CSC 505: Design and Analysis of Algorithms

Spring 2018

REPORT: Programming Assignment #01

Analyzing Sorting Algorithms

(Insertion Sort, Merge Sort, Heap Sort and Sorting Utility)

Student Name	*Student ID*	*Unity ID*
Akshay Nalwaya	200159155	analway
K. Sai Sri Harsha	200207493	skanama

DESCRIPTION OF EXPERIMENT SETUP

Machine Architecture / Specifications

Operating System		Ubuntu
Language		C++
Compiler Version		g++ 5.4.0
Processor	Type	Intel(R) Core(TM) i7-7700HQ
	Speed	2.8 GHz
Cache Size		L1d - 32K, L1i - 32K, L2 – 256K, L3 – 6144K
Memory Size		4GB RAM

Choice of Inputs

The inputs are divided into three categories: completely sorted, reverse sorted and randomly populated arrays. The correctness of all algorithms is tested by running them 10 times on a variety of input sizes, i.e., 0.1, 0.2, 0.4, 0.8, 1.6, 3.2, 6.4 and 12.8 million. The range of input values for insertion sort algorithm is slightly lower than the rest of algorithms owing to the high time complexity $(O(n^2))$ associated with this algorithm.

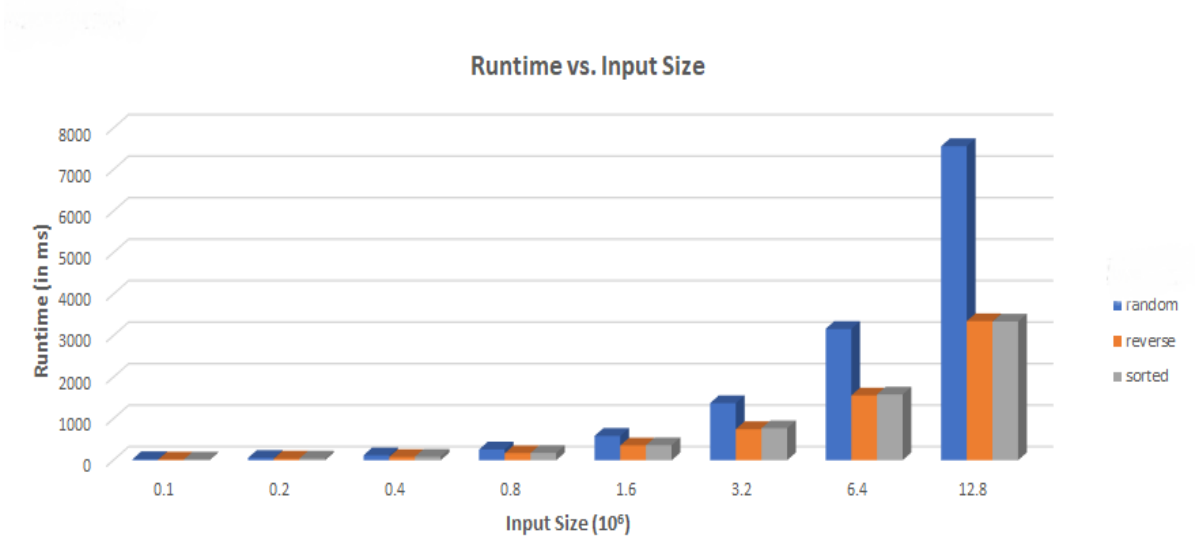
EXPERIMENTAL RESULTS

The 3 sorting algorithms were run on different input sizes and types as specified above. The results obtained are as follows

