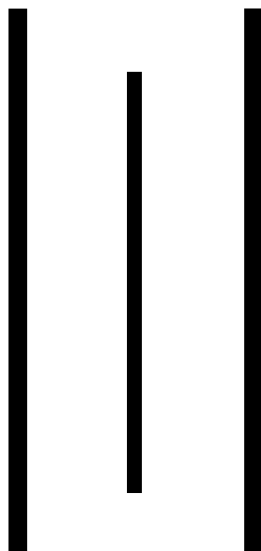




Texas International College

LAB REPORT

CRYPTOGRAPHY



BSc. CSIT 5th Semester

Submitted by:
BIPIN SAUD
Roll no : 11

Submitted to:
Sukraj Limbu
Lecturer

**TITLE: WAP TO ENCRYPT AND DECRYPT THE USER INPUT MESSAGE
AND KEY USING CAESER CIPHER.**

THEORY:

The Caesar Cipher is a simple substitution cipher where each letter in the plaintext is shifted a fixed number of positions down the alphabet, determined by a key. For encryption, each letter is shifted forward by the key value, wrapping around from 'Z' to 'A'. Decryption involves shifting each letter backward by the key value. The encryption can be expressed as $E(x)=(x+k)\bmod 26$ & decryption as $D(x)=(x-k+26)\bmod 26$. This method provides a basic but effective way to encode and decode messages.

ALGORITHM

1. Start
2. Convert each letter to its corresponding position in the alphabet
(e.g., a=0, b=1, ..., z=25).
3. Encrypt the plaintext:
 - For each letter in the plaintext, compute the ciphertext using:
$$C=E(K,P)=(P+K)\bmod 26$$
 - Where P is the position of the plaintext letter and K is the key ($0 \leq K \leq 25$).
4. Decrypt the ciphertext:
 - For each letter in the ciphertext, compute the plaintext using:
$$P=D(K,C)=(C-K+26)\bmod 26$$
 - Where C is the position of the ciphertext letter.
5. End

IDE: Vs-Code

LANGUAGE: C

SOURCE CODE

/*1. WAP to encrypt and decrypt the user input message and key using caesar cipher. */

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    char input[100], ch;
    int key, i;
    printf("Enter the input message: \t");
    gets(input);
    printf("Enter the key(0-25): \t");
    scanf("%d", &key);

    // Encryption
    for(i = 0; input[i] != '\0'; ++i){
        ch = input[i];
        if(ch >= 'a' && ch <= 'z'){
            ch = ch + key;
            if(ch > 'z'){ch = ch - 'z' + 'a' - 1;}
            input[i] = ch;
        }
        else if(ch >= 'A' && ch <= 'Z'){
            ch = ch + key;
            if(ch > 'Z'){ch = ch - 'Z' + 'A' - 1;}
            input[i] = ch;
        }
    }
    printf("Encrypted message: %s\n", input);

    // Decryption
    for(i = 0; input[i] != '\0'; ++i){
        ch = input[i];
        if(ch >= 'a' && ch <= 'z'){
            ch = ch - key;
            if(ch < 'a'){ ch = ch + 'z' - 'a' + 1; }
            input[i] = ch;
        }
        else if(ch >= 'A' && ch <= 'Z'){
            ch = ch - key;
            if(ch < 'A'){
                ch = ch + 'Z' - 'A' + 1;
            }
            input[i] = ch;
        }
    }
    printf("Decrypted message: %s\n", input);
    return 0;
}
```

OUTPUT:

```
Enter the input message:      online
Enter the key(0-25):      5
Encrypted message: tsqnsj
Decrypted message: online
bipinsaud@bipin Cryptography Labs %
```

LAB_2

Date: 2081-02-12

**TITLE: WAP TO ENCRYPT AND DECRYPT THE USER INPUT MESSAGE
AND**

KEY USING PLAYFAIR CIPHER.

THEORY:

The Playfair cipher is a digraph substitution cipher using a 5x5 matrix generated from a keyword. The matrix is filled with the keyword (omitting duplicates) and the remaining alphabet letters (combining 'I' and 'J'). For encryption, digraphs are substituted based on their positions: same row (right circular shift), same column (downward circular shift), or rectangle (opposite corners). Decryption reverses these rules. This method increases security over simple monoalphabetic ciphers.

ALGORITHM

1. Start
2. Construct the 5x5 matrix:
 - Write the keyword, omitting duplicates.
 - Fill remaining spaces with the rest of the alphabet (combine 'I' and 'J').
3. Prepare the plaintext:
 - Split plaintext into digraphs, inserting 'X' for repeated/single letters.
4. Encrypt each digraph:
 - Same row: Replace each letter with the one to its immediate right
 - Same column: Replace each letter with the one immediately below
 - Rectangle: Replace with the letters at the opposite corners of the rectangle.
5. Decrypt each digraph (reverse the encryption rules):
 - Same row: Replace each letter with the one to its immediate left.
 - Same column: Replace each letter with the one immediately above.
 - Rectangle: Replace with the letters at the opposite corners of the rectangle.
6. End

IDE: Vs-Code

LANGUAGE: C

SOURCE CODE

/*2. WAP to encrypt and decrypt the user input message and key using PlayFair cipher. */

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

#define SIZE 5

void prepareKeyTable(char key[], char keyTable[SIZE][SIZE]);
void formatMessage(char message[], char formatted[]);
void encrypt(char keyTable[SIZE][SIZE], char plaintext[],
char ciphertext[]);
void decrypt(char keyTable[SIZE][SIZE], char ciphertext[],
char plaintext[]);
void toUpper(char str[]);
void removeSpaces(char str[]);
void findPosition(char keyTable[SIZE][SIZE], char ch, int
*row, int *col);
void replaceJwithI(char str[]);
void displayKeyTable(char keyTable[SIZE][SIZE]);

int main()
{
    char key[100];
    char message[100];
    char formattedMessage[100];
    char ciphertext[100];
    char decryptedMessage[100];
    char keyTable[SIZE][SIZE];

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

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

    toUpper(key);
    removeSpaces(key);
    replaceJwithI(key);
    prepareKeyTable(key, keyTable);

    printf("Key Table:\n");
    displayKeyTable(keyTable);

    formatMessage(message, formattedMessage);
```

```

    toUpper(formattedMessage);
    replaceJwithI(formattedMessage);

    encrypt(keyTable, formattedMessage, ciphertext);
    printf("Ciphertext: %s\n", ciphertext);

    decrypt(keyTable, ciphertext, decryptedMessage);
    printf("Decrypted message: %s\n", decryptedMessage);

    return 0;
}

void prepareKeyTable(char key[], char keyTable[SIZE][SIZE])
{
    int map[26] = {0};
    int i, j, k, len = strlen(key);

    for (i = 0, k = 0; i < len; i++)
    {
        if (key[i] != 'J' && map[key[i] - 'A'] == 0)
        {
            keyTable[k / SIZE][k % SIZE] = key[i];
            map[key[i] - 'A'] = 1;
            k++;
        }
    }

    for (i = 0; i < 26; i++)
    {
        if (i != 'J' - 'A' && map[i] == 0)
        {
            keyTable[k / SIZE][k % SIZE] = 'A' + i;
            k++;
        }
    }
}

void displayKeyTable(char keyTable[SIZE][SIZE])
{
    for (int i = 0; i < SIZE; i++)
    {
        for (int j = 0; j < SIZE; j++)
        {
            printf("%c ", keyTable[i][j]);
        }
        printf("\n");
    }
}

Void formatMessage(char message[], char formatted[])

```

```

{
    int i, j = 0;
    for (i = 0; message[i] != '\0'; i++)
    {
        if (isalpha(message[i]))
        {
            formatted[j++] = toupper(message[i]);
            if (j > 1 && formatted[j - 1] == formatted[j -
2])
            {
                formatted[j - 1] = 'X';
                formatted[j++] = toupper(message[i]);
            }
        }
        if (j % 2 != 0)
        {
            formatted[j++] = 'X';
        }
        formatted[j] = '\0';
    }
}

