Peer learning document for Python assignment

Question-1

class River():

def longest\_path\_from\_this\_position(self,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 += self.longest\_path\_from\_this\_position((new\_x, new\_y), grid)

return size + 1

def find\_max\_path(self,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, self.longest\_path\_from\_this\_position((i, j), grid))

return ans

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

river=River()

print(river.find\_max\_path(grid))

Approach :-

I have created a class called River that contains two methods:

longest\_path\_from\_this\_position :- A recursive method that takes in a node and a 2D grid and returns the length of the longest connected path of ‘0’ values that starts at the node and only moves in the up, down, left, or right direction. The function updates the grid to mark visited cells as ‘1’.

This method works on BFS (Breadth First Search) algorithm.

find\_max\_path:- A method that takes in a 2D grid and returns the length of the longest connected path of ‘0’ values in the grid. The function accomplishes this by iterating through each cell in the grid and using the longest\_path\_from\_this\_position function to find the longest path that starts at that cell.

The code then creates a sample grid and an instance of the River class, calls the find\_max\_path method on the sample grid using the instance of the River class, and prints out the result. The expected output is the length of the longest connected path of ‘0’ values in the sample grid.

Time Complexity O(m\*n)

Space complexity O(1)

**Srinivas 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, he created a function that accepts parameter grid

Step-2: he iterated through the grid and whenever he found ‘0’ that is water, he tried to check for its adjacent sides by making DFS function call

Step-3: In this process, he 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, he is setting

grid[I][j] = 1.

Step-5: Finally, after all iterations completed, he get our biggest size island and printed it.

**Time Complexity: O(n\*m)**

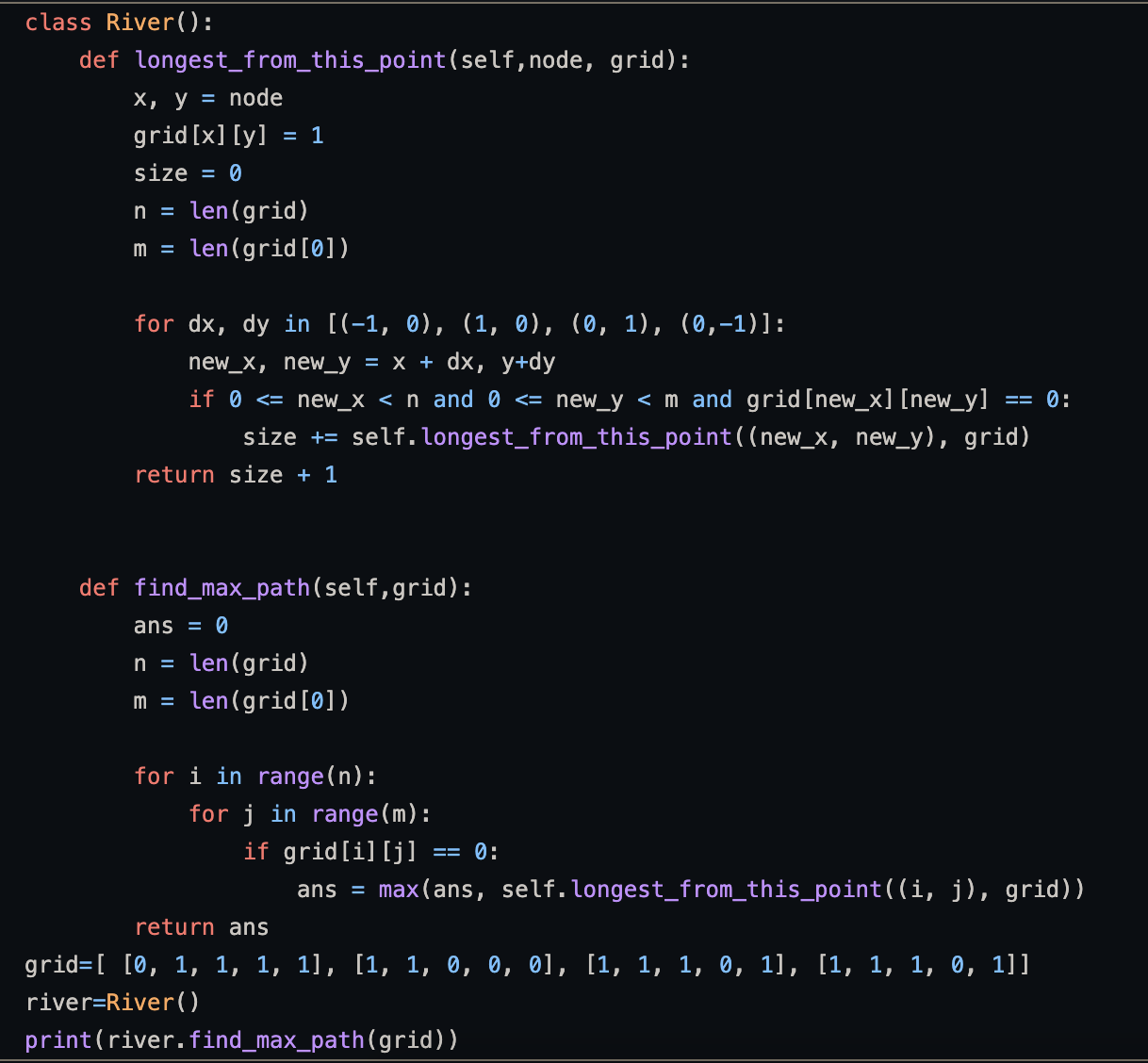
**Space Complexity: O(1)** (Auxiliary space used for recursive calls)

