



LAB REPORT OF CRYPTOGRAPHY

Submitted to

Bikram Acharya

Samriddhi College

Submitted by

Suchak Niraula

BSc.CSIT (5th Semester)

Reg No.: 5-2-1113-45-2020

Symbol No.: 26836/077

TABLE OF CONTENTS

LAB - 1: IMPLEMENTATION OF CIPHER.....	1
1.1 CEASER CIPHER.....	1
1.2 MONOALPHABETIC SUBSTITUTION CIPHER.....	3
1.3 PLAYFAIR CIPHER.....	4
1.4 HILL CIPHER.....	8
1.5 ONE TIME PAD CIPHER.....	11
1.6 VIGENERE CIPHER.....	13
1.7 RAIL FENCE CIPHER.....	15
LAB -2: DATA ENCRYPTION STANDARDS.....	17
LAB- 3: ADDITIVE INVERSE IN MODULAR ARITHMETIC.....	22
LAB- 4: EUCLIDEAN ALGORITHM.....	23
LAB- 5: EXTENDED EUCLIDEAN ALGORITHM	24
LAB- 6: RELATIVELY PRIME	26
LAB- 7: MILLER RABIN PRIMALITY TEST	27
LAB- 8: EULER TOTIENT FUNCTION.....	29
LAB- 9: PRIMITIVE ROOT.....	30
LAB- 10: DIFFIE HELLMAN ALGORITHM.....	31
LAB- 11: RSA ALGORITHM	33

LAB - 1: IMPLEMENTATION OF CIPHER

1.1 CEASER CIPHER

AIM:

To implement a program to simulate Caesar cipher.

THEORY:

The Caesar Cipher technique is one of the earliest and simplest methods of encryption technique. It's simply a type of substitution cipher, i.e., each letter of a given text is replaced by a letter with a fixed number of positions down the alphabet.

For example: with a shift of 1, A would be replaced by B, B would become C, and so on. The method is apparently named after Julius Caesar, who apparently used it to communicate with his officials.

Plaintext: It is a simple message written by the user.

Eg : ABCDEFGHIJKLMNOPQRSTUVWXYZ

Shift key: 23

Ciphertext: It is an encrypted message after applying some technique.

Eg: XYZABCDEFGHIJKLMNOPQRSTUVW

The formula of encryption is:

$$En(x) = (x + n) \bmod 26$$

The formula of decryption is:

$$Dn(x) = (xi - n) \bmod 26$$

If any case (Dn) value becomes negative (-ve), in this case, we will add 26 in the negative value.

where, E = encryption

D= decryption

x = letters value

n = key value (shift value)

SOURCE CODE:

```
#include<stdio.h>
#include<conio.h>
int main() {
    int i, f;
    char pop[100], c;
    printf("Enter a plaintext: ");
    gets(pop);
    printf("Enter key: ");
    scanf("%d", &f);
    for(i = 0; pop[i] != '\0'; i++) {
        c = pop[i];
        if(c >= 'a' && c <= 'z') {
            c = c + f;
            if(c > 'z') {
                c = c - 'z' + 'a' - 1;
            }
        }
    }
}
```

```

    }
    pop[i] = c;
}
else if(c >= 'A' && c < 'Z') {
    c = c + f;
    if(c > 'Z') {
        c = c = 'Z' + 'A' - 1;
    }
    pop[i] = c;
}
}
printf("Encrypted message: %s", pop);
for(i = 0; pop[i] != '\0'; i++) {
    c = pop[i];
    if(c >= 'a' && c <= 'z') {
        c = c - f;
        if(c < 'a') {
            c = c + 'z' - 'a' + 1;
        }
        pop[i] = c;
    }
    else if(c >= 'A' && c < 'Z') {
        c = c + f;
        if(c < 'A') {
            c = c + 'Z' - 'A' + 1;
        }
        pop[i] = c;
    }
}
printf("\nDecrypted message: %s", pop);
return 0;
}

```

OUTPUT:

```

Enter a plaintext: string
Enter key: baall
Encrypted message: pqoofkd
Decrypted message: string

```

1.2 MONOALPHABETIC SUBSTITUTION CIPHER

AIM:

To implement a program to simulate Monoalphabetic Substitution cipher.

THEORY:

Monoalphabetic Cipher is a cipher where the letters of the plain text are mapped to ciphertext letters based on a single alphabetic key. It is a one-to-one mapping.

Monoalphabetic cipher is a substitution cipher in which for a given key, the cipher alphabet for each plain alphabet is fixed throughout the encryption process.

For example, if 'A' is encrypted as 'D', for any number of occurrences in that plaintext, 'A' will always get encrypted to 'D'.

ALGORITHM:

1. Set unique monoalphabetic key
2. Input plaintext or cipher text
3. For plaintext, Set the key for each character in text For Cipher text, Set the cipher text for each character in text
4. End the program

SOURCE CODE:

```
#include<stdio.h>
char monocipher_encr(char);
char alpha[27][3] = { { 'a', 'f' }, { 'b', 'a' }, { 'c', 'g' }, { 'd', 'u' }, { 'e', 'n' }, { 'f', 'i' }, { 'g', 'j' }, { 'h', 'k' }, { 'i', 'l' }, { 'j', 'm' }, { 'k', 'o' }, { 'l', 'p' }, { 'm', 'q' }, { 'n', 'r' }, { 'o', 's' }, { 'p', 't' }, { 'q', 'v' }, { 'r', 'w' }, { 's', 'x' }, { 't', 'y' }, { 'u', 'z' }, { 'v', 'b' }, { 'w', 'c' }, { 'x', 'd' }, { 'y', 'e' }, { 'z', 'h' } };
char str[20];
int main() {
    char str[20], str2[20];
    int i;
    printf("\n Enter String: ");
    gets(str);
    for (i = 0; str[i]; i++) {
        str2[i] = monocipher_encr(str[i]);
    }
    str2[i] = '\0';
    printf("\n Plain text : %s", str);
    printf("\n After encryption :%s\n", str2);
}
char monocipher_encr(char a) {
    int i;
    for (i = 0; i < 27; i++) {
        if (a == alpha[i][0])
            break;
    }
    return alpha[i][1];
}
```

