Homework 4: Sorting

CS 1332 Section C

Fall 2021

1 Important

There are general homework guidelines you must always follow. If you fail to follow any of the following guidelines, you risk receiving a **0** for the entire assignment

1. All submitted code must compile under **JDK 11**. This includes unused code, so don't submit extra files that don't compile. Any compile errors will result in a 0.
2. Do not include any package declarations in your classes.
3. Do not change any existing class headers, constructors, instance/global variables, or method signatures. For example, do not add **throws** to the method headers since they are not necessary.
4. Do not add additional public methods. You may create helper methods, but any helper method you create should be **private**.
5. Do not use anything that would trivialize the assignment. (e.g. don't import/use java.util.ArrayList for an ArrayList assignment. Ask if you are unsure.)
6. Always consider the efficiency of your code. Even if your method is *O(n)*, traversing the structure multiple times is considered inefficient unless that is absolutely required (and that case is very rare).
7. You must submit your source code - the .java files. Do not submit compiled code - the .class files.
8. Only the last submission will be graded. Make sure your last submission has all required files. Resubmitting voids prior submissions.

2 Style and Formatting

It is important that your code is not only functional, but written clearly and with good programming style. Your code will be checked against a style checker. The style checker is provided to you and it located on Canvas. A point is deducted for every style error that occurs. Please double check before you submit that your code is in the appropriate style so that you don't lose any unnecessary points!

2.1 Javadocs

Javadocs should be written for any private helper methods that you create. They should follow a style similar to the existing javadocs on the assignment. Any javadocs you write must be useful and describe the contract, parameters, and return value of the method. Random or useless javadocs added only to appease checkstyle will lose points.

2.2 Vulgar/Obscene Language

Any submission that contains profanity, vulgar, or obscene language will receive an automatic zero on the assignment. This policy applies to all aspects of your code, such as comments, variable names, and javadocs.

2.3 Exceptions

When throwing exceptions, you must include a message by passing in a String as a parameter. **The message must be useful and tell the user what went wrong.** "Error", "Oof - Bad things are happening", and "FAIL" are *not* good messages. Additionally, the name of an exception itself is not a good message.

In addition, you may not use try catch blocks to catch an excpetion unless you are catching an exception you have explicitly thrown yourself with the **throw new ExceptionName('Exception Message");** syntax

2.4 Generics

If available, use the generic type of the class; do not use the raw type of the class. For example, use new **LinkedList<Integer>()** instead of **new LinkedList()**. Using the raw type of the class will result in a penalty.

3 Forbidden Statements

You may not use any of the following in your code at any time in CS 1332. If you are not sure whether you can use something, and it is not explicitly listed here, just ask. Debug print statements are fine, but should be either removed or commented out prior to submission. If print statements are left in, assignments are messy to grade, and checkstyle points will be deducted.

- package
- System.arraycopy()
- clone()
- assert()
- Arrays class
- Thread class
- Collections class
- Collection.toArray()
- Refleciton APIs
- Inner or nested classes
- Lambda Expressions
- Method References (using the :: operator to obtain a reference to a method)

4 JUnits

We have provided a **very basic** set of tests for your code. These tests do not guarantee the correctness of your code, nor do they guarantee you any grade. You may additionally post your own set of tests for others to use on the Georgia Tech GitHub as a gist. Do **NOT** post your tests on the public GitHub.

5 Deliverables

You must submit all of the following file(s) to the corresponding assignment on Gradescope. Make sure all file(s) listed below are in each submission, as only the last submission is graded. Make sure the filename(s) matches the filename(s) below, and that *only* the following file(s) are present.

1. Sorting.java

5.1 Sorting

For this assignment you will be coding 5 different sorts: insertion sort, cocktail sort, merge sort, LSD radix sort, and quick sort. In addition to the requirements for each sort (except LSD radix sort), to test for efficiency, we will be looking at the number of comparisons made between elements while grading.

For each of the sorting algorithms, you may assume that the arrays / lists you are sorting will not contain null elements. You should also assume that arrays may contain any number of duplicate elements.

