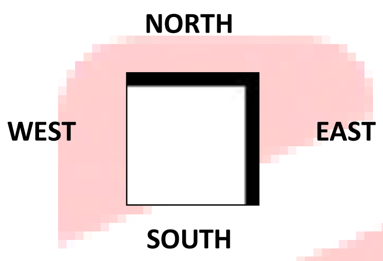
# Task-1 – Practice

Terrain navigation is one of the important challenges in robotic technology. One of the many pre-requisites for a robot rover is the ability to navigate a pre-explored terrain. These tasks involve complex challenges like image processing and path planning.

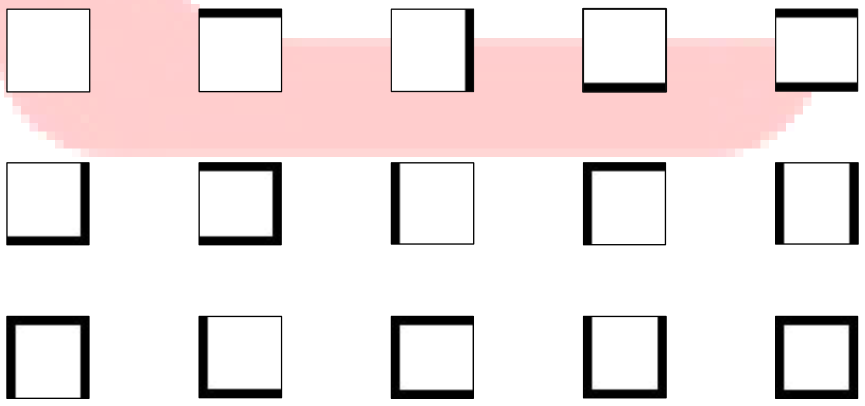
Navigating through a pre-explored or unexplored area is very much like navigating through a maze of connected passages. The following task will be an exercise in understanding these concepts. This task consists of 3 sections: **Section 1, Section 2** and **Section 3**. You have to follow the instructions and complete the sections in a sequential manner. You are expected to understand the problem at hand and use image processing techniques to generate a solution.

## Section – 1

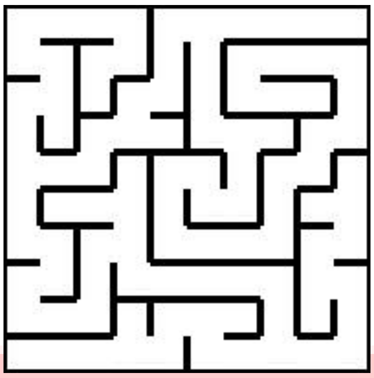
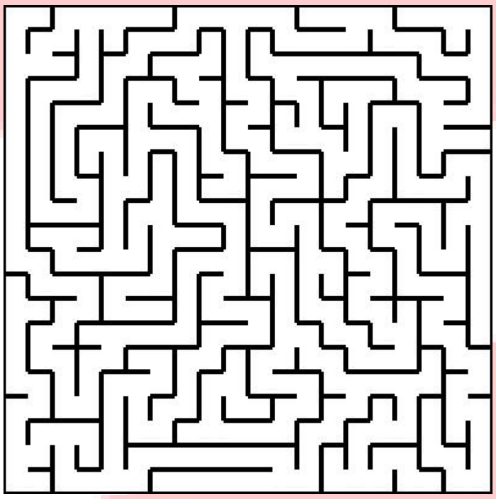
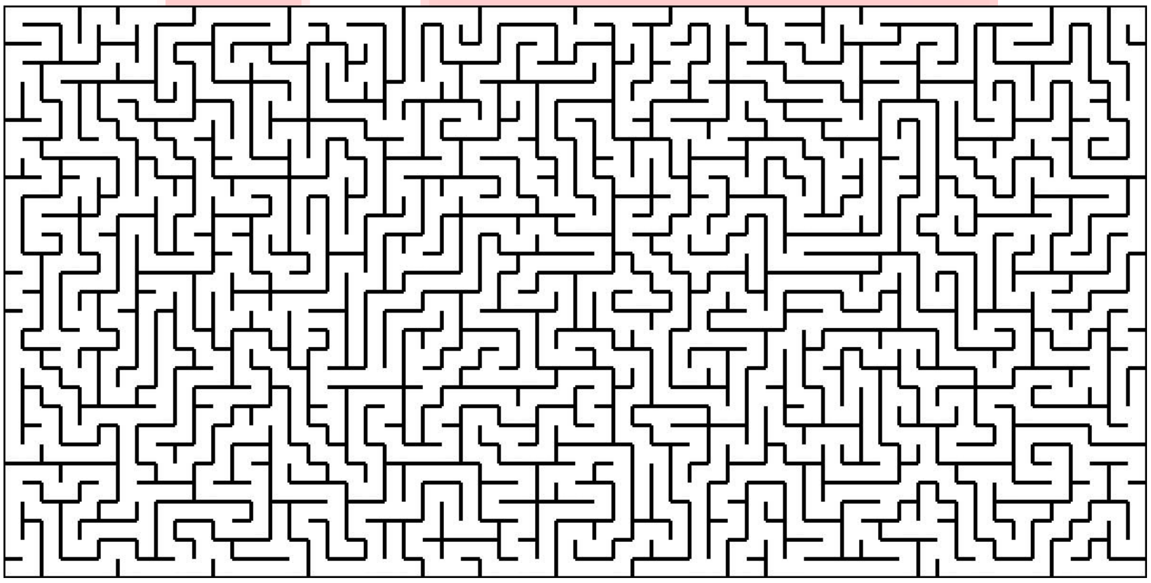
Let us look at the basic building element of a maze, the **Cell.**



