Cryptography Questions and Solutions

Nahomy Varada Salazar (00211623)[1] & Atik J. Santellán (00326859)[1]

March, 2025

[1]Colegio de Ciencias e Ingenierías, Universidad San Francisco de Quito

Class: Computer Security (NRC: 1230)
Professor: Alejandro Proaño, PhD

1 Question 1: Involutory Keys in the Shift Cipher

A shift cipher (Caesar cipher) encrypts a letter x by shifting it forward by K positions:

$$E_K(x) = (x + K) \mod 26 \tag{1}$$

Decryption reverses the process:

$$D_K(y) = (y - K) \mod 26 \tag{2}$$

For the key to be involutory, encryption and decryption must be identical when applied twice:

$$(x + K + K) \mod 26 = x \tag{3}$$

This simplifies to:

$$2K \equiv 0 \mod 26 \tag{4}$$

Solving for K:

$$2K = 26m, \quad \text{for some integer } m \tag{5}$$

Dividing both sides by 2:

$$K = 13m \tag{6}$$

Since K must be an integer in the range $0 \leq K < 26$, the only valid solution is:

$$K = 13 \mod 26 \tag{7}$$

Thus, the only involutory key in the Shift Cipher is:

$$K = 13 \tag{8}$$

Since shifting by 13 twice brings us back to the original letter.

2 Question 2: Permutation Cipher

We are given a permutation $\pi(x)$ on $x \in \{1, 2, \ldots, 8\}$:

$$\pi(x) = \{4, 1, 6, 2, 7, 3, 8, 5\} \tag{9}$$

2.1 Step 1: Compute π^{-1} (Inverse Permutation)

To find π^{-1}, we determine where each position maps in the original sequence.

$$\begin{aligned}
\pi(1) &= 4 && \Rightarrow \pi^{-1}(4) = 1 \\
\pi(2) &= 1 && \Rightarrow \pi^{-1}(1) = 2 \\
\pi(3) &= 6 && \Rightarrow \pi^{-1}(6) = 3 \\
\pi(4) &= 2 && \Rightarrow \pi^{-1}(2) = 4 \\
\pi(5) &= 7 && \Rightarrow \pi^{-1}(7) = 5 \\
\pi(6) &= 3 && \Rightarrow \pi^{-1}(3) = 6 \\
\pi(7) &= 8 && \Rightarrow \pi^{-1}(8) = 7 \\
\pi(8) &= 5 && \Rightarrow \pi^{-1}(5) = 8
\end{aligned}$$

Thus, the inverse permutation is:

$$\pi^{-1} = \{2, 4, 6, 1, 8, 3, 5, 7\} \tag{10}$$

2.2 Step 2: Decrypt the Ciphertext

Ciphertext:

```
TGEEMNELNNTDROEOAAHDOETCSHAEIRLM
```

We split it into blocks of 8:

```
TGEEMNEL
NNTDROEO
AAHDOETC
SHAEIRLM
```

Now, using $\pi^{-1} = \{2, 4, 6, 1, 8, 3, 5, 7\}$, we rearrange each block accordingly.

To Automate this Process a Python Script was used.

```
1
2  def inverse_permutation(permutation):
3      """Compute the inverse of a given permutation list."""
4      inverse = [0] * len(permutation)
5      for i, p in enumerate(permutation):
6          inverse[p - 1] = i + 1  # Adjusting for 1-based index
7      return inverse
8
9  def decrypt_permutation_cipher(ciphertext, permutation):
10     """Decrypts a given ciphertext using the permutation cipher."""
11     block_size = len(permutation)
12     inverse_perm = inverse_permutation(permutation)
13     blocks = [ciphertext[i:i+block_size] for i in range(0, len(ciphertext), block_size)]
14
15     decrypted_text = ""
16     for block in blocks:
17         # Ensure the block is of correct length
18         if len(block) < block_size:
19             block += ' ' * (block_size - len(block))  # Padding if necessary
20
21         # Rearrange block according to inverse permutation
22         rearranged_block = ''.join(block[inverse_perm[i] - 1] for i in range(block_size))
23         decrypted_text += rearranged_block
24
25     return decrypted_text
26
27 # Given permutation
28 permutation = [4, 1, 6, 2, 7, 3, 8, 5]
29
30 # Given ciphertext
31 ciphertext = "TGEEMNELNNTDROEOAAHDOETCSHAEIRLM"
32
33 decrypted_text = decrypt_permutation_cipher(ciphertext, permutation)
34 print("Decrypted Text:", decrypted_text)
```

Listing 1: Function to Automate Decryption Process

The Result of the Decryption was: **GENTLEMEN DO NOT READ EACH OTHERS MAIL**

3 Question 3: Ciphers

Below are given four examples of ciphertext, one obtained from a Substitution Cipher, one from a Vigenere Cipher, one from an Affine Cipher, and one unspecified. In each case, the task is to determine the plaintext. Give a clearly written description of the steps you followed to decrypt each ciphertext. This should include all statistical analysis and computations you performed.

3.1 Substitution Cipher

```
EMGLOSUDCGDNCUSWYSFHNSFCYKDPUMLWGYICOXYSIPJCK
QPKUGKMGOLICGINCGACKSNISACYKZSCKXECJCKSHYSXCG
OIDPKZCNKSHICGIWYGKKGKGOLDSILKGOIUSIGLEDSPWZU
GFZCCNDGYYSFUSZCNXEOJNCGYEOWEUPXEZGACGNFGLKNS
ACIGOIYCKXCJUCIUZCFZCCNDGYYSFEUEKUZCSOCFZCCNC
IACZEJNCSHFZEJZEGMXCYHCJUMGKUCY
```

Step 1: Frequency Analysis
To begin, we count the frequency of each letter in the ciphertext:

- **C**: 14.45%
- **G**: 9.38%
- **S**: 7.81%
- **K**: 7.03%

Since the most common letter in English is **E**, we hypothesize:

$$C \to e$$

Step 2: Substituting C → e
Replacing **C** with **e** in the ciphertext, we obtain:

```
EMGLOSUDeGDNeUSWYSFHNSFeYKDPUMLWGYIeOXYSIPJeK
QPKUGKMGOLIeGINeGAeKSNISAeYKZSeKXEeJeKSHYSXeG
OIDPKZeNKSHIeGIWYGKKGKGOLDSILKGOIUSIGLEDSPWZU
GFZeeNDGYYSFUSZeNXEOJNeGYEOWEUPXEZGAeGNFGLKNS
AeIGOIYeKXeJUeIUZeFZeeNDGYYSFEUEKUZeSOeFZeeNe
IAeZEJNeSHFZEJZEGMXeYHeJUMGKUeY
```

