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 displays the result.

DESCRIPTION:

The operation "XOR each character in this string with 0" means applying the XOR (exclusive OR) bitwise operation to each character of the string using the value `0`. Since XOR with `0` leaves the bits unchanged, each character will remain the same.

In other words, XORing any character with 0 does not alter the original string.

CODE:

```c
#include <stdio.h>
int main() {
    char str[] = "Hello world";
    for (int i = 0; str[i] != '\0'; i++) {
        str[i] = str[i] ^ 0;
    }
    printf("Result after XOR with 0: %s\n", str);
    return 0;
}
```

OUTPUT:

```
Result after XOR with 0: Hello world

--------------------------------
Process exited after 13.26 seconds with return value 0
Press any key to continue . . .
```

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 display the result.

DESCRIPTION:
AND each character with 127:

- o The AND operation compares each bit of the character's ASCII value with the number 127 (01111111 in binary). Since 127 has its highest bit set to 0, this effectively clears the most significant bit of each character, leaving the other bits unchanged. Characters with an ASCII value above 127 will be reduced to a value below 127.

XOR each character with 127:

- o The XOR operation flips the bits wherever the corresponding bit of the number 127 (01111111) is 1. This will invert the lower 7 bits of each character, drastically changing its ASCII value. For example, if a character is lowercase, it might turn into uppercase or a different symbol depending on its original ASCII value.

Both operations result in modified strings, but in different ways based on how bitwise AND or XOR affects the character's ASCII values.

CODE:

```c
#include <stdio.h>
int main() {
    char *str = "Hello world";
    int i;
    printf("Original string: %s\n", str);
    printf("AND with 127: ");
    for (i = 0; str[i] != '\0'; i++) {
        printf("%c", str[i] & 127);
    }
    printf("\n");
    printf("OR with 127: ");
    for (i = 0; str[i] != '\0'; i++) {
        printf("%c", str[i] | 127);
    }
    printf("\n");
    printf("XOR with 127: ");
    for (i = 0; str[i] != '\0'; i++) {
        printf("%c", str[i] ^ 127);
    }
    printf("\n");
    return 0;
```

```
}
```

OUTPUT:

```
Original string: Hello world
AND with 127: Hello world
OR with 127:
XOR with 127: 7ʿ_


rocess exited after 14.76 seconds with return value 0
Press any key to continue . . .
```
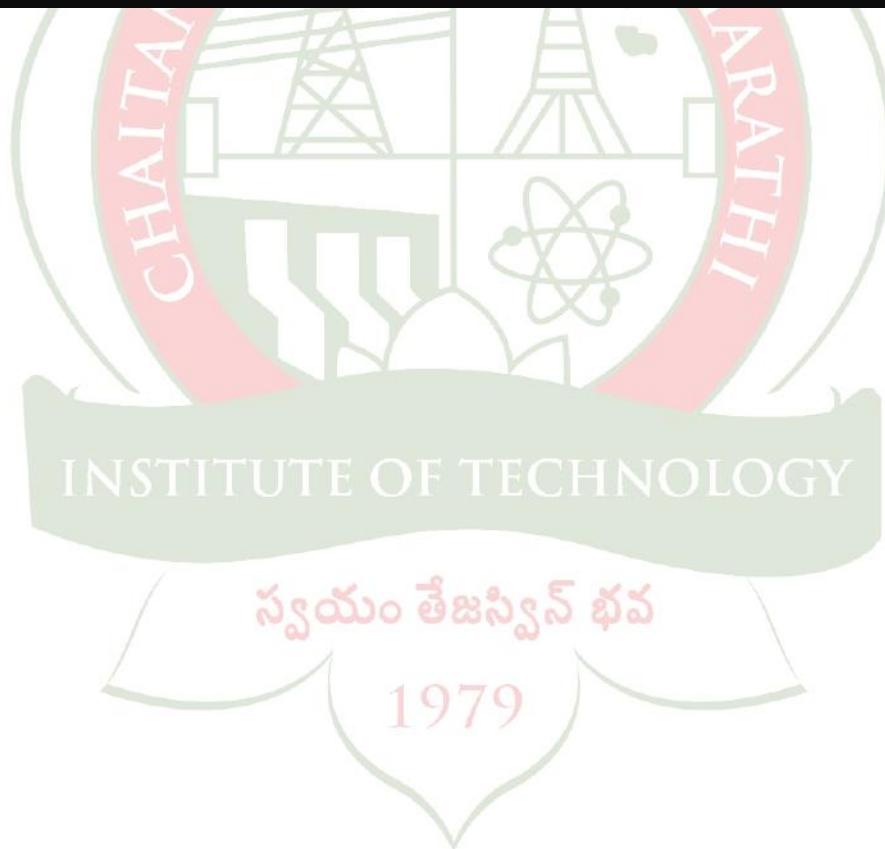
EXPERIMENT-3

AIM: Write a Java program to perform encryption and decryption using the following algorithms

a. Ceaser cipher

DESCRIPTION:
A **Caesar cipher** is one of the simplest encryption techniques. It works by shifting each letter of the plaintext by a fixed number of positions in the alphabet. For example, with a shift of 3:

