CSCI241, Winter 2019, Assignment 1
Due January ~~25th~~ 28th at 9:59pm

Your submission for this and all future homework assignments, must be your own work. You may discuss topics and concepts at a high level and brainstorm using a white board, but you cannot share, disseminate, co-author, or even view, another student's code. Please refer to the academic honesty guidelines on the syllabus for more details. If any of this is unclear, please ask for further clarification.
If you rely on any external resources (e.g., the internet, other textbooks, etc.), you MUST cite those resources in the acknowledgements section of your writeup. Under no circumstances may you cut-and-paste entire blocks of code from the internet, other current or past students, or anywhere else—if you do, you will receive an F in the course and be reported to the Dean of Students.

1 Overview

In this assignment, you will implement four sorting algorithms, an interactive command-line program that demonstrates the sorting algorithms, and create a write-up analyzing the number of comparisons performed by each sort as a function of input array size.

Your primary tasks are as follows:

- Implement the methods for insertion, merge, quick, and radix sorts in Sorts.java. You will also need to implement the merge and partition helper methods for merge sort and quick sort, respectively.
- Implement the user-facing behavior described below in SortsDriver, using the sorting methods from Sorts.java to perform the sorting.
- Create a writeup analyzing how number of comparisons grows as a function of input array size for insertion, merge, and quicksort.
- (Optional) Complete any of the optional enhancements described below (or come up with your own) and explain what you did in your writeup.

2 Getting Started

The Github Classroom invitation link for this assignment is in Assignment 1 on Canvas. Begin by accepting the invitation and cloning a local working copy of your repository as you did in Lab 1. Make sure to clone it somewhere outside your lab1 working copy (e.g., `~/csci241/a1`) to avoid nesting local repositories. Skeleton code is provided in your repository to get you started.

See Section 7 below for a suggested game plan for getting the sorts, user interface, and writeup done. The following sections provide details and hints on each subtask.

3 Sorting Algorithms

Sorts.java contains method headers for six public methods:

- `insertionSort`
- `merge`
- `mergeSort`
- `partition`
- `quickSort`
- `radixSort`

The method headers and specifications (i.e., the name, return type, parameters, and the Javadoc comment specifying the method's behavior **should not be changed**. If you change method names, call signatures, or return values, your code will not compile with the testing system and you'll receive no credit for the correctness portion of your grade.

Public methods *must* implement their specifications precisely and completely. For example, even if your quickSort method always calls `partition` using the first element as the pivot, `partition` is still required to work with any other legal pivot index specified, because such behavior is prescribed in the specification.

In Lab 2, you will write unit testing code that will help you check the correctness of the sorting methods. As you develop the sorts, you should run `gradle test` frequently and make sure that each algorithm passes all its tests before moving on to the next.

3.1 Implementation Notes

- You may write and use as many `private` helper methods as you need.
- The mergeSort and quickSort implementations must be recursive and all sorts must be asymptotically efficient.
- The Sorts class has a private member `comparisonCount`. In each of the sorts that you implement, use this counter to tally the count of comparisons that are done between entries of the array as it is sorted. For example, for insertionSort, tally the number of times that `array[j]` $<$ `inputArr[j-1]` is performed. For quickSort, tally the number of times that `A[j]` is compared to the pivot, etc. Be sure to count all comparisons made, not just those that evaluate to true. You do not need to count comparisons among loop variables (e.g., you should ignore the i ¡ n comparison in a for loop header).
- The bottom of Sorts.java has two private helper methods written for you that you may find useful.
- Radix sort requires the use of a stable sorting algorithm to sort the array on each digit. You can either use counting sort (see CLRS 8.2) or maintain a list of queues, one to store the contents of each bucket. Counting sort is algorithmically trickier. On the other hand, creating an array of queues of integers in Java can be a bit painful because of the way generics and arrays interact. The following snippet creates and populates an ArrayList of 10 buckets, each of which is a LinkedList of integers:

```
ArrayList<LinkedList<Integer>> buckets = new ArrayList<LinkedList<Integer>>(10);
for (int i = 0; i < 10; i++) {
  buckets.add(new LinkedList<Integer>());
}
```

Because buckets is an `ArrayList`, use `buckets.get(i)` to get the LinkedList storing the i'th digit. Remember that a LinkedList implements the Queue interface; see the Java documentation for details on which methods make it behave like a Queue.