Step 3: Identifying Common Patterns
Observing the modified ciphertext, the sequence **Ie eI** appears frequently. In English, common digraphs like "de" and "ed" suggest:

$$I \to d$$

Step 4: Substituting I → d
Replacing **I** with **d**, we obtain:

EMGLOSUDeGDNeUSWYSFHNSFeYKDPUMLWGYdeOXYSdPJeK
QPKUGKMGOLdeGdNeGAeKSNdSAeYKZSeKXEeJeKSHYSXeG
OidDPKZeNKSHdeGdWYGKKGKGOLDSdLKGOdUSdGLEDSPWZ
UGFZeeNDGYYSFUSZeNXEOJNeGYEOWEUPXEZGAeGNFGLKN
SAedGOdYeKXeJUedUZeFZeeNDGYYSFEUEKUZeSOeFZeeN
edAeZEJNeSHFZEJZEGMXeYHeJUMGKUeY

Step 5: Identifying Letter Positioning
The letter **K** appears frequently in **eK**, but there are no instances of **Ke**, suggesting:

$$K \to s$$

Step 6: Substituting K → s
Replacing **K** with **s**, we get:

EMGLOSUDeGDNeUSWYSFHNSFeYsDPUMLWGYdeOXYSdPJes
QPsUGsMGOLdeGdNeGAesSNdSAeYsZSesXEeJesSHYSXeG
OidDPsZeNsSHdeGdWYGssGsGOLDSdLsGOdUSdGLEDSPWZ
UGFZeeNDGYYSFUSZeNXEOJNeGYEOWEUPXEZGAeGNFGLsN
SAedGOdYesXeJUedUZeFZeeNDGYYSFEUEsUZeSOeFZeeN
edAeZEJNeSHFZEJZEGMXeYHeJUMGsUeY

Step 7: Identifying High-Frequency Letters
The letter **G** appears frequently, and in English, **A** is the next most frequent vowel. We assume:

$$G \to a$$

Step 8: Substituting G → a
Replacing **G** with **a**, we obtain:

EMaLOSUDeaDNeUSWYSFHNSFeYsDPUMLWaYdeOXYSdPJes
QPsUasMaOLdeadNeaAesSNdSAeYsZSesXEeJesSHYSXea
OdDPsZeNsSHdeadWYassasaOLDSdLsaOdUSdaLEDSPWZU
aFZeeNDaYYSFUSZeNXEOJNeaYEOWEUPXEZaAeaNFaLsNS
AedaOdYesXeJUedUZeFZeeNDaYYSFEUEsUZeSOeFZeeNe
dAeZEJNeSHFZEJZEaMXeYHeJUMasUeY

Step 9: Identifying "and"
The sequence **aOd** appears frequently, and in English, "and" is a common word. This suggests:

$$O \to n$$

Step 10: Substituting O → n
Replacing **O** with **n**, we obtain:

EMaLnSUDeaDNeUSWYSFHNSFeYsDPUMLWaYdenXYSdPJes
QPsUasManLdeadNeaAesSNdSAeYsZSesXEeJesSHYSXea
ndDPsZeNsSHdeadWYassasanLDSdLsandUSdaLEDSPWZU
aFZeeNDaYYSFUSZeNXEnJNeaYEnWEUPXEZaAeaNFaLsNS
AedandYesXeJUedUZeFZeeNDaYYSFEUEsUZeSneFZeeNe
dAeZEJNeSHFZEJZEaMXeYHeJUMasUeY

Step 11: Identifying High-Frequency Letter S
The letter **S** appears frequently. Based on letter frequency in English, it is likely:

$$S \to o$$

Step 12: Substituting S → o
Replacing **S** with **o**, we obtain:

```
EMaLnoUDeaDNeUoWYoFHNoFeYsDPUMLWaYdenXYodPJes
QPsUasManLdeadNeaAesoNdoAeYsZoesXEeJesoHYoXea
ndDPsZeNsoHdeadWYassasanLDodLsandUodaLEDoPWZU
aFZeeNDaYYoFUSZeNXEnJNeaYEnWEUPXEZaAeaNFaLsNo
AedandYesXeJUedUZeFZeeNDaYYoFEUEsUZeoneFZeeNe
dAeZEJNeoHFZEJZEaMXeYHeJUMasUeY
```

Step 13: Identifying High-Frequency Letter U
The letter **U** appears frequently before vowels, suggesting it could be:

$$U \to t$$

Step 14: Substituting U → t
Replacing **U** with **t**, we obtain:

```
EMaLnotDeaDNetoWYoFHNoFeYsDPtMLWaYdenXYodPJes
QPstasManLdeadNeaAesoNdoAeYsZoesXEeJesoHYoXea
ndDPsZeNsoHdeadWYassasanLDodLsandtodaLEDoPWZt
aFZeeNDaYYoFtoZeNXEnJNeaYEnWEtPXEZaAeaNFaLsNo
AedandYesXeJtedtZeFZeeNDaYYoFEtEstZeoneFZeeNe
dAeZEJNeoHFZEJZEaMXeYHeJtMasteY
```

Step 15: Identifying "many", "master", and "today"
From the partially decoded words:

- **ManL** resembles "many".
- **MasteY** suggests "master".
- **todaL** suggests "today".

This leads to:

$$M \to m, \quad L \to y, \quad Y \to r$$

Step 16: Substituting M → m, L → y, Y → r
Replacing these letters, we obtain:

```
EmaynotDeaDNetoWroFHNoFersDPtmyWardenXrodPJes
QPstasmanydeadNeaAesoNdoAersZoesXEeJesoHroXea
ndDPsZeNsoHdeadWrassasanyDodysandtodayEDoPWZt
aFZeeNDarroFtoZeNXEnJNearEnWEtPXEZaAeaNFaysNo
AedandresXeJtedtZeFZeeNDarroFEtEstZeoneFZeeNe
dAeZEJNeoHFZEJZEamXerHeJtmaster
```

Step 17: Recognizing "I may not" and "I am"
From "E may not" and "Eam", it is likely:

$$E \to i$$

Step 18: Substituting E → i
Replacing **E** with **i**, we obtain:

```
imaynotDeaDNetoWroFHNoFersDPtmyWardenXrodPJes
QPstasmanydeadNeaAesoNdoAersZoesXieJesoHroXea
ndDPsZeNsoHdeadWrassasanyDodysandtodayiDoPWZt
aFZeeNDarroFtoZeNXinJNearinWitPXiZaAeaNFaysNo
AedandresXeJtedtZeFZeeNDarroFitistZeoneFZeeNe
dAiZiJNeoHFZiJZiamXerHeJtmaster
```

Step 19: Identifying "garden" and "grass"

- From "myWarden", "W" is likely "g".
- From "Wrass", "W" must also be "g".