- **A** becomes **D**

- **B** becomes **E**

- **C** becomes **F**

This shifting continues for all letters in the message. When the end of the alphabet is reached, it wraps around (so **Z** would shift to **C** with a shift of 3). Non-letter characters remain unchanged.
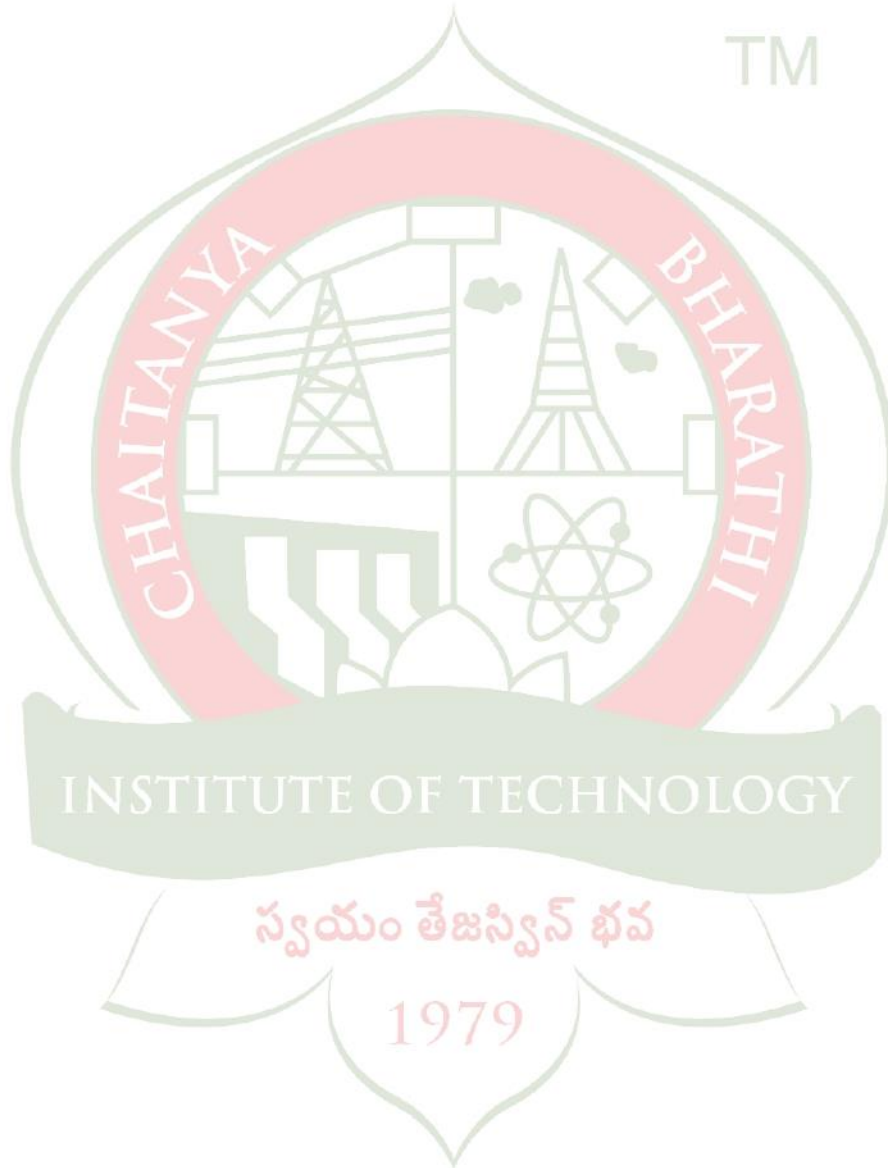
CODE:

```java
class ceaser {
    public static String encrypt(String plaintext, int key) {
        StringBuilder encryptedText = new StringBuilder();
        for (char c : plaintext.toCharArray()) {
            char shiftedChar = (char) (c + key);
            encryptedText.append(shiftedChar);
        }
        return encryptedText.toString();
    }
    public static String decrypt(String encryptedText, int key) {
        StringBuilder decryptedText = new StringBuilder();
        for (char c : encryptedText.toCharArray()) {
            char shiftedChar = (char) (c - key);
            decryptedText.append(shiftedChar);
        }
        return decryptedText.toString();
    }
    public static void main(String[] args) {
        String plaintext = "ABC";
        int key = 3;
        String encryptedText = encrypt(plaintext, key);
        System.out.println("Encrypted Text: " + encryptedText);
        String decryptedText = decrypt(encryptedText, key);
        System.out.println("Decrypted Text: " + decryptedText);
    }
}
```

OUTPUT:

```
Encrypted Text: DEF
Decrypted Text: ABC
```

b. Substitution Cipher

DESCRIPTION:

A **Substitution cipher** is a type of encryption where each letter in the plaintext is replaced with another letter according to a fixed system. Unlike the Caesar cipher, where letters are shifted uniformly, a substitution cipher uses a more complex mapping.

For example:

- **Plaintext**: HELLO

- **Ciphertext** (using a random substitution): XNZZT

CODE:

```java
class subcipher {
    public static String encrypt(String plaintext, int[] key) {
        StringBuilder encryptedText = new StringBuilder();
        for (int i = 0; i < plaintext.length(); i++) {
            char c = plaintext.charAt(i);
            char shiftedChar = (char) (c + key[i]);
            encryptedText.append(shiftedChar);
        }
        return encryptedText.toString();
    }
    public static String decrypt(String encryptedText, int[] key) {
        StringBuilder decryptedText = new StringBuilder();
        for (int i = 0; i < encryptedText.length(); i++) {
            char c = encryptedText.charAt(i);
            char shiftedChar = (char) (c - key[i]);
            decryptedText.append(shiftedChar);
        }
        return decryptedText.toString();
    }
    public static void main(String[] args) {
        String plaintext = "ABC";
        int[] key = {1, 3, 5};
        String encryptedText = encrypt(plaintext, key);
        System.out.println("Encrypted Text: " + encryptedText);
        String decryptedText = decrypt(encryptedText, key);
        System.out.println("Decrypted Text: " + decryptedText);
    }
}
```

OUTPUT:

```
Encrypted Text: BEH
Decrypted Text: ABC
```

c. Hill Cipher

DESCRIPTION:

The **Hill cipher** is a polygraphic substitution cipher that uses linear algebra, specifically matrix multiplication, to encrypt text. It encrypts blocks of letters, treating them as vectors and multiplying them by an encryption matrix.

Here's how it works:

1. **Key Matrix**: A square matrix (e.g., 2x2 or 3x3) acts as the encryption key.

2. **Plaintext Blocks**: The plaintext is divided into equally sized blocks of letters.

3. **Matrix Multiplication**: Each block is converted to a vector (using numerical values for letters), then multiplied by the key matrix.

4. **Modulo Operation**: The resulting vector is reduced modulo 26 (for the 26 letters of the alphabet) to get the ciphertext.

To decrypt, the inverse of the key matrix is used. It requires modular arithmetic and the matrix determinant to compute the inverse.

