Pabna University of Science and Technology

Faculty of Engineering and Technology

Department of Information and Communication Engineering

Lab Report

Course Title: **Cryptography and Computer Security Sessional**

Course Code: **ICE-4108**

Submitted By:	**Submitted To:**
MD. Mahamudul Hasan **Roll: 200609** **Session**: 2019-2020 4th Year 1st semester Department of Information and Communication Engineering, **PUST.**	**Md. Anwar Hossain** **Professor** Department of Information and Communication Engineering, **PUST.**

Date of Submission: **12/11/2024**

Signature

INDEX

Experiment No: 01

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

Objective:

To implement the Caesar cipher encryption technique, explore its functionality in encoding and decoding messages, and analyze its limitations in cryptographic security.

Theory:

The oldest and most basic cryptographic technique is the Caesar cipher. The Caesar cipher method, often known as a shift cipher or additive cipher, is based on a mono-alphabetic cipher. To communicate with his officers, Julius Caesar employed the shift cipher, often known as the additive cipher. The shift cipher technique is therefore known as the Caesar cipher. The Caesar cipher is a type of substitution cipher in which every letter in the plain text is swapped out for a different letter.

The Caesar cipher can be better understood by using the following example: if we shift by 1, then A will be replaced by B, B by C, C by D, D by C, and so on until the plain text is completed.

Caesar ciphers are a weak cryptography technique. It is easily hackable. It indicates that the message encrypted using this technique is simply decryptable.

- **For Encryption:** C=(P + K) mod 26 where 'P' is the character in plain text, 'K' is the key, and 'C' is the Cipher.
- **For Decryption**: P=(C-K) mod 26

Algorithm:

For Encryption:

Therefore, an integer value called a shift—which tells how many positions each letter of the text has been shifted down—is required in order to cipher a particular text.
Modular arithmetic can be used to express the encryption by first converting the letters into numbers using the following scheme: A = 0, B = 1,…, Z = 25. A mathematical description of the encryption of a letter by a shift n is as follows.

For instance, the letter A would be changed to the letter D if the shift was 3, and the letters B and C would be changed to E, F, and so forth. The alphabet is twisted so that it begins at A again after Z.

Here's an example of encrypting the message "HELLO" with a shift of three using the Caesar cypher:

1. Put the following plaintext message in writing: HELLO
2. Select a shift value. We shall employ a shift of 3 in this instance.
3. Every letter in the plaintext message should be changed to the letter three letters to the right in the alphabet.

   H becomes K (shift 3 from H)
   E becomes H (shift 3 from E)
   L becomes O (shift 3 from L)
   L becomes O (shift 3 from L)
   O becomes R (shift 3 from O)
4. The encrypted message is now “KHOOR”.

Decryption:

To decrypt the message, you simply need to shift each letter back by the same number of positions. In this case, you would shift each letter in “KHOOR” back by 3 positions to get the original message, “HELLO”.

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

(Decryption Phase with shift n)

Code:

```
def encrypt(text, shift):
    result = ""
    # Traverse text character by character
    for char in text:
        # Encrypt uppercase characters
        if char.isupper():
            result += chr((ord(char) + shift - 65) % 26 + 65)
        # Encrypt lowercase characters
        elif char.islower():
            result += chr((ord(char) + shift - 97) % 26 + 97)
        else:
            # Keep the character as it is (like spaces or punctuation)
            result += char
```

```
    return result

def decrypt(text, shift):
    # Decryption is simply the reverse of encryption
    return encrypt(text, -shift)

# Example usage
text = "Hello, World!"
shift = 3
encrypted_text = encrypt(text, shift)
print("Encrypted Text:", encrypted_text)

decrypted_text = decrypt(encrypted_text, shift)
print("Decrypted Text:", decrypted_text)
```

Input:

Text: Hello, World

Output:

Encrypted Text: Khoor, Zruog!
Decrypted Text: Hello, World!

Experiment No: 02

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

Objective:

The objective of this lab is to understand and implement the mono-alphabetic cipher, an encryption technique that uses a fixed substitution scheme to replace each letter in the plaintext with a unique corresponding letter.

Theory:

The Mono-Alphabetic Cipher is a type of substitution cipher where each letter in the plaintext is mapped to a unique letter in the ciphertext alphabet. Unlike the Caesar cipher, which shifts letters by a set amount, the mono-alphabetic cipher can use any permutation of the alphabet, providing 26! (factorial of 26) possible mappings, theoretically increasing its strength.

