# Comp 103 Assignment 10, Sort efficiency report

Daniel Armstrong;

armstrdani1;

300406381;

Introduction:

This report focuses on the efficiency of several different sort method, using the IDE intelliJ and on a laptop with an 7th gen i7 processor, so results gathered here may vary to results from others.

Method: My method for testing was relatively straight forward, instead of meticulously and slowly gathering data individual, by altering the given program, I wrote a few methods to allow me to cycle through different sort methods, with different array sizes and orders. Allowing me to efficiently get the results I need by printing the results to a csv file, which I then could sort to a table. (See the commented code attached). Using for loops, the idea behind my methods, was for each value in an arraySize array it would create an array of the size and populate it with a random assortment of strings, for each time this would happen it would select a sort method, and for each time this happened it would run that sort method five times, while checking if the average time was greater than 60 seconds, if so it would then break out of the method(this was to prevent the tests for taking too long). The program would then send each measurement of time it would be written to a csv file, which I could read and then tabulate more effectively to graph the data, which allowed me to check the complexities using the averages sent to the csv file.

Test Results:

| Selection Sort | 256 | 512 | 1024 | 8192 | 65536 | 131072 | 262144 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Unsorted | 0.0002 | 0.0006 | 0.0022 | 0.0082 | 5.4972 | 70.1366 | 139.629 | | | | |
| Sorted | 0.0002 | 0.0004 | 0.0026 | 0.1186 | 5.4918 | 27.802 | 139.225 | | | | |
| Reversed | 0.0002 | 0.0006 | 0.0028 | 0.0704 | 1.1012 | 5.5686 | 129.125 | | | | |
| | | | | | | | | | | | |
| Selection Sort 2 | 256 | 512 | 1024 | 8192 | 65536 | 131072 | 262144 | | | | |
| Unsorted | 0.0002 | 0.0012 | 0.0046 | 0.014 | 47.4956 | 64.0834 | 237.624 | | | | |
| Sorted | 0 | 0.0012 | 0.003 | 0.2132 | 10.5396 | 51.9142 | 236.514 | | | | |
| Reversed | 0.0004 | 0.0014 | 0.0046 | 0.079 | 2.137 | 10.4226 | 222.265 | | | | |
| | | | | | | | | | | | |
| Insertion Sort | 256 | 512 | 1024 | 8192 | 65536 | 131072 | 524288 | 1048576 | 4194304 | 8388608 | 1.7E+07 |
| Unsorted | 0.0002 | 0 | 0.0002 | 0.0002 | 0.013 | 0.0304 | 0.1254 | 0.3286 | 1.5122 | 3.3976 | 15.1464 |
| Sorted | 0 | 0 | 0 | 0.002 | 0.0068 | 0.011 | 0.1266 | 0.3278 | 1.5066 | 3.389 | 15.1976 |
| Reversed | 0.0002 | 0 | 0.0002 | 0.0014 | 0.0072 | 0.011 | 0.075 | 0.109 | 0.3026 | 0.679 | 3.045 |
| | | | | | | | | | | | |
