Experiment No:1

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

Objective:

1. To use the Caesar Cipher algorithm to create a basic cryptographic system.
2. The "Mustofa" text will be encrypted and decrypted using different shift positions.
3. To analyse the patterns of character mutation seen during the encryption process.
4. To confirm the decryption process's precision and dependability.
5. To find out the algorithm's security flaws and assess how susceptible it is to typical assaults.

Theory:

The Caesar Cipher is a traditional substitution cipher that shifts each character in the plaintext by a fixed number of positions within the alphabet, replacing it with a different letter. The shift value serves as the key for both encrypting and decrypting messages. While the simplicity of this cipher makes it easy to learn and implement, it is also quite vulnerable to being broken through modern cryptanalysis techniques. Despite its limitations, it serves as a foundational example in the study of cryptography. Understanding its weaknesses highlights the need for more sophisticated encryption methods. Additionally, it introduces the concept of key-based encryption, which is a crucial aspect of more advanced ciphers. The encryption and decryption are performed using these formulas:

- **Encryption Formula: C** = (P + k) mod 26

  Where P and C represent the plaintext and cipher text respectively, and k is the number of positions to shift.

- **Decryption Formula:** P = (C - k) mod 26

Where P and C represent the plaintext and cipher text respectively, and k is the number of positions to shift.

This cipher is typically not used for secure communication today due to its vulnerability to simple attacks.

Procedure:

1. **Mapping Characters:** The cipher uses the standard English alphabet (A-Z), assigning each letter a specific position ranging from 0 to 25. This mapping forms the basis for the encryption and decryption operations.
2. **Performing Encryption:** The algorithm processes the input message, shifting each letter by a predetermined number of positions according to the key. If the end of the alphabet is reached, the shift continues from the beginning, ensuring all letters remain within the A-Z range.
3. **Carrying Out Decryption:** The decryption process undoes the shift by moving each letter back by the same number of positions, effectively restoring the original plaintext message. Like the encryption, decryption wraps around the alphabet as needed.
4. **Validation and Testing:** To ensure accuracy, the system is tested by applying the same shift key for both encryption and decryption. The output is checked to confirm that the decrypted text matches the initial plaintext.
5. **Handling Edge Cases**: Consider special cases, such as non-alphabetic characters, and decide how the algorithm should treat them, either by ignoring or preserving them. This ensures the system is robust and can handle a variety of inputs.

Python Code for Caesar Cipher

```
        def caesar_encrypt(text, shift):
    encrypted_text = ""
    for char in text:
        if char.isalpha():
            shift_base = ord('A') if char.isupper() else ord('a')
            encrypted_char = chr((ord(char) - shift_base + shift) % 26 + shift_base)
            encrypted_text += encrypted_char
        else:
            encrypted_text += char
    return encrypted_text
def caesar_decrypt(text, shift):
    decrypted_text = " "
    for char in text:
        if char.isalpha():
```

```
            shift_base = ord('A') if char.isupper() else ord('a')
            decrypted_char = chr((ord(char) - shift_base - shift) % 26 + shift_base)
            decrypted_text += decrypted_char
        else:
            decrypted_text += char
    return decrypted_text
text = "Mustofa"
shift = 3
encrypted_text = caesar_encrypt(text, shift)
decrypted_text = caesar_decrypt(encrypted_text, shift)
print("Original Text:", text)
print("Encrypted Text:", encrypted_text)
print("Decrypted Text:", decrypted_text)
```

Algorithm:

1. START
2. Set the INPUT_TEXT to "Mustofa".
3. Set the SHIFT value to 3.
4. Call the CaesarEncryption function with INPUT_TEXT and SHIFT as arguments.
5. Store the result in ENCRYPTED_TEXT.
6. Call the CaesarDecryption function with ENCRYPTED_TEXT and SHIFT as arguments.
7. Store the result in DECRYPTED_TEXT.
8. Display the original INPUT_TEXT.
9. Display the ENCRYPTED_TEXT obtained from the encryption process.
10. Display the DECRYPTED_TEXT obtained from the decryption process.
11. END

Expected Output:

Enter the message: Mustofa

Caesar Cipher Encryption: pxvwrid

Caesar Cipher Decryption: Mustofa

Experiment No:2

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

1. Learn how the Mono-Alphabetic Cipher works by substituting each letter in the plaintext with a corresponding letter from a cipher key.
2. Create a predefined cipher key to map letters for encryption and decryption.
3. Encrypt a message by replacing letters with those from the cipher key.
4. Decrypt the message by reversing the encryption process using the same cipher key.
5. Understand the limitations of the cipher and verify the accuracy of encryption and decryption through practical examples.

Theory:

The Mono-Alphabetic Cipher is an early form of substitution cipher where each character in the plaintext is replaced with a corresponding character from a fixed cipher alphabet. This encryption technique relies on a one-to-one mapping between the standard alphabet and the cipher alphabet.

In this method, the key is a rearranged version of the original alphabet, and the same key is used for both encryption and decryption, making it a symmetric cipher. The simplicity of the Mono-Alphabetic Cipher makes it easy to implement and understand. However, its major flaw lies in its vulnerability to frequency analysis, a method that attackers can use to break the cipher by analysing the frequency of letter occurrences in the ciphertext. This makes it insecure for use in modern cryptography.

