Pabna University of Science and Technology

Faculty of Engineering and Technology

Department of Information and Communication Engineering

Lab Report

Course title : Cryptography and Computer Security Sessional

Course code : ICE-4108

Submitted By:

Name: Dibakor Das

Roll : 200623

Session : 2019-2020

4th Year 1st Semester

Department of ICE, PUST.

Submitted To :

Md. Anwar Hossain

Professor

Department of Information and Communication Engineering, PUST.

.....

Signature

INDEX

SL	Name of the Experiment	Page No.
1.	Write a program to implement encryption and decryption using Caesar cipher.	
2.	Write a program to implement encryption and decryption using Mono-Alphabetic cipher.	
3.	Write a program to implement encryption and decryption using Brute force attack cipher.	
4.	Write a program to implement encryption and decryption using Hill cipher.	
5.	Write a program to implement encryption using Playfair cipher.	
6.	Write a program to implement decryption using Playfair cipher.	
7.	Write a program to implement encryption using Poly-Alphabetic cipher.	
8.	Write a program to implement decryption using Poly-Alphabetic cipher.	
9.	Write a program to implement encryption using Vernam cipher.	
10.	Write a program to implement decryption using Vernam cipher.	

Experiment no. 01 :

Experiment name : Write a program to implement encryption and decryption using Caesar cipher.

Objectives:

Encryption: The goal of encryption is to convert plaintext into ciphertext using the Caesar cipher. This helps ensure the confidentiality of the message.

Decryption: The goal of decryption is to reverse the encryption process and recover the original plaintext

Theory:

The Caesar cipher is to shift each letter of the plaintext by a certain number (known as the **key**) to produce the ciphertext. The shift wraps around the end of the alphabet, so after 'Z' comes 'A' (for uppercase letters) or after 'z' comes 'a' (for lowercase letters). This shift ensures that the plaintext remains confidential, as long as the key is kept secret.

- **Plaintext**: The original, unencrypted message.
- **Ciphertext**: The encrypted message, obtained after applying the Caesar cipher.
- **Key**: The number of positions by which each letter of the plaintext is shifted.

For example, with a key of 3, 'A' becomes 'D', 'B' becomes 'E', etc.

The encryption and decryption operations are simple and symmetrical, meaning the same key is used for both processes.

How the Caesar Cipher Works:

Here's a more formal explanation of how the Caesar cipher works for both **encryption** and **decryption**:

Encryption Process:

1. **Step 1**: Take the plaintext, which consists of a sequence of letters.
2. **Step 2**: For each letter in the plaintext:
    - If the letter is an alphabetic character, shift it by the key value (for example, 3).

    - If the letter is not alphabetic (such as a space, comma, or period), leave it unchanged.
3. **Step 3**: After shifting the letters, concatenate them to form the ciphertext.

For example, with the **plaintext** "HELLO" and a **key** of 3:

- 'H' becomes 'K'
- 'E' becomes 'H'df
- 'L' becomes 'O'
- 'L' becomes 'O'
- 'O' becomes 'R'

The resulting **ciphertext** would be "KHOOR".

Decryption Process:

1. **Step 1**: Take the ciphertext, which has been encrypted using the Caesar cipher.
2. **Step 2**: For each letter in the ciphertext:
    - If the letter is an alphabetic character, shift it **back** by the key value (in the opposite direction of encryption).
    - If the letter is not alphabetic, leave it unchanged.
3. **Step 3**: After shifting the letters back, concatenate them to form the decrypted plaintext.

For example, with the **ciphertext** "KHOOR" and a **key** of 3:

- 'K' becomes 'H'
- 'H' becomes 'E'
- 'O' becomes 'L'
- 'O' becomes 'L'
- 'R' becomes 'O'

The resulting **decrypted plaintext** is "HELLO".

Algorithm:

Encryption:

1. **Input**: A plaintext string and a key (shift value).
2. For each character in the plaintext:
    - If the character is a letter (either uppercase or lowercase):
        - Shift it forward by the key positions.
        - If the shift goes beyond 'Z' (for uppercase) or 'z' (for lowercase), wrap around to the start of the alphabet.
    - If the character is not a letter (e.g., spaces or punctuation), leave it unchanged.
3. **Output**: The resulting ciphertext.

Decryption :

1. **Input**: A ciphertext string and a key (shift value).
2. For each character in the ciphertext:
    - If the character is a letter (either uppercase or lowercase):
        - Shift it backward by the key positions.
        - If the shift goes before 'A' (for uppercase) or 'a' (for lowercase), wrap around to the end of the alphabet.
    - If the character is not a letter (e.g., spaces or punctuation), leave it unchanged.
3. **Output**: The resulting plaintext.