KEY FINDINGS: HEAP SORT

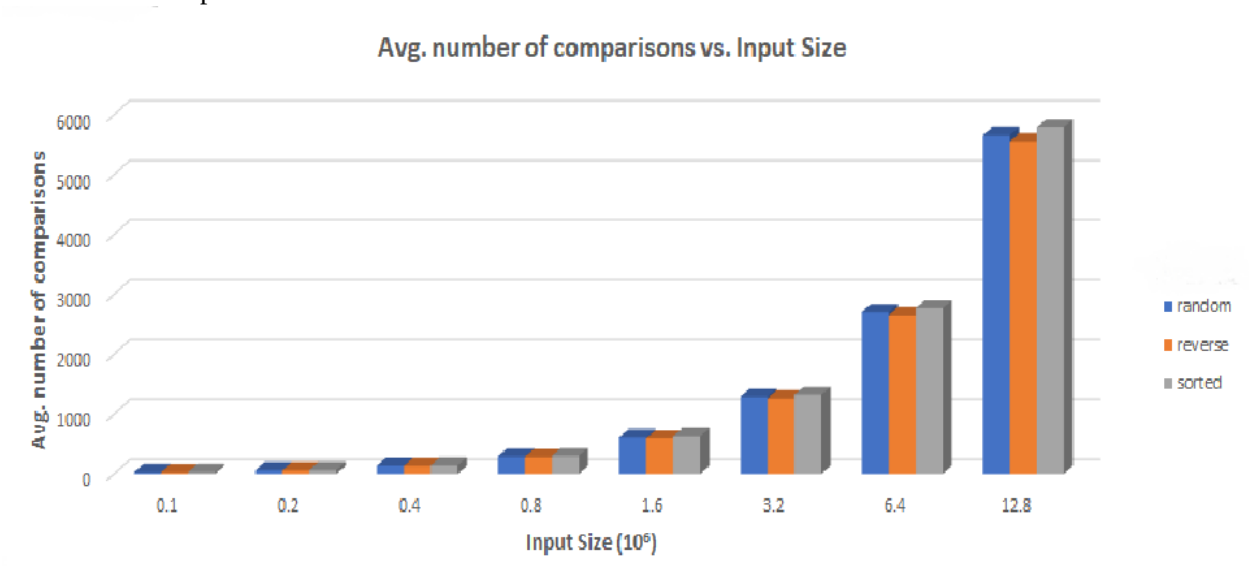
Runtime Analysis

- The runtime of sorted and reverse sorted inputs are approximately same for all the input sizes
- Runtime of this algorithm for randomly populated input array is noticeably higher than the other two input sizes



Number of Comparisons

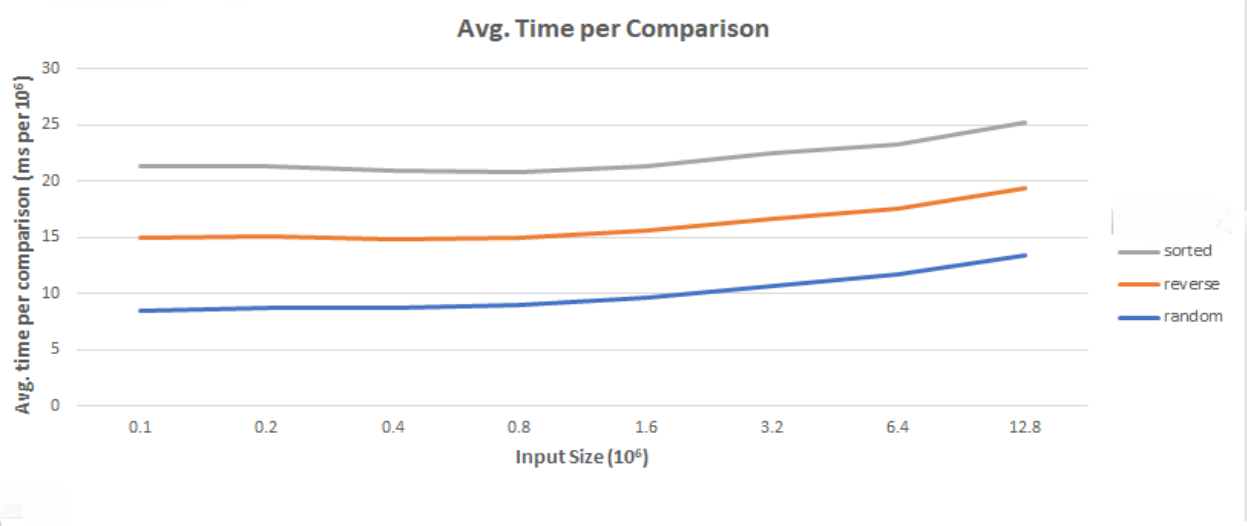
- There is not much difference in the average number of comparisons for the different input types, with randomly populated inputs having somewhat higher number of comparisons than the reverse sorted inputs



Average runtime per comparison

Average runtime per comparison is constant for all the three types of data. Also, average time per comparison for various types follows the below pattern:

Avg. time per comparison $_{sorted}$ > Avg. time per comparison $_{reverse}$ > Avg. time per comparison $_{random}$

