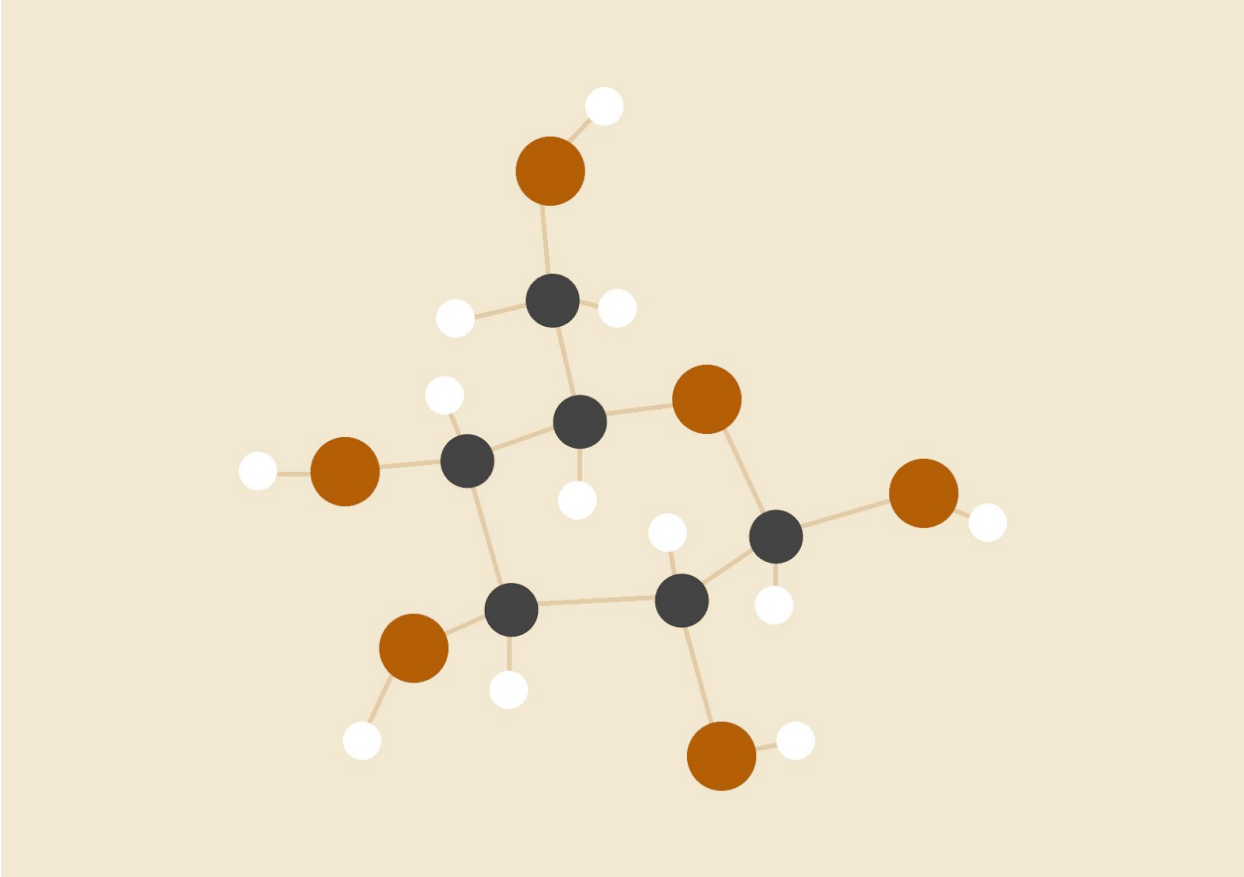
PROJECT #1

Experimentation



Azaria Fowler

October 2nd, 2019
CSE 310 TTh 10:30am

EXPERIMENTATION

Impacts on Compression Ratio → Average Compression Ratio

Experimentation Description

In order to compute the average compression ratio, I decided to analyze the relationships between the compression ratio to three distinct characteristics of an input line. Whichever characteristic has the most significant impact on the compression ratio, I would generate a large data set pertaining to the variability of the characteristic and compute the average on that.

Since the compression ratio only pertains to the encoding process, I focused on inputs to the encode program. I focused on these three characteristics of an input:

1. **Quantity of characters:** How does the length of line (number of characters in a string) affect the compression ratio?
2. **Frequency of duplicate characters:** How does the number of duplicate characters affect the compression ratio? (i.e. *race**car***, *ba**b**y*, etc.)
3. **Frequency of consecutive duplicate characters:** How does the number of duplicate characters that appear consecutively affect the compression ratio? (i.e. bu*tt*erfly, l*oo*p, *ll*ama, etc.)

