Project 4: All About Grids

Due Date: Friday 4 December 2020 by 11:59 PM

General Guidelines.

The APIs given below describe the required methods in each class. You may need additional methods or classes that will not be directly tested but may be necessary to complete the assignment.

Unless otherwise stated in this handout, you are welcome (and encouraged) to add to/alter the provided java files as well as create new java files as needed. 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! For this project, you are allowed to use the following: ***java.util.Queue, java.util.Stack, java.util.ArrayList, java.util.HashMap, java.util.PriorityQueue, java.util.TreeMap.*** *If there are others you would like to use, you must ask first!*

Note: *java.util.HashMap is a hashtable and java.util.TreeMap* is implemented as a red-black tree.

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 receive an automatic 0 on the assignment and may be subject to further consequences according to the university's policies.

Note on grading and provided tests: The provided tests are to help you as you implement the project and to give you an idea of the number of points you are likely to receive. Please note that any 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 are given five problems to solve involving grids. You will be graded both on the accuracy of your solutions and on the efficiency of your solutions. Note that efficiency is determined based on the number of times your solution accesses the grid. DO NOT change the Grid.java code in any way. Also, bear in mind that any attempt to get around the efficiency tests will be considered academic dishonesty and, at the very least, result in a 0 on the entire project. For instance, you should not copy the grid over into your code and then solve the problem so that it will appear that you only accessed the grid $O(N^2)$ times.

`Grid.java` and `Loc.java`

You are provided with two classes. *Grid.java* represents the grid. It is always a square (N X N) and is made up of location objects. Locations are indicated by the row and column and contain a String value. Note that both *Loc.java* and *Grid.java* include methods for dealing with both Strings and Integers. These are to make things easier on you as Parts 1 through 4 involve integers, while Part 5 involves Strings. Take some time to look through both classes before looking at the rest of the project.

You will not be submitting Grid.java with your code, so any changes you make should be for your use only. Your solution should not depend on any changes you make there as your code will be tested using our version of Grid.java (which is what is provided to you).

On the other hand, you will be submitting Loc.java as we expect that you may need to add to it to help with some of your solutions. You should not, however, make any changes to existing code that break the test cases.

A. Maximum Product (15 points)

Consider the grid below.

	0	1	2	3	4
0	4	5	8	1	9
1	7	8	9	7	0
2	0	1	2	8	4
3	7	2	9	1	5
4	3	9	3	5	8

The grid is of size 5×5. If we consider any set of 3 integers in the grid (horizontally, vertically, or diagonally), we find that the set with the highest product is {9, 9, 8} with a product of $9 \times 9 \times 8 = 648$.

In this part of the project, you must implement a solution to the following problem.

Given an $N \times N$ grid of integers and an integer $M \leq N$, determine the maximum product that can be made from M integers that are next to each other in the grid (horizontally, vertically, or diagonally (both directions).)

Some of the test grids will be quite large, so you need to think about the performance of your algorithm.

You should implement your solution in a class called *`Part1.java`*. The required method is described below.

`public static long getMaxProd(Grid g, int mVal)`	`@param g: the Grid`
	`@param mVal: the m value`
	`@return: the maximum product of m adjacent integers in the grid (horizontally, vertically, or diagonally)`

B. Sorting (15 points)

In this part, you will implement *SortGrid.java*, which sorts the elements in a grid of integers.

Example:

Input:

4	1	0	2	8
10	3	24	2	8
9	0	37	29	3
43	22	7	11	0
5	67	3	2	0

Output:

0	0	0	0	1
2	2	2	3	3
3	4	5	7	8
8	9	10	11	22
24	29	37	43	67

It is up to you to figure out a solution to this problem according to the requirements below. We will test your solution for accuracy (i.e. does it sort the grid?) and for efficiency. Points will be awarded based on accuracy *and* efficiency, so you should design your solution with that in mind. Keep in mind that some algorithms may be efficient on smaller grids or on specific grids but don't work as well on other grids, so be sure to test your solution on many different inputs.

The required method is described below.

public static void *sort*(Grid *grid*)	sorts *grid*

Implementation Requirements

- **You *may* use a two-dimensional array as a temporary copy in case you decide to use a variation of MergeSort. You may *not* use that 2D array to actually do the sorting in order to avoid getting more access counts.**
- **You may *not* copy the items over into a regular array of size *N* X *N*. In other words, the sorting must be done in the grid.**

C. Sequence (20 points)

Consider the following grid.

0	2	4	6	2
1	2	8	2	3
2	4	7	8	2
3	4	5	8	9
3	4	6	0	2

A sequence here is defined as a path through the grid made of consecutive integers. Your task in this part is to search for a sequence where the input is a starting location (i, j) and a value v.

Example (using the Grid above):
Input: $i = 2, j = 0, v = 6$
Output: $(2, 0)\ (3, 0)\ (3, 1)\ (3, 2)\ (4, 2)$

The output corresponds to the following sequence starting at $(2, 0)$ and ending at a location with the value of 6.

