Aaron Fox
CECS 564-01
Dr. Desoky

Project 1: Vigenere Cipher

Introduction: An implementation of decryption, encryption, attacking, and evaluating cipher texts as related to the Vigenere cipher was successfully carried out in Project 1. My classmate, Read Hughes, supplied me with a file containing text of the first 256 Extended ASCII characters encrypted with the Vigenere cipher. An encrypted text file was created using the Vigenere cipher for Mr. Hughes to crack. A function to attack the encrypted file and uncover the keyword used to encrypt the file was then implemented. The decryption of the given file was then solved using a decryption of the Vigenere cipher function also implemented. Additional report questions given in the project prompt are appended to the end of this report.

Encrypting: The Vigenere Cipher's encryption was written in the Python. First, a text file was read in with the Unicode encoding, only allowing characters of Unicode between the values of 0 and 255, inclusive. This was done to ensure that only characters from the 256 extended ASCII values were used in the encryption process. The text was stored into a character array called text that contains every character in order. The code used for the IO is found below:

```
text_file = open(text_file_path, 'r', encoding='latin1')
# Read all info of  extended ASCII characters only
text = []
i = 0
while True:
    # Read one character at a time, including only chars of first 256 characters
    char = text_file.read(1)
    if not char:
        break
    if ord(char) > 0 and ord(char) < 256:
        text.append(char)
    else:
        print("Excluding char " + str(char) + " from reading of input file because
              it's not in ASCII-256")
    i = i + 1
text_file.close()
```

The actual encryption process followed the encryption formula laid out in the Project 1 Prompt:

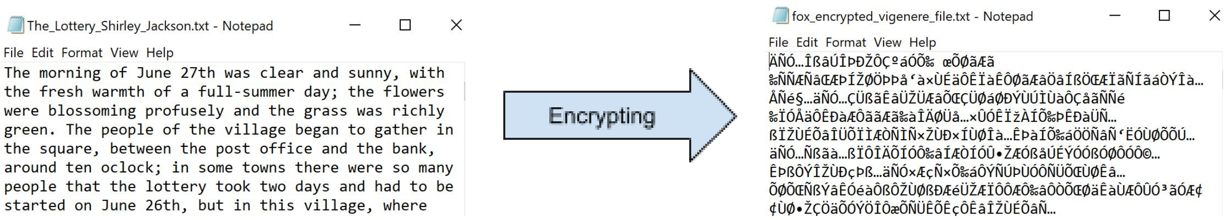
$$Y_i = (X_i + K_{i \% m}) \% 256$$

The corresponding variables for X_i *and* $K_{i \% m}$ are stored in the variables `X_i` and `K_i` below, respectively. The built-in Python `ord` function converts the input unicode character to its equivalent Unicode decimal number. The numbers are then added modulus 256 so that the encrypted characters remain in the extended ASCII code.

```
index = 0
```

```python
    # m is the key length, as specified by the Project 1 prompt formula
    m = len(key)

    # Encrypt file here
    encrypted_file = open(output_file_name, "w", encoding="latin1")
    while index < len(text):
        # Yi = (Xi + Ki % m) % 256
        X_i = ord(text[index])
        K_i = ord(key[index % m])
        Y_i = (X_i + K_i) % 256

        # Increment index each time to continue encrypting
        index = index + 1
        # Place value into output file
        encrypted_file.write(chr(Y_i))
```

The resulting files look like the following:



Before and after running the Vigenere encryption on the file.

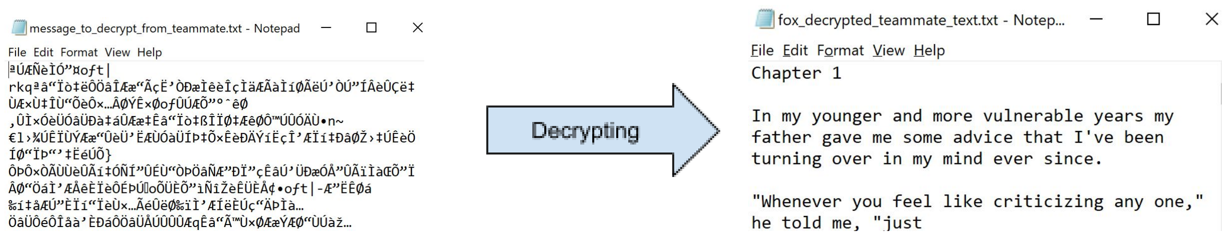
Decryption: The Vigenere's cipher decryption was coded in the `decrypt_vigenere` function. The function begins by reading through every character in the given encrypted text file given as `encrypted_text`. One character is read at a time and is continued to be read until no there is no character left, where the while loop then breaks.

```python
index = 0
m = len(key)
while True:
    # Read one character at a time
    char = encrypted_text.read(1)
    if not char:
        break

    # Xi = (Yi – Ki % m) % 256
    Y_i = ord(char)
    K_i = ord(key[index % m])
    X_i = (Y_i - K_i) % 256
    index = index + 1

    decrypted_text_file.write(chr(X_i))
```

