**Experimentation & Report:**

I created two sets of text files with randomly generated sentences and randomly spaced newlines of varying line length to test each of the sorting methods.

To better gauge the scope of the project I decided to begin testing with 100 lines in a text file. This is to better capture differences and because these programs were designed to handle very large sources. The intervals for testing are 100, 250, 500, 1000, 2500, 5000 and 10000 lines for each of the text files. Each text file has lines no longer than 70 characters but can be variably shorter.

I also ran tests on files with variable line length starting with 100 characters and going up to 10000 total.

1. **Comparing Run-Time in Seconds & Comparisons between both algorithms**

The results of this experiment surprised me a bit due to prior knowledge of the abstract run times of both insertion and merge sort. I went into this test fully expecting merge sort to dominate insertion sort at the larger test intervals, however this proved not to be the case.

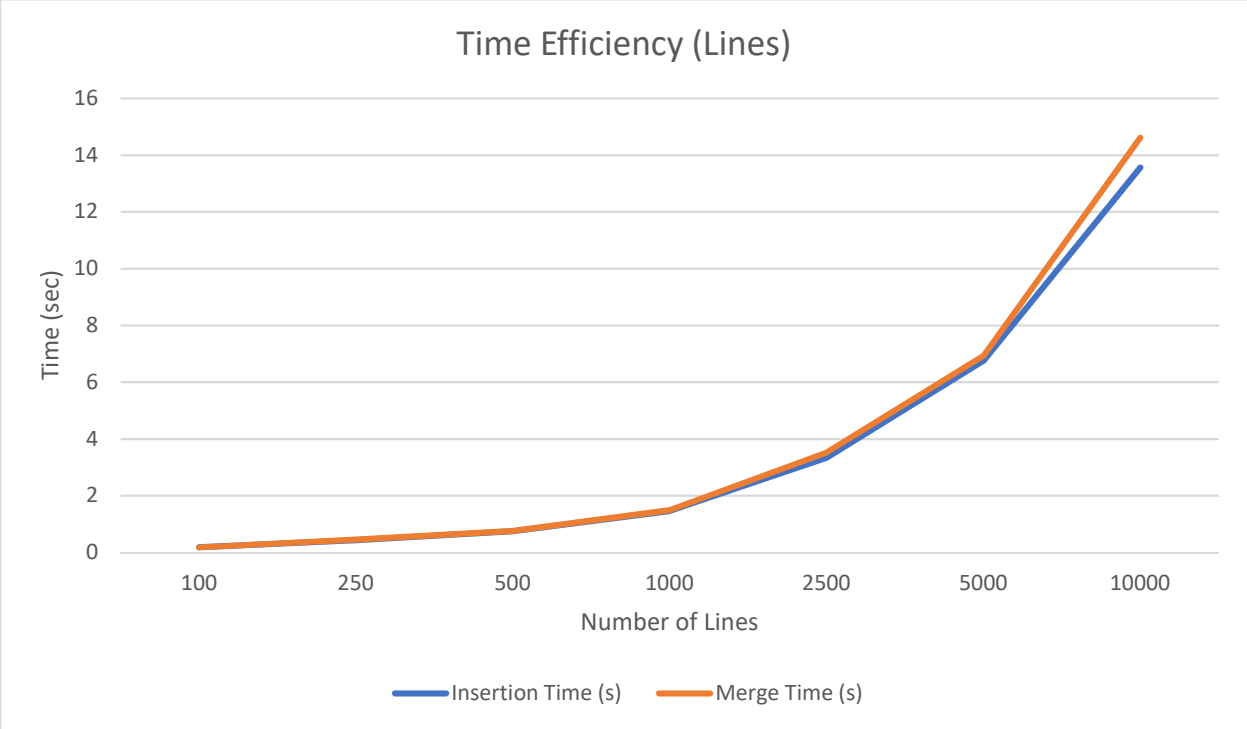


*Figure 1: Comparison of actual run time of both insertion and merge sort algorithms for number of lines (1-70 chars)*

At every single interval except for tests below 200 lines insertion sort had the purely faster run time. And once the tests were 10,000 lines long this discrepancy grew to as large as a full second difference. There are a number of possible reasons why this could be the case such as my merge sort being more inefficiently designed, potentially the Unix server I was testing on in GCC or the heavy resource usage of merge sort resulting in a slower run time.

