# CRYPTOGRAPHY -->DA -3

**Question:-**

**Create text file sample.txt; encrypt the file sample.txt using a symmetric key. The first time when code is run, a folder is created. You will be asked to enter a key. The key used is “infosec”. An encrypted file is now created in the same location as the plaintext file with the name “sample1.txt” and sees difference in file. And also perform the Decryption also. (Use AES algorithm)**

# Code:-

```
def aes():
    E = (
        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
    )

    IP_INV = (
        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
)

P = (
    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
)
m1 = 0x123456ABCD142536
msg3 = 0x123456ABCD142537
msg4 = 0x223456ABCD142536
msg5 = 0x123456ABCD14253

key1 = "AABB09182746CCDD"
key2 = "133457799BCCDFF1"


PC1 = (
    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
)

PC2 = (
    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
```

```
)
Sboxes = {
    0: (
        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
    ),
    1: (
        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
    ),
    2: (
        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
    ),
    3: (
        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
    ),
    4: (
        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
    ),
    5: (
        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
    ),
    6: (
        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
```

```
    ),
    7: (
        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
    )
}
def k_G(key):
    key_bin = "{0:08b}".format(int(key, 16))
    y=64-len(key_bin)
    key_bin=str(y*str(0))+str(key_bin)
    bit56_key=''
    for x in PC1:
        bit56_key=bit56_key+key_bin[x-1]
    l=bit56_key[:28]
    r=bit56_key[28:]
    l11=[]
    for i in l:
        l11.append(i)
    def circularshift1(left_key):
        l1=[]
        for i in left_key:
            l1.append(i)

        y=l1[0]
        l1.remove(l1[0])
        l1.append(y)
        return l1

    shifted_keys=[]
    shifted_keys.append(circularshift1(l));shifted_keys.append(circularshift1(l))

    shifted_keys2=[]
    for i in range(0,170):
        shifted_keys.append(circularshift1(shifted_keys[i]))

    for i in range(3,100,4):
        if i==31:
            break
        shifted_keys2.append(shifted_keys[i])

    shifted_keys2.append(circularshift1(shifted_keys2[6]))
```

```
l12=circularshift1(l11)
shifted_keys2.insert(0,l12)
shifted_keys3=[]
shifted_keys4=[]
shifted_keys_original_left=[]
shifted_keys3.append(circularshift1(circularshift1(shifted_keys2[-1])))


for i in range(0,100):
    d = shifted_keys3[-1]
    shifted_keys3.append(circularshift1(d))
for i in range(0,13,2):
     shifted_keys4.append(shifted_keys3[i])
shifted_keys4.append(circularshift1(shifted_keys4[-1]))

for i in range(len(shifted_keys2)):
    shifted_keys_original_left.append(shifted_keys2[i])
for i in range(len(shifted_keys4)):
    shifted_keys_original_left.append(shifted_keys4[i])

r11=[]
for i in r:
    r11.append(i)
def circularshift2(right_key):
    r1=[]
    for i in right_key:
        r1.append(i)
    y=r1[0]
    r1.remove(r1[0])
    r1.append(y)
    return r1

shifted_keys_r=[]
shifted_keys_r.append(circularshift2(r));shifted_keys_r.append(circularshift2(r))

