Cryptography and Network Security Lab
Digital Assignment 3
22BCE3939
Karan Sehgal

1. Without using library functions develop a menu-driven code to simulate the following in cpp Asymmetric algorithms.
i. RSA
ii. Elgammal
iii.delfie hellman key exchange
iv. ECC - point doubling addition
v. Key Generation – (Public,private)

NOTE: The program should have sufficient test cases to perform data validation. The output should contain intermediate results [provide user-friendly I/O messages]

Pseudocode:

Cryptography and Network Security Lab
Digital Assignment 3
22BCE3939
Karan Sehgal

Pseudocode :→

```
// Utility functions
FUNCTION mod-pow (base, exp, mod):
    if (mod = 1):
        return 0
    result = 1
    base = base % mod
    while (exp > 0):
        if ( exp % 2 = 1):
            result = ( result * base) % mod
        exp = exp >> 1
        base = (base^2) % mod

    return result

FUNCTION is-prime (n):
    if ( n < = 1) return false
    if ( n <= 3) return true
    if ( n % 2 = 0  OR  n % 3 = 0) return false
    for i from 5 to √n :
        if ( n % i = 0  OR  n % (i+2) = 0):
            return false
    return true
```

```
FUNCTION gcd (a, b):
    while (b != 0):
        temp = b
        b = a % b
        a = temp
    return (a)
FUNCTION mod_inv (a, m):
    m0 = m
    x0 = 0 , x1 = 1
    if (m = 1) return 0:

    while (a > 1):
        q = a / m
        t = m
        m = a % m
        a = t
        t = x0
        x0 = x1 - q * x0
        x1 = t

    if (x1 < 0) x1 += m0

    return x1

FUNCTION generate_randomprime (min, ma
    Using Random-device library in cpp
```

// RSA Implementation 22BCE3939

```
FUNCTION rsa_algorithm():
// Generate or input prime nos p and q
  p = GET_PRIME ("Enter prime")
  q = GET_PRIME ("Enter second prime")
  n = p * q
  phi = (p - 1) * (q - 1)
  // choose public exponent e
// if e such that 1 < e < phi and
                gcd (e, phi) = 1

  // compute private exponent d
   d = mod_inv (e, phi)

 // Encryption:
  m = INPUT ( " Enter message to encrypt
                            (<n))
   c = mod_pow (m, e, n)
  print "Encryped Msg":, c
// Decryption :
  decrypted = MOD_pow (c, d, n)
    print " Decrypted Msg: ", decrypted
  END FUNCTION
```

//22BCE3939

// Elgamal Implementation

```
FUNCTION elgamal_algorithm():
    // Choose a large prime p and a
       primitive root g
    p = GET_PRIME ("Enter prime p)
    g = GET_primitive root (p)

    // Choose a private key a
    a = Random (1, p-2)

    // Compute public key h = g^a mod p
    h = mod_pow (g, a, p)

    // Encryption
    // Input message to be encrypted (<p)
    m = Input()
    k = RANDOM (1, p-2)
    c1 = mod_pow (g, k, p) //c1 = g^k mod p
    s = mod_pow (h, k, p) //S = h^k mod p
    c2 = (m * s) % p    // c2 = m*s mod p
    print "Encrypted msg:", (c1, c2)

    // Decryption
    s_inv = mod_pow (c1, p-1-a, p)
    decrypted = (c2 * s_inv) % p
    print "Decrypted msg", decrypted
END FUNCTION
```

```
//diffie- Hellman Key Exchange                22BCE3939
FUNCTION  diffie_hellman ():
  p= GET_prime ("Enter prime p)
  g= GET_primitive root(p)
   // User A's private and public key
     a= RANDOM (1, p-2)
     A = MOD-POW (g, a, p) // g^a mod p
   // User B's private and public key
     b= RANDOM (1, p-2)
     B= MOD-POW (g, b, p) // g^b mod p

    // Shared Secret Computation
    Secret_A = mod_pow (B, a, p)
    secret_B = mod-pow (A, b, p)

    IF  Secret_A = secret_B :
          print "Key exchange succesfull"
    ElSE
          print "Key exchange failed"
    END IF
END FUNCTION
```

// ECC Operations 22BCE393

```
FUNCTION ecc-operations():
    p = GET_PRIME ("Enter prime p")
    a = INPUT ("Enter coeff a")
    b = INPUT ("Enter coeff b")

    Print "Curve : y^2 = x^3 + ax+b mod p

    WHILE TRUE
        PRINT "1. Point Addition"
        PRINT "2. Point Doubling"
        PRINT "3. Scalar Multiplication"
        PRINT "4. Return to Main Menu"

        choice = INPUT ("Enter choice")

        IF choice == 4 THEN
            BREAK
        ELSE IF choice == 1 THEN
            P = GET_POINT ()
            Q = GET_POINT ()
            R = ECC_POINT_ADD (P, Q, a, p

        ELSE IF choice == 2 THEN
            P = GETPOINT ()
            R = ECC_POINT_DOUBLING (P,

        ELSE IF choice == 3 THEN
            P = GET_POINT()
```

```
k = Input ("scalar")                    22BCE3939
R = ECC-scalar (P, k, a, p)

ELSE
    PRINT "Invalid-Choice"
  END IF
 END WHILE
END FUNCTION
```

// Key Generation

```
FUNCTION Key-generation ():
  WHILE TRUE:
    PRINT "1. RSA Key Gen"
    PRINT "2. Elgamal Key Gen"
    PRINT "3. ECC Key Gen"
    PRINT "4. Return to Main Menu"
    choice = input ("Enter choice")
   IF choice == 4 THEN
        BREAK
   ELSE IF choice == 1 THEN
        CALL rsa-key-gen ()
   ELSE IF choice == 2 THEN
        CALL elgamal-key-generation ()
   ELSE if choice == 3 THEN
        CALL ecc-key-gen ()
   ELSE
        print "Invalid Choice"
  END IF
  END WHILE
END FUNCTION
```

22BCE3939

// Sub Functions

```
FUNCTION ecc_key_gen():
    p = GET_PRIME ("Enter prime p")
    a = input ("Enter coeff a")
    b = input ("Enter coeff b")
    G = GET_POINT ("Enter base point G
    n = Input ("Enter order of G")
    d = RANDOM (1, n-1)
    Q = ECC_Scalar (G, d, a, p)
     PRINT "Public Key Q = " Q
     PRINT "Private key d=", d
  END-FUNCTION
FUNCTION elgamal_key_generation():
    p = GETPRIME ("Enter prime p")
    g = GET_primitive root (p)
    x = RANDOM (1, p-2)
    h = mod-pow (g, x, p)
     PRINT "Public key (p, g, h) = " p, g
     PRINT "Private key x = " X
  END FUNCTION
```

```
FUNCTION rsa_key_gen():                          22BCE3939
 p= GET_PRIME ("Enter prime (p)")
 q= GET_PRIME ("Enter second (q)")
  n= p*q
  phi= (p-1) * (q-1)
  e= choose_public_exp(phi)
  d= mod_inv (e, phi)
  PRINT "Public key (e, n)", e, n
  PRINT "Private key (d, n)", d, n
 END FUNCTION
```

