Department of Information and Communication Engineering
Pabna University of Science and Technology
Faculty of Engineering and Technology

B.Sc. (Engineering) 4th Year 1st Semester Exam-2023

Session: 2019-2020

Course Title : Cryptography and Computer Security Sessional.
Course Code: ICE-4108

Lab Report

Submitted By: MD RASHED
Roll No**:** 200613
Dept. of Information and Communication Engineering
Pabna University of Science and Technology
Pabna-6600, Bangladesh

Submitted To: Md. Anwar Hossain
Professor
Dept. of Information and Communication Engineering.
Pabna University of Science and Technology
Pabna-6600, Bangladesh

Date of Submission: 12-11-2024 Signature

INDEX

SL	Name of the Experiment	Page No.
1.	Write a program to implement encryption and decryption using Caesar cipher.	1-2
2.	Write a program to implement encryption and decryption using Mono-Alphabetic cipher.	3-5
3.	Write a program to implement encryption and decryption using Brute force attack cipher.	6-8
4.	Write a program to implement encryption and decryption using Hill cipher.	9-13
5.	Write a program to implement encryption using Playfair cipher.	14-17
6.	Write a program to implement decryption using Playfair cipher.	18-20
7.	Write a program to implement encryption using Poly-Alphabetic cipher.	21-22
8.	Write a program to implement decryption using Poly-Alphabetic cipher.	23-24
9.	Write a program to implement encryption using Vernam cipher.	25-26
10.	Write a program to implement decryption using Vernam cipher.	27-28

Problem No: 01
Problem Name: Write a program to implement encryption and decryption using Caesar cipher.

Theory:
The Caesar cipher is a classical encryption technique dating back to ancient times, named after Julius Caesar who used it for secure communication. It represents a straightforward example of a substitution cipher, where each character in the original message (plaintext) is substituted with another character located a fixed number of positions away in the alphabet. The fixed number of positions is known as the "shift" or "key." This technique involves modular arithmetic to handle wrapping around the alphabet.

Encryption Process: In the encryption phase, each character in the plaintext is replaced with a character shifted forward by a predetermined number of positions based on the key. For instance, if the key is 4, 'A' would become 'E', 'B' would transform into 'F', and so on. When the shift goes beyond 'Z', it wraps around to the start of the alphabet. This process can also apply to lowercase letters and other character sets.

Decryption Process: Decryption reverses the encryption process by shifting each character in the ciphertext backward by the same fixed key used for encryption. This reversion returns each character to its original form, effectively reconstructing the original plaintext. The decryption process relies on modular subtraction to account for the wrap-around in the alphabet.

Strengths and Weaknesses: The simplicity of the Caesar cipher makes it an effective learning tool for introducing cryptography concepts, such as message concealment and key-based transformations. However, its simplicity is also a major drawback, as the cipher is vulnerable to several attack methods. Given that the total number of possible shifts for the English alphabet is limited (25 different shifts), a brute-force attack can easily crack the cipher. Additionally, the regularity of letter frequencies in the ciphertext allows for straightforward frequency analysis attacks.

Applications: While not suitable for modern security, the Caesar cipher has historical significance and is used to introduce fundamental concepts in cryptography, such as encryption, decryption, and key management. It helps in understanding more advanced cryptographic algorithms that involve complex substitution and transposition techniques.

Python Source Code:
Encryption

```
def encrypt(text, shift):
    encrypted_text = ""
    for char in text:
        if char.isalpha():
            shift_base = 65 if char.isupper() else 97
            encrypted_char = chr((ord(char) - shift_base + shift) % 26 + shift_base)
```

```
            encrypted_text += encrypted_char
        else:
            encrypted_text += char
    return encrypted_text

def main():
    text = input("Enter the text to encrypt: ")
    shift = int(input("Enter the shift value: "))
    encrypted_text = encrypt(text, shift)
    print(f"Encrypted Text: {encrypted_text}")

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

Output:
Enter the text to encrypt: Information
Enter the shift value: 1
Encrypted Text: Jogpsnbujpo

Decryption

```
def decrypt(text, shift):
    decrypted_text = ""
    for char in text:
        if char.isalpha():
            shift_base = 65 if char.isupper() else 97
            decrypted_char = chr((ord(char) - shift_base - shift) % 26 + shift_base)
            decrypted_text += decrypted_char
        else:
            decrypted_text += char
    return decrypted_text

def main():
    text = input("Enter the text to decrypt: ")
    shift = int(input("Enter the shift value: "))
    decrypted_text = decrypt(text, shift)
    print(f"Decrypted Text: {decrypted_text}")

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

Output:
Enter the text to decrypt: Jogpsnbujpo
Enter the shift value: 1
Decrypted Text: Information

Problem No: 02
Problem Name: Write a program to implement encryption and decryption using Mono-Alphabetic cipher.

Theory:
The Mono-Alphabetic cipher is a form of substitution cipher in which each letter of the plaintext is substituted by a unique corresponding letter from a jumbled version of the alphabet. Unlike simpler ciphers like the Caesar cipher, where the substitution is determined by a uniform shift, the Mono-Alphabetic cipher allows for a more complex, one-to-one mapping between the characters of the original alphabet and the ciphertext alphabet. Each character in the plaintext maps to a unique character in the ciphertext, creating a more varied and less predictable encryption pattern.

Encryption Mechanism: The encryption process in a Mono-Alphabetic cipher involves creating a substitution key, which is essentially a rearranged version of the original alphabet. For example, the original alphabet "ABCDEFGHIJKLMNOPQRSTUVWXYZ" could be rearranged to "QWERTYUIOPASDFGHJKLZXCVBNM." Each letter in the plaintext is then replaced with the corresponding letter from the rearranged alphabet according to the original character's position. This substitution creates the encrypted message or ciphertext, which appears as a scrambled sequence of characters.

Decryption Mechanism: To decrypt a message encrypted with a Mono-Alphabetic cipher, the reverse mapping of the substitution key is used. The ciphertext is transformed back into the original plaintext by replacing each character with its corresponding letter from the original alphabet based on the decryption key. This process is the inverse of encryption, restoring the original sequence of letters in the message.

