Multithreaded Sorting Application:

A Comparison of Quicksort and Mergesort

Don Wapenski

San Jose State University

CS 149: Operating Systems

Instructor: Jahan Ghofraniha

May 17, 2021

I. INTRODUCTION

This report delves into the design, implementation, and measurement of the quicksort and mergesort algorithms utilizing recursive parallel processing and a threshold value to switch to a simpler sorting algorithm. Parallel processing is achieved by extending Java's RecursiveAction class, which implements fork-join parallelism for recursive programs. The simpler sorting algorithm chosen for this report is selection sort, although many other choices are available. The application designed for this report implements a multithreaded quicksort and a multithreaded mergesort and provides the capability to generate and sort an arbitrarily large number of arrays and repeat over many array sizes and threshold values. Additionally, it verifies that all sorts were performed correctly and tracks the average time taken for all sorts for each array size and threshold value.

In this implementation and with optimal threshold values, quicksort outperformed mergesort in computation time for all array sizes. However, determining the optimal threshold value is generally difficult and requires extensive timing trials that would require computation time beyond the scope of this report. Further, this report finds that the optimal threshold value could be somewhat nebulous and more of a range of possible values, which could indicate that the optimal value changes depending on the elements contained in the array. Contrary to what one may expect, the data in this report does not indicate that the optimal threshold value changes significantly with the size of the array.

II. PROBLEM STATEMENT

The problem presented in this project was to create an application that sorts an array using a multithreaded quicksort and a multithreaded mergesort, with both types of sorts

switching to a simpler algorithm, such as a selection or insertion sort, once a given threshold value has been reached.

Briefly, a quicksort sorts an array by selecting a pivot value to partition the array into subarrays. Partitioning the array involves placing all values less than the pivot on the left side of the array and all values greater than the pivot on the right side. The pivot is then in its correct place in the array and need not move again. The process is then recursively repeated for each subarray until it is completely sorted.

A mergesort performs its sort in two phases: recursively split the array in half until each half is sorted (a single value would be considered sorted), then merge each subarray such that the result is one sorted array. Since each subarray is sorted, the merge is most commonly implemented by repeatedly comparing the first values in each subarray, then moving through each subarray as the smaller value is written to the main array. In the given problem, the array is not split in half until single values are reached, but until a threshold value is reached and a simpler sorting algorithm is used. It should be noted that a threshold value of one would yield a "true" mergesort.

Both sorts must be implemented by extending Java's RecursiveAction task, which provides fork-join parallelism capability. RecursiveAction, as its name suggests, simply performs an action recursively without returning a value, while its counterpart RecursiveTask does return a value. RecursiveAction was chosen over RecursiveTask to allow all threads to work on the same array. With careful monitoring of which threads work on which ranges of values, it is not possible for a race condition to occur.

III. SOFTWARE ARCHITECTURE

This application was designed to achieve the following goals: (1) Generate and sort a list of integers with quicksort and mergesort; (2) Perform each algorithm with several array sizes and threshold values an arbitrary number of times; and, (3) Encapsulate to the greatest extent possible to ease potential future modifications (e.g., changing the method of I/O). To those ends, this application is comprised of the following Java classes: CS149Project, BatchSorter, Sorter, QuickSorter, MergeSorter, and Statistics.

CS149Project is the main class and handles all of the user input and output and only interacts with BatchSorter and Statistics. Specifically, CS149Project accepts user input from a .properties file, passes it to a BatchSorter object, then retrieves the output (stored in Statistics objects) from BatchSorter once all sorts are complete and formats it into a .csv file. This separation eases potential modification of what and how values are read in and written out. The properties needed as input are listed below:

- Iterations: The number of iterations to perform.
- Sorts per Iteration: The number of arrays to sort per iteration.
- Array Size: The initial size of the arrays to sort.
- Array Increment: How much to increase the size of the arrays per iteration.
- Threshold: The initial threshold to switch to a simpler sorting algorithm.
- Threshold Increment: How much to increase the threshold by each iteration.
- Threshold Iterations: How many times to increment the threshold per array size (i.e., overall iteration). Effectively multiplies the number of overall iterations.

Once BatchSorter has received all input values from CS149Project, it performs input verification and sets any invalid input to the minimum value that makes sense (e.g., the minimum value that makes sense for Array Size is one). Subsequently, BatchSorter creates Sorter objects for every individual array that needs to be sorted, retrieving and tracking in aggregate the time taken for each sort and number of failed sorts. For each Array Size/Threshold pair of values, it creates and stores a Statistics object that stores the aggregated sort times and failed sorts.

Each Sorter object generates an array of random values to be sorted first by quicksort then again by mergesort. It does so by creating QuickSorter and MergeSorter objects, passing in the array and threshold value to use, tracks the time taken for each sort, and verifies that each sort was performed correctly.

At last, the QuickSorter and MergeSorter classes recursively perform the actual sorts called for by the project. Both extend the RecursiveAction class and fork new threads for each subtask created. In the case of QuickSorter, a single iteration of quicksort is performed, then two further QuickSorter objects are forked to quicksort each subarray on either side of the pivot (not including the pivot itself). In the case of MergeSorter, the array is split in half into two subarrays, it then forks two MergeSorter objects to again split each subarray further. Finally, MergeSorter will then merge the two subarrays into one sorted array.

Once all sorts are completed, CS149Project retrieves the results from BatchSorter and compiles them into a .csv file with the following columns:

- Array Size: The size of the arrays that were sorted.
- Threshold: The threshold value used in the sorts.
- Average Time Quicksort: The average time to sort each array with quicksort.
- Average Time Mergesort: The average time to sort each array with mergesort.

- Failed Quicksorts: The number of quicksorts that did not yield a correctly sorted array. The expected value is zero.
- Failed Mergesorts: The number of mergesorts that did not yield a correctly sorted array. The expected value is zero.

IV. IMPLEMENTATION

As the purpose of the project is to perform a multithreaded quicksort and a multithreaded mergesort, this section will focus on the implementation of the QuickSorter and MergeSorter classes rather than going into the minutiae of the other classes, which effectively serve a management role of initiating and tracking each mergesort and quicksort. First, this section will discuss the common attributes of QuickSorter and MergeSorter, then it will delve into the issues and implementation choices associated with each algorithm.

The data given to QuickSorter and MergeSorter are an array of integers and a threshold value to use. The array of integers contains randomized values starting from zero and up to the array size times ten. The upper bound was selected as such to yield a spread of numbers while still making duplicate values somewhat likely. A certain amount of duplicate values are desired to test the performance of each algorithm in these situations. An area of potential interest for future analysis is to vary the upper bound of values in the array and compare the results. While only positive (or zero) values are included in the arrays for these tests, QuickSorter and MergeSorter also work as expected with negative values.

The implementations of the quicksort and mergesort both provide for switching to a simpler sorting algorithm once an arbitrary threshold value is reached. For the purposes of this project, a selection sort was selected as the simpler algorithm. If a user wished to perform a true

quicksort or mergesort, a threshold value of one with zero threshold iterations would yield the desired results. The selection sort is performed by following the below basic steps:

1. Find the minimum value of the array, then swap it to the start of the array. This value is now in its correct place in the sorted array.
2. Find the next minimum value from the second value of the array through the end and swap it into the second index. The first two indices of the array are now correctly sorted.
3. Repeat the above step, moving forward one index each time, until the array is completely sorted.

i. Implementation of QuickSorter

A brief description of the quicksort algorithm is given in Section II of this report. One of the primary implementation choices around the quicksort algorithm is how to select the pivot value. Given the semantics of the quicksort algorithm, the best pivot would be the median value of the array. However, calculating the median requires a sorted array, which is not available in this situation, as the purpose of this application is to sort the array. Therefore, we must find an alternative strategy to select the pivot. The strategy employed in this implementation is to estimate the true median value by selecting the first, middle, and last values of the array and calculating the median of those three values. While it is sometimes, if not often, a far cry from the true median value, it balances getting a good enough estimation and computation time.