CODE:

```java
public class HillCipher {
    // Function to calculate the determinant of a 3x3 matrix
    public static int determinant(int[][] matrix) {
        return (matrix[0][0] * (matrix[1][1] * matrix[2][2] -
matrix[1][2] * matrix[2][1])
                - matrix[0][1] * (matrix[1][0] * matrix[2][2] -
matrix[1][2] * matrix[2][0])
                + matrix[0][2] * (matrix[1][0] * matrix[2][1] -
matrix[1][1] * matrix[2][0])) % 26;
    }
    // Function to find the modular inverse of the determinant
    public static int modInverse(int d, int mod) {
        d = d % mod;
        for (int x = 1; x < mod; x++) {
            if ((d * x) % mod == 1) return x;
        }
        return 1; // Fallback
    }
    // Function to calculate the adjoint of a 3x3 matrix
    public static int[][] adjoint(int[][] matrix) {
        int[][] adj = new int[3][3];
         adj[0][0] = (matrix[1][1] * matrix[2][2] - matrix[1][2] *
matrix[2][1]) % 26;
         adj[0][1] = (matrix[0][2] * matrix[2][1] - matrix[0][1] *
matrix[2][2]) % 26;
```

```
        adj[0][2] = (matrix[0][1] * matrix[1][2] - matrix[0][2] *
matrix[1][1]) % 26;
        adj[1][0] = (matrix[1][2] * matrix[2][0] - matrix[1][0] *
matrix[2][2]) % 26;
        adj[1][1] = (matrix[0][0] * matrix[2][2] - matrix[0][2] *
matrix[2][0]) % 26;
        adj[1][2] = (matrix[0][2] * matrix[1][0] - matrix[0][0] *
matrix[1][2]) % 26;
        adj[2][0] = (matrix[1][0] * matrix[2][1] - matrix[1][1] *
matrix[2][0]) % 26;
        adj[2][1] = (matrix[0][1] * matrix[2][0] - matrix[0][0] *
matrix[2][1]) % 26;
        adj[2][2] = (matrix[0][0] * matrix[1][1] - matrix[0][1] *
matrix[1][0]) % 26;
        // Ensure positive modulo
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                if (adj[i][j] < 0) adj[i][j] += 26;
            }
        }
        return adj;
    }
    // Function to multiply two matrices and return the result mod
26
    public static int[][] multiplyMatrices(int[][] matrix1, int[][]
matrix2) {
        int[][] result = new int[3][3];
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                result[i][j] = 0;
                for (int k = 0; k < 3; k++) {
                    result[i][j] += matrix1[i][k] * matrix2[k][j];
                }
                result[i][j] = result[i][j] % 26;
                if (result[i][j] < 0) result[i][j] += 26; // Ensure
positive modulo
            }
        }
        return result;
    }
    // Function to encrypt a plaintext message
    public static String encrypt(String plaintext, int[][] key) {
        int n = plaintext.length();
        int[][] plainMatrix = new int[3][3];
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                plainMatrix[i][j] = plaintext.charAt(i * 3 + j) -
'A';
            }
```

```java
        }
        int[][] cipherMatrix = multiplyMatrices(key, plainMatrix);

        StringBuilder ciphertext = new StringBuilder();
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                ciphertext.append((char) (cipherMatrix[i][j] + 'A'));
            }
        }
        return ciphertext.toString();
    }
    // Function to decrypt a ciphertext message
    public static String decrypt(String ciphertext, int[][] key) {
        int[][] adj = adjoint(key);
        int d = determinant(key);
        int dInverse = modInverse(d, 26);
        int[][] inverseKey = new int[3][3];
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                inverseKey[i][j] = (dInverse * adj[i][j]) % 26;
                if (inverseKey[i][j] < 0) inverseKey[i][j] += 26;
// Ensure positive modulo
            }
        }
        int[][] cipherMatrix = new int[3][3];
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                cipherMatrix[i][j] = ciphertext.charAt(i * 3 + j) -
'A';
            }
        }
            int[][]  plainMatrix  =  multiplyMatrices(inverseKey,
cipherMatrix);
        StringBuilder plaintext = new StringBuilder();
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                plaintext.append((char) (plainMatrix[i][j] + 'A'));
            }
        }
        return plaintext.toString();
    }
    public static void main(String[] args) {
        int[][] key = {
            {2, 4, 5},
            {9, 2, 1},
            {3, 17, 7}
        };
        String plaintext = "ATTACKBBB";
        String ciphertext = encrypt(plaintext, key);
```

```
        System.out.println("Encrypted text: " + ciphertext);
        String decryptedText = decrypt(ciphertext, key);
        System.out.println("Decrypted text: " + decryptedText);
    }
}
```

<u>OUTPUT:</u>

```
Encrypted text: FZFBUKHUA
Decrypted text: ATTACKBBB
```
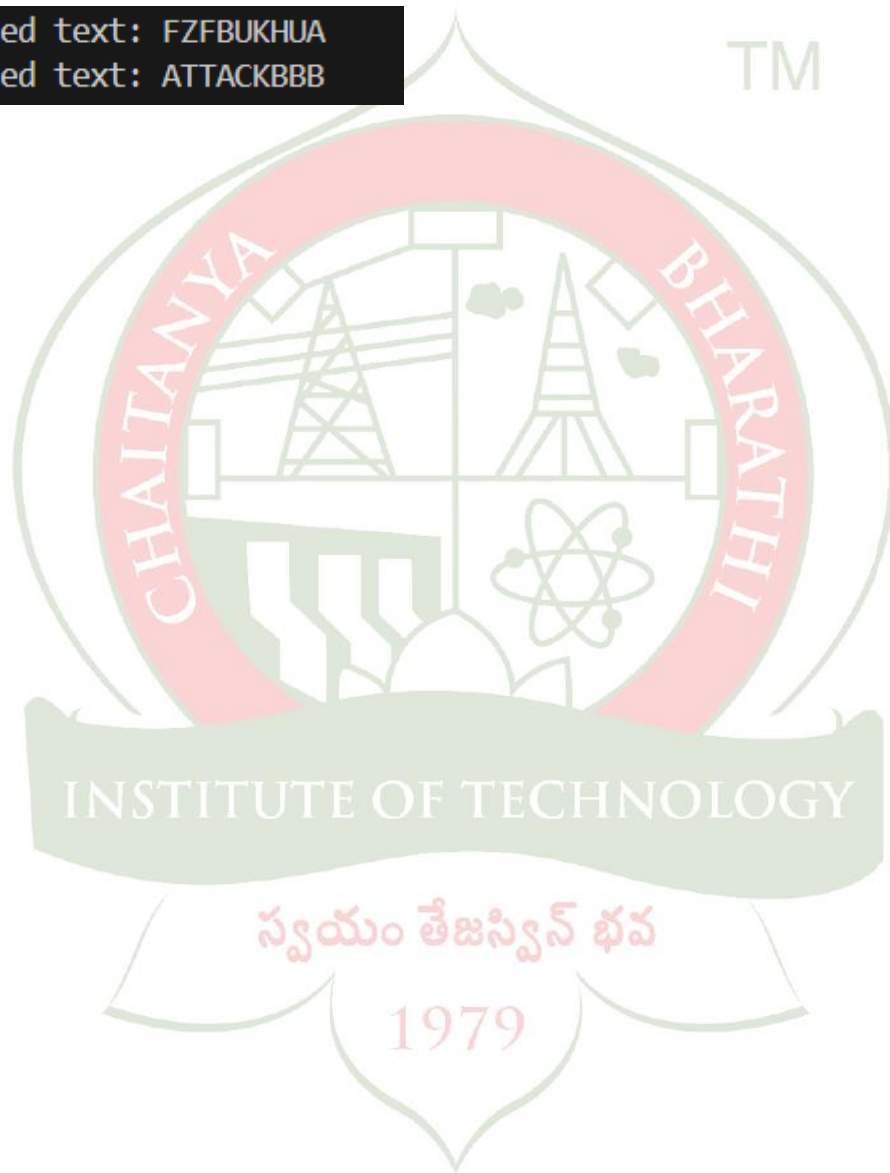
d. Play fair Cipher

DESCRIPTION:
The **Playfair Cipher** is a digraph substitution cipher that encrypts pairs of letters in plaintext. It uses a 5x5 matrix of letters, and each letter pair is encrypted based on its position in this matrix. The rules for encryption are as follows:

1. If both letters are in the same row, each letter is replaced by the letter to its right.

2. If both letters are in the same column, each letter is replaced by the letter below.

3. If the letters form a rectangle, each is replaced by the letter on the same row but in the other pair's column.

CODE:

```java
import java.util.*;
public class PlayfairCipher {
    static String removeDuplicate(String s) {
        int j, index = 0, len = s.length();
        char c[] = s.toCharArray();
        for (int i = 0; i < len; i++) {
            for (j = 0; j < i; j++) {
                if (c[i] == c[j]) break;
            }
            if (i == j) c[index++] = c[i];
        }
        s = new String((Arrays.copyOf(c, index)));
        return s;
    }
    static String removeWhiteSpace(char[] ch, String key) {
        char[] c = key.toCharArray();
        for (int i = 0; i < c.length; i++) {
            for (int j = 0; j < ch.length; j++) {
                if (c[i] == ch[j]) c[i] = ' ';
            }
        }
        key = new String(c);
        key = key.replaceAll(" ", "");
        return key;
    }
    static String makePair(String pt) {
        String s = "";
        char c = 'a';
        for (int i = 0; i < pt.length(); i++) {
            if (pt.charAt(i) == ' ') continue;
            else {
                c = pt.charAt(i);
                s += pt.charAt(i);
```

```
                }
            if (i < pt.length() - 1)
                if (pt.charAt(i) == pt.charAt(i + 1)) s += "x";
        }
        if (s.length() % 2 != 0) s += "x";
        System.out.println(s);
        return s;
    }
    static int[] findIJ(char a, char b, char x[][]) {
        int[] y = new int[4];
        if (a == 'j') a = 'i';
        else if (b == 'j') b = 'i';
        for (int i = 0; i < 5; i++) {
            for (int j = 0; j < 5; j++) {
                if (x[i][j] == a) {
                    y[0] = i;
                    y[1] = j;
                } else if (x[i][j] == b) {
                    y[2] = i;
                    y[3] = j;
                }
            }
        }
        if (y[0] == y[2]) {
            y[1] += 1;
            y[3] += 1;
        } else if (y[1] == y[3]) {
            y[0] += 1;
            y[2] += 1;
        }
        for (int i = 0; i < 4; i++) y[i] %= 5;
        return y;
    }
    static String encrypt(String pt, char x[][]) {
        char ch[] = pt.toCharArray();
        int a[] = new int[4];
        for (int i = 0; i < pt.length(); i += 2) {
            if (i < pt.length() - 1) {
                a = findIJ(pt.charAt(i), pt.charAt(i + 1), x);
                if (a[0] == a[2]) {
                    ch[i] = x[a[0]][a[1]];
                    ch[i + 1] = x[a[0]][a[3]];
                } else if (a[1] == a[3]) {
                    ch[i] = x[a[0]][a[1]];
                    ch[i + 1] = x[a[2]][a[1]];
                } else {
                    ch[i] = x[a[0]][a[3]];
                    ch[i + 1] = x[a[2]][a[1]];
                }
```

```
                }
            }
        pt = new String(ch);
        return pt;
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String pt = "instruments";
        String key = "monarchy";
        key = removeDuplicate(key);
        char[] ch = key.toCharArray();
        String st = "abcdefghiklmnopqrstuvwxyz";
        st = removeWhiteSpace(ch, st);
        char[] c = st.toCharArray();
        char[][] x = new char[5][5];
        int indexOfSt = 0, indexOfKey = 0;
        for (int i = 0; i < 5; i++) {
            for (int j = 0; j < 5; j++) {
                if (indexOfKey < key.length())
                    x[i][j] = ch[indexOfKey++];
                else
                    x[i][j] = c[indexOfSt++];
            }
        }
        for (int i = 0; i < 5; i++) {
            for (int j = 0; j < 5; j++)
                System.out.print(x[i][j] + " ");
            System.out.println();
        }
        pt = makePair(pt);
        pt = encrypt(pt, x);
        System.out.println(pt);
    }
}
```

OUTPUT:

```
m o n a r
c h y b d
e f g i k
l p q s t
u v w x z
instrumentsx
gatlmzclrqxa
```

EXPERIMENT-4

AIM: Write a C/JAVA program to implement the DES algorithm logic.

DESCRIPTION:

The **DES (Data Encryption Standard)** is a symmetric-key block cipher used for encrypting data. It operates on 64-bit blocks of plaintext and uses a 56-bit key for encryption. Here's how it works:

1. **Initial Permutation**: The plaintext is permuted using a fixed table to rearrange the bits.

2. **16 Rounds of Feistel Network**: Each round involves:

   o   Splitting the block into two 32-bit halves.

   o   Applying a series of transformations (expansion, substitution, permutation) using a round-specific key derived from the main key.

   o   XORing one half with the transformed result.

   o   Swapping the two halves.

