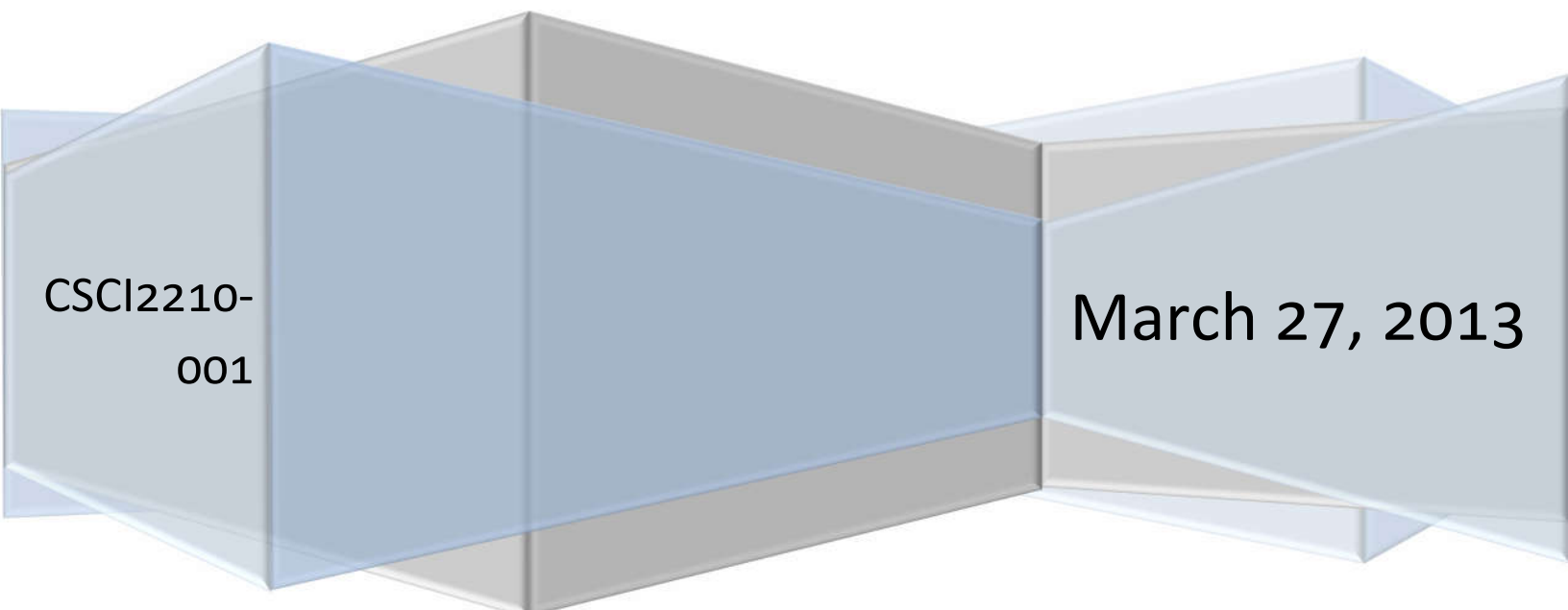


Performance and Growth Characteristics of C# Sorting Algorithms

An Analysis Using a Code Execution Profiler

Author: Isaac Styles



CSCI2210-
001

March 27, 2013

Contents

INTRODUCTION.....	2
PROBLEM STATEMENT	2
SOLUTION STATEMENT	2
OBSERVATIONS AND CONCLUSIONS ON THE DIFFERENT QUICK SORTS	2
DETERMINING THE FASTEST ALGORITHMS.....	3
ON THE INACCURACIES OF THE MODEL	4
SUMMARY	4
APPENDIX 1 – DATA AND GRAPHS.....	5
REFERENCES	10

INTRODUCTION

There are numerous ways in C# which an algorithm can sort a list of integers. Each sort algorithm takes a different approach, and will invariably have a unique profile of strengths and weaknesses when dealing with certain qualities and quantities of data. For this study I recorded the elapsed inclusive time needed to sort a List of integers using six individual algorithms: Sinking Sort, Selection Sort, Insertion Sort, Shell Sort, Quick Sort, and a modified Quick Median-of-Three Sort.