However, merge sort does redeem itself when testing the number of comparisons between the two different sorts. In this test the abstract run times were more accurately portrayed by the data. As insertion sort had much closer to a $O(n^2)$ run time whereas merge sort was trending along closer to an

average O(nlogn) run time. This suggests that my timing process was somehow flawed and gives a more realistic assessment of both algorithms.

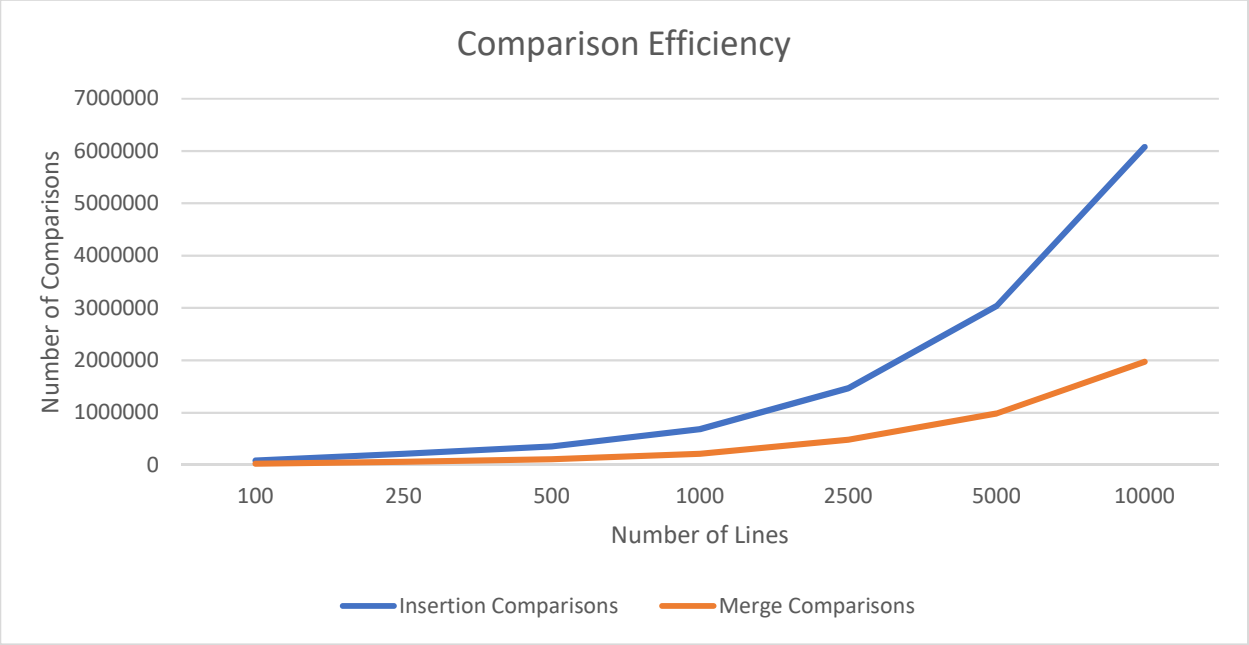


*Figure 2: Result of testing the number of comparisons for insertion and merge sort based on total number of lines*

The comparison testing went about as expected with insertion sort being more efficient for less lines and for shorter lines, but merge sort being significantly more efficient as tests got larger and longer.

For both tests the number of swaps was identical as they were sorting over the same amount of data. If there was a discrepancy in this section of data, then I would know something were wrong with the sorts I was utilizing.