Code (Python):

```
def caesar_cipher(text, key, mode='encrypt'):
    result = ""
    for char in text:
        if char.isalpha():
            shift = key if mode == 'encrypt' else -key
            base = 65 if char.isupper() else 97
            result += chr((ord(char) - base + shift) % 26 + base)
        else:
            result += char
```

```
    return result

# Example usage
plaintext = "ICE DEPARTMENT"
key = 3

ciphertext = caesar_cipher(plaintext, key, mode='encrypt')
print(f"Ciphertext: {ciphertext}")  # Encrypted text

decrypted_text = caesar_cipher(ciphertext, key, mode='decrypt')
print(f"Decrypted Text: {decrypted_text}")  # Decrypted back to original
plaintext
```

Input:

plaintext = "ICE DEPARTMENT"

key = 3

Output:

Ciphertext: LFH GHSDUWPHQW

Decrypted Text: ICE DEPARTMENT

Experiment no. 02 :

Experiment name: Write a program to implement encryption and decryption using Mono-Alphabetic cipher.

Objectives:

Encryption: The goal is to transform the plaintext into ciphertext by replacing each letter with another letter according to a fixed substitution rule (key).

Decryption: The goal is to reverse the encryption process, transforming the ciphertext back into the original plaintext using the inverse of the substitution key.

Theory :

In the Mono-Alphabetic Cipher, a key is created by randomly permuting the 26 letters of the alphabet. Each letter in the plaintext is then replaced by the letter in the same position in the key. This means each letter is substituted by another letter, and no letter is substituted with itself, ensuring one-to-one correspondence.

Encryption Process:

1. **Create a Key**: Choose a random permutation of the alphabet (a 1-to-1 substitution rule).
2. **Substitute Letters**: For each letter in the plaintext, find the corresponding letter in the key and replace it with that letter.
    - For example, if 'A' is replaced by 'Q', 'B' by 'M', and so on, then the plaintext "ABC" might turn into "QMC" based on the key.

Decryption Process:

1. **Create an Inverse Key**: Reverse the key. For instance, if 'A' is substituted by 'Q', the inverse key will ensure that 'Q' is replaced by 'A'.
2. **Substitute Letters**: For each letter in the ciphertext, use the inverse key to find and substitute it with the original letter.

Example:

Let's consider the following example:

- **Plaintext**: "HELLO"

- **Key**: QWERTYUIOPASDFGHJKLZXCVBNM (Random permutation of the alphabet)
    - 'A' → 'Q', 'B' → 'W', 'C' → 'E', ..., 'Z' → 'M'
- **Ciphertext**: The encrypted message after substituting each letter based on the key.

Example Encryption:

- Plaintext: **"HELLO"**
- Key: QWERTYUIOPASDFGHJKLZXCVBNM

The substitution would look like this:

- H → T
- E → O
- L → P
- L → P
- O → B

The resulting **ciphertext** would be: "TOPPB"

Example Decryption:

- Ciphertext: **"TOPPB"**
- Inverse Key: QWERTYUIOPASDFGHJKLZXCVBNM (inverse mapping of the key, where 'Q' → 'A', 'W' → 'B', etc.)

The decryption process reverses the substitution:

- T → H
- O → E
- P → L
- P → L
- B → O

The resulting **decrypted plaintext** would be: "HELLO".

Algorithm:

Encryption :

1. **Input**: A plaintext string and a substitution key (a permutation of the alphabet).
2. For each letter in the plaintext:
    - If it's a letter, replace it with the corresponding letter from the key.
    - Non-alphabetic characters (spaces, punctuation, etc.) remain unchanged.
3. **Output**: The ciphertext.

Decryption :

1. **Input**: A ciphertext string and a substitution key (same as used in encryption).
2. For each letter in the ciphertext:
    - If it's a letter, replace it with the corresponding letter from the inverse of the key.
    - Non-alphabetic characters remain unchanged.
3. **Output**: The decrypted plaintext.

Code (Python):

```
import string

# Simple substitution key (manual mapping)
key = 'QWERTYUIOPASDFGHJKLZXCVBNM'
alphabet = string.ascii_uppercase

# Function to encrypt or decrypt text
def mono_alphabetic(text, key, mode='encrypt'):
    key_dict = {alphabet[i]: key[i] for i in range(26)}
    inverse_key_dict = {key[i]: alphabet[i] for i in range(26)}
```

```
    return ''.join(inverse_key_dict[char] if mode == 'decrypt' else
key_dict[char] if char.isalpha() else char
                   for char in text.upper())

# Example usage
plaintext = "DIBAKOR"
ciphertext = mono_alphabetic(plaintext, key, mode='encrypt')
decrypted_text = mono_alphabetic(ciphertext, key, mode='decrypt')

print(f"Ciphertext: {ciphertext}")  # Encrypted message
print(f"Decrypted Text: {decrypted_text}")  # Decrypted back to original
plaintext
```

Input:

Plaintext: DIBAKOR

Output:

Ciphertext: ROWQAGK

Decrypted Text: DIBAKOR

Experiment no. 03 :

Experiment name: Write a program to implement encryption and decryption using Brute force attack cipher.