OUTPUT:

```
Enter String: helloworld
Plain text : helloworld
After encryption :knppscswpu
|
```

1.3 PLAYFAIR CIPHER

AIM:

To implement a program to simulate Playfair cipher.

THEORY:

Playfair cipher is an encryption algorithm to encrypt or encode a message. It is the same as a traditional cipher. The only difference is that it encrypts a digraph (a pair of two letters) instead of a single letter.

ALGORITHM:

The playfair cipher encryption algorithm:

1. Generate the key square (5×5):
 - The key square is a 5×5 grid of alphabets that acts as the key for encrypting the plaintext. Each of the 25 alphabets must be unique and one letter of the alphabet (usually J) is omitted from the table (as the table can hold only 25 alphabets). If the plaintext contains J, then it is replaced by I.
 - The initial alphabets in the key square are the unique alphabets of the key in the order in which they appear followed by the remaining letters of the alphabet in order.
2. Algorithm to encrypt the plain text:

The plaintext is split into pairs of two letters (digraphs). If there is an odd number of letters, a Z is added to the last letter.

SOURCE CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define SIZE 30
void toLowerCase(char plain[], int ps){
    int i;
    for (i = 0; i < ps; i++) {
        if (plain[i] > 64 && plain[i] < 91)
            plain[i] += 32;
    }
}
int removeSpaces(char* plain, int ps){
    int i, count = 0;
    for (i = 0; i < ps; i++)
        if (plain[i] != ' ')
            plain[count++] = plain[i];
    plain[count] = '\0';
    return count;
}
void generateKeyTable(char key[], int ks, char keyT[5][5]){
    int i, j, k, flag = 0, *dicty;
```

```

dicty = (int*)calloc(26, sizeof(int));
for (i = 0; i < ks; i++) {
    if (key[i] != 'j')
        dicty[key[i] - 97] = 2;
}
dicty['j' - 97] = 1;
i = 0;
j = 0;
for (k = 0; k < ks; k++) {
    if (dicty[key[k] - 97] == 2) {
        dicty[key[k] - 97] -= 1;
        keyT[i][j] = key[k];
        j++;
        if (j == 5) {
            i++;
            j = 0;
        }
    }
}
for (k = 0; k < 26; k++) {
    if (dicty[k] == 0) {
        keyT[i][j] = (char)(k + 97);
        j++;
        if (j == 5) {
            i++;
            j = 0;
        }
    }
}
}

void search(char keyT[5][5], char a, char b, int arr[])
{
    int i, j;

    if (a == 'j')
        a = 'i';
    else if (b == 'j')
        b = 'i';

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

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

            if (keyT[i][j] == a) {
                arr[0] = i;
            }
        }
    }
}

```

```

        arr[1] = j;
    }
    else if (keyT[i][j] == b) {
        arr[2] = i;
        arr[3] = j;
    }
}
}
}

int mod5(int a) { return (a % 5); }

int prepare(char str[], int ptrs)
{
    if (ptrs % 2 != 0) {
        str[ptrs++] = 'z';
        str[ptrs] = '\0';
    }
    return ptrs;
}

void encrypt(char str[], char keyT[5][5], int ps)
{
    int i, a[4];

    for (i = 0; i < ps; i += 2) {

        search(keyT, str[i], str[i + 1], a);

        if (a[0] == a[2]) {
            str[i] = keyT[a[0]][mod5(a[1] + 1)];
            str[i + 1] = keyT[a[0]][mod5(a[3] + 1)];
        }
        else if (a[1] == a[3]) {
            str[i] = keyT[mod5(a[0] + 1)][a[1]];
            str[i + 1] = keyT[mod5(a[2] + 1)][a[1]];
        }
        else {
            str[i] = keyT[a[0]][a[3]];
            str[i + 1] = keyT[a[2]][a[1]];
        }
    }
}

void encryptByPlayfairCipher(char str[], char key[])
{

```



```

char ps, ks, keyT[5][5];

// Key
ks = strlen(key);
ks = removeSpaces(key, ks);
toLowerCase(key, ks);

ps = strlen(str);
toLowerCase(str, ps);
ps = removeSpaces(str, ps);

ps = prepare(str, ps);

generateKeyTable(key, ks, keyT);

encrypt(str, keyT, ps);
}

// Driver code
int main()
{
    char str[SIZE], key[SIZE];

    // Key to be encrypted
    strcpy(key, "playfair");
    printf("Key text: %s\n", key);

    // Plaintext to be encrypted
    strcpy(str, "suchakniraula");
    printf("Plain text: %s\n", str);

    // encrypt using Playfair Cipher
    encryptByPlayfairCipher(str, key);

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

    return 0;
}

```

OUTPUT:

```

Key text: playfair
Plain text: suchakniraula
Cipher text: nxbkyhueblvpfw

```

1.4 HILL CIPHER

AIM:

To implement a program to simulate Hill cipher.

THEORY:

In classical cryptography, the hill cipher is a polygraphic substitution cipher based on Linear Algebra.

Hill cipher is a polygraphic substitution cipher based on linear algebra. Each letter is represented by a number modulo 26. Often the simple scheme A = 0, B = 1, ..., Z = 25 is used, but this is not an essential feature of the cipher. To encrypt a message, each block of n letters (considered as an n-component vector) is multiplied by an invertible $n \times n$ matrix, against modulus 26.

To decrypt the message, each block is multiplied by the inverse of the matrix used for encryption.

The matrix used for encryption is the cipher key, and it should be chosen randomly from the set of invertible $n \times n$ matrices (modulo 26).

Encryption: The given message string and key string is represented in the form of matrix. Then key and message matrix are multiplied. Finally, modulo 26 is taken for each element of matrix obtained by multiplication. The key matrix that we take here should be invertible, otherwise decryption will not be possible.

Decryption: The encrypted message matrix is multiplied by the inverse of key matrix and finally its modulo 26 is taken to get the original message.

SOURCE CODE:

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

int determinant(int a, int b, int c, int d) {
    return a * d - b * c;
}

int modInverse(int a, int m) {
    a = a % m;
    for (int i = 1; i < m; i++)
        if ((a * i) % m == 1)
            return i;
    return -1;
}

void hillCipherEncrypt(char *message, int key[2][2]) {
    int len = strlen(message);
    if (len % 2 != 0) {
        message[len] = 'X';
        len++;
    }
    for (int i = 0; i < len; i += 2) {
```

```

    int char1 = message[i] - 'A';
    int char2 = message[i + 1] - 'A';

    int result1 = (key[0][0] * char1 + key[0][1] * char2) % 26;
    int result2 = (key[1][0] * char1 + key[1][1] * char2) % 26;

    message[i] = 'A' + result1;
    message[i + 1] = 'A' + result2;
}
}

int main() {
    char message[100];
    int key[2][2];
    printf("Enter the message (uppercase letters only): ");
    scanf("%s", message);

    printf("Enter the 2x2 key matrix (space-separated): ");
    for (int i = 0; i < 2; i++)
        for (int j = 0; j < 2; j++)
            scanf("%d", &key[i][j]);
    hillCipherEncrypt(message, key);
    printf("Encrypted message: %s\n", message);

    return 0;
}

```

```

Enter the message (uppercase letters only): SUCHAK
Enter the 2x2 key matrix (space-separated): 4 3 2 1
Encrypted message: CEDLEK

```

1.5 ONE TIME PAD CIPHER

AIM:

To implement a program to simulate One time pad cipher.

THEORY:

It is the only available algorithm that is unbreakable (completely secure). It is a method of encrypting alphabetic plain text. It is one of the Substitution techniques which converts plain text into ciphertext. In this mechanism, we assign a number to each character of the Plain-Text.

The two requirements for the One-Time pad are:

- The key should be randomly generated as long as the size of the message.
- The key is to be used to encrypt and decrypt a single message, and then it is discarded.

ALGORITHM:

1. Enter the plaintext.
2. Generate a random key.
3. For encryption, XOR every bit of the key with the plaintext.
4. For decryption, again XOR every bit of the key with the ciphertext.
5. End the program.

SOURCE CODE:

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

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

void oneTimePad(char *message, char *key, char *result) {
    int len = strlen(message);

    for (int i = 0; i < len; i++) {
        // XOR operation
        result[i] = ((message[i] - 'A') ^ (key[i] - 'A')) + 'A';
    }
    result[len] = '\0';
}

int main() {
    srand(time(NULL));

    char message[100];
    char key[100];
```

```

char encrypted[100];
char decrypted[100];

printf("Enter the message (uppercase letters only): ");
scanf("%s", message);

generateKey(key, strlen(message));

oneTimePad(message, key, encrypted);

printf("Message: %s\n", message);
printf("Key:    %s\n", key);
printf("Encrypted message: %s\n", encrypted);

oneTimePad(encrypted, key, decrypted);

printf("Decrypted message: %s\n", decrypted);

return 0;
}

```

OUTPUT:

```

Enter the message (uppercase letters only): ABCDE
Message: ABCDE
Key:    RJIHV
Encrypted message: RIKER
Decrypted message: ABCDE

```

1.6 VIGENERE CIPHER

AIM:

To implement a program to simulate vigenere cipher.

THEORY:

Vigenere Cipher is a method of encrypting alphabetic text. It uses a simple form of polyalphabetic substitution. A polyalphabetic cipher is any cipher based on substitution, using multiple substitution alphabets. The encryption of the original text is done using the Vigenère square or Vigenère table.

- The table consists of the alphabets written out 26 times in different rows, each alphabet shifted cyclically to the left compared to the previous alphabet, corresponding to the 26 possible ceaser ciphers.
- At different points in the encryption process, the cipher uses a different alphabet from one of the rows.
- The alphabet used at each point depends on a repeating keyword.

Encryption: $E_i = (P_i + K_i) \bmod 26$

Decryption: $D_i = (E_i - K_i) \bmod 26$

ALGORITHM:

1. Input the plaintext and the keyword.
2. Repeat the keyword so as to make it as the length of the plaintext.
3. Encryption: Perform modular addition of the repeating keyword and the plaintext.
 $C_i = P_i + K_i \pmod{m}$
Where, C_i is the cipher text,
 P_i is the plaintext,
 K_i is the repeating keyword and
'm' is the length of the alphabet.
4. Decryption: Perform modular subtraction of key phrase from the cipher text.
 $P_i = C_i - K_i \pmod{m}$
5. Display the encrypted and decrypted message as well.
6. End the program.

SOURCE CODE:

```
#include<stdio.h>
#include<string.h>
int main(){
    char msg[] = "THISISCAT"; char key[] = "MEOW";
    int msgLen = strlen(msg), keyLen = strlen(key), i, j;
    char newKey[msgLen], encryptedMsg[msgLen], decryptedMsg[msgLen];
    for(i = 0, j = 0; i < msgLen; ++i, ++j){ if(j == keyLen)
        j = 0;
        newKey[i] = key[j];
    }
    newKey[i] = '\0';

    //encryption
    for(i = 0; i < msgLen; ++i)
        encryptedMsg[i] = ((msg[i] + newKey[i]) % 26) + 'A'; encryptedMsg[i] = '\0';
    //decryption
```

```
for(i = 0; i < msgLen; ++i)
    decryptedMsg[i] = (((encryptedMsg[i] - newKey[i]) + 26) % 26) + 'A'; decryptedMsg[i] = '\0';
printf("Original Message: %s", msg); printf("\nKey: %s", key);
printf("\nNew Generated Key: %s", newKey); printf("\nEncrypted Message: %s",
    encryptedMsg); printf("\nDecrypted Message: %s", decryptedMsg);
return 0;
}
```

OUTPUT:

```
Original Message: THISISCAT
Key: MEOW
New Generated Key: MEOWMEOWM
Encrypted Message: FLWOUWQWF
Decrypted Message: THISISCAT
```

1.7 RAIL FENCE CIPHER

AIM:

To implement a program to simulate rail fence cipher.

THEORY:

In a rail fence cipher, letters are not changed, but only switched around regarding their positioning in the message. This type of cipher is often called a transposition cipher, because letters are simply transposed in terms of their placement. Transposition ciphers like the rail fence cipher are relatively weak forms of encoding, and can easily be broken, especially with today's technology. These types of ciphers date back to the American Civil War, where soldiers would use the code to send encrypted messages.

In a rail fence cipher, the writer takes a message and writes it into descending lines or "rails." The rail fence cipher is sometimes called a zig zag cipher if the writer uses a zigzag or W pattern to represent text.

In order to encode the text, the user takes the letters in the top line, or rail, and puts them together. He or she then writes out the second line and the third line. The result is an encoded line of text. For example, using the phrase "hello world" and a series of three rails, the result (for a linear descent) would be HLODEORLWL.

SOURCE CODE:

```
#include<stdio.h>
#include<string.h>
void encryptMsg(char msg[], int key){
    int msgLen = strlen(msg), i, j, k = -1, row = 0, col = 0;
    char railMatrix[key][msgLen];
    for(i = 0; i < key; ++i)
        for(j = 0; j < msgLen; ++j)
            railMatrix[i][j] = '\n';
    for(i = 0; i < msgLen; ++i){
        railMatrix[row][col++] = msg[i];
        if(row == 0 || row == key-1)
            k = k * (-1);
        row = row + k;
    }
    printf("\nEncrypted Message: ");
    for(i = 0; i < key; ++i)
        for(j = 0; j < msgLen; ++j)
            if(railMatrix[i][j] != '\n')
                printf("%c", railMatrix[i][j]);
}
void decryptMsg(char enMsg[], int key){
    int msgLen = strlen(enMsg), i, j, k = -1, row = 0, col = 0, m = 0;
    char railMatrix[key][msgLen];
    for(i = 0; i < key; ++i)
        for(j = 0; j < msgLen; ++j)
            railMatrix[i][j] = '\n';
    for(i = 0; i < msgLen; ++i){
        railMatrix[row][col++] = '*';
```



```

        if(row == 0 || row == key-1)
            k= k * (-1);
        row = row + k;
    }
    for(i = 0; i < key; ++i)
        for(j = 0; j < msgLen; ++j)
            if(railMatrix[i][j] == '*')
                railMatrix[i][j] = enMsg[m++];
    row = col = 0;
    k = -1;
    printf("\nDecrypted Message: ");
    for(i = 0; i < msgLen; ++i){
        printf("%c", railMatrix[row][col++]);
        if(row == 0 || row == key-1)
            k= k * (-1);
        row = row + k;
    }
}
int main(){
    char msg[] = "Hello crypto";
    char enMsg[] = "Hoyel rpolct";
    int key = 3;
    printf("Original Message: %s", msg);
    encryptMsg(msg, key);
    decryptMsg(enMsg, key);
    return 0;
}

```

OUTPUT:

```

Original Message: Hello crypto
Encrypted Message: Hoyel rpolct
Decrypted Message: Hello crypto

```

CONCLUSION:

Hence, different cipher algorithms like ceaser cipher, monoalphabetic substitution cipher, playfair cipher, hill cipher, one time pad cipher, vigenere cipher and rail fence cipher were implemented.

LAB -2: DATA ENCRYPTION STANDARDS

AIM:

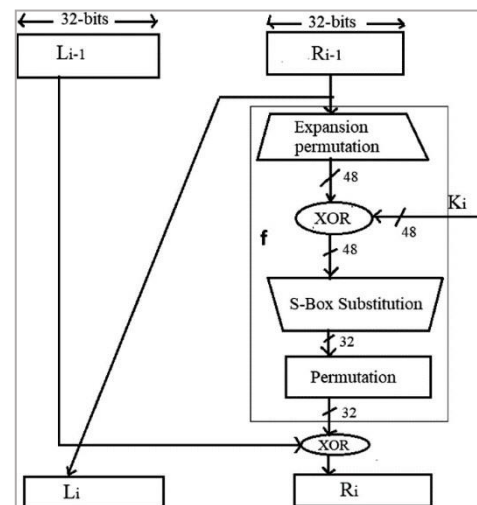
To implement DES cipher.

THEORY:

DES is the most widely used block cipher in world. DES is a careful and complex combination of the two fundamental building blocks of encryption: substitution and transposition. The algorithm derives its strength from repeated application of these two techniques (16 cycles), one on top of the other. DES is a cipher which encrypts blocks of length of 64 bits with a key of size of 56 bits. DES encrypts data in blocks of size of 64 bit each, means 64 bits of plain text goes as the input to DES, which produces 64 bits of cipher text. The same algorithm and key are used for encryption and decryption, with minor differences.

Steps of DES:

- 64-bit plaintext block is given to Initial Permutation (IP) Function.
- IP performed on 64 bit plain text block.
- IP produced two halves of the permuted block known as Left Plain Text(LPT) and Right Plain Text(RPT).
- Each LPT and RPT performed 16-rounds of encryption process.
- LPT and RPT rejoined and Final Permutation(FP) is performed on combined block.
- 64-bit Cipher text block is generated.



Initial Permutation in DES:

```
#include <iostream>
#include <bitset>
using namespace std;
// Define the IP block's permutation table
int IP[64] = {
    58, 50, 42, 34, 26, 18, 10, 2,
    60, 52, 44, 36, 28, 20, 12, 4,
    62, 54, 46, 38, 30, 22, 14, 6,
    64, 56, 48, 40, 32, 24, 16, 8,
    57, 49, 41, 33, 25, 17, 9, 1,
    59, 51, 43, 35, 27, 19, 11, 3,
    61, 53, 45, 37, 29, 21, 13, 5,
    63, 55, 47, 39, 31, 23, 15, 7
};
// Function to perform the IP block permutation
bitset<64> ipBlock(bitset<64> input) {
    bitset<64> output;
    for (int i = 0; i < 64; i++) {
        output[i] = input[IP[i]-1];
    }
}
```



```

int keyCompressionTable[] = {
    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
};

void generateKeys(uint64_t originalKey, uint64_t *subKeys) {
    uint64_t keyPerm = 0;
    for (int i = 0; i < 56; i++) {
        keyPerm <<= 1;
        keyPerm |= (originalKey >> (64 - initialPermTable[i])) & 1;
    }

    uint32_t leftHalf = keyPerm >> 28;
    uint32_t rightHalf = keyPerm & 0xFFFFFFFF;

    for (int round = 0; round < 16; round++) {
        leftHalf = ((leftHalf << 1) | (leftHalf >> 27)) & 0xFFFFFFFF;
        rightHalf = ((rightHalf << 1) | (rightHalf >> 27)) & 0xFFFFFFFF;

        uint64_t roundKey = ((uint64_t)leftHalf << 28) | rightHalf;

        uint64_t subKey = 0;
        for (int i = 0; i < 48; i++) {
            subKey <<= 1;
            subKey |= (roundKey >> (56 - keyCompressionTable[i])) & 1;
        }
        subKeys[round] = subKey;
    }
}

int main() {
    uint64_t originalKey = 0x133457799BBCDFF1;
    uint64_t subKeys[16];

    generateKeys(originalKey, subKeys);

    printf("Generated Subkeys:\n");
    for (int i = 0; i < 16; i++) {
        printf("Round %2d: 0x%012lx\n", i + 1, subKeys[i]);
    }

    return 0;
}

```

Generated Subkeys:

Round 1:	0x1b02effc7072
Round 2:	0x79aed9dbc9e5
Round 3:	0xb958bc65ea6e
Round 4:	0x55fc8a42cf99
Round 5:	0x942becb4bcda
Round 6:	0x72add6db351d
Round 7:	0x927e35ad9677
Round 8:	0x7cec07eb53a8
Round 9:	0xcd3f641feee2
Round 10:	0x63a53e507b2f
Round 11:	0xc2f6ed3ccd55
Round 12:	0xec84b7f618bc
Round 13:	0xd9d7628be4d6
Round 14:	0xf78a3ac13bfb
Round 15:	0xe0dbebede781
Round 16:	0xae2b237ba39

Substitution Box (S-BOX):

```
#include <iostream>
#include <bitset>

using namespace std;

int S[4][16] = {
    {14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7},
    {0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8},
    {4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0},
    {15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13}
};

bitset<4> sBlock(bitset<6> input) {
    int row = (input[5] << 1) + input[0];
    int col = (input[4] << 3) + (input[3] << 2) + (input[2] << 1) +
input[1];
    int value = S[row][col];
    bitset<4> output(value);
    return output;
}

int main() {
    bitset<6> input("011011");
    bitset<4> output = sBlock(input);

    cout << "Input: " << input << " (" << input.to_ulong() << ")" << endl;
    cout << "Output: " << output << " (" << output.to_ulong() << ")" <<
endl;

    return 0;
}
```

```
Input: 011011 (27)
Output: 0101 (5)
```

Permutation(P-BOX):

```
#include <iostream>
#include <bitset>
using namespace std;
int P[32] = {
    16, 7, 20, 21, 29, 12, 28, 17,
    1, 15, 23, 26, 5, 18, 31, 10,
    2, 8, 24, 14, 32, 27, 3, 9,
    19, 13, 30, 6, 22, 11, 4, 25
};
bitset<32> pBlock(bitset<32> input) {
    bitset<32> output;
    for (int i = 0; i < 32; i++) {
        output[i] = input[P[i]-1];
    }
    return output;
}

int main(){
    bitset<32> input("11001010011100011010001001100101");
    bitset<32> output = pBlock(input);
    cout << "Input: " << input << " (0x" << hex << input.to_ulong() << ")" << endl;
    cout << "Output: " << output << " (0x" << hex << output.to_ulong() << ")" << endl;
    return 0;
}
```

