Names: Anthony Fajardo, Fabrizzio Uscoovich, Pablo Encalada

1. **If an encryption function Ek is identical to the decryption function dk , then the key K is said to be an involutory key. Find all the involutory keys in the Shift Cipher**

C=Ek (P) = (P+K) mod 26

P=Dk (C)= (C-K) mod 26

We get an Involuntory Key when the cipher function P and the decoded C are equal so:

P =C

Ek (P) = Dk (P)

For P to be valid in the ecuation the next condition should be meet.

$P+K \equiv P-K \mod(26)$

$K \equiv -K \mod(26)$

$2k \equiv 0 \mod(26)$

This tells us that 2k is a multiple of 26

2k = 26 m

If m =0 then K =0

If m=1 then k= 13

So the involuntary keys in the Shift Cipher are 0 and 13

0 becaues if therew is no shifting all letters remain the same

And 13 because if we apply the cypher twice we get the same word.

2. Suppose that π is the following permutation of .

- Compute **π^-1**

  π(x)={4,1,6,2,7,3,8,5}

  π^-1= {2,4,6,1,8,3,5,7}

- Decrypt the following ciphertext, for a Permutation Cipher with m = 8, which was encrypted using the key **π** : TGEEMNEL NNTDROEO AAHDOETC SHAEIRLM

1 → 2

2 → 4

3→ 6

4→1

5→8

6→3

7→5

8→7

TGEEMNEL = GENTLEME

NNTDROEO= NDONOTRE

AAHDOETC= ADEACHOT

SHAEIRLM = HERSMAIL

Decryption: GENTLEMEN DO NOT READ EACH OTHERS MAIL.

In [39]:

```
import collections

# Ciphertext
ciphertext = """EMGLOSUDCGDNCUSWYSFHNSFCYKDPUMLWGYICOXYSIPJCK
QPKUGKMGOLICGINCGACKSNISACYKZSCKXECJCKSHYSXCG
OIDPKZCNKSHICGIWYGKKGKGOLDSILKGOIUSIGLEDSPWZU
GFZCCNDGYYSFUSZCNXEOJNCGYEOWEUPXEZGACGNFGLKNS
ACIGOIYCKXCJUCIUZCFZCCNDGYYSFEUEKUZCSOCFZCCNC
IACZEJNCSHFZEJZEGMXCYHCJUMGKUCY""".replace("\n", "")

english_freq_order = "ETAOINSHRDLCUMWFGYPBVKJXQZ"

cipher_freq = collections.Counter(ciphertext)
print(cipher_freq)
sorted_cipher_freq = [pair[0] for pair in sorted(cipher_freq.items(), key=la
print(sorted_cipher_freq)

substitution_map = {cipher: eng for cipher, eng in zip(sorted_cipher_freq, e
print(substitution_map)

def decrypt(text, key_map):
    return "".join(key_map.get(char, char) for char in text)

plaintext_guess = decrypt(ciphertext, substitution_map)
print("Initial Decryption Attempt:")
print(plaintext_guess)
```

```
Counter({'C': 37, 'G': 24, 'S': 20, 'K': 18, 'Y': 15, 'I': 15, 'U': 14, 'N':
13, 'Z': 13, 'E': 12, 'O': 10, 'F': 9, 'D': 8, 'L': 7, 'X': 7, 'J': 7, 'P':
6, 'M': 5, 'W': 5, 'H': 5, 'A': 5, 'Q': 1})
['C', 'G', 'S', 'K', 'Y', 'I', 'U', 'N', 'Z', 'E', 'O', 'F', 'D', 'L', 'X',
'J', 'P', 'M', 'W', 'H', 'A', 'Q']
{'C': 'E', 'G': 'T', 'S': 'A', 'K': 'O', 'Y': 'I', 'I': 'N', 'U': 'S', 'N':
'H', 'Z': 'R', 'E': 'D', 'O': 'L', 'F': 'C', 'D': 'U', 'L': 'M', 'X': 'W',
'J': 'F', 'P': 'G', 'M': 'Y', 'W': 'P', 'H': 'B', 'A': 'V', 'Q': 'K'}
Initial Decryption Attempt:
DYTMLASUETUHESAPIACBHACEIOUGSYMPTINELWIANGFEOKGOSTOYTLMNETNHETVEOAHNAVEIORAE
OWDEFEOABIAWETLNUGOREHOABNETNPITOOTOTLMUANMOTLNSANTMDUAGPRSTCREEHUTIIACSAREH
WDLFHETIDLPDSGWDRTVETHCTMOHAVENTLNIEOWEFSENSRECREEHUTIIACDSDOSREALECREEHENVE
RDFHEABCRDFRDTYWEIBEFSYTOSEI
```

Substitution Cipher Decryption

Ciphertext and Plaintext

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

"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."

Frequency Analysis

From the code we get the fre quency analisis

- **Ctxt:** A:5, C:37, D:8, E:12, F:9, G:24, H:5, I:15, J:7, K:18, L:7, M:5, N:13, O:10, P:6, Q:1, S:20, U:14, W:5, X:7, Y:15, Z:13

We order them from most used to least

- **Rank:** C(1), G(2), S(3), K(4), Y(5), I(6), U(7), Z(8), N(9), D(10), O(11), E(12), A(13), F(14), J(15), X(16), P(17), W(18), H(19), L(20), M(21), Q(22)

Now, we know the most common english letters order *e, t, a, o, n, i, s, h, r, d, l*.

Decryption

The approach relies on frequency analysis and pattern recognition, we mapped the high frequency ciphertext letters to common plaintext ones, then refining with digrams and trigrams. In this way we can make educated guesses to modify the mappping and discover the message.