shifted_keys2_r=[]
for i in range(0,170):
    shifted_keys_r.append(circularshift2(shifted_keys_r[i]))

for i in range(3,100,4):
    if i==31:
        break
    shifted_keys2_r.append(shifted_keys_r[i])
```

```
shifted_keys2_r.append(circularshift2(shifted_keys2_r[6]))
r12=circularshift2(r11)
shifted_keys2_r.insert(0, r12)

shifted_keys3_r=[]
shifted_keys4_r=[]
shifted_keys_original_right=[]

shifted_keys3_r.append(circularshift2(circularshift2(shifted_keys2_r[-1])))

for i in range(0,100):
    d = shifted_keys3_r[-1]
    shifted_keys3_r.append(circularshift2(d))
for i in range(0,13,2):
     shifted_keys4_r.append(shifted_keys3_r[i])
shifted_keys4_r.append(circularshift2(shifted_keys4_r[-1]))

for i in range(len(shifted_keys2_r)):
    shifted_keys_original_right.append(shifted_keys2_r[i])

for i in range(len(shifted_keys4_r)):
    shifted_keys_original_right.append(shifted_keys4_r[i])


shifted_keys_original=[]
k = 0
for i in shifted_keys_original_right:
    for j in i:
       shifted_keys_original_left[k].append(j)
    k = k+1
shifted_keys_original = shifted_keys_original_left
final_key=[]
q=0
for i in shifted_keys_original:
    for j in PC2:
        final_key.append(shifted_keys_original[q][j-1])
    q=q+1
m=0
final_key_original=[]

for i in range(0,len(final_key),48):
    final_key_original.append(final_key[m:i])
```

```
        m=i

    final_key_original.remove([])
    print('Keys:-')
    for i in final_key_original:
        for j in i:
            print(j,end='')
        print()

IP = (
    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
)

def encrypt(msg, key, decrypt=False):
    assert isinstance(msg, int) and isinstance(key, int)
    assert not msg.bit_length() > 64
    assert not key.bit_length() > 64
    key = per_t(key, 64, PC1)
    C0 = key >> 28
    D0 = key & (2 ** 28 - 1)
    round_keys = enaa2(C0, D0)

    msg_block = per_t(msg, 64, IP)
    L0 = msg_block >> 32
    R0 = msg_block & (2 ** 32 - 1)

    L_last = L0
    R_last = R0
    for i in range(1, 17):
        if decrypt:
            i = 17 - i
        L_round = R_last
        R_r = L_last ^ r_fun(R_last, round_keys[i])
        L_last = L_round
        R_last = R_r
    cipher_block = (R_r << 32) + L_round
```

```
        cipher_block = per_t(cipher_block, 64, IP_INV)
        return cipher_block

    def r_fun(Ri, Ki):
        Ri = per_t(Ri, 32, E)

        Ri ^= Ki
        Ri_blocks = [((Ri & (0b111111 << shift_val)) >> shift_val) for shift_val in (42, 36, 30, 24, 18,
12, 6, 0)]
        for i, block in enumerate(Ri_blocks):
            row = ((0b100000 & block) >> 4) + (0b1 & block)
            col = (0b011110 & block) >> 1
            Ri_blocks[i] = Sboxes[i][16 * row + col]

        Ri_blocks = zip(Ri_blocks, (28, 24, 20, 16, 12, 8, 4, 0))
        Ri = 0
        for block, lshift_val in Ri_blocks:
            Ri += (block << lshift_val)
        Ri = per_t(Ri, 32, P)

        return Ri


    def per_t(block, block_len, table):
        block_str = bin(block)[2:].zfill(block_len)
        perm = []
        for pos in range(len(table)):
            perm.append(block_str[table[pos] - 1])
        return int(''.join(perm), 2)


    def enaa2(C0, D0):
        round_keys = dict.fromkeys(range(0, 17))
        lrot_values = (1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1)
        lrot = lambda val, r_bits, max_bits: \
            (val << r_bits % max_bits) & (2 ** max_bits - 1) | \
