Com S 227
Fall 2022
Miniassignment 3
60 points (out of 30 possible)

Due Date: Friday, November 18, 11:59 pm (midnight)

5% bonus for submitting 1 day early (by 11:59 pm November 17)

10% penalty for submitting 1 day late (by 11:59 pm November 19)

No submissions accepted after November 19, 11:59 pm

General information

This assignment is to be done on your own. See the Academic Dishonesty policy in the syllabus 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 Canvas. 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.

There are two problems, both of which involve a recursive search. The first one (part A) involves searching a maze, and the second one (part B) is to implement a solver for the **`LinesGame`** *from homework 3. Both are fun, and neither one should be terribly difficult if you have a good understanding of Lab 7. The first problem is 30 points and the second problem will be counted as another 30 points "extra credit".*

This is a miniassignment and the grading is automated. If you do not submit it correctly and we have to run it by hand, you will receive at most half credit.

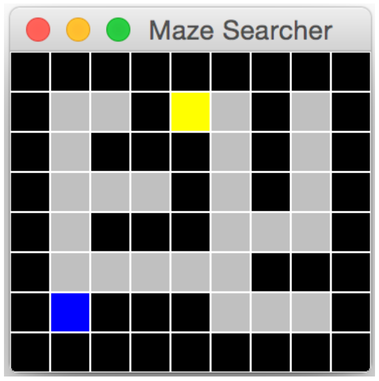
Part A - maze searcher

This is a fun problem to give you some practice with recursion. This had always been one of my favorite kinds of examples to go back to and think about when I was in my second year of college and trying to visualize how recursion works.

Your task is to implement the recursive method **`search`** in the class **`MazeSearcher`**. The sample code includes a skeleton for the **`MazeSearcher`** class that you can use. *You'll want to be sure you have done and understood lab 7 first.*

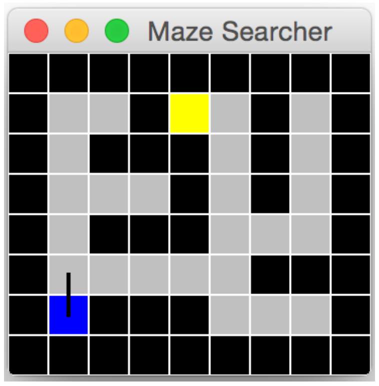
The sample code includes a lot of related stuff in the packages **`maze_api`** and **`maze_ui`**, as well all the unrelated stuff for part B in packages **`api`**, **`ui`**, and **`hw3`**. *Do not modify any of that code*, just implement the **`search`** method.

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 princess we have to rescue (the "goal"), and the black cells are boundaries ("walls").

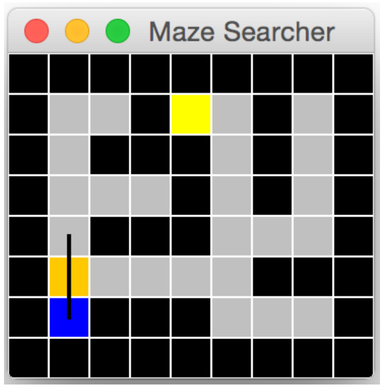


The challenge with searching a maze (or anything that is non-linear, such as a file hierarchy) is keeping track of where you've been and being able to *backtrack* and resume searching in a different direction from a previous point. It turns out that recursion is ideal for this. What does it mean to "backtrack"? Just return from the method call you're in, and resume execution where you left off!

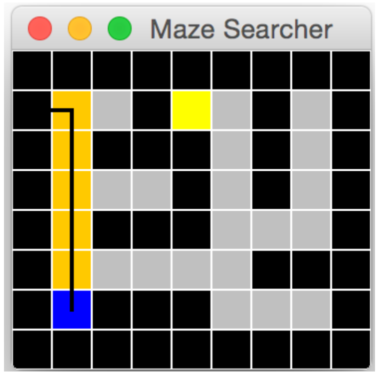
As an example, suppose you are initially standing in the blue square. You first lay down a stick to show you've started searching the cell. You plan to go to the next cell up, so you put the stick pointing up:



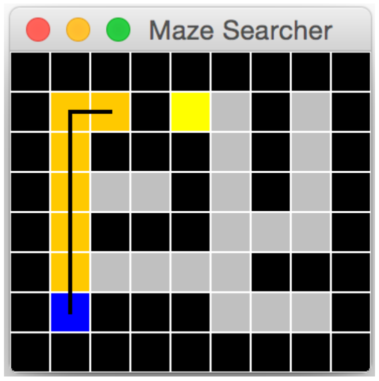
Next you step into the cell above. Again, lay down a stick, and point it upwards to show which direction you're going next:



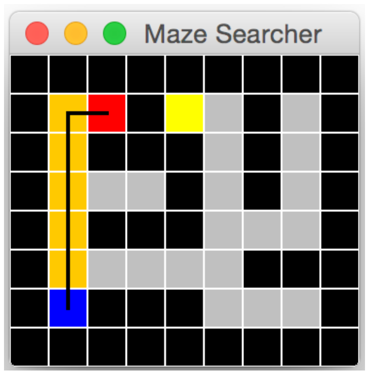
Well, keep on doing that. When you get up to the top (row 1, column 1) you can't go up, so you try other neighboring cells. If you look at the cell below, you see the stick, so you know you've already been there), so try going left:



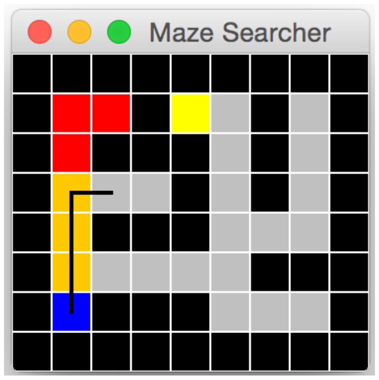
Well, that's just a wall. Try going right:



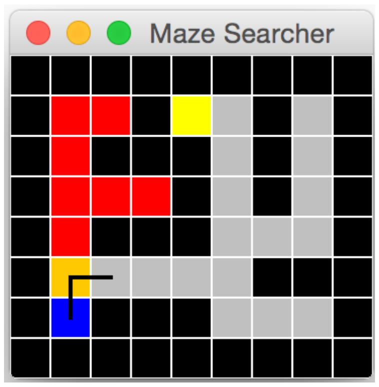
Ok, this is a cell you haven't seen yet. From here, you check up (wall), down (wall), left (already been there), and right (wall), and you're stuck. So you *backtrack* to the cell where you just were. (As you leave the cell, put a red rock on it to mark the fact that it's a dead end.):



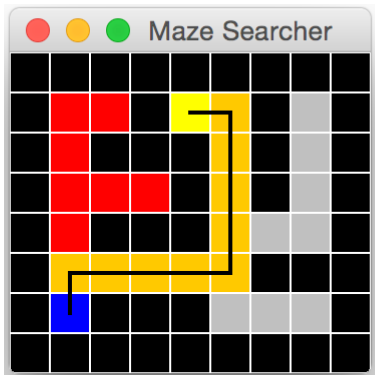
Now back at row 1, column 1, how do you know whether you've searched the other three neighbors already? You can tell since the stick is pointing right that you've already searched all its neighbors. ***That's because we're always checking neighbors in the order up, down, left, then right***. You return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



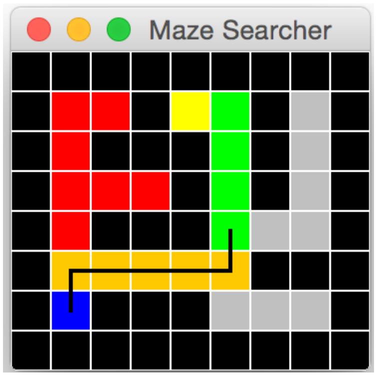
But it leads to another dead end, so you end up backtracking again. When you eventually get back to row 5, column 1 (i.e. just above the starting cell) you can again check down (already been there), left (wall) and right, to find another unexplored cell to the right:



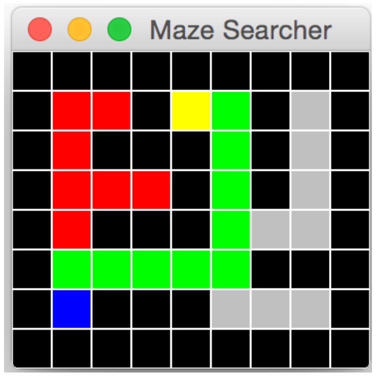
You continue searching neighboring cells, exploring up, down, left, and right (always in that order) marking with sticks as you go.



