Lab No:1

WAP to implement Caesar Cipher

The Caesar Cipher is a simple substitution cipher where each letter in the plaintext is shifted by a fixed number of positions in the alphabet.

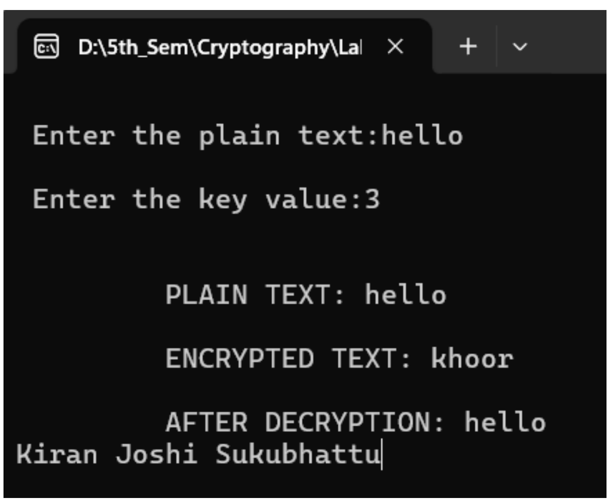
Algorithm:
1. Input: Plaintext and key (shift value).
2. Convert each letter to its corresponding shifted value using the formula.
3. Output the ciphertext.
4. For decryption, reverse the shift using the decryption formula.

Source Code:

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

int main()
{
    char plain[10], cipher[10];
    int key, i, length;
    int result;

    printf("\n Enter the plain text:");
    scanf("%s", plain);
    printf("\n Enter the key value:");
    scanf("%d", &key);
    printf("\n \n \t PLAIN TEXT: %s", plain);
    printf("\n \n \t ENCRYPTED TEXT: ");

    for(i = 0, length = strlen(plain); i < length; i++)
    {
        cipher[i] = plain[i] + key;
        if (isupper(plain[i]) && (cipher[i] > 'Z'))
```

```
        cipher[i] = cipher[i] - 26;
      if (islower(plain[i]) && (cipher[i] > 'z'))
        cipher[i] = cipher[i] - 26;
      printf("%c", cipher[i]);
    }
    printf("\n \n \t AFTER DECRYPTION: ");
    for(i = 0; i < length; i++)
    {
      plain[i] = cipher[i] - key;
      if (isupper(cipher[i]) && (plain[i] < 'A'))
        plain[i] = plain[i] + 26;
      if (islower(cipher[i]) && (plain[i] < 'a'))
        plain[i] = plain[i] + 26;
      printf("%c", plain[i]);
    }
    printf("Kiran Joshi Sukubhattu");
    return 0;
}
```

Output:



Lab No:2

WAP to implement Shift Cipher

The Shift Cipher (Caesar Cipher) is a simple substitution cipher where each letter in the plaintext is shifted by a fixed number of positions in the alphabet.

Algorithm:

1. Input: Plaintext and key (shift value).

2. Convert each letter to its corresponding shifted value using the formula.

3. Output the ciphertext.

4. For decryption, reverse the shift using the decryption formula.

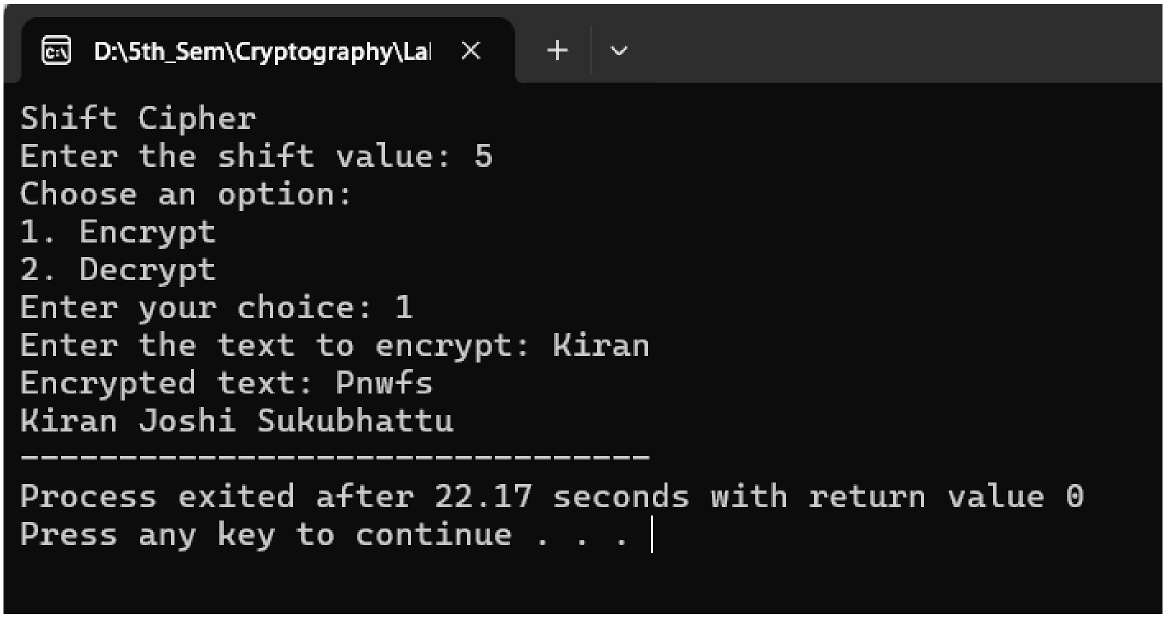
Source Code:

```
#include <stdio.h>

#include <string.h>

// Function to encrypt text using shift cipher

void encrypt(int shift) {

    char text[100];

    printf("Enter the text to encrypt: ");

    fgets(text, sizeof(text), stdin);

    text[strcspn(text, "\n")] = 0; // Remove newline character

    for (int i = 0; text[i] != '\0'; i++) {

        char ch = text[i];

        if (ch >= 'A' && ch <= 'Z') {

            text[i] = (ch - 'A' + shift) % 26 + 'A';