```

```

void toUpper(char str[])
{
    int i;
    for (i = 0; str[i] != '\0'; i++)
    {
        str[i] = toupper(str[i]);
    }
}

```

```

void removeSpaces(char str[])
{
    int i, j;
    for (i = 0, j = 0; str[i] != '\0'; i++)
    {
        if (str[i] != ' ')
        {
            str[j++] = str[i];
        }
    }
    str[j] = '\0';
}

```

```

void findPosition(char keyTable[SIZE][SIZE], char ch, int
*row, int *col)
{
    int i, j;
    for (i = 0; i < SIZE; i++)
    {
        for (j = 0; j < SIZE; j++)

```



```

        {
            if (keyTable[i][j] == ch)
            {
                *row = i;
                *col = j;
                return;
            }
        }
    }
}

void replaceJwithI(char str[])
{
    int i;
    for (i = 0; str[i] != '\0'; i++)
    {
        if (str[i] == 'J')
        {
            str[i] = 'I';
        }
    }
}

void encrypt(char keyTable[SIZE][SIZE], char plaintext[],
char ciphertext[])
{
    int i, a, b, c, d;
    for (i = 0; plaintext[i] != '\0'; i += 2)
    {
        findPosition(keyTable, plaintext[i], &a, &b);
        findPosition(keyTable, plaintext[i + 1], &c, &d);

        if (a == c)
        {
            ciphertext[i] = keyTable[a][(b + 1) % SIZE];
            ciphertext[i + 1] = keyTable[c][(d + 1) % SIZE];
        }
        else if (b == d)
        {
            ciphertext[i] = keyTable[(a + 1) % SIZE][b];
            ciphertext[i + 1] = keyTable[(c + 1) % SIZE][d];
        }
        else
        {
            ciphertext[i] = keyTable[a][d];
            ciphertext[i + 1] = keyTable[c][b];
        }
    }
    ciphertext[i] = '\0';
}

```

```

void decrypt(char keyTable[SIZE][SIZE], char ciphertext[],
char plaintext[])
{
    int i, a, b, c, d;
    for (i = 0; ciphertext[i] != '\0'; i += 2)
    {
        findPosition(keyTable, ciphertext[i], &a, &b);
        findPosition(keyTable, ciphertext[i + 1], &c, &d);

        if (a == c)
        {
            plaintext[i] = keyTable[a][(b + SIZE - 1) %
SIZE];
            plaintext[i + 1] = keyTable[c][(d + SIZE - 1) %
SIZE];
        }
        else if (b == d)
        {
            plaintext[i] = keyTable[(a + SIZE - 1) %
SIZE][b];
            plaintext[i + 1] = keyTable[(c + SIZE - 1) %
SIZE][d];
        }
        else
        {
            plaintext[i] = keyTable[a][d];
            plaintext[i + 1] = keyTable[c][b];
        }
    }
    plaintext[i] = '\0';
}

```

OUTPUT:

```

Enter key: galois
Enter message: online
Key Table:
G A L O I
S B C D E
F H K M N
P Q R T U
V W X Y Z
Ciphertext: IMOGUN
Decrypted message: ONLINE
○ bipinsaud@bipin Cryptography Labs %

```

LAB_3

Date:2081-02-29

TITLE: WAP TO ENCRYPT AND DECRYPT THE MESSAGE USING VERNAM CIPHER

THEORY:

The Vernam Cipher is a symmetric key cipher where each character in the plaintext is XORed with a corresponding character from a key of the same length. For encryption, this XOR operation produces the ciphertext, and for decryption, it retrieves the original plaintext. If the key is truly random, secret, and used only once, the Vernam Cipher is theoretically unbreakable, making it impossible for an attacker to determine the original message without the key. The operations can be expressed as follows:

- **Encryption:** $C_i = P_i \oplus K_i$
- **Decryption:** $P_i = C_i \oplus K_i$

where P_i is the i th character of the plaintext, K_i is the i th character of the key, and C_i is the i th character of the ciphertext.

ALGORITHM

1. Start
2. Generate a random key:
 - Create a random key of the same length as the plaintext message.
3. Encrypt the plaintext:
 - For each character in the plaintext:
 - Compute the ciphertext using: $C_i = P_i \oplus K_i$
 - Where P_i is the i th character of the plaintext, and K_i is the i th character of the key.
4. Decrypt the ciphertext:
 - For each character in the ciphertext:
 - Compute the plaintext using: $P_i = C_i \oplus K_i$
 - Where C_i is the i th character of the ciphertext.
5. End

IDE: Vs-Code

LANGUAGE: C

SOURCE CODE

```

/*3.WAP to encrypt and decrypt the message using Vernam
Cipher*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

// Function to generate a key matching the length of the
plaintext
char *generate_key(int plaintext_length)
{
    char *key = (char *)malloc((plaintext_length + 1) *
sizeof(char));

    // Seed the random number generator
    srand(time(NULL));

    for (int i = 0; i < plaintext_length; i++)
    {
        key[i] = 'A' + rand() % 26; // Generate a random
uppercase letter
    }
    key[plaintext_length] = '\\0';

    return key;
}

// Function to perform XOR encryption
char *encrypt(const char *plaintext, const char *key)
{
    int plaintext_length = strlen(plaintext);
    char *ciphertext = (char *)malloc((plaintext_length + 1)
* sizeof(char));

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

```

```

        ciphertext[i] = plaintext[i] ^ key[i]; // XOR
operation
    }
    ciphertext[plaintext_length] = '\0';

    return ciphertext;
}

// Function to perform XOR decryption
char *decrypt(const char *ciphertext, const char *key)
{
    int ciphertext_length = strlen(ciphertext);
    char *decrypted_text = (char *)malloc((ciphertext_length
+ 1) * sizeof(char));

    for (int i = 0; i < ciphertext_length; i++)
    {
        decrypted_text[i] = ciphertext[i] ^ key[i]; // XOR
operation
    }
    decrypted_text[ciphertext_length] = '\0';

    return decrypted_text;
}

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

    // Get the plaintext from the user
    printf("Please enter your message: ");
    fgets(plaintext, sizeof(plaintext), stdin);
    plaintext[strcspn(plaintext, "\n")] = '\0'; // Remove
newline character

    int plaintext_length = strlen(plaintext);

```

```

        // Get the key from the user
        printf("Please provide the key (must match message
length): ");
        fgets(key, sizeof(key), stdin);
        key[strcspn(key, "\n")] = '\0'; // Remove newline
character

        // Encrypt the message
        char *ciphertext = encrypt(plaintext, key);
        printf("Encrypted Message (Hexadecimal): ");
        for (int i = 0; i < plaintext_length; i++)
        {
            printf("%02X", (unsigned char)ciphertext[i]);
        }
        printf("\n");

        // Decrypt the message
        char *decrypted_text = decrypt(ciphertext, key);
        printf("Decrypted Message: %s\n", decrypted_text);

        // Free dynamically allocated memory
        free(ciphertext);
        free(decrypted_text);

        return 0;
}

```

OUTPUT:

```

Please enter your message: letsdoit
Please provide the key (must match message length): tiodstel
Encrypted Message (Hexadecimal): 180C1B17171B0C18
Decrypted Message: letsdoit
bipinsaud@bipin Cryptography Labs %

```

LAB_4

Date:2081-02-29

**TITLE: WAP TO ENCRYPT AND DECRYPT THE MESSAGE USING
RAILFENCE CIPHER**

THEORY:

The Rail Fence cipher, also known as the Rail Fence or Zigzag cipher, is a transposition cipher where the plaintext message is written diagonally across a number of "rails" or lines. The encrypted message is then formed by reading the characters row by row. Decryption involves reconstructing the original message by reversing the process. It provides a simple form of encryption, but it is vulnerable to cryptanalysis due to its predictable pattern and can be easily broken with brute force methods for small key sizes. Despite its weaknesses, the Rail Fence cipher has historical significance and serves as an introduction to the concept of transposition ciphers in cryptography education. Its simplicity makes it suitable for basic encryption tasks and classroom exercises, though it is not suitable for secure communications without additional cryptographic techniques.