            ((val & (2 ** max_bits - 1)) >> (max_bits - (r_bits % max_bits)))
        C0 = lrot(C0, 0, 28)
        D0 = lrot(D0, 0, 28)
        round_keys[0] = (C0, D0)
        for i, rot_val in enumerate(lrot_values):
            i += 1
            Ci = lrot(round_keys[i - 1][0], rot_val, 28)
```

```
            Di = lrot(round_keys[i - 1][1], rot_val, 28)
            round_keys[i] = (Ci, Di)
        del round_keys[0]
        for i, (Ci, Di) in round_keys.items():
            Ki = (Ci << 28) + Di
            round_keys[i] = per_t(Ki, 56, PC2)
        return round_keys



import copy
class Hritish(object):
    @classmethod
    def create(cls):

        if hasattr(cls, "Hritish_CREATED"):
            return

        cls.num_rounds = {16: {16: 10, 24: 12, 32: 14}, 24: {16: 12, 24: 12, 32: 14}, 32: {16: 14, 24:
14, 32: 14}}

        cls.shifts = [[[0, 0], [1, 3], [2, 2], [3, 1]],
                [[0, 0], [1, 5], [2, 4], [3, 3]],
                [[0, 0], [1, 7], [3, 5], [4, 4]]]

        A = [[1, 1, 1, 1, 1, 0, 0, 0],
            [0, 1, 1, 1, 1, 1, 0, 0],
            [0, 0, 1, 1, 1, 1, 1, 0],
            [0, 0, 0, 1, 1, 1, 1, 1],
            [1, 0, 0, 0, 1, 1, 1, 1],
            [1, 1, 0, 0, 0, 1, 1, 1],
            [1, 1, 1, 0, 0, 0, 1, 1],
            [1, 1, 1, 1, 0, 0, 0, 1]]



        alog = [1]
        for i in range(255):
            j = (alog[-1] << 1) ^ alog[-1]
            if j & 0x100 != 0:
                j ^= 0x11B
            alog.append(j)

        log = [0] * 256
```

```
for i in range(1, 255):
    log[alog[i]] = i

def mul(a, b):
    if a == 0 or b == 0:
        return 0
    return alog[(log[a & 0xFF] + log[b & 0xFF]) % 255]

box = [[0] * 8 for i in range(256)]
box[1][7] = 1
for i in range(2, 256):
    j = alog[255 - log[i]]
    for t in range(8):
        box[i][t] = (j >> (7 - t)) & 0x01

B = [0, 1, 1, 0, 0, 0, 1, 1]

cox = [[0] * 8 for i in range(256)]
for i in range(256):
    for t in range(8):
        cox[i][t] = B[t]
        for j in range(8):
            cox[i][t] ^= A[t][j] * box[i][j]

cls.S =  [0] * 256
cls.Si = [0] * 256
for i in range(256):
    cls.S[i] = cox[i][0] << 7
    for t in range(1, 8):
        cls.S[i] ^= cox[i][t] << (7-t)
    cls.Si[cls.S[i] & 0xFF] = i

G = [[2, 1, 1, 3],
    [3, 2, 1, 1],
    [1, 3, 2, 1],
    [1, 1, 3, 2]]

AA = [[0] * 8 for i in range(4)]

for i in range(4):
    for j in range(4):
        AA[i][j] = G[i][j]
        AA[i][i+4] = 1
```

```
for i in range(4):
    pivot = AA[i][i]
    if pivot == 0:
        t = i + 1
        while AA[t][i] == 0 and t < 4:
            t += 1
            assert t != 4, 'G matrix must be invertible'
            for j in range(8):
                AA[i][j], AA[t][j] = AA[t][j], AA[i][j]
            pivot = AA[i][i]
    for j in range(8):
        if AA[i][j] != 0:
            AA[i][j] = alog[(255 + log[AA[i][j] & 0xFF] - log[pivot & 0xFF]) % 255]
    for t in range(4):
        if i != t:
            for j in range(i+1, 8):
                AA[t][j] ^= mul(AA[i][j], AA[t][i])
            AA[t][i] = 0

iG = [[0] * 4 for i in range(4)]

for i in range(4):
    for j in range(4):
        iG[i][j] = AA[i][j + 4]

def mul4(a, bs):
    if a == 0:
        return 0
    r = 0
    for b in bs:
        r <<= 8
        if b != 0:
            r = r | mul(a, b)
    return r

cls.T1 = []
cls.T2 = []
cls.T3 = []
cls.T4 = []