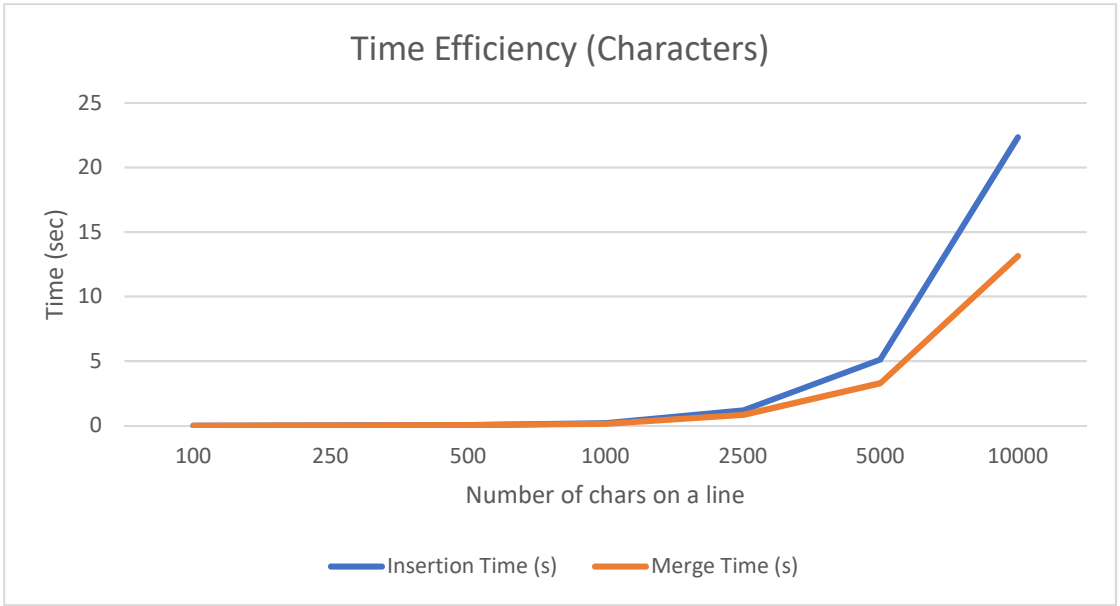


*Figure 3: Comparison of actual run time between insertion and merge sort for number of chars on a single line*

Like the number of comparisons, the time complexity abstraction is made very apparent when the number of characters in a single line increases. Merge sort excels in these tests once the number of characters in a line is about 200 long. At 200 characters in a line insertion becomes extremely inefficient at a rate much faster than that of merge sort.

2. **Comparing Compression Ratio at different line sizes**

The compression ratio for these tests was on average better than if we left the text file uncompressed, but it feels like a relatively small improvement. On average the compression ratio fluctuated between 15% and 17% depending on the number of lines with more lines resulting in a more stable fluctuation of between 16.4% and 16.7% compression.

As for the extremes worth noting that lines only containing a newline indicator on them resulted in a compression ratio of 0%. The frequency of these newlines can impact the compression ratio of the entire document because they only provide a 1-character to 1-cluster ratio. However, because of their minimal character usage this shouldn't seriously impact compression efficiency for documents with very long text lines.