```
Input: 11001010011100011010001001100101 (0xca71a265)
Output: 00011000010110001100110111001011 (0x1858cdcb)
```

CONCLUSION:

Hence, DES cipher was implemented.

LAB- 3: ADDITIVE INVERSE IN MODULAR ARITHMETIC

AIM:

To find additive inverse of given integer.

THEORY:

In modular arithmetic, the additive inverse of an integer is another integer that when added to the original integer results in a multiple of the modulus. The additive inverse of an integer a in modulo n arithmetic is denoted by $-a$ and is given by $n-a$. Therefore, if we have a number a in modulo n arithmetic, the additive inverse of a is the number b such that:

$$a + b \equiv 0 \pmod{n}$$

In other words, b is the integer that, when added to a , results in a multiple of n .

SOURCE CODE:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
int main(){
    int n;
    printf("Enter the value of n:");
    scanf("%d",&n);
    printf("\nAdditive inverse of %d:\n",n);
    for(int i=0;i<n;i++){
        for(int j=0;j<n;j++){
            if((i+j)%n==0){
                printf("(%d,%d)","i,j");
            }
        }
    }
    return 0;
}
```

OUTPUT:

```
Enter the value of n:5
Additive inverse of 5:
(0,0),(1,4),(2,3),(3,2),(4,1),
```

CONCLUSION:

Hence, additive inverse of an integer was found.

LAB- 4: EUCLIDEAN ALGORITHM

AIM:

To implement Euclidean algorithm to calculate GCD between numbers.

THEORY:

The greatest common divisor (GCD) of two numbers is the largest positive integer that divides both of them without leaving a remainder.

The Euclidean algorithm is a method for finding the greatest common divisor (GCD) of two integers. This is a recursive algorithm that involves repeatedly finding the remainder of the larger number divided by the smaller number until the remainder is zero. The GCD is then the last non-zero remainder.

$$\text{GCD}(a,b) = \text{GCD}(b, a \bmod b)$$

Euclidean Algorithm to compute GCD(a,b) is:

```
EUCLID(a,b)
  A = a; B = b
  if B = 0 return A = gcd(a, b)
  R = A mod B
  A = B
  B = R
  goto 2
```

THEORY:

```
#include <stdio.h>
#include <stdlib.h>
void GCD(int a,int b);
int main(){
    int m,n;
    printf("Enter two numbers to find their GCD:");
    scanf("%d %d",&m,&n);
    GCD(m,n);
}
void GCD(int a,int b){
    int r1=a,r2=b,q,r;
    while(r2>0){
        q=r1/r2;
        r=r1-q*r2;
        r1=r2;
        r2=r;
    }
    printf("GCD(%d,%d) = %d",a,b,r1);
}
```

OUTPUT:

```
Enter two numbers to find their GCD:4 7
GCD(4,7) = 1|
```


LAB- 5: EXTENDED EUCLIDEAN ALGORITHM

AIM:

To implement Extended Euclidean Algorithm.

THEORY:

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

EXTENDED EUCLID(m, b)

(A1, A2, A3)=(1, 0, m);

(B1, B2, B3)=(0, 1, b);

if B3 = 0

return A3 = gcd(m, b); no inverse

if B3 = 1

return B3 = gcd(m, b); B2 = b-1 mod m

Q = A3 div B3

(T1, T2, T3)=(A1 - Q B1, A2 - Q B2, A3 - Q B3)

(A1, A2, A3)=(B1, B2, B3)

(B1, B2, B3)=(T1, T2, T3)

goto 2

SOURCE CODE:

```
#include <stdio.h>
#include <stdlib.h>
void EEA(int n,int b);
int main(){
    int x,y;
    printf("Enter value of n:"); scanf("%d",&y); printf("Enter a value:"); scanf("%d",&x); EEA(y,x);
}
void EEA(int n,int b){
    int r1=n,r2=b,q,r,t1=0,t2=1,t; while(r2>0){
        q=r1/r2; r=r1-q*r2; r1=r2; r2=r;
        t=t1-q*t2; t1=t2; t2=t;
    }

    if(r1==1){
        if(t1<0){
            t1=t1%26;
            if(t1<0)
                t1=t1+n;
        }
        else{
            t1=t1%n;
            printf("\nThe multiplicative inverse of %d in Z%d is %d.",b,n,t1);
        }
    }
    else
        printf("\n\nThere is no multiplicative inverse");
}
```

Enter value of n:7
Enter a value:3
The multiplicative inverse of 3 in Z7 is 5.

CONCLUSION:

Hence, Extended Euclidean Algorithm was implemented.

LAB- 6: RELATIVELY PRIME

AIM:

To find if two integers are relatively prime.

THEORY:

Two integers a and b are said to be relatively prime (or coprime) if their greatest common divisor (GCD) is 1. In other words, they have no common factors other than 1.

$$\text{GCD}(a,b)=1$$

SOURCE CODE:

```
#include <stdio.h>

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

int areRelativelyPrime(int num1, int num2) {
    return gcd(num1, num2) == 1;
}

int main() {
    int num1, num2;
    printf("Enter the first integer: ");
    scanf("%d", &num1);

    printf("Enter the second integer: ");
    scanf("%d", &num2);
    if (areRelativelyPrime(num1, num2)) {
        printf("%d and %d are relatively prime.\n", num1, num2);
    } else {
        printf("%d and %d are not relatively prime.\n", num1, num2);
    }

    return 0;
}
```

OUTPUT:

```
Enter the first integer: 3
Enter the second integer: 7
3 and 7 are relatively prime.
```

LAB- 7: MILLER RABIN PRIMALITY TEST

AIM:

To implement Miller Rabin primality test algorithm to test if the number is prime or not.