The QuickSorter class performs a quicksort primarily using four methods: compute(), quicksort(), choosePivot(), and selectionSort(). Once a QuickSorter object is forked, it runs the compute() method, which follows the below basic steps:

1. If the number of values to sort is less than the threshold value, call selectionSort().

2. Otherwise, do the following:
    a. Choose a pivot using choosePivot().
    b. Partition the list using quicksort(), passing in the value of the pivot obtained from the previous step.
    c. Fork two new QuickSorter objects to work on each subarray on either side of the pivot, which will repeat the above steps.

quicksort() partitions the list by following the below basic steps:

1. Find the first occurence of the pivot based on the pivot value passed into it.
2. Swap the pivot with the last value of the array to put it out of the way for the next steps.
3. Iterate through the array, swapping values less than or equal to the pivot to the start of the array.
4. Swap the pivot to be immediately after the values less than or equal to it. The pivot is now in its correct place in the sorted array and does not need to move for the rest of the sort.

The implementations of choosePivot() and selectionSort() have already been discussed earlier in this section.

ii. *Implementation of Mergesort*

A brief description of the mergesort algorithm is given in Section II of this report. Compared to quicksort, mergesort does not have as many design choices to make, as it does not rely on a pivot value. It also contrasts with quicksort in how it utilizes multithreading. As shown

previously, quicksort only forks new tasks once it has partitioned the array and completed its subtask. Mergesort, however, begins by splitting the array in half and forking new tasks to further split each subarray. Once the subarrays reach a threshold value, they are sorted using a selection sort, at which point mergesort takes over again by merging the subarrays together into one sorted array. As such, this implementation of mergesort relies primarily on two methods: compute() and selectionSort(). The semantics of selectionSort() have already been discussed. The basic steps of the compute() method are given below:

1. If the number of values to sort is less than the threshold value, call selectionSort().
2. Otherwise, do the following:
    a. Split the array in half into two subarrays.
    b. Pass each subarray into two new MergeSorters and fork them.
    c. Once both the new MergeSorters have finished, merge their now-sorted subarrays together by comparing the first values in each and taking the lesser as the first value of the combined array. Repeat this comparison moving down each subarray until all values have been incorporated.

V. TESTS AND RESULTS

In this section, we begin by evaluating the results of a unit test with JUnit to ensure that this implementation of quicksort and mergesort is functioning correctly and is taking full advantage of multithreading. Next, the runtimes of each sort at different threshold values will be explored. Finally, the performance of mergesort and quicksort will be compared at each algorithm's respective optimal threshold value.

The sorts were tested with arrays of size 10,000, 20,000, … 100,000 and each size was tested with threshold values 1, 10, 20 .. 100, 200, … 1,000. For each array size/threshold value pair, 10,000 arrays were generated and sorted with quicksort and mergesort. The runtimes stated reflect the averages of the runtimes for those 10,000 arrays.

i. Unit Test

The QuickSorter and MergeSorter classes were unit tested with JUnit to ensure the accuracy of sorts and utilization of multithreading. First, the unit tests began by passing in a handful of arrays representing different types of test cases. The full list of test cases can be found in Appendix A-2 in the QuickSorterTest.java and MergeSorterTest.java files. In summary, the test cases included five "normal" arrays with the types of integers these classes are intended to work with (greater than or equal to 0 with well-distributed values), all negative values, a mix of negative and positive values, all zeroes, an already-sorted array, a reverse-sorted array, an array containing one value, and an array containing no values. In all cases, both MergeSorter and QuickSorter obtained the expected results, an array sorted in ascending order (or an empty array in the case they were given an empty array).

Next, it is essential to this project that each sort is truly being performed in parallel and that threads aren't waiting for each other before executing, which would be equivalent to a serialized implementation with more overhead. Parallelism can be difficult to ensure since, if implemented correctly, threads will execute asynchronously, which means there is no one expected result. In these unit tests, each class is given a single array to sort and statements are printed when each thread begins and finishes sorting a subarray. If the sort is happening in parallel, the output should reflect multiple threads working simultaneously.

An excerpt of the results for QuickSorterTest.java is shown below in Figure 1. It should be noted that some thread IDs are repeated because Java's ForkJoinTask base class utilizes a thread pool. In Figure 1, Thread 18 sorts indices 17 to 19 while Thread 17 completes a few single-element sorts, which indicates that they were working in parallel. For ease of reading the output, an array of size 20 was used with threshold 2, but a larger array would exemplify parallelism more frequently.

```
QuickSorter multithreaded test
...
Thread 18 sorting from 17 to 19
Thread 17 finished sorting from 0 to 1
Thread 17 sorting from 3 to 3
Thread 17 finished sorting from 3 to 3
Thread 17 sorting from 5 to 5
Thread 17 finished sorting from 5 to 5
Thread 18 finished sorting from 17 to 19
...
QuickSorter multithreaded complete
```

Figure 1: An excerpt of the print statements generated showing the activities of each thread for QuickSorter.

The same method to evaluate parallelism was employed for MergeSorter; however, since MergeSorter forks most of its tasks before performing much of the computation, a different pattern is expected for the output. In this case, we expect to see many statements indicating a sort has begun followed by many statements stating a sort has finished. In Figure 2 below, the first action MergeSorter takes is to create Thread 1 to sort the entire array, which then forks two more threads to sort each half of the array and so on. As the threshold value is reached and subarrays begin to be sorted and merged, many threads finish their tasks, ending with Thread 1 as expected.

```
MergeSorter multithreaded test
Thread 1 sorting from 0 to 19
Thread 15 sorting from 0 to 9
Thread 17 sorting from 0 to 4
Thread 18 sorting from 5 to 9
...
Thread 16 sorting from 10 to 19
...
Thread 14 sorting from 15 to 16
Thread 14 finished sorting from 15 to 16
...
Thread 15 finished sorting from 0 to 9
...
Thread 14 finished sorting from 15 to 19
Thread 16 finished sorting from 10 to 19
Thread 1 finished sorting from 0 to 19
MergeSorter multithreaded complete
```

Figure 2: An excerpt of the print statements generated showing the activities of each thread for MergeSorter.

ii. Behavior with Threshold Values Between 100 and 1,000

This report begins the analysis of optimal threshold values by examining the runtime of each algorithm with large threshold values relative to the size of the array. Specifically, the threshold values in this subsection range from 100 to 1,000 and array sizes range from 10,000 to 100,000. For simplicity, this report will only include the smallest and the largest array sizes, however trends in performance remained consistent throughout all array sizes for both quicksort and mergesort.

The performance of quicksort (Figure 3) and mergesort (Figure 4) both degraded as the threshold value approached 1,000. This is in line with expectations, as a selection sort of 1,000 elements is likely to be much slower than a quicksort or mergesort of the same set of elements. Of note, however, is where the two algorithms diverge in behavior. Quicksort's performance degrades linearly while mergesort's performance degrades in a stepwise manner. Mergesort likely exhibits this behavior because of the nature of the algorithm itself. Since mergesort always

splits arrays in half (e.g. 10,000, 5,000, 1,250, 625, 312…), a difference in threshold value of 100 may not make a difference in when the selection sort is applied. In this respect, mergesort is insensitive to changes of the threshold value unless certain breakpoints are reached, dependent upon the size of the array.

