**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.")

**5. Mono Alphabetic**

import random

from collections import Counter

import string

class MonoalphabeticCipher:

    def \_\_init\_\_(self):

        self.key = self.generate\_key()

        self.reverse\_key = {v: k for k, v in self.key.items()}

    def generate\_key(self):

        """Generate random substitution key"""

        letters = list(string.ascii\_lowercase)

        shuffled = letters.copy()

        random.shuffle(shuffled)

        return dict(zip(letters, shuffled))

    def encrypt(self, plaintext):

        """Encrypt plaintext using substitution cipher"""

        ciphertext = []

        for char in plaintext.lower():

            if char in self.key:

                ciphertext.append(self.key[char])

            else:

                ciphertext.append(char)  # Keep non-alphabet characters

        return ''.join(ciphertext)

    def decrypt\_with\_key(self, ciphertext):

        """Decrypt ciphertext using known key"""

        plaintext = []

        for char in ciphertext.lower():

            if char in self.reverse\_key:

                plaintext.append(self.reverse\_key[char])

            else:

                plaintext.append(char)

        return ''.join(plaintext)

    @staticmethod

    def frequency\_attack(ciphertext, lang='english'):

        """Improved frequency analysis attack"""

        # Enhanced frequency tables

        freq\_tables = {

            'english': {

                'letter\_freq': [

                    ('e', 12.7), ('t', 9.1), ('a', 8.2), ('o', 7.5), ('i', 7.0),

                    ('n', 6.7), ('s', 6.3), ('h', 6.1), ('r', 6.0), ('d', 4.3),

                    ('l', 4.0), ('c', 2.8), ('u', 2.8), ('m', 2.4), ('w', 2.4),

                    ('f', 2.2), ('g', 2.0), ('y', 2.0), ('p', 1.9), ('b', 1.5),

                    ('v', 1.0), ('k', 0.8), ('j', 0.2), ('x', 0.2), ('q', 0.1),

                    ('z', 0.1)

                ],

                'digrams': [

                    'th', 'he', 'in', 'er', 'an', 're', 'nd', 'at', 'on', 'nt',

                    'ha', 'es', 'st', 'en', 'ed', 'to', 'it', 'ou', 'ea', 'hi',

                    'is', 'or', 'ti', 'as', 'te', 'et', 'ng', 'of', 'al', 'de',

                    'se', 'le', 'sa', 'si', 'ar', 've', 'ra', 'ld', 'ur'

                ],

                'trigrams': [

                    'the', 'and', 'ing', 'ion', 'ent', 'her', 'for', 'tha', 'nth',

                    'int', 'ere', 'tio', 'ter', 'est', 'ers', 'ati', 'hat', 'ate',

                    'all', 'eth', 'hes', 'ver', 'his', 'oft', 'ith', 'fth', 'sth',

                    'oth', 'res', 'ont'

                ]

            }

        }

        # Get letter frequencies in ciphertext

        letters = [c for c in ciphertext.lower() if c.isalpha()]

        if not letters:

            return "No alphabetic characters found", {}

        freq = Counter(letters)

        total\_letters = len(letters)

        # Get most common letters in ciphertext (sorted by frequency)

        cipher\_freq = [item[0] for item in freq.most\_common()]

        # Create initial mapping (cipher -> plain)

        mapping = {}

        std\_letters = [item[0] for item in freq\_tables[lang]['letter\_freq']]

        # Map most frequent cipher letters to most frequent English letters

        for i in range(min(len(cipher\_freq), len(std\_letters))):

            mapping[cipher\_freq[i]] = std\_letters[i]

        # Improve mapping using digrams

        digrams = Counter(ciphertext[i:i+2].lower() for i in range(len(ciphertext)-1)

                   if ciphertext[i].isalpha() and ciphertext[i+1].isalpha())

        for digram, \_ in digrams.most\_common(10):  # Top 10 digrams

            if len(digram) != 2:

                continue

            # If we've mapped the first character to 't', second is likely 'h'

            if digram[0] in mapping and mapping[digram[0]] == 't':

                mapping[digram[1]] = 'h'

