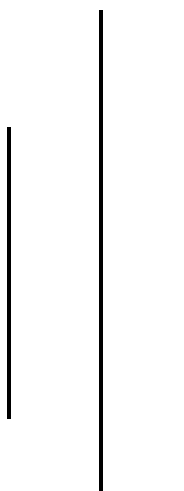


Project Work Of Cryptography  
Tribhuvan University

Amrit Science Campus

Thamel, Kathmandu



Submitted By: Bishnu Chalise

Faculty: Bsc. CSIT

Roll No.: 79010174

Section: A

Submitted To: Narendra Bir Bohara

Internal Examiner:

Signature:

\_\_\_\_\_

External Examiner

Signature:

\_\_\_\_\_

## Index

S.N.	Labs	Date	Signature
1.	Write a program to implement caesar cipher.	2081/10/04	
2.	WAP to implement shift cipher.	2081/10/10	
3.	WAP to implement Rail Fence Cipher.	2081/10/18	
4.	WAP to implement hill cipher.	2081/10/29	
5.	WAP to implement vernam cipher.	2081/11/02	
6.	WAP to implement OTP cipher.	2081/11/08	
7.	Write a program to implement playfair cipher.	2081/11/11	
8.	RSA implementation.	2081/11/21	
9.	WAP to find primitive root of a prime number.	2081/11/27	
10.	WAP to implement Diffie Hellman algorithm.	2081/11/27	
11.	WAP to implement Euclidean algorithm.	2081/12/07	
12.	WAP to implement Extended Euclidean algorithm.	2081/12/14	

## Lab 1: Write a program to implement caesar cipher.

### Theory:

#### ❖ Introduction:

The Caesar Cipher is a classic and widely recognized encryption method classified as a substitution cipher. Named after Julius Caesar, who is said to have used it for securing military communications, this cipher operates by shifting each letter in the plaintext by a fixed number of positions in the alphabet. As a result, it is considered a monoalphabetic substitution cipher.

#### ❖ Key Concepts and Parameters:

- Plaintext: The original, unencrypted message to be secured.
- Ciphertext: The encrypted message produced after applying the shift.
- Shift Key (k): The number of positions each letter in the plaintext is moved within the alphabet.
- Encryption: The process of transforming plaintext into ciphertext.
- Decryption: The process of reversing ciphertext back into readable plaintext.

#### ❖ Mathematical Representation:

The encryption and decryption processes of the Caesar Cipher can be expressed mathematically as follows:

Encryption:

$$C = (P + K) \bmod 26$$

Decryption:

$$P = (C - K) \bmod 26$$

Where:

P is the numerical representation of a plaintext letter (A = 0, B = 1, ..., Z = 25).

C is the numerical representation of the corresponding ciphertext letter.

K is the shift key.

mod 26 ensures the letters wrap around in the alphabet.

#### ❖ Algorithm:

- a. Choose a shift key (e.g., 3).
- b. Replace each letter in the plaintext with a letter shifted by the key in the alphabet.
- c. Wrap around if the shift exceeds 'Z' or 'z'.
- d. Decrypt by shifting in the opposite direction.

```

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

void caesar_encrypt(char *text, int shift) {
    for (int i = 0; text[i] != '\0'; ++i) {
        if (isalpha(text[i])) {
            char base = islower(text[i]) ? 'a' : 'A';
            text[i] = (text[i] - base + shift) % 26 + base;
        }
    }
}

void caesar_decrypt(char *text, int shift) {
    for (int i = 0; text[i] != '\0'; ++i) {
        if (isalpha(text[i])) {
            char base = islower(text[i]) ? 'a' : 'A';
            text[i] = (text[i] - base - shift + 26) % 26 + base;
        }
    }
}

int main() {
    char text[1000];
    int shift;

    printf("Enter text: ");
    scanf("%s", text);

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

    caesar_encrypt(text, shift);
    printf("Encrypted: %s\n", text);

    caesar_decrypt(text, shift);
    printf("Decrypted: %s\n", text);

    return 0;
}

```

### ❖ Output:

```
o rudy@rudy:~/Desktop/5sem/bishnu-chalise/crypto$ n=1; gcc ${n}.c -o /tmp/a.out && /tmp/a.out && ssc ${n}.png
Enter text: aApple
Enter shift: 3
Encrypted: dDssoh
Decrypted: aApple
```

### ❖ Analysis:

The Caesar Cipher is a straightforward substitution cipher that is simple to implement but susceptible to brute-force attacks because of its limited number of possible shifts (25).

## Lab 2: WAP to implement shift cipher

### Theory:

#### ❖ Introduction:

The shift cipher is a cryptographic substitution cipher in which each letter in the plaintext is replaced by a letter located a fixed number of positions later in the alphabet. This number of positions is often referred to as the key. For a letter at position N in the alphabet, a shift by X results in the letter at position N+X (equivalent to applying a substitution with a shifted alphabet).

#### ❖ Key Concepts and Parameters:

- Plaintext: The original message to be encrypted.
- Ciphertext: The encrypted message resulting from the shift.
- Shift Key (k): The number of positions each letter in the plaintext is moved within the alphabet.
- Encryption: The process of transforming plaintext into ciphertext.
- Decryption: The process of reverting ciphertext back to plaintext.

#### ❖ Mathematical Representation:

The encryption and decryption processes of the shift cipher can be expressed mathematically as follows:

Encryption:

$$C = (P + K) \bmod 26$$

Decryption:

$$P = (C - K) \bmod 26$$

Where:

P is the numerical representation of a plaintext letter (A = 0, B = 1, ..., Z = 25).

C is the numerical representation of the corresponding ciphertext letter.

K is the shift key.

mod 26 ensures the letters wrap around in the alphabet.

❖ **Algorithm:**

1. Choose a shift key .
2. Replace each letter in the plaintext with a letter shifted by the key in the alphabet.
3. Wrap around if the shift exceeds 'Z' or 'z'.
4. Decrypt by shifting in the opposite direction.