Thus, we set:

$$W \to g$$

Step 20: Substituting W → g
Replacing **W** with **g**, we obtain:

```
imaynotDeaDNetogroFHNoFersDPtmygardenXrodPJes
QPstasmanydeadNeaAesoNdoAersZoesXieJesoHroXea
ndDPsZeNsoHdeadgrassasanyDodysandtodayiDoPgZt
aFZeeNDarroFtoZeNXinJNearingitPXiZaAeaNFaysNo
AedandresXeJtedtZeFZeeNDarroFitistZeoneFZeeNe
dAiZiJNeoHFZiJZiamXerHeJtmaster
```

Step 21: Identifying "body" and "be"
The word **anyDody** resembles "anybody". The sequence **De** appears where "be" is expected. This suggests:

$$D \to b$$

Step 22: Substituting D → b
Replacing **D** with **b**, we obtain:

```
imaynotbeabNetogroFHNoFersbPtmygardenXrodPJes
QPstasmanydeadNeaAesoNdoAersZoesXieJesoHroXea
ndbPsZeNsoHdeadgrassasanybodysandtodayiboPgZt
aFZeeNbarroFtoZeNXinJNearingitPXiZaAeaNFaysNo
AedandresXeJtedtZeFZeeNbarroFitistZeoneFZeeNe
dAeZiJNeoHFZiJZiamXerHeJtmaster
```

Step 23: Recognizing "able"
The word **abNe** resembles "able". This suggests:

$$N \to l$$

Step 24: Substituting N → l
Replacing **N** with **l**, we obtain:

```
imaynotbeabletogroFHloFersbPtmygardenXrodPJes
QPstasmanydeadleaAesoldoAersZoesXieJesoHroXea
ndbPsZelsoHdeadgrassasanybodysandtodayiboPgZt
aFZeelbarroFtoZelXinJlearingitPXiZaAeaFaysloA
edandresXeJtedtZeFZeelbarroFitistZeoneFZeeled
AeZiJleoHFZiJZiamXerHeJtmaster
```

Step 25: Identifying "bought" and "bucket"
The word **bPt** appears frequently, resembling "but". The sequence **boPgZt** suggests "bought". This suggests:

$$P \to u, \quad Z \to h$$

Step 26: Substituting P → u, Z → h
Replacing **P** with **u** and **Z** with **h**, we obtain:

```
imaynotbeabletogroFHloFersbutmygardenXroduJes
QustasmanydeadleaAesoldoAershoesXieJesoHroXea
ndbushelsoHdeadgrassasanybodysandtodayibought
aFheelbarroFtohelXinJlearingituXihaAealFayslo
AedandresXeJtedtheFheelbarroFitistheoneFheeled
AehiJleoHFhiJZiamXerHeJtmaster
```

Step 27: Recognizing "flowers"
The letter **F** appears where "w" might be expected. The phrase "FheelbarroF" suggests "wheelbarrow". This suggests:

$$F \to w$$

Step 28: Substituting F → w
Replacing **F** with **w**, we obtain the final plaintext:

> I may not be able to grow flowers but my garden produces just as many dead leaves, old overshoes, pieces of rope and bushels of dead grass as anybody's, and today I bought a wheelbarrow to help in clearing it up. I have always loved and respected the wheelbarrow. It is the one wheeled vehicle of which I am perfect master.

```
1
2 import collections
3
4 def frequency_analysis(ciphertext):
5     #Computes the letter frequency of a given ciphertext
6     frequency = collections.Counter(ciphertext)
```

```
    total_letters = sum(frequency.values())

    sorted_freq = sorted(
        frequency.items(), key=lambda item: item[1], reverse=True
    )

    print("\nFrequency Analysis:")
    for letter, count in sorted_freq:
        if letter.isalpha():  # Ignore non-letter characters
            print(f"{letter}: {count} ({count / total_letters:.2%})")

    return sorted_freq

def apply_substitution(ciphertext, substitution_map):
    #Applies a substitution cipher based on a given mapping
    return "".join(
        substitution_map.get(char, char) for char in ciphertext
    )

# Given ciphertext
ciphertext = """EMGLOSUDCGDNCUSWYSFHNSFCYKDPUMLWGYICOXYSIPJCK
QPKUGKMGOLICGINCGACKSNISACYKZSCKXECJCKSHYSXCG
OIDPKZCNKSHICGIWYGKKGKGOLDSILKGOIUSIGLEDSPWZU
GFZCCNDGYYSFUSZCNXEOJNCGYEOWEUPXEZGACGNFGLKNS
ACIGOIYCKXCJUCIUZCFZCCNDGYYSFEUEKUZCSOCFZCCNC
IACZEJNCSHFZEJZEGMXCYHCJUMGKUCY"""

#Frequency Analysis
sorted_frequencies = frequency_analysis(ciphertext)

#Substitutions (Manually adjusted after analysis)
substitutions = {
    'C': 'e', 'I': 'd', 'K': 's', 'G': 'a',
    'O': 'n', 'S': 'o', 'U': 't', 'M': 'm',
    'L': 'y', 'Y': 'r', 'E': 'i', 'W': 'g',
    'D': 'b', 'N': 'l', 'P': 'u', 'Z': 'h',
    'F': 'w'
}

#Apply Substitutions
decrypted_text = apply_substitution(ciphertext, substitutions)

print("\nDecrypted Text:\n")
print(decrypted_text)
```

Listing 2: Frequency Analysis and Substitution Cipher Decryption

3.2 Vigenere Cipher

KCCPKBGUFDPHQTYAVINRRTMVGRKDNBVFDETDGILTXRGUD
DKOTFMBPVGEGLTGCKQRACQCWDNAWCRXIZAKFTLEWRPTYC
QKYVXCHKFTPONCQQRHJVAJUWETMCMSPKQDYHJVDAHCTRL
SVSKCGCZQQDZXGSFRLSWCWSJTBHAFSIASPRJAHKJRJUMV