        } else if (ch >= 'a' && ch <= 'z') {

            text[i] = (ch - 'a' + shift) % 26 + 'a';

        }

    }

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

}
```

```
// Function to decrypt text using shift cipher
void decrypt(int shift) {
    char text[100];
    printf("Enter the text to decrypt: ");
    fgets(text, sizeof(text), stdin);
    text[strcspn(text, "\n")] = 0; // Remove newline character
    for (int i = 0; text[i] != '\0'; i++) {
        char ch = text[i];
        if (ch >= 'A' && ch <= 'Z') {
            text[i] = (ch - 'A' - shift + 26) % 26 + 'A';
        } else if (ch >= 'a' && ch <= 'z') {
            text[i] = (ch - 'a' - shift + 26) % 26 + 'a';
        }
    }
    printf("Decrypted text: %s\n", text);
}

int main() {
    int shift, choice;
    printf("Shift Cipher\n");
    printf("Enter the shift value: ");
    scanf("%d", &shift);
    getchar(); // Consume newline character
    printf("Choose an option:\n1. Encrypt\n2. Decrypt\nEnter your choice: ");
    scanf("%d", &choice);
    getchar(); // Consume newline character
    if (choice == 1) {
        encrypt(shift);
    } else if (choice == 2) {
        decrypt(shift);
```

```
    } else {
        printf("Invalid choice!\n");  }
    printf("\nKiran Joshi Sukubhattu");
    return 0;
}
```

Output:



Lab No:3

WAP to implement Rail Fence Cipher

The Rail Fence Cipher is a transposition cipher that encrypts a message by placing characters in a zigzag pattern across multiple rows (rails) and then reading them row-wise.

Algorithm
Encryption:
The text is written diagonally across n rails.
The message is read row-wise to get the ciphertext.
Decryption:
The ciphertext is placed in the zigzag pattern.
The message is reconstructed by reading it row-wise.

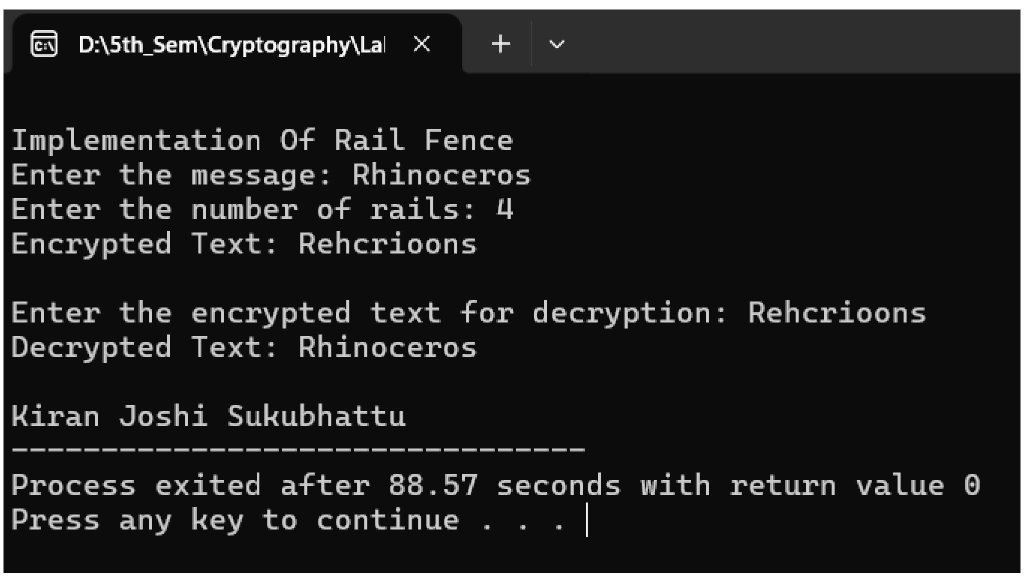
Source Code:

```
#include <stdio.h>
#include <string.h>
#define MAX 100
// Function to encrypt using Rail Fence Cipher
void encryptRailFence(char text[], int rails) {
    char rail[rails][MAX];
    int len = strlen(text), row = 0, dir = 1;
    // Initialize rail matrix
    for (int i = 0; i < rails; i++)
        for (int j = 0; j < len; j++)
            rail[i][j] = '\n';
    // Fill matrix in zigzag pattern
    for (int i = 0; i < len; i++) {
        rail[row][i] = text[i];
        row += dir;
        if (row == 0 || row == rails - 1)
            dir *= -1;
    }
    // Read matrix row-wise for ciphertext
    printf("Encrypted Text: ");
    for (int i = 0; i < rails; i++)
```

```
        for (int j = 0; j < len; j++)
            if (rail[i][j] != '\n')
                printf("%c", rail[i][j]);
    printf("\n");
}
void decryptRailFence(char cipher[], int rails) {
    char rail[rails][MAX];
    int len = strlen(cipher), row = 0, dir = 1, index = 0;
    // Initialize rail matrix
    for (int i = 0; i < rails; i++)
        for (int j = 0; j < len; j++)
            rail[i][j] = '\n';
    // Mark zigzag pattern positions
    for (int i = 0; i < len; i++) {
        rail[row][i] = '*';
        row += dir;
        if (row == 0 || row == rails - 1)
            dir *= -1;
    }
    for (int i = 0; i < rails; i++)
        for (int j = 0; j < len; j++)
            if (rail[i][j] == '*' && index < len)
                rail[i][j] = cipher[index++];
    // Read matrix in zigzag order to reconstruct the plaintext
    row = 0, dir = 1;
    printf("Decrypted Text: ");
    for (int i = 0; i < len; i++) {
        printf("%c", rail[row][i]);
        row += dir;
        if (row == 0 || row == rails - 1)
            dir *= -1;
```

