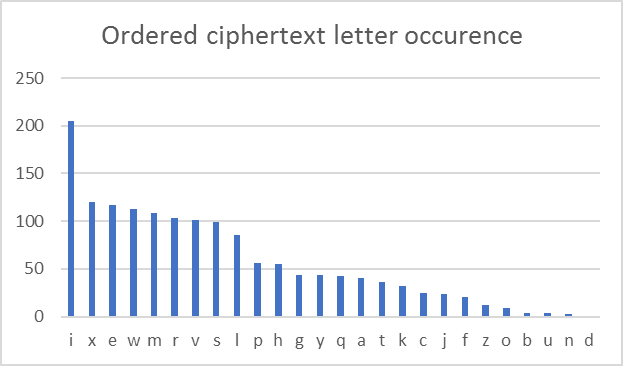
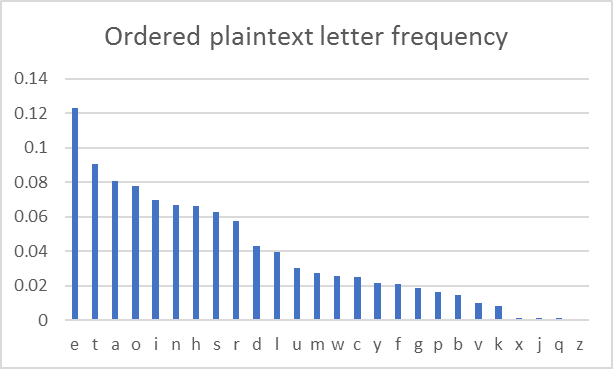
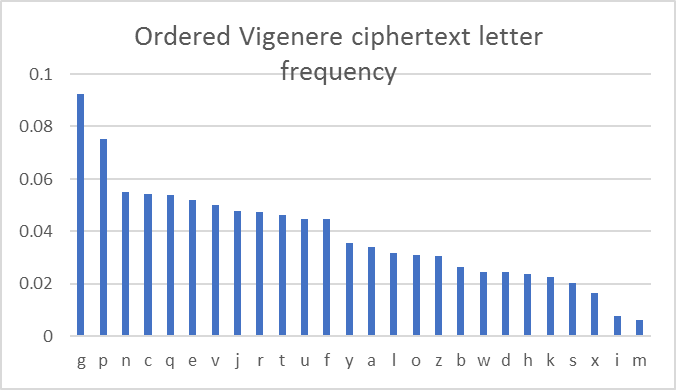
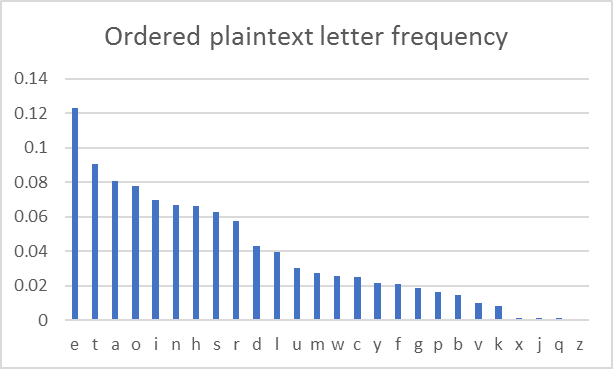
CSC3422 Cryptography Exercise 2 Report

In this report I will discuss the working of a polyalphabetic cipher, particularly the vigenere cipher. I will demonstrate how I used a user defined password to encrypt an English text file using the vigenere cipher, and how I used that same password to decrypt it. Furthermore, I will explain how I discovered the encryption key for a ciphertext generated with a vigenere cipher and recovered its subsequent plaintext.

