Performance of Different Algorithms Regarding Their Use on Various Arrangements and Sizes of Data

Matthew Hill

Primary Author

5621 Fox Ct.  
South Beloit IL  
1-608-774-538

Matthew.richard.hill@gmail.com

**ABSTRACT**

In this paper we will compare different search algorithms and analyze their time complexity by comparing their completion time for sorting different combinations and sizes of numerical arrays. This paper aims to explain each algorithm andanalyze the theoretical time complexity compared with the actual results from sorting a multitude of files.

# INTRODUCTION

The algorithms that we will be using are quick-sort, insertion-sort, and shell-sort. There are two different versions of quick-sort and insertion-sort, and one version of shell-sort.

# ALGORITHMS IN THEORY

*2.1 Insertion Sort*

Insertion sort works on the concept of building a final array one item at a time. In theory, this algorithm is the most beneficial on smaller arrays with the purpose of outclassing algorithms like bubble or selection sort. Because this algorithm works by building a final array piece by piece, the best-case scenario would be one where the pieces are already in order and the algorithm would simply check each piece, leading to a time complexity of O(n). However, if the pieces were in the exact reverse order, the algorithm would have to compare each piece to every other piece leading to a time complexity of O(n2) in the worst-case scenario. One of the key reasons insertion sort is used is because it doesn’t take any additional memory because it swaps values rather than creating a separate array.

*2.2 Quick Sort*

Quick sort works on the concept of partitioning the data into lesser and greater sub-arrays continuously until each array only contains one value. It does this by picking a Pivot(an element in the array of chosen by median, but can be altered depending on the type of data set) and then sorting elements based on whether their value is greater or less than the pivot. This algorithm is often used as a replacement for either merge-sort or heap-sort because it can be far faster. The best-case scenario using quick-sort is when the array can be divided into two equal pieces every time we pick a pivot. This minimizes the number of sub-arrays we need to create. This will result in a time of O(n log n) to complete. This is the same as the average expected time because we can usually rely on the program picking a good pivot value. A worst-case scenario would be if the arrays were always split into the most unequal sizes (1 element vs. the rest of the array). This would result in a very long call tree and a time complexity of O(n2).

*2.2 Shell Sort*

Shell sort is an in-place comparison sort that works in concept by exchanging elements that are far apart and repeating the process until it can exchange elements that are close together. The idea is that this will remove a large amount of disorder quickly and make the rest of the algorithm quicker as it progresses. There is a lot of different ways to determine how large the gap and many different authors have published their own equations on how to calculate a gap under which circumstances. The worst complexity for shell sort is O(n2) which would have achieved under a condition where the gap sequence where the integers have the greatest a greatest common denominator of one. This would result in impractical gap sequences and slow the process greatly and is only achieved when using the worst known gap sequence created by Donald Shell himself when conceptualizing the sort ( 2 (N/(2k+1)) + 1) . Using the best known gap sequence the worst case scenario drops to O(n log2n) when using the correct pattern of numbers. The best performance when using a correct pattern is O(n log n).

*2.3 Merge Sort*

Merge sort works by first dividing a list into the smallest possible pieces on chunk at a time. Only once each piece is one value in size does the sorting begin. Each of these sub-lists is merged with other adjacent sub-lists continuously by sorting them until the final array is completed. Merge sort is particularly reliable because the best case, worst case and average case scenario are all O(n log n). This makes it very good for general purpose. The downfall is that it requires external memory to operate.

# DESCRIPTION OF THE EXPERIMENT

*3.1 Given Materials*

For this experiment, twelve pre-written files containing data were tested. These consisted positive numbers with greatly ranging values. They were organized into categories where the numbers where random, sorted numerically, and sorted reverse numerically. Each category continued files with 1000, 10000, 100000, and 1000000 elements. A pre-compiled program was also given for an insertion sort algorithm and a quick sort algorithm. An additional insertions sort, quick sort, and shell sort algorithm where provided from previous student work.

*3.2 Execution*

After testing all code to make sure that it operated as intended, each given file was run through each program and the timing results were recorded in the tables below. All programs were run on Illinois State University’s OAK linux terminal in the same 24 hour span, with minimal gaps between run times. The “Time” command was used to computer program run-time and the “real” time value was used for recording purposes after being converted into seconds format.

