Homework 1 Cybersecurity

- Sebastián Navarro 00321588
- Eduardo Guerrero 00326712
- Mateo Pozo 00320780

Exercise 1

The hw-crypto assignment is available in the picoCTF platform. As we discussed in class, you need to write a notebook describing 2 alternative solutions for each problem

Interencdec

Solution 1

```
import base64
cpt =
"YidkM0JxZGtwQlRYdHFhR3g2YUhsZmF6TnFlVGwzWVROclgya3lNRFJvYTJvMmZRPT0nC
g=="
```

The use of UpperCase and lowercase letters and numbers suggest this might be a base 62, however, since this base is not really common, its better to try with base 64 first.

```
plt = base64.b64decode(cpt).decode('utf-8')
print(plt)

b'd3BqdkpBTXtqaGx6aHlfazNqeTl3YTNrX2kyMDRoa2o2fQ=='
```

The message we get seems to still be in base 64, considering we still have uppercase and lowercase letters and numbers. This time we have a full text inside aphostrophes, so we'll ignore the "b" at the beginning.

```
cpt2 = plt.strip()[2:-1]
cpt2
plt2 = base64.b64decode(cpt2).decode('utf-8')
print(plt2)

wpjvJAM{jhlzhy_k3jy9wa3k_i204hkj6}
```

Since the format of the flag has the key inside "{ }", we can guess this is the format of the picoFlag, next, considering the hint said "Engaging in various decoding processes is of utmost importance", we'll try a new decoding process, the ceaser cipher.

```
def caesar_decipher(ciphertext):
    def shift_char(char, shift):
        if char.isalpha():
            # Determine the case
```

```
            start = ord('A') if char.isupper() else ord('a')
            # Apply shift
            return chr((ord(char) - start - shift) % 26 + start)
        else:
            # Keep non-alphabetic characters unchanged
            return char

    # Print all 26 possible shifts
    for shift in range(26):
        plaintext = ''.join(shift_char(char, shift) for char in
ciphertext)
        print(f"Shift {shift}: {plaintext}")

caesar_decipher(plt2)
```

```
Shift 0: wpjvJAM{jhlzhy_k3jy9wa3k_i204hkj6}
Shift 1: voiuIZL{igkygx_j3ix9vz3j_h204gji6}
Shift 2: unhtHYK{hfjxfw_i3hw9uy3i_g204fih6}
Shift 3: tmgsGXJ{geiwev_h3gv9tx3h_f204ehg6}
Shift 4: slfrFWI{fdhvdu_g3fu9sw3g_e204dgf6}
Shift 5: rkeqEVH{ecguct_f3et9rv3f_d204cfe6}
Shift 6: qjdpDUG{dbftbs_e3ds9qu3e_c204bed6}
Shift 7: picoCTF{caesar_d3cr9pt3d_b204adc6}
Shift 8: ohbnBSE{bzdrzq_c3bq9os3c_a204zcb6}
Shift 9: ngamARD{aycqyp_b3ap9nr3b_z204yba6}
Shift 10: mfzlZQC{zxbpxo_a3zo9mq3a_y204xaz6}
Shift 11: leykYPB{ywaown_z3yn9lp3z_x204wzy6}
Shift 12: kdxjXOA{xvznvm_y3xm9ko3y_w204vyx6}
Shift 13: jcwiWNZ{wuymul_x3wl9jn3x_v204uxw6}
Shift 14: ibvhVMY{vtxltk_w3vk9im3w_u204twv6}
Shift 15: haugULX{uswksj_v3uj9hl3v_t204svu6}
Shift 16: gztfTKW{trvjri_u3ti9gk3u_s204rut6}
Shift 17: fyseSJV{squiqh_t3sh9fj3t_r204qts6}
Shift 18: exrdRIU{rpthpg_s3rg9ei3s_q204psr6}
Shift 19: dwqcQHT{qosgof_r3qf9dh3r_p204orq6}
Shift 20: cvpbPGS{pnrfne_q3pe9cg3q_o204nqp6}
Shift 21: buoaOFR{omqemd_p3od9bf3p_n204mpo6}
Shift 22: atnzNEQ{nlpdlc_o3nc9ae3o_m204lon6}
Shift 23: zsmyMDP{mkockb_n3mb9zd3n_l204knm6}
Shift 24: yrlxLCO{ljnbja_m3la9yc3m_k204jml6}
Shift 25: xqkwKBN{kimaiz_l3kz9xb3l_j204ilk6}
```

We find that shift 7 is the right one. picoCTF{caesar_d3cr9pt3d_b204adc6}

Solution 2

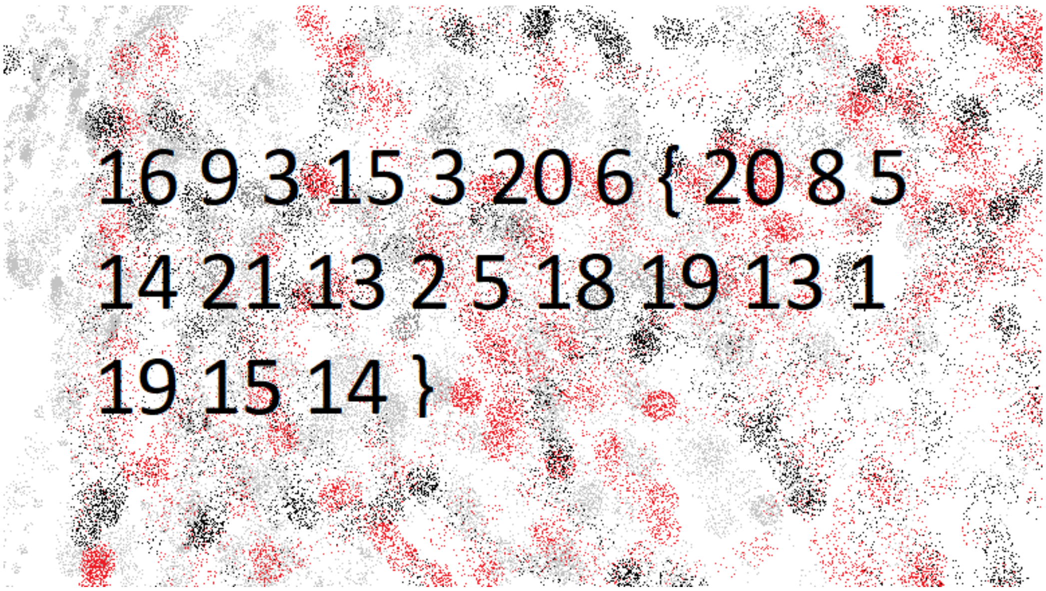
Since the ciphertext is a Base64 encoded string, we can use the terminal to decode it. Applying the decoding twice we get this output:

```
heisenberg_0702@laptopmateo:~$ echo YidkM0JxZGtwQlRYdHFhR3g2YUhsZmF6TnFlVGwzWVROclgyZzBOMm8yYXpZNWZRPT0nCg== | base64 --decode
b'd3BqdkpBTXtqaGx6aHlfazNqeTl3YTNrX2g0N2o2azY5fQ=='
heisenberg_0702@laptopmateo:~$ echo d3BqdkpBTXtqaGx6aHlfazNqeTl3YTNrX2g0N2o2azY5fQ== | base64 --decode
wpjvJAM{jhlzhy_k3jy9wa3k_h47j6k69}heisenberg_0702@laptopmateo:~$ |
```

Again, using a brute force Caesar Cipher decoder, we get the following result:

↑↓	↑↓
→7 (←19)	picoCTF{caesar_d3cr9pt3d_a47c6d69}

The Numbers



Solution 1

We can associate the numbers in the image with the common word **PICOCTF{}**. So:

- P = 16
- I = 9
- C = 3
- O = 15
- C = 3
- T = 20
- F = 6

At this point, we can assume that the numbers represent the positions of the letters of the alphabet from A = 1 to Z = 26. So, we wrote some python code to automatically decode the numbers and find the **PICOCTF{THENUMBERSMASON}** flag:

```python
# Function that takes an array of numbers and decrypts them matching with the alphabet example A=1, B=2, C=3, etc.
def decrypt_numbers(numbers):
```

```
    # Initialize the plain text
    plain_text = ""

    for i, num in enumerate(numbers):
        # Check if the number is within the range of 1-26
        if 1 <= num <= 26:
            # Convert number to corresponding uppercase letter
            plain_text += chr(num + 64)
        else:
            # Add the number as is
            plain_text += str(num)

        # Add opening brace after the 7th character
        if i == 6:
            plain_text += "{"

    # Add closing brace at the end
    plain_text += "}"

    return plain_text

numbers = [16, 9, 3, 15, 3, 20, 6, 20, 8, 5, 14, 21, 13, 2, 5, 18, 19,
13, 1, 19, 15, 14]

print(decrypt_numbers(numbers))
```

```
PICOCTF{THENUMBERSMASON}
```

Solution 2

Following the format of picoCTF flags, it's clear that the numbers represent a direct mapping to the alphabet.

```
# Define a dictionary with the number and letter of the alphabet
alphabet_dict = {i: chr(64 + i) for i in range(1, 27)}

# Print the dictionary
print(alphabet_dict)
```

```
{1: 'A', 2: 'B', 3: 'C', 4: 'D', 5: 'E', 6: 'F', 7: 'G', 8: 'H', 9:
'I', 10: 'J', 11: 'K', 12: 'L', 13: 'M', 14: 'N', 15: 'O', 16: 'P',
17: 'Q', 18: 'R', 19: 'S', 20: 'T', 21: 'U', 22: 'V', 23: 'W', 24:
'X', 25: 'Y', 26: 'Z'}
```

```
# Define a list with the numbers
numbers_list = [16, 9, 3, 15, 3, 20, 6, '{', 20, 8, 5, 14, 21, 13, 2,
5, 18, 19, 13, 1, 19, 15, 14, '}']

# Define the alphabet dictionary
alphabet_dict = {i: chr(64 + i) for i in range(1, 27)}
```

```python
# Replace the numbers with the characters
flag = ''.join([alphabet_dict[number] if isinstance(number, int) else
number for number in numbers_list])

# Print the flag
print(flag)
```

```
PICOCTF{THENUMBERSMASON}
```

C3

The code provided for the exercise is as follows:

```python
import sys
chars = ""
from fileinput import input
for line in input():
  chars += line

lookup1 = "\n \"#()*+/1:=[]abcdefghijklmnopqrstuvwxyz"
lookup2 = "ABCDEFGHIJKLMNOPQRSTabcdefghijklmnopqrst"

out = ""

prev = 0
for char in chars:
  cur = lookup1.index(char)
  out += lookup2[(cur - prev) % 40]
  prev = cur

sys.stdout.write(out)
```

We are also given the ciphertext:
`DLSeGAGDgBNJDQJDCFSFnRBIDjgHoDFCFtHDgJpiHtGDmMAQFnRBJKkBAsTMrsPSDDnEFCFtIbEDtDCIbFCFtHTJDKerFldbFObFCFtLBFkBAAAPFnRBJGEkerFlcPgKkImHnIlATJDKbTbFOkdNnsgbnJRMFnRBNAFkBAAAbrcbTKAkOgFpOgFpOpkBAAAAAAAiClFGIPFnRBaKliCgClFGtIBAAAAAAAOgGEkImHnIl`

Solution 1

Therefore, by analyzing the code provided, it is observed that the encryption is based on a system of shifts between the indices of the current character and the previous one. The current index is obtained by locating the encrypted character in `lookup1`, and then concatenating to the output the corresponding character from `lookup2`, whose index is calculated by subtracting the current index from the previous one and applying a cyclic adjustment.

Now, to decrypt the ciphertext, the reverse process must be performed, as follows:

```
def decryptC3(ciphertext):
    prev = 0
    for char in ciphertext:
        ciphered_index = lookup2.index(char)

        for i in range(len(lookup1)):
            if (i - prev) % 40 == ciphered_index:
                print(lookup1[i], end = "")
                prev = i
                break
```

The function decrypts the ciphertext by processing it character by character. For each encrypted character, its index in `lookup2` is obtained, which represents the encrypted index. Next, the index of the original character is found in `lookup1` such that, by subtracting the previous index and applying a modulus of 40, it matches the encrypted index. Finally, the characters corresponding to the indices found in `lookup1` are printed. The use of modulus 40 is due to the fact that both `lookup1` and `lookup2` contain 40 characters, which ensures that the indices stay within the bounds of these strings and allows for cyclic decryption behavior.

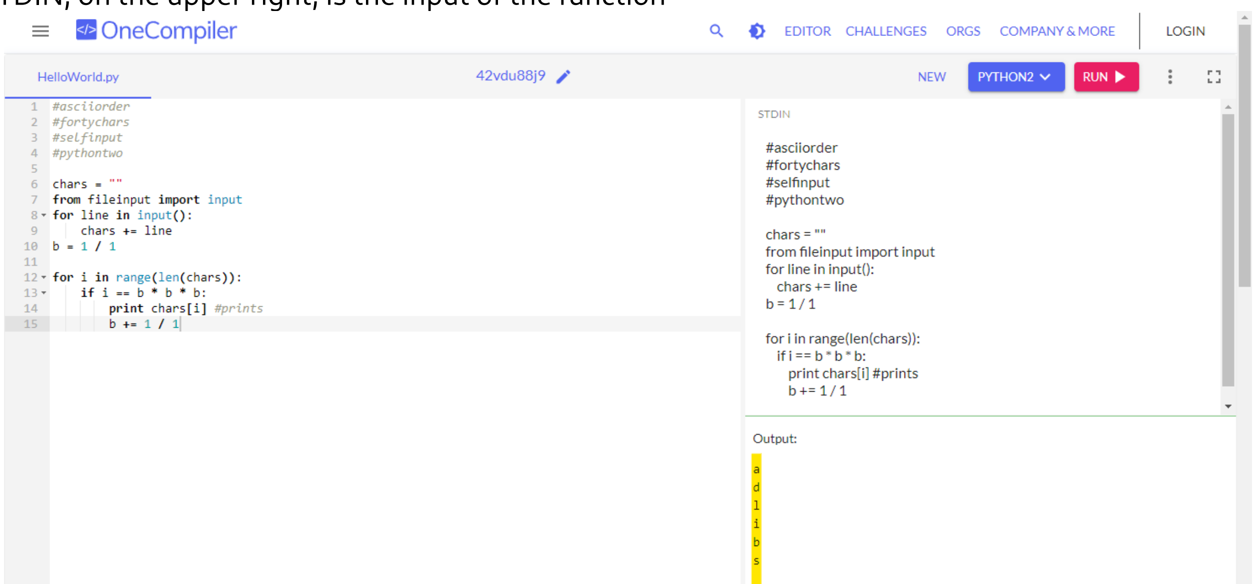
Now, applying the decryption function to the provided ciphertext:

```
ciphertext =
"DLSeGAGDgBNJDQJDCFSFnRBIDjgHoDFCFtHDgJpiHtGDmMAQFnRBJKkBAsTMrsPSDDnEF
CFtIbEDtDCIbFCFtHTJDKerFldbFObFCFtLBFkBAAAPFnRBJGEkerFlcPgKkImHnIlATJD
KbTbFOkdNnsgbnJRMFnRBNAFkBAAAbrcbTKAkOgFpOgFpOpkBAAAAAAAiClFGIPFnRBaKl
iCgClFGtIBAAAAAAAOgGEkImHnIl"
lookup1 = "\n \"#()*+/1:=[]abcdefghijklmnopqrstuvwxyz"
lookup2 = "ABCDEFGHIJKLMNOPQRSTabcdefghijklmnopqrst"


def decryptC3(ciphertext):
    prev = 0
    for char in ciphertext:
        cur = lookup2.index(char)

        for i in range(len(lookup1)):
            if (i - prev) % 40 == cur:
                print(lookup1[i], end = "")
                prev = i
                break


decryptC3(ciphertext)

#asciiorder
#fortychars
#selfinput
#pythontwo
```

```
chars = ""
from fileinput import input
for line in input():
    chars += line
b = 1 / 1

for i in range(len(chars)):
    if i == b * b * b:
        print chars[i] #prints
        b += 1 / 1
```

We can run the code obtained in python2, instead of trying to redo the script in python 3. This will help, since converting it to pyhton3 may create mistakes and problems with the decryption. For this, we use the website at https://onecompiler.com/python2

STDIN, on the upper right, is the input of the function



Finally, the flag is **picoCTF{adlibs}**.

