Dmitrius Agoston
Prof. Darrell Long
17 October 2021

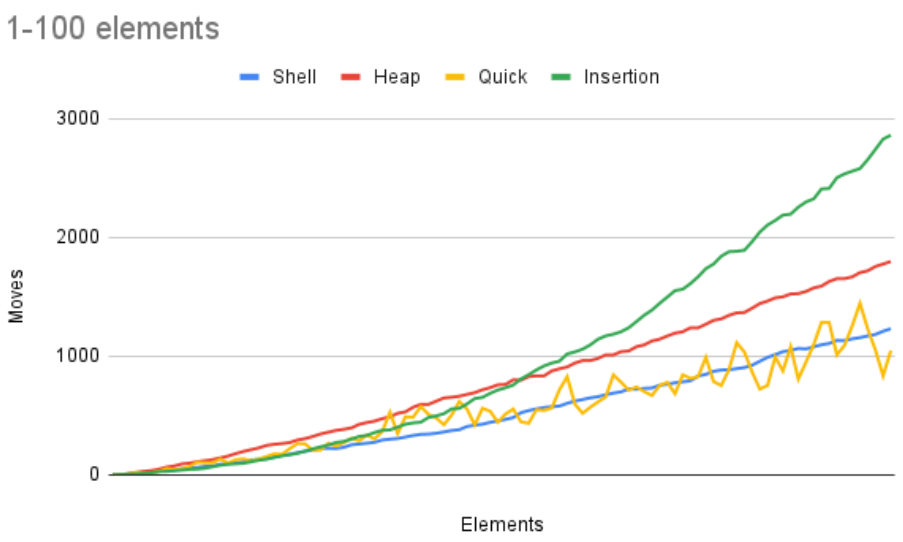
CSE 13S Fall 2021
Assignment 3: Sorting
Writeup

Abstract:
The objective of this assignment was to test the efficiency of four different sorts. The methodology used was giving the four sorts a pseudorandom array of numbers and measuring how efficiently they sorted it by using the number of moves as an indicator. In almost all cases, quick sort was the fastest and most efficient sorting algorithm.

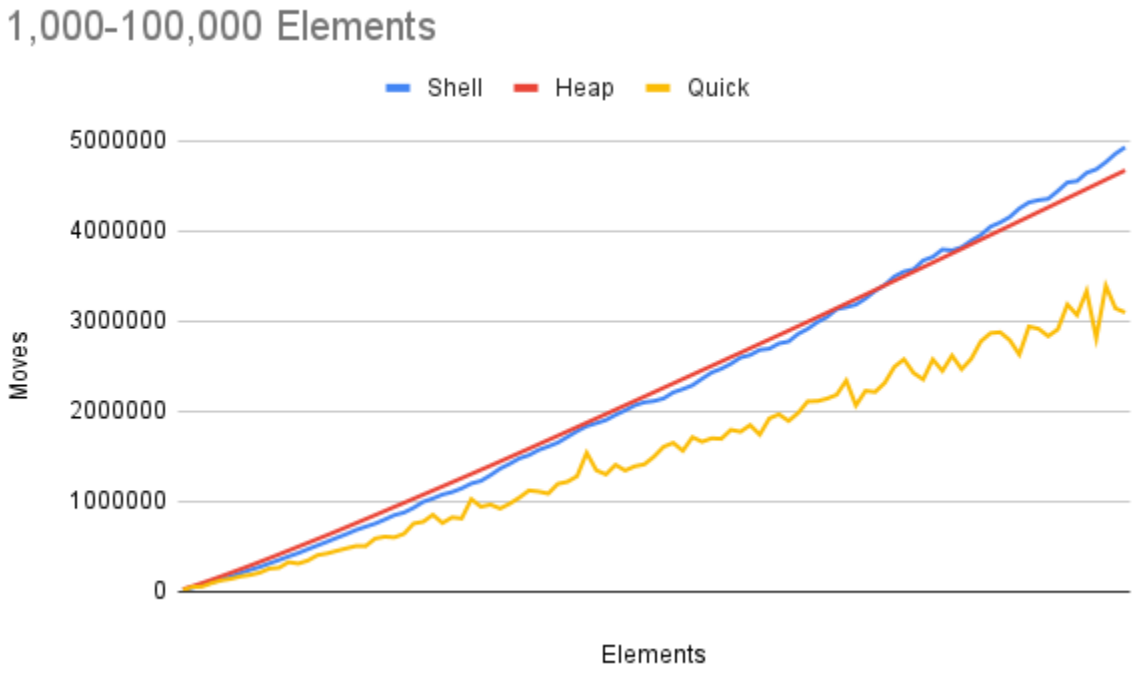
Introduction:
The purpose of this assignment was to investigate the four sorts, insertion, shell, heap, and quick sort. These sorting methods are well known and tested and usually measured by big O notation. The purpose of big O notation is to give these functions a visual to the complexity that they perform in both time and memory. To investigate these sorting algorithms, a test harness was created to run each sort with a pseudorandom array of numbers. Each sort was tasked with sorting the array and after analyzed for how efficiently it was able to sort the given array. The measurement for efficiency in these tests was counting the number of moves that it took to fully sort the algorithm. The results of this investigation show that on average, quick sort is the most efficient, but the other sorts also have their uses under certain test conditions.

