Michael Patel
CSC 505
Program 1 Report

Introduction

This report outlines the results of several experiments with insertion sort, merge sort, heapsort and the Python sorting utility. Runtime and the number of comparisons were analyzed with regards to different types of inputs (random, sorted and reverse sorted).

Experiment Environment Setup

Processor type: Intel Core i5, 4 cores
Processor speed: 2.9 GHz
Cache size: 256 KB L2 cache, 6 MB L3 cache
Memory size: 8 GB 1600 MHz DDR3
OS: macOS Sierra 10.12.6
Language: Python 3.5.2

Choice of inputs: Random, Presorted, Reverse-sorted
Input sizes: 10000, 20000, 40000, 80000, 160000, 320000, 640000, 1280000 (for heap, merge and Python utility sort)
Input sizes: 200, 400, 800, 1600, 3200, 6400, 12800, 25600 (for insertion and merge sort)

Experiment Results and Analysis

1. Growth rate in number of comparisons

Figure 1 shows this worst case analysis for heapsort, merge sort and the Python sorting utility. The y-axis represented a ratio between the actual number of comparisons and the predicted/theoretical number of comparisons. The heapsort ratio was greater than one, meaning that my implementation was a poor performer against the theoretical heapsort algorithm. I was very surprised by this result as I expected the heapsort to be the closest to the Python utility's ratio. The Python sorting utility's ratio was very low and performed better than both my heapsort and merge sort. Python's sorting utility is based on Timsort. Perhaps predictably, the optimized Python sorting algorithm had a ratio near 0.1. This means that the actual number of comparisons was only about 10% of the worst case theoretical. Since none of the trendlines hovered near 1.0, the actual growth rate in number of comparisons was not significantly accurate to the predicted number of comparisons, which I found to surprising to learn.

Michael Patel
CSC 505
Program 1 Report

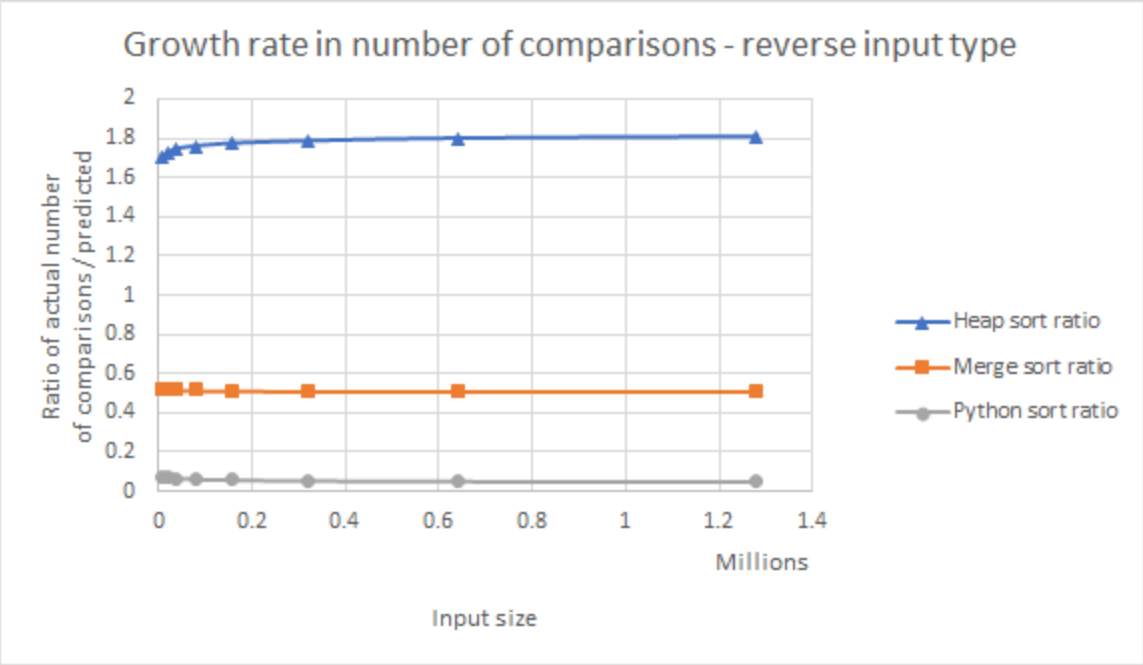

Figure 1: Growth rate in number of comparisons (heap, merge and Python sort)

2. Growth rate in runtime

To analyze the growth rate in runtime for heap, merge and the Python sorting utility, runtime was divided by nlgn and plotted against the input size n. Figure 2 shows these three algorithms on a random input type. The heap and Python implementations seemed to perform similarly (with similar constants), while merge sort had a significantly higher growth rate. This is quite unpredicted as all three sorting algorithms have big-Theta nlgn. I am surprised by such a dramatic difference. It seemed that merge sort did not behave with nlgn growth. This analysis is further supported with different input types (see Figures 15 and 16 in Appendix A).

