

Program 1 report

Amey Meher (avmeher) and Deep Mehta (dmmehta2)

February 22, 2023

Abstract

This report covers our experimental analysis of three sorting algorithms, namely Insertion sort, Merge sort, and Quick sort. The report first gives an overview of the sorting algorithm, then discusses the theoretical analysis of the three algorithms. It then proceeds to describe the experimental setup, along with the specifics of the implementation. Different types of analysis are conducted, namely stability check of these experiments, comparisons of run time, and the total number of comparisons as well as a few others are discussed in the results section. The report concludes the fact that the sorting algorithm's efficiencies vary based on numerous factors and each algorithm performs better than the others in certain situations. The paper also provides a future scope for other experiment ideas that might give further insights into the evaluation of these algorithms.

1 Introduction

Sort is an important functionality in most of the day-to-day applications, be it in database management systems for removing duplicates, ordering elements, or be it in application programs for user-specific tasks. Due to the extensive usage of sorting algorithms, it is quite essential to use an optimized way of sorting as per the requirement of the task. In this report, we mainly discuss the applicability to various situations of three sorting algorithms: Insertion sort, Merge sort, and Quick sort.

Different requirements prefer different implementations of sorting. As an example, if the input size is too small, generally insertion sort is preferred over merge and quick sort as these are heavy-duty operations that would be better suited to perform sorting on a larger set of elements. Our aim in this report is to figure out such peculiarities of these sorting algorithms which would help us understand in detail which implementation of the sorting algorithm would be better suited for the task at hand.

2 Theory

Table 1 gives the theoretical run-time analysis of the three algorithms according to each of their best, average, and worst-case complexity. From the table, we can briefly see that

Insertion sort performs the best if the inputs are already sorted or partially sorted and have a complexity of $\mathcal{O}(n)$. Merge sort and Quick sort have an average case complexity of $n \log n$, which suggests the better application of these algorithms on larger input sizes. Also, we can note that for the worst case, Quick sort performs with a complexity of n^2 .

We also aim to identify from Merge and Quick sort, which performs better as they have the same average case time complexity. Although it heavily depends upon the implementation of the algorithm, we have specified a few parameters for both algorithms which would help in comparing both algorithms. This will be discussed in further detail in the experimental analysis section.

Case analysis	Insertion sort	Merge sort	Quick sort
Best case	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$
Average case	$\mathcal{O}(n^2)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$
Worst case	$\mathcal{O}(n^2)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$

Table 1: Theoretical Run-time analysis

3 Experimental design

Below are a few of the things that we have kept in common for all the implementations which would ensure the algorithms are compared in a fair manner.

1. The input is in the form of a LinkedList having two pointers, head, and tail and only has one directional pointer; thus can be traversed in only one direction.
2. While noting the performance of each of the experiments, the input data are kept common among all three algorithms.
3. The total time taken does not take into consideration the time to read the input from the user or other auxiliary tasks such as printing the output. It only considers the time to sort the list.
4. Each set of experiments is worked on the same hardware. Below are the hardware specification for the experiments.
 - Processor: Macbook air M2
 - RAM: 16GB
 - OS: Mac OS Ventura 13.2.1
5. We have also used an Input data generator in python which accepts two parameters as input:
 - n - number of elements in the input
 - b - number of blocks. Each block will have sorted elements within them.