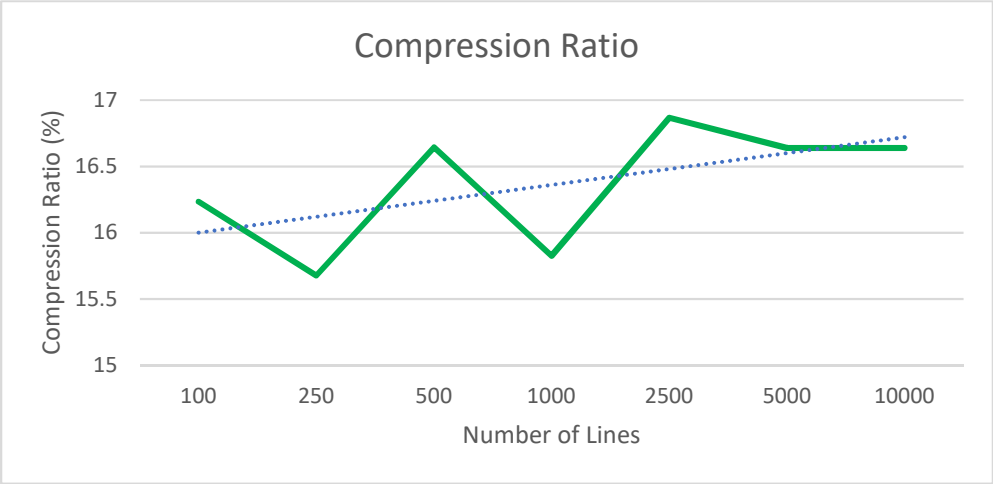


Figure 4: Average Compression Ratio based on the total number of lines in the text file; trend line shows relative increase until 16.7%

Ideally long lines with many similar characters produce the best compressible outcomes. For example, there was a line in one of the documents that contained 28 spaces and a single word with repeating characters. The compression ratio of this line was 81.5%. This is an anomaly and as the average suggests, this is a very rare occurrence.

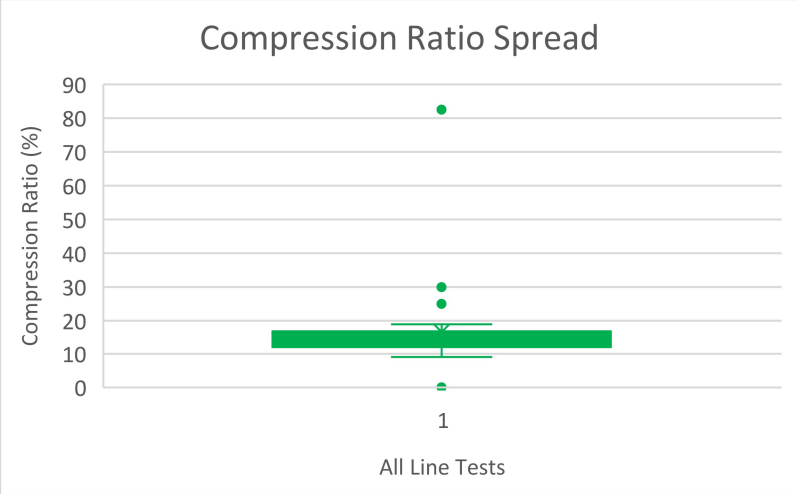


Figure 5: Chart showing the relative spread of compression ratios for each line of the text files; Max = 81.5%, Min = 0%, Average: 16.6%

### 3. Time to Decode Each Encoded Input

Similar to the encryption results, the time efficiency of both sorting algorithms in terms of total time was about the same for all line lengths. However, of note is that insertion sort was only more efficient until 1000 lines of text were sorted. Then merge sort is constantly just a little bit more efficient for the duration of the testing. The difference between both is very small, not much more than a fraction of a second, but as the text files get larger it should be expected that merge sort continues to fair better than insertion sort.

