Renuga Devi.B

192311111

Exp 5

```
def rail_fence_encrypt(text, rails):
    fence = [''] * rails
    rail, dir = 0, 1
    for char in text:
        fence[rail] += char
        rail += dir
        if rail == 0 or rail == rails - 1: dir *= -1
    return ''.join(fence)

print("Rail Fence Encrypted:", rail_fence_encrypt("HELLOWORLD", 3))
def row_col_encrypt(text, key):
    col = len(key)
    text += 'X' * ((col - len(text) % col) % col)
    rows = [text[i:i+col] for i in range(0, len(text), col)]
    order = sorted(range(col), key=lambda k: key[k])
    return ''.join(''.join(row[i] for row in rows) for i in order)

print("Row-Column Encrypted:", row_col_encrypt("HELLOWORLD", "4312"))
```

Exp 6

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

key = b'8bytekey'              # 8 bytes key
text = b'HELLO123'             # Plaintext (must be bytes)
```

```
cipher = DES.new(key, DES.MODE_ECB)

encrypted = cipher.encrypt(pad(text, 8))
print("Encrypted:", encrypted)

decipher = DES.new(key, DES.MODE_ECB)
decrypted = unpad(decipher.decrypt(encrypted), 8)
print("Decrypted:", decrypted)
```

Exp 7

```
def des_encrypt(text, key):
    encrypted = ""
    for i in range(len(text)):
        encrypted += chr(ord(text[i]) ^ ord(key[i % len(key)]))  # XOR encryption
    return encrypted

def des_decrypt(encrypted, key):
    return des_encrypt(encrypted, key)  # XOR is reversible

# Example
plaintext = "HELLODES"
key = "8bytekey"

encrypted = des_encrypt(plaintext, key)
print("Encrypted (hex):", ''.join(f'{ord(c):02x}' for c in encrypted))  # Display in hex

decrypted = des_decrypt(encrypted, key)
```

```
print("Decrypted:", decrypted)
```

Exp 8

```
# Prime number and primitive root
p = 23
g = 5

# Private keys (chosen secretly)
a = 6  # Alice
b = 15 # Bob

# Public keys (shared openly)
A = pow(g, a, p)  # Alice's public key
B = pow(g, b, p)  # Bob's public key

# Shared secret (calculated independently)
secret_a = pow(B, a, p)
secret_b = pow(A, b, p)

# Output
print("Shared Secret (Alice):", secret_a)
print("Shared Secret (Bob):", secret_b)
```

Exp 9

```
import hashlib


def generate_md5_hash(text):
```

```
    hash_object = hashlib.md5(text.encode())
    md5_hash = hash_object.hexdigest()
    return md5_hash

message = "HelloWorld"
hashed = generate_md5_hash(message)

print("Original Message:", message)
print("MD5 Hash:", hashed)
```

Exp 10

```
import hashlib


def generate_sha1_hash(text):

    hash_object = hashlib.sha1(text.encode())
    sha1_hash = hash_object.hexdigest()
    return sha1_hash

# Example usage
message = "HelloWorld"
hashed = generate_sha1_hash(message)

print("Original Message:", message)
print("SHA-1 Hash:", hashed)
```

Exp 11

```
from Crypto.Cipher import DES3
```

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

# Key and IV
key = DES3.adjust_key_parity(get_random_bytes(24))  # 3DES requires a 24-byte key
iv = get_random_bytes(8)

# Data to encrypt
data = b"Encrypt this message using 3DES CBC!"

# Encrypt
cipher_encrypt = DES3.new(key, DES3.MODE_CBC, iv)
ciphertext = cipher_encrypt.encrypt(pad(data, DES3.block_size))

# Decrypt
cipher_decrypt = DES3.new(key, DES3.MODE_CBC, iv)
plaintext = unpad(cipher_decrypt.decrypt(ciphertext), DES3.block_size)

print("Ciphertext:", ciphertext.hex())
print("Decrypted:", plaintext.decode())
```

Exp 12

```
def gcd_ext(a, b):
    if b == 0: return a, 1, 0
    g, x1, y1 = gcd_ext(b, a % b)
    return g, y1, x1 - (a // b) * y1

def modinv(e, phi):
```

```
    g, x, _ = gcd_ext(e, phi)
    return x % phi if g == 1 else None


def find_factors(n):
    for i in range(2, int(n**0.5)+1):
        if n % i == 0:
            return i, n // i


# Given
e, n = 31, 3599
p, q = find_factors(n)
phi = (p - 1) * (q - 1)
d = modinv(e, phi)


print(f"Public Key (e, n): ({e}, {n})")
print(f"Private Key d: {d}")
print(f"Factors p, q: {p}, {q}")
print(f"φ(n): {phi}")
```