The first task involved accepting a user-defined password, which would act as the key, to encrypt the text file pg1661.txt. My method Encrypt() firstly splits the password into a character array so that each character has its own index that can be referred to. I then loop through each line in the text, and because this is a normal English text I accounted for the large number of symbols and numbers that occur occur. I wanted to preserve these symbols after encryption for simplicity sake, so I added if statements to begin that would print out the symbol or number again should it find one. This was necessary for encryption because if the program mistook a symbol for a character it would treat it as a character and attempt to shift it, causing the key to be out of sync with the rest of the letters, leading to an incorrect encryption. To shift a character I used a simple equation; I minus the character ‘a’, or 97, from both the letter in the plaintext, and the password, reducing them both to between 0 and 25, the letters of the alphabet. I then add the ASCII of the password character to character of the plaintext. Should this number be beyond 25, I also included a modulus 26 operation so the character would essentially wraparound from ‘z’ to ‘a’. I then add 97 back to the result and return the ASCII as a character, resulting in the new shifted character.

When encrypting such a large text file it can be difficult to judge if it has encrypted correctly, especially when any password can be defined. To test the correctness of my method I created a text file with ‘newcastleuniversity’, and used the password ‘ncl’ to encrypt it. The resulting encryption produced the correct string of characters: ‘aghpcdgnphptigcfkel’. I knew this to be correct because, for example, if we take ‘agh’, which is ‘new’, we know that the character ‘n’ was used to rotate the character ‘n’; given that ‘n’ is at position 13 the character ‘n’, also at position 13, would have been shifted 13 times, producing ‘a’ at position 0 of the ASCII alphabet. This occur because 26 goes beyond ‘z’ and therefore needed to wrap back around to ‘a’.

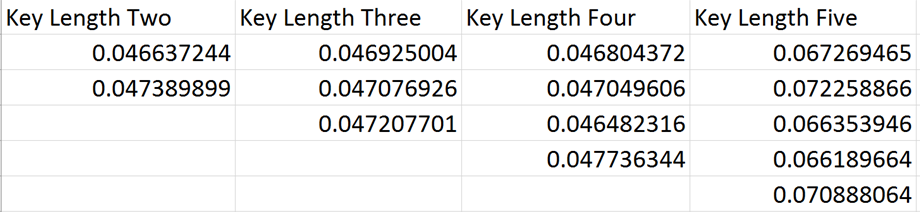
  
Pictured on the left is an ordered graph of the letter occurrences from pg1661.txt, while on the right is an ordered graph of my frequency analysis of the ciphertext from exercise one, both from most common to least common. The important aspect of these graphs is that they are very similar in appearance. The character ‘e’ in the plaintext has a very high frequency of 0.122, while subsequently the character ‘I’ in the ciphertext is the highest at 0.136. They also follow the same degradation as the character frequencies remain steadily high until a drop in the middle of the graph, which continues to fall until the frequencies are little to none. It would be relatively clear from these graphs which character in the ciphertext represents which character in the plaintext.



On the left is the same plaintext letter frequency graph, but the graph on the right is an ordered graph of the letter frequencies of the plaintext after being encrypted by a vigenere cipher with the key ‘ncl’. Unlike my previous frequency analysis, the vigenere cipher graph is slightly less similar in appearance to the plaintext graph; it still follows the same curvature, but the distribution of characters is a lot more even. The difference is that a shift cipher shifts all characters in a text by a certain number, whereas a vigenere cipher shifts characters dependant on the position the character is within the key. This gives a more random distribution of letters that cannot be deciphered by frequency analysis alone. This is evident by the graph; characters like ‘z’, ‘j’, ‘q’, and ‘x’ have almost no frequency in the plaintext, but there is no such pattern in the vigenere graph.

Before I attempted to retrieve the key and plaintext from exercise two I wanted to have a working method that would be able to decrypt a file once the key was known. This method was DecryptWithKey(), which works very similarly to my Encrypt() method, but reverses the encrypting process. The only difference between these two methods is the equation used to shift the characters; the method takes a character from the cipher and subtracts the character from the password, which consequently shifts the character back to its original character in the plaintext. For the characters to wraparound I included an if statement that will add 26 to the character should the subtraction drop below 0.

With the method to decrypt a ciphertext with a given key now written, I began the process for finding the key for the exercise two ciphertext. The first step to doing so is finding the key length. The first step to finding the key length is splitting the ciphertext into subtexts e.g. to test if the key length may be 2 you split the cipher into two equal length parts, but in the form ab so that all a’s are split into the first subtext, and all b’s are split into the second subtext. I used several KeyLength() methods to find the appropriate key length. The methods have a number of lists equal to the number of subtexts I wished to split the ciphertext into. I would then loop through each line adding each character to the appropriate list, and then printing the lists out into separate text files.

