CSCI 2720 – Fall 2025

Programming Assignment 4: Sorting

Due by 11:59 PM on November 23rd, 2025 on Odin

You can complete this assignment individually or as a group of two students.

1 Project Setup

Your source code should be kept under a package `cs2720.p4`. You are free to setup your project using Maven (recommended) or by creating the directory structure manually. Regardless, you must briefly document how to compile and run your project in your README file.

2 Project Description

For this project, you will implement Selection Sort, Merge Sort, Heap Sort, and QuickSort in Java, and you will write a report analyzing these functions. **For QuickSort, you should provide two implementations.** One implementation should use the last element of the array as the pivot. The second implementation should use a random pivot. **The sorting algorithms should sort an array of integers in ascending order.**

Start early on this assignment, since you need to submit a report along with your code. Note that most of the code is already provided in the slides/textbooks and you are free to use that code (just cite it in your README file). The only thing you need to do is to calculate the number of comparisons each algorithm makes. That is, you need to count when you compare two "data elements" in the array, not the comparison for "i" in a loop.

3 Functional Requirements

A functional requirement is *added* to your point total if satisfied.

1. You can complete this assignment by writing your own classes with the required data members and methods. You have full freedom on how you want to build your classes.
   At a minimum, you need to use an integer array to store the input and pass this integer array to all sorting algorithm methods. **You don't need to use an "ItemType" class or generics in this assignment.**
   - If you would like a more fun application of object-oriented principles, you can consider using something like the Strategy design pattern to represent sorting algorithms as interchangeable objects instead of methods that need to manually be called.
2. At least two different sorting algorithms should be written recursively (e.g., Merge and Quick, Heap and Merge, etc.).

3. When the program is run (with input filename as a command line argument as shown below), it should ask for the sorting algorithm to be used. Sorted results should then be generated using that algorithm.
4. Unlike the previous assignments, your program should terminate after the sorting is done. In other words, **do not** ask to enter the algorithm again in a loop.
5. In each of the sorting functions, you should embed statements to count the total number of comparisons used by the function for sorting the numbers. You can use a `long` variable to count comparisons. After sorting is completed, you should print this count exactly as shown in the example output.
   - Note that your exact counts may differ from ours slightly, but your counts should not be extraordinarily different.
6. For this assignment you need to do two sets of experiments and prepare a report which will include results, plots, and discussion from these two experiments. The first experiment is explained in Point 9, where you just need to find out the number of comparisons for different sorting algorithms for different types of input files.
   The second experiment is explained in Points 10 and 11, where you need to draw a plot of input size vs. number of comparisons for different sorting algorithms and then verify that the plots obtained by your experiments match with the theoretical results. You need to prepare a report summarizing all your results, plots, discussion, and conclusion from these two experiments.
7. You are given 3 input files for the first experiment on eLC.
   (a) Ordered file — Containing integers placed in ascending order from 0 to 9999
   (b) Random file — Containing 10000 integers arranged in a random order
   (c) Reversed file — Containing integers placed in reverse order from 9999 to 0
8. Here is a small example that shows what to expect when you run these algorithms on the different input files. If the input size is 10,000, for an algorithm with n^2 complexity, n^2 is going to be 100,000,000. Expect 8-9 digits in the number of comparisons.
   Similarly for an algorithm with $n \log n$ complexity, expect 5-6 digits in the number of comparisons. For an $n \log n$ algorithm, if you are getting 8-9 digits in comparisons that will indicate a problem in your comparison calculation or algorithm implementation.
9. You can use the same code snippet for reading input from the files as in the previous assignments. Read numbers from a text file, store them in an integer array, and pass this array to your sorting functions.
   Write a well-organized report of your work. Your report must be submitted as a PDF file with your name in the middle of the cover page. Submit your report (one report per group) on eLC and your code on Odin. Your report should include the following points in Experiment 1:
   (a) Provide the total number of comparisons used by each one of the algorithms and explain whether they align with the Big-O time complexity of each algorithm. In the case of QuickSort, compare and comment about the number of comparisons and complexity with the other QuickSort implementation.
      i. For Ordered file as input.
      ii. For Randomized file as input.
      iii. For Reversed file as input.

   Please use the following table format to summarize the content for point (a):

Algorithm	Input Type	# Comparisons	Comments About Time Complexity and # Comparisons
...	...	...	...

