CSCE 477 – FALL 2024

Homework 1 – Date: 09/22/2024

Abstract

Using the frequency analysis, Kasiski Test, Index of Coincidence to decrypt ciphertext
These decipher methods are used in this assignment:
Shift Cipher, Affine Cipher, Vigenère Cipher

Cong Nguyen
cnguyen46@huskers.unl.edu

CONTENTS

1. INTRODUCTION

1.1. SUMMARY THE ASSIGNMENT

In this assignment, students are required to apply the cryptographic techniques learned so far to decrypt four given ciphertexts. The following methods can be used for decryption:

- Shift Cipher
- Substitution Cipher
- Vigenère Cipher
- Permutation Cipher (Columnar Transposition)
- One-Time Pad

Important Guidelines:

- Using brute force alone for decryption is strictly prohibited.
- Students must employ cryptanalysis techniques to analyze each ciphertext and determine the most suitable decryption method.
- If students are unable to decrypt a ciphertext, they must provide an explanation using cryptanalysis to justify why decryption was not possible.

About my programming:

- I use Python and VS Code to work on this assignment. Next part is the tutorial to set up the environment to run my program.
- I designed this assignment as a fun game involving four ciphertexts provided by Professor Ghose, where you must enter the correct key to access the system.

1.2 SET UP WORKING ENVIRONMENT

I use this website to download and set up the environment for working:

https://code.visualstudio.com/docs/python/python-tutorial

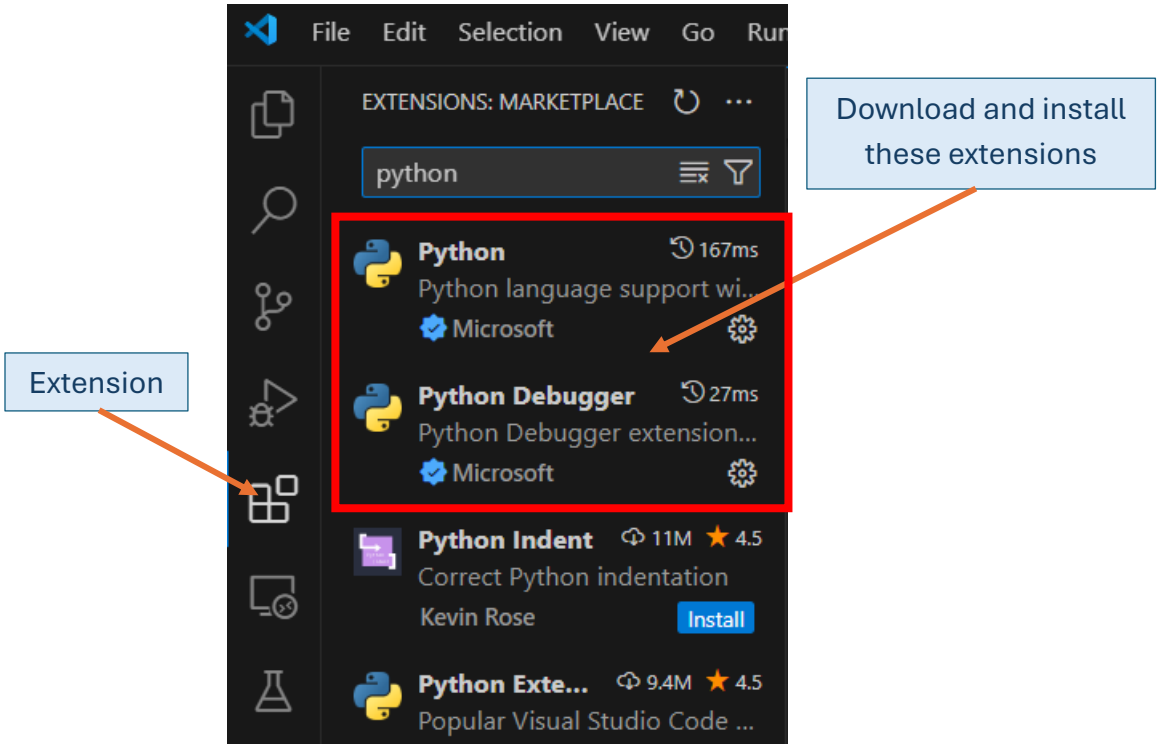
There are three prerequisites on the website:

- Python 3
- VS Code
- VS Code Python Extension

Download and Install VS Code: Click the provided link to download VS Code. Follow the installation instructions to set it up on your computer.

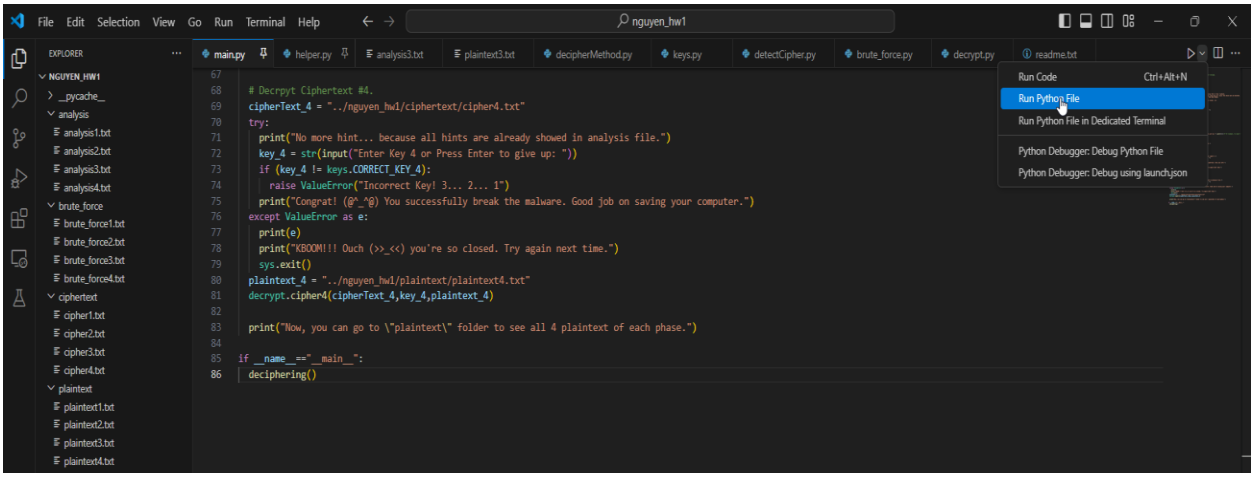
Download and Install **Python 3**: Click the provided link to download Python 3. Install it on your system by following the setup prompts.