Strengths and Limitations: The Mono-Alphabetic cipher provides stronger security compared to simple ciphers like the Caesar cipher because the number of possible keys is factorially large (26!), making brute-force attacks infeasible. However, it is still susceptible to frequency analysis attacks. In English, certain letters appear more frequently (e.g., 'E', 'T'), and by analyzing the frequency of characters in the ciphertext, an attacker can deduce the likely substitutions. Thus, while the cipher is more secure than simpler forms, it does not provide adequate protection for modern security needs.

Applications and Historical Relevance: The Mono-Alphabetic cipher has been historically used for secure communication before the advent of more sophisticated encryption techniques. Today, it serves an educational role in teaching fundamental cryptographic principles, including substitution, key management, and the limitations of classical encryption. It lays the groundwork for understanding more advanced ciphers that involve multiple alphabets (polyalphabetic ciphers) or even more complex methods of permutation and substitution.

Python Source Code:

Encryption

```
import random
import string

def generate_key():
    # Create a random permutation of the alphabet
    letters = list(string.ascii_uppercase)
    random.shuffle(letters)
    return ''.join(letters)

def encrypt(text, key):
    # Create a mapping for encryption
    mapping = {original: substituted for original, substituted in zip(string.ascii_uppercase,
key)}
    encrypted_text = ""
    for char in text.upper():
        if char in mapping:
            encrypted_text += mapping[char]
        else:
            encrypted_text += char  # Non-alphabetic characters remain unchanged
    return encrypted_text

def main():
    # Generate a random substitution key
    key = generate_key()
    print(f"Generated Key: {key}")

    # Get the text to encrypt
    text = input("Enter the text to encrypt: ")
    # Encrypt the text
    encrypted_text = encrypt(text, key)
    print(f"Encrypted Text: {encrypted_text}")

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

Output:

Generated Key: IMEBXUFNZAWOPLVCRTQKHGDSJY
Enter the text to encrypt: abcdefghijklmnopqrstuvwxyz
Encrypted Text: IMEBXUFNZAWOPLVCRTQKHGDSJY

Decryption

```
import string

def decrypt(text, key):
    # Create a reverse mapping for decryption
    reverse_mapping = {substituted: original for original, substituted in
zip(string.ascii_uppercase, key)}
    decrypted_text = ""
    for char in text.upper():
        if char in reverse_mapping:
            decrypted_text += reverse_mapping[char]
        else:
            decrypted_text += char  # Non-alphabetic characters remain unchanged
    return decrypted_text

def main():
    # Get the key for decryption from the user
    key = input("Enter the encryption key used for decryption: ").upper()
    # Get the text to decrypt
    text = input("Enter the text to decrypt: ")
    # Decrypt the text
    decrypted_text = decrypt(text, key)
    print(f"Decrypted Text: {decrypted_text}")

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

Output:

Enter the encryption key used for decryption: IMEBXUFNZAWOPLVCRTQKHGDSJY
Enter the text to decrypt: IMEBXUFNZAWOPLVCRTQKHGDSJY
Decrypted Text: ABCDEFGHIJKLMNOPQRSTUVWXYZ

Problem No: 03
Problem Name: Write a program to implement encryption and decryption using Brute force attack cipher.

Theory:
A brute force attack is a cryptographic technique used to decipher encrypted data by exhaustively trying all possible keys until the correct one is found. In the context of breaking encryption schemes, the brute force approach involves systematically testing each key to determine which one decrypts the ciphertext into a readable plaintext. This method leverages the fact that some encryption algorithms, especially simpler or weaker ciphers, have a limited number of possible keys, making them susceptible to such attacks.

Application in Breaking Simple Ciphers: Brute force attacks are particularly effective against basic encryption techniques, such as the Caesar cipher or other substitution ciphers. In a Caesar cipher, for example, each letter in the plaintext is shifted by a fixed number of positions in the alphabet. The number of possible keys for the Caesar cipher is relatively small (only 25 possible shifts for English alphabets), allowing a brute force attack to try all shifts and decrypt the message. The attacker tests each possible key to identify the one that transforms the ciphertext into a meaningful and understandable message.

Strengths and Weaknesses: The brute force attack is a straightforward and universally applicable decryption method because it does not rely on knowing the specific encryption algorithm or key management scheme. Its simplicity is a major advantage when dealing with low-complexity ciphers that have a limited key space. However, brute force attacks are computationally expensive and become impractical as the number of possible keys increases. For modern encryption algorithms with large key sizes, the number of potential keys is astronomically high, making brute force attacks infeasible due to the enormous time and computational resources required.

Implications in Cryptography: The vulnerability of a cipher to brute force attacks highlights the importance of using encryption methods with sufficiently large key spaces. In modern cryptography, algorithms are designed to withstand brute force attempts by employing large key sizes (e.g., 128-bit or 256-bit keys in advanced encryption algorithms). This ensures that the number of potential keys is so vast that a brute force attack would be computationally prohibitive.

Educational and Practical Value: Brute force attacks provide a foundational understanding of the limitations of certain cryptographic methods and the need for more complex encryption techniques. They also serve as a practical exercise for students learning about cryptographic security, illustrating how simple encryption schemes can be broken and why stronger algorithms are necessary for protecting sensitive data in real-world applications.

Python Source Code:

Encryption

```
def encrypt(text, shift):
    encrypted_text = ""
    for char in text:
        if char.isalpha():
            shift_base = 65 if char.isupper() else 97
            encrypted_char = chr((ord(char) - shift_base + shift) % 26 + shift_base)
            encrypted_text += encrypted_char
        else:
            encrypted_text += char  # Non-alphabetic characters remain unchanged
    return encrypted_text

def main():
    text = input("Enter the text to encrypt: ")
    shift = int(input("Enter the shift value: "))
    encrypted_text = encrypt(text, shift)
    print(f"Encrypted Text: {encrypted_text}")

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

Output:

Enter the text to encrypt: abcdefgh
Enter the shift value: 1
Encrypted Text: bcdefghi

Decryption

