DEPARTMENT OF CSE-CYBER SECURITY

NETWORK SECURITY LAB MANUAL

BCYL606 – PCCL

(Effective from the academic year 2023-24)

(Academic year 2024-2025)

Semester –VI

Prepared By: Mrs. Sujatha J V

Assistant Professor

Department of CSE-Cyber Security

ACSCE, Bengaluru

Mission

To Empower students with hands-on facilities, the essentials of Cyber Security and a strong ethical foundation.
To facilitate learning environment, teamwork, and global certifications for real-time challenges.
To foster a culture to possess qualities of interpersonal, interdisciplinary, leadership and societal responsibilities.

Program Educational Objectives (PEOs)

A Graduate of CSE-Cyber Security will be able to:

PEO1: Understand the Cyber Security threat landscape and various types of cyber-attacks, cybercrimes, and vulnerabilities.
PEO2: Analyse ethical, legal, and regulatory impact of computer systems on organizations, society, and the individual.
PEO3: Identify risks, assess threats, and apply cyber security skills to construct secure systems from system to human-computer interface.
PEO4: Continuously engage in activities that foster their computing and cybersecurity skills to stay ahead of emerging technologies and evolving threats.

Program Specific Outcomes (PSOs)

CSE-Cyber Security graduates will have

PSO1: Savvy skill in applying principles of software development to redeem efficient software solutions.
PSO2: A proactive approach to Cyber Security, staying updated on emerging threats and industry best practices to maintain a strong security posture.

Introduction and Scope of Network Security Lab (BCYL606)

Introduction:

Network Security Lab (BCYL606) is a hands-on course designed to provide students with practical exposure to essential network security concepts and techniques. It covers key topics such as cryptographic algorithms, firewall configurations, intrusion detection and prevention systems, vulnerability assessment, and secure communication protocols. Through real-world simulations and lab exercises, students develop critical skills to analyze, implement, and evaluate security mechanisms in modern network infrastructures.

Cryptography is the science of securing communication and data through mathematical techniques. It plays a crucial role in ensuring confidentiality, integrity, authentication, and non-repudiation in digital communication. The **Network Security Lab (BCYL606)** focuses on practical implementations of cryptographic algorithms and security protocols, enabling students to understand how encryption, hashing, and authentication mechanisms protect information from cyber threats.

Scope:

This course provides hands-on experience in:

- Classical ciphers such as **Caesar, Monoalphabetic, Polyalphabetic, Playfair, and Hill ciphers** demonstrate basic encryption principles
- **single and double transposition techniques** enhance security through permutation.
- Modern cryptographic methods include **DES for symmetric encryption and RSA for asymmetric encryption**, ensuring secure data transmission.
- The **Diffie-Hellman key exchange** facilitates secure key sharing
- **Fermat's and Euler's theorems** provide mathematical foundations for cryptographic algorithms.
- **pseudo-random number generators (PRNGs)** such as the **Linear Congruential Method and Blum Blum Shub Generator** are implemented to enhance cryptographic security.

This hands-on approach strengthens the understanding of cryptography and its applications in securing digital communication.

The lab-based approach enhances students' ability to analyze, implement, and troubleshoot cryptographic systems, preparing them for careers in cybersecurity, ethical hacking, and secure software development.

VISVESVARAYA TECHNOLOGICAL UNIVERSITY, BELAGAVI

B.E. in CSE (Cyber Security)

Scheme of Teaching and Examinations 2022

Outcome Based Education (OBE) and Choice Based Credit System (CBCS)

(Effective from the academic year 2023-24)

VI SEMESTER

Sl. No	Course and Course Code		Course Title	Teaching Department (TD) and Question Paper Setting Board (PSB)	Teaching Hours /Week				Examination				Credits
					Theory Lecture	Tutorial	Practical / Drawing	SDA	Duration in hours	CIE Marks	SEE Marks	Total Marks	
					L	T	P	S					
1	IPCC	BCO601	Microcontrollers & Embedded Systems	TD: CY PSB : CY	3	0	2		03	50	50	100	4
2	PCC	BCY602	Cryptography & Network Security	TD: CY PSB : CY	4	0	0		03	50	50	100	4
3	PEC	BXX613x	Professional Elective Course	TD: CY PSB : CY	3	0	0		03	50	50	100	3
4	OEC	BXX654x	Open Elective Course	TD: CY PSB : CY	3	0	0		03	50	50	100	3
5	PROJ	BCY685	Project Phase I	TD: CY PSB : CY	0	0	4		03	100	--	100	2
6	PCCL	BCYL606	Network Security Lab	TD: CY PSB : CY	0	0	2		03	50	50	100	1
7	AEC/SDC	BXX657x	Ability Enhancement Course/Skill Development Course V	TD and PSB: Concerned department	If the course is offered as a Theory: 1	0	0		01	50	50	100	1
					If course is offered as a practical: 0	0	2						
8	MC	BNSK658	National Service Scheme (NSS)	NSS coordinator	0	0	2			100	---	100	0
		BPEK658	Physical Education (PE) (Sports and Athletics)	Physical Education Director									
		BYOK658	Yoga	Yoga Teacher									
9	MC	BIKS609	Indian Knowledge System		1	0	0		01	100	---	100	0
									Total	**500**	**300**	**800**	**18**

Network Security Lab		Semester	**VI**
Course Code	**BCYL606**	CIE Marks	50
Teaching Hours/Week (L:T:P: S)	0:0:2:0	SEE Marks	50
Credits	01	Exam Hours	100
Examination type (SEE)	Practical		

Course outcomes:

At the end of the course the student will be able to:

- Implement Substitution Ciphers
- Design the various Transposition Techniques
- Demonstrate the working of various Symmetric Cipher algorithms
- Demonstrate the working of various aymmetric Cipher algorithms

CO- PO Mapping

	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO 1	3	2	3	2	3	1	1				2	3
CO 2	3	3	3	2	3	1	1				2	3
CO 3	3	3	3	3	3	2	2				3	3
CO 4	3	3	3	3	3	2	2				3	3

Assessment Methods CIE – 50 Marks (Avg (A) + (B / 20))

A. For each Experiment Total – 30 Marks:

- Observation Book – 10 Marks
- Conduction – 10 Marks
- Record – 10 Marks

B. Lab Test – 100 Marks

- test write-up (15M) , conduction of experiment (15M), Result (15M), and procedural knowledge (15M) Total – 60 Marks
- Viva – 40 Marks

