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 **ALAMIN** Roll No: 200610 Reg. No: 1065375 Session: 2019-2020. 4th year 1st semester, Department of Information and Communication Engineering, Pabna University of Science and Technology.	Submitted to **Md. Anwar Hossain** Professor, Department of Information and Communication Engineering, Pabna University of Science and Technology.

Submission Date: 12-11-2024

……………………….

Signature

INDEX

SL	**Name of the Experiment**	**Page No.**
1.	Write a program to implement encryption and decryption using Caesar cipher.	
2.	Write a program to implement encryption and decryption using Mono-Alphabetic cipher.	
3.	Write a program to implement encryption and decryption using Brute force attack cipher.	
4.	Write a program to implement encryption and decryption using Hill cipher.	
5.	Write a program to implement encryption using Playfair cipher.	
6.	Write a program to implement decryption using Playfair cipher.	
7.	Write a program to implement encryption using Poly-Alphabetic cipher.	
8.	Write a program to implement decryption using Poly-Alphabetic cipher.	
9.	Write a program to implement encryption using Vernam cipher.	
10.	Write a program to implement decryption using Vernam cipher.	

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

i. To study and implement encryption using Caesar cipher.
ii. To study and implement decryption using Caesar cipher.

Theory:

The Caesar Cipher, also known as the shift cipher, is one of the simplest and most widely known encryption techniques. Named after Julius Caesar, who used it in his private correspondence, the Caesar cipher shifts each letter in a plaintext by a fixed number of positions down or up the alphabet. This type of cipher is a form of substitution cipher, where each character in the plaintext is replaced by a corresponding character in the ciphertext.

In Caesar Cipher encryption, each letter in the plaintext message is shifted by a fixed number of positions down the alphabet, known as the key. This means if the key is 3, for example, the letter "A" would become "D," "B" would become "E," and so on. This process creates a "ciphertext," which appears as a jumbled version of the original text and can only be understood by those who know the key. By applying this shift consistently to all characters in the plaintext, a hidden message is generated that conceals the original content, making it readable only to those who understand the encryption method and key used.

The decryption process of a Caesar Cipher reverses the encryption by shifting each letter in the ciphertext back by the same key. For example, if the key is 3 and the letter in the ciphertext is "D," it would shift back to "A" in the plaintext. This reverse shift, or "undoing" of the encryption, allows the original message to be recovered accurately. Since there are only a limited number of possible keys (typically 1 to 25 for English letters), Caesar Cipher decryption is straightforward and efficient, though it also makes the cipher less secure against attacks due to the ease with which a ciphertext can be brute-forced by trying each possible shift value.

Algorithm:

Encryption Algorithm

```
1. Input: Plaintext message (plaintext) and a shift key.
2. Initialize: An empty string ciphertext.
3. Loop through each character in plaintext:
   o If the character is a letter:
      ▪ Shift it by key positions forward in the alphabet.
      ▪ Wrap around if needed (e.g., from "Z" to "A").
      ▪ Use the formula encryption $E_n(x) = (X + K)\ mod\ 26$
```

Where X is the current position and K is key

- If it's not a letter, keep it the same.
- Add the result to ciphertext.

4. Output: ciphertext as the encrypted message.

Decryption Algorithm

1. Input: Ciphertext message (ciphertext) and a shift key.
2. Initialize: An empty string plaintext.
3. Loop through each character in ciphertext:
   - If the character is a letter:
     - Shift it by key positions backward in the alphabet.
     - Wrap around if needed (e.g., from "A" to "Z").
     - Use the formula decryption $D_n(x) = (X - K) \bmod 26$

       Where X is the current position and K is key
   - If it's not a letter, keep it the same.
   - Add the result to plaintext.
4. Output: plaintext as the decrypted message.

Example

Plaintext: ALAMIN encrypt using Caesar cipher where key=5

Solution:

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

For encryption formula is

$$E_n(x) = (X + K) \bmod 26$$

$$For\ A = 0 \qquad E_n(x) = (0 + 5) \bmod 26 = 5 \bmod 26 = 5 = F$$
$$For\ L = 11 \qquad E_n(x) = (11 + 5) \bmod 26 = 16 \bmod 26 = 16 = Q$$
$$For\ A = 0 \qquad E_n(x) = (0 + 5) \bmod 26 = 5 \bmod 26 = 5 = F$$
$$For\ M = 12 \qquad E_n(x) = (12 + 5) \bmod 26 = 17 \bmod 26 = 17 = R$$
$$For\ I = 8 \qquad E_n(x) = (8 + 5) \bmod 26 = 13 \bmod 26 = 13 = N$$
$$For\ N = 13 \qquad E_n(x) = (13 + 5) \bmod 26 = 18 \bmod 26 = 18 = S$$

Finally, ciphertext: FQFRNS

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

Again, for decryption formula is

$$D_n(x) = (X - K) \bmod 26$$

$$For\ F = 5 \qquad D_n(x) = (5 - 5)\ mod\ 26 = 0\ mod\ 26 = 0 = A$$
$$For\ Q = 16 \qquad D_n(x) = (16 - 5)\ mod\ 26 = 11\ mod\ 26 = 11 = L$$
$$For\ F = 5 \qquad D_n(x) = (5 - 5)\ mod\ 26 = 0\ mod\ 26 = 0 = A$$
$$For\ R = 17 \qquad D_n(x) = (17 - 5)\ mod\ 26 = 12\ mod\ 26 = 12 = M$$
$$For\ N = 13 \qquad D_n(x) = (13 - 5)\ mod\ 26 = 8\ mod\ 26 = 8 = I$$
$$For\ S = 18 \qquad D_n(x) = (18 - 5)\ mod\ 26 = 13\ mod\ 26 = 13 = N$$

Finally, plaintext: ALAMIN

Code (Python):
Code for encryption:
```
def caesar_cipher_encrypt(plaintext, key):
    ciphertext = ""

    for char in plaintext:
        if char.isupper():  # Check if the character is an uppercase letter
            # Shift character by key positions and wrap around with modulo 26
            shifted_char = chr((ord(char) - ord('A') + key) % 26 + ord('A'))
            ciphertext += shifted_char
        else:
            # If the character is not an uppercase letter, keep it the same
            ciphertext += char

    return ciphertext

# Example usage:
plaintext = "ALAMIN"
key = 5
ciphertext = caesar_cipher_encrypt(plaintext, key)
print(f"The plaintext:{plaintext}")
print("Encrypted message:", ciphertext)
```

For Decryption code:
```
def caesar_cipher_decrypt(ciphertext, key):
    plaintext = ""

    for char in ciphertext:
        if char.isupper():  # Check if the character is an uppercase letter
            # Shift character backwards by key positions
            shifted_char = chr((ord(char) - ord('A') - key) % 26 + ord('A'))
            plaintext += shifted_char
        else:
            # If the character is not an uppercase letter, keep it the same
            plaintext += char

    return plaintext

# Example usage:
cipher_text = "FQFRNS"
key = 5
plaintext = caesar_cipher_decrypt(cipher_text, key)
print(f"The Ciphertext: {cipher_text}")
print("The plaintext:", plaintext)
```

Output:

Encryption:

```
The plaintext: ALAMIN
Encrypted message: FQFRNS
```

Decryption:

```
The Ciphertext: FQFRNS
The plaintext: ALAMIN
```

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

i. To study and implement encryption using Mono-Alphabetic cipher.
ii. To study and implement decryption using Mono-Alphabetic cipher.