Objectives:

Encryption: The purpose of encryption is to transform the plaintext into ciphertext using a specific key and algorithm

Brute Force Decryption: The objective of a brute force attack is to try every possible key and check the resulting plaintext until a meaningful result is found.

Theory :

A brute force attack on a cipher refers to systematically trying every possible key until the correct one is found. The main goal of a brute force attack is to decrypt a message without knowing the encryption key. This attack is commonly used on weak ciphers like the Caesar cipher, where there are a limited number of possible keys.

In this context, a brute force attack on a cipher such as the **Caesar cipher** would involve trying every possible shift of the alphabet until the plaintext is recovered.

Algorithm:

Encryption :

1. Choose a shift value k for the Caesar cipher.
2. For each letter in the plaintext:
    - Shift the letter by k positions in the alphabet.
3. If the letter is not alphabetic, leave it unchanged.
4. Return the resulting ciphertext.

Brute Force Attack Algorithm:

1. For each possible key (shift value from 1 to 25):
    - Apply the Caesar cipher decryption for the current shift.
    - Check if the resulting text makes sense (often, this can be done by looking for a meaningful word or phrase in the output).

2. Once a correct plaintext is found, stop the attack and return the result.

Code (Python) :

```
import string

# Function to encrypt text using Caesar Cipher
def caesar_encrypt(plaintext, shift):
    alphabet = string.ascii_uppercase
    return ''.join([alphabet[(alphabet.index(char) + shift) % 26] if char.isalpha() 
else char for char in plaintext.upper()])

# Function to decrypt text using Caesar Cipher (with brute force)
def brute_force_attack(ciphertext):
    alphabet = string.ascii_uppercase
    print(f"Ciphertext: {ciphertext}")
    for shift in range(1, 26):
        decrypted_text = ''.join([alphabet[(alphabet.index(char) - shift) % 26] if 
char.isalpha() else char for char in ciphertext.upper()])
        print(f"Shift {shift}: {decrypted_text}")

# Example usage
plaintext = "MY UNIVERSITY"
ciphertext = caesar_encrypt(plaintext, 4)  # Example encryption with shift of 4
print(f"Encrypted Text (Ciphertext): {ciphertext}")
print("\nBrute Force Attack Decryption:\n")
brute_force_attack(ciphertext)
```

Input :

plaintext = "MY UNIVERSITY"

Output:

Encrypted Text (Ciphertext): QC YRMZIVWMXC

Brute Force Attack Decryption:

Ciphertext: QC YRMZIVWMXC

Shift 1: PB XQLYHUVLWB

Shift 2: OA WPKXGTUKVA

Shift 3: NZ VOJWFSTJUZ

Shift 4: MY UNIVERSITY

Experiment no. 04:

Experiment name: Write a program to implement encryption and decryption using Hill cipher.

Objectives:

1. Understand the principles of the Hill Cipher, a polygraphic substitution cipher based on linear algebra.
2. Learn how to implement encryption and decryption using matrix multiplication and modular arithmetic.
3. Explore Python programming concepts to construct, encrypt, and decrypt text with the Hill Cipher algorithm.

Theory :

The Hill Cipher is a polygraphic substitution cipher that encrypts multiple characters simultaneously using linear transformations. It was invented by Lester S. Hill in 1929 and uses matrix multiplication in linear algebra.

In Hill Cipher:

- A key matrix (square matrix) of size n×n is used.
- A plaintext is divided into groups of n characters (converted to numeric values).
- Each group is represented as a column vector, and the key matrix multiplies this vector to produce the ciphertext vector.
- Modular arithmetic (mod 26) is applied to keep values within the 0-25 range for each alphabet letter.

Mathematical Formulation

Given:

- P: Plaintext column vector.
- K: Key matrix.
- C: Ciphertext vector.

Encryption:

$$C = (K \cdot P) \bmod 26$$

Decryption (only if K invertible mod 26):

$$\mathbf{P = (K^{-1}. C) \mod 26}$$

Example :

Suppose we want to encrypt "ACT" using a 2x2 key matrix K as follows:

$$K = \begin{pmatrix} 3 & 3 \\ 2 & 5 \end{pmatrix}$$

1. Convert each letter to a number (A=0, B=1, ..., Z=25):
   - Plaintext "ACT" → Vector [0, 2, 19]
2. Divide into pairs: [0, 2] and [19].
3. Encrypt each pair by multiplying it with K and applying mod 26.

Algorithm :

1. **Convert Plaintext to Numeric**: Map A-Z to 0-25.
2. **Divide Text into Chunks**: Split plaintext into groups of size n (same as key matrix).
3. **Matrix Multiplication**: Multiply each plaintext vector by the key matrix.
4. **Apply Modulo 26**: Wrap results within 0-25 to map to letters.
5. **Repeat for Decryption**: Use the inverse key matrix for decryption.

