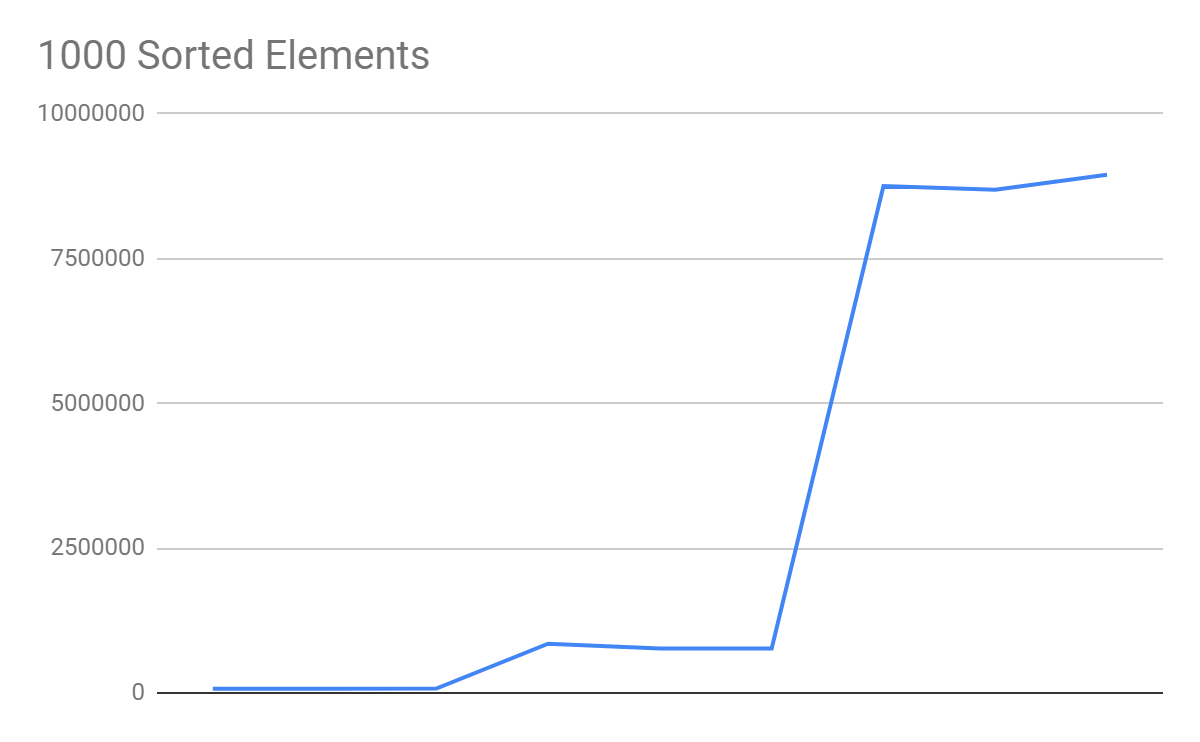
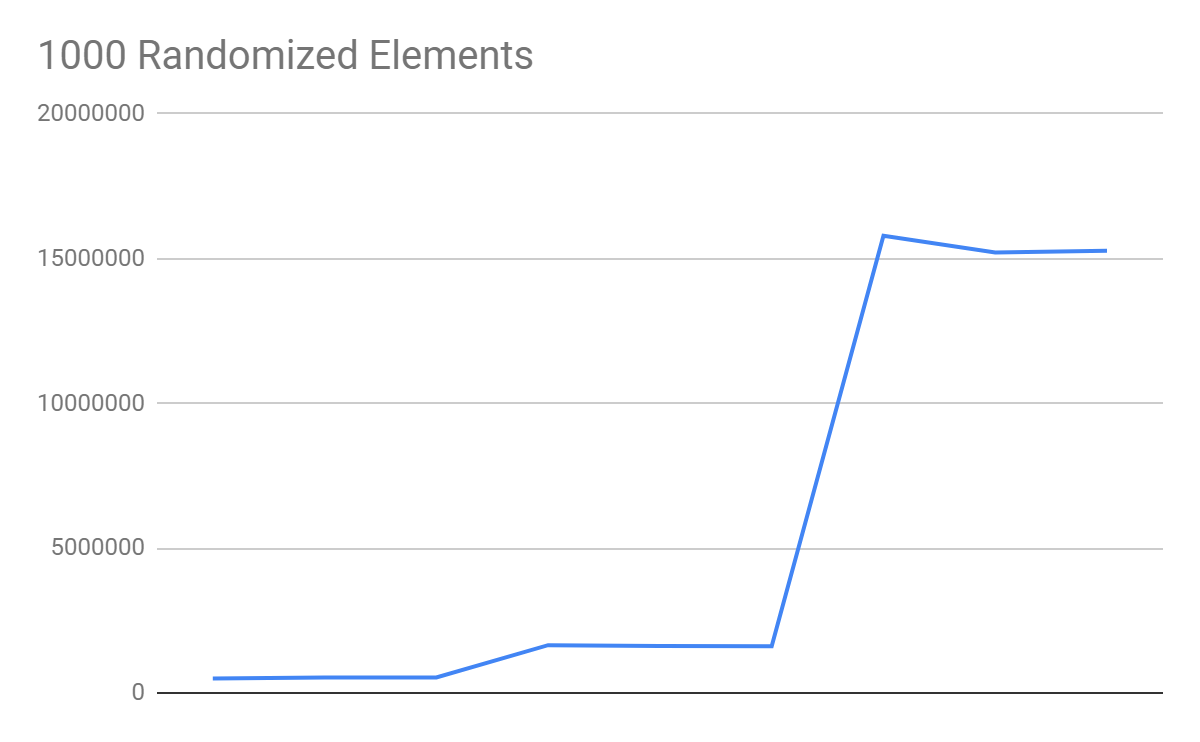
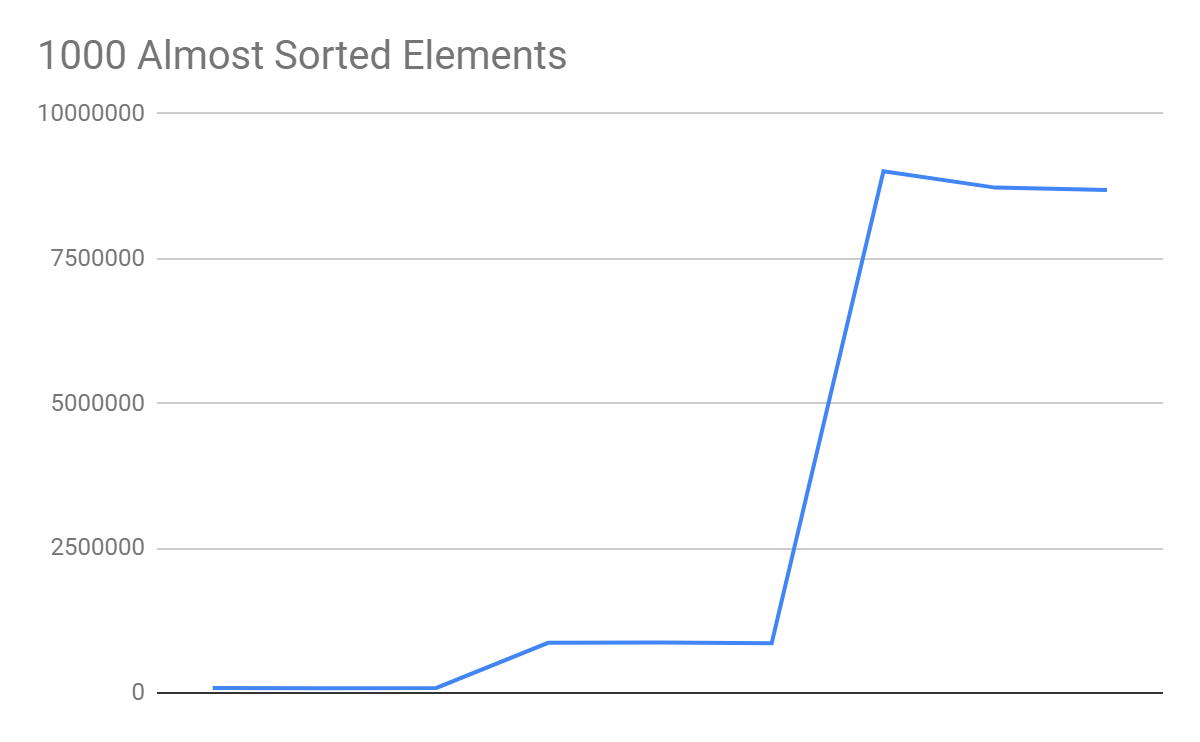
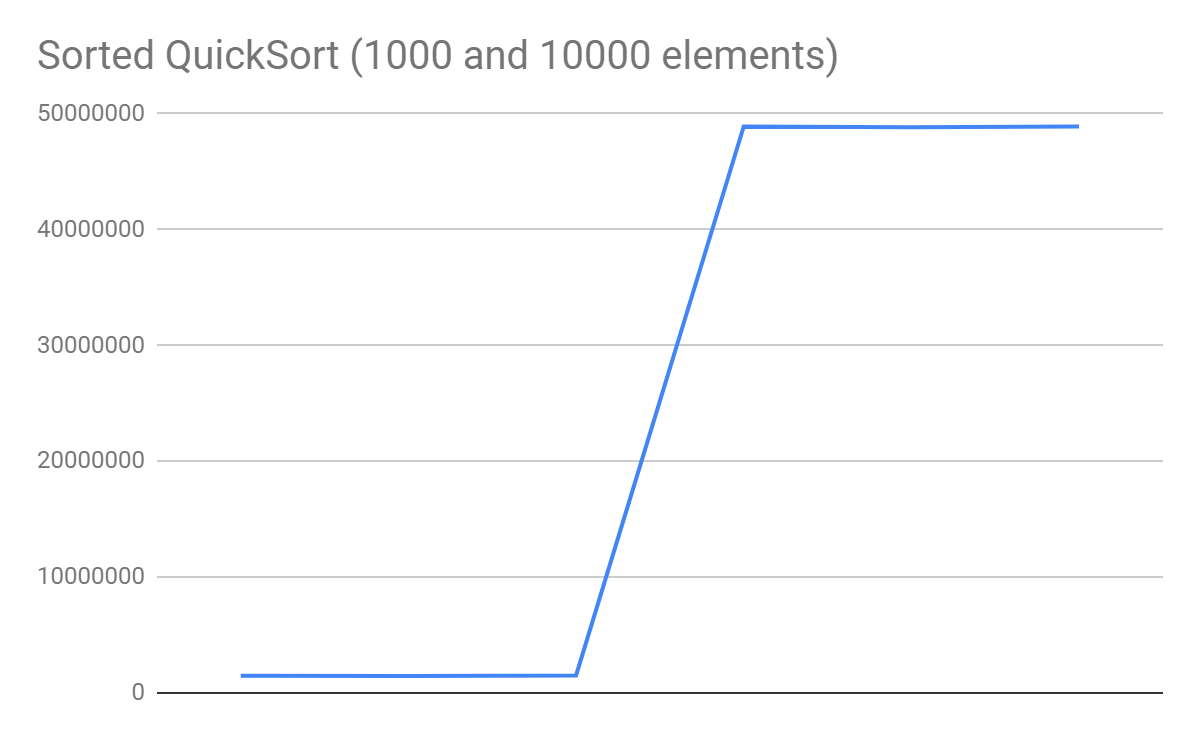
