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.

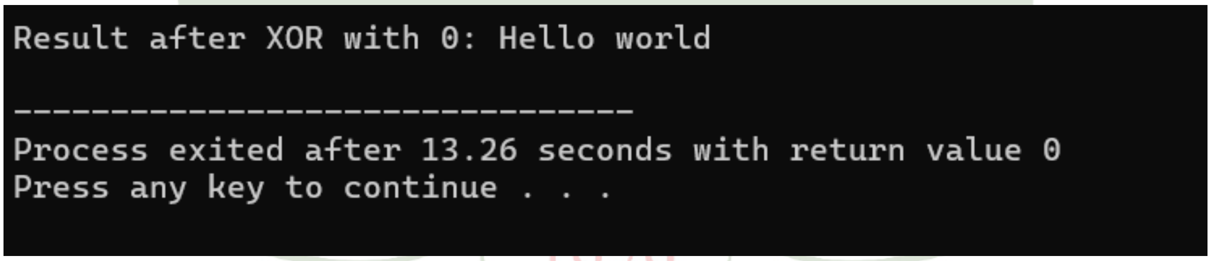
DESCRIPTION:
The operation "XOR each character in this string with 0" means applying the XOR (exclusive OR) bitwise operation to each character of the string using the value `0`. Since XOR with `0` leaves the bits unchanged, each character will remain the same.

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

CODE:

```
#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:

- 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:

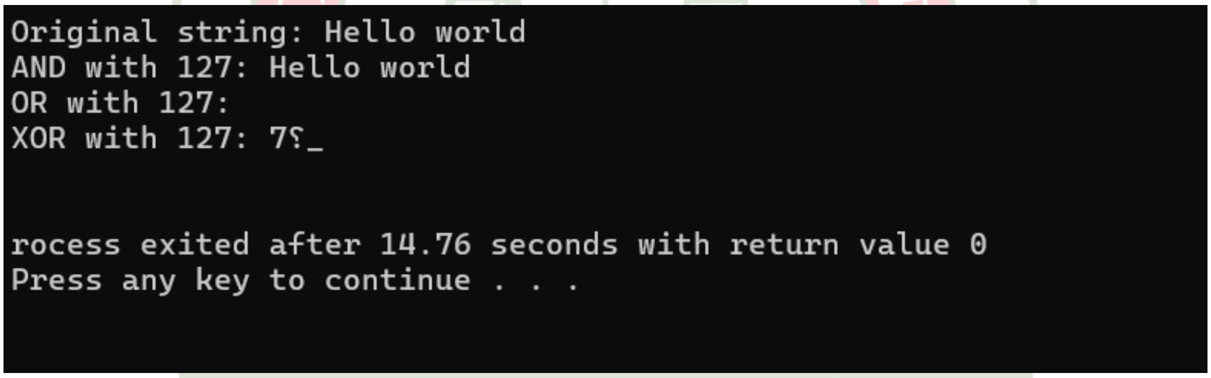
- The XOR operation flips the bits wherever the corresponding bit of the number 127 (01111111) is 1. This will invert the lower 7 bits of each character, drastically changing its ASCII value. For example, if a character is lowercase, it might turn into uppercase or a different symbol depending on its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

```
#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: 7ʕ_


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**

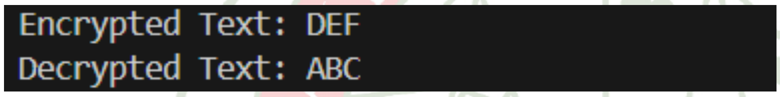
This shifting continues for all letters in the message. When the end of the alphabet is reached, it wraps around (so **Z** would shift to **C** with a shift of 3). Non-letter characters remain unchanged.

