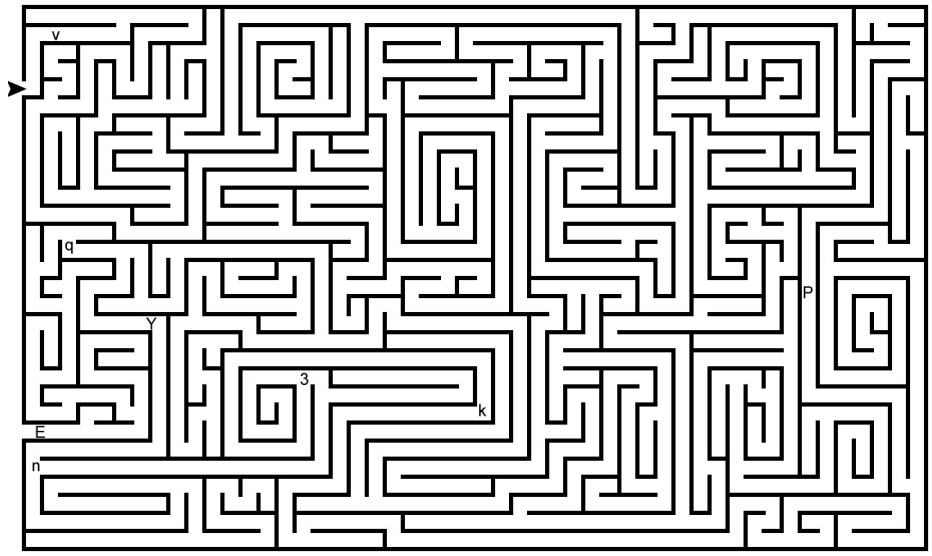
CS 15 Lab 4: Recursion



Introduction

Mazes have probably been around since before the Egyptians created gigantic labyrinths circa 2000BCE. A 'perfect' maze is a maze with only one path from a start point to an end point. As it turns out, it is relatively easy for computers to create and solve perfect mazes. In this lab, we will investigate solving mazes using recursion. The code you will be given can already create perfect mazes; today, you will write the code to solve them.

Depth First Search

There are numerous algorithms for solving mazes; today, we will explore **depth-first search** (`DFS`), one of the canonical search algorithms. With `DFS`, we follow a path as far as we can. Along the way, when we reach a fork in the road, we pick one way arbitrarily. We might choose to always go left, or to always go right, or even to choose randomly. Regardless, we continue until we either hit a dead-end or we reach the goal. If we're at a dead end, then we backtrack to the last fork, and follow a new (unexplored) route to its conclusion.

Human-Facing Maze Representation

The mazes that the starter code creates look like the diagram below on the left, and solved mazes will look like the diagram on the right; the `S` is where the maze starts, and the `F` is where the maze finishes. (We put in color here to help you see the start and finish, but your program will just deal with plain text.) Walls are marked with `X`s, and available paths are marked by blank areas. When solving a maze, `.` characters trace the solved path through the maze, and lowercase b characters show where the solving algorithm had to backtrack during the solving phase (it is actually easier to see the results on a computer screen). Look at the figure below and trace the periods to check the solution.

```
auto-generated maze                result of exploration
-------------------                ---------------------

 XXXXXXXXXXXXXXXXX                  XXXXXXXXXXXXXXXXX
 XSX   X          X                  XSX...Xbbbbbbbbbx
 X X X XXXXXXX X X                  X.X.X.XXXXXXXbXbX
 X   X X     X X X                  X...X.X.....XbXbX
 XXXXX X XXX XXX X                  XXXXX.X.XXX.XXXbX
 X   X   X X   X X                  X   X...XbX...XbX
 X XXXXXXX XXX X X                  X XXXXXXXbXXX.XbX
 X X   X     X   X                  X X...X.....X...X
 X X X X XXX XXX X                  X X.X.X.XXX.XXX.X
 X X X X X X     X                  X X.X.X.X X.....X
 X X X X X XXXXXXX                  X X.X.X.X XXXXXXX
 X X X   X X     X                  X X.X...X X     X
 X X XXXXX X X XXX                  X X.XXXXX X X XXX
 X X   X     X   X                  X X...X     X   X
 X XXX XXXXXXXXX X                  X XXX.XXXXXXXXX X
 X              FX                  X    ..........FX
 XXXXXXXXXXXXXXXXX                  XXXXXXXXXXXXXXXXX
```

Machine-Facing Maze Representation

In memory, the maze is represented by a 2-dimensional array of characters laid out in rows and columns, where `maze[row][col]` denotes the character row lines down and col positions over from the left. `maze[0][0]` is the top left corner. (As with all arrays in `C++`, indices are zero-based). Our mazes will all be square (equal number of columns and rows). For the mazes shown above, the start (`S`) is at position `maze[1][1]`, and the finish (`F`) is at position `maze[maze_size - 2][maze_size - 2]`.