```
def decrypt_with_shift(text, shift):
    decrypted_text = ""
    for char in text:
        if char.isalpha():
            shift_base = 65 if char.isupper() else 97
            decrypted_char = chr((ord(char) - shift_base - shift) % 26 + shift_base)
            decrypted_text += decrypted_char
        else:
            decrypted_text += char  # Non-alphabetic characters remain unchanged
    return decrypted_text

def brute_force_decrypt(text):
    print("Trying all possible shifts (0-25):")
    for shift in range(26):
        possible_plaintext = decrypt_with_shift(text, shift)
```

```
        print(f"Shift {shift}: {possible_plaintext}")

def main():
    encrypted_text = input("Enter the encrypted text: ")
    brute_force_decrypt(encrypted_text)

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

Output:

Enter the encrypted text: bcdefghi
Trying all possible shifts (0-25):
Shift 0: bcdefghi
Shift 1: abcdefgh
Shift 2: zabcdefg
Shift 3: yzabcdef
Shift 4: xyzabcde
Shift 5: wxyzabcd
Shift 6: vwxyzabc
Shift 7: uvwxyzab
Shift 8: tuvwxyza
Shift 9: stuvwxyz
Shift 10: rstuvwxy
Shift 11: qrstuvwx
Shift 12: pqrstuvw
Shift 13: opqrstuv
Shift 14: nopqrstu
Shift 15: mnopqrst
Shift 16: lmnopqrs
Shift 17: klmnopqr
Shift 18: jklmnopq
Shift 19: ijklmnop
Shift 20: hijklmno
Shift 21: ghijklmn
Shift 22: fghijklm
Shift 23: efghijkl
Shift 24: defghijk
Shift 25: cdefghij

Problem No: 04

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

Theory:

The Hill cipher is a polygraphic substitution cipher, which encrypts blocks of text using linear algebra. It was invented by Lester S. Hill in 1929 and is based on matrix multiplication. Unlike simpler ciphers that operate on individual characters, the Hill cipher encrypts multiple letters simultaneously, treating them as a vector and performing a series of linear transformations. This makes it a more secure and complex method of encryption compared to mono-alphabetic or simple substitution ciphers.

Encryption Mechanism: The encryption process of the Hill cipher involves dividing the plaintext into blocks of a specified size (usually 2 or 3 characters), which corresponds to the dimensions of the encryption matrix (key matrix). Each block of plaintext is represented as a vector, and the key matrix, which must be invertible in modular arithmetic, is used to transform this vector. The resulting vector is then converted back into characters to form the ciphertext.

Mathematically, the encryption formula can be expressed as:

$$\text{CipherText}, \quad C = KP \bmod 26$$

where

- K *is the key matrix.*
- P *is the plaintext*
- C *is the ciphertext vector.*
- $\bmod 26$ *ensures the result fits within the range of the alphabet.*

Decryption Mechanism: Decryption requires the inverse of the key matrix to transform the ciphertext back into the plaintext. The inverse matrix must be calculated under modular arithmetic (mod 26). Once the inverse key matrix is found, the ciphertext is converted back to plaintext using the formula:

$$\text{Plaintext}, \quad P = K^{-1}C \bmod 26$$

where

- K^{-1} *is the inverse of the key matrix.*
- P *is the plaintext*
- C *is the ciphertext vector.*
- *The modular operation ensures the decryption stays within the bounds of the alphabet.*

Conditions and Limitations: For the Hill cipher to work, the key matrix must be invertible in the given modular arithmetic. This requires that the determinant of the key matrix is non-zero and coprime with the modulus (usually 26 for the English alphabet). If these conditions are not met, the matrix does not have an inverse, making decryption impossible.

Advantages and Disadvantages: The Hill cipher's ability to encrypt multiple characters simultaneously offers stronger security than simple ciphers. It is resistant to frequency analysis since each character's encryption depends on multiple other characters. However, it is vulnerable to known-plaintext attacks, where if an attacker knows a sufficient amount of

plaintext-ciphertext pairs, they can solve for the key matrix.

Practical and Historical Importance: Historically, the Hill cipher was one of the early encryption methods to use mathematical concepts such as matrices and linear algebra, paving the way for modern cryptography. While it is not used in contemporary encryption systems due to its susceptibility to certain attacks, it provides educational value for learning about linear transformations, matrix operations, and modular arithmetic in the context of cryptography.

For example:

Given that , Plaintext = "sh"

Key = "hill"

Let us assign a numerical equivalent to each letter:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
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

Encryption:

$$\text{Plaintext, } P = \begin{bmatrix} s \\ h \end{bmatrix} = \begin{bmatrix} 18 \\ 7 \end{bmatrix}$$

$$\text{Key, } \quad K = \begin{bmatrix} h & l \\ i & l \end{bmatrix} = \begin{bmatrix} 7 & 11 \\ 8 & 11 \end{bmatrix}$$

We know that Cipher text , $C = KP \ mod\ 26$

$$= \begin{bmatrix} 7 & 11 \\ 8 & 11 \end{bmatrix} \times \begin{bmatrix} 18 \\ 7 \end{bmatrix} \ mod\ 26$$

$$= \begin{bmatrix} 203 \\ 221 \end{bmatrix} \ mod\ 26$$

$$= \begin{bmatrix} 21 \\ 13 \end{bmatrix} = \begin{bmatrix} v \\ n \end{bmatrix}$$

Decryption:

Plaintext , $P = K^{-1}C \ mod\ 26$

$$K^{-1} = \frac{1}{\det|K|} \ Adj\ K \ mod\ 26$$

$$|K| = \begin{bmatrix} 7 & 11 \\ 8 & 11 \end{bmatrix} \ mod\ 26$$

$$|K| = -11 \ \ mod\ 26$$

$$|K| = 15$$

$$K^{-1} = \frac{1}{15}\begin{bmatrix}11 & 15\\ 18 & 7\end{bmatrix} mod\ 26 \qquad \text{where,} \quad Adj\ K = \begin{bmatrix}11 & 15\\ 18 & 7\end{bmatrix}$$

$$K^{-1} = 7\begin{bmatrix}11 & 15\\ 18 & 7\end{bmatrix} mod\ 26$$

$$K^{-1} = \begin{bmatrix}77 & 105\\ 126 & 49\end{bmatrix} mod\ 26$$

$$K^{-1} = \begin{bmatrix}25 & 11\\ 22 & 23\end{bmatrix} mod\ 26$$

Plaintext , $P = K^{-1}C\ mod\ 26$

$$P = \begin{bmatrix}25 & 11\\ 22 & 23\end{bmatrix} \times \begin{bmatrix}21\\ 13\end{bmatrix} mod\ 26$$

$$P = \begin{bmatrix}668\\ 761\end{bmatrix} mod\ 26$$

$$P = \begin{bmatrix}18\\ 7\end{bmatrix} = \begin{bmatrix}s\\ h\end{bmatrix}$$

Python Source Code:

Encryption

```
import numpy as np
import string

# Helper function to process text
def process_text(text, size):
    text = text.lower().replace(" ", "")  # Remove spaces and convert to lowercase
    while len(text) % size != 0:  # Pad with 'x' if necessary
        text += 'x'
    return text

# Encryption function for Hill Cipher
def hill_encrypt(plain_text, key_matrix):
    n = key_matrix.shape[0]
    plain_text = process_text(plain_text, n)
    plain_vector = [ord(char) - ord('a') for char in plain_text]

    cipher_text = ''
    for i in range(0, len(plain_vector), n):
        chunk = np.array(plain_vector[i:i + n])
        encrypted_chunk = np.dot(key_matrix, chunk) % 26
        cipher_text += ''.join(chr(int(num) + ord('a')) for num in encrypted_chunk)

    return cipher_text
```

```
# Take input from the user
plain_text = input("Enter the plaintext to encrypt: ")
key_size = int(input("Enter the key matrix size (2 for 2x2 or 3 for 3x3): "))
key_matrix = []

print("Enter the key matrix row by row:")
for i in range(key_size):
    row = list(map(int, input(f"Row {i+1}: ").split()))
    key_matrix.append(row)

key_matrix = np.array(key_matrix)

# Encrypt the plaintext
cipher_text = hill_encrypt(plain_text, key_matrix)
print("Encrypted text:", cipher_text)
```

Output:

Enter the plaintext to encrypt: sh
Enter the key matrix size (2 for 2x2 or 3 for 3x3): 2
Enter the key matrix row by row:
Row 1: 7 8
Row 2: 11 11
Encrypted text: ap

Decryption

```
import numpy as np
from sympy import Matrix
import string

# Decryption function for Hill Cipher
def hill_decrypt(cipher_text, key_matrix):
    n = key_matrix.shape[0]
    cipher_vector = [ord(char) - ord('a') for char in cipher_text]

    try:
        # Compute the modular inverse of key_matrix
        key_inv = Matrix(key_matrix).inv_mod(26)
        key_inv = np.array(key_inv).astype(int)
    except ValueError:
        return "Decryption key matrix is not invertible modulo 26."

    decrypted_text = ''
    for i in range(0, len(cipher_vector), n):
```

```
        chunk = np.array(cipher_vector[i:i + n])
        decrypted_chunk = np.dot(key_inv, chunk) % 26
        decrypted_text += ''.join(chr(int(num) + ord('a')) for num in decrypted_chunk)

    return decrypted_text

# Take input from the user
cipher_text = input("Enter the ciphertext to decrypt: ")
key_size = int(input("Enter the key matrix size (2 for 2x2 or 3 for 3x3): "))
key_matrix = []

print("Enter the key matrix row by row:")
for i in range(key_size):
    row = list(map(int, input(f"Row {i+1}: ").split()))
    key_matrix.append(row)

key_matrix = np.array(key_matrix)

# Decrypt the ciphertext
decrypted_text = hill_decrypt(cipher_text, key_matrix)
print("Decrypted text:", decrypted_text)
```

Output:

Enter the ciphertext to decrypt: ap
Enter the key matrix size (2 for 2x2 or 3 for 3x3): 2
Enter the key matrix row by row:
Row 1: 7 8
Row 2: 11 11
Decrypted text: sh

Problem No: 05
Problem Name: Write a program to implement encryption using Playfair cipher.

Theory:
The Playfair cipher is a classical encryption technique that encrypts messages using pairs of letters, or digraphs, rather than single characters. Developed by Charles Wheatstone in 1854 and popularized by Lord Playfair, the Playfair cipher improves on simple substitution ciphers by encrypting two letters at a time. It uses a 5x5 matrix filled with a keyword to map the alphabet and encrypt the plaintext, offering better security than monoalphabetic ciphers, as letter frequencies are more dispersed. The Playfair cipher also disrupts letter patterns, making it more resistant to frequency analysis.

Procedure for Encryption Using Playfair Cipher

1. **Key Matrix Construction:**
    - Select a keyword for generating the encryption matrix. This keyword forms the basis for the 5x5 grid.
    - Remove any duplicate letters from the keyword, and then fill in the grid with the remaining letters of the alphabet in the order they appear in the keyword.
    - Usually, the letters 'I' and 'J' are treated as interchangeable, combining them to fit all 26 letters into the 25-square matrix.
2. **Divide Plaintext into Pairs (Digraphs):**
    - Separate the plaintext into digraphs (pairs of letters). If a pair consists of the same letter (like "LL"), insert a filler character (commonly 'X') between them to make the digraph unique. For an odd-length message, add a filler character at the end.
    - The text should be preprocessed by converting all characters to uppercase and replacing 'J' with 'I' (if using this convention).
3. **Encryption Rules for Each Digraph:**
    - For each digraph, find the positions of the two letters in the matrix.
        - ✓ If both letters are in the same row, shift each one to the right by one column, wrapping around to the start if necessary.
        - ✓ If both letters are in the same column, shift each one down by one row, wrapping around if needed.
        - ✓ If the letters form a rectangle, swap the columns of the two letters, so that each letter moves to the corner on the opposite side of the rectangle, maintaining the original rows.
4. **Forming the Ciphertext:**
    - Combine the encrypted pairs sequentially to produce the ciphertext.
    - Maintain the digraph structure throughout, but do not include any fillers in the final encrypted output if they were used only for alignment.

Advantages and Drawbacks: The Playfair cipher adds a layer of complexity compared to

basic substitution ciphers by encrypting digraphs instead of individual letters, reducing the patterns available for cryptanalysis. However, it can still be susceptible to certain known-plaintext attacks, especially when a significant portion of the encrypted and original texts are known.

