Given the plaintext {000102030405060708090A0B0C0D0E0F} and the key {01010101010101010101010101010101}:

a. Show the value of State after SubBytes using any programming language

b. Show the value of State after ShiftRows using any programming language

Shiftrows

```
# Generate the initial matrix from the input data
def gen_matrix(data):
    matrix = [[0] * 4 for x in range(4)]
    row = 3
    col = 3
    for x in range(15,-1,-1):
        matrix[3-row][3-col] =  (data >> (8 * (x))) & 0xFF
        if x % 4 == 0:
            col -= 1
        row = (row - 1) % 4
    return matrix

def shift_rows(matrix):
    def l_rotate_row(rowNum, shiftCount):
        for _ in range(shiftCount):
            temp_byte = matrix[rowNum][0]
            matrix[rowNum][0] = matrix[rowNum][1]
            matrix[rowNum][1] = matrix[rowNum][2]
            matrix[rowNum][2] = matrix[rowNum][3]
            matrix[rowNum][3] = temp_byte

    l_rotate_row(1, 1)
    l_rotate_row(2, 2)
    l_rotate_row(3, 3)

    return matrix

# Given plaintext
plaintext = 0x000102030405060708090A0B0C0D0E0F

# Convert input to matrix
plaintext_matrix = gen_matrix(plaintext)

# Print original input matrix
print("Original Input Matrix:")
for row in plaintext_matrix:
```

```
    print(row)

# Perform ShiftRows on plaintext
plaintext_matrix = shift_rows(plaintext_matrix)

# Print the results after ShiftRows
print("\nPlaintext after ShiftRows:")
for row in plaintext_matrix:
    print(row)
```

Subbytes

```
# AES S-box
s_box = [
    0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe,
0xd7, 0xab, 0x76,
    0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c,
0xa4, 0x72, 0xc0,
    0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71,
0xd8, 0x31, 0x15,
    0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb,
0x27, 0xb2, 0x75,
    0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29,
0xe3, 0x2f, 0x84,
    0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a,
0x4c, 0x58, 0xcf,
    0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50,
0x3c, 0x9f, 0xa8,
    0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10,
0xff, 0xf3, 0xd2,
    0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64,
0x5d, 0x19, 0x73,
    0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde,
0x5e, 0x0b, 0xdb,
    0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91,
0x95, 0xe4, 0x79,
    0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65,
0x7a, 0xae, 0x08,
    0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b,
0xbd, 0x8b, 0x8a,
    0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86,
0xc1, 0x1d, 0x9e,
    0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce,
0x55, 0x28, 0xdf,
```

```python
    0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16
]

# Generate the initial matrix from the input data
def gen_matrix(data):
    matrix = [[0] * 4 for x in range(4)]
    row = 3
    col = 3
    for x in range(15,-1,-1):
        matrix[3-row][3-col] =  (data >> (8 * (x))) & 0xFF
        if x % 4 == 0:
            col -= 1
        row = (row - 1) % 4
    return matrix

def byte_sub(matrix):
    for r in range(4):
        for c in range(4):
            matrix[r][c] = s_box[matrix[r][c]]
    return matrix

# Given plaintext
plaintext = 0x000102030405060708090A0B0C0D0E0F

# Convert input to matrix
plaintext_matrix = gen_matrix(plaintext)

# Print original input matrix
print("Original Input Matrix:")
for row in plaintext_matrix:
    print(row)

# Perform SubBytes on plaintext
plaintext_matrix = byte_sub(plaintext_matrix)

# Print the results after SubBytes
print("\nPlaintext after SubBytes:")
for row in plaintext_matrix:
    print(row)
```

Consider AES with 128-bit keys. Assume that the initial subkey SK0 is all-zero and that the plaintext block is also all-zero. What is the output of the Mix Columns in Round 1? Write your code for the above problem.

```
def mix_columns(state):
    # The AES MixColumns matrix
    matrix = [
        [2, 3, 1, 1],
        [1, 2, 3, 1],
        [1, 1, 2, 3],
        [3, 1, 1, 2]
    ]

    # Perform matrix multiplication
    result = []
    for i in range(4):
        row = []
        for j in range(4):
            val = 0
            for k in range(4):
                val ^= mul(matrix[i][k], state[k][j])
            row.append(val)
        result.append(row)
    return result

def mul(a, b):
    # AES multiplication in Galois Field (GF)
    p = 0
    for _ in range(8):
        if b & 1:
            p ^= a
        hi_bit_set = a & 0x80
        a <<= 1
        if hi_bit_set:
            a ^= 0x1B  # Rijndael's Galois field
        b >>= 1
    return p

def print_state(state):
    for row in state:
        print(' '.join(format(x, '02x') for x in row))

# Initialize the plaintext block as all-zero
plaintext_block = [[0] * 4 for _ in range(4)]
```

```python
# Compute the output of MixColumns in Round 1
mix_columns_output = mix_columns(plaintext_block)

# Print the output
print("Output of MixColumns in Round 1:")
print_state(mix_columns_output)
```