```
    }
    printf("\n");
}
int main() {
    char message[MAX];
    int rails;
    printf("\nImplementation Of Rail Fence\n");
    printf("Enter the message: ");
    scanf("%s", message);
    printf("Enter the number of rails: ");
    scanf("%d", &rails);
    encryptRailFence(message, rails);
    char cipher[MAX];
    printf("\nEnter the encrypted text for decryption: ");
    scanf("%s", cipher);
    decryptRailFence(cipher, rails);
    printf("\nKiran Joshi Sukubhattu");
    return 0;
}
```

Output:



```
D:\5th_Sem\Cryptography\Lal

Implementation Of Rail Fence
Enter the message: Rhinoceros
Enter the number of rails: 4
Encrypted Text: Rehcrioons

Enter the encrypted text for decryption: Rehcrioons
Decrypted Text: Rhinoceros

Kiran Joshi Sukubhattu
--------------------------------
Process exited after 88.57 seconds with return value 0
Press any key to continue . . .
```

Lab No:4

WAP to implement hill cipher.

The Hill Cipher is a polygraphic substitution cipher based on matrix multiplication in modular arithmetic.

Algorithm:

Encryption Process

Convert the plaintext into numerical values (A=0, B=1, ..., Z=25).
Represent the plaintext as a matrix.
Multiply the plaintext matrix by the encryption key matrix.
Compute the result mod 26 to get the ciphertext.

Decryption Process

Compute the inverse of the encryption key matrix mod 26.
Multiply the ciphertext matrix by the inverse key matrix.
Compute the result mod 26 to get the plaintext.

Source Code:

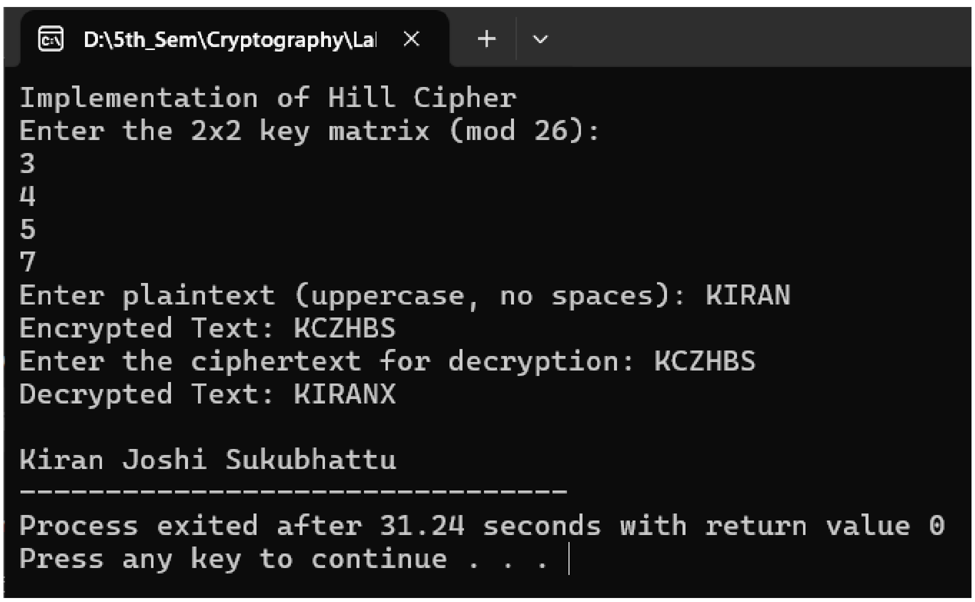
```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MOD 26
// Function to compute determinant of a 2x2 matrix
int determinant(int key[2][2]) {
   return (key[0][0] * key[1][1] - key[0][1] * key[1][0]);
}
// Function to compute modular inverse of a number under mod 26
int modInverse(int num) {
   num = num % MOD;
   for (int i = 1; i < MOD; i++)
      if ((num * i) % MOD == 1)
         return i;
   return -1;
}
// Function to compute the inverse of a 2x2 matrix mod 26
void inverseKey(int key[2][2], int invKey[2][2]) {
   int det = determinant(key);
```

```
    int detInv = modInverse(det);
    if (detInv == -1) {
        printf("Matrix is not invertible!\n");
        exit(1);
    }
    // Compute adjugate matrix
    invKey[0][0] = key[1][1];
    invKey[1][1] = key[0][0];
    invKey[0][1] = -key[0][1];
    invKey[1][0] = -key[1][0];
    // Multiply by modular inverse of determinant
    for (int i = 0; i < 2; i++)
        for (int j = 0; j < 2; j++)
            invKey[i][j] = (invKey[i][j] * detInv) % MOD;
    // Ensure values are positive mod 26
    for (int i = 0; i < 2; i++)
        for (int j = 0; j < 2; j++)
            if (invKey[i][j] < 0)
                invKey[i][j] += MOD;
}
// Function to encrypt using Hill Cipher
void encrypt(char plaintext[], int key[2][2]) {
    int len = strlen(plaintext);
    if (len % 2 != 0) {
        strcat(plaintext, "X"); // Padding if odd length
        len++;
    }
    printf("Encrypted Text: ");
    for (int i = 0; i < len; i += 2) {
        int P[2] = {plaintext[i] - 'A', plaintext[i+1] - 'A'};
        int C[2];
```

```
        // Matrix multiplication
        for (int j = 0; j < 2; j++)
            C[j] = (key[j][0] * P[0] + key[j][1] * P[1]) % MOD;
        printf("%c%c", C[0] + 'A', C[1] + 'A');
    }
    printf("\n");
}
// Function to decrypt using Hill Cipher
void decrypt(char ciphertext[], int key[2][2]) {
    int len = strlen(ciphertext);
    int invKey[2][2];
    // Compute inverse of key matrix
    inverseKey(key, invKey);
    printf("Decrypted Text: ");
    for (int i = 0; i < len; i += 2) {
        int C[2] = {ciphertext[i] - 'A', ciphertext[i+1] - 'A'};
        int P[2];
        // Matrix multiplication with inverse key
        for (int j = 0; j < 2; j++)
            P[j] = (invKey[j][0] * C[0] + invKey[j][1] * C[1]) % MOD;
        printf("%c%c", P[0] + 'A', P[1] + 'A');
    }
    printf("\n");
}
int main() {
    char plaintext[100];
    char ciphertext[100];
    int key[2][2];
    printf("Implementation of Hill Cipher\n");
    printf("Enter the 2x2 key matrix (mod 26):\n");
    for (int i = 0; i < 2; i++)
```

