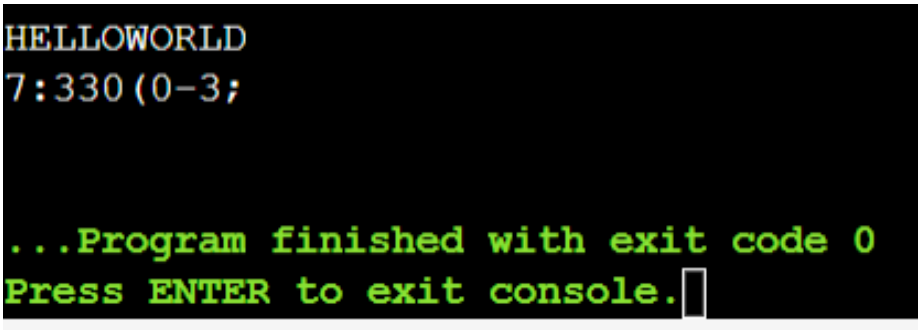
IS-1 AND XOR Encryption

Code

```
#include <iostream.h>
//using namespace std;
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <stdlib.h>
void main()
{
    //clrscr();
    char str[]="HELLOWORLD";
    char str1[11];
    char str2[11];
    int i,len;
    len = strlen(str);

    for(i=0;i<len;i++)
    {
        str1[i]=str[i] & 127;
        cout<<str1[i];
    }
    cout<<"\n";
    for(i=0;i<len;i++)
    {
        str2[i] = str[i] ^ 127;
        cout<<str2[i];
    }
    cout<<"\n";
    getch();
}
```

Output


HELLOWORLD
7:330(0-3;
...Program finished with exit code 0
Press ENTER to exit console.

IS-2 Transposition Technique- Columnar

Code

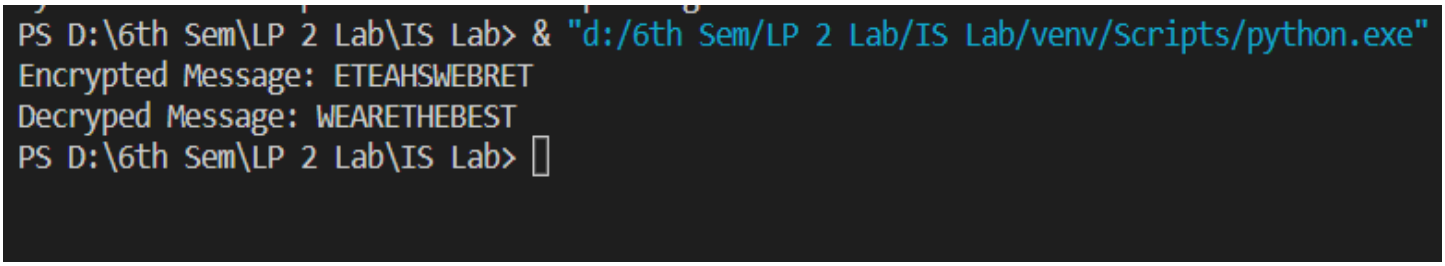
```
import math
key = "HACK"

# Encryption
def encryptMessage(msg):
    cipher = ""
    # track key indices
    k_indx = 0

    msg_len = float(len(msg))
    msg_lst = list(msg)
    key_lst = sorted(list(key))

    # calculate column of the matrix
    col = len(key)

    # calculate maximum row of the matrix
    row = int(math.ceil(msg_len / col))

    # add the padding character '_' in empty
    # the empty cell of the matix
    fill_null = int((row * col) - msg_len)
    msg_lst.extend('_' * fill_null)

    # create Matrix and insert message and
    # padding characters row-wise
    matrix = [msg_lst[i: i + col]
              for i in range(0, len(msg_lst), col)]

    # read matrix column-wise using key
    for _ in range(col):
```

```
        curr_idx = key.index(key_lst[k_indx])
        cipher += ''.join([row[curr_idx]
                           for row in matrix])
        k_indx += 1

    return cipher

# Decryption
def decryptMessage(cipher):
    msg = ""

    # track key indices
    k_indx = 0

    # track msg indices
    msg_indx = 0
    msg_len = float(len(cipher))
    msg_lst = list(cipher)

    # calculate column of the matrix
    col = len(key)

    # calculate maximum row of the matrix
    row = int(math.ceil(msg_len / col))

    # convert key into list and sort
    # alphabetically so we can access
    # each character by its alphabetical position.
    key_lst = sorted(list(key))

    # create an empty matrix to
    # store deciphered message
    dec_cipher = []
```

```
    for _ in range(row):
        dec_cipher += [[None] * col]

    # Arrange the matrix column wise according
    # to permutation order by adding into new matrix
    for _ in range(col):
        curr_idx = key.index(key_lst[k_indx])

        for j in range(row):
            dec_cipher[j][curr_idx] = msg_lst[msg_indx]
            msg_indx += 1
        k_indx += 1

    # convert decrypted msg matrix into a string
    try:
        msg = ''.join(sum(dec_cipher, []))
    except TypeError:
        raise TypeError("This program cannot",
                        "handle repeating words.")

    null_count = msg.count('_')

    if null_count > 0:
        return msg[: -null_count]

    return msg

# Driver Code
msg = "WEARETHEBEST"

cipher = encryptMessage(msg)
print("Encrypted Message: {}".
               format(cipher))
```

```
print("Decryped Message: {}".
        format(decryptMessage(cipher)))
```