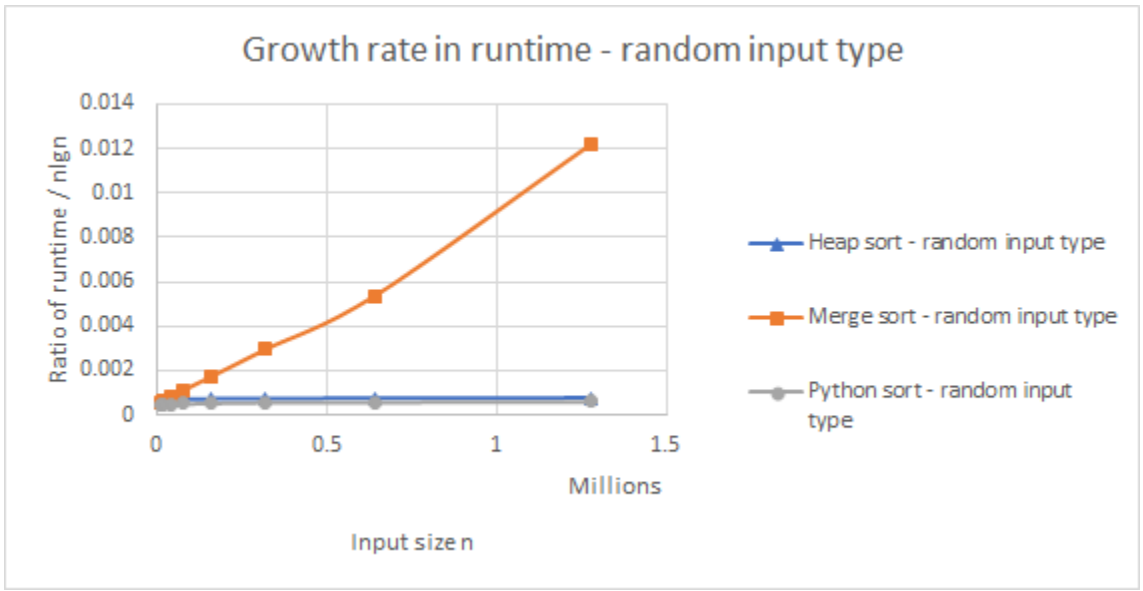

Figure 2: Growth rate in runtime (random - heap, merge and Python sort)

Runtime growth rate was further analyzed in Figures 3 and 4 which compared insertion and merge sort. Here, predictably, merge sort had a growth rate in line with nlgn and constant factor of about 0.002, and insertion sort's growth continued to far exceed merge sort. During experimentation, I was unable to find a set of input sizes that showed the intersection between insertion sort and merge sort. Many times, the merge sort runtime would be zero on small input sizes, making it virtually impossible to detect a crossing point. I imagine that crossing point between insertion and merge sort would have to be on a very small input size. Very interestingly, when the input was pre-sorted, insertion sort outperformed merge sort on every input size. The runtimes were about seven times faster for insertion sort than merge sort on these pre-sorted runs. Insertion sort in its best case is a more efficient implementation, so I would suggest that when a list is almost sorted or nearing completion, then the system should change from using a merge sort to an insertion sort. (See Figure 17 in Appendix A for reverse input).

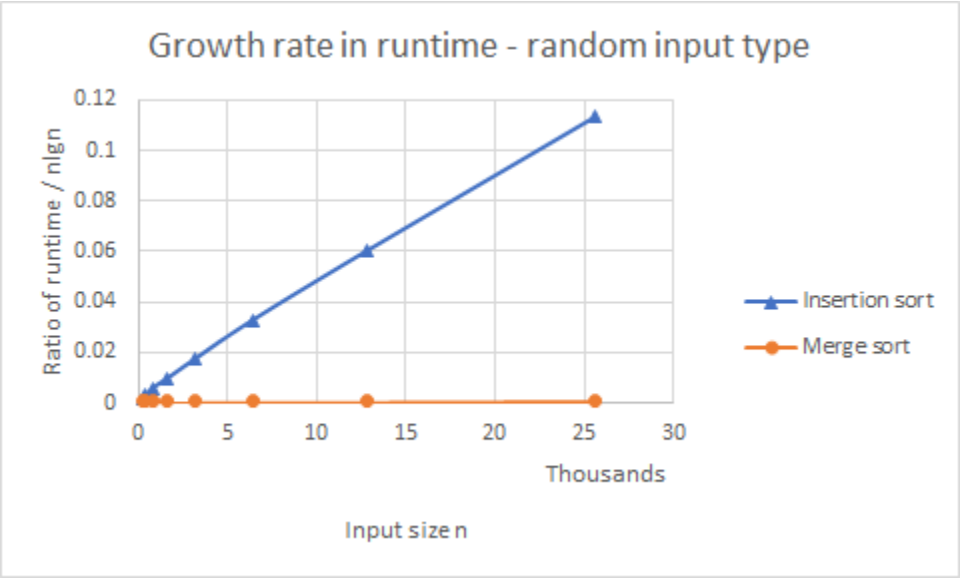

Figure 3: Growth rate in runtime (random - insertion and merge sort)