However, due to predictable letter frequencies in languages (e.g., "E" is the most common letter in English), the mono-alphabetic cipher is vulnerable to frequency analysis. By analyzing letter frequency in the ciphertext, a cryptanalyst can often deduce the substitution pattern and break the cipher. While the mono-alphabetic cipher was historically important, modern encryption methods have largely replaced it due to these weaknesses.

plain: 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

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

Algorithm:

For Encryption:

i. **Choose a Key Mapping:** Define a unique substitution for each letter in the alphabet. For instance, 'A' could map to 'Q', 'B' to 'W', etc. This key can be a randomly shuffled alphabet or a pre-defined mapping.

ii. **Substitute Each Letter:** For each letter in the plaintext:

- Locate the letter in the key mapping.
- Substitute it with the corresponding letter in the ciphertext alphabet.

iii. **Return the Ciphertext:** After substituting all letters, the result is the encrypted message.

For Decryption:

i. **Use the Inverse Mapping:** Use the inverse of the encryption key to map each letter in the ciphertext back to its original letter in the plaintext.

ii. **Substitute Each Letter:** For each letter in the ciphertext:

- Locate the letter in the inverse mapping.
- Substitute it with the corresponding plaintext letter.

iii. **Return the Plaintext:** After reversing all substitutions, the result is the decrypted message.

Code:

```
import random
import string

def generate_mono_alphabetic_cipher():
    # Create a dictionary for encryption mapping
    letters = list(string.ascii_uppercase)
    shuffled_letters = letters[:]
    random.shuffle(shuffled_letters)
    cipher = dict(zip(letters, shuffled_letters))
    return cipher

def encrypt(text, cipher):
    result = ""
    for char in text.upper():
        if char in cipher:
            result += cipher[char]  # Encrypt character using cipher mapping
        else:
            result += char  # Non-alphabet characters remain the same
    return result

def decrypt(text, cipher):
    # Create a reverse dictionary for decryption mapping
    reverse_cipher = {v: k for k, v in cipher.items()}
    result = ""
```

```
    for char in text.upper():
        if char in reverse_cipher:
            result += reverse_cipher[char]  # Decrypt character using reverse mapping
        else:
            result += char  # Non-alphabet characters remain the same
    return result

# Generate cipher and example usage
cipher = generate_mono_alphabetic_cipher()
print("Cipher Mapping:", cipher)

text = "HELLO WORLD"
encrypted_text = encrypt(text, cipher)
print("Encrypted Text:", encrypted_text)

decrypted_text = decrypt(encrypted_text, cipher)
print("Decrypted Text:", decrypted_text)
```

Input:

Text= "HELLO WORLD"

Output:

Encrypted Text: IVHHB XBJHT

Decrypted Text: HELLO WORLD

Experiment No: 03

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

Objective:

To understand and implement a brute force attack on a cipher, testing all possible keys to decrypt a message. This lab aims to demonstrate how brute force methods can break simple ciphers, such as the Caesar cipher, and to highlight the importance of using sufficiently large key spaces in encryption to resist brute-force attacks.

Theory:

Brute force is a technique used in cybersecurity to crack encrypted messages or passwords by systematically trying all possible combinations until the correct one is found. This method relies on the assumption that the encryption algorithm used is known, but the key or password is unknown. In the field of classical cryptography, brute force attacks have been historically employed to decrypt messages encrypted using various ciphers based on modular arithmetic.

Three important characteristics of this problem enabled us to use a bruteforce cryptanalysis:

1. The encryption and decryption algorithms are known.

2. There are only 25 keys to try.

3. The language of the plaintext is known and easily recognizable.

Algorithm:

For Encryption:

Define the Ciphertext:

- Start with a given ciphertext (the encrypted text) that you wish to decrypt.

Initialize Possible Keys:

- Identify the possible range of keys (shifts) to test. For example, in the Caesar cipher, the key range is from 0 to 25 (for 26 letters of the alphabet).

Loop Over All Keys:

1. Decrypt the Ciphertext:
    - Attempt to decrypt the ciphertext using the current key.
2. Evaluate the Decrypted Text:

- Check if the resulting plaintext is meaningful (in practical scenarios, this could involve checking against a dictionary of common words or language patterns).

3. Stop if Correct:
   - If the decrypted text is correct or makes sense, stop the attack and output the result.

For Decryption:

Decryption Using Brute Force Attack

1. Initialize Key Attempts: Set up a loop to try each possible key value in the cipher's keyspace.
2. Attempt Each Key:
   - For each key in the keyspace:
     - Decrypt the ciphertext by reversing the encryption process using the current key.
     - Generate a possible plaintext for each key attempt.
3. Check for Meaningful Output:
   - After each decryption attempt, check if the output is meaningful (e.g., contains recognizable words in a given language).
   - This step may involve automated checks (such as dictionary matching) or manual inspection to identify the correct plaintext.
4. Return the Correct Plaintext: Once the correct key is identified, return the successfully decrypted message.

