PEER LEARNING FOR PYTHON

My Approach :-

Problem 1 -

Using DFS
Time Complexity - O(n*m)

```
def Dfs(l,n,m,i,j):
    if i>=n or j>=m or i<0 or j<0 or l[i][j]==1:
        return 0

    l[i][j]=1
    down=Dfs(l,n,m,i+1,j)
    right=Dfs(l,n,m,i,j+1)
    left=Dfs(l,n,m,i-1,j)
    up=Dfs(l,n,m,i,j-1)

    return left+right+down+up+1



def Solve(l):
    res=0;
    n=len(l)
    m=len(l[0])
    for i in range(n):
        for j in range(m):
            ans=0
            if l[i][j]==0:
                ans=Dfs(l,n,m,i,j)
                if ans>res:
                    res=ans
    return res

def main():
    l=[[0,1,0,1,1],
       [1,1,0,0,0],
       [1,1,1,1,0],
       [1,1,1,0,0]
       ]

    print(Solve(l))
```

```
if __name__=="__main__":
   main()
```

- First of all we will traverse throughout the grid.
- If we get any 0 inside the grid then we call the recursion function.

- The recursive function implements basic dfs in order to find adjacent cells that contain 0.

- At the start of the recursive function we check if index is out of bound or if the cell contains 1 then we return 0.
- Otherwise we will change the value of the current cell from 0 to 1.
- Then we traverse the grid in four directions up ,down , left, right.
- At the end we will return the sum of values returned by all four directions and +1 to Include the current cell.
- We will take the maximum of the current maximum stored in our result variable and the value
- Returned by recursive function.
- We will print the result at the end.

Problem 2 -

Time Complexity : O(n)
Space Complexity: O(n)

```
class Logger:
    def __init__(self):
        self.dict={}
        self.log=["null"]

    def ShouldPrint(self ,timestamp ,message):

        if message in self.dict.keys():
            if self.dict[message]+10<=timestamp:
                self.dict[message]=timestamp
                self.log.append(True)
            else:
                self.log.append(False)
        else:
            self.dict[message]=timestamp
            self.log.append(True)

def main():
    query=Logger()

    l1=[[1,"foo"],[2,"bar"],[3,"foo"],[8,"bar"],[10,"foo"],[11,"foo"]]

    l1=[[1,"foo"],[2,"bar"],[3,"foo"],[8,"bar"],[10,"foo"],[11,"foo"]]

    for i in l1:
        query.ShouldPrint(i[0],i[1])




    print(query.log)


if __name__=="__main__":
    main()
```

[redacted]

My Approach

- First we create a class named Logger.
- We then define the init function to initialize the class's object.
- In the init function we declare the object variables, a dictionary to store the message Along with its timestamp and a list to store the result of each call to the ShouldPrint Function.
- Then we define the ShouldPrint function which checks if the message is already present in
- the dictionary and if the message comes after 10 seconds then we increment the   timestamp of the message and append "True" in the list.
- Otherwise we append "False"
- Coming out the if statement if the message was not already present in the dictionary then we Just insert the message as a key with its timestamp and append "True" in the list.

Purushottam’s Approach

Problem 1:

```
def dfs(node, grid):
    x, y = node
    grid[x][y] = 1
    size = 0
    n = len(grid)
    m = len(grid[0])

    for dx, dy in [(-1, 0), (1, 0), (0, 1), (0,-1)]:
        new_x, new_y = x + dx, y+dy
        if 0 <= new_x < n and 0 <= new_y < m and grid[new_x][new_y] == 0:
            size += dfs((new_x, new_y), grid)
    return size + 1


def find_max_path(grid):
    ans = 0
    n = len(grid)
    m = len(grid[0])
    for i in range(n):
        for j in range(m):
            if grid[i][j] == 0:
                ans = max(ans, dfs((i, j), grid))
    return ans
```

Review:-

- His approach is similar to mine and using the dfs function for traversing and condition is applied is similar.

Problem 2:

```
class Logger:

    def __init__(self):

        self.msg_dict = {}



    def canPrintMessage(self, timestamp, msg):

        if msg not in self.msg_dict:

            self.msg_dict[msg]=timestamp
            return True
        elif timestamp-self.msg_dict[msg] >=10:
            self.msg_dict[msg]=timestamp
            return True
        else:
            return False

logger = Logger()
```

Review:-

- His approach is similar to mine but he is returning the boolean value through the function in my I am storing these values in the list.

Srinivas's Approach

Problem 1:

```
def dfs(grid, i, j, n, m):
    if i<0 or j<0 or i>=n or j>=m or grid[i][j] == 1:
        return 0
    grid[i][j] =1
    left = dfs(grid, i, j-1, n, m)
    right = dfs(grid, i, j+1, n, m)

    up = dfs(grid, i-1, j, n, m)

    down = dfs(grid, i+1, j, n, m)

    return 1 + left + right + up + down

#function to find size of biggest river
def size_of_biggest_river(grid):

    n,m = len(grid), len(grid[0])

    biggest_size = 0

    for i in range(n):

        for j in range(m):

            if grid[i][j] == 0:

                biggest_size = max(biggest_size, dfs(grid,i,j,n,m))

    return biggest_size

grid = [ [0,1,0,1,1], [1, 1, 0, 0, 0], [1, 1, 1, 1, 0], [1, 1, 1, 0, 0] ]

print(size_of_biggest_river(grid))
```

Review -

- He applied dfs and this approach was similar to mine.

Problem 2:

```
class Logger:

    def __init__(self):

        self.msg_dict = {}

    def shouldPrintMessage(self, timestamp, message):

        if message not in self.msg_dict:

            self.msg_dict[message] = timestamp

            return True

        elif self.msg_dict[message] + 10 <= timestamp:

            self.msg_dict[message] = timestamp

            return True

        else:

            return False

logger = Logger()

print(logger.shouldPrintMessage(4, "foo"))

print(logger.shouldPrintMessage(3, "foo"))
```

Review:-

- His approach is similar to mine but he is returning the boolean value through the function in my I am storing these values in the list.