```
        for (int j = 0; j < 2; j++)
            scanf("%d", &key[i][j]);
    printf("Enter plaintext (uppercase, no spaces): ");
    scanf("%s", plaintext);
    encrypt(plaintext, key);
    printf("Enter the ciphertext for decryption: ");
    scanf("%s", ciphertext);
    decrypt(ciphertext, key);
    printf("\nKiran Joshi Sukubhattu");
    return 0;
}
```

Output:



Lab No:5

WAP to implement vernam cipher.

The Vernam Cipher is a symmetric encryption algorithm that uses a random key of the same length as the plaintext.

Algorithm:
Encryption
Convert plaintext characters to numeric values (A=0, B=1, ..., Z=25).
Convert key characters to numeric values.
Compute: $C[i]=(P[i]\oplus K[i]) \bmod 26$
Convert numbers back to letters.
Decryption
Convert ciphertext characters to numeric values.
Convert key characters to numeric values.
Compute: $P[i]=(C[i]\oplus K[i]) \bmod$
Convert numbers back to letters.

Source Code:

```
#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <time.h>

// Function to generate a random key of the same length as plaintext

void generateKey(char plaintext[], char key[]) {

    int len = strlen(plaintext);

    for (int i = 0; i < len; i++) {

        key[i] = 'A' + (rand() % 26);  // Generate random uppercase letter

    }

    key[len] = '\0';  // Null-terminate the key

}


// Function to encrypt using Vernam Cipher

void encryptVernam(char plaintext[], char key[], char ciphertext[]) {

    int len = strlen(plaintext);

    for (int i = 0; i < len; i++) {
```

```
        ciphertext[i] = ((plaintext[i] - 'A') ^ (key[i] - 'A')) + 'A';
    }
    ciphertext[len] = '\0';  // Null-terminate the ciphertext
}
// Function to decrypt using Vernam Cipher
void decryptVernam(char ciphertext[], char key[], char decryptedText[]) {
    int len = strlen(ciphertext);
    for (int i = 0; i < len; i++) {
        decryptedText[i] = ((ciphertext[i] - 'A') ^ (key[i] - 'A')) + 'A';
    }
    decryptedText[len] = '\0';  // Null-terminate the decrypted text
}
// Main driver function
int main() {
    char plaintext[100], key[100], ciphertext[100], decryptedText[100];
    printf("Implementaion of Vernam Cipher:\n");
    // Seed random number generator
    srand(time(0));
    // Input plaintext
    printf("Enter plaintext (uppercase, no spaces): ");
    scanf("%s", plaintext);
    // Generate random key
    generateKey(plaintext, key);
    printf("Generated Key: %s\n", key);
    // Encrypt
    encryptVernam(plaintext, key, ciphertext);
    printf("Encrypted Text: %s\n", ciphertext);
    // Decrypt
    decryptVernam(ciphertext, key, decryptedText);
    printf("Decrypted Text: %s\n", decryptedText);
```

```
    printf("Kiran Joshi Sukubhattu");
    return 0;
}
```

Output:

```
D:\5th_Sem\Cryptography\Lal
Implementaion of Vernam Cipher:
Enter plaintext (uppercase, no spaces): LIGHT
Generated Key: ZRSCT
Encrypted Text: SZUFA
Decrypted Text: LIGHT
Kiran Joshi Sukubhattu
--------------------------------
Process exited after 105.5 seconds with return value 0
Press any key to continue . . .
```

Lab No:6

WAP to implement OTP cipher.

The One-Time Pad (OTP) Cipher is essentially the Vernam Cipher, where the key is truly random, as long as the plaintext, and used only once.

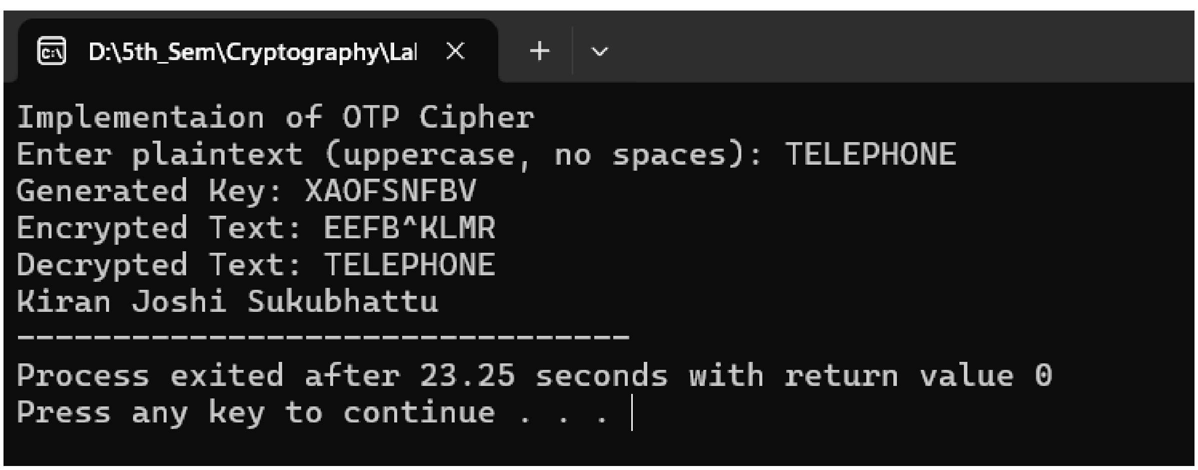
Algorithm:
Generate a random key of the same length as the plaintext.
Encrypt using: $C[i]=(P[i]\oplus K[i]) \bmod 26$
Decrypt using: $P[i]=(C[i]\oplus K[i]) \bmod 26$
Convert numbers back to letters.

Source Code:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
void generateKey(char plaintext[], char key[]) {
    int len = strlen(plaintext);
    for (int i = 0; i < len; i++) {
        key[i] = 'A' + (rand() % 26); // Generate a random uppercase letter
    }
    key[len] = '\0'; // Null-terminate the key
}
void encryptOTP(char plaintext[], char key[], char ciphertext[]) {
    int len = strlen(plaintext);
    for (int i = 0; i < len; i++) {
        ciphertext[i] = ((plaintext[i] - 'A') ^ (key[i] - 'A')) + 'A';
    }
    ciphertext[len] = '\0'; // Null-terminate the ciphertext
}
void decryptOTP(char ciphertext[], char key[], char decryptedText[]) {
    int len = strlen(ciphertext);
    for (int i = 0; i < len; i++) {
        decryptedText[i] = ((ciphertext[i] - 'A') ^ (key[i] - 'A')) + 'A';
```

```
    }
    decryptedText[len] = '\0'; // Null-terminate the decrypted text
}
int main() {
    char plaintext[100], key[100], ciphertext[100], decryptedText[100];
    printf("Implementaion of OTP Cipher\n");
    srand(time(0))
    printf("Enter plaintext (uppercase, no spaces): ");
    scanf("%s", plaintext);
    generateKey(plaintext, key);
    printf("Generated Key: %s\n", key);
    encryptOTP(plaintext, key, ciphertext);
    printf("Encrypted Text: %s\n", ciphertext);
    decryptOTP(ciphertext, key, decryptedText);
    printf("Decrypted Text: %s\n", decryptedText);
    printf("Kiran Joshi Sukubhattu");
    return 0;
}
```