CODE:

```
class CaesarCipher:
  @staticmethod
  def encrypt(plaintext, key):
    encrypted_text = []
    for c in plaintext:
      shifted_char = chr(ord(c) + key)
      encrypted_text.append(shifted_char)
    return ''.join(encrypted_text)

  @staticmethod
  def decrypt(encrypted_text, key):
    decrypted_text = []
    for c in encrypted_text:
      shifted_char = chr(ord(c) - key)
      decrypted_text.append(shifted_char)
    return ''.join(decrypted_text)
```

```
if __name__ == "__main__":
    plaintext = "ABC"
    key = 3
    encrypted_text = CaesarCipher.encrypt(plaintext, key)
    print("Encrypted Text:", encrypted_text)
    decrypted_text = CaesarCipher.decrypt(encrypted_text, key)
    print("Decrypted Text:", decrypted_text)
```

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:

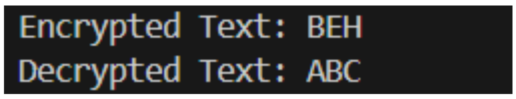
- **Plaintext**: HELLO
- **Ciphertext** (using a random substitution): XNZZT

CODE:

```
class SubCipher:

    @staticmethod

    def encrypt(plaintext, key):

        encrypted_text = []

        for i in range(len(plaintext)):

            shifted_char = chr(ord(plaintext[i]) + key[i])

            encrypted_text.append(shifted_char)

        return ''.join(encrypted_text)


    @staticmethod

    def decrypt(encrypted_text, key):

        decrypted_text = []

        for i in range(len(encrypted_text)):

            shifted_char = chr(ord(encrypted_text[i]) - key[i])

            decrypted_text.append(shifted_char)

        return ''.join(decrypted_text)



if __name__ == "__main__":

    plaintext = "ABC"

    key = [1, 3, 5]
```

```
encrypted_text = SubCipher.encrypt(plaintext, key)
print("Encrypted Text:", encrypted_text)
decrypted_text = SubCipher.decrypt(encrypted_text, key)
print("Decrypted Text:", decrypted_text)
```

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.

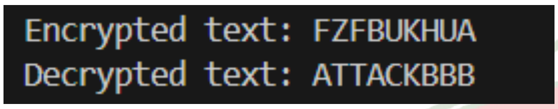
CODE:

```
class HillCipher:
    @staticmethod
    def determinant(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

    @staticmethod
    def mod_inverse(d, mod):
        d = d % mod
        for x in range(1, mod):
            if (d * x) % mod == 1:
                return x
        return 1  # Fallback

    @staticmethod
```

```
def adjoint(matrix):
    adj = [[0] * 3 for _ in range(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 i in range(3):
        for j in range(3):
            if adj[i][j] < 0:
                adj[i][j] += 26
    return adj

@staticmethod
def multiply_matrices(matrix1, matrix2):
    result = [[0] * 3 for _ in range(3)]
    for i in range(3):
        for j in range(3):
            result[i][j] = sum(matrix1[i][k] * matrix2[k][j] for k in range(3)) % 26
            if result[i][j] < 0:
                result[i][j] += 26  # Ensure positive modulo
    return result
    @staticmethod
    def encrypt(plaintext, key):
        plain_matrix = [[ord(plaintext[i * 3 + j]) - ord('A') for j in range(3)] for i in range(3)]
        cipher_matrix = HillCipher.multiply_matrices(key, plain_matrix)

        ciphertext = ''.join(chr(cipher_matrix[i][j] + ord('A')) for i in range(3) for j in range(3))
        return ciphertext

    @staticmethod
    def decrypt(ciphertext, key):
        adj = HillCipher.adjoint(key)
        d = HillCipher.determinant(key)
        d_inverse = HillCipher.mod_inverse(d, 26)

        inverse_key = [[(d_inverse * adj[i][j]) % 26 for j in range(3)] for i in range(3)]
        cipher_matrix = [[ord(ciphertext[i * 3 + j]) - ord('A') for j in range(3)] for i in range(3)]
        plain_matrix = HillCipher.multiply_matrices(inverse_key, cipher_matrix)

        plaintext = ''.join(chr(plain_matrix[i][j] + ord('A')) for i in range(3) for j in range(3))
        return plaintext


if __name__ == "__main__":
    key = [
        [2, 4, 5],
        [9, 2, 1],
        [3, 17, 7]
    ]
    plaintext = "ATTACKBBB"
```

```
ciphertext = HillCipher.encrypt(plaintext, key)
print("Encrypted text:", ciphertext)
decrypted_text = HillCipher.decrypt(ciphertext, key)
print("Decrypted text:", decrypted_text)
```