PROBLEM STATEMENT

As there is no ideal way of sorting a random dataset using only one sort algorithm, there will be a loss of algorithm performance anytime a sort is performed on a dataset it is sub-optimally designed for. An optimal sort algorithm exists for every conceivable dataset, but there is no way to correct for these losses without having prior knowledge of the characteristics of the dataset. In the study I manipulated the dataset characteristics across the various algorithms which allowed me to discover the performance of each sort algorithm without regard to the random trends within datasets. Each sort algorithm should reveal predictable changes in the elapsed inclusive time when faced with varying dataset characteristics and number of iterations.

SOLUTION STATEMENT

I created a C# algorithm that builds an array of identical `List<int>` objects containing a quantity N of random, semi-random, or sorted integers. It then passes a List to each of the sort algorithms one at a time, which are encapsulated in functions to make profiling easier.

I then recorded the performance profiles of each sort function at three distinct iterations and with various types of datasets. Specifically, the program was run at $N=10$, $N=100$, $N=1000$ iterations using Lists of 100% random integers, 90% random integers, 10% random integers, ascending integers, descending integers, and constant integers. The elapsed inclusive time of each function was collected using the Code Execution Profiler built into Visual Studio 2010.

Using the data collected from the various performance profiles I plotted the elapsed time required by each algorithm on a logarithmic scale against the number of iterations. This provided a good way to compare each algorithm's growth and Big-O across the three iterations.¹ To verify my findings I compared my observed Big-O with those found online at cprogramming.com (Allain).

OBSERVATIONS AND CONCLUSIONS ON THE DIFFERENT QUICK SORTS

Quick Sort uses a divide and conquer strategy to achieve $O(N \log(N))$ with random data, but falls short on presorted datasets achieving performance most like $O(N^2)$. It is important that the pivot not be the largest integer in the List, which is the cause of the slow-downs seen in the original Quick Sort when using an ascending dataset. The modified Quick Median-of-Three Sort finds a way around this slowdown by declaring the pivot as the median of three positions within the List, guaranteeing the existence of an integer to the right of the pivot that is greater than or equal to the pivot. The additional partition and diversion to Insertion Sort when $N \leq 10$ to avoid the increased overhead allows the Quick Median-of-Three Sort to maintain $O(N \log(N))$, even with presorted datasets.²

The Quick Median-of-Three Sort is not the only algorithm that could benefit from defining a cutoff to Insertion Sort when $N=10$. Insertion Sort performs better than Sink Sort, Selection Sort, and Quick Sort when $N=10$. Although 10 is a useful cutoff value and was used in this study, that value can be manipulated to optimize the crossover point between

¹ Refer to the Excel Datasheet titled *Growth Functions*

² Refer to chart (1) titled *Growth of Quick Sort Algorithms in Randomized vs. Ascending Datasets*

algorithms. A larger cutoff would lower the amount of overhead created when switching to lower iterations of the Quick Median-of-Three algorithm, though one would anticipate a nearby point of diminishing returns. More research is needed, but it is clear by the chart *Comparison of Algorithm Elapsed Time, N=10* that Insertion Sort is the fastest algorithm when $N=10$.³ Alternately, Shell Sort may be used because of its similar performance at $N=10$ but significantly less Big-O.

The Quick Median-of-Three algorithm is thought to be an improvement upon the original, so is this verifiable? The differences in performance between the two are menial, with the largest gain coming from switching to the Insertion Sort in lower iterations. However, a major hiccup in the Original Quick Sort caused by sorting an already In-Order dataset is cured by the Quick Median-of-Three Sort.⁴ These two characteristics lend the Quick Median-of-Three Sort an advantage, but in the end the Quick Sort still achieves a lower total elapsed time by a difference of 266.38 milliseconds. Additionally, the Original Quick Sort may be further improved by adding a cutoff to switch to an Insertion Sort when $N \leq 10$.⁵

DETERMINING THE FASTEST ALGORITHMS

To determine the overall fastest sort algorithm I compared each algorithm's sum of elapsed times for each iteration value.⁶ It is no surprise that Shell Sort, with $O(N \log(N))$, was able to outperform every other algorithm by at least a factor of 100x when talking about their averages. This is because Shell Sort scales very well with higher numbers of iterations and is more resilient to trends in the dataset. Although its performance at $N=10$ was only on par with the others, Shell Sort seemed to return the most consistent times across different datasets; whether the dataset was random, sorted, or constant the Shell Sort turned out consistently low times. These two factors make it safe to say that Shell Sort is the overall best algorithm for sorting a List.

