CMSC 201 Fall 2017
Project 3 – Maze Solver

Assignment: Project 3 – Maze Solver
Due Date:
Design Document: Friday, December 1st, 2017 by 8:59:59 PM
Project: Friday, December 8th, 2017 by 8:59:59 PM
Value: 80 points

Collaboration: For Project 3, **collaboration is not allowed** – you must work individually. You may still come to office hours for help, but you may not work with any other CMSC 201 students.

Make sure that you have a complete file header comment at the top of each file, and that all of the information is correctly filled out.

```
# File:    FILENAME.py
# Author:  YOUR NAME
# Date:    THE DATE
# Section: YOUR DISCUSSION SECTION NUMBER
# E-mail:  YOUR_EMAIL@umbc.edu
# Description:
#    DESCRIPTION OF WHAT THE PROGRAM DOES
```

For Project 3 you will have to turn in a "design document" in addition to the actual code. The design document is intended to help you practice deliberate construction of your program and how it will work, rather than coding as you go along, or starting without a plan.

Instructions

For this project, you will be creating a single program, but one that is bigger in size and complexity than any individual homework problem. This assignment will focus on file I/O, manipulating lists, calling functions, and recursion. For this assignment, more than any other this semester, **planning ahead and designing your program will be very, *very* important**!

The design for Project 3 is entirely up to you – suggestions are provided within the project description, but you are not required to use them.

At the end, your Project 3 file must run without any errors.
It must also be called proj3.py (case sensitive).

Additional Instructions – Creating the proj3 Directory

During the semester, you'll want to keep your different Python programs organized, organizing them in appropriately named folders (also known as directories).

You should create a directory in which to store your Project 3 files. We recommend calling it **`proj3`**, and creating it inside a newly-created directory called **`Projects`** inside the **`201`** directory.

If you need help on how to do this, refer back to the detailed instructions in Homework 1.

Objective

Project 3 is designed to give you practice with file I/O, two-dimensional lists, creating and calling functions, and recursion. You'll need to use practically everything you've learned so far, and will need to do some serious thinking about how all of the pieces you need to create should fit together.

Remember to enable Python 3 before running and testing your code:

```
scl enable python33 bash
```

Specification

Prior to this assignment, **you should be familiar with the entirety of the Coding Standards**, available on Blackboard under "Assignments" and linked on the course website at the top of the "Assignments" page.

You should be commenting your code (including function headers), and using constants in your code (not magic numbers or strings). Any numbers other than 0 or 1 are magic numbers!

You will **lose major points** if you do not follow the 201 coding standards.

If you have questions about commenting, whitespace, or any other coding standards, please come to office hours.

Additional Specifications

For this assignment, **you must use recursion to determine the solution to the maze.** No other functions are required to be recursive. You also must create and call the three functions described in "Requirements". All other design decisions are up to you.

You may ***NOT*** change the maze once it's read in. Do not add "walls" to the maze, or otherwise update the information. (This applies even if you make a copy of the maze – do ***not*** update or change the maze, even in a copy!)

For this assignment, you do need to worry about "input validation." You may assume that the user will enter the correct type of input (for example, an integer if one is asked for, a float if one is asked for), but the input may be negative, outside of the allowable range, or "bogus" (in the case of strings).

If the user enters a different type of data than what you asked for, your program may crash. This is acceptable.

It is also acceptable if your program crashes when a filename is entered that either does not exist, or has the wrong formatting for the minesweeper board.

Details

For this project, you will harness the computing power of Python to solve a maze, using a recursive search algorithm. You will need to understand algorithms, Python data structures, file I/O, and recursion to complete this project.

The maze will be rectangular, comprised of square spaces. The Maze Solver can move freely between two adjacent squares, as long as the movement is horizontal or vertical (no diagonal moves), and the way is not blocked by a wall. The dimensions, finishing square, and configuration of each maze are provided in a separate file. The starting square from which the maze solution is attempted is chosen by the user.

Your Maze Solver will start from the user's choice of starting position, and will search out a path to the finish square. It can travel right, down, left, or up, as long as it doesn't go through any walls. When it finds a solution, it will print out the successful path. If there is no successful path, it must print out that there is no solution.