1. **C → e**: Since its the most common, we assumed it was **e**
2. **Q → j**: Q appears only once, suggesting a rare letter like *j z or x*, after attemps we notices **j** fits the word just.
3. **Z → h**: The digram ZC appeared 7 times, the most common english pair is *he* also we discovered ZCN to be **her**
4. **U → t**: UZC digram, same as before, we assume **_he** so *the* was out first choice
5. **S → o**: We had founf **l-ved** so our first assumption was **i** but then *o* made more sense
6. **K → s**: K was also high in frequency, we noticed we had the triad **CKS** so we assumed it would be a consonant since a triad full of vowels was unlikely, *s* fitted perfectly.
7. **A → v**: We had **NCG_C** wich was already decrypted to **lea_e**, so we assumed **v**, forming **leave**
8. **W → g**: WYGKK decrypts to __*ass*, we also had **dead__ass** wich we thought it was **dead grass**
9. **L → y**: **"alwa_s"** becomes **"always"**
10. **X → p**: **res-e-ted** was assumed to be **respected** having **X** as *p*.

11. **J → c**: From the same assumtion as before

The other substitutions were deduced in the same way, filling gaps via context and frequency.

In [40]:
```python
from collections import Counter
import re

ciphertext = """
KCCPKBGUFDPHQTYAVINRRTMVGRKDNBVFDETDGILTXRGUD
DKOTFMBPVGEGLTGCKQRACQCWDNAWCRXIZAKFTLEWRPTYC
QKYVXCHKFTPONCQQRHJVAJUWETMCMSPKQDYHJVDAHCTRL
SVSKCGCZQQDZXGSFRLSWCWSJTBHAFSIASPRJAHKJRJUMV
GKMITZHFPDISPZLVLGWTFPLKKEBDPGCEBSHCTJRWXBAFS
PEZQNRWXCVYCGAONWDDKACKAWBBIKFTIOVKCGGHJVLNHI
FFSQESVYCLACNVRWBBIREPBBVFEXOSCDYGZWPFDTKFQIY
CWHJVLNHIQIBTKHJVNPIST
"""

ciphertext = ciphertext.replace("\n", "").replace(" ", "")


def extract_char_trigrams_no_spaces(text):
    trigrams = [text[i:i+3] for i in range(len(text) - 2)]
    return Counter(trigrams)

char_trigram_counts = extract_char_trigrams_no_spaces(ciphertext)
print(char_trigram_counts)
```

```
Counter({'HJV': 5, 'KFT': 3, 'MVG': 2, 'BVF': 2, 'DDK': 2, 'HCT': 2, 'RLS':
2, 'KCG': 2, 'AFS': 2, 'RWX': 2, 'VYC': 2, 'WBB': 2, 'BBI': 2, 'JVL': 2, 'VL
N': 2, 'LNH': 2, 'NHI': 2, 'KCC': 1, 'CCP': 1, 'CPK': 1, 'PKB': 1, 'KBG': 1,
'BGU': 1, 'GUF': 1, 'UFD': 1, 'FDP': 1, 'DPH': 1, 'PHQ': 1, 'HQT': 1, 'QTY':
1, 'TYA': 1, 'YAV': 1, 'AVI': 1, 'VIN': 1, 'INR': 1, 'NRR': 1, 'RRT': 1, 'RT
M': 1, 'TMV': 1, 'VGR': 1, 'GRK': 1, 'RKD': 1, 'KDN': 1, 'DNB': 1, 'NBV': 1,
'VFD': 1, 'FDE': 1, 'DET': 1, 'ETD': 1, 'TDG': 1, 'DGI': 1, 'GIL': 1, 'ILT':
1, 'LTX': 1, 'TXR': 1, 'XRG': 1, 'RGU': 1, 'GUD': 1, 'UDD': 1, 'DKO': 1, 'KO
T': 1, 'OTF': 1, 'TFM': 1, 'FMB': 1, 'MBP': 1, 'BPV': 1, 'PVG': 1, 'VGE': 1,
'GEG': 1, 'EGL': 1, 'GLT': 1, 'LTG': 1, 'TGC': 1, 'GCK': 1, 'CKQ': 1, 'KQR':
1, 'QRA': 1, 'RAC': 1, 'ACQ': 1, 'CQC': 1, 'QCW': 1, 'CWD': 1, 'WDN': 1, 'DN
A': 1, 'NAW': 1, 'AWC': 1, 'WCR': 1, 'CRX': 1, 'RXI': 1, 'XIZ': 1, 'IZA': 1,
'ZAK': 1, 'AKF': 1, 'FTL': 1, 'TLE': 1, 'LEW': 1, 'EWR': 1, 'WRP': 1, 'RPT':
1, 'PTY': 1, 'TYC': 1, 'YCQ': 1, 'CQK': 1, 'QKY': 1, 'KYV': 1, 'YVX': 1, 'VX
C': 1, 'XCH': 1, 'CHK': 1, 'HKF': 1, 'FTP': 1, 'TPO': 1, 'PON': 1, 'ONC': 1,
'NCQ': 1, 'CQQ': 1, 'QQR': 1, 'QRH': 1, 'RHJ': 1, 'JVA': 1, 'VAJ': 1, 'AJU':
1, 'JUW': 1, 'UWE': 1, 'WET': 1, 'ETM': 1, 'TMC': 1, 'MCM': 1, 'CMS': 1, 'MS
P': 1, 'SPK': 1, 'PKQ': 1, 'KQD': 1, 'QDY': 1, 'DYH': 1, 'YHJ': 1, 'JVD': 1,
'VDA': 1, 'DAH': 1, 'AHC': 1, 'CTR': 1, 'TRL': 1, 'LSV': 1, 'SVS': 1, 'VSK':
1, 'SKC': 1, 'CGC': 1, 'GCZ': 1, 'CZQ': 1, 'ZQQ': 1, 'QQD': 1, 'QDZ': 1, 'DZ
X': 1, 'ZXG': 1, 'XGS': 1, 'GSF': 1, 'SFR': 1, 'FRL': 1, 'LSW': 1, 'SWC': 1,
'WCW': 1, 'CWS': 1, 'WSJ': 1, 'SJT': 1, 'JTB': 1, 'TBH': 1, 'BHA': 1, 'HAF':
1, 'FSI': 1, 'SIA': 1, 'IAS': 1, 'ASP': 1, 'SPR': 1, 'PRJ': 1, 'RJA': 1, 'JA
H': 1, 'AHK': 1, 'HKJ': 1, 'KJR': 1, 'JRJ': 1, 'RJU': 1, 'JUM': 1, 'UMV': 1,
'VGK': 1, 'GKM': 1, 'KMI': 1, 'MIT': 1, 'ITZ': 1, 'TZH': 1, 'ZHF': 1, 'HFP':
1, 'FPD': 1, 'PDI': 1, 'DIS': 1, 'ISP': 1, 'SPZ': 1, 'PZL': 1, 'ZLV': 1, 'LV
L': 1, 'VLG': 1, 'LGW': 1, 'GWT': 1, 'WTF': 1, 'TFP': 1, 'FPL': 1, 'PLK': 1,
'LKK': 1, 'KKE': 1, 'KEB': 1, 'EBD': 1, 'BDP': 1, 'DPG': 1, 'PGC': 1, 'GCE':
1, 'CEB': 1, 'EBS': 1, 'BSH': 1, 'SHC': 1, 'CTJ': 1, 'TJR': 1, 'JRW': 1, 'WX
B': 1, 'XBA': 1, 'BAF': 1, 'FSP': 1, 'SPE': 1, 'PEZ': 1, 'EZQ': 1, 'ZQN': 1,
'QNR': 1, 'NRW': 1, 'WXC': 1, 'XCV': 1, 'CVY': 1, 'YCG': 1, 'CGA': 1, 'GAO':
1, 'AON': 1, 'ONW': 1, 'NWD': 1, 'WDD': 1, 'DKA': 1, 'KAC': 1, 'ACK': 1, 'CK
A': 1, 'KAW': 1, 'AWB': 1, 'BIK': 1, 'IKF': 1, 'FTI': 1, 'TIO': 1, 'IOV': 1,
'OVK': 1, 'VKC': 1, 'CGG': 1, 'GGH': 1, 'GHJ': 1, 'HIF': 1, 'IFF': 1, 'FFS':
1, 'FSQ': 1, 'SQE': 1, 'QES': 1, 'ESV': 1, 'SVY': 1, 'YCL': 1, 'CLA': 1, 'LA
C': 1, 'ACN': 1, 'CNV': 1, 'NVR': 1, 'VRW': 1, 'RWB': 1, 'BIR': 1, 'IRE': 1,
'REP': 1, 'EPB': 1, 'PBB': 1, 'BBV': 1, 'VFE': 1, 'FEX': 1, 'EXO': 1, 'XOS':
1, 'OSC': 1, 'SCD': 1, 'CDY': 1, 'DYG': 1, 'YGZ': 1, 'GZW': 1, 'ZWP': 1, 'WP
F': 1, 'PFD': 1, 'FDT': 1, 'DTK': 1, 'TKF': 1, 'KFQ': 1, 'FQI': 1, 'QIY': 1,
'IYC': 1, 'YCW': 1, 'CWH': 1, 'WHJ': 1, 'HIQ': 1, 'IQI': 1, 'QIB': 1, 'IBT':
1, 'BTK': 1, 'TKH': 1, 'KHJ': 1, 'JVN': 1, 'VNP': 1, 'NPI': 1, 'PIS': 1, 'IS
T': 1})
```