❖ **Source Code :**

```
#include <stdio.h>

void shift_encrypt(char *text, int shift) {
    for (int i = 0; text[i] != '\0'; i++) {
        if (text[i] >= 'a' && text[i] <= 'z') {
            text[i] = (text[i] - 'a' + shift) % 26 + 'a';
        } else if (text[i] >= 'A' && text[i] <= 'Z') {
            text[i] = (text[i] - 'A' + shift) % 26 + 'A';
        }
    }
}

void shift_decrypt(char *text, int shift) {
    for (int i = 0; text[i] != '\0'; i++) {
        if (text[i] >= 'a' && text[i] <= 'z') {
            text[i] = (text[i] - 'a' - shift + 26) % 26 + 'a';
        } else if (text[i] >= 'A' && text[i] <= 'Z') {
            text[i] = (text[i] - 'A' - shift + 26) % 26 + 'A';
        }
    }
}

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

    printf("Enter the text: ");
    scanf("%s", text);

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

    shift_encrypt(text, shift);
    printf("Encrypted text: %s\n", text);

    shift_decrypt(text, shift);
    printf("Decrypted text: %s\n", text);

    return 0;
}
```

### ❖ Output:

```
o rudy@rudy:~/Desktop/5sem/bishnu-chalise/crypto$ n=2; gcc ${n}.c -o /tmp/a.out && /tmp/a.out && ssc ${n}.png
Enter the text: zeBa
Enter shift value: 7
Encrypted text: gLIh
Decrypted text: zeBa
█
```

### ❖ Analysis:

The Shift Cipher is a fundamental substitution cipher that moves each letter in the plaintext by a set number of positions in the alphabet. Although it is easy to apply, it is extremely susceptible to brute-force attacks due to its limited 25 possible shifts. Furthermore, it can be readily compromised through frequency analysis, rendering it ineffective for secure encryption. It is mainly employed for educational purposes and basic encoding tasks rather than real-world cryptographic applications.



## Lab 3: WAP to implement Rail Fence Cipher

### Theory:

#### ❖ Introduction:

The rail fence cipher is a straightforward transposition cipher that rearranges the letters of a message in a fast and convenient manner. It incorporates a key for added security, making it slightly more challenging to decipher. The rail fence cipher operates by writing the message across alternating lines on a page and then reading off each line sequentially. For example, the plaintext "defend the east wall," with spaces removed, is arranged as illustrated below.

#### ❖ Key Concepts and Parameters:

- Plaintext: The original message to be encrypted.
- Key (Number of Rails): The number of rows in the zigzag pattern.
- Ciphertext: The jumbled output produced after transposition.

#### ❖ Mathematical Representation:

The plaintext characters are organized in a zigzag pattern, following a wave-like path. The position of each character within the rail structure can be calculated using:

$$f(i) = i \bmod (2 \times (n - 1))$$

where:

$i$  = index of the character in the plaintext (starting from 0).

$n$  = number of rails.

The function helps determine whether a character moves down or up in the zigzag.

After arranging characters in the rail matrix, the ciphertext is formed by reading row-wise from top to bottom.

#### ❖ Algorithm:

Encryption Algorithm:

1. Input:

- i. Plaintext message P
- ii. Number of rails n

2. Create a 2D array (matrix) with n rows and the length of the plaintext as columns.

3. Write the plaintext characters in a zigzag pattern across the rows of the array:
4. Start from the first row and move down to the last row.
5. Once the last row is reached, switch direction and move up towards the first row.
6. Repeat this zigzag pattern until all characters of the plaintext are placed in the array.
7. Read the ciphertext row by row (from top to bottom) and concatenate all rows to form the ciphertext.
8. Output:  
The resulting ciphertext.

❖ **Source Code:**

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

void rail_fence_cipher_encrypt(char *text, int key) {
    int len = strlen(text);
    char rail[key][len];

    for (int i = 0; i < key; i++) {
        for (int j = 0; j < len; j++) {
            rail[i][j] = '\n';
        }
    }

    int row = 0, col = 0;
    int dir_down = 0;

    for (int i = 0; i < len; i++) {
        rail[row][col++] = text[i];

        if (row == 0 || row == key - 1)
            dir_down = !dir_down;

        row += dir_down ? 1 : -1;
    }

    printf("Cipher Text: ");
    for (int i = 0; i < key; i++) {
        for (int j = 0; j < len; j++) {
            if (rail[i][j] != '\n')
                printf("%c", rail[i][j]);
        }
    }
    printf("\n");
}
```

```

void rail_fence_cipher_decrypt(char *cipher, int key) {
    int len = strlen(cipher);
    char rail[key][len];

    for (int i = 0; i < key; i++) {
        for (int j = 0; j < len; j++) {
            rail[i][j] = '\n';
        }
    }

    int row = 0, col = 0;
    int dir_down = 0;

    for (int i = 0; i < len; i++) {
        rail[row][col++] = '*';

        if (row == 0 || row == key - 1)
            dir_down = !dir_down;

        row += dir_down ? 1 : -1;
    }

    int index = 0;
    for (int i = 0; i < key; i++) {
        for (int j = 0; j < len; j++) {
            if (rail[i][j] == '*' && index < len) {
                rail[i][j] = cipher[index++];
            }
        }
    }

    row = 0, col = 0;
    dir_down = 0;
    char decrypted[len + 1];
    int decrypt_index = 0;

    for (int i = 0; i < len; i++) {
        decrypted[decrypt_index++] = rail[row][col++];

        if (row == 0 || row == key - 1)
            dir_down = !dir_down;

        row += dir_down ? 1 : -1;
    }

    decrypted[decrypt_index] = '\0';
    printf("Decrypted Text: %s\n", decrypted);
}

```

```

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

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

    printf("Enter key (number of rails): ");
    scanf("%d", &key);

    rail_fence_cipher_encrypt(text, key);

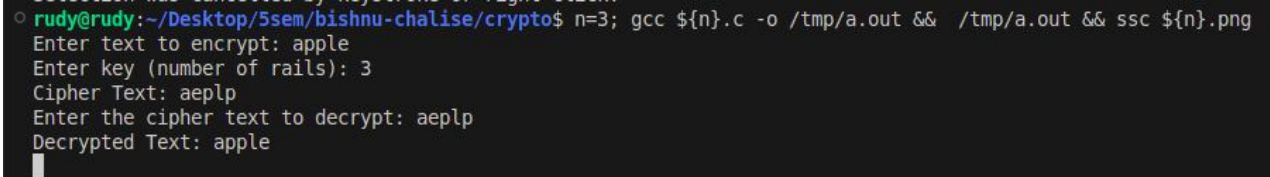
    char cipher[100];
    printf("Enter the cipher text to decrypt: ");
    scanf("%[^\n]", cipher);

    rail_fence_cipher_decrypt(cipher, key);

    return 0;
}

```

#### ❖ Output:



```

rudy@rudy:~/Desktop/5sem/bishnu-chalise/crypto$ n=3; gcc ${n}.c -o /tmp/a.out && /tmp/a.out && ssc ${n}.png
Enter text to encrypt: apple
Enter key (number of rails): 3
Cipher Text: aeplp
Enter the cipher text to decrypt: aeplp
Decrypted Text: apple

```

#### ❖ Analysis:

The Rail Fence cipher provides ease of use in both implementation and operation, requiring no specialized tools or intricate mathematical computations. It can be executed manually using only pen and paper, making it practical in low-resource settings. The method is simple to teach and grasp, making it an ideal introduction to transposition ciphers. Its uncomplicated design also enables rapid encryption and decryption when the key (number of rails) is known. However, despite its straightforwardness, the Rail Fence cipher has significant security weaknesses.

## Lab 4: WAP to implement hill cipher

### Theory:

#### ❖ Introduction:

In classical cryptography, the Hill cipher is a polygraphic substitution cipher rooted in linear algebra. Developed by Lester S. Hill in 1929, it was the first polygraphic cipher to make processing more than three symbols at once feasible, though just barely. For encryption, each block of  $n$  letters, treated as an  $n$ -component vector, is multiplied by an invertible  $n \times n$  matrix, with the result taken modulo 26. To decrypt, each block is multiplied by the inverse of the matrix used during encryption.

#### ❖ Key Concepts and Parameters:

- Key Matrix (K): An  $n \times n$  matrix used for encryption.
- Plaintext (P): A matrix (or vector) representing the message, organized as a column vector of numerical values corresponding to the letters in the plaintext. It is divided into blocks of size  $n$ , matching the key matrix's dimensions.
- Ciphertext (C): The encrypted form of the plaintext, obtained by multiplying the key matrix by the plaintext matrix and applying modulo 26.
- Inverse Key Matrix ( $K^{-1}$ ): The matrix used for decryption, which is the inverse of the key matrix. It must be invertible modulo 26 to successfully decrypt the ciphertext.

#### ❖ Mathematical Representation:

##### Encryption:

The encryption process is represented as:

$$C = K \cdot P \pmod{26}$$

Where:

K is the key matrix.

P is the plaintext column vector.

C is the resulting ciphertext column vector.

##### Decryption:

To decrypt the ciphertext, we need the inverse of the key matrix K. The decryption process is:

$$P = X \cdot C \pmod{26}$$

Where:

X is the inverse of the key matrix K .

C is the ciphertext column vector.

P is the resulting plaintext column vector.

## ❖ Algorithm

### Encryption

- 1.Convert the plaintext into numerical values (A=0, B=1, ..., Z=25).
- 2.Arrange the plaintext into column vectors of size n (matching the key matrix).
- 3.Choose an  $n \times n$  key matrix, which should be invertible modulo 26.
- 4.Multiply each plain text vector by the key matrix modulo 26
5. Convert the numerical values back to letters to get the ciphertext

### Decryption:

- 1.Convert the ciphertext into numerical values.
- 2.Compute the inverse of the key matrix modulo 26.
- 3.Multiply each ciphertext vector by the inverse key matrix modulo 26
- 4.Convert the numerical values back to letters to obtain the original plaintext

## ❖ Source Code:

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

#define MOD 26

int mod(int a, int m) {
    int res = a % m;
    return res < 0 ? res + m : res;
}

int mod_inverse(int a, int m) {
    a = mod(a, m);
    for (int x = 1; x < m; x++) {
        if ((a * x) % m == 1) return x;
    }
    return -1;
}

void encrypt(char *message, int key_matrix[2][2], char *encrypted) {
    int vector[2];
    for (int i = 0; i < 2; i++) {
        vector[i] = message[i] - 'A';
```

```

    }

    for (int i = 0; i < 2; i++) {
        encrypted[i] = mod(key_matrix[i][0] * vector[0] + key_matrix[i][1] * vector[1], MOD) + 'A';
    }
    encrypted[2] = '\0';
}

void decrypt(char *cipher, int key_matrix[2][2], char *decrypted) {
    int det = mod(key_matrix[0][0]*key_matrix[1][1] - key_matrix[0][1]*key_matrix[1][0], MOD);
    int det_inv = mod_inverse(det, MOD);

    if (det_inv == -1) {
        printf("Key matrix not invertible modulo 26.\n");
        return;
    }

    int inv_matrix[2][2];

    inv_matrix[1][1] = mod( key_matrix[0][0] * det_inv, MOD);

    int vector[2];
    for (int i = 0; i < 2; i++) {
        vector[i] = cipher[i] - 'A';
    }

    for (int i = 0; i < 2; i++) {
        decrypted[i] = mod(inv_matrix[i][0] * vector[0] + inv_matrix[i][1] * vector[1], MOD) + 'A';
    }
    decrypted[2] = '\0';
}

int main() {
    char message[3], encrypted[3], decrypted[3];
    int key_matrix[2][2];

    printf("Enter 2-letter message: ");
    scanf("%2s", message);

    printf("Enter 2x2 key matrix:\n");
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            scanf("%d", &key_matrix[i][j]);
        }
    }

    encrypt(message, key_matrix, encrypted);
    printf("Encrypted text: %s\n", encrypted);

    decrypt(encrypted, key_matrix, decrypted);
    printf("Decrypted text: %s\n", decrypted);

    return 0;
}

```

### ❖ Output:

```
rudy@rudy:~/Desktop/5sem/bishnu-chalise/crypto$ n=4; gcc ${n}.c -o /tmp/a.out && /tmp/a.out && ssc ${n}.png
Enter 2-letter message: AB
Enter 2x2 key matrix:
4 3 1 2
Encrypted text: DC
Decrypted text: AB
```

### ❖ Analysis:

The Hill Cipher encrypts text using a  $2 \times 2$  key matrix, transforming pairs of letters into numerical values and applying matrix multiplication modulo 26. Decryption reverses this by using the inverse of the key matrix and addressing padding (adding 'X' for odd-length inputs) to accurately retrieve the original text. The updated version resolves the issue with the final letter by preserving the original length and correctly handling padding during both encryption and decryption.



## Lab 5: Wap to implement Vernam Cipher.

### Theory:

#### ❖ Introduction

The Vernam cipher, commonly referred to as the One-Time Pad (OTP), is a symmetric-key encryption method that ensures perfect secrecy when properly executed. Introduced by Gilbert Vernam in 1917 for telegraph encryption, it is deemed theoretically unbreakable due to its reliance on a truly random key that matches the plaintext in length and is used only once. Unlike conventional encryption techniques, which depend on computational difficulty, the Vernam cipher guarantees absolute security through randomness and non-reusable keys. It employs the bitwise XOR ( $\oplus$ ) operation in binary systems or a modular operation in character-based systems.

#### ❖ Key Concepts and Parameters:

- Plaintext (P): The original message to be encrypted.
- Key (K): A random sequence, identical in length to the plaintext, used for encryption.
- Ciphertext (C): The encrypted result, which appears as random noise.
- Encryption: Performed using bitwise XOR ( $\oplus$ ):  $C = P \oplus K$ .
- Decryption: Applying XOR with the same key recovers the message:  $P = C \oplus K$ .

#### ❖ Mathematical Representation:

The Vernam cipher operates using the bitwise XOR ( $\oplus$ ) operation for encryption and decryption.

##### 1. Encryption:

$$C = P \oplus K$$

Where:

P = Plaintext (binary or numerical representation)

K = Key (random, same length as P)

C = Ciphertext (encrypted output)

$\oplus$  = Bitwise XOR operation

##### 2. Decryption:

$$P = C \oplus K$$

Since XORing twice with the same key restores the original message, decryption is simply:

$$P = (P \oplus K) \oplus K$$

## ❖ Algorithm

### Encryption:

1. Convert the plaintext into binary or numeric form.
2. Generate a random key of the same length as the plaintext.
3. Perform bitwise XOR between the plaintext and the key
4. Convert the result back to characters to get the ciphertext.