            # Other common digram patterns

            if digram[0] in mapping and mapping[digram[0]] == 'h':

                mapping[digram[1]] = 'e'

            if digram[0] in mapping and mapping[digram[0]] == 'a':

                mapping[digram[1]] = 'n'

        # Improve mapping using trigrams

        trigrams = Counter(ciphertext[i:i+3].lower() for i in range(len(ciphertext)-2)

                    if ciphertext[i].isalpha() and ciphertext[i+1].isalpha() and ciphertext[i+2].isalpha())

        for trigram, \_ in trigrams.most\_common(5):  # Top 5 trigrams

            if len(trigram) != 3:

                continue

            # Check for 'the' pattern

            if (trigram[0] in mapping and mapping[trigram[0]] == 't' and

                trigram[1] in mapping and mapping[trigram[1]] == 'h'):

                mapping[trigram[2]] = 'e'

            # Check for 'and' pattern

            if (trigram[0] in mapping and mapping[trigram[0]] == 'a' and

                trigram[2] in mapping and mapping[trigram[2]] == 'd'):

                mapping[trigram[1]] = 'n'

        # Decrypt using current mapping

        decrypted = []

        for char in ciphertext.lower():

            if char in mapping:

                decrypted.append(mapping[char])

            elif char.isalpha():

                decrypted.append('\_')  # Use underscore for unknown letters

            else:

                decrypted.append(char)  # Keep punctuation and spaces

        # Additional refinement: look for common words

        decrypted\_text = ''.join(decrypted)

        words = decrypted\_text.split()

        # Try to identify common short words

        for i, word in enumerate(words):

            if len(word) == 1 and word == 'a':

                # Find which cipher letter maps to 'a'

                for c in ciphertext.lower():

                    if c.isalpha() and c in mapping and mapping[c] == 'a':

                        # Look at the original word position

                        orig\_word = ciphertext.split()[i]

                        if len(orig\_word) == 1:

                            mapping[orig\_word[0].lower()] = 'a'

                            break

        # Final decryption with improved mapping

        final\_decrypted = []

        for char in ciphertext.lower():

            if char in mapping:

                final\_decrypted.append(mapping[char])

            elif char.isalpha():

                final\_decrypted.append('\_')

            else:

                final\_decrypted.append(char)

        return ''.join(final\_decrypted), mapping

# Example usage with longer text for better results

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

    print("=== Enhanced Mono-alphabetic Cipher Cracker ===")

    # Create cipher and generate key

    cipher = MonoalphabeticCipher()

    # Use a longer plaintext for better frequency analysis

    plaintext = """The quick brown fox jumps over the lazy dog. This sentence contains all

    the letters in the English alphabet. Frequency analysis works better with more text.

    The more words we have, the more accurate our letter frequency counts will be."""

    ciphertext = cipher.encrypt(plaintext)

    print("\nPlaintext:", plaintext[:50], "...")

    print("Encrypted:", ciphertext[:50], "...")

    # Frequency attack

    print("\nAttempting frequency analysis attack...")

    cracked, mapping = MonoalphabeticCipher.frequency\_attack(ciphertext)

    print("\nRecovered Mapping:")

    for c in sorted(mapping.items(), key=lambda x: x[1]):

        print(f"{c[1]} -> {c[0]}")

    print("\nCracked Message:")

    print(cracked)

    # Compare with actual decryption

    actual\_decrypted = cipher.decrypt\_with\_key(ciphertext)

    print("\nActual Decrypted Message:")

    print(actual\_decrypted)

**6. Hill Ciphering**

import numpy as np

def mod\_inverse(a, m):

    """Returns modular inverse of a under mod m, if exists"""

    a = a % m

    for x in range(1, m):

        if (a \* x) % m == 1:

            return x

    raise ValueError(f"No modular inverse for {a} under mod {m}")

def text\_to\_numbers(text):

    return [ord(c) - ord('a') for c in text.lower()]

def numbers\_to\_text(numbers):

    return ''.join([chr((num % 26) + ord('a')) for num in numbers])

def create\_key\_matrix(key, n):

    key\_nums = text\_to\_numbers(key)

    if len(key\_nums) != n \* n:

        raise ValueError(f"Key must be {n\*n} characters for {n}x{n} matrix.")