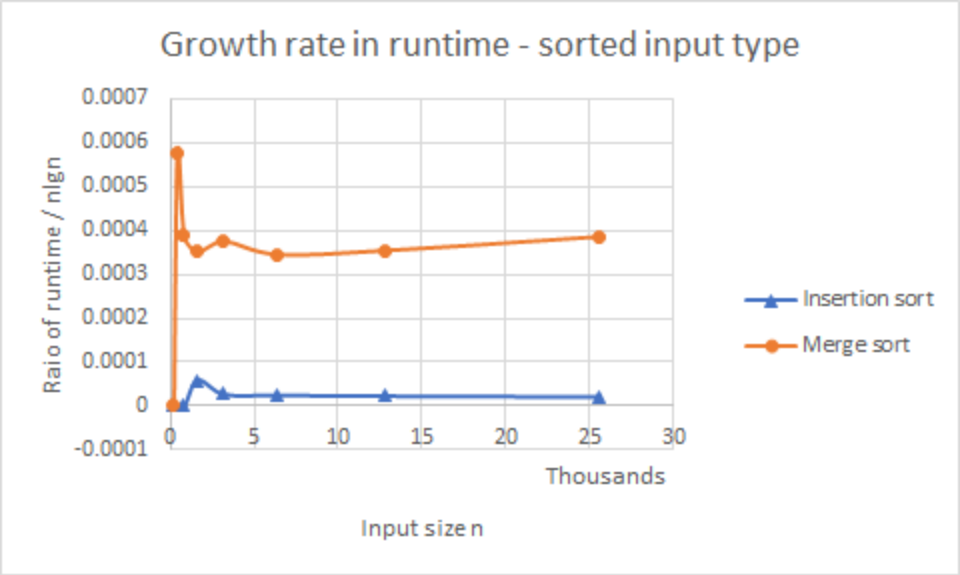

Figure 4: Growth rate in runtime (sorted - insertion and merge sort)

3. The relationship between runtime and number of comparisons

The Python sorting utility (based on Timsort) far exceeded my implementations for heap sort and merge sort with respect to the runtimes versus the number of comparisons. In Figure 5, it is shown that the Python runtime was orders of magnitude faster than merge sort, but had similar trending to heap sort. Meanwhile, the Python number of comparisons was similar to merge sort, but significantly lower than heap sort. It would seem that the Python sorting utility maximizes low runtime and low number of comparisons. The Python sort must use adapting algorithms that hybridize merge and insertion sort, so that on small inputs or nearly-sorted lists, the insertion sort strengths are utilized, while the merge sort strengths are highlighted in larger input sizes. (see Figures 18 and 19 in Appendix A for reverse and sorted inputs)

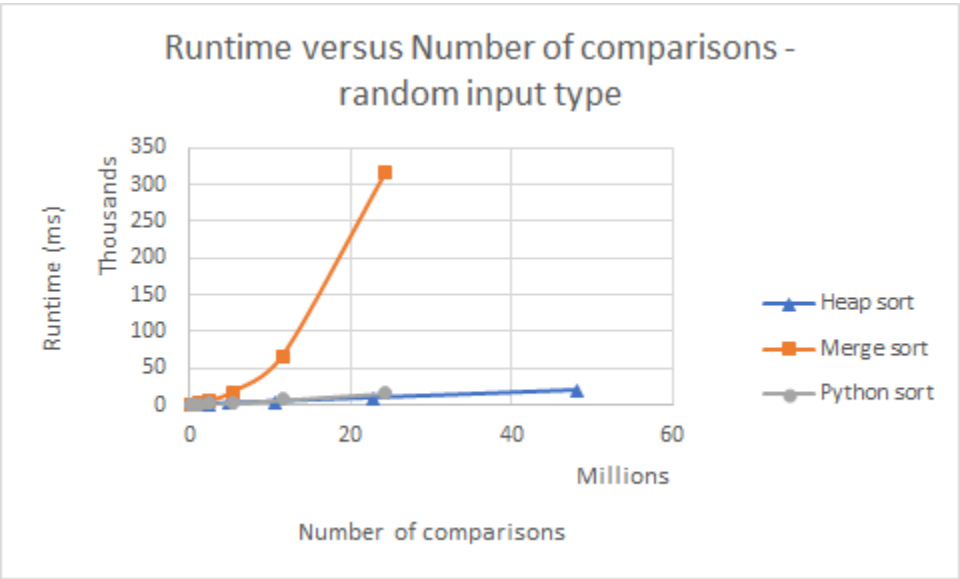

Figure 5: Runtime vs. Number of comparisons (random input)

4. Impact of different types of inputs on runtime and number of comparisons

The experiments that I learned the most from were on the runtimes compared to the input types and the comparison counts against the input types for each of the four sorting implementations. For heap sort (Figure 7), no matter the input type (random, sorted, reverse-sorted), the nlgn runtimes were all similar near a constant 0.0007. The number of comparisons were also very tight no matter the input type (see Figure 11). I learned that heap sort is fairly consistent.

Meanwhile, the Python utility performed better when the inputs were either sorted or reverse-sorted (see Figures 8 and 12). I would speculate that the Python sorting utility has some clever tricks for portions of the input list that are ordered or near-ordered.