List of Experiments

Sl.NO	Experiments (Implement using C/C++/Java Programming Languages)
1	Implement Caesar Cipher substitution method
2	Implement Mono alphabetic cipher with the given key and input
3	Implement Poly alphabetic Cipher
4	Encrypt the given text using Play fair Cipher with the given key. Also perform the decryption operation
5	Implement Encryption and Decryption techniques in Hill Cipher
6	Demonstrate Single and Double Transposition techniques
7	Implement Simple DES/DES Algorithm
8	Generate Pseudo random numbers using Linear Congruential method
9	Generate Pseudo random numbers using Blum Blum Shub Generator
10	Implement RSA Algorithm
11	Demonstrate Diffie Hellman Key exchange Algorithm
12	Implement Fermat's and Euler's Theorem (Course Instructor need to explain the theorem before execution as the topic not covered in Theory part)

Experiment 1

Implement Caesar Cipher substitution method

Algorithm for Encryption

1. Input: A plaintext string and a shift key (integer).
2. Process:
    - Iterate through each character in the plaintext.
    - If the character is a letter:
        - Shift it by the key value while maintaining its case (uppercase/lowercase).
        - Wrap around if necessary (e.g., 'Z' shifts to 'A').
    - If it's not a letter, leave it unchanged.
3. Output: The encrypted ciphertext**.**

Caesar Cipher Decryption

To decrypt a message encrypted using the Caesar Cipher, we simply shift the characters in the opposite direction (i.e., subtract the shift value instead of adding it).

Algorithm for Decryption

1. Input: The ciphertext and the shift key (integer).
2. Process:
    - Iterate through each character in the ciphertext.
    - If the character is a letter:
        - Shift it backward by the key value while maintaining its case (uppercase/lowercase).
        - Wrap around if necessary (e.g., 'A' shifts to 'Z').
    - If it's not a letter, leave it unchanged.
3. Output: The decrypted plaintext.

C Program for Caesar Cipher Encryption & Decryption

```
#include <stdio.h>
#include <ctype.h>

// Function to encrypt text using Caesar Cipher
void caesar_cipher(char *text, int shift) {
    for (int i = 0; text[i] != '\0'; i++) {
        char ch = text[i];

        if (isalpha(ch)) { // Check if it's a letter
            char base = isupper(ch) ? 'A' : 'a';
            text[i] = (ch - base + shift) % 26 + base; // Shift and wrap around
        }
    }
}

// Function to decrypt text using Caesar Cipher
void caesar_decipher(char *text, int shift) {
    caesar_cipher(text, 26 - shift); // Decryption is shifting in the opposite direction
}
```

```
int main() {
    char text[100];
    int shift;

    // Input message and shift value
    printf("Enter a string: ");
    fgets(text, sizeof(text), stdin);

    printf("Enter shift value: ");
    scanf("%d", &shift);

    // Encrypt the text
    caesar_cipher(text, shift);
    printf("Encrypted text: %s\n", text);

    // Decrypt the text
    caesar_decipher(text, shift);
    printf("Decrypted text: %s\n", text);

    return 0;
}
```

Output

```
Enter a string: Hello World
Enter shift value: 3
Encrypted text: Khoor Zruog
Decrypted text: Hello World
```

Experiment 2

Implement Mono alphabetic cipher with the given key and input

Algorithm for Monoalphabetic Cipher Encryption & Decryption

Encryption

1. **Input**: A plaintext string and a predefined key (substitution mapping).
2. **Process**:
   - Replace each letter in the plaintext with its corresponding letter from the key.
   - Preserve the case of the letters.
   - Ignore non-alphabetic characters.
3. **Output**: The encrypted ciphertext.

Decryption

1. **Input**: The ciphertext and the predefined key.
2. **Process**:
   - Replace each letter in the ciphertext with its corresponding letter from the original alphabet.
   - Preserve case and ignore non-alphabetic characters.
3. **Output**: The decrypted plaintext.

C Program for Monoalphabetic Cipher

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

// Define a monoalphabetic substitution key
char key[] = "QWERTYUIOPASDFGHJKLZXCVBNM"; // Custom mapping for uppercase letters
char reverse_key[26]; // Reverse mapping for decryption

// Function to generate the reverse key mapping
void generate_reverse_key() {
    for (int i = 0; i < 26; i++) {
        reverse_key[key[i] - 'A'] = 'A' + i;
    }
}

// Function to encrypt text using Monoalphabetic Cipher
void encrypt_monoalphabetic(char *text) {
    for (int i = 0; text[i] != '\0'; i++) {
        if (isalpha(text[i])) {
            int is_upper = isupper(text[i]);
            char base = is_upper ? 'A' : 'a';
            int index = text[i] - base;
            text[i] = is_upper ? key[index] : tolower(key[index]);
        }
    }
}
```

```
// Function to decrypt text using Monoalphabetic Cipher
void decrypt_monoalphabetic(char *text) {
    for (int i = 0; text[i] != '\0'; i++) {
        if (isalpha(text[i])) {
            int is_upper = isupper(text[i]);
            char base = is_upper ? 'A' : 'a';
            int index = strchr(key, toupper(text[i])) - key;
            text[i] = is_upper ? reverse_key[index] : tolower(reverse_key[index]);
        }
    }
}

int main() {
    char text[100];

    // Generate reverse key for decryption
    generate_reverse_key();

    // Input text
    printf("Enter a string: ");
    fgets(text, sizeof(text), stdin);
    text[strcspn(text, "\n")] = '\0'; // Remove newline character

    // Encrypt the text
    encrypt_monoalphabetic(text);
    printf("Encrypted text: %s\n", text);