| Insertion Sort 2 | 256 | 512 | 1024 | 8192 | 65536 | 131072 | 524288 | 1048576 | 4194304 | 8388608 | 1.7E+07 |
| Unsorted | 0 | 0 | 0 | 0 | 0.0044 | 0.0074 | 0.0298 | 0.0526 | 0.1736 | 0.347 | 1.632 |
| Sorted | 0.0002 | 0 | 0.0002 | 0.0008 | 0.0014 | 0.0024 | 0.024 | 0.0532 | 0.1748 | 0.3462 | 1.6394 |
| Reversed | 0 | 0 | 0 | 0.0006 | 0.0018 | 0.0028 | 0.0228 | 0.043 | 0.0696 | 0.0702 | 0.3284 |
| | | | | | | | | | | | |
| Merge Sort | 256 | 512 | 1024 | 8192 | 65536 | 131072 | 524288 | 1048576 | 4194304 | 8388608 | 1.7E+07 |
| Unsorted | 0 | 0 | 0.0002 | 0 | 0.0116 | 0.0238 | 0.0932 | 0.231 | 1.1756 | 2.1212 | 10.9234 |
| Sorted | 0 | 0 | 0 | 0.0006 | 0.0066 | 0.0116 | 0.0928 | 0.2312 | 0.9834 | 2.1288 | 10.91 |
| Reversed | 0 | 0.0004 | 0 | 0.0016 | 0.006 | 0.0094 | 0.0764 | 0.0804 | 0.1978 | 0.4284 | 1.8298 |
| | | | | | | | | | | | |
| Merge Sort 2 | 256 | 512 | 1024 | 8192 | 65536 | 131072 | 524288 | 1048576 | 4194304 | 8388608 | 1.7E+07 |
| Unsorted | 0 | 0 | 0.0002 | 0.002 | 0.0238 | 0.0286 | 0.1462 | 0.3348 | 1.7228 | 3.5506 | 12.8052 |
| Sorted | 0.0002 | 0.0002 | 0.0002 | 0.0018 | 0.0178 | 0.0268 | 0.129 | 0.3042 | 1.7856 | 1.9 | 10.0808 |
| Reversed | 0.0002 | 0 | 0.0004 | 0.0016 | 0.0138 | 0.026 | 0.0722 | 0.1082 | 0.1224 | 0.3264 | 1.6962 |
| | | | | | | | | | | | |
| Array.Sort | 256 | 512 | 1024 | 8192 | 65536 | 131072 | 524288 | 1048576 | 4194304 | 8388608 | 1.7E+07 |
| Unsorted | 0 | 0 | 0.0004 | 0.0004 | 0.0218 | 0.0182 | 0.1288 | 0.1364 | 0.6326 | 1.5892 | 4.3986 |
| Sorted | 0 | 0 | 0 | 0.0004 | 0.0012 | 0.0062 | 0.0224 | 0.0518 | 0.165 | 0.3368 | 1.6414 |
| Reversed | 0.0002 | 0 | 0 | 0.0006 | 0.0022 | 0.0026 | 0.034 | 0.0634 | 0.12 | 0.2674 | 1.1686 |
| | | | | | | | | | | | |
| quickSort | 256 | 512 | 1024 | 8192 | 65536 | 131072 | 524288 | 1048576 | 4194304 | 8388608 | 1.7E+07 |
| Unsorted | 0 | 0 | 0 | 0.0012 | 0.0138 | 0.0328 | 0.1464 | 0.3444 | 1.8382 | 4.8988 | 15.7106 |
| Sorted | 0 | 0 | 0 | 0.0018 | 0.0074 | 0.0284 | 0.1276 | 0.3432 | 1.507 | 3.2682 | 15.6262 |
| Reversed | 0 | 0.0004 | 0.0002 | 0.0014 | 0.009 | 0.013 | 0.076 | 0.0694 | 0.3018 | 0.6558 | 3.1248 |
| | | | | | | | | | | | |
| quickSort2 | 256 | 512 | 1024 | 8192 | 65536 | 131072 | 524288 | 1048576 | 4194304 | 8388608 | 1.7E+07 |
| Unsorted | 0 | 0 | 0 | 0.0002 | 0.0154 | 0.0312 | 0.1368 | 0.3524 | 1.8694 | 4.9894 | 15.7822 |
| Sorted | 0 | 0 | 0 | 0.001 | 0.0078 | 0.0356 | 0.1312 | 0.3622 | 1.5392 | 3.3404 | 15.917 |
| Reversed | 0 | 0 | 0.0002 | 0.0016 | 0.009 | 0.0112 | 0.0778 | 0.0704 | 0.3068 | 0.6732 | 3.1254 |

Figure 1, 2 and 3- https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/divide-and-conquer-algorithms (22/10/17)

Graphs

Selection Sort, And improved Selection sort:


Insertion Sort and improved Insertion sort:

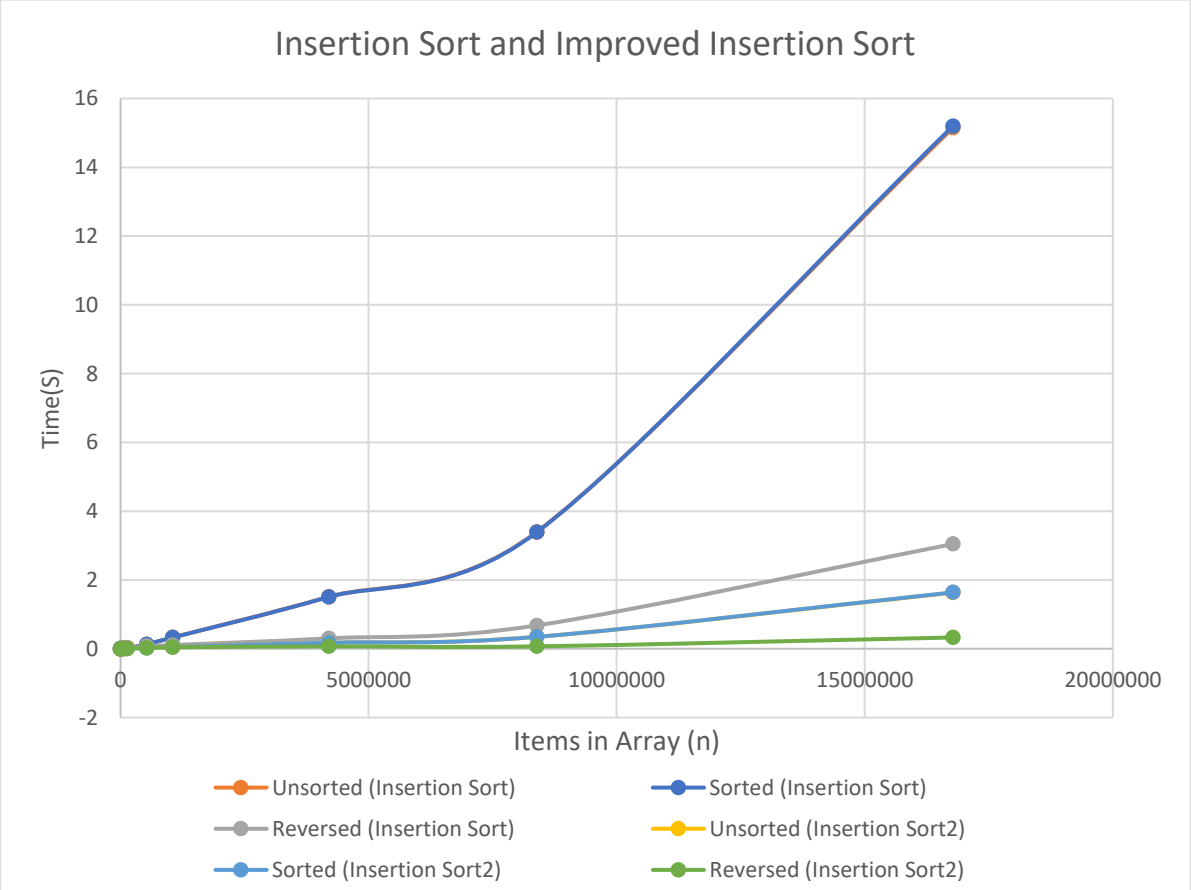
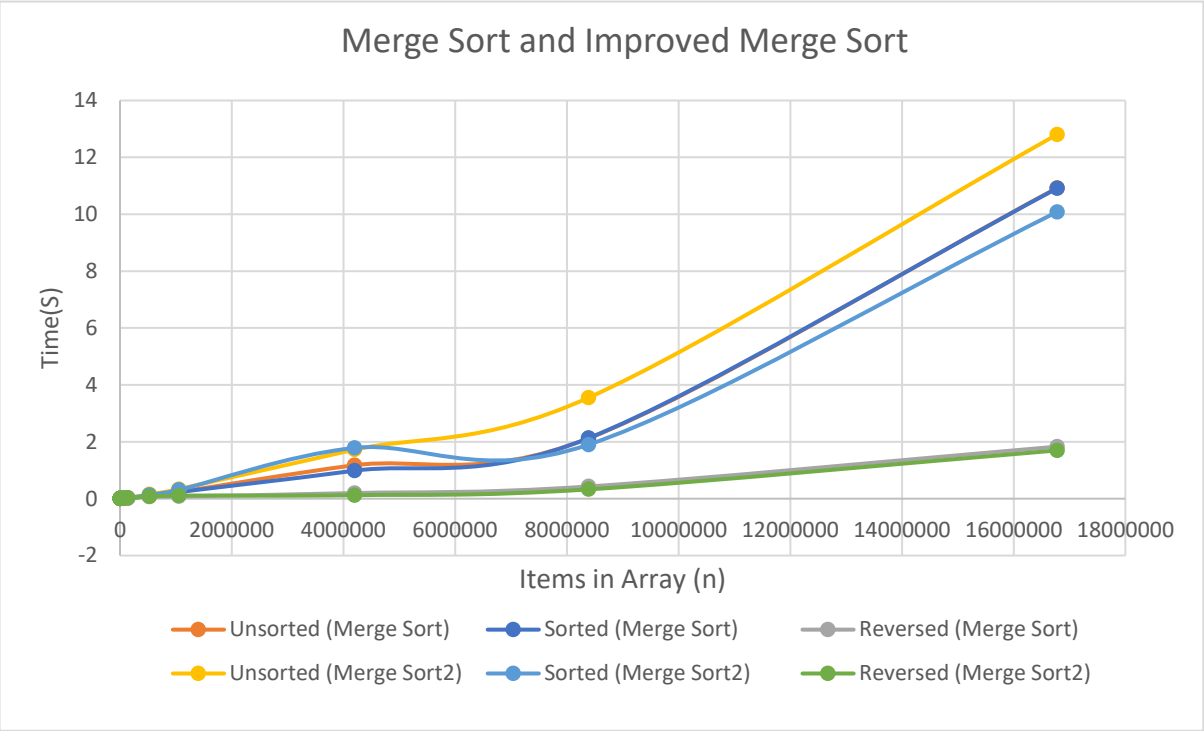