The merge sort (Figures 9 and 13) had similar behavior to the Python sort with random input pushing longer runtimes and more comparisons. I now understand why merge sort has been adapted by so many other sorting algorithms.

Lastly, and the most intriguing, insertion sort (Figures 10 and 14) had a fantastic runtime for a presorted input, mediocre for random input and generally poor for reverse-sorted input. Insertion sort also had runtimes and comparison growths that grew n^2 against the input sizes n which supported my intuitions. I learned that a robust algorithm (assuming random input and that the Python utility is not available) should use heap sort or merge sort initially, and once the list is near sorting completion, the algorithm should adapt and use insertion sort. I would not have learned this without this project assignment.

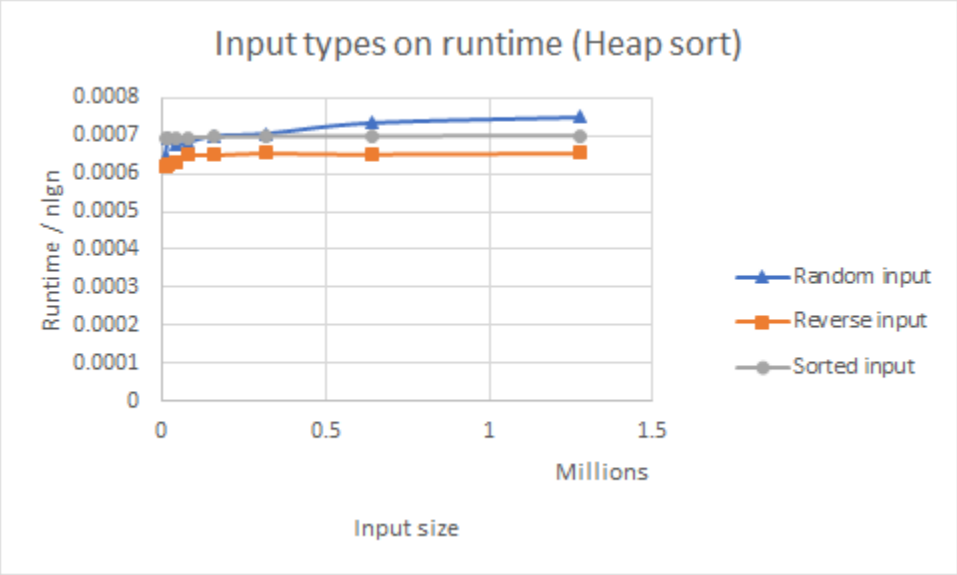

Figure 7: Input types against runtime (heap sort)

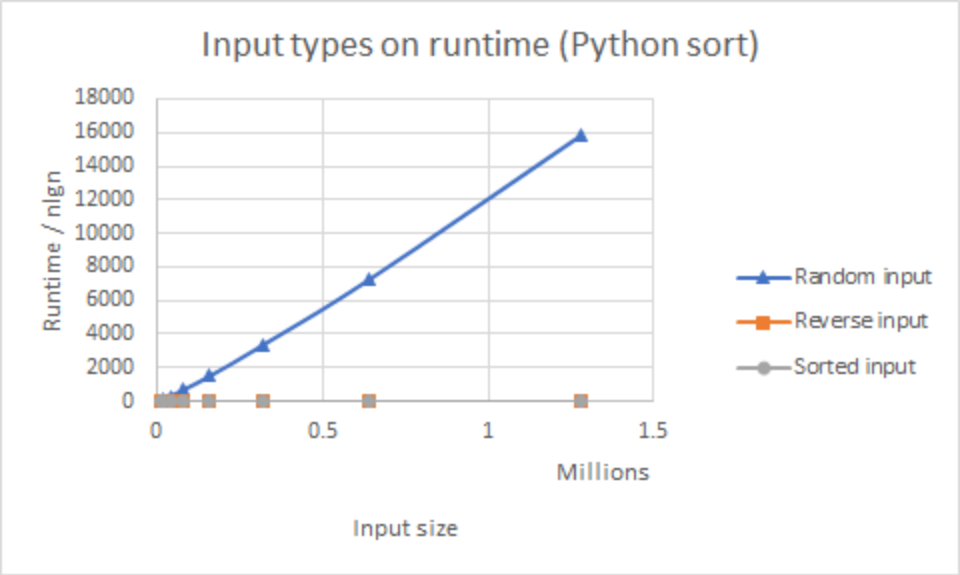

Figure 8: Input types against runtime (Python sort)