0	2	4	6	2
1	2	8	2	3
2	4	7	8	2
3	4	5	8	9
3	4	6	0	2

Write your solution in a file called *Sequence.java,* following these specifications.

The required methods are described below.

Sequence API

Sequence (Grid *grid*)	constructor; creates a new Sequence object for the specified Grid object
void *getSeq*(int *i*, int *j*, int *val*)	searches for a sequence starting at location *(i, j)* and ending at a location with the value *val*; the path, if it exists should be stored in the Stack variable called *path;* if no sequence exists, *path* should be empty
String *toString*()	returns a String representation of the sequence; use the toString method provided in Loc, and make sure you follow the format of the example above (e.g. no

	`spaces between locations)`

NOTE: When there are multiple options for steps to take in a path, you should give precedence in the following order: UP, RIGHT, DOWN, LEFT.

D. Closest Pair (20 points)

In this section, you will search for the closest matching value in a grid, given a starting value.

Consider the following grid and the starting location (0, 3). The closest pair would be (4, 2). In the same grid, if the input location is (2, 3), the closest pair would be (3, 3). In the case of the starting location (1, 0), there is no pair.

0	2	4	6	2
1	2	8	2	3
2	4	7	8	2
3	4	5	8	9
3	4	6	0	2

Given a Grid and a starting location, your task is to implement an algorithm for finding the closest pair for the value at the starting location. Write your solution in *ClosestPair.java*. The following describes the required methods.

ClosestPair API

`ClosestPair (Grid grid)`	`constructor; creates a new ClosestPair object for the specified Grid object`
`Loc search(int x, int y)`	`searches for the closest* location with a value equal to the value at (i, j); return null if no match is found`

**Closest* means that it takes the fewest number of steps to reach the new location by moving only UP, RIGHT, DOWN, or LEFT. Precedence should be given in that order in the case where two items are the same number of steps from the starting location.

E. Word Search

A word search is a puzzle containing a bunch of letters in a grid in which you search for specific words that are hidden in the grid. There is an example below.

D	R	L	B	V	W	B
P	E	E	K	P	E	O
O	U	D	K	P	S	I
U	A	R	O	A	T	L
L	E	Z	D	N	M	E
Y	D	N	I	U	E	R
I	O	P	H	H	E	Y

This is a 7×7 grid of letters.

You should implement your solution in *Puzzle.java*. The following describes the required methods.

Puzzle.java API

Puzzle(Grid grid)	This is the constructor. It constructs a Puzzle object associated with the given grid.
Loc[] search(String word)	This is the main method you are tested on. It searches the puzzle for the given word and returns a 2-element array of Loc objects: {*start, end*} where *start* is the location where the word starts and *end* is the location where the word ends. If the word cannot be found, return *null*.

Examples

A	**D**	**E**	**S**	**E**	**R**	**T**
N	**N**	**S**	K	P	O	E

O	U	**O**	K	S	S	L
S	A	**N**	**Z**	A	T	I
C	E	**O**	D	**I**	M	O
U	T	**R**	I	U	**R**	B
T	O	**A**	H	H	E	**A**

search("TUCSON"): {(6, 0), (1, 0)}
search("ARIZONA"): {(6, 6), (0, 0)}
search("DESERT"): {(0, 1), (0, 6)}
search("SONORA"): {(1, 2), (6, 2)}

A few more things…

- Words can be found diagonally (either way), bottom to top, top to bottom, left to right, or right to left.
- Words are always in a line (horizontally, vertically, or diagonally).
- You will be tested on three grids:
  - a 15×15 grid with 15 words
  - a 25×25 grid with 25 words
  - a 50×50 grid with 50 words
- You may **not** use the following String functions: *contains, contentEquals, copyValueOf, indexOf, lastIndexOf, matches, offsetByCodePoints, regionMatches, toCharArray*

Submission Procedure.

To submit, please upload the following files to **lectura** by using **turnin**. Once you log in to lectura, you can submit using the following command:

```
turnin cs345-fall20-p4 Loc.java Part1.java SortGrid.java Sequence.java ClosestPair.java Puzzle.java
```

Upon successful submission, you will see this message:

```
Turning in:
    Loc.java -- ok
    Part1.java -- ok
    SortGrid.java -- ok
    Sequence.java -- ok
    ClosestPair.java -- ok
    Puzzle.java -- ok
```

```
All done.
```

Note: Your 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 following rubric gives the basic breakdown of your grade for this Project.

Part 1	15 points
Part 2	15 points
Part 3	20 points
Part 4	20 points
Part 5	20 points

Other notes about grading:

- Any violation of the academic integrity guidelines will result in a 0 on the assignment.
- If you do not follow the directions, you may not receive credit for that part of the assignment.
- Good Coding Style includes: using indentation, using meaningful variable names, using informative comments, breaking out reusable functions instead of repeating them, etc.
- 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.