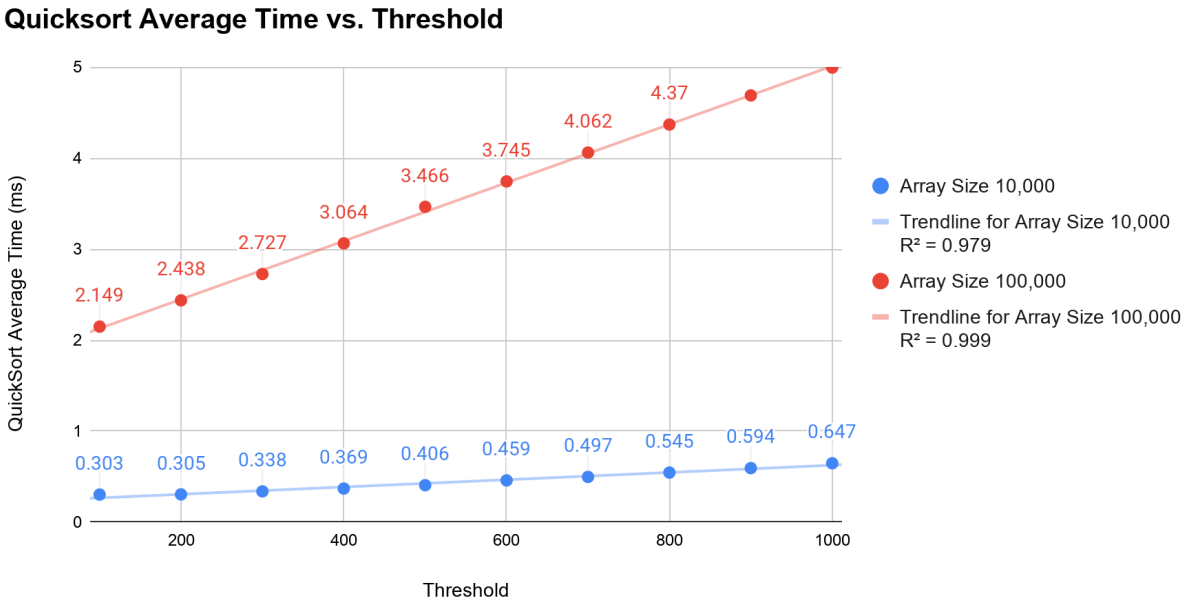

Figure 3: A scatterplot of the average runtime of quicksort in milliseconds. The linear trend in runtime is consistent across array sizes, although only sizes 10,000 and 100,000 are depicted.

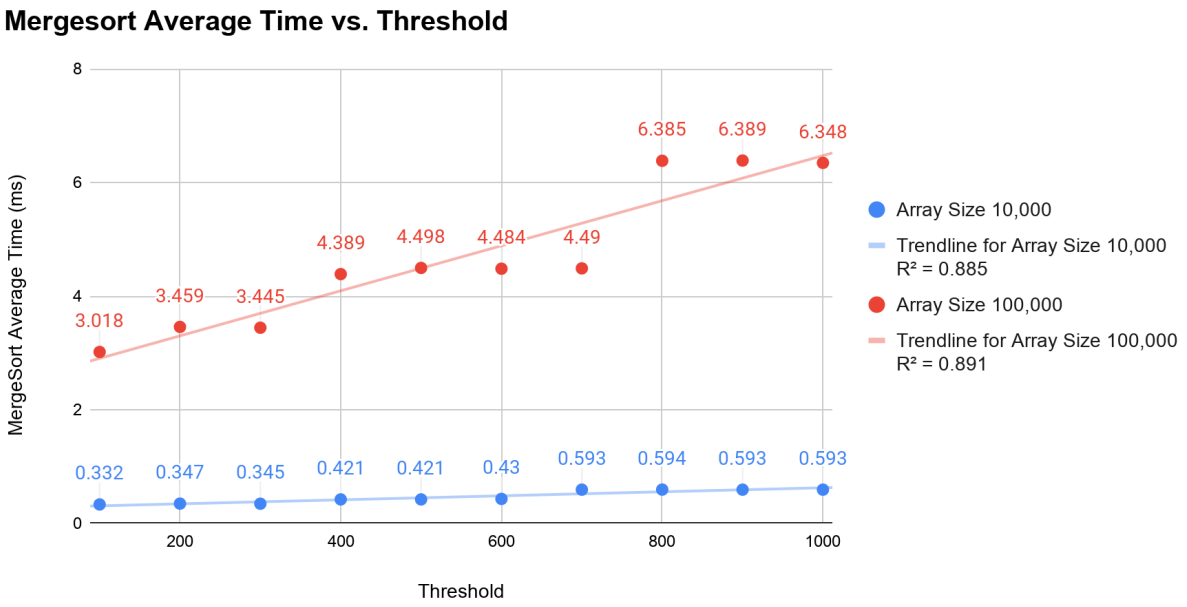

Figure 4: A scatterplot of the average runtime of mergesort in milliseconds. The increasing stepwise trend in runtime is consistent across array sizes, although only sizes 10,000 and 100,000 are depicted here. Trendlines and R^2 values are provided here for consistency with Figure 4.

iii. Behavior with Threshold Values Between 1 and 100

Both quicksort and mergesort had optimal threshold values between 1 and 100 for all array sizes, which warrants closer analysis with a smaller threshold increment. Of note, a threshold size of 1 (i.e., performing quicksort or mergesort all the way through the array) produced poor results in terms of runtime, sometimes even worse than the larger threshold values examined previously. With relatively small threshold sizes, mergesort did not appear to exhibit the same stepwise behavior as shown in Subsection ii, likely because the subarrays are much smaller and therefore have many more breakpoints than with larger threshold values.

With smaller threshold values, quicksort and mergesort showed very similar trends in performance: nearly flat. Interestingly, neither quicksort nor mergesort exhibited significant differences in runtime for threshold values between 20 and 40, with only marginal increases in runtime for other values except the large difference with 1. This is a departure from the steady

increases observed for larger threshold values; that being said, the optimal threshold values were very consistent for both quicksort and mergesort. With array size 10,000, the optimal threshold for both quicksort and mergesort were 30, whereas for all other (larger) array sizes, the optimal threshold for quicksort was 40 and for mergesort was 20. The implications of this will be discussed in Section VII.

Figure 5 below depicts the performance of quicksort and mergesort with an array size of 100,000 to accentuate the relatively small differences in runtime between threshold values. The same trend was present in all other array sizes and have been omitted from this report for brevity. As mentioned, a threshold of size 1 produces the worst runtime while values between 20 and 40 produce the best results.

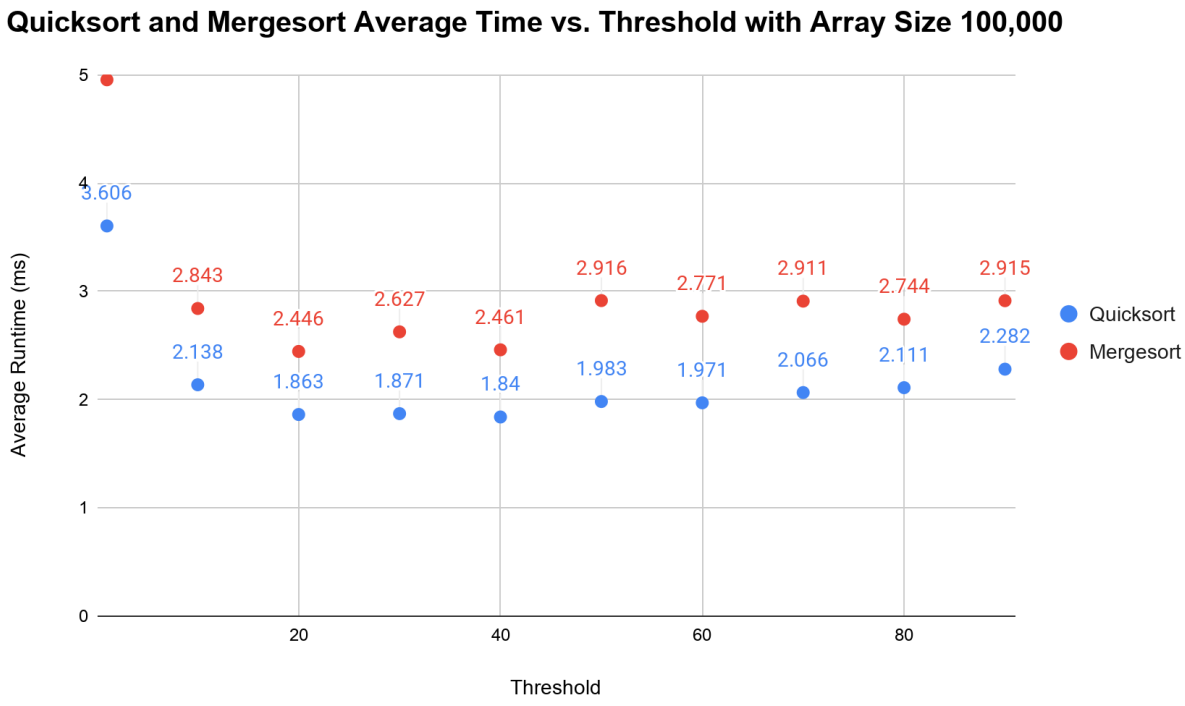

Figure 5: A chart showing the runtime for quicksort and mergesort with threshold values between 1 and 90 with an array size of 100,000. The optimal threshold value for quicksort is 40, while the optimal value for mergesort is 20.

iv. Comparison at Optimal Thresholds Values

Finally, the runtimes for quicksort and mergesort will be compared as a function of array size. For this comparison, the optimal threshold values for each array size are used. For quicksort and mergesort, the optimal threshold for array size 10,000 was 30. From 20,000 to 100,000, the optimal value for quicksort was 40 and for mergesort was 20. As shown in Figure 6, both sorts exhibited a linear growth in runtime as array size increases, with mergesort taking increasingly more time than quicksort.