Code (Python) :

```
import numpy as np

def mod_inverse(matrix, mod=26):
    det_inv = pow(int(round(np.linalg.det(matrix))), -1, mod)
    return (det_inv * np.array([[matrix[1,1], -matrix[0,1]], [-matrix[1,0],
matrix[0,0]]]) % mod).astype(int)

def hill_cipher(text, key, mode='encrypt'):
    # Convert text to numbers (A=0, B=1, ..., Z=25)
    text = [(ord(c) - 65) for c in text.upper() if c.isalpha()]
    key = mod_inverse(key) if mode == 'decrypt' else key
```

```
    # Add padding if needed
    text += [0] * (len(text) % key.shape[0])

    # Encrypt or decrypt by multiplying key with text chunks
    result = [np.dot(key, text[i:i+key.shape[0]]) % 26 for i in range(0, len(text),
key.shape[0])]
    output_text = ''.join(chr(num + 65) for vec in result for num in vec)

    return output_text.rstrip('A') if mode == 'decrypt' else output_text

# Example usage
key = np.array([[3, 3], [2, 5]])
plaintext = "ACT"
ciphertext = hill_cipher(plaintext, key)
decrypted_text = hill_cipher(ciphertext, key, mode='decrypt')

print("Plaintext:", plaintext)
print("Ciphertext:", ciphertext)
print("Decrypted Text:", decrypted_text)
```

Input :

Plaintext: ACT

Key Matrix: $\begin{pmatrix} 3 & 3 \\ 2 & 5 \end{pmatrix}$

Output :

Ciphertext: GKFM

Decrypted Text: ACT

Experiment no. 05 :

Experiment name: Write a program to implement encryption using Playfair cipher.

Objectives :

1. To encrypt a plaintext message by breaking it into pairs of letters.
2. To use a 5x5 matrix (key square) to transform these letter pairs into ciphertext.
3. To ensure a more secure cipher compared to simple substitution, as it encrypts pairs of letters instead of single letters.

Theory :

The Playfair cipher is a cryptographic technique that encrypts pairs of letters (bigrams) instead of single letters, which makes it more secure than simple substitution ciphers. It uses a 5x5 matrix of letters (usually excluding 'J' and combining it with 'I'). The encryption and decryption are based on the positions of pairs of letters in this matrix.

The Playfair cipher uses a 5x5 matrix constructed from a keyword, where each letter of the keyword fills the matrix (repeated letters are removed). If there are fewer than 25 unique letters in the keyword, the remaining spaces are filled with the rest of the alphabet (usually excluding 'J').

Plaintext is split into pairs of letters. If a pair has two identical letters (e.g., "LL"), an 'X' is inserted between them to separate them.

The rules for encryption depend on the positions of the letters in the 5x5 matrix:

1. **Same row**: Replace each letter with the one to its right (wrap around if needed).
2. **Same column**: Replace each letter with the one below it (wrap around if needed).
3. **Rectangle rule**: If neither of the above, replace the letters with the ones on the same row but in the opposite corners of the rectangle formed by the pair.

Algorithm :

1. **Prepare the Key Matrix**: Remove duplicates from the keyword. Complete the matrix with the remaining letters of the alphabet (excluding 'J').
2. Split the Plaintext into pairs of letters. If a pair consists of two identical letters, insert a filler letter (usually 'X').
3. **Encryption**: For each pair of letters, apply the encryption rules based on their positions in the matrix.
4. Return the Encrypted Text as the ciphertext.

Code (Python) :

```
import string

def create_key_square(key):
    """Create a 5x5 key square matrix for the Playfair cipher using a keyword."""
    # Remove duplicates, uppercase the key, and exclude 'J' (treating 'I' and 'J' as the same)
    key = ''.join(sorted(set(key.upper()), key=key.index)).replace('J', '')
    alphabet = string.ascii_uppercase.replace('J', '')  # Alphabet without 'J'

    # Complete the key square with the remaining letters in the alphabet
    key_square = key + ''.join(letter for letter in alphabet if letter not in key)
    return [key_square[i:i+5] for i in range(0, len(key_square), 5)]

def prepare_plaintext(plaintext):
    """Prepare the plaintext by pairing letters and adding 'Z' if needed."""
    plaintext = plaintext.replace(' ', '').upper()
    prepared_text = []
    i = 0

    while i < len(plaintext):
        # Add 'Z' if a duplicate character is found in a pair
        if i + 1 < len(plaintext) and plaintext[i] == plaintext[i + 1]:
            prepared_text.append(plaintext[i] + 'Z')
            i += 1
        elif i + 1 < len(plaintext):
            prepared_text.append(plaintext[i:i + 2])
            i += 2
        else:
            prepared_text.append(plaintext[i] + 'Z')  # Pad with 'Z' if last character is alone
```

```
            i += 1
    return prepared_text

def find_position(char, key_matrix):
    """Find the row and column of a character in the key square."""
    for row in range(5):
        for col in range(5):
            if key_matrix[row][col] == char:
                return row, col
    return None

def playfair_encrypt(plaintext, key):
    """Encrypt plaintext using the Playfair cipher."""
    key_matrix = create_key_square(key)
    prepared_text = prepare_plaintext(plaintext)
    ciphertext = []

    # Encrypt each digraph according to the Playfair cipher rules
    for digraph in prepared_text:
        first_char, second_char = digraph
        row1, col1 = find_position(first_char, key_matrix)
        row2, col2 = find_position(second_char, key_matrix)

        if row1 == row2:  # Same row: shift right
            ciphertext.extend([key_matrix[row1][(col1 + 1) % 5],
key_matrix[row2][(col2 + 1) % 5]])
        elif col1 == col2:  # Same column: shift down
            ciphertext.extend([key_matrix[(row1 + 1) % 5][col1],
key_matrix[(row2 + 1) % 5][col2]])
        else:  # Rectangle rule: swap columns
            ciphertext.extend([key_matrix[row1][col2], key_matrix[row2][col1]])

    return ''.join(ciphertext)

# Example usage:
plaintext = "DIBAKOR F"
key = "KEYWORD"
ciphertext = playfair_encrypt(plaintext, key)
print(f"Plaintext: {plaintext}")
print(f"Ciphertext: {ciphertext}")
```