Materials and Methods:
The main methods used to test these sorts were implemented in the test harness that was created. The test harness was responsible for deciding which test to run, the seed to be used, the size of the array to be sorted, and how many elements to be listed. The test was run with multiple different arrays and different sizes of those arrays. For showing special cases, the test harness was also given some specifically made arrays that were already ordered.

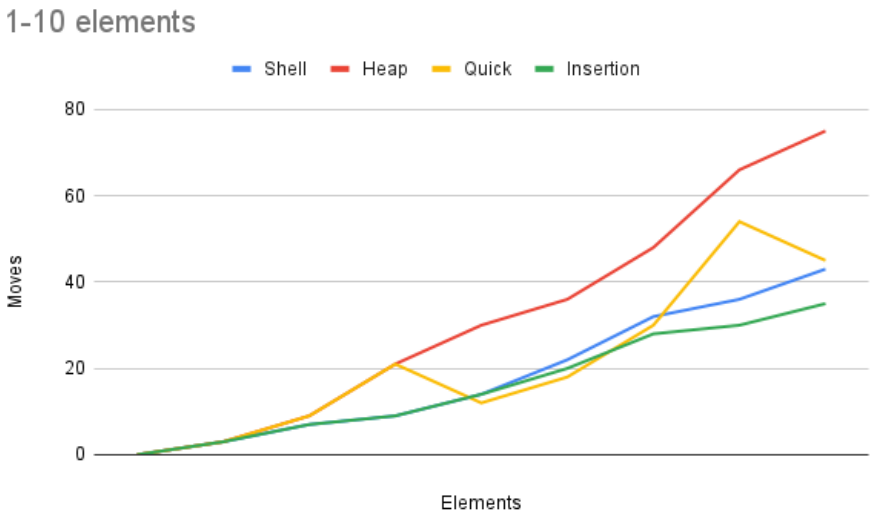
Results:
After analyzing the results from the test that I had run I have learned that there is not one sorting algorithm that best firsts every problem. Overall, quick sort on average had the most



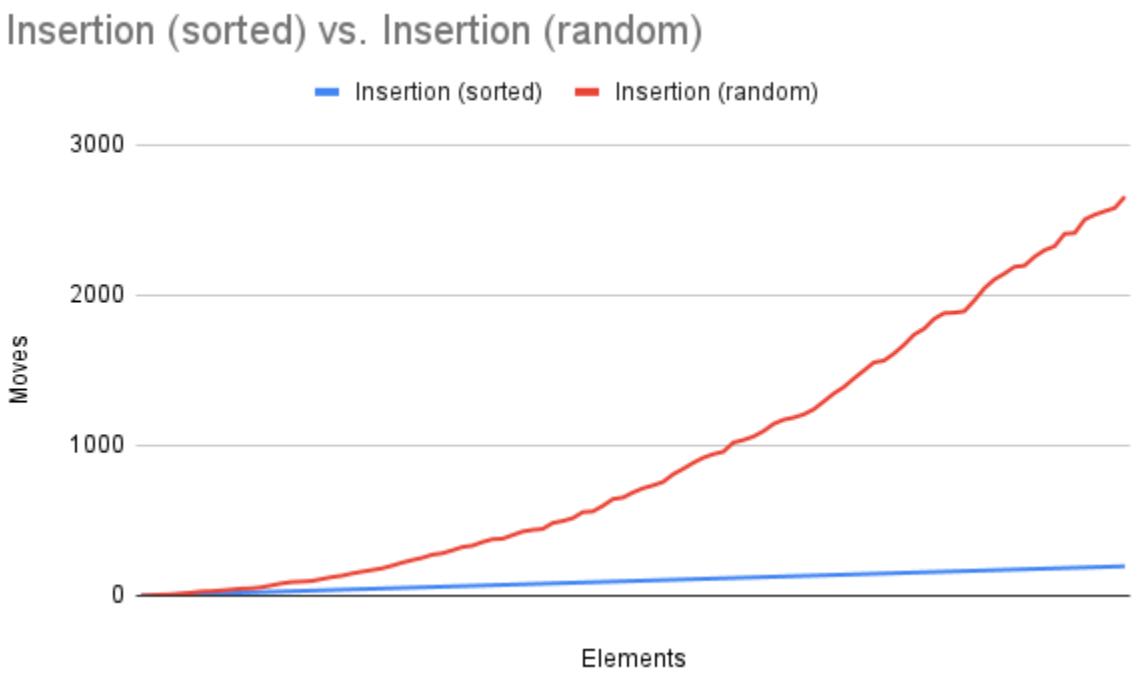
efficient results when given a random unsorted array of numbers. As seen in the graph, quick sort seems to do the best on average when only looking between 1 and 100 elements to be sorted. This may not seem like a huge difference based on this graph, but the difference becomes more prevalent when larger amounts of elements are sorted.