Your implementations must match what was taught in lecture and recitation to receive credit. Implementing a different sort or a different implementation for a sort will receive no credit even if it passes comparison checks.

5.1.1 Comparator

Each method (except LSD radix sort) will take in a Comparator and use it to compare the elements of the array in various algorithms described below and in the sorting file. You **must** use this Comparator as the number of comparisons performed with it will be used when testing your assignment. See the Java API for details about how the Comparator works and the meaning of the returned value.

5.1.2 Generic Methods

Most of the assignments for this class so far have utilized generics by incorporating them into the class declaration. However, the rest of the assignments will have you implement various algorithms as static methods in a utility class. Thus, the generics from here on will use generic methods instead of generic classes (hence the <T>in each of the method headers and javadocs). This also means any helper methods you create will also need to be static with the same <T>in the method header.

5.1.3 In-Place Sorts

Some of the sorts below are in-place sorts. This means that the items in the array passed in **should not** get copied over to another data structure. Note that you can still create variables that hold only one item; you cannot create another data structure such as an array or list in the method.

5.1.4 Adaptive Sorts

Some of the sorts below are adaptive sorts. This means that the algorithm takes advantage of existing order in the input array. The algorithm can detect existing order in the input array and optimize its performance based on that order.

5.2 Insertion Sort

Insertion sort should be in-place, stable, and adaptive. It should have a worst case running time of $O(n^2)$ and a best case running time of $O(n)$.

Note that, for this implementation, you should sort from the beginning of the array. This means that after the first pass, indices 0 and 1 should be relatively sorted. After the second pass, indices 0-2 should be relatively sorted. After the third pass, indices 0-3 should be relatively sorted, and so on.

5.3 Cocktail Sort

Cocktail sort should be in-place, stable, and adaptive. It should have a worst case running time of $O(n^2)$ and a best case running time of $O(n)$. **Note: Implement cocktail sort with the optimization where it utilizes the last swapped index.** Remembering where you last swapped will enable some optimization for cocktail sort. For example, traversing the array from smaller indices to larger indices, if you remember the index of your last swap, you know after that index, there are only the largest elements in order. Therefore, on the next traversal down the array, you start at the last swapped index, and on the next traversal up the array, you stop at the last swapped index. Make sure that both on the way up and on the way down, you only look at the indices that you do not know are sorted. Do not make extra comparisons.

Note: a single iteration for this sort is considered as a traversal up **and** down the backing array.

5.4 Merge Sort

Merge sort should be out-of-place, stable, and not adaptive. It should have a worst case running time of $O(nlogn)$ and a best case running time of $O(nlogn)$. When splitting an odd size array, the extra data should go to the **right** subarray to preserve stability.

5.5 LSD Radix Sort

LSD Radix sort should be out-of-place, stable, and not adaptive. It should have a worst case running time of $O(kn)$ and a best case running time of $O(kn)$, where k is the number of digits in the longest number and n is the number of data in the structure. You will be implementing the least significant digit version of the sort. You will be sorting ints. Note that you CANNOT change the ints into Strings at any point in the sort for this exercise. The sort **must** be done in base 10. Also, as per the forbidden statements section, you cannot use anything from the Math class besides Math.abs(). However, be wary of handling overflow if you use Math.abs()! Think about the edge case we had to consider with HashMaps when using Math.abs().

5.6 Quick Sort

Quick Sort should be in-place, unstable, and not adaptive. It should have a worst case running time of $O(n^2)$ and a best case running time of $O(nlogn)$. Your implementation must be randomized as specified in the method's javadocs.

6 Grading

Here is the grading breakdown for the assignment. There are various deductions not listed that are incurred when breaking the rules listed in the PDF and in other various circumstances. Note that for this assignment efficiency is explicitly mentioned as contributing to your point total. In contrast to past assignments, unimplemented methods will result in losing efficiency points.

- Sorting − 80 pts
- Efficiency − 10 pts
- Checkstyle − 10 pts
- Total: 100 pts