EXPERIMENT - 1

AIM: Write a C program that contains a string (char pointer) with a value ‘Hello World’. The program should XOR each character in this string with 0 and display the result.

DESCRIPTION:
In this program, we are trying to XOR each character in the given string with 0. XOR is a bitwise operator, and it stands for "exclusive or." It performs logical operations. If input bits are the same, then the output will be false(0) else true(1).

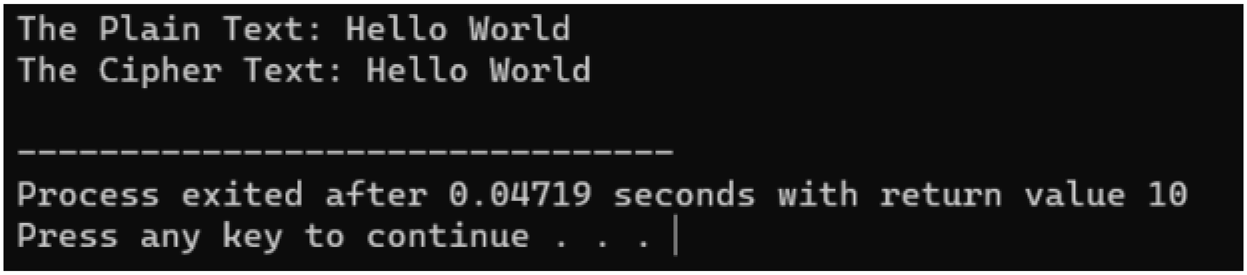
ALGORITHM:
Step-1: Start
Step-2: Initialize two character arrays: ‘str’ for the plain text and ‘str1’ for the cipher text.
Step-3: Calculate the length of the plain text using the ‘strlen’ function.
Step-4: Iterate through each character of the plain text using a loop.
Step-4.1: Print each character of the plain text to display the original message.
Step-5: After printing the original message, iterate through each character of the plain text again.
Step-5.1: Apply a bitwise XOR operation (^) with 0 to each character and store the result in the corresponding position of the ‘str1’ array.
Step-5.2: Print each character of the cipher text to display the encrypted message.
Step-6: End

PROGRAM:

```
#include<stdlib.h>
int main()
{
char str[]="Hello World";
char str1[11];
int i,len;
len=strlen(str);
printf("The Plain Text: ");
for(i=0;i<len;i++)
{
printf("%c", str[i]);
}
printf("\nThe Cipher Text: ");
for(i=0;i<len;i++)
{
```

```
str1[i]=str[i]^0;
printf("%c",str1[i]);
}
printf("\n");
}
```

OUTPUT:



```
The Plain Text: Hello World
The Cipher Text: Hello World

--------------------------------
Process exited after 0.04719 seconds with return value 10
Press any key to continue . . .
```

CONCLUSION:

By executing the above, we have successfully performed XOR operation of the given string with 0 and displayed the result.

EXPERIMENT - 2

AIM: Write a C program that contains a string (char pointer) with a value ‘Hello World’. The program should AND, OR, and XOR each character in this string with 127 and displays the result

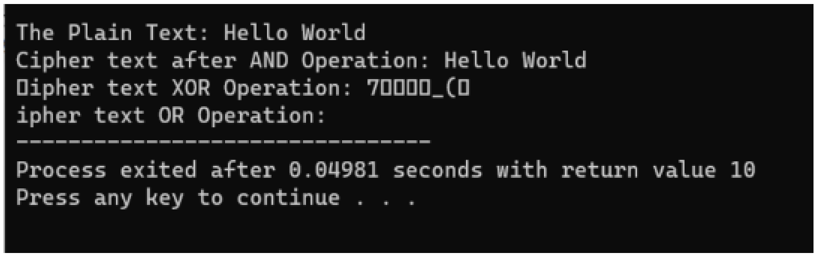
DESCRIPTION: In this program, we are trying to AND, OR and XOR. Each character in the given string with 127. AND operator is represented as the '&&' double ampersand symbol. It is a logical operator. It checks the condition of two or more operands by combining in an expression, and if all the conditions are true, the logical AND operator returns the Boolean value true or 1. Else it returns false or 0.
The logical OR operator (||) returns the boolean value true if either or both XOR is a bitwise operator, and it stands for "exclusive or." It performs logical operation. If input bits are the same, then the output will be
false(0) else true(1).

ALGORITHM:
Step-1: Start
Step-2: Initialize a character array ‘str’ with the text "Hello World" and three additional character arrays ‘str1’, ‘str2’, and ‘str3’ to store the results of the bitwise operations. Step-3: Calculate the length of the input string using the ‘strlen’ function.
Step-4: Iterate through each character of the input string and perform the following operations:
Step-4.1: AND Operation:
Step-4.1.1: Apply a bitwise AND operation (&) with the character and 127 (binary: 01111111) and store the result in the corresponding position of the ‘str1’ array.
Step-4.1.2: Print each character of the result to display the cipher text after the AND operation.
Step-4.2: XOR Operation:
Step-4.2.1: Apply a bitwise XOR operation (^) with the
character and 127 and store the result in the corresponding position of
the ‘str3’ array.
Step-4.2.2: Print each character of the result to display the
cipher text after the XOR operation.
Step-4.3: OR Operation:
Step-4.3.1: Apply a bitwise OR operation (|) with the character
and 127 and store the result in the corresponding position of the
‘str2’ array.
Step-4.3.2: Print each character of the result to display the
cipher text after the OR operation.
Step-5: End

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
void main()
{ char str[]="Hello World";
char str1[11],str2[11], str3[11];
int i,len;
len = strlen(str);
printf("The Plain Text: ");
for (i=0;i<len;i++)
{ printf("%c", str[i]);
}
printf("\nCipher text after AND Operation: ");
for(i=0;i<len;i++)
{ str1[i] = str[i]&127;
printf("%c",str1[i]);
}
printf("\nCipher text XOR Operation: ");
for(i=0;i<len;i++)
{ str3[i] = str[i]^127;
printf("%c",str3[i]);
}
printf("\nCipher text OR Operation: ");
for(i=0;i<len;i++)
{
str2[i] = str[i]|127;
printf("%c",str2[i]);
}
printf("\n");
}
```

OUTPUT:



```
The Plain Text: Hello World
Cipher text after AND Operation: Hello World
□ipher text XOR Operation: 7□□□□_(□
ipher text OR Operation:
--------------------------------
Process exited after 0.04981 seconds with return value 10
Press any key to continue . . .
```

CONCLUSION:

By executing the above, we have successfully performed AND, OR and XOR operation of the given string with 127 and displayed the result.

EXPERIMENT - 3.1

AIM: Write a C program to perform encryption and decryption using Ceaser Cipher

DESCRIPTION: In this program, we are trying to perform encryption and decryption of the given string using Ceaser Cipher algorithm. 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.

ALGORITHM:

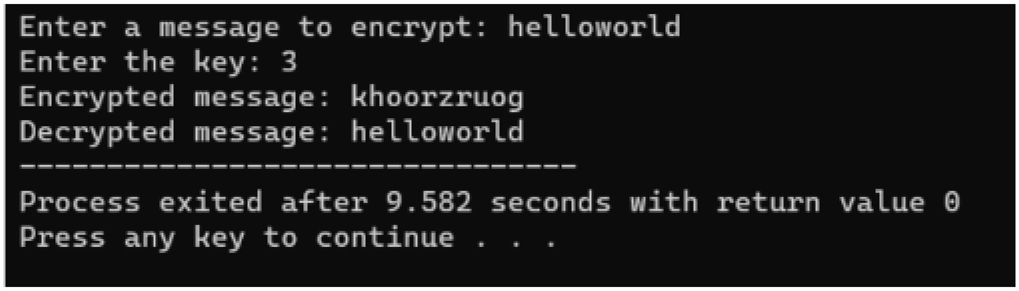
```
Step-1: Start
Step-2: Initialize a character array ‘text’ to store the input message and variables ‘ch’ and ‘key’.
Step-3: Prompt the user to enter a message to encrypt and the key.
Step-4: Encrypt the message:
Step-4.1: Iterate through each character of the input message.
Step-4.2: Check if the character is alphanumeric using isalnum function.
Step-4.3: If alphanumeric, apply the following transformations:
Step-4.3.1: For lowercase letters, shift the character by the key value using modulo arithmetic to ensure looping within the alphabet range.
Step-4.3.2: For uppercase letters, perform the same shift operation.
Step-4.3.3: For digits, shift the character by the key value within the digit range (0-9).
Step-4.4: Update the text array with the encrypted character.
Step-4.5: If the character is not alphanumeric, print "Invalid Message."
Step-5: Print the encrypted message.
Step-6: Decrypt the message:
Step-6.1: Iterate through each character of the encrypted message.
Step-6.2: Check if the character is alphanumeric using isalnum function.
Step-6.3: If alphanumeric, perform the reverse shift operation to decrypt the character.
Step-6.4: Update the text array with the decrypted character.
Step-6.5: If the character is not alphanumeric, print "Invalid Message."
Step-7: Print the decrypted message.
Step-8: End
```

PROGRAM:

```
#include<stdio.h>
#include<ctype.h>
int main()
{
```

```
char text[500], ch;
int i;
int key;
printf("Enter a message to encrypt: ");
scanf("%s", text);
printf("Enter the key: ");
scanf("%d", & key);
for (i = 0; text[i] != '\0'; ++i)
{
ch = text[i];
if (isalnum(ch)) {
if (islower(ch)) {
ch = (ch - 'a' + key) % 26 + 'a';
}
if (isupper(ch)) {
ch = (ch - 'A' + key) % 26 + 'A';
}
if (isdigit(ch)) {
ch = (ch - '0' + key) % 10 + '0';
}
}
else {
printf("Invalid Message");
}
text[i] = ch;
}
printf("Encrypted message: %s\n", text);
for (i = 0; text[i] != '\0'; ++i) {
ch = text[i];
if (isalnum(ch)) {
if (islower(ch)) {
ch = (ch - 'a' - key) % 26 + 'a';
}
if (isupper(ch)) {
ch = (ch - 'A' - key) % 26 + 'A';
}
if (isdigit(ch)) {
ch = (ch - '0' - key) % 10 + '0';
}
}
else {
printf("Invalid Message");
}
text[i] = ch;
}
printf("Decrypted message: %s", text);
return 0;
}
```

OUTPUT:



```
Enter a message to encrypt: helloworld
Enter the key: 3
Encrypted message: khoorzruog
Decrypted message: helloworld
--------------------------------
Process exited after 9.582 seconds with return value 0
Press any key to continue . . .
```

CONCLUSION:

By executing the above, we have successfully performed encryption and decryption for the given string using Ceaser Cipher.

EXPERIMENT 3.2:

AIM: Write a C program to perform encryption and decryption using Substitution Cipher

DESCRIPTION: In this program, we are trying to perform encryption and decryption of the given string using the Substitution Cipher algorithm. In a Substitution cipher, any character of plain text from the given fixed set of characters is substituted by some other character from the same set depending on a key.

ALGORITHM:

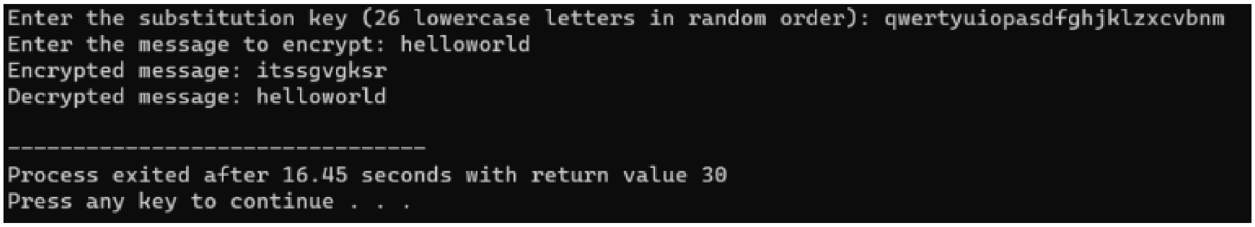
```
Step-1: Start
Step-2: Define functions encrypt and decrypt for encryption and decryption processes.
Step-3: In the encrypt function:
Step-3.1: Iterate through each character of the input message.
Step-3.2: If the character is a lowercase letter, replace it with the corresponding character from the provided key.
Step-4: In the decrypt function:
Step-4.1: Iterate through each character of the input message.
Step-4.2: For each character, iterate through the key to find the matching character and replace it with the corresponding lowercase letter.
Step-5: In the main function:
Step-5.1: Prompt the user to enter a substitution key consisting of 26 lowercase letters.
Step-5.2: Check the length and validity of the key.
Step-5.3: Prompt the user to enter the message to encrypt.
Step-5.4: Call the encrypt function to encrypt the message using the provided key.
Step-5.5: Print the encrypted message.
Step-5.6: Call the decrypt function to decrypt the message using the same key.
Step-5.7: Print the decrypted message.
Step-6: End
```

PROGRAM:

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

```
void encrypt(char *message, char *key)
{
int i;
for (i = 0; i < strlen(message); i++)
{
if (message[i] >= 'a' && message[i] <= 'z')
{
message[i] = key[message[i] - 'a'];
}
}
}
void decrypt(char *message, char *key)
{
int i,j;
for (i = 0; i < strlen(message); i++)
{
for (j = 0; j < 26; j++)
{
if (message[i] == key[j])
{
message[i] = 'a' + j;
break;
}
}
}
}
int main()
{
char key[26];
int i;
printf("Enter the substitution key (26 lowercase letters in random
order): ");
scanf("%s", key);
if (strlen(key) != 26)
{
printf("Invalid key length. Please provide 26 letters.\n");
return 1;
}
for (i = 0; i < 26; i++)
{
if (key[i] < 'a' || key[i] > 'z')
{
printf("Invalid key. Please provide only lowercase letters.\n");
return 1;
}
}
char message[100];
printf("Enter the message to encrypt: ");
scanf(" %[^\n]s", message);
encrypt(message, key);
printf("Encrypted message: %s\n", message);
decrypt(message, key);
printf("Decrypted message: %s\n", message);
}
```

OUTPUT:



```
Enter the substitution key (26 lowercase letters in random order): qwertyuiopasdfghjklzxcvbnm
Enter the message to encrypt: helloworld
Encrypted message: itssgvgksr
Decrypted message: helloworld

--------------------------------
Process exited after 16.45 seconds with return value 30
Press any key to continue . . .
```

CONCLUSION:

By executing the above, we have successfully performed encryption And decryption for the given string using Substitution Cipher.

EXPERIMENT 3.3:

AIM: Write a C program to perform encryption and decryption using Hill Cipher.

DESCRIPTION: In this program, we perform encryption and decryption using the Hill Cipher algorithm. Hill Cipher is a polygraphic substitution cipher based on linear algebra. Each letter is represented by a number (e.g., A = 0, B = 1, ..., Z = 25). The message is broken into chunks of size equal to the key matrix, and each chunk is multiplied by the key matrix to produce the cipher text. Decryption involves multiplying the ciphertext by the inverse of the key matrix.

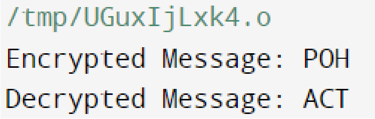
ALGORITHM:

```
Step-1: Start
Step-2: Define functions encrypt and decrypt for encryption and decryption processes.
Step-3: In the encrypt function:
Step-3.1: Break the message into vectors based on the size of the key matrix.
Step-3.2: Multiply each vector by the key matrix to produce encrypted vectors.
Step-3.3: Convert the resulting vectors back into characters to form the encrypted message.
Step-4: In the decrypt function:
Step-4.1: Multiply the encrypted vectors by the inverse of the key matrix.
Step-4.2: Convert the resulting vectors back into characters to obtain the decrypted message.
Step-5: In the main function:
Step-5.1: Prompt the user to enter a key matrix and a message to encrypt.
Step-5.2: Ensure the key matrix is valid and invertible.
Step-5.3: Call the encrypt function to encrypt the message.
Step-5.4: Call the decrypt function to decrypt the message.
Step-5.5: Print the encrypted and decrypted messages.
Step-6: End
```

PROGRAM:

```c
#include <stdio.h>
#include <string.h>
#define N 3
void encrypt(char key[], char item[], int keyMatrix[N][N], int itemMatrix[], int answer[]) {
   int i, j, k = 0, p;
   for (i = 0; i < N; i++) {
      itemMatrix[i] = item[i] - 'A';
   }
   k = 0;
   for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            keyMatrix[i][j] = key[k] - 'A';
            k++;
        }
    }
    for (i = 0; i < N; i++) {
        p = 0;
        for (j = 0; j < N; j++) {
            p += keyMatrix[i][j] * itemMatrix[j];
        }
        answer[i] = p % 26;
    }
}
void decrypt(int inverseKeyMatrix[N][N], int encryptedMatrix[], int decryptedMatrix[]) {
    int i, j, p;
    for (i = 0; i < N; i++) {
        p = 0;
        for (j = 0; j < N; j++) {
            p += inverseKeyMatrix[i][j] * encryptedMatrix[j];
        }
        decryptedMatrix[i] = p % 26;
        if (decryptedMatrix[i] < 0)
            decryptedMatrix[i] += 26;
    }
}
int main() {
    char key[] = "GYBNQKURP";
    char item[] = "ACT";
    int keyMatrix[N][N];
    int itemMatrix[N];
    int encryptedMatrix[N];
    char encryptedMessage[N + 1];
    encrypt(key, item, keyMatrix, itemMatrix, encryptedMatrix);
    for (int i = 0; i < N; i++) {
        encryptedMessage[i] = encryptedMatrix[i] + 'A';
    }
    encryptedMessage[N] = '\0';
    printf("Encrypted Message: %s\n", encryptedMessage);
    int inverseKeyMatrix[N][N] = {
        {8, 5, 10},
        {21, 8, 21},
        {21, 12, 8}
    };
    int decryptedMatrix[N];
    char decryptedMessage[N + 1];
    decrypt(inverseKeyMatrix, encryptedMatrix, decryptedMatrix);
    for (int i = 0; i < N; i++) {
        decryptedMessage[i] = decryptedMatrix[i] + 'A';
    }
    decryptedMessage[N] = '\0';
    printf("Decrypted Message: %s\n", decryptedMessage);
    return 0;
}
```

OUTPUT:



```
/tmp/UGuxIjLxk4.o
Encrypted Message: POH
Decrypted Message: ACT
```

CONCLUSION:

By executing the above program, we have successfully performed encryption and decryption for the given string using Hill Cipher.

EXPERIMENT 3.4:

AIM: Write a C program to perform encryption and decryption using Playfair Cipher.

DESCRIPTION: In this program, we perform encryption and decryption using the Playfair Cipher. The Playfair Cipher encrypts pairs of letters using a 5x5 matrix of letters, based on a keyword. Each letter is substituted by letters in the same row or column or the opposite corners of a rectangle. This demonstrates the functionality and effectiveness of the Playfair Cipher for secure communication, particularly highlighting its ability to handle digraphs and avoid frequency analysis attacks typical of simpler ciphers.

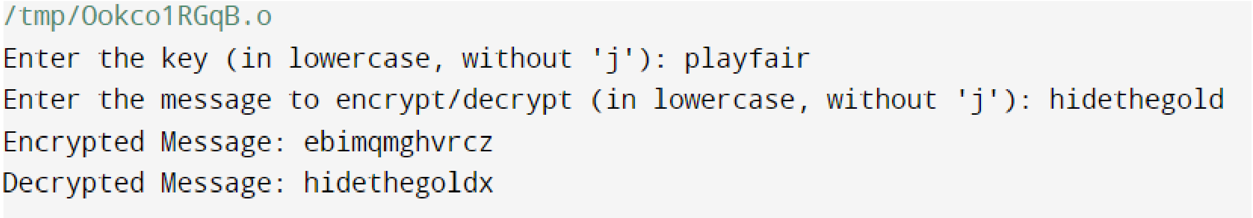
ALGORITHM:

```
Step-1: Start
Step-2: Create a 5x5 key matrix using a keyword.
Step-3: Preprocess the input message to handle pairs of letters, adding a filler letter if needed.
Step-4: For encryption:
Step-4.1: Find each pair of letters in the key matrix.
Step-4.2: Replace the pair according to the Playfair rules (same row, same column, rectangle).
Step-5: For decryption:
Step-5.1: Reverse the process to retrieve the original message.
Step-6: In the main function:
Step-6.1: Prompt the user to enter the keyword and message.
Step-6.2: Call the encrypt function to encrypt the message.
Step-6.3: Call the decrypt function to decrypt the message.
Step-6.4: Print the encrypted and decrypted messages.
Step-7: End
```

PROGRAM:

```c
#include <stdio.h>
#include <string.h>
#define SIZE 5
void generateKeyTable(char key[], char keyTable[SIZE][SIZE]) {
    int dict[26] = {0};
    int i, j, k = 0, l = 0;
    for (i = 0; i < strlen(key); i++) {
        if (key[i] != 'j' && !dict[key[i] - 'a']) {
            keyTable[k][l] = key[i];
            dict[key[i] - 'a'] = 1;
            l++;
            if (l == SIZE) {
                k++;
                l = 0;
            }
        }
    }
    for (i = 0; i < 26; i++) {
        if (!dict[i] && (i != 9)) {
            keyTable[k][l] = 'a' + i;
            l++;
            if (l == SIZE) {
                k++;
                l = 0;
            }
        }
    }
}
void search(char keyTable[SIZE][SIZE], char a, char b, int pos[]) {
    int i, j;
    if (a == 'j') a = 'i';
    if (b == 'j') b = 'i';
    for (i = 0; i < SIZE; i++) {
        for (j = 0; j < SIZE; j++) {
            if (keyTable[i][j] == a) {
                pos[0] = i;
                pos[1] = j;
            } else if (keyTable[i][j] == b) {
                pos[2] = i;
                pos[3] = j;
            }
        }
    }
}
char* encrypt(char str[], char keyTable[SIZE][SIZE]) {
    int i, a[4];
    static char encrypted[100];
    strcpy(encrypted, str);
    for (i = 0; i < strlen(encrypted); i += 2) {
        search(keyTable, encrypted[i], encrypted[i + 1], a);
        if (a[0] == a[2]) {
            encrypted[i] = keyTable[a[0]][(a[1] + 1) % SIZE];
            encrypted[i + 1] = keyTable[a[2]][(a[3] + 1) % SIZE];
        } else if (a[1] == a[3]) {
            encrypted[i] = keyTable[(a[0] + 1) % SIZE][a[1]];
            encrypted[i + 1] = keyTable[(a[2] + 1) % SIZE][a[3]];
        } else {
            encrypted[i] = keyTable[a[0]][a[3]];
            encrypted[i + 1] = keyTable[a[2]][a[1]];
        }
    }
    return encrypted;
}
void decrypt(char str[], char keyTable[SIZE][SIZE]) {
    int i, a[4];
    for (i = 0; i < strlen(str); i += 2) {
        search(keyTable, str[i], str[i + 1], a);
        if (a[0] == a[2]) {
            str[i] = keyTable[a[0]][(a[1] + SIZE - 1) % SIZE];
            str[i + 1] = keyTable[a[2]][(a[3] + SIZE - 1) % SIZE];
        } else if (a[1] == a[3]) {
            str[i] = keyTable[(a[0] + SIZE - 1) % SIZE][a[1]];
            str[i + 1] = keyTable[(a[2] + SIZE - 1) % SIZE][a[3]];
        } else {
            str[i] = keyTable[a[0]][a[3]];
            str[i + 1] = keyTable[a[2]][a[1]];
        }
    }
}
int main() {
    char key[SIZE * SIZE], str[100], keyTable[SIZE][SIZE];
    char *encryptedMessage;
    int i;
    printf("Enter the key (in lowercase, without 'j'): ");
    scanf("%s", key);
    generateKeyTable(key, keyTable);
    printf("Enter the message to encrypt/decrypt (in lowercase, without 'j'): ");
    scanf("%s", str);
    if (strlen(str) % 2 != 0) {
        strcat(str, "x");
    }
    encryptedMessage = encrypt(str, keyTable);
    printf("Encrypted Message: %s\n", encryptedMessage);
    decrypt(encryptedMessage, keyTable);
    printf("Decrypted Message: %s\n", encryptedMessage);
    return 0;
}
```

OUTPUT:



```
/tmp/Ookco1RGqB.o
Enter the key (in lowercase, without 'j'): playfair
Enter the message to encrypt/decrypt (in lowercase, without 'j'): hidethegold
Encrypted Message: ebimqmghvrcz
Decrypted Message: hidethegoldx
```

CONCLUSION:

By executing the above program, we have successfully implemented the Playfair Cipher algorithm in C. The program allows users to encrypt a plaintext message using a key and then decrypt the resulting ciphertext back into its original form.

EXPERIMENT 4:

AIM: To implement to perform Cryptography using Transposition Technique

DESCRIPTION:

In this program, we implement the Simple and Advanced Columnar Transposition techniques for encrypting a message. Columnar Transposition is a classical encryption technique where the plaintext is written row by row in a matrix of fixed dimensions (rows and columns) and then read out column by column according to a key. The key is a permutation of column indices, which determines the order in which the columns are read during encryption.

The **Simple Columnar Transposition** technique involves filling the matrix with the message row by row and then reading the message column by column according to the key to produce the encrypted message.

The **Advanced Columnar Transposition** technique applies multiple rounds of the Simple Columnar Transposition, further scrambling the message by reapplying the transposition process multiple times.

ALGORITHM:

Simple Columnar Transposition

1. **Input:** A message, a key (list of column indices), number of rows, and number of columns.
2. **Step 1:** Calculate the dimensions of the matrix based on the message length and number of columns.
3. **Step 2:** Pad the message with spaces to fit the matrix dimensions if necessary.
4. **Step 3:** Fill the matrix row by row with the characters of the padded message.
5. **Step 4:** Initialize an empty list for the encrypted message.
6. **Step 5:** For each column index in the key:
   - Read the characters down the corresponding column.
   - Append the characters to the encrypted message list.
7. **Step 6:** Return the encrypted message as a string.

Advanced Columnar Transposition

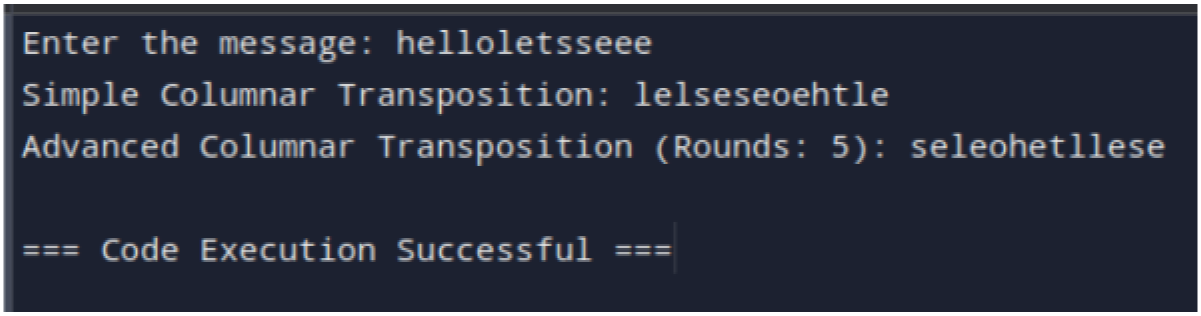
1. **Input:** A message, a key, number of rows, number of columns, and the number of rounds.

2. **Step 1:** Initialize the transposed message as the original message.
3. **Step 2:** For each round, perform the Simple Columnar Transposition on the current transposed message.
4. **Step 3:** After completing all rounds, return the final transposed (encrypted) message.

PROGRAM:

```
import random
import math

def simple_columnar_transposition(message, key, num_rows, num_cols):
    # Pad the message with spaces if needed
    padded_message = message.ljust(num_rows * num_cols)  # Pad the message to fit the matrix
    matrix = []
    x = 0

    # Fill the matrix row by row
    for i in range(num_rows):
    row = []
    for j in range(num_cols):
    row.append(padded_message[x])
    x += 1
    matrix.append(row)

    encrypted = []
    for i in key:
    for r in range(num_rows):
    if matrix[r][i] != " ":
        encrypted.append(matrix[r][i])  # No need for index correction since key starts from 0
    # print(matrix, encrypted)
    return ''.join(encrypted)

def advanced_columnar_transposition(message, key, num_rows, num_cols, rounds):
    transposed_message = message
    for _ in range(rounds):
    transposed_message = simple_columnar_transposition(transposed_message, key, num_rows, num_cols)
    return transposed_message

def generate_random_key(num_cols):
    key = list(range(num_cols))
    random.shuffle(key)
    return key

def main():
    # Input message
    message = input("Enter the message: ").replace(" ", "")

    # Generate random rows and columns
    num_cols = random.randint(2, 10)  # Random number of columns between 2 and 10
    num_rows = math.ceil(len(message) / num_cols)

    # Generate a random key
    key = generate_random_key(num_cols)

    # Simple Columnar Transposition
    encrypted_message = simple_columnar_transposition(message, key, num_rows,
num_cols)
    print(f"Simple Columnar Transposition: {encrypted_message}")

    # Advanced Columnar Transposition
    rounds = random.randint(1, 5)  # Random number of rounds between 1 and 5
    advanced_encrypted_message = advanced_columnar_transposition(message, key,
num_rows, num_cols, rounds)
    print(f"Advanced Columnar Transposition (Rounds: {rounds}):
{advanced_encrypted_message}")

if __name__ == "__main__":
    main()
```

OUTPUT:



```
Enter the message: helloletsseee
Simple Columnar Transposition: lelseseoehtle
Advanced Columnar Transposition (Rounds: 5): seleohetllese

=== Code Execution Successful ===
```

CONCLUSION:

By executing the above Python program, we successfully performed encryption using both Simple and Advanced Columnar Transposition techniques.

EXPERIMENT 5:

AIM: Implement the Diffie-Hellman Key Exchange mechanism

DESCRIPTION:

The Diffie-Hellman Key Exchange is a cryptographic protocol that enables two parties to securely establish a shared secret key over an insecure communication channel. Both parties agree on a large prime number and a primitive root modulo this prime. Each party independently selects a private key, computes a corresponding public key, and exchanges these public keys. Using the received public key and their own private key, each party computes the shared secret. Despite the insecure nature of the channel, both parties end up with the same shared secret, which can then be used for symmetric encryption, thanks to the mathematical properties of modular arithmetic.

ALGORITHM:

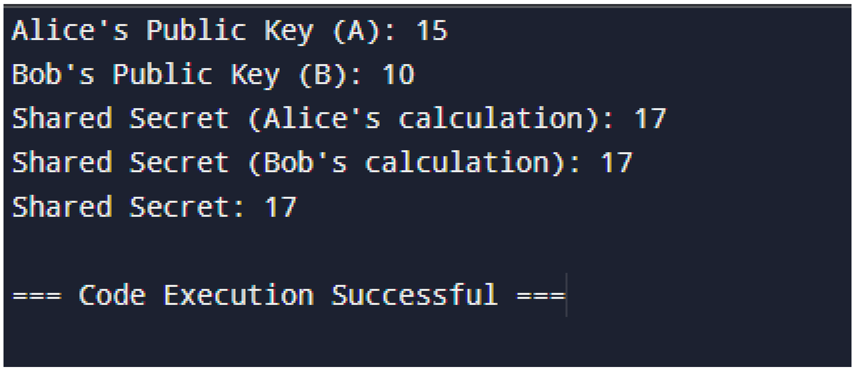
1. Choose a large prime number p.
2. Choose a primitive root g modulo p.
3. Alice selects a random private key a where 1≤ a ≤ p−1.
4. Alice computes her public key A = g^a mod p
5. Alice sends A to Bob.
6. Bob selects a random private key b where 1≤ b≤ p−1.
7. Bob computes his public key B = g^b mod p
8. Bob sends B to Alice.
9. Alice computes SA=B^a mod p
10. Bob computes SB=A^b mod p
11. Verify that SA=SB. This shared secret S is now used for symmetric encryption between Alice and Bob.

PROGRAM:

```
// Function to perform modular exponentiation
function modExp(base, exp, mod) {
   let 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 = Math.floor(exp / 2);
      base = (base * base) % mod;
   }
```

```
    return result;
}

// Diffie-Hellman parameters
const p = 23;  // Prime number
const g = 5;   // Primitive root modulo p

// Generate random private keys for Alice and Bob
const a = Math.floor(Math.random() * (p - 1)) + 1;
const b = Math.floor(Math.random() * (p - 1)) + 1;

// Compute public keys
const A = modExp(g, a, p);  // Alice's public key
const B = modExp(g, b, p);  // Bob's public key

// Compute shared secrets
const sharedSecretAlice = modExp(B, a, p);
const sharedSecretBob = modExp(A, b, p);

// Output results
console.log(`Alice's Public Key (A): ${A}`);
console.log(`Bob's Public Key (B): ${B}`);
console.log(`Shared Secret (Alice's calculation): ${sharedSecretAlice}`);
console.log(`Shared Secret (Bob's calculation): ${sharedSecretBob}`);

// Check if the shared secrets match
if (sharedSecretAlice === sharedSecretBob) {
    console.log(`Shared Secret: ${sharedSecretAlice}`);
} else {
    console.log('Error: Shared secrets do not match!');
}
```

OUTPUT:



```
Alice's Public Key (A): 15
Bob's Public Key (B): 10
Shared Secret (Alice's calculation): 17
Shared Secret (Bob's calculation): 17
Shared Secret: 17

=== Code Execution Successful ===
```

CONCLUSION:

By executing the above javascript program, we successfully performed Diffie-Hellman Key Exchange mechanism.

EXPERIMENT 6

AIM : Write a Java program to implement the DES algorithm logic.

DESCRIPTION :

The Java program implements Triple DES (3DES) encryption and decryption using the `javax.crypto` library. Triple DES, also known as DESede, enhances the security of the original DES algorithm by applying it three times with different keys. This makes it significantly more secure than DES alone.

ALGORITHM :

Step 1: The program initializes a 3DES key from a hardcoded string. This key is used for both encryption and decryption. The key must be 24 bytes long, which is ensured by the choice of the encryption key.

Step 2:

Encryption Process:

- Converts plaintext to a byte array.
- Encrypts the byte array using the 3DES algorithm.
- Encodes the encrypted byte array into a Base64 string for easy representation.

Step 3:

Decryption Process:

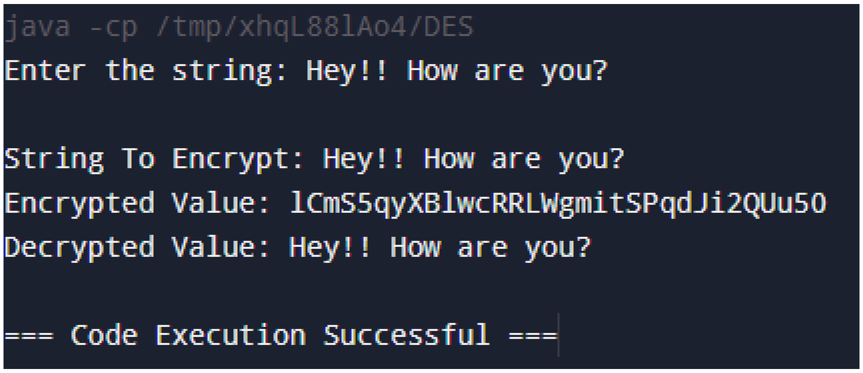
- Decodes the Base64-encoded string back to a byte array.
- Decrypts the byte array using the 3DES algorithm.
- Converts the decrypted byte array back into a string.

PROGRAM:

```
import java.util.*;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.security.spec.KeySpec;
import javax.crypto.Cipher;
import javax.crypto.SecretKey;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.DESedeKeySpec;
import java.util.Base64;
```

```
public class DES {
    private static final String UNICODE_FORMAT = "UTF8";
    public static final String DESEDE_ENCRYPTION_SCHEME = "DESede";
    private KeySpec myKeySpec;
    private SecretKeyFactory mySecretKeyFactory;
    private Cipher cipher;
    private SecretKey key;
    private String myEncryptionKey;
    private String myEncryptionScheme;
    static BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

    public DES() throws Exception {
        myEncryptionKey = "ThisIsSecretEncryptionKey"; // 24 bytes key for DESede
        myEncryptionScheme = DESEDE_ENCRYPTION_SCHEME;
        byte[] keyAsBytes = myEncryptionKey.getBytes(UNICODE_FORMAT);
        myKeySpec = new DESedeKeySpec(keyAsBytes);
        mySecretKeyFactory = SecretKeyFactory.getInstance(myEncryptionScheme);
        cipher = Cipher.getInstance(myEncryptionScheme);
        key = mySecretKeyFactory.generateSecret(myKeySpec);
    }

    public String encrypt(String unencryptedString) {
        String encryptedString = null;
        try {
            cipher.init(Cipher.ENCRYPT_MODE, key);
            byte[] plainText = unencryptedString.getBytes(UNICODE_FORMAT);
            byte[] encryptedText = cipher.doFinal(plainText);
            encryptedString = Base64.getEncoder().encodeToString(encryptedText);
        } catch (Exception e) {
            e.printStackTrace();
        }
        return encryptedString;
    }

    public String decrypt(String encryptedString) {
        String decryptedText = null;
        try {
            cipher.init(Cipher.DECRYPT_MODE, key);
            byte[] encryptedText = Base64.getDecoder().decode(encryptedString);
            byte[] plainText = cipher.doFinal(encryptedText);
            decryptedText = new String(plainText, UNICODE_FORMAT);
        } catch (Exception e) {
            e.printStackTrace();
        }
        return decryptedText;
    }

    public static void main(String args[]) throws Exception {
        System.out.print("Enter the string: ");
        DES myEncryptor = new DES();
        String stringToEncrypt = br.readLine();
        String encrypted = myEncryptor.encrypt(stringToEncrypt);
        String decrypted = myEncryptor.decrypt(encrypted);
        System.out.println("\nString To Encrypt: " + stringToEncrypt);
        System.out.println("Encrypted Value: " + encrypted);
        System.out.println("Decrypted Value: " + decrypted);
    }
}
```

OUTPUT:



```
java -cp /tmp/xhqL88lAo4/DES
Enter the string: Hey!! How are you?

String To Encrypt: Hey!! How are you?
Encrypted Value: lCmS5qyXBlwcRRLWgmitSPqdJi2QUu5O
Decrypted Value: Hey!! How are you?

=== Code Execution Successful ===
```

CONCLUSION:

This Java program showcases Triple DES (3DES) encryption using the `javax.crypto` library. It encrypts and decrypts user input with a 3DES key, demonstrating symmetric encryption. Base64 encoding is used to convert encrypted data into a readable string format, making it easy to store and transmit. This example highlights the implementation of cryptographic algorithms in Java for securing data.

EXPERIMENT - 7

AIM:- To implement RSA algorithm

DESCRIPTION:-

RSA (Rivest-Shamir-Adleman) is an asymmetric cryptographic algorithm used for secure data transmission. It relies on the mathematical properties of prime numbers and modular arithmetic.

Key Generation

1. **Select Two Primes**: Choose two distinct large prime numbers, ppp and qqq.
2. **Compute Modulus**: N=p×qN = p \times qN=p×q, used in both public and private keys.
3. **Compute Totient**: ϕ(N)=(p−1)×(q−1)\phi(N) = (p - 1) \times (q - 1)ϕ(N)=(p−1)×(q−1).
4. **Choose Public Exponent**: Select eee such that 1<e<ϕ(N)1 < e < \phi(N)1<e<ϕ(N) and gcd(e,ϕ(N))=1\text{gcd}(e, \phi(N)) = 1gcd(e,ϕ(N))=1.
5. **Compute Private Exponent**: Find ddd where d≡e−1mod ϕ(N)d \equiv e^{-1} \mod \phi(N)d≡e−1modϕ(N) (i.e., (e×d)mod ϕ(N)=1(e \times d) \mod \phi(N) = 1(e×d)modϕ(N)=1).

Encryption and Decryption

- **Encryption**: To encrypt a message MMM, compute C=Memod NC = M^e \mod NC=MemodN using the public key (N,e)(N, e)(N,e).
- **Decryption**: To decrypt ciphertext CCC, compute M=Cdmod NM = C^d \mod NM=CdmodN using the private key (N,d)(N, d)(N,d).

Digital Signatures

- **Signing**: Encrypt a hash of the message with the private key ddd to create a signature.
- **Verification**: Decrypt the signature with the public key eee and compare it to the message hash.

Security

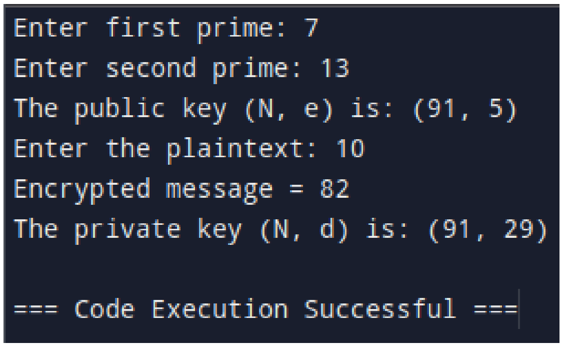
RSA's security is based on the difficulty of factoring large numbers. For robust security, primes should be sufficiently large (typically 2048 bits or more).

PROGRAM:-

