Lab No: 03

Lab Name: Implementation of the Extended Euclidean Algorithm

Objectives:

1. To implement the **Extended Euclidean Algorithm** to find the **Greatest Common Divisor (GCD)** of two integers.
2. To determine the **Bézout coefficients (x, y)** that satisfy the equation $ax + by = \text{GCD}(a, b)$.

Theory:

The **Extended Euclidean Algorithm (EEA)** is an extension of the **Euclidean Algorithm** that not only determines the **Greatest Common Divisor (GCD)** of two integers a and b, but also finds two integers x and y (called **Bézout coefficients**) that satisfy the linear equation:

$$ax + by = \text{GCD}(a, b)$$

This algorithm is widely used in **cryptography and network security**, particularly in finding **modular inverses**, which are essential in systems such as **RSA encryption**, **digital signatures**, and **key exchange protocols**.

Algorithm Steps:

1. **Input:** Two integers a and b.
2. **If** $a = 0$: return $(b, 0, 1)$, since $\text{GCD}(a, b) = b$.
3. **Else:** recursively apply the algorithm on $(b \bmod a, a)$.
4. **Back substitution:**

$$x = y_1 - \left\lfloor \frac{b}{a} \right\rfloor \times x_1$$

$$y = x_1$$

5. **Return:** (GCD, x, y)

Example:

Let $a = 35$ and $b = 21$.

Steps:

$$35 = 21(1) + 14$$

$$21 = 14(1) + 7$$

$$14 = 7(2) + 0$$

So, **GCD(35, 21) = 7**.

Now, back-substituting:

$$7 = 21 - 14(1) = 21 - (35 - 21)(1) = 2(21) - 1(35)$$

Hence,

$$x = -1, \quad y = 2$$

and

$$35(-1) + 21(2) = 7$$

Thus, the Extended Euclidean Algorithm gives **GCD = 7**, **x = -1**, and **y = 2**.

Result and Discussion:

The experiment successfully computes the **GCD** and **Bézout coefficients** using the Extended Euclidean Algorithm.
For the given example $a = 35$ and $b = 21$, the result obtained is:

$$\text{GCD} = 7, \quad x = -1, \quad y = 2$$

This verifies the relation $35(-1) + 21(2) = 7$.
The algorithm efficiently finds integer solutions that satisfy the linear combination equation.
In **cryptography**, these coefficients are crucial for computing **modular inverses**, which are used in **RSA encryption/decryption** and other **key-based security algorithms**.

Objectives:

1. To study and understand the concept of **modular arithmetic** and its fundamental properties.
2. To demonstrate modular operations using **primitive roots** for encryption and cryptographic purposes.

Theory:

Modular arithmetic is a system of arithmetic for integers where numbers "wrap around" after reaching a certain value called the **modulus**. It is often described as **clock arithmetic**, because once the value reaches the modulus, it resets to zero.

For integers a, b, and modulus n:

$$a \equiv b \pmod{n}$$

means that n divides $(a - b)$, or equivalently,

$$a \mod n = b \mod n$$

Primitive Root:

An integer g is called a **primitive root modulo p** (where p is prime) if its powers generate all integers from 1 to $p - 1$ modulo p. Formally:

$$g^1, g^2, g^3, \ldots, g^{p-1} \mod p$$

produces all integers $1, 2, \ldots, p - 1$ exactly once.

Primitive roots are widely used in **cryptography**, particularly in **Diffie–Hellman key exchange** and modular exponentiation exercises, because they demonstrate modular arithmetic properties and ensure reversibility of operations.

Algorithm Steps Using Primitive Roots:

1. **Choose a prime modulus** p.
2. **Find a primitive root** g modulo p.
3. **Compute powers of** g **modulo** p:

$$g^1 \mod p, g^2 \mod p, \ldots, g^{p-1} \mod p$$

4. **Verify** that all integers from 1 to $p - 1$ appear exactly once.
5. Use the results to demonstrate modular arithmetic properties: addition, multiplication, and exponentiation.

Example:

Let $p = 7$ (prime) and $g = 3$ (primitive root modulo 7).