Source Code:

```
#include <iostream>
#include <cmath>
#include <string>
#include <vector>
#include <random>
#include <ctime>

using namespace std;

// Utility functions
long long mod_pow(long long base, long long exponent, long long
modulus) {
    if (modulus == 1) return 0;
    long long result = 1;
    base = base % modulus;
    while (exponent > 0) {
        if (exponent % 2 == 1) {
            result = (result * base) % modulus;
        }
        exponent = exponent >> 1;
        base = (base * base) % modulus;
    }
    return result;
}

bool is_prime(long long n) {
    if (n <= 1) return false;
    if (n <= 3) return true;
    if (n % 2 == 0 || n % 3 == 0) return false;

    for (long long i = 5; i * i <= n; i += 6) {
        if (n % i == 0 || n % (i + 2) == 0) return false;
    }
    return true;
}

long long gcd(long long a, long long b) {
    while (b != 0) {
        long long temp = b;
        b = a % b;
        a = temp;
    }
    return a;
```

```
}

long long modular_inverse(long long a, long long m) {
    long long m0 = m, t, q;
    long long x0 = 0, x1 = 1;

    if (m == 1) return 0;

    while (a > 1) {
        q = a / m;
        t = m;
        m = a % m;
        a = t;
        t = x0;
        x0 = x1 - q * x0;
        x1 = t;
    }

    if (x1 < 0) x1 += m0;
    return x1;
}

long long generate_random_prime(long long min_val, long long
max_val) {
    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<long long> dist(min_val, max_val);

    long long num = dist(gen);
    // Make sure the number is odd
    if (num % 2 == 0) num++;

    while (!is_prime(num)) {
        num += 2; // Check next odd number
        if (num > max_val) num = min_val + (num % min_val);
        if (num % 2 == 0) num++;
    }

    return num;
}
```

```
// RSA Algorithm Implementation
void rsa_algorithm() {
    cout << "\n===== RSA ALGORITHM =====\n";

    // Step 1: Generate two large prime numbers
    cout << "Generating prime numbers p and q...\n";
    long long p, q;

    // User can choose to enter primes or generate them
    char choice;
    cout << "Do you want to (e)nter prime numbers or (g)enerate them? (e/g): ";
    cin >> choice;

    if (choice == 'e' || choice == 'E') {
        cout << "Enter first prime number (p): ";
        cin >> p;
        while (!is_prime(p)) {
            cout << "Not a prime number. Please enter a prime number: ";
            cin >> p;
        }

        cout << "Enter second prime number (q): ";
        cin >> q;
        while (!is_prime(q)) {
            cout << "Not a prime number. Please enter a prime number: ";
            cin >> q;
        }
    } else {
        // Generate primes between 100 and 1000 for demonstration purposes
        p = generate_random_prime(100, 1000);
        do {
            q = generate_random_prime(100, 1000);
        } while (p == q);
    }

    cout << "Prime p = " << p << endl;
    cout << "Prime q = " << q << endl;

    // Step 2: Compute n = p * q
    long long n = p * q;
```

```
    cout << "Computing n = p * q = " << p << " * " << q << " = "
<< n << endl;

    // Step 3: Compute Euler's totient function phi(n) = (p-1) * (q-1)
    long long phi = (p - 1) * (q - 1);
    cout << "Computing phi(n) = (p-1) * (q-1) = " << p-1 << " * "
<< q-1 << " = " << phi << endl;

    // Step 4: Choose e such that 1 < e < phi(n) and gcd(e, phi(n)) =
1
    long long e;
    cout << "Choosing public exponent e...\n";

    if (choice == 'e' || choice == 'E') {
        cout << "Enter public exponent e (1 < e < " << phi << " and
gcd(e, " << phi << ") = 1): ";
        cin >> e;
        while (e <= 1 || e >= phi || gcd(e, phi) != 1) {
            cout << "Invalid e. Please enter a valid public exponent: ";
            cin >> e;
        }
    } else {
        // Start with e = 3 (common choice)
        e = 3;
        while (gcd(e, phi) != 1) {
            e += 2;
        }
    }

    cout << "Public exponent e = " << e << endl;

    // Step 5: Compute d such that (d * e) % phi(n) = 1
    long long d = modular_inverse(e, phi);
    cout << "Computing private exponent d...\n";
    cout << "Private exponent d = " << d << " (the modular inverse
of e mod phi(n))\n";
    cout << "Verification: (d * e) % phi(n) = " << (d * e) % phi << "
(should be 1)\n";

    // Display public and private keys
    cout << "\nRSA Keys generated successfully!\n";
    cout << "Public Key (e, n) = (" << e << ", " << n << ")\n";
    cout << "Private Key (d, n) = (" << d << ", " << n << ")\n";
```

```
    // Encryption and decryption example
    long long message;
    cout << "\nEnter a message (a number less than " << n << "):
";
    cin >> message;

    while (message >= n) {
        cout << "Message must be less than " << n << ". Please
enter again: ";
        cin >> message;
    }

    cout << "\nOriginal message: " << message << endl;

    // Encryption: c = m^e mod n
    long long ciphertext = mod_pow(message, e, n);
    cout << "Encryption: C = M^e mod n\n";
    cout << "C = " << message << "^" << e << " mod " << n <<
"\n";
    cout << "Encrypted message (ciphertext): " << ciphertext <<
endl;

    // Decryption: m = c^d mod n
    long long decrypted = mod_pow(ciphertext, d, n);
    cout << "Decryption: M = C^d mod n\n";
    cout << "M = " << ciphertext << "^" << d << " mod " << n
<< "\n";
    cout << "Decrypted message: " << decrypted << endl;

    if (decrypted == message) {
        cout << "Verification: Original message and decrypted
message match!\n";
    } else {
        cout << "Error: Original message and decrypted message do
not match!\n";
    }
}
```

```
void elgamal_algorithm() {
    cout << "\n===== ELGAMAL ALGORITHM =====\n";

    // Step 1: Choose a large prime p and a primitive root g
    long long p;
    cout << "Enter a prime number p: ";
    cin >> p;
    while (!is_prime(p)) {
        cout << "Not a prime number. Please enter a prime number:
";
        cin >> p;
    }

    cout << "Prime p = " << p << endl;

    // For simplicity, we'll use a random number between 2 and p-2
as g
    // In a real implementation, you would need to verify that g is a
primitive root
    long long g;
    cout << "Enter a primitive root g (2 <= g <= " << p-2 << "): ";
    cin >> g;
    while (g < 2 || g > p-2) {
        cout << "Invalid g. Please enter a value between 2 and " <<
p-2 << ": ";
        cin >> g;
    }

    cout << "Primitive root g = " << g << endl;

    // Step 2: Choose a private key a
    long long a;
    cout << "Enter private key a (1 <= a <= " << p-2 << "): ";
    cin >> a;
    while (a < 1 || a > p-2) {
        cout << "Invalid a. Please enter a value between 1 and " <<
p-2 << ": ";
        cin >> a;
    }

    // Step 3: Compute public key h = g^a mod p
    long long h = mod_pow(g, a, p);
    cout << "Computing public key h = g^a mod p\n";
```

```
    cout << "h = " << g << "^" << a << " mod " << p << " = "
<< h << endl;

    // Display public and private parameters
    cout << "\nElGamal Parameters and Keys:\n";
    cout << "Public Parameters: p = " << p << ", g = " << g << "\
n";
    cout << "Public Key: h = " << h << "\n";
    cout << "Private Key: a = " << a << "\n";

    // Encryption
    long long message;
    cout << "\nEnter a message (a number less than " << p << "):
";
    cin >> message;

    while (message >= p) {
        cout << "Message must be less than " << p << ". Please
enter again: ";
        cin >> message;
    }

    cout << "Original message: " << message << endl;

    // Choose a random ephemeral key k
    long long k;
    cout << "Enter an ephemeral key k (1 <= k <= " << p-2 << "):
";
    cin >> k;
    while (k < 1 || k > p-2 || gcd(k, p-1) != 1) {
        cout << "Invalid k. Please enter a value between 1 and " <<
p-2 << " that is coprime with " << p-1 << ": ";
        cin >> k;
    }

    // Compute C1 = g^k mod p
    long long c1 = mod_pow(g, k, p);
    cout << "Computing C1 = g^k mod p\n";
    cout << "C1 = " << g << "^" << k << " mod " << p << " = "
<< c1 << endl;

    // Compute s = h^k mod p
    long long s = mod_pow(h, k, p);
    cout << "Computing shared secret s = h^k mod p\n";
```

```
    cout << "s = " << h << "^" << k << " mod " << p << " = " <<
s << endl;

    // Compute C2 = m * s mod p
    long long c2 = (message * s) % p;
    cout << "Computing C2 = m * s mod p\n";
    cout << "C2 = " << message << " * " << s << " mod " << p
<< " = " << c2 << endl;

    cout << "\nEncrypted message: (C1, C2) = (" << c1 << ", " <<
c2 << ")\n";

    // Decryption
    // Compute s' = C1^a mod p
    long long s_prime = mod_pow(c1, a, p);
    cout << "\nDecryption:\n";
    cout << "Computing shared secret s' = C1^a mod p\n";
    cout << "s' = " << c1 << "^" << a << " mod " << p << " = "
<< s_prime << endl;

    // Compute m = C2 * s'^(p-2) mod p (Using Fermat's Little
Theorem)
    long long s_inv = mod_pow(s_prime, p-2, p);
    cout << "Computing modular inverse of s' using Fermat's Little
Theorem: s'^(p-2) mod p\n";
    cout << "s'^(p-2) = " << s_prime << "^" << p-2 << " mod "
<< p << " = " << s_inv << endl;

    long long decrypted = (c2 * s_inv) % p;
    cout << "Computing m = C2 * s'^(p-2) mod p\n";
    cout << "m = " << c2 << " * " << s_inv << " mod " << p << "
= " << decrypted << endl;

    cout << "Decrypted message: " << decrypted << endl;

    if (decrypted == message) {
        cout << "Verification: Original message and decrypted
message match!\n";
    } else {
        cout << "Error: Original message and decrypted message do
not match!\n";
    }
}
```