Theory:

The Mono-Alphabetic Cipher is a type of substitution cipher in which each letter in the plaintext is substituted with a corresponding letter in the ciphertext alphabet. The substitution follows a specific, predefined mapping of each letter to another letter, where each letter of the plaintext is replaced by one letter from the ciphertext alphabet. This mapping is usually a fixed key.

The encryption process involves replacing each character of the plaintext with its corresponding character from the cipher key. Since the cipher alphabet is static, anyone who knows the key can easily encrypt a message by following the established substitution rules. However, if the key is known, the cipher is vulnerable to frequency analysis, as some letters in the ciphertext will appear more often than others, revealing patterns in the language.

Decryption of a Mono-Alphabetic Cipher is simply the reverse of encryption. To decrypt the message, the recipient must possess the same cipher key that was used for encryption. Using this key, the ciphertext is mapped back to the original plaintext alphabet, converting the encrypted text into readable form. The decryption process is analogous to encryption but involves using the inverse of the cipher alphabet, where each letter from the ciphertext is substituted back to its corresponding letter from the plain alphabet. Although the cipher is relatively simple and easy to implement, it provides weak security due to the predictability of letter frequencies, making it susceptible to cryptanalysis techniques.

Algorithm:

Algorithm for Encryption:

1. Create a key: Create a mapping between the letters of the alphabet. For example, the letters A-Z could be substituted by another random permutation of the same letters (a key).
2. Substitute the characters: Replace each character in the plaintext with the corresponding character from the ciphertext alphabet.
3. Output the ciphertext.

Algorithm for Decryption:

1. Create the inverse key: Create the inverse of the encryption key, i.e., map each letter of the cipher alphabet back to the corresponding letter of the plain alphabet.
2. Substitute the characters: Replace each character in the ciphertext with the corresponding character from the plain alphabet.
3. Output the plaintext.

Example

Plaintext "A GOOD BOY ALAMIN"

Key: QWERTYUIOPASDFGHJKLZXCVBNM

Using the mono alphabetic algorithm encrypt and decrypt

Solution

Here the plaintext is **A GOOD BOY ALAMIN**

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

Form the table compare each letter and make the ciphertext.

Ciphertext: Q UGGR WGN QSQDOF

Again plaintext from the ciphertext this table reverse

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

Plaintext: **A GOOD BOY ALAMIN**

Code(python):

Code for encryption:

```
import string
def create_mapping(key):
   # Create a dictionary mapping each letter of the alphabet to a letter in the
provided key
   alphabet = string.ascii_uppercase
   return {alphabet[i]: key[i] for i in range(len(alphabet))}

def monoalphabetic_encrypt(plaintext, key_mapping):
   ciphertext = ""

   for char in plaintext:
      if char.isupper():  # Encrypt only uppercase letters
```

```
            ciphertext += key_mapping.get(char, char)  # Substitute using the key
mapping
        else:
            ciphertext += char  # Keep non-uppercase characters unchanged

    return ciphertext

# User-provided key (cipher alphabet)
key = "QWERTYUIOPASDFGHJKLZXCVBNM"  # Example key; replace this
with your own key
key_mapping = create_mapping(key)
print("Using Key (Cipher Alphabet):", key)
# Example plaintext
plaintext = "A GOOD BOY ALAMIN"
print(f"Plaintext: {plaintext}")
ciphertext = monoalphabetic_encrypt(plaintext, key_mapping)
print("Ciphertext:", ciphertext)
```

For Decryption code:

```
#deryption
import string

def create_reverse_mapping(key):
    # Create a reverse dictionary mapping each letter in the key back to the
alphabet
    alphabet = string.ascii_uppercase
    return {key[i]: alphabet[i] for i in range(len(alphabet))}

def monoalphabetic_decrypt(ciphertext, key_mapping):
    plaintext = ""

    for char in ciphertext:
        if char.isupper():  # Decrypt only uppercase letters
            plaintext += key_mapping.get(char, char)  # Substitute using the reverse
key mapping
        else:
            plaintext += char  # Keep non-uppercase characters unchanged

    return plaintext
```

```
# User-provided key (cipher alphabet)
key = "QWERTYUIOPASDFGHJKLZXCVBNM"  # Example key; replace this
with your own key
reverse_key_mapping = create_reverse_mapping(key)
print("Using Key (Cipher Alphabet):", key)

# Given ciphertext
ciphertext = "Q UGGR WGN QSQDOF"
print(f"Ciphertext: {ciphertext}")
plaintext = monoalphabetic_decrypt(ciphertext, reverse_key_mapping)
print("Plaintext:", plaintext)
```

Output:

Encryption:

```
Using Key (Cipher Alphabet): QWERTYUIOPASDFGHJKLZXCVBNM
Plaintext: A GOOD BOY ALAMIN
Ciphertext: Q UGGR WGN QSQDOF
```

Decryption:

```
Using Key (Cipher Alphabet): QWERTYUIOPASDFGHJKLZXCVBNM
Ciphertext: Q UGGR WGN QSQDOF
Plaintext: A GOOD BOY ALAMIN
```

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

i. To study and implement encryption using Caesar cipher.
ii. To study and implement decryption using Brute force attack cipher.

Theory:

A brute force attack on a cipher involves systematically trying all possible keys until the correct one is found, allowing the attacker to decrypt the ciphertext or encrypt a message. This technique is often employed on simpler ciphers with smaller key spaces, such as the Caesar cipher or substitution ciphers. In encryption, the attacker generates all possible ciphertexts by encrypting the plaintext with every key in the key space and then compares the results to identify the correct encryption. In decryption, the attacker attempts all possible keys to decrypt the ciphertext, checking each result for meaningful plaintext. The success of a brute force attack relies on the size of the key space—smaller key spaces allow for quicker attacks, while larger key spaces make brute force infeasible.

While brute force is a straightforward and guaranteed method for breaking ciphers with small key spaces, it becomes impractical for modern encryption algorithms with large key sizes, such as AES or RSA. These algorithms are designed to withstand brute force attacks by using complex mathematical structures and large key spaces, making such attacks computationally infeasible even with significant computational power. Despite this, brute force remains a useful concept in cryptography, particularly for understanding vulnerabilities in classical ciphers and demonstrating the importance of strong key management in modern cryptographic systems.

Algorithm:

Algorithm for Brute Force Attack on Caesar Cipher
Step-by-Step Algorithm

1. Input: Ciphertext message (ciphertext).
2. Initialize: An empty list possible_plaintexts to store all possible decrypted messages.
3. Loop through each possible key from 1 to 25:
    - For each key:
        - Initialize an empty string plaintext to store the decrypted message for this key.
        - For each character ch in the ciphertext:
            - If ch is a letter:

                - Shift it by the key positions backward in the alphabet.
                - Wrap around if needed (e.g., from "A" to "Z").
            - If ch is not a letter, keep it the same.
            - Append the shifted character to plaintext.
        - Add plaintext to possible plaintexts.
4. Output: possible plaintexts, containing all decrypted messages for each key from 1 to 25.