Despite its weaknesses, the Mono-Alphabetic Cipher is an important foundational concept in the study of cryptography, illustrating the basic principles of encryption and the concept of a key in securing communications.

Procedure:

1. **Establish the Cipher Key:** Begin by creating a mapping that pairs each letter of the English alphabet (A-Z) with a different letter from a shuffled version of the alphabet. This shuffled alphabet will act as the cipher key, used consistently for both the encryption and decryption processes.
2. **Encryption Method:** During encryption, each character of the plaintext is processed individually. For every letter in the plaintext, find the corresponding letter from the cipher key and replace it. Any non-alphabetic characters, such as spaces or punctuation marks, are left unchanged and retained in their original form.

3. **Decryption Method:** To decrypt the ciphertext, the process is reversed. Use the inverse of the cipher key (the original alphabet) to map each letter of the ciphertext back to its corresponding letter in the standard alphabet, thereby recovering the original plaintext message.
4. **Verification Process:** Test the cipher by encrypting a sample message with the generated cipher key, then decrypt it using the same key to ensure the correctness of the encryption and decryption operations. The decrypted message should match the original plaintext exactly, verifying that the system works as intended.
5. **Edge Case Handling:** Consider handling any potential edge cases, such as repeated characters, unusual symbols, or the presence of capital letters, and decide how these elements should be treated in the encryption and decryption processes. This ensures that the cipher works effectively across different input types.

Python Code for Mono-Alphabetic Cipher

```
class MonoalphabeticCipher:
    def __init__(self):
        self.normal_char = ['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']

        self.coded_char = ['Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I', 'O',
                           'P', 'A', 'S', 'D', 'F', 'G', 'H', 'J', 'K',
                           'L', 'Z', 'X', 'C', 'V', 'B', 'N', 'M']

    def string_encryption(self, s):
        encrypted_string = ""
        for char in s:
            for i in range(26):
                if char == self.normal_char[i]:
                    encrypted_string += self.coded_char[i]
                    break
                elif char < 'a' or char > 'z':
                    encrypted_string += char
                    break
        return encrypted_string

    def string_decryption(self, s):
        decrypted_string = ""
        for char in s:
            for i in range(26):
                if char == self.coded_char[i]:
```

```
                decrypted_string += self.normal_char[i]
                break
            elif char < 'A' or char > 'Z':
                decrypted_string += char
                break
        return decrypted_string


def main():
    cipher = MonoalphabeticCipher()
    plain_text = "I am ICEIAN"

    print("Plain text:", plain_text)

    # Changing the whole string to lowercase
    encrypted_message = cipher.string_encryption(plain_text.lower())
    print("Encrypted message:", encrypted_message)

    decrypted_message = cipher.string_decryption(encrypted_message)
    print("Decrypted message:", decrypted_message)


main()
```

Algorithm:

1. START: Begin the algorithm.
2. Set the value of PLAIN_TEXT to "i am iceian".
3. Display the PLAIN_TEXT to show the original message.
4. Convert the PLAIN_TEXT to lowercase to standardize the input for encryption.
5. Call the String_Encryption function with the PLAIN_TEXT as input.
6. Store the result in ENCRYPTED_TEXT.
7. Display the ENCRYPTED_TEXT to show the encrypted message.
8. Call the String_Decryption function with the ENCRYPTED_TEXT as input.
9. Store the result in DECRYPTED_TEXT.
10. Display the DECRYPTED_TEXT to show the decrypted message, which should match the original PLAIN_TEXT.
11. END: Finish the algorithm.

Expected Output:

Plain text: I am ICEIAN
Encrypted message: O QD OETOQF
Decrypted message: i am iceian

Experiment No:03

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

Objective:

1. To demonstrate how a Brute Force Attack can be applied to decrypt a message encrypted with the Caesar Cipher.
2. To explore how the brute force approach works by testing all possible keys until the original plaintext is recovered.
3. To understand the vulnerabilities of simple encryption methods, like the Caesar Cipher, to brute force attacks.
4. To implement a brute force decryption method for the Caesar Cipher and observe how easily it can break the encryption.

Theory:

The Caesar Cipher is a classical substitution cipher where each letter in the plaintext is shifted by a fixed number of positions in the alphabet. For example, with a shift of 3, 'A' would be replaced by 'D', 'B' by 'E', and so on. While this encryption method is simple and easy to implement, it has a significant weakness: it is highly vulnerable to brute force attacks. In a brute force attack, an attacker systematically tests all possible shift values (from 1 to 25) to decrypt the ciphertext. Since the Caesar Cipher only has 25 possible shift keys, the key space is small, making it computationally feasible to try every shift. For instance, if the ciphertext is "nz obnf jt mlsle" and the key is unknown, a brute force attack will attempt each shift until the original plaintext, "My name is Mustofa", is revealed. The simplicity and limited key space of the Caesar Cipher make it easily breakable through brute force, and as a result, it is not suitable for secure communications in modern cryptography.

