

Measuring Algorithm Performance With Java: Patterns of Variation

Kirby McMaster
kcmcmaster@weber.edu
Computer Science
Moravian College
Bethlehem, PA 18018, USA

Samuel Sambasivam
ssambasivam@apu.edu
Computer Science
Azusa Pacific University
Azusa, CA 91702, USA

Stuart Wolthuis
stuart.wolthuis@byuh.edu
Computer & Information Sciences
BYU-Hawaii
Laie, HI 96762, USA

Abstract

Textbook coverage of algorithm performance emphasizes patterns of growth in expected and worst case execution times, relative to the size of the problem. Variability in execution times for a given problem size is usually ignored. In this research study, our primary focus is on the empirical distribution of execution times for a given algorithm and problem size. We examine CPU times for Java implementations of four sorting algorithms: selection sort, insertion sort, bubble sort, and quicksort. We measure variation in running times for these sorting algorithms. We show how the sort time distributions change as the problem size increases. With our methodology, we compare the relative stability of performance for the different sorting algorithms.

Keywords: algorithm, sorting, performance, variation, order-of-growth, Java.

1. INTRODUCTION

The performance of algorithms is addressed at different levels throughout the computing curriculum. In introductory programming courses, informal comparisons of alternative algorithms are presented without a rigorous theoretical framework (Lewis and Loftus, 2011; Liang, 2012).

In Data Structures textbooks (Koffman & Wolfgang, 2010; Lafore, 2003), the emphasis is on how to implement algorithms to support data structures of varying complexity, such as stacks, priority queues, binary search trees, and weighted graphs. A casual introduction to "Big-Oh" notation is included to relate problem size to execution time for various types of algorithms.

In Analysis of Algorithms textbooks (Cormen, Leiserson, Rivest, & Stein, 2009), the discussion of algorithm performance places greater emphasis on mathematical reasoning. A formal examination of algorithm efficiency based on resources required (primarily CPU time) looks at best case, worst case, and average case situations.

Most of the discussion centers on worst case analysis because the mathematical arguments are simpler. *Order-of-growth* is defined to ignore constants and lower order terms, so average case results are often proportional to the worst case. Worst case examples provide an upper bound on the execution time for an algorithm.

Sedgewick & Wayne (2011) present a mathematical analysis of algorithms, and then relate their mathematical models to empirical results obtained from algorithm run times on a computer. They give several algorithms for finding three numbers (from a large input file) that sum to zero. They ran each algorithm once for each input file, assuming that the only source of variation was the actual data. However, in our research we experienced situations where repeated execution of the same algorithm on the same data resulted in different execution times.

Some textbooks briefly mention that running times can vary for different inputs. However, they include no discussion of the nature of the *distribution* of execution times for random inputs. Variation includes not only *dispersion* (how spread out the scores are from a central value), but also *skewness* (how unbalanced the scores are at each end of the distribution).

Variation can be of greater importance than averages when consistency/dependability of execution time is a major requirement. This is true in systems having strict time constraints on operations, such as manufacturing systems, real-time control systems, and embedded systems (Jones, 2009).

Research Plan

The primary objective of this research is to examine how algorithm execution time distributions depend on problem size, randomness of data, and other factors. We limit our study to sorting algorithms for arrays of integers. In the next section, we list potential sources of variation for execution times. We then describe our experimental design to control sources of variation beyond algorithm structure

and problem size. Our results and conclusions are summarized later in the paper.

2. SOURCES OF VARIATION

There are many system features which can affect algorithm performance. In this research, we use CPU time as our primary measure of performance. A layered list of sources of variation in sort times is outlined below.

1. Computer hardware components: (a) CPU clock speed, pipelines, number of cores, internal caches, (b) memory architecture, amount of RAM, interleaved RAM, external caches.
2. Operating system features: (a) process scheduling algorithms, multi-tasking, parallel processing, (b) memory allocation algorithms, virtual memory.
3. For Java programs: (a) Java JIT compiler, (b) Java run-time options, (c) Java run-time behavior, especially automatic garbage collection.
4. Application program: (a) choice of algorithm, and how it is implemented, (b) size of problem, (c) amount of memory required by the algorithm, (d) data type and data source.