Code:

```
def caesar_decrypt_brute_force(cipher_text):
    # Try each possible shift (from 1 to 25) for decryption
    for shift in range(1, 26):
        decrypted_text = ""
        for char in cipher_text:
            if char.isupper():
                decrypted_text += chr((ord(char) - shift - 65) % 26 + 65)
            elif char.islower():
                decrypted_text += chr((ord(char) - shift - 97) % 26 + 97)
```

```
            else:
                decrypted_text += char  # Non-alphabet characters remain the same
        print(f"Shift {shift}: {decrypted_text}")

# Example encrypted text
cipher_text = "Khoor Zruog"  # Encrypted "Hello World" with a shift of 3
caesar_decrypt_brute_force(cipher_text)
```

Input:

cipher_text = "Khoor Zruog"

Output:

Shift 1: Jgnnq Yqtnf
Shift 2: Ifmmp Xpsme
Shift 3: Hello World
Shift 4: Gdkkn Vnqkc
Shift 5: Fcjjm Umpjb
Shift 6: Ebiil Tloia
Shift 7: Dahhk Sknhz
Shift 8: Czggj Rjmgy
Shift 9: Byffi Qilfx
Shift 10: Axeeh Phkew
Shift 11: Zwddg Ogjdv
Shift 12: Yvccf Nficu
Shift 13: Xubbe Mehbt
Shift 14: Wtaad Ldgas
Shift 15: Vszzc Kcfzr
Shift 16: Uryyb Jbeyq
Shift 17: Tqxxa Iadxp
Shift 18: Spwwz Hzcwo
Shift 19: Rovvy Gybvn
Shift 20: Qnuux Fxaum
Shift 21: Pmttw Ewztl
Shift 22: Olssv Dvysk
Shift 23: Nkrru Cuxrj
Shift 24: Mjqqt Btwqi
Shift 25: Lipps Asvph

Experiment No: 04

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

Objectives:

To understand and implement the Hill cipher for encryption and decryption using matrix-based transformations of plaintext. This lab will explore how matrix operations, specifically multiplication and modular arithmetic, are used in cryptography to encode and decode messages securely.

Theory:

The Hill cypher is a linear algebra-based polygraphic substitution cypher. A integer modulo 26 is used to represent each letter. Although it is frequently employed, the straightforward scheme A = 0, B = 1, …, Z = 25 is not a necessary component of the cypher. Each block of n letters, which is regarded as an n-component vector, is multiplied by an invertible (n × n) matrix against modulus 26 in order to encrypt a message. Each block is multiplied by the inverse of the encryption matrix in order to decrypt the message.

The cypher key, which is the encryption matrix, should be selected at random from the set of invertible (n × n) matrices (modulo 26).

Algorithm:

To encrypt the text using hill cipher, we need to perform the following operation.

$$E(K, P) = (K * P) \bmod 26$$

Where **K** is the key matrix and **P** is plain text in **vector form**. Matrix multiplication of K and P generates the encrypted ciphertext.

For Encryption:

1. **Choose a Key Matrix**:

   Select an n×nn \times nn×n matrix (key matrix) with integer values. The matrix should be invertible in modulo 26 (meaning its determinant has no common factor with 26) for decryption to be possible.

2. **Prepare the Plaintext**:

    i. Convert each letter in the plaintext into a numerical equivalent (e.g., A=0, B=1, ..., Z=25).

    ii. Divide the plaintext into blocks of size nnn. If the last block is short, pad it with extra letters (usually "X").

3. **Multiply the Key Matrix by Plaintext Vectors**:

    For each block of plaintext, create a vector and multiply it by the key matrix. Compute the resulting vector modulo 26 to get the encrypted text.

4. **Convert the Result to Ciphertext**:

    Convert each number in the resulting vector back to a letter.

For Decryption:

1. **Calculate the Inverse of the Key Matrix**:

    - Find the modular inverse of the key matrix modulo 26. This inverse will only exist if the determinant of the matrix is coprime with 26.

2. **Multiply the Ciphertext by the Inverse Matrix**:

    - For each block of ciphertext, create a vector and multiply it by the inverse key matrix. Compute the resulting vector modulo 26 to retrieve the plaintext vector.

3. **Convert the Result Back to Text**:

    - Convert each number in the resulting plaintext vector back to a letter.

