Question-1

My Solution for question 1:

```
#question 1

# function that will execute in every dfs call which checks for a
river in all 4 directions
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))
```

Approach - Using DFS

Step-1: First, I created a function that accepts parameter grid
Step-2: I iterated through the grid and whenever I found ‘0’ that is water, I tried to check for its adjacent sides by making DFS function call
Step-3: In this process, I updated the *biggest_size* as it is my final result (biggest Island)
Step-4: In DFS function, I checked the base cases like out of bound cases and summed up the result of all four direction and returned the result.
Step-5: While going through this process, ignorer to avoid multiple recursive call, I am setting grid[I][j] = 1.
Step-5: Finally, after all iterations completed, we get our biggest size island and printed it.

Time Complexity: O(n*m)
Space Complexity: O(1) (Auxiliary space used for recursive calls)

Mahesh's Solution for Question-1

```
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()
```

Review on above solution:

- Mahesh's Solution is similar to my approach.

Purushotham's Solution for Question 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 on above Solution:

- The approach is similar to me but that implementation of checking in all four direction in dis function is different.
- In above code, all directions are checked by iterating the directions in a list.

Question-2

My Solution to Question-2:

```
#question 2

class Logger:

    def __init__(self):

        self.msg_dict = {}

    def shouldPrintMessage(self, timestamp, message):

        if message not in self.msg_dict or
(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"))
```

Approach - Basic implementation

- Initialise the empty dictionary in *__init__* function of Logger class
- As the messages the coming to input stream, *shouldPrintMessage* function is called, it accepts two parameters timestamp and message.
- If the message is coming for 1st time, update the message in dictionary by timestamp and return true.
- If not, we have to check whether the previous instance of same message came before 10 sec
- If the current message came within 10 sec difference then return false
- Else update dictionary and return true

Mahesh's Solution for Question 2:

```
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"]]

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


    print(query.log)


if __name__=="__main__":
    main()



print(6 + 5 - 4 * 3 / 2 % 1)

print(3==3.0)

tup=(1,3,1,2,3,4,3,2,1)
# tup.sort()   #give error
print(sorted(tup))
print(tup)
```

Review on above solution:

- The approach is same, but the only difference is above code taking the input streaming messages in the form of list
- solutions are storing in a list for every query.

Purushotham's Solution for Question 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 on above code:

- The approach and implementation is same as mine.