Output:



Lab No:7

Write a program to implement playfair cipher

The Playfair Cipher is a digraph substitution cipher that encrypts text in paris of letters using a 5×5 key matrix.

Algorithm:

Encryption

Create the key matrix from the keyword.
Format the plaintext (remove spaces, duplicate letter rule).
Encrypt letter pairs based on Playfair rules.

Decryption

Use the same key matrix.
Reverse the Playfair rules to retrieve plaintext.

Source Code:

```
#include <stdio.h>

#include <string.h>

#include <ctype.h>

// 5x5 Playfair key matrix

char keyMatrix[5][5];

// Function to check if a character is already in the matrix

int isCharInMatrix(char c) {

    for (int i = 0; i < 5; i++)

        for (int j = 0; j < 5; j++)

            if (keyMatrix[i][j] == c)

                return 1;

    return 0;

}

// Function to generate the Playfair key matrix

void generateKeyMatrix(char key[]) {

    char tempKey[26] = {0}; // Store unique letters

    int index = 0;

    // Convert key to uppercase and remove duplicates

    for (int i = 0; key[i] != '\0'; i++) {
```

```
        char c = toupper(key[i]);
        if (c == 'J') c = 'I'; // Treat I and J as the same
        if (!isCharInMatrix(c) && isalpha(c)) {
            tempKey[index++] = c;
        }
    }
    // Fill matrix with unique letters of the key
    int x = 0, y = 0;
    for (int i = 0; i < index; i++) {
        keyMatrix[x][y++] = tempKey[i];
        if (y == 5) { x++; y = 0; }
    }
    // Fill remaining letters
    for (char c = 'A'; c <= 'Z'; c++) {
        if (c == 'J') continue; // Skip 'J'
        if (!isCharInMatrix(c)) {
            keyMatrix[x][y++] = c;
            if (y == 5) { x++; y = 0; }
        }
    }
}
// Function to display the key matrix
void displayKeyMatrix() {
    printf("\nPlayfair Key Matrix:\n");
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 5; j++) {
            printf("%c ", keyMatrix[i][j]);
        }
        printf("\n");
    }
```