Figure 6: A scatterplot with trendlines comparing the average runtimes of quicksort and mergesort with their optimal threshold values.

VI. GROUP MEMBERS

Don Wapenski is the sole group member for this project.

VII. CONCLUSIONS

In summary, this report has explored the behaviors of mergesort and quicksort at various threshold values and array sizes, where the threshold value indicates when to switch to a simpler sorting algorithm (in this case, selection sort). The most apparent conclusion from the results in Section V is that in this implementation, quicksort outperforms mergesort in all scenarios where the optimal threshold is chosen. However, as with all other conclusions this report draws, this could change with a different implementation and choice in simpler sorting algorithm. Compromises were made for computation time, but a complete study of this area would involve at least more array sizes, threshold values, and most of all testing more simple sorting algorithms.

In regards to the selection of threshold value, the data indicates that finding a range of values may yield adequate results rather than spending more computation time finding the single best value. For both sorting algorithms, a threshold value between 10 and 80 yielded comparable results with values between 20 and 40 being the “sweet spot.” It is possible that the optimal threshold changes depending on the values stored in the array, which could be the reason the optimal threshold appears to be more of a range than a specific value and would indicate a flaw in the methodology of this data collection. The number of duplicates, to what degree the array is already ordered, and how spread out the values are could all potentially influence what the optimal threshold is.

VIII. REFERENCES

For a general outline of quicksort and mergesort:

Malik, D. S. (2010). Sorting Algorithms. *Data Structures Using C++: Second Edition* (pp. 533-598). Course Technology, Cengage Learning.

For an introduction to Java's fork-join library:

Silberschatz, A., Galvin, P. B., Gagne, G. (2018). Threads & Concurrency. *Operating System Concepts: Tenth Edition* (pp. 180-183). John Wiley & Sons, Inc.

For an outline of writing JUnit tests in Netbeans:

Apache NetBeans. (n.d.). Writing JUnit Tests in NetBeans IDE. Apache NetBeans. http://netbeans.apache.org/kb/docs/java/junit-intro.html

The following Javadoc webpages were used from Oracle's website:

- https://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html
- https://docs.oracle.com/javase/8/docs/api/java/util/Random.html
- https://docs.oracle.com/javase/7/docs/api/java/io/FileInputStream.html
- https://docs.oracle.com/javase/8/docs/api/java/io/FileWriter.html
- https://docs.oracle.com/javase/7/docs/api/java/util/Properties.html
- https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html

IX. APPENDIX A-1: RAW DATA

The raw data used for Section V is below. The quicksort and mergesort times reflect the average time needed for a sort over 10,000 arrays.

Array Size	Threshold	Quicksort Average Time (ms)	Mergesort Average Time (ms)	Failed Quicksort	Failed Mergesort
10000	1	0.47	0.542	0	0
10000	10	0.312	0.302	0	0
10000	20	0.292	0.281	0	0
10000	30	0.256	0.271	0	0
10000	40	0.265	0.283	0	0
10000	50	0.263	0.309	0	0
10000	60	0.259	0.283	0	0
10000	70	0.282	0.311	0	0
10000	80	0.274	0.34	0	0
10000	90	0.3	0.365	0	0
10000	100	0.303	0.332	0	0
10000	200	0.305	0.347	0	0
10000	300	0.338	0.345	0	0
10000	400	0.369	0.421	0	0
10000	500	0.406	0.421	0	0
10000	600	0.459	0.43	0	0
10000	700	0.497	0.593	0	0
10000	800	0.545	0.594	0	0
10000	900	0.594	0.593	0	0
10000	1000	0.647	0.593	0	0
20000	1	0.91	0.927	0	0
20000	10	0.584	0.55	0	0
20000	20	0.462	0.495	0	0
20000	30	0.445	0.521	0	0
20000	40	0.433	0.536	0	0
20000	50	0.466	0.565	0	0
20000	60	0.449	0.536	0	0
20000	70	0.47	0.558	0	0
20000	80	0.476	0.642	0	0
20000	90	0.506	0.669	0	0
20000	100	0.494	0.593	0	0
20000	200	0.542	0.658	0	0

20000	300	0.602	0.661	0	0
20000	400	0.678	0.816	0	0
20000	500	0.787	0.859	0	0
20000	600	0.833	0.827	0	0
20000	700	0.878	1.122	0	0
20000	800	0.961	1.127	0	0
20000	900	1.048	1.133	0	0
20000	1000	1.113	1.116	0	0
30000	1	1.068	1.354	0	0
30000	10	0.75	0.799	0	0
30000	20	0.642	0.731	0	0
30000	30	0.611	0.786	0	0
30000	40	0.6	0.752	0	0
30000	50	0.631	0.789	0	0
30000	60	0.635	0.863	0	0
30000	70	0.68	0.901	0	0
30000	80	0.676	0.862	0	0
30000	90	0.719	0.904	0	0
30000	100	0.697	0.851	0	0
30000	200	0.771	0.917	0	0
30000	300	0.872	1.088	0	0
30000	400	0.967	1.087	0	0
30000	500	1.058	1.422	0	0
30000	600	1.147	1.416	0	0
30000	700	1.255	1.417	0	0
30000	800	1.357	1.412	0	0
30000	900	1.466	1.412	0	0
30000	1000	1.589	2.13	0	0
40000	1	1.335	1.799	0	0
40000	10	0.973	1.043	0	0
40000	20	0.805	0.959	0	0
40000	30	0.808	1.036	0	0
40000	40	0.801	1.044	0	0
40000	50	0.83	1.09	0	0
40000	60	0.831	1.047	0	0
40000	70	0.889	1.116	0	0
40000	80	0.894	1.278	0	0
40000	90	0.998	1.384	0	0

40000	100	0.89	1.144	0	0
40000	200	0.996	1.28	0	0
40000	300	1.135	1.296	0	0
40000	400	1.266	1.596	0	0
40000	500	1.373	1.576	0	0
40000	600	1.499	1.578	0	0
40000	700	1.624	2.178	0	0
40000	800	1.762	2.18	0	0
40000	900	1.905	2.188	0	0
40000	1000	2.044	2.185	0	0
50000	1	1.645	2.203	0	0
50000	10	1.144	1.334	0	0
50000	20	1.003	1.2	0	0
50000	30	0.997	1.31	0	0
50000	40	0.982	1.214	0	0
50000	50	1.004	1.439	0	0
50000	60	1.008	1.359	0	0
50000	70	1.094	1.453	0	0
50000	80	1.086	1.368	0	0
50000	90	1.138	1.408	0	0
50000	100	1.09	1.468	0	0
50000	200	1.223	1.688	0	0
50000	300	1.373	1.688	0	0
50000	400	1.528	2.151	0	0
50000	500	1.682	2.149	0	0
50000	600	1.85	2.154	0	0
50000	700	2.02	2.16	0	0
50000	800	2.178	3.113	0	0
50000	900	2.345	3.111	0	0
50000	1000	2.521	3.12	0	0
60000	1	2.116	2.739	0	0
60000	10	1.34	1.586	0	0
60000	20	1.164	1.421	0	0
60000	30	1.224	1.592	0	0
60000	40	1.128	1.504	0	0
60000	50	1.187	1.562	0	0
60000	60	1.195	1.714	0	0
60000	70	1.237	1.76	0	0

