INDEX

Experiment Number: 01

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

Objective: The primary goal of the Caesar cipher is to offer a basic yet effective method for encoding messages, making them secure by shifting each letter in the original message (plaintext) by a set number of positions in the alphabet. This results in an encrypted message (ciphertext) that can only be deciphered if the shift value is known. The main objectives of this technique include:

- **Secrecy:** To keep information hidden from unauthorized parties.
- **Ease of Use:** To provide a simple encryption method that is easy to apply and understand.
- **Basic Security:** To obscure the original message, though the cipher is relatively weak by today's cryptographic standards.

Theory: The Caesar cipher is a simple encryption method that derives its name from Julius Caesar, who reportedly used it to secure his messages with his generals. As a substitution cipher, it shifts each letter in the plaintext by a fixed number of positions in the alphabet, generating the encrypted ciphertext. This shift value, usually a small integer, is applied uniformly across the entire message.

Key Components:

1. **Shift Key:** The number of positions each character in the plaintext is moved.
2. **Plaintext:** The original, unmodified message.
3. **Ciphertext:** The encrypted message after the shift.

For example, with a shift of 3:

- "A" becomes "D"
- "B" becomes "E"
- "X" wraps around to "A"

The formula for encryption can be written as: $C = E\ (k, p) = (p + k) mod\ 26$where k is the shift value, and $k$pis the position of the plaintext letter.

Encryption Process:

To encrypt a message with the Caesar cipher:

1. **Choose a Shift Value:** Typically, a number between 1 and 25.
2. **Apply the Shift:**
    - For each letter in the plaintext, determine if it's uppercase or lowercase.
    - Move each letter forward by the shift value, wrapping from "Z" to "A" as needed.

- Non-alphabet characters remain unchanged.

3. **Form the Ciphertext:** Combine all shifted letters into the final encrypted message.

Example of Encryption:

- Plaintext: "Sayeeda Khan"
- Shift: 4
- Ciphertext: " Weciihe Oler " (each letter shifted forward by three positions)

Decryption Process:

To decrypt a message encrypted with the Caesar cipher:

1. **Use the Same Shift Key:** Apply the same shift in reverse.
2. **Shift Backwards:**
   - For each letter in the ciphertext, shift it back by the same amount within the alphabet.
3. **Reconstruct the Plaintext:** Combine the letters to return to the original message.

Example of Decryption:

- Ciphertext: " Weciihe Oler "
- Shift: 3
- Plaintext: "Sayeeda Khan" (each letter shifted back by three positions)

Code:

```
def encrypt(text, shift):
    result = ""
    for char in text:
        if char.isalpha():
            # Shift letter within alphabet
            offset = 65 if char.isupper() else 97
            result += chr((ord(char) - offset + shift) % 26 + offset)
        else:
            result += char  # Non-alphabetic characters remain the same
    return result

# Decrypt with Caesar Cipher
def decrypt(text, shift):
    return encrypt(text, -shift)


# Get user input for plaintext and shift
text = input("Enter the plain text: ")
shift = int(input("Enter the shift key (1-25): "))

# Encrypt the text
encrypted_text = encrypt(text, shift)
print("Encrypted Text:", encrypted_text)
```

```
# Decrypt the text back using known shift
decrypted_text = decrypt(encrypted_text, shift)
print("Decrypted Text:", decrypted_text)
```

Input and Output:

```
Enter the plain text: Sayeeda Khan
Enter the shift key (1-25): 4
Encrypted Text: Weciihe Oler
Decrypted Text: Sayeeda Khan
```

Experiment Number: 02

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

Objective: The goal of the monoalphabetic cipher is to improve encryption security compared to simpler ciphers, like the Caesar cipher, by substituting each letter of the plaintext with a distinct letter from a scrambled version of the alphabet. This creates a more intricate mapping, making the cipher more difficult to decipher without the correct key. The key objectives include:

- **Stronger Security:** To provide a more secure encryption method by ensuring each letter is replaced with a different character, making it harder to crack than simpler shift-based ciphers.
- **Confidentiality:** To safeguard the message from unauthorized access by transforming the plaintext with a random alphabet arrangement.
- **Ease of Use:** To offer a straightforward encryption method that remains simple to apply, even though it provides a higher level of security.

Theory: A monoalphabetic cipher is a type of substitution cipher where each letter in the plaintext is replaced by a distinct letter from a scrambled version of the alphabet. Unlike the Caesar cipher, which applies a fixed shift to each letter, monoalphabetic ciphers involve a more intricate and varied mapping, making them harder to anticipate.

Key Characteristics of Monoalphabetic Ciphers:

1. **Fixed Substitution:**
    - Each letter of the plaintext is consistently replaced with a unique letter from the cipher alphabet, creating a fixed rule for substitution.
2. **Example Mapping:**
    - For instance, the plaintext alphabet could be mapped to the cipher alphabet like this:
        - Plain alphabet: ABCDEFGHIJKLMNOPQRSTUVWXYZ
        - Cipher alphabet: QWERTYUIOPASDFGHJKLZXCVBNM
    - This means "A" in the plaintext would be encoded as "Q," "B" as "W," and so on.
3. **Key (Substitution Alphabet):**
    - The "key" is the scrambled alphabet used for substitution.
    - With 26 letters, there are 26! possible keys, providing more security than the Caesar cipher if a random key is used.
4. **Encryption and Decryption:**
    - **Encryption:** Replace each letter in the plaintext with the corresponding letter from the cipher alphabet.

- **Decryption:** Reverse the process by finding each letter in the cipher alphabet and mapping it back to the original plaintext letter.

5. **Security Considerations:**
   - Although more secure than the Caesar cipher, monoalphabetic ciphers remain vulnerable to frequency analysis. Common letters in the plaintext (such as "E" in English) will appear with a similar frequency in the ciphertext, offering hints to potential attackers.