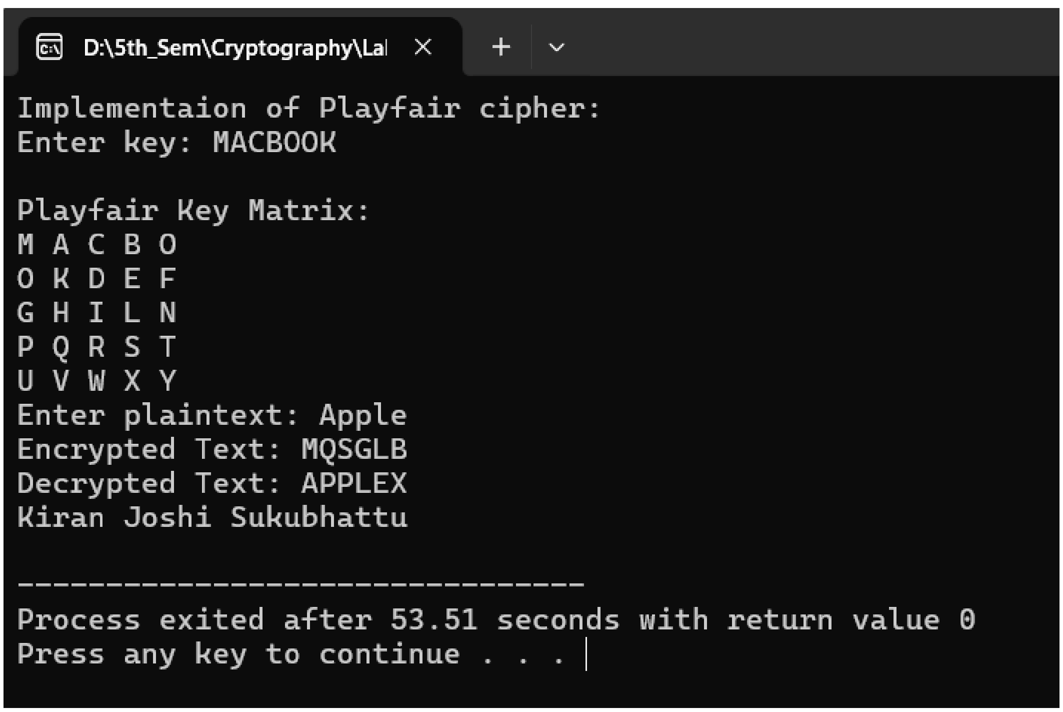
```
}
// Function to get position of a character in the key matrix
void getPosition(char c, int *row, int *col) {
    if (c == 'J') c = 'I'; // Treat 'J' as 'I'
    for (int i = 0; i < 5; i++)
        for (int j = 0; j < 5; j++)
            if (keyMatrix[i][j] == c) {
                *row = i;
                *col = j;
                return;
            }
}
// Function to prepare plaintext (remove spaces, handle duplicate letters)
void formatPlaintext(char plaintext[], char formattedText[]) {
    int len = strlen(plaintext);
    int j = 0;
    for (int i = 0; i < len; i++) {
        if (isalpha(plaintext[i])) {
            formattedText[j++] = toupper(plaintext[i]);
        }
    }
    formattedText[j] = '\0';
    // Insert 'X' between duplicate letters and pad odd-length text
    char finalText[100];
    int k = 0;
    for (int i = 0; i < j; i += 2) {
        finalText[k++] = formattedText[i];
        if (i + 1 < j) {
            if (formattedText[i] == formattedText[i + 1])
                finalText[k++] = 'X'; // Insert 'X' if pair is the same
```

```
            finalText[k++] = formattedText[i + 1];
        }
    }
    if (k % 2 != 0) finalText[k++] = 'X'; // Pad with 'X' if odd-length
    finalText[k] = '\0';
    strcpy(formattedText, finalText);
}
// Function to encrypt a pair of characters
void encryptPair(char a, char b, char *enc1, char *enc2) {
    int row1, col1, row2, col2;
    getPosition(a, &row1, &col1);
    getPosition(b, &row2, &col2);
    if (row1 == row2) { // Same row: Shift right
        *enc1 = keyMatrix[row1][(col1 + 1) % 5];
        *enc2 = keyMatrix[row2][(col2 + 1) % 5];
    } else if (col1 == col2) { // Same column: Shift down
        *enc1 = keyMatrix[(row1 + 1) % 5][col1];
        *enc2 = keyMatrix[(row2 + 1) % 5][col2];
    } else { // Rectangle swap
        *enc1 = keyMatrix[row1][col2];
        *enc2 = keyMatrix[row2][col1];
    }
}
// Function to encrypt the text
void encryptText(char plaintext[], char ciphertext[]) {
    formatPlaintext(plaintext, plaintext);
    int len = strlen(plaintext);
    for (int i = 0; i < len; i += 2) {
        encryptPair(plaintext[i], plaintext[i + 1], &ciphertext[i], &ciphertext[i + 1]);
    }
```

```
    ciphertext[len] = '\0';
}
// Function to decrypt a pair of characters
void decryptPair(char a, char b, char *dec1, char *dec2) {
    int row1, col1, row2, col2;
    getPosition(a, &row1, &col1);
    getPosition(b, &row2, &col2);
    if (row1 == row2) { // Same row: Shift left
        *dec1 = keyMatrix[row1][(col1 + 4) % 5];
        *dec2 = keyMatrix[row2][(col2 + 4) % 5];
    } else if (col1 == col2) { // Same column: Shift up
        *dec1 = keyMatrix[(row1 + 4) % 5][col1];
        *dec2 = keyMatrix[(row2 + 4) % 5][col2];
    } else { // Rectangle swap
        *dec1 = keyMatrix[row1][col2];
        *dec2 = keyMatrix[row2][col1];
    }
}
// Function to decrypt the text
void decryptText(char ciphertext[], char decryptedText[]) {
    int len = strlen(ciphertext);
    for (int i = 0; i < len; i += 2) {
        decryptPair(ciphertext[i], ciphertext[i + 1], &decryptedText[i], &decryptedText[i + 1]);
    }
    decryptedText[len] = '\0';
}
int main() {
    char plaintext[100], ciphertext[100], decryptedText[100], key[100];
    printf("Implementaion of Playfair cipher:\n");
    printf("Enter key: ");
```

```
    scanf("%s", key);
    generateKeyMatrix(key);
    displayKeyMatrix();
    printf("Enter plaintext: ");
    scanf("%s", plaintext);
    encryptText(plaintext, ciphertext);
    printf("Encrypted Text: %s\n", ciphertext);
    decryptText(ciphertext, decryptedText);
    printf("Decrypted Text: %s\n", decryptedText);
    printf("Kiran Joshi Sukubhattu\n");
    return 0;
}
```

Output:



```
D:\5th_Sem\Cryptography\Lal

Implementaion of Playfair cipher:
Enter key: MACBOOK

Playfair Key Matrix:
M A C B O
O K D E F
G H I L N
P Q R S T
U V W X Y
Enter plaintext: Apple
Encrypted Text: MQSGLB
Decrypted Text: APPLEX
Kiran Joshi Sukubhattu

--------------------------------
Process exited after 53.51 seconds with return value 0
Press any key to continue . . .
```

Lab no:8

WAP to implement RSA
The RSA algorithm is an asymmetric encryption algorithm that uses two keys:
Public Key (e, n) for encryption
Private Key (d, n) for decryption
The security of RSA relies on the difficulty of factoring large prime numbers.
Algorithm:
Generate keys (p, q, n, φ(n), e, d).
Encrypt message using $C=M^e \bmod n$
Decrypt message using $M=C^d$ dmod n

