ELEC6242: Cryptography

Cryptanalysis Coursework

Dominic H. Heaton

Student ID: 2777 9009

dhh1g15@soton.ac.uk

MEng Electronic Engineering with Mobile & Secure Systems

University of Southampton

February 25, 2019

Contents

1 Outline

In this report various cryptanalysis methods are employed to decipher three ciphertexts each of which have been encrypted using different cipher methods. For each of the ciphertexts to be broken, a Python program was developed which was capable of producing the plaintext from the provided ciphertext as well as returning the value of the key used to encrypt the plaintext in the first place. The approach to deciphering each ciphertext is described in this report as well as the code developed being included for reference in the appendices.

2 Solution for Cipher 1

Deciphered Plaintext:

Formative assessment can be viewed as a mean to enhance the learning process. Based on the results of such assessments, students will be able to assess their knowledge and identify strengths and weaknesses. The teacher will also have indication on how well the students are grasping the fundamental facts and whether he needs to alter their teaching to emphasis some important concepts.

Key: tyu

3 Cipher 1 Cryptanalysis

The first test of the ciphertext was to calculate the Index of Coincidence (IC). A python script, `indexOfCoincidence.py`, was generated to calculate the value as shown in Equation 1. The Index of Coincidence of 0.5278 is close to the value expected of written English (0.066) and therefore it was assumed that the plaintext was a message in English. From this value it was determined that the most likely ciphers to have been used to encrypt the message were substitutional ciphers.

$$IC \quad = \quad \frac{\sum_{i=A}^{Z} f_i \left(f_i - 1\right)}{N \left(N - 1\right)} \quad = \quad 0.05278 \tag{1}$$

The first decryption method attempted was the Caesar Substitutional Cipher, in which the letters in the plaintext will have been shifted by a fixed amount to generate the ciphertext. A Brute Force methodology was used in an attempt to decode the Caesar Cipher as in its simplest form there are only 25 possible keys. A script was generated (`caesar.py`) to output all of these possible deciphered plaintexts, but none of these revealed the message.

The next stage was then to attempt to decrypt the ciphertext using another substitutional cipher and therefore the Vigenere Cipher was selected. The approach for this cipher involved first carrying out the Kasisky Test to find the longest pattern of letters which were repeated within the ciphertext. The distance between these repetitions was then calculated, with the key length being a factor of this distance. This provided key lengths of `1,2,3,6,13,26,39,78` given the distance of 78 characters between the repeated pattern `tqmxqmfchm`. It was assumed that the encryption key was unlikely to have a length less than 3 and therefore keys of length 1 and 2 were ignored.

Frequency Analysis was carried out under the assumption of a key space of 3 letters, producing the graphs in Figure 1. Each graph represents a letter of the key and compares the frequency of the letters in the ciphertext to that of the frequency of a letter appearing in the English language. The graphs allow the determination of the shift applied for each letter in the plaintext to generate the ciphertext. The first attempt to decipher the message using the shifts extracted from the frequency analysis provided the plaintext shown in Figure 2.

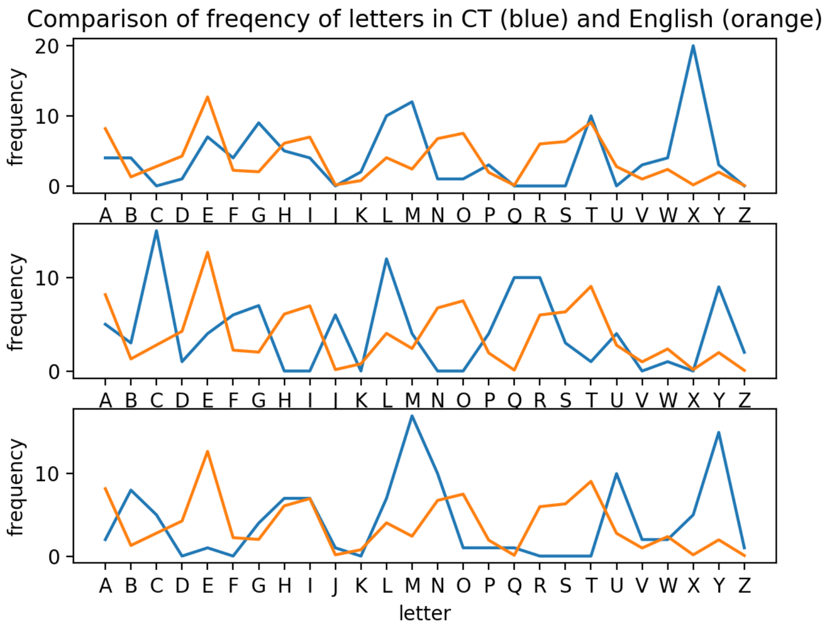

Figure 1: Frequency Analysis of the Ciphertext (CT) assuming a key space of 3 in the Vigenere Cipher. The shape of the curves on each graph allows the shift used in the Vigenere cipher to be determined. The graphs show the frequency of letters in the ciphertext (blue) and the english language (orange).

```
FodmafivqaseesemeztcmnbqviqwepasmmemntaentanoettelqarzinspraceesBmseponfhedesglteof
euctaseesemeztsetupenfswullneanlefoaeseestteidknawlqdgqanpidqntufyetrqngfhsmndieawneese
eThqtemchqrwullmlsahaheizdioatuonanhawwqllfheetupenfsadegdasbinsthqfuzdayenfalracfsazdw
tettertenqedetomltqrtteidtemchungfoeyphmsiesoyeiypodtaztcancqpte
```

Figure 2: First attempt at Vigenere Deciphering using a simple shift according to the most frequently appearing letters as shown in Figure 1. The result resembles english except for every third letter has not been deciphered properly.

From Figure 2 it can be seen that the Vigenere deciphering has revealed a plaintext which is almost a string of English words except for every third letter having been incorrectly deciphered.

This meant that the shift used for the third letter of the key was incorrect. The third graph in Figure 1 shows several sharp peaks in frequency for the ciphertext. As the highest peak in the ciphertext proved unsuccessful in deciphering the plaintext, the next highest peak was used. This shift then allowed the plaintext to be revealed as shown in Section 2. The key length of 3 had successfully deciphered the ciphertext, using the Vigenere Cipher. From these shifts it could be determined that the key is `tyu`. The `q3.py` script was generated to carry out the Vigenere Deciphering as well as the Kasisky Test and Frequency Analysis for this question. These scripts can be found in Appendix A.

4 Solution for Cipher 2

Deciphered Plaintext:

In ancient Egypt servants were smeared with honey to attract flies away from the pharaoh

Key: 0x1abc

5 Cipher 2 Cryptanalysis

This ciphertext was in the form of a '.hex' file and therefore a starting point was to look to decode the data into a read-able ASCII format to see if the message was encoded rather than encrypted into hexadecimal format. Figure 3 shows the '.hex' file alongside the ASCII representation showing that the message was not simply encoded.

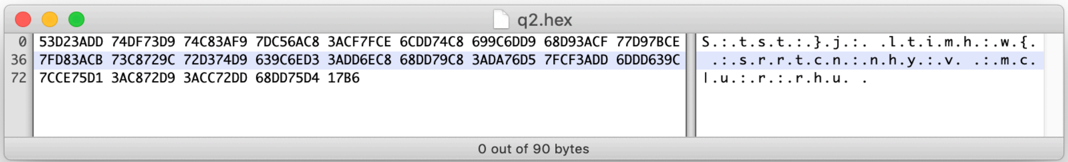

Figure 3: The hex file (left) to be decrypted in this question alongside the ASCII conversion (right) of the hexadecimal.

The ciphertext is of hexadecimal form which is simply converted to binary at which point the XOR cipher seemed a reasonable cipher to start with in attempting to decipher the plaintext. This was because the XOR cipher is reversible and therefore the hint provided with the plaintext beginning with the letter 'I' would allow a reverse engineering of the ciphertext to provide the key use to encrypt the plaintext.