Python Source Code:

```
import numpy as np

# Helper function to generate the 5x5 key matrix
def generate_key_matrix(keyword):
    keyword = keyword.replace("j", "i").lower()  # Replace 'j' with 'i'
    matrix = []
    used_chars = set()

    # Add unique letters from the keyword to the matrix
    for char in keyword:
        if char not in used_chars and char.isalpha():
            used_chars.add(char)
            matrix.append(char)

    # Add remaining letters of the alphabet
    for char in "abcdefghiklmnopqrstuvwxyz":  # 'j' is merged with 'i'
        if char not in used_chars:
            used_chars.add(char)
            matrix.append(char)

    # Convert the list into a 5x5 matrix
    key_matrix = np.array(matrix).reshape(5, 5)
    return key_matrix

# Helper function to find the position of a letter in the key matrix
def find_position(char, key_matrix):
    row, col = np.where(key_matrix == char)
    return row[0], col[0]

# Function to preprocess text into pairs for encryption
def process_text(plain_text):
    plain_text = plain_text.lower().replace(" ", "").replace("j", "i")
    processed_text = ""
    i = 0

    # Form pairs with padding if necessary
    while i < len(plain_text):
```

```
        processed_text += plain_text[i]
        if i + 1 < len(plain_text) and plain_text[i] == plain_text[i + 1]:
            processed_text += 'x'  # Insert 'x' between duplicate letters
        elif i + 1 < len(plain_text):
            processed_text += plain_text[i + 1]
            i += 1
        else:
            processed_text += 'x'  # Padding if only one letter is left at the end
        i += 1

    return processed_text

# Playfair encryption function
def playfair_encrypt(plain_text, key_matrix):
    processed_text = process_text(plain_text)
    cipher_text = ""

    for i in range(0, len(processed_text), 2):
        char1, char2 = processed_text[i], processed_text[i + 1]
        row1, col1 = find_position(char1, key_matrix)
        row2, col2 = find_position(char2, key_matrix)

        # Apply Playfair cipher rules
        if row1 == row2:  # Same row
            cipher_text += key_matrix[row1][(col1 + 1) % 5]
            cipher_text += key_matrix[row2][(col2 + 1) % 5]
        elif col1 == col2:  # Same column
            cipher_text += key_matrix[(row1 + 1) % 5][col1]
            cipher_text += key_matrix[(row2 + 1) % 5][col2]
        else:  # Rectangle rule
            cipher_text += key_matrix[row1][col2]
            cipher_text += key_matrix[row2][col1]

    return cipher_text.upper()

# Main code to run Playfair encryption
keyword = input("Enter the keyword for the key matrix: ")
plain_text = input("Enter the plaintext to encrypt: ")

# Generate key matrix and encrypt
key_matrix = generate_key_matrix(keyword)
cipher_text = playfair_encrypt(plain_text, key_matrix)
```

```
print("Key Matrix:")
print(key_matrix)
print("Encrypted Text:", cipher_text)
```

Output:

```
Enter the keyword for the key matrix: computer
Enter the plaintext to encrypt: communicate
Key Matrix:
[['c' 'o' 'm' 'p' 'u']
 ['t' 'e' 'r' 'a' 'b']
 ['d' 'f' 'g' 'h' 'i']
 ['k' 'l' 'n' 'q' 's']
 ['v' 'w' 'x' 'y' 'z']]
Encrypted Text: OMRMPCSGPTER
```

Problem No: 06
Problem Name: Write a program to implement decryption using Playfair cipher.

Theory:
The Playfair cipher is a classical encryption technique that secures messages by encrypting pairs of letters (digraphs) using a 5x5 matrix derived from a keyword. While it offers better security than simple substitution ciphers by obscuring letter frequencies and making pattern detection more difficult, decrypting a Playfair-encrypted message involves reversing the encryption steps. The decryption process reconstructs the original plaintext by applying inverse transformations to the encrypted digraphs, using the same key matrix that was used for encryption.
To decrypt, the Playfair cipher follows specific rules that involve locating the positions of encrypted digraphs within the matrix and shifting them in a direction opposite to that used during encryption. This reverses the transformations applied during encryption and retrieves the original message in plaintext form.

Procedure for Decryption Using Playfair Cipher

1. **Reconstruct the Key Matrix:**
    - Use the same keyword that was used during encryption to build the 5x5 matrix.
    - Eliminate duplicate letters from the keyword, then fill the grid with the remaining letters of the alphabet.
    - If following the convention of merging 'I' and 'J,' treat them as a single letter to ensure all 26 letters fit into the 25 cells.

2. **Split the Ciphertext into Digraphs:**
    - Divide the ciphertext into pairs of two letters (digraphs). These pairs will be decrypted one at a time to reconstruct the original message.
    - If the original plaintext had padding (e.g., added 'X' for repeated letters), note this to remove any unnecessary padding after decryption.

3. **Decrypt Each Digraph Using the Key Matrix:**
    - Locate the two letters of each digraph in the 5x5 matrix:
        - ✓ If both letters are in the same row, shift each letter one position to the left. If a letter is at the start of the row, wrap around to the last position.
        - ✓ If both letters are in the same column, shift each letter one position up. If a letter is at the top of the column, wrap around to the bottom.
        - ✓ If the letters form the opposite corners of a rectangle, replace them with the letters on the same row but at the other corners of the rectangle, retaining their original columns.

4. **Combine the Decrypted Pairs to Form the Plaintext:**
    - Concatenate the decrypted digraphs sequentially to reconstruct the original message.

- If padding was used during encryption (e.g., an 'X' added to separate repeated letters), remove it from the final plaintext, ensuring the output matches the original message as closely as possible.

Advantages and Limitations of Decryption with Playfair Cipher: Decrypting a Playfair-encrypted message requires knowledge of the keyword, which is used to generate the key matrix. While it can make frequency analysis more difficult by encrypting digraphs, the cipher can still be susceptible to cryptanalysis if the keyword is weak or if a significant amount of ciphertext is available for analysis. Additionally, reversing the transformations exactly as applied during encryption is crucial for accurate decryption.