Set Up Python in VS Code: Open VS Code, navigate to the Extensions tab, and search for the "Python" extension. Click "Install" to add it to your editor.



Go to File → "Open folder" → Choose the directory leading to *<nguyen_hw1>* → Click "Select Folder".

Run Your Python Program: You must execute the Python code on the *main.py*. Click the **Run Python File** button at the top of the window.



*** Note: If you encounter any issues running the program, feel free to contact me via email for assistance.

2. TESTING HYPOTHESIS

Hypothesis: All letters in the four cipher files are capital letters from the 26-letter alphabet (A–Z).

Testing: To test this hypothesis, we will:

- Check if the sum of the character counts in each file equals the total number of letters in the file.
- If true: The hypothesis is valid.
- If false: The hypothesis is rejected.

Discovery: After running the test on all four files, the results show that all characters are indeed capital letters. This confirms that we only need to account for the 26 alphabet letters (A–Z), thus proving the hypothesis to be true.

Approach:

We can utilize the ord() function (which returns ASCII values from 65 to 90 for A to Z) to:

- Perform frequency analysis,
- Conduct the Kasiski Test, and
- Calculate the Index of Coincidence for the cipher text in the files.

*** **Note:** The function *hypothesis* in *helper.py* is responsible for performing the hypothesis test.

```
85  '''
86  # Function to test hypothesis.
87  # Return: Print out the conclusion.
88  '''
89  def hypothesis(ciphertext, dictOf_Char):
90    result = "\nTotal number of letters in ciphertext: " + str(len(ciphertext))
91    result += "\nTotal number of 26 alphabet letters  : " + str(total_Char(dictOf_Char))
92    if (total_Char(dictOf_Char) == len(ciphertext)):
93      result += "\nHypothesis is true that ciphertext only containts 26 uppercased alphabet letters."
94      result += "\n^_^ Total sum of each letter is equal with total number of character in the file.\n"
95
96    else:
97      result += "\nHypothesis is false."
98      result += "\nT_T Total sum of each letter is not equal with total number of character in the file.\n"
99    return result
```

3. ATTEMPT BRUTE FORCE

I understand that brute force decryption is not allowed for this assignment, but out of curiosity, I decided to test it anyway. I applied a Shift Cipher, shifting the first 50 letters of each ciphertext through all 25 possible shifts, and checked if any of the resulting messages made sense. I assumed the plaintext would be in English.

Step-by-Step Process:

- Shift each character by a value **n**, where 0 < n < 26, with **n** being an integer.
- Loop through all 25 possible shifts, as there are 26 letters in the alphabet.
- Focus on the first 50 characters of the ciphertext and identify any shifts that result in meaningful content.

Findings:

- **<cipher1.txt>**: A meaningful message was found after shifting 10 times.
- **<cipher2.txt>**: A meaningful message was found after shifting 24 times.
- **<cipher3.txt>**: No meaningful content was found.
- **<cipher4.txt>**: No meaningful content was found.

Conclusion:

A new approach is required for **cipher3.txt** and **cipher4.txt**. It is recommended to apply cryptanalysis techniques such as **frequency analysis**, the **Kasiski Test**, and the **Index of Coincidence** for further investigation.

*** **Note:** The function *brute_force* in *brute_force.py* demonstrates the process in detail. All outputs are saved in the **"brute force"** folder for review.

```
'''
# Function: Shift decipher in brute force.
# Return: A string that contain all deciphered message.
'''
def brute_force(ciphertext) :
  key = 1
  result = ""
  first_50_letters = ciphertext[:50] # Only need first 50 letters.
  while (key < 26):
    current_character = 0
    plaintext = ""
    while (current_character < len(first_50_letters)):
      decipher = ord(first_50_letters[current_character]) - ord('A')
      decipher += key
      decipher = chr(decipher % 26 + ord('A'))
      plaintext += decipher
      current_character += 1
    # Print out all results.
    result += "Secret key #" + str(key) + "\n"
    result += plaintext
    result += "\n\n"
    key += 1
  return result
```

4. CRYPTANALYSIS

I wrote a program to analyze the ciphertext, and all results were saved in the **"analysis"** folder.

All details of cryptanalysis are performed in *cryptanalysis.py*.

