Variation in Best-Case and Worst-Case Performance for Sorting Algorithms

Kirby McMaster
kmcmaster@weber.edu

Brian Rague
brague@weber.edu

Weber State University
Ogden UT

Samuel Sambasivam
ssambasivam@apu.edu
Azusa Pacific University
Azusa, CA

Stuart Wolthuis
stuartlw@byuh.edu
Brigham Young University – Hawaii
Laie, HI

Abstract

This research examines variation in performance for sorting algorithms in "best-case" and "worst case" situations. Our primary focus is on central tendency and dispersion of execution times for six sorting algorithms. Arrays containing fixed integer sequences are initialized in either increasing (already sorted, representing best-case) or decreasing (reverse sorted, representing worst-case) order. Our Java simulation program replicates each sorting algorithm thousands of times for each case with various array sizes. We use the simulation results to describe how sort-time distributions change across the design region.

Keywords: algorithm, sorting, performance, variation, best-case, worst-case, simulation, Java.

1. INTRODUCTION

Algorithms and data structures have been central concepts throughout the history of computer programming education (Knuth, 1997; Wirth, 1976). In current data structures and algorithms courses, the two concepts are combined into classes, with objects representing data structures and methods implementing algorithms (Koffman & Wolfgang, 2010; Lafore, 2003).

Algorithms textbooks spend a substantial amount of time explaining the performance of algorithms relative to the size of the problem (Cormen, Leiserson, Rivest, & Stein, 2009; Sedgewick & Wayne, 2011). For example, the execution time of a sorting algorithm for an array is related to the size of the array (order-of-growth). If the algorithm is repeated for a given size array, the result will be a distribution of execution times, due primarily to variation in the data to be sorted.

Algorithms textbooks define mathematical models for performance based on expected times for sorting *random* data. Some discussion is often added for "best-case" and "worst-case" situations. In effect, the distribution of execution times is described by three numbers--the average (expected value), the minimum, and the maximum. Measures of dispersion for distributions are rarely presented. Yet, variation in execution times can be an important property of an algorithm, especially in situations where strict time constraints exist and predictability is required (Jones, 2009).

This research examines variation in execution-time distributions for sorting algorithms. A previous study (McMaster, et al, 2015) described this variation when the data to be sorted was randomly generated. We now look at variation in best-case and worst-case situations. Because best-case and worst-case are relative to the sorting algorithm, we consider arrays already sorted in *increasing* order as representing best-case. Conversely, we consider arrays already sorted in *decreasing* order as representing worst-case.

In this study, for each array size and specified sorting algorithm, we replicate a Java implementation of the algorithm on a *fixed* set of array values (increasing or decreasing) to generate samples of execution times. We then describe and characterize the sort-time sample distributions.

Since the data to be sorted does not change across replications, we might expect the execution times within a sample to be identical. This does not happen, due to other sources of variation in our experiment. One observable effect on variation is the behavior of the Java run-time environment, especially automatic garbage collection.

2. RESEARCH METHODOLOGY

Our focus in this paper is on patterns of variation in execution times due to features of sorting algorithms. To minimize performance effects from hardware and lower software layers, we ran all results on a single computer, operating system, and version of the Java compiler and run-time. We restrict our study to sorting algorithms for arrays of 32-bit integers.

Research Design
Our experimental design includes three main factors: sorting algorithm, array size, and data sequence.

1. Sorting algorithm: We chose three "slow" algorithms and three "fast" algorithms. The three slow algorithms are selection sort, insertion sort, and bubble sort. The three fast algorithms are merge sort and two versions of quicksort. The quick sort algorithms differ in their choice of pivot element.

2. Array size: The array size N ranged from 400 to 2400 in increments of 400.

3. Data sequence: Three data sequences were used. In the first sequence, the N data values *increased* from 1 to N (best-case). In the second sequence, the N data values *decreased* from N to 1 (worst-case). For comparison purposes, a third sequence consisting of N random positive integers was included. This sequence was fixed over all replications for a given algorithm and array size. A copy of the array containing the initial random sequence was used in each replication.