This method initially reverse engineered the first hexadecimal value in the ciphertext (`0x53`) to give the plaintext letter 'I', giving the key to be `0x1a`. The XOR cipher applies the key to each byte of the plaintext until the message is encrypted (or decrypted). The 1-byte key of `0x1a` was applied to each byte of the ciphertext revealing the plaintext message shown in Figure 4. This plaintext revealed no meaningful information and therefore a key length of 1-byte was unsuccessful.

```
 >Attempting decryption of the ciphertext using this 1-byte key (0x1a):
¬È ÇnÅiÃnÒ ãgßpÒ ÕeÔvÇnÒs wÃrÃ ÕmÃaÔeÂ ÑiÒh hÉnÃy tÉ ÇtÒrÇcÒ ÀlÏeÕ ÇwÇy fÔoË ÒhÃ ÖhÇrÇoÎ
```

Figure 4: Attempting to decipher the ciphertext using a 1-byte key of `0x1a` failed to produce a meaningful plaintext

Following the failure of a 1-byte key, an attempt to use a 2-byte key was used, where the first byte was the `0x1a` value which must be used for the first letter in the cipher as we know that the letter 'I' is the first plaintext letter. A 2-byte key means that the bytes of the key are applied in an alternating pattern to bytes of the ciphertext (or plaintext as XOR is reversible).

The second byte of the key was found using the Brute Force methodology, where the key was of form `1aXX` where `XX` was the second byte of the key to be determined. If a second byte was found in this method, the plaintext would also be revealed at the same time. This brute force method iterated through all possible keys (00≤XX≤FF) to produce 256 plaintexts. English language detection was then applied on these plaintexts to extract only plaintexts which resembled english. This filter meant that the total number of plaintexts printed by the developed script was limited and from the shortened list produced, the actual plaintext and key could be easily spotted. The plaintext and key are shown in Section 4. The python script (`q2.py`) developed to decipher the '.hex' file and reveal the plaintext and the key can be found in Appendix B.

6 Solution for Cipher 3

Deciphered Plaintext:

There are probably more than hundred billion galaxies in the cosmos each of those has up to trillion stars

Key Length: 8

7 Cipher 3 Cryptanalysis

The first stage of deciphering the ciphertext was to carry out frequency analysis, producing the graph in Figure 5. This figure shows a comparison of the frequency of letters appearing in the ciphertext in comparison to the frequency of letters appearing in the english language. The similar shapes of the graph imply that the ciphertext is a re-arrangement of the order of characters from the plaintext, otherwise known as a Transposition cipher.

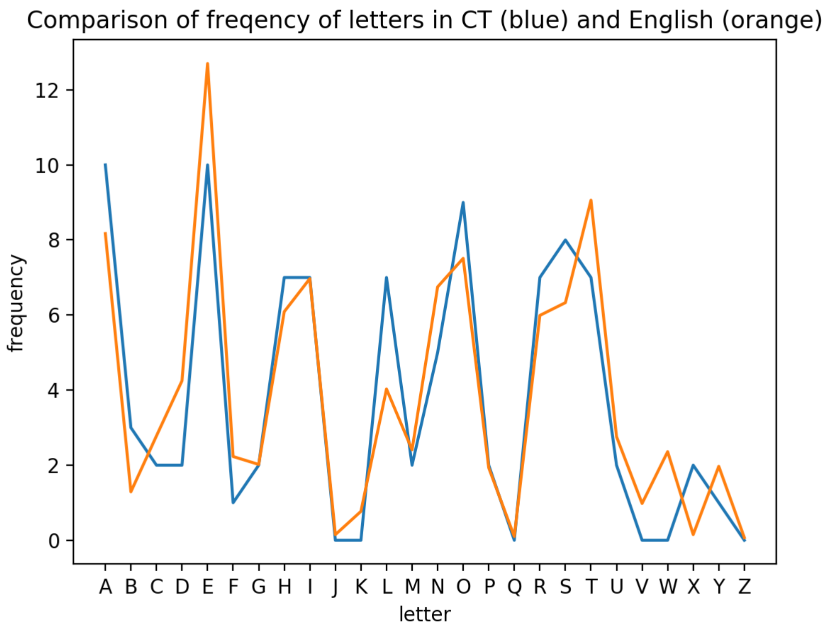

Figure 5: Frequency Analysis of the Ciphertext (CT) in comparison to text in the English language. The two graphs are very similar in shape and therefore the cipher method using is likely to be a transpositional cipher.

Two main methods exist for transposition ciphers; the simplest being the Rail Fence Cipher, and the other being the Columnar Transposition Cipher. The method of approach used here began by exploring the Rail Fence Cipher method. A script, `railTest.py` was developed to show that the implementation would allow messages to be both encrypted and decrypted using the Rail Fence Cipher. The decryption methods from this script were then used in another script, `rail.py` to Brute Force all possible combinations of the plaintext for the given ciphertext.

The ciphertext has a length of 96 characters and therefore there could be up to 95 rails used in this cipher mode. The `rail.py` script produced all the possible plaintexts when using any number of rails between 2 and 95. The result of this showed that the ciphertext had not used the Rail Fence Cipher as the decrypted messages held no useful information. Therefore the Columnar Transposition Cipher was then attempted as a method to decrypt the plaintext.

A python script was developed to once again Brute Force all possible combinations of the ciphertext to generate the potential plaintexts (`q3.py`). For this method a table is generated containing the ciphertext with a number of columns matching the length of the key being used. These columns are re-arranged using the alphabetical order of the word being used as the key. The Brute Force method employed therefore had to generate the possible tables for keys of various lengths and then try every possible combination when re-ordering the columns as to produce all possible plaintexts.

The length of the message was 96 characters long and therefore potential key lengths (excluding those with length less than 3) were `3,4,6,8,12,16,24,32,48,96`. The developed script used two language recognition libraries to examine the plaintexts produced using the re-arrangement method of the columns. A set of rules were introduced using these libraries so that the number of plaintexts output was limited. This was required due to the increasing number of potential plaintexts for increasing key number as shown in Table 1 for key lengths of up to 8.

Key Length	Calculation of Number of Plaintexts	Number of Plaintexts
3	$3! = 3 \times 2 \times 1$	6
4	$4! = 4 \times 3 \times 2 \times 1$	24
6	$6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1$	720
8	$8! = 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1$	40320

Table 1: Increasing number of plaintexts produced by the Brute Force method of decrypting the Columnar Transposition Cipher. The total number of columns in the transposition table is given by the length of the key and each column can then be re-arranged to generate a large number of permutations all providing unique potential plaintexts.

Two language libraries were used as each provided different analysis methods, neither of which was perfect, but together they provided a fairly successful method for detecting english words in the deciphered plaintexts. A filtering process implemented using language detection meant that only 48 potential plaintexts were output having run the `q3.py` script. These were then examined and the plaintext was found in amongst this list. The decrypted plaintext is found in Section 6 and the scripts used in this section in Appendix C.

Appendix A Code for solving Cipher 1

indexOfCoincidence.py

```
# Author: Dominic Heaton
# Index of Coincidence Calculator
################################################################################
from langdetect import detect
from guess_language import guess_language
from termcolor import colored
import re
import matplotlib.pylab as plt
import itertools
import os
import collections

def loadFile():
    open_file = open("q1.txt", "r")
    ciphertext = open_file.read().rstrip('\n')
    open_file.close()
    # print(ciphertext)
    return ciphertext

def indexOfCoincidence(ciphertext):
    ciphertext  = "".join([x.upper() for x in ciphertext.split() if  x.isalpha()
        ])
    N = len(ciphertext)
    sumOfFrequency = 0
    alphabet =  map(chr, range( ord('A'), ord('Z')+1))
    letterFrequency = collections.Counter(ciphertext)
    for letter in alphabet:
        sumOfFrequency += letterFrequency[letter] * (letterFrequency[letter] - 1)
    indexOfCoincidence = sumOfFrequency/(N*(N-1))
    return indexOfCoincidence

### MAIN PROGRAM
ciphertext = loadFile()
indexOfCoincidence = indexOfCoincidence(ciphertext)
print(' >Ciphertext: \n' + ciphertext + '\n')
print(' >Index of Coincidence for this Ciphertext is: ' + colored(str(
    indexOfCoincidence),'red') + '\n')
```

caesar.py

```python
# Author: Dominic Heaton
# Caeser Cipher Brute Force
# Tries 25 shifts of the alphabet to solve the CT
##################################################
from langdetect import detect
from guess_language import guess_language
from termcolor import colored

def encrypt(string, shift):
  cipher = ''
  for char in string: #Check spaces
    if char == ' ':
      cipher = cipher + char
    elif  char.isupper(): #Upper case shift
      cipher = cipher + chr((ord(char) + shift - 65) % 26 + 65)
    else: #Lower case shift
      cipher = cipher + chr((ord(char) + shift - 97) % 26 + 97)
  return cipher

def caeserDecrypt():
    print('\nAttempting Caeser Cipher Brute Force')
    print(' >ciphertext: ' + ciphertext)
    for i in range(0,26):
        decrypted_shift = encrypt(ciphertext, i)
        # if detect(decrypted_shift) == 'en': #attempt to limit output via language detection (not perfect)
            # if guess_language(decrypted_shift) == 'en': #attempt to limit output via language detection (not perfect)
        print(' >plaintext : ' + colored(decrypted_shift, 'red')) #print in colour
        print(' >shift num.: ' + str(i-26) + '\n')
    print('Finished Caeser Decrypt Attempt')

### Main Program ###
open_file = open("q1.txt", "r")
ciphertext = open_file.read().rstrip('\n')
ciphertext = ciphertext.rstrip('.')
open_file.close()

caeserDecrypt()
```

