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:	Submitted to:
Name: MD. Rakibul Hossain **Roll: 200616** **Session: 2019-2020** **4th year 1st semester.** **Department of Information and Communication Engineering,** **PUST.**	**Md. Anwar Hossain** **Professor,** **Department of Information and Communication Engineering,** **Pabna University of Science and Technology.**

Submission date: 12-11-2024 **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.	

Number of experiment: 01

Name of experiment: Write a program to implement encryption and decryption using Caesar cipher.

Objective:

1) Implement Caesar cipher encryption and decryption.
2) Ensure reversible encryption and decryption process.
3) Allow user-defined shift key for flexibility.

Theory:

The Caesar cipher is one of the simplest and oldest encryption techniques. Named after Julius Caesar, who reportedly used it to protect military messages, this cipher shifts each letter in the plaintext by a fixed number of positions in the alphabet. This shift, known as the "key," allows the sender and recipient to encode and decode messages securely as long as they share this key.

For encryption, each letter in the plaintext P is shifted K positions to get the ciphertext C.

$$C = (P+K) \bmod n.$$

For decryption, each letter in the ciphertext C is shifted back by K positions to recover the plaintext P.

$$P = (C-K) \bmod n$$

Where,

P = the plaintext letter.

C = the ciphertext letter.

K = the shift key.

n = the number of letters in the alphabet.

For example, the plaintext P = RAKIB and the shift key K=3.Now we calculate the encryption and decryption using Caesar cipher.

First, we convert each letter to its alphabet position.

R=17, A=0, K=10, I=8, B=1

For encryption using the formula,

Ciphertext $C = (P+K) \bmod n$

$R = (17+3) \bmod 26 = 20 \Rightarrow U$

A = (0+3) mod 26 = 3 =>D

K = (10+3) mod 26 = 13 =>N

I = (17+3) mod 26 = 20 =>L

B = (1+3) mod 26 = 4 =>E

So, the ciphertext, C = UDNLE

For decryption using the formula,

P = (C−K) mod n

U = (20-3) mod 26 = 17 =>R

D = (3-3) mod 26 = 0 =>A

N = (13-3) mod 26 = 10 =>K

L = (11-3) mod 26 = 8 =>I

E = (4-3) mod 26 = 1 =>B

So, the decrypted message, P=RAKIB.

Algorithm:

1. Convert all letters to upper case.
2. Convert each letter to its alphabet position.
3. Encrypted the message by using the formula C = (P+K) mod n.
4. Decrypted the message by using the formula P = (C−K) mod n.
5. Print the ciphertext and decrypted message.

Source code:

```
def caesar_encrypt(plaintext, shift):
    ciphertext = ""
    for char in plaintext.upper():  # Convert to uppercase
        if char.isalpha():
            ciphertext += chr((ord(char) - 65 + shift) % 26 + 65)
        else:
            ciphertext += char
    return ciphertext

def caesar_decrypt(ciphertext, shift):
    plaintext = ""
```

```
    for char in ciphertext.upper():  # Convert to uppercase
        if char.isalpha():
            plaintext += chr((ord(char) - 65 - shift) % 26 + 65)
        else:
            plaintext += char
    return plaintext

message = input("Enter the message: ")
shift_key = int(input("Enter the shift key (integer): "))

encrypted_message = caesar_encrypt(message, shift_key)
print("Encrypted:", encrypted_message)

decrypted_message = caesar_decrypt(encrypted_message, shift_key)
print("Decrypted:", decrypted_message)
```

Input:

Enter the message: rakib

Enter the shift key (integer): 3

Output:

Encrypted: UDNLE

Decrypted: RAKIB

Number of experiment: 02

Name of experiment: Write a program to implement encryption and decryption using Mono-Alphabetic cipher.

Objective:

1) Implement encryption and decryption with a Mono-Alphabetic cipher.
2) Support user-defined substitution keys for encryption.
3) Ensure non-alphabetic characters remain unchanged.

Theory:

The Mono-Alphabetic cipher is a substitution cipher in which each letter of the plaintext is substituted by a corresponding letter from a fixed, predefined alphabet or key. The key is a permutation (rearranged) version of the standard alphabet, and it is used to replace each letter in the plaintext message with a unique letter from the key.

Encryption Equation:
Given a plaintext letter P, the corresponding ciphertext letter C is determined by the substitution key.

$$C=K(P)$$

Where:

P = Position of the plaintext letter in the standard alphabet (A=0, B=1, ..., Z=25)

K(P) = The letter in the key that corresponds to the position of P.

For example, if the message is RAKIB and the key is ASDFGHJKLZXCVB. Now calculate the ciphertext and decripted text.

For encryption:

The standard alphabet (A-Z) is as follows:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Now, key with unused alphabet:

A S D F G H J K L Z X C V B E I M N O P Q R T U W Y

Match the Original Alphabet with the Key and find the cipher text.

Here, the cipher text is: NAXLS.

For decryption:

To decrypt the ciphertext **"NAXLS"**, we reverse the process. We will find each letter of the ciphertext in the key, and map it to the corresponding letter in the standard alphabet.

Here, the decrypted message is: RAKIB.