For each factor combination, an initial 50 replications were performed to "warm-up" the Java run-time (Boyer, 2008; Wicht, 2011), effectively leveraging all optimizations afforded by JIT compilation. The warm-up execution times were discarded. Then R (= 10,000) experimental replications were run, yielding a sample of R sort times for further analysis.

Execution times were measured using the Java nanoTime function. The values of this function are in nanoseconds, but the actual clock resolution is coarse. According to Oracle's Java documentation (Oracle, 2014), the nanoTime method "returns the current value of the most precise available system timer, in nanoseconds." This value is system dependent.

For a given hardware/software environment, sorting algorithm, and array size, we expect the distribution of execution times to result from two components--normal variation due to randomness of the data, and other sources of variation that occasionally produce outliers.

Having three unchanging input data sequences eliminates the normal variation. All remaining variation is presumed to be due to other sources. In this study, we observe the pattern of the remaining variation, but we do not attempt to identify all of the sources.

Data Collection

Our Java simulation program performs the following steps for a selected sorting algorithm, array size, and data sequence:

1. Choose a desired number of warm-up and experimental algorithm repetitions.
2. For each repetition:
   a. Fill the data array with the specified data sequence.
   b. Sort the array using the selected algo rithm.
   c. For experimental repetitions, place the execution time (measured using the Java nanoTime function) in a SortTime array.
3. After all experimental repetitions are completed, calculate various statistical summaries of the execution times.

Sample Case

The following sample case demonstrates some issues we faced in developing our research methodology. In this sample case, the sorting algorithm is bubble sort, the array size is 400, and the initial array values are increasing (best-case). The number of experimental repetitions is 10000. A frequency distribution of the 10000 sort times obtained from our Java program is shown in Table 1A.

Table 1A: Bubble Sort Distribution.
Initial array contains *increasing* values.
Sort Time is in nanoseconds (ns)
Array Size N = 400, Repetitions R = 10000

Time	Freq	CumFreq	Diff
932	19	19	---
933	8232	8251	1
1399	873	9124	466
1400	866	9990	1
1866	6	9996	466
2333	1	9997	467
3265	1	9998	932
13062	1	9999	9797
1907977	1	10000	1894915

Several interesting features of this distribution can be noted:

1. The execution times are not constant, although there are many repeat values. Only nine distinct values appear in the 10000 repetitions of the sorting algorithm. Over 99 percent of the sample is contained in the four smallest execution times.

2. The median sort time (933) is almost identical to the minimum (932), and the median equals the mode.

3. The maximum sort time (1907977) is more than 2000 times larger than the median. This largest value is clearly an outlier, but are there other outliers?

4. The mean sort time (1207) is almost 30 percent larger than the median. The distribution is *positively- skewed*, even without the highest value.

5. The standard deviation of the sort times (19071) is almost 16 times larger than the mean. This measure of dispersion is greatly inflated by outliers, since the formula involves squares of deviations.

6. The smaller sort times appear in "pairs", differing by 1 nanosecond. This could be due to rounding, since the nanoTime function returns an integer.

If we consider each pair differing by 1 as a single value, we can restate the sort-time distribution in the form of Table 1B.

Table 1B: Bubble Sort Distribution.
Initial array contains *increasing* values.
Sort Time is in nanoseconds (ns) and equivalent clock ticks.
Time pairs combined when Diff = 1.
Array Size N = 400, Repetitions R = 10000

Time	Tick	Freq	CumFreq
933	2	8251	8251
1400	3	1739	9990
1866	4	6	9996
2333	5	1	9997
3265	7	1	9998
13062	28	1	9999
1907977	4090	1	10000

In this table, only seven distinct values appear in the distribution. Over 99 percent of the distribution is concentrated in the two smallest sort-time pairs. Execution times are not constant, but variation is remarkably small, except for outliers. We note that this distinctive pattern does not hold for all algorithm/array size/data sequence combinations.

When pairs that differ by 1 are combined, the smallest difference between consecutive pairs is 466 or 467. We can interpret this difference as the resolution of our nanoTime clock (approximately 466.5). In tests on other computers, we observed nanoTime clock increments between 450ns and 550ns.