Our main focus in this paper is on patterns of variation in execution times due to features in the *application program*. We limit our research to *sorting algorithms*, including selection sort, insertion sort, bubble sort, and quicksort. We examine a range of array sizes, and repeatedly fill the arrays with random integers.

To minimize algorithm performance effects from the lower hardware and software layers, we ran all final results on a single computer. This computer had an Intel Core2 Duo CPU, Windows 7 operating system, and Version 7 of the Java compiler and run-time.

Unexpected Variation

In our research environment, we assumed that algorithm execution times would depend almost entirely on:

1. the sorting algorithm
2. the size and data type of the array
3. the randomness of the generated data

Surprisingly, this assumption was *not* supported by our test data. Unexpected patterns of

variation in performance were encountered throughout our research study.

For example, early in the exploratory phase of our study, we performed the *selection* sort algorithm 7 times on an array size of 100. For each sort operation, independent random values of type *int* were generated to fill the array. The execution times in nanoseconds (ns) for the sort module were:

```
113827
320489
 16328
 15394
 14928
 14928
 14462
```

A statistical summary of CPU times to sort these arrays is:

```
Minimum = 14462
Median   = 15394
Maximum  = 320489

Mean     = 72908
Std dev  = 115195
```

Several patterns in this data can be noted:

1. The maximum sort time is more than 20 times larger than the median. This is due to the presence of *outliers* (large sort times) in the sample.
2. The median sort time is only slightly larger than the minimum.
3. The average sort time is much larger than the median, suggesting a *positively-skewed* distribution.
4. The standard deviation of the sort times is larger than the mean. This measure of variation is greatly inflated by outliers.

3. METHODOLOGY

The above example containing outliers was not atypical in our study. Because of these unexpected patterns in execution time data, we developed a *methodology* for generating and analyzing performance data that is relatively immune to outlier effects.

CPU time measurement does not provide an "exact" performance value for an algorithm. Karl Pearson theorized that measurements

represent samplings from a *probability distribution* of values (Salsburg, 2001). For example, to answer the question of "how fast is a sprinter?", his/her running times in 100-meter dash events over a season provide a partial answer in the form of a distribution of sample values.

For a given hardware/software environment, sorting algorithm, and array size, our methodology assumes that the distribution of execution times is a *mixture* of two components: (a) *normal* variation due to randomness of the data, and (b) other sources of variation that result in outliers.

Our methodology attempts to extract the normal variation component from the combined distribution. This requires being able to *detect* possible outliers and *remove* them from the sample.

Our sort time data often contained a relatively large number of outliers. Therefore, we did not perform statistical tests to detect individual outliers. Instead, we used two general approaches for removing outliers:

1. Set limits on the perceived "normal" data, and *trim* off values outside these limits. In particular, we examine trimmed means and trimmed standard deviations.
2. Use statistics such as the median that are less susceptible to outliers.

Our performance analysis approach was developed first for the *selection sort* algorithm. Samples of execution times for selection sort were obtained for a range of array sizes starting with 100.

Our Java data generation program, initially written for selection sort, performs the following steps:

1. Input the array size (N) and number of algorithm repetitions (R).
2. For each repetition:
 - a. fill the data array with random integers.
 - b. sort the array, and place the execution time (collected using the Java System nanoTime function) in a SortTime array.
3. After all repetitions are completed, sort the execution times in the SortTime array.

4. Calculate various statistical summaries of the execution times. This part of the Java program was modified frequently throughout the study.

As data were collected for the sorting algorithm, we evaluated how well different statistics summarized essential features of the sort time distributions. When the methodology began to provide consistent results for selection sort, we applied the methodology to the remaining sorting algorithms.