As shown in the figure above, the dark bands indicate presence of walls. The above cell has open passages towards WEST and SOUTH while the NORTH and EAST passages are closed by walls. There are 15 more cells as shown below having the different possible wall configurations.



By combining cells having different wall configuration using special we can create mazes like these:

Cells are numbered in a maze using a coordinate system as given below:

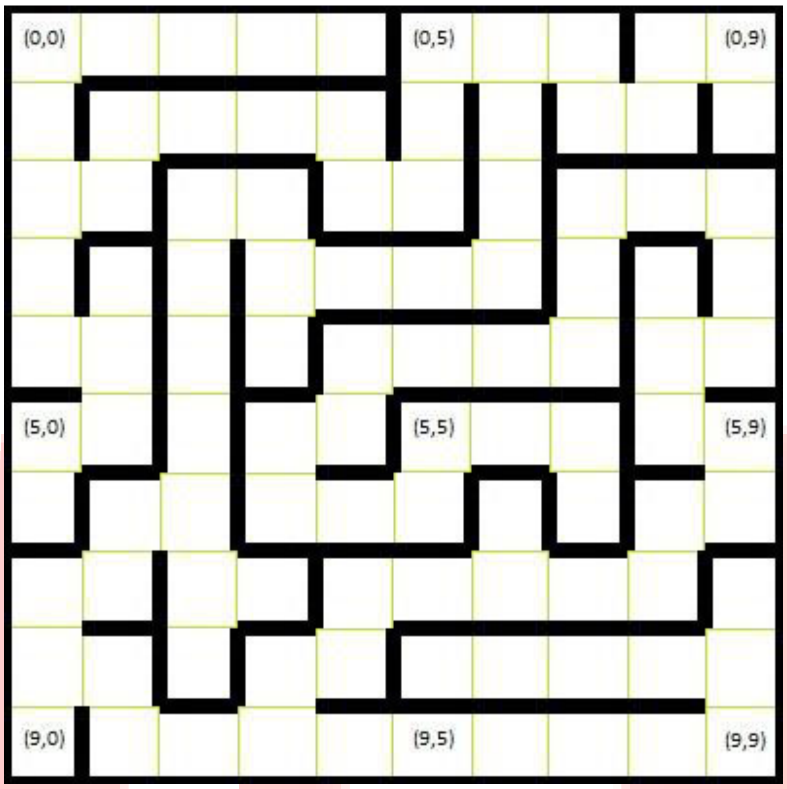


Figure 1: Coordinate Numbering in Maze Images

Given such a maze, the problem is to navigate from a **START** location in the maze to an **END** location, where the coordinates of **START** and **END** have been given.

The first step to solve such a problem is to ***examine*** cells in the maze image as well as ***analyze*** how these cells connect with each other.
You start by analyzing a maze and finding the neighbors of each of the specified cells.

1. Open the **Task1_Practice folder**.
2. In the **Section-1 folder**, open the ***section1.py file***.
3. Follow the instructions to modify functions in the *section1.py* file as given below:

| **MAIN** |
|---|
| <pre>if __name__ == '__main__':<br>    filePath = 'maze00.jpg'## Specify the filepath of image here<br>    img = main(filePath)<br>    cv2.imshow('canvas', img)<br>    cv2.waitKey(0)<br>    cv2.destroyAllWindows()</pre> |
| **This section of code calls the main() function with a filepath which you can specify. You can specify filepath as "maze00.jpg" to "maze09.jpg". Apart from the filepath please do not change any other code in this section.** |

**MAIN FUNCTION**

```
##  Main function takes the filepath of image as input.
##  You are not allowed to change any code in this function.
def main(filePath):
    img = readImage(filePath)
    coords = [(0,0),(9,9),(3,2),(4,7),(8,6)]
    string = ""
    for coordinate in coords:
        img = colourCell(img, coordinate[0], coordinate[1], 150)
        neighbours = findNeighbours(img, coordinate[0],
     coordinate[1])
        print neighbours
        string = string + str(neighbours) + "\n"
        for k in neighbours:
            img = colourCell(img, k[0], k[1], 230)
    if __name__ == '__main__':
        return img
     else:
        return string + "\t"
```