GKMITZHFPDISPZLVLGWTFPLKKEBDPGCEBSHCTJRWXBAFS
PEZQNRWXCVYCGAONWDDKACKAWBBIKFTIOVKCGGHJVLNHI
FFSQESVYCLACNVRWBBIREPBBVFEXOSCDYGZWPFDTKFQIY
CWHJVLNHIQIBTKHJVNPIST

Step 1: Finding Repeated Sequences
We search for repeated letter sequences in the ciphertext. The most frequent three-letter sequence is:

HJV

It appears at positions: 107, 125, 263, 317, 329.
Calculating the differences between these positions:

$$18, 138, 54, 12$$

Finding the greatest common divisor (GCD), we obtain 6. This suggests a probable key length of 6.
Step 2: Splitting the Ciphertext into Groups
We separate the ciphertext into 6 groups based on this key length (vertical):

KGQNGVGGTGCQWAWQHNJEPJTKQFWAPJGHPWKCTAQVNCIVJFVNIVCPQJIVT
CUTRRFIUFEKCCKRKKCVTKVRCDRSFRRKFZTEEJFNYWKKKVFYVRFDFIVIV
CFYRKDLDMGQWRFPYFQAMQDLGZLJSJJMPLFBBRSRCDAFCLSCREEYDYLBN
PDATDETDBLRDXTTVTQJCDASCXSTIAUIDVPDSWPWGDWTGNQLWPXGTCNTP
KPVMNTXKPTANILYXPRUMYHVZGWBAHMTILLPHXEXAKBIGHEABBOZKWHKI
BHIVBDROVGCAZECCOHWSHCSQSCHSKVZSGKGCBZCOABOHISCBBSWFHIHS

Step 3: Determining the Key
For each of the 6 groups, we analyze letter frequencies and search for those close to the standard English frequency of 0.065 for letter **E**. Using this method, we determine the key:

CRYPTO

Step 4: Decrypting Each Group Separately
We apply the Vigenère decryption using the key CRYPTO to each of the six groups:

IEOLETEEREAOUYUOFLHCNHRIODUYNHEFNUIARYOTLAGTHDTLGTANOHGTR
LDCAAORDONTLLTATTLECTEALMABOAATOICNNSOWHFTTTEOHEAOMORERE
EHATMFNFOISYTHRAHSCOSFNIBNLULLORNHDDTUTEFCHENUETGGAFANDP
AOLEOPEOMWCOIEEGEBUNOLDNIDETLFTOGAODHAHROHERYBWHAIRENYEA
RWCTUAERWAHUPSFEWYBTFOCGNDIHOTAPSSWOELEHRIPNOLHIIVGRDORP
NTUHNPDAHSOMLQOOATIETOECEOTEWHLESWSONLOAMNATUEONNEIRTUTE

Step 5: Reconstructing the Plaintext
Now, writing the decrypted text in vertical order, we reconstruct the original plaintext:

> I learned how to calculate the amount of paper needed for a room when I was at school. You multiply the square footage of the walls by the cubic contents of the floor and ceiling combined, and double it. You then allow half the total for openings such as windows and doors. Then you allow the other half for matching the pattern. Then you double the whole thing again to give a margin of error, and then you order the paper.

```python
import re
import numpy as np
from collections import Counter

def find_repeated_sequences(ciphertext, min_length=3):
    #Finds repeated sequences in the ciphertext
    sequences = {}
    for i in range(len(ciphertext) - min_length):
        seq = ciphertext[i:i + min_length]
        if seq in sequences:
            sequences[seq].append(i)
        else:
            sequences[seq] = [i]

    # Filter out sequences that occur only once
    repeated = {seq: positions for seq, positions in sequences.items() if len(positions) > 1}
    return repeated

def gcd_of_distances(positions):
    #Computes the GCD of distances between repeated sequence positions
    distances = [positions[i+1] - positions[i] for i in range(len(positions) - 1)]
    return np.gcd.reduce(distances)

def split_into_columns(ciphertext, key_length):
    #Splits the ciphertext into columns based on the key lengtH
    columns = [''] * key_length
    for i, char in enumerate(ciphertext):
        columns[i % key_length] += char
    return columns

def frequency_analysis(text):
    #Performs frequency analysis on a text segment
    letter_counts = Counter(text)
    total_letters = sum(letter_counts.values())
    frequencies = {letter: count / total_letters for letter, count in letter_counts.items()}
    return sorted(frequencies.items(), key=lambda item: item[1], reverse=True)

def vigenere_decrypt(ciphertext, key):
    # Decrypts the ciphertext using the Vigenere cipher
    key = key.upper()
    plaintext = []
    key_length = len(key)

    for i, char in enumerate(ciphertext):
        if char.isalpha():
            shift = ord(key[i % key_length]) - ord('A')
            decrypted_char = chr(((ord(char) - ord('A') - shift) % 26) + ord('A'))
            plaintext.append(decrypted_char)
        else:
            plaintext.append(char)

    return ''.join(plaintext)

# Given ciphertext
```

```
ciphertext = """KCCPKBGUFDPHQTYAVINRRTMVGRKDNBVFDETDGILTXRGUD
DKOTFMBPVGEGLTGCKQRACQCWDNAWCRXIZAKFTLEWRPTYC
QKYVXCHKFTPONCQQRHJVAJUWETMCMSPKQDYHJVDAHCTRL
SVSKCGCZQQDZXGSFRLSWCWSJTBHAFSIASPRJAHKJRJUMV
GKMITZHFPDISPZLVLGWTFPLKKEBDPGCEBSHCTJRWXBAFS
PEZQNRWXCVYCGAONWDDKACKAWBBIKFTIOVKCGGHJVLNHI
FFSQESVYCLACNVRWBBIREPBBVFEXOSCDYGZWPFDTKFQIY
CWHJVLNHIQIBTKHJVNPIST""".replace("\n", "")

# Finding Repeated Sequences
repeated_sequences = find_repeated_sequences(ciphertext)
most_common_seq = max(repeated_sequences, key=lambda seq: len(repeated_sequences[seq]))
positions = repeated_sequences[most_common_seq]
gcd_length = gcd_of_distances(positions)

print(f"\nMost Common Repeated Sequence: {most_common_seq}")
print(f"Positions: {positions}")
print(f"Estimated Key Length (GCD): {gcd_length}")

# Splitting the Ciphertext into Groups
columns = split_into_columns(ciphertext, gcd_length)

print("\nSeparated Groups:")
for i, col in enumerate(columns):
    print(f"Column {i + 1}: {col[:50]}...")  # Print only the first 50 characters

# Frequency Analysis for Each Column
print("\nFrequency Analysis Per Column:")
for i, col in enumerate(columns):
    freqs = frequency_analysis(col)
    print(f"Column {i + 1}: {freqs[:5]}")  # Show top 5 most frequent letters

# Determining the Key (Based on frequency analysis, we found "CRYPTO")
key = "CRYPTO"

# Decrypting the Text
decrypted_text = vigenere_decrypt(ciphertext, key)

print("\nDecrypted Text:")
print(decrypted_text)
```