    return np.array(key\_nums).reshape(n, n)

def matrix\_mod\_inverse(matrix, modulus):

    """Finds the modular inverse of a matrix under given modulus"""

    det = int(round(np.linalg.det(matrix)))  # determinant

    det\_inv = mod\_inverse(det, modulus)

    matrix\_mod\_inv = (det\_inv \* np.round(det \* np.linalg.inv(matrix)).astype(int)) % modulus

    return matrix\_mod\_inv

def hill\_cipher\_encrypt(plaintext, key):

    plaintext = plaintext.replace(" ", "").lower()

    key\_length = len(key)

    n = int(key\_length \*\* 0.5)

    if n \* n != key\_length:

        raise ValueError("Key length must be a perfect square.")

    while len(plaintext) % n != 0:

        plaintext += 'x'  # Padding

    key\_matrix = create\_key\_matrix(key, n)

    ciphertext = ""

    for i in range(0, len(plaintext), n):

        block = plaintext[i:i+n]

        block\_vector = np.array(text\_to\_numbers(block))

        encrypted\_vector = np.dot(key\_matrix, block\_vector) % 26

        ciphertext += numbers\_to\_text(encrypted\_vector)

    return ciphertext, key\_matrix

def hill\_cipher\_decrypt(ciphertext, key\_matrix):

    n = key\_matrix.shape[0]

    inverse\_key\_matrix = matrix\_mod\_inverse(key\_matrix, 26)

    plaintext = ""

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

        block = ciphertext[i:i+n]

        block\_vector = np.array(text\_to\_numbers(block))

        decrypted\_vector = np.dot(inverse\_key\_matrix, block\_vector) % 26

        plaintext += numbers\_to\_text(decrypted\_vector)

    return plaintext

# Main program

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

    plaintext = input("Enter the plaintext: ").strip()

    key = input("Enter the key (perfect square length, e.g., 4, 9, 16): ").strip()

    try:

        ciphertext, key\_matrix = hill\_cipher\_encrypt(plaintext, key)

        print("\nEncrypted ciphertext:", ciphertext)

        decrypted = hill\_cipher\_decrypt(ciphertext, key\_matrix)

        print("Decrypted plaintext:", decrypted)

    except Exception as e:

        print("Error:", e)

**7. Block Cipher**

import os

BLOCK\_SIZE = 8  # 8 bytes (64 bits)

KEY = b'\x13\x57\x9B\xDF\x02\x46\x8A\xCE'  # example 64-bit key

def xor\_bytes(a, b):

    return bytes(x ^ y for x, y in zip(a, b))

def pad(data):

    pad\_len = BLOCK\_SIZE - (len(data) % BLOCK\_SIZE)

    return data + bytes([pad\_len] \* pad\_len)

def unpad(data):

    pad\_len = data[-1]

    return data[:-pad\_len]

# Toy block cipher: XOR with fixed key

def encrypt\_block(block):

    return xor\_bytes(block, KEY)

def decrypt\_block(block):

    return xor\_bytes(block, KEY)

# ECB Mode

def ecb\_encrypt(plaintext):

    plaintext = pad(plaintext)

    blocks = [plaintext[i:i+BLOCK\_SIZE] for i in range(0, len(plaintext), BLOCK\_SIZE)]

    ciphertext = b''.join(encrypt\_block(block) for block in blocks)

    return ciphertext

def ecb\_decrypt(ciphertext):

    blocks = [ciphertext[i:i+BLOCK\_SIZE] for i in range(0, len(ciphertext), BLOCK\_SIZE)]

    plaintext = b''.join(decrypt\_block(block) for block in blocks)

    return unpad(plaintext)

