Data Analytics Assignment

Team 10:

عمر عصام الدين الأنصاري

منى محسن حسن البطران

أحمد أسامة محمد أسعد

محمد أحمد محمد عبد المنعم

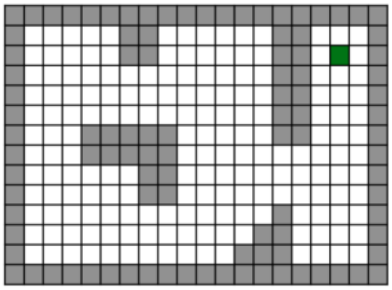
Submitted to:

Dr Ibrahim Sadeq

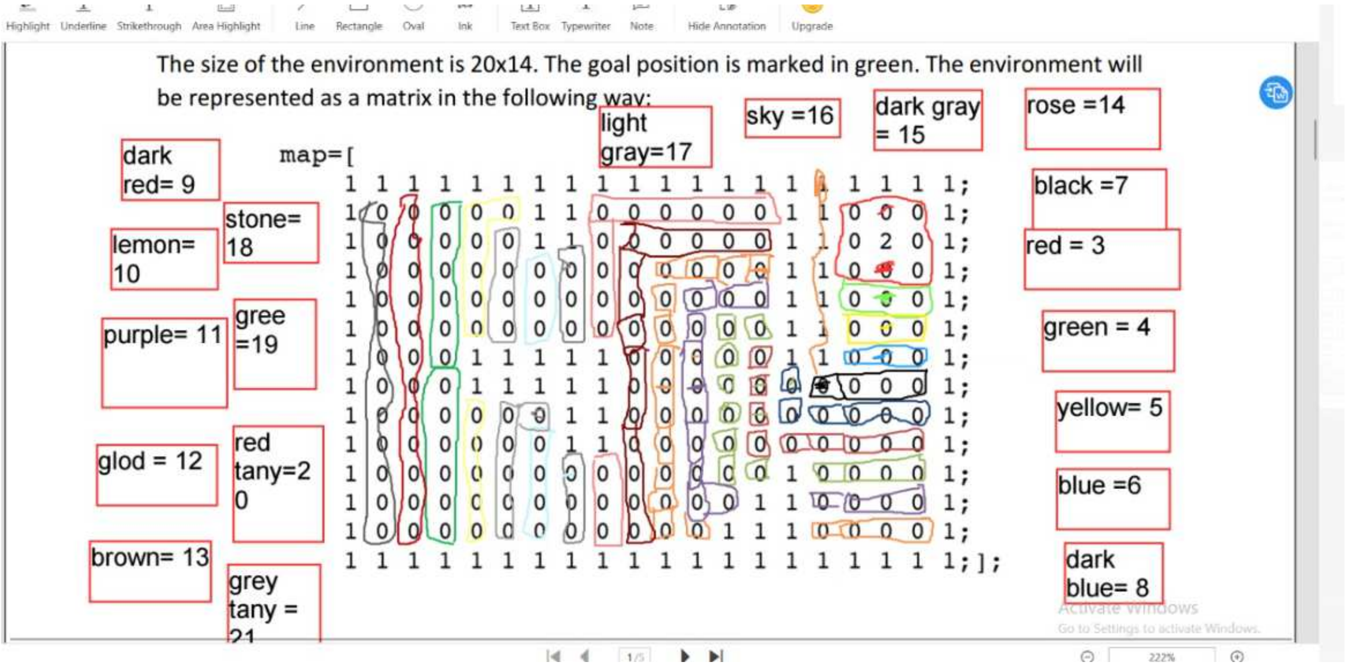
WaveFront Algorithm:

Algorithm to compute the optimal path toward the goal using 8 point connectivity.Priority is assumed :[upper, Right, Lower, Left, Upper right, lower right, Lower Left, Upper Left].

IMPORTANT NOTE: Note that the obstacle cells are marked with 1, free space with 0, and the goal position with a 2.



Brainstorming Phase-Filling Matrix Manually:



The size of the environment is 20x14. The goal position is marked in green. The environment will be represented as a matrix in the following way:

```
map=[
    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1;
    1 0 0 0 0 0 1 1 0 0 0 0 0 0 1 1 0 0 0 1;
    1 0 0 0 0 0 1 1 0 0 0 0 0 0 1 1 0 2 0 1;
    1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1;
    1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1;
    1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1;
    1 0 0 0 1 1 1 1 1 0 0 0 0 0 1 1 0 0 0 1;
    1 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 1;
    1 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 1;
    1 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 1;
    1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1;
    1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1;
    1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 1;
    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1;];
```