The Recursive Algorithm

The general recursive algorithm for solving the maze is straightforward: we need to "solve the maze" in terms of 'solving the maze." We will, therefore, try to solve the maze by going in all four possible directions from the start position. In case of multiple possible directions, the reference implementation follows the order: North, then East, then South, then West. Although this choice is technically arbitrary, **you must try the directions in the same order to match our solution:** `N`, `E`, `S`, `W`. Thus, at every point we will return the result of continuing North, East, South, then West. And, if we find ourselves at the finish line, then we have solved the maze! It might seem counter-intuitive, but that's really all there is to it.

Example Traversal

Here is an example maze. [1] Let's traverse it!

```
  012345678
0XXXXXXXXX
1XS  X   X
2XXX X X X
3X X X X X
4X X XXX X
5X X     X
6X XXXXX X
7X      FX
8XXXXXXXXX
```

A journey of 1000 miles begins with a single step. So which way to go? Remember, we always go in order of **North, East, South, West**. In our

[1] Note: to aid the discussion, we have attached row and column indices in the figures here. These are not part of the array that holds the maze.

case, North is blocked by a wall (`X`). East is free, so we go that way! As we continue forward, we mark where we came from with a breadcrumb (`.`) so we can find our way back.

```
  012345678
0XXXXXXXXX
1XS. X   X
2XXX X X X
3X X X X X
4X X XXX X
5X X     X
6X XXXXX X
7X      FX
8XXXXXXXXX
```

Keep in mind that `DFS` is not 'smart'! It will eventually finish the maze, but it might take a big detour even when it is close to the finish. For instance, after continuing with our `DFS` traversal, we will eventually wind up here.

```
  012345678
0XXXXXXXXX
1XS..X   X
2XXX.X X X
3X X.X X X
4X X.XXX X
5X X.....X
6X XXXXX X
7X      FX
8XXXXXXXXX
```

At this position, we humans can clearly see that South is the correct way to go, but based on our chosen strategy, the algorithm will first pursue the Northern way. Thus, it will continue, until reaching:

```
  012345678
0XXXXXXXXX
1XS..X...X
2XXX.X.X.X
3X X.X.X.X
4X X.XXX.X
5X X.....X
6X XXXXX X
7X      FX
8XXXXXXXXX
```

Note that the path went up, left, and down. At this point, we find walls on all sides (except where we came from), so we *backtrack*, and leave a (`b`) symbol to indicate that we backtracked from this position. When the

algorithm backtracks, it continues from the last fork; if it tried going North but had to backtrack, the next move will be to go East. If it tried North and East without success, then it will try South. In our example, eventually we wind up back at this position.

```
  012345678
 0XXXXXXXXX
 1XS..XbbbX
 2XXX.XbXbX
 3X X.XbXbX
 4X X.XXXbX
 5X X.....X
 6X XXXXX X
 7X       FX
 8XXXXXXXXX
```

Now we can go South!

```
  012345678
 0XXXXXXXXX
 1XS..XbbbX
 2XXX.XbXbX
 3X X.XbXbX
 4X X.XXXbX
 5X X.....X
 6X XXXXX.X
 7X       FX
 8XXXXXXXXX
```