OUTPUT:



```
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:

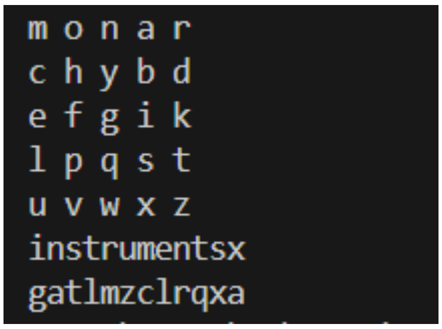
1. If both letters are in the same row, each letter is replaced by the letter to its right.
2. If both letters are in the same column, each letter is replaced by the letter below.
3. If the letters form a rectangle, each is replaced by the letter on the same row but in the other pair's column.

CODE:

```
def remove_duplicate(s):
   result = []
   for char in s:
      if char not in result:
         result.append(char)
   return ''.join(result)


def remove_white_space(ch, key):
   key_list = list(key)
   for i in range(len(key_list)):
      if key_list[i] in ch:
         key_list[i] = ' '
   key = ''.join(key_list)
   key = key.replace(' ', '')
   return key


def make_pair(pt):
   s = ''
```

```
    for i in range(len(pt)):
      if pt[i] == ' ':
        continue
      else:
        s += pt[i]
      if i < len(pt) - 1 and pt[i] == pt[i + 1]:
        s += 'x'
    if len(s) % 2 != 0:
      s += 'x'
    print(s)
    return s


  def find_ij(a, b, matrix):
    y = [-1, -1, -1, -1]
    if a == 'j':
      a = 'i'
    if b == 'j':
      b = 'i'
    for i in range(5):
      for j in range(5):
        if matrix[i][j] == a:
          y[0], y[1] = i, j
        elif matrix[i][j] == b:
          y[2], y[3] = i, j


    if y[0] == y[2]:
      y[1] = (y[1] + 1) % 5
      y[3] = (y[3] + 1) % 5
    elif y[1] == y[3]:
        y[0] = (y[0] + 1) % 5
        y[2] = (y[2] + 1) % 5
    return y


def encrypt(pt, matrix):
    ch = list(pt)
    for i in range(0, len(pt), 2):
        if i < len(pt) - 1:
            a = find_ij(pt[i], pt[i + 1], matrix)
            if a[0] == a[2]:
                ch[i], ch[i + 1] = matrix[a[0]][a[1]], matrix[a[0]][a[3]]
            elif a[1] == a[3]:
                ch[i], ch[i + 1] = matrix[a[0]][a[1]], matrix[a[2]][a[1]]
            else:
                ch[i], ch[i + 1] = matrix[a[0]][a[3]], matrix[a[2]][a[1]]

    return ''.join(ch)


def main():
    pt = "instruments"
    key = "monarchy"
    key = remove_duplicate(key)
    ch = list(key)
    st = "abcdefghiklmnopqrstuvwxyz"  # Excludes 'j'
    st = remove_white_space(ch, st)
    c = list(st)
    matrix = [['' for _ in range(5)] for _ in range(5)]
    index_of_st, index_of_key = 0, 0

    for i in range(5):
        for j in range(5):
            if index_of_key < len(key):
                matrix[i][j] = ch[index_of_key]
                index_of_key += 1
            else:
                matrix[i][j] = c[index_of_st]
                index_of_st += 1

    # Display the matrix
    for row in matrix:
        print(' '.join(row))

    pt = make_pair(pt)
    pt = encrypt(pt, matrix)
    print("Encrypted text:", pt)