60000	80	1.291	1.718	0	0
60000	90	1.365	1.803	0	0
60000	100	1.353	1.732	0	0
60000	200	1.507	1.892	0	0
60000	300	1.7	2.209	0	0
60000	400	1.875	2.221	0	0
60000	500	2.094	2.872	0	0
60000	600	2.251	2.874	0	0
60000	700	2.433	2.863	0	0
60000	800	2.652	2.89	0	0
60000	900	2.927	2.941	0	0
60000	1000	3.058	4.29	0	0
70000	1	2.459	3.367	0	0
70000	10	1.459	1.809	0	0
70000	20	1.329	1.688	0	0
70000	30	1.338	1.772	0	0
70000	40	1.31	1.79	0	0
70000	50	1.371	1.891	0	0
70000	60	1.381	1.786	0	0
70000	70	1.468	2.214	0	0
70000	80	1.522	2.11	0	0
70000	90	1.583	2.217	0	0
70000	100	1.563	2.033	0	0
70000	200	1.749	2.269	0	0
70000	300	1.967	2.714	0	0
70000	400	2.193	2.684	0	0
70000	500	2.403	2.722	0	0
70000	600	2.616	3.639	0	0
70000	700	2.835	3.637	0	0
70000	800	3.102	3.648	0	0
70000	900	3.26	3.579	0	0
70000	1000	3.487	3.575	0	0
80000	1	2.751	3.762	0	0
80000	10	1.679	2.07	0	0
80000	20	1.526	1.939	0	0
80000	30	1.525	2.03	0	0
80000	40	1.507	2.08	0	0
80000	50	1.555	2.189	0	0

80000	60	1.575	2.081	0	0
80000	70	1.694	2.212	0	0
80000	80	1.705	2.536	0	0
80000	90	1.822	2.633	0	0
80000	100	1.735	2.317	0	0
80000	200	2.003	2.64	0	0
80000	300	2.196	2.596	0	0
80000	400	2.428	3.188	0	0
80000	500	2.692	3.188	0	0
80000	600	2.903	3.159	0	0
80000	700	3.153	4.344	0	0
80000	800	3.456	4.412	0	0
80000	900	3.7	4.393	0	0
80000	1000	3.964	4.398	0	0
90000	1	3.046	4.378	0	0
90000	10	1.941	2.537	0	0
90000	20	1.646	2.184	0	0
90000	30	1.676	2.324	0	0
90000	40	1.642	2.187	0	0
90000	50	1.78	2.549	0	0
90000	60	1.768	2.405	0	0
90000	70	1.876	2.509	0	0
90000	80	1.914	2.421	0	0
90000	90	2.016	3.102	0	0
90000	100	1.942	2.679	0	0
90000	200	2.206	3.02	0	0
90000	300	2.487	3.048	0	0
90000	400	2.733	3.765	0	0
90000	500	3.012	3.767	0	0
90000	600	3.294	3.763	0	0
90000	700	3.59	3.769	0	0
90000	800	3.863	5.289	0	0
90000	900	4.169	5.297	0	0
90000	1000	4.451	5.296	0	0
100000	1	3.606	4.957	0	0
100000	10	2.138	2.843	0	0
100000	20	1.863	2.446	0	0
100000	30	1.871	2.627	0	0

100000	40	1.84	2.461	0	0
100000	50	1.983	2.916	0	0
100000	60	1.971	2.771	0	0
100000	70	2.066	2.911	0	0
100000	80	2.111	2.744	0	0
100000	90	2.282	2.915	0	0
100000	100	2.149	3.018	0	0
100000	200	2.438	3.459	0	0
100000	300	2.727	3.445	0	0
100000	400	3.064	4.389	0	0
100000	500	3.466	4.498	0	0
100000	600	3.745	4.484	0	0
100000	700	4.062	4.49	0	0
100000	800	4.37	6.385	0	0
100000	900	4.69	6.389	0	0
100000	1000	4.996	6.348	0	0

X. APPENDIX A-2: SOURCE CODE

The source code used for this project is available at the link below. In case the link does not work for any reason, the code is also copied below.

https://github.com/DonaldWapenski/CS149MultithreadedSortingApplication

CS149Project.java

```
package cs149project;

import java.io.FileInputStream;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Properties;
import java.util.Iterator;

/**
 * Main file for this project.
 * Sorts an array using two threads with Quicksort and
Mergesort.
 * @author Don Wapenski 010535538
 */
```

```
public class CS149Project {
    final private static String parametersFileName =
"parameters.properties";
    final private static String outputFileName = "output.csv";
    final private static String lineSeparator =
System.lineSeparator();

    public static void main(String argv[]) {
        System.out.println("Program Starting");
        //Start by loading the input file
        Properties parameters;
        FileInputStream in;
        BatchSorter bs = new BatchSorter();
        try {
            parameters = new Properties();
            in = new FileInputStream(parametersFileName);
            parameters.load(in);

            //Read input. Set defaults to -1 since BatchSorter
performs validation
            int it =
Integer.parseInt(parameters.getProperty("iterations","-1"));
            int sorts_per_it =
Integer.parseInt(parameters.getProperty("sorts_per_iteration","
-1"));
            int array_size =
Integer.parseInt(parameters.getProperty("array_size","-1"));
            int array_inc =
Integer.parseInt(parameters.getProperty("array_increment","-1")
);
            int thresh =
Integer.parseInt(parameters.getProperty("threshold","-1"));
            int thresh_inc =
Integer.parseInt(parameters.getProperty("threshold_increment","
-1"));
            int thresh_it =
Integer.parseInt(parameters.getProperty("threshold_iterations",
"-1"));
            in.close();

            bs.load(it, sorts_per_it, array_size, array_inc,
thresh, thresh_inc, thresh_it);
        } catch (IOException e) {
            System.out.println("ERROR loading input file.");
            e.printStackTrace();
            System.exit(-1);
```

```
        }

        bs.run();

        //extract statics and write to csv
        try {
            FileWriter fw = new FileWriter(outputFileName);
            fw.write("PARAMETERS"+lineSeparator);

fw.write("Iterations,"+bs.getIterations()+lineSeparator);
            fw.write("Sorts per
Iteration,"+bs.getSortsPerIt()+lineSeparator);
            fw.write("Array
Size,"+bs.getArraySize()+lineSeparator);
            fw.write("Array
Increment,"+bs.getArrayInc()+lineSeparator);

fw.write("Threshold,"+bs.getThresh()+lineSeparator);
            fw.write("Threshold
Increment,"+bs.getThreshInc()+lineSeparator);
            fw.write("Threshold
Iterations,"+bs.getThreshIt()+lineSeparator);
            fw.write(lineSeparator);
            fw.write("RESULTS"+lineSeparator);
            fw.write("Array Size,Threshold,QuickSort Average
Time,MergeSort Average
Time,FailedQuickSort,FailedMergeSort"+lineSeparator);
            Iterator<Statistics> it = bs.statistics.iterator();
            while (it.hasNext()) {
                Statistics stats = it.next();
                String s =
String.format("%d,%d,%.3f,%.3f,%d,%d", stats.arraySize,
stats.threshold, stats.avgTimeQS, stats.avgTimeMS,
stats.failedQS, stats.failedMS);
                //String s =
stats.arraySize+","+stats.threshold+","+stats.avgTimeQS+","+sta
ts.avgTimeMS+","+stats.failedQS+","+stats.failedMS+lineSeparato
r;
                fw.write(s+lineSeparator);
            }
            fw.close();
        } catch (IOException e) {
            System.out.println("ERROR writing output file.");
            e.printStackTrace();
        }
        System.out.println("Program Ending");
```

```
    }
}
```

