# CS21120: Data Structures and Algorithm Analysis - Sorting

[Extended Abstract]

James Euesden
Computer Science Department
Aberystwyth University
Llandinum Bldg, Penglais
Aberystwyth, Dyfed
SY23 3DB
jee22@aber.ac.uk

## ABSTRACT

This paper concerns the runtime comparissons of 7 sorting algorithms, that are tried and tested throughout computer science and software engineering. The algorithms in question are Bubble, Selection, Insertion, Shell, Radix, Merge and Quick sort. These are written and tested in Java. Each algorithm is tested under similar conditions, and their results are compared to determine if there are any interesting cross-over points in their runtimes, and if there is a 'best suited' algorithm.

## Categories and Subject Descriptors

F.2 [**ANALYSIS OF ALGORITHMS AND PROBLEM COMPLEXITY**]: Nonnumerical Algorithms and Problems—*Sorting and searching*

## General Terms

Algorithms, Performance

## Keywords

Sorting, Big O

## 1. INTRODUCTION

The task set[1] was to analyse a minimum of three sorting algorithms, written in Java, and use each on a number of test samples of data. These samples of data range from small data sets to very large data sets, with an initial 13 tests supplied and later extended to 18 tests. On each run, the elapsed time from the start of the algorithm to the completion completion time would be record and compared against other algorithms in order to reach conclusions.

From the results of the testing, we can compare and contrast each of the algorithms to determine which are the best for particular circumstances of varying data set sizes. We will also attempt to determine which are the best case average, and examine any particularly interesting results and cross-over points between the algorithms where one becomes better than another with the different sizes of data sets.

The sorting algorithms under test were Bubble Sort, Insertion Sort, Selection Sort, Shell Sort, Radix Sort, Merge Sort and Quick Sort. For this report, it is assumed that the reader has a basic knowledge of how these algorithms work, and a comprehension of their time complexity[2], expressed in Big O[3] notation.

Nowhere in this report will discuss the specifics of what an algorithm does, or how to write it in code. There are online resources in the dozens to develop these algorithms, and this report deals only with their performance in their runtimes and the effect of different sizes of data sets on each algorithm.

## 2. THE *BODY* OF THE PAPER

### 2.1 Materials & Environment

In order to get the most accurate and valid results to compare algorithms, it was necessary to have similar testing conditions for each individual test run. The environment for all tests was on a Fujitsu Lifebook AH532 Notebook, running Windows version 8.1 with a 2.6GHz processor and were allowed to use up to 3GBs of RAM for each individual test.

All algorithms and the testing suite were written in Java, SDK 1.7.0_51, using the IntelliJ IDE for both writing and conducting the tests. All tests were run under similar conditions, where the device in use was only used for running the tests and nothing else in order to best represent each algorithms performance.

### 2.2 Methods

To test the sorting algorithms, a testing suite class was written to conduct single and multiple tests on single and multiple algorithms. Using this class, the algorithms were tested multiple times on a total of 18 different data sets, each with a differing number of items in the set. By testing with many different varying sizes of data sets, it was possible to see what the effect of the size of a data set has on the different algorithms, and if some out performed others on one size, but were outperformed on other sizes, larger or smaller.

These data set sizes ranged from 10 items to 5,000,000 items, and each number contained no more than 6 digits. Some data sets had repeating data, while others did not. These are shown in a table below.

**Table 1: Test files and their data set sizes**

| Test Name | Number of items |
|---|---|
| test1 | 10 |
| test1a | 20 |
| test1b | 50 |
| test2 | 100 |
| test2a | 200 |
| test2b | 500 |
| test3 | 1,000 |
| test3a | 2,000 |
| test3b | 5,000 |
| test4 | 10,000 |
| test4a | 20,000 |
| test4b | 50,000 |
| test5 | 100,000 |
| test5a | 200,000 |
| test5b | 500,000 |
| test6 | 1,000,000 |
| test6a | 2,000,000 |
| test6b | 5,000,000 |

To get as accurate of a result as possible, each tests runtime was recorded using nanoseconds, with the Java command $System.nanotime()$. This was saved to a variable just before the call to begin the sort, and the nanotime was taken again after the completion of the algorithm. This result was then deducted from the start time to result in the elapsed time. This time was then converted into milliseconds for representation in the spreadsheet of results.