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

OUTPUT:



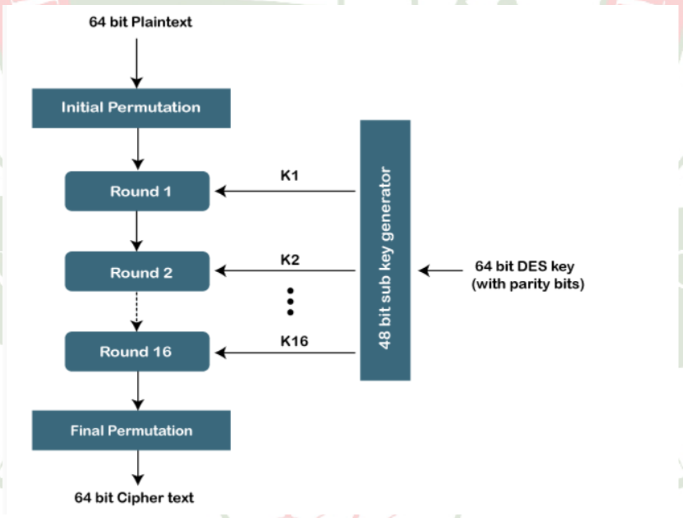
```
m o n a r
c h y b d
e f g i k
l p q s t
u v w x z
instrumentsx
gatlmzclrqxa
```

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:

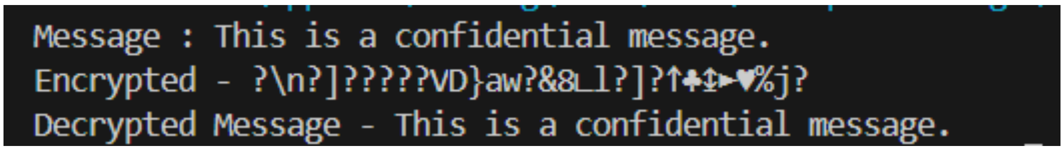
1. **Initial Permutation**: The plaintext is permuted using a fixed table to rearrange the bits.
2. **16 Rounds of Feistel Network**: Each round involves:
   - Splitting the block into two 32-bit halves.
   - Applying a series of transformations (expansion, substitution, permutation) using a round-specific key derived from the main key.
   - XORing one half with the transformed result.
   - Swapping the two halves.
3. **Final Permutation**: After 16 rounds, the two halves are combined and permuted to produce the ciphertext.



CODE:

```
from Crypto.Cipher import DES

from Crypto.Random import get_random_bytes


def pad(text):

    while len(text) % 8 != 0:

        text += ' '
    return text


def main():
    message = "This is a confidential message."
    message = pad(message)
    my_message = message.encode('utf-8')

    my_des_key = get_random_bytes(8)

    my_cipher = DES.new(my_des_key, DES.MODE_ECB)

    my_encrypted_bytes = my_cipher.encrypt(my_message)

    my_cipher = DES.new(my_des_key, DES.MODE_ECB)

    my_decrypted_bytes = my_cipher.decrypt(my_encrypted_bytes)

    encrypted_data = my_encrypted_bytes
    decrypted_data = my_decrypted_bytes.decode('utf-8').strip()

    print("Message: ", message.strip())
    print("Encrypted (in bytes): ", encrypted_data)
    print("Decrypted Message: ", decrypted_data)

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

OUTPUT:



```
Message : This is a confidential message.
Encrypted - ?\n?]?????VD}aw?&8L1?]?↑♣↕►♥%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:

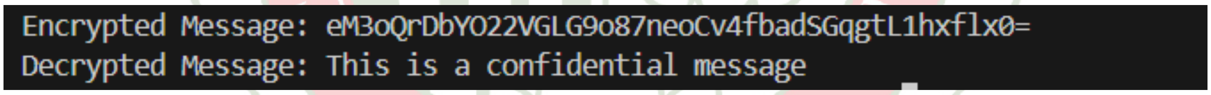
1. **Key Expansion**: The key provided is expanded into several subkeys that are stored in two arrays: the **P-array** and the **S-boxes**. Blowfish uses 18 entries in the P-array and four S-boxes containing 256 entries each.
2. **Rounds of Encryption**: Blowfish processes 16 rounds of encryption, where each round consists of:
    - XORing half the data with a subkey from the P-array.
    - Passing the result through a complex function involving the S-boxes.
    - 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:

```
from Crypto.Cipher import Blowfish

from Crypto.Random import get_random_bytes

import base64


def main():

   message = "This is a confidential message"


   secret_key = get_random_bytes(16)


   cipher = Blowfish.new(secret_key, Blowfish.MODE_ECB)


   while len(message) % 8 != 0:

      message += ' '
```

```
    encrypted_message = cipher.encrypt(message.encode('utf-8'))


    encrypted_base64 = base64.b64encode(encrypted_message).decode('utf-8')
    print("Encrypted Message:", encrypted_base64)


    cipher = Blowfish.new(secret_key, Blowfish.MODE_ECB)


    decrypted_message = cipher.decrypt(base64.b64decode(encrypted_base64)).decode('utf-8').strip()
    print("Decrypted Message:", decrypted_message)


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

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.

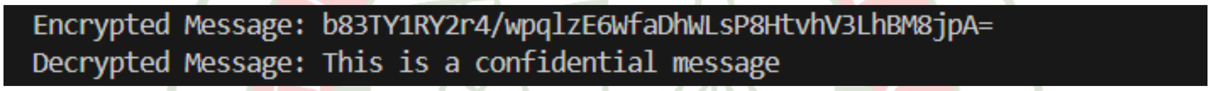
Key Features of Rijndael/AES

- **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:

```
from Crypto.Cipher import AES

from Crypto.Random import get_random_bytes

import base64


def main():

    message = "This is a confidential message"


    secret_key = get_random_bytes(16)  # AES key size of 128 bits (16 bytes)


    cipher = AES.new(secret_key, AES.MODE_ECB)


    while len(message) % 16 != 0:

        message += ' '


    encrypted_message = cipher.encrypt(message.encode('utf-8'))


    encrypted_base64 = base64.b64encode(encrypted_message).decode('utf-8')
    print("Encrypted Message:", encrypted_base64)


    cipher = AES.new(secret_key, AES.MODE_ECB)


    decrypted_message = cipher.decrypt(base64.b64decode(encrypted_base64)).decode('utf-
8').strip()

    print("Decrypted Message:", decrypted_message)


if __name__ == "__main__":

    main()
```

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)**:
   - Initializes a permutation of all 256 possible byte values (0-255) using the provided key.
   - 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)**:
   - Generates a keystream from the permuted array.
   - This keystream is XORed with the plaintext to produce ciphertext during encryption and with ciphertext to recover plaintext during decryption.

CODE:

```
def encryption():

  global S, key_list, pt, key_stream, cipher_text

  print("Plain text : ", plain_text)

  print("Key : ", key)

  print("n : ", n)


  S = [i for i in range(2 ** n)]
```

```
    print("S : ", S)
    key_list = convert_to_decimal(key)
    pt = convert_to_decimal(plain_text)
    print("Plain text (in array form): ", pt)

    diff = len(S) - len(key_list)
    if diff != 0:
        key_list.extend(key_list[:diff])

    print("Key list : ", key_list)

    KSA()
    PRGA()
    XOR()


def decryption():
    global S, key_list, pt, key_stream, original_text
    S, key_list, pt, key_stream = [], [], [], []

    S = [i for i in range(2 ** n)]
    key_list = convert_to_decimal(key)
    pt = convert_to_decimal(plain_text)

    diff = len(S) - len(key_list)
    if diff != 0:
        key_list.extend(key_list[:diff])

