Schedule
Calendar
Syllabus
Questions
Material
Assignments
Help Hours

PA0
PA1
PA2
PA3
Exam 1
PA4
PA5
PA6
Exam 2
PA7
PA8
Exam 3
Final Exam - Part 1
Final Exam - Part 2
Final Exam - Part 3

CSE 12 Programming Assignment 2

Worklists are A MAZE ING

This assignment is open to collaboration.

This assignment will teach you how to use stacks and queues as worklists, how to implement an important search algorithm, and how the worklist choice affects the algorithm.

This assignment draws ideas from an assignment by Prof Langlois and Alvarado, which in turn drew from a CSCI 151 lab assignment from Oberlin college.

This PA is due on ** **Tuesday, April 19 at 10:00pm** **

CSE Mantra: *Start early, start often!*

You will notice throughout the quarter that the PAs get harder and harder. By starting the quarter off with good habits, you can prepare yourself for future PAs that might take more time than the earlier ones.

Getting the Code

You can find the code from: https://github.com/ucsd-cse12-sp22/cse12-pa3-Mazes. There are two easy ways to download the starter files.

1. Download as a ZIP folder

   After going to the Github repository, you should see a green button that says *Code*. Click on that button. Then click on *Download ZIP*. This should download all the files as a ZIP folder. You can then unzip/extract the zip bundle and move it to wherever you would like to work.

2. Using git clone (requires terminal/command line)

   After going to the Github repository, you should see a green button that says *Code*. Click on that button. You should see something that says *Clone with HTTPS*. Copy the link that is in that section. In terminal/command line, navigate to whatever folder/directory you would like to work. Type the command `git clone _` where the `_` is replaced with the link you copied. This should clone the repository on your computer and you can then edit the files on whatever IDE you see fit.

If you are unsure or have questions about how to get the starter code, feel free to make a Piazza post or ask a tutor for help.

Code Layout

- `Maze.java` – you will edit this file
- `MazeSolver.java` – you will edit this file
- `SearchWorklist.java` – you will edit this file
- `Square.java` – you *cannot* edit this file
- `TestSolvers.java` – you will edit this file

Part 1: The Structure of a Maze Solver (30 points)

There are a few components to the maze solver:

- The data used to represent the maze
- A choice of worklist to use for keeping track of the spaces that still need visiting
- An algorithm that uses the worklist to traverse the maze and find a solution

You will implement *two* versions of the worklist, and *one* algorithm that will be parameterized to work with either type. Then you can put them together to see the different versions work, and compare them.

The Worklists

You will implement the `SearchWorklist` interface *twice*. First you will implement it with stack semantics, so `add` will "push" and `remove` will "pop", and then you will implement it with queue semantics, so `add` will "enqueue" and `remove` will "dequeue." In both cases, the `isEmpty` method should return `true` when the worklist has no items in it. These are the *only* three methods that should be implemented on these classes, and you shouldn't change any interfaces.

TODO: Methods to Implement for SearchWorklist (StackWorklist, QueueWorklist)

```
void add(Square c)
```

Adds square to the worklist.

```
Square remove()
```

Removes square from the worklist and returns the square that was removed.

```
boolean isEmpty()
```

Returns true if worklist is empty, false otherwise.

Note: You are free to use any built-in Java collections to implement these using the adapter pattern (`LinkedList`, `Stack`, etc), as long as they have the appropriate behavior. This may mean that your implementation is no more than a dozen lines of code! There is one constraint we'd like you to respect – make sure the constructors take no arguments, and initialize the worklist to be empty.

The `Maze` and `Square` Classes

There are several classes provided for you that both represent the maze and help create it.

`Square`

A `Square` represents a single square in the maze. It has the following fields:

- `row` and `col`, which represent its coordinates
- `isWall`, which is true if the square represents a wall, false if it is an empty space
- `previous`, which you will use in the search algorithm to keep track of the path from the finish back to the start
- `visited`, which is initially false, and you will use in the search algorithm to keep track of squares that have been searched already and shouldn't be re-searched

You should read the methods on the `Square` class, as you will use them to manipulate and access these fields during the search algorithm.