Source Code:

```
#include <stdio.h>

#include <stdlib.h>

#include <math.h>

// Function to compute gcd (Greatest Common Divisor)

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 modExp(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 modular inverse (d = e^(-1) mod phi)
int modInverse(int e, int phi) {
    for (int d = 2; d < phi; d++) {
        if ((d * e) % phi == 1)
            return d;
    }
    return -1; // No modular inverse found
}
int main() {
    int p, q, n, phi, e, d, message, encrypted, decrypted;
    printf("RSA Implementation:\n");
    // Step 1: Input two prime numbers p and q
    printf("Enter two prime numbers (p and q): ");
    scanf("%d %d", &p, &q);
    // Step 2: Compute n and Euler's Totient function phi(n)
    n = p * q;
    phi = (p - 1) * (q - 1);
    // Step 3: Choose an encryption exponent e (1 < e < phi, and gcd(e, phi) = 1)
    for (e = 2; e < phi; e++) {
        if (gcd(e, phi) == 1)
            break;
    }
   // Step 4: Compute the decryption exponent d
    d = modInverse(e, phi);
    if (d == -1) {
        printf("Error finding modular inverse.\n");
        return 1;
    }
```

```
    // Display the generated keys
    printf("\nPublic Key: (e = %d, n = %d)\n", e, n);
    printf("Private Key: (d = %d, n = %d)\n", d, n);
    printf("\nEnter the message (integer format): ");
    scanf("%d", &message);
    encrypted = modExp(message, e, n);
    printf("Encrypted Message: %d\n", encrypted);
    decrypted = modExp(encrypted, d, n);
    printf("Decrypted Message: %d\n", decrypted);
    printf("Kiran Joshi Sukubhattu\n");
    return 0;
}
```

Output:

```
D:\5th_Sem\Cryptography\Lal

RSA Implementation:
Enter two prime numbers (p and q): 31 7

Public Key: (e = 7, n = 217)
Private Key: (d = 103, n = 217)

Enter the message (integer format): 13
Encrypted Message: 146
Decrypted Message: 13
Kiran Joshi Sukubhattu

--------------------------------
Process exited after 83.7 seconds with return value 0
Press any key to continue . . .
```

Lab No:9

WAP to find primitive root of a prime number.

A primitive root of a prime number p is an integer g such that its powers generate all numbers from 1 to p−1modulo p.

Algorithm:

Find all factors of ϕ(p) (which is p−1).

Check if g((p−1)/factor) mod p≠ 1 for all factors.

If condition holds, g is a primitive root.

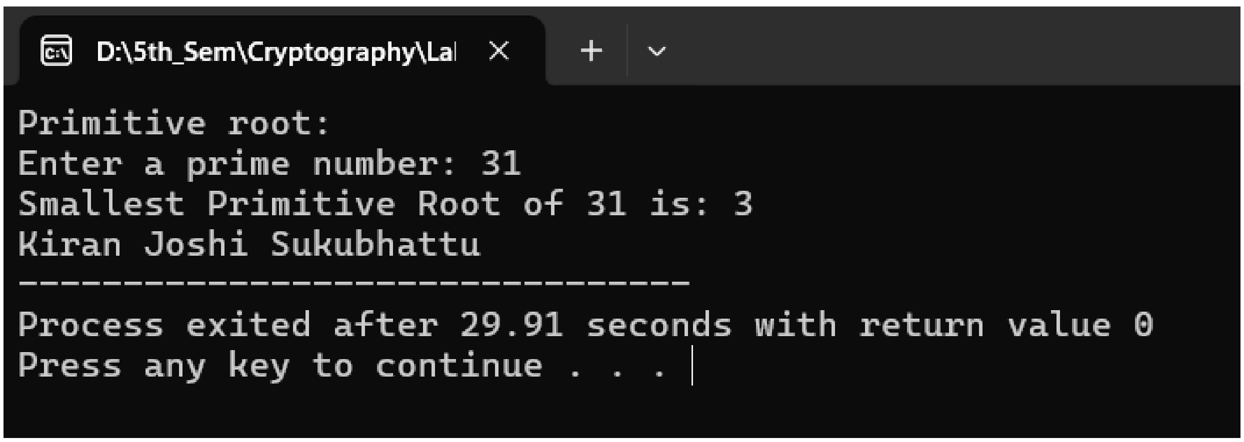
Source Code:

```
#include <stdio.h>

// Function to compute (base^exp) % mod using Modular Exponentiation

int powerMod(int base, int exp, int mod) {

    int result = 1;

    base = base % mod;

    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 find prime factors of (p-1)

int findFactors(int n, int factors[]) {

    int count = 0;

    for (int i = 2; i * i <= n; i++) {

        if (n % i == 0) {

            factors[count++] = i;

            while (n % i == 0)

                n /= i;
```

```
        }
    }
    if (n > 1) {
        factors[count++] = n;
    }
    return count;
}
// Function to find the smallest primitive root of prime p
int findPrimitiveRoot(int p) {
    int phi = p - 1; // Euler's Totient Function (p-1 for prime p)
    int factors[20];
    int numFactors = findFactors(phi, factors);
    for (int g = 2; g < p; g++) { // Start checking from 2
        int isPrimitive = 1;
        for (int i = 0; i < numFactors; i++) {
            if (powerMod(g, phi / factors[i], p) == 1) {
                isPrimitive = 0;
                break;
            }
        }
        if (isPrimitive)
            return g; // Found the smallest primitive root
    }
    return -1; // No primitive root found (shouldn't happen for prime p)
}

int main() {
    int p;
    printf("Primitive root:\n");
    printf("Enter a prime number: ");
```

```
    scanf("%d", &p);
    int root = findPrimitiveRoot(p);
    if (root != -1)
        printf("Smallest Primitive Root of %d is: %d\n", p, root);
    else
        printf("No primitive root found.\n");
    printf("Kiran Joshi Sukubhattu");
    return 0;
}
```

Output:



Lab No:10

WAP to implement Diffie Hellman algorithm.

The Diffie-Hellman key exchange algorithm is used to securely exchange cryptographic keys over a public channel. It allows two parties to generate a shared secret key, which can then be used for further encryption and decryption.

Algorithm:

1. Select a prime number p and base g (public values).
2. Generate private keys a and b (random values).
3. Calculate public keys A and B.
4. Exchange public keys.
5. Compute the shared secret from the received public key.