q1.py

```
# Author: Dominic Heaton
# Solution Script for Q1
# Performs Kasisky Test and Frequency Analysis to establish key length and
#   to decipher the final plaintext message which is exported in q1-solution.txt
# REQUIREMENTS: See libraries imported;
#   langdetect, guess-language, termcolor, re, matplotlib
###############################################################################
from langdetect import detect
from guess_language import guess_language
from termcolor import colored
import re
import matplotlib.pylab as plt

def loadFile():
    open_file = open("q1.txt", "r")
    ciphertext = open_file.read().rstrip('\n')
    open_file.close()
    # print(ciphertext)
    return ciphertext

def stripPunctuation(ciphertext):
    noPunctuation = re.sub(r'w\s]','',ciphertext) #remove all non-alphanumerics
    strippedCiphertext = noPunctuation.replace(' ', '') #remove all spaces
    return strippedCiphertext

def largestPattern(ciphertext):
    length = 0
    i=0
    j=0
    for j in range(len(ciphertext)):
        for i in range(len(ciphertext)):
            substring = ciphertext[j:i]
            if len(list(re.finditer(re.escape(substring),ciphertext))) > 1 and len(substring) > length:
                match = substring
                length = len(substring)
    return match.strip() #remove all spaces

def repeatDistance(ciphertext, repeatedString, stringLength):
    firstOccurance = ciphertext.find(repeatedString) #start of firstOccurance
    endOfFirstOccurance = firstOccurance + stringLength #end of firstOccurance
    secondOccurance = ciphertext.find(repeatedString, endOfFirstOccurance) #start finding after firstOccurance
    repeatDistance = secondOccurance - firstOccurance
    return repeatDistance

def findFactors(value):
    factors = []
    for i in range(1, value+1):
        if value % i == 0:
            factors.append(i)
```

```
    return factors

def removeSmallKeys(factors):
    print('>Ignoring keys < 3 in length as they are highly unlikely')
    factors.remove(1)
    factors.remove(2)
    return factors

#function returns three sets of characters - each set has been encrypted with the same letter of the key
def getEncryptedGroups(ciphertext, keyLength):
    keyLetter1 = ''
    keyLetter2 = ''
    keyLetter3 = ''
    for i in range(0,len(ciphertext), keyLength):
        keyLetter1 += ciphertext[i]
    for j in range(1,len(ciphertext), keyLength):
        keyLetter2 += ciphertext[j]
    for k in range(2,len(ciphertext), keyLength):
        keyLetter3 += ciphertext[k]
    return keyLetter1, keyLetter2, keyLetter3

def letterFrequency(ct):
    ct = ct.upper() #make all upper case
    freqAlphabet = {'A' : ct.count('A'), 'B' : ct.count('B'), 'C' : ct.count('C'),
                    'D' : ct.count('D'), 'E' : ct.count('E'), 'F' : ct.count('F'),
                    'G' : ct.count('G'), 'H' : ct.count('H'), 'I' : ct.count('I'),
                    'J' : ct.count('J'), 'K' : ct.count('K'), 'L' : ct.count('L'),
                    'M' : ct.count('M'), 'N' : ct.count('N'), 'O' : ct.count('O'),
                    'P' : ct.count('P'), 'Q' : ct.count('Q'), 'R' : ct.count('R'),
                    'S' : ct.count('S'), 'T' : ct.count('T'), 'U' : ct.count('U'),
                    'V' : ct.count('V'), 'W' : ct.count('W'), 'X' : ct.count('X'),
                    'Y' : ct.count('Y'), 'Z' : ct.count('Z')}
    return freqAlphabet

def englishFrequency(): #from wikipedia
    freqEnglish = {'A': 8.17, 'B': 1.29, 'C': 2.78, 'D': 4.25,'E': 12.70,
                   'F': 2.23, 'G': 2.02, 'H': 6.09, 'I': 6.97, 'J': 0.15,
                   'K': 0.77, 'L': 4.03,  'M': 2.41, 'N': 6.75, 'O': 7.51,
                   'P': 1.93, 'Q': 0.10, 'R': 5.99, 'S': 6.33, 'T': 9.06,
                   'U': 2.76, 'V': 0.98, 'W': 2.36, 'X': 0.15, 'Y': 1.97,
                   'Z': 0.07}
    return freqEnglish

def maxValKey(dictionary):
```

```
    return max(dictionary, key=dictionary.get)

def mostFrequent(frequency1, frequency2, frequency3, englishFrequency):
    mostFrequent1 = maxValKey(frequency1) #most frequent character in ciphertext for 1st key letter
    mostFrequent2 = maxValKey(frequency2) #most frequent character in ciphertext for 2nd key letter
    mostFrequent3 = maxValKey(frequency3) #most frequent character in ciphertext for 3rd key letter
    mostFrequentEnglish = maxValKey(englishFrequency) #most frequent character in ciphertext for 3rd key letter
    return mostFrequent1, mostFrequent2, mostFrequent3, mostFrequentEnglish

def findShift(mostFrequent1, mostFrequent2, mostFrequent3, mostFrequentEnglish):
    alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
    # +1 due to indexing from 0
    position1 = alphabet.find(mostFrequent1) + 1
    position2 = alphabet.find(mostFrequent2) + 1
    position3 = alphabet.find(mostFrequent3) + 1
    positionAlphabet = alphabet.find(mostFrequentEnglish) + 1
    shift1 = positionAlphabet - position1
    shift2 = positionAlphabet - position2
    shift3 = positionAlphabet - position3
    return shift1, shift2, shift3

def caeserShift(string, shift):
  cipher = ''
  for char in string: #Check spaces
    if char == ' ':
      cipher = cipher + char
    elif  char.isupper(): #Upper case shift
      cipher = cipher + chr((ord(char) + shift - 65) % 26 + 65)
    else: #Lower case shift
      cipher = cipher + chr((ord(char) + shift - 97) % 26 + 97)
  return cipher

#rebuild words
def rebuildPlaintext(shifted1, shifted2, shifted3):
    plaintext = ''
    for i in range(0, len(shifted1)):
      plaintext += shifted1[i]
      plaintext += shifted2[i]
      plaintext += shifted3[i]
    return plaintext

def removeKey(dictionary, key):
    del dictionary[key]

def nextMostFrequent(frequency3):
    mostFrequent3 = maxValKey(frequency3) #find most frequent
    removeKey(frequency3, mostFrequent3) #remove most frequent (we know it didnt work)
    nextMostFrequentVal = maxValKey(frequency3) #get the next most frequent
```

```
    return nextMostFrequentVal

def findShift3(nextMostFrequentVal, mostFrequentEnglish):
    alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
    position3 = alphabet.find(nextMostFrequentVal) + 1
    positionAlphabet = alphabet.find(mostFrequentEnglish) + 1
    newShift3 = positionAlphabet - position3
    return newShift3

def findKey(shift1, shift2, shift3):
    reversed = 'ZYXWVUTSRQPONMLKJIHGFEDCBA' #reversed as keys have been
        calculated from that perspective (handles minus [-] signs)
    theKey = ''
    theKey += reversed[shift1-1]
    theKey += reversed[shift2-1]
    theKey += reversed[shift3-1]
    return theKey.lower() #lowercase

def exportSolution(punctuatedPlaintext, theKey):
    open_file = open("q1-solution.txt", "w")
    open_file.write('Plaintext: ' + punctuatedPlaintext + '\n\nKey: ' + theKey)
    open_file.close()

### MAIN PROGRAM
ciphertext = loadFile()
print('\nStarting Vigenere Decipher Attempt...\n')
print('>Ciphertext: \n' + ciphertext + '\n')

# START KASISKY TEST
ciphertext = stripPunctuation(ciphertext)
print('>Removing spaces and punctuation: \n' + ciphertext + '\n')

print('>Calculating...\n')

largestRepeatedPattern = largestPattern(ciphertext)
print('>Largest repeated string: ' + largestRepeatedPattern)

stringLength = len(largestRepeatedPattern)
print('>Length of repeated string: ' + str(stringLength))

distanceBetweenRepetitions = repeatDistance(ciphertext, largestRepeatedPattern,
    stringLength)
print('>Distance between repetitions: ' + str(distanceBetweenRepetitions))

factors = findFactors(distanceBetweenRepetitions)
# print('>Possible key lengths: ' + colored(str(factors), 'red'))
print('>Possible key lengths: ' + str(factors))

likelyKeys = removeSmallKeys(factors)
print('>Likely keys: ' + str(likelyKeys))
print('>Attempt frequency analysis with key space of ' + str(likelyKeys[0]))
print('>Move to next key space listed if it fails')
#END OF KASISKY TEST
```

```

#Attempting first likely keylength (i.e. 3)
keyLength = 3
keyLetter1, keyLetter2, keyLetter3 = getEncryptedGroups(ciphertext, keyLength)
# print(keyLetter1)
# print(keyLetter2)
# print(keyLetter3)

#Frequency analysis
frequency1 = letterFrequency(keyLetter1)
frequency2 = letterFrequency(keyLetter2)
frequency3 = letterFrequency(keyLetter3)
englishFrequency = englishFrequency()

#graphical representation of the analysis
x1, y1 = zip(*frequency1.items())
x2, y2 = zip(*frequency2.items())
x3, y3 = zip(*frequency3.items())
xe, ye = zip(*englishFrequency.items())
plt.subplot(3,1,1)
plt.plot(x1, y1, xe, ye)
plt.title('Comparison of freqency of letters in CT (blue) and English (orange)')
plt.ylabel('frequency')
plt.subplot(3,1,2)
plt.plot(x2, y2, xe, ye)
plt.ylabel('frequency')
plt.subplot(3,1,3)
plt.plot(x3, y3, xe, ye)
plt.xlabel('letter')
plt.ylabel('frequency')
plt.show(block=False) #prevents the graph from plotting here to stop blocking of program

#find most frequent values
mostFrequent1, mostFrequent2, mostFrequent3, mostFrequentEnglish = mostFrequent(frequency1, frequency2, frequency3, englishFrequency)

#examine the shift amounts
shift1, shift2, shift3 = findShift(mostFrequent1, mostFrequent2, mostFrequent3, mostFrequentEnglish)
# print(shift1)
# print(shift2)
# print(shift3)

#shift
shifted1 = caeserShift(keyLetter1, shift1)
shifted2 = caeserShift(keyLetter2, shift2)
shifted3 = caeserShift(keyLetter3, shift3)
# print(shifted1)
# print(shifted2)
# print(shifted3)

#get plaintext
```

```
plaintext = rebuildPlaintext(shifted1, shifted2, shifted3)
print('\n >Plaintext Decipher Attempt Produces:\n' + plaintext)
print(' >This is close to resembling english but every third letter is not quite
    right')

#find the next most frequent value and therefore shift for the 3rd set of letters
nextMostFrequentVal = nextMostFrequent(frequency3)
newShift3 = findShift3(nextMostFrequentVal, mostFrequentEnglish)
newShifted3 = caeserShift(keyLetter3, newShift3)

#finally rebuild the cipher to show the final rebuildPlaintext
plaintext = rebuildPlaintext(shifted1, shifted2, newShifted3)
print('\n >2nd attempt to decipher gives:\n' + plaintext)
print(' >This is now an english sentence, just requiring the spaces and
    punctuation to be re-entered\n')
# manual addition of punctuation through inspection of the ciphertext
punctuatedPlaintext = 'Formative assessment can be viewed as a mean to enhance
    the learning process. Based on the results of such assessments, students will
    be able to assess their knowledge and identify strengths and weaknesses. The
    teacher will also have indication on how well the students are grasping the
    fundamental facts and whether he needs to alter their teaching to emphasis
    some important concepts.'
print(' >Formatting this as shown in the original Ciphertext gives:\n' +
    punctuatedPlaintext + '\n')

print(' >The key is therefore of length 3. The shifts used to decipher the text
    allows the key to be recovered as displayed below')
theKey = findKey(shift1, shift2, newShift3)
print(' >Key: ' + theKey + '\n')

exportSolution(punctuatedPlaintext, theKey)

#leave as last line to show the graphs
plt.show()
```