3. **Final Permutation**: After 16 rounds, the two halves are combined and permuted to produce the ciphertext.
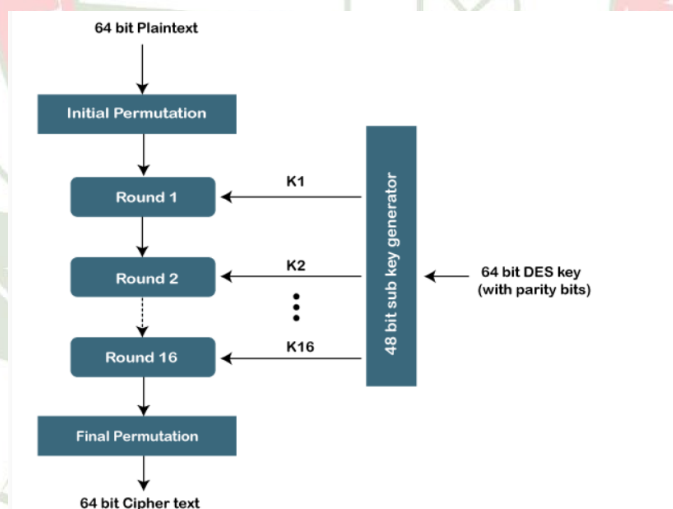
CODE:

```
import java.util.*;
import javax.crypto.BadPaddingException;
import javax.crypto.Cipher;
import javax.crypto.IllegalBlockSizeException;
import javax.crypto.KeyGenerator;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.SecretKey;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.DESKeySpec;
```
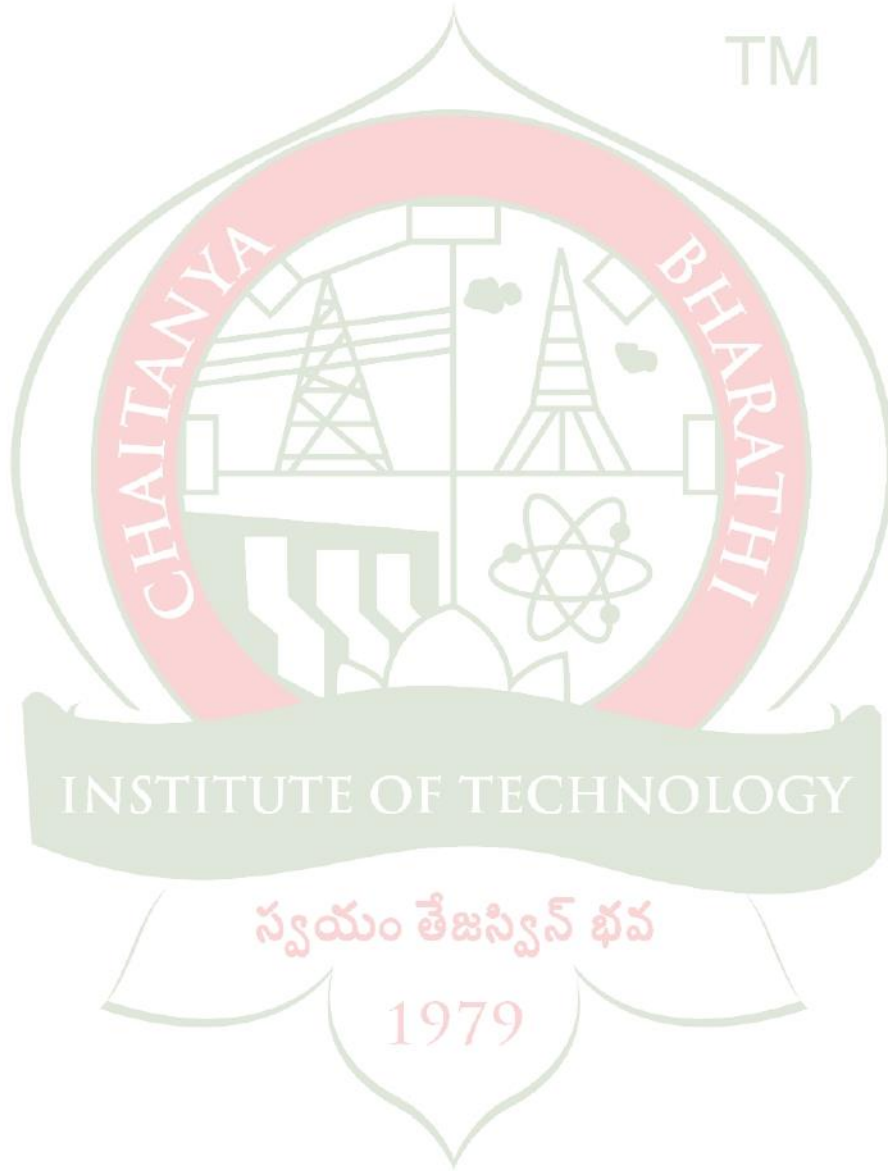
```java
import java.io.*;
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;
import java.security.spec.InvalidKeySpecException;
class des{
    public static void main(String[] args) throws IOException,
NoSuchAlgorithmException, InvalidKeyException,
InvalidKeySpecException, NoSuchPaddingException,
IllegalBlockSizeException, BadPaddingException {
        String  message="This is a confidential message.";
        byte[] myMessage =message.getBytes(); //string to byte
array as DES works on bytes
        //If you want to use your own key
        // SecretKeyFactory MyKeyFactory =
SecretKeyFactory.getInstance("DES");
        // String Password = "My Password";
        // byte[] mybyte =Password.getBytes();
        // DESKeySpec myMaterial = new DESKeySpec(mybyte);
        // SecretKey myDESKey =
MyKeyFactory.generateSecret(myMaterial);
        //Generating Key
        KeyGenerator Mygenerator =
KeyGenerator.getInstance("DES");
        SecretKey myDesKey = Mygenerator.generateKey();
        //initializing crypto algorithm
        Cipher myCipher = Cipher.getInstance("DES");
        //setting encryption mode
        myCipher.init(Cipher.ENCRYPT_MODE, myDesKey);
        byte[] myEncryptedBytes=myCipher.doFinal(myMessage);
        //setting decryption mode
        myCipher.init(Cipher.DECRYPT_MODE, myDesKey);
        byte[]
myDecryptedBytes=myCipher.doFinal(myEncryptedBytes);
        //print message in byte format
        //System.out.println(Arrays.toString(myEncryptedBytes));
        //System.out.println(Arrays.toString(myDecryptedBytes));
        String encrypteddata=new String(myEncryptedBytes);
        String decrypteddata=new String(myDecryptedBytes);
        System.out.println("Message : "+ message);
        System.out.println("Encrypted - "+ encrypteddata);
        System.out.println("Decrypted Message - "+ decrypteddata);
    }
}
```

OUTPUT:

```
Message : This is a confidential message.
Encrypted - ?\n?]?????VD}aw?&&_l?]?↑↓►♥%j?
Decrypted Message - This is a confidential message.
```

EXPERIMENT-5

AIM: Write a C/JAVA program to implement the Blowfish algorithm logic

DESCRIPTION:
Steps in Blowfish Algorithm:

1. **Key Expansion**: The key provided is expanded into several subkeys that are stored in two arrays: the **P-array** and the **S-boxes**. Blowfish uses 18 entries in the P-array and four S-boxes containing 256 entries each.

2. **Rounds of Encryption**: Blowfish processes 16 rounds of encryption, where each round consists of:

   o XORing half the data with a subkey from the P-array.

   o Passing the result through a complex function involving the S-boxes.

   o XORing this result with the other half of the data and swapping halves.

3. **Final Round**: After 16 rounds, the two halves are swapped again, and the output is XORed with two more subkeys to produce the ciphertext.

4. **Decryption**: The decryption process is the reverse of encryption, using the same key and subkeys.

CODE:

```java
import java.util.Base64;
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
public class BlowfishExample {
    public static void main(String[] args) throws Exception {
        String message = "This is a confidential message";
        // Generating a Blowfish key
                            KeyGenerator      keyGenerator      =
KeyGenerator.getInstance("Blowfish");
        keyGenerator.init(128); // key size can be 32-448 bits
        SecretKey secretKey = keyGenerator.generateKey();
        // Initializing cipher for encryption
        Cipher cipher = Cipher.getInstance("Blowfish");
        cipher.init(Cipher.ENCRYPT_MODE, secretKey);
        // Encrypting the message
                            byte[]        encryptedMessage        =
cipher.doFinal(message.getBytes());
                            String      encryptedBase64      =
Base64.getEncoder().encodeToString(encryptedMessage);
        System.out.println("Encrypted Message: " + encryptedBase64);
        // Initializing cipher for decryption
```

```
            cipher.init(Cipher.DECRYPT_MODE, secretKey);
                        byte[]      decryptedMessage      =
cipher.doFinal(Base64.getDecoder().decode(encryptedBase64));
            System.out.println("Decrypted   Message:   "  +  new
String(decryptedMessage));
    }
}
```