Exp 13

```
import math


# Given public key
n = 3599  # n = p * q
e = 31


# Let's say someone gives a plaintext block m
m = 177  # Assume this shares a factor with n
```

```
# Try to factor n using GCD
g = math.gcd(m, n)

if 1 < g < n:
    p = g
    q = n // g
    phi = (p - 1) * (q - 1)

    # Compute private key d
    def modinv(a, m):
        def egcd(a, b):
            if b == 0: return a, 1, 0
            g, y, x = egcd(b, a % b)
            return g, x, y - (a // b) * x
        g, x, _ = egcd(a, m)
        return x % m if g == 1 else None

    d = modinv(e, phi)

    print(f"Found p = {p}, q = {q}")
    print(f"Private key d = {d}")
else:
    print("No common factor found. RSA still secure.")
```

Exp 14

```
import math
```

```
# Given public key
n = 3599  # n = p * q
e = 31

# Let's say someone gives a plaintext block m
m = 177  # Assume this shares a factor with n

# Try to factor n using GCD
g = math.gcd(m, n)

if 1 < g < n:
    p = g
    q = n // g
    phi = (p - 1) * (q - 1)

    # Compute private key d
    def modinv(a, m):
        def egcd(a, b):
            if b == 0: return a, 1, 0
            g, y, x = egcd(b, a % b)
            return g, x, y - (a // b) * x
        g, x, _ = egcd(a, m)
        return x % m if g == 1 else None

    d = modinv(e, phi)

    print(f"Found p = {p}, q = {q}")
```

```
    print(f"Private key d = {d}")
else:
    print("No common factor found. RSA still secure.")
```

Exp 15

```
def encrypt(m, e, n):
    return pow(m, e, n)

# Simulate known RSA public key
e = 17
n = 3233  # Large enough to seem secure

# Build lookup table for A-Z (0–25)
lookup = {encrypt(m, e, n): chr(m + ord('A')) for m in range(26)}

# Intercepted ciphertexts (simulate Alice’s encrypted message)
ciphertext_blocks = [encrypt(ord(c) - ord('A'), e, n) for c in "HELLO"]

# Attacker decrypts using lookup table
decrypted = ''.join(lookup[c] for c in ciphertext_blocks)
print("Decrypted:", decrypted)
```

Exp 16

```
# Public values
a = 5  # primitive root mod q
q = 23  # prime modulus

# Alice and Bob's secret values
alice_secret = 6
```

```
bob_secret = 15

# Exchange values
alice_public = pow(a, alice_secret, q)
bob_public = pow(a, bob_secret, q)

# Shared key
alice_key = pow(bob_public, alice_secret, q)
bob_key = pow(alice_public, bob_secret, q)

print("Shared key (Alice):", alice_key)
print("Shared key (Bob):", bob_key)
```

Exp 17

```
import random

def simulate_sha3():
    state = [[0] * 64 for _ in range(25)]
    rate_lanes = 12  # 50% capacity, 50% rate (25 lanes total)
    for i in range(rate_lanes, 25):
        state[i] = [random.choice([0, 1]) for _ in range(64)]

    steps = 0
    while not all(any(bit == 1 for bit in lane) for lane in state[:rate_lanes]):
        steps += 1
        for i in range(rate_lanes):  # Flip random bits in capacity lanes
            state[i][random.randint(0, 63)] = 1
```

```
    return steps


steps_needed = simulate_sha3()
print(steps_needed)
```

Exp 18

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


def cbc_mac(key, message):
    cipher = AES.new(key, AES.MODE_CBC, iv=b'\x00' * 16)
    padded_message = pad(message.encode(), AES.block_size)
    mac = cipher.encrypt(padded_message)[-AES.block_size:]
    return mac


# Key and message X
key = b'Sixteen byte key'
X = "Hello1234"


# CBC MAC for one-block message X
T = cbc_mac(key, X)
print("MAC for X:", T.hex())


# Adversary computes MAC for X || (X ⊕ T)
X_xor_T = ''.join(chr(ord(a) ^ ord(b)) for a, b in zip(X, T.decode()[:len(X)]))
message = X + X_xor_T
mac_xt = cbc_mac(key, message)
```

```
print("MAC for X || (X ⊕ T):", mac_xt.hex())
```

Exp 19

```
from cryptography.hazmat.primitives.asymmetric import dsa
from cryptography.hazmat.primitives import hashes, serialization

# Generate private key
private_key = dsa.generate_private_key(key_size=1024)

# Sign the same message twice
message = b"Hello, DSA"
signature1 = private_key.sign(message, hashes.SHA256())
signature2 = private_key.sign(message, hashes.SHA256())

print("Signature 1:", signature1.hex())
print("Signature 2:", signature2.hex())
print("Are signatures different?", signature1 != signature2)
```

Exp 20

```
from collections import Counter
import string

# Standard English letter frequency (approx.)
ENGLISH_FREQ = "ETAOINSHRDLCUMWFGYPBVKJXQZ"

# Function to score text based on frequency match
def score(text):
    freq = Counter(c for c in text.upper() if c.isalpha())
```

```
    most_common = ''.join([pair[0] for pair in freq.most_common()])
    return sum([ENGLISH_FREQ.index(c) if c in ENGLISH_FREQ else 26 for c in most_common[:6]])