Due to the limited resolution of the nanoTime function, we include restated sort times in *clock-tick* units in Table 1B. These values are obtained by dividing the nanosecond time by the estimated clock increment of 466.5 (rounding slightly). Expressing execution times in clock-tick units is a reminder of the inaccuracy of our clock. In addition, this smaller unit of measurement more clearly reveals relationships in the data.

Summary Statistics
We now address an important question for our methodology. "What statistical measures should we use to summarize the fixed data sequence variation in our sort time distributions?" Two characteristics of distributions are of particular interest:

1. *Central tendency*: Where is the "center" of the distribution?

2. *Dispersion*: How widely spread are the values from the central value?

Central Tendency: Given a sorting algorithm, array size, and data sequence, we estimate the center of the distribution of sort times, ignoring outliers. Our two candidate statistics are the *trimmed mean* and the *median*. For trimmed means, we remove the upper and lower 5% and 10% of scores. Either of these trimming choices removes almost all outliers.

However, many of our distributions are concentrated in very few distinct values (as in our sample case). Trimming these distributions removes many "non-outlier" values, especially since the outliers appear on the high end of the distribution. Another problem is that the mean value will usually be between two consecutive clock ticks, which is less intuitive.

The median is a measure where all values except the middle score (or two middle scores) are removed. The median is robust, since it does not depend on the shape of the distribution. We also found in our samples that the correlation between the median and the trimmed mean is close to 1.0. Therefore, we decided to use the median to express central tendency in our data.

Dispersion: Our main candidates to describe dispersion of sort times are a trimmed standard deviation and a centile range. The standard deviation, even with trimming, is a volatile statistic. It is also difficult to interpret for our non-normal discrete distributions.

The most common centile range is the *inter-quartile range* (IQR). This statistic describes the spread of the middle 50% of the distribution (75th centile minus 25th centile). This statistic is easy to interpret. However, in our distributions the middle 50% of the scores sometimes all had the same value. For this case, the IQR measure would be 0.

We wanted a more sensitive measure of dispersion than IQR. After some trial and error, we chose the range between the 90th centile and the minimum value (0th centile). This centile range never removes low outliers (since distributions are positively skewed). However, in our sort-time samples, the 90th centile and several centiles lower than 90th are often the same value.

3. SORT-TIME CENTRAL TENDENCY

In this section, we summarize our central tendency results for the six sorting algorithms. The medians for array size and data sequence combinations are presented in Table 2A through Table 2F, with one table per algorithm.

Selection Sort
Table 2A displays the median execution times for the selection sort algorithm.

Table 2A: Selection Sort.
Median (Clock Ticks), 10000 Reps.

Size	Incr	Decr	Random
400	329	333	375
800	1284	1292	1398
1200	2864	2877	3053
1600	5068	5086	5339
2000	7898	7921	8253
2400	11380	11380	11812

The tables contain a column for each data sequence (increasing, decreasing, and random). Each row gives the median sort times for one array size. The medians are stated in clock tick units. Cell sample sizes are 10,000 replications.

In this table, for array size 400 with an increasing data sequence, 329 clock ticks corresponds to 153,478 ns (1 clock tick = 466.5 ns). The columns of medians are almost identical, indicating that the different data sequences have little effect on the resulting execution times.

The pattern of similar columns is not a surprise, considering the nature of the selection sort algorithm. In this algorithm, each main loop places the smallest remaining value in its correct position. Most of the work in the loop is comparing data values with the current candidate for smallest value. The order of the data in the array has little effect on the number of comparisons.

We observe that the empirical order-of-growth for each column is approximately N^2. For the increasing data sequence column, array size 400 has median sort time 329, and size 800 has median sort time 1284. Doubling the array size makes sorting take about four times longer. The same relationship appears throughout the table. This is why selection sort is classified as a slow algorithm.

Insertion Sort
The median execution times for insertion sort are given in Table 2B.

Table 2B: Insertion Sort.
Median (Clock Ticks), 10000 Reps.