BatchSorter.java

```
package cs149project;

import java.util.ArrayList;

/**
 * Sets up and runs Sorters and aggregates statistics
 * @author Don
 */
public class BatchSorter {
    private int iterations, sortsPerIt, arraySize, arrayInc,
thresh, threshInc, threshIt;

    protected ArrayList<Statistics> statistics;

    //Default parameters. Lowest allowable values.
    final public int DEF_ITERATIONS = 1, DEF_SORTSPERIT = 1,
DEF_ARRAYSIZE = 1,
            DEF_ARRAYINC = 0, DEF_THRESH = 1, DEF_THRESHINC =
0, DEF_THRESHIT = 0;

    /**
     * Default constructor.
     */
    public BatchSorter() {
        this.iterations = DEF_ITERATIONS;
        this.sortsPerIt = DEF_SORTSPERIT;
        this.arraySize = DEF_ARRAYSIZE;
        this.arrayInc = DEF_ARRAYINC;
        this.thresh = DEF_THRESH;
        this.threshInc = DEF_THRESHINC;
        this.threshIt = DEF_THRESHIT;
        this.statistics = new ArrayList<>();
    }

    /**
     * Constructor with input parameters. If parameters passed
in are unallowable values,
     * they will be set to the default value.
     * @param iterations Number of iterations to perform
     * @param sortsPerIt Number of arrays to sort per iteration
```

```
     * @param arraySize Initial size of array to sort
     * @param arrayInc Size to increase array per iteration
     * @param thresh Initial threshold to switch to a simpler
sorting algorithm
     * @param threshInc How much to increase the threshold size
by each iteration
     * @param threshIt Number of iterations to test threshold,
multiplies number of overall iterations
     */
    public BatchSorter(int iterations, int sortsPerIt, int
arraySize, int arrayInc, int thresh, int threshInc, int
threshIt) {
        load(iterations, sortsPerIt, arraySize, arrayInc,
thresh, threshInc, threshIt);
    }

    /**
     * Loads values given to it, similar to constructor.
     * @param iterations Number of iterations to perform
     * @param sortsPerIt Number of arrays to sort per iteration
     * @param arraySize Initial size of array to sort
     * @param arrayInc Size to increase array per iteration
     * @param thresh Initial threshold to switch to a simpler
sorting algorithm
     * @param threshInc How much to increase the threshold size
by each iteration
     * @param threshIt Number of iterations to test threshold,
multiplies number of overall iterations
     */
    public void load(int iterations, int sortsPerIt, int
arraySize, int arrayInc, int thresh, int threshInc, int
threshIt) {
        if (iterations < DEF_ITERATIONS) {
            this.iterations = DEF_ITERATIONS;
        } else {
            this.iterations = iterations;
        }
        if (sortsPerIt < DEF_SORTSPERIT) {
            this.sortsPerIt = DEF_SORTSPERIT;
        } else {
            this.sortsPerIt = sortsPerIt;
        }
        if (arraySize < DEF_ARRAYSIZE) {
            this.arraySize = DEF_ARRAYSIZE;
        } else {
            this.arraySize = arraySize;
```

```
        }
        if (arrayInc < DEF_ARRAYINC) {
            this.arrayInc = DEF_ARRAYINC;
        } else {
            this.arrayInc = arrayInc;
        }
        if (thresh < DEF_THRESH) {
            this.thresh = DEF_THRESH;
        } else {
            this.thresh = thresh;
        }
        if (threshInc < DEF_THRESHINC) {
            this.threshInc = DEF_THRESHINC;
        } else {
            this.threshInc = threshInc;
        }
        if (threshIt < DEF_THRESHIT) {
            this.threshIt = DEF_THRESHIT;
        } else {
            this.threshIt = threshIt;
        }
        this.statistics = new ArrayList<>();
    }

    /**
     * Sets up, runs, and collects statistics for each sort
     */
    public void run() {
        for (int i=0; i<iterations; i++) {
            System.out.println("Iteration " + (i+1));
            for (int j=0; j<threshIt; j++) {
                Sorter sorter = new
Sorter(arraySize+i*arrayInc, thresh+j*threshInc);

                //Trackers for average run-time for each
algorithm
                double avgTimeQS = 0;
                double avgTimeMS = 0;

                //Paranoid checks for incorrect sorting. Should
stay 0 by end of execution.
                int failedQS = 0;
                int failedMS = 0;
                for (int s=0; s<sortsPerIt; s++) {
                    sorter.run();
                    avgTimeQS += sorter.getQSTime() /
```

```
sortsPerIt;
                    avgTimeMS += sorter.getMSTime() /
sortsPerIt;
                    if (!sorter.verifyQS()) {
                        failedQS++;
                    }
                    if (!sorter.verifyMS()) {
                        failedMS++;
                    }
                }

                //Record statistics
                avgTimeQS /= 1000000; //convert ns to ms
                avgTimeMS /= 1000000;
                statistics.add(new
Statistics(arraySize+i*arrayInc, thresh+j*threshInc, avgTimeQS,
avgTimeMS, failedQS, failedMS));
            }
        }
    }

    public int getIterations() {
        return iterations;
    }

    public int getSortsPerIt() {
        return sortsPerIt;
    }

    public int getArraySize() {
        return arraySize;
    }

    public int getArrayInc() {
        return arrayInc;
    }

    public int getThresh() {
        return thresh;
    }

    public int getThreshInc() {
        return threshInc;
    }

    public int getThreshIt() {
```

```
        return threshIt;
    }
}
```

Sorter.java

```
package cs149project;

import java.util.Arrays;
import java.util.Random;

/**
 * Creates an array and sorts it separately with QuickSort and
MergeSort
 * @author Don
 */
public class Sorter {
    private int thresh;
    private int[] arr;

    private QuickSorter qs;
    private MergeSorter ms;
    private long startTime, endTime, elapsedTimeQS,
elapsedTimeMS;

    private Random rand;

    /**
     * Constructor
     * @param arraySize the size of the array to create
     * @param threshold the threshold value to use
     */
    public Sorter(int arraySize, int threshold) {
        arr = new int[arraySize];
        this.thresh = threshold;
        rand = new Random();
    }

    /**
     * Runs and times sorts for the same array using quicksort
and mergesort
     */
    public void run() {
        //First generate random values for array. Range from 0
to 10*size, duplicates allowed
```

```
        for (int i=0; i<arr.length; i++) {
            arr[i] = rand.nextInt(arr.length*10);
        }

        //Run and time the quicksort
        startTime = System.nanoTime();
        qs = new QuickSorter(0, arr.length-1, arr.clone(),
thresh);
        qs.fork();
        qs.join();
        //System.out.println("Quicksort result: " +
Arrays.toString(qs.arr));
        endTime = System.nanoTime();
        elapsedTimeQS = endTime - startTime;
        //System.out.println("");
        //Run and time the mergesort
        startTime = System.nanoTime();
        ms = new MergeSorter(0, arr.length-1, arr.clone(),
thresh);
        ms.fork();
        ms.join();
        //System.out.println("Mergesort result: " +
Arrays.toString(ms.arr));
        endTime = System.nanoTime();
        elapsedTimeMS = endTime - startTime;
    }

    /**
     * Gets the time taken for quicksort in ms
     * @return the time taken for quicksort in ms
     */
    public long getQSTime() {
        return elapsedTimeQS;
    }

    /**
     * Gets the time taken for mergesort in ms
     * @return the time taken for mergesort in ms
     */
    public long getMSTime() {
        return elapsedTimeMS;
    }

    /**
     * Verifies that the quicksort was done correctly
     * @return true if the quicksort was correct
```

```
     */
    public boolean verifyQS() {
        return verifySort(qs.arr);
    }

    /**
     * Verifies that the mergesort was done correctly
     * @return true if the mergesort was correct
     */
    public boolean verifyMS() {
        return verifySort(ms.arr);
    }

    /**
     * Verifies that an array was sorted correctly in ascending
order.
     * @param arrCheck The array to verify
     * @return true if the sort was correct
     */
    private boolean verifySort(int[] arrCheck) {
        boolean sorted = true;
        int prev = arrCheck[0];
        for (int i=1; i>arrCheck.length; i++) {
            if (prev > arrCheck[i]) {
                sorted = false;
            }
            prev = arrCheck[i];
        }
        return sorted;
    }
}
```