dark red= 9

lemon= 10

purple= 11

glod = 12

brown= 13

stone= 18

gree =19

red tany=20

grey tany = 21

light gray=17

sky =16

dark gray = 15

rose =14

black =7

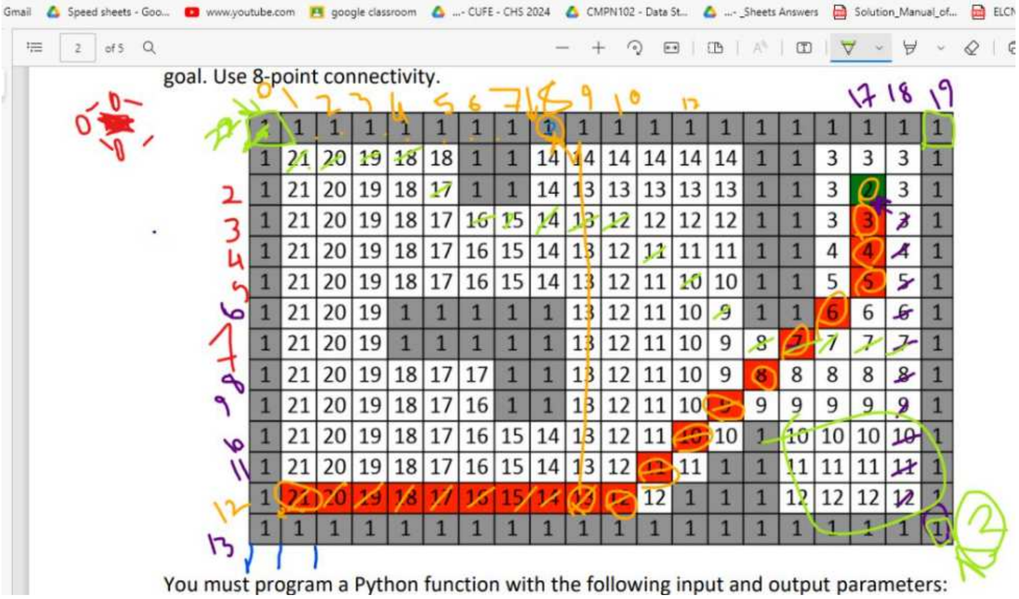
red = 3

green = 4

yellow= 5

blue =6

dark blue= 8



goal. Use 8-point connectivity.

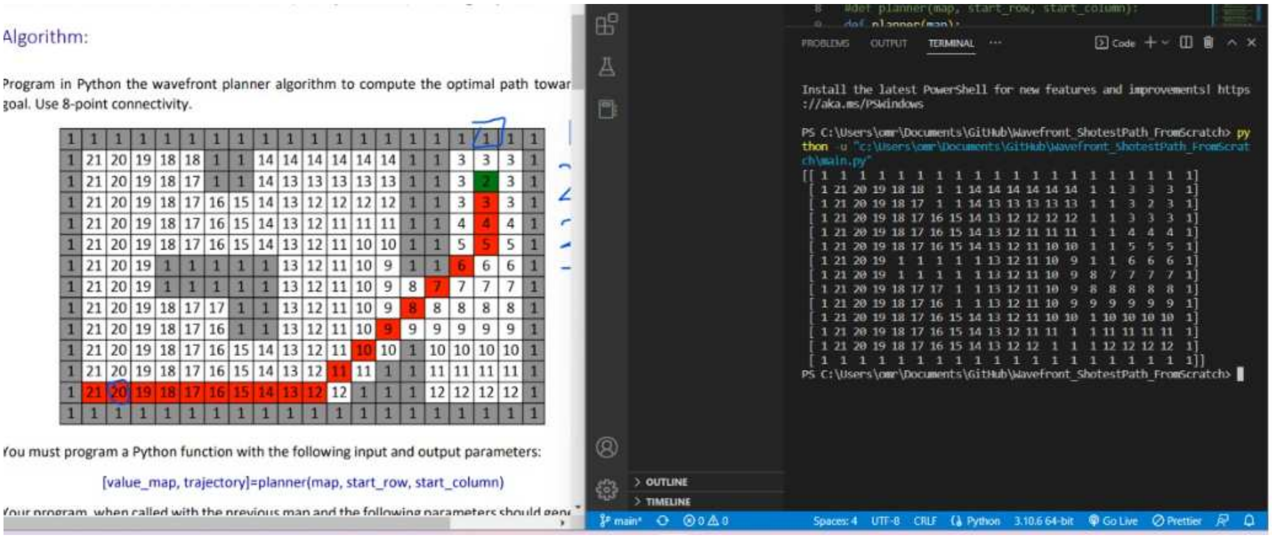
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	21	20	19	18	18	1	1	14	14	14	14	14	14	1	1	3	3	3	1
1	21	20	19	18	17	1	1	14	13	13	13	13	13	1	1	3	2	3	1
1	21	20	19	18	17	16	15	14	13	12	12	12	12	1	1	3	3	3	1
1	21	20	19	18	17	16	15	14	13	12	11	11	11	1	1	4	4	4	1
1	21	20	19	18	17	16	15	14	13	12	11	10	10	1	1	5	5	5	1
1	21	20	19	1	1	1	1	1	13	12	11	10	9	1	1	6	6	6	1
1	21	20	19	1	1	1	1	1	13	12	11	10	9	8	7	7	7	7	1
1	21	20	19	18	17	17	1	1	13	12	11	10	9	8	8	8	8	8	1
1	21	20	19	18	17	16	1	1	13	12	11	10	9	9	9	9	9	9	1
1	21	20	19	18	17	16	15	14	13	12	11	10	10	1	10	10	10	10	1
1	21	20	19	18	17	16	15	14	13	12	11	11	1	1	11	11	11	11	1
1	21	20	19	18	17	16	15	14	13	12	12	1	1	1	12	12	12	12	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

You must program a Python function with the following input and output parameters:

Algorithm Steps:

1)Matrix Filling

Output:



Algorithm:

Program in Python the wavefront planner algorithm to compute the optimal path towar
goal. Use 8-point connectivity.

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	21	20	19	18	18	1	1	14	14	14	14	14	14	1	1	3	3	3	1
1	21	20	19	18	17	1	1	14	13	13	13	13	13	1	1	3	2	3	1
1	21	20	19	18	17	16	15	14	13	12	12	12	12	1	1	3	3	3	1
1	21	20	19	18	17	16	15	14	13	12	11	11	11	1	1	4	4	4	1
1	21	20	19	18	17	16	15	14	13	12	11	10	10	1	1	5	5	5	1
1	21	20	19	1	1	1	1	1	13	12	11	10	9	1	1	6	6	6	1
1	21	20	19	1	1	1	1	1	13	12	11	10	9	8	7	7	7	7	1
1	21	20	19	18	17	17	1	1	13	12	11	10	9	8	8	8	8	8	1
1	21	20	19	18	17	16	1	1	13	12	11	10	9	9	9	9	9	9	1
1	21	20	19	18	17	16	15	14	13	12	11	10	10	1	10	10	10	10	1
1	21	20	19	18	17	16	15	14	13	12	11	11	1	1	11	11	11	11	1
1	21	20	19	18	17	16	15	14	13	12	12	1	1	1	12	12	12	12	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

You must program a Python function with the following input and output parameters:

[value_map, trajectory]=planner(map, start_row, start_column)