   After including this table, answer the following questions in your report:

i. For selection sort, what do you think about the number of comparisons when different input files are used? Are your results consistent with the theoretical values (i.e., do you expect to see the same number of comparisons for all three types of inputs or random input file results should be higher or lower than the sorted input files)?

ii. In the merge sort experiment, the number of comparisons for random input files should be higher (almost double) than the number of comparisons for the sorted input files. Provide an explanation for this observation.

iii. In the heap sort experiment the number of comparisons for all three types of files are almost identical. Provide an explanation for this observation that the comparison numbers are not fluctuating for different input files (ordered, random and reversed) like other algorithms (merge sort and quick sort). Also provide an explanation why the number of comparisons for the ordered file is higher than the number of comparisons for the reversed files.

10. After completing the above program and report, you will do the next set of experiments, where you will draw plots (input size vs no. of comparisons) for sorting algorithms and verify that the experimental results match with the theoretical results.
    You have complete freedom in how you want to implement this part of the assignment. You can create a new `main` method, or add a method to your `Sorting` class or add a method in the `SortDriver.java` file that implements this part of the assignment. Your code should use the algorithms created in `Sorting.java` to find additional insights about the five different sorting algorithms.
    For this part of the assignment you are not reading inputs from the text files; instead, you will generate the inputs within the program itself, and it should calculate the comparison values. Your code should be able to run the five algorithms (mentioned below) with different sizes of the inputs and give the number of comparisons. You can use these comparison values to draw the plots.
    For grading purposes, provide an interface in your implementation where the user can specify which sorting algorithm they want to use and the size of the input that they wish to test. Your program should then generate an array with n number of **random values** (with n being the number specified by the user) and then it should sort those values using the sorting algorithm selected by the user. The number of comparisons should also be printed.
    Please specify in your README file how to compile, run and use your implementation because it is mostly up to you for how you want to implement this part of the assignment. Also note there is no sample output for this part of the assignment.
11. Now, use the implementation for Experiment 2 to create multiple different plots for the report mentioned above. Keep track of your experiments in a table formatted like the following and include it in your report:

	Size of Input							
Algorithm	100	500	1000	5000	10000	20000	25000	30000
Selection								
Merge								
Heap								
Quick-Last[a]								
Quick-Random[b]								
Iter Algo[c]								

[a] QuickSort where Partition chooses the last element as the pivot
[b] QuickSort where Partition chooses a random element as the pivot
[c] See: Point 12

Note: **I recommend running each algorithm multiple times and taking the average result for each input size to use in this table and for your plots. You can also use more**

values of n to make your plot look smoother. However, in the report you just need to show the results for the above table.

(a) You should use some sort of graphing software like Microsoft Excel or Google Sheets to generate plots that compare the size of input n to the number of comparisons for the 6 sorting algorithms. You should have 6 separate plots (one for each sorting algorithm) with input size n on the x-axis and number of comparisons on the y-axis. Include these plots in your report.

(b) You will then provide some discussion about your results. Compare the theoretical result with your experimental result for each algorithm. Does your experimental result coincide with the Big-O of that specific algorithm? Describe if there may be some inconsistencies between your plot and what the theoretical plot looks like.

12. Finally, pick one of the recursive algorithms you wrote (not Selection Sort), and write an iterative implementation. Allow the user to call the iterative or recursive version of this function in your Experiment 2 program. Include your iterative results in the above table, include the appropriate plots in your report, and answer the following questions in your report:

    (a) Which algorithm is better in terms of efficiency? Justify your answer in terms of space complexity and stack frames. Is the runtime complexity consistent?

    (b) Which algorithm was easier to implement? Argue whether or not this should be a determining factor when choosing a recursive implementation or an iterative implementation, and defend your answer.

    (c) Which implementation was easier to understand? Defend your choice by justifying how this implementation would be easier to explain to someone with little experience with sorting algorithms (like a student who just completed CSCI 1301).

Notes:

1. All submissions must be typed. Upload a PDF of your work to the dropbox on eLC. Only one submission per group.
2. The example outputs do not print the complete list of numbers. However, you must print the complete list of numbers in your program. The number of comparisons shown in sample output below may be slightly different than the numbers that you get; however, make sure that you at least get the same number of digits so you know that it is in the correct Big-O class.

Sample Outputs

Blue text is entered by the user.

The series of "..... " in each output is cosmetic. Your implementation should print every single number in the sorted output.