SDES

```python
# Permutation functions
def permute_10(key):
    return [key[2], key[4], key[1], key[6], key[3], key[9], key[0], key[8],
key[7], key[5]]

def permute_8(key):
    return [key[5], key[2], key[6], key[3], key[7], key[4], key[9], key[8]]

def permute_4(key):
    return [key[1], key[3], key[2], key[0]]

def expand_permute(key):
    return [key[3], key[0], key[1], key[2], key[1], key[2], key[3], key[0]]

def initial_permute(plaintext):
    return [plaintext[1], plaintext[5], plaintext[2], plaintext[0], plaintext[3],
plaintext[7], plaintext[4], plaintext[6]]

def inverse_permute(plaintext):
    return [plaintext[3], plaintext[0], plaintext[2], plaintext[4], plaintext[6],
plaintext[1], plaintext[7], plaintext[5]]

# S-Boxes
s_box_0 = [
    ['01', '00', '11', '10'],
    ['11', '10', '01', '00'],
    ['00', '10', '01', '11'],
    ['11', '01', '00', '10']
]

s_box_1 = [
    ['00', '01', '10', '11'],
    ['10', '00', '01', '11'],
```

```python
    ['11', '00', '01', '00'],
    ['10', '01', '00', '11']
]

# Key generation
def generate_keys(key):
    keys = []
    key = [int(bit) for bit in key]
    key = permute_10(key)
    left_key = key[:5]
    right_key = key[5:]

    # First round of key generation
    left_key = left_shift(left_key, 1)
    right_key = left_shift(right_key, 1)
    keys.append(permute_8(left_key + right_key))

    # Second round of key generation
    left_key = left_shift(left_key, 2)
    right_key = left_shift(right_key, 2)
    keys.append(permute_8(left_key + right_key))

    return keys

# Left shift operation
def left_shift(bits, amount):
    return bits[amount:] + bits[:amount]

# XOR operation
def xor(bits1, bits2):
    return [str(int(bit1) ^ int(bit2)) for bit1, bit2 in zip(bits1, bits2)]

# S-Box substitution
def s_box_substitution(bits, s_box):
    row = int(bits[0] + bits[3], 2)
    col = int(bits[1] + bits[2], 2)
    return [int(bit) for bit in list(bin(int(s_box[row][col], 2))[2:].zfill(2))]

# S-DES encryption
def s_des_encrypt(plaintext, keys):
    plaintext = [int(bit) for bit in plaintext]
    plaintext = initial_permute(plaintext)

    left_half = plaintext[:4]
    right_half = plaintext[4:]
```

```python
    # First round of encryption
    expanded_right_half = expand_permute(right_half)
    xor_result = xor(expanded_right_half, keys[0])
    s_box_0_output = s_box_substitution(xor_result[:4], s_box_0)
    s_box_1_output = s_box_substitution(xor_result[4:], s_box_1)
    p4_input = s_box_0_output + s_box_1_output
    p4_output = permute_4(p4_input)
    new_right_half = xor(left_half, p4_output)
    new_left_half = right_half

    # Second round of encryption
    xor_result = xor(expanded_right_half, keys[1])
    s_box_0_output = s_box_substitution(xor_result[:4], s_box_0)
    s_box_1_output = s_box_substitution(xor_result[4:], s_box_1)
    p4_input = s_box_0_output + s_box_1_output
    p4_output = permute_4(p4_input)
    new_right_half = xor(new_right_half, p4_output)

    ciphertext = new_left_half + new_right_half
    ciphertext = inverse_permute(ciphertext)
    return ''.join(map(str, ciphertext))

# Main function
def main():
    key = '1010000010'
    plaintext = '01110010'
    keys = generate_keys(key)
    ciphertext = s_des_encrypt(plaintext, keys)
    print("Ciphertext:", ciphertext)

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

Ceaser cipher

```python
#A python program to illustrate Caesar Cipher Technique
def encrypt(text,s):
    result = ""

    # traverse text
    for i in range(len(text)):
        char = text[i]
```

```python
        # Encrypt uppercase characters
        if (char.isupper()):
            result += chr((ord(char) + s-65) % 26 + 65)

        # Encrypt lowercase characters
        else:
            result += chr((ord(char) + s - 97) % 26 + 97)

    return result

#check the above function
text = "ATTACKATONCE"
s = 4
print ("Text : " + text)
print ("Shift : " + str(s))
print ("Cipher: " + encrypt(text,s))
```

Playfair Cipher

```python
# Python program to implement Playfair Cipher

# Function to convert the string to lowercase


def toLowerCase(text):
    return text.lower()

# Function to remove all spaces in a string


def removeSpaces(text):
    newText = ""
    for i in text:
        if i == " ":
            continue
        else:
            newText = newText + i
    return newText

# Function to group 2 elements of a string
# as a list element
```

```python
def Diagraph(text):
    Diagraph = []
    group = 0
    for i in range(2, len(text), 2):
        Diagraph.append(text[group:i])

        group = i
    Diagraph.append(text[group:])
    return Diagraph

# Function to fill a letter in a string element
# If 2 letters in the same string matches


def FillerLetter(text):
    k = len(text)
    if k % 2 == 0:
        for i in range(0, k, 2):
            if text[i] == text[i+1]:
                new_word = text[0:i+1] + str('x') + text[i+1:]
                new_word = FillerLetter(new_word)
                break
            else:
                new_word = text
    else:
        for i in range(0, k-1, 2):
            if text[i] == text[i+1]:
                new_word = text[0:i+1] + str('x') + text[i+1:]
                new_word = FillerLetter(new_word)
                break
            else:
                new_word = text
    return new_word


list1 = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'k', 'l', 'm',
        'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']