In [41]:

```python
def find_trigram_positions(text, trigram):
    positions = [i for i in range(len(text) - 2) if text[i:i+3] == trigram]
    return positions

trigram = "HJV"
positions = find_trigram_positions(ciphertext, trigram)
print(f"Positions of '{trigram}': {positions}")
```

```
Positions of 'HJV': [107, 125, 263, 317, 329]
```

In [42]:

```python
import string
import numpy as np
```

```python
# Ciphertext from your input
ciphertext = (
    "KCCPKBGUFDPHQTYAVINRRTMVGRKDNBVFDETDGILTXRGUD"
    "DKOTFMBPVGEGLTGCKQRACQCWDNAWCRXIZAKFTLEWRPTYC"
    "QKYVXCHKFTPONCQQRHJVAJUWETMCMSPKQDYHJVDAHCTRL"
    "SVSKCGCZQQDZXGSFRLSWCWSJTBHAFSIASPRJAHKJRJUMV"
    "GKMITZHFPDISPZLVLGWTFPLKKEBDPGCEBSHCTJRWXBAFS"
    "PEZQNRWXCVYCGAONWDDKACKAWBBIKFTIOVKCGGHJVLNHI"
    "FFSQESVYCLACNVRWBBIREPBBVFEXOSCDYGZWPFDTKFQIY"
    "CWHJVLNHIQIBTKHJVNPIST"
)

# Alphabet
alphabet = string.ascii_uppercase
key_length = 6

def split_into_groups(text, length):
    groups = ["" for _ in range(length)]
    for i, char in enumerate(text):
        groups[i % length] += char
    return groups

def frequency_distribution(text):
    freq = {c: 0 for c in alphabet}
    for char in text:
        freq[char] += 1
    total = sum(freq.values())
    return [freq[c] / total if total > 0 else 0 for c in alphabet]

def mutual_index_of_coincidence(freq, english_freqs):
    return sum(f * e for f, e in zip(freq, english_freqs))

# English letter frequency for comparison
english_freqs = [
    0.082, 0.015, 0.028, 0.043, 0.127, 0.022, 0.020, 0.061, 0.070, 0.002,
    0.008, 0.040, 0.024, 0.067, 0.075, 0.019, 0.001, 0.060, 0.063, 0.091,
    0.028, 0.010, 0.023, 0.001, 0.020, 0.001
]

def find_shift(freq, english_freqs):
    shifts = [(shift, mutual_index_of_coincidence(np.roll(freq, -shift), eng
    return max(shifts, key=lambda x: x[1])[0]

def vigenere_decrypt(ciphertext, key):
    key_repeated = (key * (len(ciphertext) // len(key) + 1))[:len(ciphertext
    return "".join(alphabet[(alphabet.index(c) - alphabet.index(k)) % 26] fo

groups = split_into_groups(ciphertext, key_length)
freqs = [frequency_distribution(group) for group in groups]
key_shifts = [find_shift(freq, english_freqs) for freq in freqs]
print(key_shifts)

def deduce_key_from_shifts(shifts):
    return "".join(alphabet[s] for s in shifts)

key = deduce_key_from_shifts(key_shifts)
```

```python
plaintext = vigenere_decrypt(ciphertext, key)

print(f"Possible Key: {key}")
print(f"Decrypted Text Preview: {plaintext[:100]}")
```

```
[2, 17, 24, 15, 19, 14]
Possible Key: CRYPTO
Decrypted Text Preview: ilearnedhowtocalculatetheamountofpaperneededforaroom
wheniwasatschoolyoumultiplythesquarefootageofthe
```

