

Index

S.N.	Labs	Date	Signature
1.	Write a program to implement caesar cipher.	2081/11/05	
2.	WAP to implement shift cipher.	2081/11/05	
3.	WAP to implement Rail Fence Cipher.	2081/11/10	
4.	WAP to implement hill cipher.	2081/11/10	
5.	WAP to implement vernam cipher.	2081/11/10	
6.	WAP to implement OTP cipher.	2081/11/15	
7.	Write a program to implement playfair cipher.	2081/11/15	
8.	RSA implementation.	2081/11/15	
9.	WAP to find primitive root of a prime number.	2081/11/15	
10.	WAP to implement Diffie Hellman algorithm.	2081/11/22	
11.	WAP to implement Euclidean algorithm.	2081/11/22	
12.	WAP to implement Extended Euclidean algorithm.	2081/11/22	

Title: Write a program to implement caesar cipher.

Theory:

Introduction

The Caesar Cipher is a simple and well-known encryption technique that falls under the category of substitution ciphers. It was named after Julius Caesar, who reportedly used it to encrypt military messages. The cipher works by shifting each letter in the plaintext a fixed number of places in the alphabet, making it an example of a monoalphabetic substitution cipher.

Key Concepts and Parameters

- Plaintext: The original message that needs to be encrypted.
- Ciphertext: The encoded message after applying the shift.
- Shift Key (k): The number of positions each letter in the plaintext is shifted in the alphabet.
- Encryption: The process of converting plaintext into ciphertext.
- Decryption: The process of converting ciphertext back into plaintext.

Mathematical Representation

The encryption and decryption process of the Caesar Cipher can be mathematically represented as:

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.

Source Code:

//Encryption

```
#include<stdio.h>
#include<ctype.h>
int main() {
    char text[500], ch;
    int key;

    // Taking user input.
    printf("Enter a message to encrypt: ");
    scanf("%s", text);
    printf("Enter the key: ");
    scanf("%d", & key);

    // Visiting character by character.
    for (int i = 0; text[i] != '\0'; ++i) {
        ch = text[i];
        // Check for valid characters.
        if (isalnum(ch)) {
            //Lowercase characters.
            if (islower(ch)) {
                ch = (ch - 'a' + key) % 26 + 'a';
            }
            // Uppercase characters.
            if (isupper(ch)) {
                ch = (ch - 'A' + key) % 26 + 'A';
            }

            // Numbers.
            if (isdigit(ch)) {
                ch = (ch - '0' + key) % 10 + '0';
            }
        }
        // Invalid character.
        else {
            printf("Invalid Message");
        }

        // Adding encoded answer.
        text[i] = ch;
    }
    printf("Encrypted message: %s", text);

    return 0;
}
```

//Decryption

```
#include<stdio.h>
#include<ctype.h>
int main() {
    char text[500], ch;
    int key;

    // Taking user input.
    printf("Enter a message to encrypt: ");
    scanf("%s", text);
    printf("Enter the key: ");
    scanf("%d", & key);

    // Visiting character by character.

    for (int i = 0; text[i] != '\0'; ++i) {
        ch = text[i];
        // Check for valid characters.
        if (isalnum(ch)) {
            //Lowercase characters.
            if (islower(ch)) {
                ch = (ch - 'a' + key) % 26 + 'a';
            }
            // Uppercase characters.
            if (isupper(ch)) {
                ch = (ch - 'A' + key) % 26 + 'A';
            }

            // Numbers.
            if (isdigit(ch)) {
                ch = (ch - '0' + key) % 10 + '0';
            }
        }
        // Invalid character.
        else {
            printf("Invalid Message");
        }

        // Adding encoded answer.
        text[i] = ch;
    }

    printf("Encrypted message: %s", text);
    return 0;
}
```

Output:

Encryption:

```
PS D:\Arjun Mijar(109) Lab Reports\Cryptogrphy> .\ceaserEncryption.exe
Enter a message to encrypt: Arjun9800
Enter the key: 3
Encrypted message: Dumxq2133
```

Decryption:

```
PS D:\Arjun Mijar(109) Lab Reports\Cryptogrphy> .\ceaserDecryption.exe
Enter a message to decrypt: Dumxq2133
Enter the key: 3
Decrypted message: Arjun9800
```

Analysis:

The Caesar Cipher is a simple substitution cipher that is easy to implement but vulnerable to brute-force attacks due to its limited number of possible shifts (25). It is primarily used for educational purposes and basic encoding tasks rather than secure encryption.