# Function to generate the 5x5 key square matrix


def generateKeyTable(word, list1):
    key_letters = []
    for i in word:
        if i not in key_letters:
```

```
            key_letters.append(i)

    compElements = []
    for i in key_letters:
        if i not in compElements:
            compElements.append(i)
    for i in list1:
        if i not in compElements:
            compElements.append(i)

    matrix = []
    while compElements != []:
        matrix.append(compElements[:5])
        compElements = compElements[5:]

    return matrix


def search(mat, element):
    for i in range(5):
        for j in range(5):
            if(mat[i][j] == element):
                return i, j


def encrypt_RowRule(matr, e1r, e1c, e2r, e2c):
    char1 = ''
    if e1c == 4:
        char1 = matr[e1r][0]
    else:
        char1 = matr[e1r][e1c+1]

    char2 = ''
    if e2c == 4:
        char2 = matr[e2r][0]
    else:
        char2 = matr[e2r][e2c+1]

    return char1, char2


def encrypt_ColumnRule(matr, e1r, e1c, e2r, e2c):
    char1 = ''
    if e1r == 4:
        char1 = matr[0][e1c]
```

```python
    else:
        char1 = matr[e1r+1][e1c]

    char2 = ''
    if e2r == 4:
        char2 = matr[0][e2c]
    else:
        char2 = matr[e2r+1][e2c]

    return char1, char2


def encrypt_RectangleRule(matr, e1r, e1c, e2r, e2c):
    char1 = ''
    char1 = matr[e1r][e2c]

    char2 = ''
    char2 = matr[e2r][e1c]

    return char1, char2


def encryptByPlayfairCipher(Matrix, plainList):
    CipherText = []
    for i in range(0, len(plainList)):
        c1 = 0
        c2 = 0
        ele1_x, ele1_y = search(Matrix, plainList[i][0])
        ele2_x, ele2_y = search(Matrix, plainList[i][1])

        if ele1_x == ele2_x:
            c1, c2 = encrypt_RowRule(Matrix, ele1_x, ele1_y, ele2_x, ele2_y)
            # Get 2 letter cipherText
        elif ele1_y == ele2_y:
            c1, c2 = encrypt_ColumnRule(Matrix, ele1_x, ele1_y, ele2_x, ele2_y)
        else:
            c1, c2 = encrypt_RectangleRule(
                Matrix, ele1_x, ele1_y, ele2_x, ele2_y)

        cipher = c1 + c2
        CipherText.append(cipher)
    return CipherText


text_Plain = 'instruments'
```

```
text_Plain = removeSpaces(toLowerCase(text_Plain))
PlainTextList = Diagraph(FillerLetter(text_Plain))
if len(PlainTextList[-1]) != 2:
    PlainTextList[-1] = PlainTextList[-1]+'z'

key = "Monarchy"
print("Key text:", key)
key = toLowerCase(key)
Matrix = generateKeyTable(key, list1)

print("Plain Text:", text_Plain)
CipherList = encryptByPlayfairCipher(Matrix, PlainTextList)

CipherText = ""
for i in CipherList:
    CipherText += i
print("CipherText:", CipherText)
```

Hill Cipher

```
# Python3 code to implement Hill Cipher

keyMatrix = [[0] * 3 for i in range(3)]

# Generate vector for the message
messageVector = [[0] for i in range(3)]

# Generate vector for the cipher
cipherMatrix = [[0] for i in range(3)]

# Following function generates the
# key matrix for the key string
def getKeyMatrix(key):
    k = 0
    for i in range(3):
        for j in range(3):
            keyMatrix[i][j] = ord(key[k]) % 65
            k += 1

# Following function encrypts the message
def encrypt(messageVector):
    for i in range(3):
        for j in range(1):
```

```python
            cipherMatrix[i][j] = 0
            for x in range(3):
                cipherMatrix[i][j] += (keyMatrix[i][x] *
                                       messageVector[x][j])
            cipherMatrix[i][j] = cipherMatrix[i][j] % 26

def HillCipher(message, key):

    # Get key matrix from the key string
    getKeyMatrix(key)

    # Generate vector for the message
    for i in range(3):
        messageVector[i][0] = ord(message[i]) % 65

    # Following function generates
    # the encrypted vector
    encrypt(messageVector)

    # Generate the encrypted text
    # from the encrypted vector
    CipherText = []
    for i in range(3):
        CipherText.append(chr(cipherMatrix[i][0] + 65))

    # Finally print the ciphertext
    print("Ciphertext: ", "".join(CipherText))

# Driver Code
def main():

    # Get the message to
    # be encrypted
    message = "ACT"

    # Get the key
    key = "GYBNQKURP"

    HillCipher(message, key)