Solution 2

```
def decrypt_cyclical(ciphertext):
  lookup1 = "\n \"#()*+/1:=[]abcdefghijklmnopqrstuvwxyz"
  lookup2 = "ABCDEFGHIJKLMNOPQRSTabcdefghijklmnopqrst"

  out = ""

  # We reverse the encryption process in the loop
  prev = 0
  for char in ciphertext:
    cur = lookup2.index(char)
    out += lookup1[(cur + prev) % 40]
    prev = cur + prev % 40
```

```
  return out
```

Alternatively, we can modify the decrypted python code to be a python 3 program and get the flag.

```
#asciiorder
#fortychars
#selfinput
#pythontwo

chars = ""
from fileinput import input

input =
decrypt_cyclical("DLSeGAGDgBNJDQJDCFSFnRBIDjgHoDFCFtHDgJpiHtGDmMAQFnRB
JKkBAsTMrsPSDDnEFCFtIbEDtDCIbFCFtHTJDKerFldbFObFCFtLBFkBAAAPFnRBJGEker
FlcPgKkImHnIlATJDKbTbFOkdNnsgbnJRMFnRBNAFkBAAAbrcbTKAkOgFpOgFpOpkBAAAA
AAAiClFGIPFnRBaKliCgClFGtIBAAAAAAAOgGEkImHnIl")

for line in input:
    chars += line
b = 1 / 1

for i in range(len(chars)):
    if i == b * b * b:
        print(chars[i]) #prints
        b += 1 / 1

a
d
l
i
b
s
```

Custom encyption

Solution 1

We are given the following information

a = 97

b = 22

cipher is: [151146, 1158786, 1276344, 1360314, 1427490, 1377108, 1074816, 1074816, 386262, 705348, 0, 1393902, 352674, 83970, 1141992, 0, 369468, 1444284, 16794, 1041228, 403056, 453438, 100764, 100764, 285498, 100764, 436644, 856494, 537408, 822906, 436644, 117558, 201528, 285498]

Along with the encryption code:

```
from random import randint
import sys


def generator(g, x, p):
    return pow(g, x) % p


def encrypt(plaintext, key):
    cipher = []
    for char in plaintext:
        cipher.append(((ord(char) * key*311)))
    return cipher


def is_prime(p):
    v = 0
    for i in range(2, p + 1):
        if p % i == 0:
            v = v + 1
    if v > 1:
        return False
    else:
        return True


def dynamic_xor_encrypt(plaintext, text_key):
    cipher_text = ""
    key_length = len(text_key)
    for i, char in enumerate(plaintext[::-1]):
        key_char = text_key[i % key_length]
        encrypted_char = chr(ord(char) ^ ord(key_char))
        cipher_text += encrypted_char
    return cipher_text


def test(plain_text, text_key):
    p = 97
    g = 31
    if not is_prime(p) and not is_prime(g):
        print("Enter prime numbers")
        return
    a = randint(p-10, p)
    b = randint(g-10, g)
    print(f"a = {a}")
    print(f"b = {b}")
    u = generator(g, a, p)
    v = generator(g, b, p)
```

```python
    key = generator(v, a, p)
    b_key = generator(u, b, p)
    shared_key = None
    if key == b_key:
        shared_key = key
    else:
        print("Invalid key")
        return
    semi_cipher = dynamic_xor_encrypt(plain_text, text_key)
    cipher = encrypt(semi_cipher, shared_key)
    print(f'cipher is: {cipher}')

if __name__ == "__main__":
    message = sys.argv[1]
    test(message, "trudeau")
```

```
a = 94
b = 23
cipher is: [711568, 793672, 164208, 383152, 2052600, 2271544, 602096,
1915760, 1778920, 465256, 2517856, 2517856, 2408384, 437888, 1943128,
1751552, 1861024, 2381016, 2353648, 2435752, 1751552, 1943128,
2052600, 1888392, 164208, 2545224, 136840, 547360, 1833656, 1751552,
465256, 164208, 2189440, 2244176, 1751552, 1833656, 1806288, 1751552,
164208, 2271544, 54736, 602096, 1806288, 547360, 1888392, 2381016,
519992, 2079968, 684200, 465256, 766304, 191576, 27368, 383152,
1669448, 437888, 684200, 738936, 27368, 273680, 437888, 519992,
1122088, 164208, 629464, 27368, 793672, 574728, 547360, 848408,
1094720, 574728, 738936, 355784, 218944, 0, 711568, 1039984, 1258928,
547360, 437888, 109472, 1012616, 136840, 109472, 1395768, 1122088,
1176824, 1039984, 1094720, 1614712, 1094720, 27368, 191576, 27368,
602096, 1423136, 1122088, 2134704, 465256, 1970496, 54736, 1970496,
2079968]
```

From this we get the following information.

```python
a = 97
b = 22
p = 97
g = 31
text_key = "trudeau"
```

The plain text first goes through the dynamic_xor_encrypt function and then the encrypt function.

We also know that the share key happens when key = b_key

To reverse this process, first we reverse the functions of encryption

```python
def decrypt(ciphertext, key):
    plaintext = []
    for num in ciphertext:
        # Reverse the multiplication by dividing by (key * 311)
        orig_char = chr(num // (key * 311))
        plaintext.append(orig_char)
    return ''.join(plaintext)


def dynamic_xor_decrypt(cipher_text, text_key):
    plaintext = ""
    key_length = len(text_key)
    for i, char in enumerate(cipher_text):  # Reverse the cipher_text
        key_char = text_key[i % key_length]
        decrypted_char = chr(ord(char) ^ ord(key_char))
        plaintext += decrypted_char
    return plaintext
```

The rest of the functions remain the same

Now we add the information we previously find

```python
a = 97
b = 22
p = 97
g = 31
text_key = "trudeau"
u = generator(g, a, p)
v = generator(g, b, p)
key = generator(v, a, p)
b_key = generator(u, b, p)
shared_key = None
if key == b_key:
    shared_key = key
else:
    print("Invalid key")

cipher = [151146, 1158786, 1276344, 1360314, 1427490, 1377108,
          1074816, 1074816, 386262, 705348, 0, 1393902, 352674,
          83970, 1141992, 0, 369468, 1444284, 16794, 1041228,
          403056, 453438, 100764, 100764, 285498, 100764,
          436644, 856494, 537408, 822906, 436644, 117558,
          201528, 285498]
```

We know reverse the order of the following part of the code

```python
semi_cipher = dynamic_xor_encrypt(plain_text, text_key)
    cipher = encrypt(semi_cipher, shared_key)
    print(f'cipher is: {cipher}')
```

```
semi_plain = decrypt(cipher, shared_key)
plain_text = dynamic_xor_decrypt(semi_plain, text_key)
print(plain_text)
```

```
}7950354e_d6tp0rc2d_motsuc{FTCocip
```

We get a reversed string so we need to reverse it to get the hidden message.

```
plt = plain_text[::-1]
print(plt)
```

```
picoCTF{custom_d2cr0pt6d_e4530597}
```

Solution 2

Analyzing this part of the code

```
def dynamic_xor_encrypt(plaintext, text_key):
    cipher_text = ""
    key_length = len(text_key)
    for i, char in enumerate(plaintext[::-1]):
        key_char = text_key[i % key_length]
        encrypted_char = chr(ord(char) ^ ord(key_char))
        cipher_text += encrypted_char
    return cipher_text

def dynamic_xor_decrypt(cipher_text, text_key):
    plaintext = ""
    key_length = len(text_key)
    for i, char in enumerate(cipher_text):  # Reverse the cipher_text
        key_char = text_key[i % key_length]
        decrypted_char = chr(ord(char) ^ ord(key_char))
        plaintext += decrypted_char
    return plaintext
```

We know we can get the ciphertext by doing an XOR of the plaintext and the text_key and we can ge the plaintext by doing an XOR of the ciphertext and the text_key (This works because the inverse of XOR is XOR itself).

Meaning, we could get the text_key given the ciphertext and the plaintext.

We know part of the plaintext will be "picoCTF", se we can obtain the text_key with it.

Also, since we know a, b, p and g where used for the creation of the shared_key for the plaintext, we dont need to use this verificacion anymore

```
u = generator(g, a, p)
v = generator(g, b, p)
key = generator(v, a, p)
b_key = generator(u, b, p)
```

```python
shared_key = None
if key == b_key:
    shared_key = key
else:
    print("Invalid key")
```

And can just define shared_key as

```python
u = generator(g, a, p)
b_key = generator(u, b, p)
shared_key = b_key
```

or

```python
shared_key = generator(generator(g, a, p), b, p)

a = 97
b = 22
p = 97
g = 31
text_key = "picoCTF"
shared_key = generator(generator(g, a, p), b, p)

# cipher = [856494, 537408, 822906, 436644, 117558,
#             201528, 285498]

cipher = [285498, 201528, 117558, 436644, 822906, 537408, 856494]
```

Since we had to reverse the plaintext to get the actual plaintext, we know the cipher needs to be in reverse to work, so we only use the last 7 blocks of the cipher and we reverse the order of them.

```python
def decrypt(ciphertext, key):
    plaintext = []
    for num in ciphertext:
        # Reverse the multiplication by dividing by (key * 311)
        orig_char = chr(num // (key * 311))
        plaintext.append(orig_char)
    return ''.join(plaintext)

def dynamic_xor_decrypt(cipher_text, text_key):
    plaintext = ""
    key_length = len(text_key)
    for i, char in enumerate(cipher_text):  # Reverse the cipher_text
        key_char = text_key[i % key_length]
        decrypted_char = chr(ord(char) ^ ord(key_char))
        plaintext += decrypted_char
    return plaintext
```

```
semi_plain = decrypt(cipher, shared_key)
plain_text = dynamic_xor_decrypt(semi_plain, text_key)
print(plain_text)
```

```
aedurtu
```

Now we try this key without reversing the order of the plaintext at the end

```
a = 97
b = 22
p = 97
g = 31
text_key = "aedurtu"
shared_key = generator(generator(g, a, p), b, p)

cipher = [151146, 1158786, 1276344, 1360314, 1427490, 1377108,
          1074816, 1074816, 386262, 705348, 0, 1393902, 352674,
          83970, 1141992, 0, 369468, 1444284, 16794, 1041228,
          403056, 453438, 100764, 100764, 285498, 100764,
          436644, 856494, 537408, 822906, 436644, 117558,
          201528, 285498]

def decrypt(cipher, key):
    plaintext = ""
    for number in cipher:
        plaintext += chr(number // key // 311)
    return plaintext

def dynamic_xor_decrypt(cipher, text_key):
    decrypted_text = ""
    key_length = len(text_key)
    for i, char in enumerate(cipher[::-1]):
        key_char = text_key[i % key_length]
        decrypted_char = chr(ord(char) ^ ord(key_char))
        decrypted_text += decrypted_char
    return decrypted_text

semi_plain = decrypt(cipher, shared_key)
plain_text = dynamic_xor_decrypt(semi_plain, text_key)
print(plain_text)
```

```
picoCTF{custom_d2cr0pt6d_e4530597}
```

Rotation

Solution 1

```
cpt = "xqkwKBN{z0bib1wv_l3kzgxb3l_555957n3}"
```

Knowing the start of the plaintext is picoCTF, and guessing the cipher text was just encryoted by a rotation cipher (considering the start has the same amount of letters as "picoCTF" before having "{", followed by text and ending with "}", which follows the picoCTF flag format), we can map the letter x to p

```
shift = ord('x') - ord('p')
shift

8
```

Given 8 as the shift, we can create the following function

```
def rot_decrypt_8(ciphertext):
    decrypted = ""
    for char in ciphertext:
        if char.isalpha():
            shift_amount = 65 if char.isupper() else 97
            decrypted += chr((ord(char) - shift_amount - 8) % 26 +
shift_amount)
        else:
            decrypted += char  # Non-alphabetic characters are
unchanged
    return decrypted

plt = rot_decrypt_8(cpt)
print(plt)

picoCTF{r0tat1on_d3crypt3d_555957f3}
```

Solution 2

From the name of the challenge and the clue, we can assume that this is a type of rotation cipher, a cipher where each letter of the alphabet is shifted **n** positions.

We used the website https://www.dcode.fr/rot-cipher to decrypt the rotationally the encrypted text `xqkwKBN{z0bib1wv_l3kzgxb3l_i4j7l759}`, and using the **ROT Cipher Decoder** tool we found the flag **picoCTF{r0tat1on_d3crypt3d_a4b7d759}**

As you can see in the image, the plaintext was shifted **8 positions forward**.

Exercise 2

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

Perfect secrecy means that knowing a ciphertext c gives no information about the plaintext p

So, the probability distribution of c is independent of p

* For definition, two events are independent if:

$$Pr(A \mid B) = Pr(A)$$

In this case, where $p \in P$, $c \in C$ and $k \in K$, then:

$$① \quad Pr(P=p \mid C=c) = Pr(P=p)$$

Applying Baye's theorem:

$$② \quad Pr(P=p \mid C=c) = \frac{Pr(C=c \mid P=p) \cdot Pr(P=p)}{Pr(C=c)}$$

$$① = ②$$

$$Pr(P=p) = \frac{Pr(C=c \mid P=p) \cdot Pr(P=p)}{Pr(C=c)}$$

Multiplying $\frac{1}{Pr(P=p)}$ at the two sides:

$$\frac{1}{\cancel{Pr(P=p)}} \cdot \cancel{Pr(P=p)} = \frac{Pr(C=c \mid P=p) \cdot \cancel{Pr(P=p)}}{Pr(C=c)} \cdot \frac{1}{\cancel{Pr(P=p)}}$$

$$1 = \frac{Pr(C=c \mid P=p)}{Pr(C=c)}$$

$$Pr(C=c \mid P=p) = Pr(C=c)$$

This expression proves that for every plaintext p, the probability of producing a given ciphertext c is the same

So every ciphertext is equally probable for all plaintexts

Since that $|K| = |C| = |P|$, each key is used to map a given plaintext to a unique ciphertext so that:

If $p \in P$, $c \in C$ and $k \in K$, then:

$$Encrypt_{k_{p,c}}(p) = c$$

Then, each key in $|K|$ must be used with equal probability:

$$Pr(K) = \frac{1}{|K|}$$

Exercise 3

Suppose that APNDJI or XYGROBO are ciphertexts that are obtained from encryption using the Shift Cipher. Show in each case that there are two "meaningful" plaintexts that could encrypt to the given ciphertext.

```python
# Shift Cipher
def shift_cipher_forward(text, shift):
    result = ""
    for i in range(len(text)):
        char = text[i]
        if char.isupper():
            result += chr((ord(char) + shift - 65) % 26 + 65)
        else:
            result += chr((ord(char) + shift - 97) % 26 + 97)
    return result

def shift_cipher_backward(text, shift):
    result = ""
    for i in range(len(text)):
        char = text[i]
        if char.isupper():
            result += chr((ord(char) - shift - 65) % 26 + 65)
        else:
            result += chr((ord(char) - shift - 97) % 26 + 97)
    return result

def brute_force(text):
    print("Forward:")
    for i in range(1, 26):
        print(f"Shift {i}: {shift_cipher_forward(text, i)}")

cypher_text = "APNDJI"
brute_force(cypher_text)
```

```
Forward:
Shift 1: BQOEKJ
Shift 2: CRPFLK
Shift 3: DSQGML
Shift 4: ETRHNM
Shift 5: FUSION
Shift 6: GVTJPO
Shift 7: HWUKQP
Shift 8: IXVLRQ
Shift 9: JYWMSR
Shift 10: KZXNTS
Shift 11: LAYOUT
Shift 12: MBZPVU
Shift 13: NCAQWV
Shift 14: ODBRXW
```

```
Shift 15: PECSYX
Shift 16: QFDTZY
Shift 17: RGEUAZ
Shift 18: SHFVBA
Shift 19: TIGWCB
Shift 20: UJHXDC
Shift 21: VKIYED
Shift 22: WLJZFE
Shift 23: XMKAGF
Shift 24: YNLBHG
Shift 25: ZOMCIH
```

The two "meaningful" plaintexts that can be identified in the case of the ciphertext APNDJI are:

- Shift 5: FUSION
- Shift 11: LAYOUT

```
cypher_text = "XYGROBO"
brute_force(cypher_text)
```

```
Forward:
Shift 1: YZHSPCP
Shift 2: ZAITQDQ
Shift 3: ABJURER
Shift 4: BCKVSFS
Shift 5: CDLWTGT
Shift 6: DEMXUHU
Shift 7: EFNYVIV
Shift 8: FGOZWJW
Shift 9: GHPAXKX
Shift 10: HIQBYLY
Shift 11: IJRCZMZ
Shift 12: JKSDANA
Shift 13: KLTEBOB
Shift 14: LMUFCPC
Shift 15: MNVGDQD
Shift 16: NOWHERE
Shift 17: OPXIFSF
Shift 18: PQYJGTG
Shift 19: QRZKHUH
Shift 20: RSALIVI
Shift 21: STBMJWJ
Shift 22: TUCNKXK
Shift 23: UVDOLYL
Shift 24: VWEPMZM
Shift 25: WXFQNAN
```

The two "meaningful" plaintexts that can be identified in the case of the ciphertext XYGROBO are:

- Shift 3: ADJURER
- Shift 16: NOWHERE

Exercise 4

Compute H(K|C) and H(K|P, C) for the Affine Cipher, assuming that keys are used equiprobably and the plaintexts are equiprobable.