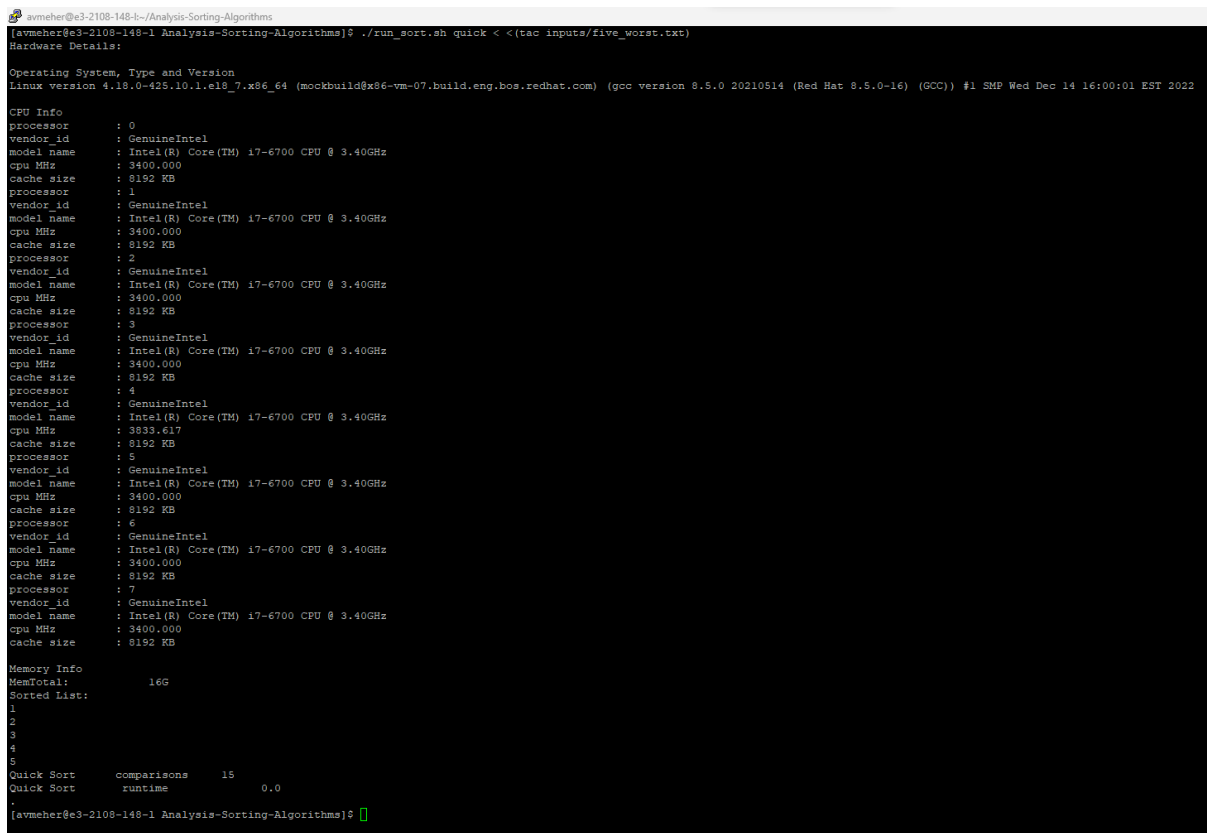
Below is the set of experiments we thought of conducting with the three algorithms:

1. **Stability test:** Testing the algorithm multiple times with the same input to check if there is any deviation in each of the runs.
2. **Correctness test:** Testing the algorithm with a known set of input data as well and also comparing if the output from the algorithm matches with the expected output. All the algorithms are tested on the below set of input files provided. We have created a set of automated triggers to test all the sorting algorithms on the below set of files.
 - 100_sorted.txt
 - 100_reversed.txt
 - b_1000000_1000.txt
 - r_128_01.txt
 - five_worst.txt
 - sixteen_reversed.txt
 - sixteen.txt
 - zr_1000000.txt
 - zr_5000000.txt
3. **Runtime test:** Monitoring the total runtime for each of the algorithms.
4. **Number of comparison test:** Monitoring the total number of comparisons for each of the algorithms.
5. **Time taken vs Number of comparisons test:** Determining which of the two metrics, the total time taken or the total number of comparisons yields the expected theoretical behavior of the algorithms.
6. **Determining the constant multiple between Merge sort and Quick sort:** Determining the constant multiple with which Merge sort and Quick sort differs by. This will be done by dividing the number of comparisons or the runtime with $n \log n$.
7. **Effect of varying block size:** The effect of varying the block size in the input data generator on the different sorting algorithms. This would work in visualizing the best and worst case scenarios of the Insertion and Quick sort as they differ from their average case
8. **Finding the k value (threshold number of inputs) for insertion and quick sort:** Vary the number of inputs or block size parameters and find the threshold point till which insertion sort outperforms quick sort.
9. **Finding the k value (threshold number of inputs) for insertion and merge sort:** Vary the number of inputs or block size parameters and find the threshold point till which insertion sort outperforms merge sort.

4 Implementation

We have implemented all three sorting algorithms as Java classes. The input data is then passed to these algorithms through a shell script wrapper. Also, for generating input data, we have used a python based code as suggested in the experimental design section.

Below is the screenshot of execution of one of the sorting algorithm on a sample file on NCSU VCL machine.



```
avmeher@e3-2108-148-1 Analysis-Sorting-Algorithms
[avmeher@e3-2108-148-1 Analysis-Sorting-Algorithms]$ ./run_sort.sh quick < <(tac inputs/five_worst.txt)
Hardware Details:
Operating System, Type and Version
Linux version 4.18.0-425.10.1.el8_7.x86_64 (mockbuild@x86-vm-07.build.eng.bos.redhat.com) (gcc version 8.5.0 20210514 (Red Hat 8.5.0-16) (GCC)) #1 SMP Wed Dec 14 16:00:01 EST 2022

CPU Info
processor       : 0
vendor_id      : GenuineIntel
model name     : Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz
cpu MHz       : 3400.000
cache size    : 8192 KB
processor       : 1
vendor_id      : GenuineIntel
model name     : Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz
cpu MHz       : 3400.000
cache size    : 8192 KB
processor       : 2
vendor_id      : GenuineIntel
model name     : Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz
cpu MHz       : 3400.000
cache size    : 8192 KB
processor       : 3
vendor_id      : GenuineIntel
model name     : Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz
cpu MHz       : 3400.000
cache size    : 8192 KB
processor       : 4
vendor_id      : GenuineIntel
model name     : Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz
cpu MHz       : 3833.617
cache size    : 8192 KB
processor       : 5
vendor_id      : GenuineIntel
model name     : Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz
cpu MHz       : 3400.000
cache size    : 8192 KB
processor       : 6
vendor_id      : GenuineIntel
model name     : Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz
cpu MHz       : 3400.000
cache size    : 8192 KB
processor       : 7
vendor_id      : GenuineIntel
model name     : Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz
cpu MHz       : 3400.000
cache size    : 8192 KB

Memory Info
MemTotal:      16G
Sorted List:
1
2
3
4
5
Quick Sort      comparisons    15
Quick Sort      runtime         0.0
[avmeher@e3-2108-148-1 Analysis-Sorting-Algorithms]$
```