Algorithm:

1. Generate a mapping of the standard alphabet (A-Z) to the key.
2. Generate a mapping of key and unused alphabet in the key. Ensure that there has no duplicate letters.

For encryption:

3. Find the position of the letter in the standard alphabet.
4. Replace it with the corresponding letter from the key.

For decryption:

5. Find the position of the letter in the key.
6. Replace it with the corresponding letter from the standard alphabet.

7. Print the ciphertext and decrypted text.

Source code:

```
# Function to create the cipher mapping from the key
def create_cipher_mapping(key):
    # Remove spaces and duplicate characters from the key
    key = ''.join(sorted(set(key), key=key.index)).replace(' ', '')

    # Define the alphabet for the mapping
    alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

    # Create a list of letters that are not in the key
    remaining_letters = [letter for letter in alphabet if letter not in key]

    # Combine key with the remaining letters to complete the substitution
    full_key = key + ''.join(remaining_letters)
    return full_key

# Function for encryption
```

```
def encrypt(plaintext, key):
    alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    cipher_mapping = create_cipher_mapping(key)

    ciphertext = ""
    for letter in plaintext.upper():
        if letter in alphabet:
            # Find the position of the letter in the alphabet and get the corresponding
letter from the key
            index = alphabet.index(letter)
            ciphertext += cipher_mapping[index]
        else:
            ciphertext += letter  # Non-alphabet characters (like spaces) remain
unchanged

    return ciphertext

# Function for decryption
def decrypt(ciphertext, key):
    alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    cipher_mapping = create_cipher_mapping(key)

    plaintext = ""
    for letter in ciphertext.upper():
        if letter in cipher_mapping:
            # Find the position of the letter in the key and get the corresponding
letter from the alphabet
            index = cipher_mapping.index(letter)
            plaintext += alphabet[index]
        else:
            plaintext += letter  # Non-alphabet characters (like spaces) remain
unchanged

    return plaintext

# Main function to demonstrate encryption and decryption
def main():
    key = input("Enter the substitution key: ").upper()
    plaintext = input("Enter the message to encrypt: ").upper()
```

```
    # Encrypt the plaintext
    ciphertext = encrypt(plaintext, key)
    print("Ciphertext: ", ciphertext)

    # Decrypt the ciphertext back to plaintext
    decrypted_text = decrypt(ciphertext, key)
    print("Decrypted message: ", decrypted_text)

# Run the main function
if __name__ == "__main__":
    main()
```

Input:

Enter the substitution key: ASDFGHJKLZXCVB

Enter the message to encrypt: RAKIB

Output:

Ciphertext: NAXLS

Decrypted message: RAKIB

Number of experiment: 03

Name of experiment: Write a program to implement encryption and decryption using Brute force attack cipher.

Objective:

1) Implement brute force to decrypt by trying all possible keys.
2) Simulate decryption without knowing the key.
3) Compare decrypted output with the original plaintext.

Theory:

A brute force attack is a method used to decrypt a message by systematically trying all possible keys until the correct one is found. In the case of a Caesar cipher (a simple substitution cipher), the brute force attack tries all possible shifts (from 1 to 25) to find the key that correctly decrypts the ciphertext into meaningful plain text. The Caesar cipher is a type of substitution cipher where each letter in the plaintext is shifted by a certain number of positions down or up the alphabet.

Algorithm:

1. Accept the plaintext message.
2. Accept the shift value (key) for encryption.
3. For each character in the plaintext, shift alphabetic characters by the key and leave non-alphabetic characters unchanged.
4. Construct the ciphertext from the modified characters.
5. Display the encrypted ciphertext.
6. For each shift value k from 1 to 25, decrypt the ciphertext by shifting characters backward by k.
7. Print the result of each decryption attempt.
8. End the process when the correct shift (original key) is found.

Source code:

```
# Function to encrypt a message with a given key (shift)

def encrypt(plaintext, shift):
    ciphertext = ""
    for char in plaintext:
        if char.isalpha():  # Check if the character is a letter
            # Convert letter to its alphabetical index (0-25)
```

```
            char_index = ord(char.upper()) - 65
            # Encrypt the letter by shifting it forwards by the given key
            new_index = (char_index + shift) % 26
            # Convert back to letter and preserve case
            encrypted_char = chr(new_index + 65) if char.isupper() else
chr(new_index + 97)
            ciphertext += encrypted_char
        else:
            # Keep non-alphabet characters unchanged
            ciphertext += char
    return ciphertext

# Function to decrypt a message with a given key (shift)
def decrypt_with_key(ciphertext, k):
    plaintext = ""
    for char in ciphertext:
        if char.isalpha():  # Check if the character is a letter
            # Convert letter to its alphabetical index (0-25)
            char_index = ord(char.upper()) - 65
            # Decrypt the letter by shifting it backwards by key k
            new_index = (char_index - k) % 26
            # Convert back to letter and preserve case
            decrypted_char = chr(new_index + 65) if char.isupper() else
chr(new_index + 97)
            plaintext += decrypted_char
        else:
            # Keep non-alphabet characters unchanged
            plaintext += char
    return plaintext
```

```
# Function for brute force attack on the Caesar cipher
def brute_force_attack(ciphertext):
    for k in range(1, 26):  # Try all shifts from 1 to 25
        decrypted_text = decrypt_with_key(ciphertext, k)
        print(f"Key {k}: {decrypted_text}")

# Main function to demonstrate encryption, brute force attack and decryption
def main():
    # Input the plaintext and the shift value
    plaintext = input("Enter the plaintext to encrypt: ")
    shift = int(input("Enter the shift value (1-25): "))

    # Encrypt the plaintext
    ciphertext = encrypt(plaintext, shift)
    print(f"Ciphertext: {ciphertext}")

    # Perform brute force attack on the ciphertext to recover the original message
    print("\nBrute force attack results:")
    brute_force_attack(ciphertext)

# Run the main function
if __name__ == "__main__":
    main()
```