$H(K|C) = H(K) + H(P) - H(C)$

26 letters

Affine cipher takes α and B

α must be coprime of 26

$\alpha = \{1, 3, 5, 7, 9, 11, 15, 17, 19, 21, 23, 25\}$

* Possible values of $\alpha = 12$

  B can be $0 \leq B < 26$

* Possible values of $B = 26$

* Possible values of $k = 26 \cdot 12 = 312$

Since that $H(x) = -\sum_{x \in X} Pr[x] \log_2 Pr[x]$

and keys are used equiprobably

$$H(K) = -\sum_{K \in K} Pr(K) \log_2 Pr(K)$$

$$= -\sum_{k=1}^{312} \frac{1}{312} \log_2 \left(\frac{1}{312}\right)$$

$$= -\cancel{312} \left(\frac{1}{\cancel{312}} \cdot \log_2 \left(\frac{1}{312}\right)\right)$$

$$= -(-8{,}29)$$

$$= 8{,}29$$

P is 26 since there are 26 letters in the alphabet

$$H(P) = -\sum_{p \in P} Pr(p) \log_2 Pr(p)$$

$$= -\sum_{p=1}^{26} \frac{1}{26} \log_2 \left(\frac{1}{26}\right)$$

$$= -\cancel{26}\left(\frac{1}{\cancel{26}} \cdot \log_2\left(\frac{1}{26}\right)\right)$$

$$= -(-4{,}70)$$

$$= 4{,}70$$

C is 26 since there are 26 letters in the alphabet

$$H(C) = -\sum_{x \in X} Pr[x] \log_2 Pr[x]$$

$$= -\sum_{x=1}^{26} \frac{1}{26} \log_2 \left(\frac{1}{26}\right)$$

$$= -26\left(\frac{1}{26} \cdot \log_2\left(\frac{1}{26}\right)\right)$$

$$= -(-4{,}70)$$

$$= 4{,}70$$

$$H(K|C) = H(K) + H(P) - H(C)$$

$$= 8{,}29 + 4{,}70 - 4{,}70 = 8{,}29$$

Since $H(K|C) = H(K,C) - H(C)$

$\therefore \quad H(K|P,C) = H(K,P,C) - H(P,C)$

From class and theorem 3.10 from the book:

$H(K,P,C) = H(K,P)$ and $H(K,P) = H(K) + H(P)$

Since they are independent

Theorem 3.8 $H(x,y) \leq H(y) + H(x)$

$H(P,C) \leq H(P) + H(C)$

$$H(K|P,C) = H(K) + H(P) - (H(P) + H(C))$$

$$= H(K) + \cancel{H(P)} - \cancel{H(P)} - H(C)$$

$$= H(K) - H(C)$$

$$= 8{,}29 - 4{,}70$$

$$= 3{,}59 \quad //$$

Exercise 5.1

Substitution Cipher:

EMGLOSUDCGDNCUSWYSFHNSFCYKDPUMLWGYICOXYSIPJCKQPKUGKMGOLICGINCGACKS
NISACYKZSCKXECJCKSHYSXCGOIDPKZCNKSHICGIWYGKKGKGOLDSILKGOIUSIGLEDSPWZUG
FZCCNDGYYSFUSZCNXEOJNCGYEOWEUPXEZGACGNFGLKNSACIGOIYCKXCJUCIUZCFZCCN
DGYYSFEUEKUZCSOCFZCCNCIACZEJNCSHFZEJZEGMXCYHCJUMGKUCY

Function for frequency analysis and Fitness score implementing letter, bigram and trigram frequency and To replace letters

Note: In this function, the lower the score, the better

```
import matplotlib.pyplot as plt
from collections import Counter

def frequency_analysis(cipher_text):
    # Remove non-alphabetic characters and convert to lowercase
    cipher_text = ''.join(filter(str.isalpha, cipher_text)).lower()

    # Count the frequency of each letter
    frequency = Counter(cipher_text)

    # Sort letters by frequency
    sorted_frequency = dict(sorted(frequency.items(), key=lambda item:
item[1], reverse=True))

    # Create a bar graph
    plt.figure(figsize=(10, 6))
    plt.bar(sorted_frequency.keys(), sorted_frequency.values(),
color='blue')
    plt.xlabel('Letters')
    plt.ylabel('Frequency')
    plt.title('Frequency Analysis of Cipher Text')
    plt.xticks(rotation=45)
    plt.grid(axis='y')
    plt.tight_layout()
    plt.show()

def bigram_frequency_analysis(cipher_text):
    # Clean the text: remove non-alphabetic characters and convert to
uppercase
    clean_text = ''.join(filter(str.isalpha, cipher_text)).upper()

    # Create bigrams
    bigrams = [clean_text[i:i+2] for i in range(len(clean_text) - 1)]

    # Count frequency of each bigram
    frequency = Counter(bigrams)
```

```
    # Get the 5 most common bigrams
    most_common_bigrams = frequency.most_common(5)

    # Prepare data for plotting
    labels, counts = zip(*most_common_bigrams)

    # Create a bar graph
    plt.figure(figsize=(10, 5))
    plt.bar(labels, counts, color='green')
    plt.title('Top 5 Most Common Bigrams')
    plt.xlabel('Bigrams')
    plt.ylabel('Frequency')
    plt.xticks(rotation=45)
    plt.tight_layout()  # Adjust layout to prevent clipping of tick-
labels
    plt.show()

def trigram_frequency_analysis(cipher_text):
    # Clean the text: remove non-alphabetic characters and convert to
uppercase
    clean_text = ''.join(filter(str.isalpha, cipher_text)).upper()

    # Create trigrams
    trigrams = [clean_text[i:i+3] for i in range(len(clean_text) - 2)]

    # Count frequency of each trigram
    frequency = Counter(trigrams)

    # Get the 5 most common trigrams
    most_common_trigrams = frequency.most_common(10)

    # Prepare data for plotting
    labels, counts = zip(*most_common_trigrams)

    # Create a bar graph
    plt.figure(figsize=(10, 5))
    plt.bar(labels, counts, color='blue')
    plt.title('Top 5 Most Common Trigrams')
    plt.xlabel('Trigrams')
    plt.ylabel('Frequency')
    plt.xticks(rotation=45)
    plt.tight_layout()  # Adjust layout to prevent clipping of tick-
labels
    plt.show()

def seven_frequency_analysis(cipher_text):
    # Clean the text: remove non-alphabetic characters and convert to
uppercase
    clean_text = ''.join(filter(str.isalpha, cipher_text)).upper()
```

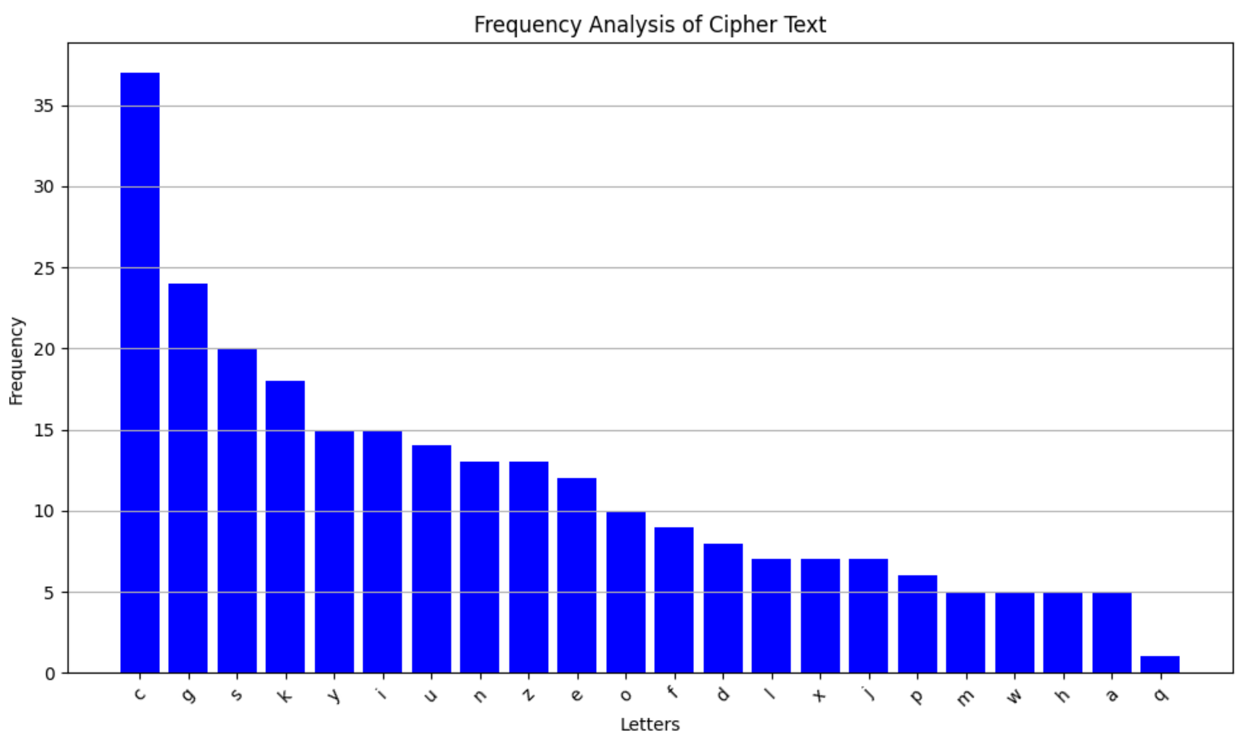
```
    # Create trigrams
    trigrams = [clean_text[i:i+7] for i in range(len(clean_text) - 6)]

    # Count frequency of each trigram
    frequency = Counter(trigrams)

    # Get the 5 most common trigrams
    most_common_trigrams = frequency.most_common(5)

    # Prepare data for plotting
    labels, counts = zip(*most_common_trigrams)

    # Create a bar graph
    plt.figure(figsize=(10, 5))
    plt.bar(labels, counts, color='blue')
    plt.title('Top 5 Most Common Trigrams')
    plt.xlabel('Trigrams')
    plt.ylabel('Frequency')
    plt.xticks(rotation=45)
    plt.tight_layout()  # Adjust layout to prevent clipping of tick-
labels
    plt.show()

def eleven_frequency_analysis(cipher_text):
    # Clean the text: remove non-alphabetic characters and convert to
uppercase
    clean_text = ''.join(filter(str.isalpha, cipher_text)).upper()

    # Create trigrams
    trigrams = [clean_text[i:i+11] for i in range(len(clean_text) -
10)]

    # Count frequency of each trigram
    frequency = Counter(trigrams)

    # Get the 5 most common trigrams
    most_common_trigrams = frequency.most_common(5)

    # Prepare data for plotting
    labels, counts = zip(*most_common_trigrams)

    # Create a bar graph
    plt.figure(figsize=(10, 5))
    plt.bar(labels, counts, color='blue')
    plt.title('Top 5 Most Common Trigrams')
    plt.xlabel('Trigrams')
    plt.ylabel('Frequency')
    plt.xticks(rotation=45)
    plt.tight_layout()  # Adjust layout to prevent clipping of tick-
```

```
labels
    plt.show()

from collections import Counter
import math

# Reference English letter frequencies (as percentages)
english_letter_freq = {
    'E': 12.70, 'T': 9.06, 'A': 8.17, 'O': 7.51, 'I': 6.97, 'N': 6.75,
'S': 6.33,
    'H': 6.09, 'R': 5.99, 'D': 4.25, 'L': 4.03, 'C': 2.78, 'U': 2.76,
'M': 2.41,
    'W': 2.36, 'F': 2.23, 'G': 2.02, 'Y': 1.97, 'P': 1.93, 'B': 1.49,
'V': 0.98,
    'K': 0.77, 'J': 0.15, 'X': 0.15, 'Q': 0.10, 'Z': 0.07
}

# Reference English bigram frequencies (as percentages)
english_bigram_freq = {
    'TH': 1.52, 'HE': 1.28, 'IN': 0.94, 'ER': 0.94, 'AN': 0.82, 'RE':
0.68,
    'ND': 0.63, 'AT': 0.59, 'ON': 0.57, 'NT': 0.56
    # Add more bigrams as needed
}

# Reference English trigram frequencies (as percentages)
english_trigram_freq = {
    'THE': 1.81, 'AND': 0.73, 'ING': 0.72, 'ENT': 0.42, 'ION': 0.42,
    'HER': 0.36, 'FOR': 0.34, 'THA': 0.33, 'NTH': 0.33, 'INT': 0.32
    # Add more trigrams as needed
}

# Function to calculate letter frequency in text
def calculate_letter_frequency(text):
    # Remove non-alphabetic characters and convert to uppercase
    clean_text = ''.join(filter(str.isalpha, text)).upper()
    letter_counts = Counter(clean_text)
    total_letters = len(clean_text)

    # Calculate percentage frequency for each letter in the text
    letter_freq = {letter: (count / total_letters) * 100 for letter,
count in letter_counts.items()}

    return letter_freq

# Function to calculate bigram frequency in text
def calculate_bigram_frequency(text):
    # Remove non-alphabetic characters and convert to uppercase
    clean_text = ''.join(filter(str.isalpha, text)).upper()
    bigrams = [clean_text[i:i+2] for i in range(len(clean_text) - 1)]
```

```
    bigram_counts = Counter(bigrams)
    total_bigrams = len(bigrams)

    # Calculate percentage frequency for each bigram in the text
    bigram_freq = {bigram: (count / total_bigrams) * 100 for bigram,
count in bigram_counts.items()}

    return bigram_freq

# Function to calculate trigram frequency in text
def calculate_trigram_frequency(text):
    # Remove non-alphabetic characters and convert to uppercase
    clean_text = ''.join(filter(str.isalpha, text)).upper()
    trigrams = [clean_text[i:i+3] for i in range(len(clean_text) - 2)]
    trigram_counts = Counter(trigrams)
    total_trigrams = len(trigrams)

    # Calculate percentage frequency for each trigram in the text
    trigram_freq = {trigram: (count / total_trigrams) * 100 for
trigram, count in trigram_counts.items()}

    return trigram_freq

# Function to calculate fitness score
def fitness_score(text):
    # Calculate letter, bigram, and trigram frequencies for the input
text
    letter_freq = calculate_letter_frequency(text)
    bigram_freq = calculate_bigram_frequency(text)
    trigram_freq = calculate_trigram_frequency(text)

    # Compare with reference letter frequencies (lower difference is
better)
    letter_score = 0
    for letter, ref_freq in english_letter_freq.items():
        observed_freq = letter_freq.get(letter, 0)  # Get observed
frequency or 0 if letter is missing
        letter_score += (observed_freq - ref_freq) ** 2  # Squared
difference

    # Compare with reference bigram frequencies
    bigram_score = 0
    for bigram, ref_freq in english_bigram_freq.items():
        observed_freq = bigram_freq.get(bigram, 0)  # Get observed
frequency or 0 if bigram is missing
        bigram_score += (observed_freq - ref_freq) ** 2  # Squared
difference

    # Compare with reference trigram frequencies
```

```
    trigram_score = 0
    for trigram, ref_freq in english_trigram_freq.items():
        observed_freq = trigram_freq.get(trigram, 0)  # Get observed
frequency or 0 if trigram is missing
        trigram_score += (observed_freq - ref_freq) ** 2  # Squared
difference

    # Combine the scores (you can adjust the weights as needed)
    total_score = math.sqrt(letter_score + bigram_score +
trigram_score)

    return total_score

def replace_letters(input_text, mapping):
    # Create a translation table for the replacements
    translation_table = str.maketrans(mapping)

    # Translate the input text using the translation table
    modified_text = input_text.translate(translation_table)

    return modified_text

def generate_substitution_key(plaintext, ciphertext):
    # Ensure both the plaintext and ciphertext are of the same length
    if len(plaintext) != len(ciphertext):
        raise ValueError("Plaintext and ciphertext must be of the same
length.")

    # Remove any non-alphabetic characters and convert to uppercase
    clean_plaintext = ''.join(filter(str.isalpha, plaintext)).upper()
    clean_ciphertext = ''.join(filter(str.isalpha,
ciphertext)).upper()

    # Create dictionaries to store mappings between the original
alphabet and the cipher
    substitution_key = {}

    # Loop through both plaintext and ciphertext to generate the
substitution key
    for p_char, c_char in zip(clean_plaintext, clean_ciphertext):
        if p_char not in substitution_key:
            substitution_key[p_char] = c_char

    # Sort the key alphabetically by the original alphabet
    sorted_key = dict(sorted(substitution_key.items()))

    # Generate the original alphabet
    original_alphabet = ''.join(sorted(sorted_key.keys()))

    # Generate the corresponding cipher alphabet
```

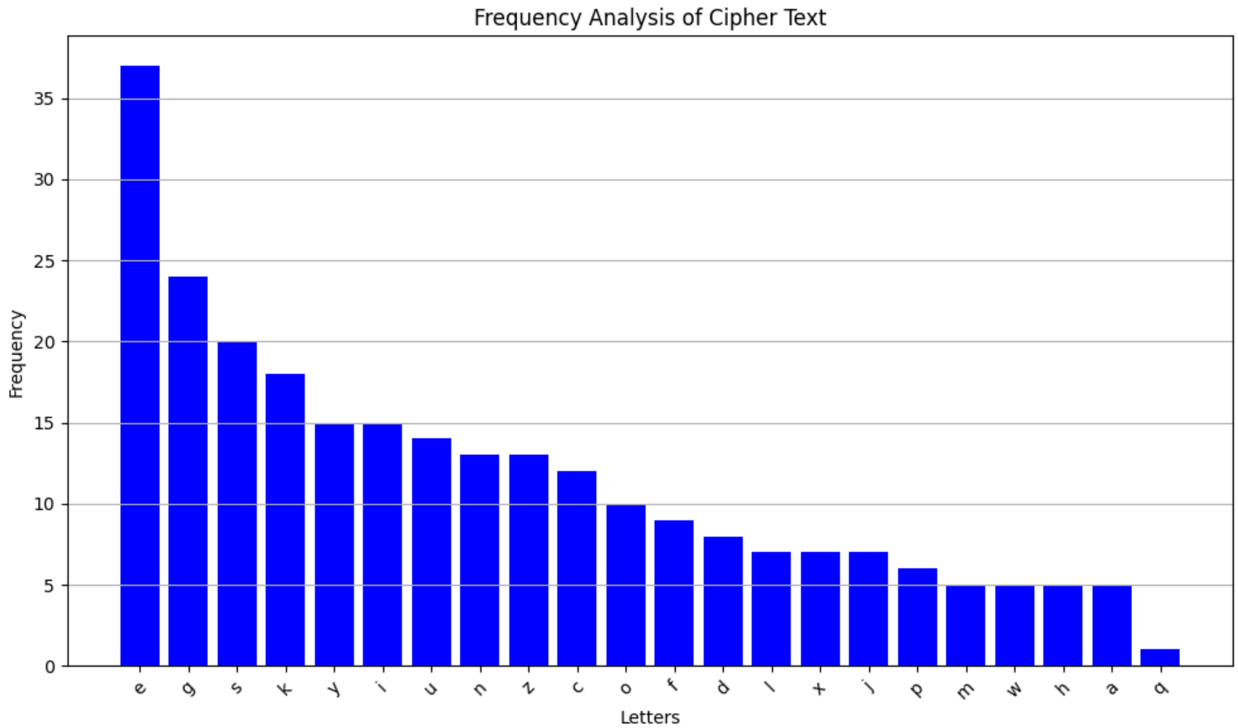
```
    cipher_alphabet = ''.join([sorted_key[letter] for letter in
original_alphabet])

    # Print the original alphabet and the substitution key
    print("Original Alphabet : ", original_alphabet)
    print("Cipher Alphabet   : ", cipher_alphabet)

    return sorted_key

ciphertext =
"EMGLOSUDCGDNCUSWYSFHNSFCYKDPUMLWGYICOXYSIPJCKQPKUGKMGOLICGINCGACKSNIS
ACYKZSCKXECJCKSHYSXCGOIDPKZCNKSHICGIWYGKKGKGOLDSILKGOIUSIGLEDSPWZUGFZC
CNDGYYSFUSZCNXEOJNCGYEOWEUPXEZGACGNFGLKNSACIGOIYCKXCJUCIUZCFZCCNDGYYSF
EUEKUZCSOCFZCCNCIACZEJNCSHFZEJZEGMXCYHCJUMGKUCY"
frequency_analysis(ciphertext)
```