Sample Output 1 (ordered.txt)

```
java SortDriver ordered.txt
selection-sort (s) merge-sort (m) heap-sort (h) quick-sort-last (q)
quick-sort-rand (r)
Enter the algorithm: s
0 1 2 3 4 5 ..... 9999
#Selection-sort comparisons: 49995000
```

```
java SortDriver ordered.txt
selection-sort (s) merge-sort (m) heap-sort (h) quick-sort-last (q)
quick-sort-rand (r)
Enter the algorithm: m
0 1 2 3 4 5 ..... 9999
#Merge-sort comparisons: 69008
```

```
java SortDriver ordered.txt
selection-sort (s) merge-sort (m) heap-sort (h) quick-sort-last (q)
quick-sort-rand (r)
Enter the algorithm: h
1 2 3 4 5 ..... 9999
#Heap-sort comparisons: 122139
```

```
java SortDriver ordered.txt
selection-sort (s) merge-sort (m) heap-sort (h) quick-sort-last (q)
quick-sort-rand (r)
Enter the algorithm: q
1 2 3 4 5 ..... 9999
#quick-sort-last comparisons: 49995000
```

```
java SortDriver ordered.txt
selection-sort (s) merge-sort (m) heap-sort (h) quick-sort-last (q)
quick-sort-rand (r)
Enter the algorithm: r
1 2 3 4 5 ..... 9999
#quick-sort-rand comparisons: 149145
```

Sample Output 2 (random.txt)

```
java SortDriver random.txt
selection-sort (s) merge-sort (m) heap-sort (h) quick-sort-last (q)
quick-sort-rand (r)
Enter the algorithm: s
1 2 3 4 5 ..... 9999
#Selection-sort comparisons: 49995000
```

```
java SortDriver random.txt
selection-sort (s) merge-sort (m) heap-sort (h) quick-sort-last (q)
quick-sort-rand (r)
Enter the algorithm: m
1 2 3 4 5 ..... 9999
#Merge-sort comparisons: 120414
```

```
java SortDriver random.txt
selection-sort (s) merge-sort (m) heap-sort (h) quick-sort-last (q)
quick-sort-rand (r)
Enter the algorithm: h
1 2 3 4 5 ..... 9999
#Heap-sort comparisons: 117687
```

```
java SortDriver random.txt
selection-sort (s) merge-sort (m) heap-sort (h) quick-sort-last (q)
quick-sort-rand (r)
Enter the algorithm: q
1 2 3 4 5 ..... 9999
#quick-sort-last comparisons: 148343
```

```
java SortDriver random.txt
selection-sort (s) merge-sort (m) heap-sort (h) quick-sort-last (q)
quick-sort-rand (r)
Enter the algorithm: r
1 2 3 4 5 ..... 9999
#quick-sort-rand comparisons: 158610
```

Sample Output 3 (reverse.txt)

```
java SortDriver reverse.txt
selection-sort (s) merge-sort (m) heap-sort (h) quick-sort-last (q)
quick-sort-rand (r)
Enter the algorithm: s
1 2 3 4 5 ..... 9999
#Selection-sort comparison: 49995000
```

```
java SortDriver reverse.txt
selection-sort (s) merge-sort (m) heap-sort (h) quick-sort-last (q)
quick-sort-rand (r)
Enter the algorithm: m
1 2 3 4 5 ..... 9999
#Merge-sort comparison: 64608
```

```
java SortDriver reverse.txt
selection-sort (s) merge-sort (m) heap-sort (h) quick-sort-last (q)
quick-sort-rand (r)
Enter the algorithm: h
1 2 3 4 5 ..... 9999
#Heap-sort comparison: 113318
```

```
java SortDriver reverse.txt
selection-sort (s) merge-sort (m) heap-sort (h) quick-sort-last (q)
quick-sort-rand (r)
Enter the algorithm: q
1 2 3 4 5 ..... 9999
#quick-sort-last comparison: 49995000
```

```
java SortDriver reverse.txt
selection-sort (s) merge-sort (m) heap-sort (h) quick-sort-last (q)
quick-sort-rand (r)
Enter the algorithm: r
1 2 3 4 5 ..... 9999
#quick-sort-rand comparison: 155231
```

Note: Make sure to quit the program after printing the number of comparisons after each sort. In other words, do not ask to enter the algorithm again in a loop.

Grading Rubric:

Grade Item	**Grade**
Selection Sort	10%
Merge Sort	10%
Heap Sort	10%
Quick Sort	10%
Driver program follows expectations	10%
Report	50%
Total	**100%**

4 Nonfunctional Requirements

A nonfunctional requirement is *subtracted* from your point total if not satisfied.

4.1 Project Structure

Your implementation should match the provided package structure and visibility of each class and method within reason. Use your best judgment when writing your own classes/methods not explicitly listed in this document. (10–100 points)

4.2 Project Dependencies

Your project may only utilize sorting algorithm implementations covered in class, the listed textbooks, and the course slides. No outside resources may be used (nor do you need them). Clearly indicate where your base implementations come from (i.e., before you add comparison counting or convert to/from iterative/recursive). Naturally, you may not use sorting algorithms already provided by Java's standard library (or other packages). (100 points)

4.3 Compiling and Running

Code that fails to compile or code that compiles but fails to run will receive a grade of zero. Your program must accept a single command-line argument as a path to the input file.