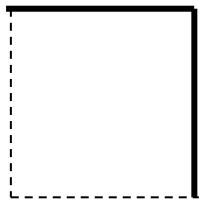
Your Maze Solver must use a recursive algorithm. Starting from the start square, your algorithm will scan for all the adjacent squares it can legally travel to in the next step. For each candidate "next square," it will first check that it has not already been there. If not, it will try adding that square to the path built so far, and will use recursion to find a path from that new square to the end. If that recursion fails, it moves on to the next candidate. If all "next square" candidates fail, this instance of the recursion itself fails.

This is what is known as a "**brute force**" method in computer science: try every possible combination until you either find an answer, or run out of options and give up. Although this method might seem very silly and slow, your Python program will beat you every time, simply because its speed at solving problems like this is much faster than yours.

Input File:

The input files will all have the following format:

- Line 1: two integers, specifying the number of rows and columns in the maze
- Line 2: two integers, specifying the row and column position of the "finish"
- Remaining lines: The remainder of the lines specify the wall layout for each of the squares in the maze, starting with the top left square, moving across the entire top row, then continuing from the left-most square on the second row, and so on.
  Each line specifies the wall layout for one square using four integers, describing the walls in order: right, bottom, left, and top.
  A "1" means there is a wall, and a "0" means there is no wall. So a square that is specified as "1 0 0 1" has a wall on the right, is open on the bottom, is open on the left, and has a wall on the top.



Notice that the way we specify the maze repeats a lot of information: most of the walls in the maze are described in the specification for both of the squares that touch it. (The only exception are the walls that make up the edge, which only touch a single square.) This makes the specification for the squares completely independent, which will actually make it easier for you to construct and use the maze data structure when your Maze Solver is searching for a solution.

You can count on the data file being consistent and correct. (In other words, if a square says it has a right wall, the square to the right of it is guaranteed to say that it has a left wall.) You can also count on it being completely boxed in, with walls around the outside.

After your program reads in and "constructs" the maze, it will initialize a few other things, ask the user for their starting point, and initiate a recursive search. At the end, it will print out its results: if a path was found, it will print out the steps taken to reach the end; otherwise, it will print out that no solution was found from that starting point. (See the sample run for details.)

Data Structures:

To represent your maze, you should use a 3-dimensional list, also known as a list-of-lists-of-lists. This might sound intimidating, but you already know enough to deal with this confidently. (Please, please come to office hours if you don't!)

As an example, if you have a Python list named "**`square_0`**", which has four numbers in it (representing the 4 walls), and you have another one much like it named "**`square_1`**", and yet another list "**`square_2`**", you can then make a list-of-lists called "**`row_0`**" with this code:

```
square_0 = [1, 1, 1, 0]
square_1 = [0, 1, 0, 1]
square_2 = [1, 0, 1, 0]

row_0 = []
row_0.append(square_0)
row_0.append(square_1)
row_0.append(square_2)
```

And if you did that a few more times to create more rows, you can then make a list-of-lists-of-lists called (for example) "**`maze`**", by doing something like:

```
maze = []
maze.append(row_0)
maze.append(row_1)
maze.append(row_2)
```

Except, of course, you would use loops to put this together from the input file.

You could then do cool things like test for a wall on a specific side in a specific square via:

```
if maze[1][2][TOP] == 1   # row 1, col 2
```

By the way, you could also have split the above into two steps by doing something like:

```
row = 1
col = 2
square = maze[row][col]
if square[TOP] == 0:
    # explore in the top direction because it's open
```

In case you didn't get the hint, **you should definitely make use of global constants for the indexes of the directions,** to help you know that the right wall is in position 0, the top wall is in position 3, etc. (In fact, if you use constants, you don't even need to remember this after you create them!)

To store your path, you will probably want to use a list, possibly of lists; for example: `[row, col]`. Remember that lists are mutable, and that before you start each recursive call, you should make a deep copy of the path list first – we'll discuss making a deep copy of a 2D list in Lecture21.

Recursive Algorithm:

What is your base case? Is there more than one base case?

Think very carefully about what information your recursive algorithm will need at each step of the way.

- Does it need a copy of the maze?
- Does it need to know the starting point?
- Does it need to know the current point?
- Does it need to know the finish point?
- Does it need to know the path so far?
- Does it need to know the number of rows?
- Does it need to know the number of columns?

The answer is not "yes" to all of these, but your recursive function will need at least a few of these at every step.

