# Coursework for CSC3621 Cryptography

## Part 1

### Exercise 2

The main method for this exercise can be found within the Vignere.java. The exercise was split into two parts which are contained within their own separate classes. An instance of each of these classes are called from the main method and their constructors are used to show how the exercise was completed. This was done for clarity between the two tasks.

#### Encrypting Sherlock with Vignere

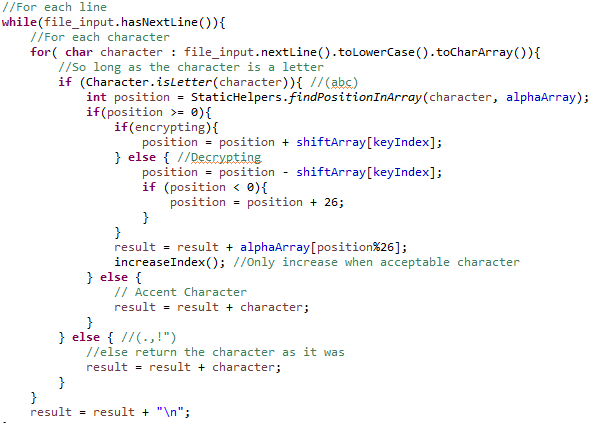
The Vignere encryption is found within the EncryptDecryptTask.java. The constructor is called and asks user for a key. This can be any length except for length 0 and must only contain letters from A-Z. It will then down case all input and use this as the key.

The next step is to set up a shift array which takes each letter within the key and assigns it a number based on its position within the alphabet. For example the key ‘NCL’ would give a shift array of [13,2,11].

Next the program will take a given file and begin encryption of that file.

Note for the Sherlock.txt file or files of a similar number of lines, it will take a minute or two to encrypt or decrypt due to the way I am reading in each line.

The method for encryption reads in the file line by line then splits the line into a character array. If the character is a letter and is not an accented character it will shift the letter by a given value deduced from the key. The first letter is shifted by the first shift array value. In the example above this is an N and therefore it shifts 13 places to the right. If adding 13 to the letters current position is above 25 then it will loop round to the start via a modulus of 26. Z + 1 = A. For each character within the line array that is valid the index for the shift value will increase. So the second letter will shift by 2 places and the third letter will shift by 11 places. The index will then reset and the 4th letter will shift by 13 places.



This is done for each line of the text.

Decryption of a file uses the same method but differ in a Boolean as a parameter. If the encrypting Boolean is true then the position will shift to the right else if it is false and decrypting it will shift to the left.

After encryption the encrypted text is printed to a file encyrptedFile.txt.

After which a character count is done on the file. This uses the same code as seen in exercise 1 to take each line and then each character on that line and count the occurrence of each letter.

The results can be seen below:

Character Count on encrypted file

Encrypted Text Frequency - { g, p, e, v, r, c, t, n, q, f, y, u, j, z, a, l, s, o, b, d, h, k, w, x, I, m }

Comparing this to other frequency arrays

Exercise 1 Cipher Text Frequency - { i, x, e, w, m, r, v, s, l, p, h, y, g, q, a, t, k, c, j, f, z, o, u, b, n }

Plain Text Book Text Frequency - { e, t, a, o, i, n, h, s, r, d, l, u, m, w, c, y, f, g, p, b, v, k, x, j, q, z }

English Letter Frequency - { e, t, a, o, i, n, s, h, r, d, l, u, c, m, f, w, y, p, v, b, g, k, j, q, x, z }

We can see that the no real correlation between the newly encrypted frequency alphabet and any of the frequency alphabets from the first exercise. It is also useful to compare the actually count of the most common characters before and after the encryption process.

|  |  |
| --- | --- |
| Before Encryption | After Encryption |
| e : 54944  t : 40511  a : 36142  o : 34869  i : 31248  n : 29731  h : 29579  s : 27965  r : 25684  d : 19100  l : 17636  u : 13636  m : 12155  w : 11534  c : 11103  y : 9760  f : 9363  g : 8299  p : 7284  b : 6638  v : 4572  k : 3681  x : 577  j : 544  q : 437  z : 153 | g : 33374  p : 31968  e : 25783  v : 25413  r : 23475  c : 22959  t : 21648  n : 21536  q : 21238  f : 20442  y : 19700  u : 19468  j : 17024  z : 15804  a : 15551  l : 15479  s : 12926  o : 12460  b : 11842  d : 11634  h : 11472  k : 10636  w : 10570  x : 6846  i : 4412  m : 3485 |

As we can see from the figure above. The distribution of the frequency is a lot lower. Compared to the extremes of the most common e : 54944 and least common z : 153 letters before encryption. If we were to increase of the size of the key from three letters (ncl) to nineteen letters (newcastleuniversity) this distribution of frequency would be even lower. This completely takes away the ability to look for the most common letter within the text that you can do with a key length of one otherwise known as a Ceaser cipher.

#### Decrypting given text

My process of decrypting the given cipher text into plain text can be found in DecipherTask.java.