THEORY:

The Miller-Rabin primality test is a probabilistic algorithm for testing whether a given number n is prime. The algorithm works by repeatedly selecting random witnesses a , computing a sequence of numbers based on a , and checking whether certain properties hold for the sequence. The Miller-Rabin test has two inputs: the number n to be tested and a parameter k that determines the accuracy of the test.

Steps in Miller Rabin test:

- i. Write $n-1$ as $2^r \times d$, where r is the largest integer such that 2^r divides $n-1$.
- ii. Choose a random integer a between 2 and $n-2$.
- iii. Compute $x_0 = a^d \bmod n$.
- iv. For $i=0$ to $r-1$, compute $x_{i+1} = x_i^2 \bmod n$. If $x_i = 1$ and $x_{i-1} \neq 1$ and $x_{i-1} \neq n-1$, then n is composite and we can return “not prime”
- v. If $x_r \neq 1$, then n is composite and we can return “not prime”
- vi. Repeat steps 2-5 k times. If n has passed all k tests, then n is probably prime.

SOURCE CODE:

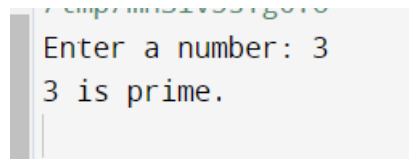
```
#include <iostream>
#include <cmath>
int modularExponentiation(int a, int b, int c) {
    int result = 1;
    a = a % c;
    while (b > 0) {
        if (b & 1)
            result = (result * a) % c;
        b = b >> 1;
        a = (a * a) % c;
    }
    return result;
}
bool isPrime(int n, int k = 4) {
    if (n <= 1 || n == 4)
        return false;
    if (n <= 3)
        return true;
    int d = n - 1;
    int r = 0;
    while (d % 2 == 0) {
        d = d / 2;
        r++;
    }
    for (int i = 0; i < k; i++){
        int a = 2 + rand() % (n - 4);
        int x = modularExponentiation(a, d, n);
        if (x == 1 || x == n - 1)
```

```

        continue;
    for (int j = 0; j < r - 1; j++) {
        x = modularExponentiation(x, 2, n);
        if (x == 1)
            return false;
        if (x == n - 1)
            break;
    }
    if (x != n - 1)
        return false;
}
return true;
}
int main() {
    int n;
    std::cout << "Enter a number: ";
    std::cin >> n;
    if (isPrime(n))
        std::cout << n << " is prime." << std::endl;
    else
        std::cout << n << " is not prime." << std::endl;
    return 0;
}

```

OUTPUT:



```

Enter a number: 3
3 is prime.

```

CONCLUSION:

Hence, miller rabin primality test was implemented.

LAB- 8: EULER TOTIENT FUNCTION

AIM:

To compute Euler Totient Function.

THEORY:

The Euler totient function, denoted as $\phi(n)$, is a mathematical function that counts the positive integers up to a given integer n that are relatively prime to n . In other words, it gives the number of integers between 1 and n (inclusive) that have no common factors with n except 1.

To compute $\phi(n)$ we need to count number of residues to be excluded.

In general we need prime factorization, but

- for p (p prime) $\phi(p) = p-1$
- for $p.q$ (p, q prime) $\phi(pq) = (p-1) \times (q-1)$

SOURCE CODE:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
int GCD(int a,int b);
int main(){
    int n,x=0;
    printf("Enter a number:"); scanf("%d",&n);
    printf("\n");
    for(int i=1;i<n;i++){
        if(GCD(i,n)==1){ x++;
        }
    }
    printf("The Euler totient of function phi(%d) is: %d",n,x);
}
int GCD(int a,int b){
    int r1=a,r2=b,q,r; while(r2>0){
        q=r1/r2; r=r1-q*r2; r1=r2; r2=r;
    }
    return r1;
}
```

OUTPUT:

```
Enter a number:7
The Euler totient of function phi(7) is: 6
```

LAB- 9: PRIMITIVE ROOT

AIM:

To find primitive root of a number.

THEORY:

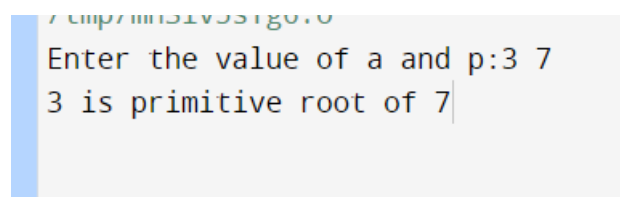
A primitive root of a prime number p is an integer g such that the powers of g modulo p generate all the integers relatively prime to p . In other words, g is a primitive root if every number in the set $\{1, 2, \dots, p-1\}$ can be written as $g^k \bmod p$ for some integer k .

Primitive roots are important in number theory and cryptography because they allow for efficient computations of discrete logarithms, which are used in many cryptographic protocols.

SOURCE CODE:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
int main(){
    int a,p,x[100],j=0;
    printf("Enter the value of a and p:"); scanf("%d %d",&a,&p);
    for(int i=1;i<p;i++){
        x[j]=fmod(pow(a,i),p); j++;
    }
    for(int m=0;m<j;m++){
        for(int n=m+1;n<j;n++){ if(x[m]==x[n]){
            printf("\n%d is not primitive root of %d",a,p); exit(0);
        }
    }
    printf("\n%d is primitive root of %d",a,p);
}
```

OUTPUT:



```
7 tmp/mms1v551go.0
Enter the value of a and p:3 7
3 is primitive root of 7
```

CONCLUSION:

Hence, primitive root was found.