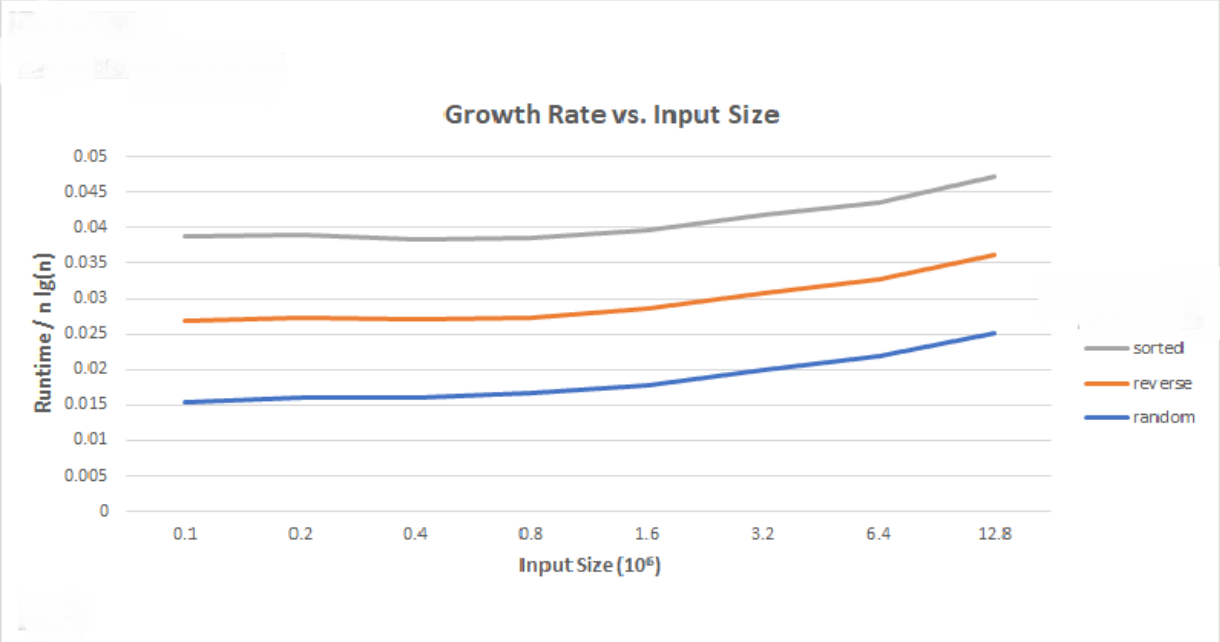


Growth Rate in runtime

Average runtime/n*lg(n) is constant for different input sizes.

Different types of data follow the following pattern:

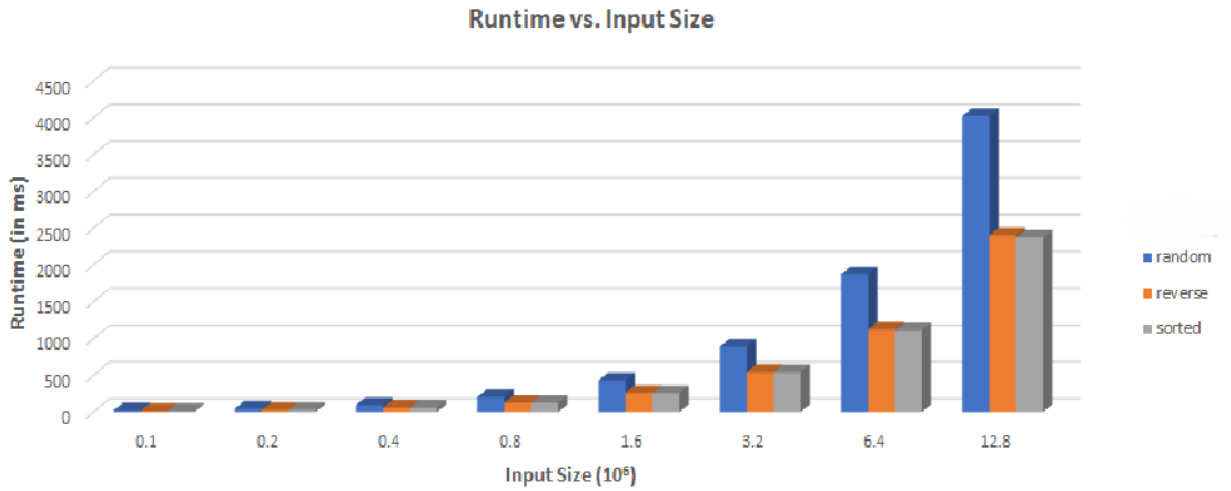
[Runtime/n * lg(n)] $_{sorted}$ > [Runtime/n * lg(n)] $_{reverse}$ > [Runtime/n * lg(n)] $_{random}$



KEY FINDINGS: MERGE SORT

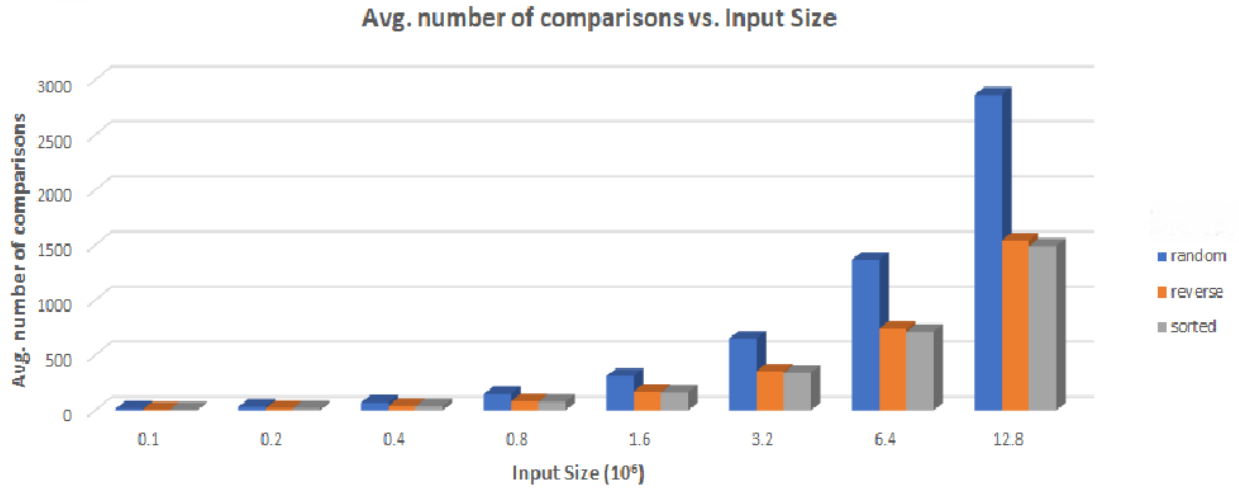
Runtime Analysis

- The runtime of sorted and reverse sorted inputs is approximately the same for all the input sizes with runtime of reverse sorted arrays being slightly higher than the sorted arrays
- This algorithm runs significantly longer when input array is randomly populated than when inputs belong to other two categories