Compression Ratio

$$\frac{number\ of\ characters - number\ of\ clusters}{number\ of\ characters} \times 100\%$$

1. Quantity of Characters

Since the english language tends to have higher and lower frequencies in certain letters, I generated random strings (not real words) for true randomness, thus only focusing on the quantity of characters. I also changed each string by increasing length sizes, so that I could better see the correlation between length and compression ratio.

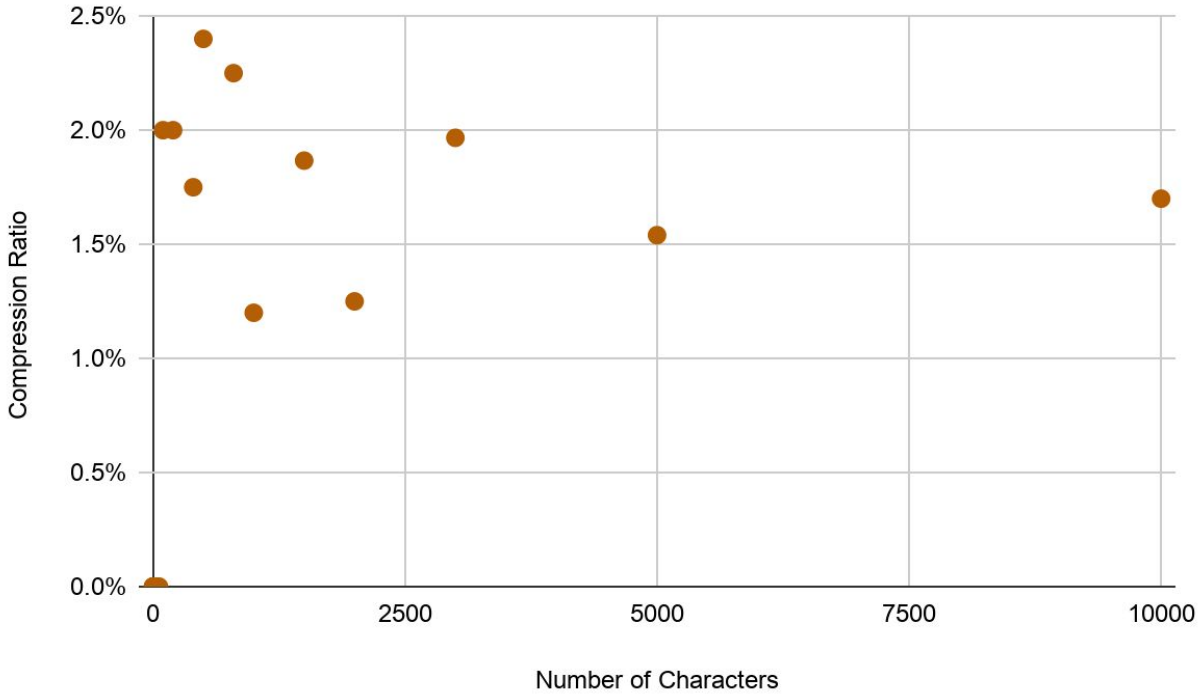
Example Sample Data:

- a
- oID43
- Ai3ReUr27aia
- t8kMT0ROtE8FTGn8cSRonOw6wg3fLiuybFr2dPoU4Ohpvfx6JtVHuBBdMbYN

I developed a small program that would run within encode on these strings of variable length and printed out the original string's number of characters, encoded line's number of clusters, and encoded line's compression ratio.

Results

The compression ratio seemed to be around 0% for strings that were smaller than 100 characters. Once we had about 100 characters, the compression ratio bounced between 1.5% and 2.5%. The average compression ratio of lines of variable character quantities was about 1.045%.



I also noticed that there wasn't a really direct correlation between the length of the characters and the compression ratio, as strings will 500 characters would have higher compression ratios than ones with 1000 characters. This suggests that **length alone does not directly correlate to the compression ratio**.