# RESULTS

*3.1 Written Presentation*

The resulting data for the random order showed similar time complexities at lower values with merge sort and quick sort outclassing the others. With larger values, the merge sort algorithm ended with a segmentation fault. The resulting data for the sorted order showed insertion sort and quick sort slower than the others and quick sort segmentation fault at one million elements. When the elements were sorted in reverse order, quick sort and merge sort performed better than the others, until they reached one million elements where they both segmentation faulted. Between shell sort and insertion sort, insertion sort performed better. Between all trials, the insertion sort provided performed better than the student written version, this was especially clear when the data was sorted.

*3.2 Graphical Presentation*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Random Ordering Element Count** | 1000 | 10000 | 100000 | 1000000 |
| Insertion Sort Provided  Time (s) | 0.005 | 0.089 | 7.709 | 754.231 |
| Quick Sort Provided  Time (s) | 0.008 | 0.015 | 0.119 | 1.135 |
| Insertion Sort Student  Time (s) | 0.007 | 0.216 | 20.357 | 2174.249 |
| Merge Sort  Time (s) | 0.005 | 0.021 | 0.158 | Seg fault |
| Shell Sort  Time (s) | 0.014 | 0.86 | 82.44 |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Reverse Ordering Element Count** | 1000 | 10000 | 100000 | 1000000 |
| Insertion Sort Provided  Time (s) | 0.011 | 0.163 | 15.159 | 1512.582 |
| Quick Sort Provided  Time (s) | 0.005 | 0.154 | 1.096 | Seg fault |
| Insertion Sort Student  Time (s) | 0.007 | 0.251 | 23.657 | 2478.901 |
| Merge Sort  Time (s) | 0.006 | 0.017 | 0.121 | Seg fault |
| Shell Sort | 0.022 | 1.692 | 164.973 |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sorted Ordering Element Count** | 1000 | 10000 | 100000 | 1000000 |
| Insertion Sort Provided Time (s) | 0.007 | 0.016 | 0.1 | 0.983 |
| Quick Sort Provided  Time (s) | 0.009 | 0.143 | 1.096 | Seg fault |
| Insertion Sort Student  Time (s) | 0.008 | 0.185 | 17.018 | 2067.734 |
| Merge Sort  Time (s) | 0.005 | 0.017 | 0.123 | Seg fault |
| Shell Sort  Time (s) | 0.007 | 0.019 | 0.163 | 1.719 |

# DATA DISCUSSION

From the results of this experiments, it is clear that these algorithms have separate and unique uses, each with their own advantages and downfall. Quick sort and merge sort were more effective than the others but had a tendency to fail at larger values. There is reason to speculate that the program provided for quick sort was not meant to handle larger loads. Merge sort probably failed at larger values due to the space required to continue sorting. This space was likely not available and caused the program to produce a segmentation fault as a result. As should be expected, insertion sort performed poorly at larger values. In a sorted array insertion sort is expected to reach its best case and outperform other algorithms and it did. However, a flaw in the technique of the student provided algorithm caused it to perform nearly as portly on a sorted array as any other. Also, as expected, merge sort performed consistently at all arrangements of the data. Merge sort provided a reasonably quick rate for all arrangements and all sizes it could perform correctly before running out of memory. Shell sort performed well in a sorted array because, like insertion sort, this represents the best case where to swaps need to be made. However, in the random order and reverse order shell sort performed significantly worse than any other. This is because shell sort needs to run through the array multiple times and when data is large it leads to a lot of actions. Quick sort performed as expected, keeping up with merge sort closely on most sizes. The benefit of quick sort compared to merge sort, as discussed earlier, is that it does not require extra space to function. This is likely why it did not experience a segmentation fault at the one million value mark for a randomly distributed array of values.

# CONCLUSION

Overall, the algorithms performed mostly as expected. A few errors were encounter in the way of segmentation faults at high element numbers, and the student created insertion sort displayed strange behavior, but otherwise each algorithm performed according to their theoretical proficiencies. Another consideration is that at smaller values the sort time may be less reliant on the algorithm and time simply due to latency or context switching. Values that are measuring a few thousandths of a second are comparably variable.

# ACKNOWLEDGMENTS

Our thanks to Hyoil Han for providing the opportunity to analyze these different algorithms for educational purposes and providing the necessary materials and knowledge base to explore the topic.