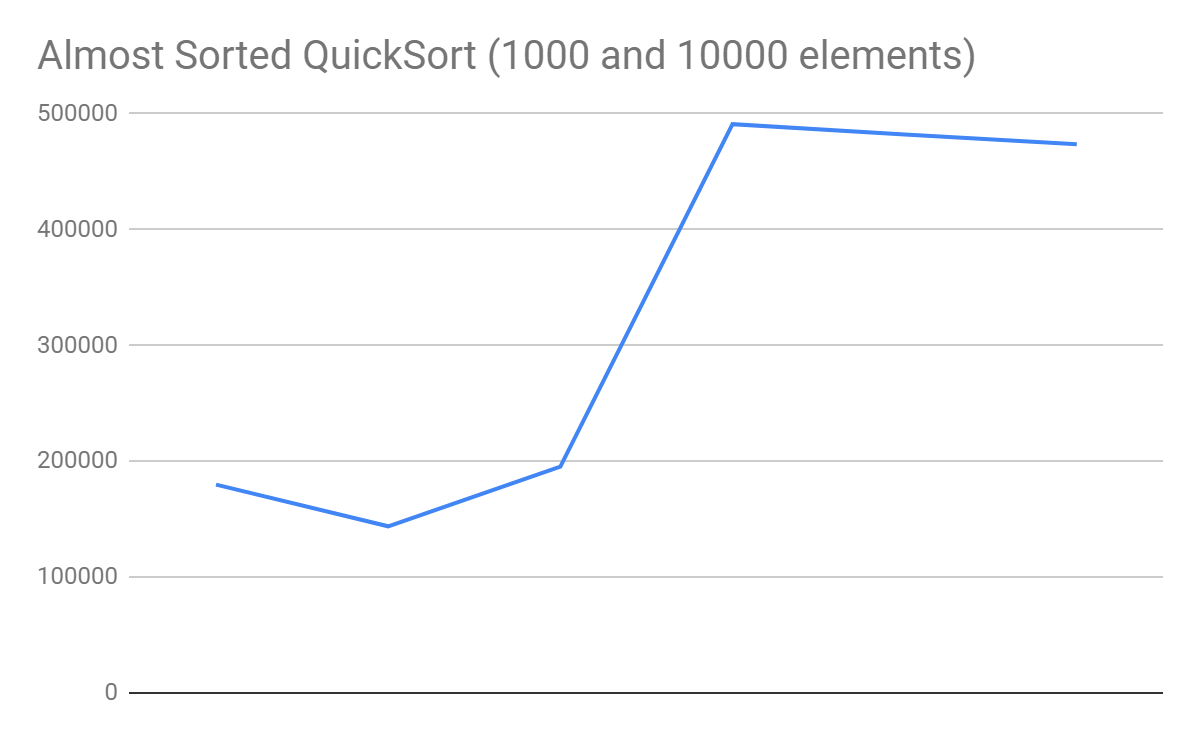
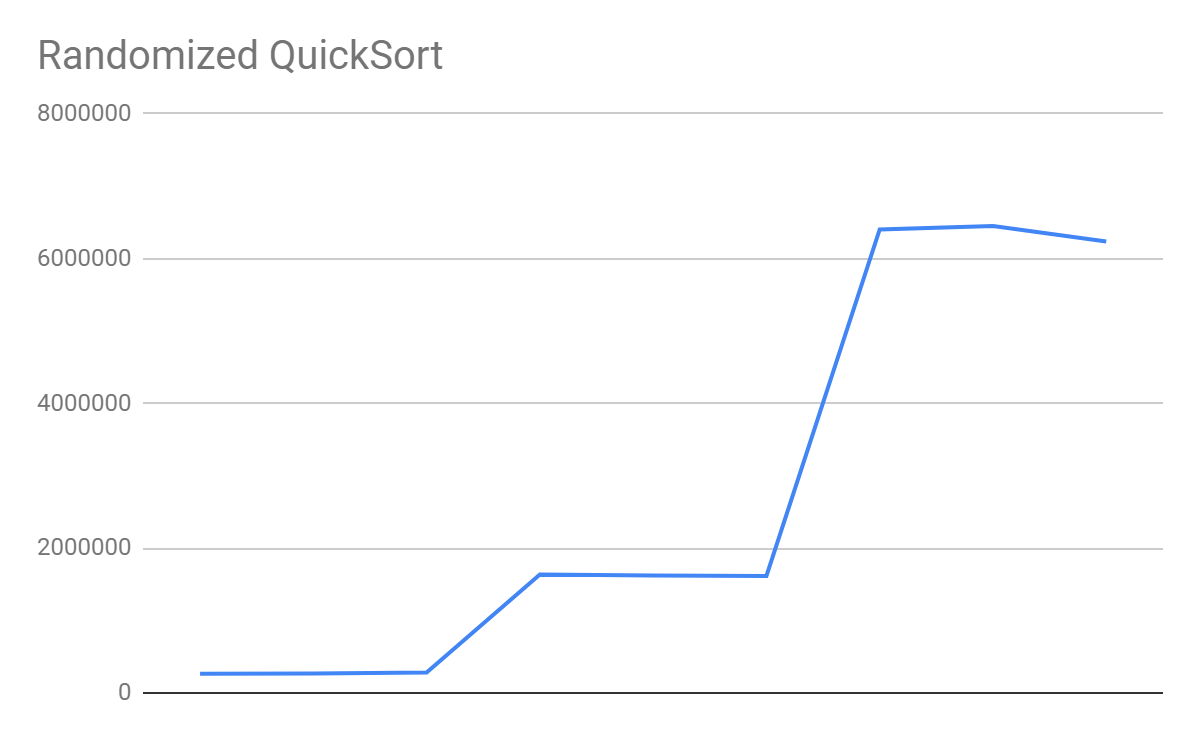
**ANALYSIS OF EXPERIMENTS** **SAMED GANIBEGOVIC**

