Depth First Search

This assignment uses *Depth first search* to find the solution to a maze.

There is a *lot* of background reading in this README, covering enumerations, 2D arrays, and depth first search. Please read the README fully to make sure you understand the concepts used in this homework.

Learning Goals

You will learn:

- Enumerations and complex data type
- 2D arrays
- More recursion
- Depth first search

Background

Enumerations

An enumeration data type specifies a set of constant (integer) values that are all distinct. You can either set the constant values yourself:

```
typedef enum Month {
  JANUARY = 1,
  FEBRUARY = 2,
  MARCH = 3,
  APRIL = 4,
  MAY = 5,
  JUNE = 6,
  JULY = 7,
  AUGUST = 8,
  SEPTEMBER = 9,
  OCTOBER = 10,
  NOVEMBER = 11,
  DECEMBER = 12
} Month;
```

Or you can let the compiler choose the values for you:

```
typedef enum Day {
  SUNDAY,
  MONDAY,
  TUESAY,
  WEDNESDAY,
  THURSDAY,
  FRIDAY,
  SATURDAY
} Day;
```

(Note that by convention we make the constants in the enumeration ALL CAPS)

You can then use these values as constants throughout your program. If you define a variable as having an enumeration type, you're saying that you want its values to be restricted to those constants:

`Month birthmonth = AUGUST;`

Enumerations are a convenient way of defining a set of constants that are related to each other. This will let us use specific constants throughout our program without having to change all of them if we decide to change the values of the constants.

In this assignment, we are using two enumerations. The first defines the "types" of maze squares: a wall, an empty space, the starting point, and the ending point:

```
typedef enum SquareType {
    WALL = '#',
    SPACE = '.',
    START = 's',
    END = 'e'
} SquareType;
```

(Note that we're taking advantage of the fact that C treats characters as integers of the appropriate ASCII value).

Throughout your code, when you want to see whether a particular value is a wall, you should test if it is equal to `WALL`, not `#`. That way, if we later want to change what walls look like in our maze, it will be easy.

The second enumeration defines the possible directions you can move in your path:

```
typedef enum PathType {
    NORTH = 'n',
    SOUTH = 's',
    EAST = 'e',
    WEST = 'w'
} PathType;
```

Complex Structures

Up until now, we have mostly dealt with structures whose fields are simple data types or arrays. But we can also have the fields of a structure be *another type* like a structure or an enumeration. Consider the types we will use to define our maze (in `maze.h`):

```
typedef struct MazeSquare {
    SquareType type;
    bool visited;
} MazeSquare;
```

This is a structure that represents a single square in the maze. It has a particular type (using the enumeration type from above), and a flag that lets us know whether we've visited this square before or not (useful for checking if your path has loops).

```
typedef struct MazePos {
    int xpos;
    int ypos;
} MazePos;
```

This is a simple structure that captures the location of a particular square in a maze.

```
typedef struct Maze {
    MazeSquare * * maze; //2D array holding maze
    int width; //Number of columns in maze
    int height; //Number of rows in maze
    MazePos start; //Location of 's'
    MazePos end; //Location of 'e'
} Maze;
```

This structure represents an entire maze. It has a width and height, it uses the `MazePos` structure to capture the starting position in the maze and the ending position in the maze, and it uses a *2D Array* of `MazeSquare`s to represent the maze grid itself.

2D Arrays

What are two-dimensional arrays? If you know how many rows and columns you want your array to have, you can define a 2D array easily:

```
float matrix[10][20]; //a 10x20 matrix
```

But what if you don't know how many rows and columns you need? We'll have to use *dynamic memory allocation*, just like we did when we needed arrays of unknown size.

To allocate a 2D array, we will build an *array of arrays*. We will have an array (representing the rows of the 2D array) where each element of the array is *another array*. Allocating 2D arrays is tricky, since we don't know how many rows and columns we need. We have to do it in two steps.