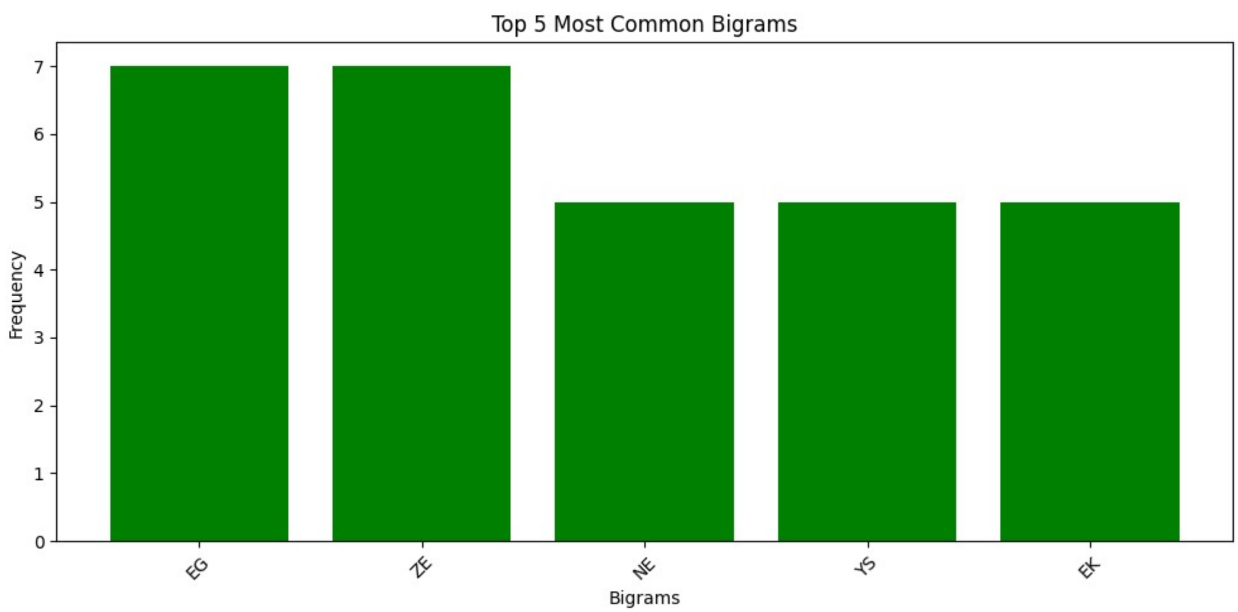


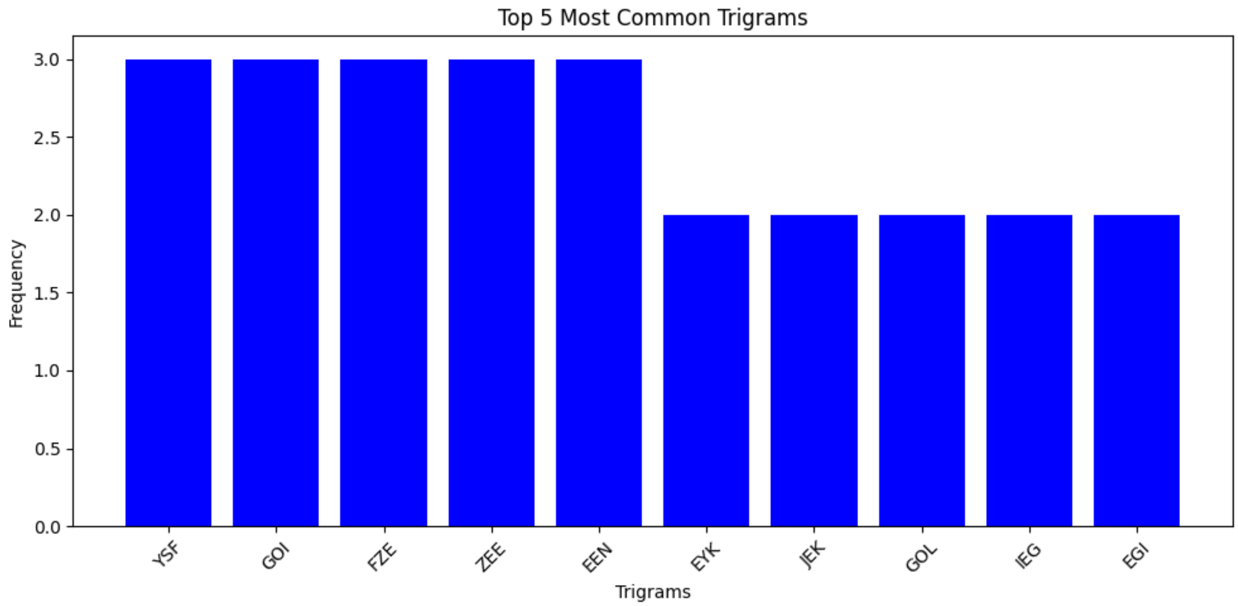
We are going to replace C with E

```
mapping = {'C': 'E', 'E': 'C'}
new_text = replace_letters(ciphertext, mapping)
print(new_text)

frequency_analysis(new_text)
bigram_frequency_analysis(new_text)
trigram_frequency_analysis(new_text)
```

```
CMGLOSUDEGDNEUSWYSFHNSFEYKDPUMLWGYIEOXYSIPJEKQPKUGKMGOLIEGINEGAEKSNISA
EYKZSEKXCEJEKSHYSXEGOIDPKZENKSHIEGIWYGKKGKGOLDSILKGOIUSIGLCDSPWZUGFZEE
NDGYYSFUSZENXCOJNEGYCOWCUPXCZGAEGNFGLKNSAEIGOIYEKXEJUEIUZEFZEENDGYYSFC
UCKUZESOEFZEENEIAEZCJNESHFZCJZCGMXEYHEJUMGKUEY
```







Lookig at the trigram frequency, THE, AND, THA are the most common trigrams, we try to map them to YSF, GOI, FZE, ZEE, EEN.

For THE

- YSF: Y is not common enough to be a T
- GOI: Since we already replaced C with E, GOI does not end with e and cant be
- FZE: F is not common enough to be a T
- ZEE: The cant be this word
- ENN: The cant be this word.

For AND

- YSF: Y is not common enough to be an A
- GOI: G is common enough to be A.
- FZE: And cant be this word
- ZEE: And cant be this word
- ENN: And cant be this word

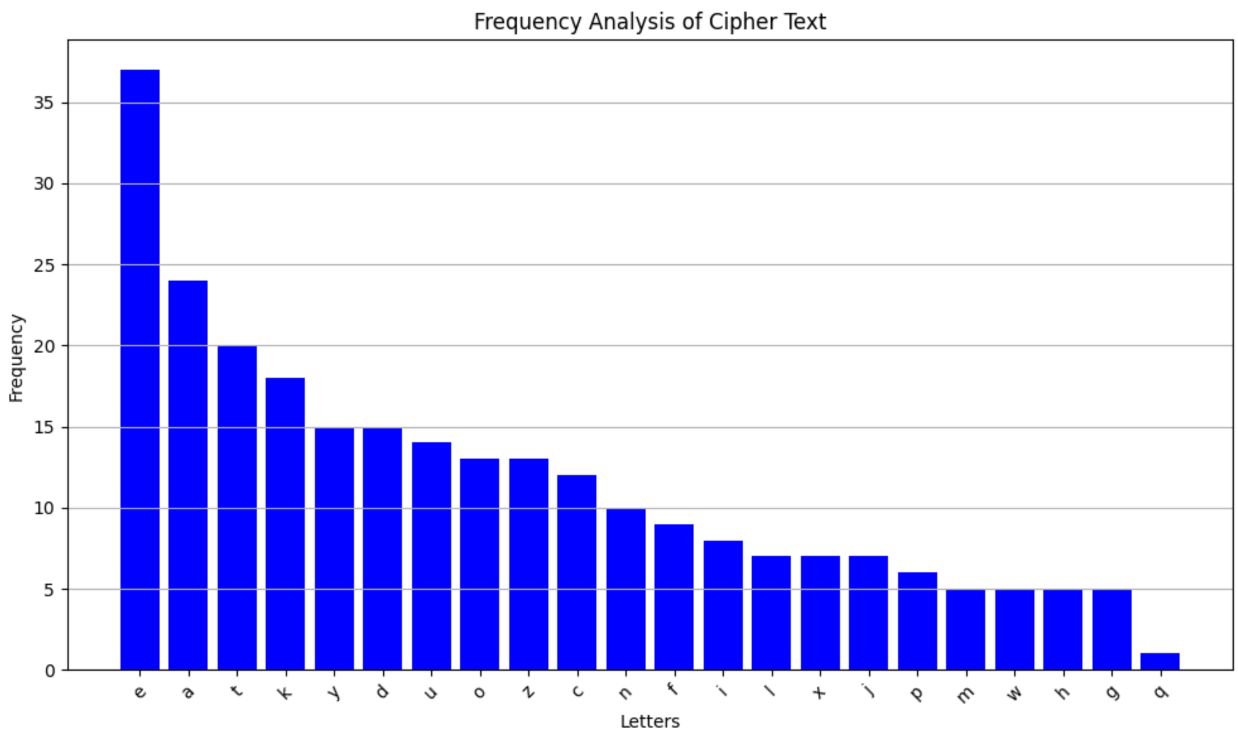
For THA

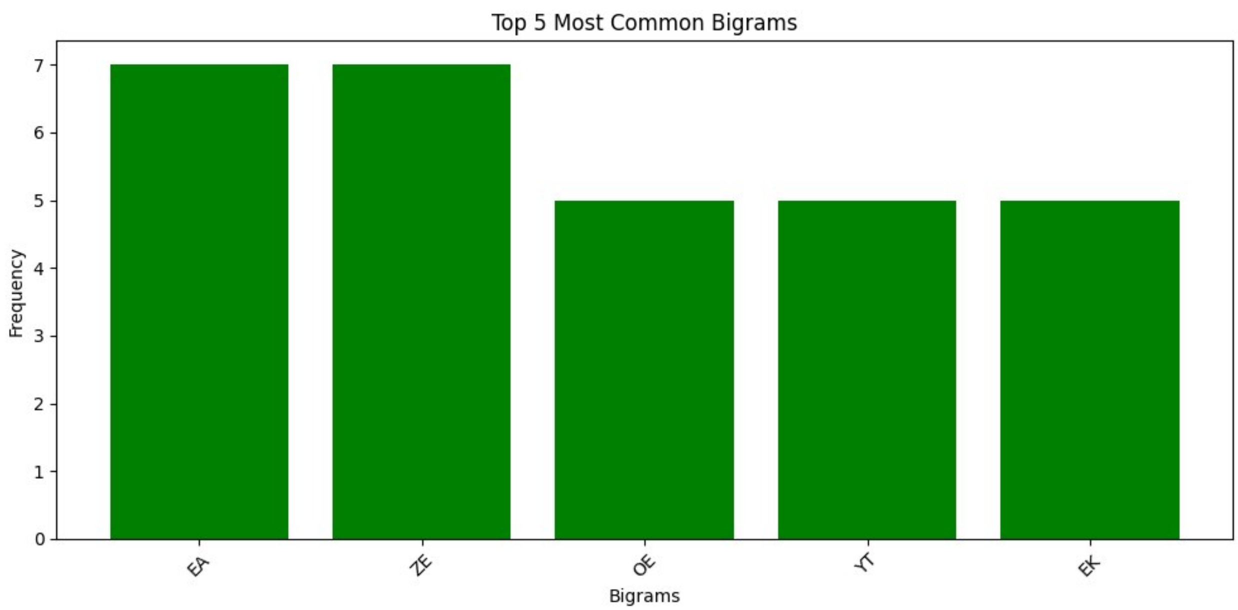
- YSF: Y is not common enough to be an T
- GOI: G is common enough to be T, I is not common enough to be an A.
- FZE: And cant be this word
- ZEE: And cant be this word
- ENN: And cant be this word

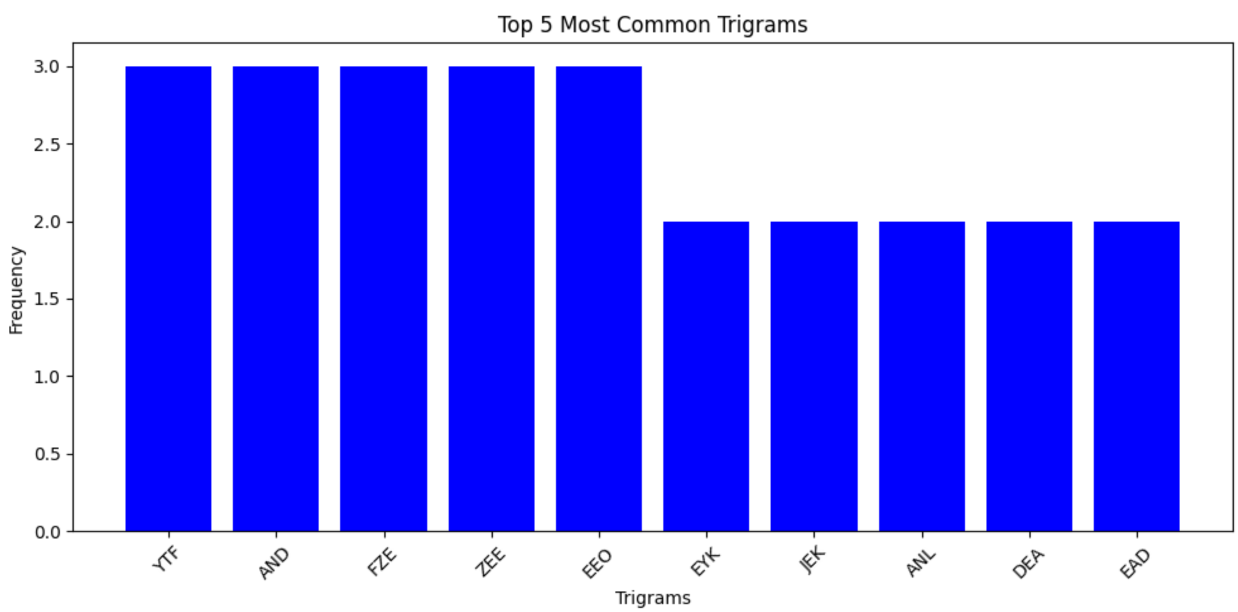
With this we can map GOI to AND and then AND to GOI and since E and A have been map, we can try to map S to T

```python
mapping = {'G': 'A', 'O': 'N', 'I': 'D', 'A': 'G', 'N': 'O', 'D': 'I',
'S': 'T', 'T': 'S'}
new_text_3 = replace_letters(new_text, mapping)
print(new_text_3)

frequency_analysis(new_text_3)
bigram_frequency_analysis(new_text_3)
trigram_frequency_analysis(new_text_3)
```

```
CMALNTUIEAIOEUTWYTFHOTFEYKIPUMLWAYDENXYTDPJEKQPKUAKMANLDEADOEAGEKTODTG
EYKZTEKXCEJEKTHYTXEANDIPKZEOKTHDEADWYAKKAKANLITDLKANDUTDALCITPWZUAFZEE
OIAYYTFUTZEOXCNJOEAYCNWCUPXCZAGEAOFALKOTGEDANDYEKXEJUEDUZEFZEEOIAYYTFC
UCKUZETNEFZEEOEDGEZCJOETHFZCJZCAMXEYHEJUMAKUEY
```







The most common bigrams are TH, HE, IN, ER, so we can try to map them to EA, ZE, OE, YT, EK

For TH

- EA: Cant be, E and A have been map already
- ZE: Cant be
- OE: Cant be
- YT: Cant be
- EK: Cant be

For HE

- EA: Cant be

- ZE: Z can be H
- OE: O can be H
- YT: Cant be
- EK: Cant be

For IN

- EA: Cant be
- ZE: Cant be
- OE: Cant be
- YT: Cant be
- EK: Cant be

For ER

- EA: Cant be
- ZE: Cant be
- OE: Cant be
- YT: Cant be
- EK: K can be R

We run a fitness function to evaluate with Z to H, O to H and K to R, we also try K to S since K is frequent enough to be map to S

```
mapping = {'Z': 'H', 'H': 'Z'}
option1 = replace_letters(new_text_3, mapping)
mapping = {'O': 'H', 'H': 'O'}
option2 = replace_letters(new_text_3, mapping)
mapping = {'K': 'R', 'R': 'K'}
option3 = replace_letters(new_text_3, mapping)
mapping = {'K': 'S', 'S': 'K'}
option4 = replace_letters(new_text_3, mapping)

# Example usage
score1 = fitness_score(option1)
print("Fitness Score:", score1)
score2 = fitness_score(option2)
print("Fitness Score:", score2)
score3 = fitness_score(option3)
print("Fitness Score:", score3)
score4 = fitness_score(option4)
print("Fitness Score:", score4)
```