Pseudo Code:

```
def create_cipher_key():
    import string
    import random
    alphabet = string.ascii_lowercase
    shuffled = list(alphabet)
    random.shuffle(shuffled)
    cipher_key = dict(zip(alphabet, shuffled))
    return cipher_key

def encrypt(plaintext, cipher_key):
    ciphertext = ''
    for char in plaintext:
        if char.isalpha():
            char_lower = char.lower()
            cipher_char = cipher_key[char_lower]
            if char.isupper():
                ciphertext += cipher_char.upper()
            else:
                ciphertext += cipher_char
        else:
            ciphertext += char
    return ciphertext

def decrypt(ciphertext, cipher_key):
    reverse_key = {v: k for k, v in cipher_key.items()}
    plaintext = ''
    for char in ciphertext:
        if char.isalpha():
            char_lower = char.lower()
            plain_char = reverse_key[char_lower]
            if char.isupper():
                plaintext += plain_char.upper()
            else:
                plaintext += plain_char
        else:
            plaintext += char
    return plaintext

# Example usage
cipher_key = create_cipher_key()

# Take input from the user
plaintext = input("Enter the plaintext to encrypt: ")

ciphertext = encrypt(plaintext, cipher_key)
decrypted_text = decrypt(ciphertext, cipher_key)
```

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

Input and Output:

```
Enter the plaintext to encrypt: Hello, My name is Sayeeda Khan.
Plaintext: Hello, My name is Sayeeda Khan.

Cipher Key: {'a': 'm', 'b': 'u', 'c': 'l', 'd': 't', 'e': 'z', 'f': 'd', 'g':
'c', 'h': 'b', 'i': 'h', 'j': 'a', 'k': 'g', 'l': 's', 'm': 'y', 'n': 'x', 'o':
'e', 'p': 'p', 'q': 'n', 'r': 'w', 's': 'j', 't': 'f', 'u': 'v', 'v': 'r', 'w':
'k', 'x': 'i', 'y': 'q', 'z': 'o'}

Ciphertext: Bzsse, Yq xmyz hj Jmqzztm Gbmx.
Decrypted Text: Hello, My name is Sayeeda Khan.
```

Experiment Number: 03

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

Objective: The goals of conducting a brute-force attack on the Caesar Cipher are:

1. **Recover the Original Message:** To decrypt the ciphertext by testing every possible shift used in the encryption process.
2. **Determine the Correct Shift:** To find the right shift value by trying all 26 possible shifts and checking which one produces meaningful text.
3. **Expose the Vulnerability of Caesar Cipher:** To demonstrate the weakness of the Caesar Cipher, as its small key space allows for easy decryption through brute-force methods.
4. **Show the Efficiency of Brute-Force:** To highlight how a brute-force approach is effective for breaking simple encryption systems, like the Caesar Cipher, due to the limited number of possible keys.

Theory: A brute-force attack on the Caesar Cipher consists of testing every possible shift value to decrypt a message. Since the cipher uses a fixed shift to replace each letter of the alphabet, the attack works by trying all possible shifts until the correct one is identified.

1. **Caesar Cipher Description:**
    - The Caesar Cipher is a substitution cipher where each letter of the plaintext is replaced by another letter that is shifted by a fixed number of positions in the alphabet.
    - For example, a shift of 3 means "A" becomes "D", "B" becomes "E", and so on, with the alphabet wrapping around after "Z".
2. **Brute-Force Method:**
    - A brute-force attack involves trying all shift values from 1 to 25 and checking each result.
    - Since the cipher operates on a 26-letter alphabet, there are only 26 possible shifts, making the brute-force attack simple.
3. **How It Works:**
    - The attacker does not need prior knowledge of the shift key. They decrypt the ciphertext using each of the 26 potential shifts.
    - After each decryption attempt, the resulting text is analyzed to check if it forms readable words or makes sense.
    - When the correct shift is found, the plaintext is uncovered.
4. **Why Brute-Force Works:**

- The limited key space of just 26 possible shifts makes brute-forcing effective. The attack is quick and straightforward since there are no complex steps involved.
- Each shift can easily be checked for meaning, so the correct one is quickly identified.

5. **Example:**
   - Given the ciphertext "khoor", the brute-force attack tests all shifts from 1 to 25. When shift 3 is applied, the plaintext "hello" is revealed, which is readable and indicates the correct shift.

Conclusion:

The simplicity and small key space of the Caesar Cipher make it highly vulnerable to brute-force attacks. Since there are only 26 possible shifts, a brute-force attack is fast and guarantees success in uncovering the plaintext. However, this method is ineffective against more complex encryption systems with larger key spaces.

Pseudo Code:

```
def encrypt(text, shift):
    result = ""
    for char in text:
        if char.isalpha():
            # Shift letter within alphabet
            offset = 65 if char.isupper() else 97
            result += chr((ord(char) - offset + shift) % 26 + offset)
        else:
            result += char  # Non-alphabetic characters remain the same
    return result

# Decrypt with Caesar Cipher
def decrypt(text, shift):
    return encrypt(text, -shift)

# Brute-force decrypt Caesar Cipher by trying all shifts
def brute_force_decrypt(cipher_text):
    print("Brute-Force Decryption Results:")
    for possible_shift in range(1, 26):  # 25 possible keys for Caesar Cipher
        decrypted_text = decrypt(cipher_text, possible_shift)
        print(f"Shift {possible_shift}: {decrypted_text}")

# Get user input for plaintext and shift
text = input("Enter the plain text: ")
shift = int(input("Enter the shift key (1-25): "))

# Encrypt the text
encrypted_text = encrypt(text, shift)
print("Encrypted Text:", encrypted_text)

# Decrypt the text back using known shift
decrypted_text = decrypt(encrypted_text, shift)
print("Decrypted Text:", decrypted_text)

# Perform brute-force decryption on encrypted text
```

```
brute_force_decrypt(encrypted_text)
```