Input:

Enter the plaintext to encrypt: RAKIB

Enter the shift value (1-25): 3

Output:

Ciphertext: UDNLE

Brute force attack results:

Key 1: TCMKD

Key 2: SBLJC

Key 3: RAKIB

Key 4: QZJHA

Key 5: PYIGZ

Key 6: OXHFY

Key 7: NWGEX

Key 8: MVFDW

Key 9: LUECV

Key 10: KTDBU

Key 11: JSCAT

Key 12: IRBZS

Key 13: HQAYR

Key 15: FOYWP

Key 16: ENXVO

Key 17: DMWUN

Key 18: CLVTM

Key 19: BKUSL

Key 20: AJTRK

Key 21: ZISQJ

Key 22: YHRPI

Key 23: XGQOH

Key 24: WFPNG

Key 25: VEOMF

Number of experiment: 04

Name of experiment: Write a program to implement encryption and decryption using Hill cipher.

Objective:

1) Encrypt using plaintext and key matrix multiplication.
2) Decrypt using the inverse of the key matrix.
3) Apply matrix operations for secure message encryption.

Theory:

Hill cipher is a classical symmetric encryption algorithm that uses linear algebra to encrypt messages. It was invented by Lester S. Hill in 1929 and operates on a matrix-based system, where the plaintext is treated as a vector and the encryption process involves multiplying this vector with a key matrix (a square matrix). This method requires the matrix to be invertible (non-singular) for decryption.

Encryption Equation:

Let P be the plaintext vector and K the key matrix. The ciphertext vector C is obtained by the following matrix multiplication:

$$C=K\times P \pmod{26}$$

Where,

K is the key matrix,

P is the plaintext vector,

C is the ciphertext vector.

Decryption Equation:

For decryption, the inverse of the key matrix $K^{-1}$is used:

$$P=K^{-1}\times C \pmod{26}$$

Where:

K^{-1} is the inverse of the key matrix,

C is the ciphertext vector,

P is the original plaintext vector.

For example, Assume the key matrix K is a 2x2 matrix:

$$K = \begin{bmatrix} 6 & 24 \\ 1 & 17 \end{bmatrix}$$

The plaintext is "HI". Convert the plaintext into a vector of numbers,

$$P = \begin{bmatrix} 7 \\ 8 \end{bmatrix}$$

Now, multiply the key matrix by the plaintext vector and take modulo 26,

$$C = K \times P = \begin{bmatrix} 6 & 24 \\ 1 & 17 \end{bmatrix} \times \begin{bmatrix} 7 \\ 8 \end{bmatrix} = \begin{bmatrix} 234 \\ 143 \end{bmatrix}$$

Now take modulo 26 of each value:

$$C = \begin{bmatrix} 234 \ mod \ 26 \\ 143 \ mode \ 26 \end{bmatrix} = \begin{bmatrix} 0 \\ 15 \end{bmatrix}$$

Which corresponds to the ciphertext "AP" (A = 0, P = 15).

To decrypt the ciphertext, calculate the inverse of the key matrix K^{-1}, and then multiply it by the ciphertext vector C modulo 26.

Let's assume we have K^{-1} (this can be computed using modular arithmetic and finding the inverse matrix).

Finally, multiply K^{-1} by the ciphertext vector to retrieve the original plaintext.

Algorithm:

1. Accept the plaintext message and key matrix.
2. Convert the plaintext into numerical representation (A = 0, B = 1, ..., Z = 25).
3. Ensure the length of the plaintext is a multiple of the matrix size, pad if necessary.
4. For encryption, multiply the key matrix by the plaintext vector modulo 26.
5. Apply the matrix multiplication result to get the ciphertext.
6. Convert the ciphertext numerical vector back to letters.
7. For decryption, find the inverse of the key matrix modulo 26.
8. Multiply the inverse key matrix by the ciphertext vector modulo 26.
9. Convert the decrypted vector back into plaintext text.
10. Display the encrypted and decrypted messages.

