MARY PWINT
CMPS 2200-01
Project1 Part3

AVL Tree Runtime Analysis

Methodology:

An AVL Tree is built in increments of an order of magnitude on each iteration until it has a million random integer entries. A total of five iterations are run. For every iteration, the time taken to perform 'contains' method is recorded in a CSV file and also printed out on console. The CSV files are then imported in an Excel sheet and the curves are plotted using a smoothing function.

Results:

The results of $log_{10}(input\_size)$ vs runtime in nanoseconds are graphed below. Series 1-5 are iterations 1-5 respectively.

Looking at Series 1, checking 'contains' in a tree of size 1, $log_{10}(1) = 0$, took 19300 ns. It is likely because on the very first call, the JVM needs to warm up. After the first couple iterations, the results later series seem to be more consistent with less fluctuation. However, surprisingly, a tree of 10^5 = 100k inputs took longer than 1M inputs for all trials except for series 4. It was not expected that searching a smaller tree would take longer. One possible reason is that on calling contains(random_int) on 100k tree, the random_int was not in the tree and thus ended up having to search through the entire tree and in the tree of 1M entries, the random_int was luckily easier to find. It is also important to note that the standard deviation was also the largest on iteration 4, from Figure 1.2.

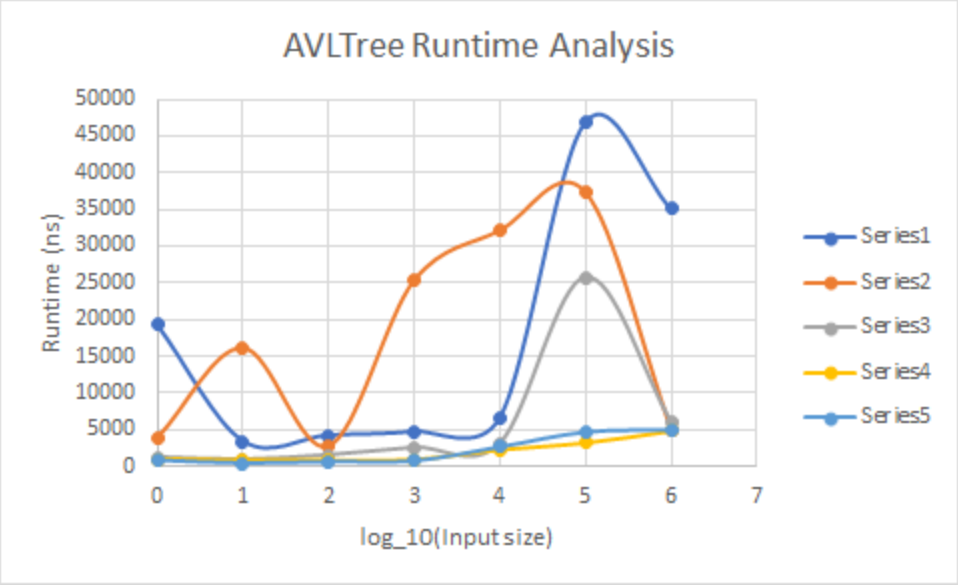

Figure 1.1: $log_{10}(input\_size)$ vs runtime graph of AVL tree showing five iterations