**Review on above solution:**

- The approach is similar to me but that implementation of checking in all four direction in dfs function is different.

- In above code, all directions are checked by iterating the directions in a list.

**Purushottam's Solution for Question-1**



**Approach** - Using DFS

1. He has created a class and two methods inside the class.
2. First method is find\_max\_path . this mathod is taking the binary grid as argument . and iterating in

given binary grid .

1. If the value of the grid will be ‘0’ then he is calling the longest\_from\_this\_point .
2. The longest\_from\_this\_grid is taking a grid as input and iterating in all four directions

and retruning the longest river as summation of result all four directions .

**Review on above Solution:**

**:-** His approach is similar as mine.

Question-2

**My Solution for Question 2:**

class Logger:

def \_\_init\_\_(self):

self.msg\_dict = {}

def shouldPrintMessage(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()

print(logger.shouldPrintMessage(0,'Sigmoid'))

print(logger.shouldPrintMessage(9,'Sigmoid'))

print(logger.shouldPrintMessage(12,'Sigmoid'))

**Approach** -

1. I have created a Logger class and created a constructor where I have created an empty dictionary.
2. I have also created a method shouldPrintMessage that will take string as an input .
3. If the string will coming at first time or comming after 10 sec then it will retrun TRUE

otherwise it will return FALSE.

1. I have created a an object named logger and calling throught this object.

**Srinivas’s 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

**Review on above Solution:**

**:-** His approach is similar as mine.

**Purushottam’s Solution for Question 2:**

class Logger:

def \_\_init\_\_(self):

self.msg\_dict = {}

def souldPrintMessage(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()

print(logger.shouldPrintMessage(0,’Sigmoid’)

print(logger.shouldPrintMessage(3,’Sigmoid’)

print(logger.shouldPrintMessage(11,’Sigmoid’)

**Approach -**

1.He have created a Logger class and created a constructor where He have created an empty dictionary.

2. He have also created a method shouldPrintMessage that will take string as an input .

3.If the string will coming at first time or comming after 10 sec then it will retrun TRUE

otherwise it will return FALSE.

4.He have created a an object named logger and calling throught this object.

**Review on above Solution:**

**:-** His approach is similar as mine.