Output



IS-3 DES Algorithm

Code

```
# 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):
            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)
```

Output

```
('Round ', 7, ' ', '10AF9D37', ' ', '308BEE97', ' ', '02765708B5BF')
('Round ', 8, ' ', '308BEE97', ' ', 'A9FC20A3', ' ', '84BB4473DCCC')
('Round ', 9, ' ', 'A9FC20A3', ' ', '2E8F9C65', ' ', '34F822F0C66D')
('Round ', 10, ' ', '2E8F9C65', ' ', 'A15A4B87', ' ', '708AD2DDB3C0')
('Round ', 11, ' ', 'A15A4B87', ' ', '236779C2', ' ', 'C1948E87475E')
('Round ', 12, ' ', '236779C2', ' ', 'B8089591', ' ', '69A629FEC913')
('Round ', 13, ' ', 'B8089591', ' ', '4A1210F6', ' ', 'DA2D032B6EE3')
('Round ', 14, ' ', '4A1210F6', ' ', '5A78E394', ' ', '06EDA4ACF5B5')
('Round ', 15, ' ', '5A78E394', ' ', '18CA18AD', ' ', '4568581ABCCE')
('Round ', 16, ' ', '14A7D678', ' ', '18CA18AD', ' ', '194CD072DE8C')
('Plain Text : ', '123456ABCD132536')
PS D:\6th Sem\LP 2 Lab\IS Lab>
```