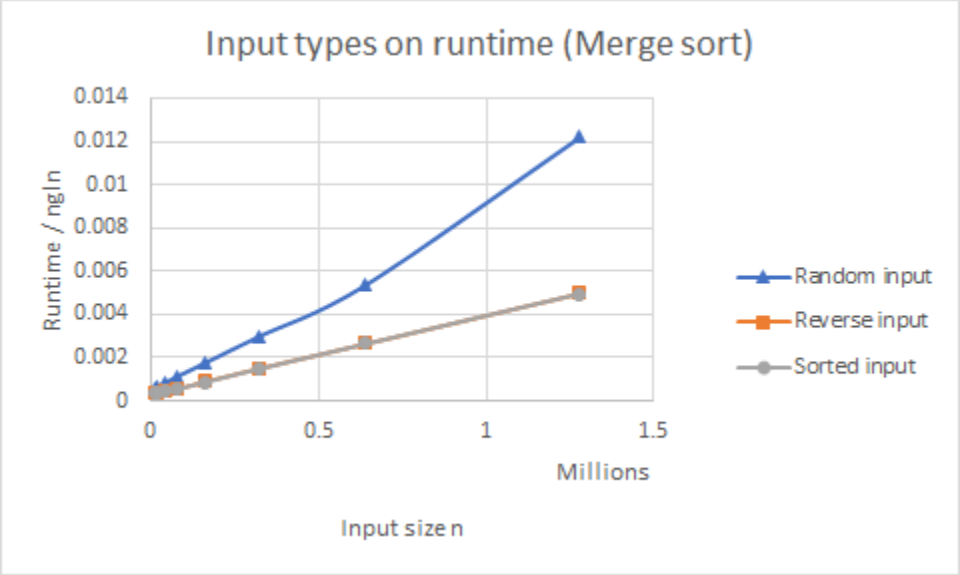

Figure 9: Input types against runtime (merge sort)

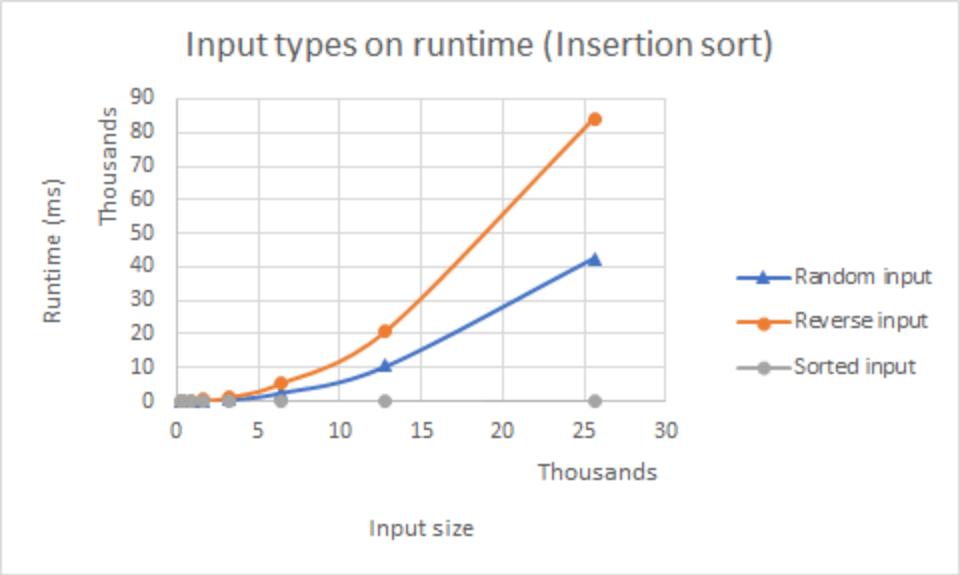

Figure 10: Input types against runtime (insertion sort)

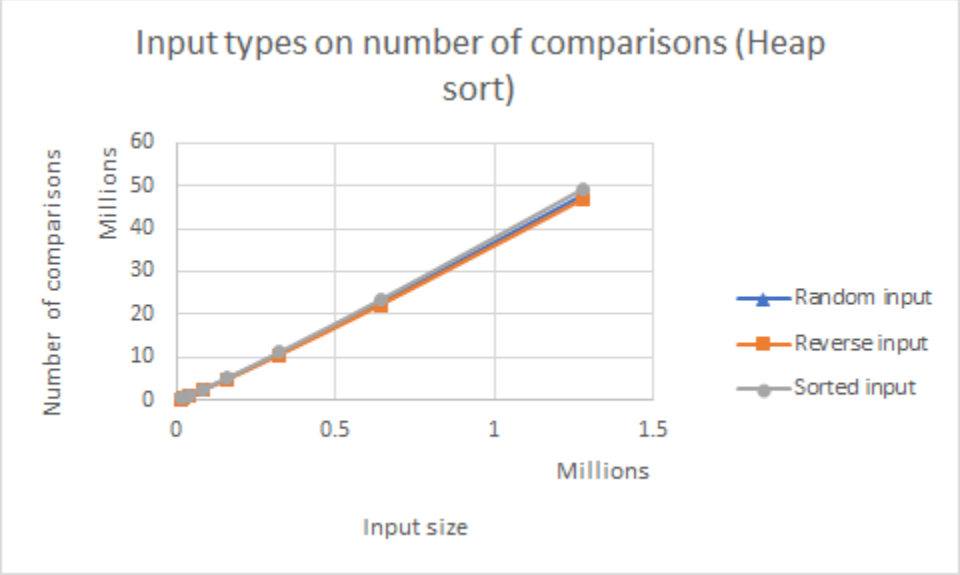

Figure 11: Input types against number of comparisons (heap sort)