OUTPUT:

```
Encrypted Message: eM3oQrDbYO22VGLG9o87neoCv4fbadSGqgtL1hxflx0=
Decrypted Message: This is a confidential message
```
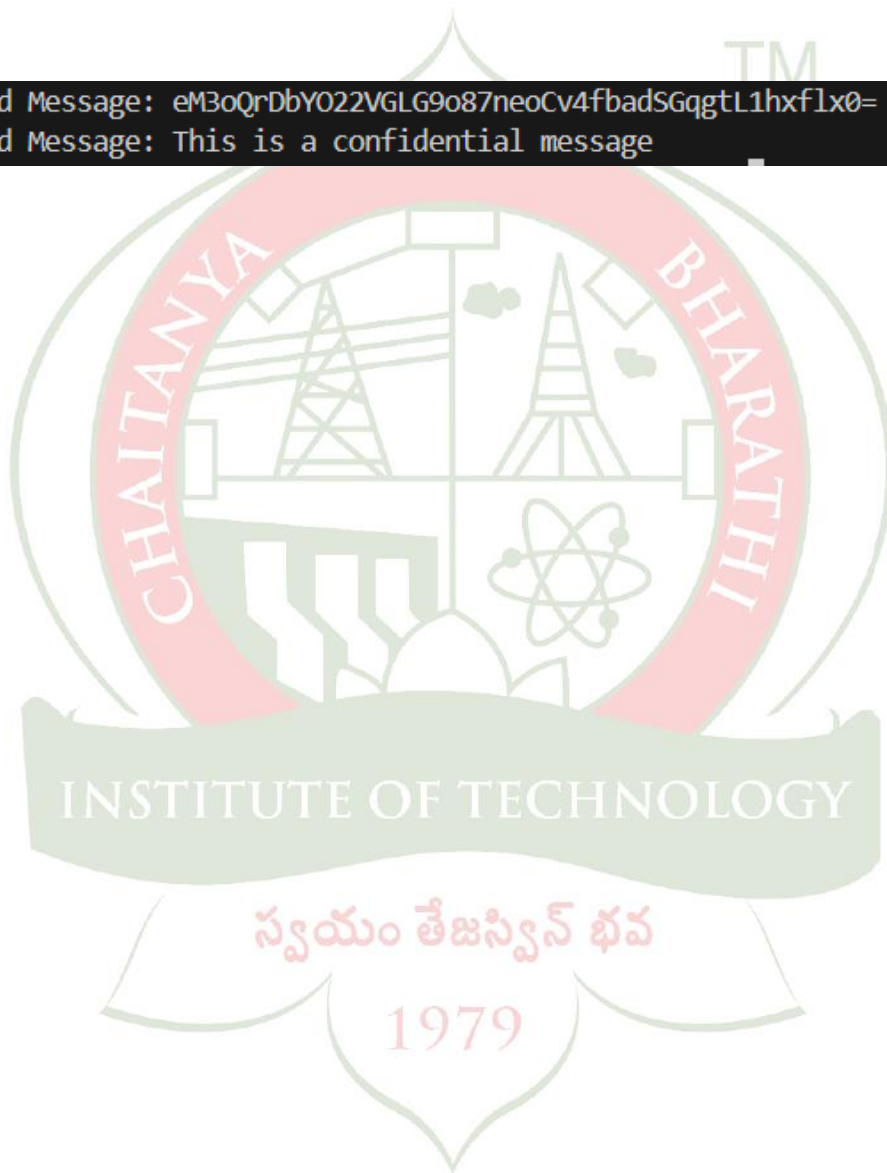
EXPERIMENT-6

AIM: Write a C/JAVA program to implement the Rijndael algorithm logic.

DESCRIPTION:

Rijndael is a symmetric key block cipher that was selected as the Advanced Encryption Standard (AES) by the U.S. National Institute of Standards and Technology (NIST) in 2001. It supports key sizes of 128, 192, and 256 bits, and operates on data blocks of 128 bits (16 bytes). Rijndael is known for its security and efficiency in both software and hardware implementations.

Key Features of Rijndael/AES

- **Block Size:** Fixed at 128 bits.

- **Key Sizes:** Supports 128, 192, or 256 bits.

- **Structure:** Composed of several rounds of processing, including substitution, permutation, and mixing operations.

CODE:

```java
import java.util.Base64;
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
public class AESExample {
    public static void main(String[] args) throws Exception {
        String message = "This is a confidential message";
        // Generating a secret key for AES
        KeyGenerator keyGenerator = KeyGenerator.getInstance("AES");
        keyGenerator.init(128); // key size can be 128, 192, or 256
bits
        SecretKey secretKey = keyGenerator.generateKey();
        // Encrypting the message
        Cipher cipher = Cipher.getInstance("AES");
        cipher.init(Cipher.ENCRYPT_MODE, secretKey);
                             byte[]     encryptedMessage     =
cipher.doFinal(message.getBytes());
                             String     encryptedBase64     =
Base64.getEncoder().encodeToString(encryptedMessage);
        System.out.println("Encrypted Message: " + encryptedBase64);
        // Decrypting the message
        cipher.init(Cipher.DECRYPT_MODE, secretKey);
                             byte[]     decryptedMessage     =
cipher.doFinal(Base64.getDecoder().decode(encryptedBase64));
            System.out.println("Decrypted  Message:  "  +  new
String(decryptedMessage));
```

```
    }
}
```

OUTPUT:

```
Encrypted Message: b83TY1RY2r4/wpqlzE6WfaDhWLsP8HtvhV3LhBM8jpA=
Decrypted Message: This is a confidential message
```
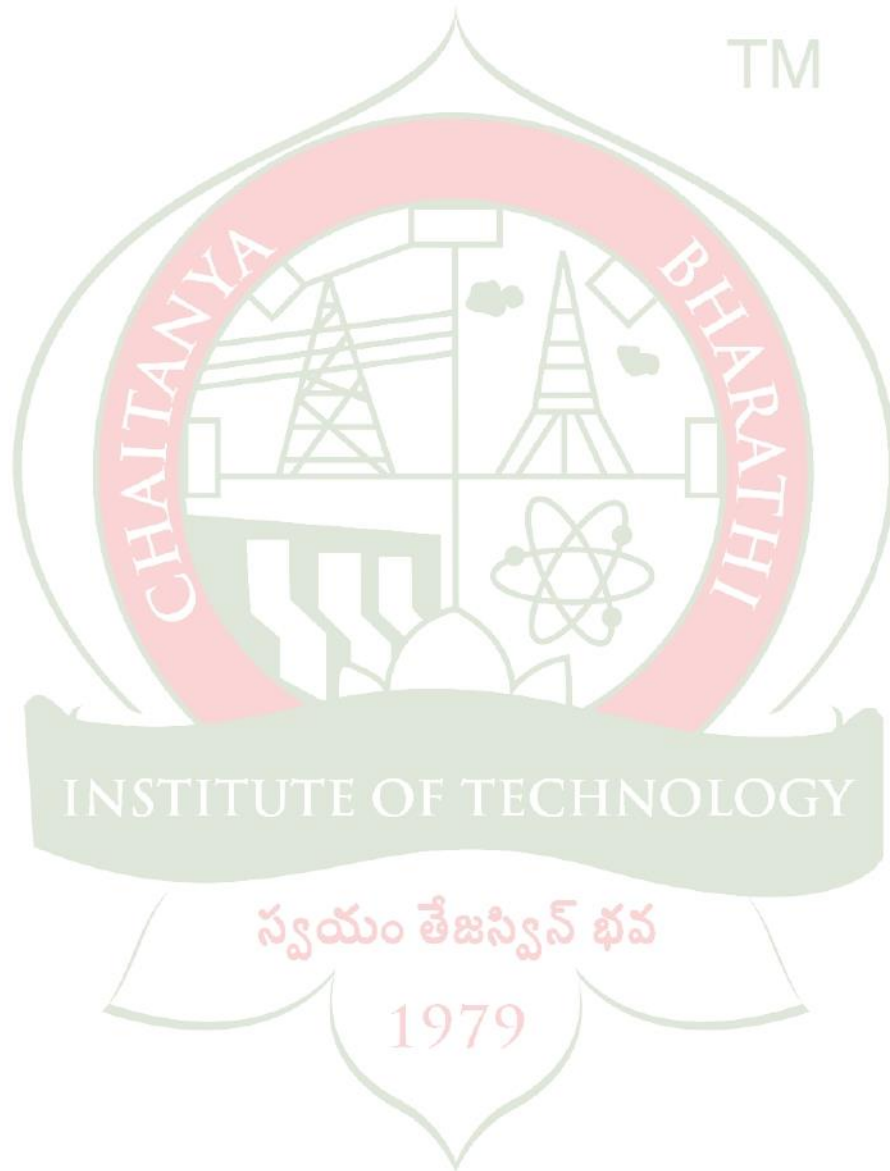
EXPERIMENT-7