Code(python):

```
def caesar_cipher_encrypt(plaintext, key):
    ciphertext = ""

    for char in plaintext:
        if char.isupper():  # Check if the character is an uppercase letter
            # Shift character by key positions and wrap around with modulo 26
            shifted_char = chr((ord(char) - ord('A') + key) % 26 + ord('A'))
            ciphertext += shifted_char
        else:
            # If the character is not an uppercase letter, keep it the same
            ciphertext += char

    return ciphertext

def caesar_cipher_decrypt(ciphertext, key):
    plaintext = ""

    for char in ciphertext:
        if char.isupper():  # Check if the character is an uppercase letter
            # Shift character by key positions backward and wrap around with
modulo 26
            shifted_char = chr((ord(char) - ord('A') - key) % 26 + ord('A'))
            plaintext += shifted_char
        else:
            # If the character is not an uppercase letter, keep it the same
            plaintext += char

    return plaintext

# Given ciphertext
# Example usage:
plaintext = "THIS IS ALAMIN"
key = 5
ciphertext = caesar_cipher_encrypt(plaintext, key)
print(f"The plaintext:{plaintext}")
print("Encrypted message:", ciphertext)

print("Ciphertext:", ciphertext)

# Brute-force attack
print("Attempting brute-force attack:")
for key in range(1, 26):
    decrypted_text = caesar_cipher_decrypt(ciphertext, key)
    print(f"Key {key}: {decrypted_text}")
```

Output:

```
The plaintext: THIS IS ALAMIN
Encrypted message: YMNX NX FQFRNS
Ciphertext: YMNX NX FQFRNS
Attempting brute-force attack:
Key 1: XLMW MW EPEQMR
Key 2: WKLV LV DODPLQ
Key 3: VJKU KU CNCOKP
Key 4: UIJT JT BMBNJO
Key 5: THIS IS ALAMIN
Key 6: SGHR HR ZKZLHM
Key 7: RFGQ GQ YJYKGL
Key 8: QEFP FP XIXJFK
Key 9: PDEO EO WHWIEJ
Key 10: OCDN DN VGVHDI
Key 11: NBCM CM UFUGCH
Key 12: MABL BL TETFBG
Key 13: LZAK AK SDSEAF
Key 14: KYZJ ZJ RCRDZE
Key 15: JXYI YI QBQCYD
Key 16: IWXH XH PAPBXC
Key 17: HVWG WG OZOAWB
Key 18: GUVF VF NYNZVA
Key 19: FTUE UE MXMYUZ
```

Key 20: ESTD TD LWLXTY
Key 21: DRSC SC KVKWSX
Key 22: CQRB RB JUJVRW
Key 23: BPQA QA ITIUQV
Key 24: AOPZ PZ HSHTPU
Key 25: ZNOY OY GRGSOT

Experiment No:04

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

Objectives:

i. To study and implement encryption using Hill cipher.

ii. To study and implement decryption using Hill cipher.

Theory:

The Hill cipher is a polygraphic substitution cipher that encrypts plaintext in blocks of letters using linear algebra, specifically matrix multiplication. The encryption process involves converting the plaintext into numerical values (typically using A=0, B=1, C=2, etc.), then multiplying these values by a key matrix (usually a square matrix) to produce the ciphertext. The size of the key matrix determines the block size for the encryption. For example, a 2x2 matrix would encrypt two letters at a time, while a 3x3 matrix would encrypt three letters at a time. The main advantage of the Hill cipher over simpler ciphers like Caesar is that it works on multiple letters simultaneously, making it harder to break through frequency analysis.

Decryption using the Hill cipher is the inverse process of encryption. Since the encryption uses matrix multiplication, decryption requires multiplying the ciphertext by the inverse of the key matrix modulo 26 (for the English alphabet). To implement decryption, it is crucial to ensure that the key matrix is invertible, meaning its determinant is non-zero and has a modular inverse. If the key matrix is invertible, the inverse matrix is calculated, and the ciphertext is multiplied by this inverse matrix to recover the original plaintext. The Hill cipher provides a more secure encryption method compared to classical ciphers, but it is still vulnerable to attacks if the key is not kept secret or if the attacker knows the key matrix.

Algorithm:

Encryption Algorithm:

1. Convert the plaintext into numerical values:
   Convert each letter of the plaintext into a number based on its position in the alphabet (A=0, B=1, C=2, ..., Z=25).
2. Organize the plaintext into blocks:
   Divide the plaintext into blocks of size equal to the order of the key matrix (e.g., 2x2 matrix → divide the plaintext into pairs of letters, 3x3 matrix → divide into triplets).
3. Define the key matrix:
   Choose a square matrix as the key matrix K, where each element is a

number between 0 and 25. The size of the matrix will depend on the size of the blocks. For example, for a 2x2 matrix, the key will be a 2x2 matrix.

4. Multiply the plaintext blocks by the key matrix:
   Each block of plaintext (as a column vector) is multiplied by the key matrix K modulo 26 to get the ciphertext.
   $$\mathrm{C} = (\mathrm{K} \times \mathrm{P})\mathrm{mod}\ 26$$
   Where:
   - P is the plaintext vector (in numerical form).
   - C is the resulting ciphertext vector.
   - K is the key matrix.
5. Convert the resulting ciphertext numbers back to letters:
   After obtaining the ciphertext numbers, convert them back to letters using the same numerical alphabet mapping (0=A, 1=B, 2=C, ..., 25=Z).
6. Repeat the process for all blocks of plaintext:
   If the plaintext block size is larger than the matrix size, repeat the encryption process for all blocks of the plaintext.

Decryption Algorithm:

1. Convert the ciphertext into numerical values:
   Convert each letter of the ciphertext into a number based on its position in the alphabet (A=0, B=1, C=2, ..., Z=25).
2. Calculate the inverse of the key matrix:
   To decrypt, calculate the inverse of the key matrix K modulo 26. This is done by finding the determinant of K and calculating its modular inverse. If the determinant is 0 (mod 26), the matrix is not invertible, and decryption is not possible. The inverse matrix $K^{-1}$is used in the decryption process.
   $$K^{-1} = \text{modular inverse of the key matrix}$$
3. Multiply the ciphertext blocks by the inverse key matrix:
   Each block of ciphertext (in numerical form) is multiplied by the inverse key matrix K^{-1} modulo 26 to get the plaintext.
   $$P = K^{-1} \times C \ \ mod\ 26$$
   Where:
   - C is the ciphertext vector.
   - P is the resulting plaintext vector.
   - K^{-1} is the inverse of the key matrix.
4. Convert the resulting plaintext numbers back to letters:
   After obtaining the plaintext numbers, convert them back to letters using the same numerical alphabet mapping (0=A, 1=B, 2=C, ..., 25=Z).

5. Repeat the process for all blocks of ciphertext:
   If the ciphertext block size is larger than the matrix size, repeat the decryption process for all blocks of the ciphertext.

Example

Plaintext: ALAMIN

Key: DDCF

Perform encryption and decryption using Hill cipher.

Solution:

We use the position table of alphabet for the ciphertext calculation.