Number of Comparisons

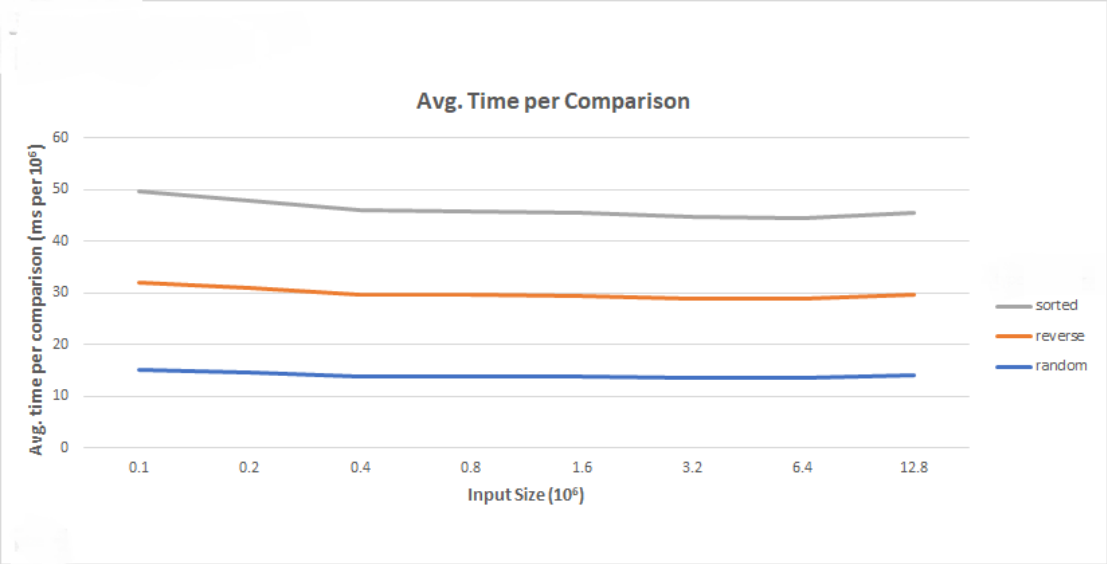
- Number of comparisons for reverse sorted array input is slightly higher than the sorted array inputs
- As in runtime analysis, randomly generated input array reports higher number of comparisons than the other two input types



Average runtime per comparison

Average runtime per comparison is constant for all the three types of data. Also, average time per comparison for various types follows the below pattern:

Avg. time per comparison $_{sorted}$ > Avg. time per comparison $_{reverse}$ > Avg. time per comparison $_{random}$

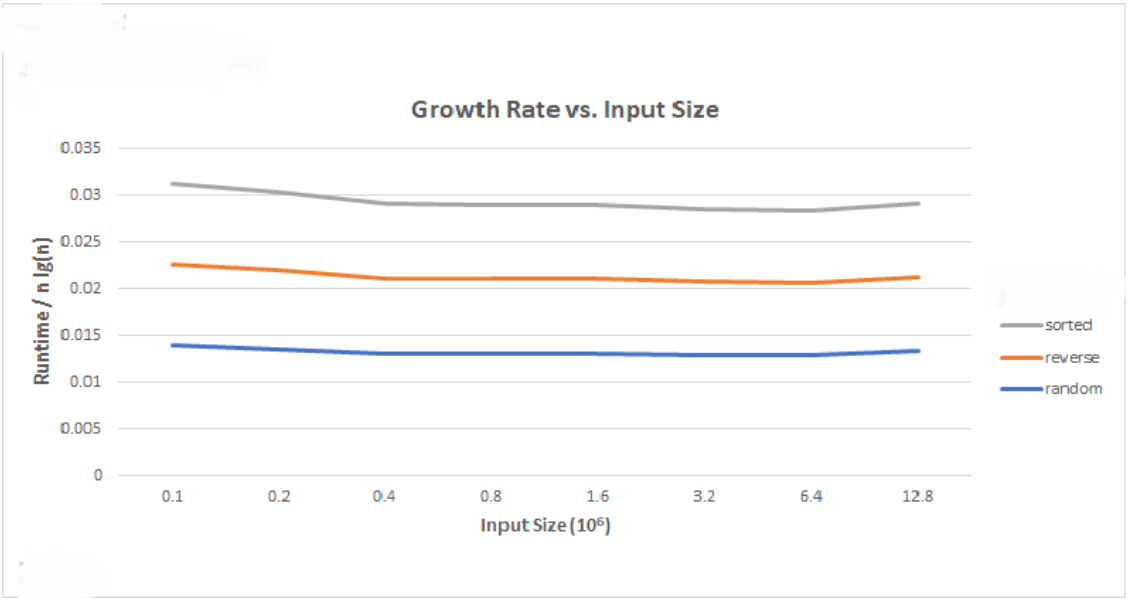


Growth Rate in runtime

Average runtime/n*lg(n) is constant for different input sizes.