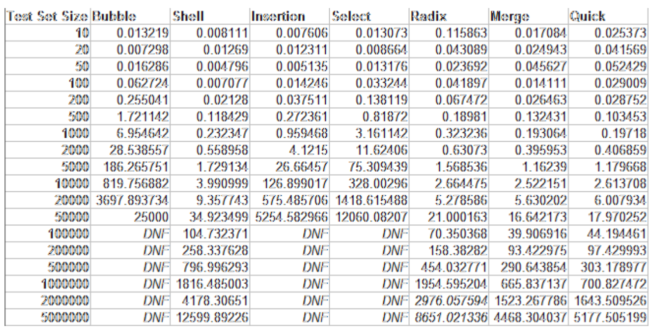
When testing each algorithm on a given data set, the runtime could sometimes be skewed by the 'warm up' of the processor as it adjusted to the algorithm, or a result might be abnormal if a sorting process was slowed down due to another process attempting to use the CPU at the same time. To combat these issues, the tests were set to run a minimum of 10 times before any elapsed times were recorded, and treated as a 'warm up' run for the algorithm,

Once an algorithm had been 'warmed' up on the system, the actual test itereations were run and the elapsed time between each run was recorded and added to a 'total' time of all runs set for testing. For the purpose of this task, each algorithm was run on each set of data a total of 100 times. The total time was then divided by the amount of times the test was run in order to form an average run time. This average run time for each test is what was considered as the result for each sorting algorithm on each test.

For a number of tests, it was necessary to cancel the sorting part way through, or to abort a test entirely. This was based on some tests taking more time than was reasonable to conclude. These will be discussed in more detail within the results section.

## 2.3 Results

As each test concluded, the results were stored in a spreadsheet to later be displayed as line graph for analysis. Below is a screencap of these results upon completion, where the numbers shown in the cells are the average runtimes of the algorithm on a particular amount of items in a data set, displayed in Milliseconds. Those cells where 'DNF' is present represent where a test 'Did Not Finish'.



| Test Set Size | Bubble | Shell | Insertion | Select | Radix | Merge | Quick |
|---|---|---|---|---|---|---|---|
| 10 | 0.013219 | 0.008111 | 0.007606 | 0.013073 | 0.115863 | 0.017084 | 0.025373 |
| 20 | 0.007298 | 0.01269 | 0.012311 | 0.008664 | 0.043089 | 0.024943 | 0.041569 |
| 50 | 0.016286 | 0.004796 | 0.005135 | 0.013176 | 0.023692 | 0.045627 | 0.052429 |
| 100 | 0.062724 | 0.007077 | 0.014246 | 0.033244 | 0.041897 | 0.014111 | 0.029009 |
| 200 | 0.255041 | 0.02128 | 0.037511 | 0.138119 | 0.067472 | 0.026463 | 0.028752 |
| 500 | 1.721142 | 0.118429 | 0.272361 | 0.81872 | 0.18981 | 0.132431 | 0.103453 |
| 1000 | 6.954642 | 0.232347 | 0.959468 | 3.161142 | 0.323236 | 0.193064 | 0.19718 |
| 2000 | 28.538557 | 0.558958 | 4.1215 | 11.62406 | 0.63073 | 0.395953 | 0.406859 |
| 5000 | 186.265751 | 1.729134 | 26.66457 | 75.309439 | 1.568536 | 1.16239 | 1.179668 |
| 10000 | 819.756882 | 3.990999 | 126.899017 | 328.00296 | 2.664475 | 2.522151 | 2.613708 |
| 20000 | 3697.893734 | 9.357743 | 575.485706 | 1418.615488 | 5.278586 | 5.630202 | 6.007934 |
| 50000 | 25000 | 34.923499 | 5254.582966 | 12060.08207 | 21.000163 | 16.642173 | 17.970252 |
| 100000 | DNF | 104.732371 | DNF | DNF | 70.350368 | 39.906916 | 44.194461 |
| 200000 | DNF | 258.337628 | DNF | DNF | 158.38282 | 93.422975 | 97.429993 |
| 500000 | DNF | 796.996293 | DNF | DNF | 454.032771 | 290.643854 | 303.178977 |
| 1000000 | DNF | 1816.485003 | DNF | DNF | 1954.595204 | 665.837137 | 700.827472 |
| 2000000 | DNF | 4178.30651 | DNF | DNF | 2976.057594 | 1523.267786 | 1643.509526 |
| 5000000 | DNF | 12599.89226 | DNF | DNF | 8651.021336 | 4468.304037 | 5177.505199 |