Procedure:

1. Encrypt a Message: Start by encrypting a message using a Caesar Cipher with a predetermined shift value. This encrypted message will serve as the ciphertext that will be subjected to the brute force attack.
2. Implement Brute Force Algorithm: Develop a brute force algorithm that systematically tries every possible shift value (from 1 to 25) for the Caesar Cipher. For each shift, decrypt the ciphertext and display the result.
3. Validate Decryption Attempts: After each decryption attempt, assess whether the output forms a coherent, meaningful message. The correct plaintext will be revealed when the proper shift is applied.
4. Test the Brute Force Attack: Apply the brute force attack to multiple ciphertexts to ensure the algorithm works effectively. The algorithm should be able to identify the correct plaintext by trying all 25 possible shift keys.

Python Code for Brute Force Attack on Caesar Cipher:

```
def brute_force_decrypt(ciphertext):
    for shift in range(26):
        decrypted_text = caesar_decrypt(ciphertext, shift)
        print(f"Shift {shift}: {decrypted_text}")


def brute_force_encrypt(plainText):
    for shift in range(26):
        encrypted_text = caesar_encrypt(plainText, shift)
        print(f"Shift {shift}: {encrypted_text}")


def caesar_encrypt(plainText, shift):
    encrypted_text = ""
    for char in plainText:
        if char.isalpha():
            if char.islower():
                encrypted_text += chr((ord(char) + shift - ord('a')) % 26 + ord('a'))
            else:
                encrypted_text += chr((ord(char) + shift - ord('A')) % 26 + ord('A'))
        else:
            encrypted_text += char
    return encrypted_text


def caesar_decrypt(ciphertext, shift):
    decrypted_text = ""
    for char in ciphertext:
        if char.isalpha():
            if char.islower():
                decrypted_text += chr((ord(char) - shift - ord('a')) % 26 + ord('a'))
```

```
            else:
                decrypted_text += chr((ord(char) - shift - ord('A')) % 26 + ord('A'))
        else:
            decrypted_text += char
    return decrypted_text


plantext='hello'
print('Brute Force Encryption for Caesar Cipher:')
brute_force_encrypt(plantext)

ciphertext = "ifmmp"
print("\nBrute Force Decryption for Caesar Cipher:")
brute_force_decrypt(ciphertext)
```

Algorithm:

1. START: Begin the algorithm.
2. GET the plaintext from the user input, which will be encrypted and decrypted using the Caesar Cipher.
3. DISPLAY the message "=== ENCRYPTION RESULTS ===" to indicate the start of the encryption process.
4. DISPLAY the original plaintext to show the input message.
5. CALL the Brute_Force_Encrypt function with the plaintext as input to initiate the encryption process.
6. ENCRYPT the plaintext using the Caesar Cipher with a shift value of 1, and continue for all possible shift values (1 to 25).
7. DISPLAY the message "=== DECRYPTION RESULTS ===" to signal the start of the decryption process.
8. DISPLAY the encrypted text for the shift value of 1, showing the result of the first encryption attempt.
9. CALL the Brute_Force_Decrypt function with the encrypted ciphertext to attempt to decrypt it using all possible shifts (1 to 25).
10. END: Finish the algorithm.

Input:

Enter the text to encrypt: hello

Output:

Brute Force Encryption for Caesar Cipher:
Shift 0: hello
Shift 1: ifmmp
Shift 2: jgnnq
Shift 3: khoor

Shift 4: lipps
Shift 5: mjqqt
Shift 6: nkrru
Shift 7: olssv
Shift 8: pmttw
Shift 9: qnuux
Shift 10: rovvy
Shift 11: spwwz
Shift 12: tqxxa
Shift 13: uryyb
Shift 14: vszzc
Shift 15: wtaad
Shift 16: xubbe
Shift 17: yvccf
Shift 18: zwddg
Shift 19: axeeh
Shift 20: byffi
Shift 21: czggj
Shift 22: dahhk
Shift 23: ebiil
Shift 24: fcjjm
Shift 25: gdkkn

Brute Force Decryption for Caesar Cipher:
Shift 0: ifmmp
Shift 1: hello
Shift 2: gdkkn
Shift 3: fcjjm
Shift 4: ebiil
Shift 5: dahhk
Shift 6: czggj
Shift 7: byffi
Shift 8: axeeh
Shift 9: zwddg
Shift 10: yvccf
Shift 11: xubbe
Shift 12: wtaad
Shift 13: vszzc
Shift 14: uryyb
Shift 15: tqxxa
Shift 16: spwwz
Shift 17: rovvy
Shift 18: qnuux
Shift 19: pmttw

Shift 20: olssv
Shift 21: nkrru
Shift 22: mjqqt
Shift 23: lipps
Shift 24: khoor
Shift 25: jgnnq

Experiment No:4

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

1. Implement the Hill cipher technique for encrypting and decrypting messages using matrix operations.
2. Understand how matrix multiplication and inversion are applied in cryptographic encryption and decryption.
3. Convert plaintext into ciphertext using a square matrix as the encryption key.
4. Decrypt ciphertext back to plaintext using the inverse of the encryption matrix.
5. Learn how the Hill cipher offers more security than basic substitution ciphers by processing text in blocks.
6. Test the encryption and decryption processes to verify their correctness and security.