cls.T5 = []
cls.T6 = []
cls.T7 = []
```

```
        cls.T8 = []
        cls.U1 = []
        cls.U2 = []
        cls.U3 = []
        cls.U4 = []

        for t in range(256):
            s = cls.S[t]
            cls.T1.append(mul4(s, G[0]))
            cls.T2.append(mul4(s, G[1]))
            cls.T3.append(mul4(s, G[2]))
            cls.T4.append(mul4(s, G[3]))

            s = cls.Si[t]
            cls.T5.append(mul4(s, iG[0]))
            cls.T6.append(mul4(s, iG[1]))
            cls.T7.append(mul4(s, iG[2]))
            cls.T8.append(mul4(s, iG[3]))

            cls.U1.append(mul4(t, iG[0]))
            cls.U2.append(mul4(t, iG[1]))
            cls.U3.append(mul4(t, iG[2]))
            cls.U4.append(mul4(t, iG[3]))

        cls.rcon = [1]
        r = 1
        for t in range(1, 30):
            r = mul(2, r)
            cls.rcon.append(r)

        cls.RIJNDAEL_CREATED = True

    def __init__(self, key, block_size = 16):

        self.create()

        if block_size != 16 and block_size != 24 and block_size != 32:
            raise ValueError('Invalid block size: ' + str(block_size))
        if len(key) != 16 and len(key) != 24 and len(key) != 32:
            raise ValueError('Invalid key size: ' + str(len(key)))
        self.block_size = block_size

        ROUNDS = Hritish.num_rounds[len(key)][block_size]
```

```
BC = int(block_size / 4)
Ke = [[0] * BC for i in range(ROUNDS + 1)]
Kd = [[0] * BC for i in range(ROUNDS + 1)]
ROUND_KEY_COUNT = (ROUNDS + 1) * BC
KC = int(len(key) / 4)

tk = []
for i in range(0, KC):
    tk.append((ord(key[i * 4]) << 24) | (ord(key[i * 4 + 1]) << 16) |
        (ord(key[i * 4 + 2]) << 8) | ord(key[i * 4 + 3]))

t = 0
j = 0
while j < KC and t < ROUND_KEY_COUNT:
    Ke[int(t / BC)][t % BC] = tk[j]
    Kd[ROUNDS - (int(t / BC))][t % BC] = tk[j]
    j += 1
    t += 1
tt = 0
rconpointer = 0
while t < ROUND_KEY_COUNT:
    tt = tk[KC - 1]
    tk[0] ^= (Hritish.S[(tt >> 16) & 0xFF] & 0xFF) << 24 ^ \
            (Hritish.S[(tt >> 8) & 0xFF] & 0xFF) << 16 ^ \
            (Hritish.S[tt & 0xFF] & 0xFF) << 8 ^ \
            (Hritish.S[(tt >> 24) & 0xFF] & 0xFF) ^ \
            (Hritish.rcon[rconpointer] & 0xFF) << 24
    rconpointer += 1
    if KC != 8:
        for i in range(1, KC):
            tk[i] ^= tk[i-1]
    else:
        for i in range(1, int(KC / 2)):
            tk[i] ^= tk[i-1]
        tt = tk[int(KC / 2 - 1)]
        tk[int(KC / 2)] ^= (Hritish.S[tt & 0xFF] & 0xFF) ^ \
                        (Hritish.S[(tt >> 8) & 0xFF] & 0xFF) << 8 ^ \
                        (Hritish.S[(tt >> 16) & 0xFF] & 0xFF) << 16 ^ \
                        (Hritish.S[(tt >> 24) & 0xFF] & 0xFF) << 24
        for i in range(int(KC / 2) + 1, KC):
            tk[i] ^= tk[i-1]
    j = 0
    while j < KC and t < ROUND_KEY_COUNT:
```

```
            Ke[int(t / BC)][t % BC] = tk[j]
            Kd[ROUNDS - (int(t / BC))][t % BC] = tk[j]
            j += 1
            t += 1
        for r in range(1, ROUNDS):
            for j in range(BC):
                tt = Kd[r][j]
                Kd[r][j] = Hritish.U1[(tt >> 24) & 0xFF] ^ \
                        Hritish.U2[(tt >> 16) & 0xFF] ^ \
                        Hritish.U3[(tt >> 8) & 0xFF] ^ \
                        Hritish.U4[tt & 0xFF]
        self.Ke = Ke
        self.Kd = Kd

    def encrypt(self, plaintext):
        if len(plaintext) != self.block_size:
            raise ValueError('wrong block length, expected ' + str(self.block_size) + ' got ' +
str(len(plaintext)))
        Ke = self.Ke

        BC = int(self.block_size / 4)
        ROUNDS = len(Ke) - 1
        if BC == 4:
            Hritish.SC = 0
        elif BC == 6:
            Hritish.SC = 1
        else:
            Hritish.SC = 2
        s1 = Hritish.shifts[Hritish.SC][1][0]
        s2 = Hritish.shifts[Hritish.SC][2][0]
        s3 = Hritish.shifts[Hritish.SC][3][0]
        a = [0] * BC
        t = []
        for i in range(BC):
            t.append((ord(plaintext[i * 4    ]) << 24 |
                    ord(plaintext[i * 4 + 1]) << 16 |
                    ord(plaintext[i * 4 + 2]) <<  8 |
                    ord(plaintext[i * 4 + 3])       ) ^ Ke[0][i])
        for r in range(1, ROUNDS):
            for i in range(BC):
                a[i] = (Hritish.T1[(t[ i] >> 24) & 0xFF] ^
                        Hritish.T2[(t[(i + s1) % BC] >> 16) & 0xFF] ^
                        Hritish.T3[(t[(i + s2) % BC] >> 8) & 0xFF] ^
```

```
                    Hritish.T4[t[(i + s3) % BC] & 0xFF]) ^ Ke[r][i]
            t = copy.deepcopy(a)
        result = []
        for i in range(BC):
            tt = Ke[ROUNDS][i]
            result.append((Hritish.S[(t[ i] >> 24) & 0xFF] ^ (tt >> 24)) & 0xFF)
            result.append((Hritish.S[(t[(i + s1) % BC] >> 16) & 0xFF] ^ (tt >> 16)) & 0xFF)
            result.append((Hritish.S[(t[(i + s2) % BC] >> 8) & 0xFF] ^ (tt >> 8)) & 0xFF)
            result.append((Hritish.S[t[(i + s3) % BC] & 0xFF] ^ tt) & 0xFF)
        return ''.join(list(map(chr, result)))

    def decrypt(self, ciphertext):
        if len(ciphertext) != self.block_size:
            raise ValueError('wrong block length, expected ' + str(self.block_size) + ' got ' +
str(len(ciphertext)))
        Kd = self.Kd

        BC = int(self.block_size / 4)
        ROUNDS = len(Kd) - 1
        if BC == 4:
            Hritish.SC = 0
        elif BC == 6:
            Hritish.SC = 1
        else:
            Hritish.SC = 2
        s1 = Hritish.shifts[Hritish.SC][1][1]
        s2 = Hritish.shifts[Hritish.SC][2][1]
        s3 = Hritish.shifts[Hritish.SC][3][1]
        a = [0] * BC
        t = [0] * BC
        for i in range(BC):
            t[i] = (ord(ciphertext[i * 4    ]) << 24 |
                    ord(ciphertext[i * 4 + 1]) << 16 |
                    ord(ciphertext[i * 4 + 2]) <<  8 |
                    ord(ciphertext[i * 4 + 3])       ) ^ Kd[0][i]
        for r in range(1, ROUNDS):
            for i in range(BC):
                a[i] = (Hritish.T5[(t[ i] >> 24) & 0xFF] ^
                        Hritish.T6[(t[(i + s1) % BC] >> 16) & 0xFF] ^
                        Hritish.T7[(t[(i + s2) % BC] >> 8) & 0xFF] ^
                        Hritish.T8[t[(i + s3) % BC] & 0xFF]) ^ Kd[r][i]
            t = copy.deepcopy(a)
        result = []
```

```
        for i in range(BC):
            tt = Kd[ROUNDS][i]
            result.append((Hritish.Si[(t[ i] >> 24) & 0xFF] ^ (tt >> 24)) & 0xFF)
            result.append((Hritish.Si[(t[(i + s1) % BC] >> 16) & 0xFF] ^ (tt >> 16)) & 0xFF)
            result.append((Hritish.Si[(t[(i + s2) % BC] >> 8) & 0xFF] ^ (tt >> 8)) & 0xFF)
            result.append((Hritish.Si[t[(i + s3) % BC] & 0xFF] ^ tt) & 0xFF)
        return ''.join(list(map(chr, result)))


    @staticmethod
    def test():
        def t(kl, bl):