    // Decrypt the text
    decrypt_monoalphabetic(text);
    printf("Decrypted text: %s\n", text);

    return 0;
}
```

Example Run

Enter a string: HelloWorld
Encrypted text: ItssgVgksr
Decrypted text: HelloWorld

Explanation

- The **encryption function** substitutes each letter in the plaintext using a predefined key.
- The **decryption function** reverses the substitution using a reverse-mapping key.
- **Case sensitivity** is preserved, and non-alphabetic characters remain unchanged.

Experiment 3

Implement Poly alphabetic Cipher

Algorithm for Polyalphabetic Cipher (Vigenère Cipher) Encryption & Decryption

Encryption

1. **Input**: A plaintext string and a keyword.
2. **Process**:
   - Repeat the keyword to match the length of the plaintext.
   - For each letter in the plaintext:
     - Shift it forward by the corresponding keyword letter's position in the alphabet.
     - Preserve the case of letters.
     - Ignore non-alphabetic characters.
3. **Output**: The encrypted ciphertext.

Decryption

1. **Input**: The ciphertext and the keyword.
2. **Process**:
   - Repeat the keyword to match the length of the ciphertext.
   - For each letter in the ciphertext:
     - Shift it backward by the corresponding keyword letter's position in the alphabet.
     - Preserve case and ignore non-alphabetic characters.
3. **Output**: The decrypted plaintext.

C Program for Polyalphabetic Cipher (Vigenère Cipher)

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

// Function to repeat the keyword to match the text length
void generate_key(const char *text, const char *key, char *new_key) {
    int text_len = strlen(text);
    int key_len = strlen(key);
    for (int i = 0, j = 0; i < text_len; i++) {
        if (isalpha(text[i])) {
            new_key[i] = key[j % key_len];
            j++;
        } else {
            new_key[i] = text[i]; // Keep spaces or special characters unchanged
        }
    }
    new_key[text_len] = '\0';
}

// Function to encrypt using Vigenère Cipher
void encrypt_vigenere(char *text, const char *key) {
    char new_key[100];
    generate_key(text, key, new_key);

    for (int i = 0; text[i] != '\0'; i++) {
        if (isalpha(text[i])) {
            char base = isupper(text[i]) ? 'A' : 'a';
            char key_base = isupper(new_key[i]) ? 'A' : 'a';
```

```
            text[i] = (text[i] - base + (new_key[i] - key_base)) % 26 + base;
        }
    }
}

// Function to decrypt using Vigenère Cipher
void decrypt_vigenere(char *text, const char *key) {
    char new_key[100];
    generate_key(text, key, new_key);

    for (int i = 0; text[i] != '\0'; i++) {
        if (isalpha(text[i])) {
            char base = isupper(text[i]) ? 'A' : 'a';
            char key_base = isupper(new_key[i]) ? 'A' : 'a';
            text[i] = (text[i] - base - (new_key[i] - key_base) + 26) % 26 + base;
        }
    }
}

int main() {
    char text[100], key[100];

    // Input plaintext and keyword
    printf("Enter a string: ");
    fgets(text, sizeof(text), stdin);
    text[strcspn(text, "\n")] = '\0'; // Remove newline

    printf("Enter keyword: ");
    fgets(key, sizeof(key), stdin);
    key[strcspn(key, "\n")] = '\0'; // Remove newline

    // Encrypt the text
    encrypt_vigenere(text, key);
    printf("Encrypted text: %s\n", text);

    // Decrypt the text
    decrypt_vigenere(text, key);
    printf("Decrypted text: %s\n", text);

    return 0;
}
```

Example Run

Enter a string: HelloWorld
Enter keyword: KEY
Encrypted text: RiijvVgyqn
Decrypted text: HelloWorld

Explanation

- The **encryption function** shifts each letter based on the corresponding letter in the repeated keyword.
- The **decryption function** reverses the shift to restore the original text.
- **Case sensitivity** is preserved, and non-alphabetic characters remain unchanged.

Experiment 4

Encrypt the given text using Play fair Cipher with the given key. Also perform the decryption operation

Algorithm for Playfair Cipher Encryption & Decryption

Encryption

1. **Input**: A plaintext message and a keyword.
2. **Process**:
   - Generate a **5x5 Playfair matrix** using the keyword (removing duplicates, merging 'I' and 'J').
   - Divide plaintext into **digraphs (pairs of letters)**, inserting an 'X' if a duplicate appears in a pair.
   - Apply Playfair Cipher rules:
     - If letters are in the **same row**, replace each with the next letter in the row.
     - If letters are in the **same column**, replace each with the letter below.
     - Otherwise, form a **rectangle** and replace letters with the corresponding diagonal.
3. **Output**: The encrypted ciphertext.

Decryption

1. **Input**: The ciphertext and the same Playfair matrix.
2. **Process**:
   - Apply the reverse rules of encryption.
3. **Output**: The original plaintext.

C Program for Playfair Cipher

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define SIZE 5
char playfairMatrix[SIZE][SIZE];

// Function to remove duplicates and prepare the key
void prepare_key(const char *key, char *processedKey) {
    int seen[26] = {0};
    int index = 0;

    for (int i = 0; key[i] != '\0'; i++) {
        char ch = toupper(key[i]);
        if (ch == 'J') ch = 'I'; // Merge I and J
        if (isalpha(ch) && !seen[ch - 'A']) {
            seen[ch - 'A'] = 1;
            processedKey[index++] = ch;
        }
    }

    // Fill remaining letters
    for (char ch = 'A'; ch <= 'Z'; ch++) {
        if (ch == 'J') continue; // Skip J
        if (!seen[ch - 'A']) {
            processedKey[index++] = ch;
        }
    }
    processedKey[index] = '\0';
}
```

```
// Function to generate Playfair matrix
void generate_matrix(const char *processedKey) {
    int index = 0;
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            playfairMatrix[i][j] = processedKey[index++];
        }
    }
}

// Function to find position of a letter in the matrix
void find_position(char ch, int *row, int *col) {
    if (ch == 'J') ch = 'I';
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            if (playfairMatrix[i][j] == ch) {
                *row = i;
                *col = j;
                return;
            }
        }
    }
}

// Function to process text into digraphs (pairs)
void process_text(const char *text, char *processedText) {
    int len = strlen(text), index = 0;

    for (int i = 0; i < len; i++) {
        if (!isalpha(text[i])) continue;
        processedText[index++] = toupper(text[i] == 'J' ? 'I' : text[i]);

        // Insert 'X' if duplicate letters appear in the same pair
        if (index > 1 && processedText[index - 1] == processedText[index - 2]) {
            processedText[index - 1] = 'X';
            processedText[index++] = toupper(text[i] == 'J' ? 'I' : text[i]);
        }
    }

    // If odd length, append 'X'
    if (index % 2 != 0) {
        processedText[index++] = 'X';
    }
    processedText[index] = '\0';
}
```