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

Vigenere Cipher

```python
# Python code to implement
# Vigenere Cipher

# This function generates the
# key in a cyclic manner until
# it's length isn't equal to
# the length of original text
def generateKey(string, key):
    key = list(key)
    if len(string) == len(key):
        return(key)
    else:
        for i in range(len(string) -
                       len(key)):
            key.append(key[i % len(key)])
    return("" . join(key))

# This function returns the
# encrypted text generated
# with the help of the key
def cipherText(string, key):
    cipher_text = []
    for i in range(len(string)):
        x = (ord(string[i]) +
            ord(key[i])) % 26
        x += ord('A')
        cipher_text.append(chr(x))
    return("" . join(cipher_text))

# This function decrypts the
# encrypted text and returns
# the original text
def originalText(cipher_text, key):
    orig_text = []
    for i in range(len(cipher_text)):
        x = (ord(cipher_text[i]) -
            ord(key[i]) + 26) % 26
        x += ord('A')
        orig_text.append(chr(x))
    return("" . join(orig_text))

# Driver code
if __name__ == "__main__":
    string = "GEEKSFORGEEKS"
    keyword = "AYUSH"
```

```python
    key = generateKey(string, keyword)
    cipher_text = cipherText(string,key)
    print("Ciphertext :", cipher_text)
    print("Original/Decrypted Text :",
        originalText(cipher_text, key))
```