Input and Output:

```
Enter the plain text: I'm Sayeeda Khan a student of ICE.
Enter the shift key (1-25): 4
Encrypted Text: M'q Weciihe Oler e wxyhirx sj MGI.
Decrypted Text: I'm Sayeeda Khan a student of ICE.
Brute-Force Decryption Results:
Shift 1: L'p Vdbhhgd Nkdq d vwxghqw ri LFH.
Shift 2: K'o Ucaggfc Mjcp c uvwfgpv qh KEG.
Shift 3: J'n Tbzffeb Libo b tuvefou pg JDF.
Shift 4: I'm Sayeeda Khan a student of ICE.
Shift 5: H'l Rzxddcz Jgzm z rstcdms ne HBD.
Shift 6: G'k Qywccby Ifyl y qrsbclr md GAC.
Shift 7: F'j Pxvbbax Hexk x pqrabkq lc FZB.
Shift 8: E'i Owuaazw Gdwj w opqzajp kb EYA.
Shift 9: D'h Nvtzzyv Fcvi v nopyzio ja DXZ.
Shift 10: C'g Musyyxu Ebuh u mnoxyhn iz CWY.
Shift 11: B'f Ltrxxwt Datg t lmnwxgm hy BVX.
Shift 12: A'e Ksqwwvs Czsf s klmvwfl gx AUW.
Shift 13: Z'd Jrpvvur Byre r jkluvek fw ZTV.
Shift 14: Y'c Iqouutq Axqd q ijktudj ev YSU.
Shift 15: X'b Hpnttsp Zwpc p hijstci du XRT.
Shift 16: W'a Gomssro Yvob o ghirsbh ct WQS.
Shift 17: V'z Fnlrrqn Xuna n fghqrag bs VPR.
Shift 18: U'y Emkqqpm Wtmz m efgpqzf ar UOQ.
Shift 19: T'x Dljppol Vsly l defopye zq TNP.
Shift 20: S'w Ckioonk Urkx k cdenoxd yp SMO.
Shift 21: R'v Bjhnnmj Tqjw j bcdmnwc xo RLN.
Shift 22: Q'u Aigmmli Spiv i abclmvb wn QKM.
Shift 23: P't Zhfllkh Rohu h zabklua vm PJL.
Shift 24: O's Ygekkjg Qngt g yzajktz ul OIK.
Shift 25: N'r Xfdjjif Pmfs f xyzijsy tk NHJ.
```

Experiment Number: 04

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

Objective: The objective of the Hill cipher is to provide a more secure method of encryption by encrypting multiple letters at once using matrix multiplication. The key goals include:

1. **Enhancing Security:** By using blocks of letters and linear algebra, the Hill cipher increases the complexity of encryption, making it more difficult to break than traditional substitution ciphers.
2. **Using Mathematical Principles:** To leverage matrix operations for both encryption and decryption, adding a layer of sophistication that improves security.
3. **Polygraphic Substitution:** To replace individual letter substitution with a polygraphic approach, which encrypts groups of letters simultaneously, strengthening the cipher's resistance to frequency analysis.

Theory: The Hill cipher is a polygraphic substitution cipher that uses linear algebra, specifically matrix multiplication, for encryption and decryption. Unlike traditional ciphers that encrypt one letter at a time, the Hill cipher works on blocks of letters. The plaintext is divided into blocks, and each block is represented as a vector. A key matrix is then multiplied by the plaintext vector to produce the ciphertext.

This approach provides higher security compared to simple substitution ciphers, as it encrypts multiple letters at once. The decryption process uses the inverse of the key matrix to transform the ciphertext back into the plaintext.

Encryption Process

1. Matrix Formation:
    - Represent the plaintext as a vector. For example, if using a block size of 2, the plaintext "HI" can be represented as a vector:

$$P = \frac{7}{8}$$

    - The corresponding numerical values for letters are based on their position in the alphabet (A=0, B=1, ..., Z=25).
2. Key Matrix:
    - Choose a square key matrix K. For instance:

$$K = \begin{pmatrix} 6 & 24 \\ 1 & 13 \end{pmatrix}$$

    - The key matrix must have an inverse modulo 26 for decryption.
3. Ciphertext Generation:
    - Multiply the key matrix by the plaintext vector, followed by taking the result modulo 26:

$$C = K \cdot P \ mod\ 26$$

- For the above example, if K is used with P:

$$C = \begin{pmatrix} 6 & 24 \\ 1 & 13 \end{pmatrix} \cdot \begin{matrix} 7 \\ 8 \end{matrix} = \begin{matrix} 6 \cdot 7 + 24 \cdot 8 \\ 1 \cdot 7 + 13 \cdot 8 \end{matrix} = \begin{matrix} 7 \\ 11 \end{matrix} \ mod\ 26$$

- The resulting ciphertext characters correspond to numerical values, such as 7and 11, representing "HL".

Decryption Process

1. Compute Inverse of the Key Matrix:
   - The inverse of the key matrix K is necessary for decryption. The inverse K^{-1} must satisfy:

$$K \cdot K^{-1} = I \ mod\ 26$$

   - This can be found using methods such as the adjugated matrix or Gaussian elimination, followed by reduction modulo 26.
2. Ciphertext Vector:
   - Represent the ciphertext similarly as a vector:

$$C = \begin{matrix} 7 \\ 11 \end{matrix}$$

3. Recover Plaintext:
   - Multiply the inverse key matrix by the ciphertext vector, applying modulo 26:

$$P' = K^{-1} \cdot C \ mod\ 26$$

   - This operation retrieves the original plaintext vector from the ciphertext.

Example

Using the matrices from earlier:

- Key Matrix:

$$K = \begin{pmatrix} 6 & 24 \\ 1 & 13 \end{pmatrix}$$

- Inverse Matrix (calculated previously):

$$K^{-1} = \begin{pmatrix} 15 & 24 \\ 2 & 17 \end{pmatrix}$$

Encrypting "HI":

1. Plaintext Vector:

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

2. Compute Ciphertext:

$$C = K \cdot P \ mod\ 26 = \begin{matrix} 7 \\ 11 \end{matrix}$$

Decrypting "HL":

1. Ciphertext Vector:

$$C = \begin{matrix} 7 \\ 11 \end{matrix}$$

2. Compute Plaintext:

$$P' = K^{-1} \cdot C \ mod\ 26$$

Conclusion

The Hill cipher effectively combines linear algebra with modular arithmetic to secure messages. Its reliance on matrix operations allows for the encryption of multiple characters simultaneously, enhancing both security and complexity. For successful implementation, it is crucial to ensure that the key matrix is invertible within the modulo 26 system.