Size	Incr	Decr	Random
400	3	323	168
800	6	1270	633
1200	10	2846	1388
1600	13	5049	2550
2000	16	7880	4072
2400	19	11340	5658

The increasing and random column patterns are quite different from selection sort. The execution times for increasing data sequences are very small--just a few clock ticks. In addition, the sort-time medians are a linear function of array size for this data sequence.

Insertion sort places each value in the appropriate order among the previously sorted values to the left. For increasing data sequences, each value is compared only once and does not have to be moved beyond any of the values on its left. Insertion sort is fast for increasing data.

Decreasing and random sequences require more comparing and moving to reposition the values. Note in Table 2B that sorting a random sequence is about twice as fast as sorting a decreasing sequence. With a decreasing sequence, each value must be moved past all of the values to its left during each iteration of the algorithm. For random sequences, on average, each value needs to be moved past about half of the values on its left.

Bubble Sort
Bubble sort median execution times are listed in Table 2C.

Table 2C: Bubble Sort.
Median (Clock Ticks), 10000 Reps.

Size	Incr	Decr	Random
400	2	496	645
800	4	1962	2477
1200	6	4395	5696
1600	8	7827	9148
2000	10	12184	14327
2400	12	17649	20702

The small bubble sort times for increasing sequences are comparable to insertion sort. Bubble sort makes repeated passes through the data, comparing consecutive values and swapping as necessary. A pass with no swaps terminates the algorithm. An array containing an increasing data sequence requires only one pass. Bubble sort is fast if the array is already sorted in increasing order.

Decreasing and random sequences require multiple passes. Decreasing sequences appear to require more passes, since the execution times are longer. Bubble sort is very slow in both cases.

Merge Sort
The recursive merge sort algorithm results are presented in Table 2D.

Table 2D: Merge Sort.
Median (Clock Ticks), 10000 Reps.

Size	Incr	Decr	Random
400	57	62	90
800	118	131	214
1200	186	207	344
1600	254	283	489
2000	319	363	632
2400	399	447	791

Sort times for increasing and decreasing data sequences are almost twice as "fast" as for random sequences. The sort times are relatively fast when compared to prior algorithms. Each doubling of the array size gives an execution time just over twice as large. Order-of-growth appears to be slightly larger than N.

Quick Sort - Low Pivot
Quicksort has several variations. Most differ in how the pivot element is chosen during each recursive function call. The version we summarize in Table 2E chooses the pivot at the *low* end of each interval to be recursively sorted. This version is described in Sedgewick and Wayne (2011).

Table 2E: Quick Sort (Low Pivot).
Median (Clock Ticks), 10000 Reps.

Size	Incr	Decr	Random
400	254	253	20
800	977	977	55
1200	2170	2169	131
1600	3831	3821	213
2000	5960	5960	301
2400	8562	8558	379

How the pivot is chosen is immaterial when the data to be sorted is random. This version of quicksort is a fast algorithm for random data. The algorithm becomes much slower if the data is already sorted (increasing or decreasing).

The speed of quicksort depends on how well the pivot elements divide each interval of data into two approximately equal halves. That ability works fairly well for random data. However, for sorted data, choosing a pivot at one end (lower or upper) divides the interval into a single slightly smaller one.

When this happens, the number of recursive call levels is linear instead of logarithmic. This example illustrates that the specific software implementation of an algorithm can have a substantial effect on performance.

Quick Sort - Mid Pivot
In Table 2F, we show the median sort times for a quicksort algorithm that chooses the pivot element from the *middle* of each interval to be divided.

Table 2F: Quick Sort (Mid Pivot).
Median (Clock Ticks), 10000 Reps.

Size	Incr	Decr	Random
400	19	21	23
800	39	45	70
1200	61	69	159
1600	87	95	253
2000	101	119	338
2400	128	147	434

This mid-pivot method has less overhead than the "median-of-three" approach. We simply calculate the middle index for the interval and use the value at that position as the pivot element (swapping as necessary).

This is a fast sorting algorithm for all three data sequences. The execution-time growth versus array size is close to linear for sorted data. A slightly higher growth rate occurs for random data.