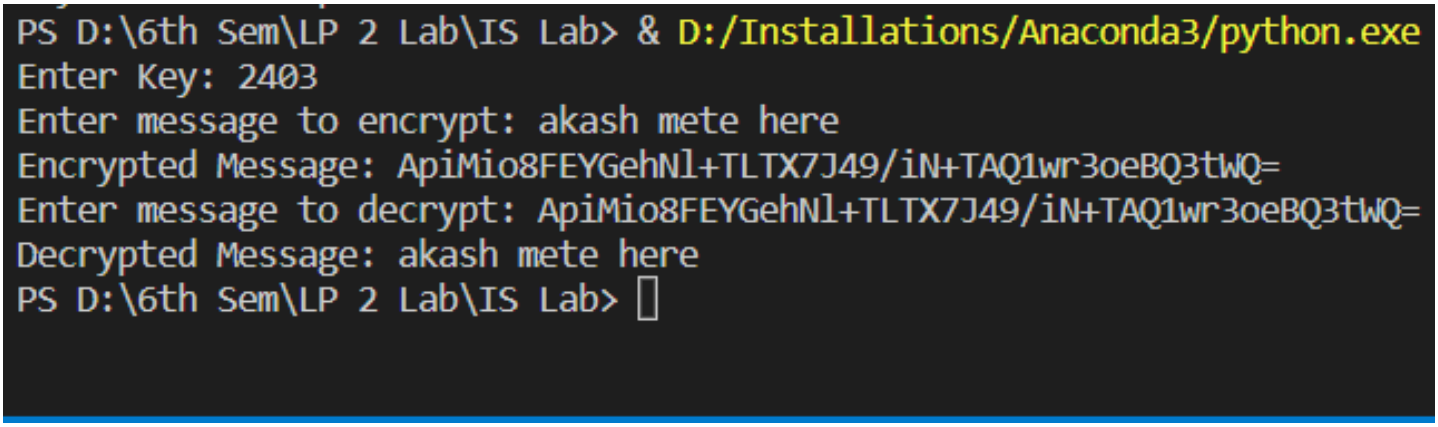
IS- 4 AES Algorithm

Code

```
import hashlib
from base64 import b64decode, b64encode

from Crypto import Random
from Crypto.Cipher import AES

class AESCipher(object):
    def __init__(self, key):
        self.block_size = AES.block_size
        self.key = hashlib.sha256(key.encode()).digest()

    def encrypt(self, plain_text):
        plain_text = self.__pad(plain_text)
        iv = Random.new().read(self.block_size)
        cipher = AES.new(self.key, AES.MODE_CBC, iv)
        encrypted_text = cipher.encrypt(plain_text.encode())
        return b64encode(iv + encrypted_text).decode("utf-8")

    def decrypt(self, encrypted_text):
        encrypted_text = b64decode(encrypted_text)
        iv = encrypted_text[:self.block_size]
        cipher = AES.new(self.key, AES.MODE_CBC, iv)
        plain_text = cipher.decrypt(encrypted_text[self.block_size:]).decode("utf-8")
        return self.__unpad(plain_text)

    def __pad(self, plain_text):
        number_of_bytes_to_pad = self.block_size - len(plain_text) % self.block_size
        ascii_string = chr(number_of_bytes_to_pad)
        padding_str = number_of_bytes_to_pad * ascii_string
        padded_plain_text = plain_text + padding_str
        return padded_plain_text

    @staticmethod
    def __unpad(plain_text):
```

```
        last_character = plain_text[len(plain_text) - 1:]
        return plain_text[:-ord(last_character)]


key = input("Enter Key: ")
aes = AESCipher(key)
message = input("Enter message to encrypt: ")
encryptedMessage = aes.encrypt(message)
print("Encrypted Message:", encryptedMessage)
message = input("Enter message to decrypt: ")
decryptedMessage = aes.decrypt(message)
print("Decrypted Message:", decryptedMessage)
```

Output



IS-5 RSA Algorithm

Code

```
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
import binascii

msg = (input("Enter message to encrypt and decrypt"))
msg = bytes(msg, 'utf-8')

keyPair = RSA.generate(3072)

pubKey = keyPair.publickey()
print(f"Public key:  (n={hex(pubKey.n)}, e={hex(pubKey.e)})")
pubKeyPEM = pubKey.exportKey()
print(pubKeyPEM.decode('ascii'))

print(f"Private key: (n={hex(pubKey.n)}, d={hex(keyPair.d)})")
privKeyPEM = keyPair.exportKey()
print(privKeyPEM.decode('ascii'))

# msg = input()
encryptor = PKCS1_OAEP.new(pubKey)
encrypted = encryptor.encrypt(msg)
print("Encrypted:", binascii.hexlify(encrypted))

decryptor = PKCS1_OAEP.new(keyPair)
decrypted = decryptor.decrypt(encrypted)
print('Decrypted:', decrypted)
```