Your program, when called with the previous map and the following parameters should gene

```
#def planner(map, start_row, start_column):
def planner(map):
```

```
Install the latest PowerShell for new features and improvements! https
://aka.ms/PSWindows

PS C:\Users\omr\Documents\GitHub\Wavefront_ShotestPath_FromScratch> py
thon -u "c:\Users\omr\Documents\GitHub\Wavefront_ShotestPath_FromScrat
ch\main.py"
[[ 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1]
 [ 1 21 20 19 18 18  1  1 14 14 14 14 14 14  1  1  3  3  3  1]
 [ 1 21 20 19 18 17  1  1 14 13 13 13 13 13  1  1  3  2  3  1]
 [ 1 21 20 19 18 17 16 15 14 13 12 12 12 12  1  1  3  3  3  1]
 [ 1 21 20 19 18 17 16 15 14 13 12 11 11 11  1  1  4  4  4  1]
 [ 1 21 20 19 18 17 16 15 14 13 12 11 10 10  1  1  5  5  5  1]
 [ 1 21 20 19  1  1  1  1  1 13 12 11 10  9  1  1  6  6  6  1]
 [ 1 21 20 19  1  1  1  1  1 13 12 11 10  9  8  7  7  7  7  1]
 [ 1 21 20 19 18 17 17  1  1 13 12 11 10  9  8  8  8  8  8  1]
 [ 1 21 20 19 18 17 16  1  1 13 12 11 10  9  9  9  9  9  9  1]
 [ 1 21 20 19 18 17 16 15 14 13 12 11 10 10  1 10 10 10 10  1]
 [ 1 21 20 19 18 17 16 15 14 13 12 11 11  1  1 11 11 11 11  1]
 [ 1 21 20 19 18 17 16 15 14 13 12 12  1  1  1 12 12 12 12  1]
 [ 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1]]
PS C:\Users\omr\Documents\GitHub\Wavefront_ShotestPath_FromScratch>
```

2)Calculating Shortest Path

3)Displaying Shortest Path Visually

General Introduction:

When enter function we create an empty list, then we do 2 nested for loops

to iterate over the 2D map array to get the goal which is the element equals to 2,

after we found our goal , we append the index of the goal as a tuple to the list

After that call fillMatrix(map,arr), inside function fillmatrix: We create an empty new list, then checks if length of list of tuples passed to the function is zero, that is the list is empty then return map, after that we iterate over the length of that list of tuples and inside the for loop for each element in the list we take the index i,j of from that tuple , also take the height and width of the map do if condition for each of the 8 neighbours to checks first if that neighbour is not out of map boundary and then checks if that element is a valid one to fill , that is if it's equal to zero, then append it to the new list ,return a recursive call which is "return self.fillMatrix(map,newArr)" so that we can apply these lines of code for every element.After that, we call the shortest path function and return list of tuples.

Explanantion of planner(self,map,i_start,j_start):

When enter function we create an empty list, then we do 2 nested for loops

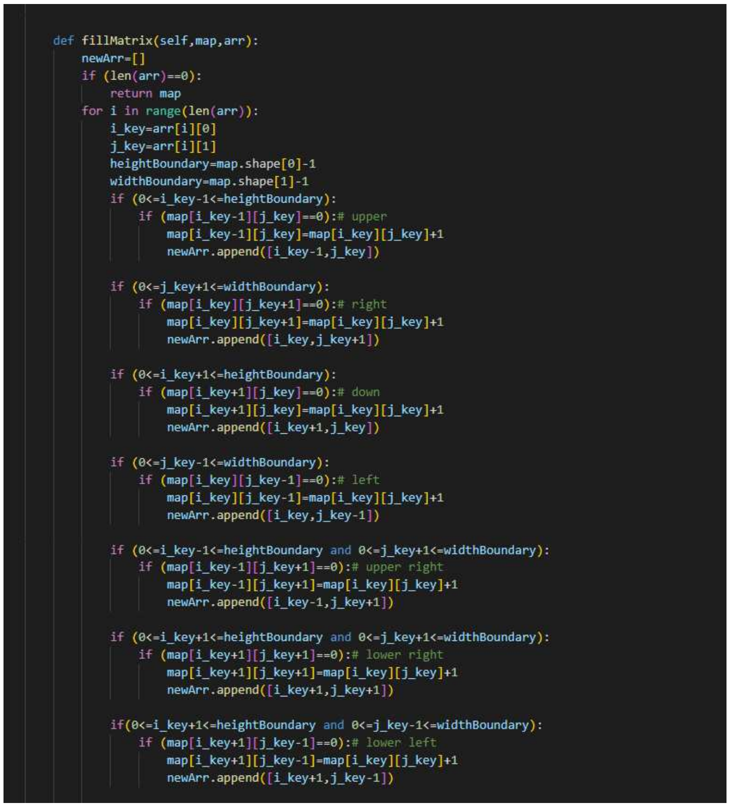
to iterate over the 2D map array to get the goal which is the element equals to 2,

after we found our goal , we append the index of the goal as a tuple to the list

After that call fillMatrix(map,arr), inside function fillmatrix:

We create an empty new list, then checks if length of list of tuples passed to the function is zero, that is the list is empty then return map, after that we iterate over the length of that list of tuples and inside the for loop for each element in the list we take the index i,j of from that tuple , also take the height and width of the map do if condition for each of the 8 neighbours to checks first if that neighbour is not out of map boundary and then checks if that element is a valid one to fill , that is if it's equal to zero, then append it to the new list. , Repeat for each neighbour ,after implementing these lines of code ,return a recursive call which is "return self.fillMatrix(map,newArr)" so that we can apply these lines of code for every element.