The theoretical efficiency of a merge sort for the all cases is Θ(n log(n)). In this experimentation of implementing the merge sort with various inputs, all sizes of elements provide similar results. In my experimentation, I ran each algorithm three times. From observation, the merge sort implementation provided an experimentation yield of Θ(n log(n)). All three array types had the same yield. 

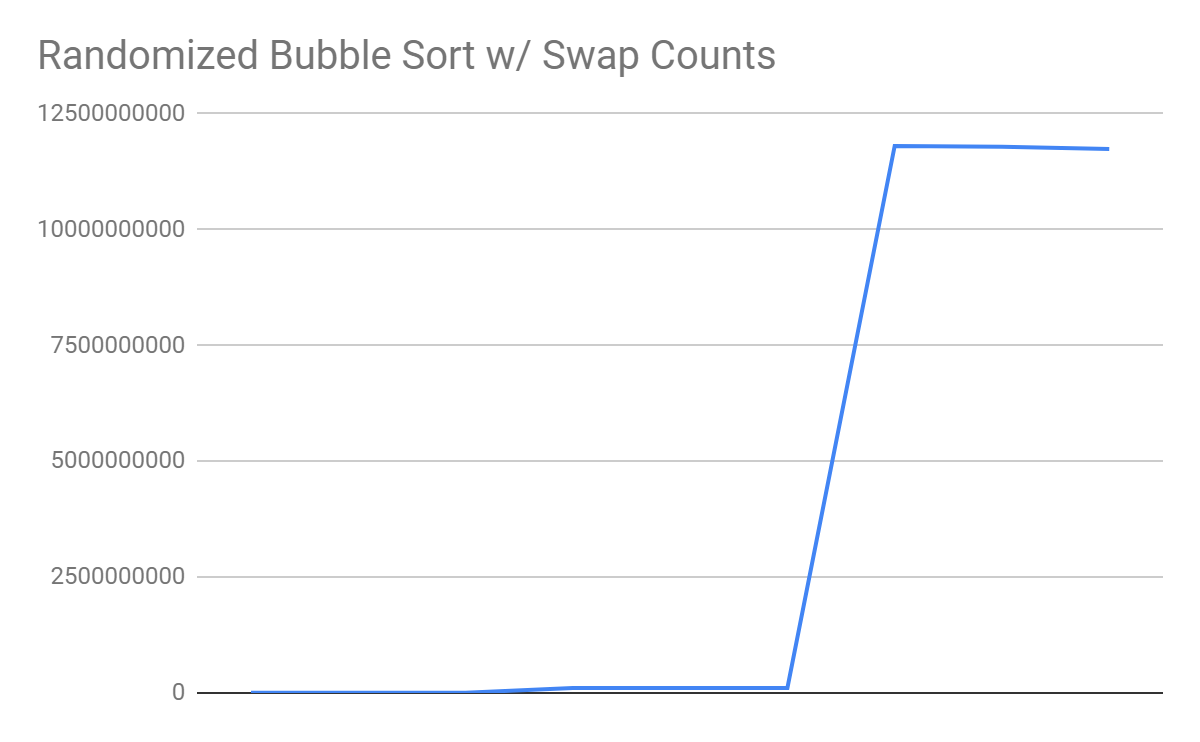
The theoretical efficiency of a quick sort for a best case and average case is Θ(n log(n)), and worst case is Θ(n2). It seems that the randomized quick sort for the three input types yields Θ(n log(n)), similar to the merge sort. Quick sort for a sorted array input and almost sorted array provides an experimental growth of Θ(n2).