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
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

Plain text divide in a three pair AL AM IN

$$P1 = \begin{bmatrix} A \\ L \end{bmatrix} = \begin{bmatrix} 0 \\ 11 \end{bmatrix}, \quad P2 = \begin{bmatrix} A \\ M \end{bmatrix} = \begin{bmatrix} 0 \\ 12 \end{bmatrix}, \quad P3 = \begin{bmatrix} I \\ N \end{bmatrix} = \begin{bmatrix} 8 \\ 13 \end{bmatrix}$$

Make a 2×2 key matrix using the position table value:

$$K = \begin{bmatrix} 3 & 3 \\ 2 & 5 \end{bmatrix}$$

For the first pair Ciphertext C1

$$C1 = (K \times P1) mod\ 26$$

$$= \left(\begin{bmatrix} 3 & 3 \\ 2 & 5 \end{bmatrix} \times \begin{bmatrix} 0 \\ 11 \end{bmatrix}\right) mod\ 26 = \begin{bmatrix} 33 \\ 55 \end{bmatrix} mod\ 26 = \begin{bmatrix} 7 \\ 3 \end{bmatrix} = \begin{bmatrix} H \\ D \end{bmatrix}$$

For the first pair Ciphertext C2

$$C2 = (K \times P2) mod\ 26$$

$$= \left(\begin{bmatrix} 3 & 3 \\ 2 & 5 \end{bmatrix} \times \begin{bmatrix} 0 \\ 12 \end{bmatrix}\right) mod\ 26 = \begin{bmatrix} 36 \\ 60 \end{bmatrix} mod\ 26 = \begin{bmatrix} 10 \\ 8 \end{bmatrix} = \begin{bmatrix} K \\ I \end{bmatrix}$$

For the first pair Ciphertext C3

$$C3 = (K \times P3) mod\ 26$$

$$= \left(\begin{bmatrix} 3 & 3 \\ 2 & 5 \end{bmatrix} \times \begin{bmatrix} 8 \\ 13 \end{bmatrix}\right) mod\ 26 = \begin{bmatrix} 63 \\ 81 \end{bmatrix} mod\ 26 = \begin{bmatrix} 11 \\ 3 \end{bmatrix} = \begin{bmatrix} L \\ D \end{bmatrix}$$

So the ciphertext: HDKILD

Again

For decrypt the ciphertext at first calculate the K^{-1}

$$K^{-1} = \frac{1}{ad - bc}\begin{bmatrix} d & -b \\ -c & a \end{bmatrix} = \frac{1}{9}\begin{bmatrix} 5 & -3 \\ -2 & 3 \end{bmatrix}$$

$$Now, \begin{bmatrix} 5 & -3 \\ -2 & 3 \end{bmatrix} mod\ 26 = \begin{bmatrix} 5 & 23 \\ 24 & 3 \end{bmatrix} and$$

$$Also\ calculate\ d^{-1}$$

$$(9 \times d^{-1})\ mod\ 26 = 1\ or\ d^{-1} = 3$$

$$K^{-1} = 3\begin{bmatrix} 5 & 23 \\ 24 & 3 \end{bmatrix} mod\ 26 = \begin{bmatrix} 15 & 69 \\ 24 & 9 \end{bmatrix} mod\ 26 = \begin{bmatrix} 15 & 17 \\ 20 & 9 \end{bmatrix}$$

$$Finally, \qquad K^{-1} = \begin{bmatrix} 15 & 17 \\ 20 & 9 \end{bmatrix}$$

For the first pair Plaintext P1

$$P1 = (K^{-1} \times C1) mod\ 26$$

$$= \left(\begin{bmatrix} 15 & 17 \\ 20 & 9 \end{bmatrix} \times \begin{bmatrix} 7 \\ 3 \end{bmatrix}\right) mod\ 26 = \begin{bmatrix} 156 \\ 167 \end{bmatrix} mod\ 26 = \begin{bmatrix} 0 \\ 11 \end{bmatrix} = \begin{bmatrix} A \\ L \end{bmatrix}$$

For the first pair Plaintext $P2$

$$P2 = (K^{-1} \times C2) mod\ 26$$

$$= \left(\begin{bmatrix} 15 & 17 \\ 20 & 9 \end{bmatrix} \times \begin{bmatrix} 10 \\ 8 \end{bmatrix}\right) mod\ 26 = \begin{bmatrix} 286 \\ 272 \end{bmatrix} mod\ 26 = \begin{bmatrix} 0 \\ 12 \end{bmatrix} = \begin{bmatrix} A \\ M \end{bmatrix}$$

For the first pair Plaintext $P3$

$$P3 = (K^{-1} \times C3) mod\ 26$$

$$= \left(\begin{bmatrix} 15 & 17 \\ 20 & 9 \end{bmatrix} \times \begin{bmatrix} 11 \\ 3 \end{bmatrix}\right) mod\ 26 = \begin{bmatrix} 216 \\ 247 \end{bmatrix} mod\ 26 = \begin{bmatrix} 8 \\ 13 \end{bmatrix} = \begin{bmatrix} I \\ N \end{bmatrix}$$

So the plaintext: ALAMIN.

Code(python):

Code for Encryption:

```
import numpy as np

def text_to_numbers(text):
    # Convert text to numbers (A=0, B=1, ..., Z=25)
    return [ord(char) - ord('A') for char in text]

def numbers_to_text(numbers):
    # Convert numbers back to text (0=A, 1=B, ..., 25=Z)
    return ''.join(chr(num + ord('A')) for num in numbers)

def hill_cipher_encrypt(plaintext, key_matrix):
    # Ensure key matrix is a numpy array
    key_matrix = np.array(key_matrix)

    # Check if the key matrix is square
    n = key_matrix.shape[0]
    if key_matrix.shape[1] != n:
        raise ValueError("Key matrix must be square")
    # Ensure plaintext length is a multiple of n
    if len(plaintext) % n != 0:
        # Add padding 'X' if plaintext length is not a multiple of matrix order
        plaintext += 'X' * (n - len(plaintext) % n)

    # Convert plaintext to numbers
    plaintext_numbers = text_to_numbers(plaintext)

    ciphertext_numbers = []

    # Encrypt each block
    for i in range(0, len(plaintext_numbers), n):
        # Take n-length block from plaintext
        block = plaintext_numbers[i:i + n]
        # Convert block to a column vector
        block_vector = np.array(block).reshape((n, 1))
        # Multiply by key matrix and take mod 26
        cipher_vector = np.dot(key_matrix, block_vector) % 26
        # Flatten and add to ciphertext numbers
        ciphertext_numbers.extend(cipher_vector.flatten())

    # Convert ciphertext numbers to text
    return numbers_to_text(ciphertext_numbers)

# Example usage:
plaintext = "ALAMIN"
key_matrix = [[3, 3], [2, 5]]  # Example 2x2 key matrix; modify as needed

ciphertext = hill_cipher_encrypt(plaintext, key_matrix)
print(f"Plaintext: {plaintext}")
print(f"Ciphertext: {ciphertext}")
```