Pseudo Code

```
import numpy as np

def create_key_matrix(key):
    """Generate a 3x3 key matrix from the given key string."""
    key_matrix = []
    k = 0
    for i in range(3):
        row = [ord(key[k + j]) % 65 for j in range(3)]
        key_matrix.append(row)
        k += 3
    return np.array(key_matrix)

def text_to_vector(text):
    """Convert text to vector form."""
    return np.array([[ord(char) % 65] for char in text])

def vector_to_text(vector):
    """Convert vector back to text."""
    return ''.join(chr(int(num) + 65) for num in vector)

def encrypt_message(plaintext, key):
    """Encrypt the plaintext message using the Hill cipher."""
    key_matrix = create_key_matrix(key)
    message_vector = text_to_vector(plaintext)
    cipher_vector = np.dot(key_matrix, message_vector) % 26
    return vector_to_text(cipher_vector)

def mod_inverse(a, m):
```

```
    """Compute modular inverse of a under modulo m."""
    for x in range(1, m):
        if (a * x) % m == 1:
            return x
    return -1

def decrypt_message(ciphertext, key):
    """Decrypt the ciphertext message using the Hill cipher."""
    key_matrix = create_key_matrix(key)
    determinant = int(np.round(np.linalg.det(key_matrix))) % 26
    determinant_inv = mod_inverse(determinant, 26)

    if determinant_inv == -1:
        raise ValueError("The key matrix is not invertible. Choose a different
key.")

    adjugate_matrix = np.round(determinant *
np.linalg.inv(key_matrix)).astype(int) % 26
    key_matrix_inv = (determinant_inv * adjugate_matrix) % 26
    cipher_vector = text_to_vector(ciphertext)
    decrypted_vector = np.dot(key_matrix_inv, cipher_vector) % 26
    return vector_to_text(decrypted_vector)

# Example usage
plaintext = "ACT"
key = "GYBNQKURP"

print("Plaintext:", plaintext)
ciphertext = encrypt_message(plaintext, key)
print("Ciphertext:", ciphertext)

try:
    decrypted_text = decrypt_message(ciphertext, key)
    print("Decrypted Text:", decrypted_text)
except ValueError as e:
    print(e)
```

Input and Output:

```
Plaintext: ACT
Ciphertext: POH
Decrypted Text: LRK
```

Experiment Number: 05

Name of the Experiment: Write a program to implement encryption using Playfair Cipher.

Objective: The objective of the Playfair cipher is to enhance security by encrypting pairs of letters (digraphs) rather than single characters, making it harder to break than basic ciphers. It improves encryption complexity, reduces susceptibility to frequency analysis, and provides a relatively simple yet more secure manual encryption method. It aims to balance security and efficiency for practical use.

Theory: The Playfair cipher is a straightforward encryption method that encodes pairs of letters, also called digraphs, instead of single letters, offering a bit more security than simpler substitution ciphers. Although it was developed by Charles Wheatstone, it is named after Lord Playfair who popularized it.

Steps for Encryption:

1. Create a 5x5 Grid:
    - Select a keyword or phrase, such as "KHAN", removing any duplicate letters.
    - Place the letters of the keyword in the 5x5 grid, followed by the remaining unused letters of the alphabet. Usually, "I" and "J" are treated as the same letter to fit all 26 letters in the grid.

      Example using "KHAN", the grid will look like this:

      K H A N B

      C D E F G

      I/J L M O P

      Q R S T U

      V W X Y Z
2. Split the Message into Pairs:
    - Divide the plaintext into pairs of letters. If you have repeated letters, insert a filler (like "X") between them. If the message has an odd number of letters, add an extra filler letter at the end.
    - For the message "SAYEEDA", we split it into pairs as: "SA", "YE", "ED", "DX".
3. Encrypt Each Pair:
    - Depending on their position in the grid, apply the following rules to each pair:
        - Same Row: If the two letters are in the same row, replace them with the letters directly to the right in the same row. If a letter is at the end, it wraps around to the beginning of the row.

- Same Column: If the two letters are in the same column, replace each with the letter directly below it. If a letter is at the bottom, it wraps to the top of the column.
- Rectangle (Different Row and Column): If the letters form a rectangle, replace each letter with the letter in its row but in the column of the other letter in the pair.

Here's a simple description of the Playfair cipher encryption for the plaintext **"SAYEEDA"** using the key **"KHAN"**:

1. **Create the Playfair Grid:**
   - First, remove duplicates from the key **"KHAN"**, resulting in **"KHAN"**.
   - Fill the rest of the alphabet (excluding "J", which is combined with "I") into the grid.
2. **Split the Plaintext into Pairs:**
   - The plaintext **"SAYEEDA"** is split into pairs: **"SA"**, **"YE"**, **"EX"**, **"DX"**.
   - Since **"E"** appears twice, we insert an "X" between them to form **"EX"**.
3. **Apply the Playfair Cipher Rules:**
   - **For "SA"**:
     - "S" (row 4, column 3) and "A" (row 1, column 3) are in the same column, so shift each letter down one row:
       - "S" becomes "X"
       - "A" becomes "E"
   - **For "YE"**:
     - "Y" (row 5, column 4) and "E" (row 2, column 3) form a rectangle, so swap the columns:
       - "Y" becomes "F"
       - "E" becomes "F"
   - **For "EX"**:
     - "E" (row 2, column 3) and "X" (row 5, column 3) are in the same column, so shift each letter down one row:
       - "E" becomes "F"
       - "X" becomes "E"
   - **For "DX"**:

- "D" (row 2, column 2) and "X" (row 5, column 3) form a rectangle, so swap the columns:
    - "D" becomes "E"
    - "X" becomes "A"

4. **Final Ciphertext:**
    - The final ciphertext after applying the rules to all the pairs is **"XEXFFEEA"**.

This is how the message **"SAYEEDA"** gets encrypted to **"XEXFFEEA"** using the Playfair cipher with the key **"KHAN"**.