Figure 1: A sample run on NCSU VCL machine

5 Analysis

1. Correctness test:

We tested all the sorting algorithms on all the input files provided in an automated workflow. Below is the screenshot of a run of the workflow which confirms that the output from all the run is as expected and matching the expected output.

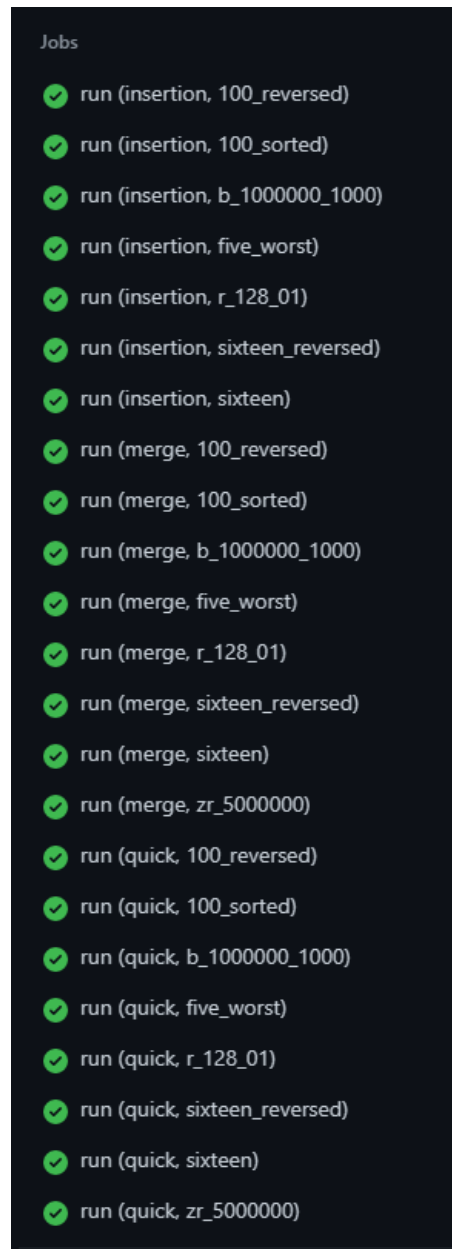


Figure 2: Correctness test

2. Stability test:

We ran all the algorithms on the same set of data multiple times to check if there is any deviation between different runs of the same data. Table 2 confirms that Insertion sort, Merge sort and Quick sort are stable as we can see that the number of comparisons for the same input has not changed. For this experiment, we have considered the value of n as 1000.

Run no.	Insertion sort	Merge sort	Quick sort
1	120409	8247	10630
2	120409	8247	10630
3	120409	8247	10630
4	120409	8247	10630
5	120409	8247	10630
6	120409	8247	10630
7	120409	8247	10630
8	120409	8247	10630
9	120409	8247	10630
10	120409	8247	10630

Table 2: Theoretical Run-time analysis

3. Runtime test:

Figures 3 and 4 represent the trend in total time taken for the sorting algorithm. For insertion sort, we have considered a different range of input as the time taken for sorting large inputs by Insertion sort is significantly higher than the other two algorithms.

We can see that there is inconsistency with the increase in the input size. This is due to the fact that there are many processes being run in the background, and for a period of time these background tasks might have finished and would have given more processing power for this task, which would explain the sudden drop in the run time from the figure 3.

From figure 4, we can see that as the input size is increasing, the time taken for quick sort is increasing at a faster rate than merge sort. This suggests that merge sort is performing better than quick sort.