**Figure 1: The full tests results table**

As is visible from the results, the simple sorts, Bubble, Insertion and Select, did not finish or were aborted for the larger test sizes. This was due to the amount of time it began to take for each sorting algorithm to run the tests. The time it would take became unreasonable and so it seemed best to abort those tests. Even with not running these tests, it is apparent from the resulting graph the way in which these $O(n^2)$ time complextity[4] algorithms would continue to run and how their results look.

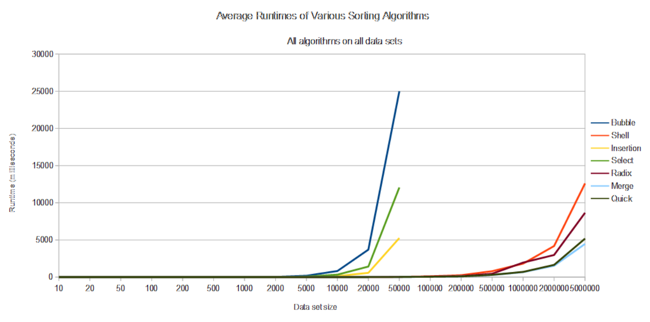


**Figure 2: The test results expressed as a line graph**

This line graph represents the full results contained in the spreadsheet, and stands as a way of comparing each sorting algorithm against the others. What immediately stands out here, is how the simpler algorithms (Bubble, Select, Insertion) begin to take much longer than those of $O(nlog(n))$ and $O(nk)$ at around the 10,000 items in a data set mark due to their exponential growth rate.

## 2.4 Discussion

To break down the results and intelligently discuss them, the results have been dissected into more appropriate forms of viewing. To begin, the smaller data (10 - 200) sets were looked at and analysed to see which sort(s) is the most appropriate for smaller data sets.

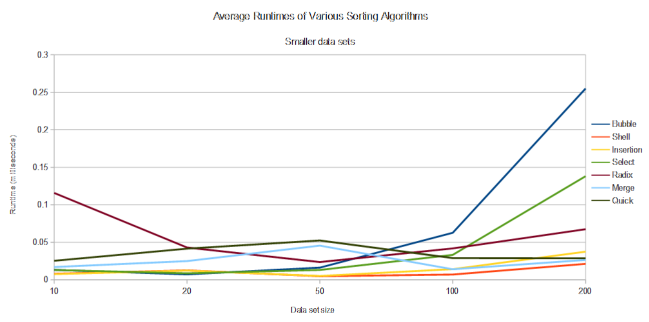


**Figure 3: Smaller data sets and their results**

What is most apparent, is that those $O(n^2)$ sorting algorithms that struggle with the larger sets actually out perform the more complicated algorithms for the data sets below 100-200 items.