Source Code:

```
#include <stdio.h>

#include <math.h>

long long modExp(long long base, long long exp, long long mod) {

    long long result = 1;

    base = base % mod; // In case base is greater than mod

    while (exp > 0) {

        if (exp % 2 == 1) {

            result = (result * base) % mod;

        }

        base = (base * base) % mod;

        exp = exp / 2;

    }

    return result;

}

int main() {

    long long p, g, a, b, A, B, secretKey1, secretKey2;

    printf("Implementation of Diffie Hellman algorithm.\n");

    // Input prime number p and base g

    printf("Enter prime number p: ");

    scanf("%d", &p);
```

```
printf("Enter base g: ");
scanf("%d", &g);

// Party 1 selects private key a
printf("\nParty 1: Select private key a: ");
scanf("%d", &a);

// Party 2 selects private key b
printf("\nParty 2: Select private key b: ");
scanf("%d", &b);

// Party 1 computes their public key A
A = modExp(g, a, p);
printf("\nParty 1 computes public key A = g^a mod p = %d\n", A);
// Party 2 computes their public key B
B = modExp(g, b, p);
printf("Party 2 computes public key B = g^b mod p = %d\n", B);

// Exchange public keys (in a real application, this happens over an insecure channel)
// Party 1 computes the shared secret key using Party 2's public key B
secretKey1 = modExp(B, a, p);
printf("\nParty 1 computes shared secret key: secretKey1 = B^a mod p = %d\n", secretKey1);

// Party 2 computes the shared secret key using Party 1's public key A
secretKey2 = modExp(A, b, p);
printf("Party 2 computes shared secret key: secretKey2 = A^b mod p = %d\n", secretKey2);

// Both parties should have the same secret key
if (secretKey1 == secretKey2) {
    printf("\nShared secret key is successfully generated: %d\n", secretKey1);
} else {
```

```
        printf("\nError in key generation. The keys do not match.\n");
    }
    printf("Kiran Joshi Sukubhattu");
    return 0;
}
```

Output:

```
D:\5th_Sem\Cryptography\Lal
Implementation of Diffie-Hellman Key Exchange Algorithm.
Enter prime number p: 23
Enter base g: 5

Party 1: Select private key a: 6

Party 2: Select private key b: 15

Party 1 computes public key A = g^a mod p = 8
Party 2 computes public key B = g^b mod p = 19

Party 1 computes shared secret key: secretKey1 = B^a mod p = 2
Party 2 computes shared secret key: secretKey2 = A^b mod p = 2

Shared secret key successfully generated: 2
Kiran Joshi Sukubhattu

--------------------------------
Process exited after 23.26 seconds with return value 0
Press any key to continue . . .
```

Lab No:11

WAP to implement Euclidean algorithm .

The Euclidean Algorithm is an efficient way to compute the greatest common divisor (GCD) of two numbers. The Euclidean algorithm works by repeatedly applying the following: gcd(a,b)=gcd(b,a%b)

Where:

- a and b are the two numbers whose GCD is to be found.
- The algorithm continues until b=0, at which point a will be the GCD.

Source Code:

```
#include <stdio.h>

// Function to compute GCD using Euclidean algorithm

int gcd(int a, int b) {

    while (b != 0) {

        int temp = b;

        b = a % b;  // Update b with the remainder of a divided by b

        a = temp;   // Update a to b (old value)

    }

    return a;  // When b becomes 0, a will hold the GCD

}

int main() {

    int a, b;

    printf("Euclidean Algorithm\n");

    // Input two numbers

    printf("Enter two numbers: ");

    scanf("%d %d", &a, &b);
```

```c
    // Compute GCD

    int result = gcd(a, b);

    // Output the result

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

    printf("Kiran Joshi Sukubhattu");

    return 0;

}
```

Output:

```
D:\5th_Sem\Cryptography\La
Euclidean Algorithm
Enter two numbers: 47 8
GCD of 47 and 8 is: 1
Kiran Joshi Sukubhattu
--------------------------------
Process exited after 46.33 seconds with return value 0
Press any key to continue . . .
```

Lab No:12

WAP to implement Extended Euclidean algorithm.

The Extended Euclidean Algorithm is an extension of the Euclidean algorithm that not only computes the greatest common divisor (GCD) of two numbers a and b, but also finds the coeffiecients x and y such that:

$a \cdot x + b \cdot y = \gcd(a,b)$

These coefficients x and y are also known as the **Bezout's coefficients**.

Source Code:

```
#include <stdio.h>
int extendedGCD(int a, int b, int *x, int *y) {
    if (b == 0) {
        *x = 1; // Base case: x = 1, y = 0
        *y = 0;
        return a; // GCD is a
    }
    int x1, y1;
    int gcd = extendedGCD(b, a % b, &x1, &y1);
    // Update x and y using the results of the previous call
    *x = y1;
    *y = x1 - (a / b) * y1;
    return gcd;
}
int main() {
    int a, b, x, y;
    printf("Extended Euckidean Algorithm:\n");
    // Input the two numbers
    printf("Enter two numbers: ");
    scanf("%d %d", &a, &b);
    // Call the extended Euclidean algorithm
```

```c
    int gcd = extendedGCD(a, b, &x, &y);
    // Output the results
    printf("GCD of %d and %d is: %d\n", a, b, gcd);
    printf("Coefficients x and y are: %d, %d\n", x, y);
    printf("Equation: %d * %d + %d * %d = %d\n", a, x, b, y, gcd);
    printf("Kiran Joshi Sukubhattu");
    return 0;
}
```

Output:

```
D:\5th_Sem\Cryptography\Lal
Extended Euclidean Algorithm:
Enter two numbers: 510 310
GCD of 510 and 310 is: 10
Coefficients x and y are: 14, -23
Equation: 510 * 14 + 310 * -23 = 10
Kiran Joshi Sukubhattu
--------------------------------
Process exited after 24.58 seconds with return value 0
Press any key to continue . . .
```