

**PA3 Report**

By

**Wes Beard**

**CSI 281**

**Data Structures & Algorithms**

**Fall 2019-20**

1. **Introduction**

This program tests and reports the runtime of 6 sorting algorithms: bubble sort, insertion sort, selection sort, shell sort, merge sort, and quick sort. The testing is based on factors of both data set size and pre-sorting state. I expect generally higher runtimes for the first three less complex algorithms, especially when it comes to large data sets. Similarly, the last three more advanced algorithms should see relatively high speeds for most tests.

1. **Background**

Bubble Sort

Works by procedurally comparing neighboring data and swapping accordingly. Very simple, takes little code to implement. Can easily tell when data is completely sorted. It has an average time complexity of *n2*.2 Especially fast at sorting best case scenario, and relatively fast at worst case depending on the data set size. Seriously lacks at sorting randomized data as well as struggles with large data sets.

Insertion Sort

Sorts data by working through data one by one and comparing it with every previous piece of data and swapping it with the last one larger than it.1 Similar to bubble sort it has an average time complexity of *n2*but also a best case of *n*.2 Another relatively simple algorithm, it is faster than bubble sort but still relatively slow. Although it does particularly excel at sorting the best case scenario.

Selection Sort

This algorithm works by creating two lists, one of unsorted data and one of the sorted data, finding the smallest data in the unsorted array and putting it at the end of the sorted array, repeating until there is no data in the unsorted array.1 It has an across-the-board complexity of *n2*.2 It’s still relatively simple compared to the more complex algorithms, but the overall speed is worse than insertion sort, although it can be more predictable as its best, worst and random, cases are the same complexity.

Shell Sort

Shell sort is an iteration of insertion sort that works by splitting up the data based on a gap and then comparing it until the gap is reduced to one and the data is sorted.1 It has a time complexity of *nlogn*, but this is highly variable.2 This implementation can be faster than previous algorithms but its speed can also be highly dependent on the gap that it is given. This algorithm also relies on bubble sort to tell when the sort is finished.

Merge Sort

This sort works by the concept of divide and conquer, halving the set recursively, sorting the pieces, and merging them back together into a sorted set.1 It has a time complexity of *nlogn*.2 Merge sort sports a very good time complexity at the cost of more total code and the risk of incorrect recursion. It is much faster than the previous algorithms for large data sets, and similar in speed for smaller ones.

Quick Sort

Similarly to merge sort, quick sort is a divide and conquer recursive algorithm. It uses a pivot that is used as a comparison for other data in the list, swapping data in relation to its value. It then works through the list recursively until sorted.1 It has a time complexity of *nlogn*.2 Again similar to merge sort it requires more code than others as well as being recursive. It has a similar runtime to merge sort but is generally faster for most applications.

**Implementation Detail**

These algorithms were implemented in a program that would procedurally test each one with different arrays representing various sorting scenarios. These scenarios were: a completely random data set, one that had been pre-sorted in ascending order representing the best case scenario, and one that had been pre-sorted in descending order representing the worst case scenario. Each algorithm was also tested with arrays of different sizes: 100, 10,000, and 500,000. The program operates by first generating a random set of data, storing it in a specified file, and then sorting it in both descending and ascending order. Once this has been done once for all future runs the program will check if the respective files exist and skip the process to save time on data generation. This data is then entered into the various arrays used in the testing process.

1. **Experimentation Detail**

Platform Specifications:

* 1. RAM: 16GB DDR4 @ 2133mhz
  2. CPU: i7 6700k 4 core 8 thread
  3. Clock: 4.2Ghz
  4. OS: Windows 64bit

**Summary Data**

The following summarized data represents the average runtime of each algorithm for a random case (Dataset #1), best case (#2), and worst case (#3). Times are represented in seconds. Expanded data tables can be seen in the Appendix.

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm: Bubble Sort** | | | |
| **N** | **Dataset #1[[1]](#footnote-0)** | **Dataset #2** | **Dataset #3** |
| 100 | 0.0000271 | 0.0000003 | 0.0000249 |
| 10,000 | 0.3145261 | 0.0000167 | 0.2597254 |
| 500,000 | 814.7690159 | 0.0008680 | 622.7260078 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm: Insertion Sort** | | | |
| **N** | **Dataset #1** | **Dataset #2** | **Dataset #3** |
| 100 | 0.0000065 | 0.0000004 | 0.0000102 |
| 10,000 | 0.1443285 | 0.0000263 | 0.1031709 |
| 500,000 | 125.6595650 | 0.0014399 | 251.5590071 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm: Selection Sort** | | | |
| **N** | **Dataset #1** | **Dataset #2** | **Dataset #3** |
| 100 | 0.0000127 | 0.0000092 | 0.0000159 |
| 10,000 | 0.0926664 | 0.0831198 | 0.1433691 |
| 500,000 | 208.9156917 | 206.9122984 | 416.1477015 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm: Shell Sort** | | | |
| **N** | **Dataset #1** | **Dataset #2** | **Dataset #3** |
| 100 | 0.0000159 | 0.0000013 | 0.0000033 |
| 10,000 | 0.1419709 | 0.0002259 | 0.0005092 |
| 500,000 | 388.9477923 | 0.0177784 | 0.0320630 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm: Merge Sort** | | | |
| **N** | **Dataset #1** | **Dataset #2** | **Dataset #3** |
| 100 | 0.0000778 | 0.0000560 | 0.0000657 |
| 10,000 | 0.0076902 | 0.0056939 | 0.0068885 |
| 500,000 | 0.8266640 | 0.6730552 | 1.3155898 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm: Quick Sort** | | | |
| **N** | **Dataset #1** | **Dataset #2** | **Dataset #3** |
| 100 | 0.0000072 | 0.0000039 | 0.0000053 |
| 10,000 | 0.0010831 | 0.0004686 | 0.0004219 |
| 500,000 | 0.0630694 | 0.0233022 | 0.0233224 |