Compute powers modulo 7:

Power	Calculation	Result mod 7
1	3^1	3
2	3^2	2
3	3^3	6
4	3^4	4
5	3^5	5
6	3^6	1

✅ All numbers from 1 to 6 appear exactly once → 3 is confirmed as a primitive root modulo 7.

This example demonstrates **modular exponentiation** and shows that primitive roots generate all residues modulo p. Such results form the basis for cryptographic algorithms like **Diffie–Hellman key exchange** and **RSA-like modular operations**.

Result and Discussion:

The experiment successfully demonstrates modular arithmetic properties using a **primitive root modulo a prime number**.

For $p = 7$ and $g = 3$, the computed powers modulo p are:

$$3^1 \mod 7 = 3, \quad 3^2 \mod 7 = 2, \quad 3^3 \mod 7 = 6$$
$$3^4 \mod 7 = 4, \quad 3^5 \mod 7 = 5, \quad 3^6 \mod 7 = 1$$

These results confirm that **all integers from 1 to 6 are generated exactly once**, verifying that 3 is indeed a primitive root modulo 7.

The experiment illustrates that **primitive roots** provide a systematic way to explore modular arithmetic operations. In **cryptography**, these roots are crucial for generating keys, performing secure modular exponentiation, and ensuring that all residues modulo p can be represented uniquely.

Lab No: 05

Lab Name: Implementation and Understanding of RSA Algorithm

Objectives:

1. To understand the **RSA algorithm** and its role in **public-key cryptography**.
2. To perform RSA **encryption and decryption** processes using small example numbers to demonstrate the algorithm.

Theory:

The **RSA algorithm** is a widely used **asymmetric cryptographic system** that relies on the difficulty of **factoring large prime numbers**.
It uses a **pair of keys**: a **public key** for encryption and a **private key** for decryption.

The steps of the RSA algorithm are as follows:

1. **Key Generation:**
   - Choose two large prime numbers p and q.
   - Compute $n = p \times q$ (the modulus).
   - Compute Euler's totient function:

$$\phi(n) = (p-1)(q-1)$$

   - Choose an encryption key e such that $1 < e < \phi(n)$ and $\text{GCD}(e, \phi(n)) = 1$.
   - Compute the decryption key d such that:

$$e \times d \equiv 1 \pmod{\phi(n)}$$

2. **Encryption:**
   For a plaintext message M, the ciphertext C is computed as:

$$C = M^e \mod n$$

3. **Decryption:**
   Recover the original message M from ciphertext C using:

$$M = C^d \mod n$$

RSA security is based on the fact that **factoring $n = p \times q$ is computationally difficult**, making it secure for practical use.

Example:

Let $p = 5, q = 11.$

1. Compute $n = 5 \times 11 = 55$
2. Compute $\phi(n) = (5-1)(11-1) = 4 \times 10 = 40$
3. Choose $e = 7$ (coprime with 40)
4. Find d such that $7 \times d \equiv 1 \pmod{40} \rightarrow d = 23$

Public key: $(e, n) = (7, 55)$
Private key: $(d, n) = (23, 55)$

Encrypt a message $M = 9$:

$$C = 9^7 \mod 55 = 9$$

Decrypt the ciphertext $C = 9$:

$$M = 9^{23} \mod 55 = 9$$

The original message is recovered successfully.

Result and Discussion:

The experiment demonstrates the complete RSA process including **key generation**, **encryption**, and **decryption**.

- Public key $(7, 55)$ and private key $(23, 55)$ were computed correctly.
- Encryption and decryption of message $M = 9$ confirmed that RSA maintains data integrity and confidentiality.
- This shows how **modular arithmetic** and the **difficulty of factoring large numbers** form the security foundation of RSA.
- In real-world applications, much larger prime numbers are used to ensure **high-level security**.

Theory:

The **Diffie–Hellman Key Exchange Algorithm** enables two users to establish a shared secret key over an insecure communication channel.
The security of the algorithm depends on the **difficulty of the discrete logarithm problem**.

Algorithm Steps:

1. Global Public Elements

- Let q be a large prime number.
- Let α be a primitive root of q, such that $\alpha < q$.
  Both q and α are public values.