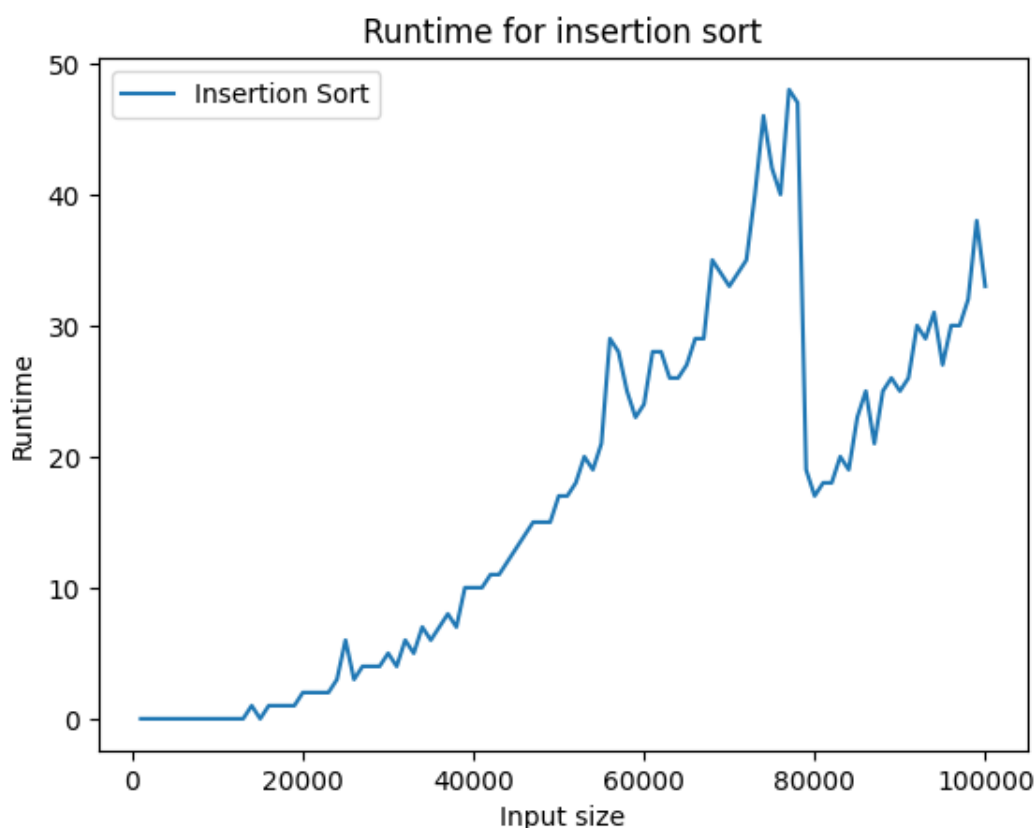


Figure 3: Runtime test for Insertion sort

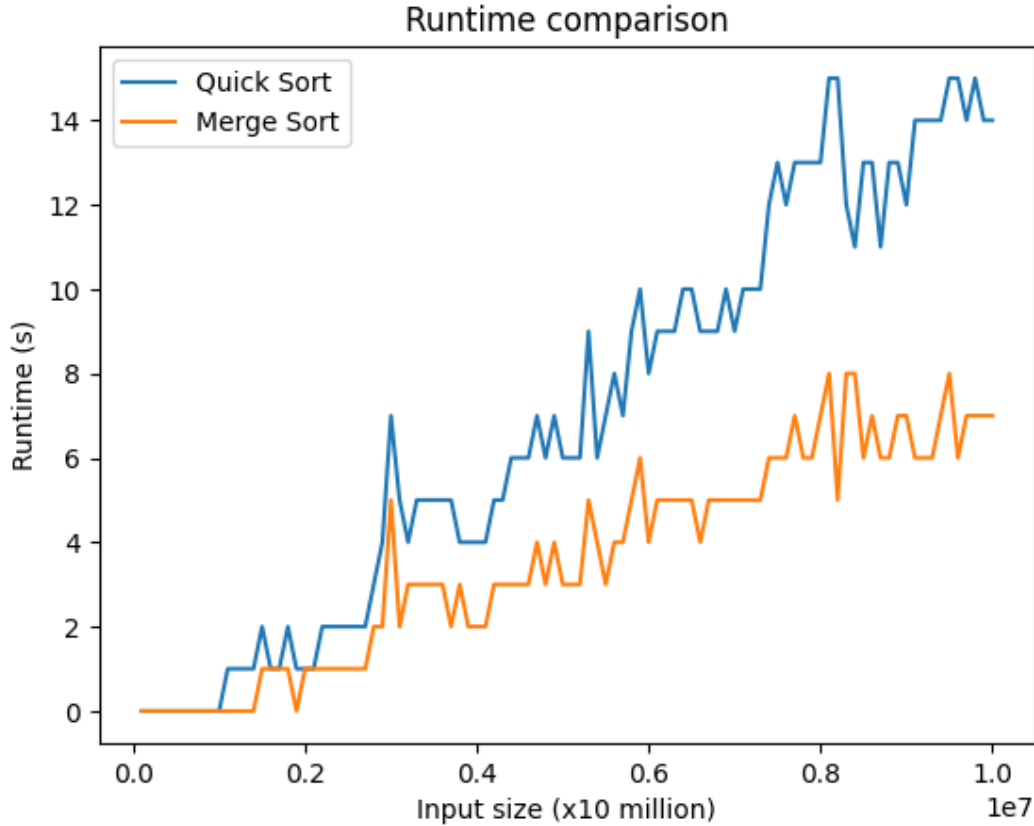


Figure 4: Runtime test for Quick and merge sort

4. **Number of comparisons test:** Figures 5 and 6 represent the trend in the number of comparisons with an increasing number of elements in the input. Over here as well, we have considered the range of a number of elements in the input differently for Insertion sort as the number of comparisons is significantly larger than the other two algorithms.

From figure 5, we can see that the graph follows the n^2 curve as we have expected in the theoretical analysis.