- Radix sort as described in class does not naturally play well with negative integers. Get it working on nonnegative numbers first, then figure out how to handle negatives. You may assume that the values to be sorted are not extremely large or small and do not approach the largest or smallest value that can be represented using an `int`.

4 Interactive Program Behavior

The main method of SortsDriver should implement a program that behaves as follows. To run the program, you can simply use `gradle run`. When the program starts, it should:

1. Prompt the user for the size of the array, n, and create an array of that size made up of integer values chosen randomly from `-n..n+1`.
2. Prompt the user to specify which sort to use (merge sort, quick sort, insertion sort, radix sort, or all). The user should be asked to enter a single letter: m, q, i and r, or a.
3. If all (a) sorts is specified, the input to each sort must be identical
4. If $n \leq 20$, the pre-sorted and sorted array's contents are printed for each sort invoked
5. If $n > 20$, the pre-sorted and sorted array's contents are NOT printed for each sort invoked
6. The count of comparisons performed is/are output.

Several sample invocations of the program are shown in Figure 1. Note the order of the prompts must be as specified, though the text does not need to be precisely the same as the example.

4.1 Implementation Notes

- Error catching is NOT required. You do not have to catch if a user specifies a negative count of entries, or inputs a letter, or provides a sort option that is not m, i, q, r nor *all*. Consider using a `switch` statement.
- The `java.util.Random` and `java.util.Scanner` classes from the Java Standard Library may come in handy.
- Do not use System.console to read user input.
- Do not create more than one `Scanner` object reading from `System.in`. Re-use the same Scanner object for all user input.
- To ensure that the *all* option works as intended, you'll need to make a deep copy of the randomly generated array to give to each sort method.

```
  % java SortsDriver
Enter sort (i[nsertion], q[uick], m[erge], r[adix], a[ll]): q
Enter n (size of array to sort): 15
Unsorted: [ -12   5  -9  15  -7  13   9  13  10  -7  14 -14 -14   2 -10 ]
  Sorted: [ -14 -14 -12 -10  -9  -7  -7   2   5   9  10  13  13  14  15 ]
Comparisons: 42

  % java SortsDriver
Enter sort (i[nsertion], q[uick], m[erge], r[adix], a[ll]): m
Enter n (size of array to sort): 1000
Comparisons: 8709

  % java SortsDriver
Enter sort (i[nsertion], q[uick], m[erge], r[adix], a[ll]): r
Enter n (size of array to sort): 12
Unsorted: [   4   1  -8   5  -2  12  -7  -4 -11  -7  -5  -8 ]
  Sorted: [ -11  -8  -8  -7  -7  -5  -4  -2   1   4   5  12 ]
Comparisons: 0

  % java SortsDriver
Enter sort (i[nsertion], q[uick], m[erge], r[adix], a[ll]): a
Enter n (size of array to sort): 10
Unsorted: [  -9   9  -6  -8  -4   0  -7  -3   6   8 ]
insertion: 13
Sorted: [  -9  -8  -7  -6  -4  -3   0   6   8   9 ]

quick: 21
Sorted: [  -9  -8  -7  -6  -4  -3   0   6   8   9 ]

merge: 23
Sorted: [  -9  -8  -7  -6  -4  -3   0   6   8   9 ]

  % java SortsDriver
Enter sort (i[nsertion], q[uick], m[erge], r[adix], a[ll]): a
Enter n (size of array to sort): 1000
insertion: 248189

quick: 10453

merge: 8695
```

Figure 1: Sample Invocations of SortDriver.java

- For the *all* option, avoid counting comparisons for multiple sorts: either reset the comparison counter (there's a handy method for this provided in Sorts.java) or create a fresh Sorts object for each sort.
- Precise comparison counts may differ based on subtle implementation choices, even across multiple correct solutions. However, the relative counts between insertion sort $O(n^2)$ and quick sort $O(n \log_2 n)$, for example, should differ greatly and clearly demonstrate their relative run-times.
- As described in the style guide on the syllabus and the rubric at the end of this document, overly long methods (e.g., with hundreds of lines of code) are considered bad programming style. Be sure that your program is broken down into sensible, modular helper methods, rather than a monolithic main method.