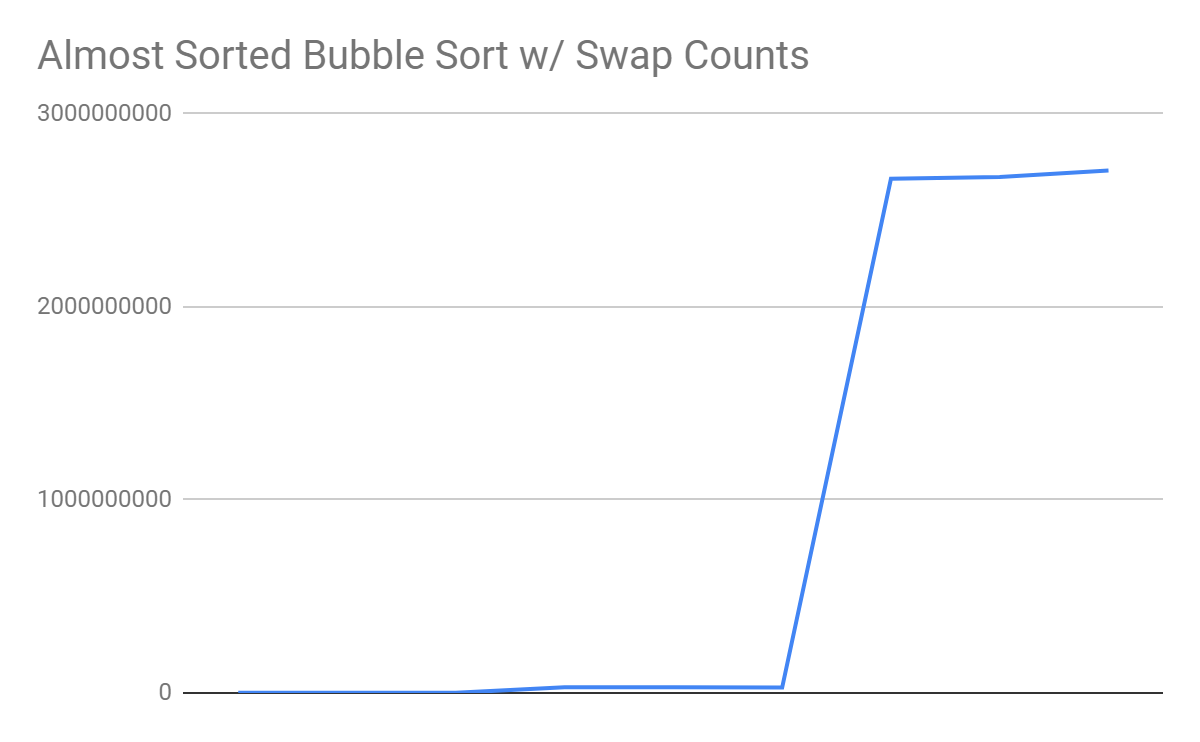
The quick sort for a sorted array and almost sorted array of one hundred thousand elements provides a StackOverflow error. The StackOverflow error occurs since a very bad pivot causes the array to split to just one number. This means the remaining numbers are the other side of the partition. When the function is recursively called again, only one integer from the sub array is removed and the remaining numbers still have to be partitioned. The recursive calls get too deep and the memory runs out, resulting in a stack overflow error.