2. Frequency of Duplicate Characters

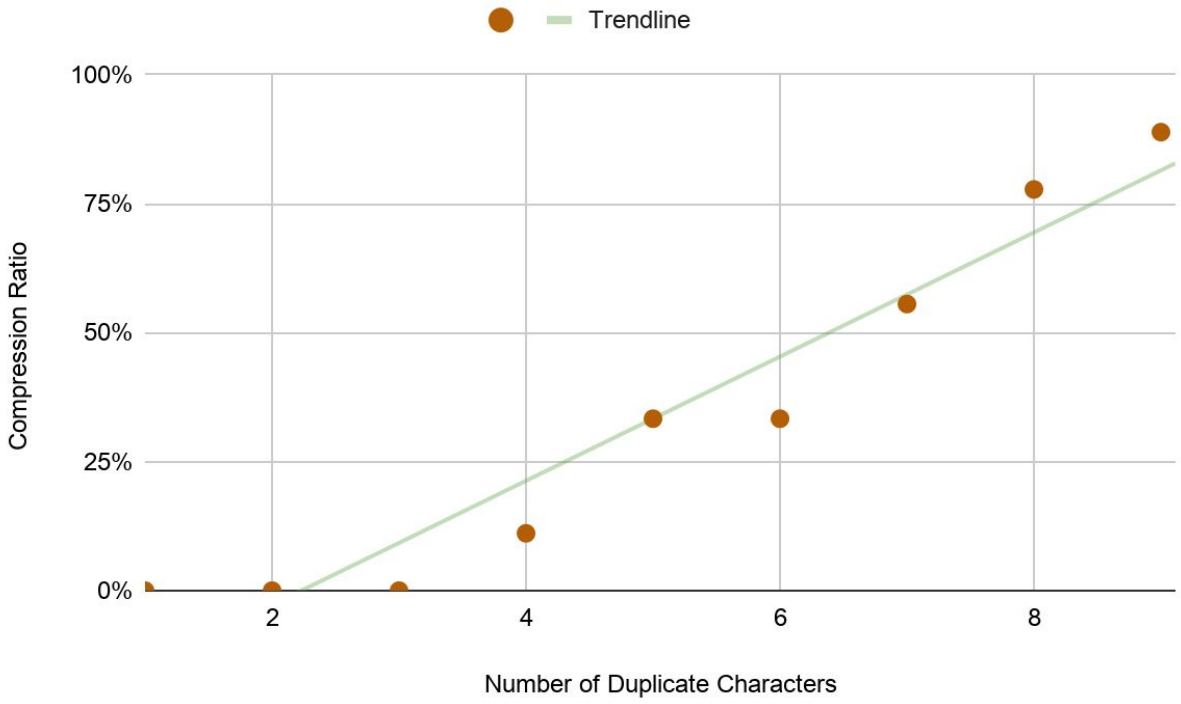
In order to accurately measure the effect of duplicate characters, I decided to run nine instances of the string: *abcefghi*. Each instance had an increasing number of the letter 'a' while not changing the length of the string. I tried to distribute the a's evenly, so that the consecutive characteristic was minimized.

Sample Inputs:

- abcdefghi
- abcaefghi
- abcaefahi
- aacaefgai
- abahaeaga
- aacaafaga
- aaaadacaa
- aaaacaaaa
- aaaaaaaaa

Results

The compression ratio seemed very dependent on the number of duplicate characters in a line. As the number of duplicate characters increased, the compression ratio also increased. This suggests that **the relationship between the compression ratio and the number of duplicate characters is linear.** The average compression ratio was about 33%.



The linear relationship between the compression ratio and the number of duplicate characters was probably exaggerated due to the fact that the length of the test strings were all the same. Therefore, the ratio between the duplicated character to the other characters started to decrease. Also, around the mark of 6 duplicate characters, I started to run into the problems of having the duplicate characters in consecutive order.

3. Frequency of Consecutive Characters

In order to analyze the effects of consecutive characters on the compression string, I decided to approach it in two ways. The first set of inputs were nine instances of the string *abcdefghi* where I added another 'a' in between the next character (see Sample Data A). The second set of inputs were nine instances of the same string where the next character was duplicated (see Sample Data B). With the previous information that the length of the string made little to no effect on the compression ratio, we can assure ourselves that both data sets will show the relationship between the consecutive characters and the compression ratio.