Different types of data follow the following pattern:

$$[\text{Runtime}/n * \lg(n)]_{\text{sorted}} > [\text{Runtime}/n * \lg(n)]_{\text{reverse}} > [\text{Runtime}/n * \lg(n)]_{\text{random}}$$

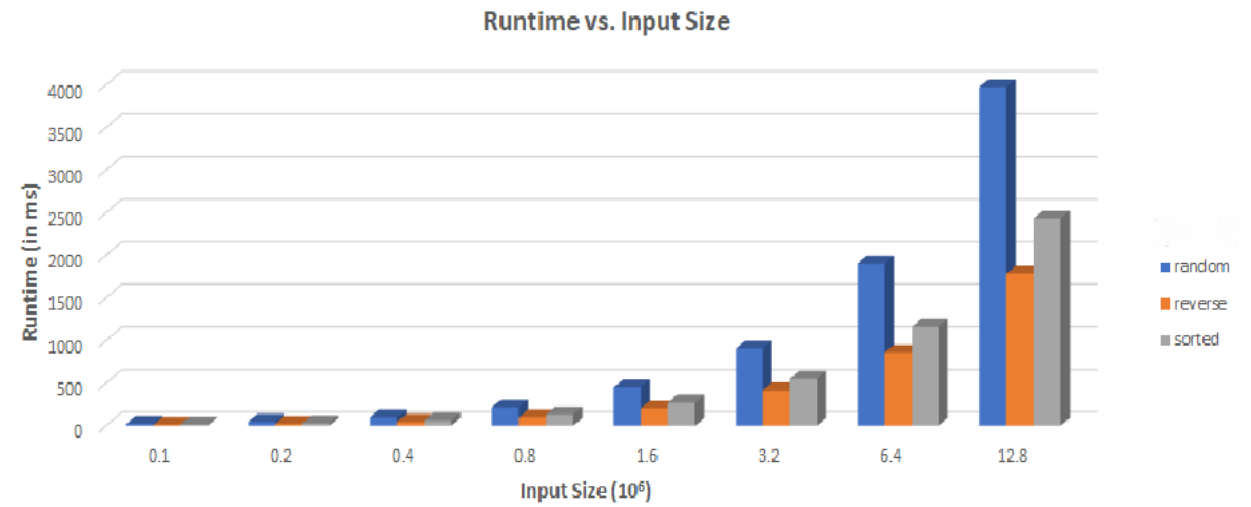


KEY FINDINGS: C++ SORTING UTILITY

Runtime Analysis

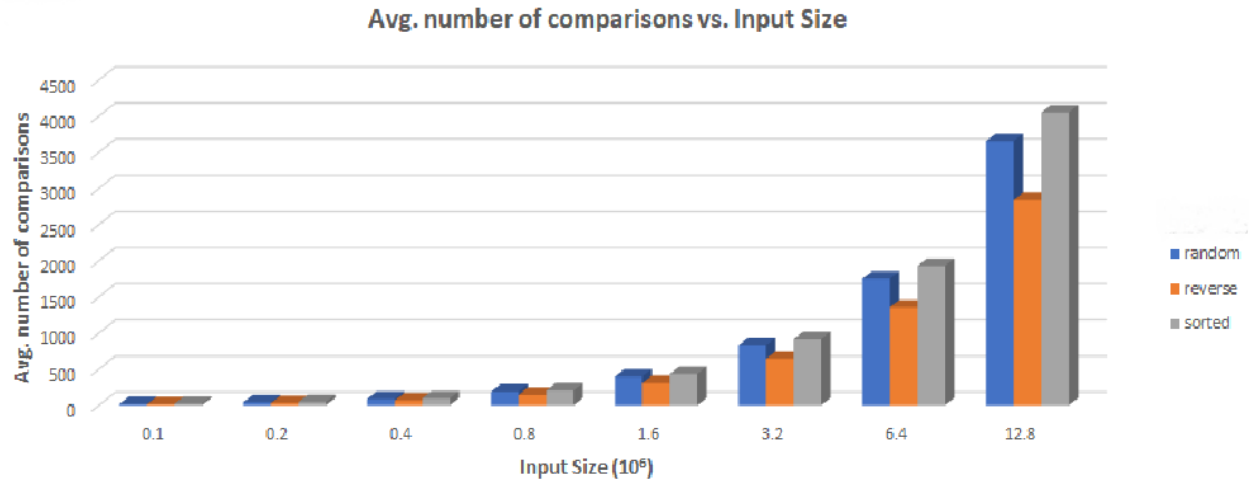
- For smaller inputs (upto 0.6-0.8 million entries), the runtime is approximately equal for all the three types of arrays (sorted, random, reverse sorted). However, for large sizes (above 0.8 million), the default sorting has the following pattern for runtime:

$$\text{Runtime}_{\text{random}} > \text{Runtime}_{\text{sorted}} > \text{Runtime}_{\text{reverse}}$$



