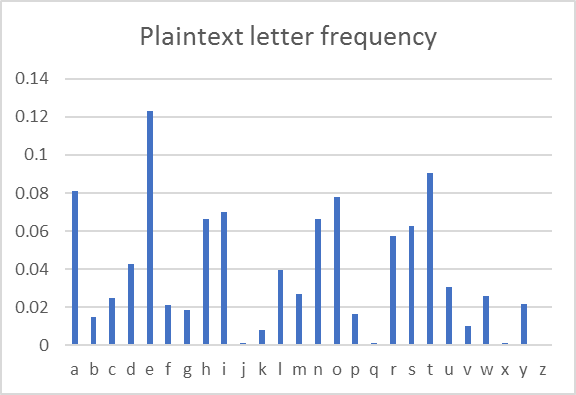
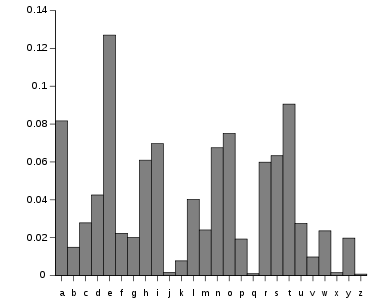
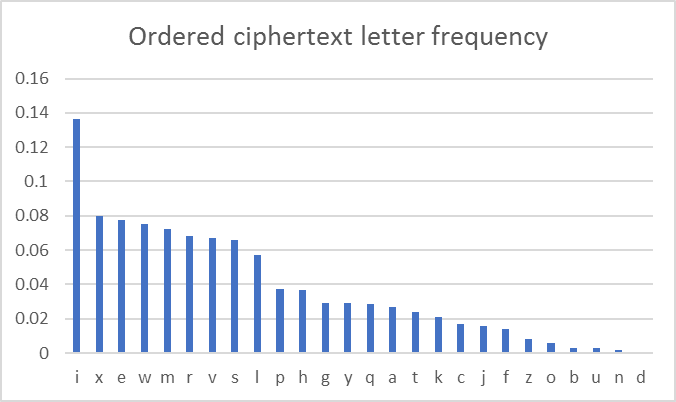
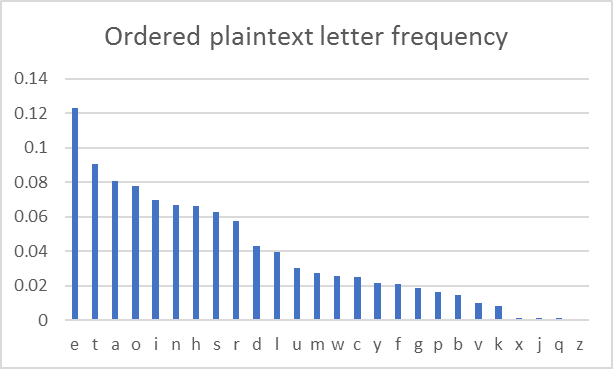
CSC3422 Cryptography Exercise 1 Report

In this report I will explain how I performed a frequency analysis on English text, and how I used that frequency analysis to cryptanalyze a cipher text. It will include a detailed explanation of how my Java program functions, and a discussion on the results, such as how my frequency analysis compares with commonly known results of the English alphabet.

Firstly, I will describe the process of my Java program. I initially created two maps, both of which held a character as the key, and an integer as the value. The 26 letters in these maps will be paired with the number of their occurrences in the plaintext and ciphertext, respectively, using my LetterCount() method. This method passes a file, and uses a loop to loop through each line of the text. Each letter in each line is then passed as a character, and each time a character is found the characters corresponding value in the map is incremented by one. The results of these maps are then output into a text file for ease of reading. I then created a Frequency() method that looped through these maps adding each value to the next to find the total number of letters. Once the total was found It would loop again, but this time dividing each value by the total to find the frequency of each letters occurrence in the texts. These values are stored in a map with its corresponding letter and printed out into a text file. While I did not use this method in the rest of my program, the results were useful, as they allowed me to compare them with the relative frequency of the English language.