From figure 6, we can see that these graphs are slightly curved as well and not a straight line, and follow the curve of $n \log n$ as we have expected theoretically. Also, as seen in the comparison between Quick and Merge sort, we can clearly see that Merge sort is performing better than Quick sort.

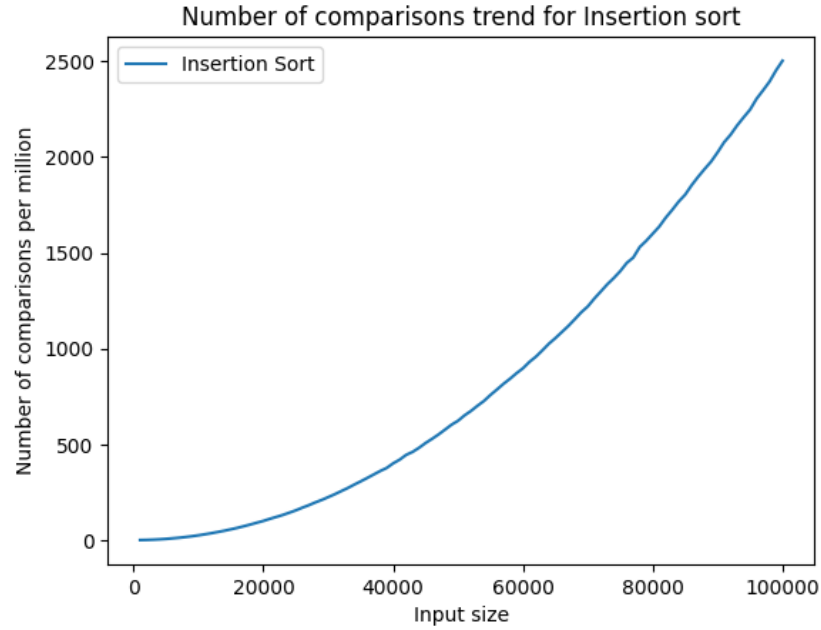


Figure 5: Number of comparisons for insertion sort

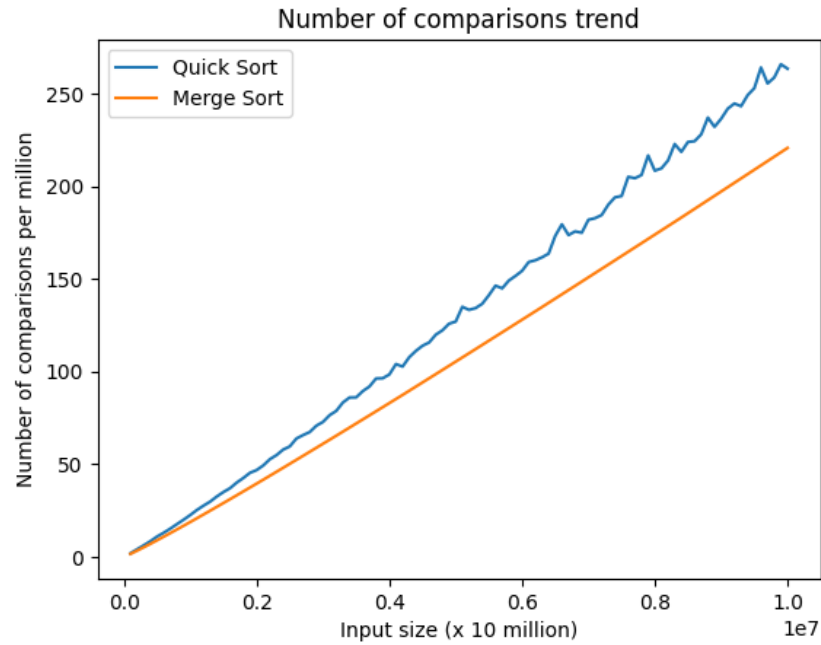


Figure 6: Number of comparisons for quick and merge sort

5. **Time taken vs Number of comparisons test:** From the figures 3, 4, 5 and, 6, we are able to observe that number of comparison metric is well suited for showcasing the complexity of the algorithm rather than the total time taken by the algorithm as there are very less deviation from the theoretical analysis.