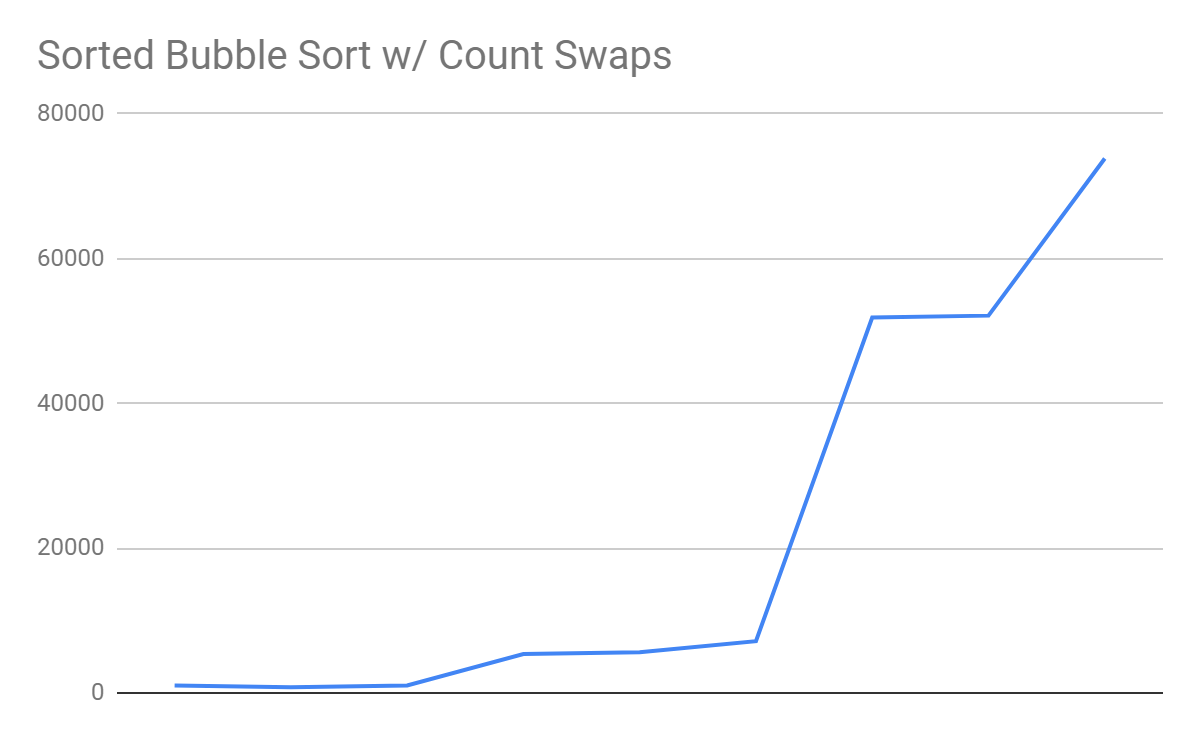


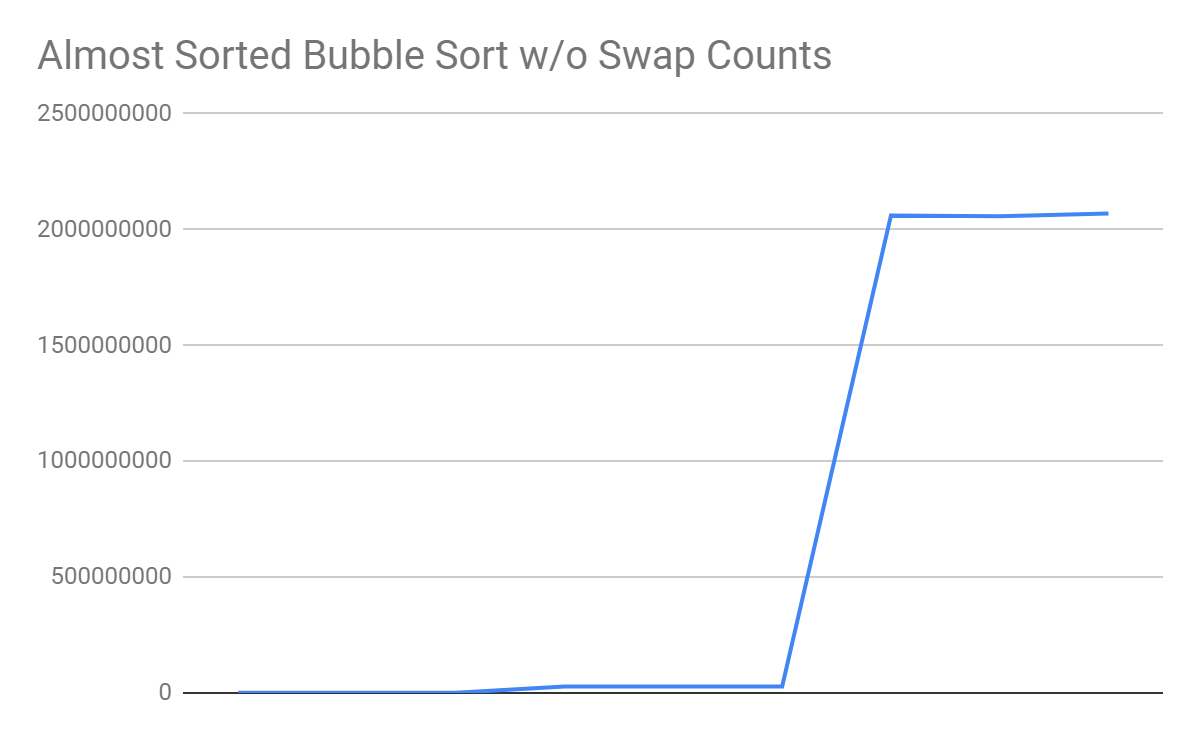
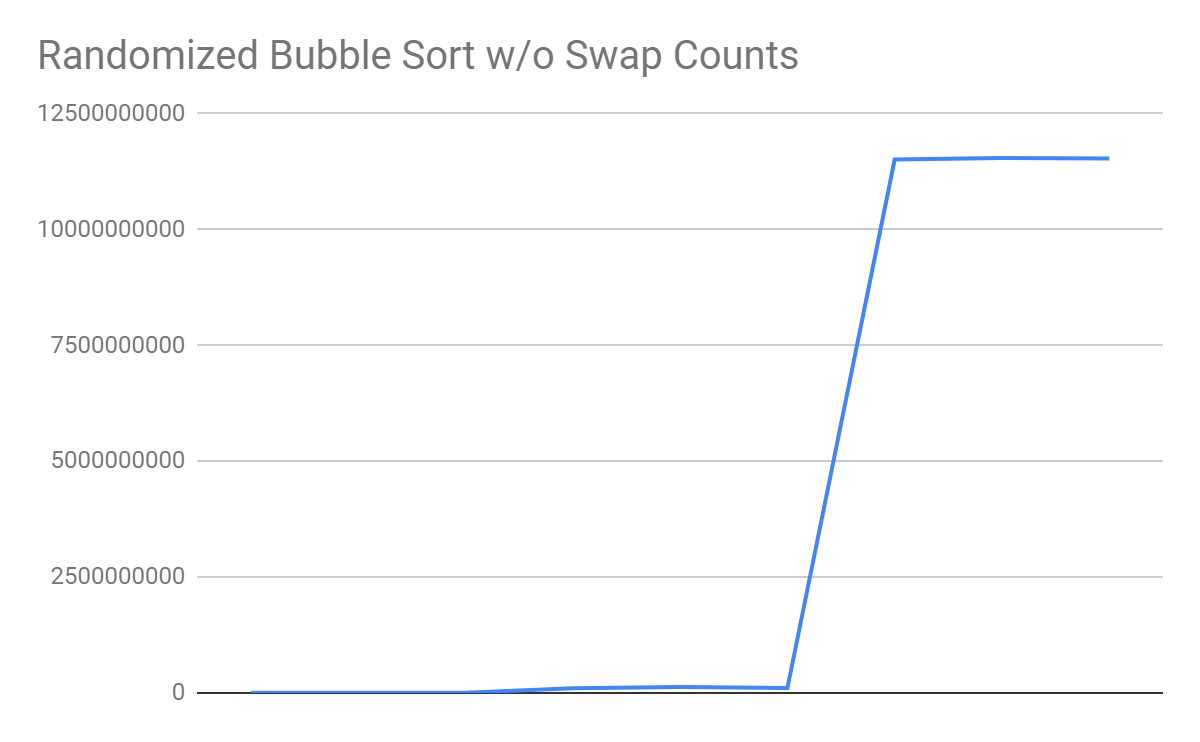
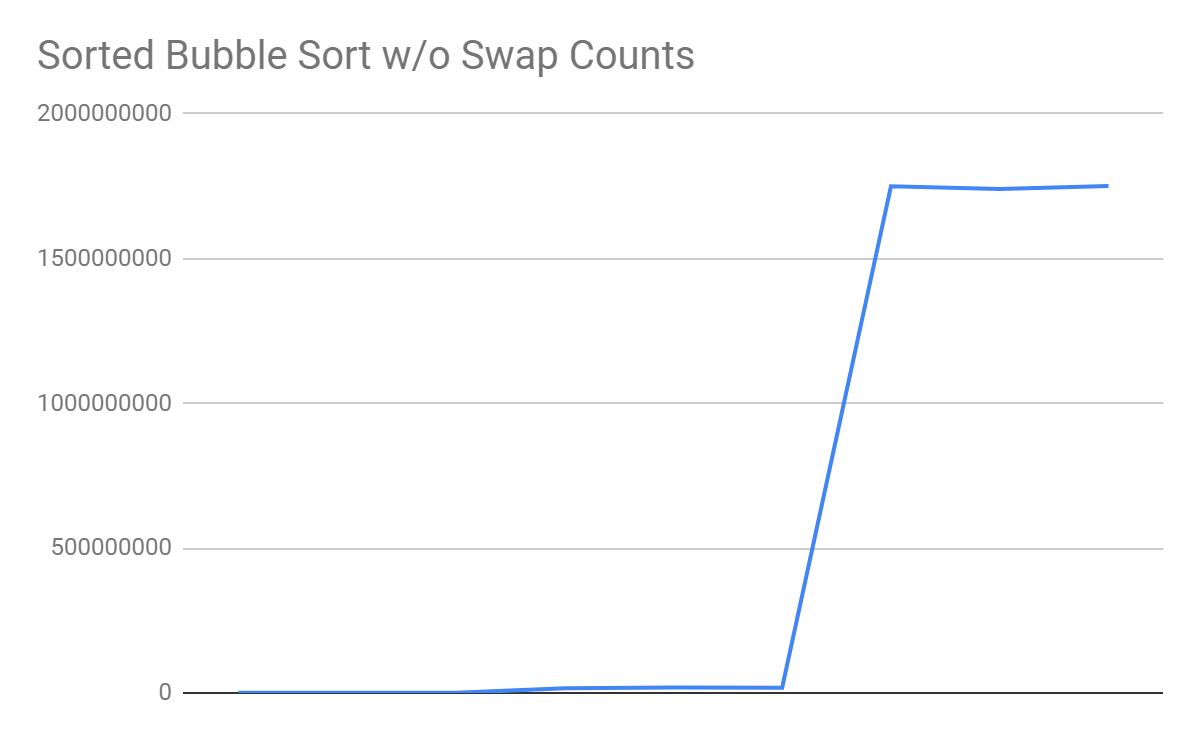
The theoretical efficiency of a bubble sort with swap counting for a best case is Θ(n), and for the average and worst case is Θ(n2). The sorted array list with a bubble sort with swap counting provides an experimental yield of Θ(n).