Though Shell Sort is consistently the fastest sort algorithm on most datasets, there are cases where Sink Sort is able to outperform it. In datasets that did not need any additional sorting, Sink Sort was able to operate at $O(N)$ instead of $O(N^2)$, which is effectively faster than any other comparative search algorithm.⁷ Datasets such as sorted ascending and constant provide the conditions necessary for Sink Sort to have a performance advantage at any number of iterations. In any other type of dataset the Sink Sort quickly becomes inefficient as N increases because it is $O(N^2)$.

The Sink Sort's stellar performance with presorted lists becomes quickly outweighed by its $O(N^2)$ in every other type of dataset. An increase in randomness of only 10% over a sorted, ascending dataset causes the Sink Sort to be outperformed by both Quick Sort and Shell Sort.⁸ This leads me to believe there is no rational reason to choose it over the Shell Sort or Quick Sort in real-world scenarios.

A small advantage may exist when using Sink Sort in environments where sorting small, presorted datasets is common, but such an advantage could be conferred to other algorithms by adding a clause to stop the sort if the dataset is already determined to be in order. At worst such a clause would compare each value once prior to the actual sort, thus adding

³ Refer to chart (2) titled *Comparison of Algorithm Elapsed Time, N=10*

⁴ Refer to chart (3) titled *Comparison of Algorithm Elapsed Time, N=1000*

⁵ Refer to chart (7) titled *Growth of Quick Sort Algorithms With and Without Cutoff to Insertion Sort at N=10*

⁶ Refer to table (1) titled *Average Elapsed Inclusive Time of All Datasets*

⁷ Refer to chart (4) titled *Growth of Sorting Algorithms in a Pre-Sorted, Ascending Dataset*

⁸ Refer to chart (5) titled *Growth of Sorting Algorithms in a 10% Random Dataset*

$O(N)$ work to the algorithm. This avoids Sink Sort's potential to perform at $O(N^2)$ while retaining the benefits when sorting a presorted, ascending list.

ON THE INACCURACIES OF THE MODEL

There is reason to believe that inaccuracies within the elapsed time could have arisen by creating a new dataset with each change in dataset type. Each iteration of the current application sorts a wholly different subset of integers, allowing an additional amount of variability in the performance profiles of the different datasets. Without averaging the elapsed time of numerous trials it becomes possible to measure the specific variances of each random dataset, and less the actual performance of the algorithm. I suspect the superlative performance of the Original Quick Sort to be influenced by this. At $N=1000$ and in a random dataset the algorithm should behave more like the Quick Median-of-Three Sort. This is reflected in the 100% Random trial, but the 90% Random trial shows performance equal with the much faster Shell Sort.⁹

Another method to defeat the impact of a random list with each iteration is to alter the application to store a single, static dataset and use it to derive each of the six types of datasets which to sort using the six types of algorithms. The elimination of this variable would allow for direct comparisons of the performance between different types of datasets.

SUMMARY

Some of the sort algorithms performed very similarly. The Sink Sort, Selection Sort and Insertion Sort all performed similarly on random and semi-random datasets, with the Selection Sort gaining an edge with datasets above $N=10$. The Selection Sort also performed more consistently with the presorted ascending and descending datasets.¹⁰ When choosing between these sort algorithms with $O(N^2)$ I recommend against using the Sink and Insertion Sorts in favor of a modified Selection Sort with a cutoff point to Insertion Sort at $N=10$.

The Sink Sort performed the worst with random and semi-random datasets, but performed at $O(N)$ when sorting a presorted ascending or constant dataset. The algorithm may have some benefits in an environment where sorting a presorted list is common, but the benefits diminish quickly with increased randomness as even a 10% random dataset completely negated the effects.

The Original Quick Sort squeezed out a victory over the Quick Median-of-Three Sort. However, each has its own advantage: the Original Quick Sort performed better with random and semi-random datasets, while the Quick Median-of-Three Sort eliminates major slowdowns when dealing with presorted and constant datasets. A further advantage may be given to the Original Quick Sort by adding a cutoff point to Insertion Sort at $N=10$.

The Shell Sort performed the best out of all the algorithms by achieving a true $O(N \log(N))$ on all datasets and iterations. Both Quick Sorts are also capable of achieving $O(N \log(N))$, but the Shell Sort is more resilient to dataset characteristics which translates to the best average performance.