From the central tendency patterns in the preceding tables, we conclude that two of the sorting algorithms have different reactions to best-case compared to worst-case inputs. This applies to insertion sort and bubble sort, which have very small median sort-times for best-case, but much larger sort times for worst-case.

The median execution time patterns for increasing versus decreasing data sequences are similar for selection sort and quicksort-low (large sort times) and merge sort and quicksort-mid (small sort times).

4. SORT-TIME DISPERSION

This section on sort-time dispersion represents a unique portion of our study, since this topic is rarely discussed in the literature. The previous results on central tendency provide empirical support for familiar order-of-growth models for sorting algorithms, especially the simulation data for best-case and worst-case sequences.

We now describe the pattern of dispersion in execution-time samples for all of the factor combinations in the design. The results are presented in Tables 3A through 3F, with each table containing dispersion measures for one sorting algorithm.

Our measure of dispersion in these tables is a specific *centile range*, the difference between the 90th centile (C90) and the minimum sort time. These centile values are transformed from nanoseconds to clock-tick units by dividing by the clock increment of 466.5. We then add 1 to the difference so that the unit of measurement becomes *clock intervals*. The dispersion measure is the number of adjacent clock intervals needed to contain the lower 90 percent of the sort times. If all of these sort times are identical, the measured dispersion is 1 clock interval (not zero).

Selection Sort
Table 3A displays the C90 centile ranges of execution times for the selection sort algorithm. As stated above, the unit of measurement is clock intervals, not clock ticks.

Table 3A: Selection Sort.
C90 - Min (Clock Intervals), 10000 Reps.

Size	Incr	Decr	Random
400	2	2	4
800	6	2	8
1200	9	14	27
1600	8	34	38
2000	35	43	51
2400	60	48	61

For selection sort, the dispersion values across all data sequences range from a low of 2 clock intervals to a high of 61. The smallest values appear for small arrays that contain increasing or decreasing data sequences. The column pattern shows that dispersion increases for larger array sizes. We should not take the exact table values literally. The centile range statistic exhibits some inherent volatility in repeated sampling, in spite of our large sample sizes. Centile ranges are less stable than medians in our simulation environment.

Insertion Sort
The sort-time dispersion statistics for the insertion sort algorithm are listed in Table 3B.

Table 3B: Insertion Sort.
C90 - Min (Clock Intervals), 10000 Reps.

Size	Incr	Decr	Random
400	2	2	2
800	2	3	2
1200	2	34	3
1600	2	36	32
2000	2	48	35
2400	2	62	35

The dispersion pattern of all 2's for increasing data sequences at all array sizes (in our design range) stands out. In these samples, the lower 90 percent of the sort times fall within two clock intervals, suggesting that the algorithm execution times are almost constant. Remember from Table 2B, the medians for increasing sequences are also quite small (between 3 and 19).

For decreasing and random data sequences, the dispersion increases for larger array sizes. The medians are also much larger (up to 11340). Dispersion for decreasing sequences is larger than for random data, which parallels the pattern for medians.

Bubble Sort
Table 3C summarizes the dispersion measures for the bubble sort algorithm.

Table 3C: Bubble Sort.
C90 - Min (Clock Intervals), 10000 Reps.

Size	Incr	Decr	Random
400	2	6	16
800	1	31	56
1200	3	49	91
1600	2	79	132
2000	3	86	187
2400	3	139	276

For bubble sort, dispersion is essentially constant for increasing data sequences, due to small medians (12 or less). For decreasing and random sequences, the growth of dispersion with array size is more pronounced than for the previous two algorithms. The growth of medians is also much larger (up to 20702 for random data). Dispersion for random sequences is twice as large as for decreasing data (opposite the pattern for insertion sort).

Merge Sort
Dispersion measures for merge sort are presented in Table 3D.

Table 3D: Merge Sort.
C90 - Min (Clock Intervals), 10000 Reps.

Size	Incr	Decr	Random
400	6	6	9
800	8	7	12
1200	9	11	13
1600	13	12	19
2000	17	17	23
2400	22	17	37

Merge sort displays relatively small dispersion for all three data sequences. The rate of growth with array size is less than in the previous algorithms. This suggests that merge sort execution times are stable over a variety of data sets. In our study, the largest median for merge sort was 791 (for random data). Dispersion for random sequences is slightly larger than for increasing and decreasing data.