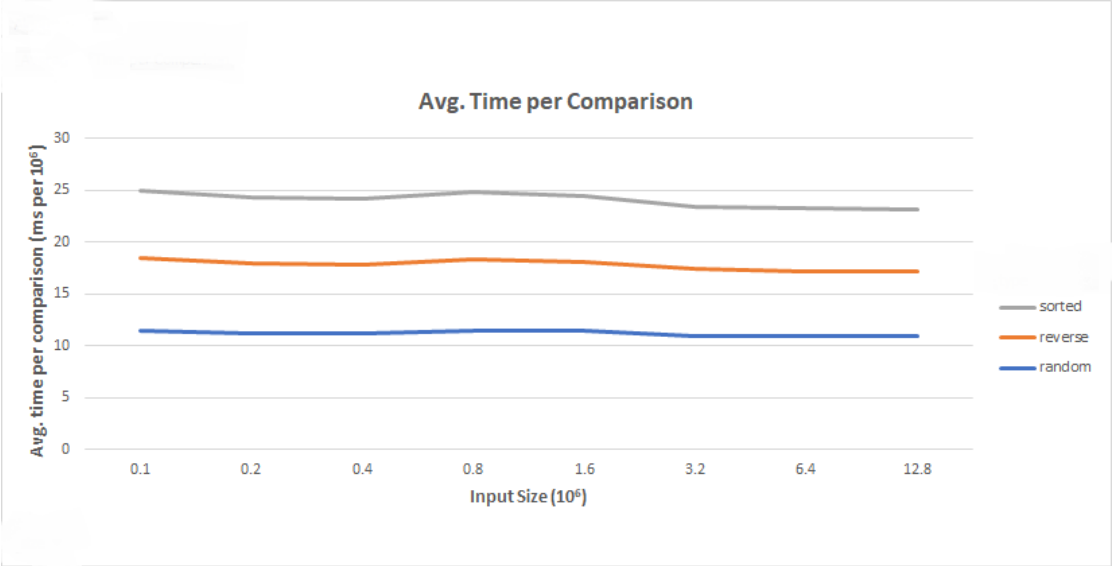
Number of Comparisons

- For smaller inputs (upto 1 million entries), the average number of comparisons are almost same for all the types of arrays (sorted, random, reverse sorted). However, for large sizes (in the range of 10 million entries), the number of comparisons for random and sorted arrays are more than that of reverse sorted arrays.



Average runtime per comparison

- It is seen that average time per comparison for all input types is approximately constant with the time for already sorted array being the lowest and random being the lowest.

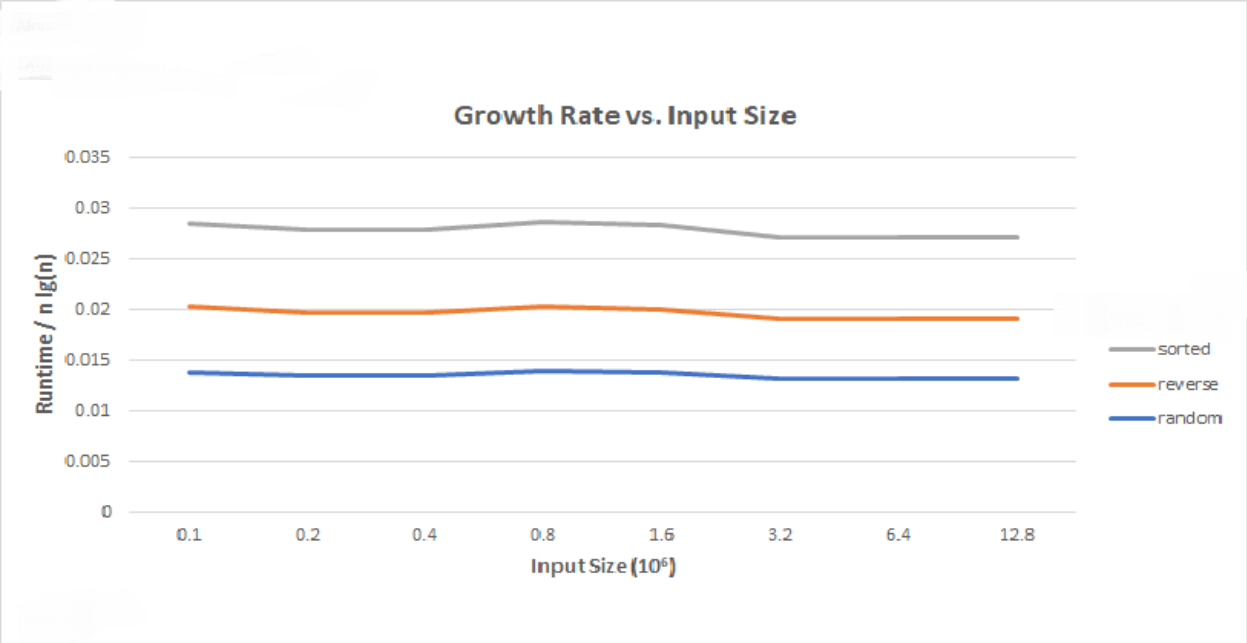


Growth Rate in runtime

Average runtime/n*lg(n) is constant for different input sizes.