Vigenère Cipher

Ciphertext and Plaintext

```
KCCPKBGUFDPHQTYAVINRRTMVGRKDNBVFDETDGILTXRGUD
DKOTFMBPVGEGLTGCKQRACQCWDNAWCRXIZAKFTLEWRPTYC
QKYVXCHKFTPONCQQRHJVAJUWETMCMSPKQDYHJVDAHCTRL
SVSKCGCZQQDZXGSFRLSWCWSJTBHAFSIASPRJAHKJRJUMV
GKMITZHFPDISPZLVLGWTFPLKKEBDPGCEBSHCTJRWXBAFS
PEZQNRWXCVYCGAONWDDKACKAWBBIKFTIOVKCGGHJVLNHI
FFSQESVYCLACNVRWBBIREPBBVFEXOSCDYGZWPFDTKFQIY
CWHJVLNHIQIBTKHJVNPIST
```

"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."

Decryption

The Vigenère Cipher works by using a keyword to shift letters

1. **Kasiski Test for Key Length**:
   From the code we see that the most common triad is *HJV* wich appears 5 times, further code shows the triad is at position [107, 125, 263, 317, 329]. Gaps between these are 18, 138, 54, 12 respoectively, we notice the gaps share factors 2, 3, and 6, suggesting a key length of 2, 3, or 6. Also, 138 = 23 × 6, so the most probable is 6.

2. **Index of Coincidence (Ic)**:
   We split the ciphertext into 6 columns because we assume the key length is 6. We calculate the shifts for every letter aand choose the one who maximizes the IC value.

3. **Key Determination**:
   After getting the shifts we map the respective character to get the key.

Decryption

- **Key**: *CRYPTO* (shifts: C=2, R=17, Y=24, P=15, T=19, O=14)
- Each ciphertext letter is decrypted using P = C - K (mod 26), where K repeats every 6 letters.

In [43]:
```
import collections

# Ciphertext
ciphertext = """KQEREJEBCPPCJCRKIEACUZBKRVPKRBCIBQCARBJCVFCUP
KRIOFKPACUZQEPBKRXPEIIEABDKPBCPFCDCCAFIEABDKP
BCPFEQPKAZBKRHAIBKAPCCIBURCCDKDCCJCIDFUIXPAFF
ERBICZDFKABICBBENEFCUPJCVKABPCYDCCDPKBCOCPERK
IVKSCPICBRKIJPKABI""".replace("\n", "")

english_freq_order = "ETAOINSHRDLCUMWFGYPBVKJXQZ"

cipher_freq = collections.Counter(ciphertext)
print(cipher_freq)
sorted_cipher_freq = [pair[0] for pair in sorted(cipher_freq.items(), key=la
print(sorted_cipher_freq)

substitution_map = {cipher: eng for cipher, eng in zip(sorted_cipher_freq, e
print(substitution_map)

def decrypt(text, key_map):
    return "".join(key_map.get(char, char) for char in text)

plaintext_guess = decrypt(ciphertext, substitution_map)
print("Initial Decryption Attempt:")
print(plaintext_guess)
```

```
Counter({'C': 32, 'B': 21, 'K': 20, 'P': 20, 'I': 16, 'E': 13, 'A': 13, 'R':
12, 'F': 10, 'D': 9, 'J': 6, 'U': 6, 'Q': 4, 'Z': 4, 'V': 4, 'O': 2, 'X': 2,
'H': 1, 'N': 1, 'Y': 1, 'S': 1})
['C', 'B', 'K', 'P', 'I', 'E', 'A', 'R', 'F', 'D', 'J', 'U', 'Q', 'Z', 'V',
'O', 'X', 'H', 'N', 'Y', 'S']
{'C': 'E', 'B': 'T', 'K': 'A', 'P': 'O', 'I': 'I', 'E': 'N', 'A': 'S', 'R':
'H', 'F': 'R', 'D': 'D', 'J': 'L', 'U': 'C', 'Q': 'U', 'Z': 'M', 'V': 'W',
'O': 'F', 'X': 'G', 'H': 'Y', 'N': 'P', 'Y': 'B', 'S': 'V'}
Initial Decryption Attempt:
AUNHNLNTEOOELEHAINSECMTAHWOAHTEITUESHTLEWRECOAHIFRAOSECMUNOTAHGONIINSTDAOTEO
REDEESRINSTDAOTEORNUOASMTAHYSITASOEEITCHEEDADEELEIDRCIGOSRRNHTIEMDRASTIETTNP
NRECOLEWASTOEBDEEDOATEFEONHAIWAVEOIETHAILOASTI
```