Title: Write a program 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>
```

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

```
void shiftCipher(char text[], int shift) {  
    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';  
        }  
    }  
}
```

```

void decryptShift(char text[], int shift) {
    shiftCipher(text, 26 - shift);
}

int main() {
    char text[100];
    int shift;
    printf("Enter Message: ");
    scanf("%s", &text);
    printf("Enter shift key: ");
    scanf("%d", &shift);

    printf("Original Text: %s\n", text);
    shiftCipher(text, shift);
    printf("Encrypted Text: %s\n", text);
    decryptShift(text, shift);
    printf("Decrypted Text: %s\n", text);
    return 0;
}

```

Output:

```

PS D:\Arjun Mijar(109) Lab Reports\Cryptogrphy> .\shiftCipher.exe
Enter Message: ArjunMijar8
Enter shift key: 7
Original Text: ArjunMijar8
Encrypted Text: HyqbuTpqhy8
Decrypted Text: ArjunMijar8

```

❖ 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.

Title: WAP to implement Rail Fence Cipher.

Theory:

Introduction:

The Rail Fence Cipher is a type of transposition cipher in which the plaintext is written in a zigzag (rail fence) pattern across multiple "rails" (rows), and then read row-wise to obtain the ciphertext. It does not alter the characters themselves, but changes their positions, making it a permutation cipher.

❖ **Key Concepts and Parameters:**

- Plaintext: The original message to be encrypted.
- Ciphertext: The rearranged message obtained after encryption.
- Number of Rails (n): The number of rows used to form the zigzag pattern.
- Encryption: Writing characters in a zigzag across rails and reading row-wise.
- Decryption: Rebuilding the zigzag structure to restore the original message.

❖ **Mathematical Representation:**

Let the plaintext be arranged in a 2D zigzag matrix with r rows (rails) and c columns (length of plaintext). Characters are placed diagonally down and then up in a zigzag until all characters are placed. The ciphertext is obtained by reading the matrix row by row.

❖ **Algorithm:**

Encryption:

1. Accept plaintext and number of rails.
2. Create a matrix with number of rows = rails and columns = length of plaintext.
3. Fill the matrix in a zigzag manner.
4. Read the matrix row-wise to get ciphertext.

Decryption:

1. Accept ciphertext and number of rails.
2. Create a zigzag pattern matrix.
3. Fill the characters row-wise.
4. Reconstruct the zigzag traversal to get the original message.

Source Code:

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

void encryptRailFence(char *text, int rails, char *result) {
    int len = strlen(text);
    char rail[rails][len];

    // Fill with null chars
    for (int i = 0; i < rails; i++)
        for (int j = 0; j < len; j++)
            rail[i][j] = '\n';

    int row = 0;
    bool down = false;

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

        if (row == 0 || row == rails - 1)
            down = !down;

        row += down ? 1 : -1;
    }

    // Read row-wise
    int idx = 0;
    for (int i = 0; i < rails; i++)
        for (int j = 0; j < len; j++)
            if (rail[i][j] != '\n')
                result[idx++] = rail[i][j];
    result[idx] = '\0';
}

void decryptRailFence(char *cipher, int rails, char *result) {
    int len = strlen(cipher);
    char rail[rails][len];

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

    int row = 0;
    bool down = false;
```

```

// Mark the positions
for (int i = 0; i < len; i++) {
    rail[row][i] = '*';

    if (row == 0 || row == rails - 1)
        down = !down;

    row += down ? 1 : -1;
}

// Fill with cipher characters
int idx = 0;
for (int i = 0; i < rails; i++)
    for (int j = 0; j < len; j++)
        if (rail[i][j] == '*')
            rail[i][j] = cipher[idx++];

// Read in zigzag
row = 0;
down = false;
idx = 0;

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

    if (row == 0 || row == rails - 1)
        down = !down;

    row += down ? 1 : -1;
}
result[idx] = '\0';
}

int main() {
    char text[100], encrypted[100], decrypted[100];
    int rails;

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

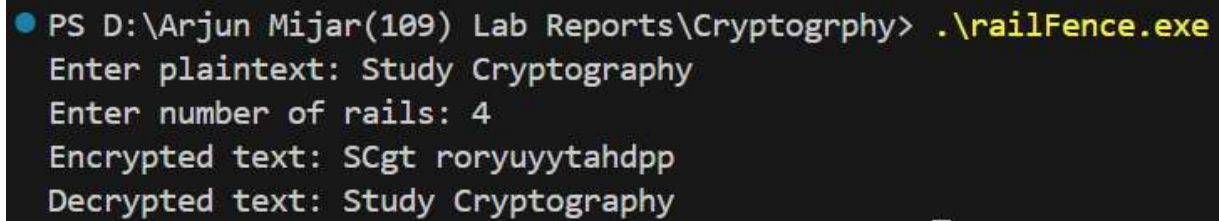
    printf("Enter number of rails: ");
    scanf("%d", &rails);

    encryptRailFence(text, rails, encrypted);
    printf("Encrypted text: %s\n", encrypted);
}