Challenges in the FilMatrix Function:



```python
def fillMatrix(self,map,arr):
    newArr=[]
    if (len(arr)==0):
        return map
    for i in range(len(arr)):
        i_key=arr[i][0]
        j_key=arr[i][1]
        heightBoundary=map.shape[0]-1
        widthBoundary=map.shape[1]-1
        if (0<=i_key-1<=heightBoundary):
            if (map[i_key-1][j_key]==0):# upper
                map[i_key-1][j_key]=map[i_key][j_key]+1
                newArr.append([i_key-1,j_key])

        if (0<=j_key+1<=widthBoundary):
            if (map[i_key][j_key+1]==0):# right
                map[i_key][j_key+1]=map[i_key][j_key]+1
                newArr.append([i_key,j_key+1])

        if (0<=i_key+1<=heightBoundary):
            if (map[i_key+1][j_key]==0):# down
                map[i_key+1][j_key]=map[i_key][j_key]+1
                newArr.append([i_key+1,j_key])

        if (0<=j_key-1<=widthBoundary):
            if (map[i_key][j_key-1]==0):# left
                map[i_key][j_key-1]=map[i_key][j_key]+1
                newArr.append([i_key,j_key-1])

        if (0<=i_key-1<=heightBoundary and 0<=j_key+1<=widthBoundary):
            if (map[i_key-1][j_key+1]==0):# upper right
                map[i_key-1][j_key+1]=map[i_key][j_key]+1
                newArr.append([i_key-1,j_key+1])

        if (0<=i_key+1<=heightBoundary and 0<=j_key+1<=widthBoundary):
            if (map[i_key+1][j_key+1]==0):# lower right
                map[i_key+1][j_key+1]=map[i_key][j_key]+1
                newArr.append([i_key+1,j_key+1])

        if(0<=i_key+1<=heightBoundary and 0<=j_key-1<=widthBoundary):
            if (map[i_key+1][j_key-1]==0):# lower left
                map[i_key+1][j_key-1]=map[i_key][j_key]+1
                newArr.append([i_key+1,j_key-1])
```

1. where is the termination point?

2. would i use iterative or recurssive?

3. if iterative do i know how many times the program will enter?

4. if reccursive what is the termination?

5. recursive function implementation?

Answers:

1. the termination happen when no other points to add

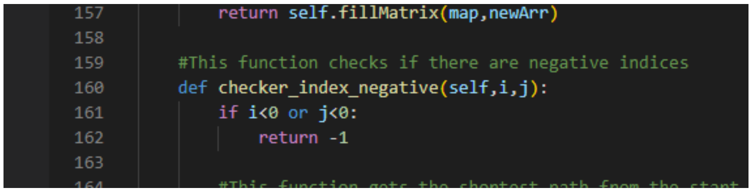
2 & 3. i couldnot use itterative since i donot know when will i end, at which itterative will the matrix filling reach to an end.

4. i could use the recurssive and the termination is when i couldont add another point.

5. the idea came from the basic thinking of filling the 8 neighbors form top,right,down,left,top-right,bottom-right,bottom-left,top-left of a point in each step the code checks if the neighbor is out of range or not if not out of range just add one to the key and place the indecies value in an array after checking and adding for all neighbors recurssvily call the function and pass the array to it so this will be the keys, if the array is empty this means no points to add so will return the map.

Explanation of shortest path function:

First we create list of neighbours without execluding the ones(obstacles) and list of neighbours with ones(obstacles) and another list of tuples that will include the trajectory indices. Also,create a goal found variable that acts as a flag when a goal is found. When we enter the function we first assign i<-starting row and j<-starting col ,also we create a tuple and add initially the first index to the list of tuples, then after that we start looping using while with a condition that goalfound==0 as we did not reach the goal yet, inside while loop, we make an if condition that checks whether the goal is reached or not ,that is when the current element is equals 2,and if reached set goalfound=1 (true) and breaks. For each 8 neighbour of the current element, call function checker and use it to check whether the index of that neighbor is negative or not, that is when exceeded the map arrray boundaries, if checker not equal -1 then no boundaries and assign that neighbor to a variable and append it to the list of neighbours:



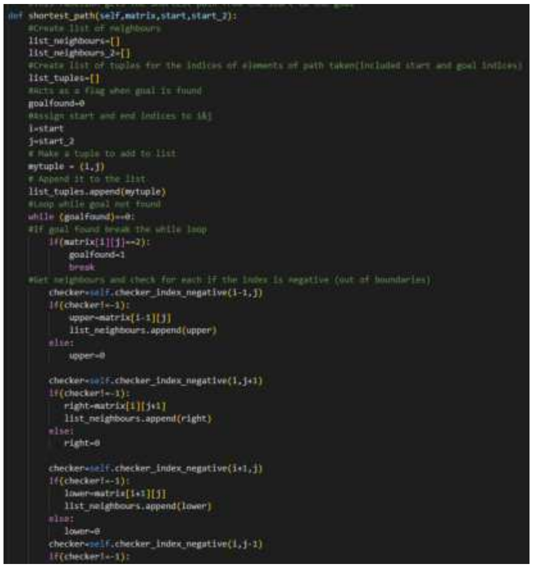
```
        return self.fillMatrix(map,newArr)

    #This function checks if there are negative indices
    def checker_index_negative(self,i,j):
        if i<0 or j<0:
            return -1

        #This function gets the shortest path from the start
```

, else ignore that neighbour, repeat for all the neighbours. After that step, loop over list of neighbours and execludes the ones (obstacles) and append to the new list of neighbours the elements not equal ones that is not obstacles .After that call min() function on list of neighbours.We then make an if statement for each one of the 8 neighbours and checks if it is the min and append , for priority , we did consider the following :[upper, Right, Lower, Left, Upper right, lower right, Lower Left, Upper Left] as our priority and to implement it code wise , we would ask first if it is the upper , else if it is not upper then check on the right , also if not check on the left and so on with the sequence mentioned before and if neighbour equals the min then append its index to the list of tuples, finally returns the list of tuples.