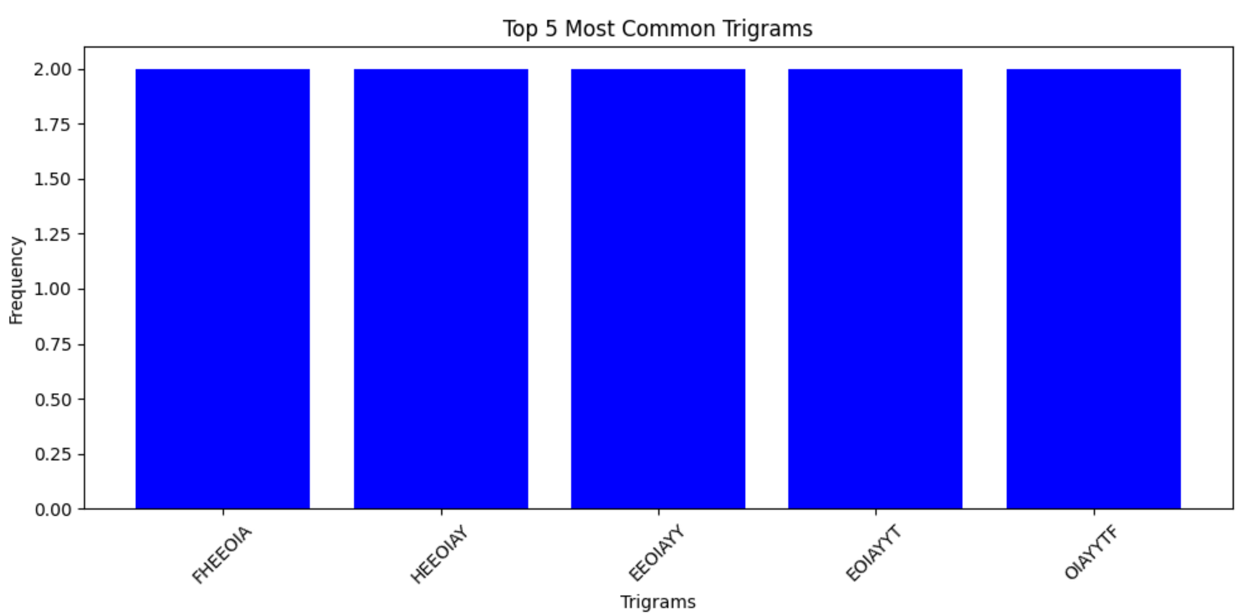
```
Fitness Score: 14.675895009309727
Fitness Score: 16.17000451088299
Fitness Score: 13.361024547772324
Fitness Score: 13.174874813485957
```

The best fitness scores imply that Z maps to H and K to S, now we see if they get a better score together

```
mapping = {'K': 'S', 'S': 'K', 'Z': 'H', 'H': 'Z'}
option5 = replace_letters(new_text_3, mapping)

score3 = fitness_score(option3)
print("Fitness Score:", score3)
score4 = fitness_score(option4)
print("Fitness Score:", score4)
score5 = fitness_score(option5)
print("Fitness Score:", score5)

Fitness Score: 13.361024547772324
Fitness Score: 13.174874813485957
Fitness Score: 11.713001081033083
```

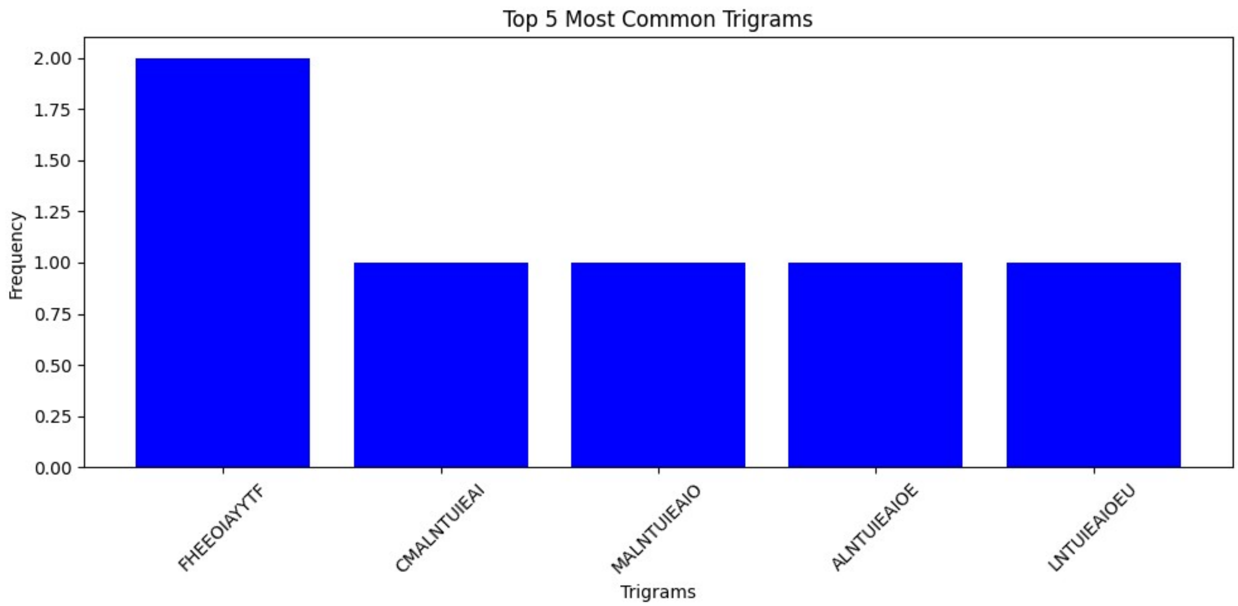
Since the fitness is lower, we do map it

```
mapping = {'K': 'S', 'S': 'K', 'Z': 'H', 'H': 'Z'}
new_text_4 = replace_letters(new_text_3, mapping)
print(new_text_4)

seven_frequency_analysis(new_text_4)

CMALNTUIEAIOEUTWYTFZOTFEYSIPUMLWAYDENXYTDPJESQPSUASMANLDEADOEAGESTODTG
EYSHTESXCEJESTZYTXEANDIPSHEOSTZDEADWYASSASANLITDLSANDUTDALCITPWHUAFHEE
OIAYYTFUTHEOXCNJOEAYCNWCUPXCHAGEAOFALSOTGEDANDYESXEJUEDUHEFHEEOIAYYTFC
UCSUHETNEFHEEOEDGEHCJOETZFHCJHCAMXEYZEJUMASUEY
```



Looking at a 7-letter word frequency analysis, we can see that the most common 7-letter words are just a big word, so we try a 11 letter analysis.

```
eleven_frequency_analysis(new_text_4)
```



Analyzing the word "FHEEOIAYYTF" we can try to guess the following:

- F needs to be a letter that goes after an H, it cant be a T since its been map already, it could be a W or a C
- Y is a letter that can be used twice, since it comes after an A, a vowel, it is not likely to be another vowel. It could be an L or R.

We try to map this and run a fitness function

```
dic = {'F': 'W', 'W': 'F'}
option6 = replace_letters(new_text_4, dic)
score = fitness_score(option6)
print("Fitness Score:", score)
dic = {'F': 'C', 'C': 'F'}
option6 = replace_letters(new_text_4, dic)
score = fitness_score(option6)
print("Fitness Score:", score)
dic = {'Y': 'L', 'L': 'Y'}
option6 = replace_letters(new_text_4, dic)
score = fitness_score(option6)
print("Fitness Score:", score)
dic = {'Y': 'R', 'R': 'Y'}
option6 = replace_letters(new_text_4, dic)
score = fitness_score(option6)
print("Fitness Score:", score)
```

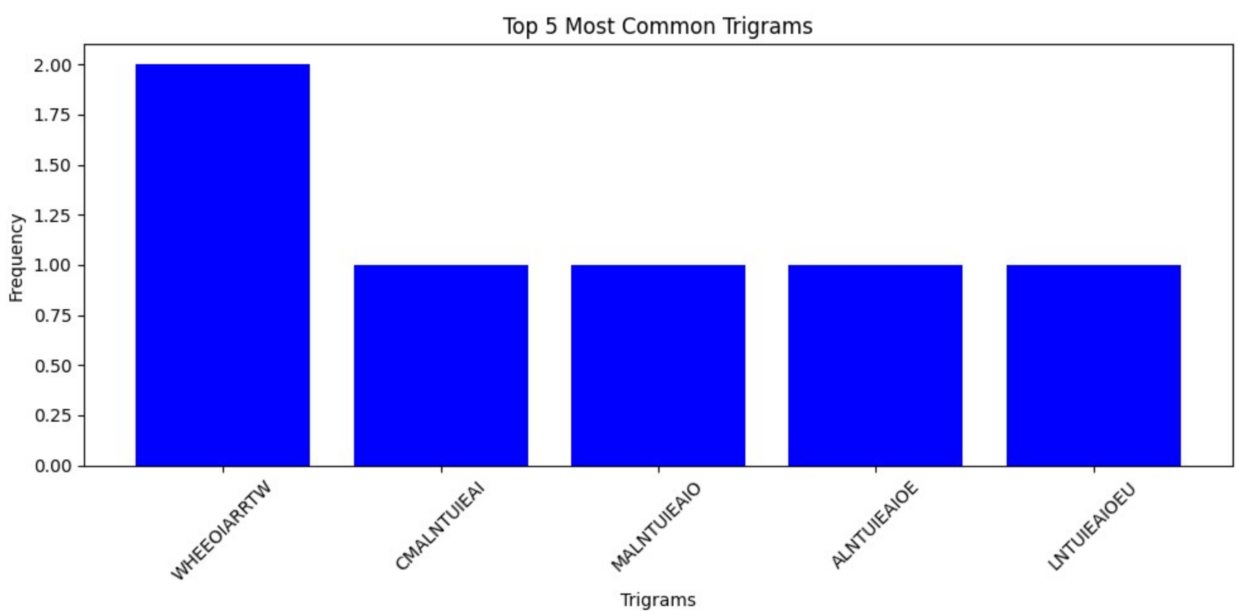
```
Fitness Score: 11.695646383346334
Fitness Score: 11.767899422763698
Fitness Score: 11.149860731160823
Fitness Score: 9.445477514031413
```

F to W and Y to R are the best options, we now put the together and see if the fitness score improves

```
dic = {'Y': 'R', 'R': 'Y', 'F': 'W', 'W': 'F'}
option6 = replace_letters(new_text_4, dic)
score = fitness_score(option6)
print("Fitness Score:", score)

Fitness Score: 9.423947976727856
```

Its slightly better, so we still map it

```
new_text_5 = replace_letters(new_text_4, dic)
print(new_text_5)

eleven_frequency_analysis(new_text_5)

CMALNTUIEAIOEUTFRTWZOTWERSIPUMLFARDENXRTDPJESQPSUASMANLDEADOEAGESTODTG
ERSHTESXCEJESTZRTXEANDIPSHEOSTZDEADFRASSASANLITDLSANDUTDALCITPFHUAWHEE
OIARRTWUTHEOXCNJOEARCNFCUPXCHAGEAOWALSOTGEDANDRESXEJUEDUHEWHEEOIARRTWC
UCSUHETNEWHEEOEDGEHCJOETZWHCJHCAMXERZEJUMASUER
```



Looking at the word WHEEOIARRTW we can guess the O being an L and getting the word WHEEL, since IARRTOW is next to WHEEL, and it repeats twice, its probably a word that begins with WHEEL, considering _ARR_W is the word, we can guess it to be WHEELBARROW and map I to B, and T to O

```
dic = {'O': 'L', 'I': 'B', 'T': 'O', 'B': 'I', 'L': 'T'}
new_text_6 = replace_letters(new_text_5, dic)
print(new_text_6)
```

```
CMATNOUBEABLEUOFROWZLOWERSBPUMTFARDENXRODPJESQPSUASMANTDEADLEAGESOLDOG
ERSHOESXCEJESOZROXEANDBPSHELSOZDEADFRASSASANTBODTSANDUODATCBOPFHUAWHEE
LBARROWUOHELXCNJLEARCNFCUPXCHAGEALWATSLOGEDANDRESXEJUEDUHEWHEELBARROWC
UCSUHEONEWHEELEDGEHCJLEOZWHCJHCAMXERZEJUMASUER
```

Looking at an extract of the semi_cipher code, we have "BEABLEUOFROWZLOWERS", which can be separated to:

BE ABLE UOFROW ZLOWERS

Which means Z maps to F. From context, UOFROW probably separates in UO FROW, and U maps to T and F to G to create GROW, a common thing done to flowers.

```
dic = {'Z': 'F', 'F': 'G', 'U': 'T', 'T': 'U', 'G': 'Z'}
new_text_7 = replace_letters(new_text_6, dic)
print(new_text_7)
```

```
CMAUNOTBEABLETOGROWFLOWERSBPTMUGARDENXRODPJESQPSTASMANUDEADLEAZESOLDOZ
ERSHOESXCEJESOFROXEANDBPSHELSOFDEADGRASSASANUBODUSANDTODAUCBOPGHTAWHEE
LBARROWTOHELXCNJLEARCNGCTPXCHAZEALWAUSLOZEDANDRESXEJTEDTHEWHEELBARROWC
TCSTHEONEWHEELEDZEHCJLEOFWHCJHCAMXERFEJTMASTER
```

Looking at the tetx after "BE ABLE TO GROW FLOWERS" we have "BPTMUGARDEN", which can be separated to

BPTMU GARDEN

By context we can see U maps to Y to create MY, meaning BPT is its own word, the only words that fit, considering B and T have been map, are

- BAT (Cant be since A has been mapped already)
- BET
- BIT
- BUT

From context, BUT is the most probable, so P maps to U

```
dic = {'U': 'Y', 'P': 'U', 'Y': 'P'}
new_text_8 = replace_letters(new_text_7, dic)
print(new_text_8)
```

```
CMAYNOTBEABLETOGROWFLOWERSBUTMYGARDENXRODUJESQUSTASMANYDEADLEAZESOLDOZ
ERSHOESXCEJESOFROXEANDBUSHELSOFDEADGRASSASANYBODYSANDTODAYCBOUGHTAWHEE
```

```
LBARROWTOHELXCNJLEARCNGCTUXCHAZEALWAYSLOZEDANDRESXEJTEDTHEWHEELBARROWC
TCSTHEONEWHEELEDZEHCJLEOFWHCJHCAMXERFEJTMASTER
```

Looking at "CMAYNOTBEABLE" we can see that C maps to I.