# Frequency attack
def frequency_attack(ciphertext, top_n=10):
    cipher_freq = Counter(c for c in ciphertext.upper() if c.isalpha())
    cipher_letters = [pair[0] for pair in cipher_freq.most_common()]
    guesses = []

    for i in range(top_n):
        mapping = dict(zip(cipher_letters, ENGLISH_FREQ[i:] + ENGLISH_FREQ[:i]))
        plaintext = ''.join([mapping.get(c.upper(), c) for c in ciphertext])
        guesses.append((plaintext, score(plaintext)))

    guesses.sort(key=lambda x: x[1])  # Lower score = better match
    return [text for text, _ in guesses]

# Example usage
ciphertext = "GSRH RH Z HVXVGRLM ULI ZMW"
top_plaintexts = frequency_attack(ciphertext, top_n=10)

print("\nTop 10 Possible Plaintexts:")
for i, text in enumerate(top_plaintexts, 1):
    print(f"{i}. {text}")
```

Exp 21

```
from math import gcd
```

```
# Encryption: C = (a * p + b) % 26
def encrypt(text, a, b):
    if gcd(a, 26) != 1:
        raise ValueError("Invalid 'a': gcd(a, 26) must be 1 for one-to-one mapping.")
    return ''.join([chr((a * (ord(c) - 65) + b) % 26 + 65) if c.isalpha() else c for c in text.upper()])

# Modular inverse of a modulo 26
def modinv(a):
    for i in range(1, 26):
        if (a * i) % 26 == 1:
            return i
    raise ValueError("No modular inverse for given 'a'.")

# Decryption: P = a_inv * (C - b) % 26
def decrypt(cipher, a, b):
    a_inv = modinv(a)
    return ''.join([chr((a_inv * ((ord(c) - 65) - b)) % 26 + 65) if c.isalpha() else c for c in cipher.upper()])

# Example usage
a, b = 5, 8  # a must be coprime with 26
plain_text = "HELLO"
cipher_text = encrypt(plain_text, a, b)
decrypted_text = decrypt(cipher_text, a, b)

print("Plaintext:", plain_text)
```

```
print("Ciphertext:", cipher_text)
print("Decrypted:", decrypted_text)
```

Exp 22

```
from cryptography.hazmat.primitives.asymmetric import dsa
from cryptography.hazmat.primitives import hashes, serialization

# Generate private key
private_key = dsa.generate_private_key(key_size=1024)

# Sign the same message twice
message = b"Hello, DSA"
signature1 = private_key.sign(message, hashes.SHA256())
signature2 = private_key.sign(message, hashes.SHA256())

print("Signature 1:", signature1.hex())
print("Signature 2:", signature2.hex())
print("Are signatures different?", signature1 != signature2)
```

Exp 23

```
from collections import Counter

# English letter frequency (most to least common)
ENGLISH_FREQ = "ETAOINSHRDLCUMWFGYPBVKJXQZ"

def frequency_attack(ciphertext, top_n=10):
    ciphertext = ciphertext.upper()
    cipher_freq = [c for c, _ in Counter(filter(str.isalpha, ciphertext)).most_common()]
    results = []
```

```python
    for i in range(top_n):
        guess_map = dict(zip(cipher_freq, ENGLISH_FREQ[i:] + ENGLISH_FREQ[:i]))
        guess = ''.join(guess_map.get(c, c) for c in ciphertext)
        results.append(guess)
    return results

# UI
ciphertext = input("Enter ciphertext: ")
top_n = int(input("Top how many plaintexts to display? "))

print("\nTop guesses:")
for i, guess in enumerate(frequency_attack(ciphertext, top_n), 1):
    print(f"{i}. {guess}")
```

Exp 24

```python
import math

# Number of unique letters in Playfair (I/J merged)
n = 25

# Number of possible keys = 25! (all permutations of the 25 letters)
keyspace = math.factorial(n)

# Convert to power of 2: log2(25!)
approx_power_of_2 = math.log2(keyspace)

print(f"Possible keys ≈ 2^{approx_power_of_2:.2f}")
```

Exp 25

```
import numpy as np
from sympy import Matrix

# Convert letter to number (A=0,...Z=25)
def text_to_nums(text):
    return [ord(c) - ord('A') for c in text.upper() if c.isalpha()]

# Build matrix from pairs
def build_matrix(pairs):
    return np.array(pairs).reshape(2, 2).T

# Inverse mod 26 using sympy
def mod26_inv(matrix):
    return Matrix(matrix).inv_mod(26)

# Known plaintext-ciphertext pairs
plaintext = "HELP"
ciphertext = "ZEBB"

P = build_matrix(text_to_nums(plaintext))     # 2x2 plaintext matrix
C = build_matrix(text_to_nums(ciphertext))    # 2x2 ciphertext matrix

# Solve for key: K = C * P_inv mod 26
P_inv = mod26_inv(P)
K = (Matrix(C) * P_inv) % 26

print("Recovered Hill Cipher Key Matrix:")
```

```
print(np.array(K).astype(int))
```