Theory

The Hill cipher, introduced by Lester S. Hill in 1929, is a cryptographic method that relies on linear algebra to encrypt blocks of text using matrix multiplication. The encryption key is a square matrix, such as a 2x2 or 3x3 matrix, and the plaintext message is divided into blocks that match the size of the matrix. Each block is then multiplied by the key matrix to produce the ciphertext.

In the encryption process, each letter in the plaintext is converted into a numerical value, with A = 0, B = 1, ..., Z = 25. The message is then split into vectors that correspond to the matrix size. These vectors are multiplied by the key matrix, and the resulting numbers are converted back to letters to form the ciphertext.

For decryption, the inverse of the key matrix is used. The ciphertext is multiplied by the inverse matrix, and the resulting product is the original plaintext message. This cipher is more secure than basic substitution ciphers because it encrypts entire blocks of text at once, rather than individual letters, which makes it harder to break using simple frequency analysis.

Source Code Implementation (Python):

```
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())
```

Algorithm:

1. **START**
2. **GET** the plaintext message and key.
3. **CREATE** a 2*2 key matrix from the key string.
4. **CONVERT** the plaintext into a vector of numbers (A=0, B=1, ..., Z=25).
5. **ENCRYPT** the plaintext by multiplying the key matrix with the message vector and applying modulo 26.
6. **CONVERT** the resulting cipher matrix back to characters.
7. **DISPLAY** the ciphertext.
8. **END**

Experiment No:5

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

Objective:

1. To implement encryption using the Playfair Cipher, a more secure method than classical ciphers like the Caesar Cipher.
2. To encrypt pairs of letters from a plaintext message by using a keyword to form a 5x5 matrix for substitution.
3. To explore the benefits of the Playfair Cipher in providing stronger encryption through digraphs (pairs of letters) rather than single letters.

Theory:

The Playfair Cipher, invented by Charles Wheatstone and later popularized by Lord Playfair, is a digraph substitution cipher that encrypts pairs of letters (digraphs) rather than individual letters. This method provides enhanced security compared to traditional ciphers like the Caesar Cipher. The encryption process begins with the creation of a 5x5 key matrix, which is generated from a keyword. This matrix contains the entire alphabet, except for the letter 'J', which is usually combined with 'I' to fit the 25 available spaces. The keyword is written into the matrix first, and the remaining letters of the alphabet are filled in sequentially.

Once the key matrix is established, the plaintext is divided into pairs of letters, or digraphs. If any pair consists of identical letters, an 'X' is inserted between them to differentiate the letters. If the plaintext contains an odd number of characters, an 'X' is added at the end to complete the final pair. Each pair of letters is then encrypted based on its position in the key matrix, following specific rules. If both letters of a pair are in the same row of the matrix, they are replaced by the letters immediately to their right, wrapping around to the beginning of the row if necessary. If the letters are in the same column, they are replaced with the letters directly below, wrapping to the top if required. Finally, if the two letters form a rectangle in the matrix, each letter is replaced by the one in the same row but in the opposite column, effectively mirroring the pair.

This approach of using digraphs and matrix-based encryption offers a more complex and secure method compared to simpler substitution ciphers, making the Playfair Cipher a stronger encryption technique.

Procedure:

1.Prepare the Key Matrix:

- Choose a keyword and remove any duplicate letters.
- Create a 5x5 matrix using the keyword, and then fill in the remaining letters of the alphabet (excluding 'J').

2.Prepare the Plaintext:

- Divide the plaintext into digraphs (pairs of letters).
- f a pair has repeated letters, replace the second one with 'X'.
- If the plaintext has an odd number of letters, append 'X' at the end.

3.Encryption:

- For each digraph, find the letters in the key matrix.
- Apply the encryption rules to transform the digraph into ciphertext.

4. Output the Ciphertext:

- After all digraphs have been encrypted, display the resulting ciphertext.