Code for Decryption:

```
import numpy as np

# Extended Euclidean Algorithm to find the modular inverse of a number
def mod_inverse(a, m):
    m0, x0, x1 = m, 0, 1
    if m == 1:
        return 0
    while a > 1:
        q = a // m
        m, a = a % m, m
        x0, x1 = x1 - q * x0, x0
    if x1 < 0:
        x1 += m0
    return x1

# Function to calculate the inverse of a 2x2 matrix mod 26
def matrix_inverse(matrix):
    # Calculate determinant (ad - bc) mod 26
    det = (matrix[0][0] * matrix[1][1] - matrix[0][1] * matrix[1][0]) % 26
    det_inv = mod_inverse(det, 26)  # Find modular inverse of the determinant

    # Calculate the inverse matrix
    inv_matrix = np.array([[matrix[1][1], -matrix[0][1]],
                          [-matrix[1][0], matrix[0][0]]]) % 26

    # Multiply the inverse matrix by the determinant inverse
    inv_matrix = (inv_matrix * det_inv) % 26
    return inv_matrix

# Function to perform Hill cipher decryption
def hill_decrypt(ciphertext, key_matrix):
    # Create the inverse key matrix
    inv_key_matrix = matrix_inverse(np.array(key_matrix))

    # Prepare ciphertext (pairs of letters)
    ciphertext = ciphertext.upper().replace(" ", "")
    ciphertext_pairs = [(ord(c1) - ord('A'), ord(c2) - ord('A')) for c1, c2 in
zip(ciphertext[::2], ciphertext[1::2])]

    # Decrypt each ciphertext pair
    plaintext = ""
    for pair in ciphertext_pairs:
        c1, c2 = pair
        # Perform matrix multiplication (K^(-1) × C) mod 26
        p1 = (inv_key_matrix[0][0] * c1 + inv_key_matrix[0][1] * c2) % 26
        p2 = (inv_key_matrix[1][0] * c1 + inv_key_matrix[1][1] * c2) % 26
        # Convert numbers back to letters
        plaintext += chr(p1 + ord('A')) + chr(p2 + ord('A'))

    return plaintext

# Example Usage
ciphertext = "HDKILD"
key_matrix = [[3, 3], [2, 5]]  # K^(-1) as described in the problem

plaintext = hill_decrypt(ciphertext, key_matrix)
print(f"Ciphertext: {ciphertext}")
print(f"Decrypted Plaintext: {plaintext}")
```

Output:

Encryption:

```
Plaintext: ALAMIN
Ciphertext: HDKILD
```

Decryption:

```
Ciphertext: HDKILD
Decrypted Plaintext: ALAMIN
```

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

To implement encryption using Playfair cipher.

Theory:

The Playfair cipher is a classical encryption technique that encrypts pairs of letters (digraphs) instead of individual letters, making it more secure than simpler substitution ciphers like the Caesar cipher. It was invented by Charles Wheatstone in 1854 but became widely known after being promoted by Lord Playfair, who was a friend of Wheatstone.

In the Playfair cipher, encryption involves substituting each pair of letters (digraphs) from the plaintext with corresponding letters from a 5x5 key square. The key square is generated using a keyword, where duplicates are removed and the letters 'I' and 'J' are typically combined into a single slot to fit within the 25 available positions. To encrypt, each digraph is processed according to three rules: if the two letters are in the same row, they are replaced by the letters immediately to their right; if they are in the same column, they are replaced by the letters immediately below; if they form a rectangle, the letters are replaced by the ones in the same row but at the opposite corners. This method of encryption, operating on digraphs rather than individual letters, makes it more secure than simpler ciphers like the Caesar cipher, though it is still vulnerable to frequency analysis.

Algorithm:

Steps of the Playfair Cipher Encryption Algorithm:

1. **Preparation of the Key Square**:
   - **Step 1**: Choose a key (e.g., "KEYWORD").
   - **Step 2**: Remove any duplicate letters from the keyword.
   - **Step 3**: Create a 5x5 matrix (key square) using the letters of the keyword, filling in the remaining spaces with the remaining letters of the alphabet (if plaintext contain, I keep I otherwise J).

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

2. **Preparation of the Plaintext:**
   - **Step 4**: Break the plaintext into pairs of letters (digraphs). If there is an odd number of letters, add a 'Z' at the end.
   - **Step 5**: If any pair of letters in the plaintext are identical, insert a 'X' between them to avoid repeating letters.
3. **Encryption of the Digraphs**: For each pair of letters (digraphs) in the plaintext, apply the following rules:
   - **Rule 1**: If both letters of the digraph appear in the same row of the key square, replace each letter with the letter immediately to its right. If the letter is at the end of the row, wrap around to the beginning of that row.
   - **Rule 2**: If both letters of the digraph appear in the same column of the key square, replace each letter with the letter immediately below it. If the letter is at the bottom of the column, wrap around to the top of that column.
   - **Rule 3**: If the letters form a rectangle (i.e., they are not in the same row or column), replace each letter with the letter in the same row but in the column of the other letter in the pair.
4. **Return the Ciphertext**: After all digraphs are encrypted, concatenate the results to form the final ciphertext.

Example

To encrypt the plaintext "ALAMIN BW MP" using the Playfair cipher with the key "KEYWORD,"

Solution.

The plaintext contains I so the key square contains I

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

Split the plaintext " ALAMIN BW MP " into digraphs (pairs of two letters): **AL**, **AM**, **IN, BW, MP**.

Here AL, AM, IN are the diagonal of the square. Replace it with other diagonal of the square.

$A \longrightarrow C$	$A \longrightarrow R$	$I \longrightarrow G$
$L \longrightarrow H$	$M \longrightarrow P$	$N \longrightarrow Q$

BW are in same column. Replace it below letter in the column

$$B \longrightarrow I$$
$$W \longrightarrow B$$

MP are in same row. Replace it right letter in the row

$$M \longrightarrow N$$
$$P \longrightarrow Q$$

So final cipher text is CH RP GQ IB NQ

Code(python):

```
import string
def create_key_square(key):
    # Step 1: Remove duplicates from the key and convert to uppercase
    key = ''.join(sorted(set(key), key=lambda x: key.index(x))).upper()

    # Step 2: Remove any letter that isn't a letter or is 'J' (we will treat 'I' and 'J' as
the same letter)
    alphabet = string.ascii_uppercase.replace('J', '')  # Alphabet without 'J'
    key_square = key + ''.join([letter for letter in alphabet if letter not in key])

    # Create the 5x5 key square matrix
    key_matrix = [key_square[i:i+5] for i in range(0, len(key_square), 5)]
    return key_matrix

def prepare_plaintext(plaintext):
    # Step 3: Remove spaces and convert to uppercase
    plaintext = plaintext.replace(' ', '').upper()

    # Step 4: Prepare the plaintext (break into pairs, handle duplicate letters)
    prepared_text = []
    i = 0
    while i < len(plaintext):
        # If the next letter is the same, insert a 'Z' in between
        if i + 1 < len(plaintext) and plaintext[i] == plaintext[i + 1]:
            prepared_text.append(plaintext[i] + 'Z')
            i += 1
        elif i + 1 < len(plaintext):
            prepared_text.append(plaintext[i] + plaintext[i + 1])
            i += 2
        else:
            prepared_text.append(plaintext[i] + 'Z')  # Add 'Z' if the length is odd
            i += 1
    return prepared_text
```

```
def find_position(char, key_matrix):
    for row in range(5):
        for col in range(5):
            if key_matrix[row][col] == char:
                return row, col
    return None

def playfair_encrypt(plaintext, key):
    key_matrix = create_key_square(key)
    prepared_text = prepare_plaintext(plaintext)

    ciphertext = []

    # Step 5: Encrypt each digraph
    for digraph in prepared_text:
        first_char, second_char = digraph

        # Find positions of each character in the key matrix
        row1, col1 = find_position(first_char, key_matrix)
        row2, col2 = find_position(second_char, key_matrix)

        # Rule 1: Same row
        if row1 == row2:
            ciphertext.append(key_matrix[row1][(col1 + 1) % 5])
            ciphertext.append(key_matrix[row2][(col2 + 1) % 5])

        # Rule 2: Same column
        elif col1 == col2:
            ciphertext.append(key_matrix[(row1 + 1) % 5][col1])
            ciphertext.append(key_matrix[(row2 + 1) % 5][col2])

        # Rule 3: Rectangle (different row, different column)
        else:
            ciphertext.append(key_matrix[row1][col2])
            ciphertext.append(key_matrix[row2][col1])

    # Join the ciphertext list into a single string
    return ''.join(ciphertext)

# Example usage:
```

```
plaintext = "ALAMIN BW MP"
key = "KEYWORD"
ciphertext = playfair_encrypt(plaintext, key)
print(f"Plaintext: {plaintext}")
print(f"Ciphertext: {ciphertext}")
```

Output:

```
          Plaintext: ALAMIN BW MP
          Ciphertext: CHRPGQIBNQ
```

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

Objective:
To implement decryption using Playfair cipher.