The actual decryption process followed the decryption formula laid out in the Project 1 Prompt:

$$X_i = (Y_i - K_{i \% m}) \% 256$$

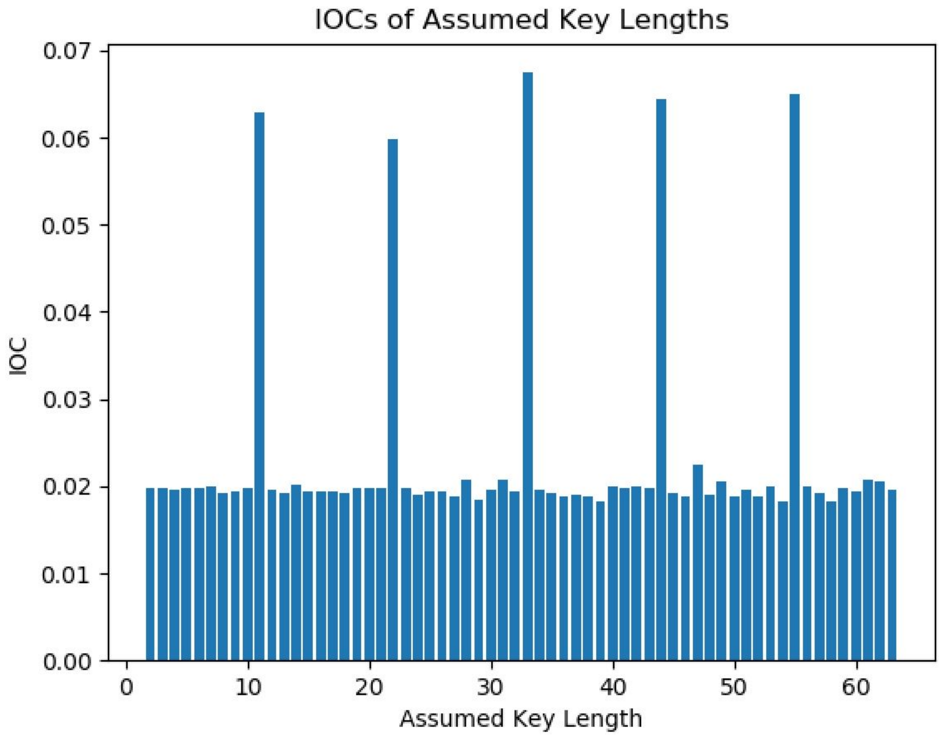
The corresponding variables are stored in `Y_i`, `X_i`, and `K_i`. The results are then written to a given `decrypted_text` file, specified in the parameters of the function. Modulus 256 is used in X_i to ensure that the resulting characters are always in the Extended ASCII 256 encoding as specified by the prompt. The results of running decrypt_vigenere using the keyword discovered by running an attack on the encrypted text can be seen below:



Before and after running the Vigenere decryption on the file.

Attacking a given Encrypted File: To allow for the decrypting of a file that has been encrypted by the Vigenere cipher, the keyword that was used to encrypt the Vigenere cipher must first be known. In order to crack the keyword, the length of the keyword must be learned. The keyword length can be discovered by analyzing the index of coincidence (IOC) based on assumed keyword lengths. The function `get_multiple_iocs` was written to evaluate the impact of the IOC on each keyword length.

For example, to evaluate an assumed keyword length of 2, every second letter of the given encrypted text is then combined into one string and then its IOC is evaluated. After this, every third letter from an encrypted text is evaluated. This continues until a certain number given in the parameter, which was given to be 64 in this function so that the pattern could easily be revealed. The periodic values of the numbers with the highest IOC can be assumed to be a multiple of the highest spikes. When cracking my classmate's given supplied text, I knew it had to be a factor of 11, because the spikes occurred every 11th key length. Because this project has a key limit length between 11 and 26, I knew I only had to attempt a key of length 11 and 22 to crack the cipher. The results of running `get_multiple_iocs` on my teammate's encrypted file can be seen below.



Assumed Key Length vs. IOC graph (note the spikes every 11th length)

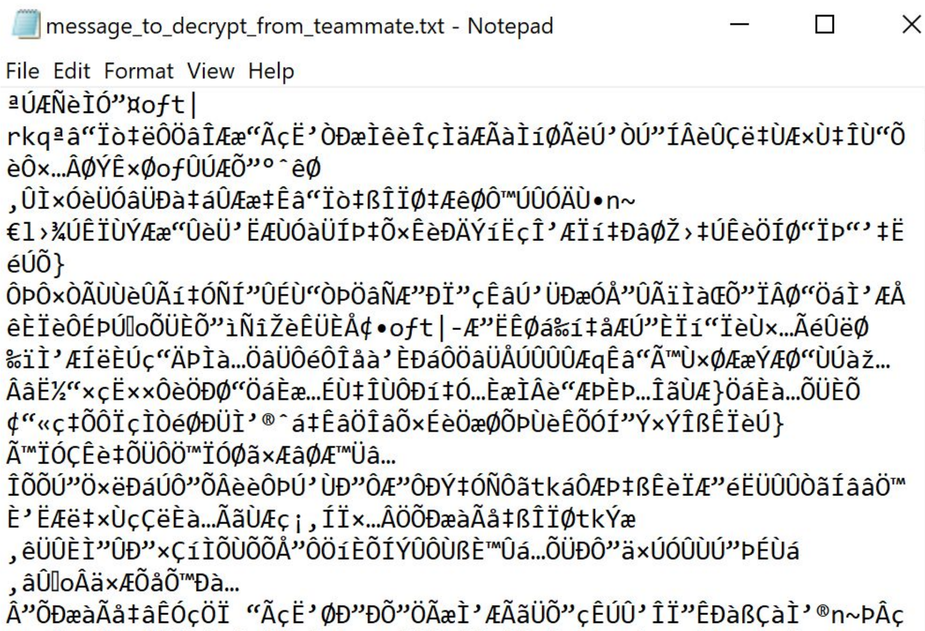
I was thus able to ascertain that the key length was 11 or 22 from the above graph, making my use of the function `attack_vigenere_cipher` go by quickly.