DSA

```python
# Python3 code for the above approach

# Hexadecimal to binary conversion


def hex2bin(s):
    mp = {'0': "0000",
        '1': "0001",
        '2': "0010",
        '3': "0011",
        '4': "0100",
        '5': "0101",
        '6': "0110",
        '7': "0111",
        '8': "1000",
        '9': "1001",
        'A': "1010",
        'B': "1011",
        'C': "1100",
        'D': "1101",
        'E': "1110",
        'F': "1111"}
    bin = ""
    for i in range(len(s)):
        bin = bin + mp[s[i]]
    return bin

# Binary to hexadecimal conversion


def bin2hex(s):
    mp = {"0000": '0',
        "0001": '1',
        "0010": '2',
        "0011": '3',
        "0100": '4',
        "0101": '5',
```

```
        "0110": '6',
        "0111": '7',
        "1000": '8',
        "1001": '9',
        "1010": 'A',
        "1011": 'B',
        "1100": 'C',
        "1101": 'D',
        "1110": 'E',
        "1111": 'F'}
    hex = ""
    for i in range(0, len(s), 4):
        ch = ""
        ch = ch + s[i]
        ch = ch + s[i + 1]
        ch = ch + s[i + 2]
        ch = ch + s[i + 3]
        hex = hex + mp[ch]

    return hex

# Binary to decimal conversion


def bin2dec(binary):

    binary1 = binary
    decimal, i, n = 0, 0, 0
    while(binary != 0):
        dec = binary % 10
        decimal = decimal + dec * pow(2, i)
        binary = binary//10
        i += 1
    return decimal

# Decimal to binary conversion


def dec2bin(num):
    res = bin(num).replace("0b", "")
    if(len(res) % 4 != 0):
        div = len(res) / 4
        div = int(div)
        counter = (4 * (div + 1)) - len(res)
        for i in range(0, counter):
```

```python
            res = '0' + res
    return res

# Permute function to rearrange the bits


def permute(k, arr, n):
    permutation = ""
    for i in range(0, n):
        permutation = permutation + k[arr[i] - 1]
    return permutation

# shifting the bits towards left by nth shifts


def shift_left(k, nth_shifts):
    s = ""
    for i in range(nth_shifts):
        for j in range(1, len(k)):
            s = s + k[j]
        s = s + k[0]
        k = s
        s = ""
    return k

# calculating xow of two strings of binary number a and b


def xor(a, b):
    ans = ""
    for i in range(len(a)):
        if a[i] == b[i]:
            ans = ans + "0"
        else:
            ans = ans + "1"
    return ans


# Table of Position of 64 bits at initial level: Initial Permutation Table
initial_perm = [58, 50, 42, 34, 26, 18, 10, 2,
                60, 52, 44, 36, 28, 20, 12, 4,
                62, 54, 46, 38, 30, 22, 14, 6,
                64, 56, 48, 40, 32, 24, 16, 8,
                57, 49, 41, 33, 25, 17, 9, 1,
                59, 51, 43, 35, 27, 19, 11, 3,
                61, 53, 45, 37, 29, 21, 13, 5,
```

```
                63, 55, 47, 39, 31, 23, 15, 7]

# Expansion D-box Table
exp_d = [32, 1, 2, 3, 4, 5, 4, 5,
        6, 7, 8, 9, 8, 9, 10, 11,
        12, 13, 12, 13, 14, 15, 16, 17,
        16, 17, 18, 19, 20, 21, 20, 21,
        22, 23, 24, 25, 24, 25, 26, 27,
        28, 29, 28, 29, 30, 31, 32, 1]

# Straight Permutation Table
per = [16, 7, 20, 21,
    29, 12, 28, 17,
    1, 15, 23, 26,
    5, 18, 31, 10,
    2, 8, 24, 14,
    32, 27, 3, 9,
    19, 13, 30, 6,
    22, 11, 4, 25]

# S-box Table
sbox = [[[14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7],
        [0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8],
        [4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0],
        [15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13]],

        [[15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10],
        [3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5],
        [0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15],
        [13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9]],

        [[10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8],
        [13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1],
        [13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7],
        [1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12]],

        [[7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15],
        [13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9],
        [10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4],
        [3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14]],

        [[2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9],
        [14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6],
        [4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14],
        [11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3]],
```

```
        [[12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11],
        [10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8],
        [9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6],
        [4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13]],

        [[4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1],
        [13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6],
        [1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2],
        [6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12]],

        [[13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7],
        [1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2],
        [7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8],
        [2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11]]]

# Final Permutation Table
final_perm = [40, 8, 48, 16, 56, 24, 64, 32,
            39, 7, 47, 15, 55, 23, 63, 31,
            38, 6, 46, 14, 54, 22, 62, 30,
            37, 5, 45, 13, 53, 21, 61, 29,
            36, 4, 44, 12, 52, 20, 60, 28,
            35, 3, 43, 11, 51, 19, 59, 27,
            34, 2, 42, 10, 50, 18, 58, 26,
            33, 1, 41, 9, 49, 17, 57, 25]


def encrypt(pt, rkb, rk):
    pt = hex2bin(pt)

    # Initial Permutation
    pt = permute(pt, initial_perm, 64)
    print("After initial permutation", bin2hex(pt))

    # Splitting
    left = pt[0:32]
    right = pt[32:64]
    for i in range(0, 16):
        # Expansion D-box: Expanding the 32 bits data into 48 bits
        right_expanded = permute(right, exp_d, 48)

        # XOR RoundKey[i] and right_expanded
        xor_x = xor(right_expanded, rkb[i])
```

```
        # S-boxex: substituting the value from s-box table by calculating row and
column
        sbox_str = ""
        for j in range(0, 8):
            row = bin2dec(int(xor_x[j * 6] + xor_x[j * 6 + 5]))
            col = bin2dec(
                int(xor_x[j * 6 + 1] + xor_x[j * 6 + 2] + xor_x[j * 6 + 3] +
xor_x[j * 6 + 4]))
            val = sbox[j][row][col]
            sbox_str = sbox_str + dec2bin(val)

        # Straight D-box: After substituting rearranging the bits
        sbox_str = permute(sbox_str, per, 32)

        # XOR left and sbox_str
        result = xor(left, sbox_str)
        left = result

        # Swapper
        if(i != 15):
            left, right = right, left
        print("Round ", i + 1, " ", bin2hex(left),
            " ", bin2hex(right), " ", rk[i])

    # Combination
    combine = left + right

    # Final permutation: final rearranging of bits to get cipher text
    cipher_text = permute(combine, final_perm, 64)
    return cipher_text


pt = "123456ABCD132536"
key = "AABB09182736CCDD"

# Key generation
# --hex to binary
key = hex2bin(key)

# --parity bit drop table
keyp = [57, 49, 41, 33, 25, 17, 9,
        1, 58, 50, 42, 34, 26, 18,
        10, 2, 59, 51, 43, 35, 27,
        19, 11, 3, 60, 52, 44, 36,
        63, 55, 47, 39, 31, 23, 15,
```

```
        7, 62, 54, 46, 38, 30, 22,
        14, 6, 61, 53, 45, 37, 29,
        21, 13, 5, 28, 20, 12, 4]

# getting 56 bit key from 64 bit using the parity bits
key = permute(key, keyp, 56)

# Number of bit shifts
shift_table = [1, 1, 2, 2,
            2, 2, 2, 2,
            1, 2, 2, 2,
            2, 2, 2, 1]

# Key- Compression Table : Compression of key from 56 bits to 48 bits
key_comp = [14, 17, 11, 24, 1, 5,
            3, 28, 15, 6, 21, 10,
            23, 19, 12, 4, 26, 8,
            16, 7, 27, 20, 13, 2,
            41, 52, 31, 37, 47, 55,
            30, 40, 51, 45, 33, 48,
            44, 49, 39, 56, 34, 53,
            46, 42, 50, 36, 29, 32]

# Splitting
left = key[0:28] # rkb for RoundKeys in binary
right = key[28:56] # rk for RoundKeys in hexadecimal

rkb = []
rk = []
for i in range(0, 16):
    # Shifting the bits by nth shifts by checking from shift table
    left = shift_left(left, shift_table[i])
    right = shift_left(right, shift_table[i])

    # Combination of left and right string
    combine_str = left + right

    # Compression of key from 56 to 48 bits
    round_key = permute(combine_str, key_comp, 48)

    rkb.append(round_key)
    rk.append(bin2hex(round_key))

print("Encryption")
cipher_text = bin2hex(encrypt(pt, rkb, rk))
```

```
print("Cipher Text : ", cipher_text)

print("Decryption")
rkb_rev = rkb[::-1]
rk_rev = rk[::-1]
text = bin2hex(encrypt(cipher_text, rkb_rev, rk_rev))
print("Plain Text : ", text)

# This code is contributed by Aditya Jain
```

