A Bagful of Coins

Submission to website: Monday, April 3, 10pm

Checkoff by LA/TA: Thursday, April 6, 10pm

This lab assumes you have Python 3.5 installed on your machine. Please use the Chromium web browser.

This lab was designed to be a 1-week lab, though you are getting 2 weeks including Spring Break. We expect that most of the work for this lab be done before Spring Break.

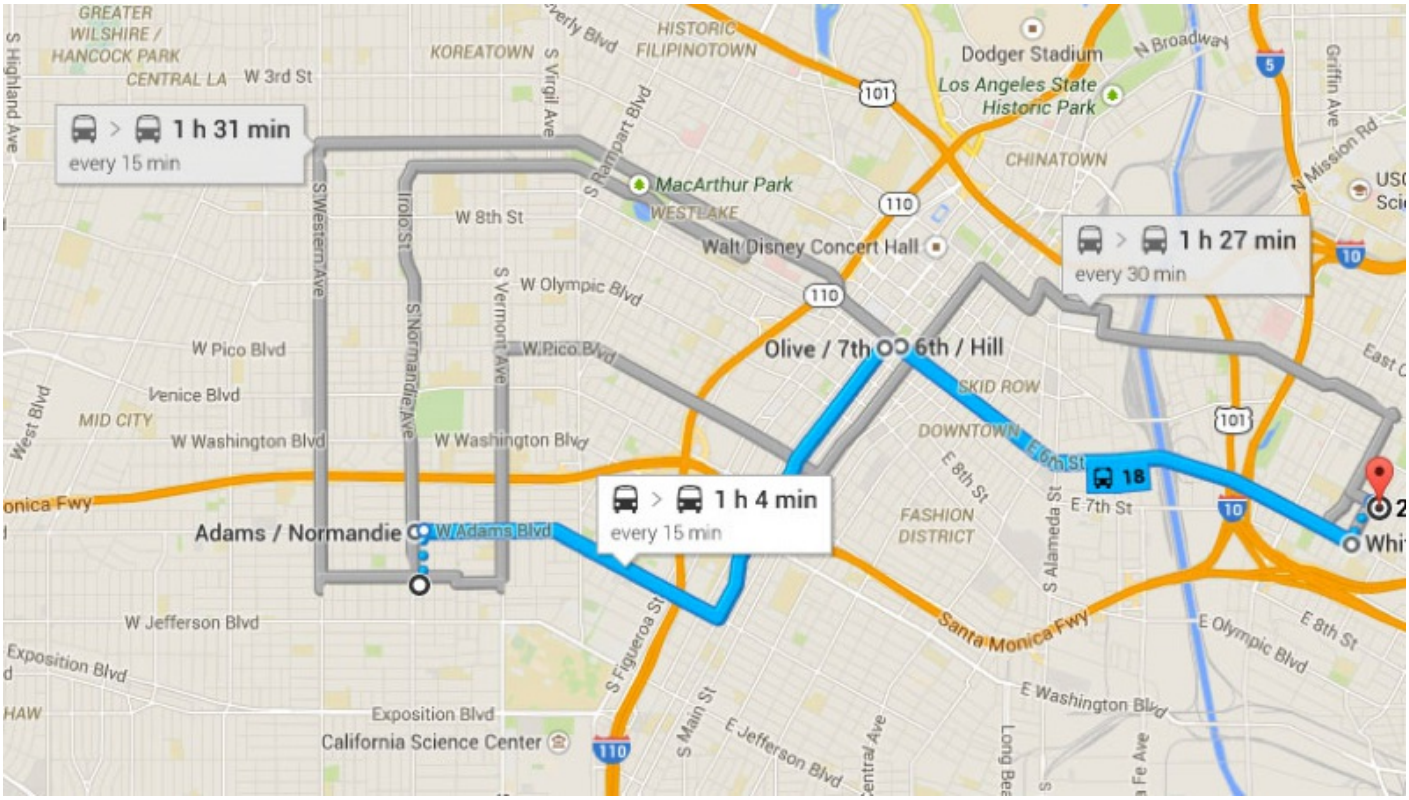
This is a backtracking problem, and we expect `solve_maze` will be **recursive**! For this lab, we will not be releasing a hints file. Remember not to use imports!

This lab has **15 tests** and **3 coding points**. The coding points will be assigned during check-off based on the organization and readability of your code.

- **Helper Functions:** appropriate use of helper functions helps avoid code repetition and enhances readability and testability.
- **Docstrings:** include docstrings for all your helper functions that explain the expected argument values/types and what's get returned. If the function modifies one of its arguments (e.g., a list or dict) that should be mentioned. Feel free to refer to python's docstring conventions for examples.
- **Comments:** good comments include a "mission statement" for each block of code: functions, loops, if's, etc. It should be possible to get an accurate sense of the algorithm you've implemented just by looking at the comments. If one has to read the code to know what's happening, you don't have enough comments yet! You don't have to comment every line, but you should include comments if it's not obvious what's happening or if there's some particular constraint your code relies on for correct operation.