ALGORITHM

1. Start
2. Encryption
 - Arrange characters of the plaintext diagonally across a predefined number of "rails".
 - Read the characters row by row to form the ciphertext.
3. Decryption
 - Reconstruct the rails with placeholders for the characters.
 - Write the ciphertext characters diagonally across the rails.
 - Read the characters row by row to retrieve the plaintext.
4. End

IDE: Vs-Code

LANGUAGE: C

SOURCE CODE

```
/*4.WAP to encrypt and decrypt the message using Railfence Cipher*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
// Function to encrypt the message using Rail Fence cipher
char *encryptRailFence(char *message, int key){
    int len = strlen(message);
    char *encrypted = (char *)malloc((len + 1) *
sizeof(char));
    int rail = 0, dir = 0;
// Initialize the rail fence matrix
    char railMatrix[key][len];
    for (int i = 0; i < key; i++){
        for (int j = 0; j < len; j++){
            railMatrix[i][j] = '\0';
        }
    }
// Fill the rail fence matrix with the message
    for (int i = 0; i < len; i++){
        railMatrix[rail][i] = message[i];
        if (rail == 0){
            dir = 0;
        }
        else if (rail == key - 1){
            dir = 1;
        }
        if (dir == 0){
            rail++;
        }
        else{
            rail--;
        }
    }
// Read the encrypted message from the rail fence matrix
    int k = 0;
    for (int i = 0; i < key; i++){
        for (int j = 0; j < len; j++){
            if (railMatrix[i][j] != '\0'){
                encrypted[k++] = railMatrix[i][j];
            }
        }
    }
    encrypted[k] = '\0';
    return encrypted;
}
// Function to decrypt the message using Rail Fence cipher
char *decryptRailFence(char *encrypted, int key){
    int len = strlen(encrypted);
```



```

    char *decrypted = (char *)malloc((len + 1) *
sizeof(char));
    int rail = 0, dir = 0;
    // Initialize the rail fence matrix
    char railMatrix[key][len];
    for (int i = 0; i < key; i++){
        for (int j = 0; j < len; j++){
            railMatrix[i][j] = '\\0';
        }
    }
    // Fill the rail fence matrix with markers indicating
the rail positions
    for (int i = 0; i < len; i++){
        railMatrix[rail][i] = '*';
        if (rail == 0){
            dir = 0;
        }
        else if (rail == key - 1){
            dir = 1;
        }

        if (dir == 0){
            rail++;
        }
        else{
            rail--;
        }
    }
    // Replace markers with characters from the encrypted
message
    int index = 0;
    for (int i = 0; i < key; i++){
        for (int j = 0; j < len; j++){
            if (railMatrix[i][j] == '*' && index < len){
                railMatrix[i][j] = encrypted[index++];
            }
        }
    }
    // Read the decrypted message from the rail fence matrix
    int k = 0;
    rail = 0, dir = 0;
    for (int i = 0; i < len; i++){
        decrypted[k++] = railMatrix[rail][i];
        if (rail == 0){
            dir = 0;
        }
        else if (rail == key - 1){
            dir = 1;
        }
        if (dir == 0){
            rail++;
        }
    }

```

```

        else{
            rail--;
        }
    }
    decrypted[k] = '\0';
    return decrypted;
}

int main(){
    char message[100], *encrypted, *decrypted;
    int key;
    printf("Enter the message to encrypt: ");
    fgets(message, sizeof(message), stdin);
    message[strcspn(message, "\n")] = '\0'; // Remove
newline
    printf("Enter the key: ");
    scanf("%d", &key);
    encrypted = encryptRailFence(message, key);
    printf("Encrypted message: %s\n", encrypted);

    decrypted = decryptRailFence(encrypted, key);
    printf("Decrypted message: %s\n", decrypted);
    free(encrypted);
    free(decrypted);
    return 0;
}

```

OUTPUT:

```

Enter the message to encrypt: hello
Enter the key: 3
Encrypted message: hoell
Decrypted message: hello
bipinsaud@bipin Cryptography Labs %

```

TITLE: WAP TO TEST IF A NUMBER IS PRIMITIVE ROOT OR NOT.

THEORY:

A primitive root modulo p is a number g that can generate all numbers from 1 to $p-1$ when raised to different powers and taken modulo p . For a prime p , g is a primitive root if the sequence $g^1 \bmod p, g^2 \bmod p, \dots, g^{p-1} \bmod p$ produces all the numbers from 1 to $p-1$ without repetition. Primitive roots are important in cryptography for creating secure keys. To check if g is a primitive root of p , we compute its powers modulo p and see if they cover all numbers from 1 to $p-1$. This ensures the number is a primitive root and useful for secure communication in cryptography.

ALGORITHM

1. Start
2. Input: Prime number p and number g .
3. Verify Prime: Check if p is a prime number.
 - i. If p is not prime, exit.
4. Calculate Totient: Compute $\phi(p) = p-1$.
5. Primitive Root Check:
 - i. For each integer i from 2 to $\phi(p)$:
 1. If $\phi(p)$ is divisible by i :
 - a. Compute $g^i \bmod p$.
 - b. If result is 1, g is not a primitive root modulo p . Exit.
6. Determine Result:
 - i. If g passes all checks (no i satisfies $g^i \equiv 1 \bmod p$ for $i < \phi(p)$), then g is a primitive root modulo p .
7. End

IDE: Vs-Code

LANGUAGE: C

SOURCE CODE

```
/*5.WAP to test if a number is primitive root or not.*/
#include <stdio.h>
#include <math.h>

int is_primitive_root(int g, int p, int values[]){
    for (int i = 0; i < p - 1; i++){
        values[i] = (int)pow(g, i) % p;
    }
    // Check if all numbers from 1 to (p-1) can be found
    for (int i = 1; i < p; i++)
    {
        int found = 0;
        for (int j = 0; j < p - 1; j++){
            if (values[j] == i){
                found = 1;
                break; // Exit inner loop if found
            }
        }
        if (!found){
            return 0;
        }
    }
    return 1;
}

int main(){
    int p, g;
    int values[100];
    printf("Enter the prime number p:");
    scanf("%d", &p);
    printf("Enter the value of g:");
    scanf("%d", &g);
    int g_mod_p = g % p;
    if (is_primitive_root(g, p, values)){
        printf("\n%d is a primitive root of %d.\n", g, p);
        printf("Values of g^i mod p are:\n");
        for (int i = 0; i < p - 1; i++){
```

```

        printf("%d^%d mod %d = %d\n", g, i, p, values[i]);
    }
    printf("Distinct values \n");
}
else{
    printf("%d is not a primitive root of %d.\n", g, p);
}
return 0;
}

```

OUTPUT:

```

Enter the prime number p:7
Enter the value of g:3

```

3 is a primitive root of 7.

Values of $g^i \bmod p$ are:

$3^0 \bmod 7 = 1$

$3^1 \bmod 7 = 3$

$3^2 \bmod 7 = 2$

$3^3 \bmod 7 = 6$

$3^4 \bmod 7 = 4$

$3^5 \bmod 7 = 5$

Distinct values

bipinsaud@bipin Cryptography Labs %

```

Enter the prime number p:7
Enter the value of g:2

```

2 is not a primitive root of 7.

bipinsaud@bipin Cryptography Labs %

**TITLE: WAP TO FIND THE GCD OF TWO INPUT INTEGERS USING
EUCLIDEAN ALGORITHM**

THEORY:

The Euclidean Algorithm is a method for finding the greatest common divisor (GCD) of two integers, which is the largest integer that divides both without leaving a remainder. It is based on the principle that the GCD of two numbers also divides their difference. The algorithm works by repeatedly replacing the larger number by its remainder when divided by the smaller number until one of the numbers becomes zero. The non-zero remainder at this point is the GCD. An important fact about this algorithm is its efficiency; it operates in logarithmic time relative to the size of the smaller number, making it very fast even for large integers. The Euclidean Algorithm also forms the basis for more advanced algorithms in number theory and computer science, such as the Extended Euclidean Algorithm, which finds integer coefficients that express the GCD as a linear combination of the original integers. Additionally, it is foundational in modern cryptographic systems, including RSA encryption.