1. **Discussion and Conclusion**

**Overall Results**

The results that were collected were more or less what was expected, bar a few exceptions and surprises. As the algorithms got more advanced the time to sort was decreased, but I was impressed at how drastically an algorithm like quicksort was able to improve sort time over an algorithm like bubble sort or selection sort. For instance what took bubble sort 15 minutes to complete took quick sort less than a second. I was expecting improved performance but not to that degree. I was also surprised by how some algorithms like bubble sort were able to sort worst case scenario data faster than they could sort random data, although this makes sense considering how bubble sort works.

**Data Set Size**

The size of the data set had a significant impact on the runtime of the less advanced algorithms like bubble and selection sort, as the data sets got bigger the runtime increased exponentially. Increasing the data set size always increased runtime, but this impact was not nearly as noticeable in the more advanced divide and conquer algorithms like merge and quicksort which were able to sort them in a reasonable time frame regardless of the size increase.

**Data Set Scenario**

The data types that were used, random, best, and worst case, also had a pretty large impact on the individual algorithms. Different algorithms had very different efficiencies at dealing with random, or sorted data. For instance, bubble sort was very slow at sorting random data and worst case data but could run through a pre-sorted set quickly regardless of size. Other algorithms like shell sort could sort both best and worst case scenarios easily but struggle greatly trying to sort through large amounts of randomized data

**Ranking**

The following are ranked best to worst based on the performance, consistency and usability they exhibited in this experiment.

1. Quick Sort
   1. Fastest and most reliable in most scenarios, always sorted in under a second independent of data type or size.
2. Merge Sort
   1. Similar to quick sort in speed and reliability but just a little bit slower and unpredictable.
3. Insertion Sort
   1. Although it doesn’t have ideal times for random and worst case data it is relatively consistent and quick for its level of complexity.
4. Selection Sort
   1. Overall slower than insertion sort but very consistent in its times.
5. Shell Sort
   1. Faster or similar to both insertion sort and selection sort for sorting best and worst case data but much slower at sorting significant amounts of randomized data, which is feel is the most prominent use case.
6. Bubble Sort
   1. Slowest for all but best case scenario, which is the least useful case.

**Favorite**

My personal favorite of the algorithms would have to be quicksort due to its raw performance. It is not the most practical in terms of code size, but it performs the best, which is frequently the driving factor in choosing an algorithm.

1. **References**

[1]"Sorting Algorithms - GeeksforGeeks", *GeeksforGeeks*, 2019. [Online]. Available: https://www.geeksforgeeks.org/sorting-algorithms/. [Accessed: 29- Sep- 2019].

[2]"Sorting algorithm", *En.wikipedia.org*, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Sorting\_algorithm. [Accessed: 29- Sep- 2019].

1. **Appendix**

**Bubble Sort**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Bubble Sort** | | | **Dataset #:** Random | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0000237 | 0.0000236 | 0.0000341 | 0.0000271 |
| 10,000 | 0.2990550 | 0.3230374 | 0.3214860 | 0.3145261 |
| 500,000 | 789.9843792 | 828.4983192 | 825.8243495 | 814.7690159 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Bubble Sort** | | | **Dataset #:** Best Case | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0000003 | 0.0000003 | 0.0000003 | 0.0000003 |
| 10,000 | 0.0000167 | 0.0000166 | 0.0000167 | 0.0000167 |
| 500,000 | 0.0008418 | 0.0009078 | 0.0008545 | 0.0008680 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Bubble Sort** | | | **Dataset #:** Worst Case | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0000247 | 0.0000250 | 0.0000250 | 0.0000249 |
| 10,000 | 0.2667036 | 0.2541139 | 0.2583589 | 0.2597254 |
| 500,000 | 614.3172865 | 632.0789192 | 621.7818177 | 622.7260078 |