Different types of data follow the following pattern:

$$[\text{Runtime}/n * \lg(n)]_{\text{sorted}} > [\text{Runtime}/n * \lg(n)]_{\text{reverse}} > [\text{Runtime}/n * \lg(n)]_{\text{random}}$$



COMPARATIVE STUDY OF ALL ALGORITHMS: Runtime

Runtime	Sorting Algorithm		
	Heapsort	Mergesort	Sorting Utility
0.1	19.03	14.42	10.67
0.2	39.92	29.53	21.89
0.4	81.06	60.33	46.17
0.8	169.70	125.48	99.20
1.6	356.14	262.94	206.54
3.2	749.09	542.44	416.63
6.4	1559.57	1123.02	859.62
12.8	3351.19	2404.64	1786.11

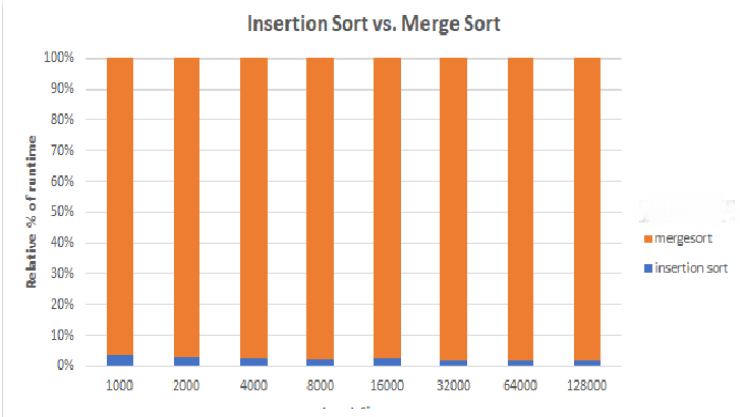
The following points can be noted from the above table:

- Sorting Utility outperforms heapsort and merge sort in terms of runtime and hence can be concluded that it's implementation is a hybrid of multiple algorithms
- Heap sort takes slightly higher time than merge sort

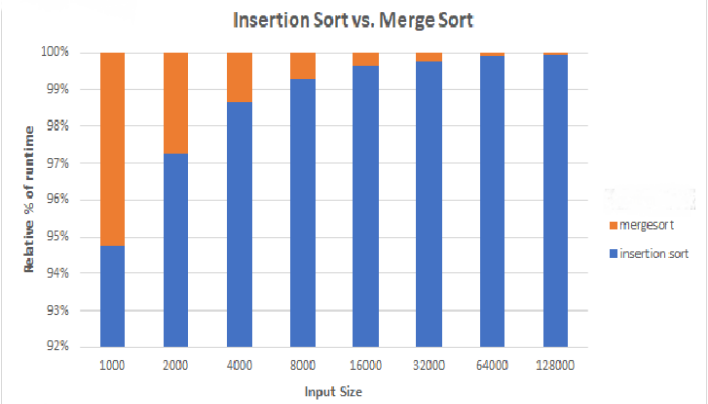
COMPARISON OF INSERTION SORT AND MERGE SORT: Runtime Analysis

CASE 1: SORTED INPUT

The runtime of insertion sort is smaller than that of merge sort irrespective of input sizes. In this case, insertion sort takes around 5% of the total time(runtime of merge sort + insertion sort)



Sorted Input



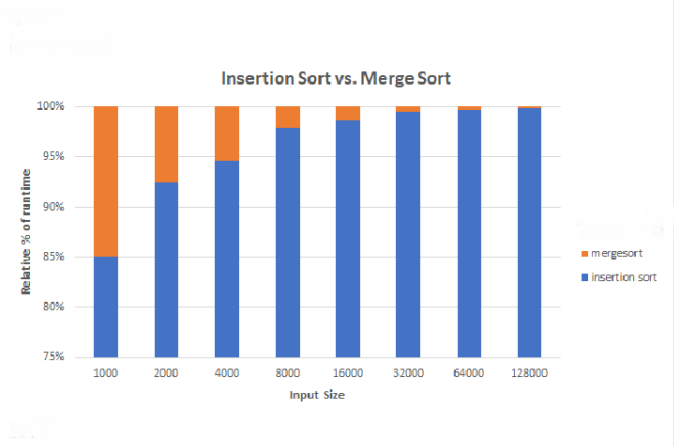
Reverse Sorted Input

CASE 2: REVERSE SORTED INPUT

The runtime of insertion sort is greater than that of merge sort. For smaller inputs, insertion sort takes around 95 - 97% of the total time (runtime of merge sort + insertion sort). On the other hand, larger inputs, insertion sort takes around 99% of the total time (runtime of merge sort + insertion sort).