6. Determining the constant multiple between Merge sort and Quick sort:

The Y-axis in the figure 7 is the total number of comparisons for Quick and Merge sort divided by $n \log n$. This graph helps in visualizing the constant multiple by which Merge sort is outperforming Quick sort. The constant comes out to be 1.189 (1.13 / 0.95)

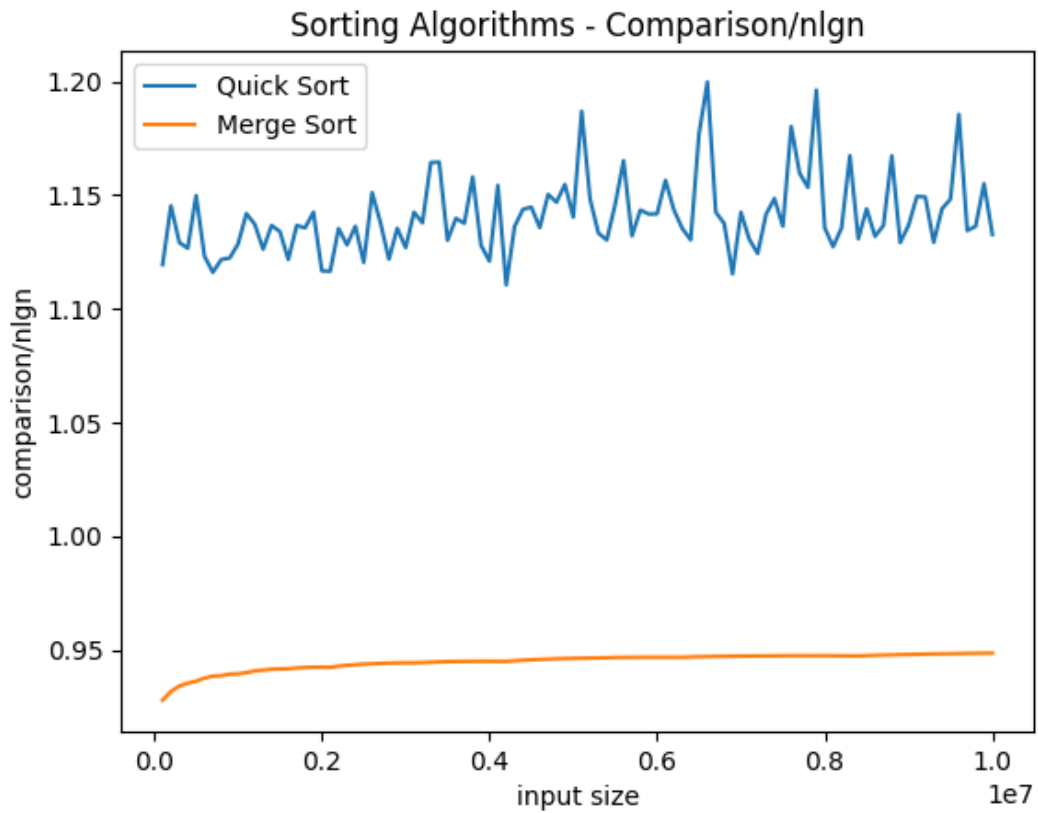


Figure 7: Number of comparisons divided by $n \log n$

7. Effect of varying block size:

From figure 8, we can observe that the runtime for insertion sort is dependent upon the block size parameter used in the input generator. For a lesser value of block size, the elements in the input are partially sorted, thus insertion sort would be outperforming till a certain point.

Also, we can see that Merge and Quick sort is not affected by the block size parameter.