Once the length of the keyword is ascertained, the `attack_vigenere_cipher` function then prints out the resulting keyword. The `attack_vigenere_cipher` function works by running a for loop that iterates `analyzed_key_length_from_graph` times, where *n* letters are found for an *n* length keyword. For every needed letter of the keyword, the break_caesar_cipher function is run. The function is called break_caesar_cipher because the cipher is effectively turned into a Caesar cipher once every nth letter is taken out of the encrypted text. This function works by first extracting out every given i^{th} number.

The function then computes the chi squared values to see how close the output text is to the expected english phrases in Z_{256}. Each letter in the given text is adjusted by one until the value with the lowest chi-squared score is found. In addition, the text with the highest amount of letters that have a probability greater than zero in the english language (e.g. letters þ have a probability of 0.0 in the English language since they are not English characters) is determined to be the most “fit,” since that particular text had the most English letters normally found in the English language. The equivalent ASCII letter is then found and returned for each i^{th} number. The `attack_vigenere_cipher` function then prints out the expected keyword.

Results: Running `attack_vigenere_cipher(encrypted_file_path,analyzed_key_length_from_graph)` on my classmate's encrypted text, where `encrypted_file_path` is the full file path of the file my classmate gave me and `analyzed_key_length_from_graph` is 11 as found in the section described above yielded the keyword *greatgatsby*.

I then ran `decrypt_vigenere(filepath_of_encrypted=encrypted_file_path, key=encryption_key, output_file_name="fox_decrypted_teammate_text.txt")` with encrypted_file_path equal to the encrypted file given by my teammate and encryption key equal to "greatgatsby." The encrypted file thus changed from



Encrypted File from classmate

to the decrypted text of the first chapter of *The Great Gatsby*:

fox_decrypted_teammate_text.txt - Notepad

File Edit Format View Help

```
Chapter 1


In my younger and more vulnerable years my father gave me some advice

that I've been turning over in my mind ever since.


"Whenever you feel like criticizing any one," he told me, "just

remember that all the people in this world haven't had the advantages

that you've had."


He didn't say any more but we've always been unusually communicative

in a reserved way, and I understood that he meant a great deal more

than that. In consequence I'm inclined to reserve all judgments,

a habit that has opened up many curious natures to me and also

made me the victim of not a few veteran bores. The abnormal mind

is quick to detect and attach itself to this quality when it

appears in a normal person, and so it came about that in college I
```

Encrypted file from classmate (the first chapter of The Great Gatsby)

The attack of my classmate's decrypted file was thus successful in both uncovering his secret keyword of *greatgatsby* and of decrypting his supplied text.

Answered Questions from the Prompt:

1. *What is the typical probability distribution of the 256 ASCII characters in .txt data?*
   After inputting several large English text files and analzing the results of the probability, a typical probability distribution of the 256 ASCII characters in .txt data found at https://norvig.com/big.txt and containing several novels such as *The Adventures of Sherlock Holmes* and many other similar English novels and essays. The most popular characters found in this large (6,337 MB) text file were the space key with a probability of 0.15974, 'e' with 0.09682, and 't' with 0.06849.

Char	Probability	Index
NUL	0.0	0
SOH	0.0	1
STX	0.0	2
ETX	0.0	3
EOT	0.0	4
ENQ	0.0	5

ACK	0.0	6
BEL	0.0	7
BS	0.0	8
HT	1.8e-06	9
LF	0.0197971	10
VT	0.0	11
FF	0.0	12
CR	0.0	13
SO	0.0	14
SI	0.0	15
DLE	0.0	16
DC1	0.0	17
DC2	0.0	18
DC3	0.0	19
DC4	0.0	20
NAK	0.0	21
SYN	0.0	22
ETB	0.0	23
CAN	0.0	24
EM	0.0	25
SUB	0.0	26
ESC	0.0	27
FS	0.0	28
GS	0.0	29
RS	0.0	30
US	0.0	31
[Space]	0.1597418	32
!	0.0006696	33
"	0.0040236	34
#	0.0001144	35
$	1.7e-05	36
%	1.2e-06	37
&	1.2e-06	38
'	0.0015439	39
(	0.0002694	40
)	0.0002694	41
*	7.54e-05	42
+	1.4e-05	43
,	0.0119709	44
-	0.0026395	45
.	0.0090422	46
/	2.03e-05	47
0	0.0004722	48
1	0.0008587	49
2	0.0004705	50
3	0.0003841	51
4	0.0003725	52
5	0.0003378	53
6	0.0003072	54
7	0.0002913	55