Pseudo Code

```
def generate_key_matrix(key):
    key = ''.join(sorted(set(key), key=lambda x: key.index(x))).replace("J",
"I")
    alphabet = "ABCDEFGHIKLMNOPQRSTUVWXYZ"
    key_matrix = []
    for char in key:
        if char not in key_matrix:
            key_matrix.append(char)
    for char in alphabet:
        if char not in key_matrix:
            key_matrix.append(char)
    # Create a 5x5 matrix
    return [key_matrix[i:i + 5] for i in range(0, 25, 5)]

def format_plaintext(plaintext):
    plaintext = plaintext.upper().replace(" ", "").replace("J", "I")
    formatted_text = ""
    i = 0
    while i < len(plaintext):
        formatted_text += plaintext[i]
        if i + 1 < len(plaintext) and plaintext[i] == plaintext[i + 1]:
            formatted_text += 'X'
        elif i + 1 < len(plaintext):
            formatted_text += plaintext[i + 1]
            i += 1
        i += 1
    # If there's an odd number of characters, add an 'X' to the end
    if len(formatted_text) % 2 != 0:
        formatted_text += 'X'

    return formatted_text

def find_position(char, key_matrix):
    for i, row in enumerate(key_matrix):
        if char in row:
            return i, row.index(char)
    return None

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

```
    if row1 == row2:  # Same row: move right
        return key_matrix[row1][(col1 + 1) % 5] + key_matrix[row2][(col2 + 1) %
5]
    elif col1 == col2:  # Same column: move down
        return key_matrix[(row1 + 1) % 5][col1] + key_matrix[(row2 + 1) %
5][col2]
    else:  # Rectangle swap
        return key_matrix[row1][col2] + key_matrix[row2][col1]


def encrypt_playfair(plaintext, key):
    key_matrix = generate_key_matrix(key)
    plaintext = format_plaintext(plaintext)
    ciphertext = ""
    for i in range(0, len(plaintext), 2):
        pair = plaintext[i:i + 2]
        ciphertext += encrypt_pair(pair, key_matrix)
    return ciphertext

plaintext = input("Enter Your Plain Text: ")
key = input("Enter Your key in Capital Letter: ")
ciphertext = encrypt_playfair(plaintext, key)
print("Plaintext:", plaintext)
print("Ciphertext:", ciphertext)
```

Input and Output:

```
Enter Your Plain Text: SAYEEDA
Enter Your key in Capital Letter: KHAN
Plaintext: SAYEEDA
Ciphertext: XEXFFEEA
```

Experiment Number: 06

Name of the Experiment: Write a program to implement decryption using Playfair Cipher.

Theory: The Playfair cipher's decryption process essentially reverses the encryption steps. Using the same 5x5 grid, each pair of letters in the encrypted message is decoded back to the original message by following these rules:

Playfair Cipher Decryption Example: "XEXFFEEA"

Given:

- **Ciphertext**: "XEXFFEEA"
- **Key**: "KHAN"
- **Playfair Grid**:

K H A N B

C D E F G

I/J L M O P

Q R S T U

V W X Y Z

1. Separate the Encrypted Message into Pairs:

- The ciphertext **"XEXFFEEA"** is split into pairs: **"XE"**, **"XF"**, **"FE"**, **"EA"**.

2. Apply Decryption Rules Based on Position:

- **Decrypt "XE"**:
    - "X" is located in **row 5, column 3**, and "E" is in **row 2, column 3**.
    - Since "X" and "E" are in the **same column**, we follow the rule for the same column: each letter is replaced by the letter directly above it.
        - "X" becomes "E" (row 5, column 3 → row 4, column 3).
        - "E" becomes "S" (row 2, column 3 → row 1, column 3).
    - "XE" decrypts to **"ES"**.
- **Decrypt "XF"**:
    - "X" is in **row 5, column 3**, and "F" is in **row 2, column 4**.
    - They form a **rectangle** (different row and column). The letters swap columns.
        - "X" becomes "Y" (row 5, column 4 → row 5, column 1).
        - "F" becomes "E" (row 2, column 3 → row 2, column 1).

- "XF" decrypts to **"YE"**.

- **Decrypt "FE"**:
  - "F" is in **row 2, column 4**, and "E" is in **row 2, column 3**.
  - Since they are in the **same row**, we shift each letter to the left.
    - "F" becomes "E" (row 2, column 4 → row 2, column 3).
    - "E" becomes "D" (row 2, column 3 → row 2, column 2).
  - "FE" decrypts to **"ED"**.

- **Decrypt "EA"**:
  - "E" is in **row 2, column 3**, and "A" is in **row 1, column 3**.
  - Since they are in the **same column**, we shift each letter up one row.
    - "E" becomes "A" (row 2, column 3 → row 1, column 3).
    - "A" becomes "Y" (row 1, column 3 → row 1, column 4).
  - "EA" decrypts to **"AY"**.

3. Combine the Decrypted Letters:

- After decoding each pair, we combine the letters to form the final decrypted message:
  - "ES", "YE", "ED", "AY"
  - Final decrypted message: **"SAYEEDAY"**

However, there is an extra "Y" at the end, which we added during encryption as filler, so we remove that to get the final **"SAYEDA"**.

Final Decrypted Message: "SAYEEDA"

By following the Playfair cipher decryption process, the encrypted message **"XEXFFEEA"** is successfully decrypted back to the original message **"SAYEEDA"**.

Pseudo Code

```
def generate_key_matrix(key):
    key = ''.join(sorted(set(key), key=lambda x: key.index(x))).replace("J", "I")
    alphabet = "ABCDEFGHIKLMNOPQRSTUVWXYZ"  # 'J' is combined with 'I'
    key_matrix = []

    for char in key:
        if char not in key_matrix:
            key_matrix.append(char)

    for char in alphabet:
        if char not in key_matrix:
            key_matrix.append(char)

