Com S 227
Fall 2017
Miniassignment 2
25 bonus points

Due Date: Tuesday, November 7, 11:59 pm (midnight)

NO LATE SUBMISSIONS

General information

This assignment is to be done on your own. See the Academic Dishonesty policy in the syllabus, http://www.cs.iastate.edu/~cs227/syllabus.html#ad , for details.

You will not be able to submit your work unless you have completed the *Academic Dishonesty policy acknowledgement* on the Homework page on Blackboard. Please do this right away.

If you need help, see your instructor or one of the TAs. Lots of help is also available through the Piazza discussions.

Note: This is a miniassignment and the grading is automated. If you do not submit it correctly, you will receive at most half credit.

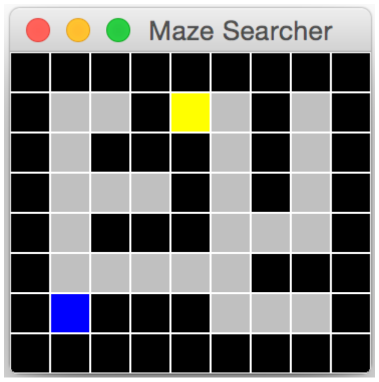
Bonus points?

The 25 points for this miniassignment are not part of the total homework points. Whatever points you get will just be added to your homework score as a bonus. It is a short and fun problem and doing it will help you understand recursion. ***However, please do not spend a lot of time on this at the expense of studying for Exam 2, which will count much more toward your grade!***

Overview

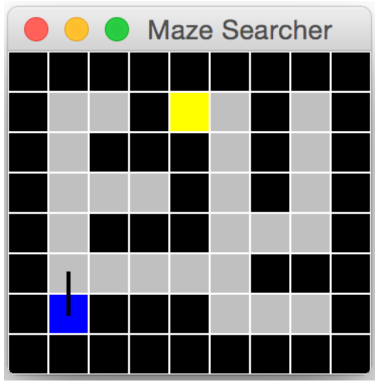
This is a short assignment intended to give you some practice using recursion before Exam 2. You will implement one recursive method called `search` in the class `Searcher`. The sample code includes a skeleton for the **`Searcher`** class that you can use. You'll want to be sure you have done lab 9 first.

The **`search`** method will recursively search for a path in a 2D maze. We can picture a 2D maze like this, where the blue cell is the starting point, the yellow cell is the goal, and the black cells are boundaries.

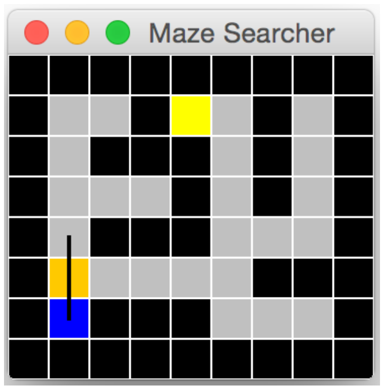


The challenge with searching a maze is keeping track of where you've been, and being able to *backtrack* and resume searching in a different direction from a previous point. It turns out that recursion is ideal for this.

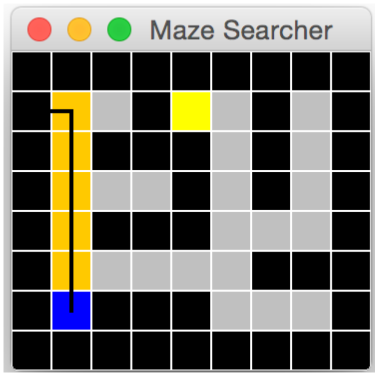
As an example, suppose you are initially standing in the blue square. You first lay down a stick to show you've started searching the cell. You plan to go to the next cell up, so you put the stick pointing up.



