## EXPERIMENT 1:

**AIM: To implement encryption and decryption using various substitution techniques such as the Caesar cipher, Playfair cipher, Hill cipher, and Vigenère cipher.**

### 1. Caesar Cipher

**Description:**
Caesar Cipher is one of the simplest and oldest encryption techniques. It works by shifting each letter of the plaintext by a fixed number of positions in the alphabet.

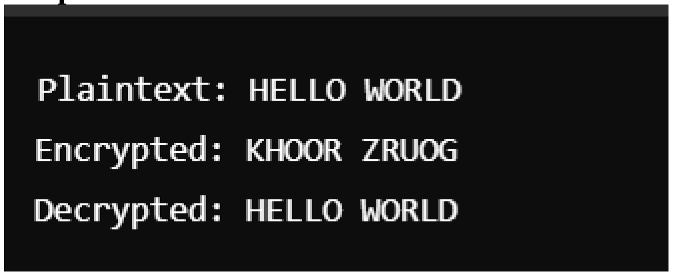
**Algorithm:**

1. Choose a shift value (key).
2. For encryption, shift each letter of the plaintext by the chosen key.
3. For decryption, shift each letter of the ciphertext by the negative of the chosen key.

**Code for Caesar Cipher:**

```
# Caesar Cipher Implementation
def caesar_cipher_encrypt(plaintext, shift):
    encrypted = ''
    for char in plaintext:
        if char.isalpha():
            shift_base = 65 if char.isupper() else 97
            encrypted += chr((ord(char) - shift_base + shift) % 26 + shift_base)
        else:
            encrypted += char
    return encrypted

def caesar_cipher_decrypt(ciphertext, shift):
    return caesar_cipher_encrypt(ciphertext, -shift)

# Example usage
plaintext = "HELLO WORLD"
shift = 3
encrypted_text = caesar_cipher_encrypt(plaintext, shift)
decrypted_text = caesar_cipher_decrypt(encrypted_text, shift)

print(f"Plaintext: {plaintext}")
print(f"Encrypted: {encrypted_text}")
print(f"Decrypted: {decrypted_text}")
```

**Output:**



```
Plaintext: HELLO WORLD
Encrypted: KHOOR ZRUOG
Decrypted: HELLO WORLD
```

## 2. Playfair Cipher

**Description:**

The Playfair cipher is a digraph substitution cipher, meaning it encrypts pairs of letters (digraphs). It uses a 5x5 matrix constructed using a keyword.

**Algorithm:**

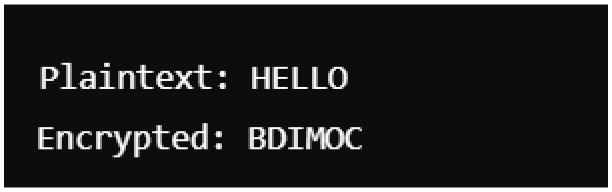
1. Generate a 5x5 matrix using a keyword.
2. Split the plaintext into digraphs (pairs of two letters). If the letters in a pair are the same, insert a filler letter ('X').
3. For each pair, use the Playfair rules to encrypt:
    - If the letters are in the same row, shift them to the right.
    - If the letters are in the same column, shift them down.
    - Otherwise, form a rectangle and swap the corners.