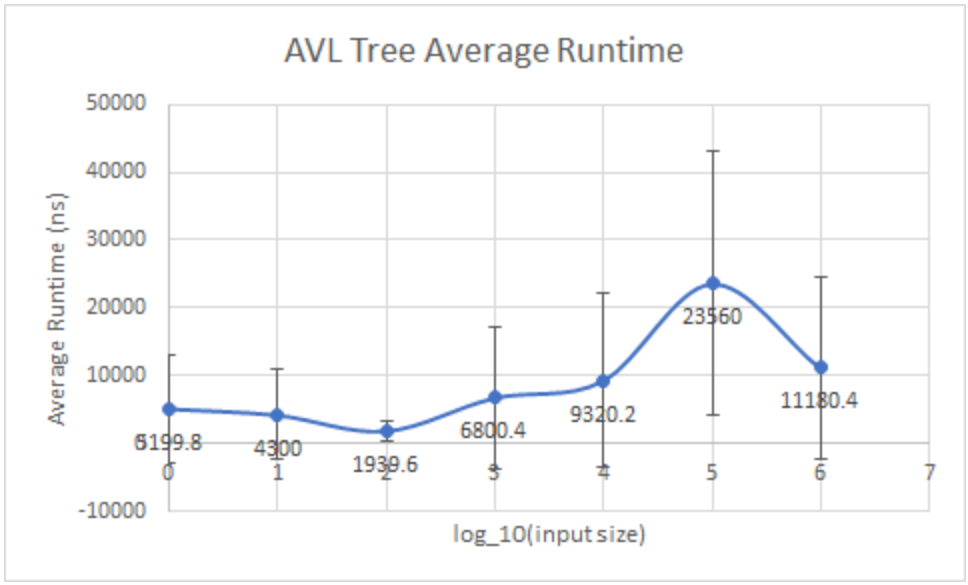

Figure 1.2: $log_{10}(input\_size)$ vs runtime graph of AVL tree showing the average of five iterations

Binary Search Tree Runtime Analysis

Methodology:

A Binary Search Tree is built in increments of an order of magnitude on each iteration until it has a million random integer entries. A total of five iterations are run. For every iteration, the time taken to perform 'contains' method is recorded in a CSV file and also printed out on console. The CSV files are then imported in an Excel sheet and the curves are plotted using a smoothing function.

Results:

The results of $log_{10}(input\_size)$ vs runtime in nanoseconds are graphed below. Series 1-5 are iterations 1-5 respectively.

Looking at Series 1, checking 'contains' in a tree of size 1, $log_{10}(1) = 0$, took 21700 ns. It is likely because on the very first call, the JVM needs to warm up. After the first couple iterations, the results later series seem to be more consistent with less fluctuation. There is an outlier in Series 2 at 1M data input, it took 657900 ns. Upon disregarding Series 2 in Figure 2.2, the data seemed much for consistent witht the expectation. Eyeballing Figure 2.3, the average runtime seems to grow linearly with the exception due to the outlier.

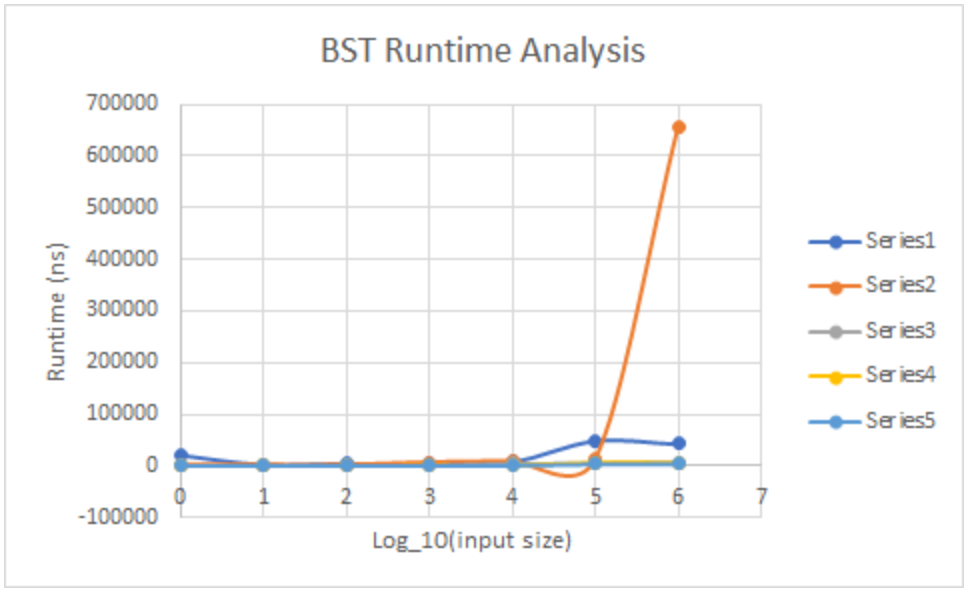

Figure 2.1: $log_{10}(input\_size)$ vs runtime graph of BST tree showing five iterations