I believe the graph doesn’t accurately represent this because the difference between the input values is extremely large, and if the graph displayed the run time at each thousand, then the graph would represent the linear growth that the algorithm would actually have. However, the randomized bubble sort and almost sorted arrays provide an experimentational yield of Θ(n2).



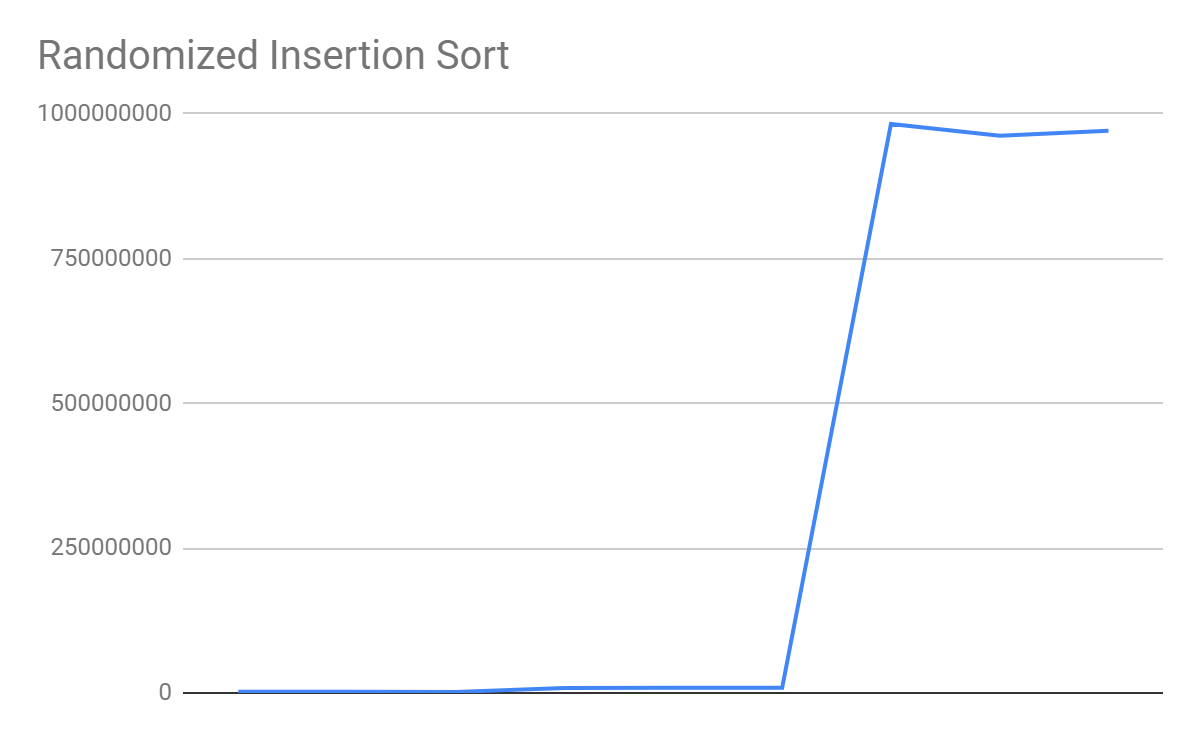


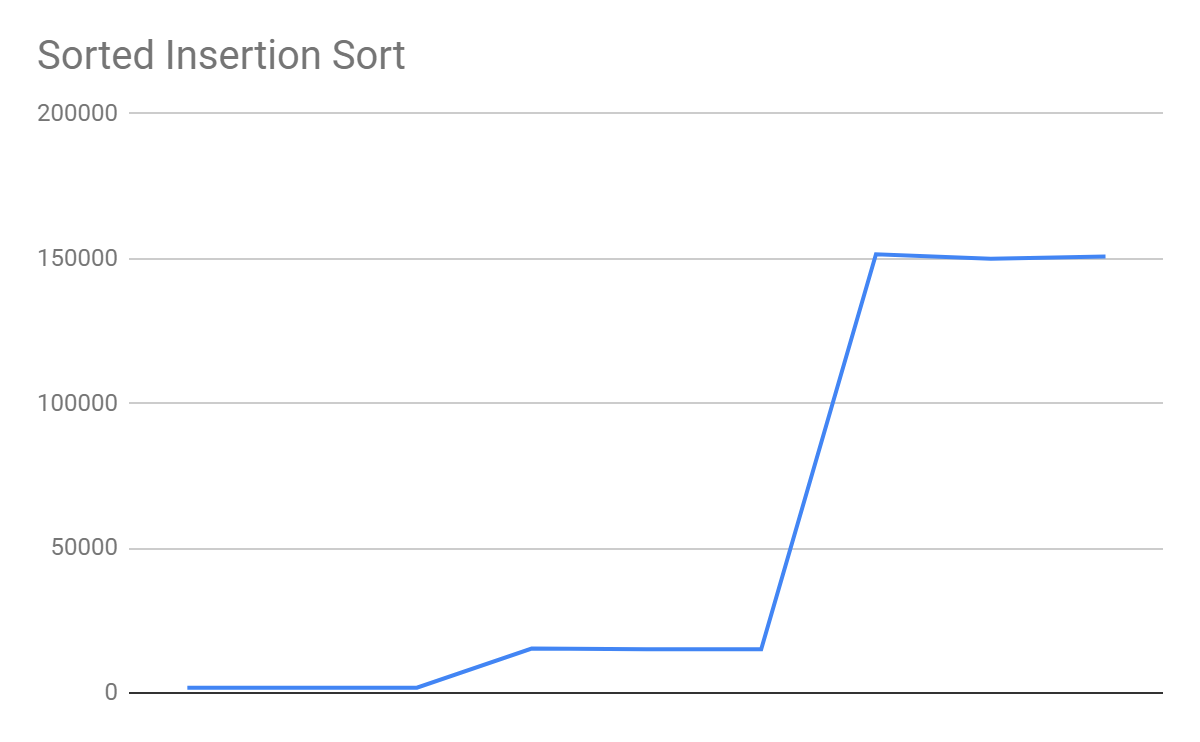
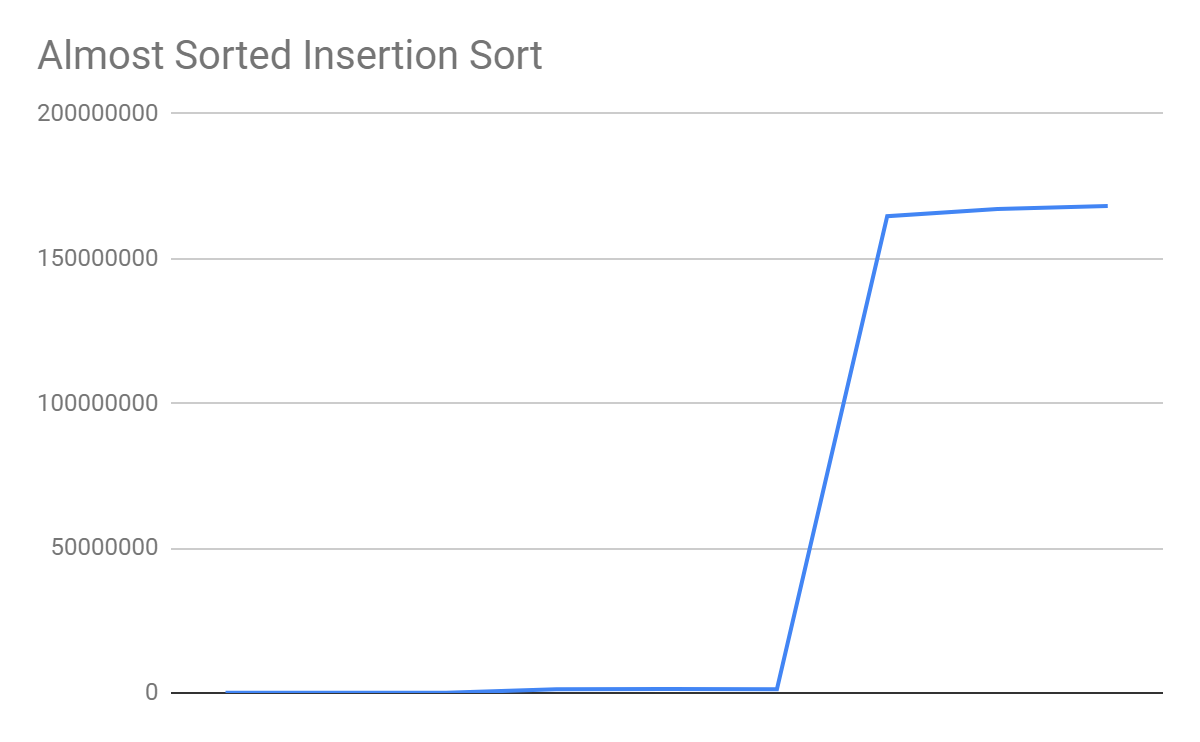