CASE 3: RANDOMLY GENERATED INPUT

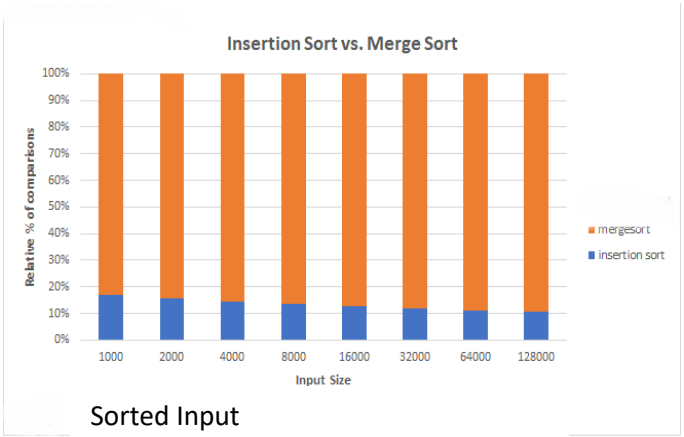
Overall, the runtime of insertion sort is greater than that of merge sort. For smaller inputs, insertion sort takes around 85 - 90% the total time (runtime of merge sort + insertion sort). While for larger inputs, insertion sort takes around 98 - 99% of the total time (runtime of merge sort + insertion sort).



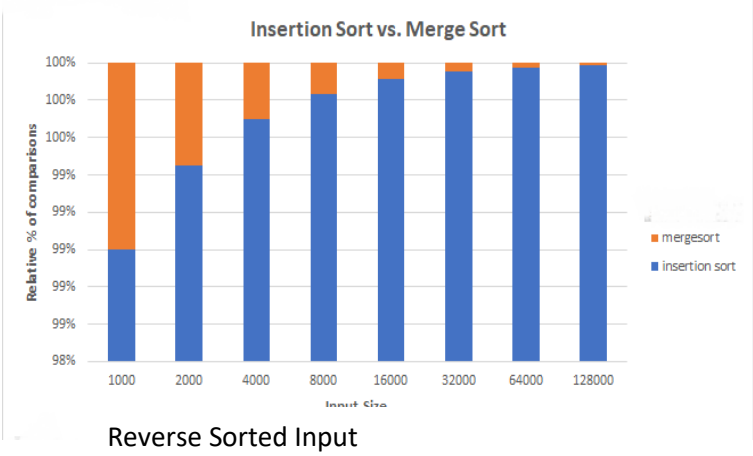
COMPARISON OF INSERTION SORT AND MERGE SORT: Number of Comparisons

CASE 1: SORTED INPUT

The number of comparisons for insertion sort is smaller than that of merge sort irrespective of input sizes. In this case, insertion sort makes around 10 - 15% of the total comparisons (comparisons made by merge sort + insertion sort).



Sorted Input



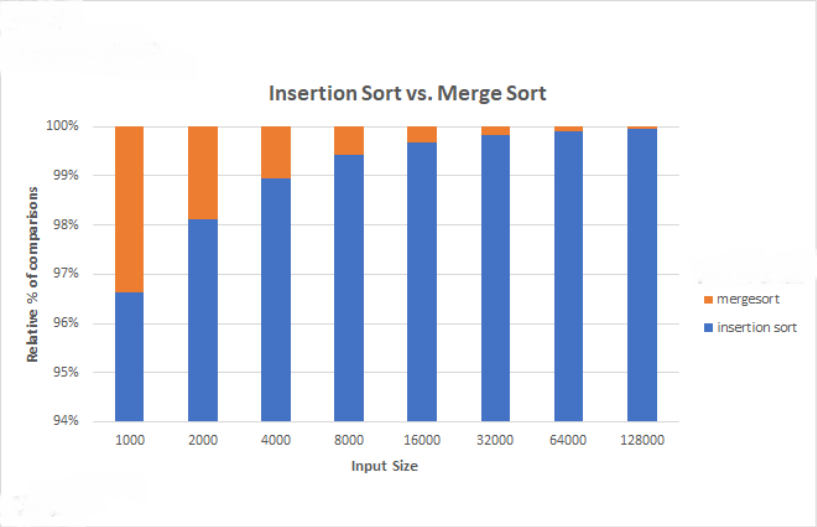
Reverse Sorted Input

CASE 2: REVERSE SORTED INPUT

In reverse sorted data, the number of comparisons for insertion sort is greater than that of merge sort. Irrespective of the size of data, insertion sort makes more than 99% of the total comparisons (comparisons made by merge sort + insertion sort).

CASE 3: RANDOMLY GENERATED INPUT

The number of comparisons for insertion sort is greater than that of merge sort. For smaller inputs, insertion sort makes around 95 - 97% the total comparisons (comparisons made by merge sort + insertion sort). For larger inputs, insertion sort makes around 98 - 99% of the total comparisons (comparisons made by merge sort + insertion sort).



SUMMARY AND CONCLUSIONS

- The experiments clearly demonstrate the claims made in theory pretty well and as explained in the theory
- Runtime for sorted and reverse sorted algorithms are nearly the same for heap sort and merge sort for all types of inputs
- Sorting utility of C++ outperforms both heap and merge sort algorithm, which indicates the implementation to be a hybrid of two or more of the algorithms