```

```
decryptRailFence(encrypted, rails, decrypted);  
printf("Decrypted text: %s\n", decrypted);  
  
return 0;  
}
```

Output:



```
PS D:\Arjun Mijar(109) Lab Reports\Cryptogrphy> .\railFence.exe  
Enter plaintext: Study Cryptography  
Enter number of rails: 4  
Encrypted text: SCgt roryuytahdpp  
Decrypted text: Study Cryptography
```

❖ Analysis:

The Rail Fence Cipher is a basic transposition cipher that provides slight obfuscation by rearranging characters in a zigzag pattern. While easy to implement and understand, it offers minimal security and is vulnerable to simple pattern analysis or brute-force attempts.

Title: WAP to implement hill cipher.

Theory:

Introduction:

The Hill Cipher is a polyalphabetic substitution cipher based on linear algebra. It uses matrix multiplication to encrypt blocks of plaintext letters. Invented by Lester S. Hill in 1929, it is one of the first ciphers to use concepts from linear algebra for encryption.

❖ **Key Concepts and Parameters:**

- Plaintext: Original message (in block format).
- Ciphertext: Encrypted message obtained after matrix multiplication.
- Key Matrix: A square matrix used for encryption. Must be invertible modulo 26.
- Block Size: Length of the key matrix (e.g., 2×2 or 3×3).
- Encryption: Uses matrix multiplication modulo 26.
- Decryption: Requires computing the inverse of the key matrix modulo 26.

❖ **Mathematical Representation:**

Let the key be a matrix K , the plaintext be a vector P , and the ciphertext be C :

Encryption: $C = (K \times P) \bmod 26$

Decryption: $P = (K^{-1} \times C) \bmod 26$

Where:

- All operations are modulo 26.
- K^{-1} is the modular inverse of matrix K .

❖ **Algorithm:**

Encryption:

1. Convert plaintext into numerical values ($A=0, B=1, \dots$).
2. Divide plaintext into blocks matching the size of the key matrix.
3. Multiply each block with the key matrix.
4. Apply modulo 26 to get ciphertext.

Decryption:

1. Compute the inverse of the key matrix modulo 26.
2. Multiply each ciphertext block with the inverse matrix.
3. Convert the resulting numbers back to characters

Source Code:

```
#include <stdio.h>
#include <string.h>
void getKeyMatrix(char key[], int keyMatrix[2][2]) {
    int k = 0;
    for (int i = 0; i < 2; i++)
        for (int j = 0; j < 2; j++)
            keyMatrix[i][j] = key[k++] % 65;
}

void encrypt(char message[], int keyMatrix[2][2], int cipherMatrix[2]) {
    for (int i = 0; i < 2; i++) {
        cipherMatrix[i] = 0;
        for (int j = 0; j < 2; j++)
            cipherMatrix[i] += keyMatrix[i][j] * (message[j] % 65);
        cipherMatrix[i] = cipherMatrix[i] % 26;
    }
}

void hillCipher(char message[], char key[]) {
    int keyMatrix[2][2];
    int cipherMatrix[2];
    char cipherText[3];

    getKeyMatrix(key, keyMatrix);
    encrypt(message, keyMatrix, cipherMatrix);

    for (int i = 0; i < 2; i++)
        cipherText[i] = cipherMatrix[i] + 65;

    cipherText[2] = '\0';

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

int main() {
    char message[3], key[5];

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

    printf("Enter 4-letter key (2x2 matrix) (in uppercase): ");
    scanf("%s", key);

    hillCipher(message, key);
}
```

```
return 0;
```

```
}
```

Output:

```
PS D:\Arjun Mijar(109) Lab Reports\Cryptogrphy> .\hillCipher.exe
Enter 2-letter plaintext (in uppercase): AR
Enter 4-letter key (2x2 matrix) (in uppercase): MIJA
Encrypted text: GA
```

❖ Analysis:

The Hill cipher is a powerful cipher for its time because it encrypts multiple letters at once. However, its security depends heavily on the invertibility of the key matrix. If the matrix is not invertible modulo 26, decryption is not possible. Also, since it's a linear cipher, it's vulnerable to known plaintext attacks.