Code:

```
import numpy as np

def mod_inverse(matrix, mod):
    # Calculate the modular inverse of a matrix under mod 26
    det = int(round(np.linalg.det(matrix)))  # Determinant of matrix
    det_inv = pow(det, -1, mod)  # Modular inverse of determinant
    matrix_mod_inv = (
        det_inv * np.round(det * np.linalg.inv(matrix)).astype(int) % mod
    )
```

```
    return matrix_mod_inv % mod

def encrypt_hill(plaintext, key_matrix):
    # Convert plaintext to numbers (A=0, B=1, ..., Z=25)
    plaintext_nums = [ord(char) - ord('A') for char in plaintext.upper()]

    # Pad plaintext if necessary
    if len(plaintext_nums) % 2 != 0:
        plaintext_nums.append(ord('X') - ord('A'))

    # Encrypt the plaintext in blocks
    ciphertext = ""
    for i in range(0, len(plaintext_nums), 2):
        block = np.array(plaintext_nums[i:i+2])
        encrypted_block = np.dot(key_matrix, block) % 26
        ciphertext += ''.join(chr(num + ord('A')) for num in encrypted_block)

    return ciphertext

def decrypt_hill(ciphertext, key_matrix):
    # Compute the modular inverse of the key matrix
    key_matrix_inv = mod_inverse(key_matrix, 26)

    # Convert ciphertext to numbers
    ciphertext_nums = [ord(char) - ord('A') for char in ciphertext.upper()]

    # Decrypt the ciphertext in blocks
    plaintext = ""
    for i in range(0, len(ciphertext_nums), 2):
        block = np.array(ciphertext_nums[i:i+2])
        decrypted_block = np.dot(key_matrix_inv, block) % 26
        plaintext += ''.join(chr(int(num) + ord('A')) for num in decrypted_block)

    return plaintext

# Example usage
key_matrix = np.array([[3, 3], [2, 5]])
plaintext = "HELLO"

print("Key Matrix:\n", key_matrix)
encrypted_text = encrypt_hill(plaintext, key_matrix)
print("Encrypted Text:", encrypted_text)

decrypted_text = decrypt_hill(encrypted_text, key_matrix)
print("Decrypted Text:", decrypted_text)
```

Input:

Key Matrix:

[[3 3]

[2 5]]

Output:

Encrypted Text: HIOZHN
Decrypted Text: HELLOX

Experiment No: 05

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

Objectives: To understand and implement the Playfair cipher for encrypting messages by substituting digraphs (pairs of letters) using a 5x5 grid of letters. This lab aims to explore how the Playfair cipher uses digraphs to provide stronger encryption than simple mono-alphabetic ciphers, reducing the effectiveness of frequency analysis attacks.

Theory:

The Playfair Cipher is a poly graphic substitution cipher that encrypts pairs of letters (digraphs) instead of single letters, which makes it more secure than mono-alphabetic ciphers like the Caesar cipher. Invented by Sir Charles Wheatstone in 1854, the cipher gained popularity after being promoted by Lord Playfair.

In the Playfair cipher, a 5x5 grid of letters, also known as the key square, is created using a keyword. The grid omits one letter (typically "J") to fit the 26 letters of the alphabet. The keyword is written into the grid, followed by the remaining letters in alphabetical order, excluding any repeats.

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

Fig: Key Matrix

The Playfair cipher's use of digraphs and its reliance on spatial relationships make it stronger than mono-alphabetic ciphers, as it obscures letter frequencies and patterns more effectively. This polygraphic substitution approach enhances encryption security and illustrates the foundational principles of positional ciphers.

Algorithm:

For Encryption:

1. **Create the Key Square**:
    - Write the keyword in the 5x5 grid, excluding repeated letters.
    - Fill the remaining spaces with the rest of the alphabet in order, omitting "J" (or combining "I" and "J").
2. **Prepare the Plaintext**:
    - Split the plaintext into digraphs (pairs of letters).
    - If a digraph contains the same letter twice, insert a filler letter (commonly "X") between them.
    - If the plaintext has an odd number of letters, add a filler letter to complete the final pair.
3. **Encrypt Each Digraph**:
    - For each pair of letters:
        - **Same Row**: Replace each letter with the letter immediately to its right, wrapping to the beginning of the row if necessary.
        - **Same Column**: Replace each letter with the letter immediately below it, wrapping to the top of the column if necessary.
        - **Rectangle Rule**: If the letters form a rectangle, replace each letter with the letter on the same row at the opposite corner of the rectangle.
4. **Combine the Encrypted Digraphs**:
    - Concatenate all encrypted digraphs to form the final ciphertext.

Code:

```
def generate_key_matrix(keyword):
    # Remove duplicates and combine 'J' with 'I'
    matrix_key = []
    for char in keyword.upper():
        if char not in matrix_key and char != 'J':
            matrix_key.append(char)
```

```
    # Add remaining letters to complete 5x5 matrix
    for char in "ABCDEFGHIKLMNOPQRSTUVWXYZ":
        if char not in matrix_key:
            matrix_key.append(char)

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

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

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

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

def prepare_text(plaintext):
    # Remove spaces and handle identical pairs
    text = plaintext.upper().replace(" ", "").replace("J", "I")
    prepared_text = ""
    i = 0
    while i < len(text):
        # Handle the case where two letters in a pair are the same
        prepared_text += text[i]
        if i + 1 < len(text) and text[i] == text[i + 1]:
            prepared_text += 'X'
            i += 1
        elif i + 1 < len(text):
            prepared_text += text[i + 1]
            i += 2
```

```
        else:
            prepared_text += 'X'  # Add 'X' if there's an odd number of letters
            i += 1
    return prepared_text

def encrypt_playfair(plaintext, keyword):
    matrix = generate_key_matrix(keyword)
    prepared_text = prepare_text(plaintext)

    ciphertext = ""
    for i in range(0, len(prepared_text), 2):
        pair = prepared_text[i:i + 2]
        ciphertext += encrypt_pair(pair, matrix)

    return ciphertext

# Example usage
keyword = "KEYWORD"
plaintext = "HELLO WORLD"
ciphertext = encrypt_playfair(plaintext, keyword)
print("Key Matrix:")
matrix = generate_key_matrix(keyword)
for row in matrix:
    print(row)
print("\nPlaintext:", plaintext)
print("Ciphertext:", ciphertext)
```

