Programming Assignment 5

Due Date: Tuesday 4 April by 12:30 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. Importing the Random 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 implement sorting algorithms for Arrays and for Grids, trying to do so in the most efficient way possible. You will also implement an iterative version of Mergesort.

Part 1. MergeSort

The pseudocode you're given for Mergesort in the slides is for a recursive version of the algorithm. However, Mergesort can also be done iteratively (i.e. without recursion). The visualization on Slide #3 in the Mergesort slides is actually showing an iterative version. Basically, it just keeps going through the array, merging successively larger pairs of subarrays until the whole thing is sorted.

In a file called *Merge.java*, implement an iterative version of Mergesort. *Your implementation may not use any recursion.* The required API is below. You must use the *Array.java* class that is provided and you must use the extra array provided through that class to do your merges. That way your access counts will be accurate. No other data structures are allowed.

API for Merge.java

Method Signature	Description	Runtime
`public static void sort(Array array)`	sort the array using an iterative Mergesort algorithm	O(NlogN)

Testing. You are provided with the *MergeTest.java* file, which tests your code and provides an expected score out of 15. Please remember that your code will also be checked manually so make sure it meets all the requirements.

Part 2. ArraySort

In this part, you will be implementing sorting algorithms for Arrays based on what you know about the input array. In each case, you are given specific information about the input array and you should try to write an algorithm that will work well for that specific array. Note that you are not required to write different sorting algorithms for each of the 7 input arrays, but using the same algorithm for all 7 inputs is unlikely to receive full credit.

Note: All the sorting methods should put the array in ascending order.

Note: You must use the *Array* class that is provided and you must be sure not to get around the access counts in any way.

Here are the seven inputs you need to consider.

1. The input array is in descending order.
2. The input array is in random order.
3. The input array is almost sorted, meaning that only a few items are out of order.
4. The input array is locality-aware. That means that every item in the input array is no more than *d* positions away from where it will end up in the sorted array.
5. The input array is mostly sorted with just a small number of unsorted elements at the end.
6. The input array is made up of a small number of sorted subarrays.
7. The input array is random, but the values in the array are relatively limited in range.

Implementation.

- Implement your solution in a file called *ArraySort.java*. All of the test cases will call a single method, whose signature is given below.

  ```
  public static void sort(Array array, int num, int d)
  ```
  - `array: the input array`
  - `num: the input number (1-7)--see above`
  - `d: the d value for #4–all other cases will ignore this`

- Specific guidelines:
  - In general, you should be doing your sorting using the Array class. Any external data structures should be limited to O(1) space and mostly O(1) time. The exception is that you are allowed to use the provided Queue class, but this can only be used *as a Queue* and *as a helping structure*. (i.e. no copying items into the queue, and then sorting the queue…)
  - The *Array.java* class does have a mechanism for providing an extra array that you can access in a way that will still count toward your overall count. For example, you might want to use this if you are doing some kind of Mergesort algorithm. But of course, this limits you to just one extra array and accessing it does affect your overall access count, so use it wisely.
- I recommend that you have separate methods for each of your sorting algorithms and just call those inside the main sorting algorithm according to what the *num* argument is.

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.
- The test cases use specific inputs for each scenario (which you can see in the text files). These are quite large because we want to get an accurate idea of your runtime.
- For each test case, the code checks accuracy (i.e. is it sorted?) and efficiency (by checking access counts) and awards points based on those results. If your algorithm is not very efficient, you may not receive full credit but there are levels of efficiency–some may be good enough for full credit and some may be good enough for extra credit.
- The total possible for each test is 3.5 points (so 24.5 total), but this part is really out of 20 points so 4.5 of those points are "extra."
- The test code should give you a reasonable idea of what you can expect from the grading code and tests.
- Note that if you use any randomization in your algorithm (e.g. Quicksort) your access counts may vary somewhat. If you don't get the grade you expect because of this, we can always run your code a few extra times to see if it improves, you will have to let us know during the regrade window.

Provided Code.

Besides the test code, you are provided with the *Array.java* class and the *Queue.java* class. You should NOT change this code in any way that will affect your code's ability to work as it will not be included in your submission. However, feel free to change it for testing purposes. As a small hint, I used the Queue in my solutions for #5 and #6, but you are by no means required to use it.

Part 3. SortGrid

In this case, you are going to "sort" a grid in two different ways based on two different definitions of a "sorted" grid.

Definition 1: A sorted grid is one where, if you laid the rows side by side, the resulting array would be sorted.

Example:

-3	-2	-2	0	4
4	4	5	5	6

7	8	9	9	9
10	11	12	12	15
16	20	22	23	25

Definition 2: A sorted grid is one where each element is less than or equal to the elements directly to the right and directly down from itself.
Example:

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

A small hint for Definition 2: It tends to surprise students how simple a solution to this can be. :-)

Implementation.

- Implement your solution in a file called *SortGrid.java*. Two methods are required and their signatures are given below.

  ```
  public static void sortA(Grid g)
  ```
  - `g: the Grid to be sorted`
  - `result: g should be sorted according to Definition 1 above`

  ```
  public static void sortB(Grid g)
  ```
  - `g: the Grid to be sorted`
  - `result: g should be sorted according to Definition 2 above`

- Specific guidelines:
  - In general, you should be doing your sorting using the Grid class.
  - The *Grid.java* class does have a mechanism for providing an extra grid that you can access in a way that will still count toward your overall count. For example, you might want to use this if you are doing some kind of Mergesort algorithm. But of course, this limits you to just one extra grid and accessing it does affect your overall access count, so use it wisely.

    - I recommend that you think about the Grid as a grid and not as an array.

Testing.

- Test cases and test code are 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.
- There are five input grids that will each be run with *sortA* and *sortB*.
- For each test case, the code checks accuracy (i.e. is it sorted?) and efficiency (by checking access counts) and awards points based on those results. If your algorithm is not very efficient, you may not receive full credit but there are levels of efficiency–some may be good enough for full credit and some may be good enough for extra credit.
- The total possible for each test is 2 points (so 20 total), but this part is really out of 15 points so 5 of those points are "extra."
- The test code should give you a reasonable idea of what you can expect from the grading code and tests.

Provided Code.

Besides the test code, you are provided with the *Grid.java* and *Loc.java*. You should NOT change this code in any way that will affect your code's ability to work as it will not be included in your submission. However, feel free to change it for testing purposes. Note: It is important to note that these grids are all made up of integers, so you will want to use the `getIntVal` and `setIntVal` methods instead of `getVal` and `setVal`.

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 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 cs345p5 Merge.java ArraySort.java SortGrid.java
```

Upon successful submission, you will see this message:

Turning in:

Merge.java – ok

ArraySort.java -- ok

SortGrid.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 autograder should give you a good idea of your grade. However, your final grade will be determined by the TAs. Below are some possible deductions.

Possible Deductions

Item	**Max Deduction Possible**
Bad Coding Style	10 points
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)
Late submission	5 points per day

Other notes about grading:

- If you do not follow the directions (e.g. importing classes without permission, 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.
- Regrade Requests and Resubmissions: See syllabus and the announcement on D2L after the grades are released.