The second step to finding the key length is using index of coincidence. Once I had split the subtexts into equal parts I would run each one through my IndexOfCoincidence() method, which would simply run the number of occurrences of each letter in the subtexts through the numerator of the index of coincidence formula, and then the denominator. The table below shows the results. 

The correct key length is five because the index of coincidence for each subtext is close to the index of coincidence for a normal English text; 0.065. I then used my Match() method to order the letter occurrences in each of the subtexts from highest to lowest, and them match these with the letter occurrences from the English text pg1661.txt. The method takes in two maps, each map featuring the character and its number of letter occurrence for both the plaintext and ciphertext. It then populates two arrays with the values from each map, and orders these from highest to lowest. The values are then matched with their corresponding characters, and these characters are put into another array, which produces two arrays with the characters ordered from highest to lowest. These two arrays are then merged into a single map to match the occurrences of the characters in both texts.

Subtext One: {e=t, t=i, a=d, o=c, i=x, n=w, h=p, s=h, r=g, d=a, l=s, u=r, m=b, w=b, c=v, y=e, f=e, g=n, p=q, b=l, v=k, k=z, x=m, j=m, q=o, z=f}

Subtext Two: {e=p, t=e, a=z, o=t, i=l, n=l, h=c, s=s, r=d, d=o, l=w, u=x, m=a, w=q, c=n, y=h, f=f, g=r, p=m, b=j, v=g, k=v, x=b, j=i, q=k, z=k}

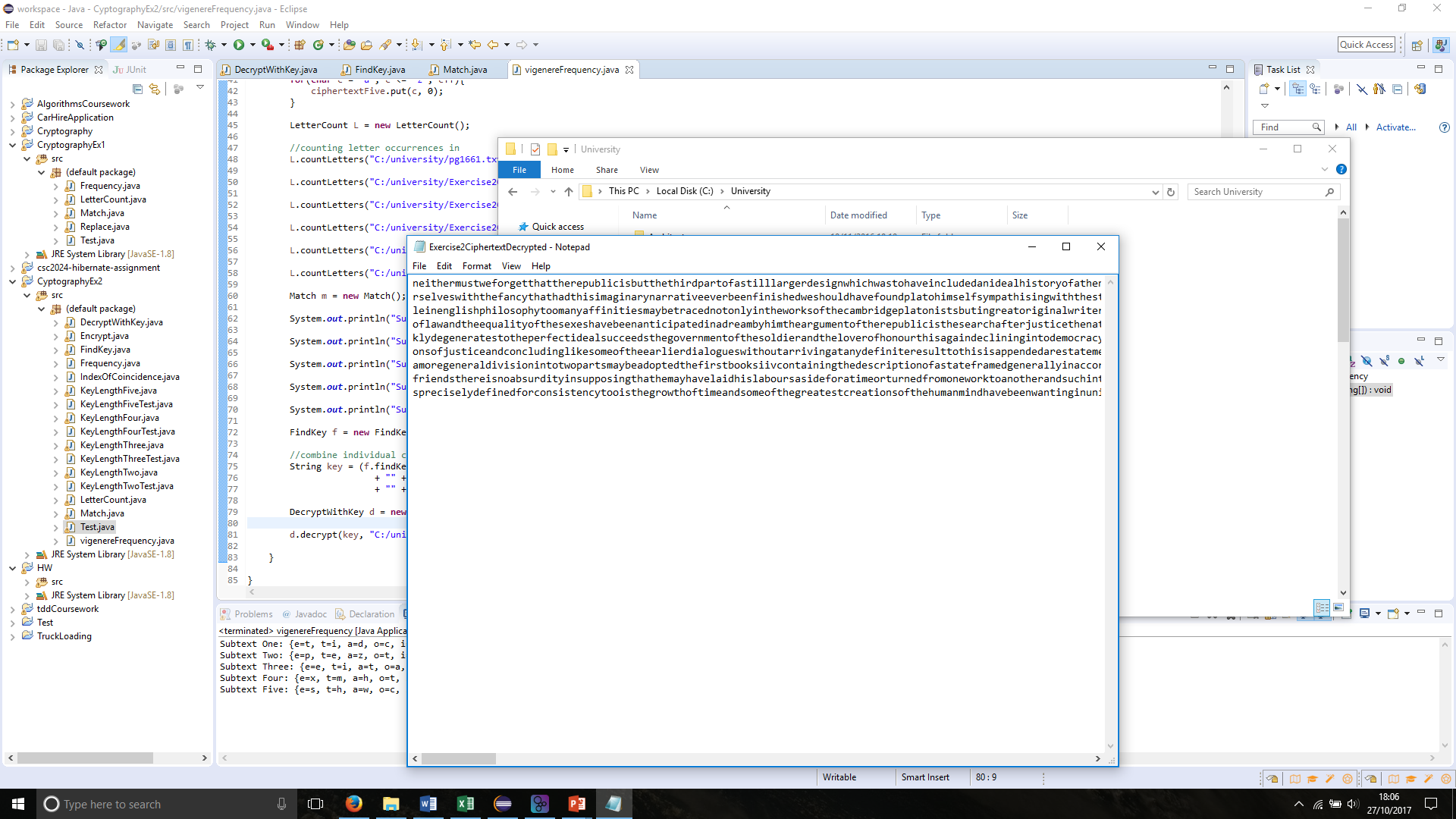
Subtext Three: {e=e, t=i, a=t, o=a, i=o, n=n, h=s, s=h, r=r, d=d, l=d, u=c, m=f, w=p, c=u, y=m, f=y, g=b, p=w, b=g, v=v, k=k, x=j, j=q, q=q, z=q}