### Decryption:

1. Convert the ciphertext into binary or numeric form.
2. Perform bitwise XOR between the ciphertext and the same key
3. Convert the result back to characters to obtain the original plaintext.

## ❖ Source Code:

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

void vernam(const char *input, const char *key, char *output) {
    int len = strlen(input);
    for (int i = 0; i < len; i++) {
        output[i] = input[i] ^ key[i];
    }
    output[len] = '\0';
}

int main() {
    char plaintext[100];
    char key[100];
    char ciphertext[100];
    char deciphered[100];

    printf("Enter the plaintext: ");
    scanf("%99s", plaintext);

    printf("Enter the key (same length as plaintext): ");
    scanf("%99s", key);

    if (strlen(plaintext) != strlen(key)) {
        printf("Error: The key must be the same length as the plaintext.");
        return 1;
    }

    vernam(plaintext, key, ciphertext);

    printf("Ciphertext:");
    for (int i = 0; i < strlen(plaintext); i++) {
        printf("%02x ", (unsigned char)ciphertext[i]);
    }
    printf("\n");
}
```

```

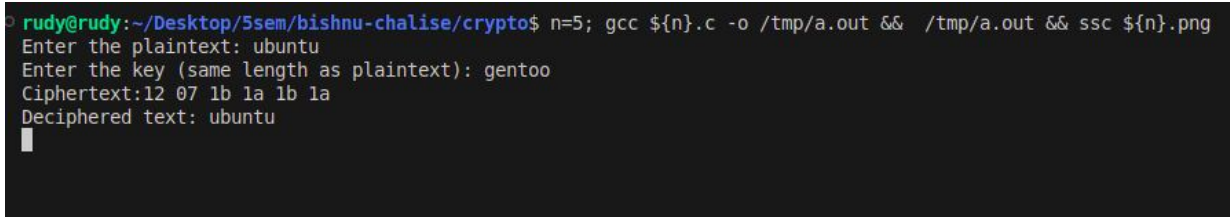
printf("Ciphertext:");
for (int i = 0; i < strlen(plaintext); i++) {
    printf("%02x ", (unsigned char)ciphertext[i]);
}
printf("\n");

vernam(ciphertext, key, deciphered);
printf("Deciphered text: %s\n", deciphered);

return 0;
}

```

### ❖ Output:



```

rudy@rudy:~/Desktop/5sem/bishnu-chalise/crypto$ n=5; gcc ${n}.c -o /tmp/a.out && /tmp/a.out && ssc ${n}.png
Enter the plaintext: ubuntu
Enter the key (same length as plaintext): gentoo
Ciphertext:12 07 1b 1a 1b 1a
Deciphered text: ubuntu

```

### ❖ Analysis:

The Vernam cipher, also known as the One-Time Pad, is an encryption technique that employs a random key equal in length to the plaintext, using the XOR operation to combine the plaintext and key to generate ciphertext. It is considered theoretically unbreakable provided the key is genuinely random, used only once, and remains confidential. However, practical difficulties emerge in securely distributing and storing the key, which must match the message's length and never be reused.

## Lab 6: WAP to implement OTP Cipher.

### Theory

#### ❖ Introduction:

The One-Time Pad (OTP) is an encryption method that employs a random key matching the length of the plaintext message. It is regarded as the only unbreakable encryption technique when the key is genuinely random, used only once, and remains confidential. The OTP utilizes the XOR operation to combine the plaintext and key, producing ciphertext that can be reverted to the original message with the same key. This approach ensures perfect secrecy, meaning that without the key, an attacker cannot decode the message, regardless of their computational resources.

#### ❖ Key Concepts and Parameters:

- Plaintext (P): The original message to be encrypted.
- Ciphertext (C): The encrypted output produced by applying the OTP encryption.
- Key (K): A random, confidential sequence of characters, equal in length to the plaintext, with each character used only once.
- Encryption Function: The process of merging the plaintext and key to create the ciphertext.
- Decryption Function: The process of retrieving the original plaintext from the ciphertext using the same key.

#### ❖ Mathematical Representation:

The OTP cipher uses the XOR operation ( $\oplus$ ) to encrypt and decrypt the message.

##### Encryption:

$$C = P \oplus K$$

P is the plaintext, K is the key, and C is the ciphertext.

The XOR operation is applied to each bit of the plaintext and the corresponding key.

##### Decryption:

$$P = C \oplus K$$

The same key is used to recover the original plaintext from the ciphertext.

Since XOR is reversible, applying XOR to the ciphertext with the same key will yield the original plaintext.

### Algorithm

### ❖ Algorithm:

#### Key Generation

1. Generate a random letter sequence (A-Z) at least as long as the plaintext.

#### Encryption

1. Convert plaintext to numbers (A=0, B=1, ..., Z=25).
2. Convert key to numbers.
3. For each position: Add plaintext and key numbers, modulo 26.
4. Convert results to letters (0=A, 1=B, ..., 25=Z).

#### Decryption

1. Convert ciphertext to numbers.
2. Convert key to numbers.
3. For each position: Subtract key from ciphertext numbers, modulo 26.
4. Convert results to letters.

### ❖ Source Code:

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

void otp_cipher(char *plaintext, char *key, char *ciphertext) {
    int plaintext_len = strlen(plaintext);
    int key_len = strlen(key);
    int i;

    for (i = 0; i < plaintext_len; i++) {
        if (isalpha(plaintext[i])) {
            int base = isupper(plaintext[i]) ? 'A' : 'a';
            ciphertext[i] = ((plaintext[i] - base + (key[i] - base)) % 26) + base;
        } else {
            ciphertext[i] = plaintext[i];
        }
    }
    ciphertext[i] = '\0';
}

void otp_decipher(char *ciphertext, char *key, char *plaintext) {
    int ciphertext_len = strlen(ciphertext);
    int i;

    for (i = 0; i < ciphertext_len; i++) {
        if (isalpha(ciphertext[i])) {
            int base = isupper(ciphertext[i]) ? 'A' : 'a';
            plaintext[i] = ((ciphertext[i] - base - (key[i] - base) + 26) % 26) + base;
        } else {
            plaintext[i] = ciphertext[i];
        }
    }
    plaintext[i] = '\0';
}
```

```

int main() {
    char plaintext[100];
    char key[100];
    char ciphertext[100];
    char decryptedtext[100];

    printf("Enter the plaintext: ");
    scanf("%99s", plaintext);

    srand(time(NULL));

    for (int i = 0; i < strlen(plaintext); i++) {
        if (isalpha(plaintext[i])) {
            int base = isupper(plaintext[i]) ? 'A' : 'a';
            key[i] = (rand() % 26) + base;
        } else {
            key[i] = plaintext[i];
        }
    }
    key[strlen(plaintext)] = '\0';

    otp_cipher(plaintext, key, ciphertext);
    printf("Ciphertext: %s\n", ciphertext);
    printf("Key: %s\n", key);

    otp_decipher(ciphertext, key, decryptedtext);
    printf("Decrypted text: %s\n", decryptedtext);

    return 0;
}