Quick Sort - Low Pivot
Sort-time centile ranges for the low-pivot quicksort algorithm are shown in Table 3E.

Table 3E: Quick Sort (Low Pivot).
C90 - Min (Clock Intervals), 10000 Reps.

Size	Incr	Decr	Random
400	2	2	3
800	3	3	7
1200	15	4	12
1600	35	17	17
2000	60	51	15
2400	58	45	15

The low-pivot version of quicksort has a small amount of dispersion for random data sequences. The growth of dispersion is larger for sorted data but is comparable to selection sort. Dispersion for random sequences remains fairly small (perhaps due to smaller medians). Apparently, the poor choice of a pivot penalizes median sort time (as large as 8562 for increasing data) much more than dispersion.

Quick Sort - Mid Pivot
Dispersion measures for the mid-pivot quicksort algorithm are given in Table 3F.

Table 3F: Quick Sort (Mid Pivot).
C90 - Min (Clock Intervals), 10000 Reps.

Size	Incr	Decr	Random
400	2	3	3
800	2	2	11
1200	2	4	14
1600	3	4	14
2000	3	5	14
2400	2	2	16

For the mid-pivot quicksort algorithm with increasing and decreasing data sequences, dispersion is small and almost constant. For random sequences, dispersion starts small and grows at a low rate (with a largest median of 434). The choice of pivot element effects dispersion in both quicksort versions, but in different ways for sorted data.

Dispersion is small for all sorting algorithms when the array size is small. This is true for both best-case and worst-case data. With increasing data sequences, dispersion remains constant across all array sizes for insertion sort, bubble sort, and mid-pivot quicksort. For decreasing sequences, only mid-pivot quicksort has constant dispersion for larger array sizes.

5. SUMMARY AND CONCLUSIONS

The primary purpose of this study was to analyze best-case and worst-case variation in the performance of sorting algorithms. Because best-case and worst-case depend on the algorithm, we represented best-case as an increasing sequence of data values. Worst-case was represented by a decreasing sequence of values.

Most of the emphasis in algorithm textbooks is on expected performance for sorting random data. We are more interested in the distribution of execution times when an algorithm is run multiple times with the *same* data (best-case or worst-case).

We designed a methodology to control hardware, operating system, and Java runtime effects. We wanted processing time variation to result primarily from three main factors--the sorting algorithm, the size of the array, and the initial data sequence.

We wrote a Java simulation program to repeatedly fill an array of size N with a fixed data sequence (increasing, decreasing, or random), sort it using a selected algorithm, and record the execution times. The sort-time data was then used to calculate statistics that summarize central tendency and dispersion.

Our experiment was performed for six sorting algorithms: selection sort, insertion sort, bubble sort, merge sort, and two versions of quicksort. For each algorithm, a range of array sizes were examined. The findings for each algorithm were reported in the previous two sections of this paper.

We reorganized the execution-time summary statistics (medians and centile ranges) for increasing (best-case) versus decreasing (worst-case) data sequences into four tables in the Appendix. Some conclusions from this rearranged summary are listed below.

1. In Table 4A (increasing sequences), the smallest median sort-time is 2 clock *ticks* (with 466.5 ns per clock tick) for bubble sort with array size 400. In this case, the replicated execution times are almost constant. The largest median sort-time is 11353 for selection sort with array size 2400.

The median growth rate is almost linear for insertion sort, bubble sort, and quicksort with mid-interval pivot elements (quicksort-mid).

2. In Table 4B (decreasing sequences), the smallest median sort-time is 21 clock ticks for quicksort-mid. The largest median sort-time is 17649 for bubble sort with array size 2400. Only quicksort-mid and merge sort perform well for decreasing data sequences. The median growth rate for these two algorithms is slightly above linear.

3. In Table 5A (increasing sequences), the smallest 90th centile range is 1 clock *interval* for bubble sort with array size 800. For array size 400, five of the algorithms (all but merge sort) have a centile range value of 2 clock intervals. In these cases, the sort-time dispersion is very small. In fact, for insertion sort, bubble sort, and quicksort-mid, the dispersion is almost constant for all array sizes.