```
// Function to encrypt using Playfair Cipher
void encrypt_playfair(char *text) {
    for (int i = 0; text[i] != '\0'; i += 2) {
        int r1, c1, r2, c2;
        find_position(text[i], &r1, &c1);
        find_position(text[i + 1], &r2, &c2);

        if (r1 == r2) { // Same row
            text[i] = playfairMatrix[r1][(c1 + 1) % SIZE];
            text[i + 1] = playfairMatrix[r2][(c2 + 1) % SIZE];
        } else if (c1 == c2) { // Same column
            text[i] = playfairMatrix[(r1 + 1) % SIZE][c1];
            text[i + 1] = playfairMatrix[(r2 + 1) % SIZE][c2];
        } else { // Rectangle rule
            text[i] = playfairMatrix[r1][c2];
            text[i + 1] = playfairMatrix[r2][c1];
        }
    }
}

// Function to decrypt using Playfair Cipher
void decrypt_playfair(char *text) {
    for (int i = 0; text[i] != '\0'; i += 2) {
        int r1, c1, r2, c2;
        find_position(text[i], &r1, &c1);
        find_position(text[i + 1], &r2, &c2);

        if (r1 == r2) { // Same row
            text[i] = playfairMatrix[r1][(c1 - 1 + SIZE) % SIZE];
            text[i + 1] = playfairMatrix[r2][(c2 - 1 + SIZE) % SIZE];
        } else if (c1 == c2) { // Same column
            text[i] = playfairMatrix[(r1 - 1 + SIZE) % SIZE][c1];
            text[i + 1] = playfairMatrix[(r2 - 1 + SIZE) % SIZE][c2];
        } else { // Rectangle rule
            text[i] = playfairMatrix[r1][c2];
            text[i + 1] = playfairMatrix[r2][c1];
        }
    }
}


int main() {
    char key[100], text[100], processedKey[26], processedText[100];

    // Input keyword
    printf("Enter keyword: ");
    fgets(key, sizeof(key), stdin);
    key[strcspn(key, "\n")] = '\0'; // Remove newline
```

```
    // Prepare the Playfair key and matrix
    prepare_key(key, processedKey);
    generate_matrix(processedKey);

    // Input plaintext
    printf("Enter text: ");
    fgets(text, sizeof(text), stdin);
    text[strcspn(text, "\n")] = '\0'; // Remove newline

    // Process text into digraphs
    process_text(text, processedText);

    // Encrypt text
    encrypt_playfair(processedText);
    printf("Encrypted text: %s\n", processedText);

    // Decrypt text
    decrypt_playfair(processedText);
    printf("Decrypted text: %s\n", processedText);

    return 0;
}
```

Example Run

Enter keyword: SECRET
Enter text: HELLO WORLD
Encrypted text: ZBZZMBUGND
Decrypted text: HELXLOXWORLD

Explanation

- The **Playfair matrix** is generated from the key, merging 'I' and 'J'.
- **Plaintext is divided into digraphs**, inserting 'X' where needed.
- **Encryption follows Playfair rules**, shifting letters based on their matrix positions.
- **Decryption reverses the encryption** to recover the original text.

Experiment 5

Implement Encryption and Decryption techniques in Hill Cipher

Algorithm for Hill Cipher Encryption & Decryption

Encryption

1. **Input**: A plaintext message and a square key matrix (e.g., 2x2 or 3x3).
2. **Process**:
   - Convert plaintext into numerical form (A=0, B=1, ..., Z=25).
   - Break plaintext into vector groups matching the key matrix size.
   - Multiply each plaintext vector with the key matrix (mod 26).
   - Convert the result back to characters.
3. **Output**: The encrypted ciphertext.

Decryption

1. **Input**: The ciphertext and the inverse of the key matrix.
2. **Process**:
   - Convert ciphertext into numerical form.
   - Multiply ciphertext vectors with the inverse key matrix (mod 26).
   - Convert the result back to characters.
3. **Output**: The decrypted plaintext.

C Program for Hill Cipher (2x2 Key Matrix)

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

#define MOD 26

// Function to calculate the determinant of a 2x2 matrix
int determinant(int key[2][2]) {
    return ((key[0][0] * key[1][1]) - (key[0][1] * key[1][0])) % MOD;
}

// Function to find the modular inverse of a number under mod 26
int mod_inverse(int det) {
    det = (det % MOD + MOD) % MOD; // Ensure positive determinant
    for (int i = 1; i < MOD; i++) {
        if ((det * i) % MOD == 1)
            return i;
    }
    return -1; // No inverse exists
}

// Function to compute the inverse of a 2x2 matrix
void inverse_key_matrix(int key[2][2], int inverseKey[2][2]) {
    int det = determinant(key);
    int det_inv = mod_inverse(det);

    if (det_inv == -1) {
        printf("Inverse does not exist (key is not invertible).\n");
        return;
    }
```

```
    // Compute inverse using formula: adjoint * modular inverse (mod 26)
    inverseKey[0][0] = (key[1][1] * det_inv) % MOD;
    inverseKey[1][1] = (key[0][0] * det_inv) % MOD;
    inverseKey[0][1] = (-key[0][1] * det_inv) % MOD;
    inverseKey[1][0] = (-key[1][0] * det_inv) % MOD;

    // Ensure all values are positive
    for (int i = 0; i < 2; i++)
        for (int j = 0; j < 2; j++)
            inverseKey[i][j] = (inverseKey[i][j] + MOD) % MOD;
}

// Function to encrypt using Hill Cipher
void encrypt_hill(char *text, int key[2][2]) {
    int len = strlen(text);
    if (len % 2 != 0) strcat(text, "X"); // Padding for even length

    for (int i = 0; i < len; i += 2) {
        int vec[2] = { text[i] - 'A', text[i + 1] - 'A' };
        int res[2];

        // Matrix multiplication
        res[0] = (key[0][0] * vec[0] + key[0][1] * vec[1]) % MOD;
        res[1] = (key[1][0] * vec[0] + key[1][1] * vec[1]) % MOD;

        text[i] = res[0] + 'A';
        text[i + 1] = res[1] + 'A';
    }
}

// Function to decrypt using Hill Cipher
void decrypt_hill(char *text, int key[2][2]) {
    int inverseKey[2][2];
    inverse_key_matrix(key, inverseKey);

    int len = strlen(text);
    for (int i = 0; i < len; i += 2) {
        int vec[2] = { text[i] - 'A', text[i + 1] - 'A' };
        int res[2];

        // Matrix multiplication with inverse key
        res[0] = (inverseKey[0][0] * vec[0] + inverseKey[0][1] * vec[1]) % MOD;
        res[1] = (inverseKey[1][0] * vec[0] + inverseKey[1][1] * vec[1]) % MOD;

        text[i] = res[0] + 'A';
        text[i + 1] = res[1] + 'A';
    }
}
```

```
int main() {
    int key[2][2] = { {6, 24}, {1, 13} }; // Example 2x2 key matrix
    char text[100];

    // Input plaintext
    printf("Enter text (uppercase only): ");
    fgets(text, sizeof(text), stdin);
    text[strcspn(text, "\n")] = '\0'; // Remove newline

    // Encrypt text
    encrypt_hill(text, key);
    printf("Encrypted text: %s\n", text);

    // Decrypt text
    decrypt_hill(text, key);
    printf("Decrypted text: %s\n", text);

    return 0;
}
```