Code for Playfair Cipher:

```
# Playfair Cipher Implementation
def generate_playfair_key_matrix(key):
    matrix = []
    alphabet = "ABCDEFGHIKLMNOPQRSTUVWXYZ"  # 'J' is usually omitted
    key = ''.join(sorted(set(key.upper()), key=lambda x: key.index(x)))
    used_chars = set(key)
    for char in key:
        matrix.append(char)
    for char in alphabet:
        if char not in used_chars:
            matrix.append(char)
    return [matrix[i:i + 5] for i in range(0, 25, 5)]

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

def playfair_encrypt(plaintext, key):
    matrix = generate_playfair_key_matrix(key)
    plaintext = plaintext.upper().replace('J', 'I')
    pairs = []
    i = 0
    while i < len(plaintext):
        a = plaintext[i]
        b = plaintext[i + 1] if i + 1 < len(plaintext) else 'X'
        if a == b:
            b = 'X'
        pairs.append((a, b))
        i += 2

    ciphertext = ""
    for a, b in pairs:
        row_a, col_a = find_position(matrix, a)
        row_b, col_b = find_position(matrix, b)
        if row_a == row_b:
            ciphertext += matrix[row_a][(col_a + 1) % 5] + matrix[row_b][(col_b + 1) % 5]
        elif col_a == col_b:
            ciphertext += matrix[(row_a + 1) % 5][col_a] + matrix[(row_b + 1) % 5][col_b]
        else:
            ciphertext += matrix[row_a][col_b] + matrix[row_b][col_a]

    return ciphertext

# Example usage
plaintext = "HELLO"
```

```
key = "PLAYFAIR"
encrypted_text = playfair_encrypt(plaintext, key)
print(f"Plaintext: {plaintext}")
print(f"Encrypted: {encrypted_text}")
```

Output:



```
Plaintext: HELLO
Encrypted: BDIMOC
```

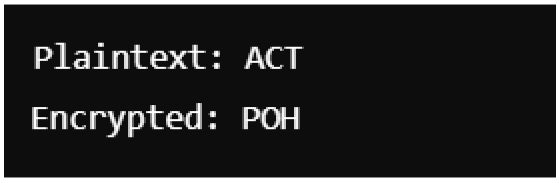
**3. Hill Cipher**
**Description:**
Hill Cipher is a polygraphic substitution cipher based on linear algebra. It uses matrix multiplication to encrypt blocks of letters.
**Algorithm:**

1. Choose a square matrix as the key.
2. Convert the plaintext into vectors and multiply by the key matrix.
3. Mod the result by 26 and convert back to letters.

Code for Hill Cipher:

```
import numpy as np

def hill_cipher_encrypt(plaintext, key_matrix):
    n = len(key_matrix)
    plaintext = plaintext.upper().replace(" ", "")
    if len(plaintext) % n != 0:
        plaintext += 'X' * (n - len(plaintext) % n)

    text_vector = [ord(char) - 65 for char in plaintext]
    encrypted_vector = np.dot(key_matrix, np.array(text_vector).reshape(-1, n).T) % 26
    encrypted_text = ''.join(chr(num + 65) for num in encrypted_vector.flatten())

    return encrypted_text

# Example usage
key_matrix = np.array([[6, 24, 1], [13, 16, 10], [20, 17, 15]])
plaintext = "ACT"
encrypted_text = hill_cipher_encrypt(plaintext, key_matrix)
print(f"Plaintext: {plaintext}")
print(f"Encrypted: {encrypted_text}")
```

Output:



```
Plaintext: ACT
Encrypted: POH
```

## 4. Vigenère Cipher

**Description:**

The Vigenère cipher is a polyalphabetic substitution cipher that uses a keyword to shift each letter by varying amounts.

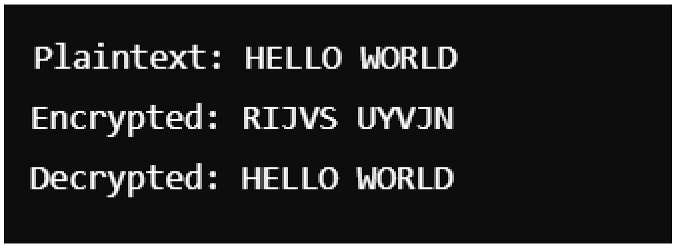
**Algorithm:**

1. Repeat the keyword to match the length of the plaintext.
2. For each letter in the plaintext, shift it by the corresponding letter in the keyword.