Part of the function:



```
def shortest_path(self,matrix,start,start_2):
    #Create list of neighbours
    list_neighbours=[]
    list_neighbours_2=[]
    #Create list of tuples for the indices of elements of path taken(included start and goal indices)
    list_tuples=[]
    #Acts as a flag when goal is found
    goalfound=0
    #Assign start and end indices to i&j
    i=start
    j=start_2
    # Make a tuple to add to list
    mytuple = (i,j)
    # Append it to the list
    list_tuples.append(mytuple)
    #Loop while goal not found
    while (goalfound)==0:
    #If goal found break the while loop
        if(matrix[i][j]==2):
            goalfound=1
            break
    #Get neighbours and check for each if the index is negative (out of boundaries)
        checker=self.checker_index_negative(i-1,j)
        if(checker!=-1):
            upper=matrix[i-1][j]
            list_neighbours.append(upper)
        else:
            upper=0

        checker=self.checker_index_negative(i,j+1)
        if(checker!=-1):
           right=matrix[i][j+1]
           list_neighbours.append(right)
        else:
           right=0

        checker=self.checker_index_negative(i+1,j)
        if(checker!=-1):
           lower=matrix[i+1][j]
           list_neighbours.append(lower)
        else:
           lower=0
        checker=self.checker_index_negative(i,j-1)
        if(checker!=-1):
```

Problems faced during implementing shortest path function:

Problems faced during implementing shortest path function

->shortest_path(self,matrix,start,end)

The First problem we faced is that when we get the neighbours of the current element,we sometimes have ones(obstacles),and therefore to solve this, we first get the neighbours if within boundaries then we loop over the list and if element value not equal one , then append to a new list , and this new list we will call the min() function to it.

Second problem we faced is getting minimium of the 8 neighbours of the current element where we would save all the neighbours in a list and call a function min(list_neighbours) to get min but if we have neighbours we would have one as our min and that is undesirable so we execluded the ones(obstacles) from the list and we would loop and append non-zero elements to this new list.

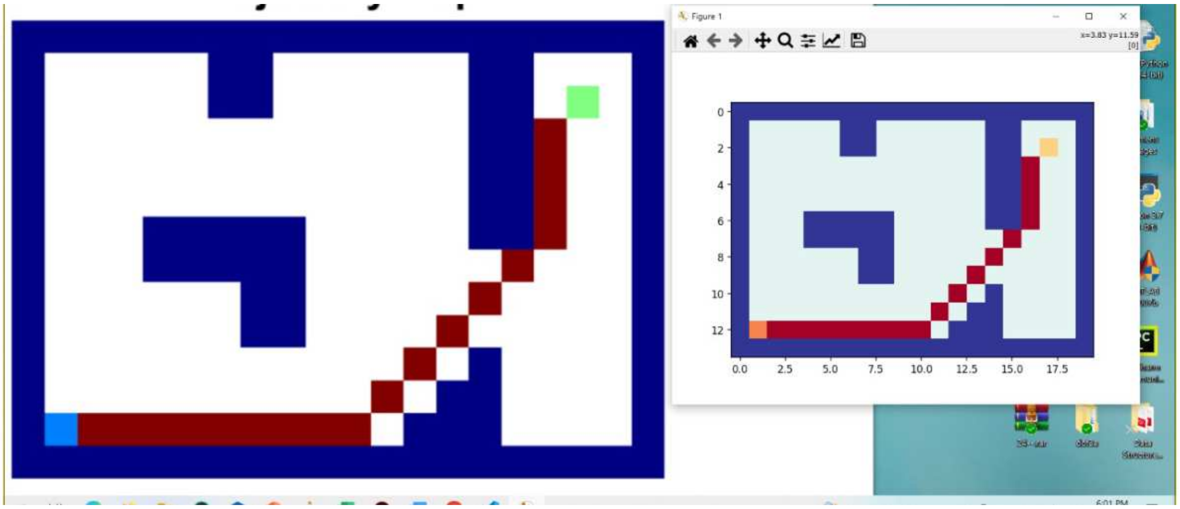
Third problem we came across is that when we start @(0,0) and the goal is @(height-1,width-1) where it would get the trajectory path in a list of tuples as follows -->[(0,0),(-1,-1)] , as shown here it has reached the last element since we are working with lists and list of list[-1] gives the last index of the array so it jumps directly from start index(0,0) to the last index the goal @(-1,-1) therefore we solved this using checker_index_negative(self,i,j) function where it checks if there are negative indices and if true , ignore this neighbor and do the same for other neighbours.

Also, another problem that we got is that in order to display the indices of the generated trajectory , we did search and decided to use a list of Tuples but we had to search again on how to use it since we did not use it before. And the result for example is as follows: [(12, 1), (12, 2), (12, 3), (12, 4), (12, 5), (12, 6), (12, 7), (12, 8), (12, 9), (12, 10), (11, 11), (10, 12), (9, 13), (8, 14), (7, 15), (6, 16), (5, 16), (4, 16), (3, 16), (2, 17)]. Another concern we had is that we should set a terminating condition for the while loop we implemented to iterate on the elements and stops and breaks when we reach our goal.The next step is plotting.