Sample Data A:

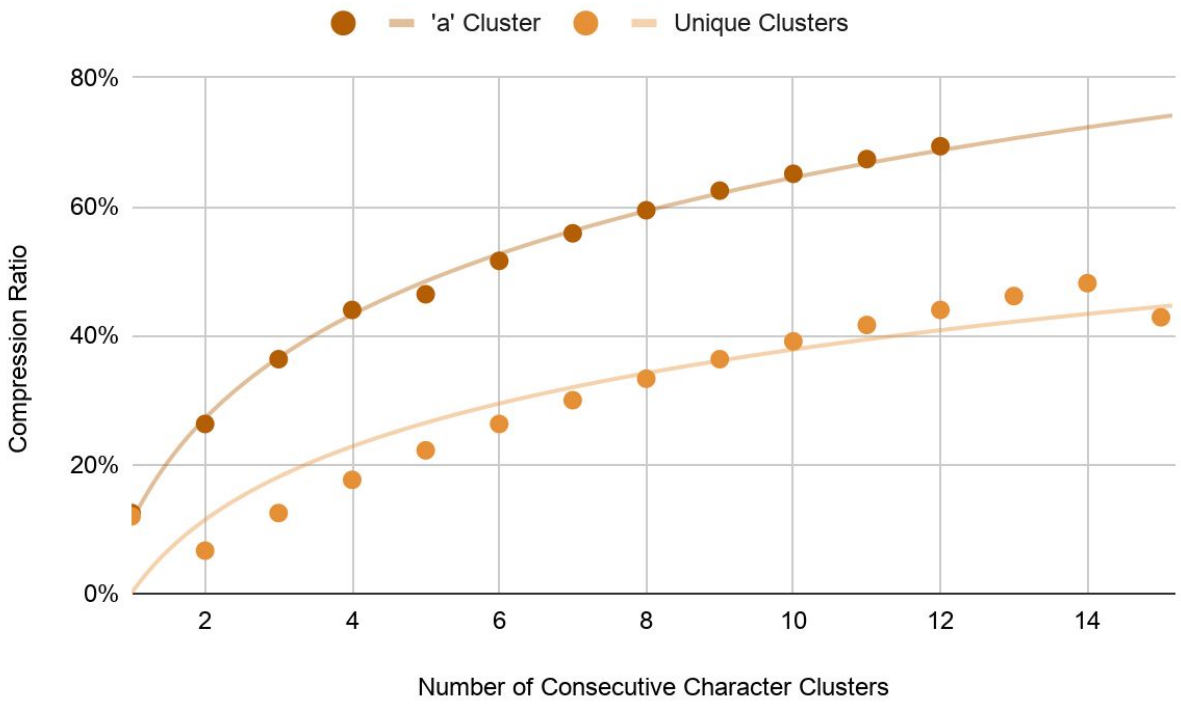
- abcdefghijklmn
- aaabcdefghijklmn
- aaabaaacdefghijklmn
- aaabaaacaaadefghijklmn
- aaabaaacaaadaaaefghijklmn
- ...

Sample Data B:

- aabcdefghijklmn
- aabbcdefghijklmn
- aabbccdefghijklmn
- aabbccddefghijklmn
- aabbccddeefghijklmn
- ...