---

2. User A Key Generation

- Select private key X_A where $X_A < q$.
- Compute public key:

$$Y_A = \alpha^{X_A} \mod q$$

---

3. User B Key Generation

- Select private key X_B where $X_B < q$.
- Compute public key:

$$Y_B = \alpha^{X_B} \mod q$$

---

4. Calculation of Secret Key by User A

$$K = (Y_B)^{X_A} \mod q$$

---

5. Calculation of Secret Key by User B

$$K = (Y_A)^{X_B} \mod q$$

Both users will obtain the **same secret key** K because:

$$(Y_B)^{X_A} \mod q = (Y_A)^{X_B} \mod q = \alpha^{X_A X_B} \mod q$$

Example:

Let:

$$q = 23, \quad \alpha = 5$$

- User A chooses $X_A = 6$
- User B chooses $X_B = 15$

Compute public keys:

$$Y_A = 5^6 \mod 23 = 8$$

$$Y_B = 5^{15} \mod 23 = 19$$

Compute secret keys:

$$K_A = (Y_B)^{X_A} \mod 23 = 19^6 \mod 23 = 2$$

$$K_B = (Y_A)^{X_B} \mod 23 = 8^{15} \mod 23 = 2$$

✅ Both users obtain the same secret key $K = 2$.

Result and Discussion:

The experiment successfully demonstrates that both users independently compute the **same shared secret key** without transmitting it.
This confirms the correctness of the **Diffie–Hellman Key Exchange Algorithm** and its practical use in cryptographic systems such as SSL/TLS and VPNs.
The security of this algorithm lies in the **difficulty of solving the discrete logarithm problem**.

Theory

DES is a symmetric key encryption technique that uses substitution, permutation, and bitwise operations to hide data. One of its most important operations is the XOR operation. XOR combines the plaintext with the key bits in every round. The reason XOR is useful is that it is reversible. If you XOR the encrypted data with the same key again, you get back the original data.

This experiment does not implement full DES. Instead, it demonstrates a simplified DES-like idea using XOR. In real DES, plaintext goes through 16 rounds where the right half is expanded, XORed with a round key, passed through S-boxes, permuted and combined with the left half. Even though our program is much simpler, it shows the same core idea: mixing data with a key using XOR.

Algorithm (XOR-Based Encryption and Decryption)

Input:

Key `k`, plaintext `t`

Process:

1. Read the key from the user.
2. Read the plaintext text from the user.
3. For encryption:
    - Convert each character of plaintext to ASCII.
    - Convert each character of the key to ASCII.
    - XOR each plaintext character with a key character.
    - If the key is shorter, repeat key characters using `i % len(k)`.
    - Store each XOR result in bytes and convert to hexadecimal.
4. For decryption:
    - Take each encrypted byte.
    - XOR it with the same repeating key character.
    - Convert the final bytes back into characters to recover the plaintext.

Output:

Encrypted data in hexadecimal form and the decrypted original text.

This algorithm models how DES uses XOR to mix key bits with data and still allow reversal using the same key.

Theory:

The **ElGamal Cryptographic Algorithm** is an **asymmetric key encryption algorithm** based on the principles of **Discrete Logarithm Problem (DLP)**.
It was proposed by **Taher ElGamal** in 1985 and is widely used in secure communication systems.

ElGamal uses two keys:

- A **public key** for encryption.
- A **private key** for decryption.

The security of this algorithm depends on the **difficulty of finding discrete logarithms in a finite field**, which makes it computationally infeasible to deduce the private key from the public key.

Algorithm Steps:

1. Key Generation:

- Choose a large prime number p.
- Choose a primitive root g of p.
- Select a random private key x, where $1 < x < p - 1$.
- Compute public key:

$$y = g^x \mod p$$

Public key = (p, g, y)
Private key = x

---

2. Encryption:

To encrypt a message M:

- Choose a random integer k (ephemeral key) such that $1 < k < p - 1$.
- Compute:

$$C_1 = g^k \mod p$$

$$C_2 = (M \times y^k) \mod p$$

Ciphertext = (C_1, C_2)

3. Decryption:

To recover M:

$$M = (C_2 \times (C_1^{p-1-x})) \mod p$$