    KSA()
    PRGA()
    do_XOR()



def KSA():
    global S
    j = 0
    N = len(S)

    for i in range(N):
        j = (j + S[i] + key_list[i]) % N
        S[i], S[j] = S[j], S[i]
        print(i, S)

    print("The initial permutation array is : ", S)



def PRGA():
    global S, key_stream
    N = len(S)
    i, j = 0, 0

    for k in range(len(pt)):
        i = (i + 1) % N
        j = (j + S[i]) % N
        S[i], S[j] = S[j], S[i]
        print(k, S)
        t = (S[i] + S[j]) % N
        key_stream.append(S[t])
    print("Key stream : ", key_stream)



def XOR():
    global cipher_text
    for i in range(len(pt)):
        c = key_stream[i] ^ pt[i]
        cipher_text.append(c)

    encrypted_to_bits = ''.join([format(i, f'0{n}b') for i in cipher_text])
    print("Cipher text : ", encrypted_to_bits)



def do_XOR():
    global original_text
    for i in range(len(cipher_text)):
        p = key_stream[i] ^ cipher_text[i]
        original_text.append(p)

    decrypted_to_bits = ''.join([format(i, f'0{n}b') for i in original_text])
    print("Decrypted text : ", decrypted_to_bits)



def convert_to_decimal(input_str):
    decimal_list = [int(input_str[i:i+n], 2) for i in range(0, len(input_str), n)]
    return decimal_list
```

```
if __name__ == "__main__":
    n = 3
    plain_text = "001010010010"
    key = "101001000001"
    S, key_list, pt, key_stream, cipher_text, original_text = [], [], [], [], [], []

    encryption()
    print("-----------------------------------------------------------")
    decryption()
```

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:

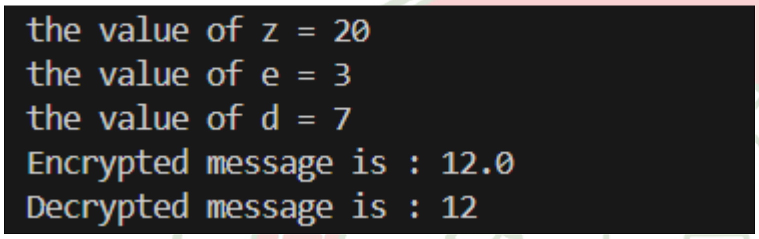
1. Key Generation: RSA involves generating a pair of keys—a public key (used for encryption) and a private key (used for decryption). The process includes:
   - Selecting two distinct large prime numbers, p and q
   - Computing n=p×q, which forms part of the public key.
   - Calculating the totient,φ(n)=(p−1)(q−1).
   - Choosing a public exponent e (typically 65537) such that 1<e<φ(n) and e is coprime to φ(n).
   - Computing the private exponent d, which is the modular multiplicative inverse of emoduloφ(n).
2. Encryption: To encrypt a message MMM:
   - Convert the plaintext message to an integer mmm such that 0≤m<n.
   - Compute the ciphertext ccc using the public key: c=m^emodn
3. Decryption: To decrypt the ciphertext ccc:
   - Use the private key to compute the original message mmm: m=c^dmodn
   - Convert the integer mmm back to plaintext.

CODE:

```
import math

from sympy import mod_inverse


def gcd(a, b):

  if a == 0:

    return b

  else:

    return gcd(b % a, a)
```

```
def main():
    p = 3
    q = 11
    n = p * q
    z = (p - 1) * (q - 1)
    print("The value of z =", z)

    # Finding e (public key exponent)
    for e in range(2, z):
        if gcd(e, z) == 1:
            break
    print("The value of e =", e)

    # Finding d (private key exponent)
    for i in range(1, 10):
        x = 1 + (i * z)
        if x % e == 0:
            d = x // e
            break
    print("The value of d =", d)

    msg = 12
    print("Original message:", msg)

    # Encryption
    c = pow(msg, e, n)
    print("Encrypted message is:", c)
    # Decryption
    msg_back = pow(c, d, n)
    print("Decrypted message is:", msg_back)


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

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
```