First, we allocate the array of rows. Because each entry needs to be an array of floats itself, the type of this array is a *pointer to a pointer to a float*:

```
float * * matrix = malloc(nrows * sizeof(float *));
```

This creates an array of `nrows` pointers to floats. Second, for *each of those pointers* we allocate an array of floats:

```
for (int i = 0; i < nrows; i++) {
  matrix[i] = malloc(ncols * sizeof(float));
}
```

We can now use `matrix` as a normal 2D array. To access the 2nd row and 5th column, we can write `matrix[1][4]`.

Freeing a 2D Array

To deallocate a 2D array, we reverse the steps. First, we free each of the rows, then we free the array that points to all the rows.

```
for (int i = 0; i < nrows; i++) {
  free(matrix[i]);
}
free(matrix);
```

Note that we can't do this in the other order! If we `free(matrix)` first, we will have no way of getting the addresses of the row arrays we need to free.

2D Array Coordinates vs. Cartesian Coordinates

Note that 2D arrays (and matrices) work very differently than Cartesian coordinates. Suppose you have a 2D grid of letters:

```
a b c d
e f g h
i j k l
```

Then your 2D grid has 4 columns (width = 4) and 3 rows (height = 3) You should thus create a 2D array with 3 rows and 4 columns (note that with arrays we list the rows first, then the columns).

The coordinate system we use for 2D arrays (and matrices) is different than Cartesian coordinates.

In Cartesian coordinates, `(0, 0)` is at the bottom left of the grid (the letter 'i'). In 2D arrays (and matrices), `[0][0]` is at the top left of the grid (the letter 'a').

In Cartesian coordinates, we list the x position first and then the y position. For 2D arrays (and matrices), we list the y position (the row) first and then the x position (the column).

In Cartesian coordinates, incrementing the "y" position moves *up*. In 2D arrays (and matrices), incrementing the row number moves *down* (the first row is row 0, the second row is row 1, etc.).

So in Cartesian coordinates, `(2, 3)` would represent the letter 'b', but in 2D arrays, `[2][3]` would represent the letter 'l'.

Depth first search

Depth first search is one of the most common search strategies and it is one that lends itself to recursion. Suppose you are standing in the middle of a maze (sort of like you are in this assignment!) and want to figure out where the exit is. How might you find your way out?

Here is one strategy: start walking through the maze. Each time you arrive at a choice of directions, pick one and keep walking. If you reach a dead end, or get to a part of the maze you've seen before, back up to the last place you made a choice and make a different one. Eventually, you will either find the exit of the maze or explore the entire maze and decide that there is no way out.

Consider trying to find your way out of this maze:

```
.#.#
...#
e#..
#s.#
```

Assume that at each square, we try to go north, then south, then east, then west.

We start at `s`, and try to go N. We hit a wall, so we back up and try to go S. We go out of bounds, so we try to go E, which succeeds. We're now in this situation, where the `*` shows where we are:

```
.#.#
...#
e#..
#s*#
```

We try to go N, which succeeds. We try to go N again, which succeeds. We try to go N again, which succeeds. We're now in this situation:

```
.#*#
...#
e#..
#s.#
```

We try to go N again, which fails (out of bounds). We try to go S, which fails (we've already been to that square!), E, which fails (we hit a wall), then W, which fails (we hit a wall). Since we don't have anywhere to go from here, we back up to the last place we made a choice:

```
.#.#
..*#
e#..
#s.#
```

Now we try to go N (fails: already been to that square); S (fails: already been to that square); E (fails: wall); then W (succeeds):

```
.#.#
.*.#
e#..
#s.#
```

Now we try to go N (fails: wall); S (fails: wall); E (fails: already been there); and W (succeeds):

```
.#.#
*..#
e#..
#s.#
```

We then go N, which succeeds:

```
*#.#
...#
e#..
#s.#
```

From here, there is no successful move to make, so we back up to the last place we made a choice:

```
.#.#
*..#
e#..
#s.#
```

and try the next choice, S, which gets us to the exit!

The reason we call this a depth-first search is that we try each possible path until we can't keep moving forward, then we back up one choice and try again, and so forth. Other searches may not explore a single path all the way to the end before trying a different path.

Depth First Search with Recursion

Depth first search can be readily solved with recursion. Each recursive call "visits" a single square in the maze and explores each of the possible paths leading out from that square. The goal of a sequence of recursive calls is to explore a specific path through the maze, and each time you call the recursive function, you're making the path one step longer and visiting one more square of the maze.