q1.py Command Line Output

```

Starting Vigenere Decipher Attempt...

 >Ciphertext:
Ymlfynbty tqmxqmfchm aug zy ogypcx tq u fcug ri xlbtlwx rbx jytphbla ipivcml.
    Zulcx hl nac lxqoerm hd mnab tqmxqmfchmq, mmsxxlnl ucej vx yvec nh ymlcml
    rbxgl dlipjywey tlx bbygrcyw mmpygenaq ugb qxyegcmlcm. Mfy mcuvfyk ucej ueqi
    aypx ghwgwtrchl ig fip uyej nac mmsxxlnl ylx eltqjbla mfy yshwygxlntj ztanl
    yhw ubxrbxp bx lyxbm mm ueryk rbxgl mcuvfcge nh cgifulgm lmgx ggimlmyhm
    aigayirm.

 >Removing spaces and punctuation:
YmlfynbtytqmxqmfchmaugzyogypcxtqufcugrixlbtlwxrbxjytphblaipivcmlZulcxhlnaclxqoermhdmnabtqmxqmfc


 >Calculating...

 >Largest repeated string: tqmxqmfchm
 >Length of repeated string: 10
 >Distance between repetitions: 78
 >Possible key lengths: [1, 2, 3, 6, 13, 26, 39, 78]
 >Ignoring keys < 3 in length as they are highly unlikely
 >Likely keys: [3, 6, 13, 26, 39, 78]
 >Attempt frequency analysis with key space of 3
 >Move to next key space listed if it fails

 >Plaintext Decipher Attempt Produces:
FodmafivqaseesemeztcmnbqviqwepasmmemntaentanoettelqarzinspraceesBmseponfhedesglteofeuctaseese

 >This is close to resembling english but every third letter is not quite right

 >2nd attempt to decipher gives:
FormativeassessmentcanbeviewedasameantoenhancethelearningprocessBasedontheresultsofsuchassessm

 >This is now an english sentence, just requiring the spaces and punctuation to
     be re-entered

 >Formatting this as shown in the original Ciphertext gives:
Formative assessment can be viewed as a mean to enhance the learning process.
    Based on the results of such assessments, students will be able to assess
    their knowledge and identify strengths and weaknesses. The teacher will also
    have indication on how well the students are grasping the fundamental facts
    and whether he needs to alter their teaching to emphasis some important
    concepts.

 >The key is therefore of length 3. The shifts used to decipher the text allows
     the key to be recovered as displayed below
 >Key: tyu
```