Sample Case

The following sample case demonstrates much of the process in developing our methodology. In this case, the array size is 100, and the number of repetitions is 1000. A frequency distribution of the 1000 sort times obtained from running our Java program once is shown below.

Table 1: Selection Sort Distribution.
Sort Time in nanoseconds (ns)
 Size N = 100, Repetitions R = 1000

SortTime	Freq	CumFreq	Diff
14461	36	36	---
14462	68	104	1
14928	472	576	466
14929	78	654	1
15394	124	778	465
15395	194	972	1
15861	17	989	466
15862	4	993	1
16328	1	994	466
17261	1	995	933
19127	1	996	1866
37320	1	997	18193
108695	1	998	71375
111028	1	999	2333
113827	1	1000	2799

Several unusual features appear in the above distribution:

1. The sample of sort times contains many repeat values. Only 15 distinct values appear in the 1000 repetitions of the sorting algorithm.
2. Among the smaller sort times, most appear in "pairs", differing only by 1 nanosecond. This is probably due to rounding, since the nanoTime function returns an integer.
3. If we consider pairs differing by 1 as a single value, over 99% of the distribution is concentrated in 4 sort time pairs.

4. Again considering pairs differing by 1 as a single value, the difference between consecutive pairs is between 466 and 467. We can interpret this difference as the resolution of the "clock tick" for our nanoTime clock. Oracle's Java documentation (Oracle, 2014) states that the System.nanoTime method "returns the current value of the most precise available system timer, in nanoseconds." Apparently, our recorded sort times are not accurate to 1 nanosecond. In tests on other computers, we observed that the clock increment is hardware specific.

5. The three largest values--113827, 111028, and 108695--are clearly outliers. But are there other outliers? The distribution is slightly skewed, even without top three values.

6. The median of the distribution is 14928, which is close to the minimum value.

We now ask the most important question for our methodology. "What characteristics of the sort time distribution are relevant for describing patterns of variation?" We will be generating sort time distributions for different sorting algorithms and various array sizes. The patterns of variation we are trying to explain should be observable within each of these separate distributions.

A related research question is: "What statistical measures best summarize the variation in sort time distributions, without being distorted by outliers?" Three characteristics of distributions are of particular interest:

1. *central tendency*: Where is the "center" of the distribution? Outliers can distort the mean of the distribution, but not the median.
2. *dispersion*: How widely spread are the values from the central value? For "normal" variation, dispersion should not be inflated by outliers.
3. *skewness*: How "unbalanced" is the distribution on both sides of the central value? Skewness can be exaggerated by outliers.

Central Tendency and Skewness

Given a sorting algorithm and an array size, we want to estimate the center of the distribution of "normal" sort times. This distribution does not include outliers. Our main statistic is the *trimmed mean*.

We must decide which scores to "trim" from the sample of sort times. We want to trim enough values so that the trimmed mean approaches

the median and is not influenced by extreme values.

In Table 2, we present several trimmed mean candidates and compare them to the median. The data is from the sample of sort times described in Table 1. The untrimmed mean is based on the entire sample, including outliers. The 99/01 trimmed mean removes the largest and smallest 1% (approximately) of the sample before calculating the mean. Other trimmed means remove the top and bottom 5%, 10%, and 20% of the sample. The median can be interpreted as the mean obtained by removing the largest and smallest 50%, but leaving the middle score(s).

Table 2: Selection Sort Trimmed Means.
 Size N = 100, Repetitions R = 1000

Trim Percent	Mean	vs. Median
Untrimmed	15367	439
99/01	15048	120
95/05	15053	125
90/10	15053	125
80/20	15099	171
50/50 (Median)	14928	-0-

Note that the median remains unchanged for all trimmed samples because we removed the same number of values from both ends of the sorted list of values. For this sample of data, removing the top and bottom 1% seems to be sufficient to remove the effect of outliers on the mean.

Dispersion

The main topic of interest in this research is patterns of variation in algorithm performance. The dispersion in the distribution of sample sort times provides a measure for performance variation. We want to determine the variation for the "normal" sort times, apart from outlier effects.