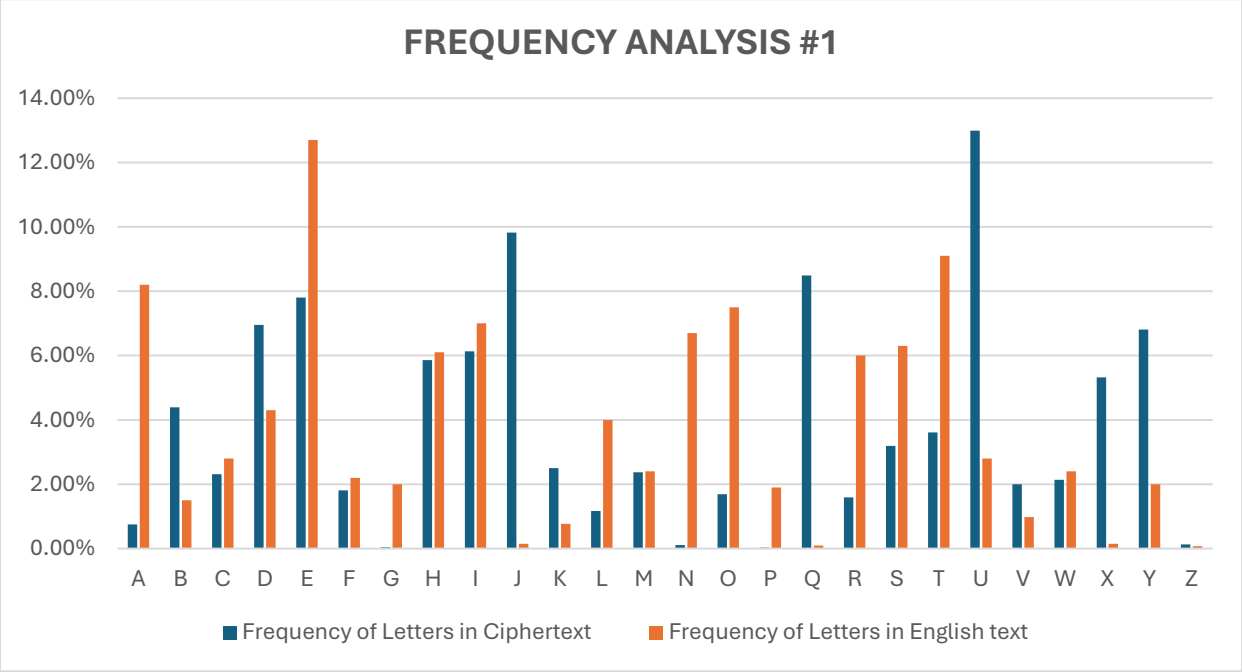
4.1. Frequency Analysis

The function *frequency_analysis* in *cryptanalysis.py* is responsible for supporting frequency analysis.

```
'''
# Function to perform frequency analysis.
# Output the frquency of repeating time of each letter.
# Return: Print out the frequency analysis.
'''
def frequency_analysis(dictOf_Char):
  # Sorting the value inside the dictionary following increasing order.
  dictOf_Char = {key : value for key, value in sorted(dictOf_Char.items(), key = lambda element: element[1], reverse = True)}
  # Print out the report.
  total = helper.total_amount(dictOf_Char)
  result = "\n-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-"
  result += "\nFREQUENCY ANALYSIS"
  result += "\n-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-"
  result += "\n%-8s %-10s %s" % ("Letter", "Amount", "Frequency (%)")
  result += "\n---------------------------------------"
  for key, value in dictOf_Char.items():
    result += "\n%-8s %-10s %s" % (key, f"{value}", f"{round(value/total * 100, 2)}")
  result += "\n-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-"
  return result
```

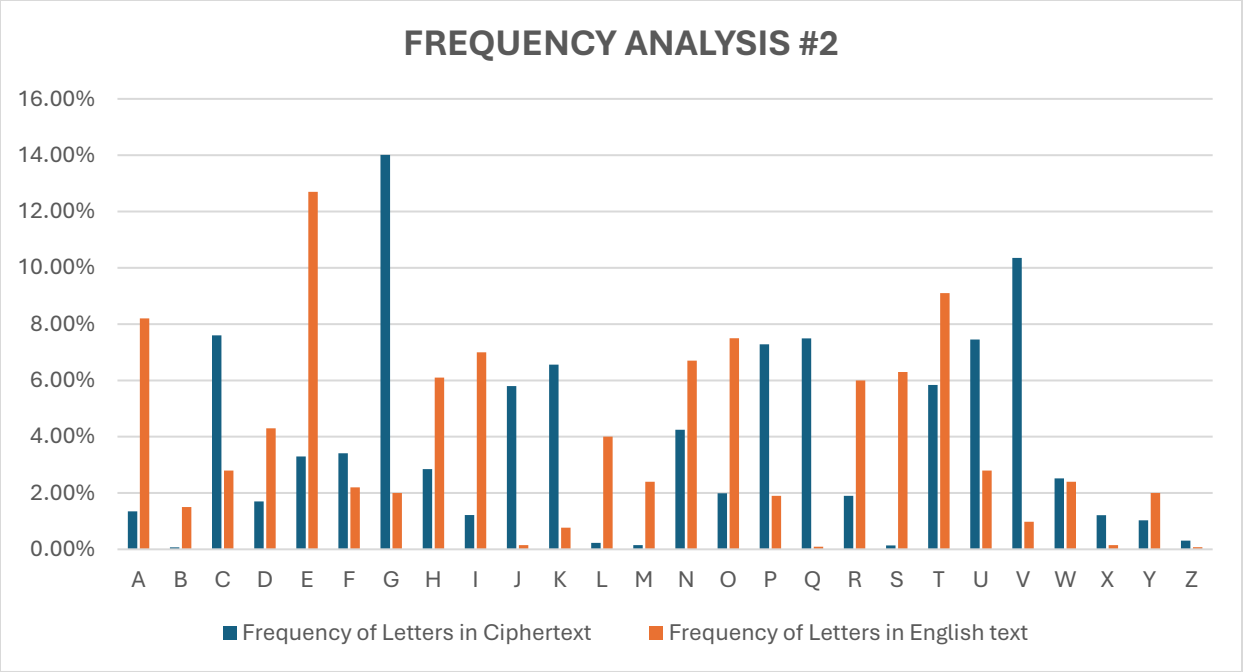
Results after running the program:

<cipher1.txt>



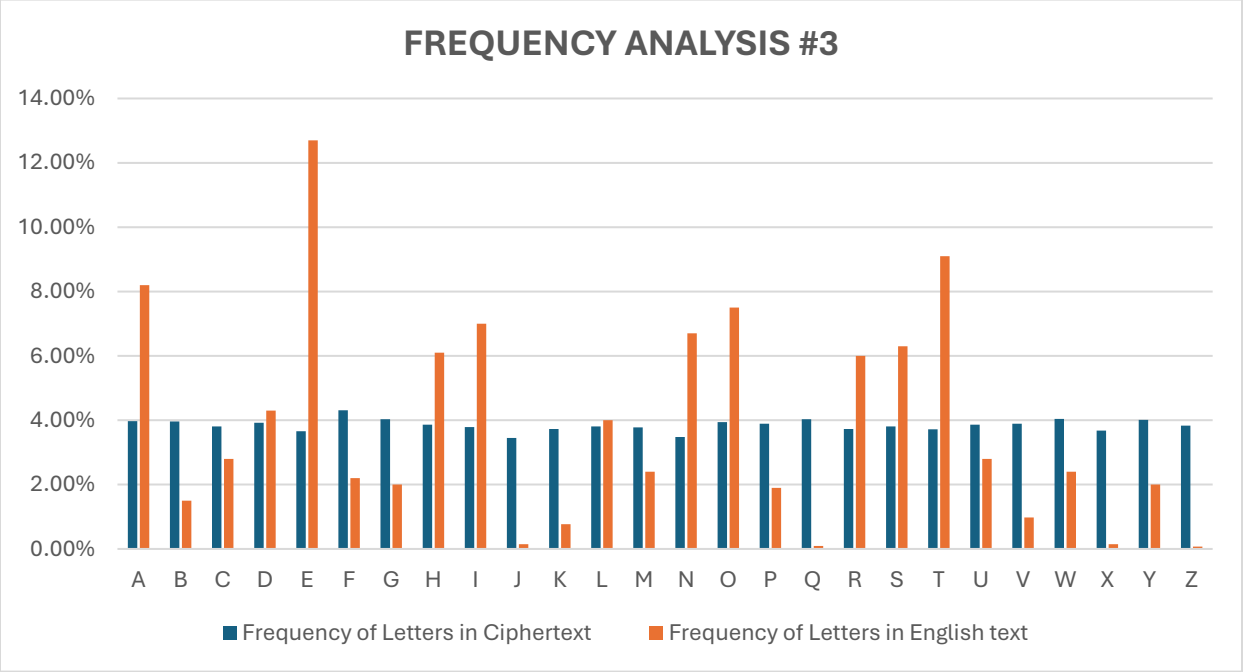
- Total number of letters in ciphertext: 9478
- Letter U repeats the most: 1231 times, with 12.99%
- Letter P repeats the least: 3 times, with 0.03%
- Initial letter is B repeats 416 times, with 4.39%
- Final letter is D repeats 659 times, with 6.95%

<cipher2.txt>



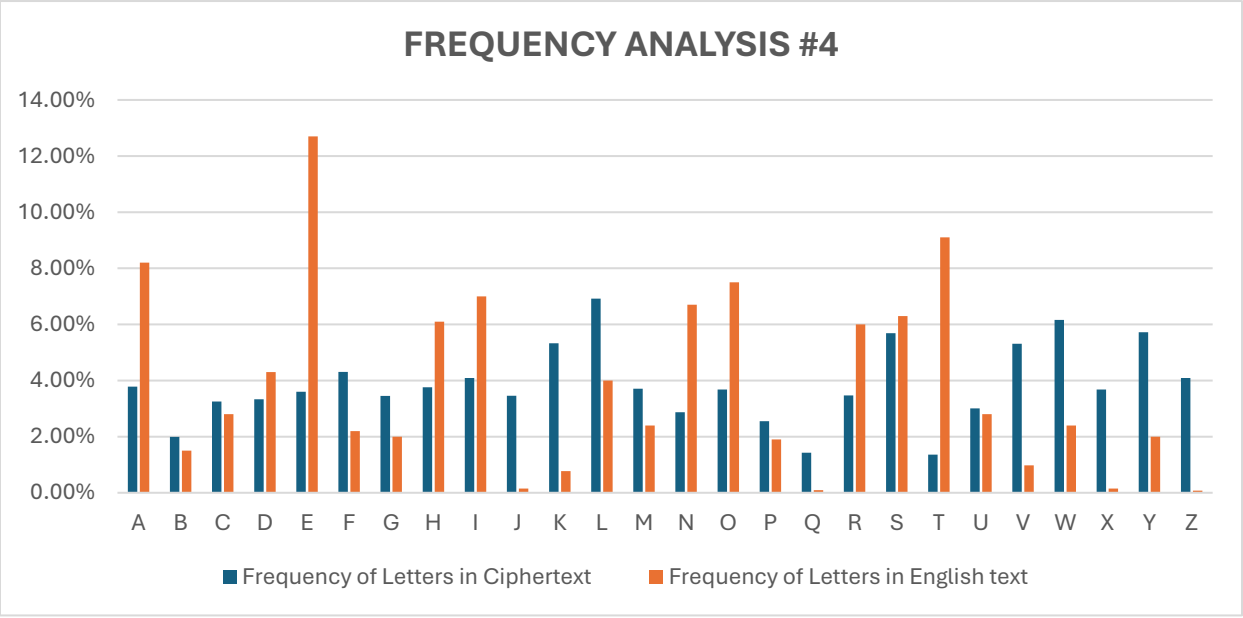
- Total number of letters in ciphertext: 20796
- Letter G repeats the most: 2913 times, with 14.01%
- Letter B repeats the least: 15 times, with 0.07%
- Initial letter is Y repeats 214 times, with 1.03%
- Final letter is J repeats 1206 times, with 5.8%