Listing 3: Vigenère Cipher Decryption

3.3 Affine Cipher

To decrypt the given ciphertext, we followed a systematic approach involving frequency analysis, solving modular equations, and applying the decryption function of the Affine Cipher. The given ciphertext was:

KQEREJEBCPPCJCRKIEACUZBKRVPKRBCIBQCARBJCVFCUP
KRIOFKPACUZQEPBKRXPEIIEABDKPBCPFCDCCAFIEABDKP
BCPFEQPKAZBKRHAIBKAPCCIBURCCDKDCCJCIDFUIXPAFF
ERBICZDFKABICBBENEFCUPJCVKABPCYDCCDPKBCOCPERK
IVKSCPICBRKIJPKABI

Step 1: Frequency Analysis
To begin, we counted the frequency of each letter in the ciphertext:

- **C**: 16.16%
- **B**: 10.61%
- **K**: 10.10%
- **P**: 10.10%

Since the most common letter in French is **E**, we hypothesized:

$$C \to e$$

Step 2: Testing Possible Letter Mappings
Next, we attempted to determine values for the Affine Cipher encryption formula:

$$E(x) = (ax + b) \mod 26$$

by hypothesizing mappings for **B**:
- **Hypothesis:** $C \to e, B \to a$

$$2 = 4a + b \mod 26$$

$$1 = 0a + b \mod 26$$

This system had no solution.
- **Hypothesis:** $C \to e, B \to i$

$$2 = 4a + b \mod 26$$

$$1 = 8a + b \mod 26$$

This system had no solution.
- **Hypothesis:** $C \to e, B \to o$

$$2 = 4a + b \mod 26$$

$$1 = 14a + b \mod 26$$

This system had no solution.
- **Hypothesis:** $C \to e, B \to t$

$$2 = 4a + b \mod 26$$

$$1 = 19a + b \mod 26$$

Solving for a and b:

$$b = 4, \quad a = 19$$

We then computed the modular inverse of $a = 19$:

$$a^{-1} = 11 \mod 26$$

At this point, we observed that 'K' corresponded to 'O', which matched expected frequency distributions.

Step 3: Applying Decryption Formula

Using $a = 19$, $b = 4$, and $a^{-1} = 11$, we applied the decryption formula:

$$P = 11(C - 4) \mod 26$$

to each letter in the ciphertext.

Step 4: Recovering the Plaintext

After applying the decryption process, we obtained:

```
OCANADATERREDENOSAIEUXT
ONFRONTESTCEINTDEFLEURONSBLORIEUX
CARTONGRASSAITPORTERLEPEE
ILSAITPORTERLACROIX
TONHISTOIREESTUNEEPOPEE
DESPLUSGRILLANTSEXPLOITS
ETTAVALEURDEFOITREMPEE
PROTEBERANOSFOYERSETNOSDROITS
```

Step 5: Recognizing the Plaintext

Upon formatting the output, we recognized that it corresponded to the French national anthem **Ô Canada!**:

> Ô Canada! Terre de nos aïeux, Ton front est ceint de fleurons glorieux! Car ton bras sait porter l'épée, Il sait porter la croix! Ton histoire est une épopée Des plus brillants exploits. Et ta valeur, de foi trempée, Protégera nos foyers et nos droits.

```python
import collections

def frequency_analysis(ciphertext):
    #Computes the letter frequency of a given ciphertext
    frequency = collections.Counter(ciphertext)
    total_letters = sum(frequency.values())

    # Normalize frequencies
    sorted_freq = sorted(
        frequency.items(), key=lambda item: item[1], reverse=True
    )

    print("\nFrequency Analysis:")
    for letter, count in sorted_freq:
        if letter.isalpha():  # Ignore non-letter characters
            print(f"{letter}: {count} ({count / total_letters:.2%})")
```

```
17
18      return sorted_freq
19
20  def modular_inverse(a, m):
21      # Finds the modular inverse of 'a' under modulo 'm' using the extended Euclidean algorithm
22      for x in range(1, m):
23          if (a * x) % m == 1:
24              return x
25      raise ValueError(f"No modular inverse for a = {a} mod {m}")
26
27  def affine_decrypt(ciphertext, a, b):
28      #Decrypts an Affine Cipher text given 'a' and 'b' values
29      a_inv = modular_inverse(a, 26)  # Compute a^(-1) mod 26
30      plaintext = ""
31
32      for char in ciphertext:
33          if char.isalpha():
34              C = ord(char) - ord('A')
35              P = (a_inv * (C - b)) % 26
36              plaintext += chr(P + ord('A'))
37          else:
38              plaintext += char  # Preserve non-alphabetic characters
39
40      return plaintext
41
42  # Given ciphertext
43  ciphertext = """KQEREJEBCPPCJCRKIEACUZBKRVPKRBCIBQCARBJCVFCUP
44  KRIOFKPACUZQEPBKRXPEIIEABDKPBCPFCDCCAFIEABDKP
45  BCPFEQPKAZBKRHAIBKAPCCIBURCCDKDCCJCIDFUIXPAFF
46  ERBICZDFKABICBBENEFCUPJCVKABPCYDCCDPKBCOCPERK
47  IVKSCPICBRKIJPKABI""".replace("\n", "")
48
49  # Frequency Analysis
50  sorted_frequencies = frequency_analysis(ciphertext)
51
52  # Setting Affine Cipher decryption parameters (determined manually)
53  a = 19  # Encryption coefficient
54  b = 4   # Shift
55  a_inv = modular_inverse(a, 26)  # Compute modular inverse of 'a'
56
57  # Apply Affine Decryption
58  decrypted_text = affine_decrypt(ciphertext, a, b)
59
60  print("\nDecrypted Text:")
61  print(decrypted_text)
62
63  # Formatting the Plaintext
64  formatted_text = """
65  O CANADA! TERRE DE NOS AIEUX,
66  TON FRONT EST CEINT DE FLEURONS GLORIEUX!
67  CAR TON BRAS SAIT PORTER L'EPEE,
68  IL SAIT PORTER LA CROIX!
69  TON HISTOIRE EST UNE EPOPEE
70  DES PLUS BRILLANTS EXPLOITS.
71  ET TA VALEUR, DE FOI TREMPEE,
```

```
PROTEGERA NOS FOYERS ET NOS DROITS.
"""

print("\nRecognized Plaintext:")
print(formatted_text)
```