8	0.0003894	56
9	0.0003081	57
:	0.0002859	58
;	0.0005411	59
<	3e-07	60
=	0.0002719	61
>	5e-07	62
?	0.0006403	63
@	1.2e-06	64
A	0.0018828	65
B	0.0009188	66
C	0.000943	67
D	0.0006209	68
E	0.0008606	69
F	0.0006937	70
G	0.0004907	71
H	0.001134	72
I	0.0026468	73
J	0.0002037	74
K	0.0002524	75
L	0.0004229	76
M	0.0009608	77
N	0.0010204	78
O	0.0006448	79
P	0.0013787	80
Q	2.3e-05	81
R	0.0008597	82
S	0.0013345	83
T	0.0025093	84
U	0.0002494	85
V	0.0002856	86
W	0.000964	87
X	9.46e-05	88
Y	0.0003522	89
Z	2.23e-05	90
[	6.7e-05	91
\	0.0	92
]	6.7e-05	93
^	3e-07	94
_	0.0008458	95
`	0.0	96
a	0.0610098	97
b	0.0103574	98
c	0.0213993	99
d	0.0326226	100
e	0.0968202	101
f	0.017935	102
g	0.0144455	103
h	0.0442808	104
i	0.0537035	105

```
j        |0.0007881  |106
k        |0.0048022  |107
l        |0.0301917  |108
m        |0.0186215  |109
n        |0.0558508  |110
o        |0.0589771  |111
p        |0.0138653  |112
q        |0.0006815  |113
r        |0.0468474  |114
s        |0.0502781  |115
t        |0.0684977  |116
u        |0.0211313  |117
v        |0.0077867  |118
w        |0.0145756  |119
x        |0.0014172  |120
y        |0.0135923  |121
z        |0.0005627  |122
{        |0.0        |123
|        |6.29e-05   |124
}        |0.0        |125
~        |3e-07      |126
Delete   |0.0        |127
128      |0.0        |128
129      |0.0        |129
130      |0.0        |130
131      |0.0        |131
132      |0.0        |132
133      |0.0        |133
134      |0.0        |134
135      |0.0        |135
136      |0.0        |136
137      |0.0        |137
138      |0.0        |138
139      |0.0        |139
140      |0.0        |140
141      |0.0        |141
142      |0.0        |142
143      |0.0        |143
144      |0.0        |144
145      |0.0        |145
146      |0.0        |146
147      |0.0        |147
148      |0.0        |148
149      |0.0        |149
150      |0.0        |150
151      |0.0        |151
152      |0.0        |152
153      |0.0        |153
154      |0.0        |154
155      |0.0        |155
```

156	0.0	156
157	0.0	157
158	0.0	158
159	0.0	159
160	0.0	160
¡	0.0	161
¢	0.0	162
£	0.0	163
¤	0.0	164
¥	0.0	165
¦	0.0	166
§	0.0	167
¨	0.0	168
©	0.0	169
ª	0.0	170
«	0.0	171
¬	0.0	172
173	0.0	173
®	0.0	174
¯	0.0	175
°	0.0	176
±	0.0	177
²	0.0	178
³	0.0	179
´	0.0	180
µ	0.0	181
¶	0.0	182
·	0.0	183
¸	0.0	184
¹	0.0	185
º	0.0	186
»	0.0	187
¼	0.0	188
½	0.0	189
¾	0.0	190
¿	0.0	191
À	0.0	192
Á	0.0	193
Â	0.0	194
Ã	0.0	195
Ä	0.0	196
Å	0.0	197
Æ	0.0	198
Ç	0.0	199
È	0.0	200
É	0.0	201
Ê	0.0	202
Ë	0.0	203
Ì	0.0	204
Í	0.0	205

```
Î         |0.0          |206
Ï         |0.0          |207
Ð         |0.0          |208
Ñ         |0.0          |209
Ò         |0.0          |210
Ó         |0.0          |211
Ô         |0.0          |212
Õ         |0.0          |213
Ö         |0.0          |214
×         |0.0          |215
Ø         |0.0          |216
Ù         |0.0          |217
Ú         |0.0          |218
Û         |0.0          |219
Ü         |0.0          |220
Ý         |0.0          |221
Þ         |0.0          |222
ß         |0.0          |223
à         |0.0          |224
á         |0.0          |225
â         |0.0          |226
ã         |0.0          |227
ä         |0.0          |228
å         |0.0          |229
æ         |0.0          |230
ç         |0.0          |231
è         |0.0          |232
é         |0.0          |233
ê         |0.0          |234
ë         |0.0          |235
ì         |0.0          |236
í         |0.0          |237
î         |0.0          |238
ï         |0.0          |239
ð         |0.0          |240
ñ         |0.0          |241
ò         |0.0          |242
ó         |0.0          |243
ô         |0.0          |244
õ         |0.0          |245
ö         |0.0          |246
÷         |0.0          |247
ø         |0.0          |248
ù         |0.0          |249
ú         |0.0          |250
û         |0.0          |251
ü         |0.0          |252
ý         |0.0          |253
þ         |0.0          |254
ÿ         |0.0          |255
```

2. What is the effect of Vigenere encryption on the data statistics of .txt data such as mode, mean, median, standard deviation and entropy?