`Maze`

The `Maze` class represents a rectangular maze with obstacles, a start, and a finish. Since it just represents data, and the fields don't change via any methods on the class, we make them all `final` and `public`, which makes access easier. So to access the `cols` field of a `Maze` instance with a reference stored in a variable `m`, just write `m.cols`. You **should not** modify anything in Maze.java **except** to implement `storePath()` function.

The fields are:

- `rows` and `cols`, which represent the number of rows and columns in the maze

- `contents`, which contains a reference to an array of arrays of `Square`s, or `Square[][]`. This represents the entire maze, and the inner arrays represent the *rows*. This means:
  - The upper left corner of the maze is at `contents[0][0]`
  - The bottom left corner is at `contents[this.rows - 1][0]`
  - The bottom right corner is at `contents[this.rows - 1][this.cols - 1]`
  - The top right corner is at `contents[0][this.cols - 1]`
- `start` and `finish`, which represent the start square and end square for searching. They contain references to the corresponding `Square`s that are in the `contents` array.

The `Maze` class has a useful constructor just for testing, which we describe in the testing section below.

```
private char[][] buildBackground()
```

Produce a grid with # for walls and _ for empty spaces, nothing else (e.g. no start and finish).

```
public String[] showSolution(ArrayList<Square> visitedHere)
```

Produce a string array like the arguments to the String[] constructor, but with a '*' in each empty space that is part of the solution given the solution array list path. This method should be called with the result of storePath method.

TODO: Methods to Implement for Maze

```
public ArrayList<Square> storePath()
```

Return the solution path as an ArrayList of square that contains all square from start to finish. If there does not exists a solution path from start to finish, return an empty ArrayList instead.

The Search Algorithm

The search algorithm we will use was presented in class, and is rewritten here:

```
// Note: DO NOT initialize a new worklist in solve method as the worklist is passed in
initialize wl to be a new empty worklist (stack _or_ queue)
add the start square to wl
mark the start as visited
while wl is not empty:
  let current = remove the first element from wl (pop or dequeue)
  if current is the finish square
    return current
  else
    for each neighbor of current that isn't a wall and isn't visited
      mark the neighbor as visited
      set the previous of the neighbor to current
      add the neighbor to the worklist (push or enqueue)

if the loop ended, return null (no path found)
```

TODO: Methods to Implement for MazeSolver

```
public static Square solve(Maze maze, SearchWorklist wl)
```

You will implement this algorithm, in Java, in the `solve` method of `MazeSolver`. The parameters of `MazeSolver` are a `Maze` instance and a (empty) worklist to use. To test the maze, you can pass in different implementations of the worklist, and sample mazes.

Note that, for testing, returning `null` is how your implementation indicates that there is no possible path from the source to the target.

There is one constraint on your implementation: When checking neighbors, you *must* add them to the worklist in the following order: **North, South, East and West**. So you should first add (if it is not a wall or out of bounds) the `Square` one row above (one *lower_index), then the* `Square` *one row below (one _higher* index, because the top row is row 0), then the `Square` one column to the right, then the `Square` one column to the left. Our reference implementation uses this order and you should as well.

Please note that this is the order in which `add` should be called, *independent* of the worklist implementation.

One place where the implementation can become complicated is when checking for available neighbors. It may be very useful to introduce a *helper method* that checks if an offset from a particular coordinate is an empty square; this method might have a signature such as:

```
// Return true if the location of s, offset by rowOffset and colOffset, is in
// bounds and not a wall, false otherwise
boolean availableNeighbor(Square[][] contents, Square s, int rowOffset, int colOffset)
```

You are not required to write this method, but doing doing so will provide useful practice for future reference in other assignments. You also might find variations of the method to be useful (i.e. return a `Square` if it is available, that take a `SearchWorklist` and add the element if it's available, etc.) when implementing certain cases of the algorithm.