**Code for Vigenère Cipher:**

```
# Vigenère Cipher Implementation
def vigenere_cipher_encrypt(plaintext, key):
    key = key.upper()
    key_repeated = (key * (len(plaintext) // len(key))) + key[:len(plaintext) % len(key)]
    encrypted = ''

    for p_char, k_char in zip(plaintext.upper(), key_repeated):
        shift = ord(k_char) - 65
        if p_char.isalpha():
            encrypted += chr((ord(p_char) - 65 + shift) % 26 + 65)
        else:
            encrypted += p_char
    return encrypted

def vigenere_cipher_decrypt(ciphertext, key):
    key = key.upper()
    key_repeated = (key * (len(ciphertext) // len(key))) + key[:len(ciphertext) % len(key)]
    decrypted = ''

    for c_char, k_char in zip(ciphertext.upper(), key_repeated):
        shift = ord(k_char) - 65
        if c_char.isalpha():
            decrypted += chr((ord(c_char) - 65 - shift) % 26 + 65)
        else:
            decrypted += c_char
    return decrypted

# Example usage
plaintext = "HELLO WORLD"
key = "KEY"
encrypted_text = vigenere_cipher_encrypt(plaintext, key)
decrypted_text = vigenere_cipher_decrypt(encrypted_text, key)

print(f"Plaintext: {plaintext}")
print(f"Encrypted: {encrypted_text}")
print(f"Decrypted: {decrypted_text}")
```

Output:



```
Plaintext: HELLO WORLD
Encrypted: RIJVS UYVJN
Decrypted: HELLO WORLD
```

## EXPERIMENT 2:

**AIM: To implement encryption and decryption using the Rail Fence transposition cipher technique.**

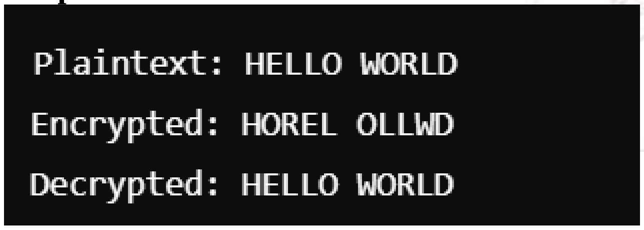
**Description:**
The Rail Fence cipher is a type of transposition cipher where the plaintext is written in a zigzag pattern (rail-like structure) across multiple "rails" and then read row by row to create the ciphertext.
**Algorithm:**

1. **Encryption:**
   - Write the plaintext letters in a zigzag pattern on a given number of rails.
   - Read the letters row by row to form the ciphertext.
2. **Decryption:**
   - Recreate the zigzag pattern based on the number of rails.
   - Fill the zigzag pattern with the ciphertext letters.
   - Read the plaintext by following the zigzag pattern.

**Code for Rail Fence Cipher:**

```
# Rail Fence Cipher Implementation

# Function to encrypt using Rail Fence Cipher
def rail_fence_encrypt(plaintext, key):
    rail = [['\n' for i in range(len(plaintext))] for j in range(key)]
    direction_down = False
    row, col = 0, 0

    for char in plaintext:
        if row == 0 or row == key - 1:
            direction_down = not direction_down
        rail[row][col] = char
        col += 1
        row += 1 if direction_down else -1

    encrypted_text = []
    for i in range(key):
        for j in range(len(plaintext)):
            if rail[i][j] != '\n':
                encrypted_text.append(rail[i][j])
    return ''.join(encrypted_text)

# Function to decrypt using Rail Fence Cipher
def rail_fence_decrypt(ciphertext, key):
    rail = [['\n' for i in range(len(ciphertext))] for j in range(key)]
    direction_down = None
    row, col = 0, 0

    for i in range(len(ciphertext)):
        if row == 0:
            direction_down = True
        if row == key - 1:
            direction_down = False
        rail[row][col] = '*'
        col += 1
        row += 1 if direction_down else -1

    index = 0
    for i in range(key):
        for j in range(len(ciphertext)):
            if rail[i][j] == '*' and index < len(ciphertext):
                rail[i][j] = ciphertext[index]
                index += 1

    decrypted_text = []
    row, col = 0, 0
    for i in range(len(ciphertext)):
        if row == 0:
            direction_down = True
        if row == key - 1:
            direction_down = False
        if rail[row][col] != '\n':
            decrypted_text.append(rail[row][col])
            col += 1
        row += 1 if direction_down else -1
    return ''.join(decrypted_text)

# Example usage
plaintext = "HELLO WORLD"
key = 3
encrypted_text = rail_fence_encrypt(plaintext, key)
decrypted_text = rail_fence_decrypt(encrypted_text, key)

print(f"Plaintext: {plaintext}")
print(f"Encrypted: {encrypted_text}")
print(f"Decrypted: {decrypted_text}")
```