The most common measure of variation for quantitative variables is the standard deviation. However, the standard deviation is very sensitive to outliers.

As with trimmed means, we calculate standard deviations from trimmed samples, hopefully with outliers removed. Since we are not testing for individual outliers, we trim different percentages of larger and smaller values from the sample.

Standard deviations, both untrimmed and trimmed, are presented in Table 3. The sample data is again from Table 1.

Table 3: Trimmed Standard Deviations.
 Size N = 100, Repetitions R = 1000

Trim Percent	Std Devn
Untrimmed	5318
99/01	311
95/05	273
90/10	273
80/20	225
Quartile Deviation	233

It is apparent that trimming the top 1% (containing the outliers) and bottom 1% leads to a substantial reduction in the standard deviation. Additional trimming has relatively little effect on the standard deviation in this case.

The *quartile deviation* is included in Table 3 for comparison purposes. The *interquartile range* (IQR) is a well-known measure of the spread of scores in a distribution. It is defined to be difference between the third quartile Q3 (75th centile) and the first quartile Q1 (25th centile). The quartile deviation is *half* the interquartile range (IRQ/2).

Higher Repetitions

The data from Table 1 represents a sample of 1000 sort times. In the early development of our methodology, we generated samples of this size for array sizes between 100 and 1000. We performed statistical analyses on data for these sample sizes.

As we became more comfortable with our methodology, we increased the number of repetitions to 10000. Each time we ran our Java data generation program, we obtained a sorted array containing 10000 execution times. With larger samples, we got a clearer picture of the stability of our results.

In Table 4, we present a frequency distribution for one sample of 10000 sort times, based on selection sort of arrays of size 100. This distribution of 10000 values is similar to the previous distribution of 1000 values.

Table 4: Selection Sort Distribution.

Size N = 100, Repetitions R = 10000

SortTime	Freq	CumFreq	Diff
* 13995	58	58	---
* 14461	2295	2353	466
* 14928	5669	8222	467
* 15394	1655	9877	466
* 15861	84	9961	467
other	38	9999	---
1866018	1	10000	---

* Consecutive values combined
 (e.g. 13995 -- 55, 13996 -- 3)

1. The sample of sort times contains thousands of repeat values.
2. The smaller sort times appear in "pairs" that differ by 1 nanosecond (shown with asterisks). The lowest five pairs comprise over 99% of the distribution. Perhaps we need a better "clock" than the one provided by Java's nanoTime method.
3. The minimum value of 13995 is one clock tick below 14461, which was the minimum value in the smaller sample. The maximum value of 1866018 is an order of magnitude larger than the earlier maximum of 113827. In our methodology, generating random data that include large sort times is not unusual.
4. The median of this second distribution remains at 14928, which is again close to the minimum value.

4. ANALYSIS OF DATA

In this section, we analyze performance variation for four sorting algorithms: selection sort, insertion sort, bubble sort, and quicksort. For each algorithm, we examine six array sizes: 200, 400, ... , 1200. Patterns of mean variation across array sizes for a given algorithm is comparable to order-of-growth models covered in algorithm textbooks.

We extend our research to describe sort time distributions *within* each algorithm/array size combination. We measured central tendency, dispersion, and skewness for these distributions. Each test case involved 10000 repetitions of one sorting algorithm for a single array size.

Sort Time Central Tendency

We measured central tendency with trimmed means and the median. Our early work with arrays of size 100 suggested that trimming the top and bottom 1% is sufficient to remove outliers. However, for larger array sizes, the amount of variation increases. We made a conservative decision to trim the top and bottom 5% of the scores from each distribution.

Trimmed means for all six array sizes for each sorting algorithm are listed in Table 5. All times are in nanoseconds.