Requirements:

Your program must have the following functions. You may include additional ones that you think are necessary:

`main()`

This should handle the calls to most of the other functions.

`readMaze(filename)`

This function should read in the maze specification from the filename specified, and return a maze data structure (however you chose to implement that structure). **You are not required to use a 3-dimensional list**, but we think it is the best option. The maze data structure should be designed to make it simple to determine what the row and column dimensions are.

`searchMaze(maze, ???)`

This function takes a maze data structure as created by **`readMaze()`**, along with some additional information that you will need to decide. This function must be recursive. After it completes and all of the recursion has ended, **`searchMaze()`** should return either a complete solution path (if it found one) or **`None`** if there was no solution.

Input Validation:

You will need to validate the following things:

- When asked to enter the starting point, the user might enter any whole number. If the row or column entered is not valid, it must reprompt, telling them what the valid options are.

Sample Maze Files:

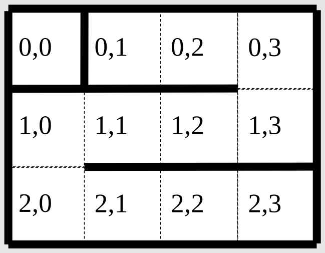
We have provided sample files for you. The first is used in the sample output, and is simple. The second one is more of a “challenge” for your Maze Solver.

```
cp /afs/umbc.edu/users/k/k/k38/pub/cs201/maze1.txt .
cp /afs/umbc.edu/users/k/k/k38/pub/cs201/maze2.txt .
```

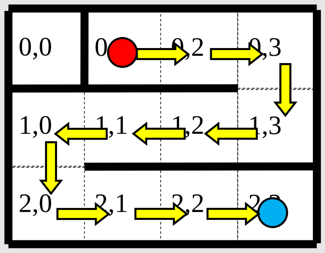
We also highly recommend creating your own simple test mazes, like a simple 1x4 “corridor”, or a 2x4 maze with a single turn. Feel free to share your maze files with other students for testing. (Remember, you can compare sample output, as long as no one sees any of your code.)

Sample Output:

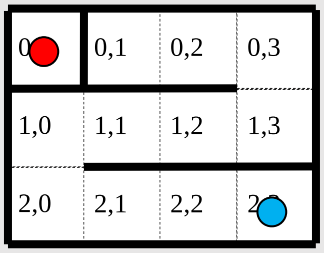
Here is some sample output, with the user input in blue.



```
bash-4.1$ python proj2.py
Welcome to the Maze Solver!
Please enter the filename of the maze: maze1.txt
Please enter the starting row: 9
Invalid, enter a number between 0 and 2 (inclusive): 22
Invalid, enter a number between 0 and 2 (inclusive): 0
Please enter the starting column: -1
Invalid enter a number between 0 and 3 (inclusive): 1
Solution found!
(0, 1)
(0, 2)
(0, 3)
(1, 3)
(1, 2)
(1, 1)
(1, 0)
(2, 0)
(2, 1)
(2, 2)
(2, 3)
```



```
bash-4.1$ python proj2.py
Welcome to the Maze Solver!
Please enter the filename of the maze: maze1.txt
Please enter the starting row: 0
Please enter the starting column: 0
No solution found!
```



Debugging:

We highly recommend the use of ***debug statements*** when you're working on this project. Debug statements are simple print statements in your code that give you a bit more information on what exactly is going on.

For example, you might want to know in which direction your Maze Solver is moving:

```
     print("Currently moving to the right")
```

Or you might want to know what the path looks like so far:

```
     print("The current path is:", path)
```

The following sample output can give you an idea of how helpful debug statements can be when figuring out what exactly your program is doing "behind the scenes." When using debug statements, **don't forget to remove them** before turning in your final project!

The debug statements are in **green** for clarity. We've turned the background a light yellow to differentiate this from normal sample output.

```
bash-4.1$ python proj2.py
Welcome to the Maze Solver!
Please enter the filename of the maze: maze1.txt
Please enter the starting row: 0
Please enter the starting column: 0
     DEBUG: Looking at row 0 and column 0
     DEBUG: Can't go right
     DEBUG: Can't go bottom
     DEBUG: Can't go left
     DEBUG: Can't go top
No solution found!
```

Again, don't forget to remove all of your debug statements before turning in your final project!

(Additional sample debugging output on the next page.)