Introduction

A GPS optimizes the path from a start to an end destination, using a variety of factors including traffic patterns, mileage, weather and more. It uses certain heuristics to make this search easier, like that freeways have higher speed limits than residential neighborhoods and that changing trains at the subway often takes added time.



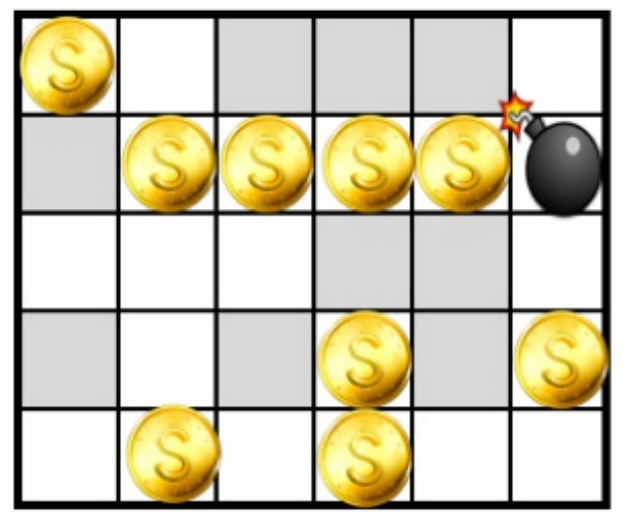
In this lab, you are going to implement a search algorithm analogous to to Google Maps. However, instead of optimizing for weather and traffic patterns in a city grid, you are going to be optimizing for coins in a fictional maze. After all, some neighborhoods in Boston are like a maze.

You must find all of the paths through the maze, collecting coins as you go, but be wary of bombs! Hitting one will blow up all coins collected up to that point along the path. To make your life easier, **you are only permitted to move down or right through the maze**

Maze Representation

In this lab, we represent mazes as `m`-by-`n` grids. Open space is represented with `0`, walls with `1`, coins with `"c"`, and bombs with `"b"`. You are free to move through open space, coins, or bombs, but cannot move through walls.

Below is an example, `maze_1`, where white is open and walls are gray:



Mazes are represented as dictionaries. `maze_1`, from above, is represented as:

```
maze_1 : {
            "dimensions": [5, 6],
            "maze": [["c", 0,  1,  1,  1,  0 ],
                     [ 1, "c","c","c","c","b"],
                     [ 0,  0,  0,  1,  1,  0 ],
                     [ 1,  0,  1, "c", 1, "c"],
                     [ 0, "c", 0, "c", 0,  0 ]]
          }
```

Notice that "maze" is an array of arrays, with a structure corresponding to the layout of the maze. The "dimensions" key corresponds to `[nrows, ncols]`. To refer to specific coordinates, this lab assumes (row, column) indexing from the top left corner. Thus, the bomb in `maze_1` has coordinates `(1, 5)`.

Lab.py

To complete this lab, you must implement the `solve_maze` function. This function takes in a maze object (described above), along with coordinates of your start and goal positions.

Your objective is to return a maze object where each square corresponds to integer max amount of coins on valid path from `start` to `goal`. If a square does not exist on a valid path from start to goal, then the corresponding square in the output maze should be an `"X"`. If you encounter a bomb along the path, you lose coins collected so far and the starting coin count resets to 0 for all paths going from the bomb's square to goal.

As mentioned above, you are only allowed to turn down and right through the maze. By default, each square with a wall in the input should return an `"X"` in the output, since you cannot move through walls. Even though a coin might be accessible moving down or right from the starting point, *it only counts* if it is located on a valid path to the the goal square. Remember to take into account that coins and bombs on the start and goal squares are counted in the final coin tally.

In the event that no valid paths exist from the start to goal, your `solve_maze` implementation should return a maze of correct dimensions with all `"X"` 's.

We've included a `pretty_print` function to help you visualize output and debug your implementation.

The following is a concrete example:

Input:

- `maze_1`
- `start = [0, 0]`
- `goal = [4, 5]`

Output:

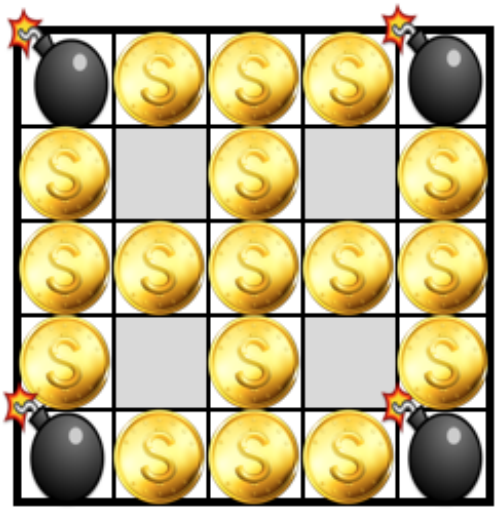
```
{
    “dimensions": [5, 6],
    “maze”:  [[ 1,  1, "X","X", "X","X"],
              ["X", 2,  3,  4,   5,  0 ],
              ["X", 2, "X","X", "X", 0 ],
              ["X", 2, "X","X", "X", 1 ],
              ["X", 3,  3,  4,   4,  4 ]]
}
```