Notably, Insertion and Shell sort share very similar time complextities for these smaller data sets. As can be seen at the 200 items point though, Shell sort continues to perform with a lower runtime, while Insertion sort's time begins to break away and the algorithm takes longer to iterate through the list. This could be expected, as Shell sort is based upon Insertion sort, and could be considered as an improvement on Insertion.

At the 200 data set test, it can also be seen that Quick Sort crosses over with the simpler sorts, beginning to perform at a similar rate as them. With any data sets less than this size, Quick Sort does not work efficiently compared to the other algorithms. It is in fact one of the slowest sorting algorithms for smaller data sets, with Radix sort being the absolute worst algorithm until a size of 50 or more items.

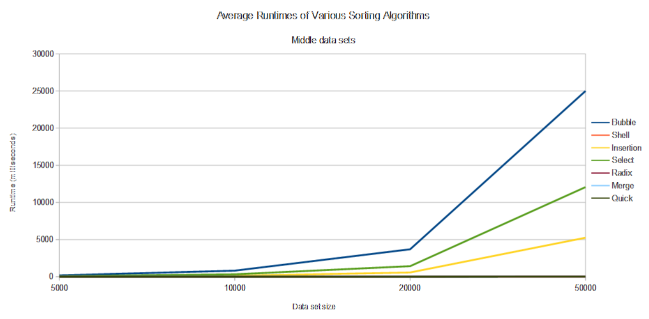


**Figure 4: The data sets from the middle of this reports testing data**

Another interesting thing to note is how Bubble sort's expoential growth is very visible by the point there are 100 items of data to process. It is at this stage that the more reliable algorithms cross over with Bubble sort, and shows why Bubble sort is fine for small data sets, but terrible when presented with anything moderately large. The same can be said about Select sort, which also becomes exponentially slower than Bubble sort at 100 items, just at a lowler rate. This is backed up by when you look at the middle set of data sets tested for this report.

While the other algorithms continue working at a relatively similar pace, Bubble, Select and Insertion sort grow exponentially. This leads them up to the tests they did not process, when they were no longer able to run within a reasonable time frame for the testing. Regardless, it is relatively easy to predict their expected outcomes for future tests based upon the results of the tests that were conducted on these algorithms.

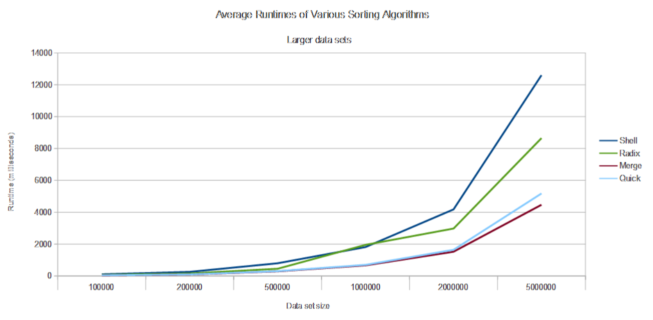


**Figure 5: The larger data sets and test results for applicable algorithms**

At the other end of the tests, where the data set sizes reach ranges of 100,000 to 5,000,000, each of the remaining sorting algorithms begins to diverge from each other in their time taken to sort the items. Merge sort and Quick sort stay very similar to one another with their results, up until 1,000,000 items. At this point, Quick sort shows itself to be slower at sorting the data set than Merge sort, although not by much. If more tests on even greater data sets were to be made, it is probable that we would see Quick sort progressively slower in it's run time than Merge sort.

As for Radix and Shell sort, while both have up to that point performed similar to Merge and Quick sort, they both become slower than the others at the 200,000 items in a data set range. Shell sorts 'curve' on the graph becomes a lot steeper, a lot faster than Radix sorts does, and we see the point at which it gets incrementally more difficult for Shell sort to run on larger data sets in comparison to the other algorithms.

