**1. Caesar Cipher**

def caesar\_encrypt(plaintext, shift):

    ciphertext = ""

    for char in plaintext:

        if char.isupper():

            ciphertext += chr((ord(char) + shift - 65) % 26 + 65)

        elif char.islower():

            ciphertext += chr((ord(char) + shift - 97) % 26 + 97)

        else:

            ciphertext += char

    return ciphertext

def caesar\_decrypt(ciphertext, shift):

    return caesar\_encrypt(ciphertext, -shift)

def brute\_force\_caesar(ciphertext):

    print("Brute-force attack results:")

    for shift in range(26):

        decrypted = caesar\_decrypt(ciphertext, shift)

        print(f"Shift {shift:2}: {decrypted}")

def main():

    while True:

        print("\nCaesar Cipher Program")

        print("1. Encrypt")

        print("2. Decrypt")

        print("3. Brute-Force Attack")

        print("4. Exit")

        choice = input("Enter your choice (1-4): ")

        if choice == '1':

            plaintext = input("Enter plaintext: ")

            shift = int(input("Enter shift value (0-25): "))

            encrypted = caesar\_encrypt(plaintext, shift)

            print(f"Encrypted text: {encrypted}")

        elif choice == '2':

            ciphertext = input("Enter ciphertext: ")

            shift = int(input("Enter shift value (0-25): "))

            decrypted = caesar\_decrypt(ciphertext, shift)

            print(f"Decrypted text: {decrypted}")

        elif choice == '3':

            ciphertext = input("Enter ciphertext to brute-force: ")

            brute\_force\_caesar(ciphertext)

        elif choice == '4':

            print("Exiting program...")

            break

        else:

            print("Invalid choice. Please try again.")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

**2. PlayFair Cipher**

def prepare\_input(text):

    # Convert to uppercase and remove non-letters

    text = ''.join(filter(str.isalpha, text.upper()))

    # Replace J with I

    text = text.replace('J', 'I')

    # Create digraphs (pairs)

    i = 0

    digraphs = []

    while i < len(text):

        a = text[i]

        b = ''

        if i + 1 < len(text):

            b = text[i + 1]

            if a == b:

                b = 'X'

                i += 1

            else:

                i += 2

        else:

            b = 'X'

            i += 1

        digraphs.append(a + b)

    return digraphs

def create\_matrix(key):

    # Remove duplicates and replace J with I

    key = key.upper().replace('J', 'I')

    seen = set()

    matrix = []

    for char in key:

        if char not in seen and char.isalpha():

            seen.add(char)

            matrix.append(char)

    for char in 'ABCDEFGHIKLMNOPQRSTUVWXYZ':

        if char not in seen:

            seen.add(char)

            matrix.append(char)

    return [matrix[i\*5:(i+1)\*5] for i in range(5)]

def find\_position(matrix, letter):

    for i in range(5):

        for j in range(5):

            if matrix[i][j] == letter:

                return i, j

    return None

def encrypt\_pair(pair, matrix):

    a\_row, a\_col = find\_position(matrix, pair[0])

    b\_row, b\_col = find\_position(matrix, pair[1])

    if a\_row == b\_row:

        return matrix[a\_row][(a\_col + 1) % 5] + matrix[b\_row][(b\_col + 1) % 5]

    elif a\_col == b\_col:

        return matrix[(a\_row + 1) % 5][a\_col] + matrix[(b\_row + 1) % 5][b\_col]

    else:

        return matrix[a\_row][b\_col] + matrix[b\_row][a\_col]

def decrypt\_pair(pair, matrix):

    a\_row, a\_col = find\_position(matrix, pair[0])

    b\_row, b\_col = find\_position(matrix, pair[1])

    if a\_row == b\_row:

        return matrix[a\_row][(a\_col - 1) % 5] + matrix[b\_row][(b\_col - 1) % 5]

    elif a\_col == b\_col:

        return matrix[(a\_row - 1) % 5][a\_col] + matrix[(b\_row - 1) % 5][b\_col]

    else:

        return matrix[a\_row][b\_col] + matrix[b\_row][a\_col]

def playfair\_encrypt(plaintext, key):

    matrix = create\_matrix(key)

    digraphs = prepare\_input(plaintext)

    return ''.join([encrypt\_pair(pair, matrix) for pair in digraphs])

def playfair\_decrypt(ciphertext, key):

    matrix = create\_matrix(key)

    digraphs = prepare\_input(ciphertext)

    return ''.join([decrypt\_pair(pair, matrix) for pair in digraphs])

# ---------- Example Usage ----------

if \_\_name\_\_ == "\_\_main\_\_":

    key = input("Enter the key: ")

    text = input("Enter the plaintext: ")

    encrypted = playfair\_encrypt(text, key)

    print(f"Encrypted: {encrypted}")

    decrypted = playfair\_decrypt(encrypted, key)

    print(f"Decrypted: {decrypted}")