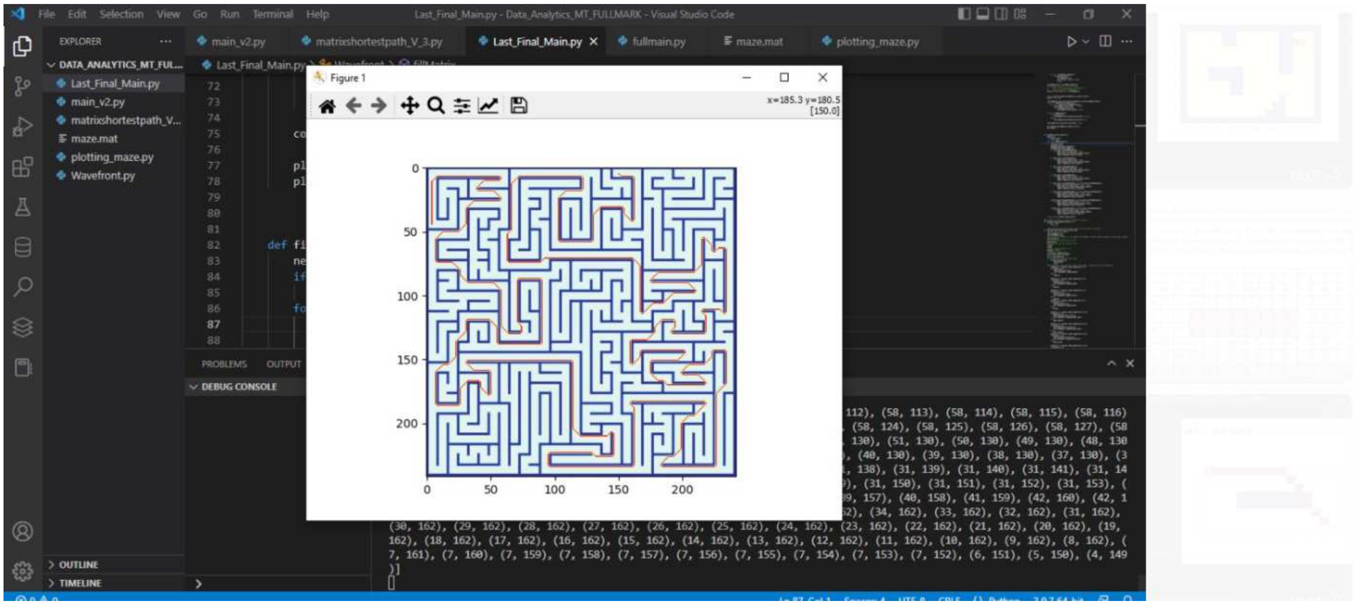
After that we do plotting for the map as follows:

Using matplotlib.pyplot library , we plotted our map:

First we create a 2D array of zeros to carry the colored values of the map to be plotted,then we loop over the filledmatrix, and assign 255 to the coloredmatrix if there are ones(obstacles) and 150 for the rest of map, values except for the starting cell index , goal index and the trajectory path, 3 different values are assigned.

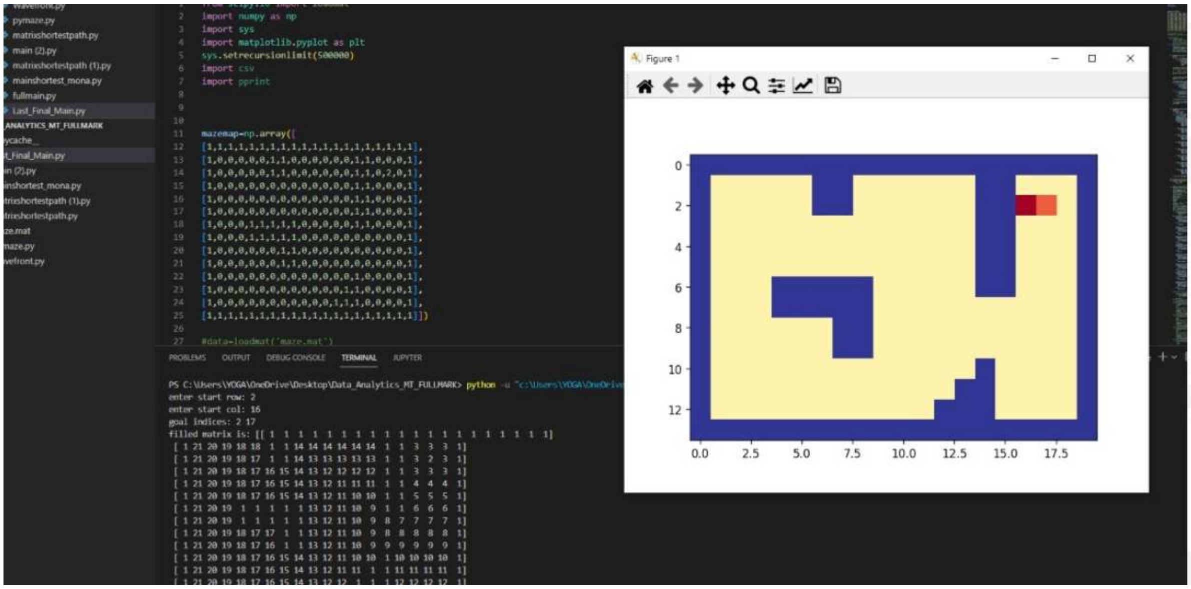


Output Samples and Test Cases:



```
def fi
```

```
112), (58, 113), (58, 114), (58, 115), (58, 116)
, (58, 124), (58, 125), (58, 126), (58, 127), (58
 130), (51, 130), (50, 130), (49, 130), (48, 130
), (40, 130), (39, 130), (38, 130), (37, 130), (3
, 138), (31, 139), (31, 140), (31, 141), (31, 14
), (31, 150), (31, 151), (31, 152), (31, 153), (
39, 157), (40, 158), (41, 159), (42, 160), (42, 1
2), (34, 162), (33, 162), (32, 162), (31, 162),
(30, 162), (29, 162), (28, 162), (27, 162), (26, 162), (25, 162), (24, 162), (23, 162), (22, 162), (21, 162), (20, 162), (19,
162), (18, 162), (17, 162), (16, 162), (15, 162), (14, 162), (13, 162), (12, 162), (11, 162), (10, 162), (9, 162), (8, 162), (
7, 161), (7, 160), (7, 159), (7, 158), (7, 157), (7, 156), (7, 155), (7, 154), (7, 153), (7, 152), (6, 151), (5, 150), (4, 149
)]
```