In [44]:
```
substitution_map['K'] = 'A'
plaintext_guess = decrypt(ciphertext, substitution_map)
print("Updated Decryption Attempt:")
print(plaintext_guess)
```

```
Updated Decryption Attempt:
AUNHNLNTEOOELEHAINSECMTAHWOAHTEITUESHTLEWRECOAHIFRAOSECMUNOTAHGONIINSTDAOTEO
REDEESRINSTDAOTEORNUOASMTAHYSITASOEEITCHEEDADEELEIDRCIGOSRRNHTIEMDRASTIETTNP
NRECOLEWASTOEBDEEDOATEFEONHAIWAVEOIETHAILOASTI
```

In [45]:
```python
substitution_map['K'] = 'O'
plaintext_guess = decrypt(ciphertext, substitution_map)
print("Updated Decryption Attempt:")
print(plaintext_guess)
```

```
Updated Decryption Attempt:
OUNHNLNTEOOELEHOINSECMTOHWOOHTEITUESHTLEWRECOOHIFROOSECMUNOTOHGONIINSTDOOTEO
REDEESRINSTDOOTEORNUOOSMTOHYSITOSOEEITCHEEDODEELEIDRCIGOSRRNHTIEMDROSTIETTNP
NRECOLEWOSTOEBDEEDOOTEFEONHOIWOVEOIETHOILOOSTI
```

In [46]:
```python
import string

# Given ciphertext
ciphertext = (
    "KQEREJEBCPPCJCRKIEACUZBKRVPKRBCIBQCARBJCVFCUP"
    "KRIOFKPACUZQEPBKRXPEIIEABDKPBCPFCDCCAFIEABDKP"
    "BCPFEQPKAZBKRHAIBKAPCCIBURCCDKDCCJCIDFUIXPAFF"
    "ERBICZDFKABICBBENEFCUPJCVKABPCYDCCDPKBCOCPERK"
    "IVKSCPICBRKIJPKABI"
)

alphabet = string.ascii_uppercase

# Function to find modular inverse using Extended Euclidean Algorithm
def mod_inverse(a, m):
    def extended_gcd(a, b):
        if a == 0:
            return b, 0, 1
        gcd, x1, y1 = extended_gcd(b % a, a)
        x = y1 - (b // a) * x1
        y = x1
        return gcd, x, y

    gcd, x, _ = extended_gcd(a, m)
    if gcd != 1:
        return None  # Modular inverse does not exist
    return x % m

def solve_affine(x1, y1, x2, y2, m=26):
    # Equations: a * x1 + b = y1 mod m, a * x2 + b = y2 mod m
    # Subtract: a * (x1 - x2) = y1 - y2 mod m
    diff_x = (x1 - x2) % m
    diff_y = (y1 - y2) % m
    a_inv = mod_inverse(diff_x, m)

    if a_inv is None:
        return None, None
    a = (diff_y * a_inv) % m
    b = (y1 - a * x1) % m
    return a, b

def decrypt_affine(ciphertext, a_inv, b):
    decrypted = []
    for char in ciphertext:
        if char.isalpha():
            y = ord(char.upper()) - ord('A')
```

```
            x = (a_inv * (y - b)) % 26
            decrypted.append(alphabet[x].lower())
        else:
            decrypted.append(char)
    return ''.join(decrypted)

def encrypt_affine(plaintext, a, b):
    encrypted = ""
    for char in plaintext:
        if char.isalpha():
            # Convert to number (a=0, ..., z=25)
            x = ord(char.upper()) - ord('A')
            # Apply encryption: 23x + 6 mod 26
            y = (a * x + b) % 26
            encrypted += alphabet[y]
        else:
            encrypted += char
    return encrypted

cipher1, plain1 = 'C', 'e'
cipher2, plain2 = 'B', 't'

x1 = ord(plain1.upper()) - ord('A')
y1 = ord(cipher1.upper()) - ord('A')
x2 = ord(plain2.upper()) - ord('A')
y2 = ord(cipher2.upper()) - ord('A')

# Calculate a and b
a, b = solve_affine(x1, y1, x2, y2)

if a is None or b is None:
    print("No valid (a, b) found.")
else:
    a_inv = mod_inverse(a, 26)

    if a_inv is None:
        print(f"No modular inverse exists for a = {a}.")
    else:
        # Test specific mappings
        decrypted_c1 = decrypt_affine(cipher1, a_inv, b)
        decrypted_c2 = decrypt_affine(cipher2, a_inv, b)

        decrypted_text = decrypt_affine(ciphertext, a_inv, b)

        print(f"a = {a}")
        print(f"b = {b}")
        print(f"a_inv = {a_inv}")
        print("Testing specific mappings (assumption C -> e, B -> t):")
        e_num = 4  # e
        encrypted_e = encrypt_affine('e', a, b)
        print(f"e (4) encrypts to: {encrypted_e} (should be C=2, actual numb

        t_num = 19  # t
        encrypted_t = encrypt_affine('t', a, b)
        print(f"t (19) encrypts to: {encrypted_t} (should be B=1, actual num
```

```
        c_decrypted = decrypt_affine('C', a_inv, b)
        print(f"C (2) decrypts to: {c_decrypted} (should be e=4, actual numb

        b_decrypted = decrypt_affine('B', a_inv, b)
        print(f"B (1) decrypts to: {b_decrypted} (should be t=19, actual num

        print(f"\nDecrypting full ciphertext with d_k(y) = {a_inv}(y - {b})
        decrypted_text = decrypt_affine(ciphertext, a_inv, b)
        print(f"Decrypted text: {decrypted_text}")
```

```
a = 19
b = 4
a_inv = 11
Testing specific mappings (assumption C -> e, B -> t):
e (4) encrypts to: C (should be C=2, actual number: 2)
t (19) encrypts to: B (should be B=1, actual number: 1)
C (2) decrypts to: e (should be e=4, actual number: 4)
B (1) decrypts to: t (should be t=19, actual number: 19)

Decrypting full ciphertext with d_k(y) = 11(y - 4) mod 26:
Decrypted text: ocanadaterredenosaieuxtonfrontestceintdefleuronsglorieuxcart
onbrassaitporterlepeeilsaitporterlacroixtonhistoireestuneepopeedesplusbrilla
ntsexploitsettavaleurdefoitrempeeprotegeranosfoyersetnosdroits
```