The debug statements are in **green** for clarity. We've turned the background a light yellow to differentiate this from normal sample output.

```
bash-4.1$ python proj2.py
Welcome to the Maze Solver!
Please enter the filename of the maze: maze1.txt
Please enter the starting row: 0
Please enter the starting column: 1
     DEBUG: Path currently [[0, 1]]
     DEBUG: Looking at row 0 and column 1
     DEBUG: Moving right
     DEBUG: Path currently [[0, 1], [0, 2]]
     DEBUG: Looking at row 0 and column 2
     DEBUG: Moving right
     DEBUG: Path currently [[0, 1], [0, 2], [0, 3]]
     DEBUG: Looking at row 0 and column 3
     DEBUG: Can't go right
     DEBUG: Moving bottom
     DEBUG: Path currently [[0, 1], [0, 2], [0, 3], [1,
3]]
     DEBUG: Looking at row 1 and column 3
     DEBUG: Can't go right
     DEBUG: Can't go bottom
     DEBUG: Moving left
     DEBUG: Path currently [[0, 1], [0, 2], [0, 3], [1,
3], [1, 2]]
     DEBUG: Looking at row 1 and column 2
     DEBUG: Can't go bottom
     DEBUG: Moving left
     DEBUG: Path currently [[0, 1], [0, 2], [0, 3], [1,
3], [1, 2], [1, 1]]
     DEBUG: Looking at row 1 and column 1
     DEBUG: Can't go bottom
     DEBUG: Moving left
     DEBUG: Path currently [[0, 1], [0, 2], [0, 3], [1,
3], [1, 2], [1, 1], [1, 0]]
     DEBUG: Looking at row 1 and column 0
     DEBUG: Moving bottom

[etc]
```

Points

The project is worth a total of 80 points. Of those points, 10 will be based on your design document (creation of it and following it), 10 will be based on following the coding standards, and the other 60 will be based on the functionality and completeness of your project.

Design Document

The design document will ensure that you begin seriously thinking about your project way early on. This will not only give you important experience doing design work, but will help you gauge the number of hours you'll need to set aside to be able to complete the project. **Your design document must be called design3.txt.**

For Project 3, you are creating the design entirely on your own.
You **may NOT work with another student** to "brainstorm" a solution or discuss any general approaches or requirements. If you need assistance with the design document, come to office hours.

Your design document must have four separate parts:

1. A file header, similar to those for your assignments
2. Constants
    a. A list of all the constants your program will need, including a short comment describing what each "group" of constants is for
3. Function headers
    a. A complete function header comment for each function your plan to create, including the description, inputs, and outputs
4. Pseudocode for main()
    a. A brief but descriptive breakdown of the steps your main() function will take to completely solve the problem; note function calls under the relevant comment (if applicable)

Your design can follow the same general format as the design for Project 1.

Your **`design3.txt`** file will be compared to the **`proj3.py`** file that you submit. Minor changes to the design are allowed. A minor change might be the addition of another function, or a small change to **`main()`**.

Major changes between the design and your project will lose you points. This would indicate that you didn't give sufficient thought to your design.
(If your submitted design doesn't work, it is generally better to lose the points on the design, and to have a functional program, rather than turning in a broken program that follows the design. The ultimate decision is up to you.)

To submit your design document, use

```
linux1[4]% submit cs201 PROJ3_DESIGN design3.txt
Submitting design3.txt...OK
linux1[5]% 
```

Submitting

Once your **`proj3.py`** or **`design3.txt`** file is complete, it is time to turn it in with the **`submit`** command. (You may also turn the design or project in multiple times, as you reach new milestones or complete each piece. To do so, run **`submit`** as normal.)

To submit your design file (which is due Friday, December 1st, 2017 by 8:59:59 PM), use the command:

```
linux1[4]% submit cs201 PROJ3_DESIGN design3.txt
Submitting design3.txt...OK
linux1[5]% 
```

To submit your project file (which is due Friday, December 8th, 2017 by 8:59:59 PM), use the command:

```
linux1[4]% submit cs201 PROJ3 proj3.py
Submitting proj3.py...OK
linux1[5]% 
```

If you don't get a confirmation like the one above, check that you have not made any typos or errors in the command.

You can check that your project and/or design was submitted by following the directions in Homework 0. Double-check that you submitted your files correctly, since **an empty file will result in a grade of zero for this assignment.**