Title: WAP to implement vernam cipher.

Theory:

Introduction:

The Vernam Cipher is a symmetric key cipher in which the plaintext is XORed with a key to produce the ciphertext. If the key is truly random and as long as the message, this becomes a One-Time Pad, which is theoretically unbreakable.

❖ **Key Concepts and Parameters:**

- Plaintext: Original message (characters A–Z).
- Key: Same length as plaintext, used for XOR operation.
- Ciphertext: Encrypted message from XORing plaintext and key.
- Encryption: $C = P \oplus K$
- Decryption: $P = C \oplus K$

❖ **Mathematical Representation:**

Let:

- P_i : i-th character of plaintext
- K_i : i-th character of key
- C_i : i-th character of ciphertext

Then:

$$C_i = (P_i \oplus K_i) \bmod 26$$

$$P_i = (C_i \oplus K_i) \bmod 26$$

Where \oplus is bitwise XOR and mod 26 maps values to A–Z.

❖ **Algorithm:**

Encryption:

1. Take the plaintext and key (same length).
2. Convert characters to numbers (A=0 to Z=25).
3. XOR each character of plaintext with corresponding key character.
4. Convert result back to characters.

Decryption:

1. XOR ciphertext with same key (reversible due to XOR).
2. Convert result back to characters.

Source Code:

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

void encryptVernam(char pt[], char key[], char ct[]) {
    for (int i = 0; pt[i] != '\0'; i++) {
        ct[i] = ((pt[i] - 'A') ^ (key[i] - 'A')) + 'A';
    }
    ct[strlen(pt)] = '\0';
}

void decryptVernam(char ct[], char key[], char pt[]) {
    for (int i = 0; ct[i] != '\0'; i++) {
        pt[i] = ((ct[i] - 'A') ^ (key[i] - 'A')) + 'A';
    }
    pt[strlen(ct)] = '\0';
}

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

    printf("Enter plaintext (A-Z only): ");
    scanf("%s", plaintext);

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

    if (strlen(plaintext) != strlen(key)) {
        printf("Error: Key and plaintext must be of same length.\n");
        return 1;
    }

    encryptVernam(plaintext, key, ciphertext);
    printf("Encrypted Text: %s\n", ciphertext);

    decryptVernam(ciphertext, key, decrypted);
    printf("Decrypted Text: %s\n", decrypted);

    return 0;
}
```


Output:

```
PS D:\Arjun Mijar(109) Lab Reports\Cryptogrphy> .\vernampCipher.exe
Enter plaintext (A-Z only): Arjun
Enter key (same length as plaintext): habit
Encrypted Text: hRI]_
Decrypted Text: Arjun
```

❖ Analysis:

The Vernam Cipher provides strong encryption when the key is truly random and used only once. However, practical use is limited by the need for secure key distribution. With a repeated or predictable key, security is compromised.

Title: WAP to implement OTP cipher.

Theory:

❖ **Introduction:**

The One-Time Pad (OTP) is an encryption technique that uses a random key that is as long as the plaintext. It is unbreakable when the key is:

- Random
- At least as long as the plaintext
- Never reused
- Kept completely secret

It is considered the only theoretically unbreakable cipher.

❖ **Key Concepts and Parameters:**

- Plaintext: Original message (A–Z).
- Key: Truly random characters (same length as plaintext).
- Ciphertext: Result of XOR between plaintext and key.
- Encryption: XOR each character with the corresponding key character.
- Decryption: XOR ciphertext with same key to get back plaintext.

❖ **Algorithm:**

Encryption:

1. Generate a random key (same length as plaintext).
2. Convert both plaintext and key characters to numbers (A=0 to Z=25).
3. XOR the two numbers, then convert back to a character.
4. Repeat for each character.

Decryption:

1. XOR ciphertext character with corresponding key character.
2. Convert result back to plaintext.

Source Code:

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

void generateRandomKey(char *key, int length) {
    for (int i = 0; i < length; i++) {
        key[i] = 'A' + rand() % 26;
    }
    key[length] = '\0';
}

void encryptOTP(char *pt, char *key, char *ct) {
    for (int i = 0; pt[i] != '\0'; i++) {
        ct[i] = ((pt[i] - 'A') ^ (key[i] - 'A')) + 'A';
    }
    ct[strlen(pt)] = '\0';
}

void decryptOTP(char *ct, char *key, char *pt) {
    for (int i = 0; ct[i] != '\0'; i++) {
        pt[i] = ((ct[i] - 'A') ^ (key[i] - 'A')) + 'A';
    }
    pt[strlen(ct)] = '\0';
}

int main() {
    char pt[100], key[100], ct[100], decrypted[100];

    printf("Enter plaintext (A–Z only, no spaces): ");
    scanf("%s", pt);

    srand(time(0));
    generateRandomKey(key, strlen(pt));

    encryptOTP(pt, key, ct);
    decryptOTP(ct, key, decrypted);

    printf("Generated Key: %s\n", key);
    printf("Encrypted Text: %s\n", ct);
    printf("Decrypted Text: %s\n", decrypted);

    return 0;
}
```