Input:

keyword = "KEYWORD"

plaintext = "HELLO WORLD"

Output:

Ciphertext: GYIZSCOKCFBU

Experiment No: 06

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

Objectives:

To understand and implement the decryption process of the Playfair cipher by reversing the encryption rules to retrieve the original plaintext from the ciphertext.

Theory:

The Playfair Cipher is a poly graphic substitution cipher that encrypts and decrypts text by processing pairs of letters (digraphs). Decryption in the Playfair cipher involves reversing the positional transformations applied during encryption.

The cipher uses a 5x5 grid (or key square) generated from a keyword, which excludes one letter of the alphabet, typically "J". During decryption, each encrypted digraph is converted back to its corresponding plaintext by following specific rules based on the relative positions of the letters in the key square.

Rules for Decryption:

1. **Same Row**: If both letters in the digraph are in the same row, replace each letter with the letter immediately to its left, wrapping to the end of the row if necessary.
2. **Same Column**: If both letters in the digraph are in the same column, replace each letter with the letter immediately above, wrapping to the bottom of the column if necessary.
3. **Rectangle Rule**: If the letters form the corners of a rectangle, replace each letter with the letter in the same row at the opposite corner of the rectangle.

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

Fig: Key Matrix

Algorithm:

For decryption:

Generate the Key Square:

- Create a 5x5 matrix (key square) using the keyword.
- Write the keyword into the matrix, excluding repeated letters.
- Fill the remaining spaces with the unused letters of the alphabet in order, excluding "J" (or combining "I" and "J").

Prepare the Ciphertext:

- Divide the ciphertext into digraphs (pairs of letters).
- Ensure that no filler letters (such as "X") interfere unless they were part of the encryption process.

Decrypt Each Digraph:

- For each digraph, locate both letters in the key square.
- Apply the following rules to find the original plaintext digraph:
  i. **Same Row**: Replace each letter with the letter immediately to its left, wrapping around to the right end of the row if necessary.
  ii. **Same Column**: Replace each letter with the letter immediately above it, wrapping around to the bottom if necessary.
  iii. **Rectangle Rule**: If the letters form a rectangle, replace each letter with the letter on the same row at the opposite corner of the rectangle.

Combine the Decrypted Digraphs:

- Concatenate all decrypted digraphs to form the plaintext message.
- If filler letters were added during encryption (e.g., "X" between repeated letters), remove them as appropriate.

Code:

```
def generate_key_matrix(keyword):
    # Remove duplicates and combine 'J' with 'I'
    matrix_key = []
    for char in keyword.upper():
```

```
        if char not in matrix_key and char != 'J':
            matrix_key.append(char)

    # Add remaining letters to complete 5x5 matrix
    for char in "ABCDEFGHIKLMNOPQRSTUVWXYZ":
        if char not in matrix_key:
            matrix_key.append(char)

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

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

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

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

def decrypt_playfair(ciphertext, keyword):
    matrix = generate_key_matrix(keyword)

    plaintext = ""
    for i in range(0, len(ciphertext), 2):
        pair = ciphertext[i:i + 2]
        plaintext += decrypt_pair(pair, matrix)

    # Remove padding 'X' added during encryption
    final_plaintext = ""
    i = 0
    while i < len(plaintext):
        final_plaintext += plaintext[i]
        # Skip padding character 'X' if it was inserted between identical letters
```

```
        if i + 2 < len(plaintext) and plaintext[i] == plaintext[i + 2] and plaintext[i + 1]
== 'X':
            i += 2
        else:
            i += 1
    return final_plaintext

# Example usage
keyword = "KEYWORD"
ciphertext = "GYIZSCOKCFBU"  # Encrypted text of "HELLO WORLD"
plaintext = decrypt_playfair(ciphertext, keyword)

print("Key Matrix:")
matrix = generate_key_matrix(keyword)
for row in matrix:
    print(row)
print("\nCiphertext:", ciphertext)
print("Plaintext:", plaintext)
```

Input:

keyword = "KEYWORD"
ciphertext = "GYIZSCOKCFBU"

Output:

Plaintext: HELLOWORLDX

Experiment No: 07

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

Objective:

To understand and implement encryption using a poly-alphabetic cipher, such as the Vigenère cipher, which uses multiple alphabets to obscure the frequency of individual letters.