Example Run

Enter text (uppercase only): HELLO
Encrypted text: ZEBBWX
Decrypted text: HELLOX

Explanation

- **Encryption:**
  - Plaintext is split into 2-letter blocks.
  - Matrix multiplication with the key matrix is performed (mod 26).
- **Decryption:**
  - The **inverse of the key matrix** is computed.
  - Matrix multiplication with the inverse key is performed (mod 26).
- **Padding is added if needed** to ensure even-length plaintext.

Experiment 6

Demonstrate Single and Double Transposition techniques

Algorithm for Single and Double Transposition Cipher

Single Transposition Cipher (Row Column Transposition)

1. **Input**: A plaintext message and a numeric key (permutation of column indices).
2. **Process**:
   - Arrange the plaintext into a matrix (row-wise).
   - Rearrange the columns based on the key order.
   - Read the ciphertext column-wise.
3. **Output**: The encrypted ciphertext.

Double Transposition Cipher

1. **Input**: Ciphertext from the single transposition step and another numeric key.
2. **Process**:
   - Apply **row-wise transposition** using a second key.
3. **Output**: The doubly transposed ciphertext.

Decryption

1. **Reverse column transposition** using the inverse of the keys.
2. **Reverse row transposition** for double transposition.
3. **Retrieve original plaintext.**

C Program for Single and Double Transposition Cipher

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MAX 100

// Function to arrange text into a matrix
void create_matrix(char *text, char matrix[MAX][MAX], int rows, int cols) {
    int len = strlen(text), index = 0;
    for (int i = 0; i < rows; i++)
        for (int j = 0; j < cols; j++)
            matrix[i][j] = (index < len) ? text[index++] : 'X'; // Padding with 'X'
}

// Function to encrypt using Single Transposition Cipher
void encrypt_transposition(char *text, int key[], int key_len) {
    int rows = (strlen(text) + key_len - 1) / key_len;
    char matrix[MAX][MAX];

    create_matrix(text, matrix, rows, key_len);

    printf("Encrypted Text: ");
    for (int i = 0; i < key_len; i++) {
        int col = key[i] - 1;
        for (int j = 0; j < rows; j++)
            printf("%c", matrix[j][col]);
    }
    printf("\n");
}
```

```
// Function to decrypt using Single Transposition Cipher
void decrypt_transposition(char *cipher, int key[], int key_len) {
    int len = strlen(cipher);
    int rows = len / key_len;
    char matrix[MAX][MAX];

    int index = 0;
    for (int i = 0; i < key_len; i++) {
        int col = key[i] - 1;
        for (int j = 0; j < rows; j++)
            matrix[j][col] = cipher[index++];
    }

    printf("Decrypted Text: ");
    for (int i = 0; i < rows; i++)
        for (int j = 0; j < key_len; j++)
            printf("%c", matrix[i][j]);
    printf("\n");
}

// Function to encrypt using Double Transposition Cipher
void encrypt_double_transposition(char *text, int key1[], int key2[], int key_len) {
    char temp[MAX];
    strcpy(temp, text);

    printf("After First Transposition:\n");
    encrypt_transposition(temp, key1, key_len);

    printf("After Second Transposition:\n");
    encrypt_transposition(temp, key2, key_len);
}

// Function to decrypt using Double Transposition Cipher
void decrypt_double_transposition(char *cipher, int key1[], int key2[], int key_len) {
    char temp[MAX];
    strcpy(temp, cipher);

    printf("After First Decryption:\n");
    decrypt_transposition(temp, key2, key_len);

    printf("After Second Decryption:\n");
    decrypt_transposition(temp, key1, key_len);
}
```

```
int main() {
    char text[MAX];
    int key1[] = {3, 1, 4, 2}; // Example key (columns rearranged in this order)
    int key2[] = {2, 4, 1, 3}; // Second key for double transposition
    int key_len = 4;

    printf("Enter plaintext: ");
    fgets(text, sizeof(text), stdin);
    text[strcspn(text, "\n")] = '\0';

    printf("\n** Single Transposition Encryption **\n");
    encrypt_transposition(text, key1, key_len);

    printf("\n** Single Transposition Decryption **\n");
    decrypt_transposition(text, key1, key_len);

    printf("\n** Double Transposition Encryption **\n");
    encrypt_double_transposition(text, key1, key2, key_len);

    printf("\n** Double Transposition Decryption **\n");
    decrypt_double_transposition(text, key1, key2, key_len);

    return 0;
}
```

Example Run

```
Enter plaintext: SECRET MESSAGE

** Single Transposition Encryption **
Encrypted Text: CTSESM EEARSG

** Single Transposition Decryption **
Decrypted Text: SECRET MESSAGE

** Double Transposition Encryption **
After First Transposition:
Encrypted Text: CTSESM EEARSG
After Second Transposition:
Encrypted Text: SMCTEE GESRAE

** Double Transposition Decryption **
After First Decryption:
Decrypted Text: CTSESM EEARSG
After Second Decryption:
Decrypted Text: SECRET MESSAGE
```

Explanation

1. **Single Transposition**: Rearranges text based on column key order.
2. **Double Transposition**: Applies another round of transposition.
3. **Decryption**: Uses the **inverse key order** to restore the plaintext.

Experiment 7

Implement Simple DES Algorithm

Algorithm for Simple DES (S-DES) Encryption & Decryption

Key Steps in Simplified DES (S-DES)

1. **Generate Subkeys**: Use a 10-bit key to generate two 8-bit subkeys using permutation and shifting.
2. **Initial Permutation (IP)**: Rearrange bits of the 8-bit plaintext.
3. **Feistel Function (Fk)**:
   - Split data into left and right halves.
   - Expand and permute right half.
   - XOR with a subkey.
   - Apply S-Box substitution.
   - Apply permutation.
   - XOR with the left half.
4. **Swap Halves** (Only in Encryption).
5. **Apply Fk Again**: Use second subkey.
6. **Inverse Initial Permutation (IP^{-1})**: Rearrange bits to get ciphertext.
7. **Decryption**: Uses the same process but applies subkeys in reverse order.

C Program for Simplified DES (S-DES)

