Sorting Algorithm Analysis

There are different sorts to be used in different applications. In this analysis, we test the speed and efficiency of 5 different sorts for two different sized arrays. Additionally, we will be seeing if the array being already sorted has an impact on how fast the sort is done by the program. We closely analyze Insertion, Heap, Radix, Quick and Merge sort. All 5 of these sorts have different time complexities, with some better than others depending on the situation. Below, the data retrieved from the testing has been displayed in tables for each sort. We will be expanding on the data in the respective section and hypothesizing that an already sorted array takes less time to sort than a randomized array.

1. Insertion Sort

Random Sort # (Sample size)	Sorted Sort # (Sample size)	Random Sort (250,000) Time (s)	Sorted Sort (250,000) Time (s)	Random Sort (500,000) Time (s)	Sorted Sort (500,000) Time (s)
1		2.993	0.002	13.521	0.001
2		3.645	0	11.832	0.001
3		3.65	0	11.896	0.001
4		2.955	0.001	11.86	0.001
5		4.166	0.001	11.815	0
6		2.949	0	11.881	0.001

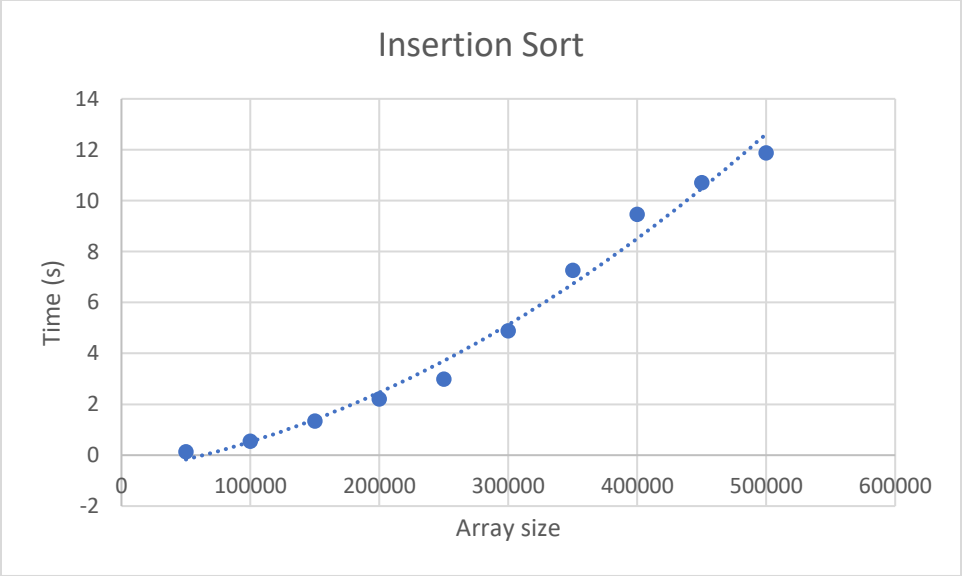

Fig 1: Insertion sort complexity

Insertion sort has a Big-O time complexity in the worst-case scenario of $O(n^2)$ and of $O(k \cdot n)$ when array is already sorted. This means that the running times increases quadratically as the number of values increases linearly for a random sort and that the running times increases linearly when the number of values increases linearly for a sorted sort. The difference between the time for the two different array sizes is large. On average, for the randomized arrays, the smaller array sorts 3.64 times