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 princess, 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 wall or 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 isn't an unexplored cell (i.e. it's a wall, or been there 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 found and return true
    // if we fall through to this point after searching all four directions…
    mark cell c as a dead end and return false
```

The enum type maze_api.CellStatus

As you search, an important part of the process is to "mark" each cell with its current state, like what we did with the sticks and the red and green rocks, above. The state of a cell could be:

```
NOT_STARTED
SEARCHING_UP
SEARCHING_DOWN
SEARCHING_LEFT
SEARCHING_RIGHT
FOUND_IT
DEAD_END
```

To keep track of these possible values, we define a set of 7 constants in the `enum` type `CellStatus` having the names above. Note that the `MazeCell` type also has two methods `isWall()` and `isGoal()` for identifying whether it's a wall or is the goal. (The status would be ignored for wall and goal cells.)

The class maze_api.MazeCell

A `MazeCell` is basically just a container for a status, along with instance variables indicating whether it's a wall or goal. See the methods `isWall()`, `isGoal()`, `getStatus()`, and `setStatus()`. 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 UIs to respond to the status change. The type of the observer is `GridObserver`, which is an example of a Java `interface`. (You don't have to worry about any of this now, but we will have a lot to say about interfaces in a couple of weeks.)

The class maze_api.TwoDMaze

The `TwoDMaze` class encapsulates a 2D array of `MazeCell` objects. You'll really just need the method `getCell()` to access the current cell in the search.

It also 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. This is good to know, in case you want to create your own mazes for testing or for fun. (The 'S' character is the start and the '$' character is the goal. The methods `getStartRow()`

and **`getStartColumn()`** can be used to find the position of the start. See the class **`ui.RunSearcher`** for more examples.)

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

The MazeSearcher UIs

There is a simple console UI that you might find useful for debugging. To run it, run **`maze_ui.ConsoleUI`**. In between cell updates, it prints the grid and pauses for you to press ENTER before continuing. In printing the grid, the cell states are represented as follows:

`NOT_STARTED`	**`(blank)`**
`SEARCHING_UP`	**`^`**
`SEARCHING_DOWN`	**`v`**
`SEARCHING_LEFT`	**`<`**
`SEARCHING_RIGHT`	**`>`**
`FOUND_IT`	**`*`**
`DEAD_END`	**`x`**

To get an idea of what to expect, the assignment posting includes a log of the console output from running the searcher with the console UI on the maze **`RunSearcher.MAZE1`**.

There is also a GUI (that was used to generate the color illustrations earlier in this pdf). The main method is in **`maze_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 animation in the assignment posting on Canvas.

Part B - LinesGame solver

This is another backtracking problem. It is not really any longer than the maze searcher from part A, but there are some additional issues to think about. Your task is simply to implement the method **`findSolutions`** in the class **`LinesGameSolver`**.

Modifications to the LinesGame code

The `LinesGame` was originally developed with a human user in mind. To support a backtracking style of recursive search, we need the ability to *undo* a move in order to try extending the line in a different direction; on the other hand, the features of `startLine` allowing us to clear an incorrect line by clicking on its endpoint, or to extend a partially completed line, are not useful in this style of search. The sample code for mini3 includes the full code for a version of `LinesGame` with the following modifications:

- There is an `undoMove` operation in `LinesGame`
- The `addCell` method includes a bounds check so that it will do nothing if the row or column is out of range.
- The `GridCell` class has a copy constructor.
- There is a class `api.Solution` that acts as a container for a completed game; its constructor makes a deep copy of a game instance so it can be saved in a list of solutions
- The `ui.GameMain` class has an additional method `display`() that can be used to view a completed game

It is important to note the limitations of the `undoMove` operation. *It will work correctly only under the assumption that* `startLine` *is only ever invoked on an endpoint of an* ***empty*** *line.* This is just what we need for this application, but it would not work more generally as part of playing the game through the GUI.

LinesGame solver algorithm

The basic idea is very similar to the maze searcher. Suppose in the simplest case that there is only one line, so we are really just finding all possible ways to connect its two endpoints. We start by calling `startLine` at the first endpoint. There are potentially eight different neighboring cells we might try to add to the line, so we try calling `addCell` on them in turn, **always starting with the cell directly above and going in counterclockwise order**. If adding the neighboring cell is successful, we recursively search, starting at the new cell. There are three eventual possibilities:

1. We get stuck before reaching the other endpoint, i.e. we reach a cell where we can't add a new cell in any direction.
2. We reach the other endpoint (i.e., the line is connected), but the game is not complete
3. We reach the other endpoint *and* the game is complete; in that case we construct a `Solution` object from the game and add to our list of solutions.

When there is only one line, then in all three cases, the next step is simply to return, backtracking to the previous cell and resuming our iteration through the neighbors. *An important point is that upon returning from the recursive call, we have to invoke the* **`undoMove`** *operation to undo the move we made, before trying the next neighbor.*

If the game has multiple lines, there is an additional possibility for case 2: when we reach the other endpoint for the line, we have to check whether there are more lines to search. If so, we have to end the current line and start another one at the first endpoint for the "next" line. **Our convention will be that we always try the lines in the order they occur in the game's line list.** Again remember that on returning from a recursive search after a **`startLine`**, we have to undo that move. The game **`undoMove`** operation is designed to handle this case - removing the first endpoint from the line, it resets the current line to be whatever it had been before that **`startLine`** invocation.

The algorithm can be roughly described in pseudocode like this:

```
if the current line is connected
    if it's the last line in the game
        if the game is complete
            add the game to solutions list
        else
            return
    else
        end the current line
        find the next line
        invoke startLine at its first endpoint
        recursively find solutions
        undo
else
    get the last location for the current line
    for each neighboring cell
        try adding the cell
        if successful (*)
            recursively find solutions
            undo
```

(*) to check whether an addCell operation is successful, you can just look at the move count before and after calling addCell

The method **`findSolutions`** has just two parameters: the game instance, and an arraylist of **`Solution`**s in which to accumulate the results. There is no need to have parameters for the current cell to be searched, since that information is always available by querying the game for the last cell in the current line. The recursive **`findSolutions`** method is first invoked from the public method **`doSolve`**, which adds the first endpoint to the first line in the game, constructs an empty list of solutions, and returns the completed list when the recursive search returns. The **`doSolve`** method is already written and should not be modified.

The skeleton for **`LinesGameSolver`** in the sample code also includes a couple of helper methods that might be useful: one for getting a list of neighbors, and one for finding the index of a line in the lines list.

Viewing LinesGame solutions

There is a sample main class in the default package with an example of how to invoke the solver and make use of the list of solutions.

A "solution" is just a completed game instance, and a **`Solution`** object is just a deep copy of the game state when it was found to be complete. You can always examine the solution by calling **`System.out.println`** on the **`Solution`** object, which invokes **`toString`** on the underlying game instance. If you want to check what the solution looks like in the GUI, that is possible too; just invoke **`ui.GameMain.display(aSolution.game()).`**

The sample code

The sample code is an archive of 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 2. 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 **`miniassignment3`**. 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 **`miniassignment3`**. *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 7, especially the exercise described on page 2, where you use the debugger to visualize what's happening on the call stack as recursive calls are made. You might 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* a value, and are no longer on the stack.

What to turn in

Please submit, on Canvas, the zip file that is created by the SpecChecker. The file will be named **`SUBMIT_THIS_mini3.zip`**. and it will be located in the directory you selected when you ran the SpecChecker. It should contain one directory, **`mini3`**, which in turn contains two files, **`MazeSearcher.java`** and **`LinesGameSolver.java`**. Always LOOK in the zip file the file to check.

Submit the zip file to Canvas using the Miniassignment3 submission link and verify that your submission was successful. If you are not sure how to do this, see the document "Assignment Submission HOWTO" which is link #9 on our Canvas front page.
We strongly recommend that you just submit the zip file created by the specchecker, AFTER CHECKING THAT IT CONTAINS THE CORRECT CODE. ***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 **mini3**, which in turn should contain the two required files. You can accomplish this by zipping up the **src** directory of your project (NOT the entire 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.