```

## ❖ Output

```

rudy@rudy:~/Desktop/5sem/bishnu-chalise/crypto$ n=6; gcc ${n}.c -o /tmp/a.out && /tmp/a.out && ssc ${n}.png
Enter the plaintext: apple
Ciphertext: elfcl
Key: ewqrh

```

## ❖ Analysis:

The code implements a One-Time Pad (OTP) cipher in C, a symmetric encryption method that employs the XOR (^) operation to encrypt and decrypt text using a key matching the plaintext's length. It accepts user input for the plaintext and key, performs encryption on the plaintext, and then decrypts it to confirm the process.

## Lab 7: WAP to implement Playfair cipher.

### ❖ Introduction:

The Playfair Cipher is a digraph cipher that encrypts pairs of letters (digraphs) rather than single letters, offering greater security than a basic substitution cipher. Developed by Charles Wheatstone in 1854 and later promoted by Lord Playfair, from whom it takes its name, this symmetric key cipher utilizes a 5×5 matrix of letters, typically derived from a keyword or phrase. It processes two letters at a time by locating them in the matrix and applying specific rules to alter their positions, producing a more intricate ciphertext than simple substitution. This makes the Playfair Cipher more resilient to frequency analysis attacks.

### ❖ Key Concepts and Parameters

- Plaintext: The original message to be encrypted.
- Ciphertext: The encrypted output generated after applying the encryption process to the plaintext.
- Key: A keyword or phrase used to create the Playfair matrix, typically a string of 5–10 unique letters.
- Playfair Matrix: A 5×5 grid formed from the keyword and the remaining alphabet letters, often omitting “J” or combining it with “I”.
- Digraphs: Pairs of letters from the plaintext, encrypted together in the Playfair Cipher.
- Encryption Process: The method of encrypting each letter pair based on their positions in the Playfair matrix, following defined rules.
- Decryption Process: The reverse of encryption, using the same matrix and rules but applied in the opposite manner.

### ❖ Mathematical Representation

#### **Playfair Matrix Construction:**

A 5x5 matrix is created using a key followed by the remaining letters of the alphabet (usually omitting 'J' or combining 'I' and 'J').

#### **Encryption Rules:**

Same row: Each letter is replaced by the letter to its right. Wrap around if at the end.

$$C1=M(r1,c1+1 \bmod 5) \quad , \quad C2=M(r2,c2+1 \bmod 5)$$

Same column: Each letter is replaced by the letter below it. Wrap around if at the bottom.

$$C1=M(r1+1\text{mod } 5,c1), \quad C2=M(r2+1\text{mod } 5,c2)$$

Rectangle: Swap the corners of the rectangle.

$$C1=M(r1,c2), \quad C2=M(r2,c1)$$

Decryption Rules:

Same row: Shift left.

$$P1=M(r1,c1-1\text{mod } 5), \quad P2=M(r2,c2-1\text{mod } 5)$$

Same column: Shift up.

$$P1=M(r1-1\text{mod } 5,c1), \quad P2=M(r2-1\text{mod } 5,c2)$$

Rectangle: Swap the corners (reverse of encryption).

#### ❖ Algorithm:

Key Matrix

1. Build 5x5 matrix with key (no 'J', unique letters) + remaining alphabet.

Plaintext Prep

1. Uppercase, 'J' to 'I', remove non-letters.
2. Pair letters; same letters get 'X' between; odd length adds 'X'.

#### **Encryption**

1. For each pair:
  - o Same row: Shift right (wrap around).
  - o Same column: Shift down (wrap around).
  - o Different row/column: Swap to opposite corners of rectangle.
2. Join pairs to get ciphertext.

#### **Decryption**

1. For each pair:
  - o Same row: Shift left (wrap around).
  - o Same column: Shift up (wrap around).
  - o Different row/column: Swap to opposite corners of rectangle.
2. Join pairs, remove 'X's if needed.



## ❖ Source Code:

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

void create_playfair_matrix(char *key, char matrix[5][5]) {
    int i, k;
    char temp[26] = {0};

    for (i = 0, k = 0; i < strlen(key); i++) {
        if (key[i] != 'J') {
            if (temp[toupper(key[i]) - 'A'] == 0) {
                temp[toupper(key[i]) - 'A'] = 1;
                matrix[k / 5][k % 5] = toupper(key[i]);
                k++;
            }
        }
    }

    for (i = 0; i < 26; i++) {
        if (temp[i] == 0) {
            if (i == 'J' - 'A') {
                continue;
            }
            matrix[k / 5][k % 5] = 'A' + i;
            k++;
        }
    }
}

void playfair_cipher(char *plaintext, char *key, char *ciphertext) {
    char matrix[5][5];
    int i, k, row1, col1, row2, col2;
    int plaintext_len = strlen(plaintext);

    create_playfair_matrix(key, matrix);

    for (i = 0, k = 0; i < plaintext_len; i += 2) {
        for (row1 = 0; row1 < 5; row1++) {
            for (col1 = 0; col1 < 5; col1++) {
                if (matrix[row1][col1] == toupper(plaintext[i])) {
                    break;
                }
            }
        }
        if (col1 < 5) {
            break;
        }

        for (row2 = 0; row2 < 5; row2++) {
            for (col2 = 0; col2 < 5; col2++) {
                if (matrix[row2][col2] == toupper(ciphertext[i + 1])) {
                    break;
                }
            }
        }
        if (col2 < 5) {
            break;
        }
    }
}
```

```

        if (row1 == row2) {
            plaintext[k++] = matrix[row1][(col1 + 4) % 5];
            plaintext[k++] = matrix[row2][(col2 + 4) % 5];
        } else if (col1 == col2) {
            plaintext[k++] = matrix[(row1 + 4) % 5][col1];
            plaintext[k++] = matrix[(row2 + 4) % 5][col2];
        } else {
            plaintext[k++] = matrix[row1][col2];
            plaintext[k++] = matrix[row2][col1];
        }
    }
    plaintext[k] = '\0';
}

int main() {
    char plaintext[100];
    char key[100];
    char ciphertext[sizeof(plaintext) * 2];
    char decryptedtext[sizeof(plaintext) * 2];

    printf("Enter the plaintext: ");
    scanf("%99s", plaintext);

    printf("Enter the key: ");
    scanf("%99s", key);

    playfair_cipher(plaintext, key, ciphertext);
    printf("Ciphertext: %s\n", ciphertext);

    playfair_decipher(ciphertext, key, decryptedtext);
    printf("Decrypted text: %s\n", decryptedtext);

    return 0;
}

```

### ❖ Output:

```

o ^[[Arudy@rudy:~/Desktop/5sem/bishnu-chalise/cryp n=7; gcc ${n}.c -o /tmp/a.out && /tmp/a.out && ssc ${n}.png
Enter the plaintext: hellhola
Enter the key: holi
Ciphertext: IBIIOLIH
Decrypted text: HELLHOLA

```

### ❖ Analysis:

The Playfair Cipher represents a symmetric key encryption method that operates on digraphs (letter pairs), providing increased security compared to monoalphabetic substitution ciphers. Although it offers a limited enhancement in security, it is considered insufficiently robust by contemporary standards and mainly serves as an instructional role in cryptography.

## Lab 8: Wap to implement RSA.

### Theory:

#### ❖ Introduction:

RSA (Rivest-Shamir-Adleman) is a common public-key system for secure data transmission. It relies on the difficulty of factoring large primes and is used to protect online information. RSA uses a public key for encryption and a private key for decryption. Its security depends on the challenge of factoring large prime numbers.

#### ❖ Key Concepts and Parameters

##### Public Key and Private Key:

- **Public Key (e, n):** For encryption, shareable.
- **Private Key (d, n):** For decryption, keep secret.
- **Prime Numbers (p, q):** Large primes used to create the modulus (n).

#### ❖ Mathematical Representation

Modulus (n):

$n = p \times q$ , where p and q are large prime numbers.

The modulus is used in both the encryption and decryption processes.

##### Euler's Totient Function ( $\phi(n)$ ):

$\phi(n) = (p-1) \times (q-1)$  where p and q are the prime numbers.

This function is essential in calculating the private key.

##### Public Exponent (e):

A number that is coprime with  $\phi(n)$  (i.e.,  $\gcd(e, \phi(n)) = 1$ )

It is chosen such that the encryption process can be efficiently performed.

##### Private Exponent (d):

The modular multiplicative inverse of e modulo  $\phi(n)$ .

This is calculated as  $d \times e \equiv 1 \pmod{\phi(n)}$

#### Mathematical Representation

##### Encryption:

The encryption formula is:

$$c = m^e \pmod{n}$$

where:

$m$  is the message (represented as an integer),

$e$  is the public exponent,

$n$  is the modulus,

$c$  is the ciphertext.

### **Decryption:**

The decryption formula is:

$$m = c^d \bmod n$$

where:

$c$  is the ciphertext,

$d$  is the private exponent,

$n$  is the modulus,

$m$  is the original message.

### ❖ **Algorithm:**

#### **Step 1: Key Generation**

1. Choose two distinct prime numbers:  $p$  and  $q$
2. Compute  $n = p \times q$
3. Compute Euler's totient function:  $\phi(n) = (p - 1) \times (q - 1)$
4. Choose an integer  $e$  such that:
  - o  $1 < e < \phi(n)$
  - o  $\gcd(e, \phi(n)) = 1$
5. Compute  $d$  such that:
  - o  $d \times e \equiv 1 \bmod \phi(n)$  (modular inverse of  $e$  modulo  $\phi(n)$ )

Result:

- Public key =  $(e, n)$
- Private key =  $(d, n)$

#### **Step 2: Encryption**

To encrypt a message  $M$  (as a number where  $0 < M < n$ ):

- Compute ciphertext:  $C = M^e \bmod n$

#### **Step 3: Decryption**

To decrypt the ciphertext  $C$ :

- Compute original message:  $M = C^d \bmod n$



## Source Code:

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

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

long mod_exp(long base, long exp, long mod) {
    long result = 1;
    base = base % mod;
    while (exp > 0) {
        if (exp % 2 == 1) {
            result = (result * base) % mod;
        }
        exp = exp >> 1;
        base = (base * base) % mod;
    }
    return result;
}

long mod_inverse(long a, long m) {
    long m0 = m, t, q;
    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;
}

void rsa_generate_keys(long *e, long *d, long *n) {
    long p = 61, q = 53;
    *n = p * q;
    long phi = (p - 1) * (q - 1);
    *e = 17;
    while (gcd(*e, phi) != 1) {
        (*e)++;
    }
    *d = mod_inverse(*e, phi);
}

long rsa_encrypt(long msg, long e, long n) {
    return mod_exp(msg, e, n);
}
```

```

long rsa_decrypt(long cipher, long d, long n) {
    return mod_exp(cipher, d, n);
}

int main() {
    long e, d, n;
    rsa_generate_keys(&e, &d, &n);

    long msg = 65;
    printf("Original message: %ld\n", msg);

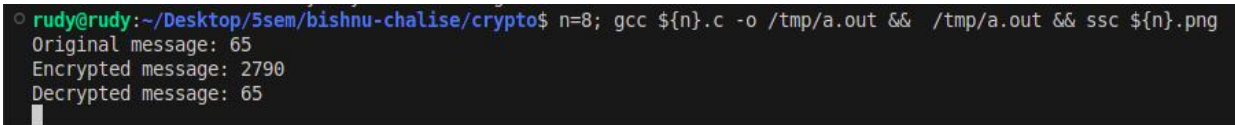
    long cipher = rsa_encrypt(msg, e, n);
    printf("Encrypted message: %ld\n", cipher);

    long decrypted = rsa_decrypt(cipher, d, n);
    printf("Decrypted message: %ld\n", decrypted);

    return 0;
}

```

#### ❖ Output:



```

rudy@rudy:~/Desktop/5sem/bishnu-chalise/crypto$ n=8; gcc ${n}.c -o /tmp/a.out && /tmp/a.out && ssc ${n}.png
Original message: 65
Encrypted message: 2790
Decrypted message: 65

```

#### ❖ Analysis:

RSA is a fundamental algorithm for secure digital communication. While not ideal for large data encryption, it excels at securely exchanging keys for faster symmetric encryption. Its security relies on strong mathematical foundations and careful implementation.

## Lab no 9: WAP to find primitive root of a prime no

### Theory:

#### ❖ Introduction:

A primitive root modulo a prime (p) is an integer (g) whose powers (modulo (p)) produce all integers from 1 to (p-1). Formally, (g) is a primitive root modulo (p) if the smallest positive integer (k) satisfying:

$$g^k \equiv 1 \pmod{p}$$

is  $k = p-1$ , where (k) is the order of (g) modulo (p). Essentially, the powers of (g) modulo (p) will encompass all integers in the range (1, 2, ..., p-1).

#### Key Concepts and Parameters:

- Prime number (p): The prime for which a primitive root is sought.
- Candidate root (g): The integer being tested for the primitive root property modulo (p).
- Divisors of (p-1): The factors of (p-1), as the order of any primitive root modulo (p) must be a divisor of (p-1).