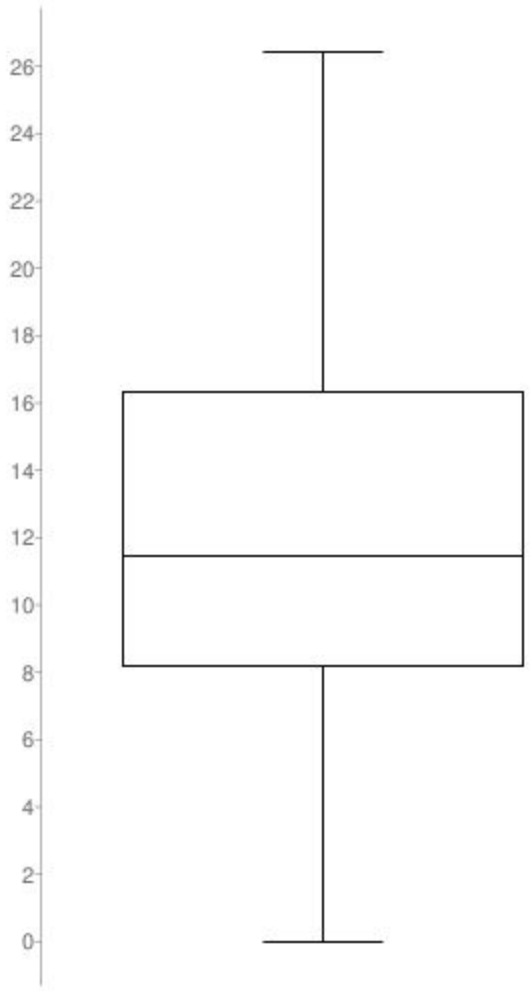
Results

The frequency of consecutive characters had an interesting relationship with the compression ratio. As the frequency of the same character that appeared consecutively rose, the compression ratio also rose. This also happened with when the frequency of any character that appeared consecutively rose. However, when there were about eight consecutive character clusters, the change in compression ratio started to decrease. This suggested that **the relationship between the frequency of consecutive characters and the compression ratio is logarithmic**. The average for 'a' cluster data set was 48% and the average for unique clusters data set was 28%.



The levelling out to a certain compression ratio suggested that the number of consecutive character clusters was most likely due to the ratio of quantity of letters to consecutive character clusters. As the clusters increased the number of characters also had to increase. If we kept the same number of characters, then we would essentially be testing the frequency of duplicate characters (the previous test) again.

Experimentation Results



The characteristic that had the most significant impact on the compression ratio was the amount of duplicate characters in a string. This makes sense because a cluster is essentially bringing duplicate characters together into one character. Thereby, offsetting the ratio between the total number of characters and the number of clusters.

Another thing I recognized is that the actual compression ratio's numerical value only reveals the *ratio* of duplicate characters to the number of clusters. Therefore, the average compression ratio only provides a unique average for the given data set. Basically, if one data set had an average compression ratio of 80% but another had 10%, then it does not reveal much about long the lines are or how many duplicate letters appear consecutively. It only tells you how close the length of the line was to the number of duplicate characters there was for every line in the data set.

Therefore, I decided to compute the average compression ratio on a set of English sentences.

According to the English language, certain letters are used at a higher frequency than others. For example, the letter E is the most commonly used letter while letters X and Z are the least commonly used. Therefore, a data set containing English sentences would be a perfect set that generates the average compression ratio.

In this data set, I generated 100 English sentences in varying lengths. In the box and whiskers chart on the left, you can see the distribution of compression ratios.

The average compression ratio turned out to be 12.13%.

The minimum was 0% and the maximum was 26.41%. The standard deviation was 5.77%.

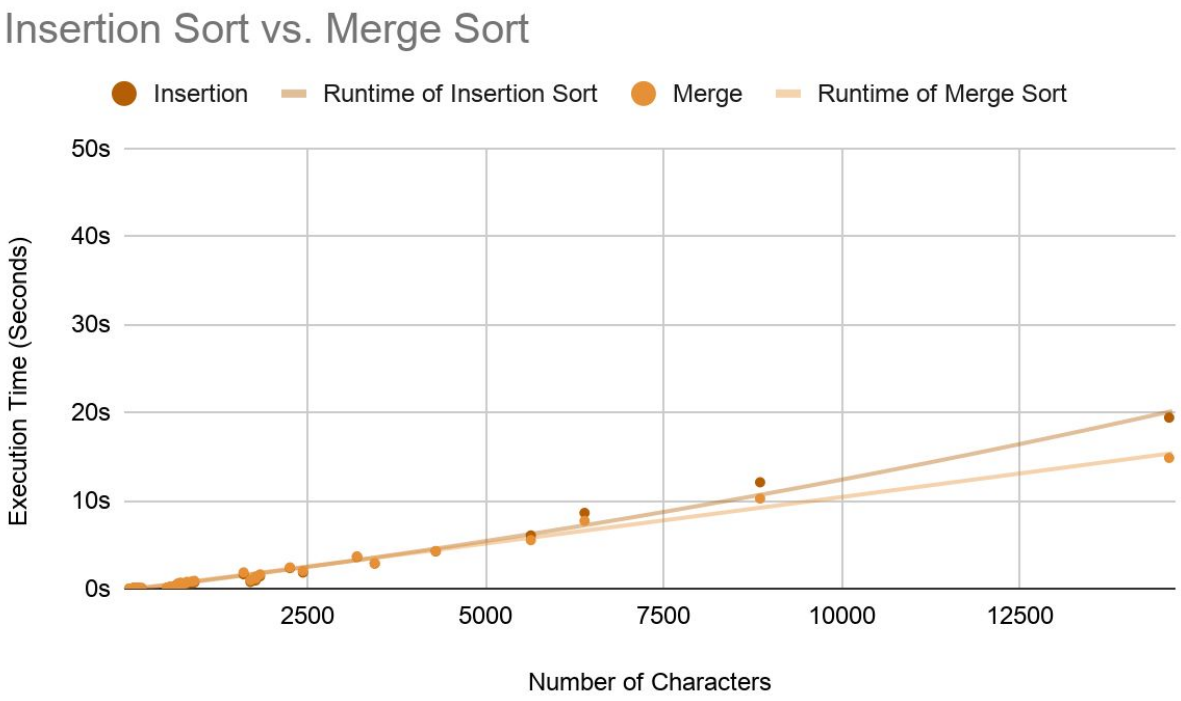
As you can see, the average compression ratio is very low. This means that the length of the sentences were significantly larger than the frequency of duplicate letters. This makes sense as while the letter E is the most frequent letter, the distribution of character frequencies are still relatively equal. Also, we only have 26 letters in the alphabet, so once a sentence becomes longer than 26 characters, the likelihood of duplicate letters certainly increases, thus providing a higher compression ratio.