Python Source Code:

```
import numpy as np

# Helper function to generate the 5x5 key matrix
def generate_key_matrix(keyword):
    keyword = keyword.replace("j", "i").lower()  # Replace 'j' with 'i'
    matrix = []
    used_chars = set()

    # Add unique letters from the keyword to the matrix
    for char in keyword:
        if char not in used_chars and char.isalpha():
            used_chars.add(char)
            matrix.append(char)

    # Add remaining letters of the alphabet
    for char in "abcdefghiklmnopqrstuvwxyz":  # 'j' is merged with 'i'
        if char not in used_chars:
            used_chars.add(char)
            matrix.append(char)

    # Convert the list into a 5x5 matrix
    key_matrix = np.array(matrix).reshape(5, 5)
    return key_matrix

# Helper function to find the position of a letter in the key matrix
def find_position(char, key_matrix):
    row, col = np.where(key_matrix == char)
    return row[0], col[0]

# Playfair decryption function
def playfair_decrypt(cipher_text, key_matrix):
```

```
    cipher_text = cipher_text.lower().replace(" ", "")
    decrypted_text = ""
    for i in range(0, len(cipher_text), 2):
        char1, char2 = cipher_text[i], cipher_text[i + 1]
        row1, col1 = find_position(char1, key_matrix)
        row2, col2 = find_position(char2, key_matrix)

        # Apply Playfair cipher decryption rules
        if row1 == row2:  # Same row
            decrypted_text += key_matrix[row1][(col1 - 1) % 5]
            decrypted_text += key_matrix[row2][(col2 - 1) % 5]
        elif col1 == col2:  # Same column
            decrypted_text += key_matrix[(row1 - 1) % 5][col1]
            decrypted_text += key_matrix[(row2 - 1) % 5][col2]
        else:  # Rectangle rule
            decrypted_text += key_matrix[row1][col2]
            decrypted_text += key_matrix[row2][col1]

    # Remove any padding 'x' that might have been added
    decrypted_text = decrypted_text.replace("x", "")
    return decrypted_text

# Main code to run Playfair decryption
keyword = input("Enter the keyword for the key matrix: ")
cipher_text = input("Enter the ciphertext to decrypt: ")

# Generate key matrix and decrypt
key_matrix = generate_key_matrix(keyword)
decrypted_text = playfair_decrypt(cipher_text, key_matrix)

print("Key Matrix:")
print(key_matrix)
print("Decrypted Text:", decrypted_text)
```

Output:

```
Enter the keyword for the key matrix: computer
Enter the ciphertext to decrypt: OMRMPCSGPTER
Key Matrix:
[['c' 'o' 'm' 'p' 'u']
 ['t' 'e' 'r' 'a' 'b']
 ['d' 'f' 'g' 'h' 'i']
 ['k' 'l' 'n' 'q' 's']
 ['v' 'w' 'x' 'y' 'z']]
Decrypted Text: communicate
```

Problem No: 07

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

Theory:

A poly-alphabetic cipher is a type of substitution cipher that uses multiple alphabets to encrypt the plaintext. Unlike mono-alphabetic ciphers, where each letter of the plaintext is replaced with the same corresponding letter of the cipher alphabet, poly-alphabetic ciphers switch between multiple cipher alphabets throughout the encryption process. This method increases the complexity of the encryption, making it more resistant to frequency analysis attacks because each letter in the plaintext could correspond to different letters in the ciphertext based on the shifting alphabet.

One of the most well-known poly-alphabetic ciphers is the Vigenère cipher, which uses a keyword to determine the shifting pattern for the letters in the plaintext. The positions of the letters in the keyword control how much each letter in the plaintext shifts, creating multiple substitution rules.

Procedure for Encryption Using Poly-Alphabetic Cipher

1. **Choose a Keyword:**
    - Select a keyword that will determine the encryption pattern. This keyword should be repeated or extended to match the length of the plaintext, ensuring that every letter in the plaintext has a corresponding letter in the keyword.

2. **Prepare the Plaintext:**
    - Convert all letters in the plaintext to uppercase and remove any non-alphabetic characters, such as punctuation or spaces, to ensure a uniform input.

3. **Encrypt Each Letter Using the Vigenère Table:**
    - For each letter in the plaintext, use the corresponding letter in the keyword to determine the shift.
        - ✓ The shift value is calculated by finding the position of the keyword letter in the alphabet (A=0, B=1, ..., Z=25).
        - ✓ Shift the plaintext letter forward by this number of positions in the alphabet.
        - ✓ If the shift causes the letter to exceed 'Z,' wrap around to the beginning of the alphabet.
    - Record the result as the encrypted letter.

4. **Form the Ciphertext:**
    - Combine all the encrypted letters sequentially to create the final ciphertext.
    - The resulting ciphertext will have the same length as the original plaintext but will appear as a scrambled version due to the multiple substitution rules applied during encryption.

Advantages and Disadvantages: The poly-alphabetic cipher significantly improves security compared to mono-alphabetic ciphers by disguising letter frequency patterns. However, it is still vulnerable to cryptanalysis techniques like Kasiski examination and frequency analysis on repeated segments if the keyword is short or if the ciphertext is long enough to reveal patterns.

Python Source Code:

```
def generate_keystream(plain_text, keyword):
    # Repeat the keyword to match the length of the plaintext
    keystream = keyword * (len(plain_text) // len(keyword)) + keyword[:len(plain_text) %
len(keyword)]
    return keystream

def poly_alphabetic_encrypt(plain_text, keyword):
    plain_text = plain_text.lower().replace(" ", "")  # Process plain text
    keyword = keyword.lower()
    keystream = generate_keystream(plain_text, keyword)

    cipher_text = ""
    for pt_char, ks_char in zip(plain_text, keystream):
        # Shift each character by the alphabetical index of the corresponding keystream
character
        pt_index = ord(pt_char) - ord('a')
        ks_index = ord(ks_char) - ord('a')
        encrypted_char = chr((pt_index + ks_index) % 26 + ord('a'))
        cipher_text += encrypted_char

    return cipher_text.upper()

# Main code to run Poly-Alphabetic (Vigenère) Cipher encryption
plain_text = input("Enter the plaintext to encrypt: ")
keyword = input("Enter the keyword for encryption: ")

# Encrypt the plaintext
cipher_text = poly_alphabetic_encrypt(plain_text, keyword)

print("Encrypted Text:", cipher_text)
```