Because the Vigenere cipher is a polyalphabetic cipher, the letters can change at different positions throughout the text and, as is the case of using the Extended ASCII like this project, the encrypted letters use may be completely different from the letters of the unencrypted text. Thus, data statistics of a .txt file that has been encrypted by the Vigenere cipher could have a completely different mode, median, standard deviation and entropy as the original unencrypted text. The amount of change is dependent on the alphabets used and the keyword used to encrypt the file.

3. What is the effect of cascading 2 Vigenere crypto systems on the security of the system?

Because encrypting sequentially, like in a cascading of 2 Vigenere crypto systems, with two keys K_a and K_b allows for a third key K_c to decrypt the system. This is a property of substitution ciphers, regardless of if they are monoalphabetic or polyalphabetic. This means that the result of cascading crypto systems is at least as vulnerable as the least secure of the two. So the security of the system is not necessarily improved.

Appendix:

The following contains binary_vigenere_fox.py, the file used to carry out the entire project 1:

binary_vigenere_fox.py:

```
# Aaron Fox
# Project 1 (Binary Vigenere Crypto System)
# CECS 564
# Dr. Desoky
# This file can encrypt, decrypt, and attack files encrypted with the Vigenere cipher system with or
without an input keyword
# NOTE: The encoding here uses extended ASCII-256 encoding based on the first 256 characters latin1
(iso-8859-1) encoding found at:
# this means that the following results from using the chr function, following the extended ASCII
# table found here: https://www.ascii-code.com/
# chr(252) == ü
# chr(253) == ý
# chr(254) == þ
# chr(255) == ÿ

# For directory usage (os.getcwd)
import os
# For graphing frequency distributions
import matplotlib.pyplot as plt
# For math.floor
import math

# encrypt_vigenere encrypts the text in a given filepath and outputs it to the given filepath
# NOTE: This uses the first 256 characters of latin1, which is equivalent
# INPUT: text_file_path (string): filepath of file containing extended ASCII text to be encrypted
#        key (string): key that is to be used to encrypt the text
```

```
#        output_file_name (string): just the file name of the output file to be placed in the
current directory
# OUTPUT: [unencrypted_text, encrypted_text] But saves encrypted .txt file to the current directory
def encrypt_vigenere(text_file_path, key, output_file_name):
    print("Encrypting text using Vigenere encryption...")
    # Open ASCII text file for reading based on input path
    text_file = open(text_file_path, 'r', encoding='latin1')
    # Read all info of  extended ASCII characters only
    text = []
    i = 0
    while True:
        # Read one character at a time, including only chars of first 256 characters
        char = text_file.read(1)
        if not char:
            break
        if ord(char) > 0 and ord(char) < 256:
            text.append(char)
        else:
            print("Excluding char " + str(char) + " from reading of input file because it's not in
ASCII-256")
        i = i + 1
    text_file.close()

    index = 0
    # m is the key length, as specified by the Project 1 prompt formula
    m = len(key)

    encrypted_text= []
    # Encrypt file here
    encrypted_file = open(output_file_name, "w", encoding="latin1")
    while index < len(text):
        # Yi = (Xi + Ki % m) % 256
        X_i = ord(text[index])
        K_i = ord(key[index % m])
        Y_i = (X_i + K_i) % 256

        # Increment index each time to continue encrypting
        index = index + 1
        # Place value into output file
        encrypted_file.write(chr(Y_i))

        # Debugging (Uncomment for testing purposes)
        # print('X_i == ' + str(X_i))
        # print('K_i == ' + str(K_i))
        # print('Y_i == ' + str(Y_i))
        # print("chr(" + str(Y_i) + ") == " + chr(Y_i))
        encrypted_text.append(chr(Y_i))

    encrypted_file.close()
    print("Successfully encrypted text. It is stored in " + str(os.getcwd()) + "\\" +
str(output_file_name))
    return [''.join(text), ''.join(encrypted_text)]


# Decrypt the text found in the filepath of filepath_of_encrypted based on the Vigenere Cipher
# INPUT: filepath_of_encrypted (string): whole filepath file to be encrypted
#        key (string): key that was used to encrypt the text
#        output_file_name (string): just the file name of the output file to be placed in the
current directory
# OUTPUT: None. But saves decrypted .txt file to the current directory
def decrypt_vigenere(filepath_of_encrypted, key, output_file_name):
```

```
    print("Decrypting text using Vigenere process...")
    encrypted_text = open(filepath_of_encrypted, 'r', encoding='latin1')
    decrypted_text_file = open(output_file_name, 'w', encoding='latin1')
    index = 0
    m = len(key)
    while True:
        # Read one character at a time
        char = encrypted_text.read(1)
        if not char:
            break

        # Xi = (Yi - Ki % m) % 256