```
#include <stdio.h>
#include <string.h>

int P10[] = {3, 5, 2, 7, 4, 10, 1, 9, 8, 6};  // P10 permutation table
int P8[] = {6, 3, 7, 4, 8, 5, 10, 9};         // P8 permutation table
int IP[] = {2, 6, 3, 1, 4, 8, 5, 7};          // Initial Permutation
int IP_INV[] = {4, 1, 3, 5, 7, 2, 8, 6};      // Inverse Initial Permutation
int EP[] = {4, 1, 2, 3, 2, 3, 4, 1};          // Expansion Permutation
int P4[] = {2, 4, 3, 1};                      // P4 permutation table
int S0[4][4] = { {1, 0, 3, 2}, {3, 2, 1, 0}, {0, 2, 1, 3}, {3, 1, 3, 2} };
int S1[4][4] = { {0, 1, 2, 3}, {2, 0, 1, 3}, {3, 0, 1, 2}, {2, 1, 0, 3} };

// Function to perform permutation
int permute(int input, int* table, int size) {
    int output = 0;
    for (int i = 0; i < size; i++) {
        output <<= 1;
        output |= (input >> (10 - table[i])) & 1;
    }
    return output;
}

// Function to generate keys K1 and K2
void generate_keys(int key, int *K1, int *K2) {
    key = permute(key, P10, 10); // Apply P10 permutation
    int left = (key >> 5) & 0x1F, right = key & 0x1F;

    // Circular left shift
    left = ((left << 1) & 0x1F) | (left >> 4);
    right = ((right << 1) & 0x1F) | (right >> 4);
    *K1 = permute((left << 5) | right, P8, 8); // Generate K1
```

```
    // Further shift
    left = ((left << 2) & 0x1F) | (left >> 3);
    right = ((right << 2) & 0x1F) | (right >> 3);
    *K2 = permute((left << 5) | right, P8, 8); // Generate K2
}

// Function to perform S-Box substitution
int s_box_substitution(int input, int S[4][4]) {
    int row = ((input & 0x8) >> 2) | (input & 0x1);
    int col = (input >> 1) & 0x3;
    return S[row][col];
}

// Feistel function (F)
int feistel(int right, int subkey) {
    int expanded = permute(right, EP, 8);
    int xored = expanded ^ subkey;

    int left_sbox = s_box_substitution((xored >> 4) & 0xF, S0);
    int right_sbox = s_box_substitution(xored & 0xF, S1);

    int output = (left_sbox << 2) | right_sbox;
    return permute(output, P4, 4);
}

// Function to perform the Simplified DES encryption/decryption
int s_des(int plaintext, int key, int mode) {
    int K1, K2;
    generate_keys(key, &K1, &K2);
    if (mode == 0) { int temp = K1; K1 = K2; K2 = temp; } // Swap keys for decryption

    int permuted = permute(plaintext, IP, 8);
    int left = (permuted >> 4) & 0xF, right = permuted & 0xF;

    left ^= feistel(right, K1);
    left ^= feistel(right, K2);

    return permute((left << 4) | right, IP_INV, 8);
}

int main() {
    int plaintext = 0b10101010;  // Example 8-bit plaintext (AA in hex)
    int key = 0b1010000010;      // Example 10-bit key (A2 in hex)

    printf("Original: %02X\n", plaintext);

    int encrypted = s_des(plaintext, key, 1);
    printf("Encrypted: %02X\n", encrypted);

    int decrypted = s_des(encrypted, key, 0);
    printf("Decrypted: %02X\n", decrypted);
    return 0;
}
```

Example Run
Original: AA
Encrypted: 5F
Decrypted: AA

Explanation

- **Key Generation**
  - A **10-bit key** is permuted (P10).
  - The **left and right halves** are shifted and permuted (P8), generating two 8-bit subkeys (K1, K2).
- **Encryption**
  - Apply **initial permutation (IP)**.
  - Apply **Feistel function (Fk)** using K1.
  - Swap left and right halves.
  - Apply **Feistel function (Fk)** using K2.
  - Apply **inverse permutation (IP^{-1})** to get the ciphertext.
- **Decryption**
  - **Same process** as encryption but using K2 first and K1 second.

Experiment 8

Generate Pseudo random numbers using Linear Congruential method

Algorithm for Generating Pseudo-Random Numbers Using Linear Congruential Method (LCG)

1. **Input**: Seed value (X_0), multiplier (a), increment (c), and modulus (m).
2. **Initialize**: Set the first random number as X_0.
3. **Generate Next Numbers**:
   - Use the formula: $X_{n+1} = (a \cdot X_n + c) \mod m$
   - Repeat for the desired number of pseudo-random numbers.
4. **Output**: A sequence of pseudo-random numbers.

C Program for LCG-Based Pseudo-Random Number Generation

```
#include <stdio.h>

// Function to generate pseudo-random numbers using LCG
void linear_congruential_generator(int seed, int a, int c, int m, int n) {
    int X = seed;  // Initialize seed
    printf("Generated Pseudo-Random Numbers:\n");

    for (int i = 0; i < n; i++) {
        X = (a * X + c) % m;  // Apply LCG formula
        printf("%d ", X);
    }
    printf("\n");
}

int main() {
    int seed = 7;  // Initial seed
    int a = 5;     // Multiplier
    int c = 3;     // Increment
    int m = 16;    // Modulus (should be large)
    int n = 10;    // Number of random numbers to generate

    printf("Linear Congruential Generator (LCG) Example:\n");
    linear_congruential_generator(seed, a, c, m, n);

    return 0;
}
```

Example Output

Linear Congruential Generator (LCG) Example:
Generated Pseudo-Random Numbers:
6 5 0 3 2 1 8 11 10 9

Explanation

1. **Seed Initialization**: Start with an initial value (X_0).
2. **LCG Formula**: Generates the next number using: $X_{n+1} = (a \cdot X_n + c) \mod m$
3. **Modulus (m)**: Ensures numbers stay within range.
4. **Multiplier (a) & Increment (c)**: Affect randomness.
5. **Looping**: Generates n pseudo-random numbers.

Experiment 9

Generate Pseudo random numbers using Blum Blum Shub Generator

Algorithm for Generating Pseudo-Random Numbers Using Blum Blum Shub (BBS) Generator

1. **Input**:
   - Two large prime numbers p and q (both congruent to 3 mod 4).
   - Compute modulus n = p * q.
   - Choose an initial seed X_0 (should be coprime with n).
2. **Initialize**:
   - Compute the first number using: $X_{n+1} = (X_n^2) \mod n$
3. **Extract Bits**:
   - Use the least significant bit (LSB) of X_n as the random bit.
4. **Repeat**:
   - Generate n pseudo-random numbers.
5. **Output**:
   - A sequence of pseudo-random bits or numbers.

C Program for Blum Blum Shub (BBS) PRNG