RSA

```
from math import gcd

# defining a function to perform RSA approch
def RSA(p: int, q: int, message: int):
    # calculating n
    n = p * q

    # calculating totient, t
    t = (p - 1) * (q - 1)

    # selecting public key, e
    for i in range(2, t):
        if gcd(i, t) == 1:
            e = i
            break

    # selecting private key, d
    j = 0
    while True:
        if (j * e) % t == 1:
            d = j
            break
        j += 1

    # performing encryption
    ct = (message ** e) % n
    print(f"Encrypted message is {ct}")

    # performing decryption
    mes = (ct ** d) % n
```

```
    print(f"Decrypted message is {mes}")

# Testcase - 1
RSA(p=53, q=59, message=89)

# Testcase - 2
RSA(p=3, q=7, message=12)
```

Deffie Hellman with man in the middle attack

```
import random

# public keys are taken
# p is a prime number
# g is a primitive root of p
p = int(input('Enter a prime number : '))
g = int(input('Enter a number : '))


class A:
    def __init__(self):
        # Generating a random private number selected by alice
        self.n = random.randint(1, p)

    def publish(self):
        # generating public values
        return (g**self.n)%p

    def compute_secret(self, gb):
        # computing secret key
        return (gb**self.n)%p


class B:
    def __init__(self):
        # Generating a random private number selected for alice
        self.a = random.randint(1, p)
        # Generating a random private number selected for bob
        self.b = random.randint(1, p)
        self.arr = [self.a,self.b]

    def publish(self, i):
        # generating public values
        return (g**self.arr[i])%p
```

```
    def compute_secret(self, ga, i):
        # computing secret key
        return (ga**self.arr[i])%p


alice = A()
bob = A()
eve = B()

# Printing out the private selected number by Alice and Bob
print(f'Alice selected (a) : {alice.n}')
print(f'Bob selected (b) : {bob.n}')
print(f'Eve selected private number for Alice (c) : {eve.a}')
print(f'Eve selected private number for Bob (d) : {eve.b}')

# Generating public values
ga = alice.publish()
gb = bob.publish()
gea = eve.publish(0)
geb = eve.publish(1)
print(f'Alice published (ga): {ga}')
print(f'Bob published (gb): {gb}')
print(f'Eve published value for Alice (gc): {gea}')
print(f'Eve published value for Bob (gd): {geb}')

# Computing the secret key
sa = alice.compute_secret(gea)
sea = eve.compute_secret(ga,0)
sb = bob.compute_secret(geb)
seb = eve.compute_secret(gb,1)
print(f'Alice computed (S1) : {sa}')
print(f'Eve computed key for Alice (S1) : {sea}')
print(f'Bob computed (S2) : {sb}')
print(f'Eve computed key for Bob (S2) : {seb}')
```

AES

```
AES 256 Encryption in Python
#Generate the initial matrix from the input data
def gen_matrix(data):
    matrix = [[0] * 4 for x in range(4)]
    #Generate the 4x4 input matrix
    row = 3
    col = 3
```

```
    for x in range(15,-1,-1):
        matrix[3-row][3-col] =  (data >> (8 * (x))) & 0xFF
        if x % 4 == 0:
            col -= 1
        row = (row - 1) % 4
    return matrix

#Add round key operation
def add_round_key(key):
    global matrix
    key_matrix = gen_matrix(key)
    for r in range(4):
        for c in range(4):
            matrix[r][c] = matrix[r][c] ^ key_matrix[r][c]
    dumpMatrix("add round keys")

for round in range(len(round_keys)-1):
    #Byte substitution
    byte_sub()

    #Shift rows
    shift_rows()

    # Final round doesn't include mix columns
    if round < 13:
        #Mix columns
        mix_columns()

    #Add round key
    add_round_key(round_keys[round+1])

#Perform the byte substitution layer
def byte_sub():
    global matrix
    for r in range(4):
        for c in range(4):
            t = matrix[r][c]
            matrix[r][c] = sbox(matrix[r][c])
    dumpMatrix("Byte sub.")

#Helper function for a leftward rotation of a matrix row
def l_rotate_row(rowNum, shiftCount):
    for x in range(shiftCount):
        global matrix
        temp_byte = matrix[rowNum][0]
```

```
        matrix[rowNum][0] = matrix[rowNum][1]
        matrix[rowNum][1] = matrix[rowNum][2]
        matrix[rowNum][2] = matrix[rowNum][3]
        matrix[rowNum][3] = temp_byte

#Shift rows operation
def shift_rows():
    global matrix
    l_rotate_row(1, 1)
    l_rotate_row(2, 2)
    l_rotate_row(3, 3)
    dumpMatrix("shift rows")

#Mix columns operation
def mix_columns():
    global matrix
    for c in range(4):
        col = [
                matrix[0][c],
                matrix[1][c],
                matrix[2][c],
                matrix[3][c]
        ]
        col = mmult(col)
        matrix[0][c] = col[0]
        matrix[1][c] = col[1]
        matrix[2][c] = col[2]
        matrix[3][c] = col[3]
    dumpMatrix("mix columns")

#Matrix multiplication done in GF(2^8)
def mmult(matb):
    c = [
            None,
            None,
            None,
            None
        ]
    c[0] = gf2mult(2, matb[0]) ^ gf2mult(3, matb[1]) ^ matb[2] ^ matb[3]

    c[1] = matb[0] ^ gf2mult(2, matb[1]) ^ gf2mult(3, matb[2]) ^ matb[3]

    c[2] = matb[0] ^ matb[1] ^ gf2mult(2, matb[2]) ^ gf2mult(3, matb[3])

    c[3] = gf2mult(3, matb[0]) ^ matb[1] ^ matb[2] ^ gf2mult(2, matb[3])
```

```
    return c

#GF(2^8) multiplication using AES irreducible polynomial
def gf2mult(x, y):
    ret = 0
    for i in range(8):
        if (y & 1) != 0:
            ret = ret ^ x
        b = (x & 0x80)
        x = (x << 1) & 0xFF
        if b:
            x = x ^ 0x1B
        y = (y >> 1) & 0xFF
    return ret

    #Reconstruct the 128-bit cipher text from the matrix
    cipher = 0
    for c in range(4):
        for r in range(4):
            cipher = cipher << 8
            b = matrix[r][c]
            cipher |= b
```