faster than the larger array. This obviously shows that the larger the array size the longer it takes for the algorithm to sort a random array. However, when the array is already sorted, the time taken to re-sort the array is very similar for both array sizes. Additionally, what is interesting is that the time taken to resort the sorted array is 2ms which is a miniscule amount of time. For example, for the first sort, the unsorted sort is 1500 times slower than the re-sort for the smaller array size and 13,521 times slower for the larger array. This is because insertion sort is an adaptive sort, sorting an already sorted is vastly faster than sorting a randomized list. However, the big takeaway from this data is that the longest time taken to sort in the table above is 13.521s, which is a very long period when it comes to algorithms. According to the values obtained from experimentation, this means that insertion sort is not a suitable algorithm for sorting large arrays. Moreover, if you look at the running times for the two array sizes across the several sorts, we see that the running time varies significantly. Fore example, in the 1st sort of the random sort of the larger array we see that it took 13.521s for the sort to be done, and then in the same category for the 5th sort, we see that it took 11.815s. These values differ by 1.706s! This is a significant difference when we examine Merge, Heap and Radix sort as they have very consistent sort times. Although the volatility in run time is not as high as for quick sort, this still classifies insertion sort as inconsistent based on our data above. The perceived growth between the two arrays is on average $O(n^2)$ which is what was expected. The perceived growth for the sorted sort is constant, with the larger array and smaller array having the same time to sort, we predicted that it would be linear with a k constant but that does not seem to be the case here. Finally, insertion sort is also an in-situ sort, therefore the memory requirements are minimal.

2. Heapsort

Random Sort # (Sample size)	Sorted Sort # (Sample size)	Random Sort (250,000) Time (s)	Sorted Sort (250,000) Time (s)	Random Sort (500,000) Time (s)	Sorted Sort (500,000) Time (s)
1		0.021	0.004	0.044	0.02
2		0.019	0.009	0.041	0.023
3		0.018	0.01	0.04	0.019
4		0.018	0.009	0.037	0.018
5		0.017	0.009	0.039	0.018
6		0.022	0.012	0.037	0.018

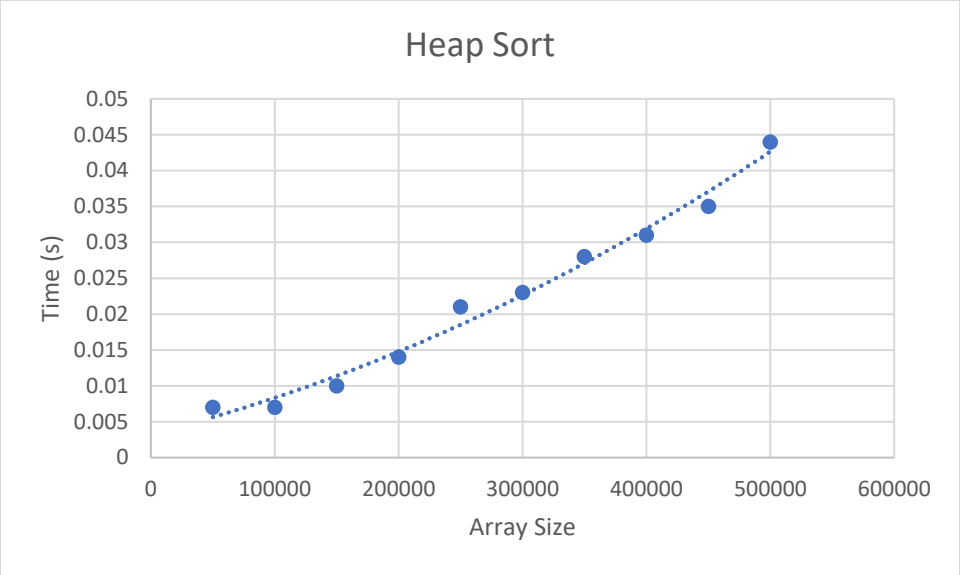

Fig 2: Heap sort complexity

Heap sort has a Big-O time complexity in the worst-case scenario of $O(n \cdot \log(n))$. This means that the running times increases logarithmically and provides us with a very efficient way to sort. The difference between the time for the two different array sizes is not large, however, the smaller array is sorted on average 2.08 times faster than the larger array. This obviously shows that the larger the array size the longer it takes for the algorithm to sort a random array. Similarly, when the array is already sorted, the time taken to re-sort the array follows the same pattern. On average, when the array is already sorted, the smaller array sorts 2.5 times faster than the larger array. However, the big takeaway from this data is that the longest time taken to sort in the table above is 44ms, which is an infinitesimal amount of time. Additionally, if you look at the running times for the same array size across the several sorts, we see that the running time on average for randomized small array is 19.16ms with the deviation only being less than 0.3ms which is again, miniscule. There is a similar pattern observed for the running times with already sorted arrays and the larger array. This shows that heapsort is extremely consistent because heapsort uses slightly fewer comparisons than the worst case-bound suggests. The perceived growth between the two arrays as shown on the graph appears to be following a linear or quadratic growth model. Not the same as $O(n \cdot \log(n))$ which is what was expected. The perceived growth for the sorted sort is the same. Finally, heapsort sort is also an in-situ sort, therefore the memory requirements are minimal.