Appendix B Code for solving Cipher 2

q2.py

```
# Author: Dominic Heaton
# Solution Script for Q2
# XOR Cipher Decryption Attempt
# Given first letter of Plaintext is 'I' (capital i), first byte of key can be
#   determined. Brute force of second byte reveals plaintext.
##########################################################################################
from langdetect import detect
from guess_language import guess_language
from termcolor import colored
import re
import matplotlib.pylab as plt
import operator
import binascii

def loadFile():
    open_file = open("q2.txt", "r")
    ciphertext = open_file.read().rstrip('\n')
    open_file.close()
    # print(ciphertext)
    return ciphertext

def hexStringToInt(hex):
    return int(hex,16)

def intToAscii(number):
    return chr(number)

def asciiToInt(string):
    return ord(string)

def exportSolution(plaintext, theKey):
    open_file = open("q2-solution.txt", "w")
    open_file.write('Plaintext: ' + plaintext + '\n\nKey: ' + theKey)
    open_file.close()

### MAIN PROGRAM
ciphertext = loadFile()
ciphertext = ciphertext.replace(" ", "")
print('\nStarting Decipher Attempt...\n')
print(' >Ciphertext: \n' + ciphertext + '\n')
key = []

#find key for first letter of text using the hint "I"
cipherByte = ciphertext[:2] #first hex byte
print(' >XOR is reversible. We know the first letter of plaintext is \'I\' and
    can therefore reverse to find the key')
hintLetter = asciiToInt("I")
key.append(hexStringToInt(cipherByte)   hintLetter)
print(' >Key to give \'I\' as plaintext: ' + hex(key[0]))
```

```
print(' >This is proved by encrypting \'I\' with key \'0x1a\' to give first ascii letter of ciphertext: ' + intToAscii(hintLetter  key[0]))

#attempting to decipher remaining plaintext using the key 0x1a
print('\n >Attempting decryption of the ciphertext using this 1-byte key (0x1a):')
plaintext = 'I'
for i in range(2, len(ciphertext), 2):
    cipherByte = ciphertext[i:i+2]
    plaintext += intToAscii(hexStringToInt(cipherByte)  key[0])
print(plaintext)
print('\n >It is clear decryption using the 1-byte key is unsuccessful at revealing plaintext')

#attempting to decipher remaining plaintext using the key 0x1a
print(' >Attempting decryption of the ciphertext using a 2-byte key of form \'1aXX\' where XX is determined by brute force:\n')
plaintext = ''
plaintext2 = ''
decipheredPlaintext = ''

#Decipher ciphertext bytes 1,3,5,7,... with key 0x1a determined before
plaintext = 'I'
for i in range(4, len(ciphertext), 4):
    cipherByte = ciphertext[i:i+2]
    plaintext += intToAscii(hexStringToInt(cipherByte)  key[0])

#Decipher ciphertext bytes 2,4,6,8,... with potential keys from 0-256
for i in range(0,256):
    for j in range(2, len(ciphertext), 4):
        cipherByte = ciphertext[j:j+2]
        plaintext2 += intToAscii(hexStringToInt(cipherByte)  i)
    #Concatenate together solutions and print potential plaintexts
    for k in range(0,len(plaintext2)):
        decipheredPlaintext += plaintext[k]
        decipheredPlaintext += plaintext2[k]
    #limit printing of plaintext to just english language
    if detect(decipheredPlaintext) == 'en': #check for english
        if guess_language(decipheredPlaintext) == 'en': #check for english
            print(' > Potential plaintext no.' + str(i) + ' using 2-byte key: 1a' + hex(i).replace('0x',''))
            print(decipheredPlaintext + '\n')
    #reset to blank for next iteration
    plaintext2 = ''
    decipheredPlaintext = ''

print(' >Here we can see a single plaintext (no.188) that reads in english using the hexadecimal key of 0x1abc')
print(' >The plaintext reads:')
decipheredPlaintext = 'In ancient Egypt servants were smeared with honey to attract flies away from the pharaoh'
theKey = '0x1abc'
print(colored(decipheredPlaintext + '\n','red'))
```

```
exportSolution(decipheredPlaintext, theKey)
```

q2.py Command Line Output

```
Starting Decipher Attempt...

 >Ciphertext:
53D23ADD74DF73D974C83AF97DC56AC83ACF7FCE6CDD74C8699C6DD968D93ACF77D97BCE7FD83ACB73C8729C72D374D9639C6ED33ADD6EC868DD

 >XOR is reversible. We know the first letter of plaintext is 'I' and can therefore reverse to find the key
 >Key to give 'I' as plaintext: 0x1a
 >This is proved by encrypting 'I' with key '0x1a' to give first ascii letter of ciphertext: S

 >Attempting decryption of the ciphertext using this 1-byte key (0x1a):
IÈ ÇnÅiÅnÒ ägßpÒ ÖeÒvÇnÒswÅrÅ ÒmÅaÒeÅ Ñ1ÒhhÉnÅytÉ ÇtÒrÇcÒ ÀlÏeÒ ÇwÇyfÒoË ÒhÅ ÖhÇrÇoÎ¬

 >It is clear decryption using the 1-byte key is unsuccessful at revealing plaintext
 >Attempting decryption of the ciphertext using a 2-byte key of form '1aXX' where XX is determined by brute force:

 > Potential plaintext no.138 using 2-byte key: 1a8a
IX WnUiSnB sgOpB EeDvWnBswSrS EmSaDeR AiBhhYnSytY WtBrWcB Pl_eE WwWyfDo[ BhS FhWrWo^<

 > Potential plaintext no.139 using 2-byte key: 1a8b
IY VnTiRnC rgNpC DeEvVnCswRrR DmRaEeS @iChhXnRytX VtCrVcC Ql^eD VwVyfEoZ ChR GhVrVo_=

 > Potential plaintext no.141 using 2-byte key: 1a8d
I_ PnRiTnE tgHpE BeCvPnEswTrT BmTaCeU FiEhh^nTyt^ PtErPcE WlXeB PwPyfCo\ EhT AhPrPoY;

 > Potential plaintext no.148 using 2-byte key: 1a94
IF InKiMn\ mgQp\ [eZvIn\swMrM [mMaZeL _i\hhGnMytG It\rIc\ NlAe[ IwIyfZoE \hM XhIrIo@"

 > Potential plaintext no.150 using 2-byte key: 1a96
ID KnIiOn^ ogSp^ YeXvKn^s
wOrO YmOaXeN ]i^h
hEnOy
tE Kt^rKc^ LlCeY KwKy
fXoG ^hO ZhKrKoB

 > Potential plaintext no.152 using 2-byte key: 1a98
IJ EnGiAnP ag]pP WeVvEnPswArA WmAaVe@ SiPhhKnAytK EtPrEcP BlMeW EwEyfVoI PhA ThErEoL.

 > Potential plaintext no.154 using 2-byte key: 1a9a
IH GnEiCnR cg_pR UeTvGnRswCrC UmCaTeB QiRhhInCytI GtRrGcR @lOeU GwGyfToK RhC VhGrGoN,

 > Potential plaintext no.155 using 2-byte key: 1a9b
II FnDiBnS bg^pS TeUvFnSswBrB TmBaUeC PiShhHnBytH FtSrFcS AlNeT FwFyfUoJ ShB WhFrFoO-

 > Potential plaintext no.156 using 2-byte key: 1a9c
IN AnCiEnT egYpT SeRvAnTswErE SmEaReD WiThhOnEytO AtTrAcT FlIeS AwAyfRoM ThE PhArAoH*

 > Potential plaintext no.159 using 2-byte key: 1a9f
IM Bn@iFnW fgZpW PeQvBnWswFrF PmFaQeG TiWhhLnFytL BtWrBcW ElJeP BwByfQoN WhF ShBrBoK)

 > Potential plaintext no.171 using 2-byte key: 1aab
Iy vntirnc Rgnpc deevvncs7wrrr dmraees `ich7hxnry7tx vtcrvcc gl~ed vwvy7feoz chr ghvrvo

 > Potential plaintext no.173 using 2-byte key: 1aad
I pnritne Tghpe becvpneslwtrt bmtaceu fiehlh~ntylt~ pterpce wlxeb pwpylfco| eht ahprpoy

 > Potential plaintext no.180 using 2-byte key: 1ab4
If inkimn| Mgqp| {ezvin|s(wmrm {mmazel i|h(hgnmy(tg it|ric| nlae( iwiy(fzoe |hm xhirio`

 > Potential plaintext no.182 using 2-byte key: 1ab6
Id kniion~ Ogsp~ yexvkn~s*woro ymoaxen }i~h*henoy*te kt~rkc~ llcey kwky*fxog ~ho zhkrkob

 > Potential plaintext no.184 using 2-byte key: 1ab8
Ij engianp Ag}pp wevvenps$wara wmaave` siph$hknay$tk etprecp blmew ewey$fvoi pha thereol

 > Potential plaintext no.186 using 2-byte key: 1aba
Ih gneicnr Cgpr uetvgnrs&wcrc umcateb qirh&hincy&ti gtrrgcr `loeu gwgy&ftok rhc vhgrgon

 > Potential plaintext no.187 using 2-byte key: 1abb
Ii fndibns Bg~ps teuvfnss'wbrb tmbauec pish'hhnby'th ftsrfcs alnet fwfy'fuoj shb whfrfoo

 > Potential plaintext no.188 using 2-byte key: 1abc
In ancient Egypt servants were smeared with honey to attract flies away from the pharaoh


 >Here we can see a single plaintext (no.188) that reads in english using the hexadecimal key of 0x1abc
 >The plaintext reads:
In ancient Egypt servants were smeared with honey to attract flies away from the pharaoh
```