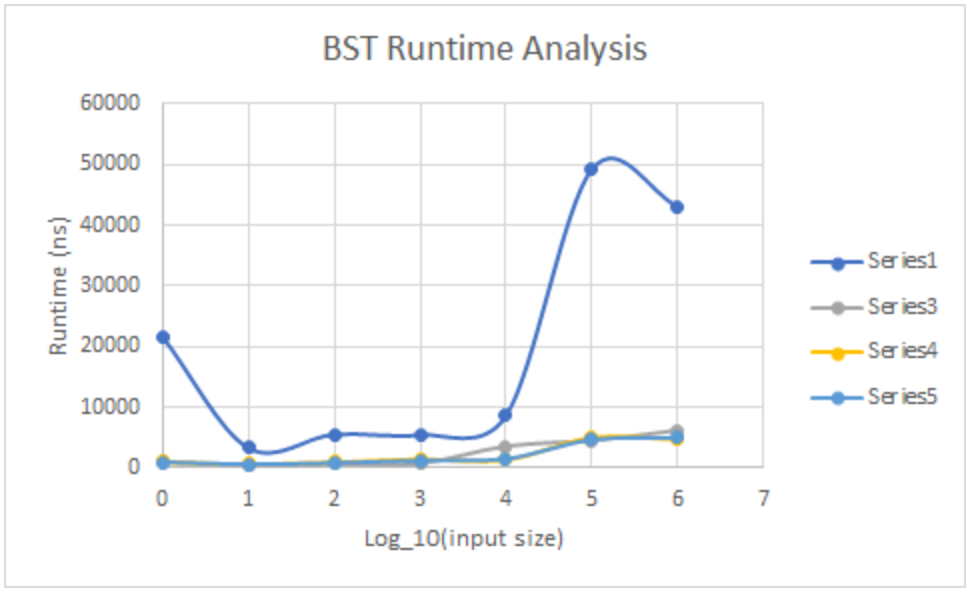

Figure 2.2: $log_{10}(input\_size)$ vs runtime graph of BST tree showing four iterations, excluding outlier Series 2

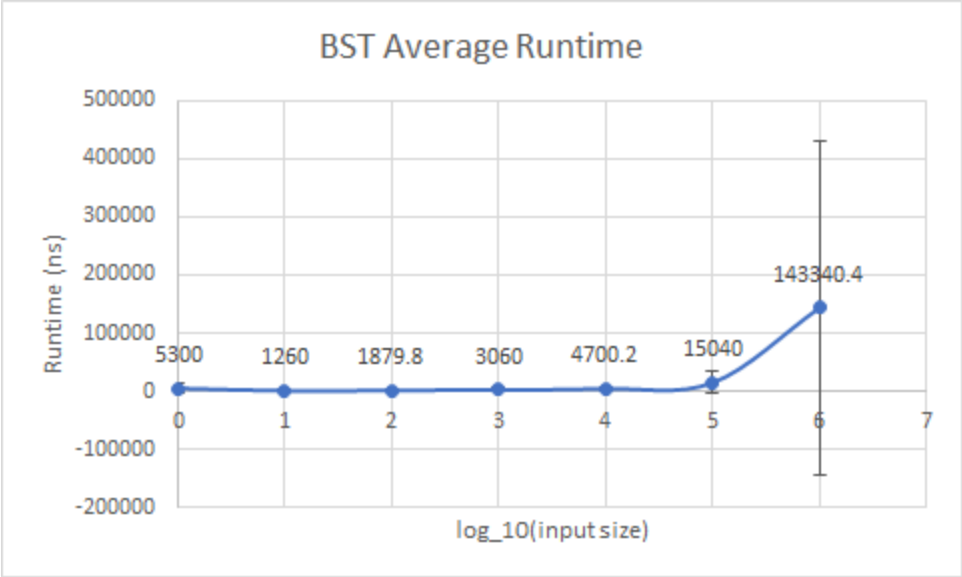

Figure 2.3: $log_{10}(input\_size)$ vs runtime graph of BST tree showing average of five iterations

From figure 2.4, runtimes of AVL tree and BST are comparable until around 100,000 nodes, after which AVL tree performs much better than a regular BST.