Table 5: Sort Time Distribution - Trimmed 95/05 Mean

Size	Select	Insert	Bubble	Quick
200	49979	21306	86374	14991
400	177308	79163	313611	32643
600	378867	173847	677236	51110
800	654887	304813	1118399	70304
1000	1004657	471508	1698413	89862
1200	1427205	674708	2415186	109896

Looking at each row separately, we see that the largest mean execution times are for bubble sort, followed by selection sort. Insertion sort are less than half the values for selection sort. Quicksort times are much smaller, especially for large array sizes.

This computer generated data is consistent with the nature of each of these sorting algorithms. For random data, bubble sort performs a large number of comparisons and swaps, while insertion sort performs many comparisons and shifts. In selection sort, the number of comparison operations is almost constant, regardless of the values in the array. The insertion sort and bubble sort algorithms can terminate early, depending on how fully sorted the data are initially. Quicksort is fastest because of its recursive design.

If we look down each column at the pattern of increasing mean execution times, the results follow traditional order-of-growth models. For selection sort, when the array size doubles (e.g. 400 -> 800), the mean sort time is approximately four times larger (177308 -> 654887). This supports an $O(N^2)$ order-of-growth model. A similar pattern occurs for insertion sort and bubble sort. Quicksort displays a noticeably smaller growth rate.

We prepared a summary table containing untrimmed means, but do not include it in this paper. With sample sizes of 10000, removing the top and bottom 5% (presumably containing outliers) had relatively little effect on the means. The trimmed means are about 1% to 2% smaller than the untrimmed means. Correlations between trimmed and untrimmed means is above 0.999 for each algorithm. As we shall see, trimming has a much greater effect on measures of dispersion.

We provide in Table 6 the medians for the sort time distributions for each algorithm/array size combination.

Table 6: Sort Time Distribution - Median

Size	Select	Insert	Bubble	Quick
200	49916	21459	86302	14928
400	177271	79305	313487	32655
600	378801	173539	677355	51314
800	654500	304627	1117730	70441
1000	1004374	471169	1698052	90034
1200	1426570	674566	2414594	110093

When the medians are compared to the trimmed means, there are minor differences, but the pattern is almost identical. This suggests that the trimming has successfully removed outliers, and the trimmed distributions are less skewed.

Sort Time Dispersion

We remind the reader that the values in the tables are not absolute. They are the results of random sampling of an algorithm. With means, the results are relatively stable, even in the presence of a small number of outliers.

The same claim cannot be made for measures of dispersion. Statistics such as the standard deviation and the range can be greatly distorted when even a few outliers are in the sample. Our main objective in this study is to characterize variation in sort time distributions. With judicious trimming, we can avoid the problem of having an unreasonable number of outliers. Even so, occasional bizarre values appeared in our data sets.

The most common measure of dispersion for a distribution is the standard deviation. To illustrate how volatile standard deviations can be with outliers, in Table 7 we present *untrimmed* standard deviations using complete samples of 10000 sort times.

In this table, untrimmed standard deviations for selection sort range in value from 6182 to 201238. Observe that increasing the array size does not always result in a larger standard deviation. The size of each standard deviation is heavily influenced by outliers. Similar irregular patterns occur for each sorting algorithm.

Table 7: Sort Time Distribution - Untrimmed Standard Deviation

Size	Select	Insert	Bubble	Quick
200	6182	10747	30301	2088
400	74580	39323	55776	3226
600	62073	34151	37991	20779
800	201238	104351	68866	36104
1000	57680	73650	43354	52899
1200	188333	32091	64344	24352

In the next table, we show how volatile variation statistics can be "tamed" with the careful use of trimming. Table 8 lists trimmed standard deviations obtained by removing the 5% largest and 5% smallest values from the sample. We chose 5% limits to be consistent with the previous trimming of means. In practice, 5% trimming might not always be enough.