A note to make about these tests, with particular attention to Radix sort, is that their runtimes and performances will be directly affected by the device they are run upon. If the device begins to have difficulties in processing such large data sets, it is very possible that the runtime would be significantly slower than they would be on a more powerful machine, even if the previous results were practically identical with smaller data sets on a less powerful machine.

It is for this reason that it may be the curves become so much steeper so quickly with data sets at this size. However, regardless of the processing power of the machine and the impact it has upon the runtimes, it can still be taken as a valid result, since this report is comparing and contrasting the runtimes of these algorithms against each other. Therefore, these results can be considered a real life example of a scenario where they are used on a device requiring them, and which algorithm is still the 'best', 'worst' and 'average' case, for ranges of data sets, small to large.

This particularily applies to Radix sort, as at the point where it is processing sets of items in the million and higher ranges, the space complexity of the algorithm makes it difficult for some devices to cope with, resulting in a slower algorithm as the devices RAM is bloated and under strain. This report still considers this to give an accurate result, based on the 'real world' example of how Radix sort compares to the other sorts with the same working environment.

From analysing and discussing the resulting data from our tests, we can see that the 'best sort' for a certain situation can be entirely situational. For example, it would be foolish to employ a Bubble sort or Selection sort on a data set larger than 1,000. It is at and after these points that these algorithms become the worst to use for their time complexity.

However, if presented with a much smaller size of items, such as one under 200, it would be acceptable to use Bubble or Selection sort over Quick and Merge sort , although most likely preferable to implement Shell or Insertion sort, as they handle these smaller data sets better than any other algorithm tested, as seen in the results.

At around the 50,000 items mark, there is a cross over point where it can safely be said that the size of the data set truly does make a difference. The simpler algorithms (Selection, Insertion, Bubble) runtime grow at such an exponential rate that it becomes pointless to use them. While previously, Quick, Merge and Radix sort were the worst case use for smaller data sets are now the much more time efficient options.

From this we could conclude that the runtimes of each algorithm are directly connected to different sizes of data sets, and some algorithms handle different sizes of data sets much better (or worse) than others. This is not to say that one should never use a Quick sort on a smaller data set, however. In fact, it is potentially the best general purpose sort.

When analysing the results of the smaller data sets below 1000, the difference in the runtime between all of the algorithms is rather minimal, and Quick sort does not perform terribly. Assuming that the algorithms will not be run on a large quantity of smaller data sets, there would most likely not be too much of a noticable difference in performance of an application.

In the chance that this is the case though, multiple algorithms could be employed through the use of conditional statements. For example, when the data set is passed to the application, an if statement could check and see the size of the data set. If it is greater than 100, then Quick sort may be applied. Otherwise, Insertion sort could be. This has often been the case with Quick sort in other applications, using the best algorithm for the task at hand dynamically. For testing purpose, however, I felt it best to keep the two tested individually.

The question may be raised, "Why not say the best average is Merge Sort? Which did just as well, if not better, than Quick sort?". The answer to this is not in the time complexity, but in the space complexity. Merge sort uses more memory than Quick sort does.

While Quick sort sorts items in-place, using only the original list of items provided and indexes to where to sort the items, Merge sort requires another array in order to separate and 'merge' the data to be processed.

Unless the device the sorting algorithm is running on has enough memory to handle this, Quick sort is the better choice for the best average case sorting algorithm. Plus, as previously stated, it's drawback with smaller data sets can be overcome with a conditional statement that processes smaller data sets with Insertion sort.

## 3. CONCLUSIONS

The overall conclusion to take from these results are that the size of a data set directly corresponds to the runtime performance of an algorithm, and that no one algorithm is best suited for every single data set, but there is an average best case in QuickSort, should an 'all-round' algorithm be required with little specification.

A developer should be aware of the differences in run time between the different types of algorithms and therefore selective of the algorithm they choose to implement when using sorting in an application. They should select based upon their expected data set sizes, their space requirements and be prepared to have conditional statements to choose the best algorithm for a problem presented to the application.