Input :

Input Plaintext: DIBAKOR F

Input Key: KEYWORD

Output :

Ciphertext: BGCBEKFM

Experiment no. 06 :

Experiment name: Write a program to implement decryption using Playfair cipher.

Objectives:

1. **To decrypt a ciphertext** that was previously encrypted using the Playfair cipher.
2. **To use the same key matrix** that was used for encryption to reverse the process and recover the original plaintext.
3. **To correctly handle special cases** such as pairs of identical letters or padded letters (like 'X') that were added during encryption.

Theory :

The decryption process in Playfair follows the same rules as encryption but in reverse:

- **Key Matrix**: The same matrix used for encryption is used for decryption. It is built from the keyword and contains 5x5 letters (excluding 'J').
- **Text Preparation**: Ciphertext is split into pairs, and the decryption process uses the inverse of the encryption rules:
    1. **Same row**: Replace each letter with the letter to its left (wrap around if needed).
    2. **Same column**: Replace each letter with the letter above it (wrap around if needed).
    3. **Rectangle rule**: If neither of the above, replace the letters with the ones on the same row but in the opposite corners of the rectangle formed by the pair.
- The main difference in decryption is that instead of shifting to the right or down as in encryption, you shift left or up.

Algorithm :

- **Prepare the Key Matrix**:

- Remove duplicates from the keyword.
- Fill the remaining letters of the alphabet (excluding 'J') to complete the matrix.
- **Prepare the Ciphertext**:
- Split the ciphertext into pairs of letters.
- Handle any special characters (like padding) that may have been added during encryption.
- **Decrypt the Text**:
- For each pair of letters, reverse the encryption rules based on their positions in the matrix.

Code(Python):

```
def create_key_square(key):
    """Create a 5x5 key square for the Playfair cipher, using a keyword."""
    key = key.upper().replace("J", "I")  # Treat I and J as the same
    seen = set()
    key_square = []

    # Add unique characters from the key to the key square
    for char in key:
        if char.isalpha() and char not in seen:
            seen.add(char)
            key_square.append(char)

    # Add remaining letters of the alphabet (excluding J)
    for char in "ABCDEFGHIKLMNOPQRSTUVWXYZ":
        if char not in seen:
            key_square.append(char)

    return [key_square[i:i + 5] for i in range(0, 25, 5)]

def find_position(char, key_square):
    """Find the row and column of a character in the key square."""
    for i, row in enumerate(key_square):
        if char in row:
```

```
            return i, row.index(char)
    return -1, -1

def decrypt_digraph(digraph, key_square):
    """Decrypt a two-letter digraph using the Playfair cipher rules."""
    row1, col1 = find_position(digraph[0], key_square)
    row2, col2 = find_position(digraph[1], key_square)

    # Apply Playfair cipher decryption rules
    if row1 == row2:  # Same row: move left
        return key_square[row1][(col1 - 1) % 5] + key_square[row2][(col2 - 1)
% 5]
    elif col1 == col2:  # Same column: move up
        return key_square[(row1 - 1) % 5][col1] + key_square[(row2 - 1) %
5][col2]
    else:  # Rectangle: swap columns
        return key_square[row1][col2] + key_square[row2][col1]

def playfair_decrypt(ciphertext, key):
    """Decrypt ciphertext encrypted with the Playfair cipher."""
    key_square = create_key_square(key)
    ciphertext = ciphertext.upper().replace(" ", "")

    # Split ciphertext into digraphs and decrypt each
    digraphs = [ciphertext[i:i + 2] for i in range(0, len(ciphertext), 2)]
    plaintext = ''.join(decrypt_digraph(digraph, key_square) for digraph in
digraphs)

    return plaintext

# Example Usage
ciphertext = "BGCBEKFM"
key = "KEYWORD"
plaintext = playfair_decrypt(ciphertext, key)
print(f"Ciphertext: {ciphertext}")
print(f"Key: {key}")
print(f"Plaintext: {plaintext}")
```

Input :

Ciphertext: BGCBEKFM

Key: KEYWORD

Output :

Plaintext: DIBAKORF

Experiment no. 07 :

Experiment name : Write a program to implement encryption using Poly-Alphabetic cipher.