Table 8: Sort Time Distribution - Trimmed 95/05 Standard Deviation

Size	Select	Insert	Bubble	Quick
200	558	761	1118	326
400	1243	2127	2838	494
600	1496	3782	3976	632
800	1712	6091	5559	794
1000	2141	8421	7520	955
1200	2852	10123	10096	1155

For the trimmed standard deviations in Table 8, the pattern in each column shows an increase in dispersion as the array size increases. These results are representative of what we *usually* obtained with 5% trimming.

The variation patterns for the four sorting algorithms is instructive. The greatest rates of increase in dispersion are for insertion sort and bubble sort. The smallest rate of increase is for quicksort.

Selection sort, as we showed in Table 5, has the second largest mean sort times. But the rate of increase in dispersion is less than for insertion and bubble sort. Why? We let the reader answer

that question. Quicksort has a lower rate of increase in dispersion than selection sort.

Coefficient of Variation

Another way of comparing dispersion among similar distributions is by measuring relative variation. In this case, we divide the trimmed standard deviation by the corresponding trimmed mean. The statistic is called the *coefficient of variation*. To make the value of the statistic easier to interpret, we multiplied it by 100, so that we express the standard deviation as a *percentage* of the mean.

Measures of relative variation for our sorting algorithms and array sizes are displayed in Figure 1. Both means and standard deviations are trimmed at the top and bottom 5% levels.

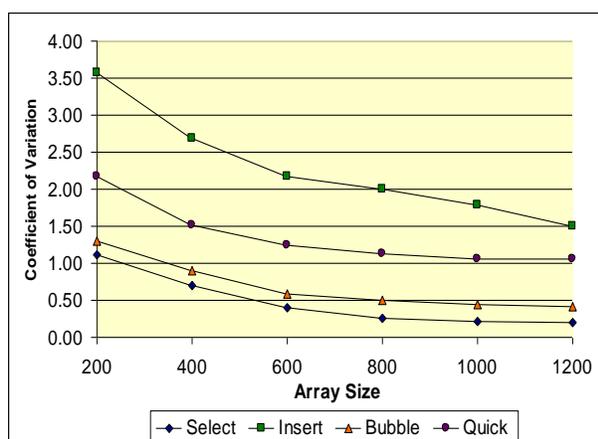


Figure 1: Sort Time Relative Variation - 95/05 Coefficient of Variation (%)

Selection sort has the smallest values for the coefficient of variation, followed closely by bubble sort. The selection sort means are more than twice as large as the times for insertion sort, but selection sort standard deviations are smaller. The result is less relative variation for selection sort.

Quicksort has smaller standard deviations and smaller means. The ratios fall in between the high and low values of the other algorithms. One interesting feature revealed by Figure 1 is that, for all four algorithms, the coefficient of variation *decreases* as the array size increases. Although the standard deviation increases for larger arrays, the mean increases at a faster rate.

It is tempting to conjecture that the ratios approach a lower limit for very large arrays. That is a question for future research.

In any case, the fact that the relative variation is small for large arrays might justify the emphasis on mean execution times in textbooks. Sort time variation could be viewed as less important for large arrays.

Sort Time Skewness

Throughout our research, we used the difference between the mean and median as a crude measure of skewness. One criteria for choosing a trim level for the sort time distributions was based on this difference being small. A comparison of the 95/05 trimmed means in Table 5 with the medians in Table 6 shows the closeness of each mean to the corresponding median. This indicates that the skewness in the trimmed distributions is relatively minor.

Our decision for the recommended amount of trimming was guided more by its effect on the standard deviation. Trimming the top and bottom 1% would be satisfactory to remove the skewness effects due to outliers. However, standard deviations are more affected by outliers, so we chose to trim 5% from the top and bottom of samples. This often led to a ten-fold reduction in the sample standard deviation.

Unexplained Variation

In our research design, we generated separate execution time distributions for specific sorting algorithm and array size combinations. The variation within these distributions was assumed to consist of a "normal" component and outliers.

We assumed that the normal component of variation would be due primarily to the randomness of the data. Measurement of this source of variation was not very accurate because of the granularity of the Java nanoTime clock. A clock increment of 466.5 nanoseconds is almost half of a microsecond. With the speed of the processor (GHZ), much of the effect of random data on algorithm performance is hidden within these 0.466 microsecond intervals.