3. Radix Sort

Random Sort # (Sample size)	Sorted Sort # (Sample size)	Random Sort (250,000) Time (s)	Sorted Sort (250,000) Time (s)	Random Sort (500,000) Time (s)	Sorted Sort (500,000) Time (s)
1		0.021	0.008	0.023	0.018
2		0.022	0.009	0.024	0.019
3		0.017	0.008	0.023	0.017

4	0.022	0.009	0.023	0.019
5	0.021	0.008	0.026	0.016
6	0.021	0.008	0.024	0.018

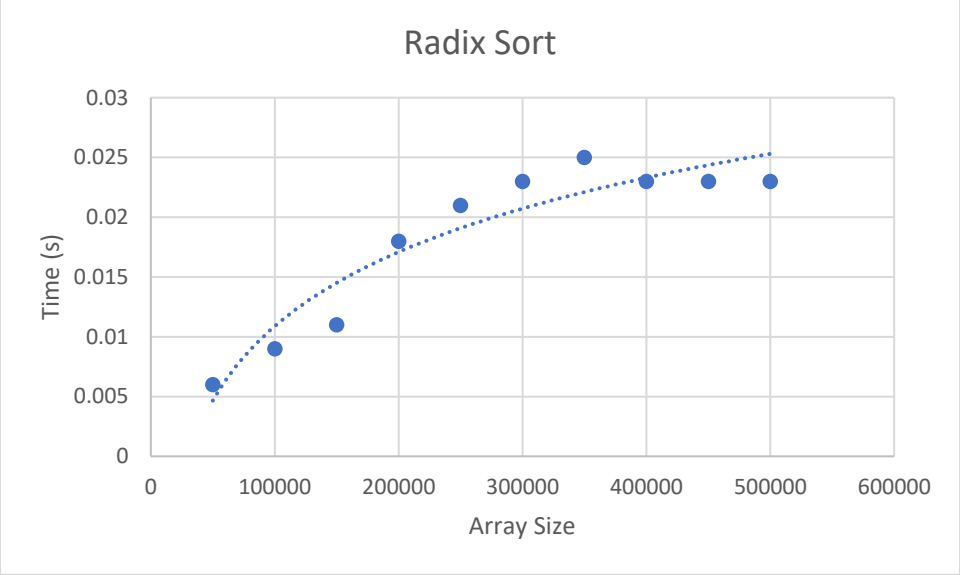

Fig 3: Radix sort time complexity

Radix sort has a Big-O time complexity in the worst-case scenario of $O(k \cdot n)$. This means that the running times increases linearly as the number of values increases linearly. The difference between the time for the two different array sizes is very similar with only a difference of 0.002s. On average, for the randomized arrays, the smaller array sorts 1.34 times faster than the larger array. No matter how miniscule the difference is, it obviously shows that the larger the array size the longer it takes for the algorithm to sort a random array. Similarly, when the array is already sorted, the time taken to re-sort the array follows the same pattern. On average, when the array is already sorted, the smaller array sorts 2.14 times faster than the larger array. This is interesting because there is a very small difference in running time for the two randomized arrays, although the difference is much more significant for the already sorted arrays, it is important to note that the already sorted arrays still sort faster than the randomized arrays. However, the big takeaway from this data is that the longest time taken to sort in the table above is 26ms, which is an infinitesimal amount of time. Additionally, if you look at the running times for the same array size across the several sorts, we see that the running time on average for randomized small array is 20.66ms with the deviation only being less than 0.4ms which is again, miniscule. There is a similar pattern observed for the running times with already sorted arrays and the larger array. This shows that radix sort based on the data above, is consistent. The perceived growth between the two arrays as shown on the graph is polynomial with a decreasing quadratic graph. $O(n)$ is what was expected, however. The perceived growth for the sorted sort is the same. Finally, radix sort is not normally an in-situ sort, therefore the memory requirements can increase with the size of the array which can be a disadvantage.

