Assignment 3 - Writeup

Introduction:

For this writeup, I will be focusing on the relative cost between different sorting algorithms, namely, insertion sort, quick sort, heap sort, and batcher sort. While each of these algorithms successfully sorts any given array, some may take longer and use up more resources than others. We will alter the length of each array using -n, and take count of how many comparisons and moves are required for each sort. Ultimately, we will compare the statistics of these different algorithms.

We will create a table of values, with the independent variable being the array size, and the dependent variables being both the move count and comparison count.

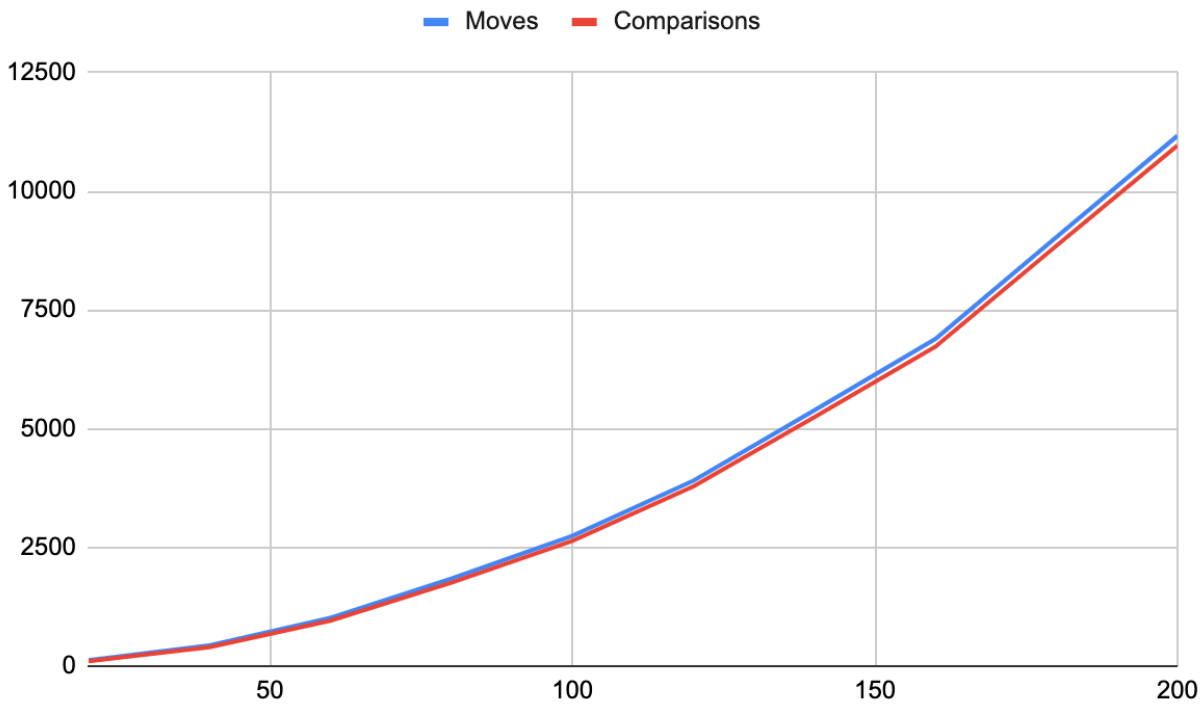
We will be utilizing the default seed of 13371453.

Insert Sort:

Insertions sort is one of the simplest sorting algorithms. The premise of insertion sort is basically to iterate through all of the elements of the array, and nested within that loop iterate through previous elements in order to find its place. This for loop and nested while loop results in time complexity O(n^2).

Record moves and comparison counts for array length from 20 to 200, incrementing by 20.