⁹ Refer to chart (6) titled *Growth of Quick Sort Algorithms in 100% Random and 90% Random Dataset*

¹⁰ Refer to chart (8) titled *Comparison of $O(N^2)$ Algorithms*

APPENDIX 1 – DATA AND GRAPHS

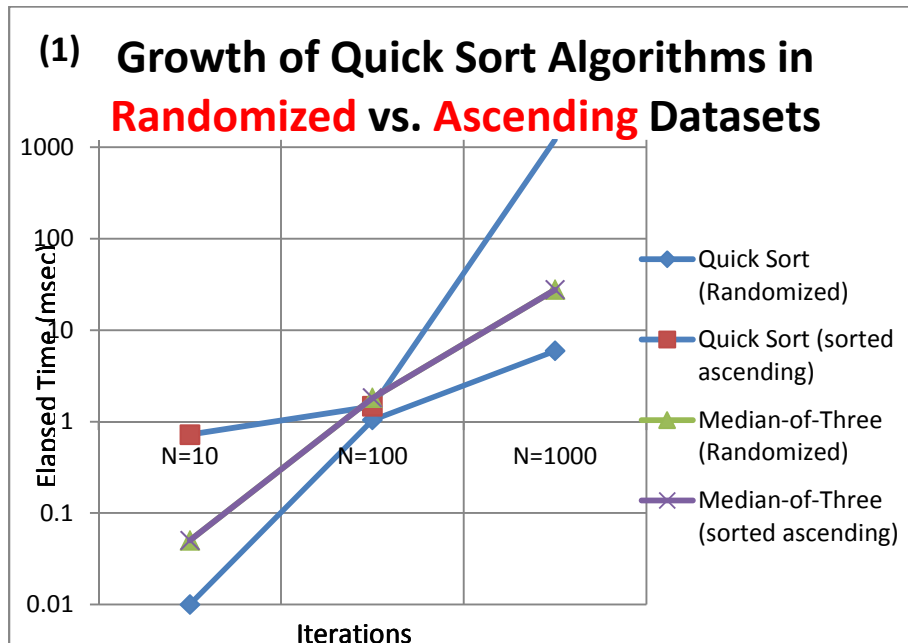
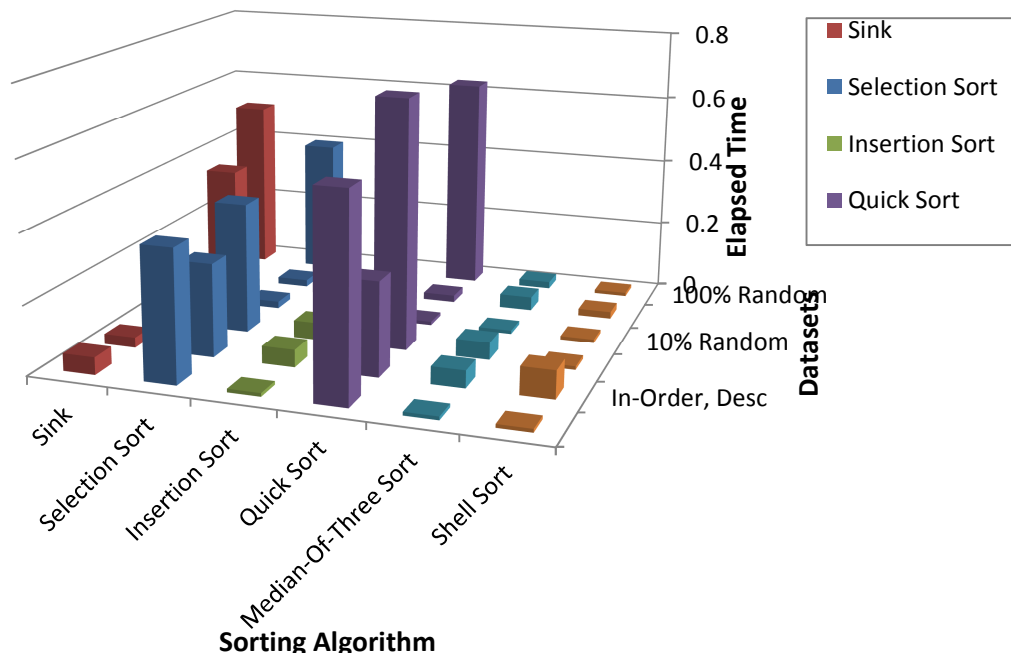