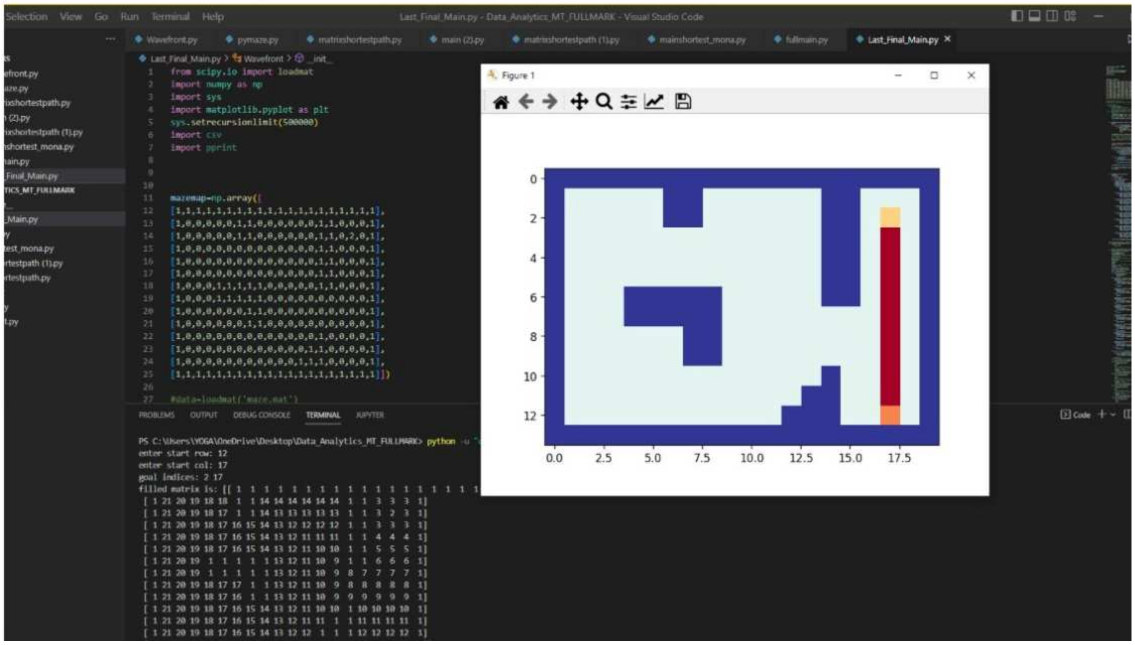
```python
import numpy as np
import sys
import matplotlib.pyplot as plt
sys.setrecursionlimit(500000)
import csv
import pprint



mazemap=np.array([
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1],
[1,0,0,0,0,0,1,1,0,0,0,0,0,0,1,1,0,0,0,1],
[1,0,0,0,0,0,1,1,0,0,0,0,0,0,1,1,0,2,0,1],
[1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,1],
[1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,1],
[1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,1],
[1,0,0,0,1,1,1,1,1,0,0,0,0,0,1,1,0,0,0,1],
[1,0,0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,1],
[1,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,1],
[1,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,1],
[1,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,1],
[1,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,1],
[1,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,0,0,1],
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]])

#data=loadmat('maze.mat')
```

```
PS C:\Users\YOGA\OneDrive\Desktop\Data_Analytics_MT_FULLMARK> python -u "c:\Users\YOGA\OneDrive
enter start row: 2
enter start col: 16
goal indices: 2 17
filled matrix is: [[ 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1]
 [ 1 21 20 19 18 18  1  1 14 14 14 14 14 14  1  1  3  3  3  1]
 [ 1 21 20 19 18 17  1  1 14 13 13 13 13 13  1  1  3  2  3  1]
 [ 1 21 20 19 18 17 16 15 14 13 12 12 12 12  1  1  3  3  3  1]
 [ 1 21 20 19 18 17 16 15 14 13 12 11 11 11  1  1  4  4  4  1]
 [ 1 21 20 19 18 17 16 15 14 13 12 11 10 10  1  1  5  5  5  1]
 [ 1 21 20 19  1  1  1  1  1 13 12 11 10  9  1  1  6  6  6  1]
 [ 1 21 20 19  1  1  1  1  1 13 12 11 10  9  8  7  7  7  7  1]
 [ 1 21 20 19 18 17 17  1  1 13 12 11 10  9  8  8  8  8  8  1]
 [ 1 21 20 19 18 17 16  1  1 13 12 11 10  9  9  9  9  9  9  1]
 [ 1 21 20 19 18 17 16 15 14 13 12 11 10 10  1 10 10 10 10  1]
 [ 1 21 20 19 18 17 16 15 14 13 12 11 11  1  1 11 11 11 11  1]
 [ 1 21 20 19 18 17 16 15 14 13 12 12  1  1  1 12 12 12 12  1]
```