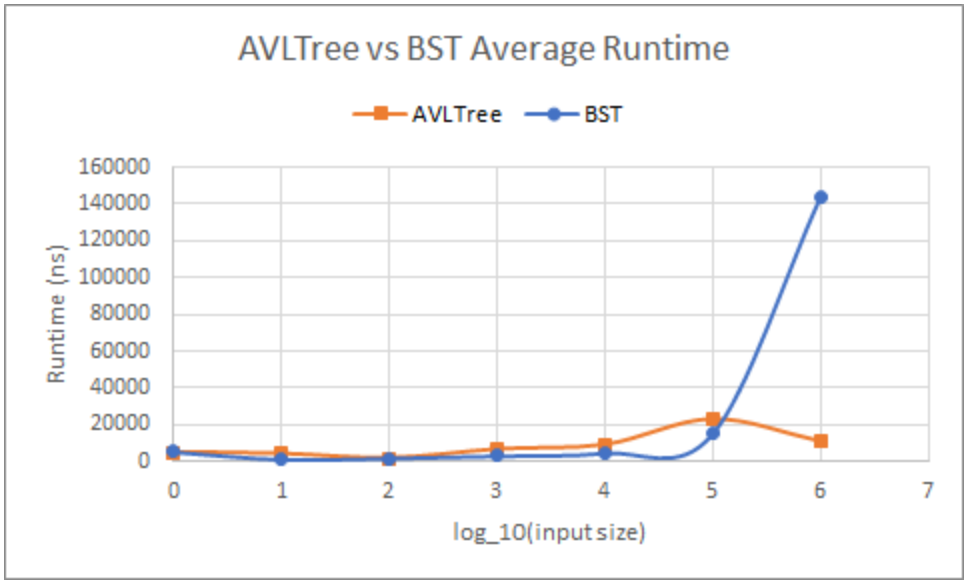

Figure 2.4: Comparison of the average runtimes of AVL Tree and BST

Sorting Algorithms Runtime Analysis:

Methodology:
Arrays of random integers are built in increments of an order of magnitude on each iteration until 10,000 random integer entries. A total of five iterations are run. For every iteration, the time taken to perform

'contains' method is recorded in a CSV file and also printed out on console. The CSV files are then imported in an Excel sheet and the curves are plotted using a smoothing function.

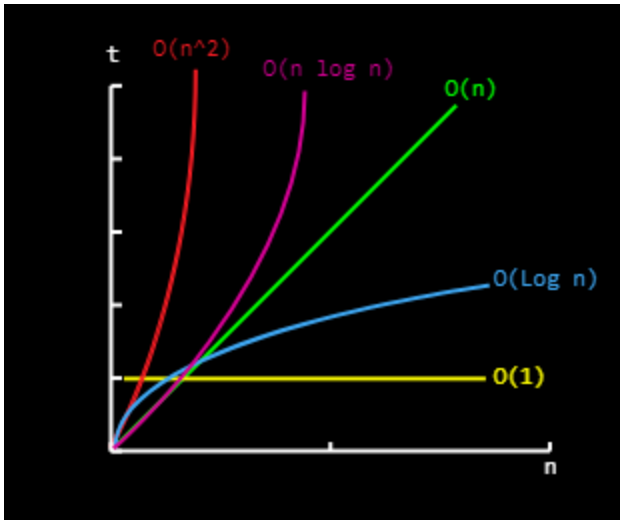

Figure 3.1: Standard curves of different Big-Oh runtimes

Mergesort:
From figures 3.2, the five trial runs were consistent and the average of the trials in figure 3.3 is comparable to the expected cure of O(n.log n) in figure 3.1.

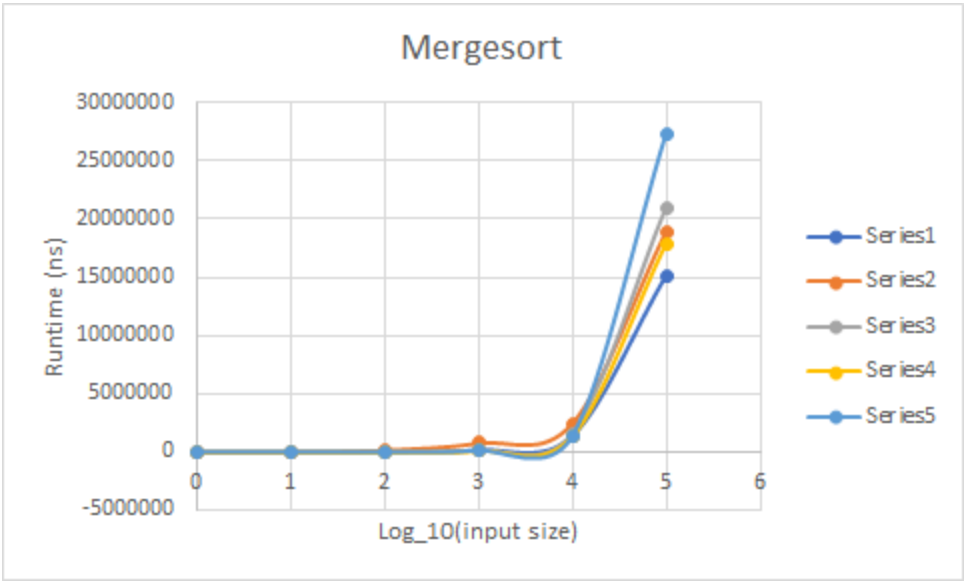

Figure 3.2: $log_{10}(input\_size)$ vs runtime graph of mergesort algorithm showing five iterations