Output:

```
PS D:\Arjun Mijar(109) Lab Reports\Cryptogrphy> .\oneTimePad.exe
Enter plaintext (A-Z only, no spaces): Cryptography
Generated Key: KHXKVHRTJGFM
Encrypted Text: Iwpfgjxcjjcu
Decrypted Text: Cryptography
```

❖ Analysis:

The One-Time Pad is provably secure when the key meets all conditions. However, practical limitations like key distribution, length, and secrecy prevent it from being widely used in modern systems.

Title: Write a program to implement playfair cipher.

Theory:

❖ **Introduction:**

The Playfair Cipher is a digraph substitution cipher that encrypts two letters (a digraph) at a time. It was invented by Charles Wheatstone in 1854 but named after Lord Playfair. It uses a 5x5 matrix of letters constructed from a keyword, omitting the letter 'J'.

❖ **Key Concepts:**

- Plaintext: Original message, split into digraphs.
- Key Matrix: A 5x5 grid constructed using a keyword (no repeated letters, 'J' is usually merged with 'I').
- Rules:
 1. Same row → replace each with the letter to the right.
 2. Same column → replace each with the letter below.
 3. Else → form rectangle and swap columns.

❖ **Algorithm:**

Encryption Steps:

1. Construct 5x5 matrix from keyword.
2. Preprocess plaintext: remove spaces, convert 'J' to 'I', insert 'X' between identical letters in a pair.
3. Break plaintext into digraphs.
4. Apply Playfair rules for encryption.

Decryption Steps:

1. Reverse the encryption rules.

Source Code:

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

char keyMatrix[5][5];

void prepareMatrix(char key[]) {
```

```

int dict[26] = {0};
int i, j, k = 0;
char ch;

for (i = 0; key[i]; i++) {
    ch = toupper(key[i]);
    if (ch == 'J') ch = 'I';
    if (!dict[ch - 'A'] && ch >= 'A' && ch <= 'Z') {
        dict[ch - 'A'] = 1;
        keyMatrix[k / 5][k % 5] = ch;
        k++;
    }
}

for (ch = 'A'; ch <= 'Z'; ch++) {
    if (ch == 'J') continue;
    if (!dict[ch - 'A']) {
        keyMatrix[k / 5][k % 5] = ch;
        k++;
    }
}

void formatText(char in[], char out[]) {
    int i, j = 0;
    for (i = 0; in[i]; i++) {
        char ch = toupper(in[i]);
        if (ch < 'A' || ch > 'Z') continue;
        if (ch == 'J') ch = 'I';
        out[j++] = ch;
    }
    out[j] = '\0';

    for (i = 0; i < j; i += 2) {
        if (out[i] == out[i + 1]) {
            for (int k = j; k > i + 1; k--)
                out[k] = out[k - 1];
            out[i + 1] = 'X';
            j++;
        }
    }
    if (j % 2 != 0)
        out[j++] = 'X';
    out[j] = '\0';
}

```