Python Implementation

```
def playfair_cipher(plaintext, key, mode):
    # Define the alphabet, excluding 'j'
    alphabet = 'abcdefghiklmnopqrstuvwxyz'
    # Remove whitespace and 'j' from the key and convert to lowercase
    key = key.lower().replace(' ', '').replace('j', 'i')
    # Construct the key square
    key_square = ''
    for letter in key + alphabet:
        if letter not in key_square:
            key_square += letter
    # Split the plaintext into digraphs, padding with 'x' if necessary
    plaintext = plaintext.lower().replace(' ', '').replace('j', 'i')
    replaceplaintext = ''
    if mode == 'encrypt':
        it = 0
        while it < len(plaintext) - 1:
            if plaintext[it] == plaintext[it + 1]:
                replaceplaintext += plaintext[it]
                replaceplaintext += 'x'
                it += 1
            else:
                replaceplaintext += plaintext[it]
                replaceplaintext += plaintext[it + 1]
                it += 2
        replaceplaintext += plaintext[-1] if it < len(plaintext) else ''
        plaintext = replaceplaintext

    if len(plaintext) % 2 == 1:
        plaintext += 'x'
    digraphs = [plaintext[i:i + 2] for i in range(0, len(plaintext), 2)]

    # Define the encryption/decryption functions
    def encrypt(digraph):
        a, b = digraph
        row_a, col_a = divmod(key_square.index(a), 5)
        row_b, col_b = divmod(key_square.index(b), 5)
        if row_a == row_b:
            col_a = (col_a + 1) % 5
            col_b = (col_b + 1) % 5
        elif col_a == col_b:
            row_a = (row_a + 1) % 5
            row_b = (row_b + 1) % 5
        else:
            col_a, col_b = col_b, col_a
```

```
        return key_square[row_a * 5 + col_a] + key_square[row_b * 5 + col_b]

    def decrypt(digraph):
        a, b = digraph
        row_a, col_a = divmod(key_square.index(a), 5)
        row_b, col_b = divmod(key_square.index(b), 5)
        if row_a == row_b:
            col_a = (col_a - 1) % 5
            col_b = (col_b - 1) % 5
        elif col_a == col_b:
            row_a = (row_a - 1) % 5
            row_b = (row_b - 1) % 5
        else:
            col_a, col_b = col_b, col_a
        return key_square[row_a * 5 + col_a] + key_square[row_b * 5 + col_b]

    # Encrypt or decrypt the plaintext
    result = ''
    for digraph in digraphs:
        if mode == 'encrypt':
            result += encrypt(digraph)
        elif mode == 'decrypt':
            result += decrypt(digraph)

    # Return the result
    return result


# Example usage
plaintext = 'mustofaj'
key = 'monkey'
ciphertext = playfair_cipher(plaintext, key, 'encrypt')
print('Cipher Text:', ciphertext)
decrypted_text = playfair_cipher(ciphertext, key, 'decrypt')
print('Decrypted Text:', decrypted_text)  # (Note: 'x' is added as padding)
```

Expected Output

Key: monkey
Plaintext: mustofa
Ciphertext: ymtpmgcv

Experiment No:6

Experiment Name: Write a program to implement Decryption using Playfair cipher.

Objective:

1. Implement the decryption process for the Playfair Cipher to convert ciphertext back into plaintext.
2. Use a 5x5 key matrix for decrypting messages.
3. Apply specific decryption rules based on the matrix to reverse the encryption process.
4. Recover the original plaintext message by applying the decryption algorithm.
5. Verify the correctness of the decryption by ensuring the ciphertext matches the original message.

Theory:

The Playfair Cipher is a digraph substitution cipher that encrypts messages by pairing letters in the plaintext and using a 5x5 key matrix. For decryption, the process involves reversing the encryption steps using the same key matrix.

The key matrix is constructed from a keyword, where each letter of the alphabet (except 'J', which is combined with 'I') is placed in a 5x5 grid. The keyword is written first, followed by the remaining unused letters of the alphabet.

In Playfair decryption, the ciphertext is split into pairs of letters (digraphs). The rules for decryption differ slightly from encryption. If both letters in a pair are in the same row, they are replaced by the letters immediately to their left, wrapping around if necessary. If both letters are in the same column, they are replaced by the letters immediately above, again wrapping to the top if needed. If the letters form a rectangle, each letter is replaced by the one in the same row but in the opposite column, mirroring the original encryption.

If the ciphertext contains an odd number of characters, an 'X' is added at the end during encryption and must be removed in decryption if it does not belong to the original message. Additionally, if identical letters appear in a pair, an 'X' is inserted between them during encryption and should be removed during decryption.

Playfair decryption thus involves reversing the steps of encryption using the same key matrix, applying these rules to recover the original plaintext message securely.

Procedure:

Prepare the Key Matrix: Construct the 5x5 key matrix from the chosen keyword. Fill the matrix with the keyword first, followed by the remaining unused letters of the alphabet (excluding 'J', which is typically combined with 'I').

Prepare the Ciphertext: Split the ciphertext into pairs of letters (digraphs). If the ciphertext has an odd number of characters, add an 'X' to the end. If a pair has identical letters (e.g., "LL"), insert an 'X' between them during encryption, and those 'X's should be removed during decryption.

Decryption Rules: For each pair of letters in the ciphertext, apply the following decryption rules:

Same Row: If both letters of the pair are in the same row of the key matrix, replace them with the letters immediately to their left. If necessary, wrap around to the end of the row.

Same Column: If both letters of the pair are in the same column, replace them with the letters directly above them. If necessary, wrap around to the top of the column. Rectangle Formation: If the letters form a rectangle in the matrix, replace each letter with the letter in the same row but in the opposite column (mirrored in the rectangle).Apply the decryption rules to each pair in the ciphertext until all digraphs are processed.

Remove Padding (if any): If an 'X' was added during encryption to fill an odd length ciphertext or to separate identical letters, remove it from the decrypted message if it does not belong to the original plaintext.

Output the Plaintext: After applying the decryption rules to all digraphs and removing any unnecessary padding, the original plaintext message is retrieved.