Array Size	Moves	Comparisons
20	124	102
40	438	396
60	1019	956
80	1842	1759
100	2741	2638
120	3906	3783
140	5390	5246
160	6892	6728



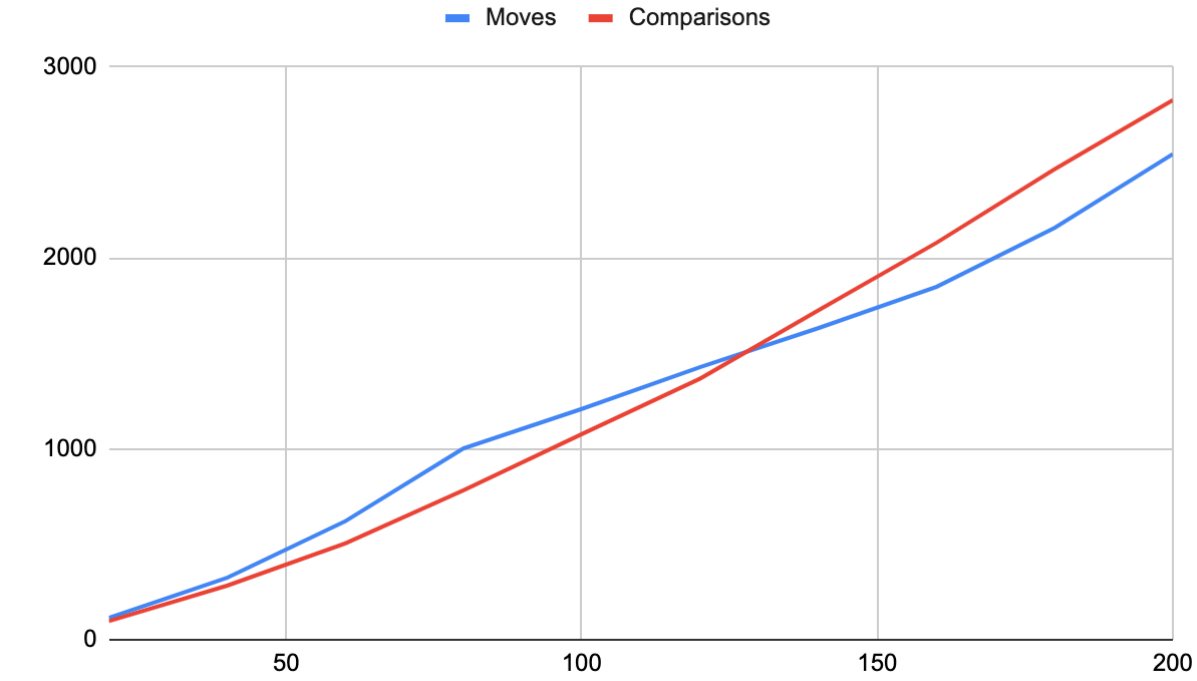
As we can see in this example, the number of moves and comparisons are almost identical. This makes sense because every comparison leads to a move in the while loop. We can also see the O(n^2) nature of this algorithm, as we see the exponential growth in the graph.

Batcher sort:

Batcher sort works by sorting items far from each other at lengths of 2^x, for example, an array with length 16 would be sorted by elements 8 values away from each other. Then it would sort elements 4 values away from each other, and so forth until all elements are sorted. This lends itself to an O($nlog(n)^2$), as the algorithm sorts through every element timed by log_2, as shown by our example.

Record moves and comparison counts for array length from 20 to 200, incrementing by 20.

Array Size	Moves	Comparisons
20	114	97
40	324	283
60	621	505
80	1002	783
100	1209	1077
120	1428	1367
140	1632	1725
160	1848	2079
180	2157	2465
200	2544	2827