5 Writeup

Your writeup should include the following:

- **Table**. For the insertion, quick, and merge sorts, run each of them on successively larger arrays, and tally the count of comparisons made as a function of n, the array size. Vary n from 10 to 500 and list the data in a table. See Figure 2 as an example. Notice that radix sort is not listed here; you be able to explain why this is the case.

n	Insertion	Quick	Merge
10	33	21	24
50	607	234	222
100	2459	550	528
200	11353	1401	1282
500	58535	4682	3852

Figure 2: Sample tally table

- **Plot**. Using the software of your choice, create a plot of the values you have tallied. Be sure to label both the x and y axes, provide a title, and make sure that you provide a legend to make it easy to identify merge, insertion, and quick sort performance. A sample plot is shown in Figure 3.
- **Discussion**. Describe any design decisions you made along the way. If you complete any of the optional enhancements (see below), describe what you did and include any associated plots or analysis.
- **Acknowledgements** If applicable, list any people you collaborated with and any external resources (i.e., anything other than the recommended texts, lecture slides, and help in office hours) you used in the course of completing this assignment

6 Optional Enhancements

The base assignment is worth 45/50 points. The final 5 points may be earned by completing one or more of the following optional enhancements. You may also come up with your own ideas - you may want to run them by the instructor to make sure they're worthwhile and will

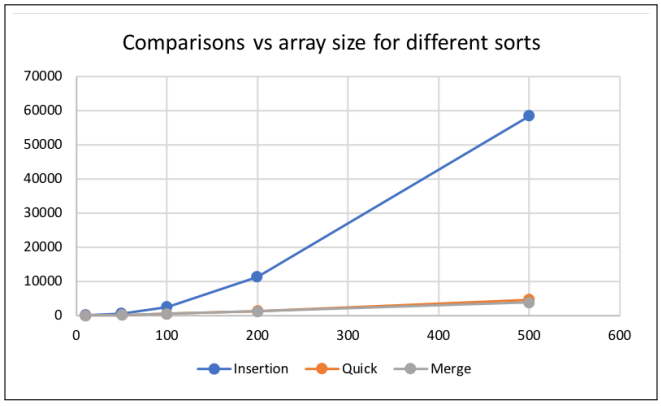

Figure 3: Sample plot of run-times for merge, insertion and quicksort

result in points awarded if successfully completed. It is highly recommended that you complete the base assignment before attempting any enhancements.

Enhancements and git The base project will be graded based on the master branch of your repository. Before you change your code in the process of completing enhancements, create a new branch in your repository (e.g., `git checkout -b enhancements`). Keep all changes related to enhancements on this branch—this way you can add functionality, without affecting your score on the base project. For example, the first enhancement asks you to add a third user prompt to choose between a sorted array and a random array. As this departs from the user-facing behavior described in the base project, such a change should only be made in your enhancements branch. Make sure you've pushed both master and enhancements branches to GitHub before the submission deadline.

1. (1 point) Real-world sorting inputs rarely come in uniformly random order. Add a prompt that allows the user to choose among the following arrays that try to model some likely real-world use cases:

   - A random array (as in the base project)
   - A fully sorted array
   - An array that is sorted, except the last 5% of its values are random
   - An array whose elements are sorted with probability 0.9, otherwise random. Tip: the `java.util.Random` class's `nextDouble()` method generates a random floating-point value between 0.0 and 1.0.

   Generate another plot for each of these types of arrays. Describe the differences—which ones can you explain? Are any surprising/inexplicable?

2. (2 points) Make a table and plot of performance in terms of elapsed time instead of number of comparisons. You may find the built-in `System.nanoTime()` function useful.