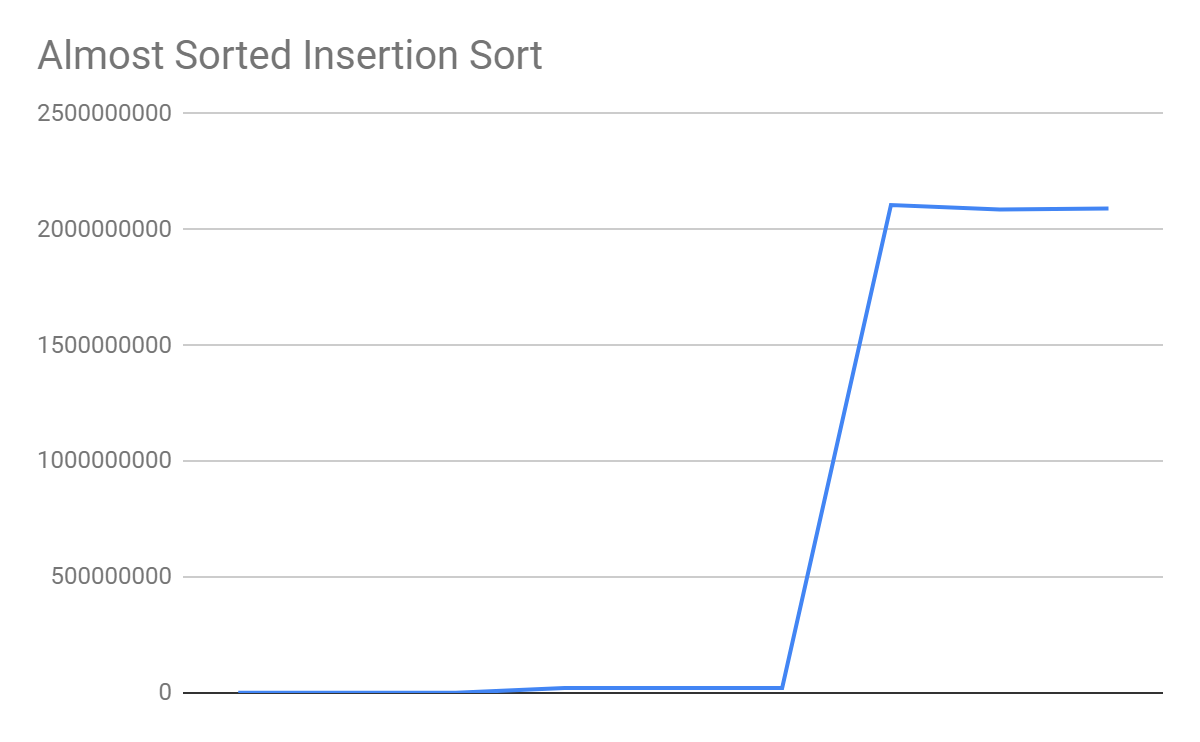
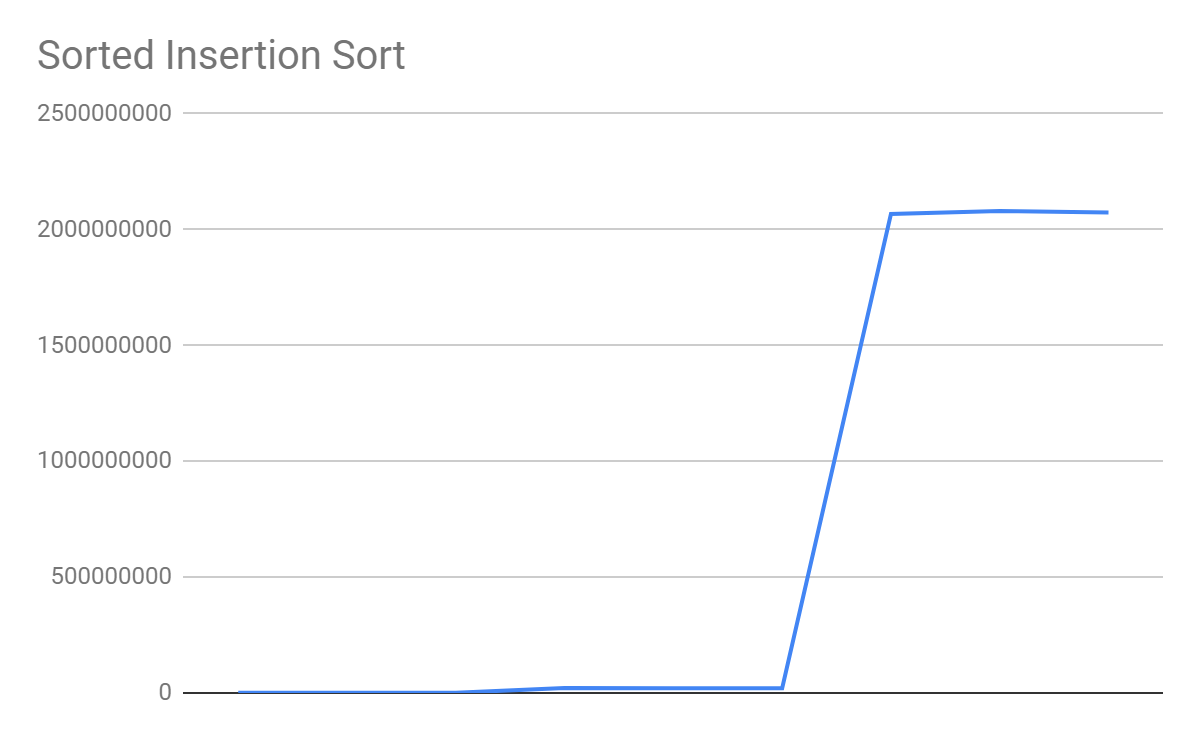
 The theoretical efficiency of a bubble sort without swap counting provides a theoretical order of growth of Θ(n2) for all cases. In all three input types for the array, all of them yielded an experimental growth of Θ(n2). This is because no matter what the input is, the algorithm checks each element in the array, whereas, with swaps counting, the algorithm stops when no swap occurs. In addition, the best case for the bubble sort with swap counts is Θ(n) for an already sorted array.

The theoretical efficiency of an insertion sort for a best case is Θ(n) and the average and worst case is Θ(n2). In this implementation of insertion sort, the sorted and almost sorted provide an experimentational growth of Θ(n).

However, as previously mentioned in the bubble sort with swap counting analysis, I don’t believe the graphs accurately represent this because of the large difference in inputs. The randomized inputs has a growth of Θ(n2).





 The theoretical efficiency of a selection sort for all cases is Θ(n2). The experimentation data observed for all three input type arrays and sizes yielded very similar results. This is due to the fact that each element has to run through both of the loops in the algorithm. The experimentation growth was the same as the theoretical growth of Θ(n2).

From the experimentational data provided in the tables in the Google Sheets document, the best algorithm for randomized data for all three input sizes was the quick sort. The algorithm is able to split the arrays into equal sizes, whereas, in the sorted array the sub array consists of one element and causes a Stack Overflow when the input is one hundred thousand integers. In addition, the almost sorted array of one hundred thousand integers causes a Stack Overflow.

The best algorithm for already sorted data regardless of the input size is the bubble sort with swaps counting. The algorithm terminates once a swap does not occur. Therefore, making it the most efficient algorithm for an already sorted array with a theoretical growth of Θ(n) for sorted data. The runner-up for this algorithm was the insertion sort which also has an theoretical growth of Θ(n). The worst algorithm for sorted data was the quick sort.

From the experimentation data, there seems to be no best algorithm for all input sizes. For an input size of one thousand, the insertion sort is the best algorithm. For an input size of ten thousand, the quick sort algorithm was the fastest, and for an input size of one hundred thousand, the merge sort was significantly faster than the rest of the algorithms.

The insertion sort algorithm is best for smaller sets of data that is already sorted or almost sorted. For this type of data the algorithm has a growth of Θ(n). The quick sort algorithm was the fastest for the input size of ten thousand. This is because an almost sorted algorithm is considered an average case for a quick sort algorithm - which has a theoretical growth of Θ(n log n).

Finally, merge sort is the best for the input size of one hundred thousand because regardless of the size of the array, the merge function splits the array into equal halves and requires linear time to merge them. Merge sort for all cases has a growth of Θ(nlogn).