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

```
def format_plaintext(plaintext):
    plaintext = plaintext.upper().replace(" ", "").replace("J", "I")
    formatted_text = ""
    i = 0

    while i < len(plaintext):
        formatted_text += plaintext[i]
        if i + 1 < len(plaintext) and plaintext[i] == plaintext[i + 1]:
            formatted_text += 'X'
        elif i + 1 < len(plaintext):
            formatted_text += plaintext[i + 1]
            i += 1
        i += 1

    if len(formatted_text) % 2 != 0:
        formatted_text += 'X'

    return formatted_text

def find_position(char, key_matrix):
    for i, row in enumerate(key_matrix):
        if char in row:
            return i, row.index(char)
    return None

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

    if row1 == row2:  # Same row: move right
        return key_matrix[row1][(col1 + 1) % 5] + key_matrix[row2][(col2 + 1) %
5]
    elif col1 == col2:  # Same column: move down
       return key_matrix[(row1 + 1) % 5][col1] + key_matrix[(row2 + 1) % 5][col2]
    else:  # Rectangle swap
        return key_matrix[row1][col2] + key_matrix[row2][col1]

def encrypt_playfair(plaintext, key):
    key_matrix = generate_key_matrix(key)
    plaintext = format_plaintext(plaintext)
    ciphertext = ""

    for i in range(0, len(plaintext), 2):
        pair = plaintext[i:i + 2]
        ciphertext += encrypt_pair(pair, key_matrix)
    return ciphertext

# Example usage
plaintext = input("Enter The Plain Text: ")
key_encrypt = input("Enter The Key in Capital Letter: ")
ciphertext = encrypt_playfair(plaintext, key_encrypt)
print("Plaintext:", plaintext)
print("The Key  is:", key_encrypt)
print("Ciphertext:", ciphertext)

def decrypt_pair(pair, key_matrix):
    row1, col1 = find_position(pair[0], key_matrix)
    row2, col2 = find_position(pair[1], key_matrix)
```

```
    if row1 == row2:  # Same row: move left
        return key_matrix[row1][(col1 - 1) % 5] + key_matrix[row2][(col2 - 1) %
5]
    elif col1 == col2:  # Same column: move up
       return key_matrix[(row1 - 1) % 5][col1] + key_matrix[(row2 - 1) % 5][col2]
    else:  # Rectangle swap
        return key_matrix[row1][col2] + key_matrix[row2][col1]

def decrypt_playfair(ciphertext, key):
    key_matrix = generate_key_matrix(key)
    plaintext = ""
    for i in range(0, len(ciphertext), 2):
        pair = ciphertext[i:i + 2]
        plaintext += decrypt_pair(pair, key_matrix)
    return plaintext

ciphertext = input("Enter the cipher text: ")
key_decrypt = input("Enter The Key in Capital Letter: ")
decrypted_text = decrypt_playfair(ciphertext, key_decrypt)
print("Decrypted Text:", decrypted_text)
```

Input and Output

```
Enter The Plain Text: SAYEEDA
Enter The Key in Capital Letter: KHAN
Plaintext: SAYEEDA
The Key  is: KHAN
Ciphertext: XEXFFEEA
Enter the cipher text: XEXFFEEA
Enter The Key in Capital Letter: KHAN
Decrypted Text: SAYEEDAX
```

Experiment Number: 07

Name of the Experiment: Write a program to implement encryption using Polyalphabetic Cipher.

Theory: The Vigenère cipher is a polyalphabetic substitution cipher, which means each letter in the message is shifted differently based on the keyword. This method enhances the security of the message by using multiple shifting patterns instead of a single, predictable shift.

Encryption Process

Choosing the Keyword:

- We begin by selecting a keyword and repeating it to match the length of our message. For the plaintext "SAYEEDA" and the keyword "KHAN", we repeat "KHAN" to form "KHANKHA" so that each letter in the message aligns with a letter in the keyword.

Aligning the Message and Keyword:

Plaintext: S A Y E E D A

Keyword: K H A N K H A

Encrypting Each Letter: For each letter in the message, we determine its position in the alphabet (A = 0, B = 1, ..., Z = 25). Then, we shift the plaintext letter by the corresponding letter in the keyword. When a shift exceeds "Z", we wrap around to "A."

Here's how the encryption works for each letter:

- **S** (position 18) shifted by **K** (position 10) gives:
    - 18 + 10 = 28, which wraps around to **2** (C).
    - "S" becomes **C**.
- **A** (position 0) shifted by **H** (position 7) gives:
    - 0 + 7 = 7.
    - "A" becomes **H**.
- **Y** (position 24) shifted by **A** (position 0) gives:
    - 24 + 0 = 24.
    - "Y" remains **Y**.
- **E** (position 4) shifted by **N** (position 13) gives:
    - 4 + 13 = 17.
    - "E" becomes **R**.
- **E** (position 4) shifted by **K** (position 10) gives:

    - 4 + 10 = 14.
    - "E" becomes **O**.
- **D** (position 3) shifted by **H** (position 7) gives:
    - 3 + 7 = 10.
    - "D" becomes **K**.
- **A** (position 0) shifted by **A** (position 0) gives:
    - 0 + 0 = 0.
    - "A" remains **A**.

Final Encrypted Message: After performing the shifts for each letter, we combine the encrypted letters:

- "S" → **C**
- "A" → **H**
- "Y" → **Y**
- "E" → **R**
- "E" → **O**
- "D" → **K**
- "A" → **A**

Thus, the encrypted message is **"CHYROKA"**.

Final Encrypted Message: "CHYROKA"

By following the Vigenère cipher encryption process, the plaintext **"SAYEEDA"** is successfully encrypted to **"CHYROKA"** using the keyword **"KHAN"**.

Pseudo Code

```
def generate_key(plaintext, key):
    key = list(key)
    if len(key) < len(plaintext):
        key = key * (len(plaintext) // len(key)) + key[:len(plaintext) %
len(key)]
    return ''.join(key)
def encrypt_vigenere(plaintext, key):
    ciphertext = []
    key = generate_key(plaintext, key).upper()
    for i in range(len(plaintext)):
        if plaintext[i].isalpha():
            shift = ord(key[i]) - ord('A')
            if plaintext[i].isupper():
                enc_char = chr(((ord(plaintext[i]) - ord('A') + shift) % 26) +
ord('A'))
            else:
```

```
                enc_char = chr(((ord(plaintext[i]) - ord('a') + shift) % 26) +
ord('a'))
            ciphertext.append(enc_char)
        else:
            ciphertext.append(plaintext[i])

    return ''.join(ciphertext)

plaintext = input("Enter The Plain Text: ")
key_encrypt = input("Enter The Key in Capital Letter: ")
ciphertext = encrypt_vigenere(plaintext, key_encrypt)
print("Plaintext:", plaintext)
print("Key:", key_encrypt)
print("Ciphertext:", ciphertext)
```

Input and Output:

```
Enter The Plain Text: SAYEEDA
Enter The Key in Capital Letter: KHAN
Plaintext: SAYEEDA
Key: KHAN
Ciphertext: CHYROKA
```

Experiment Number: 08

Name of the Experiment: Write a program to implement decryption using Polyalphabetic Cipher.