You should interpret your recursive function as implementing the following logic: "Given all the squares I have visited so far, can I exit the maze by adding this square to the solution path?"" If the answer is no, that means the current square is not part of the solution path, and the recursive method should return false. If the answer is yes, that means the current square *is* part of the solution path, and you should add the square to the current path and return true.

The *base case* for a square is that the square represents the *end* of a path:

1. It has been visited already -- the path can't be extended this way, so return false
2. It is a wall -- this path fails, so return false.
3. It is out of bounds -- this path fails, so return false.
4. It is the exit -- this path succeeds, so return true and add this square as the last square in the path

The *recursive* case for a square is that it doesn't fall into one of the above four categories: it is an empty square that you haven't visited before. In that case, the recursive case tries to make the path one step longer by visiting one more square in the maze: recursively call your search method *four times*, once for each of the four directions you could possibly move.

So what do you do with the return value of the four recursive calls you make?

1. If one of them returns true, that means the direction that call explores is on the solution path, which also means the *current* square is on the solution path. Add the current square to the solution path, and return true.
2. If the recursive call returns false, that means that direction doesn't work, so move on to the next recursive call.
3. If *all* the recursive calls return false, that means the *current* square can't be on the solution path, so return false.

(Don't forget to mark the current square as visited when the recursive method returns!)

Note that this recursive function follows the usual pattern: it calls itself and *assumes those calls do the right thing*.

What do you need to do?

In this assignment, you only need to write *one method*, `depthFirstSolve` in `solver.c`. `depthFirstSolve` is called by `solveMaze`, as you can see in the code, and is meant to be the recursive method that implements the logic described above. It takes four arguments:

1. `Maze * m`: the maze you are trying to solve. `Maze` is declared in `maze.h`. Functions to read in the maze (`readMaze`) and free the maze (`freeMaze`) are defined in `maze.c`. Pay careful attention to the comments for the `readMaze` declaration in `maze.h`; they explain in detail what the
2. `MazePos curPos`: the current square in the maze being "visited."

3. `char * path`: a character array containing the current path. The path is a *null-terminated* string using `n`, `s`, `e`, and `w` as directions to move. So, for example, the string `"wwnes"` represents a string that moves west twice, then north, then east, then south. We recommend `PathType` enum in `path.h` to represent the individual characters of the string.
4. `int step`: a counter telling you how far along the current path you are.

and returns a boolean: `true` means that the current square being visited by `depthFirstSolve` *is* part of the solution, and `false` means that current square *is not* part of the solution.

One tricky bit here is figuring out how to update `path` correctly. Two hints: (i) `step` tells you how far along in the current path you are; and (ii) if you successfully exit the maze on the `n`th step, `path[n]` should be `\0`.

> It is important that you get the format of the path correct so it is written out properly. Because the search order means that different people may find different solution paths, the way we will grade your code is by generating a solution path, then using the `checkPath` function to check that it is a valid solution to the maze.

Warning: Don't be misled by the fact that you only have to write one method. This assignment will probably have the highest ratio of *time spent thinking about what code to write* to *amount of code written* of any assignment so far. Recursive code is often quite short -- the instructors' version of `depthFirstSolve` has about 25 lines of code -- but takes a while to get right. Start thinking about your solution now, and don't be afraid to ask questions on Piazza or in office hours to make sure your logic is right.

Testing your code

Depending on the order you choose to do your recursion (do you decide to look North first, or South first, etc) you can get a different path than someone else if a maze has many paths to the exit! How can you test if your path is correct? You can use the same path checker that we will use to verify if the path you create in HW9 is valid. To do this, you read in the maze and the proposed solution path from input files, and call `checkPath` to see if it is right. There is commented-out code in `hw9.c` that shows how to do this.

We have provided several mazes for you. `maze1`, `maze3` and `maze4` have valid solutions, while `maze2` and `maze5` do not.

What do you need to turn in?

Turn in your version of `solver.c`, with your implementation of `depthFirstSolve`. You do not need to modify any other files.