```
def modcalci(m, e, N):
        m1 = m
        for _ in range(e):
        c = m1 % N
        m1 = c * m
```

```
    return c

def modinv(a, m):
    for x in range(1, m):
    if (a * x) % m == 1:
    return x
    return None  # Return None if no modular inverse is found

def gcd(x, y):
    while y:
    x, y = y, x % y
    return x

def main():
    # RSA Key Creation
    p = int(input("Enter first prime: "))
    q = int(input("Enter second prime: "))
    N = p * q
    e = 2
    phi = (p - 1) * (q - 1)

    while e < phi:
    if gcd(e, phi) == 1:
    break
    else:
    e += 1

    # Public Key
    print(f"The public key (N, e) is: ({N}, {e})")

    m = int(input("Enter the plaintext: "))

    # RSA Encryption
    c = modcalci(m, e, N)
    print(f"Encrypted message = {c}")

    # Private Key
    d = modinv(e, phi)
    print(f"The private key (N, d) is: ({N}, {d})")

main()
```

OUTPUT:-



```
Enter first prime: 7
Enter second prime: 13
The public key (N, e) is: (91, 5)
Enter the plaintext: 10
Encrypted message = 82
The private key (N, d) is: (91, 29)

=== Code Execution Successful ===
```

EXPERIMENT - 8

AIM:-Write a program to implement the BlowFish algorithm logic.

DESCRIPTION:-

This program demonstrates the use of the **Blowfish algorithm** to encrypt and decrypt a file. Blowfish is a symmetric key block cipher that is efficient and widely used for secure data encryption.

The program performs the following tasks:

1. **Key Generation**: A 128-bit (16 bytes) random key is generated. This key is used for both encryption and decryption.
2. **Cipher Initialization**: The Blowfish cipher is initialized in **CFB (Cipher Feedback)** mode, which allows the encryption of data in small segments while preserving security.
3. **Encryption**: The contents of the input file are read in chunks and encrypted using the Blowfish algorithm. The initialization vector (IV) used during encryption is also printed in base64 format for reference.
4. **Decryption**: Using the same key and IV, the encrypted data is decrypted back into its original form and written into the output file.
5. **Output**: The program displays the encrypted data in base64 format, and after decryption, the decrypted file can be verified to confirm the correctness of the process.

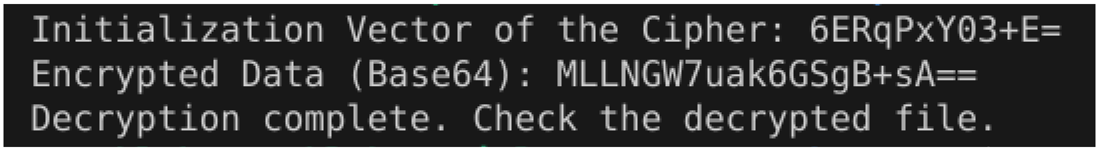
CODE:-

```
# pip3 install pycryptodome
from Crypto.Cipher import Blowfish
from Crypto.Random import get_random_bytes
from base64 import b64encode, b64decode

def encrypt_blowfish(input_file, output_file):
  # Generate a Blowfish key
  key = get_random_bytes(16)  # 128-bit key
  cipher = Blowfish.new(key, Blowfish.MODE_CFB)

  # Get the initialization vector (IV)
  iv = cipher.iv
  print("Initialization Vector of the Cipher:", b64encode(iv).decode('utf-8'))

  # Open input and output files
```

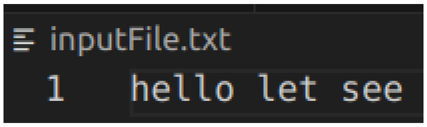
```
    encrypted_data = b""
    with open(input_file, "rb") as fin, open(output_file, "wb") as fout:
        # Encrypt data from inputFile.txt and write to outputFile.txt
        while True:
            chunk = fin.read(64)  # Read in chunks of 64 bytes
            if len(chunk) == 0:
                break
            encrypted_chunk = cipher.encrypt(chunk)
            fout.write(encrypted_chunk)
            encrypted_data += encrypted_chunk  # Collect encrypted data

    # Display encrypted data in base64 format
    print("Encrypted Data (Base64):", b64encode(encrypted_data).decode('utf-8'))

    # Return key and iv for decryption
    return key, iv

def decrypt_blowfish(key, iv, encrypted_file, decrypted_file):
    # Initialize the cipher for decryption using the same key and IV
    cipher = Blowfish.new(key, Blowfish.MODE_CFB, iv=iv)

    # Open the encrypted file and the file to write the decrypted data
    with open(encrypted_file, "rb") as fin, open(decrypted_file, "wb") as fout:
        while True:
            chunk = fin.read(64)  # Read in chunks of 64 bytes
            if len(chunk) == 0:
                break
            decrypted_chunk = cipher.decrypt(chunk)
            fout.write(decrypted_chunk)

    print("Decryption complete. Check the decrypted file.")

if __name__ == "__main__":
    # File paths for encryption and decryption
    input_file = "inputFile.txt"
    encrypted_file = "outputFile.txt"
    decrypted_file = "decryptedFile.txt"
```

```
# Encrypt the data and get the key and iv
key, iv = encrypt_blowfish(input_file, encrypted_file)

# Decrypt the data using the same key and iv
decrypt_blowfish(key, iv, encrypted_file, decrypted_file)
```

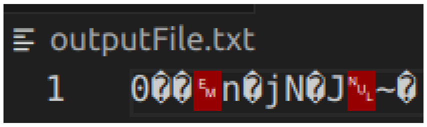
OUTPUT:-



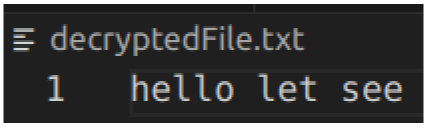
```
Initialization Vector of the Cipher: 6ERqPxY03+E=
Encrypted Data (Base64): MLLNGW7uak6GSgB+sA==
Decryption complete. Check the decrypted file.
```



```
inputFile.txt
1    hello let see
```



```
outputFile.txt
1    0��␙n�jN�J␀~�
```



```
decryptedFile.txt
1    hello let see
```

EXPERIMENT - 9

AIM:-Write a program to implement the Rijndael algorithm

DESCRIPTION:-

This program demonstrates the use of the **Rijndael (AES)** algorithm to encrypt and decrypt a message. Rijndael is the algorithm behind AES, which is widely used for secure data encryption.

The program follows these steps:

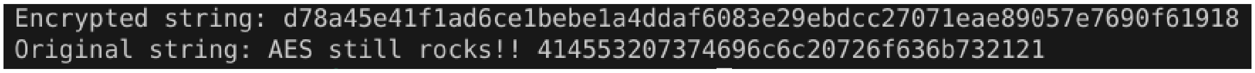
1. **Key Generation**: A 128-bit random key is generated, which is used for both encryption and decryption. AES-128 uses a 16-byte key for this purpose.
2. **Cipher Initialization**: The AES cipher is initialized in **ECB (Electronic Codebook)** mode. This mode performs encryption block by block independently, although it is less secure compared to other modes (like CBC, GCM) because identical plaintext blocks result in identical ciphertext blocks.
3. **Padding and Encryption**: The plaintext message is padded to match the block size required by AES (16 bytes), then it is encrypted using the AES algorithm.
4. **Decryption**: The encrypted message is decrypted back into its original form by reversing the encryption process. After decryption, padding is removed to recover the original message.
5. **Output**: The program displays the encrypted message as a hexadecimal string and the decrypted message as the original plaintext.

CODE:-

```
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
from Crypto.Util.Padding import pad, unpad
import binascii

def as_hex(byte_array):
  return binascii.hexlify(byte_array).decode('utf-8')

def main():
  message = "AES still rocks!!"

  # Generate a random 128-bit key
  key = get_random_bytes(16)  # AES-128, hence 16 bytes
```

```
    # Initialize AES cipher with the key
    cipher = AES.new(key, AES.MODE_ECB)

    # Encrypt the message
    encrypted = cipher.encrypt(pad(message.encode(), AES.block_size))

    print("Encrypted string: " + as_hex(encrypted))

    # Decrypt the message
    cipher_dec = AES.new(key, AES.MODE_ECB)
    decrypted = unpad(cipher_dec.decrypt(encrypted), AES.block_size)

    original_string = decrypted.decode('utf-8')
    print("Original string: " + original_string + " " + as_hex(decrypted))

if __name__ == "__main__":
    main()
```

OUTPUT:-



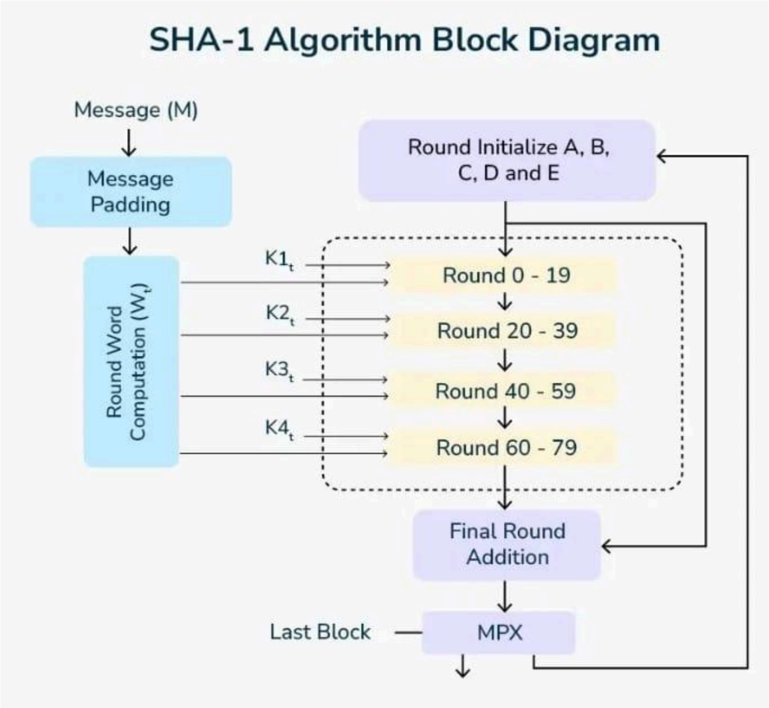
```
Encrypted string: d78a45e41f1ad6ce1bebe1a4ddaf6083e29ebdcc27071eae89057e7690f61918
Original string: AES still rocks!! 414553207374696c6c20726f636b732121
```

EXPERIMENT - 10

AIM:- Write a program to implement SHA 1 algorithm

DESCRIPTION:-

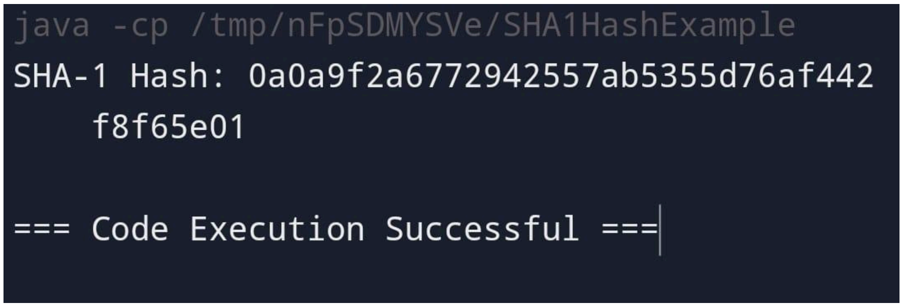
The generateSHA1 method operates similarly but uses the SHA-1 algorithm, resulting in a 160-bit (40-character hexadecimal) hash. SHA-1 produces longer and generally more secure hashes than MD5, though it is still considered vulnerable to certain cryptographic attacks.



CODE:-