Source Code Implementation for Decryption:

```
def playfair_cipher(plaintext, key, mode):
    # Define the alphabet, excluding 'j'
    alphabet = 'abcdefghiklmnopqrstuvwxyz'
    # Remove whitespace and 'j' from the key and convert to lowercase
    key = key.lower().replace(' ', '').replace('j', 'i')
    # Construct the key square
    key_square = ''
    for letter in key + alphabet:
        if letter not in key_square:
            key_square += letter
    # Split the plaintext into digraphs, padding with 'x' if necessary
    plaintext = plaintext.lower().replace(' ', '').replace('j', 'i')
    replaceplaintext = ''
    if mode == 'encrypt':
        it = 0
        while it < len(plaintext) - 1:
            if plaintext[it] == plaintext[it + 1]:
                replaceplaintext += plaintext[it]
                replaceplaintext += 'x'
                it += 1
```

```
        else:
            replaceplaintext += plaintext[it]
            replaceplaintext += plaintext[it + 1]
            it += 2
    replaceplaintext += plaintext[-1] if it < len(plaintext) else ''
    plaintext = replaceplaintext

if len(plaintext) % 2 == 1:
    plaintext += 'x'
digraphs = [plaintext[i:i + 2] for i in range(0, len(plaintext), 2)]

# Define the encryption/decryption functions
def encrypt(digraph):
    a, b = digraph
    row_a, col_a = divmod(key_square.index(a), 5)
    row_b, col_b = divmod(key_square.index(b), 5)
    if row_a == row_b:
        col_a = (col_a + 1) % 5
        col_b = (col_b + 1) % 5
    elif col_a == col_b:
        row_a = (row_a + 1) % 5
        row_b = (row_b + 1) % 5
    else:
        col_a, col_b = col_b, col_a
    return key_square[row_a * 5 + col_a] + key_square[row_b * 5 + col_b]

def decrypt(digraph):
    a, b = digraph
    row_a, col_a = divmod(key_square.index(a), 5)
    row_b, col_b = divmod(key_square.index(b), 5)
    if row_a == row_b:
        col_a = (col_a - 1) % 5
        col_b = (col_b - 1) % 5
    elif col_a == col_b:
        row_a = (row_a - 1) % 5
        row_b = (row_b - 1) % 5
    else:
        col_a, col_b = col_b, col_a
    return key_square[row_a * 5 + col_a] + key_square[row_b * 5 + col_b]

# Encrypt or decrypt the plaintext
result = ''
for digraph in digraphs:
    if mode == 'encrypt':
        result += encrypt(digraph)
    elif mode == 'decrypt':
        result += decrypt(digraph)

# Return the result
return result
```

```
# Example usage
plaintext = 'mustofa'
key = 'monkey'
ciphertext = playfair_cipher(plaintext, key, 'encrypt')
print('Cipher Text:', ciphertext)
decrypted_text = playfair_cipher(ciphertext, key, 'decrypt')
print('Decrypted Text:', decrypted_text)  # (Note: 'x' is added as padding)
```

Expected Output:

Key: monkey

Ciphertext: ymtpmgcv

Decrypted Message: mustofax

Experiment No:7

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

Objective:

1. Implement encryption using the Poly-Alphabetic Cipher, a type of substitution cipher.
2. Shift each letter of the plaintext based on a repeating key sequence.
3. Use multiple alphabets to strengthen the cipher's security.
4. Provide resistance to frequency analysis through polyalphabetic shifts.
5. Enhance security compared to simple substitution ciphers, like the Caesar Cipher.

Theory:

Polyalphabetic ciphers employ a sophisticated encryption method that involves using multiple substitution alphabets. This enhances security by obfuscating patterns that could be exploited by cryptanalysis.

The Vigenère Cipher: A Classic Example

A prominent example of a polyalphabetic cipher is the Vigenère cipher. This technique involves a repeating key that determines the shift for each letter in the plaintext. The encryption process unfolds as follows:

1. **Numerical Conversion:**
    - Each letter in the plaintext is assigned a numerical value, typically from 0 to 25, where A=0, B=1, and so on.
2. **Key-Based Shifting:**
    - The key is repeated to match the length of the plaintext.
    - Each plaintext letter is shifted by the corresponding letter in the repeated key. For instance, if the key letter is 'C' (which has a numerical value of 2), the plaintext letter is shifted two positions forward in the alphabet.
3. **Ciphertext Generation:**
    - The shifted letters are converted back to their corresponding characters, forming the ciphertext.

Breaking Down the Encryption Process:

1. **Input Acquisition:**
    - The plaintext message and the encryption key are obtained from the user.
2. **Key Repetition:**
    - If the key is shorter than the plaintext, it is repeated to match the plaintext's length.
3. **Character-by-Character Encryption:**
    - For each character in the plaintext:
        - The character is converted to its numerical equivalent.
        - The corresponding key character is used to determine the shift amount.
        - The plaintext character is shifted by the specified amount.

4. **Ciphertext Construction:**
    - The shifted characters are converted back to letters, forming the final ciphertext.

By employing multiple substitution alphabets, polyalphabetic ciphers significantly increase the complexity of cryptanalysis compared to simple substitution ciphers. However, with advancements in cryptanalysis techniques, robust key management and secure communication protocols are essential to maintain the security of these encryption methods.