// Diffie-Hellman Key Exchange

```
void diffie_hellman() {
    cout << "\n===== DIFFIE-HELLMAN KEY EXCHANGE
=====\n";

    // Step 1: Choose a prime number p and a primitive root g
    long long p;
    cout << "Enter a prime number p: ";
    cin >> p;
    while (!is_prime(p)) {
        cout << "Not a prime number. Please enter a prime number:
";
        cin >> p;
    }

    cout << "Prime p = " << p << endl;

    // For simplicity, we'll use a random number between 2 and p-2
as g
    // In a real implementation, you would need to verify that g is a
primitive root
    long long g;
    cout << "Enter a primitive root g (2 <= g <= " << p-2 << "): ";
    cin >> g;
    while (g < 2 || g > p-2) {
        cout << "Invalid g. Please enter a value between 2 and " <<
p-2 << ": ";
        cin >> g;
    }

    cout << "Primitive root g = " << g << endl;

    cout << "\nPublic parameters: p = " << p << ", g = " << g <<
endl;

    // Step 2: User A chooses a private key a
    cout << "\n--- User A ---\n";
    long long a;
    cout << "Enter User A's private key (1 <= a <= " << p-2 << "):
";
    cin >> a;
    while (a < 1 || a > p-2) {
        cout << "Invalid a. Please enter a value between 1 and " <<
p-2 << ": ";
```

```
        cin >> a;
    }

    // Step 3: User A computes public key A = g^a mod p
    long long A = mod_pow(g, a, p);
    cout << "Computing User A's public key: A = g^a mod p\n";
    cout << "A = " << g << "^" << a << " mod " << p << " = "
<< A << endl;

    // Step 4: User B chooses a private key b
    cout << "\n--- User B ---\n";
    long long b;
    cout << "Enter User B's private key (1 <= b <= " << p-2 << "):
";
    cin >> b;
    while (b < 1 || b > p-2) {
        cout << "Invalid b. Please enter a value between 1 and " <<
p-2 << ": ";
        cin >> b;
    }

    // Step 5: User B computes public key B = g^b mod p
    long long B = mod_pow(g, b, p);
    cout << "Computing User B's public key: B = g^b mod p\n";
    cout << "B = " << g << "^" << b << " mod " << p << " = "
<< B << endl;

    // Step 6: User A computes shared secret
    cout << "\n--- Shared Secret Computation ---\n";
    cout << "User A receives User B's public key: B = " << B <<
endl;
    long long secret_A = mod_pow(B, a, p);
    cout << "User A computes shared secret: s = B^a mod p\n";
    cout << "s = " << B << "^" << a << " mod " << p << " = "
<< secret_A << endl;

    // Step 7: User B computes shared secret
    cout << "User B receives User A's public key: A = " << A <<
endl;
    long long secret_B = mod_pow(A, b, p);
    cout << "User B computes shared secret: s = A^b mod p\n";
    cout << "s = " << A << "^" << b << " mod " << p << " = "
<< secret_B << endl;
```

```
    if (secret_A == secret_B) {
        cout << "\nVerification: Both users have computed the same
shared secret: " << secret_A << endl;
        cout << "Diffie-Hellman key exchange completed
successfully!\n";
    } else {
        cout << "\nError: Shared secrets do not match. Something
went wrong!\n";
    }
}

// ECC Point class to represent points on an elliptic curve
class ECPoint {
public:
    long long x;
    long long y;
    bool is_infinity;

    ECPoint() : x(0), y(0), is_infinity(true) {} // Point at infinity

    ECPoint(long long x_val, long long y_val) : x(x_val), y(y_val),
is_infinity(false) {}

    bool operator==(const ECPoint& other) const {
        if (is_infinity && other.is_infinity) return true;
        if (is_infinity || other.is_infinity) return false;
        return (x == other.x && y == other.y);
    }

    void print() const {
        if (is_infinity) {
            cout << "Point at infinity";
        } else {
            cout << "(" << x << ", " << y << ")";
        }
    }
};
```

```
// ECC operations implementation
void ecc_operations() {
    cout << "\n===== ELLIPTIC CURVE CRYPTOGRAPHY OPERATIONS =====\n";

    // Step 1: Define the elliptic curve E: y^2 = x^3 + ax + b (mod p)
    long long a, b, p;

    cout << "Enter the prime modulus p: ";
    cin >> p;
    while (!is_prime(p)) {
        cout << "Not a prime number. Please enter a prime number: ";
        cin >> p;
    }

    cout << "Enter coefficient a for the curve y^2 = x^3 + ax + b (mod " << p << "): ";
    cin >> a;

    cout << "Enter coefficient b for the curve y^2 = x^3 + ax + b (mod " << p << "): ";
    cin >> b;

    // Check that 4a^3 + 27b^2 != 0 (mod p) to ensure the curve is non-singular
    long long discriminant = (4 * mod_pow(a, 3, p) + 27 * mod_pow(b, 2, p)) % p;
    if (discriminant == 0) {
        cout << "Error: The curve is singular (4a^3 + 27b^2 = 0). Please choose different parameters.\n";
        return;
    }

    cout << "\nElliptic Curve: y^2 = x^3 + " << a << "x + " << b << " (mod " << p << ")\n";

    // Function to check if a point is on the curve
    auto is_on_curve = [a, b, p](const ECPoint& point) -> bool {
        if (point.is_infinity) return true;

        long long left_side = (point.y * point.y) % p;
```

```
        long long right_side = (mod_pow(point.x, 3, p) + (a * point.x)
% p + b) % p;

        return (left_side == right_side);
    };

    // Point operations
    while (true) {
        cout << "\n--- ECC Operations Menu ---\n";
        cout << "1. Check if a point is on the curve\n";
        cout << "2. Point Addition\n";
        cout << "3. Point Doubling\n";
        cout << "4. Scalar Multiplication\n";
        cout << "5. Return to main menu\n";
        cout << "Enter your choice: ";

        int choice;
        cin >> choice;

        if (choice == 5) break;

        switch (choice) {
            case 1: {
                long long x, y;
                cout << "Enter point coordinates (x, y):\n";
                cout << "x: ";
                cin >> x;
                cout << "y: ";
                cin >> y;

                ECPoint point(x, y);
                if (is_on_curve(point)) {
                    cout << "The point "; point.print(); cout << " is on the
curve.\n";
                } else {
                    cout << "The point "; point.print(); cout << " is NOT
on the curve.\n";
                }
                break;
            }
            case 2: {
                // Point addition P + Q
                long long x1, y1, x2, y2;
                cout << "Enter coordinates for point P:\n";
```

```
            cout << "x1: ";
            cin >> x1;
            cout << "y1: ";
            cin >> y1;

            cout << "Enter coordinates for point Q:\n";
            cout << "x2: ";
            cin >> x2;
            cout << "y2: ";
            cin >> y2;

            ECPoint P(x1, y1);
            ECPoint Q(x2, y2);

            // Check if points are on the curve
            if (!is_on_curve(P) || !is_on_curve(Q)) {
                cout << "Error: One or both points are not on the
curve.\n";
                break;
            }

            // Check for point at infinity cases
            if (P.is_infinity) {
                cout << "P is the point at infinity, so P + Q = Q = ";
Q.print(); cout << endl;
                break;
            }
            if (Q.is_infinity) {
                cout << "Q is the point at infinity, so P + Q = P = ";
P.print(); cout << endl;
                break;
            }

            // Check if P = -Q
            if (P.x == Q.x && (P.y == (p - Q.y) % p || (P.y == 0 &&
Q.y == 0))) {
                cout << "P + Q = point at infinity (P = -Q)\n";
                break;
            }

            // Point addition formula
            long long lambda;
            if (P.x == Q.x && P.y == Q.y) {
                // Point doubling when P = Q
```

```
            // L = (3x_P^2 + a) / (2y_P) mod p
            long long numerator = (3 * mod_pow(P.x, 2, p) + a) %
p;
            long long denominator = (2 * P.y) % p;
            long long denom_inv = modular_inverse(denominator,
p);
            lambda = (numerator * denom_inv) % p;

            cout << "Computing lambda for point doubling:\n";
            cout << "L = (3x_P^2 + a) / (2y_P) mod p\n";
            cout << "L = (3*" << P.x << "^2 + " << a << ") / (2*"
<< P.y << ") mod " << p << "\n";
            cout << "L = " << numerator << " / " <<
denominator << " mod " << p << "\n";
            cout << "L = " << numerator << " * " << denom_inv
<< " mod " << p << " = " << lambda << endl;
        } else {
            // Point addition when P != Q
            // L = (y_Q - y_P) / (x_Q - x_P) mod p
            long long numerator = (Q.y - P.y + p) % p;
            long long denominator = (Q.x - P.x + p) % p;
            long long denom_inv = modular_inverse(denominator,
p);
            lambda = (numerator * denom_inv) % p;

            cout << "Computing lambda for point addition:\n";
            cout << "L = (y_Q - y_P) / (x_Q - x_P) mod p\n";
            cout << "L = (" << Q.y << " - " << P.y << ") / (" <<
Q.x << " - " << P.x << ") mod " << p << "\n";
            cout << "L = " << numerator << " / " <<
denominator << " mod " << p << "\n";
            cout << "L = " << numerator << " * " << denom_inv
<< " mod " << p << " = " << lambda << endl;
        }

        // x_R = L^2 - x_P - x_Q mod p
        long long x3 = (mod_pow(lambda, 2, p) - P.x - Q.x + 2*p)
% p;

        // y_R = L(x_P - x_R) - y_P mod p
        long long y3 = (lambda * (P.x - x3 + p) % p - P.y + p) % p;

        ECPoint R(x3, y3);
```

```
            cout << "Computing result coordinates:\n";
            cout << "x_R = L^2 - x_P - x_Q mod p\n";
