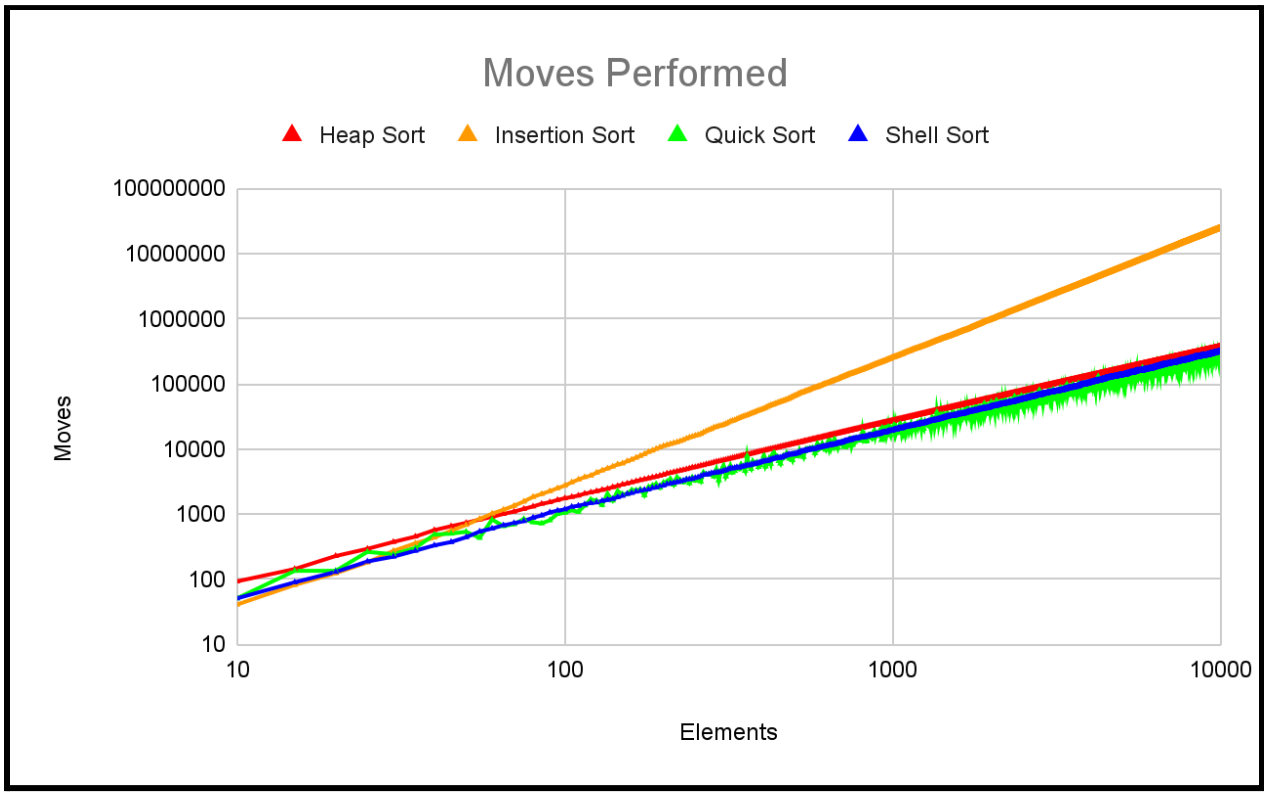
Assignment 3 WRITEUP



From this graph and the assignment data, I learnt how increasing the number of elements would affect would be as the number of elements increases for each sort.

Heap Sort: In this graph, increasing the number of elements increased the number of moves required to sort the array. It took approximately 100 moves to sort 10 elements, slightly more than a 1000 moves to sort 100 elements, around 50,000 moves to sort 1,000 elements, and around 500,000 moves to sort 10,000 elements. This algorithm's complexity is O(n log n), and is faster than the insertion sort algorithm. Although it is less beneficial for an array with less than 75 elements, it is much more efficient for

arrays with a larger number of elements. I learnt that the left child of an element would be 2 * index, and the right child would be (2 * index) + 1.

Insertion Sort: In this graph, increasing the number of elements significantly increases the required number of moves to sort the array. To sort 10 elements, it would take 80 moves, but sorting 100 elements would take around 2700 moves. As the number of elements increased, the number of required moves to sort the array would substantially increase. Because of this, it took more than a million iterations to sort an array of 10,000 elements.

I learnt that this sort compares the nth element with the preceding elements that exist in the array. It also compares consecutive elements, and moves them down if the higher-indexed element is less than the lower-indexed element. Insertion sort's time-complexity is O(n^2). From it's complexity, I was able to see the correlation because it took over a million moves to sort 10,000 elements.

Quick Sort: In this graph, increasing the number of elements increased the required number of moves to sort the array. Its complexity is also O(n log n), and using this algorithm for any number of elements would usually require slightly less moves than the heap sort algorithm. However, the number of elements required to sort the algorithm would fluctuate because this algorithm uses partitions and recursive calls to sort the array. To sort 10 elements, it would take approximately 80 moves; to sort 100 elements, it would take approximately 1000 moves; to sort

1000 elements, it would take more than 10,000 moves, and it would take around 500,000 moves to sort 10,000 elements.

Shell Sort: In this graph, increasing the number of elements increased the number of moves required to sort the array. Its algorithm complexity is O(n log n), which is the same as that of heap and quick sort. Insertion sort's complexity is different at O(n^2). It takes approximately the same number of moves as quicksort and heapsort to sort arrays, for any given number of elements because they all have the same time complexity. The shell sort's logic is similar to the insertion sort, except that it sorts numbers that are a gap apart from each other. The gap value starts at log(3 + 2 * n) / log(3), and decrements by one after each full array pass. This allows for shell sort to sort faster than the insertion sort algorithm.

From this assignment, I learnt that insertion sort is more efficient for sorting arrays with a smaller number of efficients, but not as efficient as the other sorts for an array with a large number of elements. I learnt that shell sort, heap sort, and quicksort sort elements at a similar rate because they have the same time complexity. I also learnt about the different algorithms each sort has. I experimented with the sorts by creating a test array which consisted of 30-bit pseudorandom numbers, and calculated the number of moves and comparisons each sort took. After performing calculations with different numbers of elements, I saw that the number of moves required to perform an insertion sort significantly increased, but that for heapsort, quicksort, and shell sort increased at a much slower rate. I also saw that the number

of moves required to perform heapsort, quicksort, and shell sort increased at a relatively similar rate.