Source code:

```
import string
import numpy as np

alphabet = string.ascii_lowercase
letter_to_index = dict(zip(alphabet, range(len(alphabet))))
index_to_letter = dict(zip(range(len(alphabet)), alphabet))

def egcd(a, b):
    if a == 0:
        return b, 0, 1
    else:
        gcd, x, y = egcd(b % a, a)
        return gcd, y - (b // a) * x, x

def mod_inv(det, modulus):
    gcd, x, y = egcd(det, modulus)
    if gcd != 1:
        raise Exception("Matrix is not invertible.")
    return (x % modulus + modulus) % modulus

def matrix_mod_inv(matrix, modulus):
    det = int(np.round(np.linalg.det(matrix)))
    det_inv = mod_inv(det, modulus)
    matrix_modulus_inv = (
            det_inv * np.round(det * np.linalg.inv(matrix)).astype(int) % modulus
```

```
    )
    return matrix_modulus_inv.astype(int)

def encrypt_decrypt(message, K):
    msg = ""
    message_in_numbers = [letter_to_index[letter] for letter in message]
    split_P = [
        message_in_numbers[i: i + len(K)]
        for i in range(0, len(message_in_numbers), len(K))
    ]
    for P in split_P:
        P = np.transpose(np.asarray(P))[:, np.newaxis]
        while P.shape[0] != len(K):
            P = np.append(P, letter_to_index[" "])[:, np.newaxis]
        numbers = np.dot(K, P) % len(alphabet)
        n = numbers.shape[0]
        for idx in range(n):
            number = int(numbers[idx, 0])
            msg += index_to_letter[number]
    return msg

message = "help"
K = np.matrix([[3, 3], [2, 5]])
Kinv = matrix_mod_inv(K, len(alphabet))

encrypted_message = encrypt_decrypt(message, K)
decrypted_message = encrypt_decrypt(encrypted_message, Kinv)
```

```
print("Original message: " + message.upper())
print("Encrypted message: " + encrypted_message.upper())
print("Decrypted message: " + decrypted_message.upper())
```

Output:

Original message: HELP

Encrypted message: HIAT

Decrypted message: HELP

Number of experiment: 05

Name of experiment: Write a program to implement encryption using Playfair cipher.

Objective:

1) Implement Playfair cipher encryption with a 5x5 key matrix.
2) Encrypt plaintext by processing digraphs using Playfair rules.
3) Handle padding and duplicate letters during encryption.

Theory:

The Playfair cipher is a symmetric encryption technique that encrypts pairs of letters (digraphs) instead of single letters, providing better security compared to simple substitution ciphers. It was developed by Charles Wheatstone in 1854 and was later promoted by Lord Playfair, after whom the cipher is named. A 5x5 matrix is used to hold the letters of the alphabet (excluding 'J', which is typically merged with 'I' for 25 letters).

Matrix Making:

1. Take the key and remove any duplicate letters.
2. Use the key to fill a 5x5 matrix, row by row.
3. Add the remaining letters of the alphabet (excluding 'J', which is combined with 'I') to the matrix to fill all 25 slots.

Plaintext Preparation:

1. The plaintext is divided into digraphs (pairs of two letters). If there is an odd number of letters, an 'X' or another padding character is added at the end to complete the last pair.
2. If a pair consists of two identical letters (e.g., "LL"), one of the letters is replaced by a filler character (e.g., "LX").

Rules for Encryption: For each pair of letters:

1. If the letters appear in the same row of the matrix, each letter is replaced by the letter immediately to its right (circularly).
2. If the letters appear in the same column, each letter is replaced by the letter immediately below (circularly).
3. If the letters form a rectangle (i.e., they are neither in the same row nor the same column), each letter is replaced by the letter in its row but in the column of the other letter of the pair.

Algorithm:

1. Input the plaintext and key.
2. Convert all letters to uppercase.
3. Remove any duplicate letters in the key.
4. If the plaintext has an odd number of characters, append an extra letter (usually 'X') to make it even.
5. Split the plaintext into digraphs (pairs of two letters).
6. Construct a 5x5 matrix using the key and fill in the remaining letters of the alphabet (except 'J', which is combined with 'I').
7. For each digraph:
   - If the letters appear in the same row, replace them with the letters immediately to their right (wrap around to the beginning if necessary).
   - If the letters appear in the same column, replace them with the letters immediately below (wrap around to the top if necessary).
   - If the letters form a rectangle, swap their row and column positions.
8. Combine the transformed digraphs to form the ciphertext.
9. Output the final ciphertext.

Source code:

```
def create_matrix(key):
    # Create a 5x5 matrix using the key
    key = key.upper().replace('J', 'I')  # Replace J with I as per Playfair rules
    matrix = []
    used_chars = set()
    for char in key:
        if char.isalpha() and char not in used_chars:
            matrix.append(char)
            used_chars.add(char)
```

```
    for char in 'ABCDEFGHIKLMNOPQRSTUVWXYZ':  # Note: No J
        if char not in used_chars:
            matrix.append(char)
            used_chars.add(char)

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

def find_position(matrix, char):
    for i in range(5):
        for j in range(5):
            if matrix[i][j] == char:
                return i, j
    return None

def prepare_text(text):
    text = ''.join(c.upper() for c in text if c.isalpha())
    text = text.replace('J', 'I')

    pairs = []
    i = 0
    while i < len(text):
        if i == len(text) - 1:  # If the last character is left
            pairs.append(text[i] + 'X')
            break
        elif text[i] == text[i + 1]:  # If two consecutive letters are the same
            pairs.append(text[i] + 'X')
```

```
            i += 1
        else:
            pairs.append(text[i] + text[i + 1])
            i += 2

    return pairs

def encrypt_pair(matrix, pair):
    row1, col1 = find_position(matrix, pair[0])
    row2, col2 = find_position(matrix, pair[1])

    if row1 == row2:  # Same row
        return (matrix[row1][(col1 + 1) % 5] +
                matrix[row2][(col2 + 1) % 5])
    elif col1 == col2:  # Same column
        return (matrix[(row1 + 1) % 5][col1] +
                matrix[(row2 + 1) % 5][col2])
    else:  # Rectangle case
        return matrix[row1][col2] + matrix[row2][col1]

def playfair_encrypt(key, plaintext):
    matrix = create_matrix(key)
    pairs = prepare_text(plaintext)
    ciphertext = ''

    for pair in pairs:
        ciphertext += encrypt_pair(matrix, pair)
```

```
    return ciphertext

if __name__ == "__main__":
    key = "RAKIB"
    plaintext = "ICEDEPARTMENT"

    print("Key:", key)
    print("Original text:", plaintext)

    encrypted = playfair_encrypt(key, plaintext)
    print("\nEncrypted text:", encrypted)
```

