

CSCI241 Fall 2020 Assignment 1

Due: Friday, October 16th 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 with your fellow classmates, 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 clarification.

If you rely on any external resources (e.g., the internet, other textbooks, etc.), you **MUST** cite those resources in the open-ended question of the A1 Survey and an acknowledgements section of your writeup, if you're submitting one. Under no circumstances may you cut-and-paste entire blocks of code from the internet, other current or past students, or anywhere else. Plagiarizing code and allowing your code to be plagiarized are both academic honesty violations that are handled through the formal university process, resulting in an F in the course and a record of the violation on file with the registrar.

1 Overview

In this assignment, you will implement four sorting algorithms, and an interactive command-line program that demonstrates the sorting algorithms. For extra credit, you may also perform experiments to measure and analyze their runtime.

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.

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 everything done in plenty of time. 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 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 .
2. 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 $-n \dots n+1$.
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 printed.

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 check whether a user specifies a negative count of entries, or inputs a letter, or provides a sort option that is not one of the valid options (i , q , m , r , a). Consider using a `switch` statement.
- The `java.util.Random` and `java.util.Scanner` classes from the Java Standard Library may come in handy.
- Don't use `System.console` to read user input.
- Don't 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.

```

$ gradle run -q
Enter sort (i[nsertion], q[uick], m[erge], r[adix], a[ll]): i
Enter n (size of array to sort): 15
Unsorted: [ 5 -14 5 -11 -15 -11 14 -8 4 9 -6 11 15 2 1 ]
Sorted: [ -15 -14 -11 -11 -8 -6 1 2 4 5 5 9 11 14 15 ]
Comparisons: 37

$ gradle run -q
Enter sort (i[nsertion], q[uick], m[erge], r[adix], a[ll]): q
Enter n (size of array to sort): 1000
Comparisons: 9662

$ gradle run -q
Enter sort (i[nsertion], q[uick], m[erge], r[adix], a[ll]): r
Enter n (size of array to sort): 12
Unsorted: [ 3 -11 5 -7 3 10 -9 -6 -7 8 -12 3 ]
Sorted: [ -12 -11 -9 -7 -7 -6 3 3 3 5 8 10 ]
Comparisons: 0

$ gradle run -q
Enter sort (i[nsertion], q[uick], m[erge], r[adix], a[ll]): a
Enter n (size of array to sort): 10
Unsorted: [ -3 3 -1 -4 0 -1 5 -9 8 -7 ]
insertion: 22
Sorted: [ -9 -7 -4 -3 -1 -1 0 3 5 8 ]
quick: 19
Sorted: [ -9 -7 -4 -3 -1 -1 0 3 5 8 ]
merge: 23
Sorted: [ -9 -7 -4 -3 -1 -1 0 3 5 8 ]
Sorted: [ -9 -7 -4 -3 -1 -1 0 3 5 8 ]
radix: 0

$ gradle run -q
Enter sort (i[nsertion], q[uick], m[erge], r[adix], a[ll]): a
Enter n (size of array to sort): 1000
insertion: 256539
quick: 9331
merge: 8703
radix: 0

```

Figure 1: Sample Invocations of SortDriver.java. The `-q` flag simply tells Gradle to suppress output about the tasks it's running.

- 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 Enhancements

You can earn up to 5 points of extra credit by completing one or more of the following enhancements. You may also come up with your own ideas, but you should probably run them by me to make sure they're worthwhile and will 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, prerequisite to further enhancements) Create a writeup containing a table and graph plotting the comparisons performed by each sorting algorithm as a function of input size; I recommend plotting comparisons for input sizes ranging from about 10 to about 200. Be sure to label your graph's axes and provide a title and legend. If you completed further enhancements, be sure to include a description of what experiments you ran and any design decisions you made, alongside plots and tables reporting your results.
2. (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 in which a randomly-chosen 90% of elements are, amongst themselves, in sorted order, while the other 10% are not sorted (e.g., have random values). 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?

3. (1 point) 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.
4. (1 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. Repeat this experiment, but run the sorts on sorted arrays and nearly-sorted arrays.
5. (Up to 2 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 and analyze its performance relative to the other sorts. You may find it interesting to note differences in performance measured by number of comparisons vs elapsed time. 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.

6 A1 Survey

After pushing your final changes to github, you will complete your submission by filling out the **A1 Survey** quiz on Canvas. At a minimum, you need to provide an estimate of the number of hours you worked on this assignment and report any collaborators or external resources you used. You can also let me know how the assignment went in the open-ended question. The number of hours and any other comments will not affect your grade, but **your assignment will not be considered submitted until you submit the survey**. If the survey is not filled out by the deadline, we will assume that you are planning to submit late and will not grade your submission.

7 Game Plan

Start small, test incrementally, and git commit often. Please keep track of the number of hours you spend on this assignment so you can report it in the A1 Survey.

Here's a suggested timeline for completing this project on a low-stress schedule.

1. By 10/2: Implement insertion sort.
2. By 10/5: 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. You can implement the `i` option, or simply hard-code the driver to perform insertion sort for now and implement the options later when the rest of the sorts are completed.
3. By 10/7: If you haven't finished in lab, complete your implementation of the test helper methods and make sure insertion sort is bug-free.
4. By 10/9: Implement the `merge` helper method (you should pass tests 10–11) and `mergeSort` (should pass tests 12–14).

5. By 10/11: Implement the `partition` helper method (you should pass tests 20–22) and `quickSort` (should pass tests 23–24).
6. By 10/13: Implement `radixSort` under the assumption that all numbers are negative and the maximum number of digits is known. At this point your code should pass tests 30–32. Next, generalize your code to handle negative numbers so that your code passes tests 33–34.
7. By 10/15: Finish the `SortsDriver` functionality: implement the individual sort options (`i`, `m`, `q`, `r`), then implement the *all* option. Try to avoid copy/pasting code you’ve already written—instead, factor useful pieces into private methods that you can call more than once.
8. By 10/16: complete any extra credit enhancements you plan to do in an `enhancements` branch in your repository. Commit all new changes to this branch and **don’t** merge it into the master branch. Include a description and analysis of each enhancement in a writeup and upload it to Canvas.
9. By 10/16: 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.
10. By 10pm on 10/16: Fill out the A1 Survey to complete your submission.

8 How and What to Submit

To submit the assignment, push your final changes to GitHub (`git push origin master` or just `git push`) before the deadline, then complete the **A1 Survey** on Canvas. Don’t forget that committing changes does not automatically push them to GitHub! If you’re submitting a writeup, upload it to Canvas in PDF format.

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.

Submission	
Code is pushed to github and A1 Survey is submitted with the number of hours worked	1 point
Code : Correctness	
Sorting algorithms and helper methods are implemented correctly as determined by unit tests (1.5 points per test).	30
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, with the same input array values.	3
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
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