Project 3

Due Date: Friday 7 October by 11:59 PM

General Guidelines.

The method signatures provided in the skeleton code and/or API in the handout indicate the required methods. You may need additional methods or classes that will not be directly tested but may be necessary to complete the assignment, and you are welcome to add those into the classes.

Unless otherwise stated in this handout, you are welcome to add to/alter any provided java files as well as create new java files as needed, but make sure that your code works with the provided test cases as you will not be submitting the test code as part of your submission. Your solution must be coded in Java.

In general, you are not allowed to import any additional classes in your code without explicit permission from your instructor!
Importing the Math class is okay.

Note on academic dishonesty: Please note that it is considered academic dishonesty to read anyone else's solution code, whether it is another student's code, code from a textbook, or something you found online. You MUST do your own work! It is also considered academic dishonesty to share your code with another student. Anyone who is found to have violated this policy will be subject to consequences according to the syllabus and university policy.

Note on grading and provided tests: The provided tests are to help you as you implement the project and (in the case of auto-graded sections) to give you an idea of the number of points you are likely to receive. Please note that the points indicated when you run these tests locally are not your final grade. Your solution will be graded by the TAs after you submit. Please also note that these test cases are not likely to be exhaustive and find every possible error. Part of programming is learning to test and debug your own code, so if something goes wrong, we can help guide you in the debugging process but we will not debug your code for you.

Project Overview.

In this project, you will use a variation of DFS and BFS with a Deque structure to solve a word search.

Part 1. Word Search

In this part, the grid you are given will contain individual letters represented as Strings, and you must implement a word search. A word can occur as any path from the first letter to the last where a step in the path can go up, right, down, or left. (No diagonals). Also, it is possible for the same letter to be used more than once in the same word. (See ARIZONA example below.)

	0	1	2	3	4	5	6	7	8	9
0	A	N	O	D	E	F	G	H	I	J
1	R	O	N	N	O	P	Q	W	E	R
2	I	Z	A	E	R	P	A	S	D	F
3	G	H	J	V	S	L	G	H	J	K
4	A	S	W	S	R	F	F	G	H	J
5	B	A	M	N	I	V	C	X	Z	A
6	S	W	Y	U	R	E	V	T	B	Y
7	N	D	C	I	S	R	P	O	G	H
8	I	L	A	T	Y	S	C	V	R	F
9	W	S	T	A	S	D	F	G	H	E

The output for this search should be a String representation of the locations in the path as shown below.

Examples:

```
ARIZONA: (0, 0)(1, 0)(2, 0)(2, 1)(1, 1)(0, 1)(0, 0)
UNIVERSITY: (6, 3)(5, 3)(5, 4)(5, 5)(6, 5)(7, 5)(7, 4)(7, 3)(8, 3)(8, 4)
WILDCATS: (9, 0)(8, 0)(8, 1)(7, 1)(7, 2)(8, 2)(9, 2)(9, 1)
```

Additionally, the search function will take two integer parameters that indicate the row and column that your search will start from.

Your implementation should follow these guidelines.

- Use a BFS algorithm to find the first character in the word. This search will not be included in the final path that is printed out, but it is included in the overall grid access counts, so it is important to use BFS in order to find the closest option first and go on from there.
- Once you find a possible starting place for the word, use a DFS algorithm to search for the word itself.
- **Always search a location's neighbors in the following order: UP, RIGHT, DOWN, LEFT. If you don't, you may get an alternate route that does not match the test cases.**

Example: Let's say we are searching the above grid for ARIZONA starting at (2, 3). First we search for the closest "A" using a BFS algorithm, which is at (2, 2). We then use DFS to search for the word. When we don't find it, we continue the BFS for the next closest "A". Eventually, we find the correct "A" and find the path as noted above.

Testing.

- Test cases and test code is provided for you. Keep in mind that these may not necessarily catch all possible errors, so it's up to you to make sure you are handling any corner cases.
- Note: The test cases search for multiple words in the same grid, so you will need to reset some things either at the beginning or end of each word search in order to make sure you get accurate results in the subsequent searches.
- To avoid any differences in String output, use the *toString* method provided in *Loc.java* to print out the locations. Do not add any spaces or characters in between them.

Provided Code.

Besides the test code, you are provided with the *Grid.java* class. You should NOT change this code as it will not be included in your submission. However, you may change the *Loc.java* class, and you should include your version of *Loc.java* in your submission even if you don't change it. You are also provided with the *Deque.java* and the *EmptyDequeException.java* class, which you are allowed to change as well. You are not allowed to use other imported structures, but a Deque should be enough for your BFS and DFS implementations as it can function as both a Stack and a Queue. You also may need to use multiple Deques for storing different things (e.g. it might be useful for keeping track of Loc items that need to be reset in between word searches.)

API.
Implement your solution in a class called *Puzzle.java*. The required methods are listed below.

Method Signature	**Description**
`public Puzzle(Grid grid)`	`constructor`
`public String find(String word, int r, int c)`	find and return (as a String) the path containing *word*; start the search at (r, c)

BFS and DFS

BFS (breadth-first-search) is a graph/tree traversal algorithm that searches closer items before items that are further away. This means that we can use it to find shortest paths between items. In our case, we will use it to find the first letter in a word (we check the elements closest to our starting position first.)