Theory:
The Playfair cipher is a classical encryption technique that encrypts pairs of letters (digraphs) instead of individual letters, making it more secure than simpler substitution ciphers like the Caesar cipher. It was invented by Charles Wheatstone in 1854 but became widely known after being promoted by Lord Playfair, who was a friend of Wheatstone.

To decrypt a message encrypted with the Playfair cipher, the same key used for encryption is required. The process of decryption follows rules that are the reverse of encryption.

Algorithm:

Steps for Playfair Cipher Decryption:

1. **Preparation of the Key Square**:
   This is the same as in the encryption process.
   Step 1: Choose a key (e.g., "KEYWORD").
   Step 2: Remove any duplicate letters from the keyword.
   Step 3: Create a 5x5 matrix (key square) using the letters of the keyword, filling in the remaining spaces with the remaining letters of the alphabet (if plaintext contain, I keep I otherwise J).
   Example of key square:

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

2. **Preparation of the Ciphertext:**
   Step 4: Break the ciphertext into pairs of letters (digraphs).
   Step 5: If there is an odd number of letters, add a 'Z' at the end.
   Step 6: If any pair of letters in the ciphertext are identical, insert a 'X' between them to avoid repeating letters.

3. **Decryption of the Digraphs**: For each pair of letters (digraphs) in the ciphertext, apply the following rules:

Rule 1: If both letters of the digraph appear in the same row of the key square, replace each letter with the letter immediately to its left. If the letter is at the beginning of the row, wrap around to the end of that row.
Rule 2: If both letters of the digraph appear in the same column of the key square, replace each letter with the letter immediately above it. If the letter is at the top of the column, wrap around to the bottom of that column.
Rule 3: If the letters form a rectangle (i.e., they are not in the same row or column), replace each letter with the letter in the same row but in the column of the other letter in the pair.

4. **Return the Plaintext**: After all digraphs are decrypted, concatenate the results to form the final plaintext.

Example

To decrypt the ciphertext " CHRPGQIBNQ " using the Playfair cipher with the key "KEYWORD,"

Solution.

The ciphertext contains I so the key square contains I

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

Split the ciphertext " CH RP GQ IB NQ " into digraphs (pairs of two letters): CH RP GQ IB NQ

Here CH RP GQ are the diagonal of the square. Replace it with other diagonal of the square.

$C \longrightarrow A$	$R \longrightarrow A$	$G \longrightarrow I$
$H \longrightarrow L$	$P \longrightarrow M$	$Q \longrightarrow N$

BW are in same column. Replace it upper letter in the column

$$I \longrightarrow B$$
$$B \longrightarrow W$$

MP are in same row. Replace it left letter in the row

$$N \longrightarrow M$$
$$Q \longrightarrow P$$

So final plaintext is ALAMIN BW MP

Code(python):

```
def create_key_square(key):
    key = key.upper().replace("J", "I")  # Combine I/J
    seen = set()
    key_square = []

    # Add unique characters from the key
    for char in key:
        if char not in seen and char.isalpha():
            seen.add(char)
            key_square.append(char)

    # Add remaining letters of the alphabet
    alphabet = "ABCDEFGHIKLMNOPQRSTUVWXYZ"  # Exclude 'J'
    for char in alphabet:
        if char not in seen:
            key_square.append(char)
            seen.add(char)

    return [key_square[i:i+5] for i in range(0, len(key_square), 5)]

def find_position(char, key_square):
    for i, row in enumerate(key_square):
        if char in row:
            return i, row.index(char)
    return -1, -1

def decrypt_digraph(digraph, key_square):
    row1, col1 = find_position(digraph[0], key_square)
    row2, col2 = find_position(digraph[1], key_square)

    # Rule 1: Same row - move left
    if row1 == row2:
        return key_square[row1][(col1 - 1) % 5] + key_square[row2][(col2 - 1) % 5]

    # Rule 2: Same column - move up
    elif col1 == col2:
        return key_square[(row1 - 1) % 5][col1] + key_square[(row2 - 1) % 5][col2]

    # Rule 3: Rectangle - swap corners
    else:
        return key_square[row1][col2] + key_square[row2][col1]
```

```
def playfair_decrypt(ciphertext, key):
    key_square = create_key_square(key)
    ciphertext = ciphertext.upper().replace(" ", "")

    # Split ciphertext into digraphs
    digraphs = [ciphertext[i:i+2] for i in range(0, len(ciphertext), 2)]

    # Decrypt each digraph
    plaintext = ''.join(decrypt_digraph(digraph, key_square) for digraph in digraphs)

    return plaintext

# Example Usage
ciphertext = "CHRPGQIBNQ"
key = "KEYWORD"

plaintext = playfair_decrypt(ciphertext, key)
print(f"Ciphertext: {ciphertext}")
print(f"Key: {key}")
print(f"Plaintext: {plaintext}")
```

Output:

```
        Ciphertext: CHRPGQIBNQ
        Key: KEYWORD
        Plaintext: ALAMINBWMP
```

Experiment No:07

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

Objective:

To implement encryption using Poly-Alphabetic cipher.

Theory:

The Vigenère cipher is a method of encryption that uses a keyword to shift each letter of the plaintext by a varying number of positions. It is a type of poly-alphabetic cipher, meaning that multiple substitution alphabets are used throughout the encryption process, making it more secure than mono-alphabetic ciphers like the Caesar cipher. The cipher uses a repeating keyword to determine the shift for each letter in the plaintext, which makes the pattern of shifts harder to detect.

The Vigenère cipher offers better security than simpler ciphers like the Caesar cipher by using multiple shifting alphabets, which makes it more resistant to frequency analysis. It is also flexible, allowing for varying lengths of both plaintext and keyword. However, it has some vulnerabilities. If the keyword is reused or is too short in relation to the plaintext, it can be susceptible to attacks like the Kasiski examination. Additionally, when the keyword is short and repeated, patterns in the ciphertext can emerge, making the cipher more vulnerable to decryption.

Algorithm:

Simple Algorithm in Steps:

1. Input:
    - Plaintext (Message to be encrypted).
    - Keyword (Secret key).
2. Extend the Keyword to match the length of the plaintext.
3. For each character in the plaintext:
    - Convert the character and corresponding keyword letter to a number (A = 0, B = 1, ..., Z = 25).
    - Add the numbers.
    - Take the result modulo 26.
    - Convert the result back to a letter.
4. Output the Ciphertext.