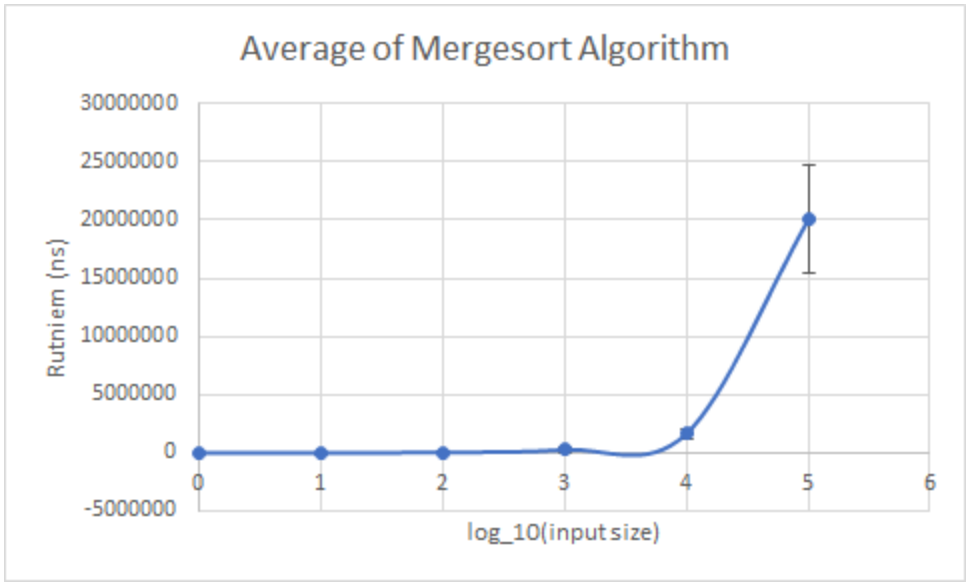

Figure 3.3: $log_{10}(input\_size)$ vs runtime graph of mergesort algorithm showing the average of five iterations

Quicksort:

The average graph for quicksort was very similar to mergesort (figure 3.5) except for Series 1 with 100,000 inputs. However, noting that the worst case of quicksort algorithm is $O(n^2)$, it was likely that Series 2-5 were good/average cases and that Series 1 was a worse case. It is also important to note that the implemented quicksort algorithm is optimized to avoid the worst possible cases such as bad choice of pivot. Therefore, from the five tests, it had comparable runtime to the mergesort algorithm.

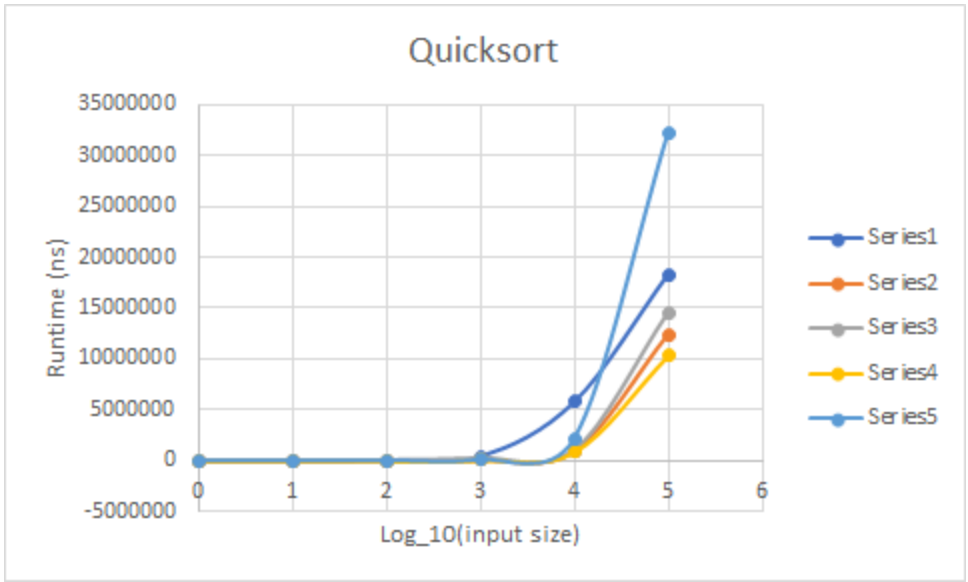

Figure 3.4: $log_{10}(input\_size)$ vs runtime graph of quicksort algorithm showing five iterations