3. (2 or 3 points) Implement the median-of-three (first, middle, last) pivot in quicksort. Plot the number of comparisons done by both variants of quickSort and insertionSort. (1 additional point) Repeat this experiment, but run the sorts on sorted arrays and nearly-sorted arrays from Enhancement 1.

4. (Up to 4 points) Most real-world sorts built into modern programming languages are hybrid algorithms that combine more than one algorithm depending on the array size,

ordering, etc. Implement a hybrid sorting algorithm. Try to outperform both quicksort and insertionsort on random, sorted, and mostly-sorted arrays. You may search the internet for inspiration and strategies, but please cite your sources, write your own code, and explain your algorithm in the writeup. A good-faith attempt that does not beat insertion and quicksort may still receive some credit.

7 Game Plan

Start small, test incrementally, and git commit often. Please keep track of the number of hours you spend on this assignment; when you are finished, write the approximate number of hours you spent as a lone integer in hours.txt and make sure it is included when you push your final changes to github to submit. Hours spent will not affect your grade.

A suggested approach is outlined below:

1. Implement the `insertionSort` method in `Sorts.java`. Use the skeleton code provided for you in `SortsDriver.java` test out the insertionSort method to sort on a small fixed array. When you have this working, commit your changes to git.

2. In SortsDriver, implement and test random array generation. Prompt the user for array size and which sort, and add functionality to print the array before and after sorting.

3. Implement each of the remaining sorts, testing using both the driver and the unit tests as you go. Commit your changes git frequently, and push to github each time you get a new piece working. Recommended order: merge, quick, radix.

4. Implement the *all* option. Avoid copy/pasting code you've already written—instead, factor useful pieces into private methods that you can call more than once. Commit your changes.

5. Create the table and plot in your writeup. Don't forget axis labels, legend, and title.

6. If you plan to do any optional enhancements, create an `enhancements` branch in your repository to track the changes beyond the base project (`git checkout -b enhancements`). Commit all new changes to this branch and **don't** merge it into the master branch. Add a description and analysis of each enhancement to your writeup.

7. Fill in hours.txt, run the tests one last time, and read through the rubric to make sure you've finished everything. Read through the code clarity section to be sure you won't lose style points; see the syllabus for more details on coding style.

8 How and What to Submit

Submitting the assignment is as simple as pushing your final changes to GitHub (`git push origin master` or just `git push`) before the deadline. Don't forget that committing changes does not automatically push them to GitHub! Submit the writeup with runtime tallies, plot, and description of any enhancements in pdf format on Canvas.

Rubric

For the coding portion of each assignment, you earn points for the correctness and efficiency of your program. Points can be deducted for errors in commenting, style, clarity, etc.

Hours	
Code is pushed to github and hours spent appear as a lone integer in hours.txt	1 point
Writeup	
Tally table of comparison counts for quicksort, merge sort, and insertion sort.	3
Plot of tallied numbers, including axis labels, title, and legend.	3
Code : Correctness	
Sorting algorithms and helper methods are implemented correctly as determined by unit tests (1 point per test).	20
Program prompts user for number of integers desired, relies on random number generator to populate the array, and prompts for type of sort to run (m, i, q, and r, a).	3
Each invocation of a sort correctly tallies the count of comparisons made.	3
When the a option is specified, all four sorts are invoked, whose input is the **same** array.	2
If $n \leq 20$, pre-sorted and sorted array(s) are printed; otherwise, the arrays are not printed.	2
Code : Efficiency	
Mergesort runs in O(n log n)	2
QuickSort runs in O(n log n) in the expected case	2
Insertion sort runs in-place, and runs O(n^2).	2
Radix makes a constant number of O(n) passes over the input.	2
Enhancements	
See Optional Enhancements section above.	up to 5
Clarity deductions (up to 2 points each)	
Include author, date and purpose in a comment comment at the top of each file you write any code in	
Methods you introduce should be accompanied by a precise specification	
Non-obvious code sections should be explained in comments	
Indentation should be consistent	
Method should be written as concisely and clearly as possible	
Methods should not be too long - use private helper methods	
Code should not be cryptic and terse	
Variable and function names should be informative	
Total	50 points