**3. RSA Algorithm**

import random

# ---------- Extended Euclidean Algorithm ----------

def extended\_gcd(a, b):

    if b == 0:

        return a, 1, 0

    gcd, x1, y1 = extended\_gcd(b, a % b)

    x = y1

    y = x1 - (a // b) \* y1

    return gcd, x, y

# ---------- Modular Inverse ----------

def mod\_inverse(e, phi):

    gcd, x, \_ = extended\_gcd(e, phi)

    if gcd != 1:

        raise Exception("Modular inverse doesn't exist.")

    else:

        return x % phi

# ---------- Check for Primality (simple) ----------

def is\_prime(n):

    if n <= 1:

        return False

    for i in range(2, int(n\*\*0.5)+1):

        if n % i == 0:

            return False

    return True

# ---------- Generate Public and Private Keys ----------

def generate\_keys():

    # Choose two distinct prime numbers p and q

    while True:

        p = random.randint(50, 100)

        q = random.randint(50, 100)

        if is\_prime(p) and is\_prime(q) and p != q:

            break

    n = p \* q

    phi = (p - 1) \* (q - 1)

    # Choose public exponent e such that 1 < e < phi and gcd(e, phi) = 1

    e = 3

    while True:

        if extended\_gcd(e, phi)[0] == 1:

            break

        e += 2

    # Compute private key d

    d = mod\_inverse(e, phi)

    return (e, n), (d, n)

# ---------- Encryption ----------

def encrypt(message, public\_key):

    e, n = public\_key

    cipher = [pow(ord(char), e, n) for char in message]

    return cipher

# ---------- Decryption ----------

def decrypt(cipher, private\_key):

    d, n = private\_key

    decrypted = ''.join([chr(pow(char, d, n)) for char in cipher])

    return decrypted

# ---------- File Operations ----------

def read\_file(filename):

    with open(filename, 'r') as file:

        return file.read()

def write\_file(filename, content):

    with open(filename, 'w') as file:

        file.write(str(content))

# ---------- Main Function ----------

if \_\_name\_\_ == "\_\_main\_\_":

    public\_key, private\_key = generate\_keys()

    print(f"Public Key (e, n): {public\_key}")

    print(f"Private Key (d, n): {private\_key}")

    # Read plaintext from file

    input\_file = "plaintext.txt"

    message = read\_file(input\_file)

    print(f"Original message from file: {message}")

    # Encrypt and save to file

    cipher = encrypt(message, public\_key)

    write\_file("encrypted.txt", cipher)

    print("Encrypted message saved to encrypted.txt")

    # Decrypt and save to file

    decrypted\_message = decrypt(cipher, private\_key)

    write\_file("decrypted.txt", decrypted\_message)

    print("Decrypted message saved to decrypted.txt")

    # Print results

    print(f"Encrypted data: {cipher}")

    print(f"Decrypted message: {decrypted\_message}")

**4. Diffie Hellman**

import random

def power\_mod(base, exponent, mod):

    """Efficient modular exponentiation"""

    result = 1

    base = base % mod

    while exponent > 0:

        if exponent % 2 == 1:

            result = (result \* base) % mod

        exponent = exponent >> 1

        base = (base \* base) % mod

    return result

def diffie\_hellman(p, g, private\_a, private\_b):

    # Compute public values

    public\_a = power\_mod(g, private\_a, p)

    public\_b = power\_mod(g, private\_b, p)

    # Compute shared secret

    shared\_secret\_a = power\_mod(public\_b, private\_a, p)

    shared\_secret\_b = power\_mod(public\_a, private\_b, p)

    return public\_a, public\_b, shared\_secret\_a, shared\_secret\_b

# Example usage

if \_\_name\_\_ == "\_main\_":

    # Prime modulus and primitive root

    p = int(input("Enter a prime number (p): "))

    g = int(input("Enter a primitive root modulo p (g): "))

    # Private keys (randomly chosen)

    private\_a = random.randint(2, p - 2)

    private\_b = random.randint(2, p - 2)

    print(f"Alice's Private Key: {private\_a}")

    print(f"Bob's Private Key: {private\_b}")

    public\_a, public\_b, secret\_a, secret\_b = diffie\_hellman(p, g, private\_a, private\_b)

    print(f"Alice's Public Key: {public\_a}")

    print(f"Bob's Public Key: {public\_b}")

    print(f"Alice's Shared Secret: {secret\_a}")

    print(f"Bob's Shared Secret: {secret\_b}")

    if secret\_a == secret\_b:

        print("✅ Shared secret successfully established.")

    else:

        print("❌ Shared secret mismatch.")