        Y_i = ord(char)
        K_i = ord(key[index % m])
        X_i = (Y_i - K_i) % 256
        index = index + 1

        decrypted_text_file.write(chr(X_i))

    decrypted_text_file.close()
    print("Successfully decrypted text. It is stored in " + str(os.getcwd()) + "\\" +
str(output_file_name))

# graph_probability_of_each_character plots a probability bar chart of each character in a text
# INPUT: dictionary (dict): key: letter, value: probability
# OUTPUT: None, just displays a graph
def graph_probability_of_each_character(probabilities_dict):
    # Display bar chart of probabilities of each letter, ignoring characters that aren't in text
    plotting_values = {}
    for key, val in probabilities_dict.items():
        if val != 0:
            plotting_values[key] = val

    plt.bar(plotting_values.keys(), plotting_values.values())
    plt.xlabel('Character')
    plt.ylabel('Probability')
    plt.title('Probability of each character in a typical Text file')
    # plt.show()


# get_index_of_coincidence returns the index of coincidence (IOC) of a text
def get_index_of_coincidence(text):
    # Calculate frequency of each char
    char_frequency = {}
    for i in range(256):
        char_frequency[chr(i)] = 0

    for char in text:
        char_frequency[char] = char_frequency[char] + 1

    sum = 0
    N = len(text)
    for key in char_frequency.keys():
        sum = sum + (char_frequency[key] * (char_frequency[key] - 1))

    return sum / (N * (N - 1))


# get_multiple_iocs gets multiple iocs from the text based on various sizes of keywords
# e.g. get every second letter, every third letter, every fourth letter
# and determine how that influences the IOC
```

```
# INPUT: num_to_stop (int): number to stop checking of every nth letter to slice out
#        text (string): text to analyze the IOCs from
# OUTPUT: iocs (list): ordered list of IOCs from 2 to num_to_stop
def get_multiple_iocs(num_to_stop, text):
    iocs = []
    for i in range(2, num_to_stop):
        new_text = []
        for i in range(0, len(text), i):
            new_text.append(text[i])

        ioc = get_index_of_coincidence(''.join(new_text))
        iocs.append(ioc)
    return iocs

# graph_iocs simply graphs out the iocs
def graph_iocs(iocs):
    x = list(range(2, 2 + len(iocs)))
    plt.bar(x, iocs)
    plt.xlabel('Assumed Key Length')
    plt.ylabel('IOC')
    plt.title('IOCs of Assumed Key Lengths')
    plt.show()


# get_probability_dist_of_text gets the probability density function of a typical text file
# INPUT: value_to_increment_letter_by (int): Given value of letter to increment by for other
encodings
#        filepath_of_text_to_get_pdf_from (string): file path of text to get probability of
# OUTPUT: [probabilities_dict (dict), letter_count_dict (dict)]: An array containing a
# dictionary with each extended ASCII character and the probability of that character occuring and
# a dictionary with the count of each letter
def get_probability_dist_of_text(value_to_increment_letter_by, filepath_of_text_to_get_pdf_from):
    print("Getting probability distribution of text...")
    text_file = open(filepath_of_text_to_get_pdf_from, "r", encoding="latin1")

    # Build out each character in Z_256 of probabilities dictionary with a count of 0 initially
    probabilities_dict = {}
    for i in range(0, 256):
        probabilities_dict[chr((i + value_to_increment_letter_by) % 256)] = 0

    # Record total number of characters for calculating the probability distribution
    total_number_of_characters = 0
    while True:
        # Read one character at a time, incrementing by certain value
        char = text_file.read(1)
        if not char:
            break
        else:
            char = chr((ord(char) + value_to_increment_letter_by) % 256)

        if char in probabilities_dict:
            probabilities_dict[char] = probabilities_dict[char] + 1
            total_number_of_characters = total_number_of_characters + 1
        else:
            print("Skipping character " + str(char) + ", because it is not in the extended ASCII
table + value_to_increment_letter_by")

    # Before getting probability, save it in the letter_count_dict
    # for finding the index of coincidence of each letter
    letter_count_dict = probabilities_dict.copy()
```

```python
    # Get percentage of each characters usage
    for i in range(0, 256):
        probabilities_dict[chr((i + value_to_increment_letter_by) % 256)] =
probabilities_dict[chr((i + value_to_increment_letter_by) % 256)] / total_number_of_characters

    # graph_probability_of_each_character(probabilities_dict)

    return [probabilities_dict, letter_count_dict]



# break_caesar_cipher uses the Chi-squared method to help determine the lowest Chi-squared
# value which should correspond to the value which should crack the given cipher
# INPUT: encrypted_text (string): string of encrypted text to break
#        key_length (int): assumed length of keyword
#        starting_letter_index (int): the nth letter to start at for Caesar cipher
# OUTPUT: equivalent_ascii (char): the most likely letter or key that is used to break the cipher
# NOTE: For project 1, it is assumed that the key is only of lowercase alphabetic letters
def break_caesar_cipher(encrypted_text, key_length, starting_letter_index):