Affine Ciphe

Ciphertext and Plaintext `KQEREJEBCPPCJCRKIEACUZBKRVPKRBCIBQCARBJCVFCUP` `KRIOFKPACUZQEPBKRXPEIIEABDKPBCPFCDCCAFIEABDKP` `BCPFEQPKAZBKRHAIBKAPCCIBURCCDKDCCJCIDFUIXPAFF` `ERBICZDFKABICBBENEFCUPJCVKABPCYDCCDPKBCOCPERK IVKSCPICBRKIJPKABI`

"ocanadaterredenosaieuxtonfrontestceintdefleur onsglorieuxcartonbrassaitporterlepeeilsaitpor terlacroixtonhistoireestuneepopeedesplusbrill antsexploitsettavaleurdefoitrempeeprotegerano sfoyersetnosdroits"

Frequency Analysis

From the code we get the frequency analisis

- **Ciphertext Frequencies**: A:13, B:21, C:32, D:9, E:13, F:10, H:1, I:16, J:6, K:20, P:20, Q:4, R:12, S:1, U:6, V:4, X:2, Y:1, Z:4

We order them as following:

- **Rank**: C(1), B(2), K(3), P(4), I(5), A(6), E(7), R(8), F(9), D(10), J(11), U(12)

Decryption Steps

1. **Frequency Analysis**:
   First we assume, C (32) maps to *E*, the most common letter in english and B

(21) maps to *T*, the second most common. We also tested K, the third mwost repeated character in the cyphertext, to *A* and *O*

2. **Equations**:
   The Affine Cipher uses the formula (e(x) = ax + b % 26)

   - C (2) → e (4): (4a + b = 2 % 26)
   - B (1) → t (19): (19a + b = 1 % 26) We assume this equations based on the hypothesis for C and B mapping we previously made.
   - Subtract: (15a = 17 % 26) → (15a = -9 % 26).
3. **Solve for (a) and (b)**:

   - (15a = 17): Solve using modular inverse of 15 (7, since (15 · 7 = 105 = 1 % 26)).
   - (a = 17 · 7 = 119 = 19 % 26), so (a = 19).
   - Substitute: (4 · 19 + b = 76 + b = 2 % 26), (b = 2 + 26 - 76 = 4).
   - Check: (19 · 1 + 4 = 23 = 19 % 26) (t), correct. -Calculate inverse of a
4. **Decryption**:

   - Encryption: (e(x) = 19x + 4 % 26)
   - Inverse: (d(y) = 11(y - 4) % 26) (since (19^{-1} = 11 % 26), verified by (19 · 11 = 209 = 1 % 26)).

In [47]:
```
import string
from collections import Counter

def index_of_coincidence(text):
    text = ''.join(filter(str.isalpha, text)).lower()

    if len(text) < 2:
        return 0

    frequency = Counter(text)
    n = len(text)
    ic = sum(f * (f - 1) for f in frequency.values()) / (n * (n - 1))
    return ic

def find_possible_key_lengths(ciphertext):
    key_length_ic_values = {}

    for key_length in range(1, 21):
        shifted_text = ['' for _ in range(key_length)]

        for i, char in enumerate(ciphertext):
            if char.isalpha():
                shifted_text[i % key_length] += char

        ic_values = [index_of_coincidence(segment) for segment in shifted_te
        avg_ic = sum(ic_values) / len(ic_values) if ic_values else 0

        key_length_ic_values[key_length] = avg_ic
```

```python
    return key_length_ic_values

ciphertext = """BNVSNSIHQCEELSSKKYERIFJKXUMBGYKAMQLJTYAVFBKVT
                DVBPVVRJYYLAOKYMPQSCGDLFSRLLPROYGESEBUUALRWXM
                MASAZLGLEDFJBZAVVPXWICGJXASCBYEHOSNMULKCEAHTQ
                OKMFLEBKFXLRRFDTZXCIWBJSICBGAWDVYDHAVFJXZIBKC
                GJIWEAHTTOEWTUHKRQVVRGZBXYIREMMASCSPBNLHJMBLR
                FFJELHWEYLWISTFVVYFJCMHYUYRUFSFMGESIGRLWALSWM
                NUHSIMYYITCCQPZSICEHBCCMZFEGVJYOCDEMMPGHVAAUM
                ELCMOEHVLTIPSUYILVGFLMVWDVYDBTHFRAYISYSGKVSUU
                HYHGGCKTMBLRX"""

ciphertext = ciphertext.replace("\n", "").replace(" ", "")
key_length_ic = find_possible_key_lengths(ciphertext)

print("Key Length | Average IC")
print("------------------------")
for key_length, ic in key_length_ic.items():
    print(f"{key_length:^10} | {ic:.5f}")

probable_lengths = [k for k, v in key_length_ic.items() if abs(v - 0.06) < 0

print("\nMost probable key lengths:", probable_lengths)
```

```
Key Length | Average IC
------------------------
    1      | 0.04138
    2      | 0.04527
    3      | 0.04684
    4      | 0.04856
    5      | 0.04151
    6      | 0.06062
    7      | 0.04183
    8      | 0.05127
    9      | 0.04445
    10     | 0.04627
    11     | 0.04241
    12     | 0.06475
    13     | 0.04079
    14     | 0.04657
    15     | 0.04697
    16     | 0.04718
    17     | 0.03965
    18     | 0.05500
    19     | 0.04109
    20     | 0.04843

Most probable key lengths: [6, 8, 12, 18]
```