The largest 90th centile ranges are 60 for selection sort and 58 for quicksort with low-endpoint pivot elements (quicksort-low), when the array size is 2400. Compared to the medians of the distributions, this is not a large amount of dispersion. Remember that the execution times within each distribution are based on initial data that does not change.

4. In Table 5B (decreasing sequences), the smallest 90th centile range is 2 clock intervals. These values occur for selection sort, insertion sort, quicksort-low, and quicksort-mid for various array sizes. The dispersion is almost constant across array sizes only for quicksort-mid. The largest centile range is 139 clock intervals for bubble sort with array size 2400.

Future Research

Our planned future research activities include:

1. Extend our analysis of variation to other sorting algorithms, array sizes, and data sequences.

2. Use our methodology on algorithms written in other programming languages, such as C and C++. One complication is that C and C++ compilers provide a variety of timer functions in different operating environments.

3. Study the behavior of Java's nanoTime function across different hardware and software environments.

6. REFERENCES

Boyer, Brent (2008). Robust Java benchmarking, Part 1: Issues. IBM DeveloperWorks.

Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., & Stein, Clifford (2009). *Introduction to Algorithms* (3rd ed). MIT Press.

Jones, Nigel (2009). Sorting (in) embedded systems. Stack Overflow.

Knuth, Donald (1997). *The Art of Computer Programming, Vol. 1: Fundamental Algorithms* (3rd ed). Addison-Wesley.

Koffman, Elliot, & Wolfgang, Paul (2010). *Data Structures: Abstraction and Design Using Java* (2nd ed). Wiley.

Lafore, Robert (2003). *Data Structures & and Algorithms in Java* (2nd ed). Sams Publishing.

McMaster, Kirby, Sambasivam, Samuel, Rague, Brian, & Wolthius, Stuart (2015). Distribution of Execution Times for Sorting Algorithms Implemented in Java. Proceedings of Informing Science & IT Education Conference (InSITE) 2015, 269-283.

Oracle (2014). java.lang Class System. www.docs.oracle.com

Sedgewick, Robert, & Wayne, Kevin (2011). *Algorithms* (4th ed). Addison-Wesley.

Wicht, Baptiste (2011). Java Micro-Benchmarking: How to write correct benchmarks. www.javacodegeeks.com

Wirth, Niklaus (1976). *Algorithms + Data Structures = Programs*. Prentice-Hall.

7. APPENDIX

Table 4A. Median (Clock Ticks). Increasing array values.

Size		Select	Insert	Bubble		Merge	Quick-L	Quick-M
400		329	3	2		57	254	19
800		1284	6	4		118	977	39
1200		2864	10	6		186	2170	61
1600		5068	13	8		254	3831	87
2000		7898	16	10		319	5960	101
2400		11353	19	12		399	8562	128

Table 4B. Median (Clock Ticks). Decreasing array values.

Size		Select	Insert	Bubble		Merge	Quick-L	Quick-M
400		333	323	496		62	253	21
800		1292	1270	1962		131	977	45
1200		2877	2846	4395		207	2169	69
1600		5086	5049	7827		283	3829	95
2000		7921	7880	12184		363	5960	119
2400		11380	11340	17649		447	8558	147

Table 5A. C90 - Min (Clock Intervals). Increasing array values.

Size		Select	Insert	Bubble		Merge	Quick-L	Quick-M
400		2	2	2		6	2	2
800		6	2	1		8	3	2
1200		9	2	3		9	15	2
1600		8	2	2		13	35	3
2000		35	2	3		17	60	3
2400		60	2	3		22	58	2

Table 5B. C90 - Min (Clock Intervals). Decreasing array values.

Size		Select	Insert	Bubble		Merge	Quick-L	Quick-M
400		2	2	6		6	2	3
800		2	3	31		7	3	2
1200		14	34	49		11	4	4
1600		34	36	79		12	17	4
2000		43	48	86		17	51	5
2400		48	62	139		17	45	2