Below is a log of all of the steps that were produced by the solution (line numbers added by us). You may find similar debug output to be very useful! Note that it takes 71 steps to solve this tiny maze! See if you can follow at least some of the steps.

```
1:   row 1 and col 1, Marking with period (.)
2:   Trying north, row 0 and col 1, Hit wall, Back at row 1 and col 1
3:   Trying east, row 1 and col 2, Marking with period (.)
4:   Trying north, row 0 and col 2, Hit wall, Back at row 1 and col 2,
5:   Trying east, row 1 and col 3, Marking with period (.)
6:   Trying north, row 0 and col 3, Hit wall, Back at row 1 and col 3,
7:   Trying east, row 1 and col 4, Hit wall, Back at row 1 and col 3,
8:   Trying south, row 2 and col 3, Marking with period (.)
9:   Trying north, row 1 and col 3, We came from here, Back at row 2 and col 3,
10:  Trying east, row 2 and col 4, Hit wall, Back at row 2 and col 3,
11:  Trying south, row 3 and col 3, Marking with period (.)
12:  Trying north, row 2 and col 3, We came from here, Back at row 3 and col 3,
13:  Trying east, row 3 and col 4, Hit wall, Back at row 3 and col 3,
14:  Trying south, row 4 and col 3, Marking with period (.)
15:  Trying north, row 3 and col 3, We came from here, Back at row 4 and col 3,
16:  Trying east, row 4 and col 4, Hit wall, Back at row 4 and col 3,
17:  Trying south, row 5 and col 3, Marking with period (.)
18:  Trying north, row 4 and col 3, We came from here, Back at row 5 and col 3,
19:  Trying east, row 5 and col 4, Marking with period (.)
```

```
20:  Trying north, row 4 and col 4, Hit wall, Back at row 5 and col 4,
21:  Trying east, row 5 and col 5, Marking with period (.)
22:  Trying north, row 4 and col 5, Hit wall, Back at row 5 and col 5,
23:  Trying east, row 5 and col 6, Marking with period (.)
24:  Trying north, row 4 and col 6, Hit wall, Back at row 5 and col 6,
25:  Trying east, row 5 and col 7, Marking with period (.)
26:  Trying north, row 4 and col 7, Marking with period (.)
27:  Trying north, row 3 and col 7, Marking with period (.)
28:  Trying north, row 2 and col 7, Marking with period (.)
29:  Trying north, row 1 and col 7, Marking with period (.)
30:  Trying north, row 0 and col 7, Hit wall, Back at row 1 and col 7,
31:  Trying east, row 1 and col 8, Hit wall, Back at row 1 and col 7,
32:  Trying south, row 2 and col 7, We came from here, Back at row 1 and col 7,
33:  Trying west, row 1 and col 6, Marking with period (.)
34:  Trying north, row 0 and col 6, Hit wall, Back at row 1 and col 6,
35:  Trying east, row 1 and col 7, We came from here, Back at row 1 and col 6,
36:  Trying south, row 2 and col 6, Hit wall, Back at row 1 and col 6,
37:  Trying west, row 1 and col 5, Marking with period (.)
38:  Trying north, row 0 and col 5, Hit wall, Back at row 1 and col 5,
39:  Trying east, row 1 and col 6, We came from here, Back at row 1 and col 5,
40:  Trying south, row 2 and col 5, Marking with period (.)
41:  Trying north, row 1 and col 5, We came from here, Back at row 2 and col 5,
42:  Trying east, row 2 and col 6, Hit wall, Back at row 2 and col 5,
43:  Trying south, row 3 and col 5, Marking with period (.)
44:  Trying north, row 2 and col 5, We came from here, Back at row 3 and col 5,
45:  Trying east, row 3 and col 6, Hit wall, Back at row 3 and col 5,
46:  Trying south, row 4 and col 5, Hit wall, Back at row 3 and col 5,
47:  Trying west, row 3 and col 4, Hit wall, Back at row 3 and col 5,
48:  Failed. Marking bad path with b. Back at row 2 and col 5,
49:  Trying west, row 2 and col 4, Hit wall, Back at row 2 and col 5,
50:  Failed. Marking bad path with b. Back at row 1 and col 5,
51:  Trying west, row 1 and col 4, Hit wall, Back at row 1 and col 5,
52:  Failed. Marking bad path with b. Back at row 1 and col 6,
53:  Failed. Marking bad path with b. Back at row 1 and col 7,
54:  Failed. Marking bad path with b. Back at row 2 and col 7,
55:  Trying east, row 2 and col 8, Hit wall, Back at row 2 and col 7,
56:  Trying south, row 3 and col 7, We came from here, Back at row 2 and col 7,
57:  Trying west, row 2 and col 6, Hit wall, Back at row 2 and col 7,
58:  Failed. Marking bad path with b. Back at row 3 and col 7,
59:  Trying east, row 3 and col 8, Hit wall, Back at row 3 and col 7,
60:  Trying south, row 4 and col 7, We came from here, Back at row 3 and col 7,
61:  Trying west, row 3 and col 6, Hit wall, Back at row 3 and col 7,
62:  Failed. Marking bad path with b. Back at row 4 and col 7,
63:  Trying east, row 4 and col 8, Hit wall, Back at row 4 and col 7,
64:  Trying south, row 5 and col 7, We came from here, Back at row 4 and col 7,
65:  Trying west, row 4 and col 6, Hit wall, Back at row 4 and col 7,
66:  Failed. Marking bad path with b. Back at row 5 and col 7,
67:  Trying east, row 5 and col 8, Hit wall, Back at row 5 and col 7,
68:  Trying south, row 6 and col 7, Marking with period (.)
69:  Trying north, row 5 and col 7, We came from here, Back at row 6 and col 7,
70:  Trying east, row 6 and col 8, Hit wall, Back at row 6 and col 7,
71:  Trying south, row 7 and col 7, Found the Finish
```

Getting Started

Create a directory for this lab and move into it. Copy the files from the usual place. Use the provided `Makefile`, and do not edit it. Remember: write a little, compile/link, test, repeat.