Effects of Encoding Runtime with Insertion and Mergesort

Experimentation Description

In order to see the effect of the sorting algorithm on the runtime of the encode function, I decided to provide the same data set to the encode function that ran insertion and then mergesort. The data set contained 142 English sentences with lengths ranging from 1 to 14, 596 characters. In other words, n is the length of the English sentence. I chose English sentences as the variability of duplicate characters would be relatively similar, decreasing the impact of duplicate characters to the sorting algorithms. I also ran a simple C++ program that calculated the amount of microseconds each line inputted into the encode function took to encode and print.

Experimentation Results



As you can see in the scatterplot graph above, **the choice of sorting algorithm seemed to have no effect on the growth of runtime of the program**. At first, I thought this was a mistake, as mergesort's algorithm is O(nlogn) and insertion sort's algorithm is $O(n^2)$ for n characters in a line. However, when looking at my code, I realized the choice of algorithm did affect the runtime of the encode program, yet other parts of my code simply overpowered it.

For example, in my encode, before I run the sorting method, I have the following code:

```
for (int i = 0; i < length; i++) // O(n) {

     int j = i; // Index in line
     int k = 0; // Index in shiftedLine

     while (line[j] != '\0') // O(n) {
          // Copy line[j] into shiftedLine[k]
     }

     j = 0;

     while (j < i) //O(n) {
          // Copy line[j] into shiftedLine[k]
     }
     ...
}
```

This excerpt is populating the vector cyclicShifts of all the cyclic shifts of the string. Since each line has n characters, the number of cyclic shifts will also be n. For each index in the line, it copies the letters in the original line from the index to the end of the line into the shiftedLine with one while loop. Then, it copies letters from the beginning of the line to the index with another while loop.

The for loop iterates onto every character in the line, so it is O(n).

The while loop is slightly more complicated. Since the shiftedLine is k characters long, then in the first loop we will iterate up to k-j of line for each iteration of the for loop. Then, the second while loop will iterate up to j times. That means, both the while loops will run k-j+j = k times, which is also n. So, both the while loops together run O(n).

Together, the runtime of the entire for loop is O(n) * O(n) which is $O(n^2)$. That is already bigger than mergesort, and therefore "overpowers" the mergesort algorithm to be $O(n^2)$. It also is the same as insertion sort, and since insertion sort runs sequentially in relation to the for loop, then it stays about $O(n^2)$. In the graph, however, you can see that insertion sort runs longer than merge sort by an increasing constant, which suggests that perhaps at large enough values, merge sort will run significantly shorter than insertion sort.

The running time of encode is $O(n^2)$.