AIM: Write the RC4 logic in Java Using Java cryptography.

DESCRIPTION:
RC4 (Rivest Cipher 4) is a symmetric stream cipher designed by Ron Rivest in 1987. It is known for its simplicity and speed, making it a popular choice for various applications, including SSL/TLS and WEP. Here's a brief overview of its key characteristics and working principle:

Key Characteristics

- **Symmetric Key Cipher**: The same key is used for both encryption and decryption.

- **Stream Cipher**: Encrypts data one byte at a time, producing a keystream that is XORed with the plaintext to create ciphertext.

- **Key Size**: Supports variable key lengths, typically ranging from 40 to 2048 bits, but commonly used with 128-bit keys.

- **Simplicity**: The algorithm is relatively straightforward, which allows for efficient implementation.

Working Principle

1. **Key Scheduling Algorithm (KSA)**:

   o Initializes a permutation of all 256 possible byte values (0-255) using the provided key.

   o The key is repeated to fill the array of size 256, and the array is then scrambled based on the key values.

2. **Pseudo-Random Generation Algorithm (PRGA)**:

   o Generates a keystream from the permuted array.

   o This keystream is XORed with the plaintext to produce ciphertext during encryption and with ciphertext to recover plaintext during decryption.

CODE:

```java
import java.util.*;
public class RC4 {
    static int n=3;
    static String plain_text="001010010010";
    static String key="101001000001";
    static List<Integer> S=new ArrayList<>();
    static List<Integer> key_list=new ArrayList<>();
    static List<Integer> pt=new ArrayList<>();
    static List<Integer> key_stream=new ArrayList<>();
```

```java
    static List<Integer> cipher_text=new ArrayList<>();
    static List<Integer> original_text=new ArrayList<>();
    public static void main(String[] args) {
        encryption();
         System.out.println("-----------------------------------------
-------------------");
        decryption();
    }
    // Function for encryption
    public static void encryption() {
        System.out.println("Plain text : "+plain_text);
        System.out.println("Key : "+key);
        System.out.println("n : "+n);
        // The initial state vector array
        for (int i=0;i<Math.pow(2,n);i++) {
            S.add(i);
        }
        System.out.println("S : "+S);
        key_list=convertToDecimal(key);
        pt=convertToDecimal(plain_text);
        System.out.println("Plain text ( in array form ): "+pt);
        // Making key_stream equal to length of state vector
        int diff=S.size()-key_list.size();
        if (diff!=0) {
            for (int i=0;i<diff;i++) {
                key_list.add(key_list.get(i));
            }
        }
        System.out.println("Key list : "+key_list);
        // Perform the KSA algorithm
        KSA();
        // Perform PGRA algorithm
        PGRA();
         // Performing XOR between generated key stream and plain
text
        XOR();
    }
    // Function for decryption of data
    public static void decryption() {
        S.clear();
        key_list.clear();
        pt.clear();
        key_stream.clear();
        // The initial state vector array
        for (int i=0;i<Math.pow(2,n);i++) {
            S.add(i);
        }
        key_list=convertToDecimal(key);
        pt=convertToDecimal(plain_text);
```

```java
        // Making key_stream equal to length of state vector
        int diff=S.size()-key_list.size();
        if (diff!=0) {
            for (int i=0;i<diff;i++) {
                key_list.add(key_list.get(i));
            }
        }
        // KSA algorithm
        KSA();
        // Perform PRGA algorithm
        PGRA();
        // Perform XOR between generated key stream and cipher text
        do_XOR();
    }
    // KSA algorithm
    public static void KSA() {
        int j=0;
        int N=S.size();
        // Iterate over the range [0, N]
        for (int i=0;i<N;i++) {
            j=(j+S.get(i)+key_list.get(i))%N;
            // Update S[i] and S[j]
            Collections.swap(S,i,j);
            System.out.println(i+" "+S);
        }
        System.out.println("The initial permutation array is : "+S);
    }
    // PGRA algorithm
    public static void PGRA() {
        int N=S.size();
        int i=0,j=0;
        // Iterate over [0, length of pt]
        for (int k=0;k<pt.size();k++) {
            i=(i+1)%N;
            j=(j+S.get(i))%N;
            // Update S[i] and S[j]
            Collections.swap(S,i,j);
            System.out.println(k+" "+S);
            int t=(S.get(i)+S.get(j))%N;
            key_stream.add(S.get(t));
        }
        // Print the key stream
        System.out.println("Key stream : "+key_stream);
    }
    // Perform XOR between generated key stream and plain text
    public static void XOR() {
        for (int i=0;i<pt.size();i++) {
            int c=key_stream.get(i)^pt.get(i);
            cipher_text.add(c);
```

```
        }
        // Convert the encrypted text to bits form
        String encrypted_to_bits="";
        for (int i:cipher_text) {
            encrypted_to_bits+=String.format("%0"+n+"d",Integer.pa
rseInt(Integer.toBinaryString(i)));
        }
        System.out.println("Cipher text : "+encrypted_to_bits);
    }
    // Perform XOR between generated key stream and cipher text
    public static void do_XOR() {
        for (int i=0;i<cipher_text.size();i++) {
            int p=key_stream.get(i)^cipher_text.get(i);
            original_text.add(p);
        }
        // Convert the decrypted text to the bits form
        String decrypted_to_bits="";
        for (int i:original_text) {
            decrypted_to_bits+=String.format("%0"+n+"d",Integer.pa
rseInt(Integer.toBinaryString(i)));
        }
        System.out.println("Decrypted text : "+decrypted_to_bits);
    }
    // Convert to decimal
    public static List<Integer> convertToDecimal(String input) {
        List<String> list=new ArrayList<>();
        List<Integer> decimalList=new ArrayList<>();
        for (int i=0;i<input.length();i+=n) {
            list.add(input.substring(i,Math.min(input.length(),i+n
)));
        }
        for (String s:list) {
            decimalList.add(Integer.parseInt(s,2));
        }
        return decimalList;
    }
}
```