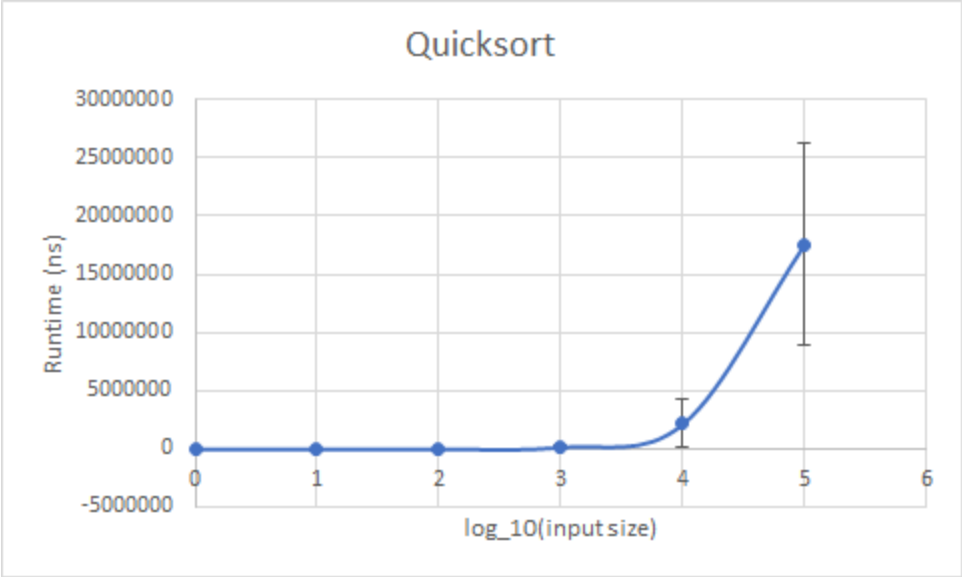

Figure 3.5: $log_{10}(input\_size)$ vs runtime graph of quicksort algorithm showing the average of five iterations

Heapsort:

Heapsort performance is also very similar to both mergesort and heapsort. However, the performance was unexpected for $log_{10}(3) = 1000$ elements array. It could be have a coincidence that Series 2 and 4 were worse cases for 1000 inputs.

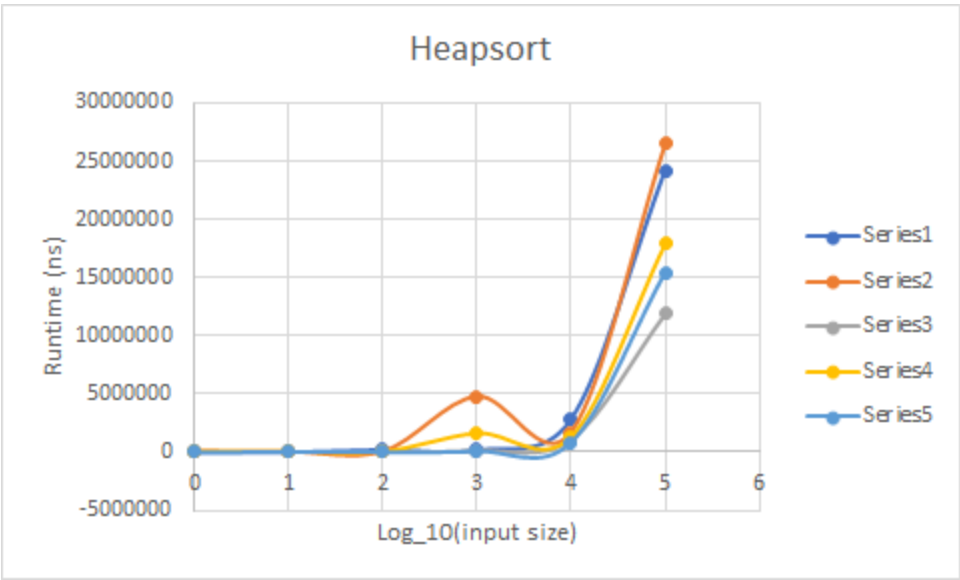

Figure 3.6: $log_{10}(input\_size)$ vs runtime graph of quicksort algorithm showing five iterations