            cout << "x_R = " << lambda << "^2 - " << P.x << " - "
<< Q.x << " mod " << p << " = " << x3 << endl;
            cout << "y_R = L(x_P - x_R) - y_P mod p\n";
            cout << "y_R = " << lambda << "(" << P.x << " - " <<
x3 << ") - " << P.y << " mod " << p << " = " << y3 << endl;

            cout << "P + Q = "; R.print(); cout << endl;

            // Verify the result
            if (is_on_curve(R)) {
                cout << "Verification: The resulting point is on the
curve.\n";
            } else {
                cout << "Error: The resulting point is NOT on the
curve. Something went wrong.\n";
            }
            break;
        }
        case 3: {
            // Point doubling P + P
            long long x, y;
            cout << "Enter coordinates for point P:\n";
            cout << "x: ";
            cin >> x;
            cout << "y: ";
            cin >> y;

            ECPoint P(x, y);

            // Check if point is on the curve
            if (!is_on_curve(P)) {
                cout << "Error: The point is not on the curve.\n";
                break;
            }

            // Check for special cases
            if (P.is_infinity) {
                cout << "P is the point at infinity, so 2P = Point at
infinity\n";
                break;
            }
```

```
        if (P.y == 0) {
            cout << "2P = Point at infinity (when y = 0)\n";
            break;
        }

        // Point doubling formula
        // L = (3x_P^2 + a) / (2y_P) mod p
        long long numerator = (3 * mod_pow(P.x, 2, p) + a) % p;
        long long denominator = (2 * P.y) % p;
        long long denom_inv = modular_inverse(denominator, p);
        long long lambda = (numerator * denom_inv) % p;

        cout << "Computing lambda for point doubling:\n";
        cout << "L = (3x_P^2 + a) / (2y_P) mod p\n";
        cout << "L = (3*" << P.x << "^2 + " << a << ") / (2*"
<< P.y << ") mod " << p << "\n";
        cout << "L = " << numerator << " / " << denominator
<< " mod " << p << "\n";
        cout << "L = " << numerator << " * " << denom_inv
<< " mod " << p << " = " << lambda << endl;

        // x_R = L^2 - 2x_P mod p
        long long x3 = (mod_pow(lambda, 2, p) - 2 * P.x + p) % p;

        // y_R = L(x_P - x_R) - y_P mod p
        long long y3 = (lambda * (P.x - x3 + p) % p - P.y + p) % p;

        ECPoint R(x3, y3);

        cout << "Computing result coordinates:\n";
        cout << "x_R = L^2 - 2x_P mod p\n";
        cout << "x_R = " << lambda << "^2 - 2*" << P.x << "
mod " << p << " = " << x3 << endl;
        cout << "y_R = L(x_P - x_R) - y_P mod p\n";
        cout << "y_R = " << lambda << "(" << P.x << " - " <<
x3 << ") - " << P.y << " mod " << p << " = " << y3 << endl;

        cout << "2P = "; R.print(); cout << endl;

        // Verify the result
        if (is_on_curve(R)) {
            cout << "Verification: The resulting point is on the
curve.\n";
        } else {
```

```
                cout << "Error: The resulting point is NOT on the
curve. Something went wrong.\n";
            }
            break;
        }
        case 4: {
            // Scalar multiplication kP
            long long x, y, k;
            cout << "Enter coordinates for point P:\n";
            cout << "x: ";
            cin >> x;
            cout << "y: ";
            cin >> y;

            cout << "Enter scalar k (positive integer): ";
            cin >> k;
            while (k <= 0) {
                cout << "k must be positive. Please enter again: ";
                cin >> k;
            }

            ECPoint P(x, y);
            // Check if point is on the curve
            if (!is_on_curve(P)) {
                cout << "Error: The point is not on the curve.\n";
                break;
            }

            // Double-and-add algorithm for scalar multiplication
            ECPoint result;
            result.is_infinity = true; // Initialize with point at infinity
(identity element)

            ECPoint temp = P; // Copy of the original point

            cout << "\nComputing " << k << "P using double-and-
add algorithm:\n";
            cout << "Start with R = Point at infinity (identity
element)\n";

            while (k > 0) {
                if (k % 2 == 1) {
                    // If k is odd, add temp to the result
```

```
            cout << "k = " << k << " is odd, so add current
point to result\n";

            // Handle the case when result is the point at infinity
            if (result.is_infinity) {
                result = temp;
                cout << "Result = "; result.print(); cout << endl;
            } else if (temp.is_infinity) {
                // Do nothing, keep result as is
            } else if (result.x == temp.x && result.y == (p -
temp.y) % p) {
                // If result = -temp, then result + temp = infinity
                result.is_infinity = true;
                cout << "Result = Point at infinity\n";
            } else {
                // Regular point addition
                long long lambda;
                if (result.x == temp.x && result.y == temp.y) {
                    // Point doubling
                    long long numerator = (3 * mod_pow(result.x,
2, p) + a) % p;
                    long long denominator = (2 * result.y) % p;
                    long long denom_inv =
modular_inverse(denominator, p);
                    lambda = (numerator * denom_inv) % p;
                } else {
                    // Point addition
                    long long numerator = (temp.y - result.y + p)
% p;
                    long long denominator = (temp.x - result.x + p)
% p;
                    long long denom_inv =
modular_inverse(denominator, p);
                    lambda = (numerator * denom_inv) % p;
                }

                long long x3 = (mod_pow(lambda, 2, p) - result.x -
temp.x + 2*p) % p;
                long long y3 = (lambda * (result.x - x3 + p) % p -
result.y + p) % p;

                result = ECPoint(x3, y3);
                cout << "Result = "; result.print(); cout << endl;
            }
```

```
                }

                // Double the temporary point
                cout << "Double the temporary point\n";

                // Handle special cases for doubling
                if (temp.is_infinity || temp.y == 0) {
                    temp.is_infinity = true;
                    cout << "Temp = Point at infinity\n";
                } else {
                    // Regular point doubling
                    long long numerator = (3 * mod_pow(temp.x, 2, p)
+ a) % p;
                    long long denominator = (2 * temp.y) % p;
                    long long denom_inv =
modular_inverse(denominator, p);
                    long long lambda = (numerator * denom_inv) % p;

                    long long x3 = (mod_pow(lambda, 2, p) - 2 * temp.x
+ p) % p;
                    long long y3 = (lambda * (temp.x - x3 + p) % p -
temp.y + p) % p;

                    temp = ECPoint(x3, y3);
                    cout << "Temp = "; temp.print(); cout << endl;
                }

                k /= 2; // Right shift k
            }

            cout << "\nFinal result " << "kP = "; result.print(); cout
<< endl;

            // Verify the result
            if (is_on_curve(result)) {
                cout << "Verification: The resulting point is on the
curve.\n";
            } else {
                cout << "Error: The resulting point is NOT on the
curve. Something went wrong.\n";
            }
            break;
        }
        default:
```

```
            cout << "Invalid choice. Please try again.\n";
        }
    }
}
// Key generation function for asymmetric algorithms
void key_generation() {
    cout << "\n===== KEY GENERATION FOR ASYMMETRIC
ALGORITHMS =====\n";

    while (true) {
        cout << "\n--- Key Generation Menu ---\n";
        cout << "1. RSA Key Generation\n";
        cout << "2. ElGamal Key Generation\n";
        cout << "3. ECC Key Generation\n";
        cout << "4. Return to main menu\n";
        cout << "Enter your choice: ";

        int choice;
        cin >> choice;

        if (choice == 4) break;

        switch (choice) {
            case 1: {
                // RSA Key Generation
                cout << "\n--- RSA Key Generation ---\n";

                // Step 1: Generate two large prime numbers
                cout << "Generating prime numbers p and q...\n";
                long long p, q;

                char user_choice;
                cout << "Do you want to (e)nter prime numbers or
(g)enerate them? (e/g): ";
                cin >> user_choice;

                if (user_choice == 'e' || user_choice == 'E') {
                    cout << "Enter first prime number (p): ";
                    cin >> p;
                    while (!is_prime(p)) {
                        cout << "Not a prime number. Please enter a prime
number: ";
                        cin >> p;
                    }
```

```
        cout << "Enter second prime number (q): ";
        cin >> q;
        while (!is_prime(q)) {
            cout << "Not a prime number. Please enter a prime
number: ";
            cin >> q;
        }
    } else {
        // Generate primes between 100 and 1000 for
demonstration purposes
        p = generate_random_prime(100, 1000);
        do {
            q = generate_random_prime(100, 1000);
        } while (p == q);
    }

    cout << "Prime p = " << p << endl;
    cout << "Prime q = " << q << endl;