LAB- 10: DIFFIE HELLMAN ALGORITHM

AIM:

To implement Deffe Hellman key exchange algorithm.

THEORY:

The Diffie-Hellman key exchange algorithm is a method for two parties to agree on a shared secret key over an insecure communication channel. The algorithm was proposed by Whitfield Diffie and Martin Hellman in 1976 and is widely used in modern cryptography.

The algorithm works as follows:

1. **Initialization:** The two parties, Alice and Bob, agree on a large prime number p and a primitive root g modulo p . These values are made public, but they do not need to be kept secret.
2. **Private key generation:** Each party generates a private key, which is a random number between 1 and $p-1$. Let's call Alice's private key a , and Bob's private key b .
3. **Public key generation:** Each party computes a public key, which is a value that can be shared with the other party. Alice computes $A = g^a \text{ mod } p$, and Bob computes $B = g^b \text{ mod } p$.
4. **Key exchange:** Alice sends A to Bob, and Bob sends B to Alice. Both A and B are exchanged over the insecure communication channel.
5. **Secret key computation:** Each party computes a shared secret key using the public key received from the other party and their own private key. Alice computes $s = B^a \text{ mod } p$, and Bob computes $s = A^b \text{ mod } p$. The result is the same for both parties and is the shared secret key that they will use for secure communication.

SOURCE CODE:

```
#include<stdio.h>
long long int power(int a, int b, int mod){
    long long int t;
    if(b==1)
        return a;
    t=power(a,b/2, mod);
    if(b%2==0)
        return (t*t)%mod;
    else
        return (((t*t) %mod) *a) %mod;
}
long int calculateKey (int a, int x, int n){
    return power(a,x,n);
}
int main(){
    int en,g,x,a,y,b;
    printf ("Enter the value of n and g : ");
    scanf ("%d %d", &en, &g);
    printf("Enter the value of x for the first person : ");
    scanf ("%d",&x);
    a=power(g,x,en);
    printf("Enter the value of y for the second person : ");
    scanf ("%d", &y);
```

```
b=power(g,y,en);  
printf("key for the first person is :%lld\n",power(b,x,en));  
printf("key for the second person is : %lld\n",power(a,y,en));  
}
```

OUTPUT:

```
Enter the value of n and g : 12  
63  
Enter the value of x for the first person : 21  
Enter the value of y for the second person : 73  
key for the first person is :3  
key for the second person is : 3  
|
```

CONCLUSION:

Hence, Deffe Hellman was implemented.

LAB- 11: RSA ALGORITHM

THEORY:

RSA is a public-key cryptographic algorithm used for secure communication over insecure networks. The RSA algorithm relies on the use of a public key and a private key. The public key is used for encryption, while the private key is used for decryption. The security of the algorithm is based on the fact that it is computationally infeasible to factor large prime numbers.

Working of RSA algorithm:

1. Key Generation:

- Choose two large prime numbers, p and q .
- Calculate $n = p * q$. This is the modulus for the public and private keys.
- Calculate the totient of n , $\phi(n) = (p-1) * (q-1)$.
- Choose an integer e such that $1 < e < \phi(n)$ and e is coprime to $\phi(n)$. e is the public key exponent.
- Calculate the private key exponent d such that $(d * e) \bmod \phi(n) = 1$.

2. Encryption:

- Convert the plaintext message to a number m , such that $0 \leq m < n$.
- Compute the ciphertext c as $c = m^e \bmod n$. The ciphertext is the encrypted message.

- Decryption:** Given the ciphertext c , compute the plaintext m as $m = c^d \bmod n$. The plaintext is the decrypted message.

SOURCE CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int GCD(int e, int phi);
int main(){
    int p,q,n,phi;
    int d=2,e=2,m;
    double c,m1;
    printf("Enter 1st prime number: ");
    scanf("%d",&p);
    printf("Enter 2nd prime number: ");
    scanf("%d",&q);
    printf("Enter value of m: ");
    scanf("%d",&m);
    n=p*q;
    phi=(p-1)*(q-1);
    printf("\nValue of n is: %d",n);
    printf("\nValue of phi is: %d\n",phi);
    while(GCD(e,phi)!=1){
        e++;
        if(GCD(e,phi)==1){
            while((d*e)%phi!=1){
                d++;
            }
        }
    }
}
```

```

printf("\nValue of e=%d",e);
printf("\nValue of d=%d\n",d);
printf("\nEncryption");
c=fmod(pow(m,e),n);
printf("\nValue of c=%lf",c);
printf("\n\nDecryption");
m1=fmod(pow(c,d),n);
printf("\nValue of m=%lf",m1);
}
int GCD(int e,int phi) {
    int i, gcd;
    for(i=1; i <= e && i <= phi; ++i){
        if(e%i==0 && phi%i==0)
            gcd = i;
    }
    return gcd;
}

```

OUTPUT:

```

Enter 1st prime number: 67
Enter 2nd prime number: 11
Enter value of m: 54
Value of n is: 737
Value of phi is: 660

Value of e=7
Value of d=283

Encryption
Value of c=65.000000

Decryption
Value of m=-nan

```

CONCLUSION:

Hence, RSA algorithm was implemented.