Output

```
PS D:\6th Sem\LP 2 Lab\IS Lab> & D:/Installations/Anaconda3/python.exe "d:/6th Sem/LP 2 Lab/IS Lab/5. RSA.py"
Enter message to encrypt and decrypt AKASH
Public key:  (n=0x9d7f4dbf56c3265b13d96e71ec55a4870133cf40422b242e8179e85e94f8c82fd16dea0778f2f1ed641f2e30be7e6027936492e19ee2ff35407faea3e44b5623cff9dc0b490f8
b97206d87f0f8f685cf9e551917f5c6dc3470653f24709eee22d8352e2acac0c21afd102fa9b8cf74cf11f7093c3f761dffaae7a916518be16d1869b22d5df4af06b6be8ae24982f794612d6db72ae6
80c819222de3dc995d387ec9c98c935e437589d0d38dec097d833cc01d22610a4f6710c0f48e16d655db3172e82d54c0a59e9e124f09a9776913a5494f877f9d875bda95d38bdd957f9512956315bc8
899ab48c130fd2ca93b6258854455ffa8ad748acafe70beb88f73861f2cbcf191472cc6e0fa9f4e91385eccc12c003f122a8c55776415333586db10fb847d5d94f354b1d87f583bdc3590d946b8d5ec
5ec38ddf1095e6e7198cdee451bfaac46759dad29e09baa5f5477db7c7f6ec76dd32216d0e1b709859116712cf99918d58a15a0837ef726cf598b02cda00bfd42b3ddc7dbc629321fd06e7, e=0x100
01)
-----BEGIN PUBLIC KEY-----
MIIBojANBgkqhkiG9w0BAQEFAAOCAY8AMIIBigKCAYEAnX9Nv1bDJlsT2W5x7FWk
hwEzz0BCKyQugXnoXpT4yC/RbeoHePLx7WQfLjC+fmAnk2SS4Z7i/zVAf66j5EtW
I8/53AtJD4uXIG2H8Pj2hc+eVRkX9cbcNHBlPyRwnu4i2DUuKsrAwhr9EC+puM90
zxH3CTw/dh3/quepFlGL4W0YabItXfSvBra+iuJJgveUYS1ttyrmgMgZIi3j3Jld
OH7JyYyTXkN1idDTjewJfYM8wB0iYQpPZxDA9I4W1lXbMXLoLVTApZ6eEk8JqXdp
E6VJT4d/nYdb2pXTi92Vf5USlWMVvIiZq0jBMP0sqTtiWIVEVf+orXSKyv5wvriP
c4YfLLzxkUcsxuD6n06ROF7MwSwAPxIqjFV3ZBUzNYbbEPuEfV2U81Sx2H9YO9w1
kNlGuNXsXsON3xCV5ucZjN7kUb+qxGdZ2tKeCbql9Ud9t8f27HbdMiFtDhtwmFkR
ZxLPmZGNWKFaCDfvcmz1mLAs2gC/1Cs93H28YpMh/QbnAgMBAAE=
-----END PUBLIC KEY-----
Private key: (n=0x9d7f4dbf56c3265b13d96e71ec55a4870133cf40422b242e8179e85e94f8c82fd16dea0778f2f1ed641f2e30be7e6027936492e19ee2ff35407faea3e44b5623cff9dc0b490f8
b97206d87f0f8f685cf9e551917f5c6dc3470653f24709eee22d8352e2acac0c21afd102fa9b8cf74cf11f7093c3f761dffaae7a916518be16d1869b22d5df4af06b6be8ae24982f794612d6db72ae6
80c819222de3dc995d387ec9c98c935e437589d0d38dec097d833cc01d22610a4f6710c0f48e16d655db3172e82d54c0a59e9e124f09a9776913a5494f877f9d875bda95d38bdd957f9512956315bc8
899ab48c130fd2ca93b6258854455ffa8ad748acafe70beb88f73861f2cbcf191472cc6e0fa9f4e91385eccc12c003f122a8c55776415333586db10fb847d5d94f354b1d87f583bdc3590d946b8d5ec
5ec38ddf1095e6e7198cdee451bfaac46759dad29e09baa5f5477db7c7f6ec76dd32216d0e1b709859116712cf99918d58a15a0837ef726cf598b02cda00bfd42b3ddc7dbc629321fd06e7, d=0x45c
bfa750eaaba4f478b26ce76e97759db170e44506e9500c52174f0a17d651e07e6797de94465218aaae4c9f649c42c15be4ef78cf5d51ad0e424cd7de94319e8dbb6800bd74749e8b591b6a16682e381