```
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

public class SHA1HashExample {
    public static String generateSHA1(String input) {
        try {
            MessageDigest sha1 = MessageDigest.getInstance("SHA-1");
            byte[] hashBytes = sha1.digest(input.getBytes());

            StringBuilder sb = new StringBuilder();
            for (byte b : hashBytes) {
                sb.append(String.format("%02x", b));
            }
            return sb.toString();
        } catch (NoSuchAlgorithmException e) {
            throw new RuntimeException(e);
        }
    }

    public static void main(String[] args) {
        String input = "Hello, World!";
        System.out.println("SHA-1 Hash: " + generateSHA1(input));
    }
}
```

OUTPUT:-



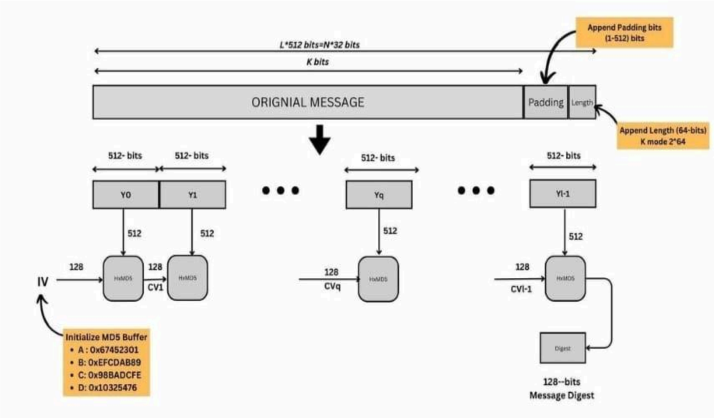
```
java -cp /tmp/nFpSDMYSVe/SHA1HashExample
SHA-1 Hash: 0a0a9f2a6772942557ab5355d76af442
    f8f65e01

=== Code Execution Successful ===
```

EXPERIMENT - 11

AIM:- Write a program to implement MD 5 algorithm

DESCRIPTION:-

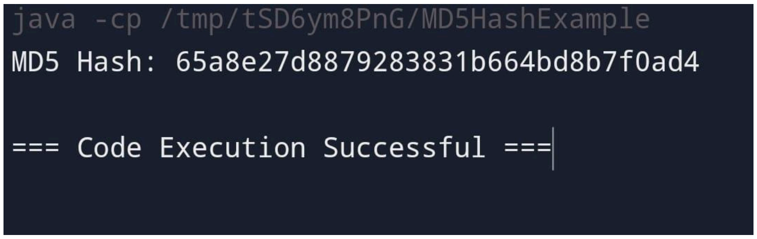
The generateMD5 method uses the MD5 algorithm, which produces a 128-bit (32-character hexadecimal) hash. This hash is generated by converting the input string to bytes, applying the MD5 hash function, and then formatting each byte as a two-character hexadecimal string



CODE:-

```
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

public class MD5HashExample {
    public static String generateMD5(String input) {
        try {
            MessageDigest md = MessageDigest.getInstance("MD5");
            byte[] hashBytes = md.digest(input.getBytes());

            StringBuilder sb = new StringBuilder();
            for (byte b : hashBytes) {
                sb.append(String.format("%02x", b));
            }
            return sb.toString();
        } catch (NoSuchAlgorithmException e) {
            throw new RuntimeException(e);
        }
    }

    public static void main(String[] args) {
        String input = "Hello, World!";
        System.out.println("MD5 Hash: " + generateMD5(input));
    }
}
```

OUTPUT:-



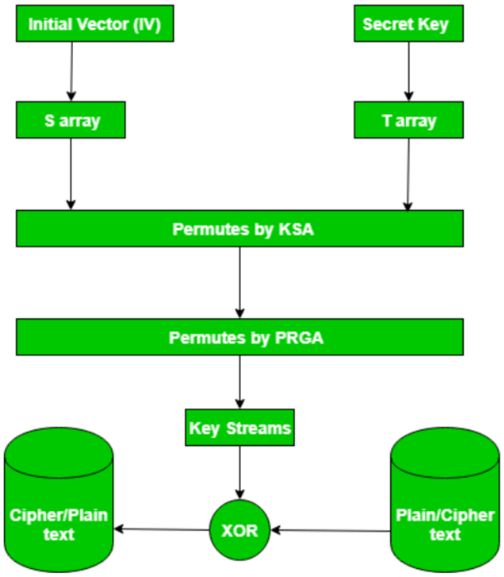
```
java -cp /tmp/tSD6ym8PnG/MD5HashExample
MD5 Hash: 65a8e27d8879283831b664bd8b7f0ad4

=== Code Execution Successful ===
```

EXPERIMENT - 12

AIM:- Write a program to implement RC4 algorithm

DESCRIPTION:-

RC4 (Rivest Cipher 4) is a fast, symmetric stream cipher widely used in protocols like SSL/TLS and WEP for secure data transmission. It generates a pseudo-random keystream to encrypt or decrypt data by XORing each byte of the plaintext with the keystream byte. RC4's simplicity and speed made it popular, but vulnerabilities (like biased output and susceptibility to certain attacks) have led to its gradual phase-out from most modern encryption standards.



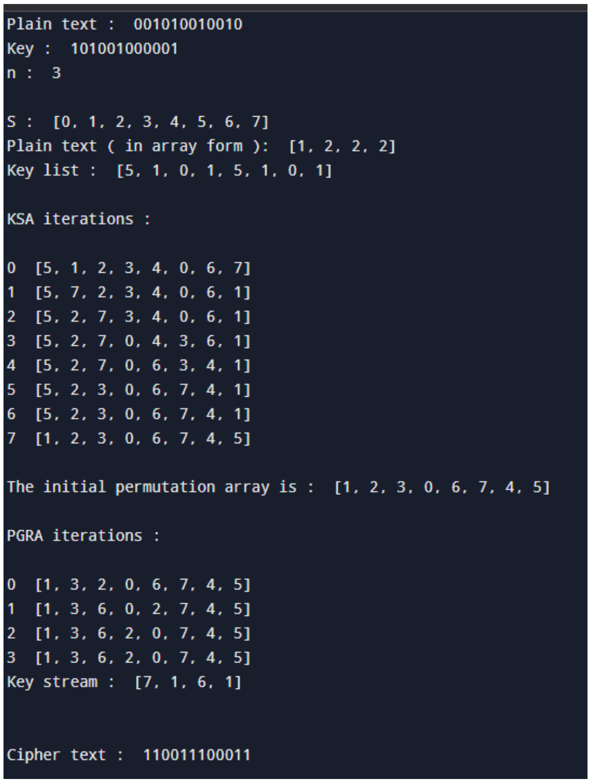
CODE:-

```
# Python3 program for the RC4 algorithm
# for encryption and decryption

# Initialize text and key
plain_text = "001010010010"
key = "101001000001"
n = 3  # No. of bits to consider at a time

# The initial state vector array
S = [i for i in range(0, 2**n)]
key_list = [int(key[i:i + n], 2) for i in range(0, len(key), n)]
pt = [int(plain_text[i:i + n], 2) for i in range(0, len(plain_text), n)]
```

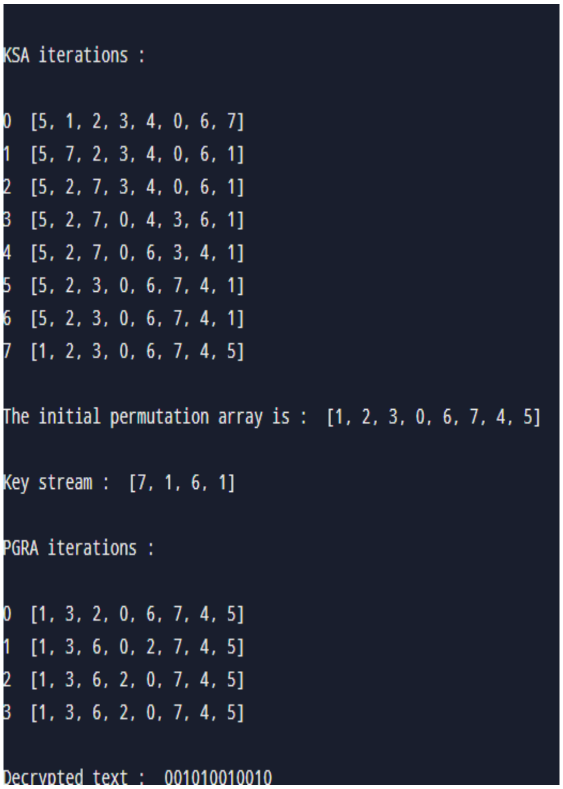
```
# Adjust key_list length to match S
if len(S) != len(key_list):
    key_list += key_list[:len(S) - len(key_list)]

# Key Scheduling Algorithm (KSA)
def KSA(S, key_list):
    j = 0
    for i in range(len(S)):
        j = (j + S[i] + key_list[i]) % len(S)
        S[i], S[j] = S[j], S[i]
    return S

# Pseudo-Random Generation Algorithm (PRGA)
def PRGA(S, text_length):
    i = j = 0
    key_stream = []
    for _ in range(text_length):
        i = (i + 1) % len(S)
        j = (j + S[i]) % len(S)
        S[i], S[j] = S[j], S[i]
        t = (S[i] + S[j]) % len(S)
        key_stream.append(S[t])
    return key_stream