QuickSorter.java

```
package cs149project;

import java.util.Arrays;
import java.util.concurrent.*;

/**
 * Performs a quicksort on an array using RecursiveAction
 * @author Don
 */
public class QuickSorter extends RecursiveAction {
    private int begin, end, thresh;
```

```
    protected int[] arr;

    //FOR TESTING PURPOSES ONLY. If true, causes a few extra
statements to print showing threads.
    protected static boolean testMultiThreading;

    /**
     * Constructor.
     * @param begin the starting index to sort from
     * @param end the ending index to stop sorting inclusive
     * @param arr the array to sort
     * @param threshold the threshold to apply a simpler
sorting algorithm
     */
    public QuickSorter(int begin, int end, int[] arr, int
threshold) {
        this.begin = begin;
        this.end = end;
        this.arr = arr;
        this.thresh = threshold;
        if (this.thresh < 1) {
            this.thresh = 1;
        }
    }

    /**
     * Overridden from RecursiveAction, performs the actual
quicksort,
     * recursively forking threads in the process. Basic
structure of
     * code is taken from page 183 of the textbook:
     * Operating System Concepts, by Silberchatz, Galvin, and
Gagne
     * 10th Edition, Wiley, ISBN: 978-1119456339
     */
    @Override
    protected void compute() {
        if (testMultiThreading) {
            System.out.println("Thread " +
Thread.currentThread().getId() + " sorting from " + this.begin
+ " to " + this.end);
        }
        if (end - begin < thresh) {
            selectionSort();
            if (testMultiThreading) {
                System.out.println("Thread " +
```

```
Thread.currentThread().getId() + " finished sorting from " +
this.begin + " to " + this.end);
            }
        } else {
            int pivot = quicksort(choosePivot());
            QuickSorter qs1 = new QuickSorter(begin, pivot-1,
arr, thresh);
            QuickSorter qs2 = new QuickSorter(pivot+1, end,
arr, thresh);
            if (testMultiThreading) {
                System.out.println("Thread " +
Thread.currentThread().getId() + " finished sorting from " +
this.begin + " to " + this.end);
            }
            qs1.fork();
            qs2.fork();
            qs1.join();
            qs2.join();
        }

    }

    /**
     * Partitions a list based on the value of the pivot
     * @param pivot the VALUE of the pivot, not the index
     * @return the INDEX the pivot is placed in after the
quicksort
     */
    private int quicksort(int pivot) {
        //First, find index of pivot value
        int pivotIndex = begin;
        while (arr[pivotIndex] != pivot) { pivotIndex++; }

        //Next, partition the array. Start by swapping pivot to
end
        swap(pivotIndex, end);
        int endOfLowerPart = begin;
        for (int i=endOfLowerPart; i<end; i++) {
            if (arr[i] <= pivot) {
                swap(endOfLowerPart, i);
                endOfLowerPart++;
            }
        }
        swap(endOfLowerPart, end); //undo initial action
swapping pivot to end
        return endOfLowerPart;
```

```
    }

    /**
     * Calculates the pivot based on an estimation of the
median. Estimation uses
     * the first, middle, and last values of the array and
finds the median
     * of those three values.
     * @return the VALUE of the pivot
     */
    private int choosePivot() {
        int first = arr[begin];
        int last = arr[end];
        int mid = arr[(end - begin)/2];
        if ((first < mid && mid < last) ||
                (last < mid && mid < first)) {
            return mid; //mid is median
        } else if ((first < last && last < mid) ||
                (mid < last && last < first)) {
            return last; //last is median
        } else {
            return first; //first is median
        }
    }

    /**
     * Swaps two elements in the array
     * @param first the index of the first element to swap
     * @param second the index of the second element to swap
     */
    private void swap(int first, int second) {
        int temp = arr[first];
        arr[first] = arr[second];
        arr[second] = temp;
    }

    /**
     * Performs an iterative selection sort
     */
    private void selectionSort() {
        for (int i=begin; i<end; i++) {
            int min = i;
            for (int j=i+1; j<=end; j++) {
                if (arr[j] < arr[min]) {
                    min = j;
                }
```

```
            }
            swap(i, min);
        }
    }
}
```

MergeSorter.java

```
package cs149project;

import java.util.Arrays;
import java.util.concurrent.*;

/**
 * Performs a mergesort on an array using RecursiveAction
 * @author Don
 */
public class MergeSorter extends RecursiveAction {
    private int begin, end, thresh;
    protected int[] arr;

    //FOR TESTING PURPOSES ONLY. If true, causes a few extra
statements to print showing threads.
    protected static boolean testMultiThreading;

    /**
     * Constructor.
     * @param begin the starting index to sort from
     * @param end the ending index to stop sorting inclusive
     * @param arr the array to sort
     * @param threshold the threshold to apply a simpler
sorting algorithm
     */
    public MergeSorter(int begin, int end, int[] arr, int
threshold) {
        this.begin = begin;
        this.end = end;
        this.arr = arr;
        this.thresh = threshold;
        if (this.thresh < 1) {
            this.thresh = 1;
        }
    }

    @Override
```

```
    protected void compute() {
        if (testMultiThreading) {
            System.out.println("Thread " +
Thread.currentThread().getId() + " sorting from " + this.begin
+ " to " + this.end);
        }
        if (end - begin < thresh) {
            selectionSort();
            if (testMultiThreading) {
                System.out.println("Thread " +
Thread.currentThread().getId() + " finished sorting from " +
this.begin + " to " + this.end);
            }
        } else {
            //Begin by doing "sort" portion of mergesort by
splitting array in half
            int mid = (end - begin)/2+begin;
            MergeSorter ms1 = new MergeSorter(begin, mid, arr,
thresh);
            MergeSorter ms2 = new MergeSorter(mid+1, end, arr,
thresh);
            ms1.fork();
            ms2.fork();
            ms1.join();
            ms2.join();

            //Finish by merging the two halves back together
            int[] tempArr = new int[end-begin+1]; //temporary
array to store results of merge
            int half1 = begin; //tracks where we are in the
first half of the array
            int half2 = mid+1; //same as above but for second
half
            for (int i=0; i<tempArr.length; i++) {
                if (half1 == mid+1) {
                    tempArr[i] = arr[half2];
                    half2++;
                } else if (half2 == end+1) {
                    tempArr[i] = arr[half1];
                    half1++;
                } else if (arr[half1] < arr[half2]) {
                    tempArr[i] = arr[half1];
                    half1++;
                } else { //arr[half2] < arr[half1]
                    tempArr[i] = arr[half2];
                    half2++;
```

```
                }
            }
            //Write the merged temporary array to the real
array
            for (int i=0; i<tempArr.length; i++) {
                arr[begin+i] = tempArr[i];
            }
            if (testMultiThreading) {
                System.out.println("Thread " +
Thread.currentThread().getId() + " finished sorting from " +
this.begin + " to " + this.end);
            }
        }
    }

    /**
     * Swaps two elements in the array
     * @param first the index of the first element to swap
     * @param second the index of the second element to swap
     */
    private void swap(int first, int second) {
        int temp = arr[first];
        arr[first] = arr[second];
        arr[second] = temp;
    }

    /**
     * Performs an iterative selection sort
     */
    private void selectionSort() {
        for (int i=begin; i<end; i++) {
            int min = i;
            for (int j=i+1; j<=end; j++) {
                if (arr[j] < arr[min]) {
                    min = j;
                }
            }
            swap(i, min);
        }
    }
}
```

Statistics.java

```
package cs149project;
```

```
/**
 * Provides a container for statistics of a sort.
 * @author Don
 */
public class Statistics {
    public int arraySize, threshold;
    public double avgTimeQS, avgTimeMS;
    public int failedQS, failedMS;

    /**
     * Constructor with parameters to initialize values.
     * @param arraySize Size of array that was sorted
     * @param threshold Threshold value used to switch to
simpler algorithm
     * @param avgTimeQS Average run-time of quicksorts in ms
     * @param avgTimeMS Average run-time of mergesorts in ms
     * @param failedQS Number of incorrect quicksorts. Expected
to be 0
     * @param failedMS Number of incorrect mergesorts. Expected
to be 0
     */
    public Statistics(int arraySize, int threshold, double
avgTimeQS, double avgTimeMS, int failedQS, int failedMS) {
        this.arraySize = arraySize;
        this.threshold = threshold;
        this.avgTimeQS = avgTimeQS;
        this.avgTimeMS = avgTimeMS;
        this.failedQS = failedQS;
        this.failedMS = failedMS;
    }
}
```