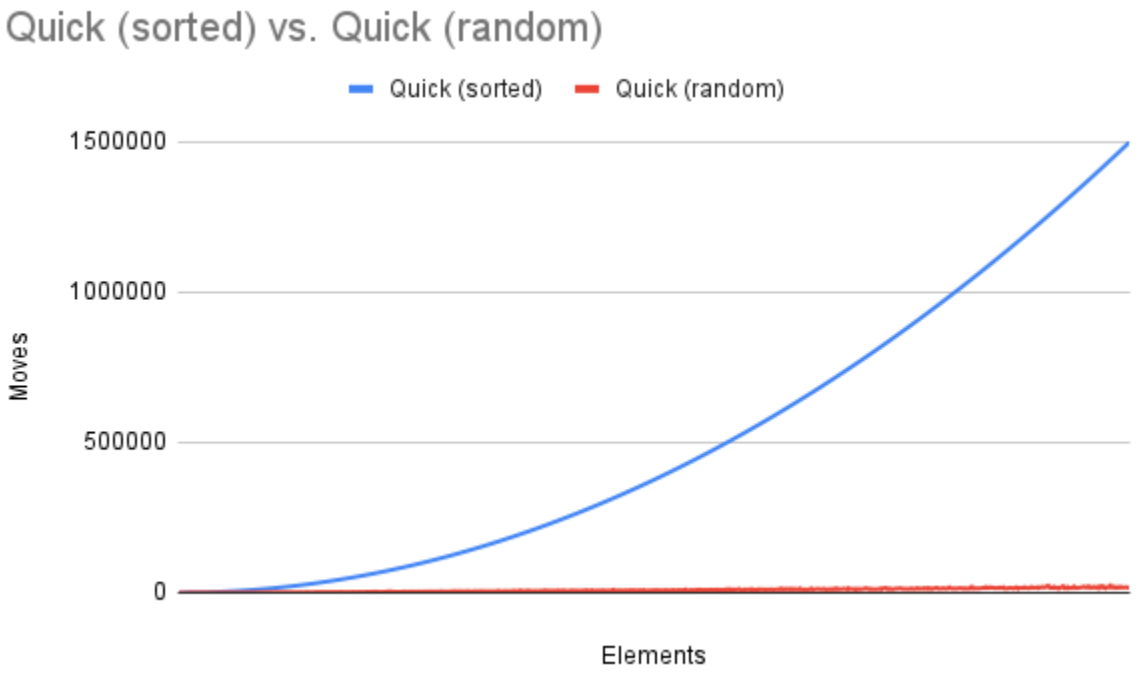
When the tests were given 1,000 to 100,000 elements to sort, it is clear that quick sort holds superiority over both shell and heap sort in this case. While this is true, it is also important to realize that when these sorts are handling such vast amounts of quantities that heap sort actually ends up surpassing shell sort. When only comparing an array of 100 elements it may seem that shell sort is the better sort, but when it comes to larger arrays, heap sort may be the more appropriate choice. It was also intentional that insertion sort was left out of this graph since its results were drastically more inefficient when compared to these three that it made the differences in these appear to be negligible. Though insertion short should not be completely disregarded either. When the sorts were given only 1-10 elements to sort, there are very different results. Looking at the graph, it would appear that quick sort is



no longer the most efficient option for sorting. It is even beaten by shell sort and insertion sort in this case. Given a small array, it seems that insertion sort would actually be the most appropriate option when it comes to cases like these. Furthermore, insertion sort can perform much better given an array that is already sorted.



In the specific case where an array given to insertion sort is already sorted, it handles the sorting process incredibly well. In most average cases insertion sort usually takes $O(n^2)$ to fully sort an array, as presented in the graphs. Though when given its best case it would appear that it now functions at O(n). There are also special cases for quick sort



as well when given a fully sorted array. On average, quick sort is an incredibly efficient array. Though as seen in the graph, the required number of moves that quick sort requires is drastically more inefficient when compared to its average performance. In an average case given an unsorted array, quick sort normally functions at $O(n \log(n))$, making it one of the most efficient options. Though in its worst case, quick sort requires $O(n^2)$ to complete the sort. In terms of the other two sorts, it would seem that heap sort is the most consistent where it does not really have a worst case. Heapsort on average seems to constantly take $O(n \log(n))$ making it a viable general option, especially when dealing with an extremely large amount of elements. When looking at shell sort though, it seems to have a slightly less efficient time complexity taking $O(n (\log(n))^2)$. Shell sort is still practical though when dealing with small to medium sorts since it is essentially an extension of insertion sort that can handle bigger sort tasks a bit better.

Conclusion:
What I can conclude from these tests and the data gathered is that, on average, the most efficient sorting algorithm is quick sort. Even though quick sort tends to be the most efficient, it does not mean that this sort is the best one to use for every problem. As shown in the data from the graphs above, there are different opportunities and cases where it might be more reasonable to implement a different sort. In short, there is not a “one size fits all” when it comes to sorting algorithms.