
    {
      if(row+rowNbr[i]>=0 && row+rowNbr[i]<R &&
        col+colNbr[i]>=0 && col+colNbr[i]<C &&
        grid[row+rowNbr[i]][col+colNbr[i]]==1  &&
        !visited[row+rowNbr[i]][col+colNbr[i]])
      {
        cnt++;
        ones(row+rowNbr[i] , col+colNbr[i],grid ,visited,cnt, R ,C);
      }
    }

  }
  //Function to find unit area of the largest region of 1s.
  int findMaxArea(vector<vector<int>>& grid)
  {
    //vector<vector<bool>>visited;
    int m=grid[0].size();
    int n=grid.size();
    vector<vector<int>> visited(n,vector<int>(m,false));
    //fill(visited.begin(), visited.end(), false);
```

```
        //memset(visited,false,sizeof(visited));

        int cnt,res=INT_MIN;
        for(int i=0;i<n;i++)
        {
            for(int j=0;j<m;j++)
            {
                if(grid[i][j]&&!visited[i][j])
                    {
                        cnt=1;
                        ones(i,j,grid,visited,cnt , n , m );
                        res=max(res,cnt);
                    }
            }
        }
        return res;
        // Code here
    }
```

Word Search

Given a 2D board of letters and a word. Check if the word exists on the board. The word can be constructed from letters of adjacent cells only. ie - horizontal or vertical neighbors. The same letter cell can not be used more than once.

Example 1:

Input: board = {{a,g,b,c},{q,e,e,l},{g,b,k,s}},
word = "geeks"
Output: 1
Explanation: The board is-
a g b c
q e e l
g b k s
The letters which are used to make the
"geeks" are colored.

Example 2:

Input: board = {{a,b,c,e},{s,f,c,s},{a,d,e,e}},
word = "sabfs"
Output: 0
Explanation: The board is-
a b c e
s f c s
a d e e
Same letter can not be used twice hence ans is 0

Expected Time Complexity: O(N * M * 4^L) where N = No. of rows in board, M = No. of columns in board, L = Length of word

Expected Space Complexity: O(L), L is length of word.

Constraints:

1 ≤ N, M ≤ 100

1 ≤ L ≤ N*M

```
bool path(int row,int col,vector<vector<char>>&board,string word,int R,int C,int l,int x)
  {
    if(board[row][col]==' ')
        return false;

    char temp=board[row][col];
    board[row][col]=' ';

    x++;
    if(x==l)
        return true;



    int r[4]={0,0,1,-1};
    int c[4]={1,-1,0,0};

    for(int i=0;i<4;i++)
    {
        if(row+r[i]>=0&&row+r[i]<R&&col+c[i]>=0&&col+c[i]<C&&
        board[row+r[i]][col+c[i]] == word[x])
            {
                if(path(row+r[i],col+c[i],board,word,R,C,l,x))
                return true;
            }
    }
    board[row][col]=temp;
```

```
        return false;
    }
    bool isWordExist(vector<vector<char>>& board, string word)
    {
        int n=board.size();
        int m=board[0].size();
        int x=0;
        for(int i=0;i<n;i++)
        {
            for(int j=0;j<m;j++)
            {
                if(board[i][j]==word[0] && path(i,j,board,word,n,m,word.length(),x))
                    return 1;
            }
        }
        return 0;
        // Code here
    }
```

Find the string in grid

Given a 2D grid of n*m of characters and a word, find all occurrences of a given word in a grid. A word can be matched in all 8 directions at any point. Word is said to be found in a direction if all characters match in this direction (not in zig-zag form). The 8 directions are, horizontally left, horizontally right, vertically up, vertically down and 4 diagonal directions.

Example 1:

Input: grid = {{a,b,c},{d,r,f},{g,h,i}},word = "abc"
Output: {{0,0}}
Explanation: From (0,0) one can find "abc" in the horizontally right direction.

Example 2:

Input: grid = {{a,b,a,b},{a,b,e,b},{e,b,e,b}} ,word = "abe"
Output: {{0,0},{0,2},{1,0}}
Explanation: From (0,0) one can find "abe" in the right-down diagonal. From (0,2) one can find "abe" in the left-down diagonal. From(1,0) one can find "abe" in Horizontally right direction.

Expected Time Complexity: O(n*m*k) where k is constant

Expected Space Complexity: O(1)

Constraints:

1 <= n <= m <= 100

1 <= |word| <= 10