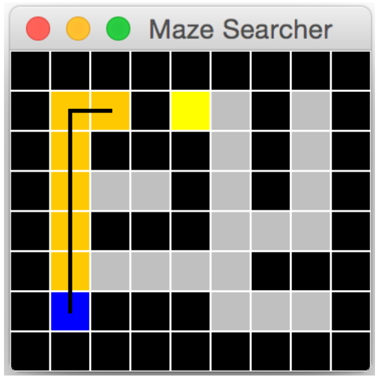
Next you step into the cell above. Again, lay down a stick, and point it upwards to show which direction you're going next.



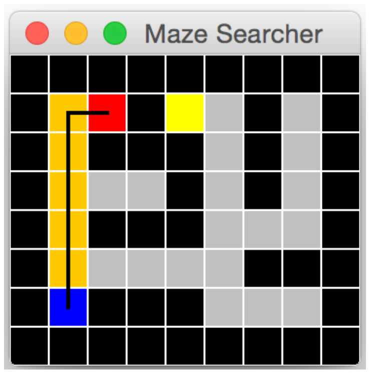
Well, keep on doing that. When you get up to the top (row 1, column 1) you can't go up, so you try other neighboring cells. If you look at the cell below, you see the stick, so you know you've already been there), so try going left:



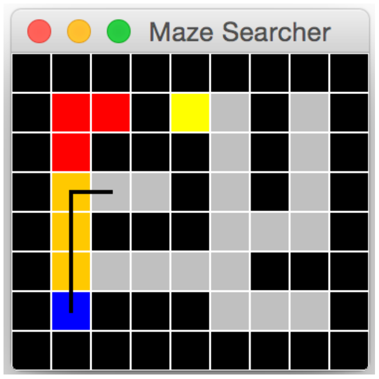
Well, that's just a wall. Try going right:



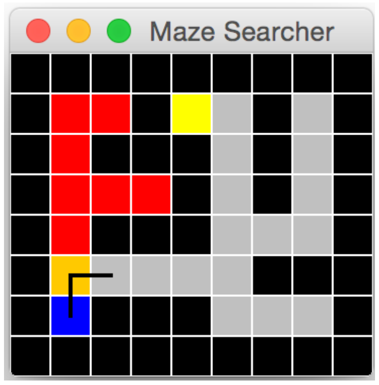
Ok, this is an unexplored cell. From here, you check up (wall), down (wall), left (already explored), and right (wall), and you're stuck. So you *backtrack* to the cell where you just were. (As you leave the cell, put a red rock on it to mark the fact that it's a dead end.)



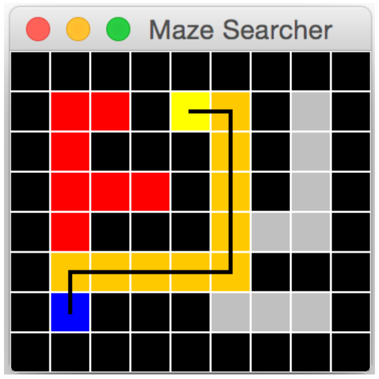
Now back at row 1, column 1, how do you know whether you've searched the other three neighbors already? You can tell since the stick is pointing right that you've already searched all its neighbors. That's because we're always checking neighbors in the order up, down, left, then right. You keep backtracking; at row 2, column 1, you see the stick pointing up, so you check down (already explored), left (wall), and right (wall). Backtrack to row 3, column 1, again check down (already explored), to the left (wall) and then to the right, discovering an unexplored cell:



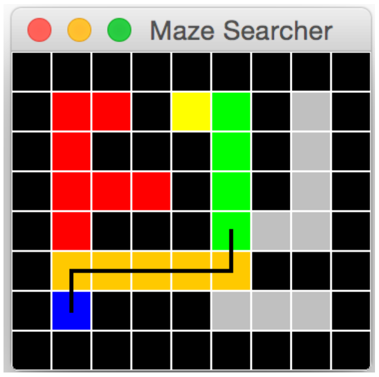
But it leads to another dead end, so you backtrack again. When you eventually get back to row 5, column 1 (i.e. just above the starting cell) you can again check down (already explored), left (wall) and right to find another unexplored cell to the right:



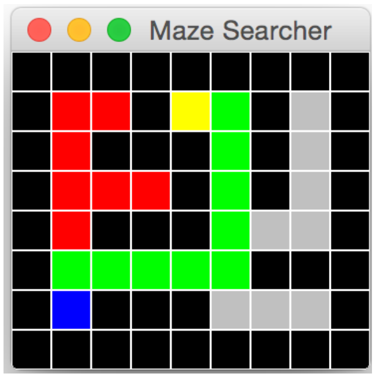
You continue searching neighboring cells, exploring up, down, left, and right (always in that order) marking with sticks as you go.



At row 1, column 4, you discover you're at the goal! Now, as you backtrack to your previous cell, put down a green rock.



This is a bit different from backtracking from a dead end: since you've already found the goal, you don't bother to search any more neighboring cells. Some parts of the maze remain unexplored when you finally backtrack all the way to the start:



This is a recursive algorithm, because basically what we're saying is:

```
to search a maze starting from cell c:
    search the maze starting at the neighboring cells of c
```

This reduces the problem to a smaller problem, provided that we avoid searching from cells we've already visited. And we have base cases: finding the goal, or a dead end. The process we carried out in pictures above could be written in pseudocode like this:

```
to search a maze starting from cell c:
    if c is the goal
        return true (success)
    else if c is not an unexplored cell (i.e. a wall, or explored already)
        return false
    else
        for each neighboring cell d (in the order up, down, left, right)
            mark cell c to record the direction of d
            recursively search the maze starting from d
            if the search returns true
                mark cell c as succeeded and return true
    if we fall through to this point, mark cell c as failed (dead end) and return false
```

Specification

The specification for this asssignment includes this pdf, any "official" clarifications posted on Piazza, and the javadoc.

The enum type api.Status

As you search, an important part of the process is to "mark" each cell with its current state. The state of a cell could be:

```
WALL
GOAL
UNEXPLORED
EXPLORING_UP
EXPLORING_DOWN
EXPLORING_LEFT
EXPLORING_RIGHT
SUCCEEDED
FAILED
```

To keep track of these possible values, we define a set of 9 constants in the type **`Status`** having the names above. Note that we *don't* define arbitrary integer constants like this:

```
public class Status {
    public static final int WALL = 0;
    public static final int GOAL = 1;

    // and so on...
  }
```

Instead, we follow conventional Java practice and define **`Status`** as an **`enum`** type. You can use enums just like integer constants, except you can't assign numeric values to them. You compare them with == and you don't construct them like objects. For example, you can write code like this:

```
if (currentCell.getStatus() == Status.UNEXPLORED)
{
  currentCell.setStatus(Status.EXPLORING_UP);
  //...
}
```

(See "Special Topic 5.4" in Section 5.5 of the text for more information on enums.) Note that cells with the WALL and GOAL status do not change.

The class api.MazeCell

A **`MazeCell`** is just a container for a status. If you look at the code, you'll notice there is also something called an "observer". This is not something you have to worry about - the idea of the observer is that when a cell's status changes, it will call the observer's **`statusChanged`** method. This can be used by an application such as the sample GUI to respond to the status change. The type of the observer is **`MazeObserver`**, which is an example of a Java **`interface`**.

The class api.Maze

The **`Maze`** class encapsulates a 2D array of **`MazeCell`** objects. It has a constructor that takes a string array that can be used to initialize a maze. For example, the maze illustrated at the beginning of this document was initialized from the string array below. (The 'S' character is the start and the '$' character is the goal. See the class **`ui.runSearcher`** for more details.)

```
public static final String[] MAZE2 = {
    "#########",
    "#  #$ # #",
    "# ### # #",
    "#   # # #",
    "# ###   #",
    "#     ###",
    "#S###   #",
    "#########",
  };
```

The UI

You can use the sample UI to try out your search method by running `ui.RunSearcher`. You can edit the `main` method to initialize with one of several predefined mazes to try. You can also adjust the speed of the animation by adjusting the value of the `sleepTime` variable. To get an idea what to expect, there is a brief movie here,

http://web.cs.iastate.edu/~cs227/homework/mini2/maze.mp4

The sample code

The sample code is a complete Eclipse project you can import. The instructions for importing are the same as for homework 3, so refer to the pdf for homework 3 if you don't remember how to do it.