The most puzzling aspect of our performance measurement was the frequent appearance of outliers. Outliers can have multiple causes. In our study, the "chief suspect" is the Java runtime environment. This software performs various actions to improve the performance of a running program. The feature most relevant

seems to be Java's *automatic garbage collection* (Boyer, 2008; Wicht, 2011).

At various points during the execution of a program, the Java runtime chooses to free memory that is currently unreferenced. Generally, this is considered a good thing. However, automatic garbage collection makes it difficult to benchmark program performance.

The simple solution for running benchmark programs with Java would be to turn off Java's garbage collection feature. That is not an option. Our solution is to remove outliers from our sample. Garbage collection takes varying amounts of time. In our samples, the largest times were often 10 to 100 times larger than normal sample values.

5. SUMMARY AND CONCLUSIONS

The primary purpose of this study was to analyze variation in the performance of sorting algorithms written in Java. Most of the emphasis in algorithm textbooks is on average and worst case performance. We are more interested in the *distribution* of execution times when an algorithm is run multiple times.

We designed a methodology to control hardware, operating system, and Java runtime effects. We wanted processing time variation to result primarily from the sorting algorithm selected, the size of the array, and the randomness of the data. We wrote a Java test program to repeatedly fill an array, sort it, and record and save the execution times. The execution time data was then used to calculate statistics that summarize the distribution in terms of central tendency, dispersion, and skewness.

Our experiment was performed for four sorting algorithms: selection sort, insertion sort, bubble sort, and quicksort. For each algorithm, a range of array sizes were examined. A number of results were reported, including the following:

1. Execution time distributions were discrete, with relatively few distinct values. This was primarily due to the limited resolution of the Java nanoTime function.
2. Distributions were positively skewed and included a few very large outliers. As a result, samples had to be trimmed to remove outliers before calculating statistics.

3. For all sorting algorithms, the mean sort time increased as the array size increased. This was expected. The differing observed rates of increase were consistent with well-known order-of-growth models for the algorithms.

4. For each sorting algorithm, the standard deviation of execution times increased with array size. The algorithms differed in the amount of variation and the pattern of growth. These patterns can be explained in terms of the structure of each algorithm.

5. For each algorithm, the standard deviation grew at a slower rate than the mean. This was demonstrated by a decreasing coefficient of variation as the array size grew larger.

Three conclusions can be drawn from our results. *First*, sort time variation exists and may be an important factor in systems with real-time constraints. *Second*, sort time variation is less important for very large arrays because the amount of variation is small compared to the mean. *Third*, beware of outliers in the data, especially when using the Java runtime environment for benchmarks.

Future Research

A good research study generates more questions than it answers. That was true in this study. Our planned future research activities include:

1. Extend our analysis of variation to other sorting algorithms, such as merge sort and shell sort.
2. Use our methodology on algorithms written in other programming languages. An obvious next language is C++. One problem is that C++ provides different timer functions in different operating environments.
3. Study the behavior of Java's nanoTime function in different hardware and software environments. The statement by Oracle that nanoTime provides the "most precise available system timer" is intriguing and suggests a number of practical questions for further research.

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.
- 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.
- Lewis, John, & Loftus, William (2011). *Java Software Solutions, Foundations of Program Design* (7th ed). Addison-Wesley.
- Liang, Y. Daniel (2012). *Introduction to Java Programming* (9th ed). Prentice Hall.
- Oracle (2014). java.lang Class System. www.docs.oracle.com
- Salsburg, David (2001). *The Lady Tasting Tea*. W. H. Freeman.
- Sedgewick, Robert, & Wayne, Kevin (2011). *Algorithms* (4th ed). Addison-Wesley.
- Wicht, Baptiste (2011). Java Micro-Benchmarking: How to write correct benchmarks. www.javacodegeeks.com