Theory:

A **Poly-Alphabetic Cipher** is an encryption technique that uses multiple substitution alphabets to encrypt a message. Unlike mono-alphabetic ciphers, which use a single fixed shift or substitution throughout, poly-alphabetic ciphers change the substitution at intervals, based on a keyword or sequence.

The **Vigenère Cipher** is one of the most famous poly-alphabetic ciphers. It employs a keyword to determine the shift for each letter in the plaintext. Each letter in the keyword corresponds to a shift value (A=0, B=1, …, Z=25), and this shift value is applied cyclically across the message.

Key Aspects of the Vigenère Cipher:

1. **Keyword**: The keyword determines the shift for each letter in the plaintext. The length and repetition of the keyword impact the encryption's strength.
2. **Shifting Mechanism**: Each letter in the plaintext is shifted based on the corresponding letter in the keyword, creating a poly-alphabetic pattern that disguises letter frequencies.
3. **Security**: The Vigenère cipher is stronger than mono-alphabetic ciphers because it avoids a fixed substitution pattern. However, it is still vulnerable to advanced frequency analysis, especially if the keyword is short or the ciphertext is long.
4.

Example

Plaintext: HELLO
Keyword: KEY

1. Convert "HELLO" and "KEY" to numeric form:
    - Plaintext: H=7, E=4, L=11, L=11, O=14
    - Keyword (repeated): K=10, E=4, Y=24, K=10, E=4

2. Encrypt each character by shifting:
    - H (7) + K (10) = R (17)
    - E (4) + E (4) = I (8)
    - L (11) + Y (24) = J (9) (mod 26)
    - L (11) + K (10) = V (21)
    - O (14) + E (4) = S (18)

Ciphertext: RIJVS

Algorithm:

For Encryption

1. **Convert Keyword and Plaintext to Numeric Form**:
    - Assign each letter a numerical value (A=0, B=1, …, Z=25).
    - Convert the keyword into a sequence of numbers based on these values.
    - Repeat the keyword to match the length of the plaintext (e.g., if the keyword is "KEY" and the plaintext has 15 characters, repeat the keyword as "KEYKEYKEYKEYKEY").
2. **Encrypt Each Character**:
    - For each letter in the plaintext:
        - Find the corresponding letter in the repeated keyword.
        - Shift the plaintext letter forward in the alphabet by the numeric value of the keyword letter (modulo 26 to wrap around if necessary).
        - Convert the shifted numeric value back to a letter to form the encrypted character.
3. **Construct the Ciphertext**:
    - Concatenate all encrypted characters to form the final ciphertext.

Code:

```
def vigenere_encrypt(plaintext, keyword):
    # Prepare the keyword by repeating it to match the length of the plaintext
    keyword = keyword.upper()
    plaintext = plaintext.upper()
      keyword_repeated = (keyword * ((len(plaintext) // len(keyword)) + 1))[:len(plaintext)]

    ciphertext = ""
    for p, k in zip(plaintext, keyword_repeated):
        if p.isalpha():  # Only encrypt alphabetic characters
            # Shift character by the keyword's character value
            shift = ord(k) - ord('A')
            encrypted_char = chr(((ord(p) - ord('A') + shift) % 26) + ord('A'))
            ciphertext += encrypted_char
        else:
            ciphertext += p  # Non-alphabet characters remain unchanged

    return ciphertext

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

print("Plaintext:", plaintext)
print("Keyword:", keyword)
print("Ciphertext:", ciphertext)
```

Input:

Plaintext: HELLO WORLD

Keyword: KEY

Output:

Ciphertext: RIJVS GSPVH

Experiment No: 08

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

Objectives:

To understand and implement decryption of a poly-alphabetic cipher, such as the Vigenère cipher, by reversing the encryption process using a keyword-based shifting mechanism.

Theory:

A **Poly-Alphabetic Cipher** uses multiple shifting alphabets to encode a message, making it more secure than simple mono-alphabetic ciphers by varying the substitution for each letter. The **Vigenère Cipher** is a popular example that uses a keyword to generate a sequence of shifts for each letter in the ciphertext. Each letter in the ciphertext is shifted backward by a certain number of positions determined by the corresponding letter in the keyword.

In the Vigenère cipher:

1. **Keyword**: The keyword is used to generate the shifts required for decryption. Each letter in the keyword corresponds to a shift value (A=0, B=1, …, Z=25), and this value is used to determine how far to shift each letter in the ciphertext backward.
2. **Inverse Shifting**: The decryption process involves reversing the shifts applied during encryption. Each letter in the ciphertext is shifted backward by the value corresponding to the repeated keyword letter.
3. **Security**: The Vigenère cipher's poly-alphabetic nature makes it resistant to simple frequency analysis, but it can still be vulnerable to attacks if the keyword is too short or if the ciphertext is very long.