ALGORITHM

1. Start
2. Input two integers, a and b.
3. If b is greater than a, swap their values.
4. Apply the Euclidean Algorithm recursively:
 - If b is zero, return a.
 - Otherwise, print the step $(a = b * (a // b) + (a \% b))$ and recursively call the function with arguments (b, a % b).
5. Print the steps to find the GCD and the result.
6. End

IDE: Vs-Code

LANGUAGE: C

SOURCE CODE

/*6. WAP to find the GCD the gcd of two input integer using Euclidean Algorithm.*/

```
#include <stdio.h>
// Recursive function to find GCD & print steps
int euclidean_algorithm(int a, int b){
    if (b == 0){
        return a;
    }
    else{
        printf("%d = %d * %d + %d\n", a, b, a / b, a % b);
        return euclidean_algorithm(b, a % b);
    }
}

int main(){
    int a, b;

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

    // Swap values if b is greater than a
    if (b > a){
        int temp = a;
        a = b;
        b = temp;
    }

    printf("Steps to find GCD of %d and %d:\n", a, b);
    int gcd = euclidean_algorithm(a, b);
    printf("GCD of %d and %d is %d\n", a, b, gcd);

    if (gcd == 1){
        printf("Numbers %d %d are relatively prime\n", a, b);
    }
}
```

```
else{
    printf("Numbers %d %d are not relatively
    prime\n", a, b);
}
return 0;
}
```

OUTPUT:

```
Enter two numbers a and b: 23 7
Steps to find GCD of 23 and 7:
23 = 7 * 3 + 2
7 = 2 * 3 + 1
2 = 1 * 2 + 0
GCD of 23 and 7 is 1
Numbers 23 7 are relatively prime
bipinsaud@bipin Cryptography Labs %
```

```
Enter two numbers a and b: 21 7
Steps to find GCD of 21 and 7:
21 = 7 * 3 + 0
GCD of 21 and 7 is 7
Numbers 21 7 are not relatively prime
bipinsaud@bipin Cryptography Labs %
```


TITLE: WAP TO FIND MODULAR INVERSE OF INPUT INTEGER IN MOD N

THEORY:

The modular inverse of integer a under modulo n is an integer x such that $a \times x \equiv 1 \pmod{n}$. This means x is the number that, when multiplied by a , yields a remainder of 1 when divided by n . The Extended Euclidean Algorithm is used to find this modular inverse. It not only computes the greatest common divisor (GCD) of two integers a and b but also finds integers x and y satisfying $ax + by = \text{GCD}(a, b)$. For the modular inverse to exist, a and n must be coprime, i.e., their GCD must be 1. The algorithm involves recursive calls to break down the problem and back-substitution to find the coefficients, ensuring the solution x is correctly determined even if initially negative by taking $(x \% n + n) \% n$.

ALGORITHM

1. Start
2. Input: Read integers a and n .
3. Check Coprimality:
 - Use the Extended Euclidean Algorithm to find the GCD of a and n .
 - If $\text{GCD} \neq 1$, print "Modular inverse does not exist" and exit.
4. Extended Euclidean Algorithm:
 - Base Case: If $a=0$, return n as GCD and set $x=0$, $y=1$.
 - Recursively call the function with $b\%a$ and a .
 - Update x and y based on recursive results.
5. Compute Modular Inverse:
 - If GCD is 1, compute the modular inverse as $(x \% n + n) \% n$ to ensure it's positive.
6. Output: Print the modular inverse.
7. End

IDE: Vs-Code

LANGUAGE: C

SOURCE CODE

```
/*7. WAP to find modular inverse of input integer in mod
n.*/

#include <stdio.h>
int gcdExtended(int a, int b, int *x, int *y)
{
    // Base Case
    if (a == 0)
    {
        *x = 0;
        *y = 1;
        return b;
    }

    int x1, y1; // To store results of recursive call
    int gcd = gcdExtended(b % a, a, &x1, &y1);

    // Update x and y using results of recursive call
    *x = y1 - (b / a) * x1;
    *y = x1;

    return gcd;
}

// Function to find the modular inverse of a under modulo n
int modInverse(int a, int n)
{
    int x, y;
    int gcd = gcdExtended(a, n, &x, &y);

    // If a and n are not coprime, the modular inverse does
    not exist
    if (gcd != 1){
        printf("Modular inverse does not exist.\n");
        return -1;}
    else{
        // x might be negative, so take it positive
        int result = (x % n + n) % n;
        return result;
    }
}

int main(){
    int a, n;
    printf("Enter an integer a: ");
    scanf("%d", &a);
    printf("Enter modulus n: ");
```

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

int inverse = modInverse(a, n);
if (inverse != -1){
    printf("The modular inverse of %d under modulo
    %d is %d\n", a, n, inverse);
}
return 0;
}
```

OUTPUT:

```
Enter an integer a: 3
Enter modulus n: 11
The modular inverse of 3 under modulo 11 is 4
bipinsaud@bipin Cryptography Labs %
```

```
Enter an integer a: 6
Enter modulus n: 9
Modular inverse does not exist.
bipinsaud@bipin Cryptography Labs %
```

LAB_8

Date: 2081-02-29

TITLE: WAP TO CALCULATE THE EULER TOTIENT FUNCTION OF INPUT INTEGER

THEORY:

The Euler Totient Function, $\phi(n)$, counts the integers up to n that are coprime with n , meaning their greatest common divisor with n is 1. For prime n , $\phi(n) = n-1$, as all numbers less than n are coprime with it. For any integer n , the function can be determined by iterating through all numbers from 1 to $n-1$ and checking if each number is coprime with n . This is done using the gcd (greatest common divisor) function. The logic is to increment a count for each number that satisfies $\text{gcd}(i, n) = 1$. The overall result gives the count of integers less than n that do not share any common factors with n , effectively providing $\phi(n)$.

ALGORITHM

1. Input: Read an integer n .
2. Initialization: Set a variable result to 1.
3. Iteration:
 - Loop through numbers i from 2 to $n-1$.
 - Check if $\text{gcd}(i, n)$ equals 1.
 - If true, increment result by 1.
4. Output: result is the Euler Totient Function $\phi(n)$, the count of integers less than n that are coprime with n .
5. End

IDE: Vs-Code

LANGUAGE: C

SOURCE CODE

```
/*8.WAP to calculate the Euler totient function of input
integer.*/
#include <stdio.h>
// Function to return gcd of a and b
int gcd(int a, int b)
{
    if (a == 0)
        return b;
    return gcd(b % a, a);
}

// A simple method to evaluate Euler Totient Function
int phi(unsigned int n)
{
    unsigned int result = 1; // Start with result = 1
    because gcd(n, n) is 1
    for (int i = 2; i < n; i++)
        if (gcd(i, n) == 1)
            result++;
    return result;
}

// Function to print the numbers relatively prime to n
void printRelativelyPrimeNumbers(int n)
{
    printf("Numbers relatively prime to %d: ", n);
    for (int i = 1; i < n; i++)
    {
        if (gcd(i, n) == 1)
            printf("%d ", i);
    }
    printf("\n");
}

// Driver program to test the above functions
int main()
```

```
{  
    int n;  
    printf("Enter an integer n: ");  
    scanf("%d", &n);  
  
    printf("φ(%d) = %d\n", n, phi(n));  
    printRelativelyPrimeNumbers(n);  
  
    return 0;  
}
```

OUTPUT:

```
Enter an integer n: 12  
φ(12) = 4  
Numbers relatively prime to 12: 1 5 7 11  
bipinsaud@bipin Cryptography Labs %
```

LAB_9

Date: 2081-02-29

**TITLE: WAP TO TEST IF AN INPUT IS PRIME OR NOT USING
FERMAT'S LITTLE THEOREM**

THEORY:

Fermat's Little Theorem states that if n is a prime number and a is any integer such that $1 \leq a < n$, then $a^{(n-1)} \equiv 1 \pmod{n}$. Using this theorem, a probabilistic test can determine if a number n is prime. The algorithm selects a random base a and checks if $a^{(n-1)} \bmod n \equiv 1$. This process is repeated for k iterations to reduce the probability of a false positive. If any iteration fails the condition, n is declared composite. If all iterations pass, n is likely prime. This method is efficient for large numbers but may produce false positives for certain composite numbers that behave like primes in this test.

ALGORITHM

1. Input: Read an integer n to test for primality.
2. Handle Special Cases:
 - If $n \leq 1$ or $n = 4$, return composite (not prime).
 - If $n \leq 3$, return prime.
3. Repeat k Times:
 - Select a random integer a in the range $[2, n-2]$.
 - Compute $a^{(n-1)} \bmod n$ using the power function.
 - If $a^{(n-1)} \bmod n \neq 1$, return composite (not prime).
4. Output: If all iterations pass, return prime.
5. End

IDE: Vs-Code

LANGUAGE: C

SOURCE CODE

/*9.WAP to test if an input is prime or not using little Fermat's Theorem.*/

```
#include <stdio.h>
#include <stdlib.h>
// Function to compute x^y under modulo m
int power(int x, unsigned int y, unsigned int m)
{
    int result = 1; // Initialize result
    x = x % m;      // Update x if it is more than or
    equal to m

    if (x == 0)
        return 0; // In case x is divisible by m;

    while (y > 0)
    {
        // If y is odd, multiply x with result
        if (y & 1)
            result = (result * x) % m;

        // y must be even now
        y = y >> 1; // y = y/2
        x = (x * x) % m; // Change x to x^2
    }
    return result;
}

// Function to test if n is prime using Fermat's little
theorem
int isPrime(unsigned int n, int k)
{
    // Corner cases
    if (n <= 1 || n == 4)
        return 0;
```



```

    if (n <= 3)
        return 1;
    // Try k times
    while (k > 0)
    {
        // Pick a random number in [2..n-2]
        // Above corner cases make sure that n > 4
        int a = 2 + rand() % (n - 4);
        // Fermat's little theorem
        if (power(a, n - 1, n) != 1)
            return 0;
        k--;
    }
    return 1;
}

int main(){
    int n;        // Number to test for primality
    int k = 5;    // Number of iterations
    printf("Enter a number to check if it is prime: ");
    scanf("%d", &n);
    if (isPrime(n, k))
        printf("%d is prime\n", n);
    else
        printf("%d is not prime\n", n);
    return 0;
}

```

OUTPUT:

```

Enter a number to check if it is prime: 21
21 is not prime
bipinsaud@bipin Cryptography Labs % cd "/Us

```

```

Enter a number to check if it is prime: 23
23 is prime
bipinsaud@bipin Cryptography Labs % █

```

**TITLE: WAP TO SIMULATE DIFFIE-HELLMAN KEY EXCHANGE
ALGORITHM**

THEORY:

The Diffie-Hellman key exchange algorithm is a cryptographic protocol that allows two parties to securely exchange keys over a public channel. This enables them to establish a shared secret key, which can then be used for encrypted communication. The security of the Diffie-Hellman algorithm relies on the difficulty of computing discrete logarithms in a finite field, ensuring the shared key remains secure even if an attacker can observe the key exchange. Each party selects a private key and generates a corresponding public key. These public keys are exchanged between the parties. Using their own private key and the other party's public key, each party computes the same shared secret key. The method's security is based on the computational difficulty of deriving the private key from the public key, making it resistant to eavesdropping attacks.

ALGORITHM

1. Input: Read prime number P , primitive root G , and private keys a and b .
2. Calculate Public Keys:
 - Compute Alice's public key $x = G^a \bmod P$.
 - Compute Bob's public key $y = G^b \bmod P$.
3. Exchange Public Keys:
 - Alice and Bob exchange their public keys x and y .
4. Compute Secret Keys:
 - Alice computes the secret key $k_a = y^a \bmod P$.
 - Bob computes the secret key $k_b = x^b \bmod P$.
5. Output: Print the secret keys k_a and k_b .
6. Verification: Verify if k_a equals k_b , confirming identical keys.
7. End

IDE: Vs-Code

LANGUAGE: C

SOURCE CODE

```
/*10.WAP to simulate Diffie-Hellman Key Exchange
Algorithm.*/

#include <stdio.h>

//Function to perform modular exponent returns (a^b) % P
long long int power(long long int a, long long int b, long
long int P){
    long long int result = 1;
    a = a % P; // Update 'a' if is more than or equal to P

    while (b > 0){
        // If b is odd, multiply 'a' with the result
        if (b % 2 == 1)
            result = (result * a) % P;
        // b must be even now
        b = b >> 1;      // b = b/2
        a = (a * a) % P; // Change 'a' to 'a^2'
    }
    return result;
}

// Driver program
int main()
{
    long long int P, G, x, a, y, b, ka, kb;

    // Prompt the user to enter the value of P
    printf("Enter the value of P (a prime number): ");
    scanf("%lld", &P);

    // Prompt the user to enter the value of G
    printf("Enter the value of G (a primitive root of P):
");
    scanf("%lld", &G);

    // Alice will choose the private key a
```

```

printf("Enter the private key a for Alice: ");
scanf("%lld", &a);
x = power(G, a, P); // gets the generated key

// Bob will choose the private key b
printf("Enter the private key b for Bob: ");
scanf("%lld", &b);
y = power(G, b, P); // gets the generated key

// Generating the secret key after the exchange of keys
ka = power(y, a, P); // Secret key for Alice
kb = power(x, b, P); // Secret key for Bob

printf("Secret key for Alice is: %lld\n", ka);
printf("Secret key for Bob is: %lld\n", kb);

if (ka == kb)
{
    printf("Identical keys are exchanged.");
}

return 0;
}

```

OUTPUT:

```

Enter the value of P (a prime number): 23
Enter the value of G (a primitive root of P): 5
Enter the private key a for Alice: 3
Enter the private key b for Bob: 2
Secret key for Alice is: 8
Secret key for Bob is: 8
Identical keys are exchanged.%
bipinsaud@bipin Cryptography Labs %

```

LAB_11

Date: 2081-02-29

TITLE: WAP TO GENERATE SUBKEYS FOR USER INPUT KEY(BITS) IN DES

THEORY:

The Data Encryption Standard (DES) uses a 64-bit key to generate 16 subkeys, each 48 bits long, for the encryption process. The initial key undergoes a permutation using the PC-1 table, reducing it from 64 to 56 bits by excluding every 8th bit. This permuted key is split into two 28-bit halves. For each of the 16 rounds, these halves are subjected to a left circular shift based on predefined shift values. After shifting, the halves are combined and permuted again using the PC-2 table to produce a 48-bit subkey. This process ensures that each round uses a unique subkey derived from the original 64-bit key. The subkeys are essential for the DES encryption's round transformations.

ALGORITHM

1. Input: Read a 64-bit key in hexadecimal.
2. Permutation Choice 1 (PC-1):
 - Permute the key using the PC-1 table to obtain a 56-bit key.
3. Split the Key:
 - Divide the 56-bit key into two 28-bit halves: CCC and DDD.
4. Generate 16 Subkeys:
 - For each of the 16 rounds:
 - Perform left circular shifts on both CCC and DDD according to a predefined schedule.
 - Combine the shifted halves.
 - Permute the combined halves using the PC-2 table to obtain a 48-bit subkey.
 - Print the subkey.
5. Output: Print the 16 generated subkeys.
6. End

IDE: Vs-Code

LANGUAGE: C

SOURCE CODE