Input:

Key: RAKIB

Original text: ICEDEPARTMENT

Output:

Encrypted text: RFFECSKASNFMSY

Number of experiment: 06

Name of experiment: Write a program to implement decryption using Playfair cipher.

Objective:

1) Implement decryption of a message encrypted using the Playfair cipher.
2) Reverse the encryption process using the Playfair cipher rules for pairs of letters.
3) Use the same key to decrypt the ciphertext back into the original plaintext message.

Theory:

The Playfair cipher is a digraph substitution cipher that encrypts pairs of letters (bigrams) rather than single letters. It uses a 5x5 matrix of letters as the key, and the encryption and decryption processes follow specific rules depending on the relative positions of the letters in the matrix.

Decryption Process:

Matrix Construction: The key is used to create a 5x5 matrix. The matrix contains all letters of the alphabet (except for 'J', which is usually merged with 'I'). Any duplicate letters in the key are removed.

Handling the Ciphertext:

- The ciphertext is divided into digraphs (pairs of letters).
- If the ciphertext has an odd number of characters, an additional filler letter (often 'X') is added to complete the last pair.

Decryption Rules:

Same Row: If both letters in the pair are in the same row, each letter is replaced by the letter to its immediate left (circularly, so the left of the first letter in the row is the last letter in the row).

Same Column: If both letters are in the same column, each letter is replaced by the letter immediately above it (again, circularly).

Rectangle Case: If the letters form a rectangle, each letter is replaced by the letter in the same row but in the column of the other letter of the pair.

Algorithm:

1. Take the key and remove any duplicate letters.
2. Construct a 5x5 matrix from the key by filling in the letters of the alphabet (excluding 'J', which is treated as 'I').
3. Convert the ciphertext to uppercase and remove any non-alphabetic characters.
4. Split the ciphertext into digraphs (pairs of letters).
5. If a digraph consists of identical letters, replace the second letter with 'X'.
6. For each digraph, find the position of the letters in the matrix.
    - If the letters are in the same row, replace them with the letters to their immediate left.
    - If the letters are in the same column, replace them with the letters immediately above.
    - If the letters form a rectangle, replace them with the letters that are in the same row but in the columns of the other letter in the pair.
7. Combine the decrypted digraphs to form the plaintext.
8. Return the decrypted plaintext.

Source code:

```
def create_matrix(key):
    key = key.upper().replace('J', 'I')  # Replace J with I as per Playfair rules
    matrix = []
    used_chars = set()
    for char in key:
        if char.isalpha() and char not in used_chars:
            matrix.append(char)
            used_chars.add(char)

    # Fill the matrix with remaining letters of the alphabet (excluding 'J')
    for char in 'ABCDEFGHIKLMNOPQRSTUVWXYZ':  # Note: No J
        if char not in used_chars:
```

```
            matrix.append(char)
            used_chars.add(char)

    # Create a 5x5 matrix
    return [matrix[i:i+5] for i in range(0, 25, 5)]


def find_position(matrix, char):
    # Find the position of the character in the matrix
    for i in range(5):
        for j in range(5):
            if matrix[i][j] == char:
                return i, j
    return None


def decrypt_pair(matrix, pair):
    # Find positions of the two characters in the matrix
    row1, col1 = find_position(matrix, pair[0])
    row2, col2 = find_position(matrix, pair[1])

    if row1 == row2:  # Same row
        return (matrix[row1][(col1 - 1) % 5], matrix[row2][(col2 - 1) % 5])
    elif col1 == col2:  # Same column
        return (matrix[(row1 - 1) % 5][col1], matrix[(row2 - 1) % 5][col2])
    else:  # Rectangle case
        return (matrix[row1][col2], matrix[row2][col1])
```

```
def playfair_decrypt(ciphertext, key):
    ciphertext = ciphertext.upper().replace('J', 'I')
    ciphertext = ''.join(c for c in ciphertext if c.isalpha())  # Remove non-alphabetic chars
    matrix = create_matrix(key)
    decrypted = []

    # Decrypt in pairs
    for i in range(0, len(ciphertext), 2):
        if i + 1 < len(ciphertext):
            pair = decrypt_pair(matrix, (ciphertext[i], ciphertext[i + 1]))
            decrypted.extend(pair)

    decrypted_text = ''.join(decrypted)

    # Remove the trailing 'X' if it exists (added during encryption)
    if decrypted_text.endswith('X'):
        decrypted_text = decrypted_text[:-1]

    return decrypted_text


def main():
    key = "RAKIB"
    ciphertext = "RFFECSKASNFMSY"  # Example ciphertext
    print("Key:", key)
```

```
    print("Ciphertext:", ciphertext)
    decrypted = playfair_decrypt(ciphertext, key)
    print("Decrypted:", decrypted)


if __name__ == "__main__":
    main()
```

Input:

Key: RAKIB

Ciphertext: RFFECSKASNFMSY

Output:

Decrypted: ICEDEPARTMENT

Number of experiment: 07

Name of experiment: Write a program to implement encryption using Poly-Alphabetic cipher.

