PEER LEARNING FOR PYTHON

My Approach -

Problem 1-

Using DFS
Time Complexity - O(n*m)

```
def dfs(grid , i,j,n,m):
      if i<0 or j<0 or i>=n or j>=m or grid[i][j]==2 or grid[i][j]==1:
         return 0

      grid[i][j] =2

      a= dfs(grid, i+1, j, n, m)
      b= dfs(grid, i-1, j, n, m)
      c=dfs(grid, i, j+1, n, m)
      d=dfs(grid, i, j-1, n, m)
      return 1+a+b+c+d


def max_path(grid):
   n= len(grid)
   m= len(grid[0])
   ans =0

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

```
    return ans

lst = [[1,0,0,0,1]]

print(max_path(lst))
```

Algo :-

1. Created function name max__path which takes grid as the argument.
2. When I get water which is ‘0’ then recursively call the dfs function for every four directions.
3. And when I call this dfs function in the main function traversing updating the max value to maximum island after all the iteration.

Problem 2-
My Approach
Simple hashing the value into the dictionary and checking whether an element is present in the hash or not .

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

Code:-

```
class Logger:

    def __init__(self):
        self.msg ={}

    def shouldPrintMessage(self, timestamp, message):
```

```
    if message not in self.msg:
      self.msg[message] = timestamp
      return True

    if self.msg[message] + 10 <= timestamp:
      self.msg[message] = timestamp
      return True
    else:
      return False

logger = Logger()

print(logger.shouldPrintMessage(10, "foo"))
```

Algo -

1. Creating the class logger which is given in the question.
2. Having a constructor method in class **self.msg={}.**
3. And Also Initializing the method inside the class which shouldPrintMessage(timestamp,message).
4. This above method takes two parameters: timestamp , message.
5. And inside this method checking whether the message is present or not if not present in the msg dict then updating the values and returning true.
6. If present then it should present after 10 seconds and this will return true. Else it would return false.

Sarthak’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 different from mine .
- Sarthak used bfs checking nearest neighbor first then moving forward and applied the same approach in the main function.

Problem 2

```
class Logger(object):

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self._msg_set = set()
        self._msg_queue = deque()

    def shouldPrintMessage(self, timestamp, message):
        """
        Returns true if the message should be printed in the given timestamp, otherwise returns false.
        """
        while self._msg_queue:
            msg, ts = self._msg_queue[0]
            if timestamp – ts >= 10:
                self._msg_queue.popleft()
                self._msg_set.remove(msg)
            else:
                break

        if message not in self._msg_set:
            self._msg_set.add(message)
            self._msg_queue.append((message, timestamp))
            return True
        else:
            return False
```

Review:

- His approach is different from mine .
- He set and deque for solving the question

- If message is not present he add it to set and also append message and timestamp in deque
- If a message is there the function check whether it is pop out of the queue until that message is not there. While poping out of the queue whether it is more than 10 second or not .If not then it would return false.

Karan’s Approach

Problem 1:

```
def isSafe(grid,row,col,r,c):
    if(r>=row or c>=col or r<0 or c<0 or grid[r][c]== 1 ):
        return False
    return True

def dfsTraverse(grid, row, col, r, c):

    #Checking for corner cases
    if not isSafe(grid, row, col, r, c):
        return 0

    #Making the current cell as 1 to avoid infinite calling during the below
recursive calls for checking for
    # left right down and up cell
    grid[r][c]=1
    #left
    left=dfsTraverse(grid, row, col, r, c – 1)
    #right
    right=dfsTraverse(grid, row, col, r, c + 1)
    #up
    up=dfsTraverse(grid, row, col, r - 1, c)
    #Down
    down=dfsTraverse(grid, row, col, r + 1, c)

    return 1+left+right+up+down


def main():
    grid=[[0,1,0,1,1], [1, 1, 0, 0, 0], [1, 1, 1, 1, 0], [1, 1, 1, 0, 0]]
    row=len(grid)
    col=len(grid[0])
    result=0
    for r in range(row):
        for c in range(col):
            if(grid[r][c] == 0):
                result=max(result,dfsTraverse(grid,row,col,r,c))
    print(result)

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

Review -

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

Problem 2-

```
class logger:
    output_list=[]
    def __init__(self):
        self.message_dict={}
        self.output_list.append('null')
    def shouldPrintMessage(self,timestamp,message):
        # print(timestamp,message)
        if message not in self.message_dict:
            self.message_dict[message]=timestamp
            return True
        elif self.message_dict[message]+10<=timestamp:
            self.message_dict[message]=timestamp
            return True
        else:
            return False

def main():
    log_obj=logger()
    a=[[],[1, "foo"], [2, "bar"], [3, "foo"], [8, "bar"], [10, "foo"], [11,
"foo"]]
    for item in a:
        if item:

log_obj.output_list.append(log_obj.shouldPrintMessage(item[0],item[
1]))

    print(log_obj.output_list)

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

Review:-

- His approach is similar to mine and using dict for hashing and condition is applied is similar.