```
/*11.WAP to generate subkeys for user innput key(bits) in
DES.*/

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

// Permutation choice 1 (PC-1) table
int PC1[56] = {
    57, 49, 41, 33, 25, 17, 9,
    1, 58, 50, 42, 34, 26, 18,
    10, 2, 59, 51, 43, 35, 27,
    19, 11, 3, 60, 52, 44, 36,
    63, 55, 47, 39, 31, 23, 15,
    7, 62, 54, 46, 38, 30, 22,
    14, 6, 61, 53, 45, 37, 29,
    21, 13, 5, 28, 20, 12, 4};

// Permutation choice 2 (PC-2) table
int PC2[48] = {
    14, 17, 11, 24, 1, 5,
    3, 28, 15, 6, 21, 10,
    23, 19, 12, 4, 26, 8,
    16, 7, 27, 20, 13, 2,
    41, 52, 31, 37, 47, 55,
    30, 40, 51, 45, 33, 48,
    44, 49, 39, 56, 34, 53,
    46, 42, 50, 36, 29, 32};

// Left shifts for each round
int left_shifts[16] = {
    1, 1, 2, 2, 2, 2, 2, 2,
    1, 2, 2, 2, 2, 2, 2, 1};

// Function to perform left circular shift
unsigned long leftCircularShift(unsigned long n, int shifts,
int totalBits)
```

```

{
    return ((n << shifts) & ((1 << totalBits) - 1)) | (n >>
(totalBits - shifts));
}

```

// Function to generate subkeys

```
void generateSubkeys(unsigned long key)
```

```

{
    unsigned long permutedKey = 0;
    int i;

    // Perform PC-1 permutation
    for (i = 0; i < 56; i++)
    {
        permutedKey <<= 1;
        permutedKey |= (key >> (64 - PC1[i])) & 1;
    }

    // Split the key into two halves
    unsigned long C = (permutedKey >> 28) & 0xFFFFFFFF;
    unsigned long D = permutedKey & 0xFFFFFFFF;

    // Generate the 16 subkeys
    for (i = 0; i < 16; i++)
    {
        // Perform the left circular shift
        C = leftCircularShift(C, left_shifts[i], 28);
        D = leftCircularShift(D, left_shifts[i], 28);

        // Combine the halves
        unsigned long combined = (C << 28) | D;

        // Apply PC-2 permutation
        unsigned long subkey = 0;
        for (int j = 0; j < 48; j++)
        {
            subkey <<= 1;

```

```

        subkey |= (combined >> (56 - PC2[j])) & 1;
    }

    // Print the subkey
    printf("Subkey %2d: ", i + 1);
    for (int k = 47; k >= 0; k--)
    {
        printf("%lu", (subkey >> k) & 1);
    }
    printf("\n");
}
}

int main()
{
    unsigned long key;
    printf("Enter a 64-bit key in hexadecimal (e.g.,
0x133457799BBCDFF1): ");
    scanf("%lx", &key);

    generateSubkeys(key);

    return 0;
}

```

OUTPUT:

```

Enter a 64-bit key in hexadecimal (e.g., 0x133457799BBCDFF1): 0x0F1571C947D9E859
Subkey 1: 011110000011001111000011001000001101101001110000
Subkey 2: 001010110001101001110100110010100100100011011000
Subkey 3: 100011000111100011011000100000011101001100011101
Subkey 4: 000101100110011101111000100100110001011010100000
Subkey 5: 110011100101110100000001110110000000101100100101
Subkey 6: 010010111010101101001101000100100110101010011100
Subkey 7: 000010011111010010001011011100010011000110010001
Subkey 8: 011100010000110111101010101000110010000000101011
Subkey 9: 000100101001101010111000001100110100011111000011
Subkey 10: 100111000011100001100110000111101000000100000011
Subkey 11: 101000100110111001001100110001100110010101000100
Subkey 12: 010010000111011100100100011010001010001111001000
Subkey 13: 110000001001110101111001111100001101010000001011
Subkey 14: 110001011110001001100011010011100001011000101010
Subkey 15: 101000111101111110000010100111000111100101101000
Subkey 16: 101001100001001000001011010011010100110000100101
bipinsaud@bipin Cryptography Labs %

```


TITLE: WAP TO CONVERT THE PLAINTEXT IN BITS INTO STATE ARRAY

THEORY:

The Advanced Encryption Standard (AES) algorithm uses a two-dimensional array of bytes, known as the "state," for its internal processes. This state array consists of four rows, each containing four bytes, determined by dividing the block length by 32 ($N_b=4$ for standard AES). Each byte in the state is indexed by its row (r) and column (c) positions, which range from 0 to 3. During AES encryption or decryption, an initial array of 16 bytes ($in_0, in_1, \dots, in_{15}$) is copied into the state array. The encryption or decryption operations are then executed on this state array. After these processes, the final values in the state array are transferred to the output array of bytes ($out_0, out_1, \dots, out_{15}$).

ALGORITHM

1. Start
2. Input Plaintext:
 - Read plaintext input (up to 16 characters) from the user.
3. Initialization:
 - Initialize a 4x4 state array (state) and a 16x8 bits array (bits) with zeros.
4. Plaintext to State Array Conversion:
 - Iterate over the first 16 characters of the plaintext.
 - Convert alphabetic characters to uppercase and map them to a value (0-25), placing the value in the state array.
 - Pad with 0 for non-alphabetic characters.
5. Plaintext to Bits Conversion:
 - For each character, convert to uppercase, map to a value (0-25), convert to 8-bit binary, and store each bit in the bits array.
6. Output:

Print the bits array as a 4x4 matrix.

 - Print the state array in hexadecimal format.
7. End

IDE: Vs-Code

LANGUAGE: C

SOURCE CODE

```
/*12.WAP to convert the plaintext in bits into state
array.*/

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

// Function to convert plaintext to a state array of
corresponding positions (0-25)
void plaintext_to_state_array(const char *plaintext, unsigned
char state[4][4])
{
    for (int i = 0; i < 16; i++)
    {
        int row = i / 4;
        int col = i % 4;
        if (i < strlen(plaintext) && isalpha(plaintext[i]))
        {
            // Convert character to uppercase and find its
position (0-25)
            state[row][col] = toupper(plaintext[i]) - 'A';
        }
        else
        {
            state[row][col] = 0; // Padding with 0 if the
character is not a letter
        }
    }
}

// Function to convert plaintext characters to their binary
representation and store in a 4x4 matrix
void plaintext_to_bits(const char *plaintext, unsigned char
bits[16][8])
{
    for (int i = 0; i < 16; i++)
    {
        unsigned char ch = (i < strlen(plaintext) &&
isalpha(plaintext[i])) ? toupper(plaintext[i]) - 'A' : 0;
```

```

        for (int j = 0; j < 8; j++)
        {
            bits[i][j] = (ch >> (7 - j)) & 1;
        }
    }

// Function to print the state array in hexadecimal
void print_state_array_hex(unsigned char state[4][4])
{
    printf("State array in hex:\n");
    for (int row = 0; row < 4; row++)
    {
        for (int col = 0; col < 4; col++)
        {
            printf("%02X ", state[row][col]);
        }
        printf("\n");
    }
}

// Function to print the bit representation of the plaintext
in a 4x4 matrix
void print_bits_matrix(unsigned char bits[16][8])
{
    printf("Plaintext in bits (4x4 matrix):\n");
    for (int i = 0; i < 16; i++)
    {
        for (int j = 0; j < 8; j++)
        {
            printf("%d", bits[i][j]);
        }
        printf(" ");
        if ((i + 1) % 4 == 0)
        {
            printf("\n");
        }
    }
}

int main()
{
    char plaintext[256];

```

```

printf("Enter the plaintext (up to 16 characters): ");
fgets(plaintext, sizeof(plaintext), stdin);
plaintext[strcspn(plaintext, "\n")] = '\0';

unsigned char state[4][4] = {0}; // Initialize state array
with zeros
unsigned char bits[16][8] = {0}; // Initialize bits array
with zeros

plaintext_to_state_array(plaintext, state);
plaintext_to_bits(plaintext, bits);

print_bits_matrix(bits);
print_state_array_hex(state);

return 0;
}

```

OUTPUT:

```

Enter the plaintext (up to 16 characters): bipinsaud
Plaintext in bits (4x4 matrix):
00000001 00001000 00001111 00001000
00001101 00010010 00000000 00010100
00000011 00000000 00000000 00000000
00000000 00000000 00000000 00000000
State array in hex:
01 08 0F 08
0D 12 00 14
03 00 00 00
00 00 00 00
bipinsaud@bipin Cryptography Labs %

```

TITLE: WAP TO ENCRYPT AND DECRYPT THE MESSAGE USING RSA

THEORY:

RSA (Rivest-Shamir-Adleman) is a widely-used public-key cryptography algorithm that relies on the difficulty of factoring large integers into their prime factors, making it computationally infeasible to reverse-engineer the private key from the public key. The core concept involves generating a public key consisting of a modulus (n) and an exponent (e), along with a corresponding private key. The encryption process involves raising the plaintext message to the power of the public exponent (e) modulo the modulus (n), while decryption involves raising the ciphertext to the power of the private exponent (d) modulo the same modulus (n). This ensures that only the holder of the private key can decrypt the message, providing secure communication over an insecure channel.

ALGORITHM

1. Start
2. Key Generation:
 - a. Choose two large prime numbers, p and q .
 - b. Compute $n = p * q$.
 - c. Calculate $\phi = (p - 1) * (q - 1)$.
 - d. Find a public exponent e such that $\text{GCD}(e, \phi) == 1$.
 - e. Determine a private exponent d such that $(d * e) \% \phi == 1$.
3. Encryption:
 - a. Convert the plaintext message into numerical form (m).
 - b. Encrypt each character by computing $c = (m^e) \% n$.
 - c. The encrypted characters represent the ciphertext.
4. Decryption:
 - a. Decrypt each ciphertext character by computing $m = (c^d) \% n$.
 - b. Convert the numerical result back into characters.
 - c. The decrypted characters represent the original plaintext message.
5. Stop

IDE: Vs-Code

LANGUAGE: C

SOURCE CODE

```
/*13.WAP to encrypt and decrypt the message using RSA.*/

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

// Function to compute GCD of two numbers
int GCD(int a, int b)
{
    if (b == 0)
        return a;
    return GCD(b, a % b);
}

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

int main()
{
    int p, q, n, phi;
    int d = 2, e = 2;
    char message[100];
    long long c, m1;

    // Input prime numbers p and q
    printf("Enter 1st prime number: ");
    scanf("%d", &p);
    printf("Enter 2nd prime number: ");
    scanf("%d", &q);
    printf("Enter message: ");
    scanf("%s", message);

    // Calculate n and phi
```

```

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

printf("\nFormulas for key generation:");
printf("\nn = p * q");
printf("\nphi = (p - 1) * (q - 1)");
printf("\nFind e such that GCD(e, phi) == 1");
printf("\nFind d such that (d * e) %% phi == 1\n");

printf("\nValues:");
printf("\nValue of n is: %d", n);
printf("\nValue of phi is: %d\n", phi);

// Find e such that GCD(e, phi) == 1
while (GCD(e, phi) != 1)
{
    e++;
}

// Find d such that (d * e) % phi == 1
while ((d * e) % phi != 1)
{
    d++;
}
printf("\nValue of e = %d", e);
printf("\nValue of d = %d\n\n", d);

printf("\n\nEncryption\n");
printf("Encryption formula: c = (m^e) %% n\n");
// Encrypt the message
printf("Encrypted characters:\n");
for (int i = 0; i < strlen(message); i++)
{
    int m = message[i];
    c = mod_exp(m, e, n);
    int encrypted_mod26 = c % 26;
    char encrypted_char = 'A' + encrypted_mod26;
    printf("Character '%c' (ASCII %d) encrypted to %lld\n",
(mod 26: %d, as char: '%c')\n",
        message[i], m, c, encrypted_mod26,
encrypted_char);
}

printf("\n\nDecryption\n");
printf("Decryption formula: m = (c^d) %% n\n");
// Decrypt the message
printf("Decrypted characters:\n");
for (int i = 0; i < strlen(message); i++)
{
    int m = message[i];
    c = mod_exp(m, e, n);

```

```

        m1 = mod_exp(c, d, n);
        printf("Encrypted value %lld decrypted to character
'%c' (ASCII %lld)\n", c, (char)m1, m1);
    }

    return 0;

```

OUTPUT:

```

Enter 1st prime number: 23
Enter 2nd prime number: 7
Enter message: hello

Formulas for key generation:
n = p * q
phi = (p - 1) * (q - 1)
Find e such that GCD(e, phi) == 1
Find d such that (d * e) % phi == 1

Values:
Value of n is: 161
Value of phi is: 132

Value of e = 5
Value of d = 53

Encryption
Encryption formula: c = (m^e) % n
Encrypted characters:
Character 'h' (ASCII 104) encrypted to 41 (mod 26: 15, as char: 'P')
Character 'e' (ASCII 101) encrypted to 54 (mod 26: 2, as char: 'C')
Character 'l' (ASCII 108) encrypted to 75 (mod 26: 23, as char: 'X')
Character 'l' (ASCII 108) encrypted to 75 (mod 26: 23, as char: 'X')
Character 'o' (ASCII 111) encrypted to 34 (mod 26: 8, as char: 'I')

Decryption
Decryption formula: m = (c^d) % n
Decrypted characters:
Encrypted value 41 decrypted to character 'h' (ASCII 104)
Encrypted value 54 decrypted to character 'e' (ASCII 101)
Encrypted value 75 decrypted to character 'l' (ASCII 108)
Encrypted value 75 decrypted to character 'l' (ASCII 108)
Encrypted value 34 decrypted to character 'o' (ASCII 111)
bipinsaud@bipin Cryptography Labs % █

```


LAB_14

Date: 2081-02-29

**TITLE: WAP TO ENCRYPT AND DECRYPT THE MESSAGE USING
ELGAMEL**

THEORY:

The ElGamal cryptosystem is a well-known public-key cryptographic system used for secure data encryption and digital signatures. Based on the Diffie-Hellman key exchange protocol, it operates within a cyclic group, typically involving integers modulo a prime number. The ElGamal system includes three main stages: key generation, encryption, and decryption.

ALGORITHM

1.Start

2.Key Generation

- Choose a large prime number p .
- Select a primitive root α such that $\alpha < q$.
- Select a private key X_A such that $X_A < q-1$.
- Calculate $Y_A = \alpha^{X_A} \bmod q$.
- Obtaining a public key $PU = \{ q, \alpha, Y_A \}$

3.Encryption

- Enter the plaintext.
- Select a random integer r such that $r < q$.
- Calculate key, $K = (Y_A)^r \bmod q$.
- Calculate two values $C1$ and $C2$.

$$C1 = \alpha^r \bmod q$$

$$C2 = KM \bmod q$$

4.Decryption

- Calculate Key, $K = (C1)^{X_A} \bmod q$.
- Plaintext, $M = (C2K^{-1}) \bmod q$.

5.End

IDE: Vs-Code

LANGUAGE: C

SOURCE CODE

```
/*14.WAP to encrypt and decrypt the message using Elgamel.*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

// Function to compute gcd
int gcd(int a, int b)
{
    if (b == 0)
        return a;
    return gcd(b, a % b);
}