BFS is often done with a Queue because we can place the neighboring positions onto a queue and visit them in FIFO order (thereby checking all the closest ones before the farther ones.)

Below is basic BFS pseudocode. You may need to adjust it somewhat to work for this problem, but the main idea will be the same.

```
Algorithm bfs
Input grid G source location L
    Q := a queue
    Q.enqueue(L)
    while !Q.isEmpty()
        v := Q.dequeue()
        visit v
        for all w next to v:
            if w is not in the queue and not already visited:
                Q.enqueue(w)
            end if
        end for
    end while
end bfs
```

Some things to think about:

- The pseudocode above is for traversing the whole grid. You are not trying to do that. Instead, you are looking for a specific item.
- You will need to have a way to determine which locations have already been visited and which have already been put into the queue. I suggest adding fields to the Loc class to do that.
- You may need to come back and continue your *bfs* after attempting a *dfs* that fails, so you'll need to think about how to do that.

DFS (depth-first-search) is another way to traverse a graph, tree, or grid. Instead of searching closest items first, it follows a particular path until it reaches a dead end and then backtracks. Because of the backtracking, it makes sense to use a Stack for this algorithm because you can easily pop off the last location each time you have to backtrack. **Also, DFS is generally easier to do recursively than iteratively.**

Below is some basic pseudocode for DFS.

```
Algorithm dfs
Input: Grid G and starting position s
Output: path, a Stack representing the steps in the
word from the first letter to the last
path := an empty Stack
push the current location onto the Stack
if you've reached the end of the word then return
else check the next neighbor and if it is the
character you are looking for, do a recursive call on
that location to keep searching
but if there are no neighbors left to check, then
backtrack—which means returning to the previous
recursive call and popping the last location off the
stack
return path
```

Some things to think about:

- You should use DFS to search for the word AFTER you have already found the first letter.
- Again, you are not trying to traverse the whole grid. You are trying to find a particular word. Think of a possible path as “the next letter matches what you’re looking for.” If it doesn’t match, that’s a dead end–so backtrack.
- By “steps”, we mean UP, RIGHT, DOWN, or LEFT, but keep in mind that there is no possible step unless that character actually matches what you’re looking for.
- Make sure that when you backtrack, you don’t end up trying the same steps over and over again.
- Keep in mind that each set of tests comes from the same Puzzle, so the same grid. You will need to reset some things after each word–particularly fields that you set up when you do BFS. I recommend keeping track of Locations that need to be reset by putting them on a Deque so that you don’t have to run through the whole grid.

REMINDER: Always search a location’s neighbors in the following order: UP, RIGHT, DOWN, LEFT. If you don’t, you may get an alternate route that does not match the test cases.

Testing your code on lectura.

Once you have transferred your files,

- you can compile a java file in the terminal with the following command:
  ```
  javac <filename>
  ```
- you can can compile all the .java files with this command:
  ```
  javac *.java
  ```
- you can run a compiled file with the following command:
  ```
  java <filename>
  ```

Submission Procedure.

To submit your code, please upload the following files to **lectura** and use the **turnin** command below. Once you log in to lectura and transfer your files, you can submit using the following command:

turnin cs345p3 Puzzle.java Deque.java Loc.java

Upon successful submission, you will see this message:

Turning in:

Puzzle.java -- ok
Deque.java-- ok
Loc.java-- ok

All done.

Note: Your code submission must be able to run on **lectura**, so make sure you give yourself enough time before the deadline to check that it runs and do any necessary debugging. I recommend finishing the project locally 24 hours before the deadline to make sure you have enough time to deal with any submission issues.

Grading.

The test cases are worth a total of 60 points, some for finding the path and some for not using too many grid accesses.

Possible Deductions

Item	**Max Deduction Possible**
Bad Coding Style	5 points
Inefficient Solution	25% of total points for that part
Not following instructions	100% of the points you would have earned (i.e. automatic 0)
Not submitted correctly on lectura	100% of the points you would have earned (i.e. automatic 0)
Does not compile/run on lectura	100% of the points you would have earned (i.e. automatic 0)

Other notes about grading:

- If you do not follow the directions (e.g. importing classes without permission, not using an array for the Deque, etc.), you may not receive credit for that part of the assignment.
- Good Coding Style includes: using good indentation, using meaningful variable names, using informative comments, breaking out reusable functions instead of repeating them, etc.
- If you implement something correctly but in an inefficient way, you may not receive full credit.
- In cases where efficiency is determined by counting something like array accesses, it is considered academic dishonesty to *intentionally* try to avoid getting more access counts without actually improving the efficiency of the solution, so if you are in doubt, it is always best to ask. If you are asking, then we tend to assume that you are trying to do the project correctly.

- If you have questions about your graded project, you may contact the TAs and set up a meeting to discuss your grade with them in person. Regrades on programs that do not work will only be allowed under limited circumstances, usually with a standard 15-point deduction. All regrade requests should be submitted according to the guidelines in the syllabus and within the allotted time frame.