Lab Cat A9

(a) An army general about 2000 years ago sent by messenger a note to the emperor telling how many troops he had. But the number was encrypted in the following way: dividing 602 by 3 gives a remainder of 2 dividing 602 by 5 gives a remainder of 2 dividing 602 by 7 gives a remainder of 0 dividing 602 by 11 gives a remainder of 8 What is the message? Write a Java code to show the encryption method as well.

```
public class TroopEncryption {
    public static void main(String[] args) {
        int troops = 602;
        int remainder3 = troops % 3;
        int remainder5 = troops % 5;
        int remainder7 = troops % 7;
        int remainder11 = troops % 11;

        System.out.println("Troops: " + troops);
        System.out.println("Remainder when dividing by 3: " + remainder3);
        System.out.println("Remainder when dividing by 5: " + remainder5);
        System.out.println("Remainder when dividing by 7: " + remainder7);
        System.out.println("Remainder when dividing by 11: " + remainder11);
    }
}
```

(b) Encrypt the word “cryptology” with the Hill Cipher using the Key 6 7 3 11. Write the java code as well

```
import java.util.Arrays;

public class HillCipher {
    private static final int[][] KEY = {{6, 7}, {3, 11}};
    private static final int MOD = 26;

    public static void main(String[] args) {
        String plaintext = "cryptology";
        String encryptedText = encrypt(plaintext);
        System.out.println("Encrypted text: " + encryptedText);
    }

    private static String encrypt(String plaintext) {
        StringBuilder result = new StringBuilder();

        // Remove spaces and convert to uppercase
        plaintext = plaintext.replaceAll("\\s", "").toUpperCase();

        // Pad the plaintext if its length is not even
        if (plaintext.length() % 2 != 0) {
            plaintext += "X";
        }

        // Iterate over the plaintext in pairs
        for (int i = 0; i < plaintext.length(); i += 2) {
```

```
            // Convert pair of letters to numbers
            int[] pair = {
                    plaintext.charAt(i) - 'A',
                    plaintext.charAt(i + 1) - 'A'
            };

            // Perform matrix multiplication
            int[] resultPair = {
                    (KEY[0][0] * pair[0] + KEY[0][1] * pair[1]) % MOD,
                    (KEY[1][0] * pair[0] + KEY[1][1] * pair[1]) % MOD
            };

            // Convert back to letters
            result.append((char) (resultPair[0] + 'A'));
            result.append((char) (resultPair[1] + 'A'));
        }

        return result.toString();
    }
}
```

General Hill Cipher Code:

```
// Jav code to implement Hill Cipher
class hill {

    // Following function generates the
    // key matrix for the key string
    static void getKeyMatrix(String key, int keyMatrix[][]) {
        int k = 0;
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                keyMatrix[i][j] = (key.charAt(k)) % 65;
                k++;
            }
        }
    }

    // Following function encrypts the message
    static void encrypt(int cipherMatrix[][],
            int keyMatrix[][],
            int messageVector[][]) {
        int x, i, j;
        for (i = 0; i < 3; i++) {
            for (j = 0; j < 1; j++) {
                cipherMatrix[i][j] = 0;

                for (x = 0; x < 3; x++) {
                    cipherMatrix[i][j] += keyMatrix[i][x] * messageVector[x][j];
```

```java
            }

            cipherMatrix[i][j] = cipherMatrix[i][j] % 26;
        }
    }
}

// Function to implement Hill Cipher
static void HillCipher(String message, String key) {
    // Get key matrix from the key string
    int[][] keyMatrix = new int[3][3];
    getKeyMatrix(key, keyMatrix);

    int[][] messageVector = new int[3][1];

    // Generate vector for the message
    for (int i = 0; i < 3; i++)
        messageVector[i][0] = (message.charAt(i)) % 65;

    int[][] cipherMatrix = new int[3][1];

    // Following function generates
    // the encrypted vector
    encrypt(cipherMatrix, keyMatrix, messageVector);

    String CipherText = "";

    // Generate the encrypted text from
    // the encrypted vector
    for (int i = 0; i < 3; i++)
        CipherText += (char) (cipherMatrix[i][0] + 65);

    // Finally print the ciphertext
    System.out.println("Ciphertext:" + CipherText);
}

// Driver code
public static void main(String[] args) {
    // Get the message to be encrypted
    String message = "CRYPTOGRAPHy";

    // Get the key
    String key = "GYBNQKURP";

    HillCipher(message, key);
```

```
    }
}
```