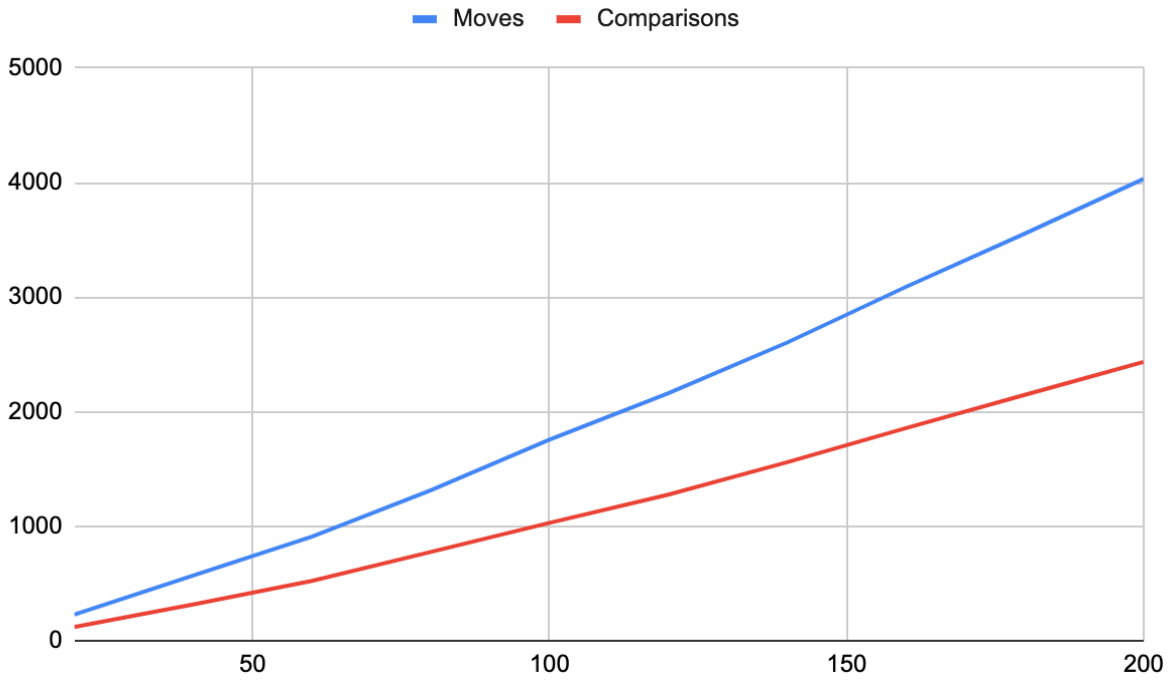
As we can see in this graph, we see what seems like a linear graph. However, this is not the case, the graph follows an O($nlog(n)^2$) pattern. This can explain why the blue line is is not smooth, as it is most obvious at the array size 80 to 160. Interestingly, these two numbers are a multiple of 2 apart, meaning that the log(n) would only increment by 1. Compounding on this $log(n)$ is squared making its impact even larger. This could explain the relative flatness of this line in comparison to the slopes that come before and after this section.

Heap Sort:

Heap sort consists of building a heap in which the parent nodes are larger than their child nodes. After this heap is corrected, we remove the largest value, and then fix the heap, making sure that all parent nodes are larger than their child nodes. This function is time complexity O($nlog(n)$).

Record moves and comparison counts for array length from 20 to 200, incrementing by 20.

Array Size	Moves	Comparisons
20	228	120
40	570	317
60	909	522
80	1314	775
100	1755	1029
120	2160	1275
140	2604	1559
160	3090	1856
180	3555	2146
200	4032	2434



