Project 2

Due Date: Friday 19 March 2021 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 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!

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 Part 1, you will implement four sorting algorithms that take advantage of locality awareness in the input array. In Part 2, you will sort a grid.

Note: All the sorting here will be from smallest to largest.

Part 1. Locality Aware Sorting

In this part, we consider sorting input integer arrays that are *locality aware*. What we mean by that is that we know the range of possible indexes that a particular element may end up in after the array is sorted. The range is determined by the value *d*.

What is meant by *locality awareness.* Let *A* be an array that is locality aware with *d* = 7. Let *x* = *A[i]* in the original array. Let *f* be the index of *x* in the final, sorted array. According to the property of locality awareness, $i - d \leq f \leq i + d$. Note that these possible indexes must also be bound by the bounds of the array, but the point is that any element can move up to *d* places away from its original position when the array is sorted.

Example. The following array is locality aware with *d = 3.*
`[3, 0, 4, 1, 5, 2, 9, 7, 8, 6]`
When sorted, the array becomes:
`[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`
At most, each element moved 3 positions.

Locality Aware Sorting. If you know that an input array has this property, you should be able to take advantage of that in order to sort the array more efficiently. In this section, you will implement four sorting methods (based on the sorting methods we covered in class) that take advantage of the locality awareness. For each one, you will need to think about each sorting method and how it would change if you know the value of *d*. The sections that follow are meant to give you some guidelines and help you think through the problem.

Locality-Aware Selection Sort. In regular Selection Sort, for each position in the array, we loop through the rest of the array to find the *minimum* value in order to put it into that position. But if you know that no element will move more than *d* positions, how much of the array do you actually have to check for the next *minimum?* **The runtime for this implementation should be no worse than O(dN).**

Locality-Aware Insertion Sort. In regular Insertion Sort, we consider elements from index 1 to the end of the array and insert them into the sorted array to the left. This means that in the worst case (if the array is sorted in the opposition direction), we may have to swap elements $1 + 2 + 3 + ... + N$ times. But if you know that no element will move more than *d* positions, how much of the sorted list do you have to consider when inserting? **The runtime for this implementation should be no worse than O(dN).**

Locality-Aware Heapsort. In regular Heapsort, we turn the array into a heap and then use the sortdown method to pull out the *maximum* value and put it in its final position (at the end of the currently unsorted array. That increases the sorted part and decreases the size of the heap. To take locality awareness into account, you need to decide on a few options. Doing regular Heapsort in place is pretty straightforward. However, doing locality-aware Heapsort in place is more challenging. Because of that, you have the option to do it in place or to use an extra data structure (a heap). *If you do it in place, you can earn up to 5 points of extra credit!*

Guidelines for Heapsort using an extra data structure (not in place):

- The size of your heap should not be bigger than *d+1.*
- The first thing you should determine is what kind of heap you need--min or max? Think of your heap as a kind of sliding window for the elements. What do you do when the window is full? You'll remove an element and put it into the array--but which one and where does that element go? Then what do you do with the next element?
- **The runtime of this implementation should not be worse than O(Nlogd).**

Guidelines for in-place Heapsort:

- If you do this in place, the overall algorithm becomes a little trickier. You can use the sliding window idea, but this time everything in the window should be sorted before you move on to the next window.
- You need to consider how big the window should be.
- You also need to consider how far to "slide" the window after you're done sorting it.
- Since this is done in place, you will not need both *sink* and *swim* because you can do your *heap construction* bottom up using *sink,* which is also used in *sortDown.* To get all the extra credit, you should do it this way, which is more efficient than doing top-down heap construction.
- **The runtime of this implementation should not be worse than** $O(\frac{N}{d} log d)$**.**

Locality-Aware Mergesort. In regular Mergesort, we divide the array into smaller arrays (until we only have 1-element arrays, which are sorted), then merge the smaller arrays together. This is a recursive algorithm. If the array is locality-aware, you don't have to sort the entire array all at once. Instead, you can use the "sliding window" method where you sort part of the array, then slide the window and sort the next set of elements. The questions you need to consider are: How big should the sliding window be? and How much do I slide the window by after sorting?