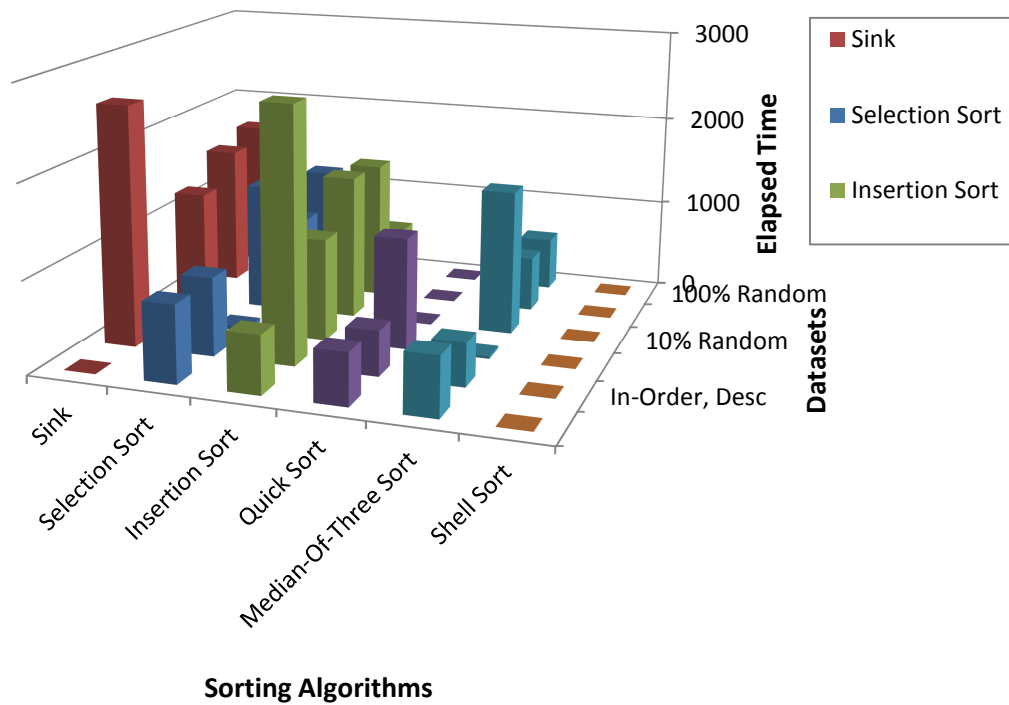
Table (1)

	Average Elapsed Inclusive Time of All Datasets (msec)			SUM:
	N=10	N=100	N=1000	
Shell	.02	.32	3.33	3.67
Selection	.25	1.38	849.69	851.31
Insertion	.03	1.51	1350.47	1352.01
Quick	.37	1.33	377.53	379.24
Quick Median-of-Three	.03	1.65	643.94	645.62
Sink	.16	1.64	1172.89	1174.69

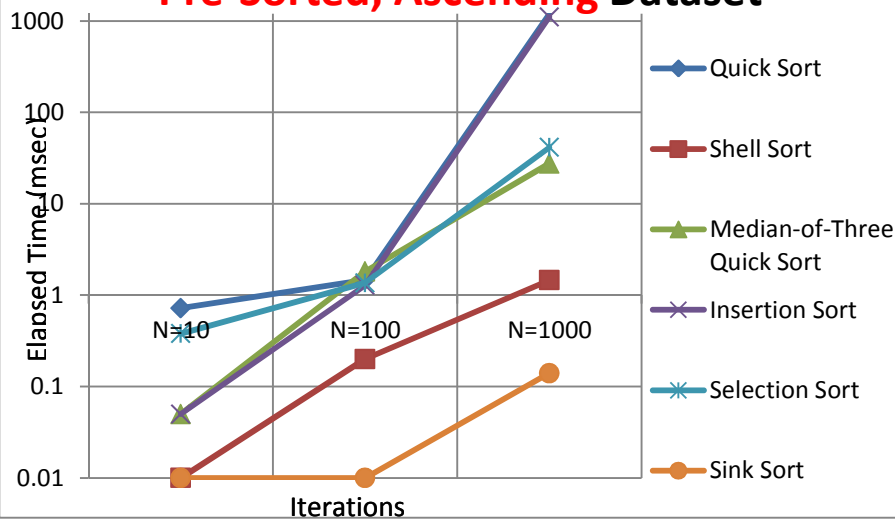
(2) Comparison of Algorithm Elapsed Time, N=10