## 4. REFERENCES

[1] R. C. Shipman, *"CS21120: Data Structures and Algorithm Analysis Assignment 2 - Sorting*, 25th February 2014.

[2] Big-O Algorithm Complexity Cheat Sheet [webpage]. Available: http://bigocheatsheet.com/ Accessed: 12/03/14

[3] Big O notation - Wikipedia, the free encyclopedia [webpage]. Available: http://en.wikipedia.org/wiki/Big_O_notation Accessed: 12/03/14

[4] Time complexity - Wikipedia, the free encyclopedia [webpage]. Available: http://en.wikipedia.org/wiki/Time_complexity Accessed: 12/03/14

## APPENDIX

### .1 Introduction

### .2 The Body of the Paper

#### *.2.1 Environment & Tools*

#### *.2.2 Methods*

#### *.2.3 Results*

#### *.2.4 Discussion*

### .3 Conclusions

### .4 References

## A. EXECUTIVE SUMMARY

### A.1 Mission Statement

To test the runtimes of different sorting algorithms on data sets of small and large sizes to see the performance of each algorithm and analyse the results using line graphs and discussion.

## A.2 Methods

Testing the algorithms using averaged time recorded in nanoseconds on a singular device under similar circumstances for multiple tests and iterations. Recording the averaged run time results in a spreadsheet for analysis.

## A.3 Outcomes

Concluding that the runtime of a sorting algorithm is affected by the size of a data set and that some algorithms are best suited for larger amounts of sets but bad for smaller sizes, and visa versa. Determining which algorithm could be best suited as a 'best average case' sorting algorithm when presented with any size of data set.