And, "QUSTASMANYDEADLEAZES" can be separated to

QUST AS MANY DEAD LEAZES

So Q maps to J, and from context of talking about garden and flowers, Z maps to V.

```
dic = {'Q': 'J', 'J': 'Q', 'Z': 'V', 'V': 'Z', 'C': 'I', 'I': 'C'}
new_text_9 = replace_letters(new_text_8, dic)
print(new_text_9)
```

```
IMAYNOTBEABLETOGROWFLOWERSBUTMYGARDENXRODUQESJUSTASMANYDEADLEAVESOLDOV
ERSHOESXIEQESOFROXEANDBUSHELSOFDEADGRASSASANYBODYSANDTODAYIBOUGHTAWHEE
LBARROWTOHELXINQLEARINGITUXIHAVEALWAYSLOVEDANDRESXEQTEDTHEWHEELBARROWI
TISTHEONEWHEELEDVEHIQLEOFWHIQHIAMXERFEQTMASTER
```

Looking at "IHAVEALWAYSLOVEDANDRESXEQTEDTHEWHEELBARROW" we separate them into

I HAVE ALWAYS LOVED AND RESXEQTED THE WHEELBARROW

Meaning X maps to P and Q to C.

```
dic = {'X': 'P', 'P': 'X', 'Q': 'C', 'C': 'Q'}
new_text_10 = replace_letters(new_text_9, dic)
print(new_text_10)
```

```
IMAYNOTBEABLETOGROWFLOWERSBUTMYGARDENPRODUCESJUSTASMANYDEADLEAVESOLDOV
ERSHOESPIECESOFROPEANDBUSHELSOFDEADGRASSASANYBODYSANDTODAYIBOUGHTAWHEE
LBARROWTOHELPINCLEARINGITUPIHAVEALWAYSLOVEDANDRESPECTEDTHEWHEELBARROWI
TISTHEONEWHEELEDVEHICLEOFWHICHIAMPERFECTMASTER
```

We have:

I MAY NOT BE ABLE TO GROW FLOWERS BUT MY GARDEN PRODUCES JUST AS MANY DEAD LEAVES OLD OVER SHOES PIECES OF ROPE AND BUSHELS OF DEAD GRASS AS ANYBODYS 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

```
plaintext = new_text_10
key = generate_substitution_key(ciphertext, plaintext)

# Print the key (mapping between original and ciphertext)
print("\nSubstitution Key (Original -> Cipher):")
```

```
for p_char, c_char in key.items():
    print(f"{p_char} -> {c_char}")
```

```
Original Alphabet :  ACDEFGHIJKLMNOPQSUWXYZ
Cipher Alphabet   :  VEBIWAFDCSYMLNUJOTGPRH

Substitution Key (Original -> Cipher):
A -> V
C -> E
D -> B
E -> I
F -> W
G -> A
H -> F
I -> D
J -> C
K -> S
L -> Y
M -> M
N -> L
O -> N
P -> U
Q -> J
S -> O
U -> T
W -> G
X -> P
Y -> R
Z -> H
```

Exercise 5.2

Vigenere Cipher:

KCCPKBGUFDPHQTYAVINRRTMVGRKDNBVFDETDGILTXRGUDDKOTFMBPVGEGLTGCKQRAC
QCWDNAWCRXIZAKFTLEWRPTYCQKYVXCHKFTPONCQQRHJVAJUWETMCMSPKQDYHJVDAH
CTRLSVSKCGCZQQDZXGSFRLSWCWSJTBHAFSIASPRJAHKJRJUMVGKMITZHFPDISPZLVLGWT
FPLKKEBDPGCEBSHCTJRWXBAFSPEZQNRWXCVYCGAONWDDKACKAWBBIKFTIOVKCGGHJV
LNHIFFSQESVYCLACNVRWBBIREPBBVFEXOSCDYGZWPFDTKFQIYCWHJVLNHIQIBTKHJVNPIS
T

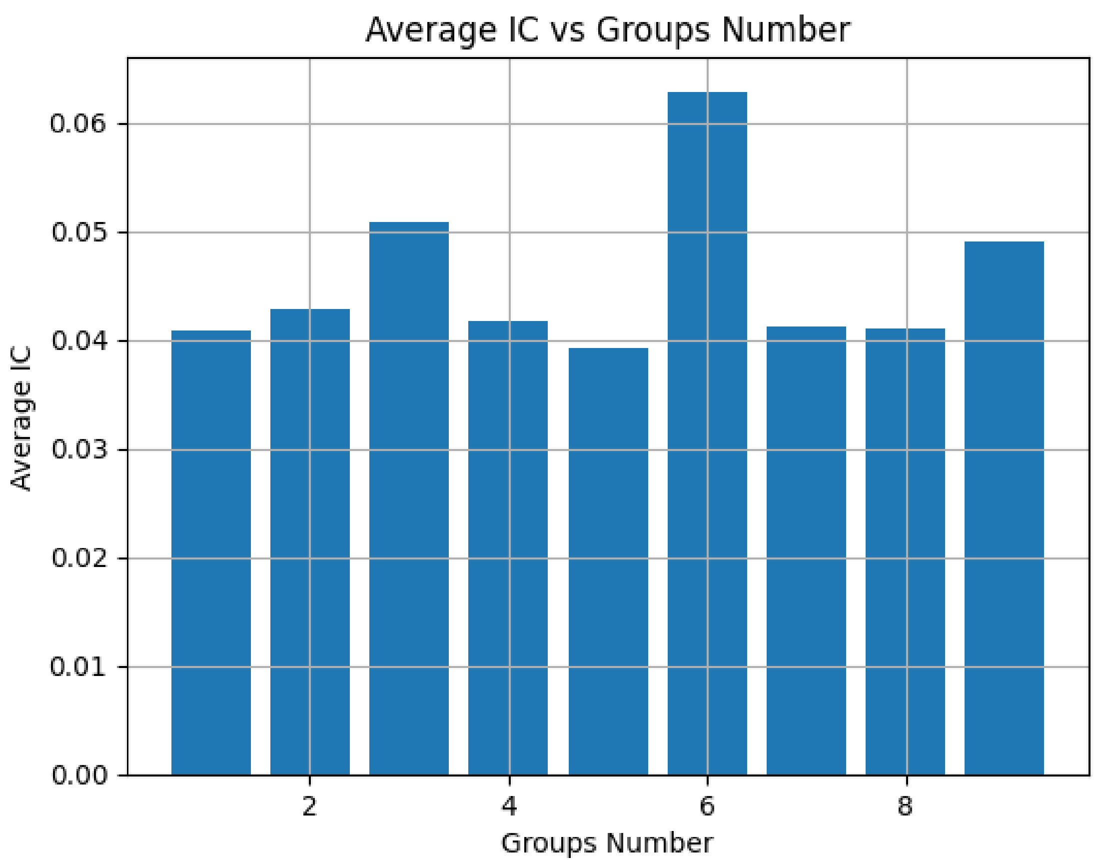
We know that the ciphertext corresponds to a Vigenere cipher. So we can start by determining the key length using the Friedman test, which relies on frequency analysis to estimate how likely it is that two random letters in the ciphertext match. This test gives us an approximation of the key length by calculating the match rate of the ciphertext and comparing it to the expected match rate of an English text.

```
import matplotlib.pyplot as plt
```

```python
# Friedman Test
def calculate_IC(text):
    Alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
    counts = {char: 0 for char in Alphabet}
    numerator = 0
    denominator = len(text) * (len(text) - 1)
    for char in text:
        if char in Alphabet:
            counts[char] += 1

    for char in Alphabet:
        numerator += counts[char] * (counts[char] - 1)
    return numerator / denominator

# Divide the text into n groups
def divide_text(text, n):
    return [text[i::n] for i in range(n)]

text =
'KCCPKBGUFDPHQTYAVINRRTMVGRKDNBVFDETDGILTXRGUDDKOTFMBPVGEGLTGCKQRACQCW
DNAWCRXIZAKFTLEWRPTYCQKYVXCHKFTPONCQQRHJVAJUWETMCMSPKQDYHJVDAHCTRLSVSK
CGCZQQDZXGSFRLSWCWSJTBHAFSIASPRJAHKJRJUMVGKMITZHFPDISPZLVLGWTFPLKKEBDP
GCEBSHCTJRWXBAFSPEZQNRWXCVYCGAONWDDKACKAWBBIKFTIOVKCGGHJVLNHIFFSQESVYC
LACNVRWBBIREPBBVFEXOSCDYGZWPFDTKFQIYCWHJVLNHIQIBTKHJVNPIST'

num_groups = range(1, 10)
average_ICs = []

for i in num_groups:
    groups = divide_text(text, i)
    group_IC = {i: calculate_IC(group) for i, group in
enumerate(groups)}
    average_IC = sum(group_IC.values()) / len(group_IC)
    average_ICs.append(average_IC)
    print(f'Key Length {i}: Avg IC: {average_IC:.4f}')

# Graficar los valores de average_IC vs número de grupos como un
diagrama de barras
plt.bar(num_groups, average_ICs)
plt.xlabel('Groups Number')
plt.ylabel('Average IC')
plt.title('Average IC vs Groups Number')
plt.grid(True)
plt.show()
```

```
Key Length 1: Avg IC: 0.0409
Key Length 2: Avg IC: 0.0428
Key Length 3: Avg IC: 0.0508
Key Length 4: Avg IC: 0.0417
Key Length 5: Avg IC: 0.0393
```

```
Key Length 6: Avg IC: 0.0628
Key Length 7: Avg IC: 0.0412
Key Length 8: Avg IC: 0.0410
Key Length 9: Avg IC: 0.0491
```



By looking the plot of the match rate, we can see that the most probable **key length is 6**. Then, we can divide the ciphertext into 6 groups and solve each group as a Caesar cipher. For that, we can use a fitness function to evaluate the quality of the decryption for each possible key letter. The fitness function is based on the frequency of the letters in the English language.

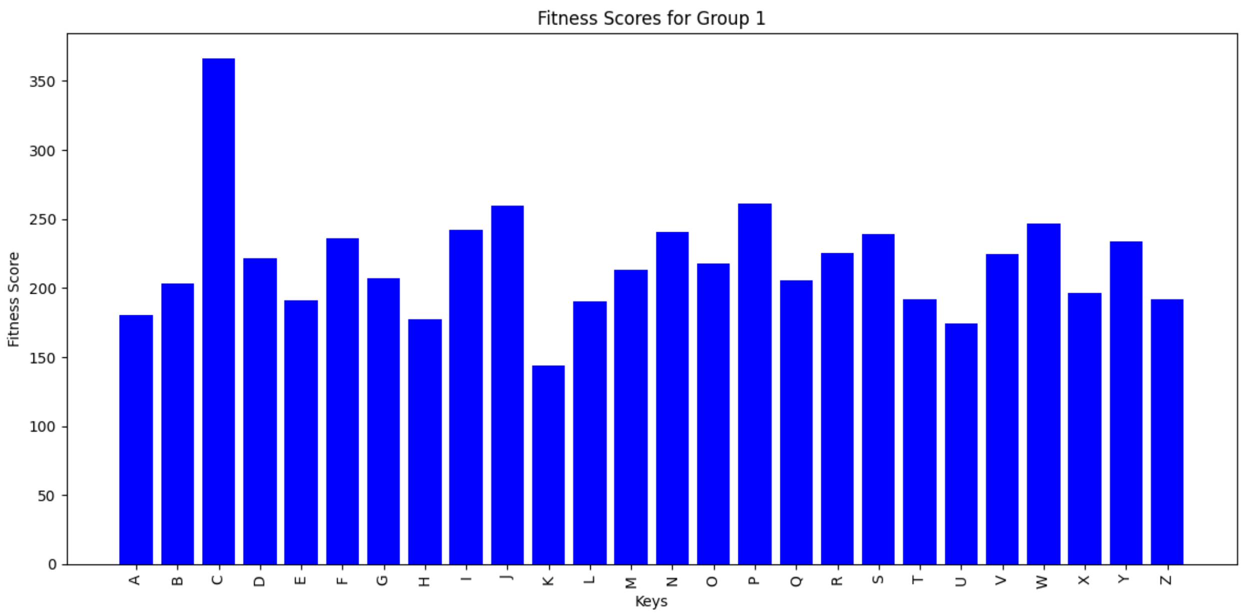
In this code, we will get the best two possible keys for each group that gives us the best fitness score.

```
import matplotlib.pyplot as plt

# Variable to store the best keys for each group
stored_keys = {}

# Divide the ciphertext into groups
groups = divide_text(text, 6)

# Fitness function to calculate the score of a plaintext
def fitness(pt):
```

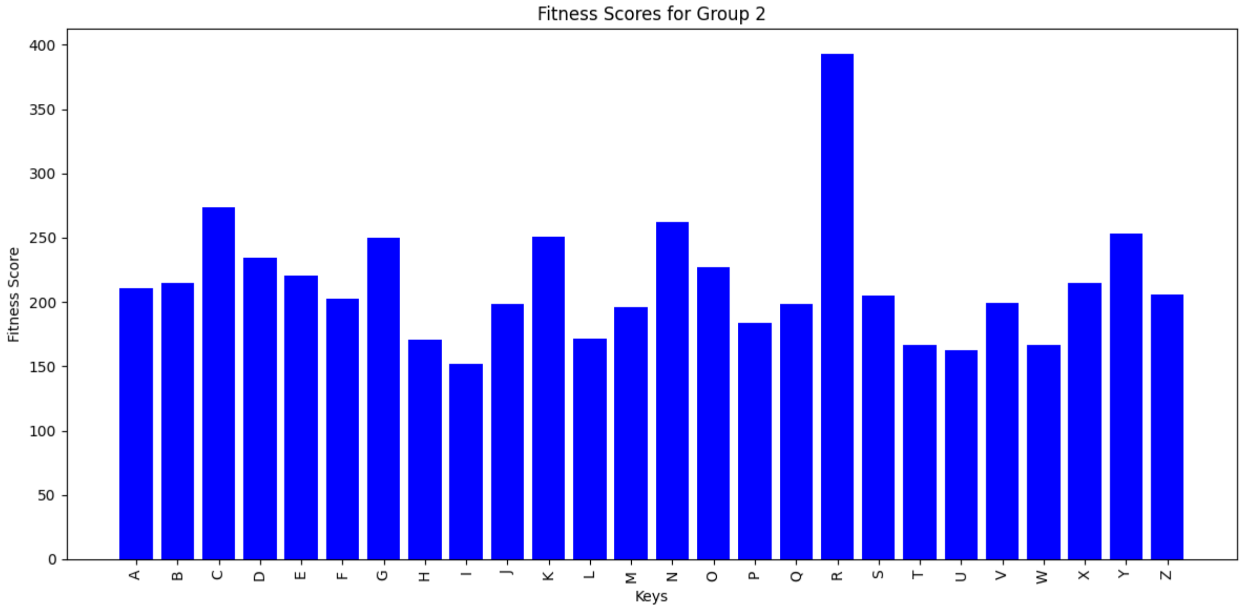
```
    english_frequencies = {
        'E': 12.02, 'T': 9.10, 'A': 8.12, 'O': 7.68,
        'I': 7.00, 'N': 6.95, 'S': 6.28, 'H': 6.09,
        'R': 5.99, 'D': 4.25, 'L': 4.03, 'C': 2.78,
        'U': 2.76, 'M': 2.41, 'W': 2.36, 'F': 2.23,
        'G': 2.02, 'Y': 1.97, 'P': 1.93, 'B': 1.49,
        'V': 0.98, 'K': 0.77, 'J': 0.15, 'X': 0.15,
        'Q': 0.10, 'Z': 0.07
    }

    score = 0
    for letter in pt:
        if letter in english_frequencies:
            score += english_frequencies[letter]
    return score

# Function to decrypt a Vigenere ciphertext
def decrypt_vigenere(ciphertext, key):
    plaintext = ''
    key_length = len(key)
    for i, char in enumerate(ciphertext):
        key_char = key[i % key_length]
        shift = ord(key_char) - ord('A')
        plaintext += chr((ord(char) - ord('A') - shift) % 26 +
ord('A'))
    return plaintext

# Iterate over the groups and find the best key for each group
for i, group in enumerate(groups):
    keys = []
    scores = []
    alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
    for char in alphabet:
        key = char
        pt = decrypt_vigenere(group, key)
        fitness_score = fitness(pt)
        keys.append(f'{key}')
        scores.append(fitness_score)

    print(f'Group: {i +1} : {group}')

    # Order the keys and scores in descending order
    sorted_keys_scores = sorted(zip(keys, scores), key=lambda x: x[1],
reverse=True)
    sorted_keys, sorted_scores = zip(*sorted_keys_scores)

    # Store the 2 best keys for each group
    stored_keys[i + 1] = sorted_keys[:2]
```

```
    print(f'Best Keys: {stored_keys[i+1]}')

    # Plot the fitness scores for each key
    plt.figure(figsize=(12, 6))
    plt.bar(keys, scores, color='blue')
    plt.xlabel('Keys')
    plt.ylabel('Fitness Score')
    plt.title(f'Fitness Scores for Group {i+1}')
    plt.xticks(rotation=90)
    plt.tight_layout()
    plt.show()
```

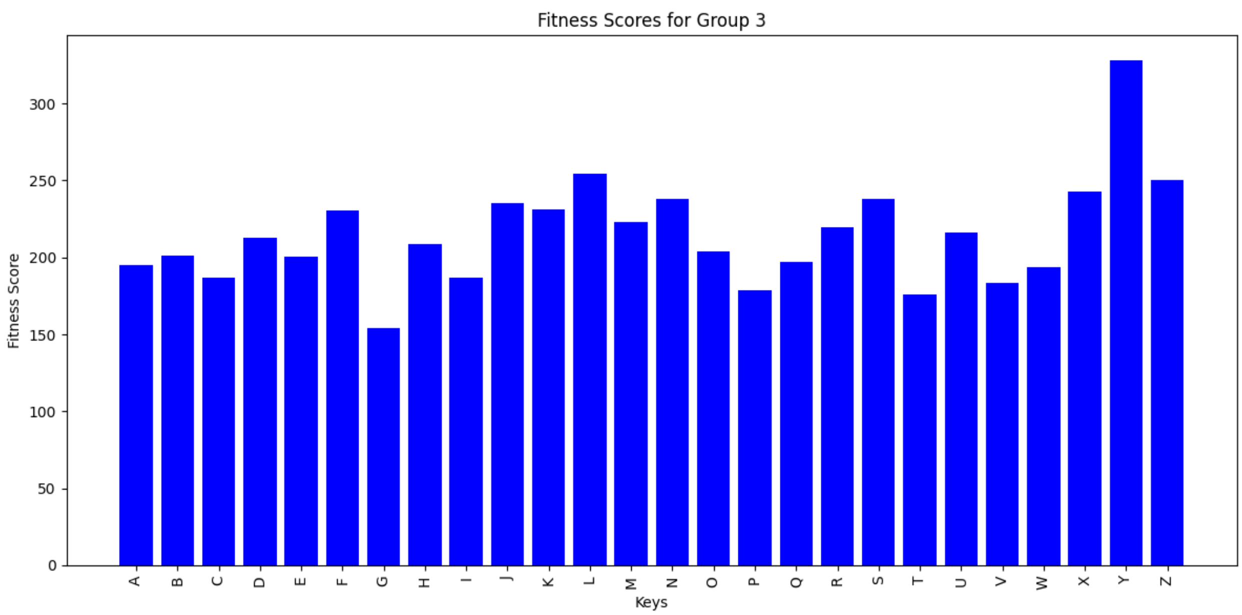
```
Group: 1 : KGQNGVGGTGCQWAWQHNJEPJTKQFWAPJGHPWKCTAQVNCIVJFVNIVCPQJQJT
Best Keys: ('C', 'P')
```