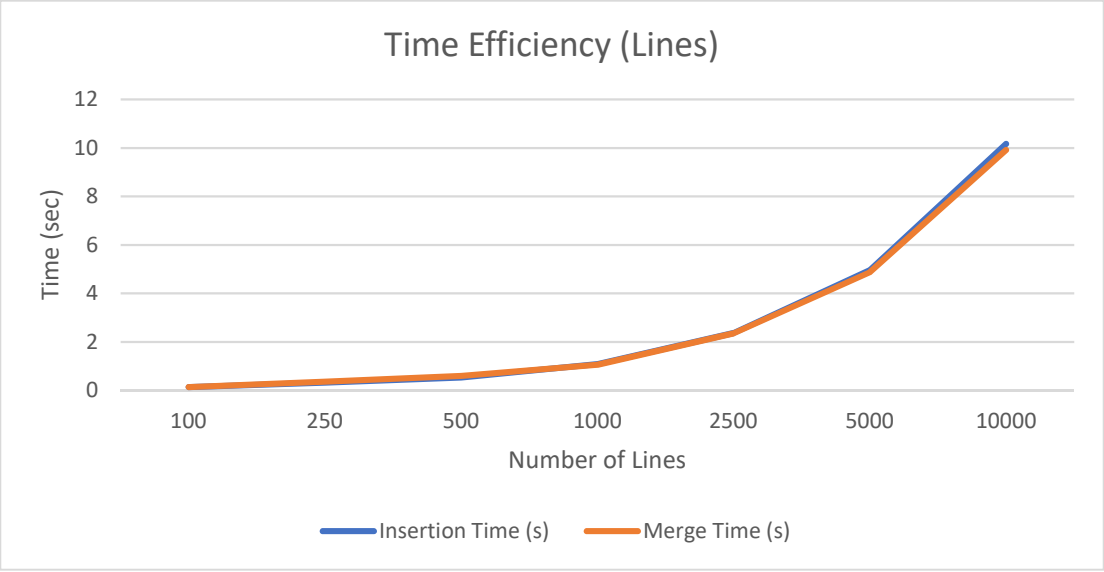


*Figure 6: Comparison of actual run time of both insertion and merge sort algorithms for number of lines (1-70 chars)*

So what changed between the sorts for encryption and decryption to cause merge sort to be more efficient at decrypting the text files? Probably the largest discrepancy can be attributed to the usage of 2D string arrays in the encryption code which significantly increased the amount of memory needed by merge sort to perform its role, whereas the decryption code only used a pair of strings to decode the

input. This change is also likely why decrypting the text file took as much as 5 seconds less total time to run.

Just like in the encryption observations, the merge sort handily bested insertion sort when it came to total element comparisons with merge sort being better again somewhere around 50 lines of text in and continuing to increase the gap. This significant difference can be attributed to the run time abstraction mentioned before with insertion sort having $O(n^2)$ and merge sort having a lower O(nlogn) time complexity.

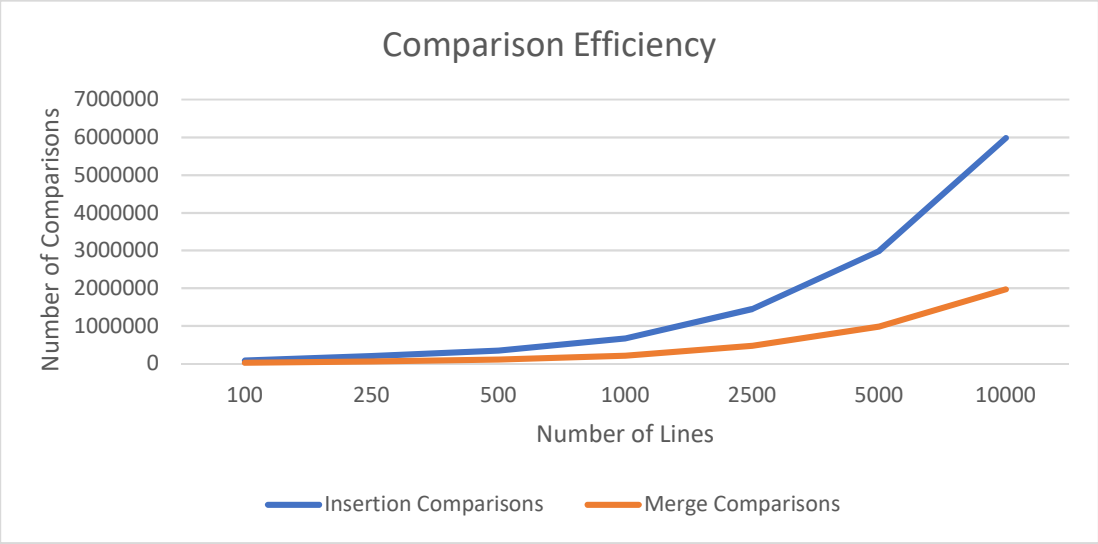


Figure 7: *Result of testing the number of comparisons for insertion and merge sort based on total number of lines*