    text_to_be_evaluated = []
    # Append every nth letter beginning from 0, where n is every key_length letter
    for i in range(math.floor(len(encrypted_text) / key_length)):
        text_to_be_evaluated.append(encrypted_text[i * key_length + starting_letter_index])

    text_to_be_evaluated = ''.join(text_to_be_evaluated)

    # print("text_to_be_evaluated[0:5] == " + str(text_to_be_evaluated[0:5]))

    # Get chi-squared values of all sequences
    # Sum from Z=0 to Z=255((C_i - E_i)^2 / E_i)
    chi_squared_values = []

    lowest_chi_squared = float("inf")
    lowest_values = []
    # debug_file = open(r"C:\Users\aaron\Classes_11th_Semester\CECS
564\CECS-564-Cryptography\Project 1\debug_file.txt", "w", encoding="latin1")
    [probabilities_dict, _] = get_probability_dist_of_text(0,
r"C:\Users\aaron\Classes_11th_Semester\CECS 564\CECS-564-Cryptography\Project
1\The_Lottery_Shirley_Jackson.txt")

    value_to_increment_letter_by_with_highest_valid_letter_count = 0
    highest_num_valid_letter_count = 0
    total_num_chars_in_text = 0
    for v in probabilities_dict.values():
            total_num_chars_in_text = total_num_chars_in_text + v
    # Then iterate over every possible sequence
    # e.g. for text_to_be_evaluated = ACB, evaluate BDC, CED, etc.
    for value_to_increment_letter_by in range(0, 256):
        # Make sure text_to_be_evaluated is correctly manipulated with Caesar cipher first
        new_text_to_be_evaluated = []
        for i in range(len(text_to_be_evaluated)):
            new_text_to_be_evaluated.append(chr((ord(text_to_be_evaluated[i]) +
value_to_increment_letter_by) % 256))

        new_text_to_be_evaluated = ''.join(new_text_to_be_evaluated)
        # debug_file.write(str(new_text_to_be_evaluated))



        # print("new_text_to_be_evaluated[0:5] == " + str(new_text_to_be_evaluated[0:5]))
        # First, get count of each letter in a dict
```

```
        freq_dict_for_sequence = {}
        for i in range(0, 256):
            freq_dict_for_sequence[chr(i)] = 0

        for char in new_text_to_be_evaluated:
            freq_dict_for_sequence[char] = freq_dict_for_sequence[char] + 1

        # debug_file.write("\n\n" + str(freq_dict_for_sequence) + "\n\n")


        # Then, use that frequency dictionary along with the expected count of the letters
        # to find the chi-squared values
        chi_squared_sum = 0
        num_valid_letters = 0
        for char in freq_dict_for_sequence.keys():
            # First, make sure character is in new text
            if freq_dict_for_sequence[char] != 0:
                # If that char is an english letter, then carry on, else penalize
                if probabilities_dict[char] != 0:
                    chi_squared_sum = chi_squared_sum + ((freq_dict_for_sequence[char] -
total_num_chars_in_text * probabilities_dict[char]) ** 2) / (total_num_chars_in_text *
probabilities_dict[char])
                    num_valid_letters = num_valid_letters + 1
                else:
                    penalty_value_for_not_english_letter = 10000
                    chi_squared_sum = chi_squared_sum + penalty_value_for_not_english_letter
        # print(str(value_to_increment_letter_by) +".) chi_squared_sum == " + str(chi_squared_sum))
        # debug_file.write("\n\n" +str(value_to_increment_letter_by) + ".) chi_squared_sum == " +
str(chi_squared_sum) + "\n")
        # debug_file.write("\n" + "num_valid_letters == " + str(num_valid_letters) + "\n")
        if lowest_chi_squared > chi_squared_sum:
            lowest_chi_squared = chi_squared_sum
            lowest_values = str(value_to_increment_letter_by) +".) chi_squared_sum == " +
str(chi_squared_sum)
        if highest_num_valid_letter_count < num_valid_letters:
            value_to_increment_letter_by_with_highest_valid_letter_count =
value_to_increment_letter_by
            highest_num_valid_letter_count = num_valid_letters

    # print("lowest_values == " + str(lowest_values))
    # print("lowest_chi_squared == " + str(lowest_chi_squared))
    # print("highest num_valid_letters == " + str(highest_num_valid_letter_count))
    equivalent_ascii = chr(97 + (159 -
value_to_increment_letter_by_with_highest_valid_letter_count))
    # print("highest value_to_increment_letter_by_with_highest_valid_letter_count == " +
str(value_to_increment_letter_by_with_highest_valid_letter_count) + ", or " +
str(chr(value_to_increment_letter_by_with_highest_valid_letter_count - 32)))
    # print("equivalent_ascii == " + equivalent_ascii)
    # debug_file.close()
    return equivalent_ascii

# attack_vigenere_cipher attacks the text of a file encrypted with the Vigenere cipher
# INPUT: encrypted_text_filepath (string): string of full filepath of a file encrypted using
vigenere
#        analyzed_key_length_from_graph (int): keyword length that is found by spikes of graph using
get_multiple_iocs