   Formula for making cipher text

$$C_j = (P_j + K_j) MOD\ 26$$

$$Where\ C_j\ is\ the\ cipher\ text, P_j\ is\ the\ plaintext\ and\ K_j\ is\ the\ key$$

Example:

Plaintext: I LOVE CODING

Key: SHE LIKE CATS

Using Vigenère to make the ciphertext.

Solution:

We use the position table of alphabet for the ciphertext calculation.

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
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

Plaintext: I LOVE CODING

Key: SHE LIKE CATS

Plain text	I	L	O	V	E	C	O	D	I	N	G
Position P_j	8	11	14	21	4	2	14	3	8	13	6
Key	S	H	E	L	I	K	E	C	A	T	S
Position K_j	18	7	4	11	8	10	4	2	0	19	18
$C_j = P_j + K_j$	26	18	18	32	12	12	18	5	8	32	24
Ciphertext position $C_j \ mod\ 26$	0	18	18	6	12	12	18	5	8	6	24
Ciphertext	A	S	S	G	M	M	S	F	I	G	Y

So the ciphertext is: ASSGNNSFIGY

Code(python):

```
def vigenere_encrypt(plaintext, key):
    # Convert plaintext and key to uppercase and remove spaces
    plaintext = plaintext.replace(" ", "").upper()
    key = key.replace(" ", "").upper()

    # Initialize the ciphertext as an empty string
    ciphertext = ""

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

    # Encrypt each character in the plaintext
    for i in range(len(plaintext)):
        # Get the position of the plaintext character (A=0, B=1, ..., Z=25)
        P_j = ord(plaintext[i]) - ord('A')

        # Get the position of the key character (A=0, B=1, ..., Z=25)
        K_j = ord(key_extended[i]) - ord('A')
        # Apply the encryption formula (C_j = (P_j + K_j) % 26)
        C_j = (P_j + K_j) % 26
        # Convert the ciphertext position back to a letter
        ciphertext += chr(C_j + ord('A'))
    return ciphertext
# Example usage:
plaintext = "I LOVE CODING"
key = "SHE LIKE CATS"
ciphertext = vigenere_encrypt(plaintext, key)
print(f"Plaintext: {plaintext}")
print(f"Key: {key}")
print(f"Ciphertext: {ciphertext}")
```

Output:

```
Plaintext: I LOVE CODING
Key: SHE LIKE CATS
Ciphertext: ASSGMMSFIGY
```

Experiment No:08
Experiment Name: Write a program to implement decryption using Poly-Alphabetic cipher (Vigenère cipher).
Objective:

To implement decryption using Poly-Alphabetic cipher (Vigenère cipher).

Theory:

The Vigenère cipher is a method of encryption that uses a keyword to shift each letter of the plaintext by a varying number of positions. It is a type of poly-alphabetic cipher, meaning that multiple substitution alphabets are used throughout the encryption process, making it more secure than mono-alphabetic ciphers like the Caesar cipher. The cipher uses a repeating keyword to determine the shift for each letter in the plaintext, which makes the pattern of shifts harder to detect.

Decryption using the Vigenère cipher involves reversing the encryption process by applying the same keyword used for encryption. To decrypt, the ciphertext is analyzed by subtracting the corresponding letter of the keyword from each letter in the ciphertext. The result of this subtraction is then adjusted using modulo 26 to ensure it falls within the range of the alphabet. If the result is negative, 26 is added to bring it back within the valid alphabet range. This process is repeated for each letter in the ciphertext, using the extended keyword, and eventually restores the original plaintext message.

Algorithm:

Vigenère Cipher Decryption Algorithm:

1. Input:
   - Ciphertext (the encrypted message).
   - Keyword (the secret key used for encryption).
2. Extend the Keyword:
   - If the keyword is shorter than the ciphertext, repeat the keyword until it matches the length of the ciphertext.
3. Decrypt the Ciphertext:
   - For each letter in the ciphertext:
     1. Convert the ciphertext letter and the corresponding keyword letter to their corresponding positions in the alphabet (A = 0, B = 1, ..., Z = 25).
     2. Subtract the position of the keyword letter from the position of the ciphertext letter.
     3. If the result is negative, add 26 to ensure it remains within the alphabet range.

    4. Take the result modulo 26 to wrap around if necessary.
    5. Convert the result back to a letter.
4. Output:
    - The resulting plaintext (the original message).

Formula for making plaintext

$$P_j = (C_j - K_j) MOD\ 26$$

$$Where\ C_j\ is\ the\ cipher\ text, P_j\ is\ the\ plaintext\ and\ K_j\ is\ the\ key$$

Example:

Ciphertext: ASSGNNSFIGY
Key: SHE LIKE CATS

Using Vigenère to make the plaintext.

Solution:

We use the position table of alphabet for the ciphertext calculation.

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
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

Ciphertext: ASSGNNSFIGY
Key: SHE LIKE CATS

Ciphertext	A	S	S	G	M	M	S	F	I	G	Y
Position C_j	0	18	18	6	12	12	18	5	8	6	24
Key	S	H	E	L	I	K	E	C	A	T	S
Position K_j	18	7	4	11	8	10	4	2	0	19	18
$P_j = C_j - K_j$	-18	11	14	-5	4	2	14	3	8	-13	6
Ciphertext position $P_j\ mod\ 26$	8	11	14	21	4	2	14	3	8	13	6
Plaintext	I	L	O	V	E	C	O	D	I	N	G

So the ciphertext is: I LOVE CODING

Code(python):

```
def vigenere_decrypt(ciphertext, key):
   # Convert ciphertext and key to uppercase and remove spaces
   ciphertext = ciphertext.replace(" ", "").upper()
   key = key.replace(" ", "").upper()
   # Initialize the plaintext as an empty string
   plaintext = ""

   # Extend the key to match the length of the ciphertext
   key_extended = (key * (len(ciphertext) // len(key))) + key[:len(ciphertext) %
len(key)]
```

```
    # Decrypt each character in the ciphertext
    for i in range(len(ciphertext)):
        # Get the position of the ciphertext character (A=0, B=1, ..., Z=25)
        C_j = ord(ciphertext[i]) - ord('A')

        # Get the position of the key character (A=0, B=1, ..., Z=25)
        K_j = ord(key_extended[i]) - ord('A')

        # Apply the decryption formula (P_j = (C_j - K_j) % 26)
        P_j = (C_j - K_j) % 26

        # Convert the plaintext position back to a letter
        plaintext += chr(P_j + ord('A'))

    return plaintext

# Example usage:
ciphertext = "A SSGM MSFIGY"
key = "SHE LIKE CATS"

plaintext = vigenere_decrypt(ciphertext, key)

print(f"Ciphertext: {ciphertext}")
print(f"Key: {key}")
print(f"Plaintext: {plaintext}")
```

Output:

```
Ciphertext: A SSGM MSFIGY
Key: SHE LIKE CATS
Plaintext: ILOVECODING
```

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

To study and implement encryption using Vernam cipher.

Theory:

The Vernam cipher, also known as the one-time pad cipher, is a type of symmetric encryption technique that is considered theoretically unbreakable when implemented correctly. It was invented by Gilbert Vernam in 1917 and operates by combining the plaintext with a random key or "pad" that is as long as the plaintext itself.

The Vernam cipher’s security relies on the following conditions:

- The key must be truly random.
- The key must be as long as the message.
- The key can only be used once, hence the term "one-time pad."