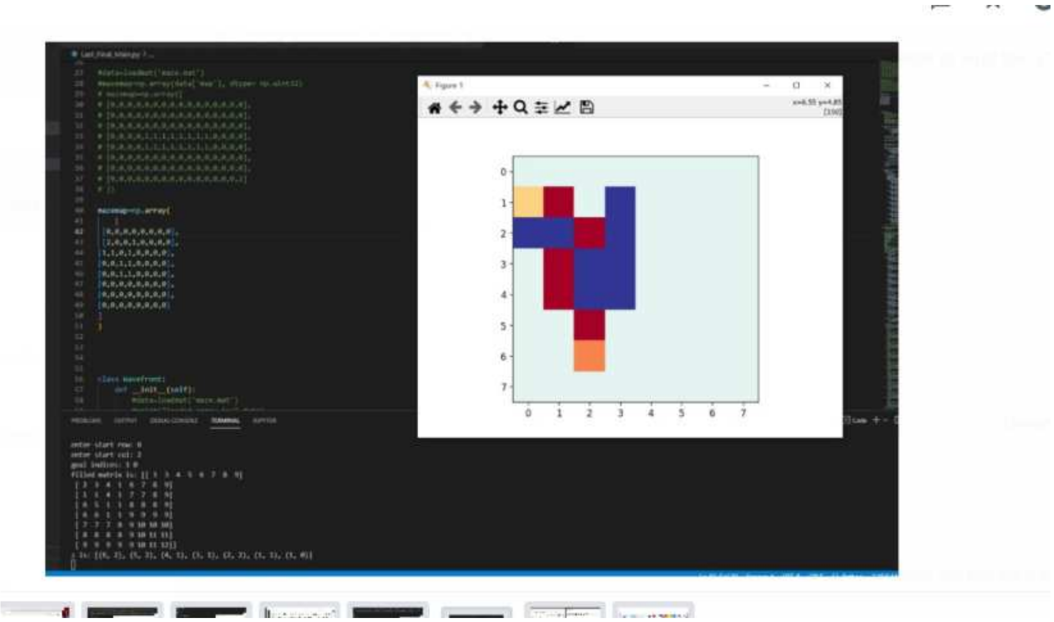
```
from scipy.io import loadmat
import numpy as np
import sys
import matplotlib.pyplot as plt
sys.setrecursionlimit(500000)
import csv
import pprint



mazemap=np.array([
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1],
[1,0,0,0,0,0,1,1,0,0,0,0,0,0,1,1,0,0,0,1],
[1,0,0,0,0,0,1,1,0,0,0,0,0,0,1,1,0,2,0,1],
[1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,1],
[1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,1],
[1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,1],
[1,0,0,0,1,1,1,1,1,0,0,0,0,0,1,1,0,0,0,1],
[1,0,0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,1],
[1,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,1],
[1,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,1],
[1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,1],
[1,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,1],
[1,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,0,1],
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]])

#data=loadmat('maze.mat')
```

```
PS C:\Users\YOGA\OneDrive\Desktop\Data_Analytics_MT_FULLMARK> python -u "
enter start row: 12
enter start col: 17
goal indices: 2 17
filled matrix is: [[ 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
 [ 1 21 20 19 18 18  1  1 14 14 14 14 14 14  1  1  3  3  3  1]
 [ 1 21 20 19 18 17  1  1 14 13 13 13 13 13  1  1  3  2  3  1]
 [ 1 21 20 19 18 17 16 15 14 13 12 12 12 12  1  1  3  3  3  1]
 [ 1 21 20 19 18 17 16 15 14 13 12 11 11 11  1  1  4  4  4  1]
 [ 1 21 20 19 18 17 16 15 14 13 12 11 10 10  1  1  5  5  5  1]
 [ 1 21 20 19  1  1  1  1  1 13 12 11 10  9  1  1  6  6  6  1]
 [ 1 21 20 19  1  1  1  1  1 13 12 11 10  9  8  7  7  7  7  1]
 [ 1 21 20 19 18 17 17  1  1 13 12 11 10  9  8  8  8  8  8  1]
 [ 1 21 20 19 18 17 16  1  1 13 12 11 10  9  9  9  9  9  9  1]
 [ 1 21 20 19 18 17 16 15 14 13 12 11 10 10  1 10 10 10 10  1]
 [ 1 21 20 19 18 17 16 15 14 13 12 11 11  1  1 11 11 11 11  1]
 [ 1 21 20 19 18 17 16 15 14 13 12 12  1  1  1 12 12 12 12  1]
```



```
#data=loadmat('maze.mat')
#mazemap=np.array(data['map'], dtype= np.uint32)
# mazemap=np.array([
# [illegible]
# [illegible]
# [illegible]
# [illegible]
# [illegible]
# [illegible]
# [illegible]
# [illegible]
# ])

mazemap=np.array(
    [
    [0,0,0,0,0,0,0,0],
    [2,0,0,1,0,0,0,0],
    1,1,0,1,0,0,0,0],
    [0,0,1,1,0,0,0,0],
    [0,0,1,1,0,0,0,0],
    [0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0]
    ]
)




class Wavefront:
    def __init__(self):
        #data=loadmat('maze.mat')
```

```
enter start row: 6
enter start col: 2
goal indices: 1 0
filled matrix is: [[ 1  3  4  5  6  7  8  9]
 [ 2  3  4  1  6  7  8  9]
 [ 1  1  4  1  7  7  8  9]
 [ 6  5  1  1  8  8  8  9]
 [ 6  6  1  1  9  9  9  9]
 [ 7  7  7  8  9 10 10 10]
 [ 8  8  8  8  9 10 11 11]
 [ 9  9  9  9  9 10 11 12]]
 is: [(6, 2), (5, 2), (4, 1), (3, 1), (2, 2), (1, 1), (1, 0)]
```

Examples for Shortest Distance(Cases and Debugging):

Result: [(1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (1,

```
7), (1, 8), (1, 9), (1, 10), (2, 11), (3, 12), (4, 12), (5, 13), (6, 14), (7, 15)]
```

```
 Result:  [(5, 3), (5, 4), (5, 5), (5, 6), (5, 7), (5, 8), (5, 9), (5, 10), (5, 11), (5, 12), (5,
13), (6, 14), (7, 15)]
```

```
Result:  [(1, 1), (1, 2), (1, 3), (1, 4), (2, 5), (3, 6), (3, 7), (3, 8), (3, 9),

(3, 10), (4, 11), (5, 12), (6, 13), (7, 14), (7, 15), (6, 16), (5, 16), (4, 16), (3, 16), (2, 17)

Result:  [(0, 1), (1, 2), (1, 3), (1, 4), (2, 5), (3, 6), (3, 7), (3, 8), (3, 9),

(3, 10), (4, 11), (5, 12), (6, 13), (7, 14), (7, 15), (6, 16), (5, 16), (4, 16), (3, 16), (2, 17)
```

```
Result:  [(0, 1), (1, 2), (1, 3), (1, 4), (2, 5), (3, 6), (3, 7), (3, 8), (3, 9),

(3, 10), (4, 11), (5, 12), (6, 13), (7, 14), (7, 15), (6, 16), (5, 16), (4, 16), (3, 16), (2,
17)]
```

```
Result:  [(13, 19), (12, 18), (11, 18), (10, 18), (9, 18), (8, 18), (7, 18), (6, 18), (5, 18),
(4, 18), (3, 18), (2, 17)]
```

```
Result:  [(12, 1), (12, 2), (12, 3), (12, 4), (12, 5), (12, 6), (12, 7), (12, 8),

(12, 9), (12, 10), (11, 11), (10, 12), (9, 13), (8, 14), (7, 15), (6, 16), (5, 16), (4, 16), (3,
16), (2, 17)]
```