The SpecChecker

Import and run the SpecChecker as for miniassignment 1. It will run a number of functional tests and then bring up a dialog offering to create a zip file to submit. Remember that error messages will appear in the console output. *Always start reading the errors at the top and make incremental corrections in the code to fix them.*

When you are happy with your results, click "Yes" at the dialog to create the zip file.

See the document "SpecChecker HOWTO", which can be found in the Piazza pinned messages under "Syllabus, office hours, useful links" if you are not sure what to do.

Documentation and style

Since this is a miniassignment, the grading is automated and in most cases we will not be reading your code. Therefore, there are no specific documentation and style requirements.

If you have questions

For questions, please see the Piazza Q & A pages and click on the folder **`miniassignment2`**. If you don't find your question answered, then create a new post with your question. Try to state the question or topic clearly in the title of your post, and attach the tag **`miniassignment2`**. *But remember, do not post any source code for the classes that are to be turned in.* It is fine to post source code for general Java examples that are not being turned in. (In the Piazza editor, use the button labeled "pre" to have Java code formatted the way you typed it.)

If you have a question that absolutely cannot be asked without showing part of your source code, make the post "private" so that only the instructors and TAs can see it. Be sure you have stated a specific question; vague requests of the form "read all my code and tell me what's wrong with it" will generally be ignored.

Of course, the instructors and TAs are always available to help you. See the Office Hours section of the syllabus to find a time that is convenient for you. We do our best to answer every question carefully, short of actually writing your code for you, but it would be unfair for the staff to fully review your assignment in detail before it is turned in.

Any posts from the instructors on Piazza that are labeled "Official Clarification" are considered to be part of the spec, and you may lose points if you ignore them. Such posts will always be placed in the Announcements section of the course page in addition to the Q&A page. (We promise that no official clarifications will be posted within 24 hours of the due date.)

Advice

Be sure you have done lab 9, especially the exercise described here,
http://web.cs.iastate.edu/~cs227/labs/lab9/page02.html
where you use the debugger to visualize what's happening on the call stack as recursive calls are made. You should try the same thing for the file lister problem later in the lab. Notice that as you return from a recursive call on a subdirectory, you can pick up right where you left off in iterating over the list.

The maze animation is really just another way of visualizing the call stack. The orange cells are the frames that are still active (method call has not yet returned) and the red and green cells are calls that have returned and are no longer on the stack.

What to turn in

Note: You will need to complete the "Academic Dishonesty policy questionnaire," found on the Homework page on Blackboard, before the submission link will be visible to you.

Please submit, on Blackboard, the zip file that is created by the SpecChecker. The file will be named **`SUBMIT_THIS_mini2.zip`**. and it will be located in the directory you selected when you ran the SpecChecker. It should contain one directory, **`mini2`**, which in turn contains one file, **`Searcher.java`**.

Submit the zip file to Blackboard using the Miniassignment 2 submission link and verify that your submission was successful by checking your submission history page. If you are not sure how to do this, see the document "Assignment Submission HOWTO" which can be found in the Piazza pinned messages under "Syllabus, office hours, useful links."

We strongly recommend that you just submit the zip file created by the specchecker. If you mess something up and we have to run your code manually, you will receive at most half the points.

> We strongly recommend that you submit the zip file as created by the specchecker. If necessary for some reason, you can create a zip file yourself. The zip file must contain the directory **mini2**, which in turn should contain the file **Searcher.java**. You can accomplish this by zipping up the **src** directory of your project. The file must be a zip file, so be sure you are using the Windows or Mac zip utility, and not a third-party installation of WinRAR, 7-zip, or Winzip.