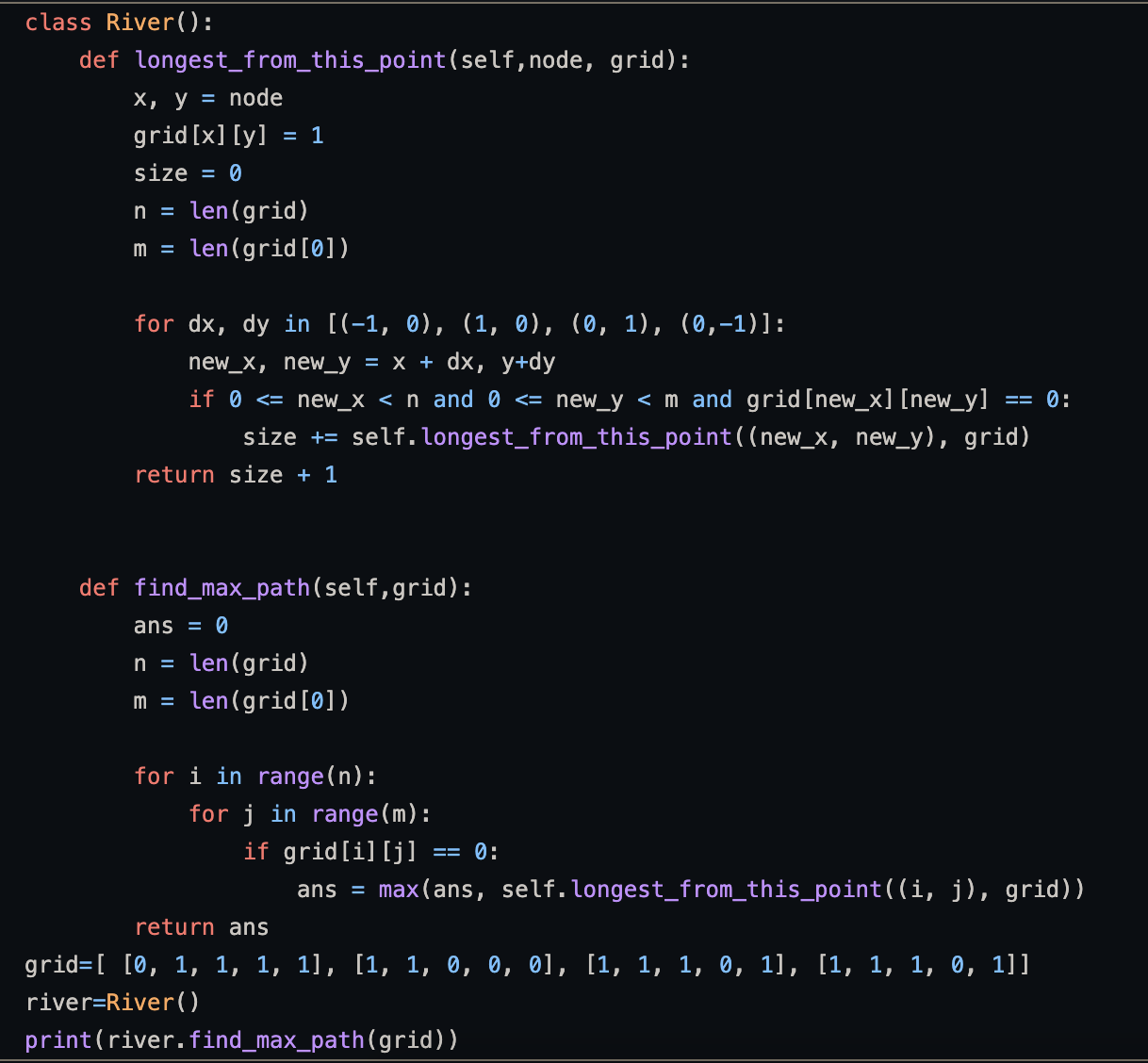
Peer learning document python

Question-1



Approach

This code defines a class called River that contains two methods:

longest\_from\_this\_point: A recursive function 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 algorithm

find\_max\_path: A function 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\_from\_this\_point 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)

**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.

**srinivas’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 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’)

**Review on above code:**

**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

**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.