**Output:**



```
Plaintext: HELLO WORLD
Encrypted: HOREL OLLWD
Decrypted: HELLO WORLD
```

## EXPERIMENT 3:

**AIM: Implement Data Encryption Standard (DES) algorithm for Symmetric key encryption..**

**Description:**
The Data Encryption Standard (DES) is a symmetric-key block cipher that was adopted as a standard by the U.S. government in the 1970s. It encrypts data in blocks of 64 bits using a 56-bit key. DES performs 16 rounds of complex operations, including substitution and permutation, based on a series of subkeys derived from the main key.

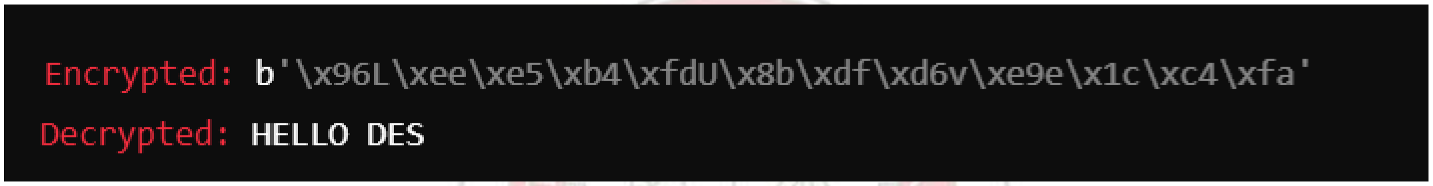
**Algorithm Steps:**

1. Key Generation:
   - Generate 16 subkeys, each of 48 bits, from the 56-bit key using permutation and shift operations.
2. Initial Permutation (IP):
   - Permute the 64-bit plaintext using a fixed permutation table.
3. 16 Rounds of Encryption:
   - Divide the permuted block into two halves (left and right).
   - For each round, perform expansion, substitution, permutation, and XOR operations using one of the subkeys.
   - Swap the halves after each round.
4. Final Permutation (FP):
   - After the 16th round, combine the left and right halves and perform the final permutation to obtain the ciphertext.

**Code for DES Algorithm:**

```
from Crypto.Cipher import DES
from Crypto.Util.Padding import pad, unpad

# Function to encrypt using DES
def des_encrypt(plaintext, key):
    # Create a DES cipher object
    des = DES.new(key, DES.MODE_ECB)

    # Padding the plaintext to be multiple of 8 bytes (block size)
    padded_text = pad(plaintext.encode(), DES.block_size)

    # Encrypt the padded plaintext
    encrypted_text = des.encrypt(padded_text)
    return encrypted_text

# Function to decrypt using DES
def des_decrypt(ciphertext, key):
    # Create a DES cipher object
    des = DES.new(key, DES.MODE_ECB)

    # Decrypt the ciphertext
    decrypted_text = des.decrypt(ciphertext)

    # Unpad the decrypted text
    unpadded_text = unpad(decrypted_text, DES.block_size)
    return unpadded_text.decode()

# Example usage
plaintext = "HELLO DES"
key = b'8bytekey'  # DES uses a 8-byte key (64 bits)

# Encrypt the plaintext
encrypted_text = des_encrypt(plaintext, key)
print(f"Encrypted: {encrypted_text}")

# Decrypt the ciphertext
decrypted_text = des_decrypt(encrypted_text, key)
print(f"Decrypted: {decrypted_text}")
```