Output:

Enter the plaintext to encrypt: geeksforgeeks
Enter the keyword for encryption: ayush
Encrypted Text: GCYCZFMLYLEIM

Problem No: 08

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

Theory:

The poly-alphabetic cipher is an advanced encryption technique that utilizes multiple substitution alphabets to enhance security during the encryption process. This method is significantly more complex than mono-alphabetic ciphers, where a single substitution rule is applied uniformly. In a poly-alphabetic cipher, the substitutions vary based on the position of the letters in the plaintext and a predetermined keyword. The most recognized poly-alphabetic cipher is the Vigenère cipher, which employs a repeating keyword to dictate the shifts for each letter in the plaintext, providing greater protection against frequency analysis.

Decryption in a poly-alphabetic cipher involves reversing the encryption process by utilizing the same keyword used during encryption. The key aspect of decryption is to shift the letters of the ciphertext backward according to the corresponding letters of the keyword, effectively retrieving the original plaintext. This approach ensures that even if the ciphertext is intercepted, without knowledge of the keyword, deciphering the message becomes exceedingly challenging.

Procedure for Decryption Using Poly-Alphabetic Cipher

1. **Utilize the Keyword:**
    - To begin the decryption process, the same keyword employed during encryption is required. This keyword will dictate the shifts used to revert the ciphertext back to its original form.
2. **Prepare the Ciphertext:**
    - Convert all letters in the ciphertext to uppercase and ignore any non-alphabetic characters (such as spaces and punctuation) to maintain consistency in processing the text.
3. **Extend the Keyword to Match the Ciphertext:**
    - Repeat the keyword so that its length matches the length of the ciphertext. This ensures that each letter in the ciphertext has a corresponding letter in the keyword for the decryption process.
4. **Decrypt Each Character:**
    - For each letter in the ciphertext, find its corresponding letter in the repeated keyword:
        - ✓ Calculate the shift value based on the position of the keyword letter in the alphabet (A=0, B=1, ..., Z=25).
        - ✓ Shift the ciphertext letter backward by this amount:
            - If the letter goes before 'A,' wrap around to 'Z' to stay within the bounds of the alphabet.
    - Append the decrypted letter to the final plaintext string.
5. **Combine Decrypted Letters:**
    - Once all characters have been processed, combine the decrypted letters to form the final plaintext message. If non-alphabetic characters were present in the

ciphertext, ensure to reintegrate them into the final output at their respective positions.

Advantages and Limitations: The poly-alphabetic cipher greatly enhances security compared to mono-alphabetic systems, as it complicates frequency analysis and disguises patterns. However, it is still vulnerable to certain attacks if the keyword is short or poorly chosen, allowing attackers to analyze repeated sequences. Additionally, if the ciphertext is lengthy and the keyword is short, patterns may emerge that can be exploited.

Python Source Code:

```
def generate_keystream(cipher_text, keyword):
    # Repeat the keyword to match the length of the ciphertext
    keystream = keyword * (len(cipher_text) // len(keyword)) + keyword[:len(cipher_text)
% len(keyword)]
    return keystream

def poly_alphabetic_decrypt(cipher_text, keyword):
    cipher_text = cipher_text.lower().replace(" ", "")  # Process ciphertext
    keyword = keyword.lower()
    keystream = generate_keystream(cipher_text, keyword)

    decrypted_text = ""
    for ct_char, ks_char in zip(cipher_text, keystream):
        # Shift each character backward by the alphabetical index of the corresponding
keystream character
        ct_index = ord(ct_char) - ord('a')
        ks_index = ord(ks_char) - ord('a')
        decrypted_char = chr((ct_index - ks_index + 26) % 26 + ord('a'))
        decrypted_text += decrypted_char

    return decrypted_text

# Main code to run Poly-Alphabetic (Vigenère) Cipher decryption
cipher_text = input("Enter the ciphertext to decrypt: ")
keyword = input("Enter the keyword used for encryption: ")

# Decrypt the ciphertext
decrypted_text = poly_alphabetic_decrypt(cipher_text, keyword)

print("Decrypted Text:", decrypted_text)
```

Output:

Enter the ciphertext to decrypt: gcyczfmlyleim
Enter the keyword used for encryption: ayush
Decrypted Text: geeksforgeeks

Problem No: 09
Problem Name: Write a program to implement encryption using Vernam cipher.

Theory:
The Vernam cipher, commonly referred to as the one-time pad, is a symmetric encryption technique celebrated for its outstanding security attributes. It operates by combining each letter of the plaintext with a corresponding letter from a randomly generated key that is equal in length to the plaintext. This encryption process employs the XOR (exclusive OR) operation, resulting in a ciphertext that theoretically provides perfect secrecy, provided certain conditions are met. A fundamental characteristic of the Vernam cipher is that the key must be entirely random, match the length of the plaintext, and be utilized solely a single time for each encryption process. This ensures that each instance of encryption generates a unique ciphertext, rendering it impossible for an attacker to infer any information about the plaintext without access to the specific key used. Theoretically, if the key is kept confidential and never reused, the cipher is unbreakable, as every potential plaintext could correspond to a valid ciphertext.
Despite its strong theoretical foundation, the practical implementation of the Vernam cipher is limited by challenges related to key distribution and management. The necessity for a unique key of equal length to the plaintext complicates its use in various real-world scenarios, especially when handling substantial amounts of data.