```

void findPosition(char ch, int *row, int *col) {
    int i, j;
    if (ch == 'J') ch = 'I';
    for (i = 0; i < 5; i++)
        for (j = 0; j < 5; j++)
            if (keyMatrix[i][j] == ch) {
                *row = i;
                *col = j;
                return;
            }
}

void encryptPlayfair(char pt[], char ct[]) {
    int i, row1, col1, row2, col2;
    for (i = 0; pt[i]; i += 2) {
        findPosition(pt[i], &row1, &col1);
        findPosition(pt[i + 1], &row2, &col2);
        if (row1 == row2) {
            ct[i] = keyMatrix[row1][(col1 + 1) % 5];
            ct[i + 1] = keyMatrix[row2][(col2 + 1) % 5];
        } else if (col1 == col2) {
            ct[i] = keyMatrix[(row1 + 1) % 5][col1];
            ct[i + 1] = keyMatrix[(row2 + 1) % 5][col2];
        } else {
            ct[i] = keyMatrix[row1][col2];
            ct[i + 1] = keyMatrix[row2][col1];
        }
    }
    ct[i] = '\0';
}

void decryptPlayfair(char ct[], char pt[]) {
    int i, row1, col1, row2, col2;
    for (i = 0; ct[i]; i += 2) {
        findPosition(ct[i], &row1, &col1);
        findPosition(ct[i + 1], &row2, &col2);
        if (row1 == row2) {
            pt[i] = keyMatrix[row1][(col1 + 4) % 5];
            pt[i + 1] = keyMatrix[row2][(col2 + 4) % 5];
        } else if (col1 == col2) {
            pt[i] = keyMatrix[(row1 + 4) % 5][col1];
            pt[i + 1] = keyMatrix[(row2 + 4) % 5][col2];
        } else {
            pt[i] = keyMatrix[row1][col2];
            pt[i + 1] = keyMatrix[row2][col1];
        }
    }
}

```

```

    pt[i] = '\0';
}

int main() {
    char key[100], plaintext[100], formatted[100], ciphertext[100], decrypted[100];

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

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

    formatText(plaintext, formatted);
    encryptPlayfair(formatted, ciphertext);
    decryptPlayfair(ciphertext, decrypted);

    printf("Formatted Plaintext: %s\n", formatted);
    printf("Encrypted: %s\n", ciphertext);
    printf("Decrypted: %s\n", decrypted);

    return 0;
}

```

Output:

```

PS D:\Arjun Mijar(109) Lab Reports\Cryptogrphy> .\playfair.exe
Enter key: message
Enter plaintext: Cryptography
Formatted Plaintext: CRYPTOGRAPHY
Encrypted: DQVTUNSUMTFZ
Decrypted: CRYPTOGRAPHY

```

❖ Analysis:

The Playfair cipher is stronger than monoalphabetic ciphers due to digraph encryption. However, it's still vulnerable to frequency analysis and not suitable for modern cryptographic needs.

Title: RSA implementation.

Theory:

❖ **Introduction:**

RSA is a widely used asymmetric cryptographic algorithm that uses two keys: a public key for encryption and a private key for decryption. It was proposed by Rivest, Shamir, and Adleman in 1977. RSA is based on the mathematical difficulty of factoring large prime numbers.

❖ **Key Concepts:**

- Public Key (e, n): Used to encrypt the message.
- Private Key (d, n): Used to decrypt the message.
- Plaintext (P): The original message.
- Ciphertext (C): The encrypted message.

❖ **RSA Algorithm Steps:**

Key Generation:

1. Choose two large distinct prime numbers p and q.
2. Compute $n = p * q$.
3. Compute Euler's totient function: $\phi(n) = (p-1)(q-1)$.
4. Choose e such that $1 < e < \phi(n)$ and $\gcd(e, \phi(n)) = 1$.
5. Compute d such that $(d * e) \% \phi(n) = 1$.

Encryption: $C = (P^e) \% n$

Decryption: $P = (C^d) \% n$

Source Code:

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

```
int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a % b);
}
```

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

```

    long 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;
}

int main() {
    int p, q;
    printf("Enter two prime numbers (p and q): ");
    scanf("%d %d", &p, &q);

    int n = p * q;
    int phi = (p - 1) * (q - 1);

    int e;
    for (e = 2; e < phi; e++) {
        if (gcd(e, phi) == 1) break;
    }

    int d;
    for (d = 1; d < phi; d++) {
        if ((d * e) % phi == 1) break;
    }

    printf("Public Key: {%d, %d}\n", e, n);
    printf("Private Key: {%d, %d}\n", d, n);

    int message;
    printf("Enter message (integer < %d): ", n);
    scanf("%d", &message);

    long long encrypted = modExp(message, e, n);
    long long decrypted = modExp(encrypted, d, n);

    printf("Encrypted message: %lld\n", encrypted);
    printf("Decrypted message: %lld\n", decrypted);

    return 0;
}

```

Output:

```
PS D:\Arjun Mijar(109) Lab Reports\Cryptogrphy> .\rsa.exe
Enter plaintext: Arjun
Enter number of rails: 3
Encrypted text: Anruj
Decrypted text: Arjun
```

❖ Analysis:

RSA ensures secure data transmission using separate keys for encryption and decryption. Its security depends on the difficulty of factoring large numbers, making it effective but computationally expensive for large data.

Title: WAP to find primitive root of a prime number.

Theory:

❖ **Introduction:**

A primitive root of a prime number p is an integer g such that every number from 1 to $p-1$ can be expressed as a power of g modulo p . In other words, g is a generator of the multiplicative group modulo p .

❖ **Key Concepts:**

- A number g is a primitive root modulo p if:

$$g^k \bmod p \text{ for } k = 1, 2, \dots, p-1$$

produces all integers from 1 to $p-1$ in some order (i.e., distinct results).

- Primitive roots exist for all prime numbers.

❖ **Algorithm:**

1. Input a prime number p .
2. Compute $\phi = p - 1$.
3. Find all prime factors of ϕ .
4. For each number g from 2 to ϕ , check:
 - If $g^{\phi/f} \bmod p$ not equal to 1 for all prime factors f , then g is a primitive root.

Source Code:

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

int isPrime(int n) {
    if (n <= 1) return 0;
    for (int i = 2; i*i <= n; i++)
        if (n % i == 0) return 0;
    return 1;
}
```

```

int powerMod(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;
}

void findPrimeFactors(int n, int factors[], int *count) {
    *count = 0;
    if (n % 2 == 0) {
        factors[*count++] = 2;
        while (n % 2 == 0) n /= 2;
    }
    for (int i = 3; i <= sqrt(n); i += 2) {
        if (n % i == 0) {
            factors[*count++] = i;
            while (n % i == 0) n /= i;
        }
    }
    if (n > 2)
        factors[*count++] = n;
}

int isPrimitiveRoot(int g, int p, int phi, int factors[], int count) {
    for (int i = 0; i < count; i++) {
        if (powerMod(g, phi / factors[i], p) == 1)
            return 0;
    }
    return 1;
}

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

    if (!isPrime(p)) {
        printf("Number is not prime.\n");
        return 0;
    }
}

```

```

int phi = p - 1;
int factors[20], count;

findPrimeFactors(phi, factors, &count);

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

```

Output:

```

PS D:\Arjun Mijar(109) Lab Reports\Cryptogrphy> .\primitiveRoot.exe
Enter a prime number: 41
Primitive roots of 41 are: 6 7 11 12 13 15 17 19 22 24 26 28 29 30 34 35
PS D:\Arjun Mijar(109) Lab Reports\Cryptogrphy>
PS D:\Arjun Mijar(109) Lab Reports\Cryptogrphy> .\primitiveRoot.exe
Enter a prime number: 7
Primitive roots of 7 are: 3 5

```

Title: WAP to implement Diffie Hellman algorithm.

Theory:

❖ **Introduction:**

The Diffie-Hellman algorithm is a key exchange protocol that allows two parties to securely share a secret key over a public channel. The shared secret key can then be used for symmetric encryption. It is based on the difficulty of computing discrete logarithms in a finite field.

❖ **Key Concepts:**

- Public parameters:
 - p : a large prime number
 - g : a primitive root modulo p (also called base or generator)
- Private keys: Chosen secretly by each party (e.g., a and b)
- Public keys: Computed and shared using $A = g^a \bmod p$, $B = g^b \bmod p$
- Shared secret: Computed by each side as $s = B^a \bmod p = A^b \bmod p$

❖ **Algorithm:**

1. Choose a large prime number p and its primitive root g .
2. User A selects a private key a , computes $A = g^a \bmod p$, and sends A to B.
3. User B selects a private key b , computes $B = g^b \bmod p$, and sends B to A.
4. Both compute the shared key:
 - A computes $K = B^a \bmod p$
 - B computes $K = A^b \bmod p$
5. Now, both A and B share the same secret key K .