Very surprising are the results from the number of characters per line. I expected these results to continue displaying the time complexity abstraction as visualized earlier in the encryption algorithm, however that was not the case. Merge Sort and Insertion sort were almost equal for all line sizes when it came to decryption. The likely conclusion why this occurred is due to the base setup for the sorts contributing more to run time than properties of the sorts themselves.

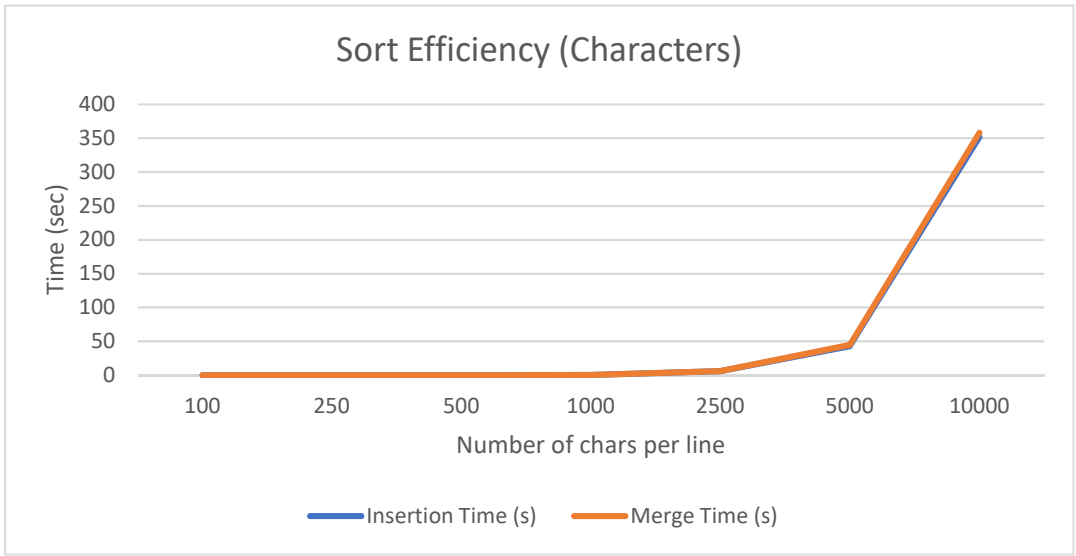


*Figure 8: Comparison of actual run time between insertion and merge sort for number of chars on a single line*

4. **Encoding Multiple Lines at a Time**

Depending on how the encryption program is written this could significantly impact the compression ratio to an extent. As the program is right now it only encrypts text files line by line, but if it were optimized to handle more lines at once this could cause the compression ratio to improve.

This improvement could be attributed to increasing the potential number of clusters that are found in each group of lines which would improve compression efficiency more than simply analyzing a single line at a time.

However, the time efficiency of the program seems as though it would suffer from reading in much larger data sets at once. In the case of insertion sort, which has a time efficiency of $O(n^2)$ this would be devastating to its run time. As the number of n elements in the sort grows it gets worse by a power of 2 each time. Merge sort would be better able to handle multiple line inputs with its time efficiency of O(nlogn), but at very large n values which would result from handling multiple lines at once it would also slow down significantly. In this instance taking in single lines for the sorts would be beneficial over attempting to sort multiple lines at once.

Overall, I think that increasing the number of lines to be sorted would result in a net loss of time efficiency rather than simply iterating through the text files line by line.