```
bool DFS_up(int u, int v, vector<vector<char>>grid, string word, int k)
{
  if(k==word.length())
    return true;
  if(grid[u][v]==word[k])
    return DFS_up(u-1,v,grid,word,k+1);
  return false;
}
bool DFS_down(int u, int v, vector<vector<char>>grid, string word, int k)
{
  if(k==word.length())
    return true;
  if(grid[u][v]==word[k])
    return DFS_down(u+1,v,grid,word,k+1);
  return false;
}
bool DFS_left(int u, int v, vector<vector<char>>grid, string word, int k)
{
  if(k==word.length())
    return true;
  if(grid[u][v]==word[k])
    return DFS_left(u,v-1,grid,word,k+1);
  return false;
}
bool DFS_right(int u, int v, vector<vector<char>>grid, string word, int k)
{
  if(k==word.length())
    return true;
  if(grid[u][v]==word[k])
    return DFS_right(u,v+1,grid,word,k+1);
  return false;
}
```

```
bool DFS_diarightdown(int u, int v, vector<vector<char>>grid, string word, int k)
{
  if(k==word.length())
     return true;
  if(grid[u][v]==word[k])
     return DFS_diarightdown(u+1,v+1,grid,word,k+1);
  return false;
}
bool DFS_diarightup(int u, int v, vector<vector<char>>grid, string word, int k)
{
  if(k==word.length())
     return true;
  if(grid[u][v]==word[k])
     return DFS_diarightup(u-1,v+1,grid,word,k+1);
  return false;
}
bool DFS_dialeftdown(int u, int v, vector<vector<char>>grid, string word, int k)
{
  if(k==word.length())
     return true;
  if(grid[u][v]==word[k])
     return DFS_dialeftdown(u+1,v-1,grid,word,k+1);
  return false;
}
bool DFS_dialeftup(int u, int v, vector<vector<char>>grid, string word, int k)
{
  if(k==word.length())
     return true;
  if(grid[u][v]==word[k])
     return DFS_dialeftup(u-1,v-1,grid,word,k+1);
  return false;
}
```

```

class Solution {
public:
        vector<vector<int>>searchWord(vector<vector<char>>grid, string word){
            // Code here
            int n=grid.size();
            int m=grid[0].size();
            int k=word.length();
            vector<vector<int>> ans;
            for(int i=0; i<n; i++)
            {
                for(int j=0; j<m; j++)
                {
                    vector<int> p;
                    p.push_back(i);
                    p.push_back(j);
                    if(grid[i][j]==word[0])
                    {
                        if(i-k+1>=0 && DFS_up(i-1,j,grid,word,1))
                           ans.push_back(p);
                        else if(i+k-1<n && DFS_down(i+1,j,grid,word,1))
                           ans.push_back(p);
                        else if(j-k+1>=0 && DFS_left(i,j-1,grid,word,1))
                           ans.push_back(p);
                        else if(j+k-1<m && DFS_right(i,j+1,grid,word,1))
                           ans.push_back(p);
                  else if(i-k+1>=0 && j-k+1>=0 && DFS_dialeftup(i-1,j-1,grid,word,1))
                           ans.push_back(p);
                 else if(i-k+1>=0 && j+k-1<m && DFS_diarightup(i-1,j+1,grid,word,1))
                           ans.push_back(p);
                else if(j-k+1>=0 && i+k-1<n && DFS_dialeftdown(i+1,j-1,grid,word,1))
                           ans.push_back(p);
```

```
      else if(i+k-1<n && j+k-1<m && DFS_diarightdown(i+1,j+1,grid,word,1))
             ans.push_back(p);
          }
        }
      }
      return ans;
    }
};
```