Lab Cat A9

(a) An army general about 2000 years ago sent by messenger a note to the emperor telling how many troops he had. But the number was encrypted in the following way: dividing 602 by 3 gives a remainder of 2 dividing 602 by 5 gives a remainder of 2 dividing 602 by 7 gives a remainder of 0 dividing 602 by 11 gives a remainder of 8 What is the message? Write a Java code to show the encryption method as well.

```
public class TroopEncryption {
    public static void main(String[] args) {
        int troops = 602;
        int remainder3 = troops % 3;
        int remainder5 = troops % 5;
        int remainder7 = troops % 7;
        int remainder11 = troops % 11;

        System.out.println("Troops: " + troops);
        System.out.println("Remainder when dividing by 3: " + remainder3);
        System.out.println("Remainder when dividing by 5: " + remainder5);
        System.out.println("Remainder when dividing by 7: " + remainder7);
        System.out.println("Remainder when dividing by 11: " + remainder11);
    }
}
```

(b) Encrypt the word “cryptology” with the Hill Cipher using the Key 6 7 3 11. Write the java code as well

```
import java.util.Arrays;

public class HillCipher {
    private static final int[][] KEY = {{6, 7}, {3, 11}};
    private static final int MOD = 26;

    public static void main(String[] args) {
        String plaintext = "cryptology";
        String encryptedText = encrypt(plaintext);
        System.out.println("Encrypted text: " + encryptedText);
    }

    private static String encrypt(String plaintext) {
        StringBuilder result = new StringBuilder();

        // Remove spaces and convert to uppercase
        plaintext = plaintext.replaceAll("\\s", "").toUpperCase();
```

```
        // Pad the plaintext if its length is not even
        if (plaintext.length() % 2 != 0) {
            plaintext += "X";
        }

        // Iterate over the plaintext in pairs
        for (int i = 0; i < plaintext.length(); i += 2) {
            // Convert pair of letters to numbers
            int[] pair = {
                    plaintext.charAt(i) - 'A',
                    plaintext.charAt(i + 1) - 'A'
            };

            // Perform matrix multiplication
            int[] resultPair = {
                    (KEY[0][0] * pair[0] + KEY[0][1] * pair[1]) % MOD,
                    (KEY[1][0] * pair[0] + KEY[1][1] * pair[1]) % MOD
            };

            // Convert back to letters
            result.append((char) (resultPair[0] + 'A'));
            result.append((char) (resultPair[1] + 'A'));
        }

        return result.toString();
    }
}
```

General Hill Cipher Code:

```
// Jav code to implement Hill Cipher
class hill {

    // Following function generates the
    // key matrix for the key string
    static void getKeyMatrix(String key, int keyMatrix[][]) {
        int k = 0;
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                keyMatrix[i][j] = (key.charAt(k)) % 65;
                k++;
            }
        }
    }

    // Following function encrypts the message
    static void encrypt(int cipherMatrix[][],
            int keyMatrix[][],
            int messageVector[][]) {
        int x, i, j;
```

```java
        for (i = 0; i < 3; i++) {
            for (j = 0; j < 1; j++) {
                cipherMatrix[i][j] = 0;

                for (x = 0; x < 3; x++) {
                    cipherMatrix[i][j] += keyMatrix[i][x] * messageVector[x][j];
                }

                cipherMatrix[i][j] = cipherMatrix[i][j] % 26;
            }
        }
    }

    // Function to implement Hill Cipher
    static void HillCipher(String message, String key) {
        // Get key matrix from the key string
        int[][] keyMatrix = new int[3][3];
        getKeyMatrix(key, keyMatrix);

        int[][] messageVector = new int[3][1];

        // Generate vector for the message
        for (int i = 0; i < 3; i++)
            messageVector[i][0] = (message.charAt(i)) % 65;

        int[][] cipherMatrix = new int[3][1];

        // Following function generates
        // the encrypted vector
        encrypt(cipherMatrix, keyMatrix, messageVector);

        String CipherText = "";

        // Generate the encrypted text from
        // the encrypted vector
        for (int i = 0; i < 3; i++)
            CipherText += (char) (cipherMatrix[i][0] + 65);

        // Finally print the ciphertext
        System.out.println("Ciphertext:" + CipherText);
    }

    // Driver code
    public static void main(String[] args) {
        // Get the message to be encrypted
```

```
        String message = "CRYPTOGRAPHy";

        // Get the key
        String key = "GYBNQKURP";

        HillCipher(message, key);
    }
}
```

```
from sympy.ntheory.modular import solve_congruence

# Define the remainders and moduli
remainders = [2, 2, 0, 8]
moduli = [3, 5, 7, 11]

# Use Chinese Remainder Theorem to find the solution
solution = solve_congruence(remainders, moduli)

# The solution will be in the form (x, LCM), where x is the smallest solution
x = solution[0]

print("The message is:", x)
```