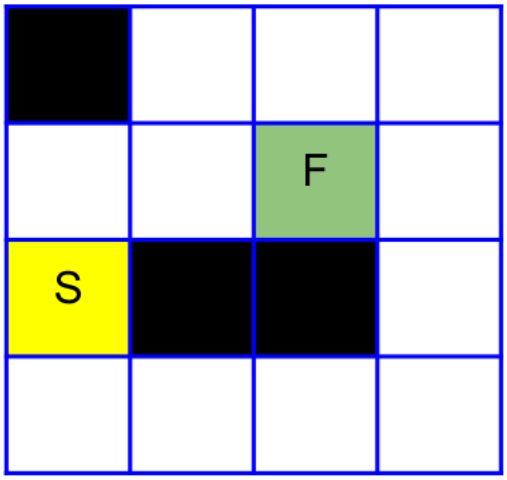
Writing Tests

You should test your storePath, solver and the worklist implementations. Here's some advice and help on doing it.

First, there is a constructor for `Maze` that accepts a `String[]` as an argument. The input uses a plain text format where:

- `#` indicates a wall
- `_` indicates an empty space
- `F` indicates the finish square
- `S` indicates the start square

For example, see the following maze:



If the S quare is the starting point, the F square is the finishing point, and the black squares are walls, we can represent the maze with the following lines of strings:

```
#___
__F_
S##_
____
```

To build a new maze with the above pattern when testing, you can write the following:

```
Maze m = new Maze(new String[] {
            "#___",
            "__F_",
            "S##_",
            "____"
        });
```

On a successful run of a solvable maze, your `solve` method will have set previous pointers from `finish` back to `start`. We wrote a method called`showSolution` that will produce a similar array as a result, but with a `*` for each square that was part of the path from start to finish given by the `storePath` function. Thus, we suggest writting tests to test `storePath` function first. For this example, the solution with a `StackWorklist`, and the add order specified above, is:

```
#___
__F*
S##*
****
```

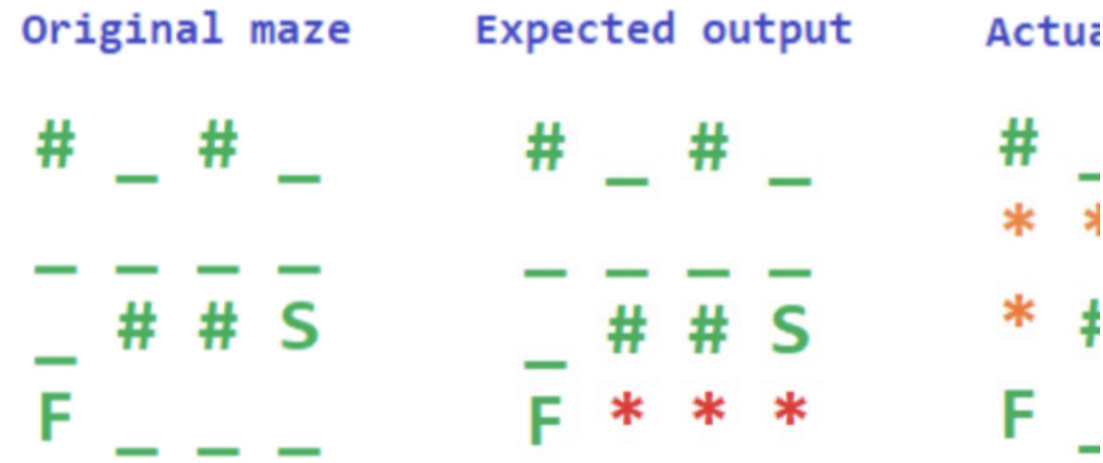
With a `QueueWorklist`, the answer should be