When these conditions are met, the Vernam cipher is considered unbreakable because, without knowing the key, all possible messages of that length are equally likely to be the original message.

Algorithm:

Steps:

i. Convert the Plaintext and Key to Binary:
   a. If the plaintext and key are in text form, convert each character into its binary representation (ASCII or UTF-8 binary encoding).

ii. XOR Each Bit of Plaintext with the Key:
   a. For each bit in the plaintext, apply the XOR (exclusive OR) operation with the corresponding bit in the key:

$$\text{Ciphertext\_Bit} = \text{Plaintext\_Bit} \oplus \text{Key\_Bit}$$

   b. XOR ($\oplus$) produces 1 if the bits are different and 0 if they are the same.

iii. Concatenate the Encrypted Bits:
   a. Combine the resulting bits to form the encrypted ciphertext.

iv. Output the Ciphertext:
   a. The ciphertext is now the encrypted message in binary form (according the position convert back to text form if needed).

Example

Plaintext: ABCDE
Key: BCDEF
Using this plaintext and key generate a ciphertext.

Solution:

We use the position table of alphabet for the ciphertext calculation.

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
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

		Position of alphabet	Binary Value of this positions
Plaintext:	ABCDE	0 1 2 3 4	0 01 10 011 100
Key:	BCDEF	1 2 3 4 5	1 10 11 100 101
		XOR operation:	1 11 01 111 001
		Position of alphabet:	1 3 1 7 1
		Ciphertext:	B D B H B

So the Ciphertext is: BDBHB

Code(python):

```
def char_to_bin(char):
    # Convert a character to its corresponding binary value (5 bits, 0 to 25)
    return format(ord(char) - ord('A'), '05b')

def bin_to_char(binary):
    # Convert a 5-bit binary value back to the corresponding character (0 to 25 ->
A to Z)
    return chr(int(binary, 2) + ord('A'))

def xor_encrypt(plaintext, key):
    # Ensure plaintext and key are the same length by repeating the key if
necessary
    plaintext = plaintext.upper().replace(" ", "")
    key = key.upper().replace(" ", "")

    # Initialize the ciphertext as an empty string
    ciphertext = ""

    # XOR each character in the plaintext with the corresponding character in the
key
    for i in range(len(plaintext)):
        # Get the binary representations of the plaintext and key characters
        plaintext_bin = char_to_bin(plaintext[i])
        key_bin = char_to_bin(key[i % len(key)])  # Repeat the key if necessary
        # XOR the binary values
        xor_result = ''.join(['1' if plaintext_bin[j] != key_bin[j] else '0' for j in
range(5)])

        # Convert the XOR result back to a character
        ciphertext_char = bin_to_char(xor_result)
        ciphertext += ciphertext_char

        # Print the details
        #print(f"Plaintext: {plaintext[i]} (Binary: {plaintext_bin})")
        #print(f"Key: {key[i % len(key)]} (Binary: {key_bin})")
        #print(f"XOR Result: {xor_result} (Ciphertext: {ciphertext_char})\n")

    return ciphertext

# Example usage:
plaintext = "ABCDE"
key = "BCDEF"

ciphertext = xor_encrypt(plaintext, key)
print(f"Plaintext: {plaintext}")
print(f"Key: {key}")
print(f"Ciphertext: {ciphertext}")
```

Output:

```
Plaintext: ABCDE
Key: BCDEF
Ciphertext: BDBHB
```

Experiment No:10
Experiment Name: Write a program to implement decryption using Vernam cipher.
Objective:

To study and implement decryption using Vernam cipher.

Theory:

The Vernam cipher, also known as the one-time pad cipher, is a type of symmetric encryption technique that is considered theoretically unbreakable when implemented correctly. It was invented by Gilbert Vernam in 1917 and operates by combining the plaintext with a random key or "pad" that is as long as the plaintext itself.

The decryption process of the Vernam cipher is straightforward due to the symmetric nature of the XOR operation. Since the encryption involves XORing each bit of the plaintext with a corresponding bit from a key, the same process is used for decryption. To decrypt the ciphertext, we simply XOR it again with the same key. This works because of the fundamental property of XOR:

$$\text{Cipher\_text} \oplus Key\_bit = \text{Plain\_text}$$

As long as the key is the same length as the plaintext, truly random, and used only once, the Vernam cipher guarantees perfect secrecy, meaning that the original plaintext can be perfectly recovered. The security of the Vernam cipher lies in the key, and as long as the key remains secret and is not reused, the encryption is theoretically unbreakable.

Algorithm:

Steps:

i. Convert the Ciphertext and Key to Binary: The ciphertext and the key are converted to binary representations (if not already in binary form).
ii. Apply XOR Operation on Each Bit: For each bit in the ciphertext, apply the XOR operation with the corresponding bit in the key.
iii. Concatenate the Decrypted Bits: Combine the resulting bits to form the original binary plaintext.
iv. Convert Back to Text: The decrypted binary data is then converted back to readable text form.

Example

Ciphertext: BDBHB
Key: BCDEF

Using this plaintext and key generate a ciphertext.

Solution:

We use the position table of alphabet for the plaintext calculation.

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
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

		Position of alphabet	Positions binary Value
Ciphertext:	BDBHB	1 3 1 7 1	1 11 01 111 001
Key:	BCDEF	1 2 3 4 5	1 10 11 100 101
		XOR operation:	0 01 10 011 100
		Position of alphabet:	0 1 2 3 4
		Ciphertext:	A B C D E

So the Ciphertext is: ABCDE

Code(python):

```
def char_to_bin(char):
    # Convert a character to its corresponding binary value (5 bits, 0 to 25)
    return format(ord(char) - ord('A'), '05b')

def bin_to_char(binary):
    # Convert a 5-bit binary value back to the corresponding character (0 to 25 ->
A to Z)
    return chr(int(binary, 2) + ord('A'))

def xor_decrypt(ciphertext, key):
    # Ensure ciphertext and key are the same length by repeating the key if
necessary
    ciphertext = ciphertext.upper().replace(" ", "")
    key = key.upper().replace(" ", "")

    # Initialize the plaintext as an empty string
    plaintext = ""

    # XOR each character in the ciphertext with the corresponding character in
the key
    for i in range(len(ciphertext)):
        # Get the binary representations of the ciphertext and key characters
        ciphertext_bin = char_to_bin(ciphertext[i])
        key_bin = char_to_bin(key[i % len(key)])  # Repeat the key if necessary

        # XOR the binary values
        xor_result = ''.join(['1' if ciphertext_bin[j] != key_bin[j] else '0' for j in
range(5)])

        # Convert the XOR result back to a character
        plaintext_char = bin_to_char(xor_result)
        plaintext += plaintext_char

        # Print the details
        #print(f"Ciphertext: {ciphertext[i]} (Binary: {ciphertext_bin})")
        #print(f"Key: {key[i % len(key)]} (Binary: {key_bin})")
        #print(f"XOR Result: {xor_result} (Plaintext: {plaintext_char})\n")

    return plaintext

# Example usage:
ciphertext = "BDBHB"
key = "BCDEF"

plaintext = xor_decrypt(ciphertext, key)
print(f"Ciphertext: {ciphertext}")
print(f"Key: {key}")
print(f"Plaintext: {plaintext}")
```

Output:

```
Ciphertext: BDBHB
Key: BCDEF
Plaintext: ABCDE
```