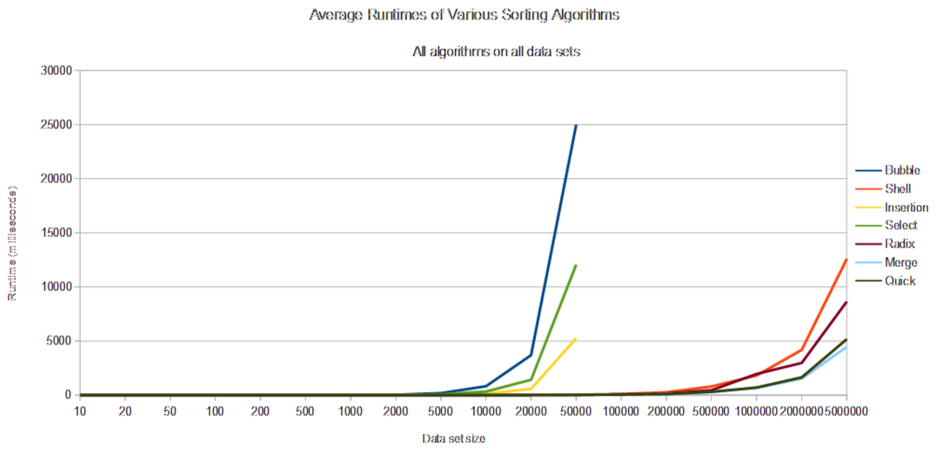


**Figure 6: The test results expressed as a line graph**

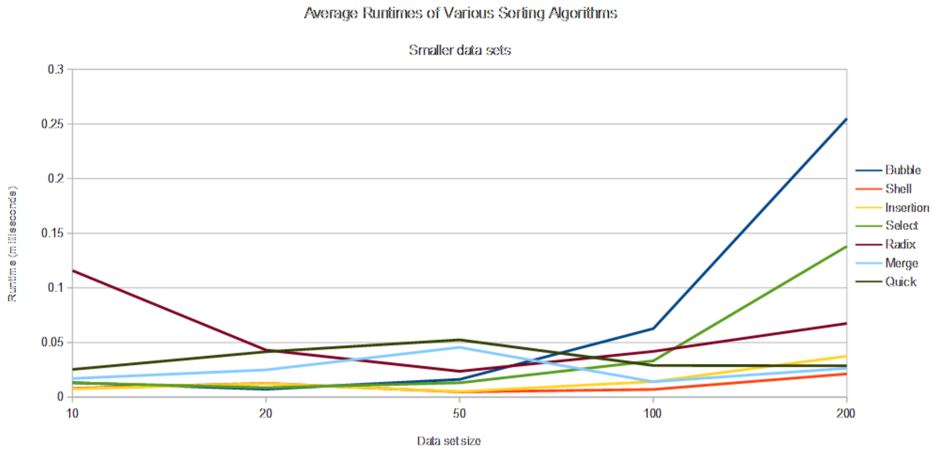


**Figure 7: Smaller data sets and their results**

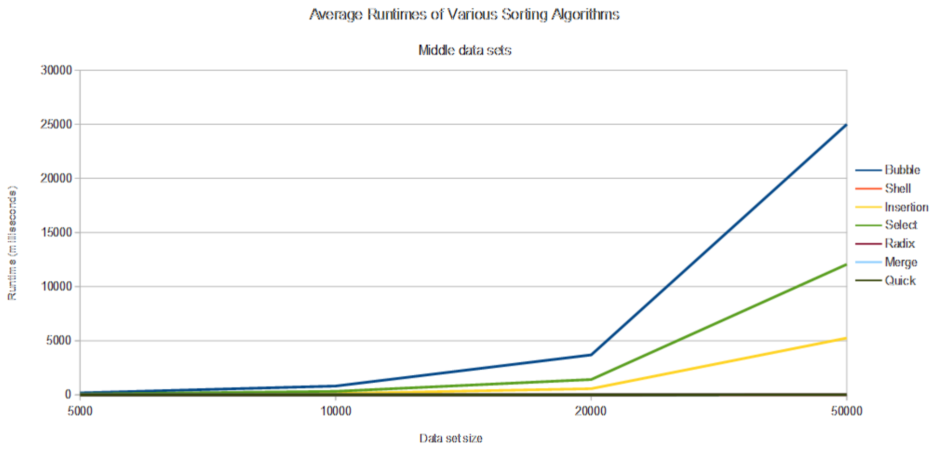


**Figure 8: The data sets from the middle of this reports testing data**

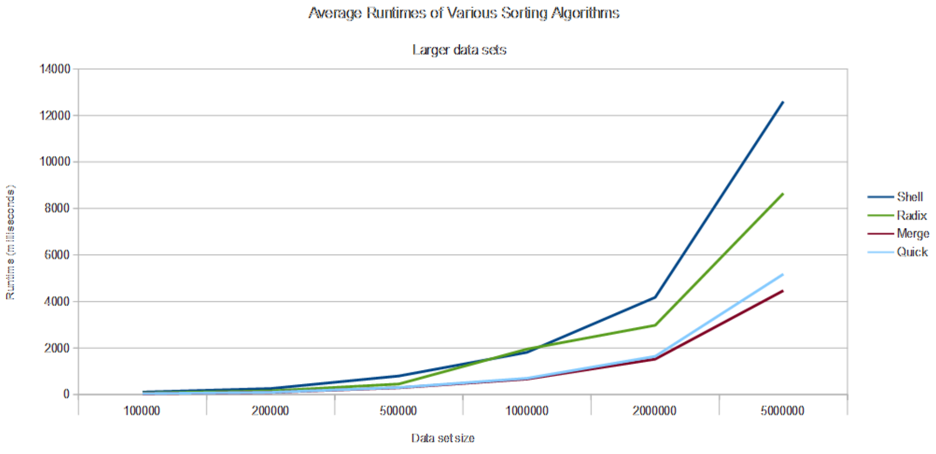


**Figure 9: The larger data sets and test results for applicable algorithms**