# XOR between generated key stream and input text
def XOR(input_text, key_stream):
    return [input_text[i] ^ key_stream[i] for i in range(len(input_text))]

# Encryption function
def encryption():
    print("Plain text:", plain_text)
    print("Key:", key)
    S_init = KSA(S[:], key_list)  # Initial permutation
    key_stream = PRGA(S_init, len(pt))
    cipher_text = XOR(pt, key_stream)
    encrypted_to_bits = ".join(f"{bin(c)[2:]:0>{n}}" for c in cipher_text)
    print("Cipher text:", encrypted_to_bits)
    return cipher_text
```

```
# Decryption function
def decryption(cipher_text):
    S_init = KSA(S[:], key_list)
    key_stream = PRGA(S_init, len(pt))
    original_text = XOR(cipher_text, key_stream)
    decrypted_to_bits = ''.join(f"{bin(p)[2:]:0>{n}}" for p in original_text)
    print("Decrypted text:", decrypted_to_bits)

# Driver Code
cipher_text = encryption()
print("-------------------------------------------------------")
decryption(cipher_text)
```

OUTPUT:-



```
Plain text :  001010010010
Key :  101001000001
n :  3

S :  [0, 1, 2, 3, 4, 5, 6, 7]
Plain text ( in array form ):  [1, 2, 2, 2]
Key list :  [5, 1, 0, 1, 5, 1, 0, 1]

KSA iterations :

0  [5, 1, 2, 3, 4, 0, 6, 7]
1  [5, 7, 2, 3, 4, 0, 6, 1]
2  [5, 2, 7, 3, 4, 0, 6, 1]
3  [5, 2, 7, 0, 4, 3, 6, 1]
4  [5, 2, 7, 0, 6, 3, 4, 1]
5  [5, 2, 3, 0, 6, 7, 4, 1]
6  [5, 2, 3, 0, 6, 7, 4, 1]
7  [1, 2, 3, 0, 6, 7, 4, 5]

The initial permutation array is :  [1, 2, 3, 0, 6, 7, 4, 5]

PGRA iterations :

0  [1, 3, 2, 0, 6, 7, 4, 5]
1  [1, 3, 6, 0, 2, 7, 4, 5]
2  [1, 3, 6, 2, 0, 7, 4, 5]
3  [1, 3, 6, 2, 0, 7, 4, 5]
Key stream :  [7, 1, 6, 1]


Cipher text :  110011100011
```



```
KSA iterations :

0  [5, 1, 2, 3, 4, 0, 6, 7]
1  [5, 7, 2, 3, 4, 0, 6, 1]
2  [5, 2, 7, 3, 4, 0, 6, 1]
3  [5, 2, 7, 0, 4, 3, 6, 1]
4  [5, 2, 7, 0, 6, 3, 4, 1]
5  [5, 2, 3, 0, 6, 7, 4, 1]
6  [5, 2, 3, 0, 6, 7, 4, 1]
7  [1, 2, 3, 0, 6, 7, 4, 5]

The initial permutation array is :  [1, 2, 3, 0, 6, 7, 4, 5]

Key stream :  [7, 1, 6, 1]

PGRA iterations :

0  [1, 3, 2, 0, 6, 7, 4, 5]
1  [1, 3, 6, 0, 2, 7, 4, 5]
2  [1, 3, 6, 2, 0, 7, 4, 5]
3  [1, 3, 6, 2, 0, 7, 4, 5]

Decrypted text :  001010010010
```

EXPERIMENT - 13

AIM:- Write a program to implement Euclidean Algorithm and Advanced Euclidean Algorithm

DESCRIPTION:-

The Euclidean algorithm is a way to find the greatest common divisor of two positive integers. The GCD of two numbers is the largest number that divides both of them. A simple way to find GCD is to factorise both numbers and multiply common prime factors.

Basic Euclidean Algorithm for GCD:

The algorithm is based on the below facts.

- If we subtract a smaller number from a larger one (we reduce a larger number), GCD doesn't change. So if we keep subtracting repeatedly the larger of two, we end up with GCD.
- Now instead of subtraction, if we divide the larger number, the algorithm stops when we find the remainder 0.

Extended Euclidean Algorithm:

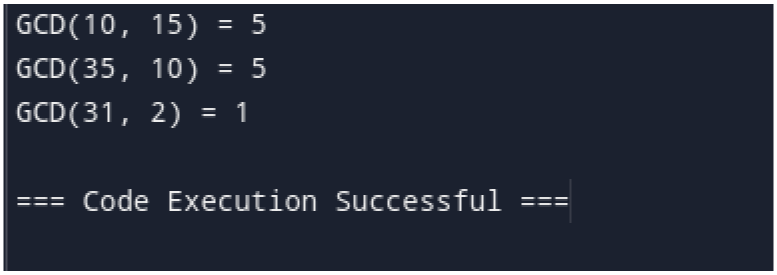
Extended Euclidean algorithm also finds integer coefficients x and y such that:ax + by = gcd(a, b). The extended Euclidean algorithm updates the results of gcd(a, b) using the results calculated by the recursive call gcd(b%a, a).

CODE:

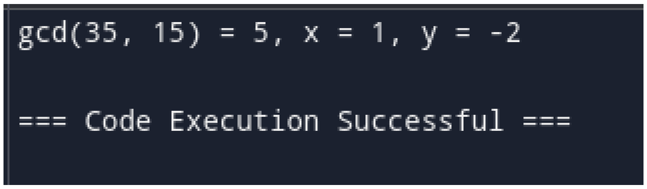
```
#Euclidean algorithm

def gcd(a, b):
   if a == 0:
      return b
   return gcd(b % a, a)

# Driver code
a, b = 10, 15
print(f"GCD({a}, {b}) = {gcd(a, b)}")

a, b = 35, 10
print(f"GCD({a}, {b}) = {gcd(a, b)}")

a, b = 31, 2
```

```
print(f"GCD({a}, {b}) = {gcd(a, b)}")

# Advanced Euclidean Algorithm

def gcd_extended(a, b):
    if a == 0:
        return b, 0, 1
    gcd, x1, y1 = gcd_extended(b % a, a)
    x = y1 - (b // a) * x1
    y = x1
    return gcd, x, y

# Driver Program
a = 35
b = 15
gcd, x, y = gcd_extended(a, b)
print(f"gcd({a}, {b}) = {gcd}, x = {x}, y = {y}")
```

OUTPUT:



```
GCD(10, 15) = 5
GCD(35, 10) = 5
GCD(31, 2) = 1

=== Code Execution Successful ===
```



```
gcd(35, 15) = 5, x = 1, y = -2

=== Code Execution Successful ===
```

EXPERIMENT - 14

AIM:- To know about cryptographic tools.

DESCRIPTION:-

Cryptlib

- Cryptlib is an open-source, cross-platform software security toolkit designed for building cryptographic security into applications. It provides a wide range of cryptographic operations including encryption, digital signatures, secure data transport, and more.
- Cryptlib is distributed under the Sleepycat License, which is compatible with the GNU General Public License (GPL). It is also available under a commercial license for proprietary use.
- Recent updates have focused on expanding support for modern cryptographic algorithms like AES-GCM, Curve25519, and improved support for secure multi-party computations. Cryptlib offers APIs that make it suitable for embedding cryptographic functions in mobile, web, and server applications.

Crypto++

- Crypto++ (also known as CryptoPP, libcrypto++, and libcryptopp) is a widely-used, free and open-source C++ library of cryptographic algorithms, written by Wei Dai. It supports a comprehensive suite of cryptographic primitives and schemes, including block ciphers (AES, DES), stream ciphers, public-key cryptography (RSA, DSA, ECC), and various hash functions (SHA-256, SHA-3).
- Crypto++ is frequently used in research and industry projects. It is maintained actively, with recent enhancements including support for newer encryption standards like ChaCha20-Poly1305, BLAKE3 hashing, and the EdDSA signature algorithm.
- The library emphasizes high performance and portability, making it a preferred choice for C++ developers in security-sensitive applications.

LibreSSL

- LibreSSL is an open-source implementation of the Transport Layer Security (TLS) protocol, developed by the OpenBSD project. It was forked from OpenSSL in 2014 in response to the Heartbleed vulnerability, with the goal of modernizing the codebase, improving security, and reducing complexity.
- LibreSSL has focused on removing obsolete features and improving code quality. It no longer includes deprecated cryptographic algorithms like SSLv2/SSLv3 and MD5, focusing on secure, modern standards like TLS 1.3, SHA-3, and Ed25519.

- Recent updates have included enhancements in cryptographic algorithm support, better random number generation, and improved performance on modern CPU architectures. It remains popular among UNIX-like systems for providing secure communication channels.

OpenSSL

- **Description**: OpenSSL is one of the most widely used open-source libraries for implementing the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols. It also provides a comprehensive suite of cryptographic functions, including public-key cryptography, symmetric encryption, and hashing algorithms.
- **Updates**: OpenSSL 3.0 introduced a new "Provider" architecture that allows for better flexibility in integrating cryptographic algorithms. It also includes improved support for post-quantum cryptography, ChaCha20-Poly1305, and updated TLS 1.3 support.
- **Use Cases**: Secure web communication (HTTPS), VPNs, and cryptographic operations in software applications.

PyCryptodome

- **Description**: PyCryptodome is a self-contained Python package of low-level cryptographic primitives. It is a fork of the old PyCrypto library and includes many improvements and additional features such as support for AES, RSA, DSA, and hash functions.
- **Updates**: The latest versions have improved support for AES-GCM, RSA-PSS signatures, and hardware-accelerated encryption on modern processors.
- **Use Cases**: Cryptographic operations in Python-based projects, such as secure file encryption and digital signatures.

Tink (by Google)

- **Description**: Tink is an open-source cryptographic library developed by Google. It provides easy-to-use and secure APIs for cryptographic operations such as encryption, digital signatures, and key management. Tink aims to make cryptography safer and simpler for developers.
- **Updates**: Tink now supports hybrid encryption schemes, better integration with cloud key management services (e.g., Google Cloud KMS, AWS KMS), and enhanced support for AES-SIV for misuse-resistant encryption.
- **Use Cases**: Secure application development, cloud-based encryption, and secure data storage.

EXPERIMENT 15

AIM:- To implement Triple DES (3DES) encryption and decryption.

DESCRIPTION:-

Triple DES (3DES) is an enhancement of the DES algorithm that applies the DES cipher algorithm three times to each data block. This implementation uses the ECB (Electronic Codebook) mode for simplicity. The process involves:

- Generating a 24-byte key with adjusted parity.
- Encrypting the input data using the 3DES algorithm, with padding to make the data length a multiple of 8 bytes.
- Decrypting the encrypted data using the same key.

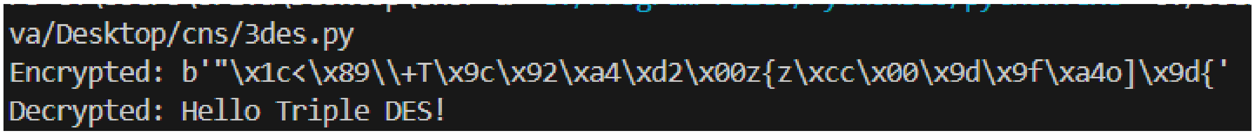
The `triple_des_encrypt` function encrypts the data, and the `triple_des_decrypt` function reverses the process to retrieve the original plaintext.

PROGRAM:-