Task 1: Solving the Maze

solv function

Complete the implementation of the `solve` function. Specifically, implement either `bool solv(int row, int col)` or the `bool solv(Position p)` in the file `maze.cpp`. They both do the same thing, but the first one has you manipulate the row and column indices directly, and the second uses a `Position` object. The `Position` object encodes a row and a column in one `struct`, and also has constructors and helpful functions for getting the position to the North, East, South, West of a given position. If you look in `position.h`, you'll notice that there are also some useful functions on positions. For example, if `p` holds a `Position` object, then `p.north()` returns a `Position` object with row and column indices of the cell just to the north of `p`. However, you can use either strategy. Pick the one that you're most comfortable with. There is no need to do both!

N, E, S, W

Remember to solve the maze by going in all four possible directions from each position, starting at the start position (1, 1). That is, we'll hunt for solutions starting at (0, 1) (to the North), then at (1, 2) (to the East), then at (2, 1) (to the South), then at (1, 0) (to the West). Remember to use the same direction order (N, E, S, W)! It will help with grading and debugging. That's really all there is to it: if any of the recursive paths we chose return true, we have solved the maze! If none work out, then there is no solution. Sad.

Tips on Recursion

As with any recursive solution, we must go through a series of ordered steps which help us visualize any problem, no matter the complexity.

Base Cases

- If a cell would be outside the maze (the indices are not in the bounds of the matrix), then we know there can't be a solution starting from the given position. What should we return?
- If a cell is blocked, then we also know there can't be a solution starting from the given position. How could a cell be blocked?
  - There's a wall (`X`) there.
  - We've been here before, i. e., we see a bread crumb (`.`), or a previously failed path (`b`).
- If we have reached the goal (`F`). In this case, we have succeeded in finding a solution!

Change Something, Adjust Yourself, Recurse

Don't forget to change things along the way!

- Remember to mark the current location with a `.` *before* recursing through the directions. This will help avoid investigating loops, and because this might be a solution path.
- Remember to mark the current position with a `b` *after* recursing through all the directions. This way we indicate to subsequent calls that this step is not on the solution path.
- Now get moving!

Task 2: Finding the path length, first method

Introduction

Suppose we found a path. We're going to find the length of the path in two ways. While we could find the length by following the bread crumb (`.`) trail, we can also notice that, if we've solved the maze, we could just count the dots! The length of the path is the number of dots plus one (convince yourself that's true!).

Functions to Implement

For this task, you will implement two functions:

1. `int num_dots_by_rows(int row)`
2. `int num_dots_in_row(int row, int col)`

For the finding the length of the path, we want you to focus on understanding what is happening on each recursive call and practice doing calculations with recursion. If you have a 2-dimensional array, and you want to traverse it recursively, probably the easiest way is to write a function `num_dots_by_rows` that recurses over the rows (or just the row index if the array is a data member). This function adds up the number of dots in all the rows, but calls another function to get the number of dots in a particular row. This second function `num_dots_in_row` recurses over the column index and returns the number of dots in a row.

Task 3: Solving and Measuring Together

The first two tasks will likely take most of all of the time in lab. If you have time, and would like further practice, you may complete the implementation of `solve_and_count`, which solves the maze and returns the number of steps it takes all at once. The function should `return NOT_FOUND;` if it cannot find a path from start to finish. The behavior is essentially the same as for the solve function above, but rather than return a boolean value, it returns how far you are from the finish. Consider these cases:

- You did not find a solution
- You are at the solution
- You are not at the solution, but you found a solution from a neighboring cell of length `L`

Other Details

When you run the test program, it does the following:

1. prompts for a maze size (make sure to test with small and large numbers).

2. prints out an unsolved maze and then the solved maze using your recursive method.
3. prints out the length of the path using `path_length`
4. builds and prints a new maze
5. solves it using your `solve_and_count`
6. prints out the path length and the final solved maze from the previous step

We strongly encourage you to solve the first two tasks. The third one is for extra practice and understanding of recursion. And recursion, as many experienced engineers believe, is the future.

Providing

There is a `provide` target in the `Makefile`

Acknowledgements

Recursive maze problems are widely used in CS. The are particularly common in settings with physical or virtual robots. The overall idea for this lab was borrowed from `https://www.cs.bu.edu/teaching/alg/maze/`. The lab was adapted by Chris Gregg, and then modified by Eliza Schreibman and Julia Cooper and Mark Sheldon. The maze creation code was taken from here: `https://azerdark.wordpress.com/2009/03/29/588/`.