Figure 1, 2 and 3- https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/divide-and-conquer-algorithms (22/10/17)

Merge Sort and Merge insertion sort




Quick Sort and Quick insertion sort

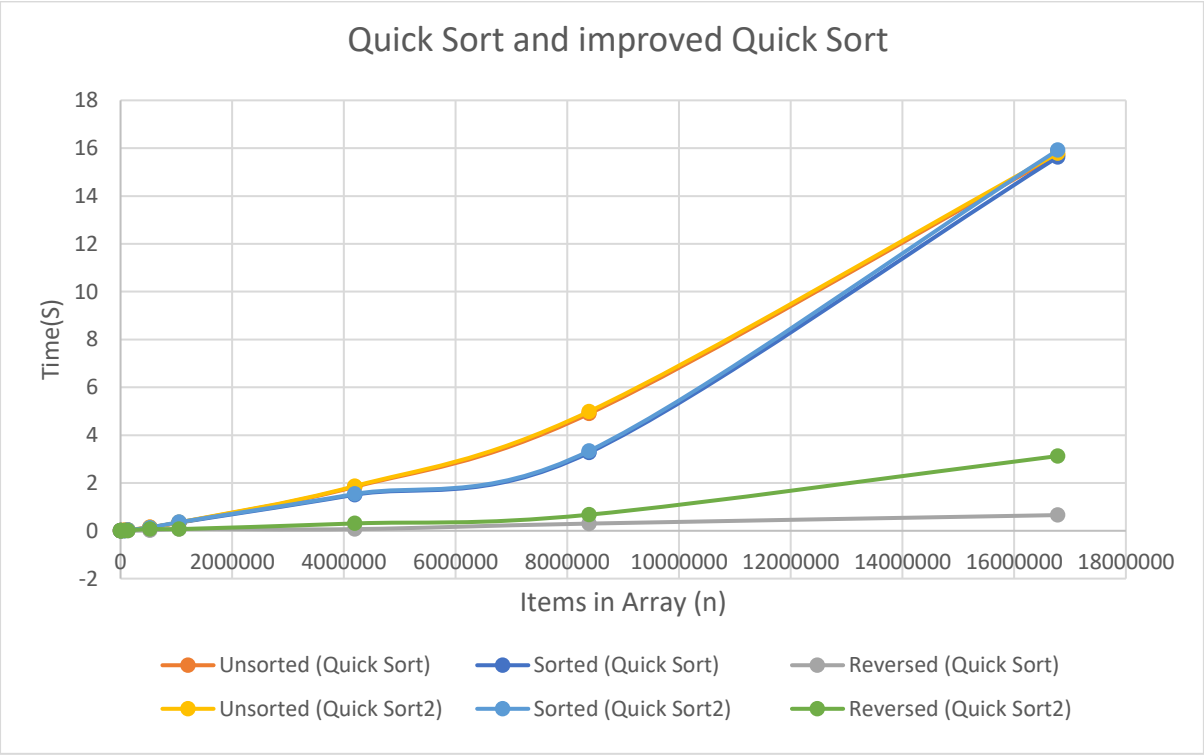
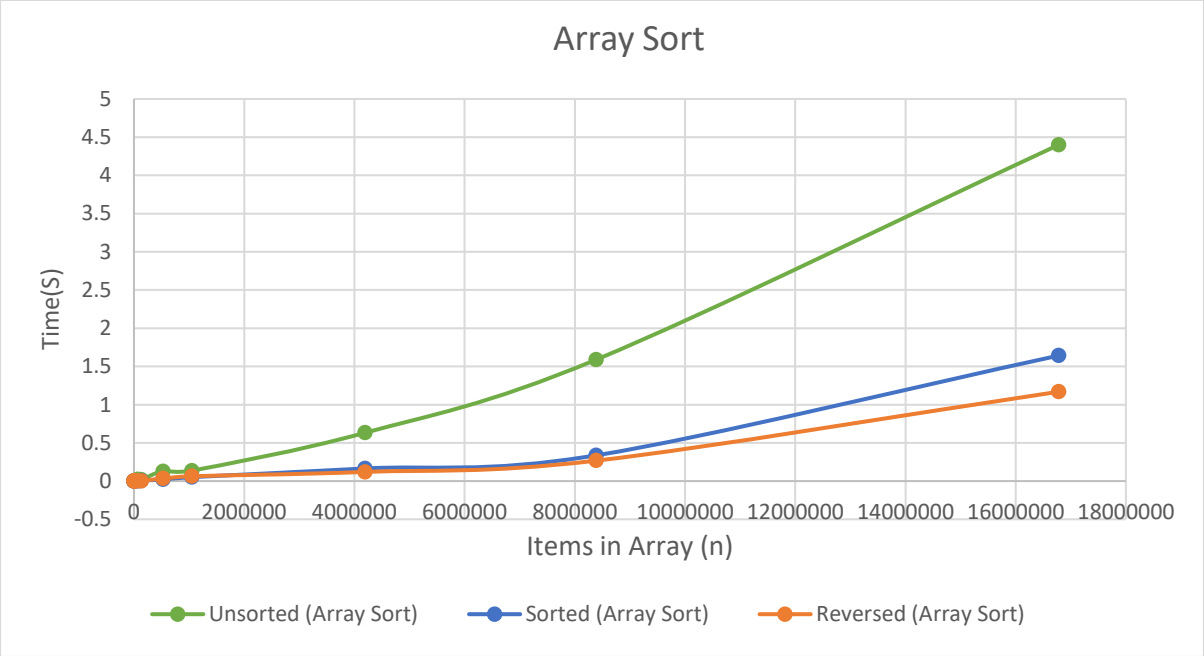



Figure 1, 2 and 3- https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/divide-and-conquer-algorithms (22/10/17)

Array Sort:




Discussion:

After running and graphing the data, we see an obvious trend, from the data we can see that the longest time taken is for unsorted arrays. The most likely cause for this is because Unsorted often is the most average case in complexity. Looking at the sorted Arrays the cost is drop drastically, and we see the best case for many of the sorting algorithms, not all. And lastly the reversed arrays often are the worst-case complexity, because for algorithms like selection sort the algorithm must then move all items, not just the out of place items. With reverse arrays it often is not as bad ad unsorted, as an unsorted array has a random order which takes more computing time to solve. With the improved algorithms only, a few of them drastically improved performant one notable algorithm improved was insertion sort as from what we see at the graph with the maximum number of items (16 million). Coming ahead of the array sort which up until that test was the fasted and most efficient algorithm. Using so many items, was to effectively test the power of the sorting algorithms the range used was 2^i, i being 1→24 to get an appropriate range of data sets. The sorting algorithms used in this investigation were of varying complexities with the main complexity of O(n^2).

