CMSC 201 Fall 2016
Project 2 – Maze Solver

Assignment: Project 2 – Maze Solver
Due Date: Tuesday, December 13th, 2016 by 8:59:59 PM
Value: 80 points

Collaboration: For Project 2, 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.

Your collaboration statement should state that
`Collaboration was not allowed on this assignment.`

Make sure that you have a complete file header comment at the top of your 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
# Collaboration:
#    COLLABORATION STATEMENT GOES HERE
```

Project 2 is an assignment where we won't be telling you exactly what to do! You will get the chance to make your own decisions about how you want your program to handle things, what its functions should be called, and how you want to go about designing it.

Project 2 will also be ***substantially longer*** than any of the single homework assignments you've completed so far, so make sure to take the time to plan ahead, and don't do any "cowboy" coding!

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

```
scl enable python33 bash
```

Instructions

For this assignment, you'll need to follow the class coding standards, a set of rules designed to make your code clear and readable. The class coding standards are on the website, linked at the top of the "Assignments" page. You can also access them directly via this URL (http://goo.gl/yEoGfC).

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

Adhere to the coding standards by including function header comments for each of the functions (other than main). Follow the instructions and example provided in the coding standards document when creating your function header comments. Failing to include function header comments will lose you points.

Re-read the coding standards!

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

NOTE: Your filename for this project must be **`proj2.py`**

NOTE: You must use **`main()`** in your file.

Objective:

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 the 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.

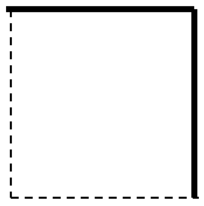
Your Maze Solver must use a recursive algorithm. Starting from the start square, your algorithm will scan for all the adjacent squares it can legally travel to in the next step. For each candidate “next square,” it will first check that it has not already been there. If not, it will try adding that square 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'll need to 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, col = 1, 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 know this after you create them!)

To store your path, you will probably want to use a list, possibly of tuples; 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 list first.

Recursive Algorithm:

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:

For this project, we will require that you validate input from the user. You can assume that the user will enter the right type of input, but not that they will enter a correct value. In other words, a user will always give an integer when you expect one, but it may be a negative or otherwise invalid value.

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.

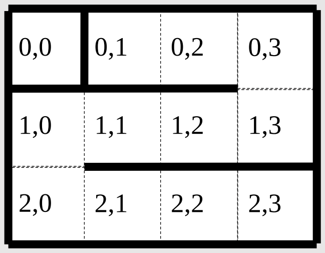
Sample Maze Files:

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.

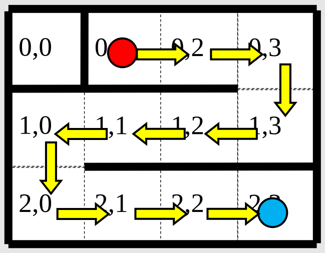
```
cp /afs/umbc.edu/users/k/k/k38/pub/cs201/maze1.txt .
cp /afs/umbc.edu/users/k/k/k38/pub/cs201/maze2.txt .
```

Sample Output:

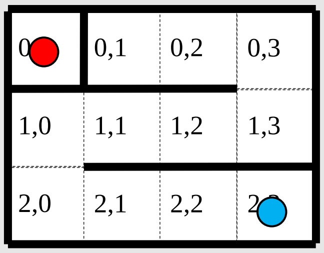
Here is some sample output, with the user input in blue.



```
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 statement, **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]
```

Submitting

Once your **`proj2.py`** file is complete, it is time to turn it in with the **`submit`** command. (You **may turn in the file multiple times** as you complete each milestone. To do so, simply submit **`proj2.py`** each time you complete a part of the project. Each new **`submit`** will overwrite the old file.)

You must be logged into your GL account, and you must be in the same directory as your Project 2 python file. To double-check this, you can type **`ls`**.

```
linux1[3]% ls
proj2.py
linux1[4]% █
```

To submit your Project 2 python files, we use the **`submit`** command, where the class is **`cs201`**, and the assignment is **`PROJ2`**. Type in (all on one line) **`submit cs201 PROJ2 proj2.py`** and press enter.

```
linux1[4]% submit cs201 PROJ2 proj2.py
Submitting proj2.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 homework was submitted by following the directions in Homework 0. Double-check that you submitted your homework correctly, since **an empty file will result in a grade of zero for this assignment.**