Subtext Four: {e=x, t=m, a=h, o=t, i=g, n=b, h=a, s=a, r=k, d=w, l=e, u=v, m=y, w=n, c=i, y=f, f=r, g=r, p=p, b=u, v=o, k=d, x=c, j=c, q=j, z=s}

Subtext Five: {e=s, t=h, a=w, o=c, i=o, n=b, h=v, s=g, r=f, d=z, l=q, u=t, m=r, w=i, c=u, y=k, f=a, g=p, p=d, b=m, v=j, k=y, x=x, j=l, q=e, z=e}

Using this information, I manually counted the difference between each letter, which would be the number of shifts it would have taken to shift the character in the ciphertext to the character in the plaintext. For example, ‘t’ is at position 19 in the alphabet, and ‘e’ is at position 4, so it would have taken 15 shifts to shift ‘t’ to ‘e’. The character ‘p’ is at position 15, which means ‘p’ may have been used as the first letter of the key. I continued this process with every pair of letters for each subtext. I eventually discovered that the individual letters spelled out ‘plato’, so I input this as the key through my DecryptWithKey() method, which output a deciphered text file with coherent words.

Frequency analysis is never completely accurate, so I found the differences between letters varied greatly, nevertheless their appeared to be a common number of shifts between some letters, so I judged the character at this position to be the letter of the key. I noticed that whatever letter was matched to the character ‘e’ always had the same number of shifts as the most common number of shifts. This would be because ‘e’ is the highest frequently occurring letter in the English language, so a frequency analysis would almost always match this character correctly. I created a class called FindKey() that would calculate the difference between each pair of letters from the map and store these in a list. It would then return the difference between the character ‘e’ and its corresponding letter as a character itself. Passing each subtext through this method produced the 5 correct letters that made up the key; ‘plato’. A clear weakness of the vigenere cipher is that the key can be found quite easily once the key length is known, as all it takes is to know the number of shifts between the letter ‘e’ and its corresponding character.



In conclusion, the vigenere cipher is built using several ceaser ciphers making it more secure than a single ceaser cipher. It requires some transposition as the key length must first be found before the actual key, which requires breaking the plaintext into several subtexts. However, once the key length is found it becomes a simple task of breaking each individual ceaser cipher to find the letters that construct the whole key, almost as simple as looking at the character ‘e’ and counting the shifts. The vigenere cipher also still contains some discernible patterns in the ciphertext.