Code Implementation for Poly-Alphabetic Cipher Encryption:

```
Poly_alphabetic cipher
alphabet = "abcdefghijklmnopqrstuvwxyz".upper()
mp = dict(zip(alphabet, range(len(alphabet))))
mp2 = dict(zip(range(len(alphabet)), alphabet))


def generateKey(plainText, keyword):
    key = ''
    for i in range(len(plainText)):
        key += keyword[i % len(keyword)]
    return key

def cipherText(plainText, key):
    cipher_text = ""
    for i in range(len(plainText)):
        shift = mp[key[i].upper()] - mp['A']
        newChar = mp2[(mp[plainText[i].upper()] + shift) % 26]
        cipher_text += newChar
    return cipher_text


def decrypt(cipher_text, key):
    plainText = ''
    for i in range(len(cipher_text)):
        shift = mp[key[i].upper()] - mp['A']
        newChar = mp2[(mp[cipher_text[i].upper()] - shift + 26) % 26]
        plainText += newChar
    return plainText


plainText = "wearediscoveredsaveyourself"
keyword = "deceptive"
key = generateKey(plainText, keyword)
cipher_text = cipherText(plainText, key)
print("Ciphertext :", cipher_text)
print("Decrypted Text :", decrypt(cipher_text, key))
```

Sample Output:

For the input:

- **Plaintext**: wearediscoveredsaveyourself
- **Key**: deceptive

The output will be:

Plaintext: wearediscoveredsaveyourself
Key: deceptive
Encrypted Message: ZICVTWQNGRZGVTWAVZHCQYGLMGJ

Experiment No:8

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

Objective:

1. Implement decryption using the Poly-Alphabetic Cipher, a substitution cipher with a repeating key.
2. Use the key sequence to determine the decryption shift for each character.
3. Reverse the encryption process to recover the original plaintext message.
4. Successfully decrypt the ciphertext to retrieve the original message.

Theory:

The Poly-Alphabetic Cipher is a form of substitution cipher in which each letter in the plaintext is encrypted using a different letter from a set of multiple alphabets, which are determined by a repeating key. Unlike simpler ciphers like the Caesar Cipher, which shifts all letters by the same amount, the Poly-Alphabetic Cipher provides greater security by applying a unique shift for each letter of the message.

The key in a Poly-Alphabetic Cipher is typically a word or phrase, which repeats itself throughout the plaintext. Each letter of the key is used to determine the shift for the corresponding letter in the plaintext. For example, if the key is "KEY", then the first letter of the plaintext is shifted by the value corresponding to 'K', the second by 'E', and the third by 'Y', and so on.

To encrypt, the plaintext letters are shifted forward by the number corresponding to the letter in the key. For decryption, the same key is used, but each letter of the ciphertext is shifted backwards by the key's corresponding letter value. This two-way encryption and decryption process enhances security by preventing frequency analysis attacks, making it much harder to break than monoalphabetic ciphers.

Overall, the Poly-Alphabetic Cipher significantly improves upon simpler ciphers by introducing variability in the shifts, providing stronger encryption and making it resistant to basic cryptographic attacks like frequency analysis.The decryption process involves:

1. Converting each character in the ciphertext back to a number (A=0, B=1, ..., Z=25).
2. Shifting each character by subtracting the corresponding letter in the key (also converted to a number).
3. Converting the result back to letters to form the decrypted message.

Procedure:

1. Prepare the Key Sequence:
2. Repeat the key sequence so that it matches the length of the ciphertext.
3. Convert Ciphertext to Letters:
4. Convert the ciphertext into letters (if it's in any other form, such as numbers, convert them to letters using a standard mapping like A=0, B=1, etc.).
5. For Each Character in the Ciphertext:
6. For each letter in the ciphertext, apply the decryption process using the corresponding letter of the key.
7. Shift the Ciphertext Letter Back:
8. Use the key letter to determine the shift and subtract the shift value from the ciphertext letter's position in the alphabet.
9. Handle Alphabet Wrap-Around:
10. If the subtraction results in a negative number, wrap it around to the end of the alphabet.
11. Repeat for All Characters:
12. Repeat the decryption for each letter in the ciphertext, using the corresponding letter from the repeated key sequence.
13. Reconstruct the Plaintext:
14. After applying the decryption rule for every letter, you will recover the original plaintext message.

Source Code Implementation for Poly-Alphabetic Cipher Decryption:

```
# Poly_alphabetic cipher

alphabet = "abcdefghijklmnopqrstuvwxyz".upper()
mp = dict(zip(alphabet, range(len(alphabet))))
mp2 = dict(zip(range(len(alphabet)), alphabet))


def generateKey(plainText, keyword):
    key = ''
    for i in range(len(plainText)):
        key += keyword[i % len(keyword)]
    return key


def cipherText(plainText, key):
    cipher_text = ""
    for i in range(len(plainText)):
        shift = mp[key[i].upper()] - mp['A']
        newChar = mp2[(mp[plainText[i].upper()] + shift) % 26]
        cipher_text += newChar
    return cipher_text


def decrypt(cipher_text, key):
    plainText = ''
    for i in range(len(cipher_text)):
```

```
        shift = mp[key[i].upper()] - mp['A']
        newChar = mp2[(mp[cipher_text[i].upper()] - shift + 26) % 26]
        plainText += newChar
    return plainText


plainText = "wearediscoveredsaveyourself"
keyword = "deceptive"
key = generateKey(plainText, keyword)
cipher_text = cipherText(plainText, key)
print("Ciphertext :", cipher_text)
print("Decrypted Text :", decrypt(cipher_text, key))
```