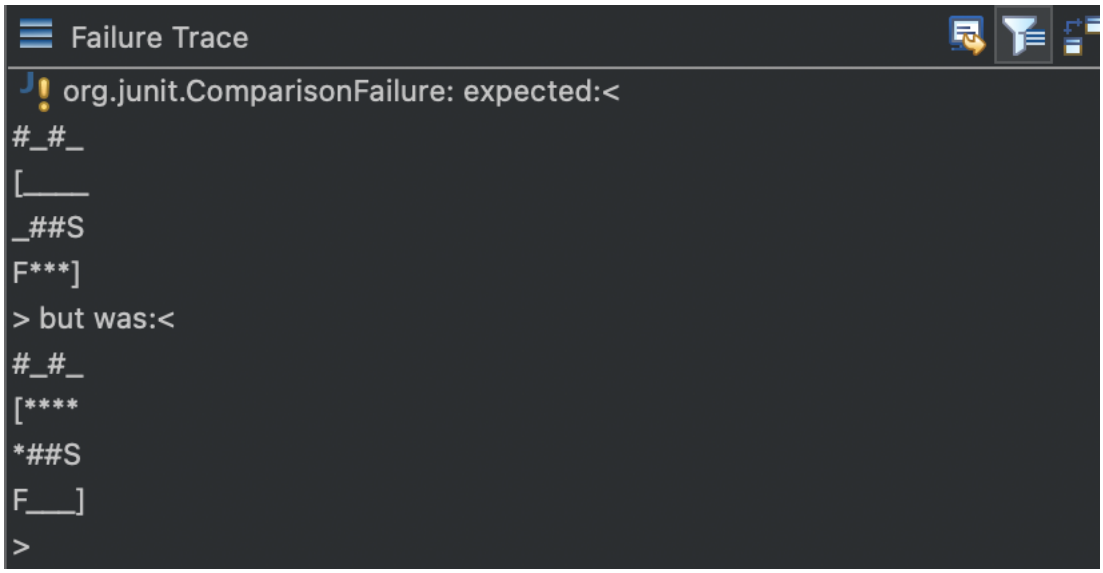
```
#___
**F_
S##_
____
```

You can use the `assertArrayEquals()` method to check if your Maze solution matches the expected solution and `assertEquals` to compare Arraylist string representation. If you want a more detailed solution that tells you exactly which parts of your maze are incorrect, simply run the `formatMaze()` helper function to your actual and expected Mazes, and `assertEquals()` them.

Here is what the JUnit output looks like on a failed solution (incorrect orientation) testing with showSolution:

Using a StackWorklist





Note that this is assuming a StackWorklist was used. The JUnit output will show you what segments of your mazes were different (in this case, rows 1-3).

Evaluating Tests

The thoroughness and correctness of your tests will be graded automatically. To test correctness, we will run your tests against our reference implementation, and they should all succeed. The thoroughness will be assessed by running your tests against each buggy implementation and checking if the results are different than on the reference implementation.

Your implementation will also be graded for correctness. On Gradescope, we have tests that will be run against your implementation and they should all succeed. It would be a good idea to write tests for your own implementation as well.

You will be able to see the correctness information in Gradescope to confirm that your tests and implmementation match our expected behavior.

The following table shows the test case breakdown along with some descriptions to help you as you write your own tests. Note, **this is NOT comprehensive** as you will have to think of some of your own test cases but this will help guide you.

Test Cases	Description	Points
chaff implementations	The following are examples of bad implementations where your tests will be expected to catch the bugs. Look at the names to help get an idea of what the bug could be. For example, `chaffsolve_stopearly` could indicate that the maze solver stops too early (this could mean that the maze solver stopped before it checked all the squares needed for the correct solution). - chaffsolve_diagonalmoves - chaffsolve_stopearly - chaffsolve_difforder - chaffwalls_ignore	8
wheat implementations	TestSolvers.java will be used against a correct implementation. This will check if the tests written are correct and do not flag any errors for the wheat implementation.	2
method tests	Each method (`add`, `remove`, `isEmpty`) of the Stack and Queue will be tested. Make sure that you have the correct behavior for both data structures.	6
implementation tests	There are also tests that your implementation needs to pass. This is not a comprehensive list but here are some of the cases you might want to write tests for. Keep in mind there may be other tests on Gradescope that are currently hidden. - the maze has no walls - the maze has a lot of walls - finish and start are right next to each other - finish and start are in two different corners	16

