1. **Design and implement a product cipher using substitution and transposition ciphers.**

def caesar\_encrypt(text, shift):

result = ""

for char in text:

if char.isalpha():

base = ord('A') if char.isupper() else ord('a')

result += chr((ord(char) - base + shift) % 26 + base)

else:

result += char

return result

def caesar\_decrypt(cipher, shift):

return caesar\_encrypt(cipher, -shift)

def rail\_fence\_encrypt(text, rails):

fence = [['\n' for \_ in range(len(text))] for \_ in range(rails)]

row, direction = 0, False

for i, char in enumerate(text):

fence[row][i] = char

if row == 0 or row == rails - 1:

direction = not direction

row += 1 if direction else -1

result = ''

for r in range(rails):

for c in range(len(text)):

if fence[r][c] != '\n':

result += fence[r][c]

return result

def rail\_fence\_decrypt(cipher, rails):

fence = [['\n' for \_ in range(len(cipher))] for \_ in range(rails)]

row, direction = 0, False

for i in range(len(cipher)):

fence[row][i] = '\*'

if row == 0 or row == rails - 1:

direction = not direction

row += 1 if direction else -1

index = 0

for r in range(rails):

for c in range(len(cipher)):

if fence[r][c] == '\*' and index < len(cipher):

fence[r][c] = cipher[index]

index += 1

result = ''

row, direction = 0, False

for i in range(len(cipher)):

result += fence[row][i]

if row == 0 or row == rails - 1:

direction = not direction

row += 1 if direction else -1

return result

def product\_cipher\_encrypt(text, shift, rails):

text = text.replace(" ", "")

substituted = caesar\_encrypt(text, shift)

encrypted = rail\_fence\_encrypt(substituted, rails)

return encrypted

def product\_cipher\_decrypt(cipher, shift, rails):

transposed = rail\_fence\_decrypt(cipher, rails)

decrypted = caesar\_decrypt(transposed, shift)

return decrypted

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

mode = input("Enter mode ('encrypt' or 'decrypt'): ").lower()

message = input("Enter message: ")

shift = int(input("Enter Caesar cipher shift value: "))

rails = int(input("Enter number of Rail Fence rails: "))

if mode == "encrypt":

result = product\_cipher\_encrypt(message, shift, rails)

elif mode == "decrypt":

result = product\_cipher\_decrypt(message, shift, rails)

else:

result = "Invalid mode!"

print(f"\nResult: {result}")

**OUTPUT**

Enter mode ('encrypt' or 'decrypt'): encrypt

Enter message: hello world

Enter Caesar cipher shift value: 3

Enter number of Rail Fence rails: 4

Result: krhzuoroog

Enter mode ('encrypt' or 'decrypt'): decrypt

Enter message: krhzuoroog

Enter Caesar cipher shift value: 3

Enter number of Rail Fence rails: 4

Result: helloworld

1. **Implement encryption and decryption of the affine cipher.**

from math import gcd

# Compute modular inverse using Extended Euclidean Algorithm

def mod\_inverse(a, m):

a = a % m

for x in range(1, m):

if (a \* x) % m == 1:

return x

return None

# Encrypt using Affine Cipher

def affine\_encrypt(text, a, b):

if gcd(a, 26) != 1:

raise ValueError("Key 'a' must be coprime with 26.")

result = ""

for char in text:

if char.isalpha():

base = ord('A') if char.isupper() else ord('a')

x = ord(char) - base

encrypted = (a \* x + b) % 26

result += chr(encrypted + base)

else:

result += char

return result

# Decrypt using Affine Cipher

def affine\_decrypt(cipher, a, b):

a\_inv = mod\_inverse(a, 26)

if a\_inv is None:

raise ValueError(f"Modular inverse of {a} does not exist. Decryption impossible.")

result = ""

for char in cipher:

if char.isalpha():

base = ord('A') if char.isupper() else ord('a')

y = ord(char) - base

decrypted = (a\_inv \* (y - b)) % 26

result += chr(decrypted + base)

else:

result += char

return result

# Main program

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

mode = input("Enter mode ('encrypt' or 'decrypt'): ").lower()

message = input("Enter message: ")

a = int(input("Enter key 'a' (must be coprime to 26): "))

b = int(input("Enter key 'b': "))

if mode == "encrypt":

result = affine\_encrypt(message, a, b)

elif mode == "decrypt":

result = affine\_decrypt(message, a, b)

else:

result = "Invalid mode!"

print(f"\nResult: {result}")

**OUTPUT**

PS S:\Cryptographics> py .\affine-cipher.py

Enter mode ('encrypt' or 'decrypt'): encrypt

Enter message: Vulnerability

Enter key 'a' (must be coprime to 26): 5

Enter key 'b': 4

Result: Fahrylejshsvu

PS S:\Cryptographics> py .\affine-cipher.py

Enter mode ('encrypt' or 'decrypt'): decrypt

Enter message: Fahrylejshsvu

Enter key 'a' (must be coprime to 26): 5

Enter key 'b': 4

Result: Vulnerability

1. **Implement Diffie-Hellman Key Exchange Algorithm.**

def mod\_exp(base, exp, mod):

res = 1

base %= mod

while exp:

if exp % 2:

res = res \* base % mod

base = base \* base % mod

exp //= 2

return res

def diffie\_hellman(p, g, a, b):