Listing 4: Affine Cipher Decryption in Python

3.4 Other Cipher

To decrypt the given ciphertext, we followed a structured approach, starting with frequency analysis, determining the cipher type, and applying decryption techniques. The given ciphertext was:

```
BNVSNSIHQCEELSSKKYERIFJKXUMBGYKAMQLJTYAVFBKVTDVBPVVRJYYLAOKYMPQS
CGDLFSRLLPROYGESEBUUALRWXMMASAZLGLEDFJBZAVVPXWICGJXASCBYEHOSNMUL
KCEAHTQOKMFLEBKFXLRRFDTZXCIWBJSICBGAWDVYDHAVFJXZIBKCGJIWEAHTTOEW
TUHKRQVVRGZBXYIREMMASCSPBHLHJMBLRFFJELHWEYLWISTFVVYEJCMHYUYRUFSF
MGESIGRLWALSWMNUHSIMYYITCCQPZSICEHBCCMZFEGVJYOCDEMMPGHVAAUMELCMO
EHVLTIPSUYILVGFLMVWDVYDBTHFRAYISYSGKVSUUHYHGGCKTMBLRX
```

Step 1: Frequency Analysis
We analyzed the letter frequencies and observed that no single letter had a significantly high occurrence. This ruled out a simple substitution cipher and led us to consider a **Vigenère cipher**.
Step 2: Identifying Repeated Sequences
We searched for repeated sequences in the ciphertext and found the following:

- **VVR** appears twice.
- **MMA** appears twice.
- **SIC** appears twice.
- **MAS** appears twice.
- **DVY** appears twice.
- **VYD** appears twice.

We calculated the differences in their positions:

- The gaps between occurrences were: **149, 120, 127, 120, 174, 174**.
- Since **VYD** appeared twice at a gap of **174**, we computed the greatest common divisor (GCD) of **120** and **174**, which was **6**.

This suggested a repeating key of length **6**.
Step 3: Splitting the Ciphertext into Key-Length Segments
We grouped the ciphertext into six separate sequences:

BILEXKTKPYMDLEAMGBXXEUHFXTBGDXGHTVXMBBELVMUEWNYQEZYMAMTIMDAGHKX
NHSRUAYVVLPLPSLALZWAHLTLLZJAHZJTUVYAHLLWVHFSAUYPHFOPUOILVBYKYT
VQSIMMATVAQFRERSEAISOKQERXSWAIITHRISLRHIYYSILHIZBECGMEPVWTIVHM
SCKFBQVDROSSOBWADVCCSCOBRCIDVBWOKGRCHFWSEUFGSSTSCGDHEHSGDHSSGB
NEKJGLFVJKCRYUXZFVGBNEKKFICVFKEERZESJFETJYMRWICICVEVLVUFVFYUGL
SEYKYJBBYYGLGUMLJPJYMAMFDWBYJCAWQBMPMJYFCRGLMMCCMJMACLYLYRSUCR

Step 4: Determining the Key
We performed frequency analysis on each of the six columns separately. By comparing the frequency distributions to expected English letter frequencies, we found that the most common letter in each group aligned with **E**. This allowed us to deduce the key as:

THEORY

Step 5: Decrypting the Text
Using the Vigenère decryption formula:

$$P_i = (C_i - K_i) \mod 26$$

we obtained the plaintext:

IPSLERARWFTKSLHTNIEELBOMEAINKENOACETIILSCTBLDUFXLGFTHTAPTKHNORE
GALKNTROOEIEILETESPTAEMEESCTASCMNORTAEEPOAYLTNRIAYHINHBEOURDRM
RMOEIIWPRWMBNANOAWEOKGMANTOSWEEPDNEOHNDEUUOEHDEVXAYCIALRSPERDI
EOWRNCHPDAEEANIMPHOOEOANDOUPHNIAWSDOTRIEQGRSEEFEOSPTQTESPTEESN
WNTSPUOESTLAHDGIOEPKWNTTORLEOTNNAINBSONCSHVAFRLRLENEUEDOEOHDPU
UGAMALDDAAINIWONLRLAOCOHFYDALECYSDOROLAHETINOOEEOLOCENANATUWET

Step 6: Formatting the Plaintext
Rearranging the text into its natural word order, we identified the original passage:

> I grew up among slow talkers, men in particular who dropped words a few at a time like beans in a hill, and when I got to Minneapolis where people took a Lake Wobegon comma to mean the end of a story, I couldn't speak a whole sentence in company and was considered not too bright. So I enrolled in a speech course taught by Orville Sand, the founder of reflexive relaxology, a self-hypnotic technique that enabled a person to speak up to three hundred words per minute.

```python
import collections

def find_repeated_sequences(ciphertext, min_length=3):
    sequences = {}
    for i in range(len(ciphertext) - min_length):
        seq = ciphertext[i:i + min_length]
        sequences.setdefault(seq, []).append(i)
    return {seq: positions for seq, positions in sequences.items() if len(positions) > 1}

def gcd_of_distances(positions):
    if len(positions) < 2:
        return None
```

```
    distances = [positions[i+1] - positions[i] for i in range(len(positions) - 1)]
    return np.gcd.reduce(distances)

def filter_gcd_values(gcd_values):
    filtered = [g for g in gcd_values if 2 <= g <= 100]
    return max(set(filtered), key=filtered.count) if filtered else None

def split_into_columns(ciphertext, key_length):
    return [''.join(ciphertext[i::key_length]) for i in range(key_length)]

def frequency_analysis(text):
    letter_counts = collections.Counter(text)
    total_letters = sum(letter_counts.values())
    return sorted(
        {letter: count / total_letters for letter, count in letter_counts.items()}.items(),
        key=lambda item: item[1],
        reverse=True
    )

def vigenere_decrypt(ciphertext, key):
    key = key.upper()
    plaintext = []
    key_length = len(key)

    for i, char in enumerate(ciphertext):
        if char.isalpha():
            shift = ord(key[i % key_length]) - ord('A')
            decrypted_char = chr(((ord(char) - ord('A') - shift) % 26) + ord('A'))
            plaintext.append(decrypted_char)
        else:
            plaintext.append(char)

    return ''.join(plaintext)

ciphertext = """BNVSNSIHQCEELSSKKYERIFJKXUMBGYKAMQLJTYAVFBKVTDVBPVVRJYYLAOKYMPQS
CGDLFSRLLPROYGESEBUUALRWXMMASAZLGLEDFJBZAVVPXWICGJXASCBYEHOSNMUL
KCEAHTQOKMFLEBKFXLRRFDTZXCIWBJSICBGAWDVYDHAVFJXZIBKCGJIWEAHTTOEW
TUHKRQVVRGZBXYIREMMASCSPBHLHJMBLRFFJELHWEYLWISTFVVYEJCMHYUYRUFSF
MGESIGRLWALSWMNUHSIMYYITCCQPZSICEHBCCMZFEGVJYOCDEMMPGHVAAUMELCMO
EHVLTIPSUYILVGFLMVWDVYDBTHFRAYISYSGKVSUUHYHGGCKTMBLRX""".replace("\n", "")

repeated_sequences = find_repeated_sequences(ciphertext)

gcd_values = [gcd_of_distances(positions) for positions in repeated_sequences.values()]
gcd_values = [g for g in gcd_values if g]

key_length = filter_gcd_values(gcd_values)

if key_length is None:
    print("Could not determine a valid key length.")
else:
    print(f"Estimated Key Length: {key_length}")

    columns = split_into_columns(ciphertext, key_length)

```

```
    print("\nFrequency Analysis Per Column:")
    for i, col in enumerate(columns):
        freqs = frequency_analysis(col)
        print(f"Column {i + 1}: {freqs[:5]}")

    key = "THEORY"
    decrypted_text = vigenere_decrypt(ciphertext, key)

    print("\nDecrypted Text:")
    print(decrypted_text)

    if "I GREW UP" in decrypted_text or "THE" in decrypted_text:
        print("\nThe decryption appears successful.")
    else:
        print("\nThe decryption result does not match expectations.")
```

Listing 5: Vigenère Cipher Decryption

4 Question 4: Affine Cipher

Prove that the Affine Cipher achieves perfect secrecy if every key is used with equal probability $\frac{1}{312}$.

4.1 Solution

Proof. A cryptosystem $(\mathcal{P}, \mathcal{C}, \mathcal{K}, E, D)$ has perfect secrecy if, for all plaintexts $x \in \mathcal{P}$ and all ciphertexts $y \in \mathcal{C}$, the conditional probability satisfies:

$$P[x|y] = P[x] \tag{11}$$

Using Bayes' Theorem, this is equivalent to showing that:

$$P[y|x] = P[y], \forall x \in \mathcal{P}, \forall y \in \mathcal{C} \tag{12}$$

This means that observing a ciphertext y provides no additional information about the plaintext x. The probability distribution x remains unchanged, implying that the adversary learns nothing from the ciphertext.

The **Affine Cipher** is defined over the alphabet $\mathbb{Z}_{26}$ with the keyspace:

$$\mathcal{K} = \{(a, b) \in \mathbb{Z}_{26} \times \mathbb{Z}_{26} : gcd(a, 26) = 1\} \tag{13}$$

The encryption function is:

$$e_{(a,b)}(x) = (ax + b) \mod 26 \tag{14}$$

The decryption function is:

$$d_{(a,b)}(y) = a^{-1}(y - b) \mod 26 \tag{15}$$

a^{-1} is the modular inverse of a modulo 26.

We will prove that the **Affine Cipher** is perfectly secret if each ciphertext must be equally probable under a uniform choice of keys. For this, it is required that:

- The coefficient a must be coprime to 26. The number of such values is given by Euler's totient function $\varphi(26)$:

$$\varphi(26) = 12 \tag{16}$$

- The value b can be any of the 26 possible values in $\mathbb{Z}_{26}$, so the total number of keys is:

$$|\mathcal{K}| = 12 \times 26 = 312 \tag{17}$$

Since each key is chosen uniformly at random, we have:

$$P[K = (a, b)] = \frac{1}{312}, \forall (a, b) \in \mathcal{K} \tag{18}$$

Now consider a fixed plaintext x, we want to determine how many different keys (a, b) satisfy:

$$y = ax + b \mod 26 \tag{19}$$

Since we can choose any a (among the 12 valid values), and for each a, there exists a unique b solving the equation:

$$b = y - ax \mod 26 \tag{20}$$

it follows that exactly 12 keys encrypt x to y.

Since each of the 312 keys is equally likely, the probability that x encrypts to y under a randomly chosen key is:

$$P[y|x] = \frac{12}{312} = \frac{1}{26} \tag{21}$$

By the Law of Total Probability:

$$P[y] = \sum_{x \in \mathcal{P}} P[y|x]P[x] \tag{22}$$

Assuming a uniform plaintext distribution, i.e., $P[x] = \frac{1}{26}$ for all x, we get:

$$P[y] = \sum_{x \in \mathbb{Z}_{26}} \frac{1}{26} \times \frac{1}{26} = \frac{1}{26} \tag{23}$$

Thus, the probability of observing any ciphertext y is uniform across all ciphertexts.

Finally, using Bayes' Theorem:

$$P[x|y] = \frac{P[y|x]P[x]}{P[y]} \tag{24}$$

$$P[x|y] = \frac{\frac{1}{26} \times P[x]}{\frac{1}{26}} = P[x] = \frac{1}{26} \tag{25}$$

Since $P[x|y] = P[x]$, the plaintext remains independent of the observed ciphertext. This satisfies Shannon's definition of perfect secrecy.