Appendix C Code for solving Cipher 3

railTest.py

```
# Author: Dominic Heaton
# Rail Fence Transpoition Cipher Test Script
#   Encrypts message 'thisisatestoftherailfencedecoder' using max number of
#   possible rails for length of the message
#   Decrypts the ciphertest also to show it can be returned to original text
###################################################################################
from langdetect import detect
from guess_language import guess_language
from termcolor import colored
import re
import matplotlib.pylab as plt

def findFactors(value):
    factors = []
    for i in range(1, value+1):
        if value % i == 0:
            factors.append(i)
    return factors

def railEncrypt(plaintext, numberOfRails):
    #generate matrix
    railMatrix = []
    for i in range(numberOfRails):
        railMatrix.append([])
    for row in range(numberOfRails):
        for column in range(len(plaintext)):
            railMatrix[row].append('.')

    #assign plaintext to matrix
    row = 0
    check = 0
    for i in range(len(plaintext)):
        if check == 0:
            railMatrix[row][i] = plaintext[i]
            row += 1
        if row == numberOfRails:
            check = 1
            row -= 1
        elif check == 1:
            row -= 1
            railMatrix[row][i] = plaintext[i]
            if row == 0:
                check = 0
                row = 1

    #form ciphertext from matrix
    ciphertext = ''
    for i in range(numberOfRails):
        for j in range(len(plaintext)):
```

```python
            ciphertext += railMatrix[i][j]
    ciphertext = re.sub(r"\.","",ciphertext)
    return ciphertext

def railDecrypt(ciphertext, numberOfRails):
    #generate matrix
    railMatrix = []
    for i in range(numberOfRails):
        railMatrix.append([])
    for row in range(numberOfRails):
        for column in range(len(ciphertext)):
            railMatrix[row].append('.')

    #assign ciphertext to matrix
    row = 0
    check = 0
    for i in range(len(ciphertext)):
        if check == 0:
            railMatrix[row][i] = ciphertext[i]
            row += 1
            if row == numberOfRails:
                check = 1
                row -= 1
        elif check == 1:
            row -= 1
            railMatrix[row][i] = ciphertext[i]
            if row == 0:
                check = 0
                row = 1

    #sort matrix
    value = 0
    for i in range(numberOfRails):
        for j in range(len(ciphertext)):
            tempVal = railMatrix[i][j]
            if re.search("\\.", tempVal):
                continue
            else:
                railMatrix[i][j] = ciphertext[value]
                value += 1

    #form plaintext from matrix
    check = 0
    row = 0
    plaintext = ''
    for i in range(len(ciphertext)):
        if check == 0:
            plaintext += railMatrix[row][i]
            row += 1
            if row == numberOfRails:
                check = 1
                row -= 1
        elif check == 1:
```

```
            row -=1
            plaintext += railMatrix[row][i]
            if row == 0:
                check = 0
                row = 1
    plaintext = re.sub(r"\.","",plaintext)
    return plaintext

msg = 'thisisatestoftherailfencedecoder'
msgLength = str(len(msg))
keyLengths = str(findFactors(int(msgLength)))
print('\nStarting Rail Fence Decipher Attempt...\n')
print(' >Message: \n' + msg + '\n')
print(' >Length of Ciphertext: ' + msgLength)
maxNumberOfRails = str(int(msgLength)-1)
print(' >Max number of Rails is therefore: ' + maxNumberOfRails + '\n') #max rail number is one less than length of text

#brute force to show encrypted and decrypted for max number of rails possible for length of message
for numberOfRails in range(2, int(maxNumberOfRails)+1):
    print(' >No. of Rails: ' + str(numberOfRails))
    ciphertext = railEncrypt(msg, numberOfRails)
    print(' >Ciphertext: ' + ciphertext)
    plaintext = railDecrypt(ciphertext, numberOfRails)
    print(' >Plaintext: ' + plaintext + '\n')
```

rail.py

```python
# Author: Dominic Heaton
# Rail Fence Transpoition Cipher Decoder Attempt
################################################################################
from langdetect import detect
from guess_language import guess_language
from termcolor import colored
import re
import matplotlib.pylab as plt

def loadFile():
    open_file = open("q3.txt", "r")
    ciphertext = open_file.read().rstrip('\n')
    open_file.close()
    # print(ciphertext)
    return ciphertext

def findFactors(value):
    factors = []
    for i in range(1, value+1):
        if value % i == 0:
            factors.append(i)
    return factors

def railEncrypt(plaintext, numberOfRails):
    #generate matrix
    railMatrix = []
    for i in range(numberOfRails):
        railMatrix.append([])
    for row in range(numberOfRails):
        for column in range(len(plaintext)):
            railMatrix[row].append('.')

    #assign plaintext to matrix
    row = 0
    check = 0
    for i in range(len(plaintext)):
        if check == 0:
            railMatrix[row][i] = plaintext[i]
            row += 1
        if row == numberOfRails:
            check = 1
            row -= 1
        elif check == 1:
            row -= 1
            railMatrix[row][i] = plaintext[i]
            if row == 0:
                check = 0
                row = 1

    #form ciphertext from matrix
    ciphertext = ''
```

```
    for i in range(numberOfRails):
        for j in range(len(plaintext)):
            ciphertext += railMatrix[i][j]
    ciphertext = re.sub(r"\.","",ciphertext)
    return ciphertext

def railDecrypt(ciphertext, numberOfRails):
    #generate matrix
    railMatrix = []
    for i in range(numberOfRails):
        railMatrix.append([])
    for row in range(numberOfRails):
        for column in range(len(ciphertext)):
            railMatrix[row].append('.')

    #assign ciphertext to matrix
    row = 0
    check = 0
    for i in range(len(ciphertext)):
        if check == 0:
            railMatrix[row][i] = ciphertext[i]
            row += 1
            if row == numberOfRails:
                check = 1
                row -= 1
        elif check == 1:
            row -= 1
            railMatrix[row][i] = ciphertext[i]
            if row == 0:
                check = 0
                row = 1

    #sort matrix
    value = 0
    for i in range(numberOfRails):
        for j in range(len(ciphertext)):
            tempVal = railMatrix[i][j]
            if re.search("\\.", tempVal):
                continue
            else:
                railMatrix[i][j] = ciphertext[value]
                value += 1

    #form plaintext from matrix
    check = 0
    row = 0
    plaintext = ''
    for i in range(len(ciphertext)):
        if check == 0:
            plaintext += railMatrix[row][i]
            row += 1
            if row == numberOfRails:
                check = 1
```

```
                row -= 1
        elif check == 1:
            row -=1
            plaintext += railMatrix[row][i]
            if row == 0:
                check = 0
                row = 1
    plaintext = re.sub(r"\.","",plaintext)
    return plaintext

ciphertext = loadFile()
cipherLength = str(len(ciphertext))
keyLengths = str(findFactors(int(cipherLength)))
print('\nStarting Rail Fence Decipher Attempt...\n')
print(' >Ciphertext: \n' + ciphertext + '\n')
print(' >Length of Ciphertext: ' + cipherLength)
maxNumberOfRails = str(int(cipherLength)-1)
print(' >Max number of Rails is therefore: ' + maxNumberOfRails + '\n') #max rail number is one less than length of text

#brute force to show decrypted cipher for max number of rails possible for length of ciphertext
for numberOfRails in range(2, int(maxNumberOfRails)+1):
    print(' >No. of Rails: ' + str(numberOfRails))
    plaintext = railDecrypt(ciphertext, numberOfRails)
    print(' >Decrypted Plaintext: ' + plaintext + '\n')
    if detect(plaintext) == 'en': #attempt to limit output via language detection (not perfect)
        if guess_language(plaintext) == 'en': #attempt to limit output via language detection (not perfect)
            print(colored(' >English Recognised', 'red'))
```