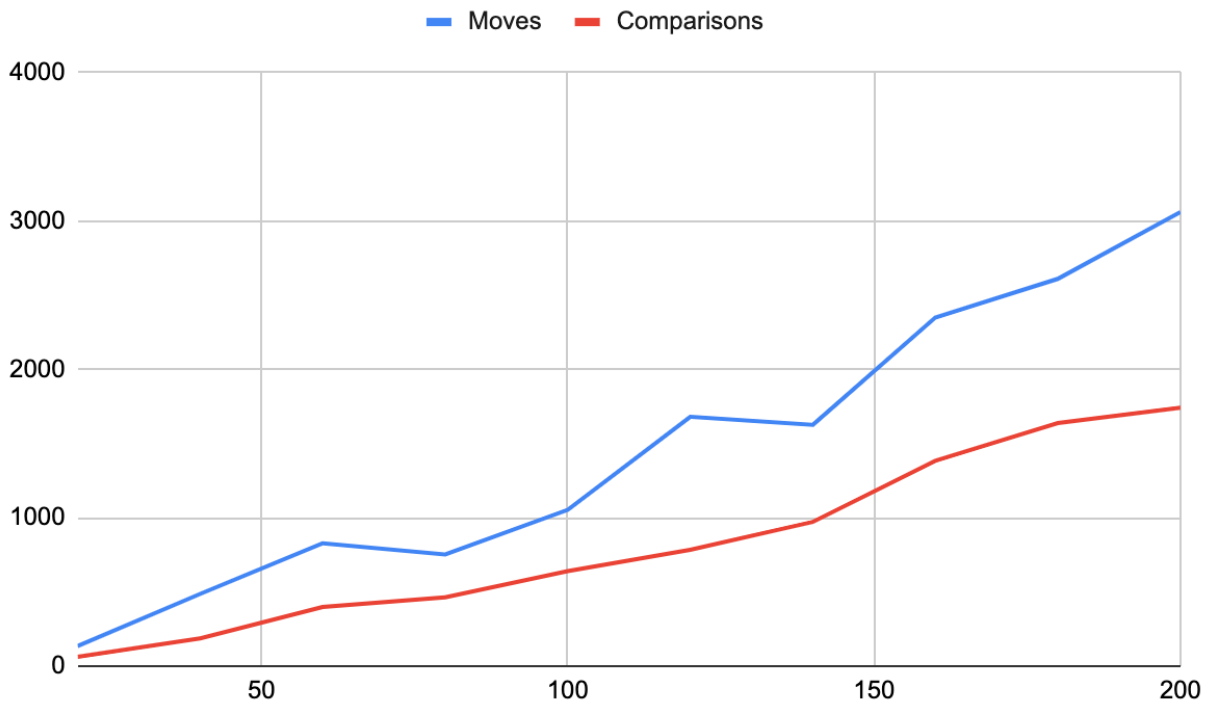
Here, again we see an almost linear line, just like we saw in batcher sort. Here we see a smoothness to the line, which makes sense since the creation of the heaps is quite predictable, as we just find father nodes and children nodes that are smaller. Additionally, we just swap the greatest value to the back one at a time, which can also contribute to the lack of bumps. This algorithm seems to be quite predictable in the number of moves and comparisons it will take because of this quality.

Quicksort:

Quicksort creates pivot points and looks for values smaller than this pivot point and values larger than the pivot point. Quicksort then sorts smaller and smaller segments by creating more pivot points. This algorithm also happens to be O($nlog(n)$), however, has much more variance, as the pivot points could be very lucky and cut down on the number of moves, or be very unlucky and create a need for more moves.