**Output:**



```
Encrypted: b'\x96L\xee\xe5\xb4\xfdU\x8b\xdf\xd6v\xe9e\x1c\xc4\xfa'
Decrypted: HELLO DES
```

**EXPERIMENT 4:**

**AIM:Implement Advanced Encryption Standard (AES) algorithm for Symmetric key encryption.**

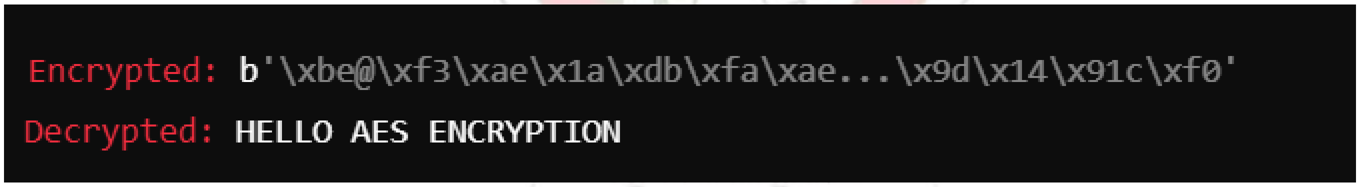
**Description:**
The **Advanced Encryption Standard (AES)** is a symmetric key encryption algorithm that encrypts data in fixed blocks of 128 bits. It supports key sizes of 128, 192, or 256 bits. AES uses substitution and permutation steps, along with multiple rounds of processing (10 rounds for a 128-bit key, 12 for 192-bit, and 14 for 256-bit). It is widely used due to its efficiency and strong security properties.
**Algorithm Steps:**

1. **Key Expansion:**
   - The key is expanded into multiple round keys, which are used for each round of encryption.
2. **Initial Round:**
   - The 128-bit plaintext block undergoes an initial key addition step (AddRoundKey).
3. **Main Rounds (9/11/13 rounds depending on key size):**
   - **SubBytes:** A substitution step where each byte is replaced using an S-box.
   - **ShiftRows:** A transposition step where rows of the state are shifted cyclically.
   - **MixColumns:** A matrix multiplication step for diffusion.
   - **AddRoundKey:** Each byte of the state is XORed with a round key.
4. **Final Round:**
   - Similar to the main rounds but without the MixColumns step.

**Code for AES Algorithm:**
The following implementation requires the pycryptodome library, which provides the AES algorithm.

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes

# Function to encrypt using AES
def aes_encrypt(plaintext, key):
    # Generate a random initialization vector (IV)
    iv = get_random_bytes(16)

    # Create a cipher object using AES (CBC mode) with the provided key and IV
    cipher = AES.new(key, AES.MODE_CBC, iv)

    # Pad the plaintext to a multiple of 16 bytes (block size)
    padded_plaintext = pad(plaintext.encode(), AES.block_size)

    # Encrypt the padded plaintext
    encrypted_text = cipher.encrypt(padded_plaintext)

    # Return the IV + encrypted text (both are required for decryption)
    return iv + encrypted_text

# Function to decrypt using AES
def aes_decrypt(ciphertext, key):
    # Split the IV and the ciphertext
    iv = ciphertext[:16]
    encrypted_text = ciphertext[16:]

    # Create a cipher object using AES (CBC mode) with the provided key and IV
    cipher = AES.new(key, AES.MODE_CBC, iv)

    # Decrypt the ciphertext
    decrypted_text = cipher.decrypt(encrypted_text)

    # Unpad the decrypted text to remove the padding
    unpadded_text = unpad(decrypted_text, AES.block_size)

    return unpadded_text.decode()

# Example usage
plaintext = "HELLO AES ENCRYPTION"
key = get_random_bytes(16)  # AES-128 requires a 16-byte (128-bit) key

# Encrypt the plaintext
encrypted_text = aes_encrypt(plaintext, key)
print(f"Encrypted: {encrypted_text}")

# Decrypt the ciphertext
decrypted_text = aes_decrypt(encrypted_text, key)
print(f"Decrypted: {decrypted_text}")
```

**Output:**