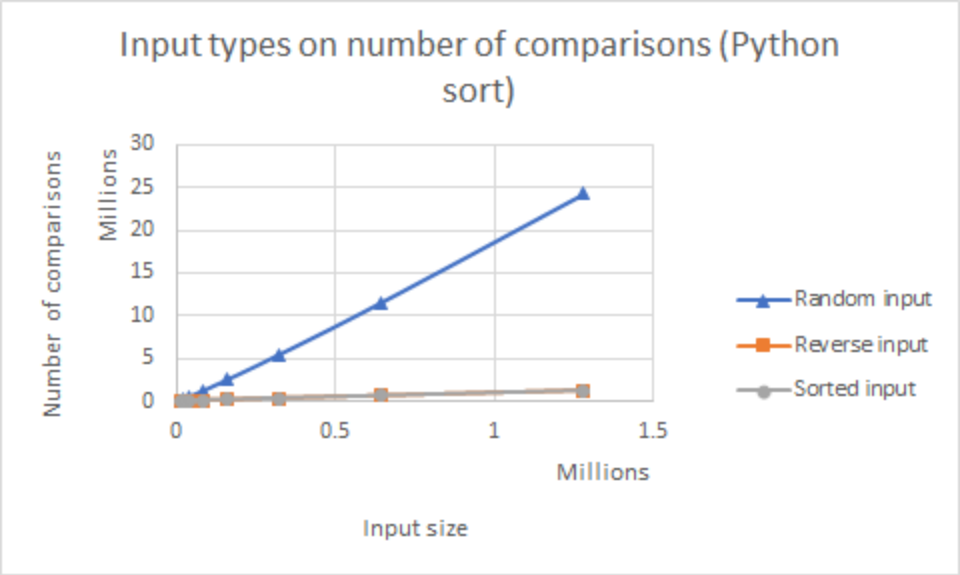

Figure 12: Input types against number of comparisons (Python sort)

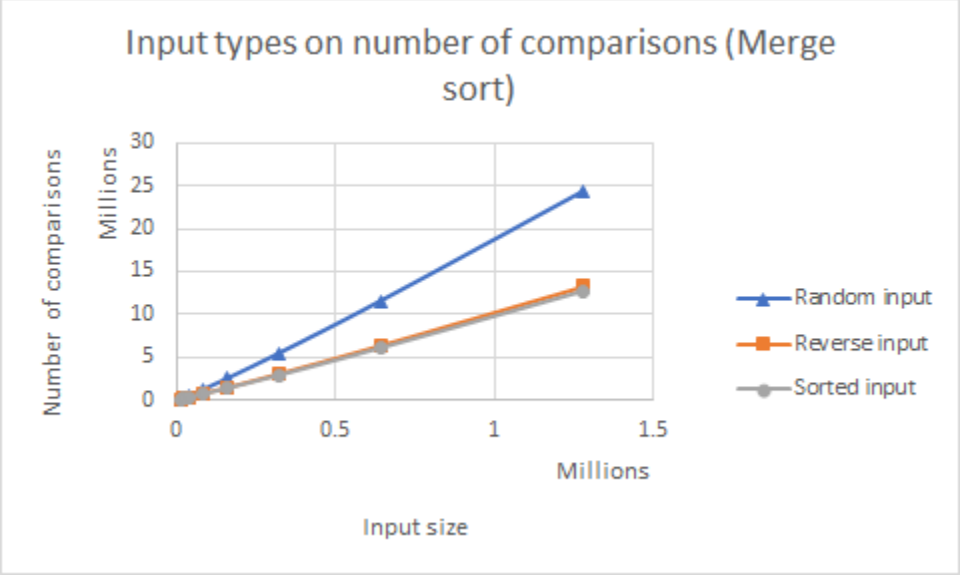

Figure 13: Input types against number of comparisons (merge sort)

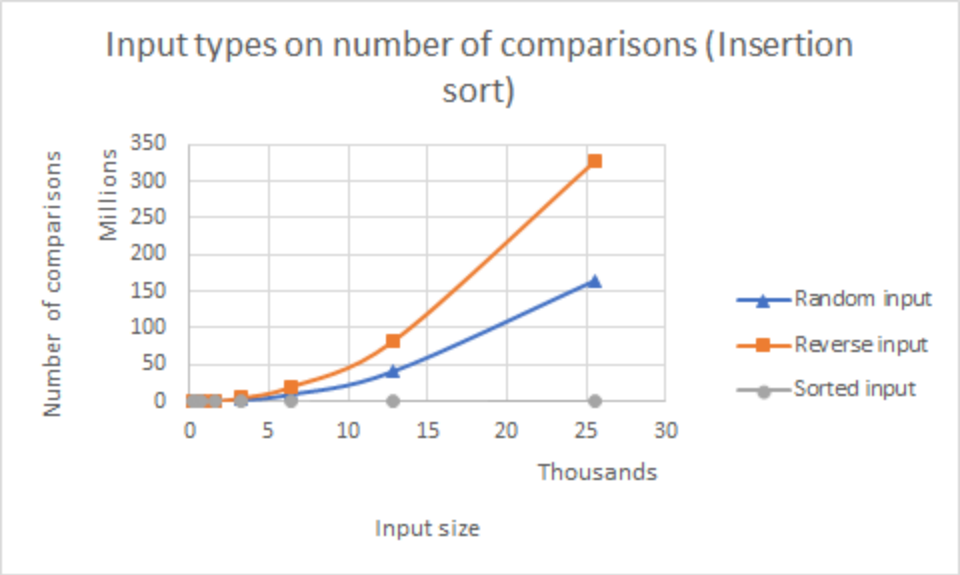

Figure 14: Input types against number of comparisons (insertion sort)