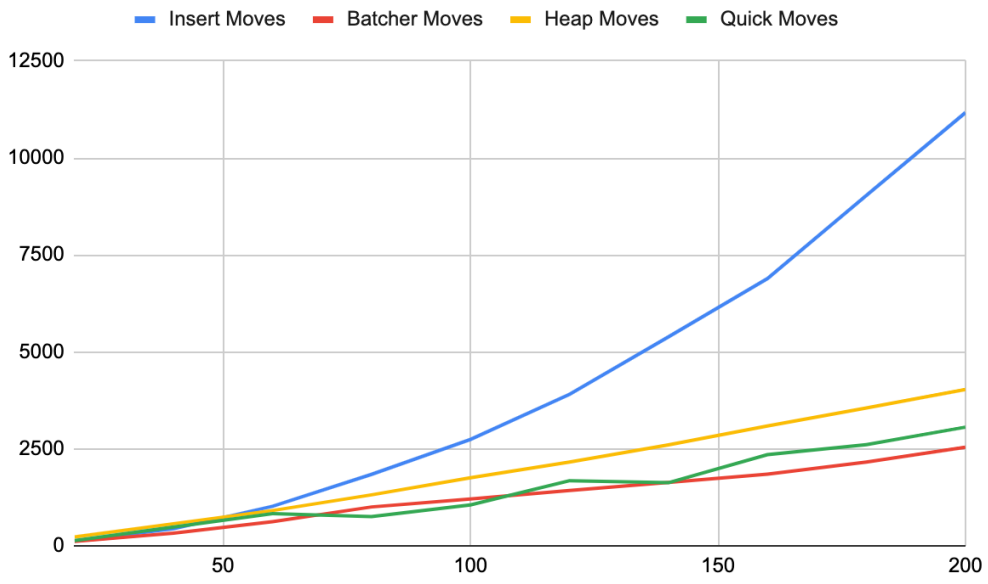
Record moves and comparison counts for array length from 20 to 200, incrementing by 20.

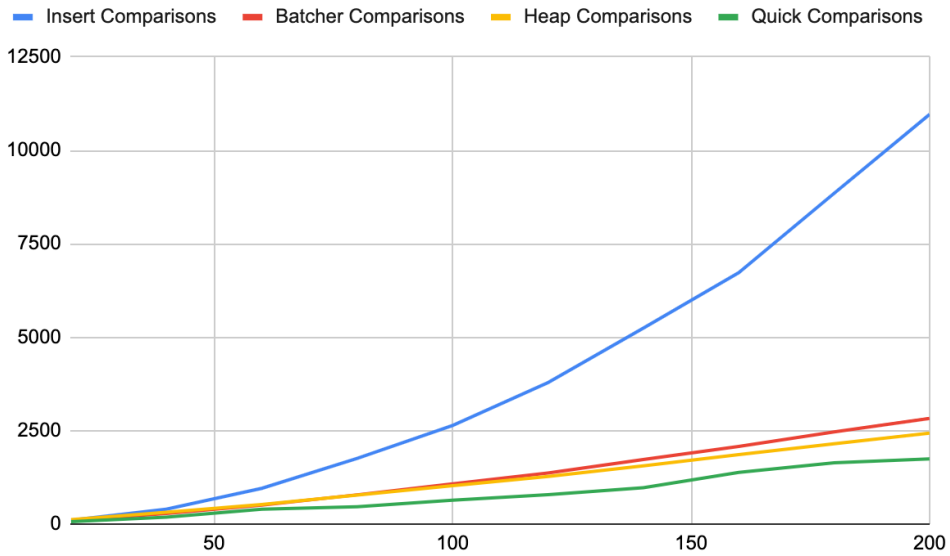
Array Size	Moves	Comparisons
20	135	63
40	486	186
60	828	398
80	753	463
100	1053	640
120	1680	784
140	1626	972
160	2349	1383
180	2610	1638
200	3060	1741



Here, we can see our theory come to fruition. We see the most non-linear graph, and we can reason that it is because of the inherent luck that comes with creating pivot points. This makes me think that quicksort could have the best time complexity in some cases while having much worse time complexities for other cases. Quicksort seems to have the most variance.

All Sorting Algorithms Moves and Comparisons:





Here, we see what we have been reasoning by looking at each graph and set of values individually. We see that insertion sort has the highest time complexity by far, of O(n^2), while the other algorithms have time complexity of some sort of O($nlog(n)$).

Conclusion:

in conclusion, I saw first-hand the importance of a good algorithm. While insertion sort works as accurately as any other sorting algorithm, it is by far the most expensive having the largest time complexity. I saw just how quickly O(n^2) diverges from the O($nlog(n)$) time complexity. Even though I only went up to 200 length arrays, I can only imagine the differences between insertion sort and the other algorithms to compound even more.