<cipher3.txt>



- Total number of letters in ciphertext: 8037
- Letter F repeats the most: 346 times, with 4.31%
- Letter J repeats the least: 277 times, with 3.45%
- Initial letter is D repeats 315 times, with 3.92%
- Final letter is K repeats 300 times, with 3.73%

<cipher4.txt>



- Total number of letters in ciphertext: 8037
- Letter L repeats the most: 556 times, with 6.92%
- Letter T repeats the least: 109 times, with 1.36%
- Initial letter is J repeats 278 times, with 3.46%
- Final letter is M repeats 298 times, with 3.71%

4.2. Kasiski Test

The function *kasiski_test* in *cryptanalysis.py* is responsible for supporting Kasiski test.

```
'''
# Kisiski Test: To print out the top common triagrams, and diagrams.
# It should be at least 5 most common triagrams, and diagrams.
# The number of triagrams and diagrams is recommended having the same number for easier analysis.
'''
def kasiski_test(trigrams, digrams):
  # Preparing data for format printing.
  output_list = trigrams + digrams
  rows, cols = len(trigrams), 2
  table_2D = [[0]*cols for _ in range(rows)]
  element = 0
  for i in range(cols):
    for j in range(rows):
      if (element < len(output_list)):
        table_2D[j][i] = output_list[element]
        element += 1
  # Output the top most common triagram, and diagram.
  result = "\n\n-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-"
  result += "\nKasiski Test:"
  result += "\nTop 10 common of trigrams and digrams"
  result += "\n-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-"
  for row in table_2D:
    if len(row) == 2 and isinstance(row[0], tuple) and isinstance(row[1], tuple):
      first_string, first_integer = row[0]
      second_string, second_integer = row[1]
      result += "\n%s: %-5s %s: %s" % (first_string, first_integer,
        second_string, second_integer)
    else:
      result += "\nUnexpected row format:", row
  result += "\n-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-\n"
  return result
```

Results after running the program:

<cipher1.txt>

Kasiski Test: Top 10 common of trigrams and diagrams			
Trigrams	**Amount**	**Digrams**	**Amount**
JXU	152	JX	297
QDT	92	XU	201
JXQ	82	QD	175
XQJ	82	YD	147
YDW	66	UH	147
UJX	48	UJ	144
UHU	41	QJ	140
VEH	40	XQ	139
XUH	39	HU	137
DJX	37	JE	135

<cipher2.txt>

Kasiski Test: Top 10 common of trigrams and diagrams			
Trigrams	**Amount**	**Digrams**	**Amount**
VJG	523	VJ	676
CNN	236	JG	573
CPF	202	GU	524
UJC	194	TG	357
JCN	191	QP	344
CVG	168	QH	340
QHV	159	VG	340

VCV	159	GT	331
GPV	155	CV	328
HVJ	149	GP	320

<cipher3.txt>

Kasiski Test: Top 10 common of trigrams and diagrams			
Trigrams	**Amount**	**Digrams**	**Amount**
VZO	5	VZ	26
GYL	4	GJ	23
IGN	4	MM	22
IWF	4	MF	22
RUA	4	OP	21
ZTA	4	YO	21
OPT	4	XA	21
UFE	4	BR	21
PBR	4	AF	21
PLO	4	GK	21

<cipher4.txt>

Kasiski Test: Top 10 common of trigrams and diagrams			
Trigrams	**Amount**	**Digrams**	**Amount**
DLV	28	LR	56
LRI	26	DL	54
SXH	24	KZ	54
XYW	24	YY	52

LOY	23	LL	50
ABW	22	WK	50
NZO	20	LW	49
KZL	20	LO	49
UFN	15	ZL	48
ZLL	13	YW	43

4.3. Index of Coincidence

The function *index_of_coincidence* in *cryptanalysis.py* is responsible for calculating index of coincidence.

```
64  '''
65  # Function to calculation index of coincidence.
66  # Return the value of index of coincidence.
67  '''
68  def index_of_coincidence(ciphertext):
69      total_Char = len(ciphertext)
70      freqs = Counter(ciphertext)
71      numerator = sum(freq * (freq - 1) for freq in freqs.values())
72      denominator = total_Char * (total_Char - 1)
73      if total_Char > 1:
74          result = numerator / denominator
75      else:
76          result = 0
77      return round(result, 4)
```

Results after running the program:

<cipher1.txt> Index of Coincidence: 0.0671

<cipher2.txt> Index of Coincidence: 0.0708

<cipher3.txt> Index of Coincidence: 0.0384

<cipher4.txt> Index of Coincidence: 0.0430

*** Note: In this assignment, the index of coincidence of English text is 0.065, and the index of coincidence of random text is 0.038.

5. DETECT CIPHER METHOD AND DECRYPT CIPHERTEXT

I wrote a program to detect and decrypt the ciphertext, performed by *detectCipher.py*, and *decipherMethod.py*, respectively.

All plaintexts after deciphering were outputted in **"plaintext"** folder.