# CBC Mode

def cbc\_encrypt(plaintext, iv):

    plaintext = pad(plaintext)

    blocks = [plaintext[i:i+BLOCK\_SIZE] for i in range(0, len(plaintext), BLOCK\_SIZE)]

    ciphertext = b''

    prev = iv

    for block in blocks:

        xor\_block = xor\_bytes(block, prev)

        enc\_block = encrypt\_block(xor\_block)

        ciphertext += enc\_block

        prev = enc\_block

    return ciphertext

def cbc\_decrypt(ciphertext, iv):

    blocks = [ciphertext[i:i+BLOCK\_SIZE] for i in range(0, len(ciphertext), BLOCK\_SIZE)]

    plaintext = b''

    prev = iv

    for block in blocks:

        dec\_block = decrypt\_block(block)

        plain\_block = xor\_bytes(dec\_block, prev)

        plaintext += plain\_block

        prev = block

    return unpad(plaintext)

# CFB Mode

def cfb\_encrypt(plaintext, iv):

    plaintext = pad(plaintext)

    ciphertext = b''

    prev = iv

    for i in range(0, len(plaintext), BLOCK\_SIZE):

        keystream = encrypt\_block(prev)

        block = plaintext[i:i+BLOCK\_SIZE]

        cipher\_block = xor\_bytes(block, keystream)

        ciphertext += cipher\_block

        prev = cipher\_block

    return ciphertext

def cfb\_decrypt(ciphertext, iv):

    plaintext = b''

    prev = iv

    for i in range(0, len(ciphertext), BLOCK\_SIZE):

        keystream = encrypt\_block(prev)

        block = ciphertext[i:i+BLOCK\_SIZE]

        plain\_block = xor\_bytes(block, keystream)

        plaintext += plain\_block

        prev = block

    return unpad(plaintext)

# OFB Mode

def ofb\_encrypt\_decrypt(data, iv):

    result = b''

    prev = iv

    for i in range(0, len(data), BLOCK\_SIZE):

        keystream = encrypt\_block(prev)

        block = data[i:i+BLOCK\_SIZE]

        output\_block = xor\_bytes(block, keystream[:len(block)])

        result += output\_block

        prev = keystream

    return result

# Main function to test

def main():

    message = b"This is a test message for block cipher modes."

    iv = os.urandom(BLOCK\_SIZE)

    print("\n--- ECB ---")

    ctext = ecb\_encrypt(message)

    print("Ciphertext:", ctext.hex())

    print("Decrypted :", ecb\_decrypt(ctext))

    print("\n--- CBC ---")

    ctext = cbc\_encrypt(message, iv)

    print("Ciphertext:", ctext.hex())

    print("Decrypted :", cbc\_decrypt(ctext, iv))

    print("\n--- CFB ---")

    ctext = cfb\_encrypt(message, iv)

    print("Ciphertext:", ctext.hex())

    print("Decrypted :", cfb\_decrypt(ctext, iv))

    print("\n--- OFB ---")

    padded\_message = pad(message)

    ctext = ofb\_encrypt\_decrypt(padded\_message, iv)

    print("Ciphertext:", ctext.hex())

    ptext = ofb\_encrypt\_decrypt(ctext, iv)

    print("Decrypted :", unpad(ptext))

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

    main()

**8. ECC**

def main():

    print("Elliptic Curve Cryptography Implementation")

    print("========================================\n")

    p = int(input("Enter the value of p of Ep(a,b): "))

    a = int(input("Enter the value of a of Ep(a,b): "))

    b = int(input("Enter the value of b of Ep(a,b): "))

    g\_input = input("Enter the generator point G (format: x,y): ")

    g = tuple(map(int, g\_input.split(',')))

    print(f"\nThe elliptic curve equation is: y² = x³ + {a}x + {b} mod {p}")

    print(f"Generator point G = {g}\n")

    # ========== Key Exchange ==========

    print("Key Exchange Phase:")

    print("------------------")