1	1	X	X	X	X
X	2	3	4	5	0
X	2	X	X	X	0
X	2	X	X	X	1
X	3	3	4	4	4

In this case, there are exactly two valid paths from `(0,0)` to `(4,5)` , and integers

representing coin counts along the paths are found in the output. Notice that in square `(1,5)`, the count for the path gets reset to 0, since we encountered a *bomb*. Also notice that the value of coins in the goal square `(4,5)`, where the two paths merge, contains the maximum coin value along the two paths. Below is a slightly more complicated example:

Input:

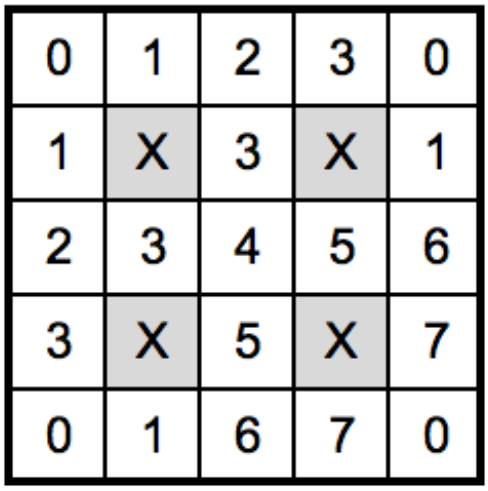


```
maze_2 = {
        "dimensions": [5, 5],
        "maze": [["b","c","c","c","b"],
                 ["c", 1, "c", 1, "c"],
                 ["c","c","c","c","c"],
                 ["c", 1, "c", 1, "c"],
                 ["b","c","c","c","b"]]
            }
```

- `maze_2`
- `start = [0, 0]`
- `goal = [4, 4]`

Output

```
{
    "dimensions": [5, 5],
    "maze": [[0, 1, 2, 3, 0],
             [1,"X",3,"X",1],
             [2, 3, 4, 5, 6],
             [3,"X",5,"X",7],
             [0, 1, 6, 7, 0]]
}
```



We have a few interesting things happening in this example. First, we have bombs on both the start and goal squares. Ensure that your implementation correctly returns 0 on these squares. Next, we have certain squares with multiple valid paths passing through them. For instance, `(3,5)` has 3 valid paths passing through while `(4,4)` has 6. In the output, the value of these squares is the maximum value of coins collected along a path to the squares, or 0 in the case of a bomb. Finally, notice that the squares corresponding to walls in the input have `"X"` in the output, since no valid path exists through these squares to goal.

In-Browser UI (./server.py)

Once your `solve_maze` outputs a valid result, you can visualize the output to help you debug your code. Run `server.py` and open your browser to localhost:8000. You will be able to select the test case and visualize the maze and your paths.

Auto-grader (./test.py)

As in the previous labs, we provide you with a `test.py` script to help you **verify** the correctness of your code. The script will call `solve_maze` from `lab.py` with the test cases in the `cases` folder and verify their output.

You will find that `test.py` is not very useful for debugging. We encourage you to use the UI and to write your own test cases to help you diagnose any problems and further verify correctness. (You may need to clear your browser cache if you create your own test cases and are using the UI to debug.)

We wish you an a-maze-ing time on this lab! Good luck. Start early!

Does your lab work? Do all tests in `test.py` pass? You're done! Submit your `lab.py` online,

and get your lab checked off by a friendly staff member. Consider tackling the bonus section below.

Bonus

NOTE: This component is not a required portion of the lab.

Now that you have enumerated all of the paths, now we want to find best one! This corresponds to the optimal navigation route through a city via GPS. In this case, it's a path that results in the highest possible number of coins collected from `start` to `goal`. In case of ties, return any such path. Write a function `find_max_path` that takes in your output maze, `start` coordinate, and `goal` coordinate and gives the maximum weight path in tuples of coordinates from `start` to `goal`. This function should output `None` if no path exists.

For example, back to `maze_1`:

Input:

```
output_1 = {
    “dimensions": [5, 6],
    “maze”:  [[ 1,  1, "X","X", "X","X"],
              ["X", 2,  3,  4,   5,  0 ],
              ["X", 2, "X","X", "X", 0 ],
              ["X", 2, "X","X", "X", 1 ],
              ["X", 3,  3,  4,   4,  4 ]]
            }
```

1	1	X	X	X	X
X	2	3	4	5	0
X	2	X	X	X	0
X	2	X	X	X	1
X	3	3	4	4	4

- `output_1`
- `start = [0, 0]`
- `goal = [4, 5]`

Output:

```
[(0, 0), (0, 1), (1, 1), (2, 1), (3, 1), (4, 1), (4, 2), (4, 3), (4, 4), (4, 5)]
```