*Figure 1*

| Algorithm | Worst-case running time | Best-case running time | Average-case running time |
|---|---|---|---|
| Selection sort | $\Theta(n^2)$ | $\Theta(n^2)$ | $\Theta(n^2)$ |
| Insertion sort | $\Theta(n^2)$ | $\Theta(n)$ | $\Theta(n^2)$ |
| Merge sort | $\Theta(n \lg n)$ | $\Theta(n \lg n)$ | $\Theta(n \lg n)$ |
| Quicksort | $\Theta(n^2)$ | $\Theta(n \lg n)$ | $\Theta(n \lg n)$ |

Figure 1, 2 and 3- https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/divide-and-conquer-algorithms (22/10/17)

The above complexities can be graphed, which can be seen in the above graphs though not as prominent. The different sorting algorithms have different ideas behind them when it comes to merge and quick sort, both employ a divide and conquer idea that works best with recursion like the binary search algorithm we examined previously this pretty much means it takes a list then divides that list into two sections based on a centre point or pivot point. The correct subsists created by the divide constantly get smaller until it reaches the solution this method is relatively quick which is shown in the above data as quick sort is a very efficient algorithm. The recursive algorithm can be shown like in figure 3. The results that were gathered depended on other factors, Hardware (The computer the tests were run on was modern hence better hardware), what was being run on the computer (How the memory is being used at the time). So not just the algorithm and how the algorithm is written. By looking at the results we do see the improved versions of each the algorithms are more efficient some more than others but there are improvements. We see the complexities of each of the algorithms do agree with the ones researched.

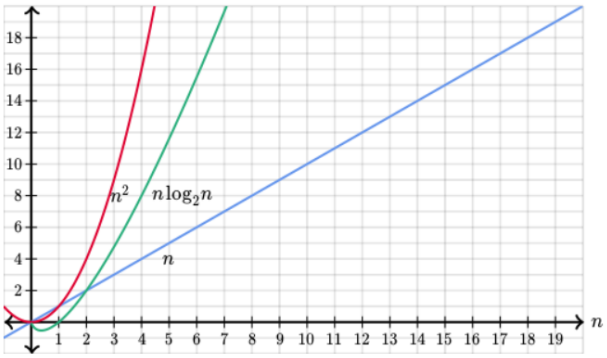



*Figure 2*

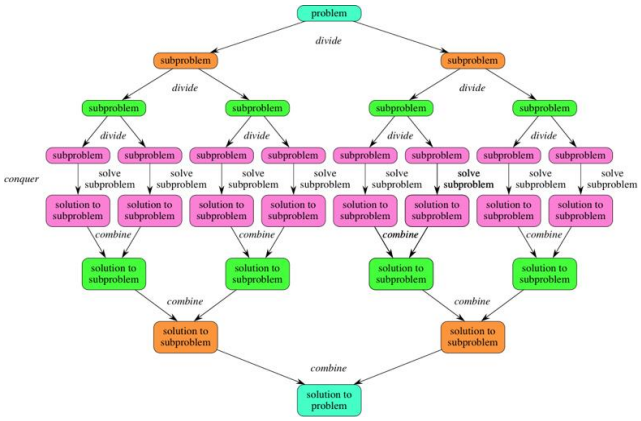



*Figure 3*

Comment. This report is horribly written I apologize this assignment took me ages I do apologize and yeah, thank you for not failing me this trimester good luck with the exams peeps.

Figure 1, 2 and 3- https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/divide-and-conquer-algorithms (22/10/17)

Figure 1, 2 and 3- https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/divide-and-conquer-algorithms (22/10/17)