Example

Ciphertext: RIJVS
Keyword: KEY

1. Convert "RIJVS" and "KEY" to numeric form:
    - Ciphertext: R=17, I=8, J=9, V=21, S=18
    - Keyword (repeated): K=10, E=4, Y=24, K=10, E=4

2. Decrypt each character by shifting backward:
    - R (17) - K (10) = H (7) (mod 26)
    - I (8) - E (4) = E (4)
    - J (9) - Y (24) = L (11) (mod 26)
    - V (21) - K (10) = L (11)
    - S (18) - E (4) = O (14)

Plaintext: HELLO

Algorithm:

For Decryption:

1. **Convert Keyword and Ciphertext to Numeric Form**:
    - Assign each letter a numerical value (A=0, B=1, …, Z=25).
    - Convert the keyword into a sequence of numbers based on these values.
    - Repeat the keyword until it matches the length of the ciphertext, creating a keyword sequence for each letter.
2. **Decrypt Each Character**:
    - For each letter in the ciphertext:
        - Find the corresponding letter in the repeated keyword.
        - Shift the ciphertext letter backward by the numeric value of the keyword letter (using modulo 26 to wrap around if necessary).
        - Convert the shifted numeric value back to a letter to form the decrypted character.
3. **Plaintext**:
    - Concatenate all decrypted characters to form the final plaintext message.

Code:

```
def vigenere_decrypt(ciphertext, keyword):
    # Prepare the keyword by repeating it to match the length of the ciphertext
    keyword = keyword.upper()
    ciphertext = ciphertext.upper()
      keyword_repeated = (keyword * ((len(ciphertext) // len(keyword)) + 1))[:len(ciphertext)]

    plaintext = ""
    for c, k in zip(ciphertext, keyword_repeated):
        if c.isalpha():  # Only decrypt alphabetic characters
            # Reverse the shift using the keyword's character value
            shift = ord(k) - ord('A')
            decrypted_char = chr(((ord(c) - ord('A') - shift + 26) % 26) + ord('A'))
            plaintext += decrypted_char
        else:
            plaintext += c  # Non-alphabet characters remain unchanged

    return plaintext

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

print("Ciphertext:", ciphertext)
print("Keyword:", keyword)
print("Plaintext:", plaintext)
```

Input:

Ciphertext: RIJVS GSPVH

Keyword: KEY

Output:

Plaintext: HELLO WORLD

Experiment No: 09

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

Objective:

To understand and implement encryption using the Vernam cipher, an early example of a perfectly secure cryptographic system when used with a truly random key of the same length as the plaintext.

Theory:

The Vernam Cipher, also known as the One-Time Pad, is a symmetric key cipher developed by Gilbert Vernam in 1917. It is based on the principle that each character in the plaintext is combined with a corresponding character in a key of the same length using the XOR (exclusive OR) operation. If the key is truly random and used only once, the Vernam cipher provides theoretically perfect security, as each plaintext character is completely obscured by a unique, unpredictable key character.

Key Characteristics of the Vernam Cipher:

1. **Key Length**: The key must be exactly the same length as the plaintext to prevent patterns from emerging, which ensures that each character has a unique encryption key.
2. **One-Time Use**: For perfect security, the key must never be reused for any other encryption.
3. **XOR Operation**: The cipher uses the XOR operation, which is both reversible and secure when used with a truly random key.
4. **Security**: When used correctly, the Vernam cipher is **unbreakable** because it doesn't allow for frequency analysis or other cryptographic attacks, as each character is independently and randomly encrypted.

XOR Operation: The XOR operation works as follows:

- 1 XOR 1 = 0
- 0 XOR 0 = 0
- 1 XOR 0 = 1
- 0 XOR 1 = 1

The Vernam cipher, each bit of the plaintext is XORed with the corresponding bit of the key to generate the ciphertext. This operation is symmetric, meaning the same process is used to decrypt as to encrypt.

Example

Plaintext: "HELLO"
Key: "XMCKL" (same length as the plaintext and truly random)

1. Convert "HELLO" and "XMCKL" to binary:
    - H = 01001000, E = 01000101, L = 01001100, L = 01001100, O = 01001111
    - X = 01011000, M = 01001101, C = 01000011, K = 01001011, L = 01001100
2. Apply XOR between each corresponding binary pair:
    - H (01001000) XOR X (01011000) = 00010000 (P)
    - E (01000101) XOR M (01001101) = 00001000 (H)
    - L (01001100) XOR C (01000011) = 00001111 (O)
    - L (01001100) XOR K (01001011) = 00000111 (G)
    - O (01001111) XOR L (01001100) = 00000011 (C)
3. Convert binary output back to characters:
    - Ciphertext: "PHOGC"

Algorithm:

For Encryption:

1. **Convert Plaintext and Key to Binary**:
    - Represent each character in the plaintext and the key as an 8-bit binary value (ASCII or Unicode).
2. **Apply XOR Operation**:
    - For each bit in the plaintext, apply the XOR operation with the corresponding bit in the key.
    - The result of each XOR operation is the encrypted bit for that position in the ciphertext.
3. **Convert to Ciphertext**:
    - Convert the binary output of the XOR operations back into characters to form the final ciphertext.
4. **Output the Encrypted Message**:
    - The ciphertext should appear random and bear no resemblance to the plaintext, ensuring high security.