    // Step 2: Compute n = p * q
    long long n = p * q;
    cout << "Computing n = p * q = " << p << " * " << q
<< " = " << n << endl;

    // Step 3: Compute Euler's totient function phi(n) = (p-1)
* (q-1)
    long long phi = (p - 1) * (q - 1);
    cout << "Computing phi(n) = (p-1) * (q-1) = " << p-1 <<
" * " << q-1 << " = " << phi << endl;

    // Step 4: Choose e such that 1 < e < phi(n) and gcd(e,
phi(n)) = 1
    long long e;
    cout << "Choosing public exponent e...\n";

    if (user_choice == 'e' || user_choice == 'E') {
        cout << "Enter public exponent e (1 < e < " << phi
<< " and gcd(e, " << phi << ") = 1): ";
        cin >> e;
        while (e <= 1 || e >= phi || gcd(e, phi) != 1) {
            cout << "Invalid e. Please enter a valid public
exponent: ";
            cin >> e;
```

```
            }
        } else {
            // Start with e = 3 (common choice)
            e = 3;
            while (gcd(e, phi) != 1) {
                e += 2;
            }
        }

        cout << "Public exponent e = " << e << endl;

        // Step 5: Compute d such that (d * e) % phi(n) = 1
        long long d = modular_inverse(e, phi);
        cout << "Computing private exponent d...\n";
        cout << "Private exponent d = " << d << " (the modular
inverse of e mod phi(n))\n";
        cout << "Verification: (d * e) % phi(n) = " << (d * e) %
phi << " (should be 1)\n";

        // Display public and private keys
        cout << "\nRSA Keys generated successfully!\n";
        cout << "Public Key (e, n) = (" << e << ", " << n << ")\
n";
        cout << "Private Key (d, n) = (" << d << ", " << n <<
")\n";
        break;
    }
    case 2: {
        // ElGamal Key Generation
        cout << "\n--- ElGamal Key Generation ---\n";

        // Step 1: Choose a large prime p and a primitive root g
        long long p;
        cout << "Enter a prime number p: ";
        cin >> p;
        while (!is_prime(p)) {
            cout << "Not a prime number. Please enter a prime
number: ";
            cin >> p;
        }

        cout << "Prime p = " << p << endl;
```

```
            // For simplicity, we'll use a random number between 2
and p-2 as g
            // In a real implementation, you would need to verify that
g is a primitive root
            long long g;
            cout << "Enter a primitive root g (2 <= g <= " << p-2
<< "): ";
            cin >> g;
            while (g < 2 || g > p-2) {
                cout << "Invalid g. Please enter a value between 2
and " << p-2 << ": ";
                cin >> g;
            }

            cout << "Primitive root g = " << g << endl;

            // Step 2: Choose a private key a
            long long a;
            cout << "Enter private key a (1 <= a <= " << p-2 << "):
";
            cin >> a;
            while (a < 1 || a > p-2) {
                cout << "Invalid a. Please enter a value between 1 and
" << p-2 << ": ";
                cin >> a;
            }

            // Step 3: Compute public key h = g^a mod p
            long long h = mod_pow(g, a, p);
            cout << "Computing public key h = g^a mod p\n";
            cout << "h = " << g << "^" << a << " mod " << p << "
= " << h << endl;

            // Display public and private parameters
            cout << "\nElGamal Parameters and Keys:\n";
            cout << "Public Parameters: p = " << p << ", g = " <<
g << "\n";
            cout << "Public Key: h = " << h << "\n";
            cout << "Private Key: a = " << a << "\n";
            break;
        }
        case 3: {
            // ECC Key Generation
            cout << "\n--- ECC Key Generation ---\n";
```

```
    // Step 1: Define the elliptic curve E: y^2 = x^3 + ax + b
(mod p)
    long long a, b, p;

    cout << "Enter the prime modulus p: ";
    cin >> p;
    while (!is_prime(p)) {
        cout << "Not a prime number. Please enter a prime
number: ";
        cin >> p;
    }

    cout << "Enter coefficient a for the curve y^2 = x^3 +
ax + b (mod " << p << "): ";
    cin >> a;

    cout << "Enter coefficient b for the curve y^2 = x^3 +
ax + b (mod " << p << "): ";
    cin >> b;

    // Check that 4a^3 + 27b^2 != 0 (mod p) to ensure the
curve is non-singular
    long long discriminant = (4 * mod_pow(a, 3, p) + 27 *
mod_pow(b, 2, p)) % p;
    if (discriminant == 0) {
        cout << "Error: The curve is singular (4a^3 + 27b^2
= 0). Please choose different parameters.\n";
        break;
    }

    cout << "\nElliptic Curve: y^2 = x^3 + " << a << "x + "
<< b << " (mod " << p << ")\n";

    // Step 2: Choose a base point G
    long long gx, gy;
    cout << "Enter coordinates for base point G:\n";
    cout << "x: ";
    cin >> gx;
    cout << "y: ";
    cin >> gy;

    ECPoint G(gx, gy);
```

```
        // Check if G is on the curve
        long long left_side = (G.y * G.y) % p;
        long long right_side = (mod_pow(G.x, 3, p) + (a * G.x) %
p + b) % p;

        if (left_side != right_side) {
            cout << "Error: The base point G is not on the curve.\
n";
            break;
        }

        cout << "Base point G = "; G.print(); cout << endl;

        // Step 3: Choose a private key d
        long long d;
        cout << "Enter a private key d (a large positive integer):
";
        cin >> d;
        while (d <= 0) {
            cout << "d must be positive. Please enter again: ";
            cin >> d;
        }

        // Step 4: Compute the public key Q = dG
        cout << "Computing public key Q = dG...\n";

        // Double-and-add algorithm for scalar multiplication
        ECPoint Q;
        Q.is_infinity = true; // Initialize with point at infinity

        ECPoint temp = G; // Copy of the base point
        long long k = d;  // Copy of the private key

        while (k > 0) {
            if (k % 2 == 1) {
                // If k is odd, add temp to Q

                // Handle the case when Q is the point at infinity
                if (Q.is_infinity) {
                    Q = temp;
                } else if (temp.is_infinity) {
                    // Do nothing, keep Q as is
                } else if (Q.x == temp.x && Q.y == (p - temp.y) %
p) {
```

```
                // If Q = -temp, then Q + temp = infinity
                Q.is_infinity = true;
            } else {
                // Regular point addition
                long long lambda;
                if (Q.x == temp.x && Q.y == temp.y) {
                    // Point doubling
                    long long numerator = (3 * mod_pow(Q.x, 2, p)
+ a) % p;
                    long long denominator = (2 * Q.y) % p;
                    long long denom_inv =
modular_inverse(denominator, p);
                    lambda = (numerator * denom_inv) % p;
                } else {
                    // Point addition
                    long long numerator = (temp.y - Q.y + p) % p;
                    long long denominator = (temp.x - Q.x + p) %
p;
                    long long denom_inv =
modular_inverse(denominator, p);
                    lambda = (numerator * denom_inv) % p;
                }

                long long x3 = (mod_pow(lambda, 2, p) - Q.x -
temp.x + 2*p) % p;
                long long y3 = (lambda * (Q.x - x3 + p) % p - Q.y
+ p) % p;

                Q = ECPoint(x3, y3);
            }
        }

        // Double the temporary point
        if (temp.is_infinity || temp.y == 0) {
            temp.is_infinity = true;
        } else {
            // Regular point doubling
            long long numerator = (3 * mod_pow(temp.x, 2, p)
+ a) % p;
            long long denominator = (2 * temp.y) % p;
            long long denom_inv =
modular_inverse(denominator, p);
            long long lambda = (numerator * denom_inv) % p;
```

```
                long long x3 = (mod_pow(lambda, 2, p) - 2 * temp.x
+ p) % p;
                long long y3 = (lambda * (temp.x - x3 + p) % p -
temp.y + p) % p;

                temp = ECPoint(x3, y3);
            }

            k /= 2; // Right shift k
        }

        cout << "Public key Q = "; Q.print(); cout << endl;

        // Display keys
        cout << "\nECC Keys generated successfully!\n";
        cout << "Private Key: d = " << d << "\n";
        cout << "Public Key: Q = "; Q.print(); cout << endl;
        break;
    }
    default:
        cout << "Invalid choice. Please try again.\n";
    }
  }
}

// Main function
int main() {
  cout << "===== ASYMMETRIC CRYPTOGRAPHY
SIMULATION =====\n";

  while (true) {
    cout << "\n--- Main Menu ---\n";
    cout << "1. RSA Algorithm\n";
    cout << "2. ElGamal Algorithm\n";
    cout << "3. Diffie-Hellman Key Exchange\n";
    cout << "4. ECC Point Operations (Addition, Doubling)\n";
    cout << "5. Key Generation (Public, Private)\n";
    cout << "6. Exit\n";
    cout << "Enter your choice: ";

    int choice;
    cin >> choice;

    if (choice == 6) {
```

```
            cout << "Exiting program. Goodbye!\n";
            break;
        }

        switch (choice) {
            case 1:
                rsa_algorithm();
                break;
            case 2:
                elgamal_algorithm();
                break;
            case 3:
                diffie_hellman();
                break;
            case 4:
                ecc_operations();
                break;
            case 5:
                key_generation();
                break;
            default:
                cout << "Invalid choice. Please try again.\n";
        }
    }

    return 0;
}
```