Source Code:

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

// Function to compute (base^exp) % mod
long long powerMod(long long base, long long exp, long long mod) {
    long 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;
}

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

    // Publicly known prime number and primitive root
    printf("Enter prime number (p): ");
    scanf("%lld", &p);
    printf("Enter primitive root modulo p (g): ");
    scanf("%lld", &g);

    // Private keys for A and B
    printf("Enter private key for User A: ");
    scanf("%lld", &a);
    printf("Enter private key for User B: ");
    scanf("%lld", &b);

    // Public keys
    long long A = powerMod(g, a, p);
    long long B = powerMod(g, b, p);

    // Shared secret key
    long long secretA = powerMod(B, a, p);
    long long secretB = powerMod(A, b, p);

    printf("User A's Public Key: %lld\n", A);
    printf("User B's Public Key: %lld\n", B);
    printf("User A's Shared Secret: %lld\n", secretA);
    printf("User B's Shared Secret: %lld\n", secretB);

    if (secretA == secretB)
        printf("Key exchange successful! Shared key: %lld\n", secretA);
    else
        printf("Key exchange failed.\n");

    return 0;
}

```


Output:

```
PS D:\Arjun Mijar(109) Lab Reports\Cryptogrphy> .\diffiHellman.exe
Enter prime number (p): 11
Enter primitive root modulo p (g): 8
Enter private key for User A: 5
Enter private key for User B: 4
User A's Public Key: 10
User B's Public Key: 4
User A's Shared Secret: 1
User B's Shared Secret: 1
Key exchange successful! Shared key: 1
```

Title: WAP to implement Euclidean algorithm.

Theory:

❖ **Introduction:**

The Euclidean Algorithm is a method to compute the greatest common divisor (GCD) of two integers. It is one of the oldest algorithms in mathematics, attributed to the Greek mathematician Euclid. It is based on the principle that the GCD of two numbers also divides their difference.

❖ **Key Concepts:**

- GCD (Greatest Common Divisor): The largest number that divides both integers without leaving a remainder.
- The Euclidean algorithm repeatedly replaces the larger number by its remainder when divided by the smaller number.

❖ **Algorithm:**

1. Input two integers a and b.
2. While b is not zero:
 - Set temp = b
 - Set b = a % b
 - Set a = temp
3. When b becomes 0, a contains the GCD.

Source Code:

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

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

```
int main() {  
    int a, b;  
    printf("Enter two integers: ");  
    scanf("%d %d", &a, &b);
```

```
int gcd = euclideanGCD(a, b);  
printf("GCD of %d and %d is: %d\n", a, b, gcd);  
return 0;  
}
```

Output:

```
PS D:\Arjun Mijar(109) Lab Reports\Cryptogrphy> .\euclidean.exe  
Enter two integers: 7 11  
GCD of 7 and 11 is: 1
```

Title: WAP to implement Extended Euclidean algorithm.

Theory:

❖ **Introduction:**

The Extended Euclidean Algorithm not only finds the greatest common divisor (GCD) of two integers a and b , but also finds integers x and y such that:

$$ax + by = \gcd(a, b)$$

This feature is especially useful in cryptography (e.g., for finding modular inverses in RSA).

❖ **Key Concepts:**

- **GCD:** Greatest Common Divisor.
- **Bezout's Identity:** For integers a and b , there exist integers x and y such that $ax + by = \gcd(a, b)$.
- The Extended Euclidean Algorithm calculates x , y , and $\gcd(a, b)$.

❖ **Algorithm:**

1. If $b == 0$, return ($\gcd = a$, $x = 1$, $y = 0$)
2. Otherwise:
 - Recursively compute \gcd , $x1$, $y1 = \text{extendedEuclid}(b, a \% b)$
 - Set $x = y1$
 - Set $y = x1 - (a / b) * y1$
3. Return \gcd , x , y

Source Code:

```
#include <stdio.h>

// Function to perform Extended Euclidean Algorithm
int extendedEuclid(int a, int b, int* x, int* y) {
    if (b == 0) {
        *x = 1;
        *y = 0;
        return a;
    }

    int x1, y1;
    int gcd = extendedEuclid(b, a % b, &x1, &y1);
```

```

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

    return gcd;
}

int main() {
    int a, b, x, y;
    printf("Enter two integers: ");
    scanf("%d %d", &a, &b);

    int gcd = extendedEuclid(a, b, &x, &y);
    printf("GCD of %d and %d is: %d\n", a, b, gcd);
    printf("Coefficients x and y such that ax + by = gcd:\n");
    printf("x = %d, y = %d\n", x, y);

    return 0;
}

```

Output:

```

PS D:\Arjun Mijar(109) Lab Reports\Cryptography> .\extendedEuclidean.exe
Enter two integers: 11 17
GCD of 11 and 17 is: 1
Coefficients x and y such that ax + by = gcd:
x = -3, y = 2

```