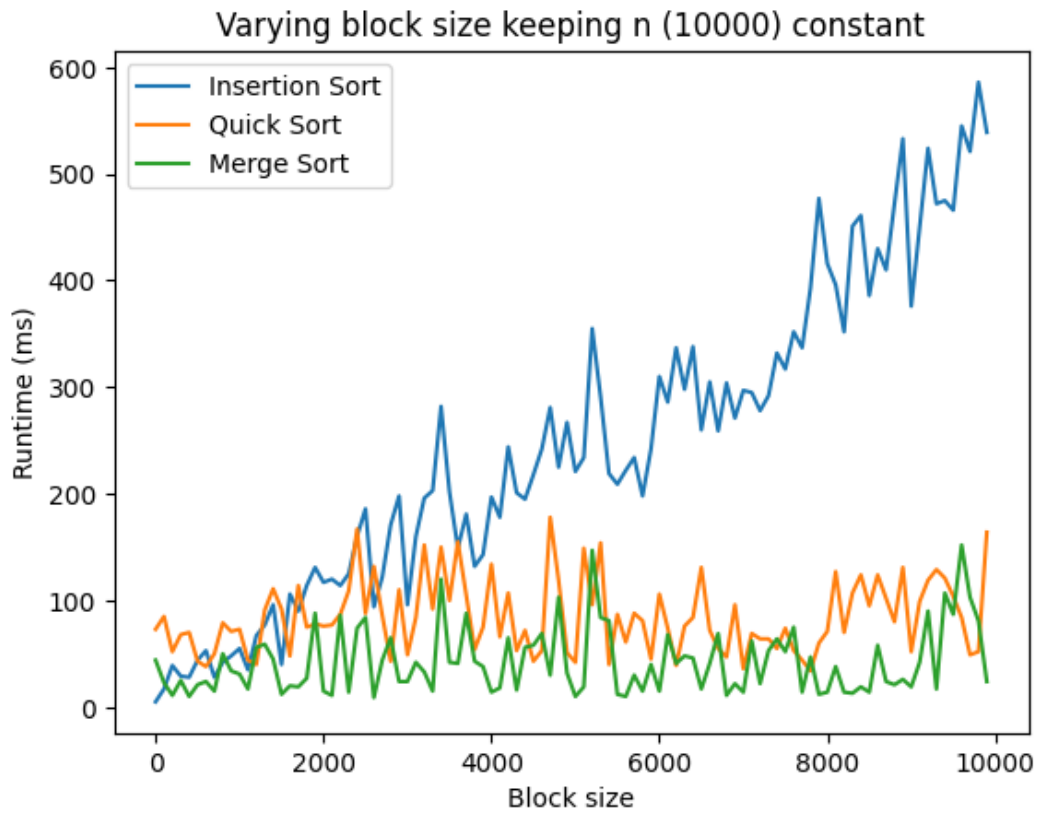


Figure 8: Varying block size for all sorts

8. Finding the k value (threshold number of inputs) for insertion and quick sort:

From the figure 9, we can see that the input size nearby 0.37 million in this case, Insertion sort was performing better or at par with Quick sort in total run time metric. This can be considered the threshold value (k) for this run of the sorting algorithms.

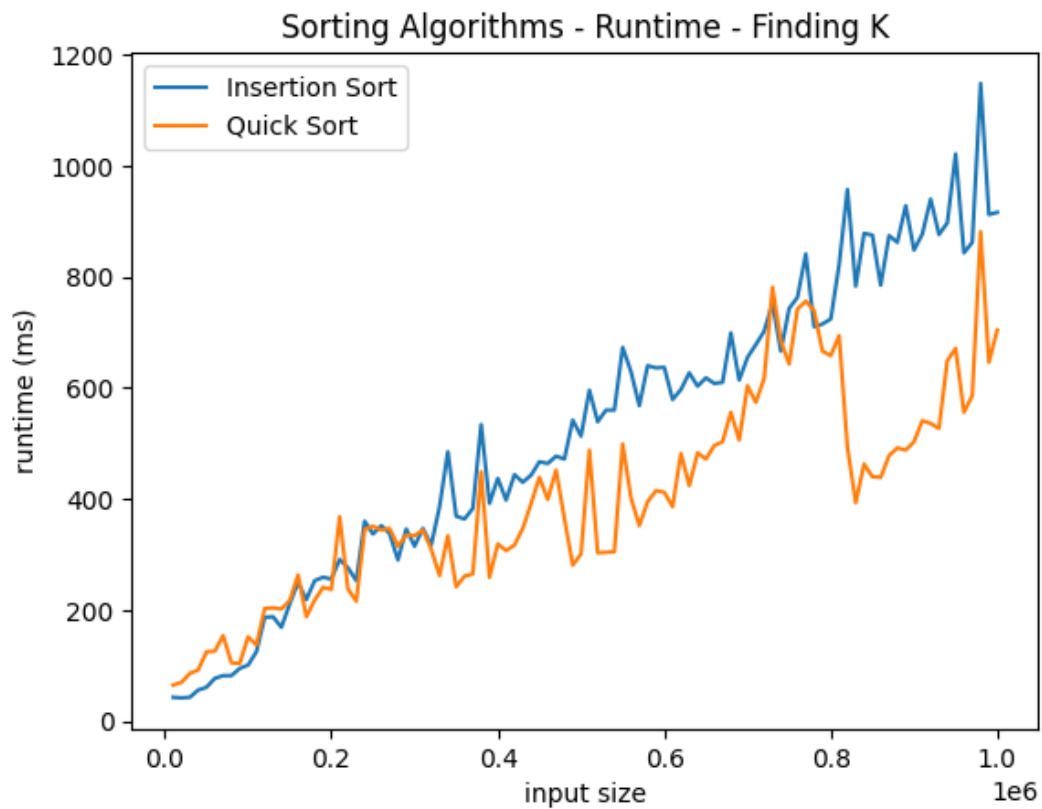


Figure 9: Finding threshold point for Insertion with Quick sort

9. Finding the k value (threshold number of inputs) for insertion and merge sort:

From the figure 10, we can see that the input size nearby 1800 in this case, Insertion sort was performing better or at par with Merge sort in total run time metric. This can be considered the threshold value (k) for this run of the sorting algorithms.