**This is the main() function which is being called in the previous section. The functions called in this main() function will be explained in detail . You are expected to understand what this snippet of code does, on your own. You can refer to online resources to learn the python syntax such as given here: https://www.tutorialspoint.com/python/**

The following section will explain the functions used in the ***section1.py*** script in detail:

**FUNCTION**

```
def readImage(filePath):
    ##############  Add your Code here  #################


    #####################################################
    return binaryImage
```

**The readImage( ) accepts the file path of the test image as input and returns the image in binary format. Please refer to the image processing resources on the portal for information on how to convert an image to its binary equivalent.**

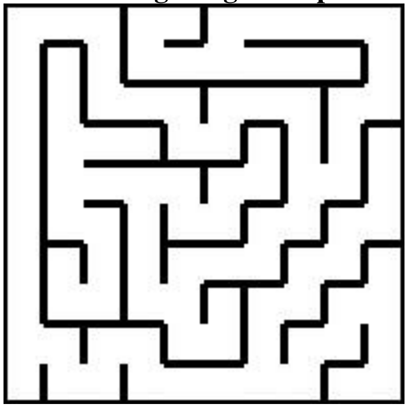
**FUNCTION**

```
def findNeighbours(img,row,column):
    neighbours = []
    ##############  Add your Code here   #################


    ######################################################
    return neighbours
```

**The findNeighbours function accepts a binary image and the row and column coordinates of a particular cell as input and returns the neighbouring cells which are traversable from that particular cell.**

For example suppose we provide the following image as input to the function:



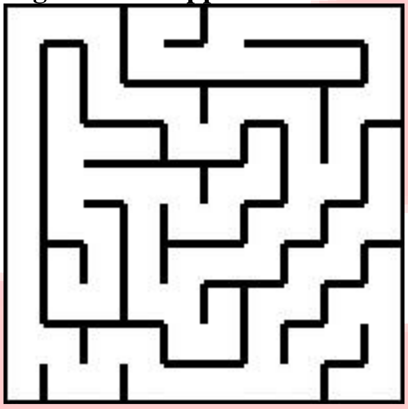
Assuming that we pass the other arguments as findNeighbours(img, 0,0) then this function will return [(0,1) , (1,0)] as output.
If we use the same image and pass other arguments as findNeighbours(img, 4,4) which means the current cell coordinates are (4,4) then the function will return the value [(4,3), (5,4)] as output.

**FUNCTION**

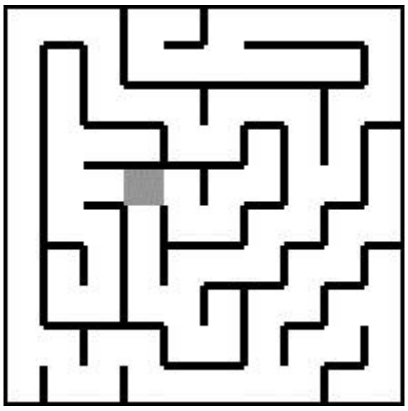
```
def colourCell(img,row,column,colourVal):
    ##############    Add your Code here    ################


    #######################################################
    return img
```

This function will highlight or 'paint' the cell whose coordinates have been specified by passing the appropriate row and column arguments. Suppose we use the following maze image:

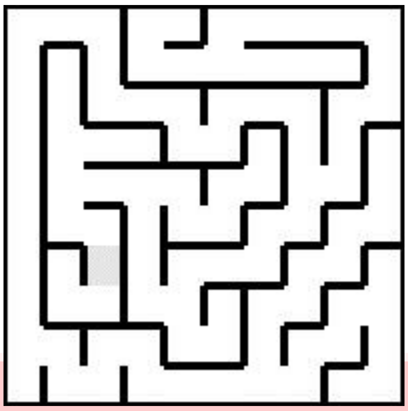


Assume that we pass the other arguments as colourCell(img, 4,3,150). The function will return the following output image:



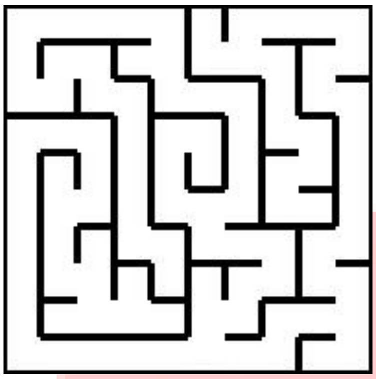
The last argument colourVal will be used to specify the intensity of colour we use to highlight the

cell.
If we pass the arguments as colourCell(img, 6, 2, 230) , the function will return the following output image:



**EXPECTED OUTPUT OF PROGRAM**

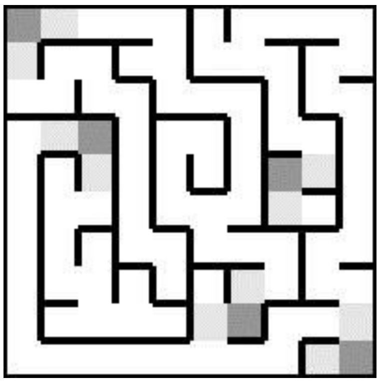
For the *section1.py* script, if we provide the filepath of the following image in the space indicated:



The expected output for the program will be:

```
>>>
[(0, 1), (1, 0)]
[(8, 9), (9, 8)]
[(4, 2), (3, 1)]
[(4, 8), (5, 7)]
[(7, 6), (8, 5)]
```

Output image will be:

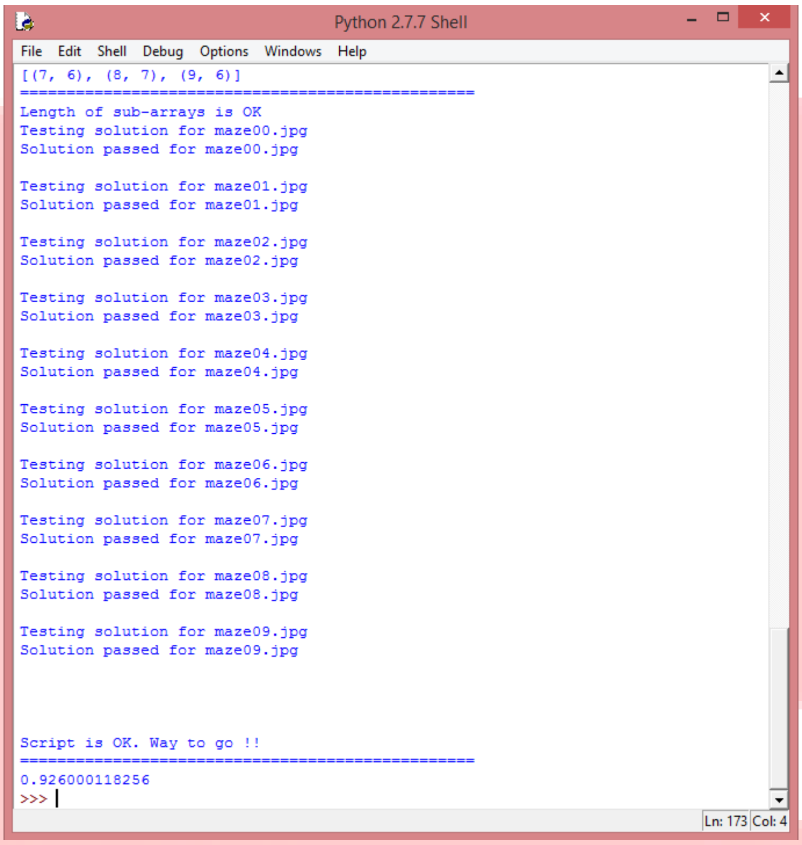


## Testing the Solution

In the **Section-1 folder,** in addition to maze test images and ***section1.py*** there are three more files, namely the ***section1.txt*** file, ***hash.txt file*** and the ***TestSuite_1.py.***

You are **not allowed** to make any changes to these two files. Anybody found to have tampered with these files will be disqualified.

After you are done modifying the code in the ***section1.py*** file, open the ***TestSuite_1.py*** file and run it in the python shell. The output of **TestSuite_1** script should resemble the following screenshot.



```
[(7, 6), (8, 7), (9, 6)]
==================================================
Length of sub-arrays is OK
Testing solution for maze00.jpg
Solution passed for maze00.jpg

Testing solution for maze01.jpg
Solution passed for maze01.jpg

Testing solution for maze02.jpg
Solution passed for maze02.jpg

Testing solution for maze03.jpg
Solution passed for maze03.jpg

Testing solution for maze04.jpg
Solution passed for maze04.jpg

Testing solution for maze05.jpg
Solution passed for maze05.jpg

Testing solution for maze06.jpg
Solution passed for maze06.jpg

Testing solution for maze07.jpg
Solution passed for maze07.jpg

Testing solution for maze08.jpg
Solution passed for maze08.jpg

Testing solution for maze09.jpg
Solution passed for maze09.jpg



Script is OK. Way to go !!
==================================================
0.926000118256
>>> 
```

If it runs successfully without any errors then your solution is correct and it passed all 10 test cases provided to you.

Please note that files submitted by you will be run through similar test cases.

Now you can proceed to Section-2 of Task 1.

Cheers!!