Theory: The Vigenère cipher's decryption process essentially reverses the encryption steps. Using the same keyword and the encrypted message, each letter in the ciphertext is decoded back to the original message by following the decryption rules.

Decryption Process

Choosing the Keyword:

- The keyword remains the same as during encryption. In this case, the keyword is "KHAN". We repeat it to match the length of the encrypted message "CHYROKA" to form "KHANKHA".

Aligning the Encrypted Message and Keyword:

Ciphertext: C H Y R O K A

Keyword: K H A N K H A

Decrypting Each Letter:

For each letter in the encrypted message, we determine its position in the alphabet (A = 0, B = 1, ..., Z = 25). Then, we shift the ciphertext letter backward by the corresponding letter in the keyword. When a shift goes below "A", we wrap around to "Z".

Here's how the decryption works for each letter:

- **C** (position 2) shifted backward by **K** (position 10) gives:
    - 2 - 10 = -8, which wraps around to 18 (S).
    - "C" becomes **S**.
- **H** (position 7) shifted backward by **H** (position 7) gives:
    - 7 - 7 = 0.
    - "H" becomes **A**.
- **Y** (position 24) shifted backward by **A** (position 0) gives:
    - 24 - 0 = 24.
    - "Y" remains **Y**.
- **R** (position 17) shifted backward by **N** (position 13) gives:
    - 17 - 13 = 4.
    - "R" becomes **E**.
- **O** (position 14) shifted backward by **K** (position 10) gives:

    - 14 - 10 = 4.
    - "O" becomes **E**.
- **K** (position 10) shifted backward by **H** (position 7) gives:
    - 10 - 7 = 3.
    - "K" becomes **D**.
- **A** (position 0) shifted backward by **A** (position 0) gives:
    - 0 - 0 = 0.
    - "A" remains **A**.

Final Decrypted Message: After performing the shifts for each letter, we combine the decrypted letters:

- "C" → **S**
- "H" → **A**
- "Y" → **Y**
- "R" → **E**
- "O" → **E**
- "K" → **D**
- "A" → **A**

Thus, the decrypted message is **"SAYEEDA"**.

Final Decrypted Message: "SAYEEDA"

By following the Vigenère cipher decryption process, the encrypted message **"CHYROKA"** is successfully decrypted back to the original plaintext **"SAYEEDA"** using the keyword **"KHAN"**.

Pseudo Code

```
def generate_key(plaintext, key):
    key = list(key)
    if len(key) < len(plaintext):
        key = key * (len(plaintext) // len(key)) + key[:len(plaintext) %
len(key)]
    return ''.join(key)
def decrypt_vigenere(ciphertext, key):
    plaintext = []
    key = generate_key(ciphertext, key).upper()
    for i in range(len(ciphertext)):
        if ciphertext[i].isalpha():
            shift = ord(key[i]) - ord('A')
            if ciphertext[i].isupper():
                dec_char = chr(((ord(ciphertext[i]) - ord('A') - shift + 26) %
26) + ord('A'))
            else:
```

```
                dec_char = chr(((ord(ciphertext[i]) - ord('a') - shift + 26) %
26) + ord('a'))
            plaintext.append(dec_char)
        else:
            plaintext.append(ciphertext[i])
    return ''.join(plaintext)

ciphertext = input("Enter The Cipher Text: ")
key = input("Enter The Key in capital letter to decrypt: ")
decrypted_text = decrypt_vigenere(ciphertext, key)
print("Ciphertext:", ciphertext)
print("Key:", key)
print("Decrypted text:", decrypted_text)
```

Input and Output:

```
Enter The Cipher Text: CHYROKA
Enter The Key in capital letter to decrypt: KHAN
Ciphertext: CHYROKA
Key: KHAN
Decrypted text: SAYEEDA
```

Experiment Number: 09

Name of the Experiment: Write a program to implement encryption using Vernam Cipher.

Theory: The Vernam cipher, also known as the one-time pad, is an encryption method that provides perfect security when the key is random, as long as the message, used only once, and kept secret. Each character in the plaintext is encrypted by performing an XOR (exclusive OR) operation with the corresponding character from the key.

Step-by-Step Process for the Vernam Cipher:

1. Key Generation:

We have chosen the key "KHANKHA," which is repeated to match the length of the plaintext. The plaintext is 7 characters long, and the key is also 7 characters long, so no repetition is necessary in this case.

Plaintext: S A Y E E D A

Key: K H A N K H A

2. Convert Plaintext to Binary: Each character in the plaintext is converted to its binary equivalent using ASCII codes.

- S = 83 (ASCII) = 01010011 (binary)
- A = 65 (ASCII) = 01000001 (binary)
- Y = 89 (ASCII) = 01011001 (binary)
- E = 69 (ASCII) = 01000101 (binary)
- E = 69 (ASCII) = 01000101 (binary)
- D = 68 (ASCII) = 01000100 (binary)
- A = 65 (ASCII) = 01000001 (binary)

Plaintext in binary:

S = 01010011

A = 01000001

Y = 01011001

E = 01000101

E = 01000101

D = 01000100

A = 01000001

3. Convert Key to Binary: Similarly, convert the key "KHANKHA" into binary.

- K = 75 (ASCII) = 01001011 (binary)
- H = 72 (ASCII) = 01001000 (binary)
- A = 65 (ASCII) = 01000001 (binary)
- N = 78 (ASCII) = 01001110 (binary)
- K = 75 (ASCII) = 01001011 (binary)
- H = 72 (ASCII) = 01001000 (binary)
- A = 65 (ASCII) = 01000001 (binary)

Key in binary:

K = 01001011

H = 01001000

A = 01000001

N = 01001110

K = 01001011

H = 01001000

A = 01000001

4. Apply XOR Operation:

Now, perform the XOR operation between the plaintext and the key. Each bit of the plaintext is XORed with the corresponding bit of the key.