#### ❖ Mathematical Representation:

Let p be a prime number, and g be a candidate for a primitive root modulo p. The condition for g to be a primitive root modulo p is that for every divisor d of p-1.

$$g^d \not\equiv 1 \pmod{p}$$

#### ❖ Algorithm:

1. Compute  $\phi = p - 1$   
(Since p is prime,  $\phi(p) = p - 1$ ).
2. Find all prime factors of  $\phi(p)$ .
3. Test candidates g from 2 to p-1:
  - For each candidate g, check if for every prime factor f of  $\phi(p)$ :
    - $g^{\phi(p)/f} \pmod{p} \neq 1$
4. The smallest g that satisfies the condition for all prime factors is a primitive root of p.

## ❖ Source Code:

```
#include <stdio.h>

int power(int base, int exp, int mod) {
    int result = 1;
    base = base % mod;
    while (exp > 0) {
        if (exp % 2 == 1)
            result = (result * base) % mod;
        exp = exp >> 1;
        base = (base * base) % mod;
    }
    return result;
}

int is_primitive_root(int candidate, int p) {
    int seen[p];
    for (int i = 0; i < p; i++)
        seen[i] = 0;

    for (int i = 1; i < p; i++) {
        int val = power(candidate, i, p);
        if (seen[val])
            return 0;
        seen[val] = 1;
    }
    return 1;
}

int main() {
    int p;
    printf("Enter a prime number: ");
    scanf("%d", &p);

    if (p <= 1) {
        printf("Input must be a prime number greater than 1\n");
        return 0;
    }

    printf("Primitive roots of %d are:\n", p);
    for (int g = 2; g < p; g++) {
        if (is_primitive_root(g, p)) {
            printf("%d ", g);
        }
    }
    printf("\n");
    return 0;
}
```



### ❖ Output :

```
rudy@rudy:~/Desktop/5sem/bishnu-chalise/crypto/files$ n=9; gcc ${n}.c -o /tmp/a.out && /tmp/a.out && ssc ${n}.png && clear
Enter a prime number: 17
Primitive roots of 17 are:
3 5 6 7 10 11 12 14
█
```

### ❖ Analysis:

The algorithm efficiently locates primitive roots by initially finding the divisors of  $(p-1)$  and then verifying if a candidate  $(g)$  raised to the power of these divisors is not congruent to 1 modulo  $(p)$ . This confirms that  $(g)$  generates all integers from 1 to  $(p-1)$  because  $(g^d \not\equiv 1 \pmod{p})$  for any divisor  $(d)$  of  $(p-1)$ . The algorithm's time complexity is primarily determined by the modular exponentiation, resulting in an overall complexity of  $(O(p \log p))$ , where  $(p)$  is the prime number.

## Lab 10: WAP to implement Diffie Hellman algorithm.

### Theory:

#### ❖ Introduction:

The Diffie-Hellman key exchange is a cryptographic protocol enabling two parties to securely establish a shared secret key across an insecure communication channel. Its security relies on the computational difficulty of the discrete logarithm problem.

In this exchange, both participants agree on a large prime number ( $p$ ) and a primitive root ( $g$ ) modulo ( $p$ ). Each party then generates a unique private key, calculates a corresponding public value, and exchanges these public values. Subsequently, using the received public value and their own private key, each party can independently compute the identical shared secret key.

#### ❖ Key Concepts and Parameters :

- Prime Number ( $p$ ): A large prime number publicly agreed upon.
- Primitive Root ( $g$ ): A primitive root modulo ( $p$ ), also agreed upon by both parties.
- Private Key: A secret number known only to each individual party.
- Public Key: A number calculated as  $(g^{\text{private key}}) \bmod p$ .
- Shared Secret: The identical key that both parties can compute using the other's public key and their own private key.

#### ❖ Mathematical Representation:

Both parties agree on a prime number  $p$  and a primitive root  $g$ .

Each party selects a private key  $a$  and  $b$  (these are kept secret).

The public keys are calculated as:

$$A = g^a \bmod p \quad (\text{for party A})$$

$$B = g^b \bmod p \quad (\text{for party B})$$

The shared secret is computed as:

$$\text{Shared Secret} = B^a \bmod p \quad (\text{for party A})$$

$$\text{Shared Secret} = A^b \bmod p \quad (\text{for party B})$$

Both will end up with the same shared secret.

#### ❖ Algorithm:

##### 1. Public Parameters:

- Both parties agree on a prime number  $p$  and a base  $g$  (publicly known).

##### 2. Private Keys:

- Party A picks a private key  $a$ .
- Party B picks a private key  $b$ .

### 3. Public Keys:

- Party A calculates  $A = g^a \% p$  and sends A to Party B.
- Party B calculates  $B = g^b \% p$  and sends B to Party A.

### 4. Shared Secret:

- Party A computes the shared secret  $s = B^a \% p$ .
- Party B computes the shared secret  $s = A^b \% p$ .

(Both parties now have the same shared secret s.)

### ❖ Source Code :

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

```
long long int power_mod(long long int base, long long int exp, long long int mod) {
    long long int result = 1;
    while (exp > 0) {
        if (exp % 2 == 1) {
            result = (result * base) % mod;
        }
        base = (base * base) % mod;
        exp /= 2;
    }
    return result;
}
```

```
int main() {
    long long int p, g, a_private, b_private, a_public, b_public, shared_secret_a, shared_secret_b;

    p = 23;
    g = 5;

    printf("Enter private key for A: ");
    scanf("%lld", &a_private);

    printf("Enter private key for B: ");
    scanf("%lld", &b_private);

    a_public = power_mod(g, a_private, p);
    b_public = power_mod(g, b_private, p);

    shared_secret_a = power_mod(b_public, a_private, p);
    shared_secret_b = power_mod(a_public, b_private, p);

    printf("Public key of A: %lld\n", a_public);
    printf("Public key of B: %lld\n", b_public);
    printf("Shared secret computed by A: %lld\n", shared_secret_a);
    printf("Shared secret computed by B: %lld\n", shared_secret_b);

    return 0;
}
```

### ❖ Output:

```
rudy@rudy:~/Desktop/5sem/bishnu-chalise/crypto$ n=10; gcc ${n}.c -o /tmp/a.out && /tmp/a.out && ssc ${n}.png
Enter private key for A: 1234
Enter private key for B: 4321
Public key of A: 2
Public key of B: 11
Shared secret computed by A: 6
Shared secret computed by B: 6
```

### ❖ Analysis:

This implementation of the Diffie-Hellman key exchange illustrates how two parties can arithmetic and the computational hardness of the discrete logarithm problem, this secure communication protocols.

securely establish a common secret across an insecure channel. Leveraging modular algorithm is a fundamental component of contemporary cryptography, particularly in

## Lab 11: WAP to implement Euclidean algorithm.

### Theory:

#### ❖ Introduction:

The Euclidean Algorithm provides an efficient way to determine the Greatest Common Divisor (GCD) of two integers. The GCD is the largest integer that divides both numbers evenly. The algorithm operates on the principle that  $\text{GCD}(a,b)=\text{GCD}(b,a\text{mod}b)$  when  $a>b$ .