How the Length of an Encoded Input Effects Decode

Experimentation Description

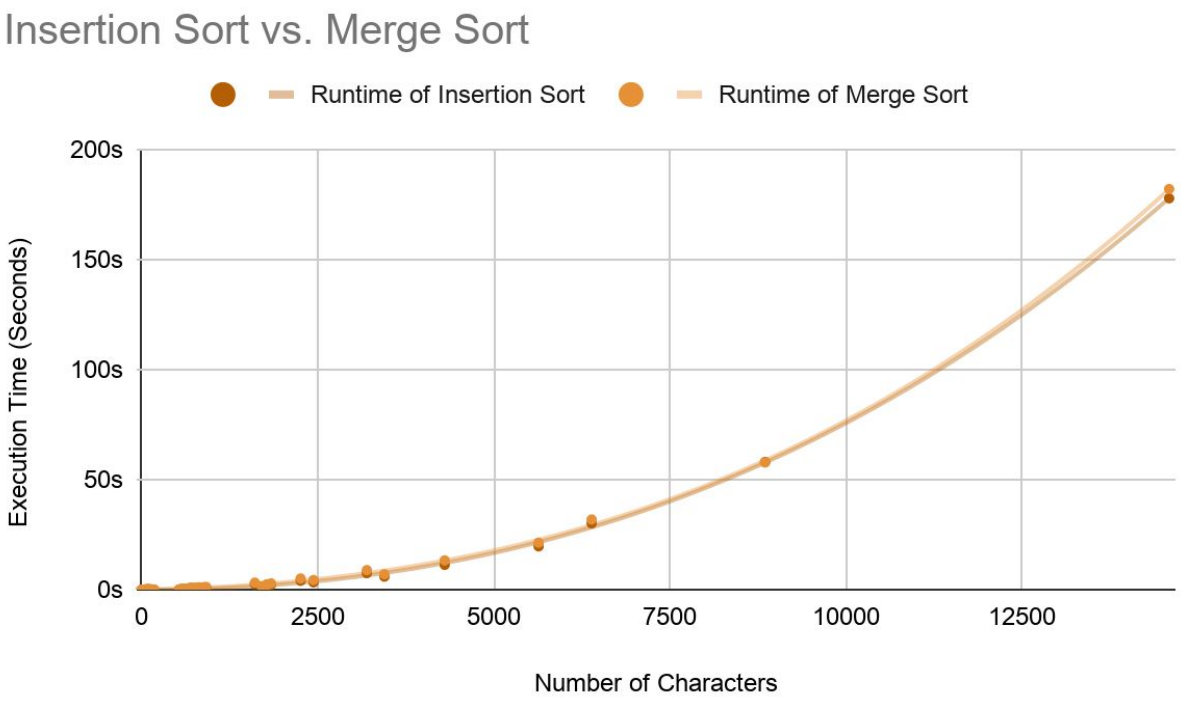
In order to see the effect of the sorting algorithm on the runtime of the decode function, I decided to provide the same data set to the decode function that ran insertion and then mergesort. The data set contained 142 *encoded* English sentences with lengths ranging from 2 to 14,596 characters. Since the decode function depends on the length of the encoded sentences, I will be comparing the run time of decode with respect to

the encoded line length. In other words, n is the length of the encoded line. (At least, for now).

I decided to not use clusters because it would be essentially half the length of the encoded line. Therefore, the graph would simply be skewed. I also decided to not include the index in the calculation of the length of an encoded line, because there will always be one index that is read at constant time. Therefore, it would not contribute significantly to the growth rate.

I chose English sentences as the variability of duplicate characters would be relatively similar, decreasing the impact of duplicate characters to the sorting algorithms. I also ran a simple C++ program that calculated the amount of microseconds each line inputted into the encode function took to encode and print.

Experimentation Results



As you can see in the scatterplot graph above, **the choice of sorting algorithm seemed to have no effect on the growth of runtime of the program**. This makes sense as other parts of my code has a runtime bigger than $O(n^2)$, the upper bound of insertion sort. However, what is interesting is that the length of the encoded line also had no effect on the growth of the runtime. In fact, it was actually the length of the original string that really determined the growth of the runtime. Therefore in this analysis, n will now be the length of the original string.

After the encoded line is parsed into last, the characters of the original string are now decompressed. As last consists of the last letter in each cyclic shift, the length of last is also the length of the original string. After last is sorted and stored as first, the following code runs:

```
bool inNext(int next[], int value, int size) {
    for (int i = 0; i < size; i++) {
        if (next[i] == value) {
            return true;
        }
    }
    return false;
}

...

for (i = 0; i < first.size(); i++) {
     for (j = 0; j < last.size(); j++) {
          if (last[j] == first[i] && !inNext(next, j, first.size())) {
               next[i] = j;
               break;
          }
     }
}
```

If you look at this excerpt of my code in decode, you can see where I generate the next array.

Basically, for each element in first, I check if that element is in last. If that element is in the next array already, you keep going through last. In the worst case, first would have to be in next's recently populated index. That would mean if next had a length of n characters in the original string, the first for loop would have to run n times. For each iteration, the second for loop would also have to run (in the worst case) n times. Lastly, for each iteration in the second for loop, we run inNext() which iterates through the length of next to check if the spot in next is already taken. Since the length of next is the length of last, it (in the worst case) runs n times.