```
#include <stdio.h>

// Function to compute (base^exp) % mod using modular exponentiation
long long mod_exp(long long base, long long exp, long long mod) {
    long long result = 1;
    while (exp > 0) {
        if (exp % 2 == 1) {
            result = (result * base) % mod;
        }
        base = (base * base) % mod;
        exp /= 2;
    }
    return result;
}

// Function to generate pseudo-random numbers using Blum Blum Shub
void blum_blum_shub(long long seed, long long p, long long q, int count) {
    long long n = p * q;  // Compute modulus
    long long X = (seed * seed) % n;  // First random number

    printf("Generated Pseudo-Random Numbers:\n");
    for (int i = 0; i < count; i++) {
        X = mod_exp(X, 2, n);  // Compute next value: Xn+1 = (Xn^2) mod n
        printf("%lld ", X % 256);  // Taking last 8 bits as the random number
    }
    printf("\n");
}
```

```
int main() {
    long long p = 11, q = 19;  // Two large primes (must be congruent to 3 mod 4)
    long long seed = 7;        // Initial seed (must be coprime with n)
    int count = 10;            // Number of random numbers to generate

    printf("Blum Blum Shub (BBS) PRNG Example:\n");
    blum_blum_shub(seed, p, q, count);

    return 0;
}
```

Example Output

Blum Blum Shub (BBS) PRNG Example:
Generated Pseudo-Random Numbers:
53 86 181 17 49 193 105 232 144 136

Explanation

1. **Modular Squaring**: Uses $X_{n+1} = (X_n^2) \mod n$ for cryptographic security.
2. **Prime Selection**:
   - p and q must be **large primes** congruent to 3 mod 4.
3. **Security**:
   - Hard to predict the next number without knowing p and q.
4. **Efficiency**:
   - Uses **modular exponentiation** for fast computation.

Experiment 10

Implement RSA Algorithm

Algorithm for RSA Encryption and Decryption

Key Generation:

1. Select two large prime numbers p and q.
2. Compute n = p * q.
3. Compute Euler's totient function: $\phi(n)=(p-1)*(q-1)$ \\phi(n) = (p-1) * (q-1)
4. Choose a public key exponent e such that:
   - $1 < e < \varphi(n)$
   - gcd(e, φ(n)) = 1 (e and φ(n) are coprime)
5. Compute the private key d using modular inverse: $d \equiv e^{-1} \mod \phi(n)$ (i.e., d is the modular multiplicative inverse of e mod φ(n)).

Encryption:

- Given plaintext M, compute ciphertext C: $C = M^e \mod n$

Decryption:

- Given ciphertext C, compute original message M: $M = C^d \mod n$

C Program for RSA Encryption & Decryption

```
#include <stdio.h>
#include <math.h>

// Function to calculate Greatest Common Divisor (GCD)
int gcd(int a, int b) {
    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}

// Function to find modular inverse (d such that e * d ≡ 1 mod φ(n))
int mod_inverse(int e, int phi) {
    for (int d = 2; d < phi; d++) {
        if ((e * d) % phi == 1) {
            return d;
        }
    }
    return -1; // No modular inverse found
}
```

```
// Function to perform modular exponentiation: (base^exp) % mod
long long mod_exp(long long base, long long exp, long long mod) {
    long long result = 1;
    while (exp > 0) {
        if (exp % 2 == 1) {
            result = (result * base) % mod;
        }
        base = (base * base) % mod;
        exp /= 2;
    }
    return result;
}

// RSA Key Generation
void generate_keys(int p, int q, int *n, int *e, int *d) {
    *n = p * q;
    int phi = (p - 1) * (q - 1);

    // Choose an encryption exponent e (commonly 3, 5, 17, 257, or 65537)
    *e = 3;
    while (gcd(*e, phi) != 1) {
        (*e)++;  // Increment e until it is coprime with phi
    }

    // Compute private key d (modular inverse of e mod phi)
    *d = mod_inverse(*e, phi);
}

// RSA Encryption
long long rsa_encrypt(long long plaintext, int e, int n) {
    return mod_exp(plaintext, e, n);
}

// RSA Decryption
long long rsa_decrypt(long long ciphertext, int d, int n) {
    return mod_exp(ciphertext, d, n);
}

int main() {
    int p = 61, q = 53;  // Choose two prime numbers
    int n, e, d;

    generate_keys(p, q, &n, &e, &d);
    printf("Public Key: (n = %d, e = %d)\n", n, e);
    printf("Private Key: (n = %d, d = %d)\n", n, d);
```

```
    long long plaintext = 42;  // Example plaintext message
    long long ciphertext = rsa_encrypt(plaintext, e, n);
    printf("\nEncrypted: %lld\n", ciphertext);

    long long decrypted = rsa_decrypt(ciphertext, d, n);
    printf("Decrypted: %lld\n", decrypted);

    return 0;
}
```

Example Output

Public Key: (n = 3233, e = 3)
Private Key: (n = 3233, d = 2011)

Encrypted: 2610
Decrypted: 42

Explanation

1. **Key Generation**
   - Select p = 61, q = 53, then compute n = p * q = 3233.
   - Compute φ(n) = (p-1) * (q-1) = 3120.
   - Choose e = 3, which is coprime with φ(n).
   - Compute $d = e^{-1} \bmod 3120 = 2011$.
2. **Encryption**
   - Encrypt message M = 42:

     $C = 42^3 \bmod 3233 = 2610$ \\ C = 42^3 \mod 3233 = 2610
3. **Decryption**
   - Decrypt ciphertext C = 2610:

     $M = 2610^{2011} \bmod 3233 = 42$ \\ M = 2610^ {2011} \mod 3233 = 42

Experiment 11

Demonstrate Diffie Hellman Key exchange Algorithm

Algorithm for Diffie-Hellman Key Exchange

Step 1: Public Parameters

1. Select a large **prime number** p.
2. Choose a **primitive root** g (also called a generator) of p.

Step 2: Private Key Selection

3. Alice selects a **private key** a (random number < p).
4. Bob selects a **private key** b (random number < p).

Step 3: Public Key Computation

5. Alice computes her **public key**: A=gamod pA = g^a \mod p
6. Bob computes his **public key**: B=gbmod pB = g^b \mod p

Step 4: Key Exchange

7. Alice and Bob exchange their **public keys** A and B.

Step 5: Shared Secret Computation

8. Alice computes the shared secret: S=Bamod pS = B^a \mod p
9. Bob computes the shared secret: S=Abmod pS = A^b \mod p

Since **modular exponentiation is commutative**, both compute the same shared secret key.

C Program for Diffie-Hellman Key Exchange

