Project 4

Due Date: Wednesday 30 March by 11:59 PM

General Guidelines.

The method signatures provided in the skeleton code 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. 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!

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.

This project has three parts. The first part requires you to implement a version of Shellsort. The second part requires you to implement a second sorting method for an array wrapper class. The third part requires you to sort a grid.

Provided Files

- `Array.java:` This is a wrapper class for an integer array. You must write your sorting algorithms for Parts 1 and 2 to sort an array of this kind. Note that this file will not be submitted so any changes you make to it should be for debugging purposes only. You should familiarize yourself with this class. There are different ways to interact with the underlying array, and each array access is counted. Note that there is also a temporary array included in there, which you can also access. This is in case you implement some kind of sorting algorithm that requires an extra copy of the array (like Mergesort). Accessing the temporary array is also included in the overall access count.
- `Deque.java:` This is the double-ended queue used in the last project. It is included as a structure that you can use for keeping track of the h-values when you implement Shellsort. This file will not be submitted, so use it as is.
- `EmptyDequeException.java:` This goes alone with the Deque class. Again, this is not included in the submission so use it as is.
- `Grid.java`: This is the grid class that you will use for Part 3. Similar to the Array class, this class keeps track of the grid accesses. It will not be submitted, so any changes you make to it should be for debugging only. Like the Array class, a temporary copy of the grid is provided in case you implement a sorting algorithm that requires an extra copy (like Mergesort). Please note that all the grids for this assignment will contain integers, so you should use the Grid functions that have to do with integers.
- `Loc.java:` This is what the Grid is made up of. In this case, this class will not be submitted, so any changes you make should be for debugging purposes only.
- Testing files: `ShellsortTest.java, ArraySortTest.java, SortGridTest.java.` You should transfer these to lectura to test your code, but do not include them in your submission.
- Test input files: `array*.txt (* = 1-5)` are used for both Parts 1 and 2, and `testGrid*.txt (* = 1-5)` are used for Part 3. You should transfer these to lectura to test your code, but do not include them in your submission.

Part 1. Shellsort

In a file called `Shellsort.java`, implement a Shellsort method. The method must take an `Array` parameter and sort it. You may not use any extra space (including the temporary array in the `Array` class.) You may use a `Deque` to keep track of the h-values.

Although you may use the h-sequence given in class, you are encouraged to experiment with h-sequences to see if you can do better.

Required Method

Method Signature	**Description**
`public static void sort(Array a)`	`sorts the Array using Shellsort`

Part 1 Grading

- Please note that these grading guidelines assume that you follow all the instructions and do not get any deductions.
- There are five input arrays that this method is run on. For each one, we check that the array is correctly sorted. If so, the automated tests should return 20 points total (4 points per test), which is full credit.
- We also check the number of array accesses for each test and compare them against the accesses required by my implementation of Shellsort using the h-sequence given in class. If your implementation uses fewer array accesses, you can get 0.5 extra credit points per test (up to 2.5 EC points total).
- Additionally, we will look at the total array accesses for all 5 tests. If your total is less than my implementation, then you will be eligible to get more extra credit points in a student competition. First place will get 3 EC points, second place will get 2 EC points, and 3rd place will get 1 EC point. Again, this only applies to people who beat my implementation.

Part 2. ArraySort

In a file called `ArraySort.java`, implement another sorting method. The method must take an `Array` parameter and sort it. The only extra space you are allowed is the temporary array provided in the Array class.

- You may use any comparator-based sorting method except the exact Shellsort method used in Part 1.This means you can still do a version of Shellsort if you want, but it needs to differ from the one you did in Part 1 in some way.

- If you use a sorting method that is too simple, you may not get full credit due to the array access count. (e.g. Bubblesort will sort the array, but it probably won't pass the array access count test.)

Required Method

Method Signature	**Description**
`public static void sort(Array a)`	`sorts the Array`

Part 2 Grading

- Please note that these grading guidelines assume that you follow all the instructions and do not get any deductions.
- There are five input arrays that this method is run on. For each one, we check that the array is correctly sorted. If so, the automated tests should return 17.5 points out of 20 (3.5 points per test.)
- We also will look at the total array accesses for all 5 tests. If your access count is better than my implementation of Bubblesort, you will get another 1.5 points. If it is better than a second cutoff, then you will get another point. The second cutoff is approximately the average number from several different sorting algorithms that I implemented. Note that if you pass both these tests, then you get full credit (20 points).
- Additionally, your access count will be compared to a few more cutoffs, giving you the opportunity to earn some extra credit points (up to 2 points).
- Finally, if you get at least 20 points (i.e. your access count is better than the second cutoff), your access count will be compared to other students' access counts and I will award extra credit points as follows: 3 points for the lowest count, 2 points for the next lowest, and 1 point for the 3rd lowest. Note that this only applies if your access count is better than the second cutoff.

Part 3. SortGrid

In a file called `SortGrid.java`, implement a sorting method for a grid. The method must take a `Grid` parameter and sort it. The only extra space you are allowed is the temporary grid provided in the `Grid` class.

A grid is considered sorted when each row is sorted and the last element in each row is less than or equal to the first element in the next row.

I highly recommend that you work with this as a grid and not treat it as a one-dimensional array. That is, take advantage of the fact that it is a grid with rows and columns.

Example:
Original Grid

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

Sorted Grid

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

Required Method

Method Signature	**Description**
`public static void sort(Grid g)`	`sorts the Grid`

Part 2 Grading

- Please note that these grading guidelines assume that you follow all the instructions and do not get any deductions.
- There are five input grids that this method is run on. For each one, we check that the grid is correctly sorted. If so, the automated tests should return 15 points out of 20 (3 points per test.)
- We will also look at the array access count for each of the five tests. There are two cutoffs for this. The higher cutoff is three times the lower cutoff, which is based on my own implementation. As long as you're within the higher cutoff, you will get 1 point per test (so full credit). If you equal or beat my implementation, you will get 1.5 points per test (full credit + 2.5 points extra credit). Keep in mind

that these are tested for each input, though, so you could get a score somewhere in between if your method works better on one input than another.

- Additionally, we will look at the total access count for all five tests. For students who earned full credit (i.e. your access count was better than 3 times my count), we will compare total access counts and award extra credit as follows: 3 points for the lowest count, 2 points for the next lowest, and 1 point for the 3rd lowest. Note that this only applies if you got full credit.

Other notes about grading:

- If you do not follow the directions (e.g. importing classes without permission, using unauthorized extra space, 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 or grid 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 20-point deduction. All regrade requests should be submitted according to the guidelines in the syllabus and within the allotted time frame.

Submission

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 csc345p4 Shellsort.java ArraySort.java SortGrid.java
```

Upon successful submission, you will see this message:

```
Turning in:
      Shellsort.java -- ok
      ArraySort.java-- ok
      SortGrid.java-- ok
      All done.
```

Testing your code on lectura.
It is always a good idea to compile and run your code on lectura before you submit. In order to do that, you need to transfer all files to lectura that are necessary for testing (but only *submit* the ones that are required for submission.) Once the files are submitted, you can use the following commands in the terminal as needed.

`javac <name of file to be compiled>` – this will compile the given file
`javac *.java` – this will compile all .java files in the current folder
`java <name of file to be run>` – this will run the compiled file