**Insertion Sort**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Insertion Sort** | | | **Dataset #:** Random3 | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0000073 | 0.0000061 | 0.0000062 | 0.0000065 |
| 10,000 | 0.0525832 | 0.0589164 | 0.3214860 | 0.1443285 |
| 500,000 | 120.6891151 | 127.6681485 | 128.6214314 | 125.6595650 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Insertion Sort** | | | **Dataset #:** Best Case | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0000003 | 0.0000004 | 0.0000005 | 0.0000004 |
| 10,000 | 0.0000262 | 0.0000263 | 0.0000266 | 0.0000263 |
| 500,000 | 0.0013128 | 0.0016844 | 0.0013227 | 0.0014399 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Insertion Sort** | | | **Dataset #:** Worst Case | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0000102 | 0.0000104 | 0.0000102 | 0.0000102 |
| 10,000 | 0.1065162 | 0.1040604 | 0.0989362 | 0.1031709 |
| 500,000 | 249.8482032 | 256.6971668 | 248.1316514 | 251.5590071 |

**Selection Sort**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Selection Sort** | | | **Dataset #:** Random | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0000115 | 0.0000116 | 0.0000152 | 0.0000127 |
| 10,000 | 0.0886637 | 0.0971310 | 0.0922046 | 0.0926664 |
| 500,000 | 204.0476371 | 211.1778774 | 211.5215608 | 208.9156917 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Selection Sort** | | | **Dataset #:** Best Case | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0000099 | 0.0000090 | 0.0000089 | 0.0000092 |
| 10,000 | 0.0838200 | 0.0833525 | 0.0821870 | 0.0831198 |
| 500,000 | 206.1481340 | 211.1468489 | 203.4419125 | 206.9122984 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Selection Sort** | | | **Dataset #:** Worst Case | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0000171 | 0.0000153 | 0.0000153 | 0.0000159 |
| 10,000 | 0.1464605 | 0.1457395 | 0.1379073 | 0.1433691 |
| 500,000 | 415.6892027 | 422.0704858 | 410.6834161 | 416.1477015 |

**Shell Sort**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Shell Sort** | | | **Dataset #:** Random | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0000123 | 0.0000122 | 0.0000159 | 0.0000159 |
| 10,000 | 0.1281087 | 0.1546391 | 0.1431651 | 0.1419709 |
| 500,000 | 381.8031382 | 389.5613517 | 395.4788870 | 388.9477923 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Shell Sort** | | | **Dataset #:** Best Case | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0000012 | 0.0000014 | 0.0000013 | 0.0000013 |
| 10,000 | 0.0002259 | 0.0002260 | 0.0002260 | 0.0002259 |
| 500,000 | 0.0167119 | 0.0200051 | 0.0166183 | 0.0177784 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Shell Sort** | | | **Dataset #:** Worst Case | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0000033 | 0.0000034 | 0.0000032 | 0.0000033 |
| 10,000 | 0.0006135 | 0.0004626 | 0.0004515 | 0.0005092 |
| 500,000 | 0.0329878 | 0.0324784 | 0.0307228 | 0.0320630 |

**Merge Sort**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Merge Sort** | | | **Dataset #:** Random | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0000769 | 0.0000609 | 0.0000956 | 0.0000778 |
| 10,000 | 0.0079261 | 0.0069574 | 0.0081871 | 0.0076902 |
| 500,000 | 1.4121511 | 0.5534283 | 0.5144127 | 0.8266640 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Merge Sort** | | | **Dataset #:** Best Case | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0000585 | 0.0000534 | 0.0000562 | 0.0000560 |
| 10,000 | 0.0065782 | 0.0054683 | 0.0050354 | 0.0056939 |
| 500,000 | 1.4304127 | 0.3087977 | 0.2799552 | 0.6730552 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Merge Sort** | | | **Dataset #:** Worst Case | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0000574 | 0.0000555 | 0.0000844 | 0.0000657 |
| 10,000 | 0.0092209 | 0.0055600 | 0.0058847 | 0.0068885 |
| 500,000 | 1.4095758 | 1.3205363 | 1.2166574 | 1.3155898 |

**Quick Sort**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Quick Sort** | | | **Dataset #:** Random | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0000070 | 0.0000066 | 0.0000081 | 0.0000072 |
| 10,000 | 0.0010223 | 0.0010232 | 0.0012040 | 0.0010831 |
| 500,000 | 0.0610542 | 0.0630991 | 0.0650550 | 0.0630694 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Quick Sort** | | | **Dataset #:** Best Case | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0000040 | 0.0000038 | 0.0000039 | 0.0000039 |
| 10,000 | 0.0004992 | 0.0004989 | 0.0004078 | 0.0004686 |
| 500,000 | 0.0224977 | 0.0256009 | 0.0218080 | 0.0233022 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Quick Sort** | | | **Dataset #:** Worst Case | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0000061 | 0.0000045 | 0.0000053 | 0.0000053 |
| 10,000 | 0.0004391 | 0.0004325 | 0.0003943 | 0.0004219 |
| 500,000 | 0.0233698 | 0.0234783 | 0.0231193 | 0.0233224 |

1. Dataset #1 is average case, dataset #2 is best case and dataset #3 is worst case. [↑](#footnote-ref-0)