A, B = mod\_exp(g, a, p), mod\_exp(g, b, p)

s1, s2 = mod\_exp(B, a, p), mod\_exp(A, b, p)

return A, B, s1, s2

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

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

g = int(input("Enter primitive root g: "))

a = int(input("Enter Alice's private key: "))

b = int(input("Enter Bob's private key: "))

A, B, s1, s2 = diffie\_hellman(p, g, a, b)

print("\n--- Key Exchange ---")

print(f"Alice's Public Key: {A}")

print(f"Bob's Public Key: {B}")

print(f"Alice's Shared Secret: {s1}")

print(f"Bob's Shared Secret: {s2}")

if s1 == s2:

print("Shared secret key successfully established!")

else:

print("Error: Shared secrets do not match.")

**OUTPUT**

Enter prime p: 103

Enter primitive root g: 3

Enter Alice's private key: 6

Enter Bob's private key: 2

--- Key Exchange ---

Alice's Public Key: 8

Bob's Public Key: 9

Alice's Shared Secret: 64

Bob's Shared Secret: 64

Shared secret key successfully established!

1. **Implement RSA Public Key Cryptosystem.**

import random

from math import gcd

*# Generate a small prime number (for demo only; use large primes for real use)*

def is\_prime(n):

    if n <= 1:

        return False

    if n <= 3:

        return True

    if n % 2 == 0 or n % 3 == 0:

        return False

    i = 5

    while i \* i <= n:

        if n % i == 0 or n % (i + 2) == 0:

            return False

        i += 6

    return True

def generate\_prime(min\_val=100, max\_val=300):

    while True:

        p = random.randint(min\_val, max\_val)

        if is\_prime(p):

            return p

*# Extended Euclidean Algorithm to find modular inverse*

def mod\_inverse(e, phi):

    def egcd(a, b):

        if a == 0:

            return (b, 0, 1)

        g, y, x = egcd(b % a, a)

        return (g, x - (b // a) \* y, y)

    g, x, \_ = egcd(e, phi)

    if g != 1:

        raise Exception('Modular inverse does not exist')

    return x % phi

*# RSA Key Generation*

def generate\_keys():

    p = generate\_prime()

    q = generate\_prime()

    while q == p:

        q = generate\_prime()

    n = p \* q

    phi = (p - 1) \* (q - 1)

    e = random.randrange(2, phi)

    while gcd(e, phi) != 1:

        e = random.randrange(2, phi)

    d = mod\_inverse(e, phi)

    return (e, n), (d, n)

*# Encryption*

def encrypt(plaintext, public\_key):

    e, n = public\_key

    ciphertext = [pow(ord(char), e, n) for char in plaintext]

    return ciphertext

*# Decryption*

def decrypt(ciphertext, private\_key):

    d, n = private\_key

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

    return plaintext

*# Main*

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

    print("RSA Key Generation")

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

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

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

    message = input("\nEnter a message to encrypt: ")

    encrypted = encrypt(message, public\_key)

    print(f"\nEncrypted: {encrypted}")

    decrypted = decrypt(encrypted, private\_key)

    print(f"Decrypted: {decrypted}")

**OUTPUT**

RSA Key Generation

Public Key (e, n): (51747, 64291)

Private Key (d, n): (61675, 64291)

Enter a message to encrypt: RSAIsGodPlayer

Encrypted: [20631, 10307, 41105, 41311, 25727, 16114, 30620, 58818, 53113, 27601, 26838, 50324, 12260, 16650]

Decrypted: RSAIsGodPlayer

1. **WAP to encrypt a message using a given P-box.**

def pbox\_encrypt(message, pbox):

    size = len(pbox)

*# Pad message to be a multiple of block size*

    pad\_len = (size - len(message) % size) % size

    message += ' ' \* pad\_len

    encrypted = ''

    for i in range(0, len(message), size):

        block = message[i:i + size]

        encrypted += ''.join(block[pbox[j]] for j in range(size))

    return encrypted

def pbox\_decrypt(ciphertext, pbox):

    size = len(pbox)

    inverse = [0] \* size

    for i, pos in enumerate(pbox):

        inverse[pos] = i

    decrypted = ''

    for i in range(0, len(ciphertext), size):

        block = ciphertext[i:i + size]

        decrypted += ''.join(block[inverse[j]] for j in range(size))

    return decrypted

*# --- Main Program ---*

message = input("Enter the message: ")

pbox = list(map(int, input("Enter P-box (0-based, space-separated): ").split()))

encrypted = pbox\_encrypt(message, pbox)

decrypted = pbox\_decrypt(encrypted, pbox)

print(f"\nEncrypted: '{encrypted}'")

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

**OUTPUT**

Enter the message: Information Security

Enter P-box (0-based, space-separated): 4 2 3 1 0

Encrypted: 'rfonIotiamcSe nyitru'

Decrypted: 'Information Security'