Output:

RSA Algorithm:

1. Basic Encryption and Decryption:

a. Generating random Prime numbers

```
===== ASYMMETRIC CRYPTOGRAPHY SIMULATION =====

--- Main Menu ---
1. RSA Algorithm
2. ElGamal Algorithm
3. Diffie-Hellman Key Exchange
4. ECC Point Operations (Addition, Doubling)
5. Key Generation (Public, Private)
6. Exit
Enter your choice: 1

===== RSA ALGORITHM =====
Generating prime numbers p and q...
Do you want to (e)nter prime numbers or (g)enerate them? (e/g): g
Prime p = 751
Prime q = 397
Computing n = p * q = 751 * 397 = 298147
Computing phi(n) = (p-1) * (q-1) = 750 * 396 = 297000
Choosing public exponent e...
Public exponent e = 7
Computing private exponent d...
Private exponent d = 212143 (the modular inverse of e mod phi(n))
Verification: (d * e) % phi(n) = 1 (should be 1)

RSA Keys generated successfully!
Public Key (e, n) = (7, 298147)
Private Key (d, n) = (212143, 298147)

Enter a message (a number less than 298147): 567

Original message: 567
Encryption: C = M^e mod n
C = 567^7 mod 298147
Encrypted message (ciphertext): 24140
Decryption: M = C^d mod n
M = 24140^212143 mod 298147
Decrypted message: 567
Verification: Original message and decrypted message match!
```

b. Inputing prime numbers:

```
--- Main Menu ---
1. RSA Algorithm
2. ElGamal Algorithm
3. Diffie-Hellman Key Exchange
4. ECC Point Operations (Addition, Doubling)
5. Key Generation (Public, Private)
6. Exit
Enter your choice: 1

===== RSA ALGORITHM =====
Generating prime numbers p and q...
Do you want to (e)nter prime numbers or (g)enerate them? (e/g): e
Enter first prime number (p): 3
Enter second prime number (q): 7
Prime p = 3
Prime q = 7
Computing n = p * q = 3 * 7 = 21
Computing phi(n) = (p-1) * (q-1) = 2 * 6 = 12
Choosing public exponent e...
Enter public exponent e (1 < e < 12 and gcd(e, 12) = 1): 5
Public exponent e = 5
Computing private exponent d...
Private exponent d = 5 (the modular inverse of e mod phi(n))
Verification: (d * e) % phi(n) = 1 (should be 1)

RSA Keys generated successfully!
Public Key (e, n) = (5, 21)
Private Key (d, n) = (5, 21)

Enter a message (a number less than 21): 15

Original message: 15
Encryption: C = M^e mod n
C = 15^5 mod 21
Encrypted message (ciphertext): 15
Decryption: M = C^d mod n
M = 15^5 mod 21
Decrypted message: 15
Verification: Original message and decrypted message match!
```

2. Invalid Input

a. non prime input :

```
===== RSA ALGORITHM =====
Generating prime numbers p and q...
Do you want to (e)nter prime numbers or (g)enerate them? (e/g): e
Enter first prime number (p): 5
Enter second prime number (q): 6
Not a prime number. Please enter a prime number: 8
Not a prime number. Please enter a prime number: 7
Prime p = 5
Prime q = 7
```

b. non relative prime public exponent

```
Prime p = 5
Prime q = 7
Computing n = p * q = 5 * 7 = 35
Computing phi(n) = (p-1) * (q-1) = 4 * 6 = 24
Choosing public exponent e...
Enter public exponent e (1 < e < 24 and gcd(e, 24) = 1): 3
Invalid e. Please enter a valid public exponent: 5
Public exponent e = 5
Computing private exponent d...
Private exponent d = 5 (the modular inverse of e mod phi(n))
Verification: (d * e) % phi(n) = 1 (should be 1)

RSA Keys generated successfully!
Public Key (e, n) = (5, 35)
Private Key (d, n) = (5, 35)
```

c. input greater than n

```
RSA Keys generated successfully!
Public Key (e, n) = (5, 35)
Private Key (d, n) = (5, 35)

Enter a message (a number less than 35): 37
Message must be less than 35. Please enter again: 34

Original message: 34
Encryption: C = M^e mod n
C = 34^5 mod 35
Encrypted message (ciphertext): 34
Decryption: M = C^d mod n
M = 34^5 mod 35
Decrypted message: 34
Verification: Original message and decrypted message match!
```

Elgamal Algorithm:

1. Basic Encryption and Decryption:

```
--- Main Menu ---
1. RSA Algorithm
2. ElGamal Algorithm
3. Diffie-Hellman Key Exchange
4. ECC Point Operations (Addition, Doubling)
5. Key Generation (Public, Private)
6. Exit
Enter your choice: 2

===== ELGAMAL ALGORITHM =====
Enter a prime number p: 13
Prime p = 13
Enter a primitive root g (2 <= g <= 11): 4
Primitive root g = 4
Enter private key a (1 <= a <= 11): 7
Computing public key h = g^a mod p
h = 4^7 mod 13 = 4

ElGamal Parameters and Keys:
Public Parameters: p = 13, g = 4
Public Key: h = 4
Private Key: a = 7

Enter a message (a number less than 13): 11
Original message: 11
Enter an ephemeral key k (1 <= k <= 11): 5
Computing C1 = g^k mod p
C1 = 4^5 mod 13 = 10
Computing shared secret s = h^k mod p
s = 4^5 mod 13 = 10
Computing C2 = m * s mod p
C2 = 11 * 10 mod 13 = 6

Encrypted message: (C1, C2) = (10, 6)

Decryption:
Computing shared secret s' = C1^a mod p
s' = 10^7 mod 13 = 10
Computing modular inverse of s' using Fermat's Little Theorem: s'^(p-2) mod p
s'^(p-2) = 10^11 mod 13 = 4
Computing m = C2 * s'^(p-2) mod p
m = 6 * 4 mod 13 = 11
Decrypted message: 11
Verification: Original message and decrypted message match!
```

2. Edge Case: Small Prime:

```
--- Main Menu ---
1. RSA Algorithm
2. ElGamal Algorithm
3. Diffie-Hellman Key Exchange
4. ECC Point Operations (Addition, Doubling)
5. Key Generation (Public, Private)
6. Exit
Enter your choice: 2

===== ELGAMAL ALGORITHM =====
Enter a prime number p: 11
Prime p = 11
Enter a primitive root g (2 <= g <= 9): 2
Primitive root g = 2
Enter private key a (1 <= a <= 9): 3
Computing public key h = g^a mod p
h = 2^3 mod 11 = 8

ElGamal Parameters and Keys:
Public Parameters: p = 11, g = 2
Public Key: h = 8
Private Key: a = 3

Enter a message (a number less than 11): 7
Original message: 7
Enter an ephemeral key k (1 <= k <= 9): 9
Computing C1 = g^k mod p
C1 = 2^9 mod 11 = 6
Computing shared secret s = h^k mod p
s = 8^9 mod 11 = 7
Computing C2 = m * s mod p
C2 = 7 * 7 mod 11 = 5

Encrypted message: (C1, C2) = (6, 5)

Decryption:
Computing shared secret s' = C1^a mod p
s' = 6^3 mod 11 = 7
Computing modular inverse of s' using Fermat's Little Theorem: s'^(p-2) mod p
s'^(p-2) = 7^9 mod 11 = 8
Computing m = C2 * s'^(p-2) mod p
m = 5 * 8 mod 11 = 7
Decrypted message: 7
Verification: Original message and decrypted message match!
```

3 . Invalid Input :

a. non prime input:

```
--- Main Menu ---
1. RSA Algorithm
2. ElGamal Algorithm
3. Diffie-Hellman Key Exchange
4. ECC Point Operations (Addition, Doubling)
5. Key Generation (Public, Private)
6. Exit
Enter your choice: 2

===== ELGAMAL ALGORITHM =====
Enter a prime number p: 14
Not a prime number. Please enter a prime number:
```

b. invalid primitive root :

```
--- Main Menu ---
1. RSA Algorithm
2. ElGamal Algorithm
3. Diffie-Hellman Key Exchange
4. ECC Point Operations (Addition, Doubling)
5. Key Generation (Public, Private)
6. Exit
Enter your choice: 2

===== ELGAMAL ALGORITHM =====
Enter a prime number p: 14
Not a prime number. Please enter a prime number: 11
Prime p = 11
Enter a primitive root g (2 <= g <= 9): 55
Invalid g. Please enter a value between 2 and 9:
```

c.Message Input greater than p:

```
===== ELGAMAL ALGORITHM =====
Enter a prime number p: 14
Not a prime number. Please enter a prime number: 11
Prime p = 11
Enter a primitive root g (2 <= g <= 9): 55
Invalid g. Please enter a value between 2 and 9: 5
Primitive root g = 5
Enter private key a (1 <= a <= 9): 7
Computing public key h = g^a mod p
h = 5^7 mod 11 = 3

ElGamal Parameters and Keys:
Public Parameters: p = 11, g = 5
Public Key: h = 3
Private Key: a = 7

Enter a message (a number less than 11): 13
Message must be less than 11. Please enter again:
```

4. Invalid ephemeral key:

```
Enter a message (a number less than 11): 13
Message must be less than 11. Please enter again: 4
Original message: 4
Enter an ephemeral key k (1 <= k <= 9): 8
Invalid k. Please enter a value between 1 and 9 that is coprime with 10: 6
Invalid k. Please enter a value between 1 and 9 that is coprime with 10: 7
Computing C1 = g^k mod p
C1 = 5^7 mod 11 = 3
Computing shared secret s = h^k mod p
s = 3^7 mod 11 = 9
Computing C2 = m * s mod p
C2 = 4 * 9 mod 11 = 3

Encrypted message: (C1, C2) = (3, 3)

Decryption:
Computing shared secret s' = C1^a mod p
s' = 3^7 mod 11 = 9
Computing modular inverse of s' using Fermat's Little Theorem: s'^(p-2) mod p
s'^(p-2) = 9^9 mod 11 = 5
Computing m = C2 * s'^(p-2) mod p
m = 3 * 5 mod 11 = 4
Decrypted message: 4
Verification: Original message and decrypted message match!
```

Diffie-Hellman Key Exchange:

1. Basic Key Exchange:

```
--- Main Menu ---
1. RSA Algorithm
2. ElGamal Algorithm
3. Diffie-Hellman Key Exchange
4. ECC Point Operations (Addition, Doubling)
5. Key Generation (Public, Private)
6. Exit
Enter your choice: 3

===== DIFFIE-HELLMAN KEY EXCHANGE =====
Enter a prime number p: 31
Prime p = 31
Enter a primitive root g (2 <= g <= 29): 22
Primitive root g = 22

Public parameters: p = 31, g = 22

--- User A ---
Enter User A's private key (1 <= a <= 29): 14
Computing User A's public key: A = g^a mod p
A = 22^14 mod 31 = 7

--- User B ---
Enter User B's private key (1 <= b <= 29): 23
Computing User B's public key: B = g^b mod p
B = 22^23 mod 31 = 3

--- Shared Secret Computation ---
User A receives User B's public key: B = 3
User A computes shared secret: s = B^a mod p
s = 3^14 mod 31 = 10
User B receives User A's public key: A = 7
User B computes shared secret: s = A^b mod p
s = 7^23 mod 31 = 10

Verification: Both users have computed the same shared secret: 10
Diffie-Hellman key exchange completed successfully!
```

2. Edge Case: Small Prime:

```
--- Main Menu ---
1. RSA Algorithm
2. ElGamal Algorithm
3. Diffie-Hellman Key Exchange
4. ECC Point Operations (Addition, Doubling)
5. Key Generation (Public, Private)
6. Exit
Enter your choice: 3

===== DIFFIE-HELLMAN KEY EXCHANGE =====
Enter a prime number p: 11
Prime p = 11
Enter a primitive root g (2 <= g <= 9): 2
Primitive root g = 2

Public parameters: p = 11, g = 2

--- User A ---
Enter User A's private key (1 <= a <= 9): 3
Computing User A's public key: A = g^a mod p
A = 2^3 mod 11 = 8

--- User B ---
Enter User B's private key (1 <= b <= 9): 7
Computing User B's public key: B = g^b mod p
B = 2^7 mod 11 = 7

--- Shared Secret Computation ---
User A receives User B's public key: B = 7
User A computes shared secret: s = B^a mod p
s = 7^3 mod 11 = 2
User B receives User A's public key: A = 8
User B computes shared secret: s = A^b mod p
s = 8^7 mod 11 = 2

Verification: Both users have computed the same shared secret: 2
Diffie-Hellman key exchange completed successfully!
```