4. Quick Sort

Random Sort # (Sample size)	Sorted Sort # (Sample size)	Random Sort (250,000) Time (s)	Sorted Sort (250,000) Time (s)	Random Sort (500,000) Time (s)	Sorted Sort (500,000) Time (s)

1	6.007	4.117	29.989	18.388
2	6.214	4.075	22.038	16.818
3	6.742	4.144	32.809	29.279
4	7.356	4.552	33.208	21.956
5	6.7	4.109	32.424	29.243
6	5.947	3.93	37.491	29.323

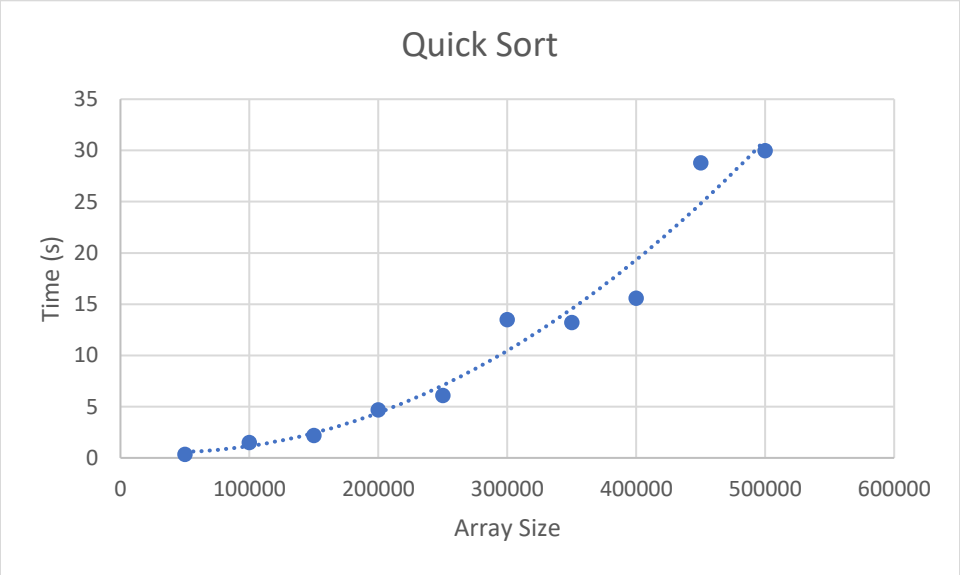

Fig 4: Quick sort time complexity

Quick sort has a Big-O time complexity in the worst-case scenario of $O(n^2)$. This means that the running times increases quadratically as the number of values increases linearly. The difference between the time for the two different array sizes is vastly different. On average, for the randomized arrays, the smaller array sorts 4.83 times faster than the larger array. For example, in the 6th sort, the smaller sort is 6.30 times faster than the larger array. Similarly, when the array is already sorted, the time taken to re-sort the array follows the same pattern. On average, when the array is already sorted, the smaller array sorts 5.84 times faster than the larger array. This is interesting because we used a hybrid sort in this case, where Quick sort is used if the sample size of the array is larger than 10. However, the big takeaway from this data is that the longest time taken to sort in the table above is 37.491s, which is a very long period when it comes to algorithms. According to the values obtained from experimentation, this means that quicksort is not a suitable algorithm for sorting large arrays. Additionally, if you look at the running times for the two array sizes across the several sorts, we see that the running time varies significantly. Fore example, in the 6th sort of the random sort of the larger array we see that it took 37.491s for the sort to be done, and then in the same category for the 2nd sort, we see that it took 22.038s. These values differ by 15.453s! This is a huge difference when we examine Merge, Heap and Radix sort as they have very consistent sort times. There is a similar pattern observed for the running times with already sorted arrays and the larger array. This shows that quick sort based on the data above, is inconsistent. The perceived growth between the two arrays as shown on the graph is $O(n^2)$ which is what was expected. The perceived growth for the sorted sort is the same, which is what was predicted. Finally, quick sort is not normally an in-situ sort, therefore the memory requirements can increase with the size of the array which can be a disadvantage.