q3.py

```
# Author: Dominic Heaton
# Solution Script for Q3
# Columnar Transposition for a range of keys   3 in length
# Brute force attack which limits the potential plaintexts it outputs through
#   english language detection. Detection scans first 3 letters, then first 5
#   letters and then the entire potential plaintext looking for english words.
#   On satisfying the english detection the plaintexts are exported to a file
#   as well as the command line
##########################################################################
from langdetect import detect
from guess_language import guess_language
from termcolor import colored
import re
import matplotlib.pylab as plt
import itertools
import os

def loadFile():
    open_file = open("q3.txt", "r")
    ciphertext = open_file.read().rstrip('\n')
    open_file.close()
    # print(ciphertext)
    return ciphertext

def removeOldExport():
    try:
        os.remove('q3-plaintexts.txt')
    except OSError:
        pass

def findFactors(value):
    factors = []
    for i in range(1, value+1):
        if value % i == 0:
            factors.append(i)
    return factors

def removeSmallKeys(keyLengths):
    print(' >Ignoring keys < 3 in length as they are highly unlikely')
    keyLengths.remove(1)
    keyLengths.remove(2)
    return keyLengths

def letterFrequency(ct):
    ct = ct.upper() #make all upper case
    freqAlphabet = {'A' : ct.count('A'), 'B' : ct.count('B'), 'C' : ct.count('C')
        ,
                    'D' : ct.count('D'), 'E' : ct.count('E'), 'F' : ct.count('F')
                        ,
                    'G' : ct.count('G'), 'H' : ct.count('H'), 'I' : ct.count('I')
                        ,
```

```
                    'J' : ct.count('J'), 'K' : ct.count('K'), 'L' : ct.count('L'),
                    'M' : ct.count('M'), 'N' : ct.count('N'), 'O' : ct.count('O'),
                    'P' : ct.count('P'), 'Q' : ct.count('Q'), 'R' : ct.count('R'),
                    'S' : ct.count('S'), 'T' : ct.count('T'), 'U' : ct.count('U'),
                    'V' : ct.count('V'), 'W' : ct.count('W'), 'X' : ct.count('X'),
                    'Y' : ct.count('Y'), 'Z' : ct.count('Z')}
    return freqAlphabet

def englishFrequency(): #from wikipedia
    freqEnglish = {'A': 8.17, 'B': 1.29, 'C': 2.78, 'D': 4.25,'E': 12.70,
                'F': 2.23, 'G': 2.02, 'H': 6.09, 'I': 6.97, 'J': 0.15,
                'K': 0.77, 'L': 4.03,  'M': 2.41, 'N': 6.75, 'O': 7.51,
                'P': 1.93, 'Q': 0.10, 'R': 5.99, 'S': 6.33, 'T': 9.06,
                'U': 2.76, 'V': 0.98, 'W': 2.36, 'X': 0.15, 'Y': 1.97,
                'Z': 0.07}
    return freqEnglish

def columnarDecrypt(ciphertext, likelyKeys):
    print('Columnar Decrypting...\n')
    for i in range(0, 4): #INCREASE THIS NUMBER FOR MORE ITERATIONS
        print(' >Key Length: ' + str(likelyKeys[i]))
        numberRows = int(len(ciphertext)/likelyKeys[i])
        print(' >Number of Rows: ' + str(numberRows))
        # generateMatrix(ciphertext, numberRows, likelyKeys[i])
        matrix = cipherToMatrix(ciphertext, numberRows)
        # print(matrix)
        matrix = transposeMatrix(matrix)
        # print(matrix)
        plaintextArray = matrixToPlaintext(matrix, likelyKeys[i], numberRows)
        printPlaintexts(plaintextArray, likelyKeys[i])
        # exportPlaintexts(plaintextArray, likelyKeys[i])

def cipherToMatrix(ciphertext, keyLength):
    return [ciphertext[i:i+keyLength] for i in range(0, len(ciphertext), keyLength)]

def transposeMatrix(matrix): #matrix of rows: key=3 gives a matrix of strings of length 3
    return [*zip(*matrix)]

def matrixToPlaintext(matrix, keyLength, numberRows): #matrix of columns: key=3 gives 3 strings
    plaintextArray = []
    tempList = []
    tempString = ''
    stringstring = ''
    combinedList = []
    numberOfColumns = list(range(0,keyLength)) #list number of columns
```

```python
    columnCombinations = list(set(itertools.permutations(numberOfColumns))) #list of all column permutations
    # print('>Column permutations to attempt: ' + str(columnCombinations) + '\n')
    for i in range(0, len(columnCombinations)): #for iterating through column combinations list
        for j in range(0, keyLength): #for iterating through individual columns in combinations
            tempList = [row[columnCombinations[i][j]] for row in matrix] #fetch column in matrix
            # print(tempList)
            tempString += ''.join(tempList) #turn column into string
            if j == (keyLength-1): #if we've got all the columns in the combination
                formattedString = fixString(tempString, keyLength, numberRows)
                # print('>>>Formatted String: ' + formattedString)
                plaintextArray.append(formattedString) #add string to plaintext array
                # print(plaintextArray)
                tempString = ''
    return plaintextArray

#before function we have the columns concatentated as a string <column0><column1><column2>
#after function we have a string which reads the rows in the order 0,1,2 (or whichever order it has been passed)
def fixString(tempString, keyLength, numberRows):
    lengthCipher = keyLength * numberRows # lengthCipher=96=len(tempString), rows and key change
    formattedString = ''
    for i in range(0, numberRows):
        formattedString += tempString[i::numberRows] #start at index i, appending every 'numberRows' value
    # print(formattedString)
    return formattedString

#1053 matches found when scanning for english only in entire plaintext
#2154 matches found when scanning just first 3 letters of word
#205 matches found when scanning first 3 letters and then entire plaintext
#48 matches found when scanning first 3, then 5, then entire plaintext
def printPlaintexts(plaintextArray, keyLength):
    print('>Potential plaintexts for key length of ' + str(keyLength) + ':')
    for i in range(0, len(plaintextArray)):
        if detect(plaintextArray[i][:3]) == 'en': #check for english word in first 3 letters
            if guess_language(plaintextArray[i][:3]) == 'en': #check for english word in first 3 letters
                if detect(plaintextArray[i][:5]) == 'en': #check for english word in first 5 letters
                    if guess_language(plaintextArray[i][:5]) == 'en': #check for english word in first 5 letters
                            if detect(plaintextArray[i]) == 'en': #check for english word in entire phrase
```

```
                        if guess_language(plaintextArray[i]) == 'en': #check for english word in entire phrase
                            print('   ' + str(i+1) + '. ' + plaintextArray[i])
                            exportToFile(plaintextArray[i])
    print() #blank line

def exportToFile(plaintext):
    open_file = open("q3-plaintexts.txt", "a")
    open_file.write(plaintext + '\n')
    open_file.close()

### MAIN PROGRAM
ciphertext = loadFile()
removeOldExport()
cipherLength = str(len(ciphertext))
keyLengths = findFactors(int(cipherLength))
print('\nStarting Decipher Attempt...\n')
print(' >Ciphertext: \n' + ciphertext + '\n')
print(' >Length of Ciphertext: ' + cipherLength)
print(' >Key lengths: ' + str(keyLengths) + '\n')
likelyKeys = removeSmallKeys(keyLengths)
print(' >Likely keys: ' + str(likelyKeys) + '\n')

# FREQUENCY ANALYSIS
ciphertextFrequency = letterFrequency(ciphertext)
englishFrequency = englishFrequency()

#graphical representation of the analysis
x1, y1 = zip(*ciphertextFrequency.items())
xe, ye = zip(*englishFrequency.items())
plt.plot(x1, y1, xe, ye)
plt.title('Comparison of freqency of letters in CT (blue) and English (orange)')
plt.xlabel('letter')
plt.ylabel('frequency')
plt.show(block=False)

#Columnar Transposition Attempt
columnarDecrypt(ciphertext, likelyKeys)
print(' >From this list it can be seen from inspection that potential plaintext 26593 can be punctuated to read:')
print(colored('There are probably more than hundred billion galaxies in the cosmos each of those has up to trillion stars','red'))
#leave as final line - plot graph
plt.show()
```

q3.py Command Line Output

```

Starting Decipher Attempt...

 >Ciphertext:
eynbanotaulsabhenssoaaiaeatroeohghrtrbediiccestsrladgimflslrhroulaheoitoeornlxeaseontpmhiltshxp


 >Length of Ciphertext: 96
 >Key lengths: [1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 48, 96]

 >Ignoring keys < 3 in length as they are highly unlikely
 >Likely keys: [3, 4, 6, 8, 12, 16, 24, 32, 48, 96]

Columnar Decrypting...

 >Key Length: 3
 >Number of Rows: 32
 >Potential plaintexts for key length of 3:

 >Key Length: 4
 >Number of Rows: 24
 >Potential plaintexts for key length of 4:

 >Key Length: 6
 >Number of Rows: 16
 >Potential plaintexts for key length of 6:

 >Key Length: 8
 >Number of Rows: 12
 >Potential plaintexts for key length of 8:
  227.
      haterererbpolybaohmranetuehndbdrlnilgaioaslxiniehstemocoeosaftchoahslaegiaxesushtipolltro

  642.
      haterererbpalobyohmtarenuehrdndblniogliaasleixinhstomecoeoshfactoahglseaiaxhsesutiprlotlo

  794.
      eartheerablproybthamorneredhunbdongillaiesilaxniosmtheochofseatcgalhosaehasxieusrilptoltt

  1353.
      therereaprybolabmonerathhubdndreilailgonlanixiesthocemossetcafhohoaeslgaxiuseshaptltolrii

  1370.
      earthereabbprolythemoranredhundboniillgaesilaxinoscthemohocseaftgaehoslahasxiesuritptollt

  1684.
      eaterehrobpabyrlrhmtenoanehrdbudlnioialgxsleinaiestocohmaoshctefsahgeaoleaxhsuisoiprtltln

  4181.
      earthereybbprolanhemoratbedhundraniillgonsilaxieoscthemotocseafhaaehoslguasxieshlitptolrs
```

37 6790.
thereaerprolybabmoranhtehundberdillganoilaxinseithemosocseaftohchoslaagexiesuahsptollirti