In [48]:

```python
from collections import Counter

def calculate_relative_shifts(ciphertext, key_length):
    shifted_text = ['' for _ in range(key_length)]

    for i, char in enumerate(ciphertext):
```

```python
        if char.isalpha():
            shifted_text[i % key_length] += char

    most_common_letters = []
    for segment in shifted_text:
        if segment:
            most_common_letter, _ = Counter(segment).most_common(1)[0]
            most_common_letters.append(most_common_letter)
        else:
            most_common_letters.append(None)

    shifts = []
    for letter in most_common_letters:
        if letter:
            shift = (ord(letter.lower()) - ord('e')) % 26
            shifts.append(shift)
        else:
            shifts.append(None)

    return shifts



if probable_lengths:
    chosen_key_length = probable_lengths[0]
    shifts = calculate_relative_shifts(ciphertext, chosen_key_length)

    print(f"\nRelative shifts for key length {chosen_key_length}: {shifts}")
else:
    print("\nNo clear key length found.")

def deduce_key_from_shifts(shifts):
    return "".join(alphabet[s] for s in shifts)

key = deduce_key_from_shifts(key_shifts)
plaintext = vigenere_decrypt(ciphertext, key)

print(f"Possible Key: {key}")
print(f"Decrypted Text Preview: {plaintext[:100]}")
```

```
Relative shifts for key length 6: [19, 7, 4, 14, 0, 20]
Possible Key: CRYPTO
Decrypted Text Preview: zwxduegqsnlqjbuvrkcakqqwvdomnkijobsvrhcgmnievocnnexc
qkwuczrkkysdjsbuhdyxjytzfscbgmbgyutheykjulgxeugo
```

Other Cipher

Ciphertext and Plaintext

`KQEREJEBCPPCJCRKIEACUZBKRVPKRBCIBQCARBJCVFCUP`

`KRIOFKPACUZQEPBKRXPEIIEABDKPBCPFCDCCAFIEABDKP`

`BCPFEQPKAZBKRHAIBKAPCCIBURCCDKDCCJCIDFUIXPAFF`

`ERBICZDFKABICBBENEFCUPJCVKABPCYDCCDPKBCOCPERK IVKSCPICBRKIJPKABI`

"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."

Decryption

1. **Index of Coincidence (Ic)**:
   We assumed there was some type of key, we applied the Ic test and choose the top results for key length: Most probable key lengths: [6, 8, 12, 18]. We started with an assumed key of length 6.

2. **Relative Shifts**:
   With the key, we split the ciphertext into 6 columns because we assume the key length is 6. We calculated all of the shifts until one was consistent. Relative shifts for key length 6: [19, 7, 4, 14, 0, 20]

3. **Key Determination**:
   After getting the shifts we mapped the respective character to get the key. The key was THEORY, in this case, then we use it to decrypt the message.

4)

Affine Cipher

Let $P = C = \mathbb{Z}_{26}$ and

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

Let $k = (a, b) \in K$,

$$e_k(x) = (ax + b) \bmod 26$$

and $d_k(y) = a^{-1}(y - b) \bmod 26$ for $x, y \in \mathbb{Z}_{26}$

With this in mind, we know there are 312 keys because $12 \times 26 = 312$, (12 coprimes y 26 % 26)

If we choose one key, encryption for one cyphertext for all posibilities we have:

$$Pr[y] = \sum_{k \in K} Pr[k] \, Pr[d_k(y)] = \frac{\cancel{12}^{1}}{\cancel{312}_{26}} Pr[a] + \cdots + \frac{\cancel{12}^{1}}{\cancel{312}_{26}} Pr[z]$$

$$= \frac{1}{26} Pr[a] + \cdots + \frac{1}{26} Pr[z]$$

Appliying Baye's Theorem for probability:

$$Pr[x \mid y] = \frac{Pr[x] \, Pr[y \mid x]}{Pr[y]} = \frac{Pr[x] \, \cancel{\frac{1}{26}}^{1}}{\cancel{\frac{1}{26}}_{1}}$$

$$= Pr[x]$$

$\therefore$ We have 12 keys that encrypt x to y. so we proof that we achieve perfect secrecy, since the probability of each plaintext mapping to a specific cypher text is the same. $\blacksquare$

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

Definition of Perfect Secrecy

A cryptosystem has perfect secrecy if, for any message $p \in P$ and any ciphertext $c \in C$, the conditional probability that p is the original message given that c is observed is equal to the prior probability of p:

$$P(P = p|C = c) = P(P = p)$$

This means that observing a ciphertext provides no additional information about the original message.

Condition $|K| = |C| = |P|$

If the number of keys is equal to the number of messages and the number of ciphertexts ($|K| = |C| = |P|$), then each key defines a bijection between messages and ciphertexts. This implies that for each message p, there exists a unique key k that transforms it into any given ciphertext c. Since the keys are chosen uniformly at random, each message has the same probability of being encrypted into any possible ciphertext.

Computing the Probability of a Ciphertext

The probability that a specific ciphertext c occurs is the sum of the probabilities of all message-key combinations that produce it:

$$P(C = c) = \sum_{p \in P} P(C = c|P = p)P(P = p)$$

Since each message has a unique key that transforms it into every ciphertext, and since the keys are chosen uniformly, the ciphertexts are uniformly distributed:

$$P(C = c) = \frac{1}{|C|}$$

That is, every ciphertext occurs with the same probability.