```
Encrypted: b'\xbe@\xf3\xae\x1a\xdb\xfa\xae...\x9d\x14\x91c\xf0'
Decrypted: HELLO AES ENCRYPTION
```

## EXPERIMENT 5:

**AIM: Implement RSA Asymmetric key encryption algorithm.**

**Description:**
The **Rivest–Shamir–Adleman (RSA)** algorithm is an asymmetric encryption technique that uses a pair of keys: a **public key** for encryption and a **private key** for decryption. The security of RSA is based on the difficulty of factoring large prime numbers.

- **Public Key**: Used to encrypt the message.
- **Private Key**: Used to decrypt the message.
- **Key Generation**: Involves generating two large prime numbers, multiplying them to get the modulus, and selecting public and private exponents.

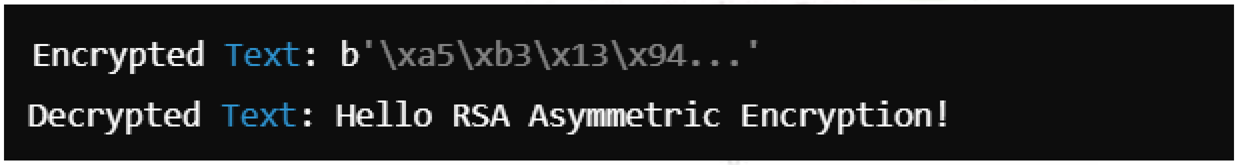
**Steps for RSA Algorithm:**

1. **Key Generation**:
   - Choose two large prime numbers ppp and qqq.
   - Compute n=p×qn = p \times qn=p×q (modulus).
   - Compute ϕ(n)=(p−1)(q−1)\phi(n) = (p-1)(q-1)ϕ(n)=(p−1)(q−1).
   - Choose a public exponent eee such that 1<e<ϕ(n)1 < e < \phi(n)1<e<ϕ(n) and gcd(e,ϕ(n))=1\gcd(e, \phi(n)) = 1gcd(e,ϕ(n))=1.
   - Compute the private key ddd as the modular inverse of eee modulo ϕ(n)\phi(n)ϕ(n), i.e., d×e≡1mod ϕ(n)d \times e \equiv 1 \mod \phi(n)d×e≡1modϕ(n).
2. **Encryption**:
   - The ciphertext CCC is computed as C=Memod nC = M^e \mod nC=Memodn, where MMM is the plaintext message.
3. **Decryption**:
   - The plaintext message MMM is computed as M=Cdmod nM = C^d \mod nM=Cdmodn, where CCC is the ciphertext.

**Code for RSA Algorithm:**
We can use the pycryptodome library to implement RSA encryption and decryption.

```
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
from Crypto.Random import get_random_bytes

# Function to generate RSA key pair
def generate_rsa_keys():
    # Generate a new RSA key pair (public and private)
    key = RSA.generate(2048)  # 2048-bit RSA key
    private_key = key.export_key()
    public_key = key.publickey().export_key()

    return private_key, public_key

# Function to encrypt using RSA public key
def rsa_encrypt(plaintext, public_key):
    # Import the public key and create a cipher object
    public_key = RSA.import_key(public_key)
    cipher = PKCS1_OAEP.new(public_key)
    # Encrypt the plaintext
    encrypted_text = cipher.encrypt(plaintext.encode())

    return encrypted_text

# Function to decrypt using RSA private key
def rsa_decrypt(ciphertext, private_key):
    # Import the private key and create a cipher object
    private_key = RSA.import_key(private_key)
    cipher = PKCS1_OAEP.new(private_key)

    # Decrypt the ciphertext
    decrypted_text = cipher.decrypt(ciphertext)

    return decrypted_text.decode()

# Example usage
plaintext = "Hello RSA Asymmetric Encryption!"

# Generate RSA keys (public and private)
private_key, public_key = generate_rsa_keys()

# Encrypt the plaintext using the public key
encrypted_text = rsa_encrypt(plaintext, public_key)
print(f"Encrypted Text: {encrypted_text}")

# Decrypt the ciphertext using the private key
decrypted_text = rsa_decrypt(encrypted_text, private_key)
print(f"Decrypted Text: {decrypted_text}")
```

**Output:**



```
Encrypted Text: b'\xa5\xb3\x13\x94...'
Decrypted Text: Hello RSA Asymmetric Encryption!
```

**Conclusion:**
The RSA algorithm has been successfully implemented for asymmetric encryption and decryption. RSA uses public and private keys, making it suitable for secure communication where only the recipient can decrypt the message using their private key.