38 8480.
eaterhreobpalrbyrhmtaoennehrdudblniogliaxsleiainestomhcoaoshfectsahgloeaeaxhsisuoiprlttl

39 9351.
therearepr olabbymorathenhundredbillgonialaxiesinthemoscoseafhocthoslgaeaxieshasuptolritli

40 9804.
hatereerrbpolyabohmranteuehndbrdlnilgaoiaslxineihstemooceosafthcoahslageiaxesuhstipollrto

41 10349.
earthereablprobythamorenredhundbongilliaesilaxinosmthecohofseactgalhoseahasxiesurilptotlt

42 13633.
hatereerrbpaboylohmternauehrdnbdlnioilagasleixnihstoceomeoshcatfoahgesaliaxhseustiprtollo

43 14434.
therereaprobylabmorenathhundbdreilliagonlaxiniesthecomosseactfhohosealgaxiesushaptotllrii

44 14787.
thereearprolyabbmoranthehundbredillgaonilaxinesithemooscseafthochoslagaexiesuhasptollriti

45 15893.
thereearprolaybbmoratnhehundrbedillgoanilaxiensithemooscseafhtochoslgaaexieshuasptolrliti

46 16167.
hatereerrbpalyobohmtanreuehrdbndlniogaliasleinxihstomoeceoshftacoahglaseiaxhsuestiprllotc

47 17121.
thereearpryloabbmonarthehubdnredilaglonilanixesithomeoscsetfahochoalsgaexiusehasptlloriti

48 17295.
thereraeprolabbymoratehnhundrdebillgoinalaxieisnthemocsoseafhcothoslgeaaxieshsauptolrtili

49 17619.
therereaprolybabmoranethhundbdreillgaionlaxiniesthemocosseaftchohoslaegaxiesushaptolltrii

50 18127.
thereareprobyblamorenhathundbedrilliangolaxinsiethecosmoseactofhhoseaalgxiesuashptotlilri

51 19266.
haterreerbpabloyohmtearnuehrddnblnioiglaasleiixnhstocmeoeoshcfatoahgelsaiaxhsseutiprtlolc

52 19953.
thereearprobayblmoretnhahundrbedillioanglaxiensithecoosmseachtofhosegaalxieshuasptotrlili

53 20410.
thereaerprabobylmoterhnahurdnebdiloilnaglaeixsnithocesomsehcaotfhogesaalxihseausptrtoilli

54 21365.
eaterherabpylrobthmnaorerehbdundoniaglllieslniaxiostomhechostfeacgahalosehaxusiesriplltott

55 21467.
thereaerpralobybmotarhnehurdnebdiloglnailaeixsnithomesocsehfaotchoglsaaexihseausptrloilti

56 22805.
therearepraboblymoterhanhurdnedbiloilngalaeixsinthocesmosehcaofthogesalaxihseasuptrtoilli

57 23849.
thereareprybablomonetharhubdrednilaionglIaniesixthocosmesetchofahoaegalsxiushaseptltriloi

58 24192.
thereaerprobabylmorethnahundrebdillionaglaxiesnithecosomseachotfhosegaalxieshausptotrilli

59 25516.
thereearpryboablmonerthahubdnredilailonglanixesithoceosmsetcahofhoaesgalxiusehasptltorili

60 25973.
eaterehrobpybarlrhmnetoanehbdrudlniaiolgxslnieaiestocohmaostchefsahaegoleaxushisoipltrtlr

61 26214.
thereraeprobylbamorenahthundbderilliagnolaxinisethecomsoseactfohhosealagxiesusahptotllliri

62 26593.
thereareprobablymorethanhundredbilliongalaxiesinthecosmoseachofthosegalaxieshasuptotrilli

63 26824.
therereaprolabybmoratenhhundrdbeillgoianlaxieinsthemocosseafhctohoslgeaaxieshsuaptolrtlii

64 27366.
eaterherybpalrobnhmtaorebehrdundaniogllinsleiaxiostomhectoshfeacaahgloseuaxhsiesliprltots

65 28497.
eaterehrybpaborlnhmteroabehrdnudanioillgnsleixaiostocehmtoshcaefaahgesoluaxhseisliprtotls

66 29445.
thereraeprolybbamoranehthundbderillgainolaxinisethemocsoseaftcohhoslaeagxiesusahptollltiri

67 30818.
earthereyblprobanhamoretbedhundrangillionsilaxieosmthecotofseachaalhoseguasxieshlilptotrs

68 30908.
therereaprabylobmotenarhhurdbdneiloiaglnlaeinixsthocomessehctfaohogealsaxihsuseaptrtlloii

69 30930.
thereraeprabolbymoterahnhurdndebiloilgnalaeixisnthocemsosehcafothogeslaaxihsesauptrtolili

70 31478.
hatereerrbpaloybohmtarneuehrdnbdlnioglaiasleixnihstomeoceoshfatcoahglsaeiaxhseustiprloltc

71 32233.
hatererererbpylobaohmnaretuehbdndrlniaglioaslnixiehstomecoeostfachoahalsegiaxuseshtipllotrc

72 33851.

```
    hatereeerrbpyboalohmnertauehbdnrdlniailogaslnixeihstoceomeostcahfoahaesgliaxusehstipltorlo

  34720.
    thereraeprobalbymoretahnhundrdebilliognalaxieisnthecomsoseachfothoseglaaxieshsauptotrlili

  34869.
    thereaerprobybalmorenhtahundberdillianoglaxinseithecosomseactohfhoseaaglxiesuahsptotlirli

  37564.
    thereareprolybbamoranhethundbedrillganiolaxinsiethemoscoseaftochhoslaaegxiesuashptollitri

  37806.
    hatereeerrbpabyolohmtenrauehrdbndlnioialgasleinxihstocoemeoshctafoahgeasliaxhsuestiprtlolo

  38045.
    eatererhybpabolrnhmteraobehrdnduanioilglnsleixiaostocemhtoshcafeaahgeslouaxhsesiliprtolts

  38678.
    thereeearprobyablmorenthahundbredilliaonglaxinesithecoosmseacthofhoseagalxiesuhasptotlrili

  39400.
    eartheerabbproylthemornaredhunbdoniillagesilaxniosctheomhocseatfgaehosalhasxieusritptollt

  39619.
    thereeearpralyobbmotanrhehurdbnedilogalnilaeinxsithomoescsehftaochoglasaexihsueasptrlloiti

  39764.
    eaterehrabpyborlthmneroarehbdnudoniaillgeslnixaiostocehmhostcaefgahaesolhaxuseisripltotlt

  40121.
    thereaerprolabybmorathnehundrebdillgonailaxiesnithemosocseafhotchoslgaaexieshausptolrilti


 >From this list it can be seen from inspection that potential plaintext 26593
    can be punctuated to read:
There are probably more than hundred billion galaxies in the cosmos each of those
    has up to trillion stars
```

Appendix D Word Count

The total word count for each section was found using Texcount and the output is below showing that no individual section of the report is longer than the 500 word limit.

```
File: Crypto.tex
Encoding: ascii
Words in text: 1667
Words in headers: 39
Words outside text (captions, etc.): 207
Number of headers: 11
Number of floats/tables/figures: 7
Number of math inlines: 2
Number of math displayed: 1
Subcounts:
  text+headers+captions (#headers/#floats/#inlines/#displayed)
  88+1+0 (1/0/0/0) Section: Outline} \label{Outline
  66+4+0 (1/0/0/0) Section: Solution for Cipher 1} \label{Solution for Cipher 1
  500+3+85 (1/2/0/1) Section: Cipher 1 Cryptanalysis} \label{Cipher 1 Cryptanalysis
  19+4+0 (1/0/0/0) Section: Solution for Cipher 2} \label{Solution for Cipher 2
  423+3+35 (1/2/2/0) Section: Cipher 2 Cryptanalysis} \label{Cipher 2 Cryptanalysis
  24+4+0 (1/0/0/0) Section: Solution for Cipher 3} \label{Solution for Cipher 3
  497+3+87 (1/2/0/0) Section: Cipher 3 Cryptanalysis} \label{Cipher 3 Cryptanalysis
  7+5+0 (1/0/0/0) Section: Code for solving Cipher 1} \label{Code for solving Cipher 1
  5+5+0 (1/1/0/0) Section: Code for solving Cipher 2} \label{Code for solving Cipher 2
  7+5+0 (1/0/0/0) Section: Code for solving Cipher 3} \label{Code for solving Cipher 3
  31+2+0 (1/0/0/0) Section: Word Count} \label{Word Count
```