Objectives :

1. **Understand the concept of Poly-Alphabetic Cipher**: Learn how to encrypt and decrypt text using multiple alphabets.
2. **Learn how to use a keyword**: The cipher uses a keyword to shift letters and encrypt text, making the encryption more secure than the Caesar cipher.
3. **Implement the encryption algorithm**: Write a Python code to encrypt text using the Poly-Alphabetic Cipher.

Theory :

The Poly-Alphabetic Cipher (also known as the Vigenère Cipher) is a cipher that uses multiple Caesar ciphers based on a keyword. It shifts each letter of the plaintext by a number of positions determined by the corresponding letter of the keyword.

Key Features:

- The encryption shifts each letter of the plaintext using a different shift value, depending on the corresponding letter of the keyword.
- If the keyword is shorter than the plaintext, it is repeated to match the length of the plaintext.
- The key used in the cipher ensures that the same letter in the plaintext might be encrypted to different ciphertext letters, depending on the corresponding letter in the keyword.

Encryption Formula:

For encryption, each letter of the plaintext is shifted by an amount defined by the corresponding letter of the key. The formula for encryption is:

$$\mathbf{Ci = (Pi + Ki) \mod 26}$$

Where:

- Ci is the i-th letter of the ciphertext,
- Pi is the i-th letter of the plaintext,
- Ki is the i-th letter of the keyword,
- Modulo 26 ensures the shift stays within the bounds of the alphabet.

Example :

- **Plaintext**: "HELLO"
- **Keyword**: "KEY"
- **Step 1**: Repeat the keyword until it matches the length of the plaintext: "KEYKE"
- **Step 2**: Encrypt using the Vigenère cipher.
    - 'H' (shifted by 'K' which is 10) → 'R'
    - 'E' (shifted by 'E' which is 4) → 'I'
    - 'L' (shifted by 'Y' which is 24) → 'J'
    - 'L' (shifted by 'K' which is 10) → 'V'
    - 'O' (shifted by 'E' which is 4) → 'S'
- **Ciphertext**: "RIJVS"

Algorithm :

- **Input**:
- Plaintext and keyword (both in uppercase).
- **Preprocessing**:
- Remove non-alphabet characters from the plaintext.
- Repeat the keyword until it matches the length of the plaintext.
- **Encryption**:

- For each letter in the plaintext:
    - Find the shift value from the corresponding letter of the keyword.
    - Encrypt by shifting the plaintext letter by the shift value.
- **Output**:
- Return the ciphertext.

Code (Python):

```
def poly_alphabetic_encrypt(plaintext, keyword):
    # Remove non-alphabetic characters and convert to uppercase
    plaintext = ''.join([char.upper() for char in plaintext if char.isalpha()])

    # Repeat the keyword until it matches the length of the plaintext
    keyword = ''.join([char.upper() for char in keyword if char.isalpha()])
    keyword_repeated = (keyword * (len(plaintext) // len(keyword))) +
keyword[:len(plaintext) % len(keyword)]

    # Encrypt the plaintext
    ciphertext = ''
    for p, k in zip(plaintext, keyword_repeated):
        # Calculate shift for each character
        shift = ord(k) - ord('A')  # Key letter shift (0 for A, 1 for B, ..., 25 for Z)
        encrypted_char = chr((ord(p) - ord('A') + shift) % 26 + ord('A'))
        ciphertext += encrypted_char

    return ciphertext

# Example usage
plaintext = "HELLO"
keyword = "KEY"
ciphertext = poly_alphabetic_encrypt(plaintext, keyword)

# Display input and output
print("Input Plaintext:", plaintext)
print("Input Keyword:", keyword)
print("Encrypted Ciphertext:", ciphertext)
```

Input :

Input Plaintext: HELLO

Input Keyword: KEY

Output :

Encrypted Ciphertext: RIJVS

Experiment no. 08 :

Experiment name: Write a program to implement decryption using Poly-Alphabetic cipher.

Objectives :

1. **Understand the concept of Poly-Alphabetic Cipher decryption**: Learn how to decrypt the ciphertext using a keyword.
2. **Learn how to reverse the encryption process**: Decryption involves shifting the letters in the opposite direction compared to encryption.
3. **Implement the decryption algorithm in Python**: Write Python code that implements the Poly-Alphabetic Cipher decryption process.