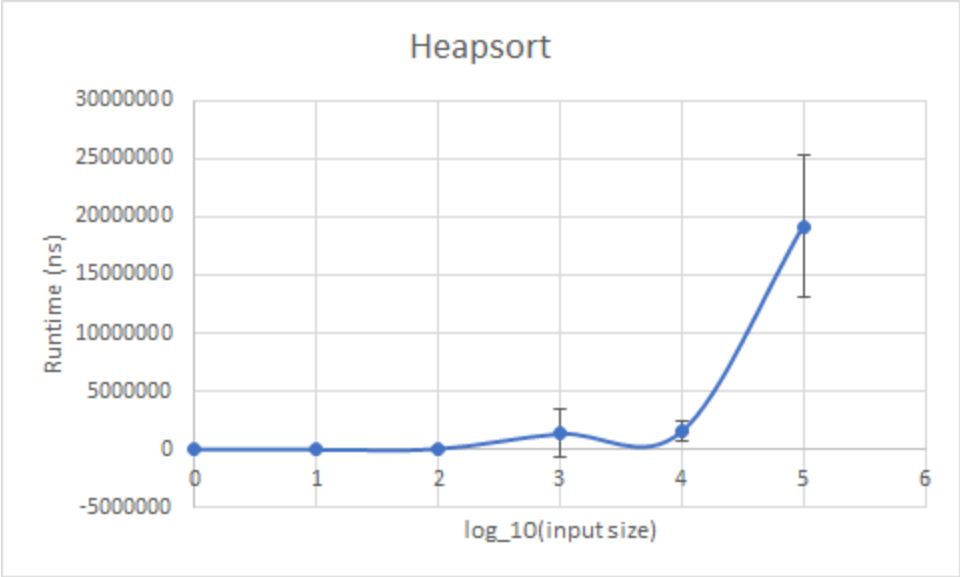

Figure 3.7: $log_{10}(input\_size)$ vs runtime graph of quicksort algorithm showing the average of five iterations

Comparing the average runtimes of these algorithms in figure 3.8, it seems that for most cases, all of these algorithms have comparable runtime, unless for worst cases in quicksort which can blow up to $O(n^2)$.

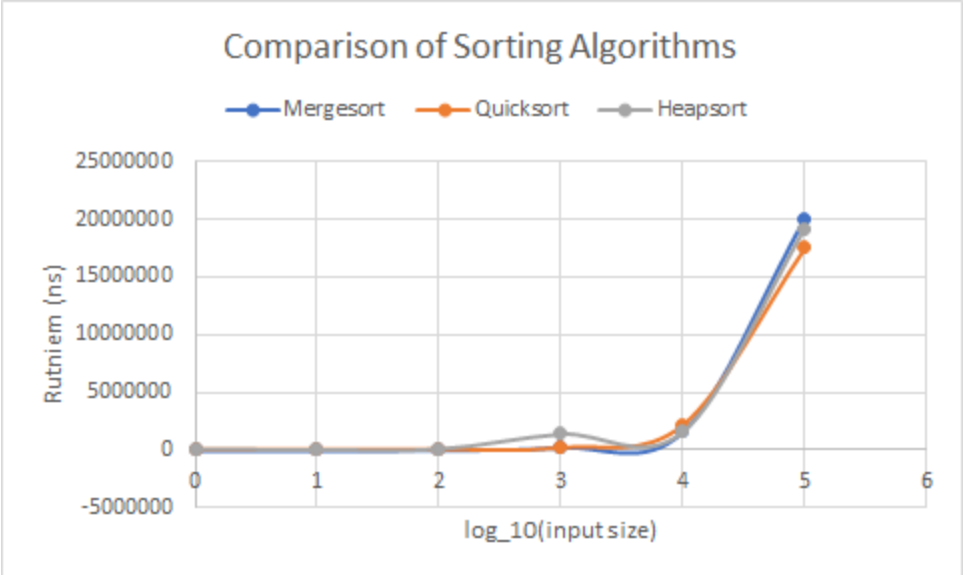

Figure 3.8: Comparison of sorting algorithms - mergesort, quicksort, and heapsort