4.4 README

Commands to run and compile the program should be documented clearly in the README file.

You must include your full name and university email address in your README file. If you are doing a project in a group of two, list the full names and email addresses of all two group members. If you are in a group, you must also describe the contributions of each group member in the assignment. (5 points)

4.5 Style Requirements

All submitted source files must conform to the CSCI 1302 style guide. Please see the guide on eLC for setting up style checks. (5 points for each style violation, up to 20)

A lack of appropriate inline comments may also constitute improper documentation.

4.6 Unit Tests

You are *encouraged* to write unit tests for your project. In CSCI 1302, you may have done this using custom classes, variables, and methods. Here, we recommend using a framework like JUnit to test your linked list features. Like integrating Checkstyle with your build process, testing frameworks provide another convenient way to speed-up your development process.

A sample test file is included on eLC. Although the package for `SortedLinkListTest` is the same as `SortedLinkList`, note that the test file is under `src/test/java/cs2720/p1`, while the main file is under `src/main/java/cs2720/p1`.

Feel free to check out the JUnit 5 documentation for more details.

You should make sure your JUnit and Surefire dependencies are compatible.

See the Project 1 instructions on eLC for more details.

4.6.1 Note on Checkstyle

Unit tests are meant to be descriptive by design, particularly since they are often only a few lines of code. Because of this, you are not explicitly required to Javadoc methods that are given the `@Test` annotation (aka the actual unit tests that are run). Bear in mind that you are still expected to document a longer test whose purpose is not immediately clear.

4.7 Late Submission Policy

Projects must be submitted before the specified deadline in order to receive full credit. Projects submitted late will be subject to the following penalties:

- If submitted 0–24 hours after the deadline 20% will be deducted from the project score.
- If submitted 24-48 hours after the deadline 40% points will be deducted from the project score.
- If submitted more than 48 hours after the deadline a score of 0 will be given for the project.
- Assignment extensions will not be granted, regardless of the situation. If you need extra time, submit up to two days late and consider the late penalty waiver option discussed in the syllabus.

4.8 Submission Notes

You must include your full name and university email address in your README.txt file. If you are doing a project in a group of two, list the full names and email addresses of all two group members. If you are in a group, you must also describe the contributions of each group member in the assignment.

Submit the following files on Odin:

- Your Java files

- Input text files
- README.txt

Odin Login Instructions

Projects should be submitted through the School of Computing's Odin server. You can ssh into Odin using your myID credentials. For example,

```
$ ssh abc12345@odin.cs.uga.edu
```

Group Submissions:

****Note: This assignment can be done individually or in a group of two students. The members of the group should be from the same section. Submit a README file with your names, UGA emails and any other information that you want to give the TA to run your code. If you are doing it in a group of two make sure to include the contribution of each student in the README file.**

If you are working in a group of two, your submission folder must be named as follows:

```
LastName1_LastName2_assignment4
```

When you are submitting the folder you should use the submit command as follows:

```
$ submit LastName1_LastName2_assignment4 <odin-submission-destination>
```

Make sure to replace "`<odin-submission-destination>`" with the correct destination from the table on the bottom of this document.

If you are working in a group, you must **submit the assignment from the Odin account of only one of the members** in the group

Individual Submissions:

If you are working individually, your submission folder must be named as follows:

```
LastName_assignment4
```

Submission command should be used as follows:

```
$ submit LastName_assignment4 <odin-submission-destination>
```

Make sure to replace "`<odin-submission-destination>`" with the correct destination from the table on the bottom of this document.

****For all submissions, the folder name and README file are very important in order to be able to locate your assignment and grade it.**

****Note: Make sure you submit your assignment to the right folder to avoid grading delays.**

When using the submit command, first set your current directory of the terminal to the immediate parent of the assignment folder.

You can check your current working directory by typing:

```
$ pwd
```

For example if your assignment is in the following directory,

```
/documents/csci2720/assignments/LastName_assignment4
```

Before executing the submit command, change to the assignment's immediate parent directory:

```
$ cd /documents/csci2720/assignments/
```

Then execute the submit command:

```
$ submit LastName_assignment4 <odin-submission-destination>
```

Make sure to replace "`<odin-submission-destination>`" with the correct destination from the table on the bottom of this document.

To check if your submission command was valid, there should be a receipt file.

```
$ ls LastName_assignment4
```

This should include something like "`rec54321`."

Selecting the Odin Submission Destination

You must select your Odin submission destination using the following mapping based on the section of the course you are registered to.

Section	Class Time	Odin Submission Destination
CRN 42634	TR 11:10 AM	`csci-2720f`