Numerical Example

Suppose we have $|P| = |C| = |K| = 3$:

Message p	**Key k**	**Ciphertext c**
A	1	X

Message p	Key k	Ciphertext c
A	2	Y
A	3	Z
B	1	Y
B	2	Z
B	3	X
C	1	Z
C	2	X
C	3	Y

If each key has probability $\frac{1}{3}$, then the probability of observing any ciphertext is:

$$P(C = X) = P(A, k_1) + P(B, k_3) + P(C, k_2) = \frac{1}{3} + \frac{1}{3} + \frac{1}{3} = \frac{1}{3}$$

Similarly:

$$P(C = Y) = \frac{1}{3}, \quad P(C = Z) = \frac{1}{3}$$

which confirms that all ciphertexts are equally probable.

Conclusion

The condition $|K| = |C| = |P|$ ensures that each message has a unique key that transforms it into every possible ciphertext. Since the keys are chosen uniformly at random, the ciphertext distribution is also uniform.

This satisfies the property of perfect secrecy and guarantees that every ciphertext is equally probable, preventing an attacker from gaining information about the original message.

In [ ]:

6- Use the extended Euclidean Algorithm to compute the following multiplicatives inverses:

The result is find the modular inverse of number a with respect to m, meaning we need to find x such that:

$a \cdot x = 1 \mod m$

For express 1 as a linear combination of a and m, we use Euclidean Extend Algorithm

1- $17^{-1} \mod 101$

gcd (101, 17)   We divide $101 \div 17 = 5$

$$\begin{array}{r|l} 101 & 17 \\ \underline{-85} & 5 \\ 16 & \end{array}$$

$101 - (5 \times 107) = 101 - 85 = 16$

$17 \div 16 = 1$, $17 - (1 \times 16) = 1$

$$\begin{array}{r|l} 17 & 16 \\ \underline{-16} & 1 \\ 1 & \end{array}$$

$16 \div 1 = 16$, $16 - (16 \times 1) = 0$

$1 = 17 - 1(16)$

Since $16 = 101 - 5(17)$, substitute:

$1 = 17 - 1(101 - 5(17))$

$1 = 17 - 1 \cdot 101 + 5 \cdot 17$

$1 = 6 \cdot 17 - 1 \cdot 101$

$x = 6$

$17^{-1} = 6 \mod 101$

$17^{-1} \mod 101 = 6$

2- $357^{-1} \bmod 1234$

$\gcd(1234, 357)$

We divide : $1234 \div 357$

$$\begin{array}{r|l} 1234 & 357 \\ -1071 & 3 \\ \hline 163 & \end{array}$$

$1234 - (3 \times 357) = 1234 - 1071 = 163$

$$\begin{array}{r|l} 357 & 163 \\ -326 & 2 \\ \hline 31 & \end{array}$$

$357 - (2 \times 163) = 31$

$$\begin{array}{r|l} 163 & 31 \\ -155 & 5 \\ \hline 8 & \end{array}$$

$163 - (5 \times 31) = 8$

$$\begin{array}{r|l} 31 & 8 \\ -24 & 3 \\ \hline 7 & \end{array}$$

$31 - (8 \times 3) = 7$

$$\begin{array}{r|l} 8 & 7 \\ -7 & 1 \\ \hline 1 & \end{array}$$

$8 - (1 \times 7) = 7$

$$\begin{array}{r|l} 7 & 1 \\ 0 & 7 \end{array}$$

$7 - (7 \times 1) = 0$

$1 = 8 - 1(7)$

Since $7 = 31 - 3(8)$, substitute:

$1 = 8 - 1(31 - 3(8)) = 4(8) - 1(31)$

Since $8 = 163 - 5(31)$:

$1 = 4(163 - 5(31)) - 1(31) = 4(163) - 21(31)$

Since $163 = 1234 - 3(357)$:

$1 = 46(1234 - 3(357)) - 21(357)$

$1 = 46(1234 - 159(357))$

$x = 1075$ $\qquad$ $357^{-1} \bmod 1234 = 1075$

3: $3125^{-1} \bmod 4987$

$\gcd(4987, 3125)$

$4987 \div 3125 = 3$

$$\begin{array}{r|l} 4987 & 3125 \\ -4375 & 3 \\ \hline 612 & \end{array}$$

$3125 \div 612 = 5$, $3125 - (5 \times 612) = 65$

$$\begin{array}{r|l} 3125 & 612 \\ -3060 & 5 \\ \hline 65 & \end{array}$$

$612 \div 65 = 9$, $612 - (9 \times 65) = 27$

$$\begin{array}{r|l} 612 & 65 \\ -585 & 9 \\ \hline 27 & \end{array}$$

$65 \div 27 = 2$, $65 - (2 \times 27) = 11$

$$\begin{array}{r|l} 65 & 27 \\ -54 & 2 \\ \hline 11 & \end{array}$$

$27 \div 11 = 2$, $27 - (2 \times 11) = 5$

$$\begin{array}{r|l} 27 & 11 \\ -22 & 2 \\ \hline 5 & \end{array}$$

$11 \div 5 = 2$, $11 - (2 \times 5) = 1$

$$\begin{array}{r|l} 11 & 5 \\ -1 & 2 \\ \hline 0 & \end{array}$$

$5 - 1 = 0$, $5 - (5 \times 1) = 0$

Since: $1 = 11 - 2(5)$, Substitute:

$$1 = 11 - 2(27 - 2(11)) = 5(11) - 2(27)$$

Substitute $11 = 65 - 2(27)$:

$$1 = 5(65 - 2(27)) - 2(27) = 5(65) - 12(27)$$

Substitute $27 = 612 - 9(65)$:

$$1 = 5(65) - 12(612 - 9(65)) = 113(65) - 12(612)$$

Substitute $65 = 3125 - 5(612)$:

$$1 = 113(3125 - 5(612)) - 12(612) = 113(3125) - 577(612)$$

Substitute $612 = 9987 - 3(3125)$.

$$1 = 113(3125) - 577(9987 - 3(3125))$$

$$1 = 1844(3125) - 577(9987)$$

$$3125^{-1} = 18844 \bmod 9987$$

$$3125^{-1} \bmod 9987 = 18844$$