Objective:

1) Implement encryption using multiple shifting alphabets to increase security.
2) Explore the use of different keys to create variable ciphertexts for the same plaintext.
3) Demonstrate the complexity added by Poly-Alphabetic cipher to protect against frequency analysis.

Theory:

The Poly-Alphabetic cipher is an encryption technique that uses multiple substitution alphabets to increase security. Unlike simple substitution ciphers, which use a single alphabet, poly-alphabetic ciphers switch between several alphabets based on the characters of a repeating key. This method obscures frequency patterns in the plaintext, making the cipher resistant to frequency analysis. In this cipher, each letter in the plaintext is shifted by a different amount, determined by the corresponding letter in the key. When the key is shorter than the plaintext, it repeats to cover the entire message length.

Encryption Equation

For each character in the plaintext:

$$C\ i = (P_i + K_i) \bmod 26$$

Where,

C_i is the i-th character of the ciphertext,

P_i is the i-th character of the plaintext,

K_i is the shift derived from the i-th character of the key,

The position of each character in the alphabet (0 for A, 1 for B, etc.) is used for calculation.

For example, the message is RAKIB and the key is ICE. Now we calculate encrypted message using Poly-Alphabetic cipher.

Since the key "ICE" is shorter than the message, we repeat it to match the length of the message:

Message: "RAKIB"
Key: "ICEIC"

Convert Each Letter to Numerical Positions

Plaintext (RAKIB):
R = 17,,A = 0,,K = 10,,I = 8,,B = 1

Key (ICEIC):
I = 8,,C = 2,,E = 4,,I = 8,,C = 2

For encryption, each character in the plaintext and corresponding character in the key:

$Ci = (P_i + K_i) \bmod 26$

$C1 = (17+8) \bmod 26 = 25 \rightarrow Z$

$C2 = (0+2) \bmod 26 = 2 \rightarrow C$

$C3 = (10+4) \bmod 26 = 14 \rightarrow O$

$C4 = (8+8) \bmod 26 = 16 \rightarrow Q$

$C5 = (1+2) \bmod 26 = 3 \rightarrow D$

The ciphertext for the plaintext "RAKIB" with the key "ICE" is "ZCOQD".

Algorithm:

1. Input the plaintext and key.
2. Repeat the key until its length matches the length of the plaintext.
3. Convert each character in the plaintext and key to its numeric position (A = 0, B = 1, ..., Z = 25).
4. For each character position:
   - Add the plaintext position and key position.
   - Take the result modulo 26 to ensure it wraps within the alphabet.
   - Convert the resulting number back to a letter.
5. Concatenate all encrypted letters to form the ciphertext.

6. Output the final ciphertext.

Source code:

```
# Define the alphabet and mappings
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
mp = dict(zip(alphabet, range(len(alphabet))))
mp2 = dict(zip(range(len(alphabet)), alphabet))


# Function for Poly-Alphabetic Cipher Encryption
def polyalphabetic_encrypt(plaintext, key):
    plaintext = plaintext.upper()  # Ensure plaintext is in uppercase
    key = key.upper()  # Ensure key is in uppercase


    # Repeat the key to match the length of the plaintext
    repeated_key = (key * (len(plaintext) // len(key))) + key[:len(plaintext) %
len(key)]


    ciphertext = ""
    for i in range(len(plaintext)):
        # Get the numeric positions for plaintext and key characters
        pt_pos = mp[plaintext[i]]
        key_pos = mp[repeated_key[i]]


        # Apply the encryption formula
        encrypted_pos = (pt_pos + key_pos) % 26
        ciphertext += mp2[encrypted_pos]  # Convert back to letter and add to
ciphertext
    return ciphertext
```

```
# Main program
plaintext = input("Enter the plaintext: ")
key = input("Enter the key: ")
ciphertext = polyalphabetic_encrypt(plaintext, key)
print("Ciphertext:", ciphertext)
```

Input:

Enter the plaintext: RAKIB

Enter the key: ICE

Output:

Ciphertext: ZCOQD

Number of experiment: 08

Name of experiment: Write a program to implement decryption using Poly-Alphabetic cipher.

Objective:

1) Decrypt Poly-Alphabetic ciphers to retrieve the original message.
2) Use the key to reverse the encryption shifts.
3) Ensure accurate restoration of the original plaintext.

Theory:

The Poly-Alphabetic cipher encryption uses multiple shifting alphabets based on a key. Decryption is the reverse process where the ciphertext is transformed back into the plaintext using the same key.