*** **Note:** Using the analysis' results from section 4 to decide the decipher method.

An Index of Coincidence (IC) of 0.065 indicates a monoalphabetic cipher, while an IC of 0.038 suggests a polyalphabetic cipher.

For **monoalphabetic ciphers**, here is step by step that we can do to determine the type of decipher method:

- Analyze the frequency of letters in the ciphertext and compare them to the expected frequencies in the English. This can help identify potential substitutions.
- Look for repeating patterns or digrams (two-letter combinations) in the ciphertext that might correspond to common words or letter pairings in the plaintext.
- If the letter frequency distribution in the ciphertext closely matches the standard frequencies (e.g., the most frequent letters in the ciphertext align with common letters like E, T, A), it suggests a **Shift cipher**.
- If the frequency distribution shows significant deviations from standard frequencies (e.g., if there were many unique letters and certain letters appeared much more often), it suggests a **Substitution cipher**.
- Calculating the greatest common divisor (gcd) of a, and 26 (default amount for English alphabet) from this system formular:

$$\begin{cases} ax_1 + b \equiv y_1 \ mod\ 26 \\ ax_2 + b \equiv y_2 \ mod\ 26 \end{cases}$$

a, b: root of the system formular needed to be found x_1, x_2: number of the letters having the most frequency in English. y_1, y_2: number of the letters having the most frequency in ciphertext.

- Using GCD to Define the Cipher (Mono alphabet):
  - If a = 1, gcd (a, 26) = 1: This cipher is likely **Shift or Affine Cipher**.
  - If a ≠ 1, gcd (a, 26) = 1: This cipher is likely **Affine Cipher**.
  - If a ≠ 1, gcd (a, 26) ≠ 1: Still unknow, so we should try different x_1, x_2, y_1, y_2.

- For **Substitution cipher**, because all letters can be random, but they are still replaced by a different letter based on a fixed substitution rule. In this case, it suggests constructing a substitution key based on frequency analysis and letter patterns.

For **polyalphabetic ciphers**, here is step by step that we can do to determine the type of decipher method:

- Analyzing the frequency of letters in the ciphertext: the frequency of letters in **Vigenère cipher** still somewhat reflect natural language frequencies, while the frequency of letters in **One-Time Pad cipher** has no noticeable frequency distribution.
- Using Kasiski test: the difference of frequency among trigrams in **Vigenère cipher** is easily detected, while in **One-Time Pad cipher**, the difference is nearly insignificant.
- Using Index of Coincidence (IC): For **Vigenère cipher**, since the key is repeated, the IC for different sections of the ciphertext should resemble that of natural language (around 0.065 for English), while for **One-Time Pad cipher,** the IC should be closer to 0.038 (the value expected for random data) because the key is entirely random and non-repeating.
- If the cipher is **One-Time Pad cipher**, it seems impossible for the class's requirement, and better to give up trying to decrypt cyphertext, as the key's characteristics are:
    - Truly random: The key must be a sequence of random bits or characters, with no discernible pattern.
    - As long as the plaintext: Each character or bit of the key is used exactly once for each character or bit of the plaintext.
    - Used only once: The same key is never reused for multiple messages.
    - Kept completely secret: If the key is never exposed, so is the encryption.
- If the cipher is **Vigenère cipher**, we can use the Kasiski or the Index of Coincidence (IC) method to estimate the length of the key. Once we determine the key length, we can calculate the IC for each block of that length to recover the keyword.

5.1. Ciphertext #1

We calculated the Index of Coincidence as 0.0671, indicating that the cipher is likely monoalphabetic. Through frequency analysis, we observed that the letters **U** and **J** appear most frequently. Using the Kasiski test, we identified that the trigram **JXU** and digrams **JX, XU** are repeated often. The letter frequency distribution in the ciphertext closely matches the standard frequencies. Thus, this suggests following the Shift cipher's approach.

By applying this to the system formula, I discover:

- Mapped **U** to **E** and **J** to **T**.
- Found values for the affine cipher: a = 1, b = 16.
- Since gcd (1, 26) = 1, we can use either the Shift Cipher or the Affine Cipher.

Finally, using a = 1 and b = 16 with the *affine_decrypt* function from *decipherMethod.py*, we successfully decrypted the ciphertext and retrieved the plaintext.

Expressed numerically, we have the following result:

Cipher letter	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Number	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Decipher letter	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
Number	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	0	1	2	3	4	5	6	7	8	9

When applied to the initial letters of the ciphertext, we observe the following:

Ciphertext : BUJCURUWYDROIQOYDWJXQDAIJEQBB

Plaintext : LETMEBEGINBYSAYINGTHANKSTOALL

Thus, ciphertext #1 can be decrypted using the Affine cipher. This conclusion is supported by a brute-force approach, which confirms that a Shift cipher could also decode the ciphertext. Therefore, we conclude that ciphertext #1 can be successfully decrypted using either the **Shift cipher or the Affine cipher**.

*** **Note:** In the program, you will notice that I used the Shift cipher, which is performed by the function *shift_decrypt* for ciphertext #1. This choice aligns with the structure of the game I designed, where the first key only requires a single number. We can still solve this problem by using the function *affine_decrypt*.

5.2. Ciphertext #2

We calculated the Index of Coincidence as 0.0708, indicating that the cipher is likely monoalphabetic. Through frequency analysis, we observed that the letters **G** and **V** appear most frequently. Using the Kasiski test, we identified that the trigram **VJG** and digrams **VJ, JG** are repeated often. The letter frequency distribution in the ciphertext closely matches the standard frequencies. Thus, this suggests following the Shift cipher's approach.

By applying this to the system formula, I discover:

- Mapped **G** to **E** and **V** to **T**.
- Found values for the affine cipher: a = 1, b = 2.