Procedure for Encryption Using Vernam Cipher

1. **Key Preparation:**
    - Create a key comprised of random characters that is equal in length to the plaintext, guaranteeing that every character in the plaintext aligns with a specific character in the key.
2. **Standardizing the Plaintext:**
    - Convert the plaintext to a consistent format, typically by transforming all characters to uppercase. Decide on how to handle non-alphabetic characters (e.g., spaces and punctuation)—whether to include them unchanged or ignore them.
3. **Encrypting the Plaintext:**
    - Loop through each character of the plaintext and its corresponding character in the key:
        - ✓ Convert the plaintext and key characters into numerical representations based on their positions in the alphabet (A=0, B=1, ..., Z=25).
        - ✓ Determine the encrypted value by performing the XOR operation between the plaintext and key characters, followed by applying the modulo 26 operation to ensure the outcome is constrained within the range of alphabetic characters.
        - ✓ Convert the resulting numerical value back to a letter.
4. **Constructing the Ciphertext:**
    - Assemble all the encrypted letters to form the final ciphertext. If non-alphabetic characters were present, they should be included in the output as they appeared

in the plaintext.

5. **Displaying the Encrypted Output:**
    - The resultant ciphertext represents the securely encrypted message. To decrypt the message, the identical key must be utilized in the same manner, allowing the original plaintext to be retrieved.

Advantages and Limitations

The main advantage of the Vernam cipher is its theoretical perfect secrecy when implemented correctly. However, its practical limitations include the complexities of key management, as each key must remain confidential and be used exclusively for one instance of encryption. Reusing a key or allowing it to be compromised would severely weaken the cipher's security.

Python Source Code:

```
def vernam_encrypt(plain_text, key):
    if len(plain_text) != len(key):
        raise ValueError("The key must be the same length as the plaintext for the Vernam cipher.")

    # Convert text to uppercase for simplicity and remove spaces
    plain_text = plain_text.upper().replace(" ", "")
    key = key.upper().replace(" ", "")

    cipher_text = ""
    for pt_char, k_char in zip(plain_text, key):
        # Perform XOR between characters and wrap result to alphabet range
        pt_index = ord(pt_char) - ord('A')
        k_index = ord(k_char) - ord('A')
        encrypted_index = (pt_index ^ k_index) % 26
        encrypted_char = chr(encrypted_index + ord('A'))
        cipher_text += encrypted_char
    return cipher_text

# Main code to run Vernam cipher encryption
plain_text = input("Enter the plaintext to encrypt: ")
key = input("Enter the key (must be the same length as the plaintext): ")

# Encrypt the plaintext
cipher_text = vernam_encrypt(plain_text, key)

print("Encrypted Text:", cipher_text)
```

Output:

Enter the plaintext to encrypt: QAK
Enter the key (must be the same length as the plaintext): SON
Encrypted Text: COH

Problem No: 10

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

Theory:

The Vernam cipher, also known as the one-time pad, is a symmetric encryption technique distinguished by its exceptional security features. Its decryption process is fundamentally linked to the encryption method, relying on the same key that was utilized during the encryption phase. The uniqueness of the Vernam cipher lies in the requirement that the key must be completely random, match the length of the plaintext, and be used only once. This ensures that every encrypted message is distinct, providing a level of security that is theoretically unbreakable when the key is kept confidential.

The process of decrypting in the Vernam cipher utilizes the XOR (exclusive OR) operation, mirroring the method used during encryption. By applying the same key to the ciphertext, the original plaintext can be accurately retrieved. This duality of the XOR operation means that the processes of encrypting and decrypting are essentially the same. If the key is correct and used in the same manner, the decryption yields the original message, thus preserving the integrity of the data. Despite its robust theoretical framework, the practical application of the Vernam cipher is often hindered by challenges associated with key management and distribution. The necessity for a unique key of equal length to the plaintext complicates its real-world usability, particularly in scenarios involving large datasets or frequent message exchanges.

Procedure for Decryption Using Vernam Cipher

1. **Key Retrieval:**
    - Retrieve the identical key that was employed for encryption. It is crucial that this key is kept secret and has not been reused.
2. **Prepare the Ciphertext:**
    - Format the ciphertext to ensure uniformity. Convert all characters to uppercase for consistency and decide how to handle non-alphabetic characters in the ciphertext.
3. **Decrypting the Ciphertext:**
    - Iterate through each character of the ciphertext and its corresponding character in the key:
        - ✓ Convert the ciphertext and key characters into numerical representations based on their positions in the alphabet (A=0, B=1, ..., Z=25).
        - ✓ Apply the XOR operation to these numerical values:
            - Compute the result as (C XOR K) % 26, where C represents the ciphertext character and K is the key character. This operation will yield the corresponding plaintext character.
        - ✓ Convert the resulting numerical value back to a letter.
4. **Constructing the Original Plaintext:**
    - Combine all the decrypted characters to form the final plaintext. If non-alphabetic characters were present in the ciphertext, they should be preserved in their original positions.

5. **Output the Decrypted Message:**
    - The final output is the original plaintext that has been successfully retrieved from the ciphertext. This demonstrates the effectiveness of the Vernam cipher in ensuring secure communication.

Advantages and Limitations

The primary advantage of the Vernam cipher lies in its theoretical perfect secrecy, provided the key is genuinely random and used only once. However, practical limitations arise from the complexities of key management, as maintaining the confidentiality and uniqueness of the key can be challenging in real-world applications.

Python Source Code:

```
def vernam_decrypt(cipher_text, key):
    if len(cipher_text) != len(key):
        raise ValueError("The key must be the same length as the ciphertext for the Vernam cipher.")

    # Convert text to uppercase for simplicity and remove spaces
    cipher_text = cipher_text.upper().replace(" ", "")
    key = key.upper().replace(" ", "")

    decrypted_text = ""
    for ct_char, k_char in zip(cipher_text, key):
        # Perform XOR between characters and wrap result to alphabet range
        ct_index = ord(ct_char) - ord('A')
        k_index = ord(k_char) - ord('A')
        decrypted_index = (ct_index ^ k_index) % 26  # XOR operation
        decrypted_char = chr(decrypted_index + ord('A'))  # Map back to alphabet
        decrypted_text += decrypted_char
    return decrypted_text

# Main code to run Vernam cipher decryption
cipher_text = input("Enter the ciphertext to decrypt: ")
key = input("Enter the key (must be the same length as the ciphertext): ")

# Decrypt the ciphertext
decrypted_text = vernam_decrypt(cipher_text, key)

print("Decrypted Text:", decrypted_text)
```

Output:

```
Enter the ciphertext to decrypt: COH
Enter the key (must be the same length as the ciphertext): SON
Decrypted Text: QAK
```