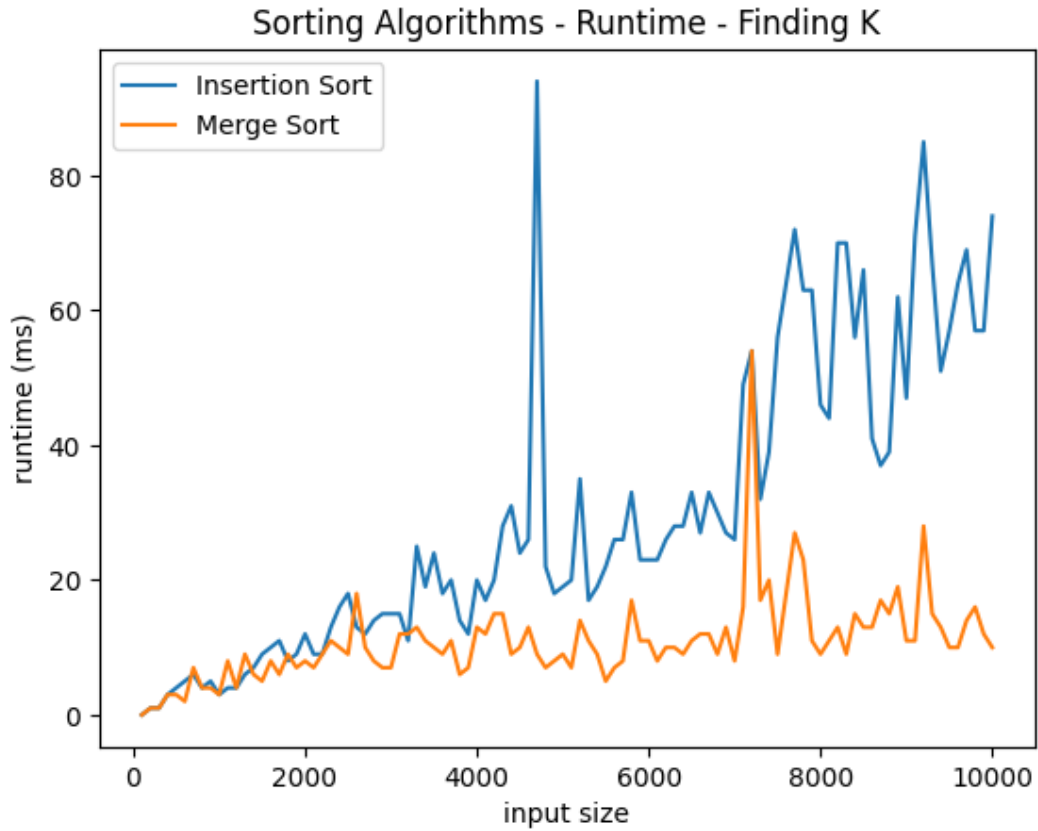


Figure 10: Finding threshold point for Insertion with Merge sort

6 Conclusions and Future work

After performing various comparisons based on a number of scenarios for the three algorithms: Insertion Sort, Merge Sort, and Quick Sort, we draw the conclusion that for a small input size, insertion sort performs better when compared to merge and quick sort. As the input size increases we notice that the merge sort rises above the other two in terms of the run time. Also, we can conclude that the number of comparisons is a better metric for comparing sorting algorithms rather than the total run time as it represents a behavior more closely to the theoretical behavior. Furthermore, while the block size shows a great impact on insertion sort, we notice that the merge and quick sort aren't affected by it. All three algorithms have been implemented in a stable fashion and this is proved by the fact that with constant input, running the algorithm yields the same number of comparisons.

As part of future work the following experiment can be performed: Some experiments can be performed to test the quicksort algorithm's efficiency by calculating the left and right partitions in parallel, as opposed to the current approach where they are calculated in serial order. The reason why they can be calculated in parallel is due to the known fact that the elements in the left partition will always be less than the elements in the right partition and thus their calculations will be independent of one another.

References

- [1] A Theoretician's Guide to the Experimental Analysis of Algorithms, *David S. Johnson*
AT&T Labs – Research
- [2] How to Present a Paper on Experimental Work with Algorithms, *Catherine C. McGeoch*,
Bernard M.E. Moret