- Since gcd (1, 26) = 1, we can use either the Shift cipher or the Affine cipher.

Finally, using a = 1 and b = 2 with the *affine_decrypt* function from *decipherMethod.py*, we successfully decrypted the ciphertext and retrieved the plaintext.

Expressed numerically, we have the following result:

Cipher letter	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Number	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Decipher letter	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O	P	Q	R	S	T	U	V	W	X
Number	24	25	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

When we apply to the initial letters of the ciphertext, we observe the following:

Ciphertext : YGVJGRGQRNGQHVJGWPKVGFUVCVGU

Plaintext: : WETHEPEOPLEOFTHEUNITEDSTATES

Thus, ciphertext #2 can be decrypted using the Affine cipher. This conclusion is supported by a brute-force approach, which confirms that a Shift cipher could also decode the ciphertext. Therefore, we conclude that ciphertext #2 can be successfully decrypted using either the **Shift cipher or the Affine cipher**.

5.3. Ciphertext #3

We calculated the Index of Coincidence as 0.0384, suggesting the cipher is likely polyalphabetic. Frequency analysis shows that the letter frequencies in the ciphertext are evenly distributed. The Kasiski test reveals that the trigram **VZO** appears most frequently, occurring 5 times. However, other trigrams consistently appear with a frequency of 4. A similar pattern is observed with digrams, where their frequencies are relatively close to each other. This pattern strongly indicates that the ciphertext is likely encrypted using a **One-Time Pad cipher**.

Furthermore, when we use the Kasiski method and Index of Coincidence method, we cannot find the length of the keyword. For more detail, you can look at my *analysis3.txt* stored in '**analysis**' folder. Upon examining ciphertext #3, which contains 8,037 characters, I noticed it has the exact same length as ciphertext #4. This led me to suspect that both ciphertexts may share the same plaintext but were encrypted using different cipher methods. As a result, I opted to bypass decryption of ciphertext #3 and instead focused my efforts on deciphering ciphertext #4.

5.4. Ciphertext #4

We calculated the Index of Coincidence as 0.0430, suggesting the cipher is likely polyalphabetic. Frequency analysis shows that the letter frequencies in the ciphertext seem naturally distributed, reflecting natural language frequencies. The Kasiski test reveals that the difference of frequency among trigrams and digrams appears significantly. It likely suggests that the cipher method is **Vigenère cipher.**

To uncover the key, we can use either the Kasiski method or the Index of Coincidence method.

Using the Kasiski method: <Running function *find_key_length_kasiski_method* in *helper.py*>

```
200 '''
201 # Function to find keyword's length using Kasiski Test.
202 '''
203 def find_key_length_kasiski_method(all_position_of_one_trigram):
204     group_of_gcd = []
205     for i in range(len(all_position_of_one_trigram)-2):
206         group_of_gcd.append(gcd(all_position_of_one_trigram[i+1] - all_position_of_one_trigram[i],
207                                 all_position_of_one_trigram[i+2] - all_position_of_one_trigram[i+1]))
208     key_length = Counter(group_of_gcd).most_common(1)[0][0]
209     return Counter(group_of_gcd), key_length
210
```

We identified the trigram **DLV** as the most frequent. We then compiled all the positions where this trigram appears:

[1011, 1207, 1578, 1676, 2068, 2180, 2187, 2313, 2327, 2355, 3650, 3671, 3853, 4168, 4665, 4714, 5421, 5428, 5645, 5652, 5743, 6114, 6331, 6674, 7234, 7472, 7717, 7920]

Examining these positions, we found that the greatest common divisor (gcd) of the differences between two adjacent positions is consistently 7. For instance:

- gcd(1207 – 1011, 1578 – 1207) = gcd (196, 371) = 7
- gcd(1578 – 1207, 1676 - 1578) = gcd (371, 98) = 7

This suggests that the length of the keyword is likely 7.

Using Index of Coincidence: <Running function *find_key_length_ic_method* in *helper.py*>

```
221 '''
222 # Function to find keyword's length using Index of Coincidence method.
223 '''
224 def find_key_length_ic_method(ciphertext, max_key_len):
225     average_ics = {}
226     for length in range(1, max_key_len + 1):
227         avg_ic = average_ic_for_keyword_length(ciphertext, length)
228         average_ics[length] = avg_ic
229     return average_ics
230
```

We found that:

Key length	Index of Coincidence (IC)
1	0.0430
2	0.0430
3	0.0429
4	0.0430
5	0.0430
6	0.0429
7	0.0661
8	0.0429
9	0.0428
10	0.0429

At a key length of 7, the Index of Coincidence (IC) aligns closely with that of typical English text. This strongly suggests that the keyword length is likely 7.

After we know that the key length is 7, here is the step by step to decipher the ciphertext:

- Divide into 7 groups, where each group contain the letter at position:
  - Group 1: [0, 7, 14, …]
  - Group 2: [1, 8, 15, …]
  - Group 3: [2, 9, 16, …]

    …
  - Group 7: [6, 13, 20, …]
- Count the occurrences of each letter in each group.
- Calculate index of coincidence of each group by using this formula:

  $$\text{IC} = \frac{\sum_{i=0}^{25}(p_i * f_{i+g})}{n}$$

  Where:

  - p_i is the probability of each letter (a to z) in alphabet text,

  - f_{i+g} is the frequency of the letter in alphabet shifted by g ($0 \leq g \leq 25$, g is an integer)

  - n is the total amount of letters in each group.

After calculating the index of coincidence of each group, this is what we found.