For decryption, each character of the ciphertext is shifted in the opposite direction of encryption by the corresponding key character. The equation for decryption is:

$$P_i = (C_i - K_i + 26) \bmod 26$$

where:

P_i is the i-th character of the plaintext,

C_i is the i-th character of the ciphertext,

K_i is the shift value from the i-th character of the key,

This formula subtracts the key's shift value from the ciphertext, and if the result is negative, adding 26 ensures it wraps around properly.

For example, Consider the ciphertext "ZCOQD" and key "ICE". We calculate the decrypted message using Poly-Alphabetic cipher.

Since the ciphertext "ZCOQD" has 5 characters, repeat the key "ICE" to match: "ICEIC".

Convert Letters to Positions:

Ciphertext:
Z = 25, C = 2, O = 14, Q = 16, D = 3

Key:
I = 8, C = 2, E = 4, I = 8, C = 2

Apply the Decryption Equation:

$P1=(25-8+26) \bmod 26 = 17 \rightarrow R$

$P2=(2-2+26) \bmod 26 = 0 \rightarrow A$

$P3=(14-4+26) \bmod 26 = 10 \rightarrow K$

$P4=(16-8+26) \bmod 26 = 8 \rightarrow I$

$P5=(3-2+26) \bmod 26 = 1 \rightarrow B$

So, The plaintext is "RAKIB".

Algorithm:

1. Input the ciphertext and the key.
2. Repeat the key to match the length of the ciphertext.
3. Convert each character in the ciphertext and key to their numeric positions (A = 0, B = 1, ..., Z = 25).
4. For each character position in the ciphertext:

- Subtract the key position from the ciphertext position.
- If the result is negative, add 26 to ensure it is within the alphabet range.
- Convert the result back to a letter.

5. Concatenate all decrypted letters to form the plaintext.
6. Output the decrypted plaintext.

Source code:

```
# Define the alphabet and mappings
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
mp = dict(zip(alphabet, range(len(alphabet))))
mp2 = dict(zip(range(len(alphabet)), alphabet))

# Function for Poly-Alphabetic Cipher Decryption
def polyalphabetic_decrypt(ciphertext, key):
    ciphertext = ciphertext.upper()  # Ensure ciphertext is in uppercase
    key = key.upper()  # Ensure key is in uppercase

    # Repeat the key to match the length of the ciphertext
    repeated_key = (key * (len(ciphertext) // len(key))) + key[:len(ciphertext) % len(key)]

    plaintext = ""
    for i in range(len(ciphertext)):
        # Get the numeric positions for ciphertext and key characters
        ct_pos = mp[ciphertext[i]]
        key_pos = mp[repeated_key[i]]
```

```
        # Apply the decryption formula
        decrypted_pos = (ct_pos - key_pos + 26) % 26
        plaintext += mp2[decrypted_pos]  # Convert back to letter and add to plaintext
    return plaintext

# Main program
ciphertext = input("Enter the ciphertext: ")
key = input("Enter the key: ")
plaintext = polyalphabetic_decrypt(ciphertext, key)
print("Plaintext:", plaintext)
```

Input:

Enter the ciphertext: ZCOQD

Enter the key: ICE

Output:

Plaintext: RAKIB

Number of experiment: 09

Name of experiment: Write a program to implement encryption using Vernam cipher.

Objective:

1) Implement encryption using a one-time pad equal in length to the plaintext.
2) Encrypt by performing a bitwise XOR between plaintext and key.
3) Use a random, secret key for secure encryption.

Theory:

The Vernam cipher, also known as the One-Time Pad (OTP), is a symmetric key cipher that is theoretically unbreakable when used with a truly random key that is as long as the plaintext and used only once. It is a stream cipher, meaning it encrypts the message one bit at a time. If the key is truly random, as long as the plaintext, and used only once, the Vernam cipher is unbreakable.

If P is the plaintext bit, K is the key bit, and C is the ciphertext bit:

$$C=P \oplus K$$

Where $\oplus$ denotes the bitwise XOR operation.

Key Generation:

1. The key must be a truly random sequence of bits that is the same length as the plaintext.
2. The key is used only once (hence the term "one-time" pad) and must be kept secret between the sender and receiver.

Encryption:

1. The encryption is performed by applying a bitwise XOR operation between the plaintext and the key.
2. For each corresponding bit in the plaintext and key, the XOR operation is applied to produce the ciphertext

For example, If plain text is RAKIB and key is DTFXE. Now calculate the encrypted message using vernam cipher.

We first need to convert each letter of the plaintext and the key to their binary equivalents.

Plaintext: "RAKIB"

R = 01010010, A = 01000001, K = 01001011, I = 01001001, B = 01000010

Key: "DTFXE"

D = 01000100, T = 01010100, F = 01000110, X = 01011000, E = 01000101

Now, we will apply the XOR operation between the corresponding bits of the plaintext and the key.

R⊕D=00010110 (16 in decimal, which corresponds to the letter "S")

A⊕T=00010101 (**21** in decimal, which corresponds to the letter "U")

K⊕F=00001101 (**13** in decimal, which corresponds to the letter "M")

I⊕X=00010001 (which is **17** in decimal, which corresponds to the letter "Q")

B⊕E=00000111 (which is **7** in decimal, which corresponds to the letter "G")

So,the ciphertext is SUMQG.