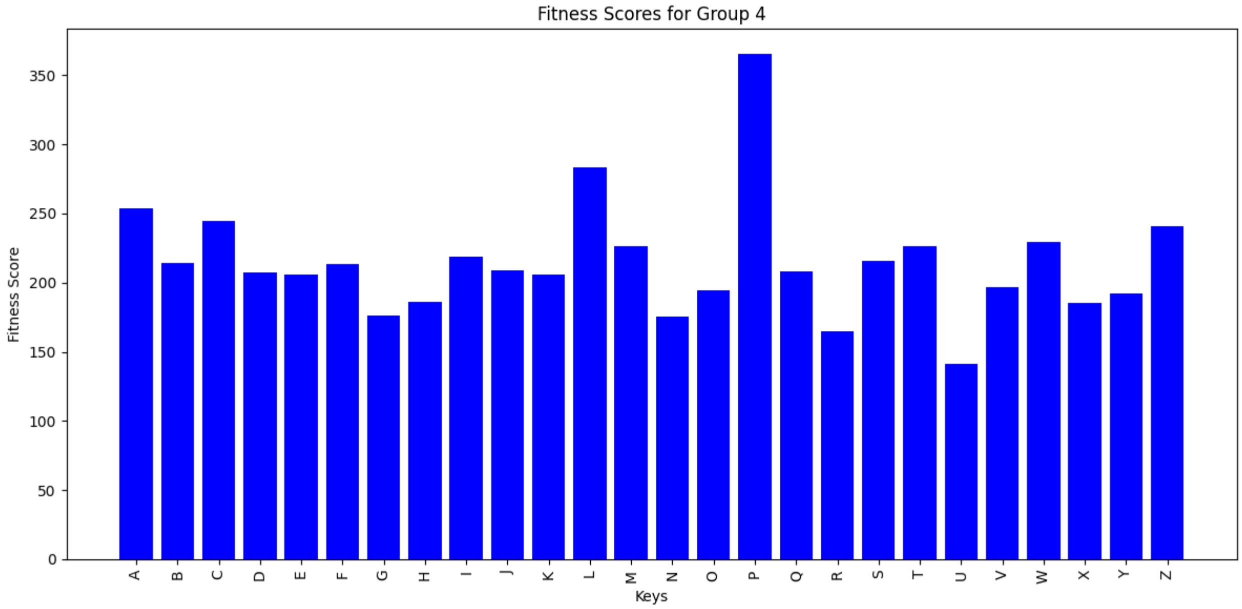

```
Group: 2 : CUTRRFIUFEKCCKRKKCVTKVRCDRSFRRKFZTEEJFNYWKKKVFYVRFDFIVIV
Best Keys: ('R', 'C')
```



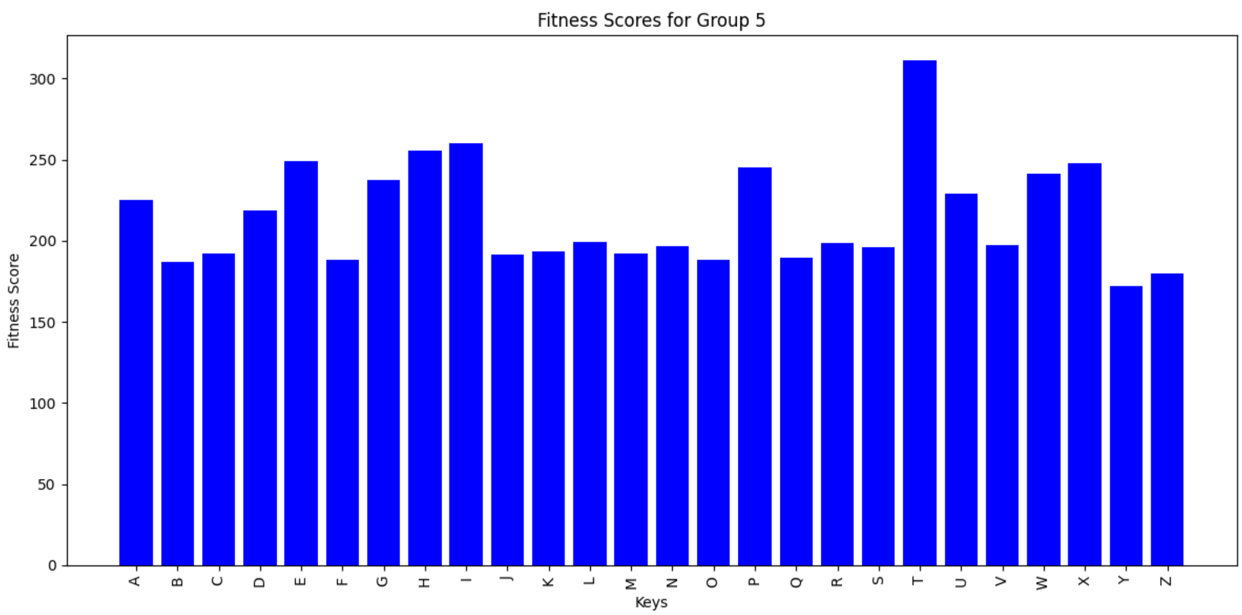

```
Group: 3 : CFYRKDLDMGQWRFPYFQAMQDLGZLJSJJMPLFBBRSRCDAFCLSCREEYDYLBN
Best Keys: ('Y', 'L')
```



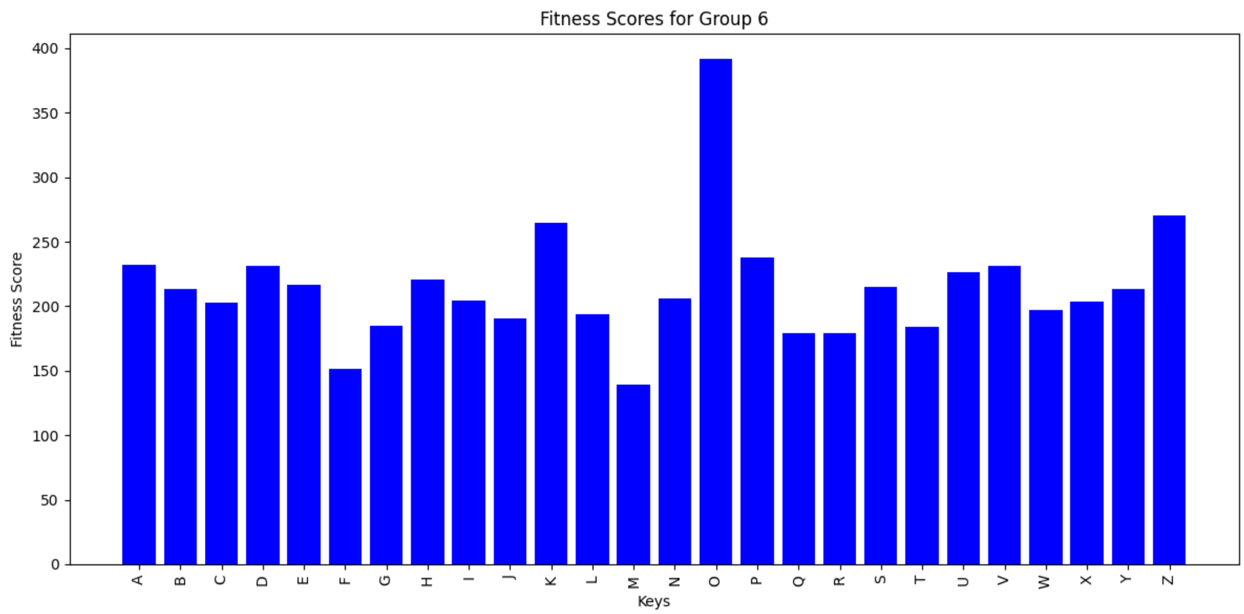

```
Group: 4 : PDATDETDBLRDXTTVTQJCDASCXSTIAUIDVPDSWPWGDWTGNQLWPXGTCNTP
Best Keys: ('P', 'L')
```




```
Group: 5 : KPVMNTXKPTANILYXPRUMYHVZGWBAHMTILLPHXEXAKBIGHEABBOZKWHKI
Best Keys: ('T', 'I')
```




```
Group: 6 : BHIVBDROVGCAZECCOHWSHCSQSCHSKVZSGKGCBZCOABOHISCBBSWFHIHS
Best Keys: ('O', 'Z')
```



The best two keys obtained for each group are:

- Group 1: 'C' and 'P'
- Group 2: 'R' and 'C'
- Group 3: 'Y' and 'L'
- Group 4: 'P' and 'L'
- Group 5: 'T' and 'Y'
- Group 6: 'O' and 'Z'

Then, we can get all possible combinations of the keys (6^2) and decrypt the ciphertext using the Vigenere cipher. Finally, we can check the key that gives the decryption with the most fitness.

```
# Function to generate all possible keys
def generate_keys(stored_keys):
    keys = []
    for key1 in stored_keys[1]:
        for key2 in stored_keys[2]:
            for key3 in stored_keys[3]:
                for key4 in stored_keys[4]:
                    for key5 in stored_keys[5]:
                        for key6 in stored_keys[6]:
                            key = key1 + key2 + key3 + key4 + key5 +
key6
                            keys.append(key)
    return keys

keys = generate_keys(stored_keys)

# Decrypt the text using the keys and print the fitness score
for key in keys:
```

```
    print(f'Key: {key} Fitness: {fitness(decrypt_vigenere(text,
key)):.2f}')
```

```
Key: CRYPTO Fitness: 2154.05
Key: CRYPTZ Fitness: 2032.74
Key: CRYPIO Fitness: 2103.49
Key: CRYPIZ Fitness: 1982.18
Key: CRYLTO Fitness: 2071.92
Key: CRYLTZ Fitness: 1950.61
Key: CRYLIO Fitness: 2021.36
Key: CRYLIZ Fitness: 1900.05
Key: CRLPTO Fitness: 2080.35
Key: CRLPTZ Fitness: 1959.04
Key: CRLPIO Fitness: 2029.79
Key: CRLPIZ Fitness: 1908.48
Key: CRLLTO Fitness: 1998.22
Key: CRLLTZ Fitness: 1876.91
Key: CRLLIO Fitness: 1947.66
Key: CRLLIZ Fitness: 1826.35
Key: CCYPTO Fitness: 2034.86
Key: CCYPTZ Fitness: 1913.55
Key: CCYPIO Fitness: 1984.30
Key: CCYPIZ Fitness: 1862.99
Key: CCYLTO Fitness: 1952.73
Key: CCYLTZ Fitness: 1831.42
Key: CCYLIO Fitness: 1902.17
Key: CCYLIZ Fitness: 1780.86
Key: CCLPTO Fitness: 1961.16
Key: CCLPTZ Fitness: 1839.85
Key: CCLPIO Fitness: 1910.60
Key: CCLPIZ Fitness: 1789.29
Key: CCLLTO Fitness: 1879.03
Key: CCLLTZ Fitness: 1757.72
Key: CCLLIO Fitness: 1828.47
Key: CCLLIZ Fitness: 1707.16
Key: PRYPTO Fitness: 2049.59
Key: PRYPTZ Fitness: 1928.28
Key: PRYPIO Fitness: 1999.03
Key: PRYPIZ Fitness: 1877.72
Key: PRYLTO Fitness: 1967.46
Key: PRYLTZ Fitness: 1846.15
Key: PRYLIO Fitness: 1916.90
Key: PRYLIZ Fitness: 1795.59
Key: PRLPTO Fitness: 1975.89
Key: PRLPTZ Fitness: 1854.58
Key: PRLPIO Fitness: 1925.33
Key: PRLPIZ Fitness: 1804.02
Key: PRLLTO Fitness: 1893.76
Key: PRLLTZ Fitness: 1772.45
Key: PRLLIO Fitness: 1843.20
```

```
Key: PRLLIZ Fitness: 1721.89
Key: PCYPTO Fitness: 1930.40
Key: PCYPTZ Fitness: 1809.09
Key: PCYPIO Fitness: 1879.84
Key: PCYPIZ Fitness: 1758.53
Key: PCYLTO Fitness: 1848.27
Key: PCYLTZ Fitness: 1726.96
Key: PCYLIO Fitness: 1797.71
Key: PCYLIZ Fitness: 1676.40
Key: PCLPTO Fitness: 1856.70
Key: PCLPTZ Fitness: 1735.39
Key: PCLPIO Fitness: 1806.14
Key: PCLPIZ Fitness: 1684.83
Key: PCLLTO Fitness: 1774.57
Key: PCLLTZ Fitness: 1653.26
Key: PCLLIO Fitness: 1724.01
Key: PCLLIZ Fitness: 1602.70
```

Finally, the key with the best fitness is 'CRYPTO'. We use that key to decrypt the ciphertext and get the plaintext.

```
key = 'CRYPTO'
ciphertext =
'KCCPKBGUFDPHQTYAVINRRTMVGRKDNBVFDETDGILTXRGUDDKOTFMBPVGEGLTGCKQRACQCW
DNAWCRXIZAKFTLEWRPTYCQKYVXCHKFTPONCQQRHJVAJUWETMCMSPKQDYHJVDAHCTRLSVSK
CGCZQQDZXGSFRLSWCWSJTBHAFSIASPRJAHKJRJUMVGKMITZHFPDISPZLVLGWTFPLKKEBDP
GCEBSHCTJRWXBAFSPEZQNRWXCVYCGAONWDDKACKAWBBIKFTIOVKCGGHJVLNHIFFSQESVYC
LACNVRWBBIREPBBVFEXOSCDYGZWPFDTKFQIYCWHJVLNHIQIBTKHJVNPIST'
print(decrypt_vigenere(ciphertext, key))
```

```
ILEARNEDHOWTOCALCULATETHEAMOUNTOFPAPERNEEDEDFORAROOMWHENIWASATSCHOOLYO
UMULTIPLYTHESQUAREFOOTAGEOFTHEWALLSBYTHECUBICCONTENTSOFTHEFLOORANDCEIL
INGCOMBINEDANDDOUBLEITYOUTHENALLOWHALFTHETOTALFOROPENINGSSUCHASWINDOWS
ANDDOORSTHENYOUALLOWTHEOTHERHALFFORMATCHINGTHEPATTERNTHENYOUDOUBLETHEW
HOLETHINGAGAINTOGIVEAMARGINOFERRORANDTHENYOUORDERTHEPAPER
```

Exercise 5.3

Affine Cipher:

KQEREJEBCPPCJCRKIEACUZBKRVPKRBCIBQCARBJCVFCUPKRIOFKPACUZQEPBKRXPEIIEABD
KPBCPFCDCCAFIEABDKPBCPFEQPKAZBKRHAIBKAPCCIBURCCDKDCCJCIDFUIXPAFFERBICZD
FKABICBBENEFCUPJCVKABPCYDCCDPKBCOCPERKIVKSCPICBRKIJPKABI

```
ciphertext =
'KQEREJEBCPPCJCRKIEACUZBKRVPKRBCIBQCARBJCVFCUPKRIOFKPACUZQEPBKRXPEIIEA
BDKPBCPFCDCCAFIEABDKPBCPFEQPKAZBKRHAIBKAPCCIBURCCDKDCCJCIDFUIXPAFFERBI
CZDFKABICBBENEFCUPJCVKABPCYDCCDPKBCOCPERKIVKSCPICBRKIJPKABI'
```

To break the cipher, we will use a brute force approach. Thus, we will try all the possible values of a (the multiplicative key), taking into account that a and m must be coprime. Also, we must take into account that the additive key will be in the range of 0 to 26.

```
# Function to find the gdc
def gdc(a, b):
  if b == 0:
    return a
  else:
    return gdc(b, a % b)

# Function to find the multiplicative modular inverse
def modular_inverse(a, m):
  if gdc(a, m) != 1:
    return None
  else:
    for x in range(1, m):
      if (a * x) % m == 1:
        return x

# Function to decrypt Affine Cipher given a and b
def decrypt_affine(ciphertext, a, b):
    plaintext = []
    # Convert to uppercase
    ciphertext = ciphertext.upper()
    # Iterate over the ciphertext
    for c in ciphertext:
        if c.isalpha():
            # Decrypt the letter
            decrypted_char = (modular_inverse(a, 26) * (ord(c) -
ord('A') - b)) % 26
            plaintext.append(chr(decrypted_char + ord('A')))
        else:
            # Keep non-alphabetic characters unchanged
            plaintext.append(c)
    return ''.join(plaintext)

# Initialize a list of coprime numbers to 26
possible_a = []
for i in range(1, 27):
  if gdc(i, 26) == 1:
    possible_a.append(i)

# Possible b values will be in the range of 1 to 26
possible_b = list(range(1, 27))
```