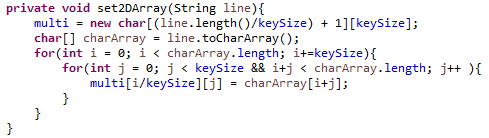
Knowing that the text was encrypted using the vignere cipher the first step I took to decipher the text was to find the key length. Knowing the key length would enable me to sort the text into a key-length number of columns and then each of these columns would be equivalent to their own Ceaser Shift Cipher. From the review above around frequency analysis, if I knew a set of characters that were all shifted by the same amount (by a key of one) I could run a character count and find the shift between the most common letters.

To calculate the key length I used an Index of Coincidence. The logic for calculating the index of coincidence can be found within IndexOfCoincidence.java.

IndexOfCoincidence is instantiated with a given file which contains the text you are trying to decipher and a given key length. It reads in the file and concatenates all lines into one string with no spaces. A 2D array is then generated for string where the number of rows is the number of characters divided by the key length, and the number of columns is the key length.

The set2DArray fills up the array by looping around the rows and then around each column and places each character one by one into the array. Such that if you have a string of ‘abcdefgh’ you would get the 2 D arrays

Key Length 2

[ [ a, b ]

[ c, d ]

[ e, f ]

[ g, h] ]

Key Length 3

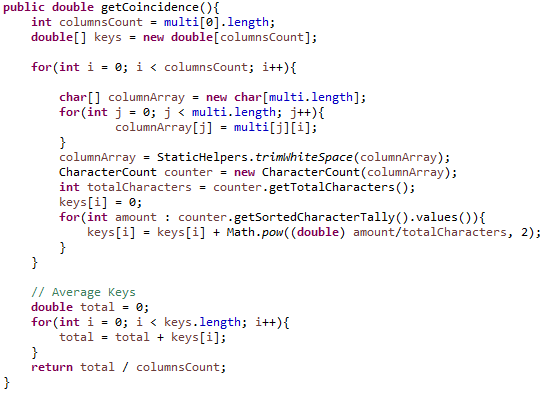
[ [ a, b, c ]

[ d, e, f ]

[ g, h ] ]

The getCoincidence method can then be called on the object. This uses the 2D array to generate a key length number of smaller arrays (the columns). A character count is then run on each of these individual columns. For each of characters that appear in the array we take the amount they appear, divide it by the total number of characters and then times it by itself (to the power of two). We add these together for each appearing character to give us an index for that particular column.

Finally we have to take an average of each column to give us the overall index for the given key length.



The point of these indexes is we want to find a result that is close to the index of normal English text which is around 0.065. The key length with the average key length close to that of the English text is likely to be the key length that was used to encrypt the text.

Running the code up to a key length of 20 we get the following output

Key Length: 1 Index: 0.04523184579952513

Key Length: 2 Index: 0.04534155620833542

Key Length: 3 Index: 0.045503916625638956

Key Length: 4 Index: 0.04556229037357947

Key Length: 5 Index: 0.06675956094939621

Key Length: 6 Index: 0.04581856713006008

Key Length: 7 Index: 0.045772719691729906

Key Length: 8 Index: 0.046029031003618714

Key Length: 9 Index: 0.04624043031590964

Key Length: 10 Index: 0.06720749025073522

Key Length: 11 Index: 0.04640206043204396

Key Length: 12 Index: 0.04645085449387423

Key Length: 13 Index: 0.046283139912250984

Key Length: 14 Index: 0.04650881830285445

Key Length: 15 Index: 0.06777863467131673

Key Length: 16 Index: 0.046789968723483295

Key Length: 17 Index: 0.04704394748469526

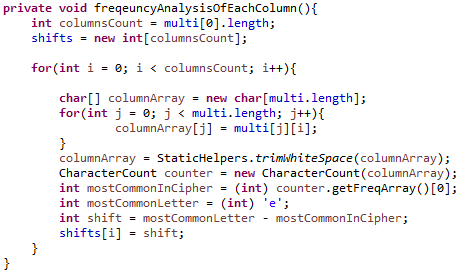
Key Length: 18 Index: 0.04717119926304315

Key Length: 19 Index: 0.047246530225626124

Key Length: 20 Index: 0.0684281003418803

As we can see the number closest to 0.065 is found at key length 5. Also note that the multiples of 5 also have a strong high index close to that of the English alphabet. If you were to repeat the key e.g. ‘abcdeabcdeabcde’ you would get the same result as ‘abcde’ and therefore all multiples of 5 have an index close to one another.

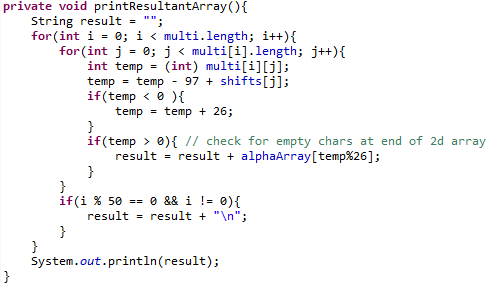
The last step is to take the 2D array for the best key length, key length 5 and try and break the shift cipher for each column. We know for a fact that each letter within that column has been shifted by the same number of places so we can run another character count on that column, find the most common letter and find the shift between that letter and the most common letter in the English alphabet ‘e’.



Shift Array = [-15, -11, 0, -19, -14]

Translates to key – PLATO

The code below does this for each column and builds up a shift array. This shift array can then be used to print out each decrypted letter as shown below.



The result of this method is the fully deciphered text. A small part of this can be found here:



This is an excerpt from The Republic by Plato and hence the key of PLATO