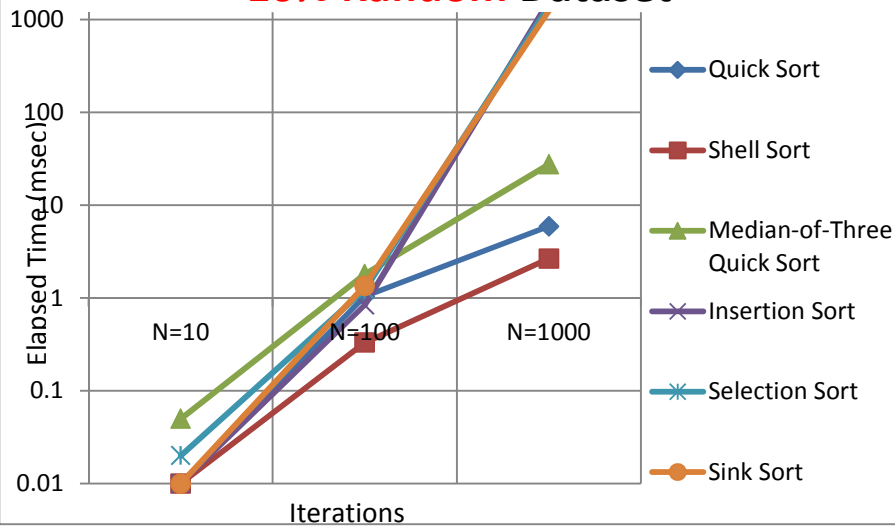
(3) Comparison of Algorithm Elapsed Time, N=1000



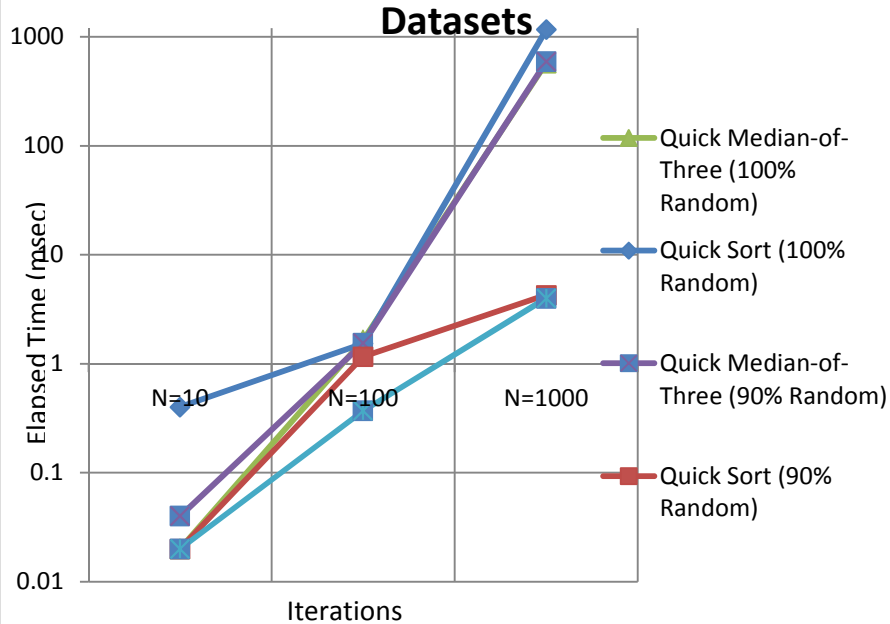
(4) Growth of Sorting Algorithms in a Pre-Sorted, Ascending Dataset