Once we have the lists with all the possible values of a and b, we try all the possible combinations and score them. The frequency distribution was taken from:
https://www3.nd.edu/~busiforc/handouts/cryptography/letterfrequencies.html

```
# Letter frequency distribution of the English alphabet
letter_frequencies = {
    'E': 11.1607, 'A': 8.4966, 'R': 7.5809, 'I': 7.5448, 'O': 7.1635,
'T': 6.9509,
    'N': 6.6544, 'S': 5.7351, 'L': 5.4893, 'C': 4.5388, 'U': 3.6308,
'D': 3.3844,
    'P': 3.1671, 'M': 3.0129, 'H': 3.0034, 'G': 2.4705, 'B': 2.0720,
'F': 1.8121,
    'Y': 1.7779, 'W': 1.2899, 'K': 1.1016, 'V': 1.0074, 'X': 0.2902,
'Z': 0.2722,
    'J': 0.1965, 'Q': 0.1962
}

# Score the possible plaintext
def score_plaintext(plaintext):
  score = 0
  frequencies = {}

  # Iterate over the letters and update frequencies
  for letter in plaintext:
    if letter in frequencies:
      frequencies[letter] += 1
    else:
      frequencies[letter] = 1

  # Iterate over the dictionary and compare frequencies
  for letter, count in frequencies.items():
    observed_frequency = count / len(plaintext) * 100
    expected_frequency = letter_frequencies.get(letter, 0)
    score += (observed_frequency - expected_frequency) ** 2

  # Return the negative score to rank the most simmilar distribution
higher
  return -score

# Initialize possible plaintexts list
possible_plaintexts = []

# Try all the possible combinations
for a in possible_a:
  for b in possible_b:
    plaintext = decrypt_affine(ciphertext, a, b)
    possible_plaintexts.append((plaintext, score_plaintext(plaintext),
a, b))

possible_plaintexts.sort(key=lambda x: x[1], reverse=True)

# Print the top 5 possible plaintexts
for i in range(5):
  print(f"Possible plaintext {i + 1}: {possible_plaintexts[i][0]}")
```

```
    print(f"Score: {possible_plaintexts[i][1]}")
    print(f"a: {possible_plaintexts[i][2]}")
    print(f"b: {possible_plaintexts[i][3]}")
    print()
```

```
Possible plaintext 1:
OCANADATERREDENOSAIEUXTONFRONTESTCEINTDEFLEURONSGLORIEUXCARTONBRASSAIT
PORTERLEPEEILSAITPORTERLACROIXTONHISTOIREESTUNEEPOPEEDESPLUSBRILLANTSE
XPLOITSETTAVALEURDEFOITREMPEEPROTEGERANOSFOYERSETNOSDROITS
Score: -93.51337165518926
a: 19
b: 4

Possible plaintext 2:
AKQDQHQLERREHEDAOQSEIZLADBRADLEOLKESDLHEBJEIRADOYJARSEIZKQRLADNRQOOQSL
XARLERJEXEESJOQSLXARLERJQKRASZLADVSOLASREEOLIDEEXAXEEHEOXJIONRSJJQDLOE
ZXJASLOELLQFQJEIRHEBASLREGXEEXRALEYERQDAOBAWEROELDAOHRASLO
Score: -214.21814931791658
a: 11
b: 10

Possible plaintext 3:
SWOBONOZERRENEBSIOUEQPZSBVRSBZEIZWEUBZNEVTEQRSBIMTSRUEQPWORZSBFROIIOUZ
JSRZERTEJEEUTIOUZJSRZERTOWRSUPZSBDUIZSUREEIZQBEEJSJEENEIJTQIFRUTTOBZIE
PJTSUZIEZZOHOTEQRNEVSUZREKJEEJRSZEMEROBSIVSGERIEZBSINRSUZI
Score: -252.34832850205794
a: 21
b: 22

Possible plaintext 4:
JDPCPKPSREERKRCJLPTRZUSJCYEJCSRLSDRTCSKRYORZEJCLFOJETRZUDPESJCWEPLLPTS
QJESREORQRRTOLPTSQJESREOPDEJTUSJCMTLSJTERRLSZCRRQJQRRKRLQOZLWETOOPCSLR
UQOJTSLRSSPGPORZEKRYJTSERVQRRQEJSRFREPCJLYJBRELRSCJLKEJTSL
Score: -270.1106562557954
a: 25
b: 19

Possible plaintext 5:
LNJWJCJIREERCRWLTJZRXQILWGELWIRTINRZWICRGSRXELWTVSLEZRXQNJEILWYEJTTJZI
ALEIRESRARRZSTJZIALEIRESJNELZQILWKZTILZERRTIXWRRALARRCRTASXTYEZSSJWITR
QASLZITRIIJMJSRXECRGLZIERHARRAELIRVREJWLTGLFRETRIWLTCELZIT
Score: -275.6552210547853
a: 3
b: 3
```

Therefore, the keys are a=19, b=4

Exercise 5.4

unspecified cipher:

BNVSNSIHQCEELSSKKYERIFJKXUMBGYKAMQLJTYAVFBKVTDVBPVVRJYYLAOKYMPQSCGDLFS
RLLPROYGESEBUUALRWXMMASAZLGLEDFJBZAVVPXWICGJXASCBYEHOSNMULKCEAHTQOK
MFLEBKFXLRRFDTZXCIWBJSICBGAWDVYDHAVFJXZIBKCGJIWEAHTTOEWTUHKRQVVRGZBX
YIREMMASCSPBNLHJMBLRFFJELHWEYLWISTFVVYFJCMHYUYRUFSFMGESIGRLWALSWMNUH
SIMYYITCCQPZSICEHBCCMZFEGVJYOCDEMMPGHVAAUMELCMOEHVLTIPSUYILVGFLMVWDV
YDBTHFRAYISYSGKVSUUHYHGGCKTMBLRX

```
ciphertext =
'BNVSNSIHQCEELSSKKYERIFJKXUMBGYKAMQLJTYAVFBKVTDVBPVVRJYYLAOKYMPQSCGDLF
SRLLPROYGESEBUUALRWXMMASAZLGLEDFJBZAVVPXWICGJXASCBYEHOSNMULKCEAHTQOKMF
LEBKFXLRRFDTZXCIWBJSICBGAWDVYDHAVFJXZIBKCGJIWEAHTTOEWTUHKRQVVRGZBXYIRE
MMASCSPBNLHJMBLRFFJELHWEYLWISTFVVYFJCMHYUYRUFSFMGESIGRLWALSWMNUHSIMYYI
TCCQPZSICEHBCCMZFEGVJYOCDEMMPGHVAAUMELCMOEHVLTIPSUYILVGFLMVWDVYDBTHFRA
YISYSGKVSUUHYHGGCKTMBLRX'
```

To identify the cipher method being used, we can calculate specific metrics on the ciphertext to make an educated guess about the encryption technique, and then attempt to break it accordingly.

First we will start with the index of coincidence of the ciphertext. In the case we have a monoalphabetic substitution cipher, we would expect to have an index of coincidence value similar to that of the English language (0.065).

```
# Function to compute the Index of Coincidence for a given text
def get_frequencies(text):
  frequencies = {}
  for letter in text:
    if letter in frequencies:
      frequencies[letter] += 1
    else:
      frequencies[letter] = 1
  return frequencies

def index_of_coincidence(text):
    n = len(text)
    if n <= 1:
        return 0
    frequency = get_frequencies(text)
    ic = sum(f * (f - 1) for f in frequency.values()) / (n * (n - 1))
    return ic
```

Now we will compute the index of coincidence of the ciphertext.

```
ciphertext_ic = index_of_coincidence(ciphertext)
print("Index of coincidence of the ciphertext: ", ciphertext_ic)
```

```
Index of coincidence of the ciphertext:  0.04138199429213872
```

Considering that a random string will have an index of coincidence of a value around 0.038, we can assume that the cipher technique being used is not a monoalphabetic substitution.

Now, we can check for polyalphabetic substitution. For this, we can compute the index of coincidence at the end of each nth letter. If a polyalphabetic substitution method is in use, the index of coincide will increase around n being the length of the key.

```
# Function to break the ciphertext in n groups
def get_groups(ciphertext, n):
  groups = []
  for i in range(n):
    groups.append(ciphertext[i::n])
  return groups

# Calculate the average ic for a given key length
def get_ic_for_key_length(ciphertext, key_length):
  groups = get_groups(ciphertext, key_length)
  ic = []
  for group in groups:
    ic.append(index_of_coincidence(group))
  return sum(ic) / len(ic)
```

We will compute the ic for key lengths in the range from 2 to 10 to check if this is a polyalphabetic substitution cipher.

```
for i in range(2, 11):
  ic = get_ic_for_key_length(ciphertext, i)
  print(f"Index of coincidence for key length {i}: {ic}")
```

```
Index of coincidence for key length 2: 0.045270386826364056
Index of coincidence for key length 3: 0.04683591223009004
Index of coincidence for key length 4: 0.04855992559657028
Index of coincidence for key length 5: 0.041509564358879425
Index of coincidence for key length 6: 0.06062272138422323
Index of coincidence for key length 7: 0.041832730511975794
Index of coincidence for key length 8: 0.05127454003494707
Index of coincidence for key length 9: 0.04445089688992128
Index of coincidence for key length 10: 0.046269954164691006
```

Since the value of the index of coincidence for key length 6 is similar to that of an English language string, we can try for a polyalphabetic substitution cipher, in this case, Vigenere Cipher.

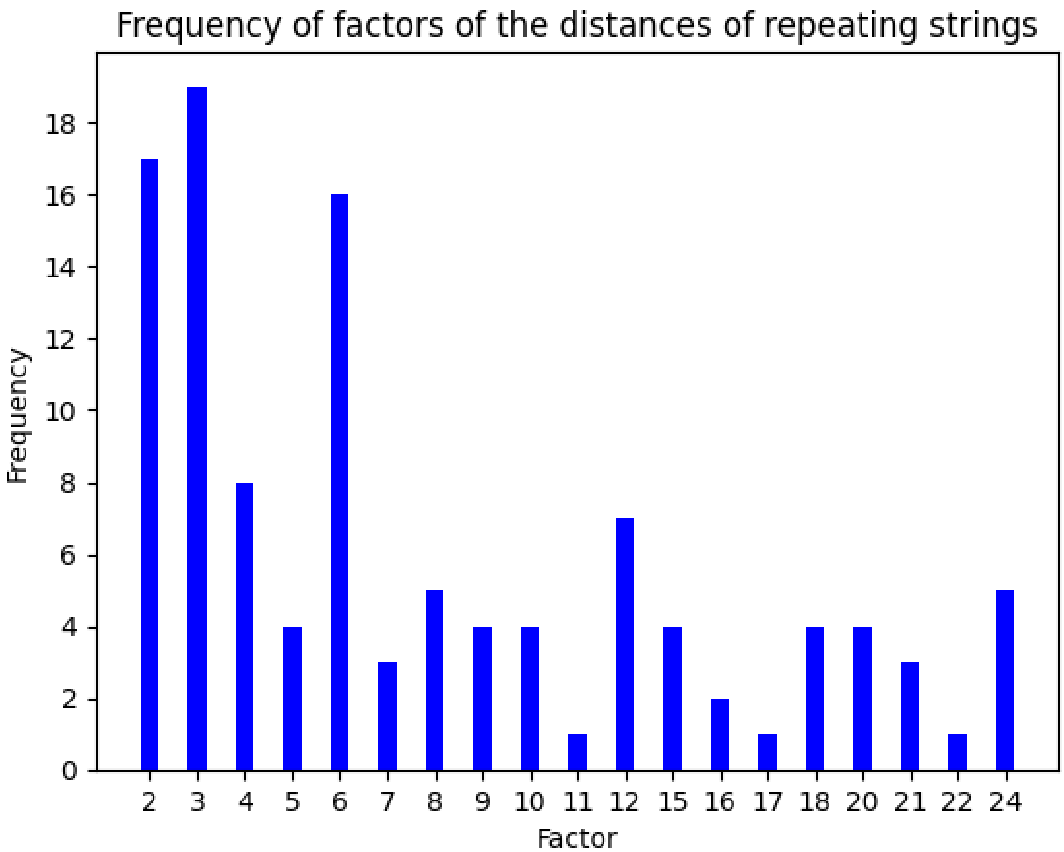
We will start by performing Kasiski Anlaysis.

```
# Check for repeating sequences of length in the range 3 to 10
def kasiski_analysis(ciphertext, length_range):
    # Initialize dictionary
```

```
    sequences = {}
    # Try different repeating string lengths
    for length in range(3, length_range + 1):
        # Iterate over the ciphertext with a step of 1 to avoid
skipping
        for i in range(0, len(ciphertext) - length + 1):
            # Get the sequence
            sequence = ciphertext[i:i+length]
            # Check if the sequence is in the dictionary
            if sequence not in sequences:
                sequences[sequence] = []
            # Append the index
            sequences[sequence].append(i)
    return sequences

# Function to get a list of common factors to a list of numbers
def get_factors_of_list(numbers):
  factors = []
  for number in numbers:
    for i in range(2, 25):
      if number % i == 0:
        factors.append(i)

  return list(set(factors))

# Get a dictionary of the repreated sequences and the factors below 25
of their distances
def get_factors(sequences):
  # Initialize dictionary
  factors = {}
  # Iterate over the sequences
  for sequence, indices in sequences.items():
    if len(indices) > 1:
      distances = []
      # Iterate over the indices
      for i in range(len(indices) - 1):
        # Get the distance
        distance = indices[i + 1] - indices[i]
        # Append the distance
        distances.append(distance)
      # Get the factors
      factors[sequence] = get_factors_of_list(distances)
  return factors

common_sequences = kasiski_analysis(ciphertext, 5)
factors = get_factors(common_sequences)
```

Since Kasiski analysis will tell us the factors of the distances between repeating strings, we can check the most common factors from the repeating strings to get the possible length of the key.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator

# Initialize a dictionary with the numbers of 2 to 25 to count
frequencies
factors_count = {i: 0 for i in range(2, 26)}

# Iterate over the factors
for sequence, factor_list in factors.items():
  for factor in factor_list:
    factors_count[factor] += 1

# Create lists to store the data
factors = list(factors_count.keys())
counts = list(factors_count.values())

# Remove positions where the value is 0
factors = [str(factor) for factor, count in zip(factors, counts) if
count != 0]
counts = [count for count in counts if count != 0]

# Initialize the bar plot
plt.bar(factors, counts, color='blue', width=0.4)
plt.xlabel("Factor")
plt.ylabel("Frequency")
plt.title("Frequency of factors of the distances of repeating
strings")

# Ensure that the y-axis contains only integers
plt.gca().yaxis.set_major_locator(MaxNLocator(integer=True))
```



By analyzing the plot, it is evident that the most likely key is either 3 or 6, given that a key of length 2 is very unlikely, the remaining options are 3 and 6.

In the previous step, we computed the index of coincidence for various key lengths. According to Stinson and Paterson's Cryptography Theory and Practice, the expected index for English text is 0.065. Based on our calculations and the Kasiski examination, we can confidently conclude that the key length is 6.

Now, with the known key length, we can solve the cipher by grouping the ciphertext into 6 columns, applying frequency analysis on each of them. By doing this, we can get the actual key.

```
import string

# Frequency distribution of letters in English
letter_frequencies = {
    'A': 8.167, 'B': 1.492, 'C': 2.782, 'D': 4.253, 'E': 12.702, 'F':
2.228,
    'G': 2.015, 'H': 6.094, 'I': 6.966, 'J': 0.153, 'K': 0.772, 'L':
4.025,
    'M': 2.406, 'N': 6.749, 'O': 7.507, 'P': 1.929, 'Q': 0.095, 'R':
5.987,
    'S': 6.327, 'T': 9.056, 'U': 2.758, 'V': 0.978, 'W': 2.360, 'X':
0.150,
```

```
    'Y': 1.974, 'Z': 0.074
}

# Your score_plaintext function
def score_plaintext(plaintext):
    score = 0
    frequencies = {}

    # Iterate over the letters and update frequencies
    for letter in plaintext:
        if letter in frequencies:
            frequencies[letter] += 1
        else:
            frequencies[letter] = 1

    # Iterate over the dictionary and compare frequencies
    for letter, count in frequencies.items():
        observed_frequency = count / len(plaintext) * 100
        expected_frequency = letter_frequencies.get(letter, 0)
        score += (observed_frequency - expected_frequency) ** 2

    # Return the negative score to rank the most similar distribution
higher
    return -score

# Vigenère cipher decryption function
def vigenere_decrypt(ciphertext, key):
    decrypted_text = []
    key_len = len(key)
    alphabet = string.ascii_uppercase

    for i, char in enumerate(ciphertext):
        if char in alphabet:
            char_index = alphabet.index(char)
            key_char = key[i % key_len]
            key_index = alphabet.index(key_char)
            decrypted_char = alphabet[(char_index - key_index) % 26]
            decrypted_text.append(decrypted_char)
        else:
            decrypted_text.append(char)  # Non-alphabet characters
remain unchanged

    return ''.join(decrypted_text)

# Function to break Vigenère cipher given the key length
def break_vigenere(ciphertext, key_length):
    alphabet = string.ascii_uppercase
    best_key = None
    best_score = float('-inf')
```

```
    # Try all possible keys by brute force on each position of the key
    def try_key(key_chars):
        key = ''.join(key_chars)
        decrypted = vigenere_decrypt(ciphertext, key)
        return score_plaintext(decrypted)

    # Iterate over all key positions
    key_chars = ['A'] * key_length
    for i in range(key_length):
        best_char_for_position = 'A'
        best_score_for_position = float('-inf')

        # Test all letters A-Z for this key position
        for letter in alphabet:
            key_chars[i] = letter
            score = try_key(key_chars)
            if score > best_score_for_position:
                best_score_for_position = score
                best_char_for_position = letter

        key_chars[i] = best_char_for_position  # Fix the best letter
for this position

    return ''.join(key_chars)

# Finding the best key and decrypting
best_key = break_vigenere(ciphertext, 6)
decrypted_message = vigenere_decrypt(ciphertext, best_key)

print("Best key:", best_key)
print("Decrypted message:", decrypted_message)

Best key: THEORY
Decrypted message:
IGREWUPAMONGSLOWTALKERSMENINPARTICULARWHODROPPEDWORDSAFEWATATIMELIKEBE
ANSINAHILLANDWHENIGOTTOMINNEAPOLISWHEREPEOPLETOOKALAKEWOBEGONCOMMATOME
ANTHEENDOFASTORYICOULDNTSPEAKAWHOLESENTENCEINCOMPANYANDWASCONSIDEREDNO
TTOOBRIGHTSOIENROLLEDINASPEECHCOURSETAUGHTBYORVILLESANDTHEFOUNDEROFREF
LEXIVERELAXOLOGYASELFHYPNOTICTECHNIQUETHATENABLEDAPERSONTOSPEAKUPTOTHR
EEHUNDREDWORDSPERMINUTE
```