    # Alice's key generation

    alpha = int(input("\nEnter Alice's private key α (1 ≤ α ≤ n-1): "))

    pa = point\_multiply(alpha, g, a, p)

    print(f"Alice's public key P\_A = αG = {pa}")

    # Bob's key generation

    beta = int(input("Enter Bob's private key β (1 ≤ β ≤ n-1): "))

    pb = point\_multiply(beta, g, a, p)

    print(f"Bob's public key P\_B = βG = {pb}\n")

    # Shared key computation

    alice\_shared\_key = point\_multiply(alpha, pb, a, p)

    bob\_shared\_key = point\_multiply(beta, pa, a, p)

    print(f"Alice computes shared key: αP\_B = {alice\_shared\_key}")

    print(f"Bob computes shared key: βP\_A = {bob\_shared\_key}")

    print("Key exchange successful!" if alice\_shared\_key == bob\_shared\_key else "Key exchange failed!")

    # ========== Encryption/Decryption ==========

    print("\nEncryption/Decryption Phase:")

    print("---------------------------")

    # Message to encrypt

    pm = get\_valid\_point(p, a, b, "\nEnter message point P\_m to encrypt (format: x,y): ")

    # Alice's encryption

    alpha\_enc = int(input("Enter Alice's random integer α for encryption: "))

    c1 = point\_multiply(alpha\_enc, g, a, p)

    alpha\_pb = point\_multiply(alpha\_enc, pb, a, p)

    c2 = point\_add(pm, alpha\_pb, a, p)

    ciphertext = (c1, c2)

    print(f"\nEncrypted ciphertext C\_m = {ciphertext}")

    # Bob's decryption

    c1, c2 = ciphertext

    beta\_c1 = point\_multiply(beta, c1, a, p)

    beta\_c1\_inv = (beta\_c1[0], (-beta\_c1[1]) % p)  # Negative of the point

    decrypted = point\_add(c2, beta\_c1\_inv, a, p)

    print(f"\nDecrypted message = {decrypted}")

    print(f"Original message was {pm}")

    print("Decryption successful!" if decrypted == pm else "Decryption failed!")

def point\_add(p, q, a, p\_prime):

    """Add two points on the elliptic curve"""

    if p == (0, 0):

        return q

    if q == (0, 0):

        return p

    if p[0] == q[0] and p[1] != q[1]:

        return (0, 0)  # point at infinity

    if p != q:

        # Point addition

        m = (q[1] - p[1]) \* pow(q[0] - p[0], -1, p\_prime)

    else:

        # Point doubling

        m = (3 \* p[0]\*p[0] + a) \* pow(2 \* p[1], -1, p\_prime)

    x\_r = (m\*m - p[0] - q[0]) % p\_prime

    y\_r = (m\*(p[0] - x\_r) - p[1]) % p\_prime

    return (x\_r, y\_r)

def point\_multiply(k, point, a, p\_prime):

    """Multiply point by scalar k using double-and-add algorithm"""

    result = (0, 0)  # point at infinity

    addend = point

    while k > 0:

        if k & 1:

            result = point\_add(result, addend, a, p\_prime)

        addend = point\_add(addend, addend, a, p\_prime)

        k >>= 1

    return result

def is\_point\_on\_curve(point, a, b, p):

    """Check if a point lies on the elliptic curve y² ≡ x³ + ax + b (mod p)"""

    if point == (0, 0):  # Point at infinity (handled separately)

        return True

    x, y = point

    return (y \*\* 2) % p == (x \*\* 3 + a \* x + b) % p

def get\_valid\_point(p, a, b, prompt="Enter point (format: x,y): "):

    """Get and validate a point on the curve"""

    while True:

        try:

            point\_input = input(prompt)

            x, y = map(int, point\_input.split(','))

            # Check if point is on the curve

            if is\_point\_on\_curve((x, y), a, b, p):

                return (x, y)

            else:

                print(f"Error: Point ({x}, {y}) is not on the curve")

        except ValueError:

            print("Please enter coordinates in format 'x,y'")

        except Exception as e:

            print(f"Error: {str(e)}")

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

    main()