Essentially, three nested loops of a runtime of O(n) result into an overall $O(n^3)$. Since this runs sequentially after the sorting, the $O(n^3)$ overpowers both insertion and mergesort's runtime. In the graph, however, you can see that merge sort runs longer than insertion sort by a small constant. This is because in my merge sort implementation, I traversed through last to generate a vector version of last before I invoked mergesort, and then iterated through the sorted vector version to store in first. These two extra iterations possibly caused merge sort to have a runtime of O(n + nlogn + n) = O(2n+nlogn) which at small values run longer than insertion sort's $O(n^2)$.

The Impact of the Number of Lines on Compression Ratio

Experimentation Description

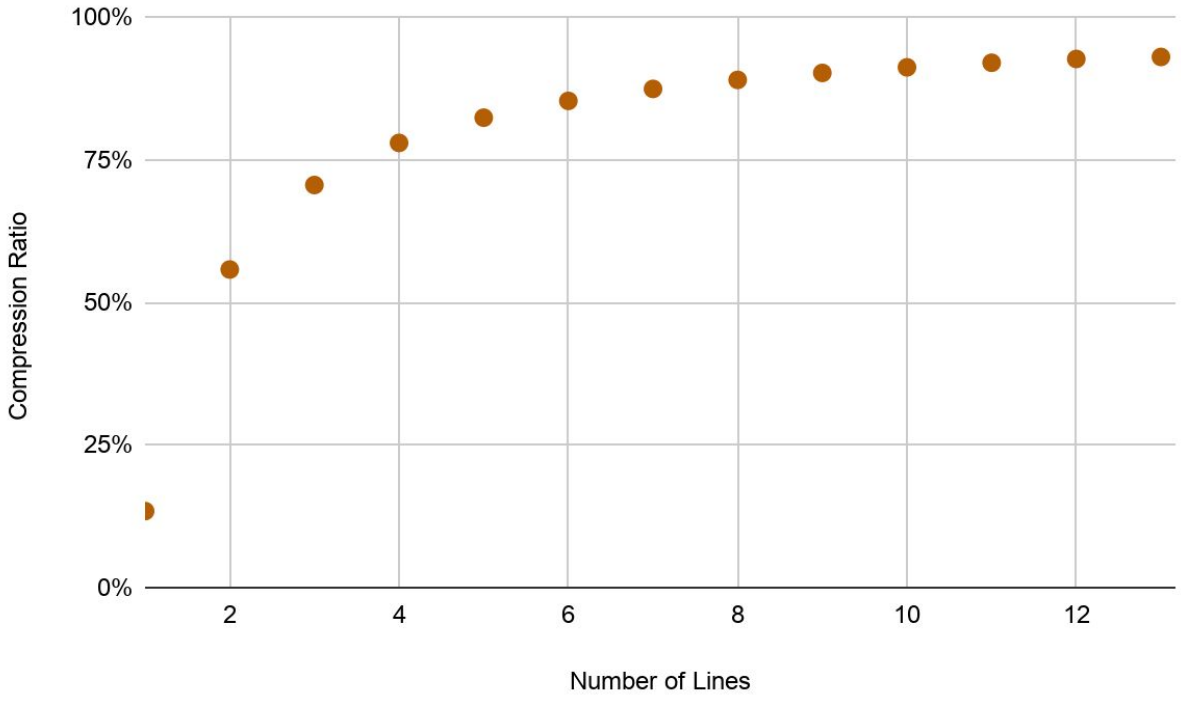
In order to see the impact of the number of lines on the compression ratio, I decided to run encode on a data set containing one unique sentence:

> This is the last random sentence I will be writing and I am going to stop mid-sent

As the number of lines increased, I just concatenated the same sentence. This is to eliminate the variability

of different letter frequencies and distributions in a different sentence on the compression ratio, and to thereby only focus on the increasing number of 'lines'.

Experimentation Results



Clearly, as you can see in the scatterplot above, the number of lines had the same effect on the compression ratio as the relationship between the frequency of consecutive characters and the compression ratio. This makes sense as the increasing of lines is basically the increasing of the length of the line. As the length increases, the number of limited characters being repeated also increases. Since they both increase, the ratio between the length of the line and the number of duplicate characters grows logarithmic.

This suggests that if the encoding took in more than one line, the compression ratio would be higher than if it took one line. However, if you kept increasing the number of lines, the change in compression ratio would decrease.