Appendix A: Additional Charts

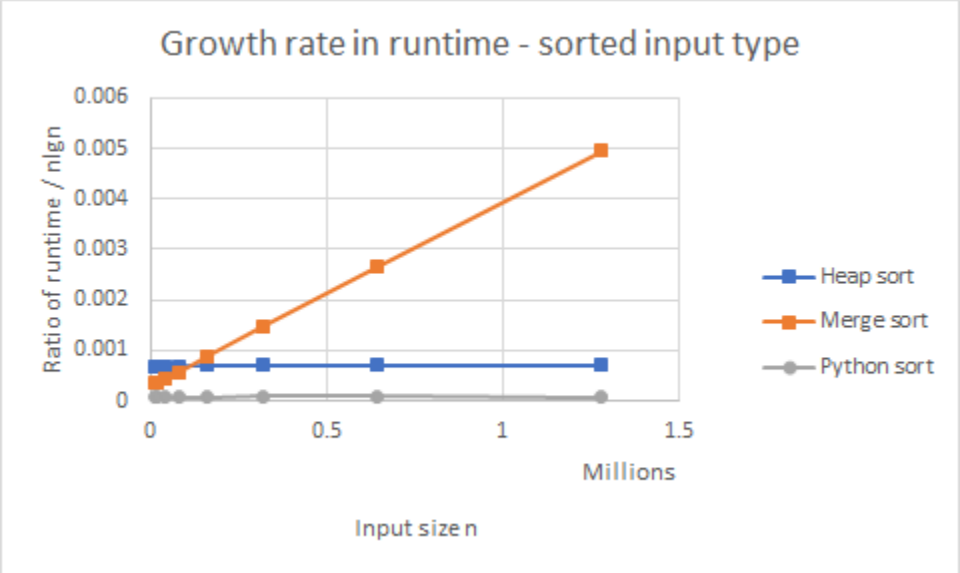

Figure 15: Growth rate in runtime (sorted - heap, merge and Python sort)

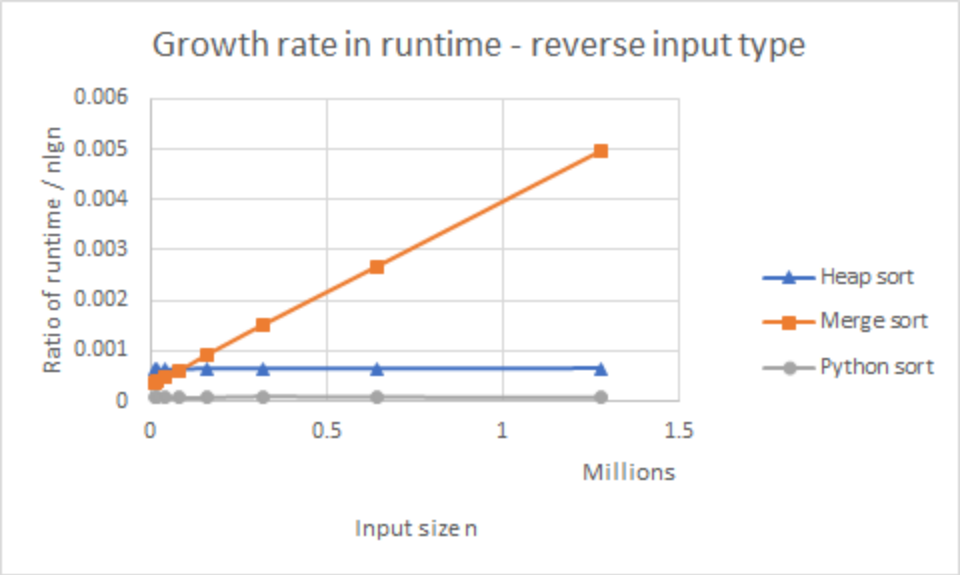

Figure 16: Growth rate in runtime (reverse - heap, merge and Python sort)