#### ❖ Key Concepts and Parameters:

- Greatest Common Divisor (GCD): The largest integer that divides two or more numbers without any remainder.
- Modular Arithmetic: The process of finding the remainder of a division operation, a core component of the Euclidean Algorithm.
- Iterative Process: The algorithm repeatedly replaces the larger number with its remainder when divided by the smaller number. This continues until one of the numbers becomes 0, at which point the non-zero number is the GCD.

#### ❖ Mathematical Representation:

Given two integers a and b, where  $a>b$ , the Euclidean algorithm follows these steps:

$a=b\times q+r$  , where r is the remainder.

Replace a with b and b with r.

Repeat until  $b=0$ . The GCD is the last non-zero remainder.

#### ❖ Algorithm:

1. Given two integers a and b, where  $a \geq b$ .
2. Divide a by b, getting a quotient q and a remainder r:
  - $a = b * q + r$
3. If  $r = 0$ , then  $\text{gcd}(a, b) = b$ .
4. Otherwise, replace a with b and b with r, and repeat the process.

#### ❖ Source Code:

```
#include <stdio.h>

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

```

#include <stdio.h>

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

int main() {
    long long int a, b;

    printf("Enter two numbers: ");
    scanf("%lld %lld", &a, &b);

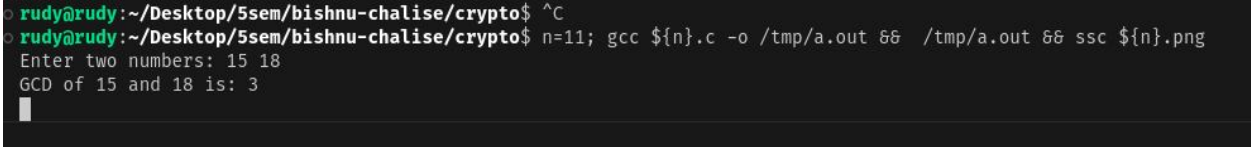
    long long int gcd = euclidean_algorithm(a, b);

    printf("GCD of %lld and %lld is: %lld\n", a, b, gcd);

    return 0;
}

```

### ❖ Output:



```

rudy@rudy:~/Desktop/5sem/bishnu-chalise/crypto$ ^C
rudy@rudy:~/Desktop/5sem/bishnu-chalise/crypto$ n=11; gcc ${n}.c -o /tmp/a.out && /tmp/a.out && ssc ${n}.png
Enter two numbers: 15 18
GCD of 15 and 18 is: 3

```

### ❖ Analysis:

The Euclidean algorithm is a straightforward yet effective technique for calculating the GCD of two integers. Its efficiency renders it well-suited for applications within cryptography and number theory. The provided C implementation showcases the practical application of this algorithm, delivering an efficient means of computing the greatest common divisor.

## Lab 12: WAP to implement Extended Euclidean algorithm.

### Theory:

#### ❖ Introduction:

The Extended Euclidean Algorithm is a more comprehensive version of the standard Euclidean Algorithm. In addition to calculating the Greatest Common Divisor (GCD) of two integers (a) and (b), it also determines integers (x) and (y) (known as Bézout's coefficients) that satisfy the equation:

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

This capability is valuable in several applications, including solving Diophantine equations and finding modular inverses in cryptography (for example, in RSA).

#### ❖ Key Concepts and Parameters:

- Bézout's Identity: This fundamental theorem asserts that the GCD of two integers (a) and (b) can be expressed as a linear combination of (a) and (b), specifically as  $(ax + by = \text{GCD}(a, b))$  for some integer coefficients (x) and (y).
- Extended Euclidean Algorithm: This algorithm not only computes the GCD of (a) and (b) but simultaneously finds the corresponding Bézout's coefficients (x) and (y).
- Modular Inverse: The Extended Euclidean Algorithm can be employed to find the modular inverse of a number modulo (m), a particularly useful operation in various cryptographic algorithms.

#### ❖ Mathematical Representation:

The Extended Euclidean Algorithm solves the equation:

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

By recursively applying the Euclidean algorithm, we can find x and y. The steps are as follows:

If  $b=0$ , then  $\text{GCD}(a,b)=a$ , and we set  $x=1$  and  $y=0$ .

Otherwise, apply the Euclidean algorithm to a and b to get the quotient  $q=a/b$  and the remainder  $r=a \% b$ .

Recursively compute the GCD for b and r, and update the coefficients x and y using the recurrence relations.

### ❖ Algorithm:

1. Apply the Euclidean algorithm to find the GCD of a and b.
2. Track the coefficients x and y during the process.
3. Once the GCD is found, you can also extract x and y from the results.
4. Given two integers a and b, apply the division:

$$a = b \cdot q_1 + r_1$$

Continue until the remainder becomes zero. The last non-zero remainder is the GCD.

5. The extended part calculates x and y in the equation:

$$\text{GCD}(a,b) = a \cdot x + b \cdot y$$

### ❖ Source Code:

```
#include <stdio.h>

long long int extended_euclidean_algorithm
(
    long long int a,
    long long int b,
    long long int *x,
    long long int *y
) {
    if (b == 0) {
        *x = 1;
        *y = 0;
        return a;
    }

    long long int x1, y1;
    long long int gcd = extended_euclidean_algorithm(b, a % b, &x1, &y1);

    *x = y1;
    *y = x1 - (a / b) * y1;

    return gcd;
}

int main() {
    long long int a, b, x, y;

    printf("Enter two numbers: ");
    scanf("%lld %lld", &a, &b);

    long long int gcd = extended_euclidean_algorithm(a, b, &x, &y);

    printf("GCD of %lld and %lld is: %lld\n", a, b, gcd);
    printf("Coefficients x and y are: x = %lld, y = %lld\n", x, y);

    return 0;
}
```



### ❖ Output:

```
o rudy@rudy:~/Desktop/5sem/bishnu-chalise/crypto$ n=12; gcc ${n}.c -o /tmp/a.out && /tmp/a.out && ssc ${n}.png
Enter two numbers: 15 18
GCD of 15 and 18 is: 3
Coefficients x and y are: x = -1, y = 1
█
```

### ❖ Analysis:

The Extended Euclidean Algorithm is a robust enhancement of the Euclidean algorithm. Beyond calculating the GCD of two integers, it also determines the coefficients (x) and (y) that fulfill the linear Diophantine equation ( $ax + by = \text{GCD}(a, b)$ ). This capability is especially valuable for solving modular equations and in various cryptographic applications. The provided C implementation effectively illustrates how to compute both the GCD and Bézout's coefficients.