Algorithm:

1. Input the plaintext and key.
2. Convert each letter of the plaintext and key to their binary equivalents.
3. Ensure the key is the same length as the plaintext. If not, pad the key with random characters (not recommended for security).
4. For each character in the plaintext, perform the bitwise XOR operation between the corresponding characters from the plaintext and key.
5. Store the result of each XOR operation.
6. Convert the binary result of each XOR operation back to a character.
7. Output the ciphertext.

Source code:

```
import random

alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

mp = dict(zip(alphabet, range(len(alphabet))))

mp2 = dict(zip(range(len(alphabet)), alphabet))
```

```
# Function to generate a random key of given length
def generate_key(length):
    key = ""
    for i in range(length):
        key += random.choice(alphabet)  # Randomly choose a letter from the alphabet
    return key


# Function to encrypt the plaintext using the key
def encrypt(plaintext, key):
    ciphertext = ""
    for i in range(len(plaintext)):
        xor = mp[plaintext[i]] ^ mp[key[i]]  # XOR operation
        ciphertext += mp2[(mp['A'] + xor) % 26]  # Create the ciphertext
    return ciphertext


# Main function
def main():
    plaintext = input("Enter the plain text: ")  # Example plaintext
    plaintext = plaintext.upper()  # Ensure the plaintext is in uppercase
    key = generate_key(len(plaintext))  # Generate a random key with the same length as the plaintext
    ciphertext = encrypt(plaintext, key)  # Encrypt the plaintext using the key
    print("Key:", key)  # Print the generated key
    print("Ciphertext:", ciphertext)  # Print the ciphertext


# Run the program
if __name__ == "__main__":
```

```
main()
```

Input:

Enter the plain text: RAKIB

Output:

Key: AMLJH

Ciphertext: RMBBG

Number of experiment: 10

Name of experiment: Write a program to implement decryption using Vernam cipher.

Objective:

1) Use XOR operation to reverse the encryption process and recover the original message.
2) Ensure the same key used in encryption is applied for decryption.
3) Reconstruct the plaintext by applying XOR between the ciphertext and the key.

Theory:

The Vernam cipher is a symmetric cipher that uses a key of the same length as the plaintext to encrypt and decrypt messages using the XOR (exclusive OR) operation. Since XOR is a reversible operation, the Vernam cipher encryption can be undone by applying the same XOR operation to the ciphertext with the same key, recovering the original plaintext.

Decryption Equation:

The decryption equation for each character in the ciphertext is:

$$P_i = C_i \oplus K_i$$

where:

P_i is the i-th character of the plaintext,

C_i is the i-th character of the ciphertext,

K_i is the i-th character of the key,

$\oplus$ denotes the XOR operation.

Since XOR is its own inverse, applying it again with the same key undoes the encryption.

Suppose we have the ciphertext "SUMQG" and the key "DTFXE".

Convert each character in both the ciphertext and the key to binary.

Ciphertext (SUMQG):
S = 01010011, U = 01010101, M = 01001101, Q = 01010001, G = 01000111

Key (DTFXE):
D = 01000100, T = 01010100, F = 01000110, X = 01011000, E = 01000101

Apply XOR between each pair of ciphertext and key bits to retrieve the plaintext:

S⊕D: 01010011⊕01000100=01010010 (R)

U⊕T: 01010101⊕01010100=01000001 (A)

M⊕F: 01001101⊕01000110=01001011 (K)

Q⊕X: 01010001⊕01011000=01001001 (I)

G⊕E: 01000111⊕01000101=01000010 (B)

The resulting plaintext is "RAKIB", matching the original message.

Algorithm:

1. Define the alphabet and create mappings from letters to positions and back.
2. Take the ciphertext and key as inputs, ensuring they are the same length.
3. Initialize an empty string for the plaintext.
4. For each character position in the ciphertext:
   - Map the ciphertext and key characters to their numeric positions.
   - Apply the XOR operation between the ciphertext and key values.
   - Map the XOR result back to a letter.
   - Append the letter to the plaintext.
5. Output the decrypted plaintext.

Source code:

```
# Dictionary to map letters to numerical positions and vice versa
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
mp = dict(zip(alphabet, range(len(alphabet))))
mp2 = dict(zip(range(len(alphabet)), alphabet))

# Function to decrypt the ciphertext using the key
```

```
def vernam_decrypt(ciphertext, key):
    plaintext = ""
    for i in range(len(ciphertext)):
        # XOR the ciphertext character with the key character to get the original
plaintext character
        xor = mp[ciphertext[i]] ^ mp[key[i]]
        plaintext += mp2[xor % 26]  # Convert result back to a letter
    return plaintext

# Main function
def main():
    ciphertext = input("Enter ciphertext: ")  # Example ciphertext
    key = input("Enter key: ")  # Example key used during encryption

    # Decrypt the ciphertext using the key
    decrypted_text = vernam_decrypt(ciphertext, key)
    print("Decrypted text:", decrypted_text)  # This should match the original
plaintext

# Run the program
if __name__ == "__main__":
    main()
```

Input:

Enter ciphertext: RMBBG

Enter key: AMLJH

Output:

Decrypted text: RAKIB