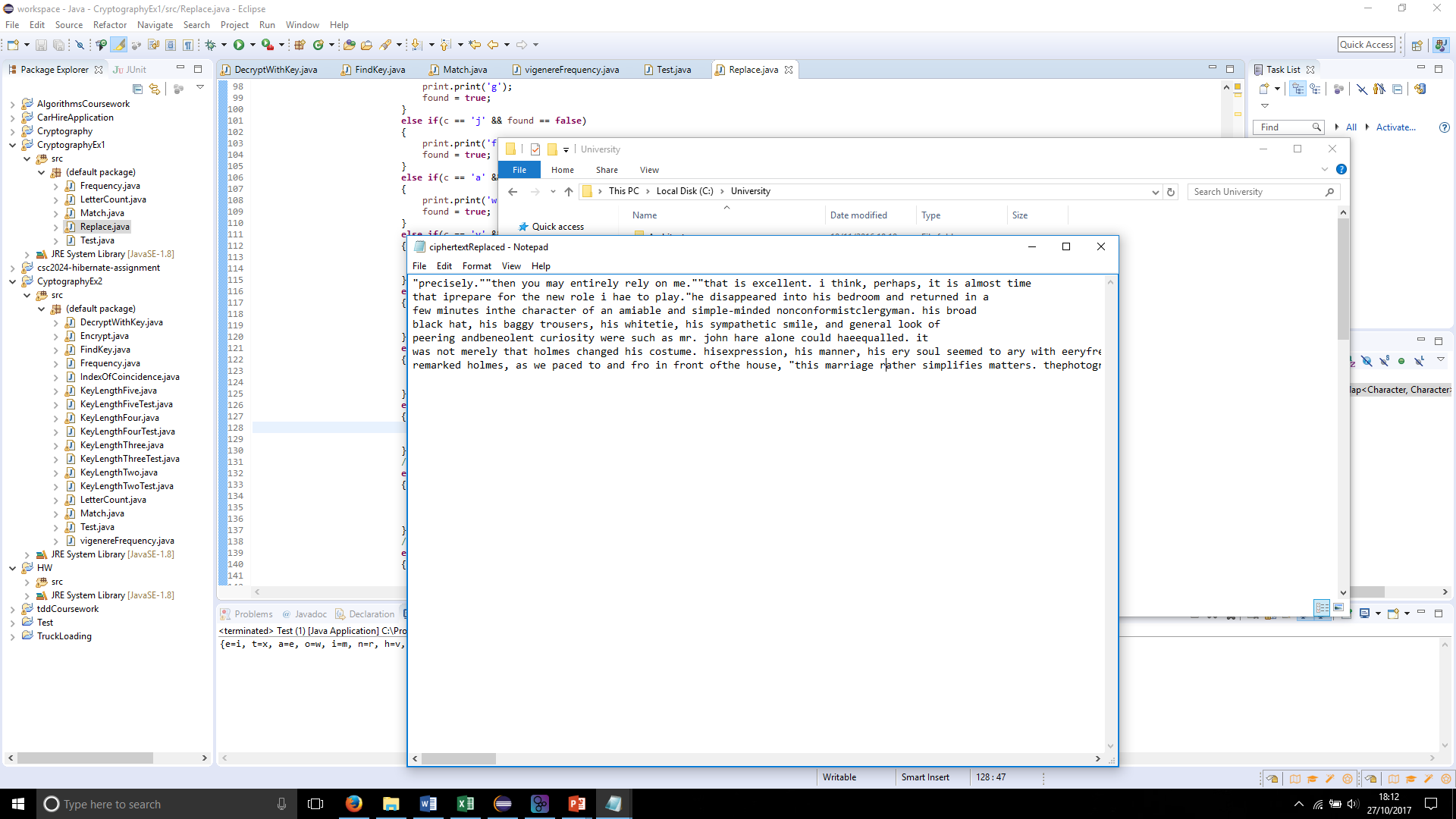
Using an Excel spreadsheet, I inputted the letters and values and created a graph. On the left is the graph found at <http://en.wikipedia.org/wiki/Letter_frequency> that maps out the relative frequency of the English language i.e. the most commonly known results of letter occurrence, while the graph on the right is taken from my frequency analysis of the plaintext. It is very clear to see that these graphs are almost identical, with letters like ‘e’, ‘t’ and ‘a’ being the most common, and ‘z’, ‘q’ and ‘j’ being the least common. This evidence supports the validity of my own frequency analysis as it follows the general trend of the most commonly known results.



On the left is the same frequency analysis of the plaintext, but ordered from the most common to the least common, and the graph on the right is my frequency analysis of the ciphertext ordered in the same fashion. Again, these graphs are very similar in appearance. We can therefore deduce that the letters in the ciphertext graph represent the corresponding letters in the plaintext graph. For example, ‘I’ has the most common occurrence in the ciphertext by a large margin, so we can therefore assume it represents the letter ‘e’, as this has the most common occurrence in the plaintext. However, many of the letters have similar occurrences meaning they do not necessarily all match, which is especially true for letters such as ‘g’ and ‘y’, which have the exact same frequency of 0.0292. The issue then arises that the Java program does not immediately print out a coherent decrypted ciphertext, and manual tuning is required.

I found this conclusion by using my Match() and Replace() method. The method passes two maps, which are the maps containing the keys and values created after using the LetterCount() method. The method stores the values, i.e. the letter occurrences, from the map in an array and then sorts these values in descending order. The method then loops through both the array and the map finding the value in the array that matches to the value in the map, and stores the corresponding letter into another array. This essentially sorts the characters from the most common occurrences to the characters with the least common occurrences. It then combines the contents of these two arrays into another map to produce:  
{e=i, t=x, a=e, o=w, i=m, n=r, h=v, s=s, r=l, d=p, l=h, u=g, m=g, w=q, c=a, y=t, f=k, g=c, p=j, b=f, v=z, k=o, x=b, j=b, q=n, z=d}  
The map shows that the character ‘I’ in the ciphertext is equal to the character ‘e’ in the plaintext, and so on.

My final method, Replace(), passes the map produced from the Match() method, the ciphertext, and a text file to print the result. The method loops through each letter in the ciphertext and compares them against the values in the map. It then prints out the corresponding letter from the key. The result was less than impressive as most of the text had not become coherent words or sentences, and therefore required a great deal of manual tuning. The cryptoanalysis had decrypted some of the letters correctly allowing me to piece together by eye other letters that could be changed to produce full words. To do this I included if statements within the Replace() method. I would look in the ciphertext and notice the letter ‘r’, for example, which I believed to be ‘h’ in the plaintext. I would look in the map for the character that ‘r’ had been transformed from, which was ‘l’. I would then overwrite this change with the aforementioned if statements by writing for the program to change all ‘l’s to ‘h’s instead. I would then repeat this process until the entire ciphertext was decrypted back into the original plaintext. One final inclusion to make this change was to include if statements for symbols as well such as ‘.’ and ‘”’ to perfectly recreate the plaintext.



In conclusion, I found that if you have a large enough sample plaintext, 447145 characters in this case, you can accurately recreate the relative frequency of the English language, which is accurate in judging the most frequent and least frequent characters in a ciphertext. While it is less accurate at judging characters with more similar frequencies, and in this case only matched 8 characters correctly out of the possible 26 it provides enough information to manually tune the rest of the characters in the text by eye until you have the original plaintext.