Guidelines for Mergesort:

- **The runtime for this implementation should not be worse than O(Nlogd).**
- To get full credit here, you should only use one temporary array for the whole method. Do *not* create new arrays each time you call Mergesort.

Other things to consider.

- These methods will be tested on various input sizes, which you can see in the *LocSortTest.java* file. Note that for very large arrays, we are using relatively small *d* values. This should make it possible to run *as long as your implementation is efficient*.
- At least one of the test cases uses a *d* that is greater than *N*. You should handle this case. Think about what that actually means...what should happen in that case?

Overall Guidelines for Part 1.

- Implement your solutions in the *LocSort.java* file. The method signatures and descriptions are provided for you. Do not change the method signatures.
- The *LocSortTest.java* file is provided with test cases. Since the actual data is randomized, these will basically be the same as what is used for grading (though the sizes may change somewhat.) The score given is the *expected* score, but keep in mind that this can change because we do take other things into account when we grade. For example, we look at your code to see if you followed the instructions and if your solution fits within the expected runtime.
- You are also provided with *ArrayGen.java*, which is a class that helps produce different kinds of arrays, including locality-aware ones. This is used in the tests, but you can also utilize it to create specific test cases as you are working on your project.

Part 2. Sorting a Grid

In this part, you will be sorting a Grid.

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. You can get full credit just for accuracy. You may get *extra credit* points depending on the efficiency of your solution. Efficiency is determined by the number of *grid accesses* which are counted through the *Grid.java* class. If you attempt to "trick" the grader into thinking your solution is more efficient than it really is, you will not get any credit and may face consequences related to academic dishonesty, so make sure you follow the guidelines below.

Implementation Requirements (regardless of whether you go for the extra credit or not).

- **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 in the grid. That is, the only thing it can be used for is for the *merge* process and copying the items back to the Grid.**
- **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 itself.**

Note: The idea of this project is to think about the sorting methods and how you might adjust them to work better for the Grid. It turns out that a straight implementation of an efficient method may not work as well on the Grid as it would if you made some alterations to it that take the Grid into account.

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, but we will only be testing this method on Grids of integers. Take some time to look through both classes before starting your implementation.

You will not be submitting Grid.java or Loc.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 versions of Grid.java and Loc.java (which is what is provided to you).

Put your implementation in *SortGrid.java*, which is provided for you with a method signature for the method you must implement.

Some useful things.

- You should be very careful about looking at code as it is *not permissible* to copy code and when you look at actual implementations of things, it is hard to draw the line between looking for understanding and actually copying.
- Instead, look at pseudocode. There has been lots of pseudocode provided throughout this course so far that may be useful to you in this project.
    - Insertion Sort: HW 1 Problem 4 & Project 1 Part1D
    - Quicksort: HW 2 Problem 2 (also the *pivot* method is shown in Project 1 Part1C)
    - Bottom-up Heap Construction: Project 1 Part1E (Note that *foo* is *sink*).
    - Merge: Problem Set #3 Question 2
    - Other pseudocode provided in lecture slides.

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 csc345-spring21-p2 LocSort.java SortGrid.java
```

Upon successful submission, you will see this message:

```
Turning in:
        LocSort.java -- ok
        SortGrid.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 rubrica give the basic breakdown of your grade for this Project.

Baseline

Item	**Points**
Locality-aware sorting tests	80
SortGrid: Correctness	20
Total	**100**

Possible Deductions

Item	**Max Deduction Possible**
Bad Coding Style	5 points
Inefficient Solution	25% of total points for that part
Not following instructions	100 points
Not submitted correctly	100 points
Does not compile	100 points

Possible Extra Points

Item	**Max Extra Points Possible**
Doing Heapsort in place	5 points
Efficient Solution for SortGrid	10 points

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 25% point deduction. All regrade requests should be submitted according to the guidelines in the syllabus and within the allotted time frame.