// Extended Euclidean Algorithm to find d
int extended_gcd(int a, int b, int *x, int *y)
{
    if (a == 0)
    {
        *x = 0;
        *y = 1;
        return b;
    }
    int x1, y1;
    int gcd = extended_gcd(b % a, a, &x1, &y1);
    *x = y1 - (b / a) * x1;
    *y = x1;
    return gcd;
}

int mod_inverse(int a, int m)
{
    int x, y;
    int g = extended_gcd(a, m, &x, &y);
    if (g != 1)
    {
        printf("Inverse doesn't exist");
        exit(1);
    }
    return (x % m + m) % m;
}

int mod_exp(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 main()
{
    int q, alpha, Xa, Ya, c1, r, K1, K2, K_inverse;
    char message[256];

    printf("Enter a prime number q: ");
    scanf("%d", &q);
    printf("Enter alpha such that alpha is primitive root of
q: ");
    scanf("%d", &alpha);

    printf("Key generation by Alice: \n");
    printf("Enter a private key(Xa) such that Xa < q: ");
    scanf("%d", &Xa);
    Ya = mod_exp(alpha, Xa, q);
    printf("Public key = {%d, %d, %d}\n", q, alpha, Ya);

    printf("Encryption by Bob with Alice's Public Key:\n");
    printf("Enter a message (uppercase letters only): ");
    scanf("%s", message);

    int length = strlen(message);
    int m[length], c2[length];

    printf("Message\tNumber\n");
    for (int i = 0; i < length; i++)
    {
        m[i] = message[i] - 'A';
        printf("%c\t%d\n", message[i], m[i]);
    }

    printf("Select a random integer r (r < q): ");
    scanf("%d", &r);

    K1 = mod_exp(Ya, r, q);
    printf("K1 = %d\n", K1);

    c1 = mod_exp(alpha, r, q);
    printf("c1 = %d\n", c1);
    printf("Message(0-25)\tNumber\n");
    for (int i = 0; i < length; i++)
    {
        c2[i] = (K1 * m[i]) % q;
        printf("%d\t\t%d\n", m[i], c2[i]);
    }
}

```

```

printf("CipherText: (c1, c2)\n");
printf("c1 = %d, c2 = ", c1);
for (int i = 0; i < length; i++)
{
    printf("%d ", c2[i]);
}
printf("\n");

printf("Decryption by Alice with Alice's Private key:
\n");
K2 = mod_exp(c1, Xa, q);
printf("K2 = %d\n", K2);

K_inverse = mod_inverse(K2, q);
printf("K_inverse = %d\n", K_inverse);

printf("Decrypted message: ");
for (int i = 0; i < length; i++)
{
    int M = (c2[i] * K_inverse) % q;
    printf("%c", M + 'A');
}
printf("\n");

return 0;
}

```

OUTPUT:

```

Enter a prime number q: 101
Enter alpha such that alpha is primitive root of q: 2
Key generation by Alice:
Enter a private key(Xa) such that Xa < q: 10
Public key = {101, 2, 14}
Encryption by Bob with Alice's Public Key:
Enter a message (uppercase letters only): BIPIN
Message Number
B      1
I      8
P     15
I      8
N     13
Select a random integer r (r < q): 4
K1 = 36
c1 = 16
Message(0-25)   Number
1               36
8               86
15              35
8               86
13              64
CipherText: (c1, c2)
c1 = 16, c2 = 36 86 35 86 64
Decryption by Alice with Alice's Private key:
K2 = 36
K_inverse = 87
Decrypted message: BIPIN
bipinsaud@bipin Cryptography Labs %

```

TITLE: WAP TO SIMULATE MALICIOUS LOGIC

THEORY:

This code illustrates how malicious logic can be embedded in software to perform harmful actions. It prompts the user for their name and then executes shutdown commands for both Unix-like and Windows systems, leading to immediate system shutdown. The code serves as an example to understand how malicious software can cause damage and highlights the importance of security awareness and caution when executing unfamiliar code.

ALGORITHM

1. Initialize Program:

- Start the program.

2. Prompt for User Input:

- Display the message "Enter your name: " on the screen.
- Read the user's input and store it in the variable name.

3. Display Malicious Message:

- Print a message "You are hacked [user's name]" to the screen, where [user's name] is the value entered by the user.

4. Execute Shutdown Commands:

- For Unix-like Systems:
 - Execute the command `sudo shutdown -h now` to immediately shut down the computer.
- For Windows Systems:
 - Execute the command `shutdown /s /t 0` to immediately shut down the computer.

5. End Program:

- Exit the program.

IDE: Vs-Code

LANGUAGE: C

SOURCE CODE

```
// 15. WAP TO SIMULATE MALICIOUS LOGIC.
#include <stdio.h>
#include <stdlib.h>

int main()
{
    char name[100];

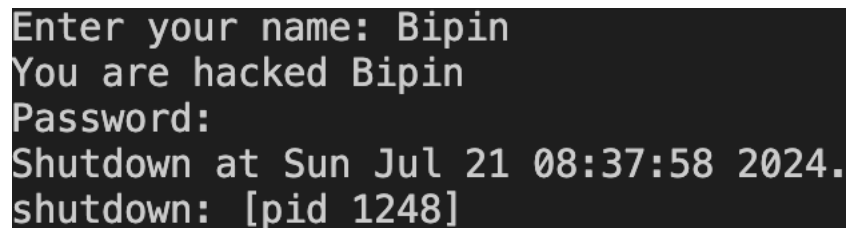
    printf("Enter your name: ");
    scanf("%s", name);

    printf("You are hacked %s\n", name);
    // mac
    system("sudo shutdown -h now");

    //(Windows)
    system("shutdown /s /t 0");

    return 0;
}
```

OUTPUT

A terminal window with a black background and white text. The output of the program is displayed as follows:

```
Enter your name: Bipin
You are hacked Bipin
Password:
Shutdown at Sun Jul 21 08:37:58 2024.
shutdown: [pid 1248]
```

A small grey rectangular cursor is visible at the bottom left of the terminal window.

TITLE: BIRTHDAY PARADOX DEMONSTRATION IN EXCEL

THEORY:

The Birthday Paradox demonstrates that in a group of 23 people, there's over a 50% chance two share the same birthday, based on probability theory. This theoretical probability is shown through simulations and calculations in Excel. By setting up a spreadsheet to generate random birthdays and calculating the probability of shared birthdays, you can visually and mathematically understand this counterintuitive concept.

PROCEDURE :

1. Set Up Your Spreadsheet:

- Column A: Label as "No. of People" and list numbers from 0 to 23.
- Column B: Label as "p(no match)" to calculate the probability that no two people share the same birthday.
- Column C: Label as "p(match)" to calculate the probability that at least two people share the same birthday.
- Column D: Label as "percentage" to convert the probability in Column C to a percentage.

2. Generate Probabilities:

- In cell B1, enter 1 because with 0 people, the probability of no matches is 100%.
- In cell C1, enter 0 because with 0 people, the probability of a match is 0%.
- In cell D1, enter 0 as the percentage of match probability for 0 people.

3. Calculate Probabilities:

- For cell B2, enter =1 because with 1 person, the probability of no matches is 100%.
- For cell C2, enter =0 because with 1 person, the probability of a match is 0%.
- For cell D2, enter =0 as the percentage of match probability for 1 person.

4. Fill the Formulas:

- In cell B3, enter the formula: $=B2 * (365 - (A3 - 1)) / 365$.
 - This formula calculates the probability that the third person does not share a birthday with the first two people, and multiplies it by the probability that the first two people do not share a birthday.

- Drag this formula down from B3 to B24 to fill the probabilities for "p(no match)" for 2 to 23 people.
 - Example: For cell B4, the formula would be $=B3 * (365 - (A4 - 1)) / 365$.

5. Calculate Matching Probabilities:

- In cell C3, enter the formula: $=1 - B3$.
 - This formula calculates the probability that at least two people share a birthday by subtracting the probability of no matches from 1.
- Drag this formula down from C3 to C24 to fill the probabilities for "p(match)" for 2 to 23 people.

6. Convert to Percentages:

- In cell D3, enter the formula: $=C3 * 100$.
- Drag this formula down from D3 to D24 to fill the percentages for the matching probabilities for 2 to 23 people.

OUTPUT :