2bfe4c677fc2ebbc59aa3368738e5e43c9c1957638196d6373ee637d42f6e5a3750293401e84e9de5ffec7a4c8e454782ab32b36c5ae613fd04e753d15bcc9758d6fdf6a342fa33dbc2f71193ca01ad
ed9672814d103fd4d5f8db989fea4b069d64bb306254f0e025738a9d5861412b6e642121f579f072f9a8dd2bae649efed106938fa1b639c719f5c70a913f5bab6945bab8ea841ee3cff96c28d8f69da
d1ec127ce803df4f211fca7ba53dd02e8e9e67861c7cc375e287b6d8089644f279261eda39f7ab807e575fac8bb8b377bb04f0b3ed30ec29cbce7ac1fe7e3b45b229e9dbe7ea07981891d537ca7b80c
a2096c95ef20fc025ab7f022f413d9e1fd6c831a7b4e4256b58f4b0394e801f4bcff02fc448df237ab56df43cdaf7ce33cbbef7f6f9a291be572168cc06f32475)
-----BEGIN RSA PRIVATE KEY-----
CiZF8a4XkxBezlcfTq328WEZyEeLDD7g+3LqKj3eWCM8bZcyl6U2Cdz5WB94qouO
i3IhUwNV6yJzeGm47Ndu07afSM+Vz9uILQKBwQDURGgMhi8R6mKoxPBfMJcJa5dC
Pqg5TXl46Obcgqvuzf2k90jrXA4Oq1QziKpGcwn90IpZJvM2K1z1P4qu1W1mUlu
JnKnQujUZhPqRoq6Z4ypPzZNan8yztG6nwvnHDM2DQY0aJ32EWmiqXF7RLQgrYX9
/rNUv9cAv44NfZ2BzjDmGj4h9A2fQtlHPaj2ZNbDq2iK+x5Yl2tcXjBQ1YtOU8PS
HQFkWe+TKRfjve0c6VWqCgt247SrdegPoS6dw+MCgcEAoGFiXHDYyFied66EBVNl
DQgjX42CpDW8P2RcCG1gyUDBVe5vOGFh26vJtXhny1ZgT/graY+2bqao0uGhxkou
wUmeDmDrXMaeWYp6Xglqpx7arNQmOH+HQTyYH53q6JOjR7pCtLCZFqS8VyHbwFOD
JnvEltnbDymV4EnXo2YGityjte8USGond8Pgh/2SSOmIrmk8qAC98HwhwyQaMM/Z
A4O4YYSh1pzrX1MgOD2GdePKR8OmSKBaXvWOZ6/wYp2OlxpoqRwJyZ94PRAEZfsf
9cNT/vQpQxXpHRCbAbzQVvIWLfLek+lMzvfDyJDc6f/itkU3AoHAS3R49GXKdWbj
D71vzcs30zIKc2XwC5tVOSb8sBXutr3mYH2gty4rVVXSHTMA3tS+UvOaBt6mFIn4
sf3L+5abjwtaP0b0BjEp9czzrurvidAoHYsAwvZ2vd2C9rP7jqg4qB76AxEDgMqZ
TDcwmryaFXobp1WqX4qXtLK/Im8AUV4hhHB4vloTwrUkRYsQShBpJLF7Sc115Kgn
-----END RSA PRIVATE KEY-----
Encrypted: b'6c4d79dcaa8ab41d5c4e3c3ff3c324431b6a43ef1629d2dd0eb020e7feb21f80bf9b2b39f72a87084c5dbce4a7f92f9d37a0bda7c1e160bc85d08899332917d73c737d65f5822bf670
83e533e26982ba803624bcef2c1ea648ad8cc49d153830e90dfb29969d66c07232a25f653c119055b73052ce9354fae0f0a42345bb9e088add14ed840403e36cec767d3a4d16801a4c2905af0f7b482
fc62a343e69cd4b3556e1842411fc5b9911ab47fb9dd1d8e6726b8184e16dd537641789a74599820c17c7fa910a70bd928d49d39159171a1dd868e9dfa5aaeff0be0148cddaf3c845b6194290885247
2eda8d8a51718a2d7de5bce9ee9092edcaaf15b149eb22dae75832b2c6226e87ca6c921883e940e6f81fa96fc2949f07c9fe00a9cc1045d05df3322bceab5b22feb40e3e0794db5c6f623f3044ea799
77c373806d5b8001d2d7de9986b14a06df7da32b068f0f8229a342344f7b40eeb656a3c1c8ed5b7da01c3cf56c7552032867ef8dbca09a3b16616eab324516fe84b24c8f8b7958392'
Decrypted: b' AKASH'
PS D:\6th Sem\LP 2 Lab\IS Lab>
```