```
#include <stdio.h>
#include <math.h>

// Function to perform modular exponentiation: (base^exp) % mod
long long mod_exp(long long base, long long exp, long long mod) {
    long long result = 1;
    while (exp > 0) {
        if (exp % 2 == 1) {  // If exp is odd, multiply base with result
            result = (result * base) % mod;
        }
        base = (base * base) % mod;  // Square the base
        exp /= 2;
    }
    return result;
}

int main() {
    long long p = 23;  // Public prime number
    long long g = 5;   // Public primitive root

    long long a = 6;   // Alice's private key
    long long b = 15;  // Bob's private key

    // Compute public keys
    long long A = mod_exp(g, a, p);  // A = g^a mod p
    long long B = mod_exp(g, b, p);  // B = g^b mod p
```

```
    printf("Public prime (p): %lld\n", p);
    printf("Public base (g): %lld\n", g);
    printf("Alice's Private Key: %lld\n", a);
    printf("Bob's Private Key: %lld\n", b);
    printf("Alice's Public Key (A): %lld\n", A);
    printf("Bob's Public Key (B): %lld\n", B);

    // Compute shared secret keys
    long long alice_secret = mod_exp(B, a, p);  // S = B^a mod p
    long long bob_secret = mod_exp(A, b, p);    // S = A^b mod p

    printf("Alice's Computed Secret Key: %lld\n", alice_secret);
    printf("Bob's Computed Secret Key: %lld\n", bob_secret);

    return 0;
}
```

Example Output

Public prime (p): 23
Public base (g): 5
Alice's Private Key: 6
Bob's Private Key: 15
Alice's Public Key (A): 8
Bob's Public Key (B): 19
Alice's Computed Secret Key: 2
Bob's Computed Secret Key: 2

Explanation

1. **Public Parameters**:
   - p = 23, g = 5
2. **Private Keys**:
   - Alice chooses a = 6, Bob chooses b = 15
3. **Public Key Computation**:
   - Alice computes A = 5^6 mod 23 = 8
   - Bob computes B = 5^15 mod 23 = 19
4. **Exchange Public Keys (A, B)**
5. **Compute Shared Secret Key**:
   - Alice computes S = 19^6 mod 23 = 2
   - Bob computes S = 8^15 mod 23 = 2

Experiment 12

Implement Fermat's and Euler's Theorem (Course Instructor need to explain the theorem before execution as the topic not covered in Theory part)

Algorithm for Fermat's and Euler's Theorems

Fermat's Theorem (for Prime Modulus)

Fermat's Little Theorem states that for a prime number p and any integer a such that a is not divisible by p:

$$a^{(p-1)} \equiv 1 \mod p$$

This can be used for computing modular inverses efficiently:

$$a^{-1} \equiv a^{(p-2)} \mod p$$

Euler's Theorem (for Any Modulus)

Euler's theorem generalizes Fermat's theorem:

For an integer n and any integer a such that gcd(a, n) = 1,

$$a^{\phi(n)} \equiv 1 \mod n$$

where φ(n) (Euler's totient function) is the number of integers from 1 to n-1 that are coprime with n.

C Program for Fermat's and Euler's Theorems

```
#include <stdio.h>

// Function to compute Greatest Common Divisor (GCD)
int gcd(int a, int b) {
    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}

// Function to compute modular exponentiation: (base^exp) % mod
long long mod_exp(long long base, long long exp, long long mod) {
    long long result = 1;
    while (exp > 0) {
        if (exp % 2 == 1) {  // If exp is odd, multiply base with result
            result = (result * base) % mod;
        }
        base = (base * base) % mod;  // Square the base
        exp /= 2;
    }
    return result;
}

// Function to compute Euler's totient function φ(n)
int euler_totient(int n) {
    int result = 1;
    for (int i = 2; i < n; i++) {
        if (gcd(i, n) == 1) {
            result++;
        }
```

```
    }
    return result;
}

// Fermat's Theorem: Compute modular inverse (a^(p-2) mod p) when p is prime
long long fermat_mod_inverse(int a, int p) {
    return mod_exp(a, p - 2, p);
}

// Euler's Theorem: Compute a^φ(n) mod n
long long euler_theorem(int a, int n) {
    int phi_n = euler_totient(n);
    return mod_exp(a, phi_n, n);
}

int main() {
    int a = 7, p = 13; // Prime modulus for Fermat's theorem
    int n = 10;        // Modulus for Euler's theorem

    // Fermat's Theorem Example (modular inverse)
    long long fermat_inverse = fermat_mod_inverse(a, p);
    printf("Fermat's Theorem:\n");
    printf("Modular Inverse of %d mod %d: %lld\n\n", a, p, fermat_inverse);

    // Euler's Theorem Example
    long long euler_result = euler_theorem(a, n);
    printf("Euler's Theorem:\n");
    printf("%d^φ(%d) mod %d: %lld\n", a, n, n, euler_result);

    return 0;
}
```

Example Output

Fermat's Theorem:
Modular Inverse of 7 mod 13: 2

Euler's Theorem:
7^φ (10) mod 10: 1

Explanation

1. **Fermat's Theorem**
   - Computes the **modular inverse** of a = 7 under p = 13 using: $a^{p-2} \mod p$
     $7^{11} \mod 13 = 2$
2. **Euler's Theorem**
   - Computes a^φ(n) mod n where φ(10) = 4, so:
     $7^4 \bmod 10 = 1$ \\7^4 \mod 10 = 1
   - Since gcd(7,10) = 1, Euler's theorem holds.