Theory :

The Poly-Alphabetic Cipher (Vigenère Cipher) encryption uses a keyword to shift each letter of the plaintext by a corresponding amount based on the keyword letter. Decryption reverses the process and shifts each letter of the ciphertext back by the corresponding letter in the keyword.

Decryption Formula:

For decryption, the formula is:

$$Pi = (Ci – Ki) \mod 26$$

Where:

- Pi is the i-th letter of the plaintext,
- Ci is the i-th letter of the ciphertext,
- Ki is the i-th letter of the keyword,
- The modulo 26 ensures the shift stays within the bounds of the alphabet.

Example :

- **Ciphertext**: "RIJVS"
- **Keyword**: "KEY"

Decryption Steps:

1. Repeat the keyword to match the length of the ciphertext: "KEYKE".
2. Decrypt each letter by shifting it back by the position of the corresponding keyword letter:
    - 'R' (shifted by 'K' which is 10) → 'H'
    - 'I' (shifted by 'E' which is 4) → 'E'
    - 'J' (shifted by 'Y' which is 24) → 'L'
    - 'V' (shifted by 'K' which is 10) → 'L'
    - 'S' (shifted by 'E' which is 4) → 'O'

- **Plaintext**: "HELLO"

Algorithm :

- **Input**:
- The ciphertext to be decrypted.
- The keyword used during encryption.
- **Preprocessing**:
- Convert the ciphertext and the keyword to uppercase.
- Remove any non-alphabetic characters from both ciphertext and keyword.
- Repeat the keyword until its length matches the length of the ciphertext.
- **Decryption Process**:
- For each character in the ciphertext:
    - Find the shift value corresponding to the letter of the keyword.
    - Decrypt by shifting the ciphertext letter in the opposite direction by the shift value.

- **Output**:

- Return the decrypted plaintext.

Code (Python) :

```
def poly_alphabetic_decrypt(ciphertext, keyword):
    # Remove non-alphabetic characters and convert to uppercase
    ciphertext = ''.join([char.upper() for char in ciphertext if char.isalpha()])

    # Repeat the keyword until it matches the length of the ciphertext
    keyword = ''.join([char.upper() for char in keyword if char.isalpha()])
    keyword_repeated = (keyword * (len(ciphertext) // len(keyword))) +
keyword[:len(ciphertext) % len(keyword)]

    # Decrypt the ciphertext
    plaintext = ''
    for c, k in zip(ciphertext, keyword_repeated):
        # Calculate the shift for each character
        shift = ord(k) - ord('A')  # Key letter shift (0 for A, 1 for B, ..., 25 for Z)
        decrypted_char = chr((ord(c) - ord('A') - shift) % 26 + ord('A'))
        plaintext += decrypted_char

    return plaintext

# Example usage
ciphertext = "RIJVS"
keyword = "KEY"
plaintext = poly_alphabetic_decrypt(ciphertext, keyword)

# Display input and output
print("Input Ciphertext:", ciphertext)
print("Input Keyword:", keyword)
print("Decrypted Plaintext:", plaintext)
```

Input :

Input Ciphertext: RIJVS

Input Keyword: KEY

Output :

Decrypted Plaintext: HELLO

Experiment no. 09 :

Experiment name: Write a program to implement encryption using Vernam cipher.

Objectives:

1. Understand the concept of the Vernam cipher.
2. Implement the encryption and decryption processes for the Vernam cipher.
3. Ensure data security by using a key of the same length as the message.

Theory:

The Vernam cipher, also known as the one-time pad, is a symmetric encryption technique that provides theoretically unbreakable security when used under strict conditions. In this theory, implementing the Vernam cipher involves using a purely random key that matches the length of the plaintext, with the key used only once and securely shared between sender and receiver.

The Pure Vernam Cipher Implementation theory is based on three core principles:

1. **One-Time Key Generation**: The security of the Vernam cipher depends on the key being as long as the message and completely random. Each character in the key is independently chosen with equal probability, ensuring there is no predictability or pattern.
2. **Exclusive OR (XOR) Encryption**: For encryption, each character in the plaintext is XORed with the corresponding character in the key. This operation is mathematically simple but effective because it transforms the plaintext into an unrecognizable ciphertext that can only be decrypted with the same key.
3. **Strict Key Management and Destruction**: After one use, the key is permanently destroyed to ensure it cannot be reused, as reuse would compromise the cipher's security. Keys are shared securely between the communicating parties before encryption, and their secrecy is critical to the Vernam cipher's unbreakability.

Example :

Let's consider a simple example with plaintext **"HERO"**:

1. Suppose the random key generated is **"ABCDE"** (same length as the plaintext).

2. Each character in "HERO" is XOR-ed with the corresponding character in "ABCDE" to produce the ciphertext.

Example Calculation:

- For encryption of "H" with "X": ASCII('H')=72, ASCII('A')=65 , and 72 $\oplus$ 65 = 9, which corresponds to some character in the ASCII table.
- Repeat for all character.