QuickSorterTest.java

```
package cs149project;

import java.util.Arrays;
import static org.junit.jupiter.api.Assertions.*;

/**
 * Unit test for QuickSorter to ensure accuracy of test
 * @author Don
 */
public class QuickSorterTest {
```

```
    /**
     * Test of compute method, of class QuickSorter.
     */
    @org.junit.jupiter.api.Test
    public void testCompute() {
        System.out.println("QuickSorter compute");
        final int[] testThresh = new int[] {0, 1, 2, 3, 4};
        final int[][] testCases = new int[][] {
            {91, 178, 12, 132, 48, 38, 147, 172, 145, 60, 188,
6, 194, 43, 35, 46, 24, 32, 97, 13},
            {157, 177, 59, 80, 76, 132, 138, 78, 186, 72, 137,
34, 190, 100, 45, 193, 33, 31, 51, 11},
            {51, 174, 3, 37, 112, 10, 130, 59, 25, 122, 33, 20,
195, 104, 68, 61, 44, 34, 46, 92},
            {40, 137, 157, 21, 67, 125, 18, 123, 38, 169, 76,
168, 200, 65, 166, 116, 115, 107, 56, 52},
            {166, 125, 9, 61, 2, 85, 14, 168, 183, 136, 37,
151, 101, 96, 146, 11, 45, 177, 119, 97},
            {-166, -125, -9, -61, -2, -85, -14, -168, -183,
-136, -37, -151, -101, -96, -146, -11, -45, -177, -119, -97},
            {-166, -125, -9, 61, -2, -85, -14, 168, -0, 136,
37, -151, -101, 96, -146, 11, -45, -177, -119, -97},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0},
            {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16, 17, 18, 19, 20},
            {20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8,
7, 6, 5, 4, 3, 2, 1},
            {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 0, 12, 13, 14, 15,
16, 17, 18, 19, 20},
            {1, 2, 3, 4, 5, 6, 7, 8, 9, -1, 0, 12, 13, 14, 15,
16, 17, 18, 19, 20},
            {1},
            {}
        };
        int[][] expectedResults = new int[testCases.length][];
        for (int i=0; i<testCases.length; i++) {
            expectedResults[i] = testCases[i].clone();
            Arrays.sort(expectedResults[i]);
        }

        //The test itself. Test each threshold value for each
array
        for (int i=0; i<testCases.length; i++) {
            System.out.println("Input array: " +
```

```
Arrays.toString(testCases[i]));
            System.out.println("Expected output: " +
Arrays.toString(expectedResults[i]));
            for (int j=0; j<testThresh.length; j++) {
                QuickSorter instance = new QuickSorter(0,
testCases[i].length-1, testCases[i].clone(), testThresh[j]);
                instance.compute();
                assertArrayEquals(expectedResults[i],
instance.arr);
                System.out.println("    Results with threshold
" + testThresh[j] + ": " + Arrays.toString(instance.arr));
            }
            //Also test with threshold size equal to the array
and greater than the array
            QuickSorter instance = new QuickSorter(0,
testCases[i].length-1, testCases[i].clone(),
testCases[i].length);
            instance.compute();
            assertArrayEquals(expectedResults[i],
instance.arr);
            System.out.println("    Results with threshold " +
testCases[i].length + ": " + Arrays.toString(instance.arr));

            instance = new QuickSorter(0,
testCases[i].length-1, testCases[i].clone(),
testCases[i].length+1);
            instance.compute();
            assertArrayEquals(expectedResults[i],
instance.arr);
            System.out.println("    Results with threshold " +
(testCases[i].length+1) + ": " +
Arrays.toString(instance.arr));
            System.out.println();
        }
        System.out.println("QuickSorter compute complete\n");

        System.out.println("QuickSorter multithreaded test");
        int[] multithreadTestCase = new int[] {5, 1, 5, 2, 7,
2, 10, 23, 15, 11, 21, 19, 6, 30, 26, 17, 13, 29, 6, 14};
        QuickSorter instance = new QuickSorter(0,
multithreadTestCase.length-1, multithreadTestCase, 2);
        QuickSorter.testMultiThreading = true;
        instance.fork();
        instance.join();
        System.out.println("QuickSorter multithreaded
complete");
```

```
    }
}
```

MergeSorterTest.java

```
package cs149project;

import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
import java.util.Arrays;

/**
 * Unit test for MergeSorter to ensure accuracy of test
 * @author Don
 */
public class MergeSorterTest {
    /**
     * Test of compute method, of class MergeSorter.
     */
    @Test
    public void testCompute() {
        System.out.println("MergeSorter compute");
        final int[] testThresh = new int[] {0, 1, 2, 3, 4};
        final int[][] testCases = new int[][] {
            {91, 178, 12, 132, 48, 38, 147, 172, 145, 60, 188,
6, 194, 43, 35, 46, 24, 32, 97, 13},
            {157, 177, 59, 80, 76, 132, 138, 78, 186, 72, 137,
34, 190, 100, 45, 193, 33, 31, 51, 11},
            {51, 174, 3, 37, 112, 10, 130, 59, 25, 122, 33, 20,
195, 104, 68, 61, 44, 34, 46, 92},
            {40, 137, 157, 21, 67, 125, 18, 123, 38, 169, 76,
168, 200, 65, 166, 116, 115, 107, 56, 52},
            {166, 125, 9, 61, 2, 85, 14, 168, 183, 136, 37,
151, 101, 96, 146, 11, 45, 177, 119, 97},
            {-166, -125, -9, -61, -2, -85, -14, -168, -183,
-136, -37, -151, -101, -96, -146, -11, -45, -177, -119, -97},
            {-166, -125, -9, 61, -2, -85, -14, 168, -0, 136,
37, -151, -101, 96, -146, 11, -45, -177, -119, -97},
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0},
            {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16, 17, 18, 19, 20},
            {20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8,
7, 6, 5, 4, 3, 2, 1},
            {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 0, 12, 13, 14, 15,
```

```
16, 17, 18, 19, 20},
            {1, 2, 3, 4, 5, 6, 7, 8, 9, -1, 0, 12, 13, 14, 15,
16, 17, 18, 19, 20},
            {1},
            {}
        };
        int[][] expectedResults = new int[testCases.length][];
        for (int i=0; i<testCases.length; i++) {
            expectedResults[i] = testCases[i].clone();
            Arrays.sort(expectedResults[i]);
        }

        //The test itself. Test each threshold value for each
array
        for (int i=0; i<testCases.length; i++) {
            System.out.println("Input array: " +
Arrays.toString(testCases[i]));
            System.out.println("Expected output: " +
Arrays.toString(expectedResults[i]));
            for (int j=0; j<testThresh.length; j++) {
                MergeSorter instance = new MergeSorter(0,
testCases[i].length-1, testCases[i].clone(), testThresh[j]);
                instance.compute();
                assertArrayEquals(expectedResults[i],
instance.arr);
                System.out.println("    Results with threshold
" + testThresh[j] + ": " + Arrays.toString(instance.arr));
            }
            //Also test with threshold size equal to the array
and greater than the array
            MergeSorter instance = new MergeSorter(0,
testCases[i].length-1, testCases[i].clone(),
testCases[i].length);
            instance.compute();
            assertArrayEquals(expectedResults[i],
instance.arr);
            System.out.println("    Results with threshold " +
testCases[i].length + ": " + Arrays.toString(instance.arr));

            instance = new MergeSorter(0,
testCases[i].length-1, testCases[i].clone(),
testCases[i].length+1);
            instance.compute();
            assertArrayEquals(expectedResults[i],
instance.arr);
            System.out.println("    Results with threshold " +
```

```
(testCases[i].length+1) + ": " +
Arrays.toString(instance.arr));
            System.out.println();
        }
        System.out.println("MergeSorter compute complete\n");

        System.out.println("MergeSorter multithreaded test");
        int[] multithreadTestCase = new int[] {5, 1, 5, 2, 7,
2, 10, 23, 15, 11, 21, 19, 6, 30, 26, 17, 13, 29, 6, 14};
        MergeSorter instance = new MergeSorter(0,
multithreadTestCase.length-1, multithreadTestCase, 2);
        MergeSorter.testMultiThreading = true;
        instance.fork();
        instance.join();
        System.out.println("MergeSorter multithreaded
complete");
    }
}
```

parameters.properties

```
#Parameters used to run the sorting algorithm
#The _increment properties are how much is added to the
corresponding property with each iteration.
iterations=10
sorts_per_iteration=10000
array_size=10000
array_increment=10000

#How small the array has to be before using a simpler sorting
algorithm
#NOTE: All threshold values will be tested for each array size,
multiplying the number of iterations
threshold=20
threshold_increment=20
threshold_iterations=4
```