# OUTPUT: None. Only a string is printed to the console, which is the keyword
def attack_vigenere_cipher(encrypted_text_filepath, analyzed_key_length_from_graph):
    print("Attacking file encrypted with Vigenere cipher...")

    encrypted_text = open(encrypted_text_filepath, "r", encoding="latin1")
```

```
    text = []
    while True:
        # Read one character at a time
        c = encrypted_text.read(1)
        if not c:
            break
        text.append(c)

    encrypted_text = ''.join(text)

    keyword = []
    for i in range(0, analyzed_key_length_from_graph):
        keyword.append(break_caesar_cipher(encrypted_text, analyzed_key_length_from_graph, i))

    print("keyword: " + ''.join(keyword))

# get_text returns a string of the text contained within a text file
# INPUT: file_path (string): filepath of text to turn into a string
# OUTPUT: output_text (string): string of all text contained inside filepath
def get_text(file_path):
    text = open(file_path, "r", encoding="latin1")
    output_text = []
    while True:
        # Read one character at a time
        c = text.read(1)
        if not c:
            break
        output_text.append(c)

    output_text = ''.join(output_text)
    return output_text


if __name__ == "__main__":
    #### Encrypting file ####
    # file_path = r"C:\Users\aaron\Classes_11th_Semester\CECS 564\CECS-564-Cryptography\Project 
1\MansNotHot.txt"
    file_path = r"C:\Users\aaron\Classes_11th_Semester\CECS 564\CECS-564-Cryptography\Project 
1\The_Lottery_Shirley_Jackson.txt"
    encryption_key='pineapple'
    [unencrypted_text, encrypted_text] = encrypt_vigenere(text_file_path=file_path, 
key=encryption_key, output_file_name="fox_encrypted_vigenere_file_the_lottery.txt")

    #### Decrypting file ####
    # encrypted_file_path = r"C:\Users\aaron\Classes_11th_Semester\CECS 
564\CECS-564-Cryptography\Project 1\fox_encrypted_vigenere_file.txt"
    # encrypted_file_path = r"C:\Users\aaron\Classes_11th_Semester\CECS 
564\CECS-564-Cryptography\Project 1\encrypted.txt"
    encrypted_file_path = r"C:\Users\aaron\Classes_11th_Semester\CECS 
564\CECS-564-Cryptography\Project 1\message_to_decrypt_from_teammate.txt"
    encryption_key="greatgatsby"
    decrypt_vigenere(filepath_of_encrypted=encrypted_file_path, key=encryption_key, 
output_file_name="fox_decrypted_teammate_text.txt")

    # Obtaining probability distribution of a typical text
    typical_text = r"C:\Users\aaron\Classes_11th_Semester\CECS 564\CECS-564-Cryptography\Project 
1\The_Lottery_Shirley_Jackson.txt"
    typical_text = r"C:\Users\aaron\Classes_11th_Semester\CECS 564\CECS-564-Cryptography\Project 
1\big.txt"
    [probabilities_dict, letter_count_dict] = get_probability_dist_of_text(0, typical_text)
    print("probabilities_dict == " + str(probabilities_dict))
```

```python
    print("{:<8} {:<6}".format('Char', '|Probability'))
    i = 0
    for k, v in probabilities_dict.items():
        # label, num = v
        if i < 33 or i > 126 and i < 161:
            k = i
        print("{:<8} |{:<10} |{:<10}".format(k, round(v, 7), i))
        i = i + 1

    # Get characters with the highest probability
    from collections import Counter
    counter = Counter(probabilities_dict)

    # Find 8 most common characters
    highest_values = counter.most_common(8)

    print("Dictionary with 8 highest values:")
    print("Keys: Values")

    for val in highest_values:
        print(val[0], " :", val[1], " ")

    # table_file = open(r"C:\Users\aaron\Classes_11th_Semester\CECS 
564\CECS-564-Cryptography\Project 1\typical_256_distribution.txt", "w", encoding="latin1")
    # Write probabilities_dict to text so that it can be easily converted to a table in Excel/Google 
Sheets
    # filename = r"C:\Users\aaron\Classes_11th_Semester\CECS 564\CECS-564-Cryptography\Project 
1\typical_256_distributionxl.txt"

    # table_file.close()
    # Pretty printing probabilities dict:
    # Attacking encrypted file
    # ioc = get_index_of_coincidence(unencrypted_text)
    # print("ioc == " + str(ioc))

    encrypted_text = get_text(encrypted_file_path)
    multiple_iocs = get_multiple_iocs(64, encrypted_text)
    # graph_iocs(multiple_iocs)

    # Analyzing the IOC graph above for the assumed key lengths, we can determine the length of the 
key
    # based on the (probably) least common multiple of all the occurring spikes
    # (e.g. if the key is 9 letters long, there is an IOC spike every 9th assumed key length in the 
graph)
    # CHANGE THIS VALUE WHEN KEY LENGTH IS ASCERTAINED FROM GRAPH
    analyzed_key_length_from_graph = 11

    # Once we know the length of the key, the problem is effectively the same as solving the Caesar 
Cipher
    # Problem. So we can use the Chi-squared calculations to solve for the letters of the key
    # (The lowest chi-squared is most likely to be the key, although it is not guaranteed)
    # There are thus analyzed_key_length_from_graph Caesar ciphers to break
    encrypted_text_filepath = r"C:\Users\aaron\Classes_11th_Semester\CECS 
564\CECS-564-Cryptography\Project 1\encrypted.txt"
    attack_vigenere_cipher(encrypted_file_path, analyzed_key_length_from_graph)
```