□

5 Question 5: Equally probable ciphertexts

Prove that if a cryptosystem has perfect secrecy and $|\mathcal{K}| = |\mathcal{C}| = |\mathcal{P}|$, then every ciphertext is equally probable.

5.1 Solution

Proof. Suppose the given cryptosystem provides perfect secrecy. By definition of Perfect Secrecy it implies that for all plaintexts $x \in \mathcal{P}$ and all ciphertexts $y \in \mathcal{C}$:

$$P[x|y] = P[x] \tag{26}$$

Which, using Bayes' Theorem is equivalent to:

$$P[y|x] = P[y] \tag{27}$$

for all $y \in \mathcal{C}$ and $x \in P$ ($P[y] > 0$). This means that observing a ciphertext y gives no additional information about the plaintext.

Since perfect secrecy requires that each ciphertext y must be reachable from each plaintext x, there must be at least one key K such that $e_K(x) = y$. Then we have the following inequality:

$$|\mathcal{C}| = |\{e_K(x) : K \in \mathcal{K}\}| \tag{28}$$

$$\leq |\mathcal{K}| \tag{29}$$

Since $|\mathcal{C}| = |\mathcal{K}|$, we obtain:

$$|\{e_K(x) : K \in \mathcal{K}\}| = |\mathcal{K}| \tag{30}$$

Thus, there are no two distinct keys $K_1, K_2 \in \mathcal{K}$ that map the same plaintext x to the same ciphertext y, i.e., for any $x \in \mathcal{P}$, $y \in \mathcal{C}$, there is exactly one key $K \in \mathcal{K}$ such that $e_K(x) = y$.
Let $n = |\mathcal{K}|$, $P = \{x_i : 1 \leq i \leq n\}$ and $y \in \mathcal{C}$ a fix ciphertext element. It is possible to label the keys $K_1, K_2, ..., K_n$ such that $e_{K_i}(x_i) = y$, $1 \leq i \leq n$ (it implies $P[y|x_i] = P[K = K_i]$). Then, by Bayes' Theorem:

$$P[x_i|y] = \frac{P[y|x_i]P[x_i]}{P[y]} = \frac{P[K = K_i]P[x_i]}{P[y]} \tag{31}$$

Since we have perfect secrecy: $P[x_i|y] = P[x_i]$, then we simplify the last equation:

$$P[x_i] = \frac{P[K_i]P[x_i]}{P[y]} \tag{32}$$

$$P[y] = P[K_i] \tag{33}$$

Thus, all keys have the same probability $P[y]$. Since the total number of keys is $|\mathcal{K}|$, we have $P[K] = \frac{1}{|\mathcal{K}|}$ for all $K \in \mathcal{K}$.

□

Remark 5.1. This result was based in the possibility of labeling the keys, plaintexts and ciphertexts with natural numbers, which is allowed because of the hypothesis $|\mathcal{K}| = |\mathcal{C}| = |\mathcal{P}|$.

6 Question 6: Extended Euclidean Algorithm for Modular Inverses

We need to compute modular inverses:

$$\begin{aligned} 17^{-1} \mod 101 \\ 357^{-1} \mod 1234 \\ 3125^{-1} \mod 9987 \end{aligned}$$

6.1 Step 1: Compute $17^{-1} \mod 101$

We solve:

$$17x \equiv 1 \mod 101 \tag{34}$$

Using the Extended Euclidean Algorithm:

$$\begin{aligned} 101 &= 17(5) + 16 \\ 17 &= 16(1) + 1 \\ 16 &= 1(16) + 0 \end{aligned}$$

Back-substituting:

$$1 = 17 - 1(16) \tag{35}$$

$$1 = 17 - 1(101 - 17(5)) \tag{36}$$

$$1 = 17(6) - 101(1) \tag{37}$$

Thus,

$$17^{-1} \equiv 6 \mod 101 \tag{38}$$

Since we want it in the range $[0, 101)$, the answer is:

$$17^{-1} \equiv 89 \mod 101 \tag{39}$$

6.2 Step 2: Compute $357^{-1} \mod 1234$

Using the Extended Euclidean Algorithm:

$$\begin{aligned} 1234 &= 357(3) + 163 \\ 357 &= 163(2) + 31 \\ 163 &= 31(5) + 8 \\ 31 &= 8(3) + 7 \\ 8 &= 7(1) + 1 \\ 7 &= 1(7) + 0 \end{aligned}$$

Back-substituting:

$$1 = 8 - 1(7) \tag{40}$$

$$1 = 8 - 1(31 - 8(3)) = 8(4) - 31(1) \tag{41}$$

$$1 = 8(4) - (163 - 31(5)) = 8(4) - 163 + 31(5) \tag{42}$$

$$1 = 8(4) + 31(5) - 163 \tag{43}$$

$$1 = (163 - 31(5))(4) + 31(5) - 163 = 163(4) - 31(20) + 31(5) \tag{44}$$

$$1 = 163(4) - 31(15) = 163(4) - (357 - 163(2))(15) \tag{45}$$

$$1 = 163(4) - 357(15) + 163(30) = 163(34) - 357(15) \tag{46}$$

$$1 = (1234 - 357(3))(34) - 357(15) \tag{47}$$

$$1 = 1234(34) - 357(102) \tag{48}$$

Thus,

$$357^{-1} \equiv 173 \mod 1234 \tag{49}$$

6.3 Step 3: Compute $3125^{-1} \mod 9987$

Using the algorithm:

$$9987 = 3125(3) + 616$$
$$3125 = 616(5) + 77$$
$$616 = 77(8) + 0$$

Since 77 divides 616 exactly, we back-substitute:

$$1 = 77 - 616(8) \tag{50}$$

$$1 = 77 - (3125 - 616(5))(8) \tag{51}$$

$$1 = 77 - 3125(8) + 616(40) \tag{52}$$

$$1 = (616 - 77(8)) - 3125(8) + 616(40) \tag{53}$$

$$1 = 616(41) - 77(8) - 3125(8) \tag{54}$$

$$1 = 616(41) - (9987 - 3125(3))(8) - 3125(8) \tag{55}$$

$$1 = 616(41) - 9987(8) + 3125(24) - 3125(8) \tag{56}$$

$$1 = 616(41) - 9987(8) + 3125(16) \tag{57}$$

$$1 = 3125(16) - 9987(8) \tag{58}$$

Thus,

$$3125^{-1} \equiv 1571 \mod 9987 \tag{59}$$