For example:

- "S" (01010011) XOR "K" (01001011) = 00011000 (in binary, which is 24 in decimal, corresponding to "18" in hexadecimal)

Continuing the XOR operation for each pair:

S (01010011) XOR K (01001011) = 00011000 → 18 (hex)

A (01000001) XOR H (01001000) = 00001001 → 09 (hex)

Y (01011001) XOR A (01000001) = 00011000 → 18 (hex)

E (01000101) XOR N (01001110) = 00001011 → 0B (hex)

E (01000101) XOR K (01001011) = 00001110 → 0E (hex)

D (01000100) XOR H (01001000) = 00001100 → 0C (hex)

A (01000001) XOR A (01000001) = 00000000 → 00 (hex)

5. Construct the Ciphertext:

After performing the XOR operation on each pair of plaintext and key, the resulting hexadecimal ciphertext is:

Ciphertext: 1809180b0e0c00

Final Result:

The plaintext **"SAYEEDA"** encrypted with the key **"KHANKHA"** using the Vernam cipher produces the ciphertext:

Ciphertext: 1809180b0e0c00

This is the encrypted message, represented in hexadecimal format. Each pair of hex digits corresponds to the result of the XOR operation between the plaintext and the key.

Pseudo Code

```
def vernam_encrypt(plaintext, key):
    ciphertext = []
    # Ensure key length matches the length of the plaintext
    if len(plaintext) != len(key):
        raise ValueError("Key length must be equal to the plaintext length.")
    for i in range(len(plaintext)):
        # XOR between the binary representation of plaintext and key characters
        encrypted_char = chr(ord(plaintext[i]) ^ ord(key[i]))
        ciphertext.append(encrypted_char)

    return ''.join(ciphertext)
plaintext = input("Enter The Plain Text: ")
key_encrypt = input("Enter The Key in Capital Letter: ")
ciphertext = vernam_encrypt(plaintext, key_encrypt)
print("Plaintext:", plaintext)
print("Key:",key_encrypt)
print("Ciphertext:", ''.join(format(ord(c), '02x') for c in ciphertext))
```

Input and Output:

```
Enter The Plain Text: SAYEEDA
Enter The Key in Capital Letter: KHANKHA
Plaintext: SAYEEDA
Key: KHANKHA
Ciphertext: 1809180b0e0c00
```

Experiment Number: 10

Name of the Experiment: Write a program to implement decryption using Vernam Cipher.

Theory: The decryption process of the Vernam cipher (one-time pad) essentially reverses the encryption process. Since the same key is used for both encryption and decryption, the XOR operation applied during encryption is also applied during decryption to retrieve the original plaintext.

Decryption Steps:

1. **Obtain the Ciphertext**: The ciphertext is the result of the XOR operation between the plaintext and the key. In this case, the ciphertext is: 1809180b0e0c00
2. **Convert the Ciphertext to Binary**:
   First, we convert the ciphertext back to binary. Since the ciphertext is in hexadecimal format, we need to convert each pair of hexadecimal digits into their 8-bit binary equivalent.

The hexadecimal ciphertext 1809180b0e0c00 converts to binary as follows:

Ciphertext in hex: 18 09 18 0b 0e 0c 00

Binary form:

18 (hex) = 00011000 (binary)

09 (hex) = 00001001 (binary)

18 (hex) = 00011000 (binary)

0b (hex) = 00001011 (binary)

0e (hex) = 00001110 (binary)

0c (hex) = 00001100 (binary)

00 (hex) = 00000000 (binary)

3. **Apply XOR Operation**:
   The decryption process uses the same key, "KHANKHA", and the XOR operation is applied between each byte of the ciphertext and the corresponding byte of the key. Since the key and the ciphertext are the same length, each pair of binary digits from the ciphertext is XORed with the corresponding binary digits from the key to retrieve the original plaintext.

For example:

- Ciphertext "18" (00011000) XOR Key "K" (01001011) = "S" (01010011)
- Ciphertext "09" (00001001) XOR Key "H" (01001000) = "A" (01000001)
- Ciphertext "18" (00011000) XOR Key "A" (01000001) = "Y" (01011001)

- Ciphertext "0b" (00001011) XOR Key "N" (01001110) = "E" (01000101)
- Ciphertext "0e" (00001110) XOR Key "K" (01001011) = "E" (01000101)
- Ciphertext "0c" (00001100) XOR Key "H" (01001000) = "D" (01000100)
- Ciphertext "00" (00000000) XOR Key "A" (01000001) = "A" (01000001)

4. **Convert Binary to Plaintext**:
   After applying the XOR operation for each pair of ciphertext and key, we get the binary values corresponding to each character of the original plaintext.

The decrypted binary values are:

00010011 (S)

01000001 (A)

01011001 (Y)

01000101 (E)

01000101 (E)

01000100 (D)

01000001 (A)

Converting these back to text gives us:

Plaintext: SAYEEDA

Final Result:

The ciphertext **1809180b0e0c00** is decrypted back to the original plaintext **"SAYEEDA"** using the key **"KHANKHA"** with the same XOR operation.

Pseudo Code

```
def vernam_decrypt_hex(ciphertext_hex, key):
    decrypted_text = []
    # Ensure the key length matches the plaintext length
    if len(ciphertext_hex) // 2 != len(key):
        raise ValueError("Key length must be equal to the ciphertext character
length divided by 2.")
    # Convert each pair of hex digits to its integer value
    ciphertext_bytes = [int(ciphertext_hex[i:i+2], 16) for i in range(0,
len(ciphertext_hex), 2)]
    for i in range(len(ciphertext_bytes)):
        # XOR each byte of ciphertext with each corresponding character in key
        decrypted_char = chr(ciphertext_bytes[i] ^ ord(key[i]))
        decrypted_text.append(decrypted_char)
    return ''.join(decrypted_text)

ciphertext_hex = input("Enter The Cipher Text: ")
key = input("Enter The Key in Capital Letter: ")
```

```
decrypted_text = vernam_decrypt_hex(ciphertext_hex, key)
print("Ciphertext (Hex):", ciphertext_hex)
print("Key:", key)
print("Decrypted Text:", decrypted_text)
```

Input and Output:

```
Enter The Cipher Text: 1809180b0e0c00
Enter The Key in Capital Letter: KHANKHA
Ciphertext (Hex): 1809180b0e0c00
Key: KHANKHA
Decrypted Text: SAYEEDA
```