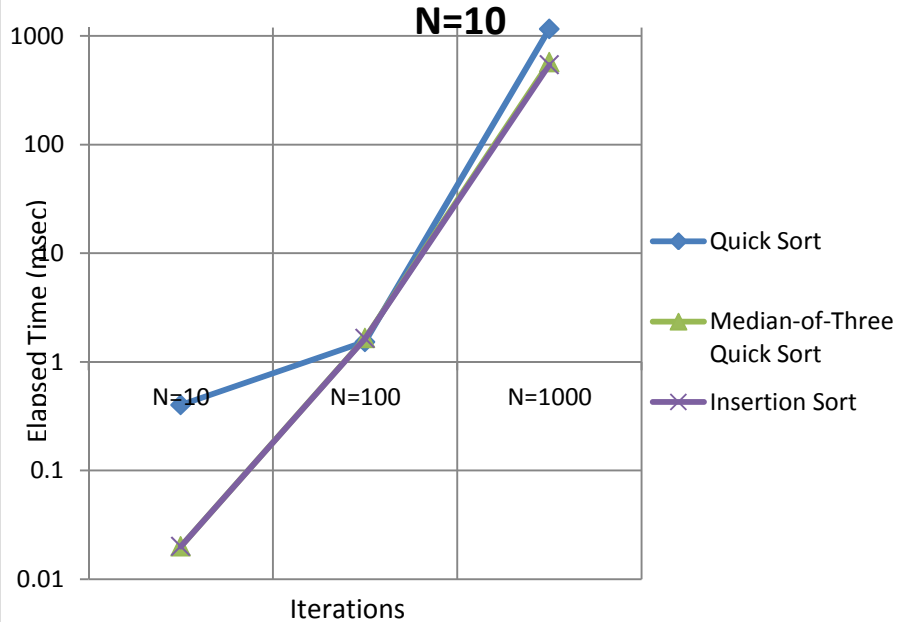
(5) Growth of Sorting Algorithms in a 10% Random Dataset



(6) Growth of Quick Sort Algorithms in 100% Random and 90% Random Datasets

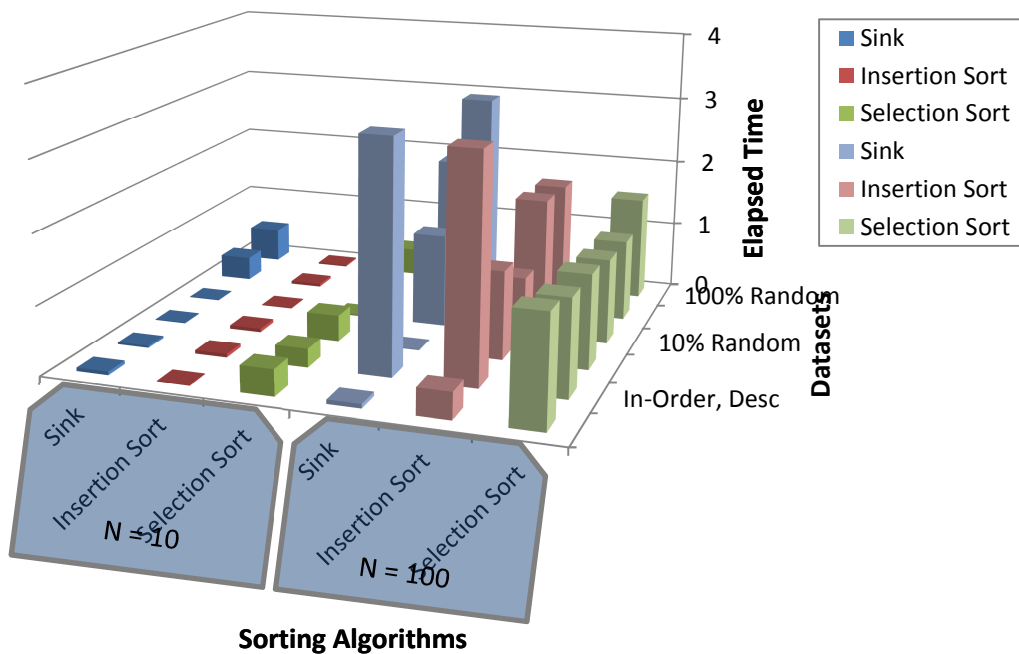


(7) Growth of Quick Sort Algorithms With and Without Cutoff to Insertion Sort at



(8)

Comparison of $O(N^2)$ Algorithms



REFERENCES

Allain, Alex. "Sorting Algorithm Comparison." Sorting Algorithms Compared.

Cprogramming.com, 2011. Web. 27 Mar. 2013.

<<http://www.cprogramming.com/tutorial/computersciencetheory/sortcomp.html>>.