5. Merge Sort

Random Sort # (Sample size)	Sorted Sort # (Sample size)	Random Sort (250,000) Time (s)	Sorted Sort (250,000) Time (s)	Random Sort (500,000) Time (s)	Sorted Sort (500,000) Time (s)
1		0.025	0.009	0.046	0.02
2		0.025	0.009	0.051	0.022
3		0.026	0.012	0.049	0.024
4		0.024	0.009	0.044	0.018
5		0.026	0.009	0.05	0.023
6		0.032	0.011	0.049	0.024

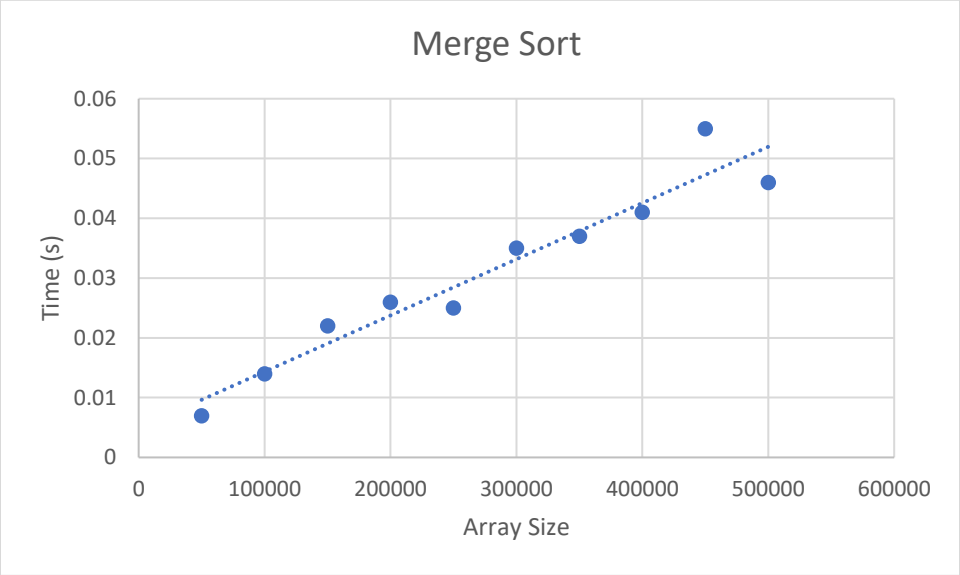

Fig 5: Merge sort time complexity

Merge sort has a Big-O time complexity in the worst-case scenario of $O(n \cdot \log(n))$. This means that the running times increases logarithmically. It is also a great example of recursion. The difference between the time for the two different array sizes is not large, however, the smaller array is sorted on average 1.84 times faster than the larger array. This obviously shows that the larger the array size the longer it takes for the algorithm to sort a random array. Similarly, when the array is already sorted, the time taken to re-sort the array follows the same pattern, with the smaller sorted array, on average, being 2.23 times faster than the larger sorted array. However, the big takeaway from this data is that the longest time taken to sort in the table above is 51ms, which is an infinitesimal amount of time. Additionally, if you look at the running times for the same array size across the several sorts, we see that the running time on average for randomized small array is 26.33ms with the deviation only being less than 0.6ms which is again, miniscule. There is a similar pattern observed for the running times with already sorted arrays and the larger array. This shows that merge sort is extremely consistent. The perceived growth between the two arrays as shown on the graph looks to be linear which is what was not expected but is not completely surprising either. What we expected was $O(n \cdot \log(n))$. The perceived growth for the sorted sort is the same. Merge sort is also not adaptive; therefore, the sorted

array times are not like insertion sort as we observed above. Finally, merge sort is not normally an in-situ sort, therefore the memory requirements can increase with the size of the array which can be a disadvantage.

In conclusion, our hypothesis is proven to be correct, the time it takes for the several algorithms to sort a randomized array differs throughout, but it is always higher than the time it takes to sort an already sorted array. The data above, along with the visualizations should be proof, as the sample size increased the running time increased.