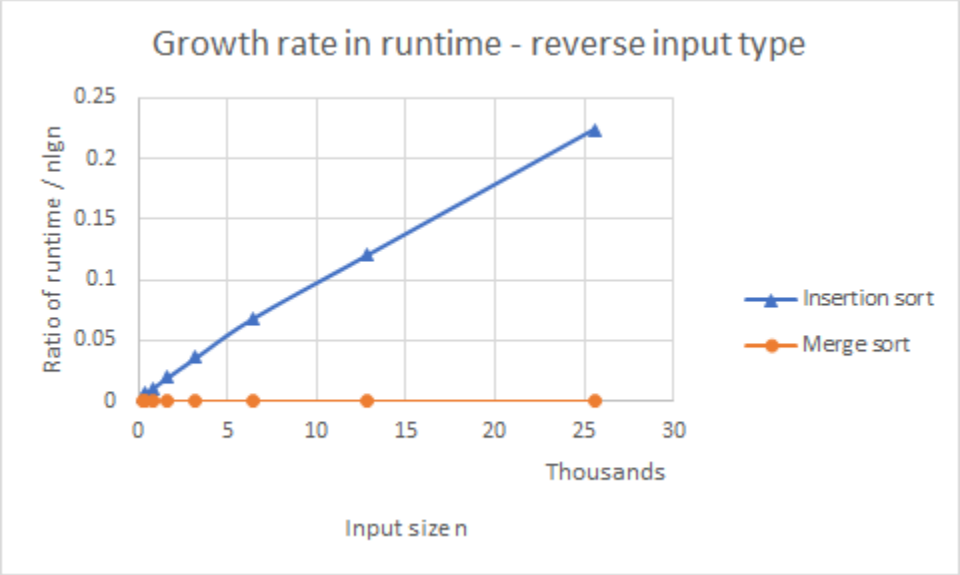

Figure 17: Growth rate in runtime (reverse - insertion and merge sort)

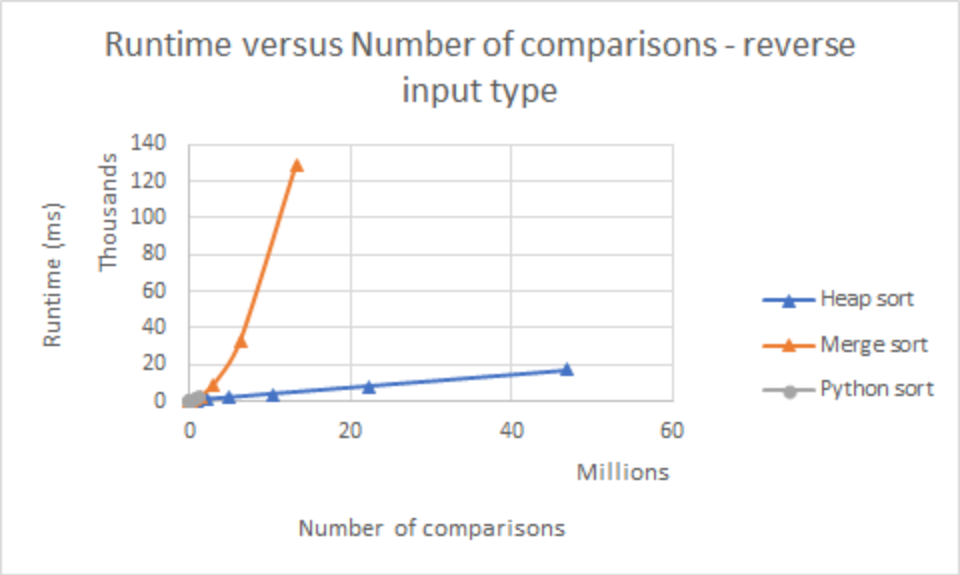

Figure 18: Runtime vs. Number of comparisons (reverse - heap, merge and Python sort)

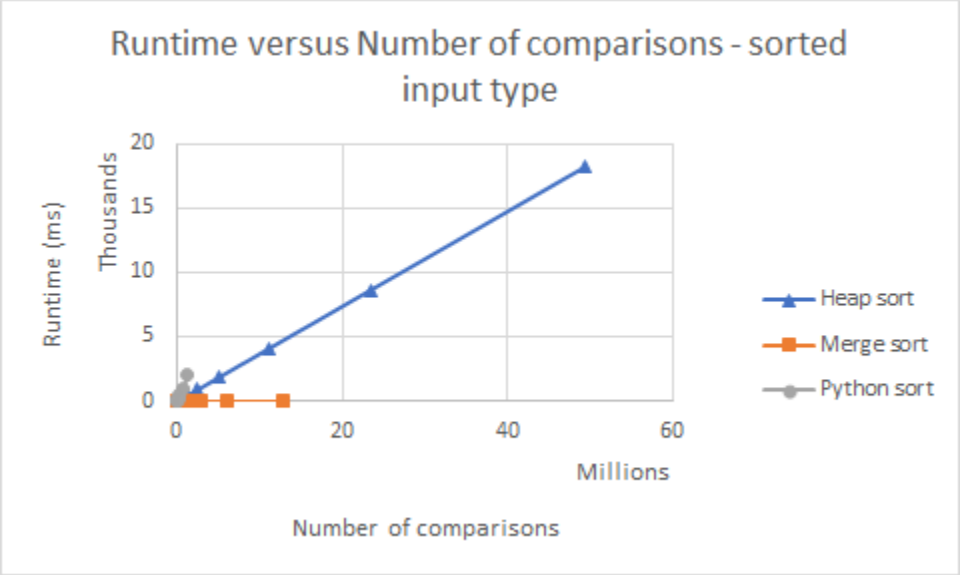

Figure 19: Runtime vs. Number of comparisons (sorted - heap, merge and Python sort)