Code:

```
def vernam_encrypt(plaintext, key):
    # Ensure the key is the same length as the plaintext
    if len(plaintext) != len(key):
        raise ValueError("Key must be the same length as plaintext")

    # Convert plaintext and key to uppercase for uniformity
    plaintext = plaintext.upper()
    key = key.upper()

    ciphertext = ""
    for p, k in zip(plaintext, key):
        # XOR the ASCII values of each character and convert back to a character
        encrypted_char = chr(((ord(p) - ord('A')) ^ (ord(k) - ord('A'))) + ord('A'))
        ciphertext += encrypted_char

    return ciphertext

# Example usage
plaintext = "HELLO"
key = "XMCKL"  # Key must be the same length as plaintext
ciphertext = vernam_encrypt(plaintext, key)

print("Plaintext:", plaintext)
print("Key:", key)
print("Ciphertext:", ciphertext)
```

Input:

Plaintext: HELLO

Key: XMCKL

Output:

Ciphertext: QIJBF

Experiment No: 10

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

Objective:

To understand and implement the decryption process of the Vernam cipher (One-Time Pad), which uses an XOR operation with a one-time, random key of the same length as the ciphertext.

Theory:

The Vernam Cipher or One-Time Pad is a symmetric cipher, meaning the same key is used for both encryption and decryption. The encryption process applies an XOR operation between each bit of the plaintext and a corresponding bit of a randomly generated key. Since XOR is a reversible operation, the ciphertext can be decrypted by applying XOR with the same key.

Key Characteristics:

1. **Key Length**: The key must be as long as the ciphertext and entirely random to maintain perfect security.
2. **One-Time Use:** For absolute security, the key must be unique and never reused for another message.
3. **XOR Reversibility:** XOR operation is self-reversing, meaning $C \oplus K = P$ if $P \oplus K = C$ (where C is ciphertext, P is plaintext, and K is the key).
4. **Perfect Security**: When used correctly, the Vernam cipher is theoretically unbreakable, as each ciphertext bit is independent of others due to random key application.

Example

Ciphertext: "PHOGC"
Key: "XMCKL" (same key used during encryption)

1. Convert "PHOGC" and "XMCKL" to binary:
    - P = 00010000, H = 00001000, O = 00001111, G = 00000111, C = 00000011

- X = 01011000, M = 01001101, C = 01000011, K = 01001011, L = 01001100

2. Apply XOR between each corresponding binary pair:
    - P (00010000) XOR X (01011000) = H (01001000)
    - H (00001000) XOR M (01001101) = E (01000101)
    - O (00001111) XOR C (01000011) = L (01001100)
    - G (00000111) XOR K (01001011) = L (01001100)
    - C (00000011) XOR L (01001100) = O (01001111)
3. Convert binary output back to characters:
    - Plaintext: "HELLO"

Algorithm:

Decryption

1. Convert Ciphertext and Key to Binary:
    - Convert each character of the ciphertext and the key to binary form (e.g., 8-bit ASCII or Unicode).
2. Apply XOR Operation:
    - For each binary bit in the ciphertext, apply the XOR operation with the corresponding bit in the key.
    - This operation effectively reverses the encryption, retrieving the original plaintext bit by bit.
3. Plaintext:
    - Convert the binary result from the XOR operations back into characters to reconstruct the original plaintext.
4. Decrypted Message:
    - The final result is the original plaintext message, which should be identical to the message encrypted with the Vernam cipher if the key was correct and has not been reused.

Code:

```
def vernam_decrypt(ciphertext, key):
    # Ensure the key is the same length as the ciphertext
    if len(ciphertext) != len(key):
        raise ValueError("Key must be the same length as ciphertext")

    # Convert ciphertext and key to uppercase for uniformity
    ciphertext = ciphertext.upper()
    key = key.upper()

    plaintext = ""
    for c, k in zip(ciphertext, key):
        # XOR the ASCII values of each character and convert back to a character
        decrypted_char = chr(((ord(c) - ord('A')) ^ (ord(k) - ord('A'))) + ord('A'))
        plaintext += decrypted_char

    return plaintext

# Example usage
ciphertext = "QIJBF"
key = "XMCKL"  # Key must be the same length as ciphertext
plaintext = vernam_decrypt(ciphertext, key)

print("Ciphertext:", ciphertext)
print("Key:", key)
print("Plaintext:", plaintext)
```

Input:

Ciphertext: QIJBF
Key: XMCKL

Output:

Plaintext: HELLO