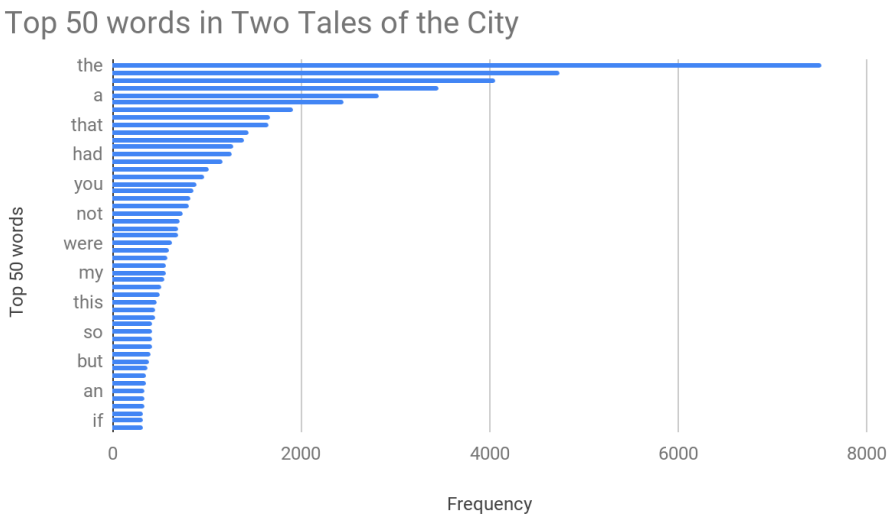
Henry Chang Lab6 CMSC201

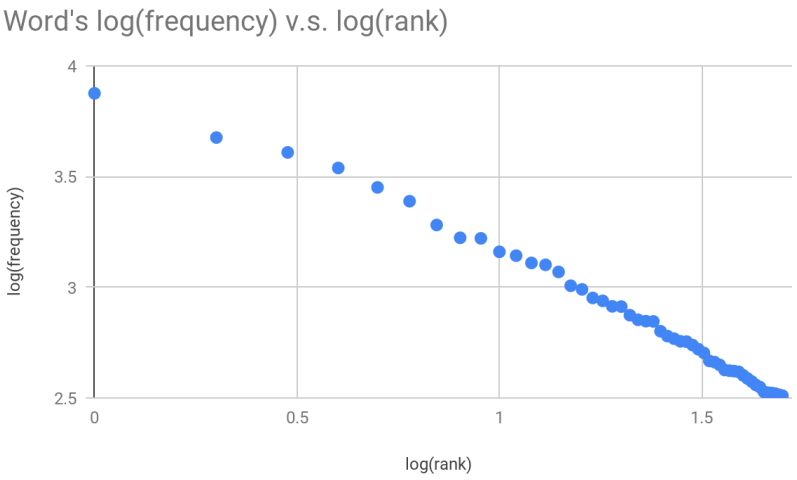
In Part I, I create an analysis on the text file of Two Tales of the City downloaded from the Gutenberg Project, to examine whether Zipf's Law exists in this dataset. After making sure that the phenomenon described by Zipf's Law that the frequency of words are proportional to the inverse of their ranks exists in my dataset, we compare in the context of Zipf's Law, the performance of counting word frequencies with Symbol Tables implemented based on Sequential Search, which orders keys by their first entries, and Self-Organizing Search, which puts the most recently inserted key into the front of the symbol table (Part II).

Part I: Zipf's Law

A quick view of the frequency of the 50 most frequently appeared words in my dataset shows that the trend that the frequency of a word in the dataset inversely proportional to its rank.



A closer analysis with the log-log plot reveals that data points follow a trend that is approximately a straight line. This shows that Zipf's Law applies to this dataset, at least for the top 50 ranked words.



Part II: Compare SequentialSearchST & SelfOrganizingST
To compare the performance of SelfOrganizingST and SequentialSearchST, I collect their runtime for counting the frequency of words for six times and average the results. I get:

SequentialSearchST: 22.3s
SelfOrganizingST: 22.6s

As the dataset I used follows Zipf's Law, and some commonly used words (e.g. the, and, he and she) tend to appear a lot more frequent than any other word, I expect SelfOrganizingST to run faster than SequentialSearchST. However, the result showed that there is little difference between their performance on speed of counting word frequencies. This may be due to the incompleteness of the strip function of my text reader, which will put "and" and ",and" as two different words. Thus, the overall appearances of some specific combination of word that only appears a few times would perhaps take a huge percentage of the overall appearances. They would clog the front of the table so that even the counting the frequency of the most frequently appeared word tends to take time as well. Also, their insertion would usually loop through the entire search table for the contains() method. This would also generate considerable runtime.

Bonus

Find a specific problem or application where a symbol table based on:
1)Binary-search is better than self-organizing search
When the frequency of words are uniformly distributed, for example { keys = [0,1,2,3,4,5,6,7,8,9,10]; value(frequencies) = [1,1,1,1,1,1,1,1,1,1,1] }, self-organizing search would work similar to a linear search that require an average of ~N/2 runtime whereas binary search would only require a lesser ~lgN runtime, for N as the size of the symbol table. In the real world, counting the frequencies of numbers as a result of rolling a dice would generate this kind of uniform distribution, and in this case, Binary-search is better than self-organizing search.

2)Self-organizing search is better than binary search
When the most frequently appeared word takes a large percentage of the overall appearances , for example { keys = [0,1,2,3,4,5,6,7,8,9,10]; value(frequencies) = [99,1,1,1,1,1,1,1,1,1,1] }, self-organizing search would only require ~1 runtime whereas binary search would still require a larger ~lgN runtime, for N as the size of the symbol table. In reality, counting the frequencies of names appearing in media would be a case in which self-organizing search is better than binary search because famous people's names(e.g. Trump) appear extremely more frequently than any random person.