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]

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]

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

----

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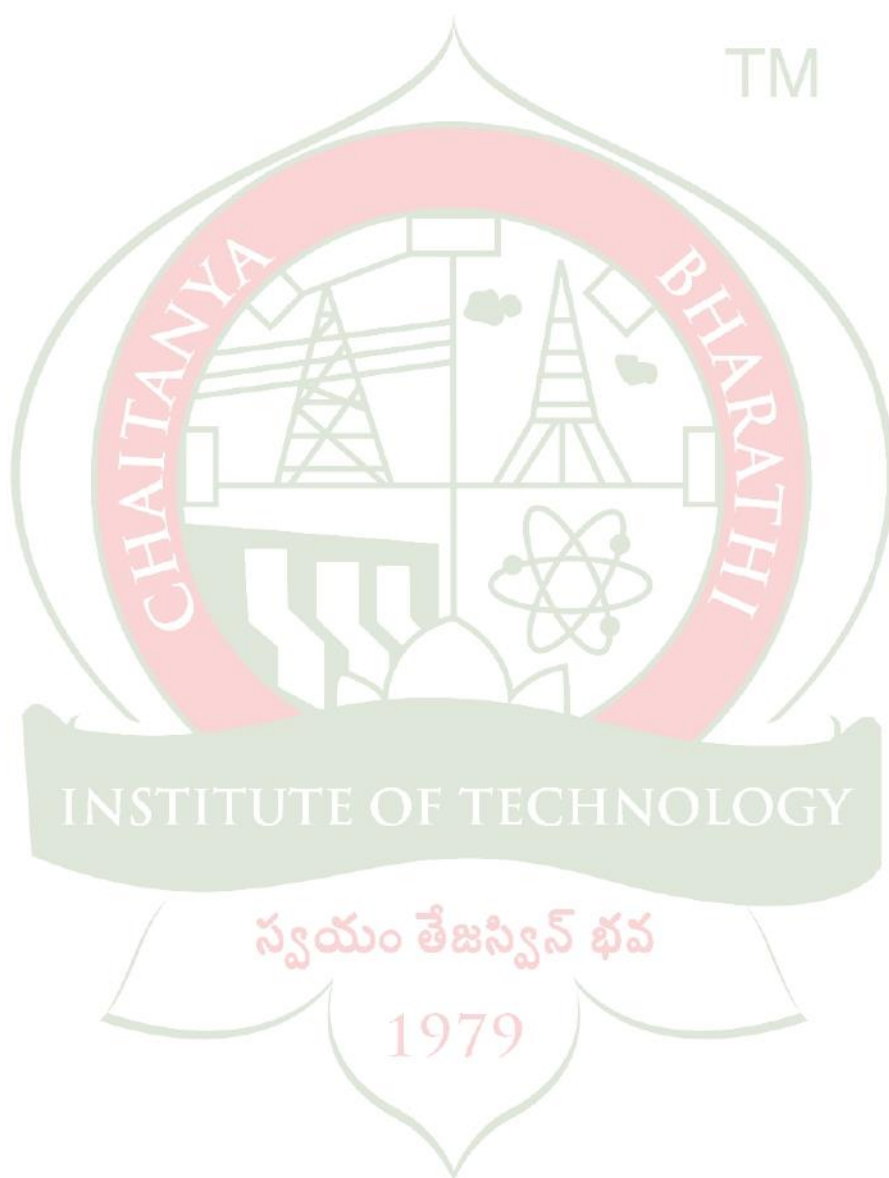
0 [1, 3, 2, 0, 6, 7, 4, 5]

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]

Decrypted text : 001010010010

EXPERIMENT-8

AIM: Write a Java program to implement RSA algorithm.

DESCRIPTION:

RSA (Rivest-Shamir-Adleman) is a widely used asymmetric encryption algorithm that relies on the mathematical properties of prime numbers. Here's a brief description:

1. Key Generation: RSA involves generating a pair of keys—a public key (used for encryption) and a private key (used for decryption). The process includes:

   o Selecting two distinct large prime numbers, p and q

   o Computing n=p×q, which forms part of the public key.

   o Calculating the totient,$\phi(n)=(p-1)(q-1)$.

   o Choosing a public exponent e (typically 65537) such that $1<e<\phi(n)$ and e is coprime to $\phi(n)$.

   o Computing the private exponent d, which is the modular multiplicative inverse of emodulo$\phi(n)$.

2. Encryption: To encrypt a message MMM:

   o Convert the plaintext message to an integer mmm such that $0\leq m<n$.

   o Compute the ciphertext ccc using the public key: c=m^emodn

3. Decryption: To decrypt the ciphertext ccc:

   o Use the private key to compute the original message mmm: m=c^dmodn

   o Convert the integer mmm back to plaintext.

CODE:

```java
// Java Program to Implement the RSA Algorithm
import java.math.*;
class RSA {
    public static void main(String args[])
    {
        int p, q, n, z, d = 0, e, i;
        // The number to be encrypted and decrypted
        int msg = 12;
        double c;
        BigInteger msgback;
        // 1st prime number p
```

```
        p = 3;
        // 2nd prime number q
        q = 11;
        n = p * q;
        z = (p - 1) * (q - 1);
        System.out.println("the value of z = " + z);
        for (e = 2; e < z; e++) {
            // e is for public key exponent
            if (gcd(e, z) == 1) {
                break;
            }
        }
        System.out.println("the value of e = " + e);
        for (i = 0; i <= 9; i++) {
            int x = 1 + (i * z);
            // d is for private key exponent
            if (x % e == 0) {
                d = x / e;
                break;
            }
        }
        System.out.println("the value of d = " + d);
        c = (Math.pow(msg, e)) % n;
        System.out.println("Encrypted message is : " + c);
        // converting int value of n to BigInteger
        BigInteger N = BigInteger.valueOf(n);
        // converting float value of c to BigInteger
        BigInteger C = BigDecimal.valueOf(c).toBigInteger();
        msgback = (C.pow(d)).mod(N);
        System.out.println("Decrypted message is : "
                            + msgback);
    }
    static int gcd(int e, int z)
    {
        if (e == 0)
            return z;
        else
            return gcd(z % e, e);
    }
}
```

OUTPUT:

```
the value of z = 20
the value of e = 3
the value of d = 7
Encrypted message is : 12.0
Decrypted message is : 12
```