Group	Index of Coincidence
1	0.0389 0.0355 0.0329 0.0449 0.0360 0.0319 0.0375 **0.0659** 0.0396 0.0311 0.0326 0.0456 0.0324 0.0364 0.0393 0.0352 0.0331 0.0378 0.0437 0.0396 0.0440 0.0386 0.0443 0.0370 0.0344 0.0332
2	0.0373 0.0387 0.0321 0.0336 0.0392 0.0453 0.0374 0.0441 0.0395 0.0434 0.0377 0.0357 0.0336 0.0378 0.0354 0.0321 0.0428 0.0342 0.0317 0.0399 **0.0662** 0.0387 0.0313 0.0346 0.0457 0.0330
3	0.0343 0.0324 0.0376 0.0454 0.0396 0.0421 0.0397 0.0450 0.0371 0.0338 0.0344 0.0391 0.0360 0.0330 0.0432 0.0337 0.0315 0.0379 **0.0659** 0.0403 0.0318 0.0316 0.0451 0.0353 0.0369 0.0385
4	0.0384 0.0330 0.0337 0.0396 0.0359 0.0329 0.0445 0.0348 0.0304 0.0383 **0.0665** 0.0408 0.0313 0.0331 0.0448 0.0335 0.0351 0.0396 0.0349 0.0329 0.0365 0.0458 0.0405 0.0405 0.0384 0.0456
5	0.0454 0.0336 0.0310 0.0404 **0.0660** 0.0393 0.0312 0.0341 0.0427 0.0332 0.0358 0.0392 0.0319 0.0349 0.0397 0.0454 0.0386 0.0433 0.0398 0.0446 0.0382 0.0327 0.0324 0.0370 0.0370 0.0340
6	0.0337 0.0390 0.0455 0.0391 0.0427 0.0394 0.0426 0.0367 0.0330 0.0360 0.0392 0.0354 0.0320 0.0446 0.0362 0.0320 0.0383 **0.0646** 0.0396 0.0310 0.0330 0.0439 0.0327 0.0361 0.0403 0.0348
7	0.0338 0.0321 0.0368 0.0457 0.0390 0.0431 0.0394 0.0426 0.0381 0.0345 0.0350 0.0393 0.0360 0.0309 0.0432 0.0349 0.0318 0.0385 **0.0656** 0.0400 0.0309 0.0352 0.0468 0.0329 0.0351 0.0400

By selecting the Index of Coincidence of approximately 0.065 from each group, we determined the positions of the key to be *K = [7, 20, 18, 10, 4, 17, 18]*. Converting these numbers to letters results in *K = [H, U, S, K, E, R, S]*.

When we apply to the initial letters of the ciphertext, we observe the following:

Key : HUSKERS

Ciphertext : JUDVQVAZBEKICKVGWIIRJZUY

Plaintext: : CALLMEISHMAELSOMEYEAR

Cipher letter	J	U	V	Q	V	A	Z	B	E	K	I	C	K	V	G	W	I	I	R	J	Z	U	Y
Number	9	20	21	16	21	0	25	1	4	10	8	2	10	21	6	22	8	8	17	9	25	20	24
Key	H	U	S	K	E	R	S	H	U	S	K	E	R	S	H	U	S	K	E	R	S	H	U
Number	7	20	18	10	4	17	18	7	20	18	10	4	17	18	7	20	18	10	4	17	18	7	20
Decipher letter	C	A	L	L	M	E	I	S	H	M	A	E	L	S	O	M	E	Y	E	A	R	S	A
Decipher letter	2	0	11	11	12	4	8	18	7	12	0	4	11	18	14	12	4	24	4	0	17	18	0

Therefore, the key is **"HUSKERS".** This means we can use the **Vigenère cipher** to decrypt ciphertext #4.

6. CONCLUSION

I successfully decrypted 3 out of 4 ciphertexts from this assignment: ciphertexts #1, #2, and #4. Ciphertext #3, however, uses a One-Time Pad, making it nearly impossible to break since the key is randomly generated and difficult to determine the correct key length. This is merely my guess that the plaintext for #3 are the same for #4, due to having the same ciphertext's length. #1 and #2 can be decrypted using either a Shift cipher or an Affine cipher, while #4 can be decrypted with the Vigenère cipher. Here's a summary of the key and cipher method for each ciphertext:

* #1 (Monoalphabetic cipher): - Using Shift decipher - Key: 10
  - Using Affine decipher - Key: [1,16]
* #2 (Monoalphabetic cipher): - Using Shift decipher - Key: 24
  - Using Affine decipher - Key: [1,2]
* #3 (Polyalphabetic cipher): - One-Time Pads cipher - Key: Impossible to find the key.
* #4 (Polyalphabetic cipher): - Using Vigenère decipher - Key: HUSKERS

7. TRACKING CODES MAP

readme.txt: Contain many important notes. Basically, it is my journal, and draft of my workshop.
main: Main program. Using for executing the program.
cryptanalysis: Analyze Cipher program. Contain frequency analysis, Kasiski test, and Index of Coincidence. Using to analyze cipher supporting in identifying cipher method.
identifyCipher.py: Use to detect cipher method, whether it is mono or poly alphabet, Shift or Affine or Substitution, Vigenère or One-Time Pads.
decipherMethod.py: Contain decipher methods, such as Shift, Affine, and Vigenère decipher.
decrypt.py: Contain whole process of each ciphertext, including cryptanalysis, identify cipher, and decryption method.
helper.py: Contain many helper methods supporting the whole program.
textfile.py: File I/O program.
keys.py: Keys for the little game that I set up. TA or any graders can use this program to bypass the system.
brute_force.py: Shift Brute force programming. This is just for my fun, and curiosity to use brute force to decrypt ciphertexts.