Part 2: Gradescope Assignment (6 points)

Make sure to submit directly to the Gradescope assignment: "Programming Assignment 3 - questions". You will also answer the following questions on Gradescope regarding the assignment:

- In your implementation, could the `setPrevious()` method ever be called twice on the same square during a single run of `solve()`? Give an example of when it would happen, or argue why it can't.
- Argue for or against this statement: "Solving a solvable maze with a queue worklist will always produce a path with length less than or equal to solving the maze with the stack worklist." Either provide a counterexample, or write a sentence or two about why this must be true.
- Argue for or against this statement: "Solving a solvable maze with a queue worklist will always visit equal or fewer squares than solving the maze with a stack worklist." Either provide a counterexample, or write a sentence or two about why this must be true.

In addition, put any collaborators you worked with in the Collaborator section of the gradescope assignment as described in the collaboration policy for open assignments.

Style (4 points)

On this PA, we will give deductions for violating the following style guidelines:

- Lines longer than 100 characters
- Inconsistent indentation
- Test method names that don't have meaning related to the test
- Helper method names that aren't meaningful

We are also introducing some new guidelines. These new guidelines won't be graded for credit on PA3, but may be on future PAs, and you may get feedback on them:

- File headers
- If you write a helper method with a body longer than 2 statements, we recommend adding a header comment (a comment above the method) that summarizes what it does in English.
- Avoid redundant in-line commenting; Some examples of redundant comments are:

```
// Check if n is null and throw an exception if it is:
if(n == null) { throw new NullPointerException(); }
```

```
// Iterate from 1 to twice the array's length
for(int i = 0; i < array.length * 2; i += 1) {
  ...
```

  Write comments only when they describe an assumption, summarize, or bring up an interesting point that isn't directly described by the code. Focus on making the code understandable on its own.

Submission Checklist

- Implementations of StackWorklist and QueueWorklist (3 methods each)
- Implementation of solve(), including any needed helpers
- Tests for StackWorklist, QueueWorklist, and solve()
- Good style for all of the above
- Answers to Gradescope Assignment questions

Submitting

Part 1

On the Gradescope assignment **Programming Assignment 3 - code** please submit the following files:

- Maze.java
- MazeSolver.java
- SearchWorklist.java
- Square.java
- TestSolvers.java

You may encounter errors if you submit extra files or directories. You may submit as many times as you like till the deadline.

Part 2

Please submit your answers to the questions from part 2 on the Gradescope assignment **Programming Assignment 3 - questions**. You may submit as many times as you like till the deadline.

Scoring (42 points total)

- **Coding Style** (4 points)
- **Correctness** (32 points)
  - Does your code compile? If not, you will get 0 points.
  - Does it pass all of the provided unit tests?
- **Gradescope Assignment** (6 points)

Extension

This is not for credit, but you may enjoy trying it! Feel free to discuss on Piazza or with each other.

Provide an implementation of `SearchWorklist` that, in the `remove()` method, picks the Square to remove in the following way:

- Let the *path distance so far* of each square be the number of nodes on the path from the square back to the start.
- Let the *best possible ending* for each square be the *Manhattan Distance* between that square and the exit square.
- Let the *Manhattan Distance* between two squares be:

  ```
  |row1 - row2| + |col1 - col2|
  ```

- Choose the square in the worklist with the *smallest sum* of its best possible ending and path distance so far to remove in each call to remove().

Test out your implementation. In what ways is it better and/or worse than the stack and queue worklists above? Do you need to change the worklist algorithm at all in order to use it? What information did you need to provide in the constructor in order to implement remove in this way?

If you implement this, please *don't* include tests in `TestSolvers.java` that use this new worklist, as it may not work with the autograder. Feel free to include them in a separate test file.