Sample Output:

For the input:

- **Ciphertext**: ZICVTWQNGRZGVTWAVZHCQYGLMGJ
- **Key**: deceptive

The output will be:

Ciphertext: ZICVTWQNGRZGVTWAVZHCQYGLMGJ
Key: deceptive
Decrypted Message: wearediscoveredsaveyourself

Experiment No:9

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

Objective:

The objective of this lab is to implement encryption using the **Vernam Cipher** (also known as the One-Time Pad cipher). This cipher employs a unique, random key equal in length to the plaintext to create ciphertext, ensuring that each encryption is unique and theoretically unbreakable if the key is used only once.

Theory:

The **Vernam Cipher** is a symmetric encryption technique where each character in the plaintext is combined with a corresponding character in the key using a bitwise XOR operation. This operation produces a unique encrypted character that depends on both the plaintext and the key. The Vernam Cipher is unbreakable when:

1. The key is truly random.
2. The key length matches the plaintext length.
3. The key is used only once.

In this method:

1. Each character in the plaintext is XORed with the corresponding character in the key.
2. This XOR operation yields an encrypted character, producing the ciphertext.

Procedure:

1 Input the Plaintext and Key:

- Ensure the key length is equal to the plaintext length.

2. **Convert Plaintext and Key Characters to Numeric Values:**

- Map each character to its corresponding ASCII value or its position in the alphabet.

3. **Perform XOR Operation on Each Character:**

- For each character, XOR the plaintext character with the corresponding key character.
- Convert the result back to a character.

4. **Generate Plaintext:**

- Combine all encrypted characters to form the final ciphertext.

Code Implementation for Vernam Cipher Encryption:

```
def stringEncryption(text, key):
    cipherText = ""
    cipher = []
    for i in range(len(key)):
        cipher.append(ord(text[i]) - ord('A') + ord(key[i]) - ord('A'))
    for i in range(len(key)):
        if cipher[i] > 25:
            cipher[i] = cipher[i] - 26
    for i in range(len(key)):
        x = cipher[i] + ord('A')
        cipherText += chr(x)
    return cipherText
plainText = "howareyou"
key = "ncbtzqarx"
encryptedText = stringEncryption(plainText.upper(), key.upper())
print("Cipher Text - " + encryptedText)
```

Sample Output:

For the input:

- **Plaintext**: howareyou
- **Key**: ncbtzqarx

The output will be:

Plaintext: howareyou
Key: ncbtzqarx
Cipher Text: UQXTQUY

Experiment No:10

Experiment Name: Write a program to implement Deycryption using Vernam cipher.

Objective:

The objective of this lab is to implement decryption using the **Vernam Cipher**. This cipher, also known as the One-Time Pad, is a symmetric encryption technique that uses a unique, random key equal in length to the ciphertext for decryption. When the key is known, this method provides a theoretically unbreakable means of recovering the original message.

Theory:

The **Vernam Cipher** relies on a bitwise XOR operation to encrypt and decrypt messages. The XOR operation is reversible, meaning that if a character in the plaintext is XORed with a character in the key to produce the ciphertext, then XORing the ciphertext character with the same key character will recover the original plaintext character. This decryption process relies on:

1. The ciphertext and key lengths being equal.
2. The key being known and used only once.

The decryption process involves:

1. Each character in the ciphertext is XORed with the corresponding character in the key.
2. This operation yields the original plaintext character.

Procedure:

1 Input the Ciphertext and Key:

- Ensure the key length matches the ciphertext length.

2. **Convert Cipertext and Key Characters to Numeric Values:**

- Map each character to its corresponding position in the alphabet.

3. **Perform XOR Operation on Each Character:**

- XOR each character in the ciphertext with the corresponding character in the key..
- Convert the result back to a character.

4. **Construct the Plaintext:**

- Combine all decrypted characters to form the plaintext.

Code Implementation for Vernam Cipher Decryption:

```
def stringDecryption(s, key):
    plainText = ""
    plain = []
    s = s.upper()
    key = key.upper()
    for i in range(len(key)):
        plain.append(ord(s[i]) - ord('A') - (ord(key[i]) - ord('A')))
    for i in range(len(key)):
        if plain[i] < 0:
            plain[i] = plain[i] + 26
    for i in range(len(key)):
        x = plain[i] + ord('A')
        plainText += chr(x)
    return plainText
cipherText = "uqxtquyfr"  # Encrypted text from the encryption script output
key = "ncbtzqarx"
decryptedText = stringDecryption(cipherText, key)
print("Decrypted Message - " + decryptedText)
```

Sample Output:

For the input:

- **Ciphertext**: uqxtquyfr
- **Key**: ncbtzqarx

The output will be:

Ciphertext: uqxtquyfr
Key: ncbtzqarx
Decrypted Message: HOWAREYOU