**CNS Lab**

**Assignment 2**

**Title : To Study 1. Crypt Analysis And 2. Columnar transposition cipher**

**Information:**

Encryption:

Step1:Select a Keyword or Key Phrase: Choose a unique keyword or key phrase. The distinct characters within this keyword determine the arrangement of columns in the transposition grid.

Step2:Grid-Based Message Entry: Write the plaintext message row by row into a grid, using the keyword to guide the order of columns. Place the characters of the message in the columns based on the keyword's character sequence.

Step3:Columnar Reading: To generate the encrypted message, read the characters column by column, starting with the column associated with the first character of the keyword and moving on accordingly. This forms the ciphertext.

Decryption:

Step1:Keyword Utilization: Utilize the same keyword to determine the column order in the transposition grid. The keyword's unique character sequence is key to reconstructing the grid.

Step2:Grid-Based Entry: Fill the grid column by column with the encrypted message. Use the keyword's character sequence to guide the placement of characters within the columns.

Step3:Row-Based Retrieval: To recover the original plaintext message, read the characters row by row from the grid. Begin with the row containing the first character of the keyword and proceed accordingly.

**Code of Crypt Analysis:**

import nltk

from nltk.corpus import words as nltk\_words

# nltk.download("words")

def is\_valid\_word(word):

    return word.lower() in nltk\_words.words()

def decrypt\_with\_meaning(encrypted\_text):

    best\_decrypted\_text = ""

    best\_word\_count = 0

    for shift in range(26):

        decrypted\_text = ""

        for char in encrypted\_text:

            if char.isalpha():

                if char.islower():

                    decrypted\_text += chr((ord(char) - ord('a') - shift) % 26 + ord('a'))

                elif char.isupper():

                    decrypted\_text += chr((ord(char) - ord('A') - shift) % 26 + ord('A'))

            else:

                decrypted\_text += char

        decrypted\_words = decrypted\_text.split()

        valid\_word\_count = sum(1 for word in decrypted\_words if is\_valid\_word(word))

        if valid\_word\_count > best\_word\_count:

            best\_word\_count = valid\_word\_count

            best\_decrypted\_text = decrypted\_text

    return best\_decrypted\_text

# Input encrypted text

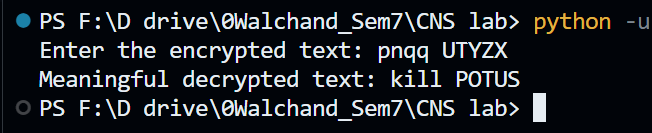
encrypted\_text = input("Enter the encrypted text: ")

# Decrypt and find the most meaningful result

meaningful\_decrypted\_text = decrypt\_with\_meaning(encrypted\_text)

print("Meaningful decrypted text:", meaningful\_decrypted\_text)

**Output:**

****

**Code for Columnar Transposition Cipher :**

def encrypt(text, key):

    # Remove spaces and convert text to uppercase

    text = text.replace(" ", "").upper()

    # Calculate the number of rows needed

    rows = len(text) // len(key)

    if len(text) % len(key) != 0:

        rows += 1

    # Pad the text with extra characters to make it evenly divisible

    padding = rows \* len(key) - len(text)

    text += "X" \* padding

    # Create a matrix for the plaintext

    matrix = [list(text[i:i+len(key)]) for i in range(0, len(text), len(key))]

    # Create a list to store the encrypted columns

    encrypted\_columns = [''] \* len(key)

    # Rearrange columns according to the key

    for idx in range(len(key)):

        col\_idx = key.index(idx + 1)

        encrypted\_columns[col\_idx] = ''.join(row[idx] for row in matrix)

    # Generate the ciphertext by reading rows

    ciphertext = ''.join(encrypted\_columns)

    return ciphertext

def decrypt(ciphertext, key):

    # Calculate the number of rows needed

    rows = len(ciphertext) // len(key)

    # Create a matrix for the ciphertext

    matrix = [list(ciphertext[i:i+rows]) for i in range(0, len(ciphertext), rows)]

    # Create a list to store the decrypted rows

    decrypted\_rows = [''] \* rows

    # Rearrange rows according to the key

    for idx in range(len(key)):

        row\_idx = key.index(idx + 1)

        for i in range(rows):

            decrypted\_rows[i] += matrix[row\_idx][i]

    # Generate the plaintext by reading rows

    plaintext = ''.join(decrypted\_rows)

    return plaintext

# Example usage

plaintext = "HELLO Shree"

key = [2, 1, 4, 3]  # Example key

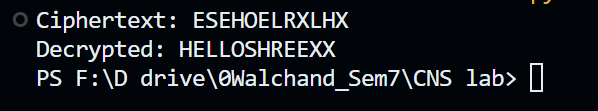
ciphertext = encrypt(plaintext, key)

print("Ciphertext:", ciphertext)

decrypted\_text = decrypt(ciphertext, key)

print("Decrypted:", decrypted\_text)

**Output:**

****

**Limitations:**

1. Key Management: Securely managing and distributing the encryption key can be challenging, especially when dealing with a large number of users or when the key needs to be changed frequently. If the key is compromised, all encrypted messages become vulnerable.

2. Lack of Security: Columnar ciphers, especially simple implementations, provide limited security. They are susceptible to known-plaintext attacks and frequency analysis, as they do not hide the frequency distribution of letters in the plaintext.

3. Small Key Space: The key space for columnar ciphers is relatively small compared to modern encryption algorithms. This makes them vulnerable to brute-force attacks, where an attacker can try all possible key permutations to decrypt the message.

4. Dependency on Key Length: The security of a columnar cipher is heavily dependent on the length and complexity of the encryption key. Short or easily guessable keys can be exploited by attackers.

5. Known Structure: Columnar ciphers have a known structure, which means that the length of the ciphertext often reveals information about the length of the plaintext. This can be used by attackers to make educated guesses about the plaintext.

6. No Authentication or Integrity Protection: Columnar ciphers only provide confidentiality; they do not offer any form of authentication or data integrity protection. This means that an attacker can tamper with the ciphertext without detection.

7. Inefficiency for Large Data: Columnar ciphers are not efficient for encrypting large volumes of data. The process of rearranging the characters in columns can be computationally expensive, especially for long messages.

8. Limited Applicability: Columnar ciphers are designed for text-based data and may not work well for other types of data or data with different character sets.

9. Complexity for Users: Implementing and using a columnar cipher can be more complex for users compared to some modern encryption tools, as it requires a good understanding of how to choose and manage keys and how to apply the cipher correctly.