3. Applying the same XOR operation on the ciphertext with the key will retrieve the original plaintext "HERO".

Algorithm:

Input: Plaintext message and a key of equal length.

Check: Ensure that the key length matches the plaintext length.

Encryption:

- For each character in the plaintext:
    - Convert the plaintext character and the corresponding key character to ASCII.
    - Perform XOR between the ASCII values of the plaintext character and key character.
    - Convert the result to a character and append it to the ciphertext.

Decryption:

- For each character in the ciphertext:
    - Perform XOR between the ASCII values of the ciphertext character and key character.
    - Convert the result to a character and reconstruct the plaintext.

Code (python):

```
def char_to_bin(char):
    """Convert a character to its corresponding 5-bit binary value (A=0 to Z=25)."""
    return format(ord(char) - ord('A'), '05b')

def bin_to_char(binary):
    """Convert a 5-bit binary value to the corresponding character (0=A to 25=Z)."""
    return chr(int(binary, 2) + ord('A'))

def xor_encrypt(plaintext, key):
    """Encrypt plaintext using an XOR cipher with a repeating key."""
    plaintext, key = plaintext.upper().replace(" ", ""), key.upper().replace(" ", "")
    ciphertext = ""

    for i in range(len(plaintext)):
        pt_bin = char_to_bin(plaintext[i])
        key_bin = char_to_bin(key[i % len(key)])

        # Perform XOR on the binary values
        xor_result = ''.join('1' if pt_bin[j] != key_bin[j] else '0' for j in range(5))

        # Append the XOR result as a character
        ciphertext += bin_to_char(xor_result)

    return ciphertext

# Example usage:
plaintext = "HERO"
key = "ABCDE"
ciphertext = xor_encrypt(plaintext, key)
print(f"Plaintext: {plaintext}")
print(f"Key: {key}")
print(f"Ciphertext: {ciphertext}")
```

Input:

Plaintext: HERO

Key (adjusted): ABCDE

Output:

Ciphertext (encrypted): HFTN

Experiment no.10 :

Experiment name: Write a program to implement decryption using Vernam cipher.

Objectives:

1. To understand how the Vernam cipher works for encryption and decryption, using XOR between the plaintext and the key.
2. To implement and explain the decryption mechanism of the Vernam cipher, which is identical to encryption.
3. To write Python code that decrypts the ciphertext back to plaintext using the same key used for encryption

Theory:

The Vernam cipher also called the one-time pad is a symmetric encryption technique, meaning the same key is used for both encryption and decryption. The key must be as long as the plaintext and should be completely random, ensuring that it is used only once.

Key Concepts:

- **Plaintext (P)**: The original message.
- **Key (K)**: A random string of the same length as the plaintext.
- **Ciphertext (C)**: The encrypted message.

Encryption:

The encryption process of the Vernam cipher works as follows:

$C = P \oplus K$

Where:

- $\oplus$ denotes the XOR operation.
- Each character in the plaintext is XORed with the corresponding character in the key.

Decryption:

The decryption process of the Vernam cipher works the same way as encryption:

$P = C \oplus K$

Since XORing a value twice with the same key retrieves the original value, decryption with the same key results in the original plaintext.

Algorithm :

1. Convert the ciphertext and key into binary.
2. XOR each bit of the ciphertext with the corresponding bit of the key.
3. Convert the resulting binary back to text.

Code (Python) :

```
def char_to_bin(char):
    """Convert a character to its corresponding 5-bit binary value (A=0 to Z=25)."""
    return format(ord(char) - ord('A'), '05b')

def bin_to_char(binary):
    """Convert a 5-bit binary value to the corresponding character (0=A to 25=Z)."""
    return chr(int(binary, 2) + ord('A'))

def xor_decrypt(ciphertext, key):
    """Decrypt ciphertext using an XOR cipher with a repeating key."""
    ciphertext, key = ciphertext.upper().replace(" ", ""), key.upper().replace(" ", "")
    plaintext = ""

    for i in range(len(ciphertext)):
        ct_bin = char_to_bin(ciphertext[i])
        key_bin = char_to_bin(key[i % len(key)])

        # Perform XOR on the binary values
        xor_result = ''.join('1' if ct_bin[j] != key_bin[j] else '0' for j in range(5))

        # Append the XOR result as a character
        plaintext += bin_to_char(xor_result)

    return plaintext

# Example usage:
ciphertext = "HFTN"
key = "ABCDE"
plaintext = xor_decrypt(ciphertext, key)
print(f"Ciphertext: {ciphertext}")
```

```
print(f"Key: {key}")
print(f"Plaintext: {plaintext}")
```

Input :

Ciphertext: HFTN

Key: ABCDE

Output :

Plaintext: HERO