```
from Crypto.Cipher import DES3
from Crypto.Random import get_random_bytes

# Generate a 3DES key (24 bytes)
key = DES3.adjust_key_parity(get_random_bytes(24))

# Encryption
def triple_des_encrypt(data, key):
        cipher = DES3.new(key, DES3.MODE_ECB)
        padded_data = data + (8 - len(data) % 8) * ' '  # Padding
        encrypted_data = cipher.encrypt(padded_data.encode())
        return encrypted_data

# Decryption
def triple_des_decrypt(encrypted_data, key):
        cipher = DES3.new(key, DES3.MODE_ECB)
        decrypted_data = cipher.decrypt(encrypted_data).decode().rstrip()
        return decrypted_data

# Example usage
data = "Hello Triple DES!"
encrypted_data = triple_des_encrypt(data, key)
```

```
decrypted_data = triple_des_decrypt(encrypted_data, key)
print("Encrypted:", encrypted_data)
print("Decrypted:", decrypted_data)
```

OUTPUT:-



```
va/Desktop/cns/3des.py
Encrypted: b'"\x1c<\x89\\+T\x9c\x92\xa4\xd2\x00z{z\xcc\x00\x9d\x9f\xa4o]\x9d{'
Decrypted: Hello Triple DES!
```

CONCLUSION:-

The Triple DES (3DES) implementation successfully encrypts and decrypts data, providing enhanced security by applying the DES algorithm three times. However, ECB mode has vulnerabilities in terms of pattern recognition and is not recommended for secure communications in modern applications.

EXPERIMENT 16.1

AIM:- To implement the Rail Fence cipher for encrypting and decrypting text.

DESCRIPTION:-

The Rail Fence cipher is a form of transposition cipher where the plaintext is written in a zigzag pattern across multiple rows (rails), determined by the key. The process involves:

- Writing the plaintext in a zigzag pattern across the specified number of rails.
- Reading the characters row by row to produce the encrypted text.
- Decrypting by reconstructing the zigzag pattern and retrieving the original message.

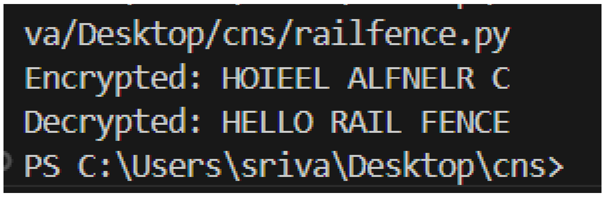
The `rail_fence_encrypt` function creates the encrypted message, while the `rail_fence_decrypt` function reverses the process.

PROGRAM:-

```
# Encryption
def rail_fence_encrypt(text, key):
    # Create a list of strings for each rail
    rails = ['' for _ in range(key)]
    direction_down = False
    row = 0

    # Traverse the text and place characters in the appropriate rail
    for char in text:
        rails[row] += char
        # Change direction when we reach the top or bottom rail
        if row == 0 or row == key - 1:
            direction_down = not direction_down
        # Move up or down the rails
        row += 1 if direction_down else -1

    # Join all the rails to get the encrypted text
    return ''.join(rails)

# Decryption
def rail_fence_decrypt(encrypted_text, key):
    # Create a matrix to mark the zigzag pattern
```

```
n = len(encrypted_text)
zigzag = [['\n' for _ in range(n)] for _ in range(key)]
direction_down = None
row, col = 0, 0

# Mark the positions in the zigzag pattern
for i in range(n):
    if row == 0:
        direction_down = True
    elif row == key - 1:
        direction_down = False

    # Mark this position
    zigzag[row][col] = '*'
    col += 1

    # Move up or down the rails
    row += 1 if direction_down else -1

# Fill the zigzag pattern with the encrypted text
index = 0
for i in range(key):
    for j in range(n):
        if zigzag[i][j] == '*' and index < n:
            zigzag[i][j] = encrypted_text[index]
            index += 1

# Read the matrix in zigzag pattern to get the decrypted text
result = []
row, col = 0, 0
for i in range(n):
    if row == 0:
        direction_down = True
    elif row == key - 1:
        direction_down = False

    # Append the character
    if zigzag[row][col] != '\n':
            result.append(zigzag[row][col])
            col += 1

        # Move up or down the rails
        row += 1 if direction_down else -1

    return ''.join(result)

# Example usage
text = "HELLO RAIL FENCE"
key = 3
encrypted = rail_fence_encrypt(text, key)
decrypted = rail_fence_decrypt(encrypted, key)
print("Encrypted:", encrypted)
print("Decrypted:", decrypted)
```

OUTPUT:-



```
va/Desktop/cns/railfence.py
Encrypted: HOIEEL ALFNELR C
Decrypted: HELLO RAIL FENCE
PS C:\Users\sriva\Desktop\cns>
```

CONCLUSION:-

The Rail Fence cipher successfully encrypts and decrypts the text using a simple zigzag pattern. It provides a basic level of security and is mainly used for educational purposes rather than practical applications due to its simplicity and vulnerability to frequency analysis.

EXPERIMENT 16.2

AIM:-To implement the Vernam cipher, a symmetric key cipher.

DESCRIPTION:-

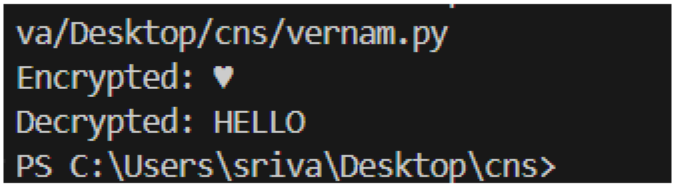
The Vernam cipher, also known as the One-Time Pad, is a symmetric encryption technique that uses a key of the same length as the plaintext. The encryption and decryption processes involve:

- Applying the XOR operation between each character of the plaintext and the corresponding character of the key.
- The `vernam_cipher_encrypt` function performs this operation to encrypt the message.
- The `vernam_cipher_decrypt` function reverses the XOR operation to retrieve the original plaintext.

PROGRAM:-

```
def vernam_cipher_encrypt(plain_text, key):
        encrypted_text = ''.join([chr(ord(plain_text[i]) ^ ord(key[i])) for i in range(len(plain_text))])
        return encrypted_text

def vernam_cipher_decrypt(encrypted_text, key):
        decrypted_text = ''.join([chr(ord(encrypted_text[i]) ^ ord(key[i])) for i in
range(len(encrypted_text))])
        return decrypted_text

# Example usage
plain_text = "HELLO"
key = "XMCKL"  # Key must be the same length as plain_text
encrypted_text = vernam_cipher_encrypt(plain_text, key)
decrypted_text = vernam_cipher_decrypt(encrypted_text, key)
print("Encrypted:", encrypted_text)
print("Decrypted:", decrypted_text)
```

OUTPUT:-



```
va/Desktop/cns/vernam.py
Encrypted: ♥
Decrypted: HELLO
PS C:\Users\sriva\Desktop\cns>
```

CONCLUSION:-

The Vernam cipher effectively encrypts and decrypts the message using the XOR operation, ensuring strong security when the key is random, unique, and used only once. However, its practical use is limited due to the requirement for a key of equal length to the plaintext and the need for secure key distribution.

EXPERIMENT 16.3

AIM:- To implement the Vigenère cipher, a polyalphabetic substitution cipher.

DESCRIPTION:-

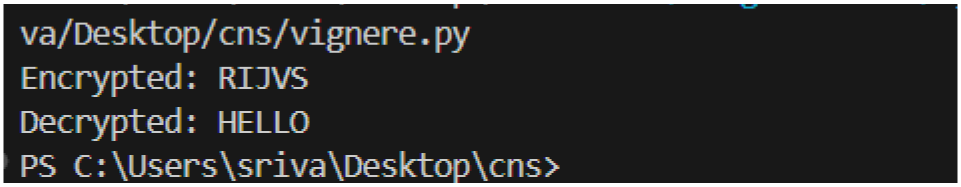
The Vigenère cipher is a method of encrypting alphabetic text using a keyword where each letter of the plaintext is shifted according to the corresponding letter of the repeating key. The process involves:

- Generating a repeating key that matches the length of the plaintext.
- Shifting each letter of the plaintext according to the corresponding letter of the key.
- The `vigenere_encrypt` function encodes the text, and the `vigenere_decrypt` function reverses the process.

PROGRAM:-

```
def vigenere_encrypt(text, key):
        key = key * (len(text) // len(key)) + key[:len(text) % len(key)]
        encrypted_text = ''.join([chr(((ord(text[i]) + ord(key[i])) % 26) + 65) for i in range(len(text))])
        return encrypted_text

def vigenere_decrypt(encrypted_text, key):
        key = key * (len(encrypted_text) // len(key)) + key[:len(encrypted_text) % len(key)]
        decrypted_text = ''.join([chr(((ord(encrypted_text[i]) - ord(key[i]) + 26) % 26) + 65) for i in
range(len(encrypted_text))])
        return decrypted_text

# Example usage
text = "HELLO"
key = "KEY"
encrypted = vigenere_encrypt(text, key)
decrypted = vigenere_decrypt(encrypted, key)
print("Encrypted:", encrypted)
print("Decrypted:", decrypted)
```

OUTPUT:-



```
va/Desktop/cns/vignere.py
Encrypted: RIJVS
Decrypted: HELLO
PS C:\Users\sriva\Desktop\cns>
```

CONCLUSION:-

The Vigenère cipher effectively encrypts and decrypts the text by using a repeating key to apply multiple Caesar shifts. It provides better security than monoalphabetic ciphers but is still vulnerable to cryptanalysis, especially when the keyword is short or can be guessed.