3. Invalid Input:

a. non prime input:

```
--- Main Menu ---
1. RSA Algorithm
2. ElGamal Algorithm
3. Diffie-Hellman Key Exchange
4. ECC Point Operations (Addition, Doubling)
5. Key Generation (Public, Private)
6. Exit
Enter your choice: 3

===== DIFFIE-HELLMAN KEY EXCHANGE =====
Enter a prime number p: 14
Not a prime number. Please enter a prime number:
```

b. Invalid primitive root:

```
===== DIFFIE-HELLMAN KEY EXCHANGE =====
Enter a prime number p: 14
Not a prime number. Please enter a prime number: 31
Prime p = 31
Enter a primitive root g (2 <= g <= 29): 45
Invalid g. Please enter a value between 2 and 29:
```

c. Invalid public key:

```
Enter a prime number p: 14
Not a prime number. Please enter a prime number: 31
Prime p = 31
Enter a primitive root g (2 <= g <= 29): 45
Invalid g. Please enter a value between 2 and 29: 20
Primitive root g = 20

Public parameters: p = 31, g = 20

--- User A ---
Enter User A's private key (1 <= a <= 29): 66
Invalid a. Please enter a value between 1 and 29: 4
Computing User A's public key: A = g^a mod p
A = 20^4 mod 31 = 9
```

ECC Operations:

For the curve $E_{11}(1,6)$ **,** I will be displaying ECC Arithmetic Operations :

```
--- Main Menu ---
1. RSA Algorithm
2. ElGamal Algorithm
3. Diffie-Hellman Key Exchange
4. ECC Point Operations (Addition, Doubling)
5. Key Generation (Public, Private)
6. Exit
Enter your choice: 4

===== ELLIPTIC CURVE CRYPTOGRAPHY OPERATIONS =====
Enter the prime modulus p: 11
Enter coefficient a for the curve y^2 = x^3 + ax + b (mod 11): 1
Enter coefficient b for the curve y^2 = x^3 + ax + b (mod 11): 6

Elliptic Curve: y^2 = x^3 + 1x + 6 (mod 11)
```

1.Checking if the point is on the curve:

```
--- ECC Operations Menu ---
1. Check if a point is on the curve
2. Point Addition
3. Point Doubling
4. Scalar Multiplication
5. Return to main menu
Enter your choice: 1
Enter point coordinates (x, y):
x: 2
y: 4
The point (2, 4) is on the curve.

--- ECC Operations Menu ---
1. Check if a point is on the curve
2. Point Addition
3. Point Doubling
4. Scalar Multiplication
5. Return to main menu
Enter your choice: 1
Enter point coordinates (x, y):
x: 5
y: 6
The point (5, 6) is NOT on the curve.
```

2. Point Addition:

```
--- ECC Operations Menu ---
1. Check if a point is on the curve
2. Point Addition
3. Point Doubling
4. Scalar Multiplication
5. Return to main menu
Enter your choice: 2
Enter coordinates for point P:
x1: 2
y1: 4
Enter coordinates for point Q:
x2: 2
y2: 7
P + Q = point at infinity (P = -Q)
```

one or both points not on curve:

```
--- ECC Operations Menu ---
1. Check if a point is on the curve
2. Point Addition
3. Point Doubling
4. Scalar Multiplication
5. Return to main menu
Enter your choice: 2
Enter coordinates for point P:
x1: 5
y1: 2
Enter coordinates for point Q:
x2: 8
y2: 9
Error: One or both points are not on the curve.
```

3. Point Doubling:

```
--- ECC Operations Menu ---
1. Check if a point is on the curve
2. Point Addition
3. Point Doubling
4. Scalar Multiplication
5. Return to main menu
Enter your choice: 3
Enter coordinates for point P:
x: 2
y: 4
Computing lambda for point doubling:
λ = (3x_P^2 + a) / (2y_P) mod p
λ = (3*2^2 + 1) / (2*4) mod 11
λ = 2 / 8 mod 11
λ = 2 * 7 mod 11 = 3
Computing result coordinates:
x_R = λ^2 - 2x_P mod p
x_R = 3^2 - 2*2 mod 11 = 5
y_R = λ(x_P - x_R) - y_P mod p
y_R = 3(2 - 5) - 4 mod 11 = 9
2P = (5, 9)
Verification: The resulting point is on the curve.
```

4.Scalar Multiplication:

```
--- ECC Operations Menu ---
1. Check if a point is on the curve
2. Point Addition
3. Point Doubling
4. Scalar Multiplication
5. Return to main menu
Enter your choice: 4
Enter coordinates for point P:
x: 2
y: 7
Enter scalar k (positive integer): 3

Computing 3P using double-and-add algorithm:
Start with R = Point at infinity (identity element)
k = 3 is odd, so add current point to result
Result = (2, 7)
Double the temporary point
Temp = (5, 2)
k = 1 is odd, so add current point to result
Result = (8, 3)
Double the temporary point
Temp = (10, 2)

Final result kP = (8, 3)
Verification: The resulting point is on the curve.
```

point not on curve:

```
--- ECC Operations Menu ---
1. Check if a point is on the curve
2. Point Addition
3. Point Doubling
4. Scalar Multiplication
5. Return to main menu
Enter your choice: 4
Enter coordinates for point P:
x: 2
y: 3
Enter scalar k (positive integer): 5
Error: The point is not on the curve.
```

5.Key Generation:

1. RSA key generation by generating and inputing primes :

```
--- Main Menu ---
1. RSA Algorithm
2. ElGamal Algorithm
3. Diffie-Hellman Key Exchange
4. ECC Point Operations (Addition, Doubling)
5. Key Generation (Public, Private)
6. Exit
Enter your choice: 5

===== KEY GENERATION FOR ASYMMETRIC ALGORITHMS =====

--- Key Generation Menu ---
1. RSA Key Generation
2. ElGamal Key Generation
3. ECC Key Generation
4. Return to main menu
Enter your choice: 1

--- RSA Key Generation ---
Generating prime numbers p and q...
Do you want to (e)nter prime numbers or (g)enerate them? (e/g): g
Prime p = 439
Prime q = 787
Computing n = p * q = 439 * 787 = 345493
Computing phi(n) = (p-1) * (q-1) = 438 * 786 = 344268
Choosing public exponent e...
Public exponent e = 5
Computing private exponent d...
Private exponent d = 206561 (the modular inverse of e mod phi(n))
Verification: (d * e) % phi(n) = 1 (should be 1)

RSA Keys generated successfully!
Public Key (e, n) = (5, 345493)
Private Key (d, n) = (206561, 345493)
```

```
--- RSA Key Generation ---
Generating prime numbers p and q...
Do you want to (e)nter prime numbers or (g)enerate them? (e/g): e
Enter first prime number (p): 3
Enter second prime number (q): 5
Prime p = 3
Prime q = 5
Computing n = p * q = 3 * 5 = 15
Computing phi(n) = (p-1) * (q-1) = 2 * 4 = 8
Choosing public exponent e...
Enter public exponent e (1 < e < 8 and gcd(e, 8) = 1): 7
Public exponent e = 7
Computing private exponent d...
Private exponent d = 7 (the modular inverse of e mod phi(n))
Verification: (d * e) % phi(n) = 1 (should be 1)

RSA Keys generated successfully!
Public Key (e, n) = (7, 15)
Private Key (d, n) = (7, 15)
```

2. *Elgamal key generation :*

```
--- Key Generation Menu ---
1. RSA Key Generation
2. ElGamal Key Generation
3. ECC Key Generation
4. Return to main menu
Enter your choice: 2

--- ElGamal Key Generation ---
Enter a prime number p: 13
Prime p = 13
Enter a primitive root g (2 <= g <= 11): 4
Primitive root g = 4
Enter private key a (1 <= a <= 11): 5
Computing public key h = g^a mod p
h = 4^5 mod 13 = 10

ElGamal Parameters and Keys:
Public Parameters: p = 13, g = 4
Public Key: h = 10
Private Key: a = 5
```

3. *ECC Key Generation:*

```
--- Key Generation Menu ---
1. RSA Key Generation
2. ElGamal Key Generation
3. ECC Key Generation
4. Return to main menu
Enter your choice: 3

--- ECC Key Generation ---
Enter the prime modulus p: 11
Enter coefficient a for the curve y^2 = x^3 + ax + b (mod 11): 1
Enter coefficient b for the curve y^2 = x^3 + ax + b (mod 11): 6

Elliptic Curve: y^2 = x^3 + 1x + 6 (mod 11)
Enter coordinates for base point G:
x: 2
y: 7
Base point G = (2, 7)
Enter a private key d (a large positive integer): 4
Computing public key Q = dG...
Public key Q = (10, 2)

ECC Keys generated successfully!
Private Key: d = 4
Public Key: Q = (10, 2)
```