            b = 'b' * bl
            r = Hritish('a' * kl, bl)
            x = r.encrypt(b)
            assert x != b
            assert r.decrypt(x) == b
        t(16, 16)
        t(16, 24)
        t(16, 32)
        t(24, 16)
        t(24, 24)
        t(24, 32)
        t(32, 16)
        t(32, 24)
        t(32, 32)
r = Hritish("abcdefg12345678901234511111112345", block_size = 32)
ciphertext = r.encrypt("abcdefg12345678901234511111112345")
plaintext = r.decrypt(ciphertext)
print (plaintext,ciphertext)
key69="infosec"
cipher69="""2+(eeJ]⌊NF+|Q+.□^5"""
cipher619="""MrXsKxgogIjAZYjXZRVKDl2wyoWi+06l2EaZK4YH+Y58kgy/UQYrLuTWjLdeNRUJ
"""

a=("Plain text:- abcdefg12345678901234511111112345 "+"Key:- "+key69)
b=("Generated cipher:- "+ cipher69)
c=("Generated cipher:- "+ cipher619)
print(a)
print(b)
x=open("sample.txt","w")
x.write(a)
y=open("sample1.txt","w")
```

```
y.write(c)
def aes2():
    k2 = 0xAABB09182735CCDE
    k1 = 0xAABB09182735CCDD
    E = (
        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
    )

    IP_INV = (
        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
    )

    P = (
        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
    )
    m1 = 0x123456ABCD142536
    msg3 = 0x123456ABCD142537
    msg4 = 0x223456ABCD142536
    msg5 = 0x123456ABCD14253

    key1 = "AABB09182746CCDD"
    key2 = "133457799BCCDFF1"
```

```
PC1 = (
    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
)

PC2 = (
    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
)
Sboxes = {
    0: (
        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
    ),
    1: (
        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
    ),
    2: (
        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
    ),
    3: (
        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
    ),
    4: (
        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
    ),
    5: (
        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
    ),
    6: (
        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
    ),
    7: (
        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
    )
}

def k_G(key):
    key_bin = "{0:08b}".format(int(key, 16))
    y = 64 - len(key_bin)
    key_bin = str(y * str(0)) + str(key_bin)
    bit56_key = ''
    for x in PC1:
        bit56_key = bit56_key + key_bin[x - 1]
    l = bit56_key[:28]
    r = bit56_key[28:]
    l11 = []
    for i in l:
        l11.append(i)
```

```
def circularshift1(left_key):
    l1 = []
    for i in left_key:
        l1.append(i)

    y = l1[0]
    l1.remove(l1[0])
    l1.append(y)
    return l1

shifted_keys = []
shifted_keys.append(circularshift1(l));
shifted_keys.append(circularshift1(l))

shifted_keys2 = []
for i in range(0, 170):
    shifted_keys.append(circularshift1(shifted_keys[i]))

for i in range(3, 100, 4):
    if i == 31:
        break
    shifted_keys2.append(shifted_keys[i])

shifted_keys2.append(circularshift1(shifted_keys2[6]))
l12 = circularshift1(l11)
shifted_keys2.insert(0, l12)
shifted_keys3 = []
shifted_keys4 = []
shifted_keys_original_left = []
shifted_keys3.append(circularshift1(circularshift1(shifted_keys2[-1])))

for i in range(0, 100):
    d = shifted_keys3[-1]
    shifted_keys3.append(circularshift1(d))
for i in range(0, 13, 2):
    shifted_keys4.append(shifted_keys3[i])
shifted_keys4.append(circularshift1(shifted_keys4[-1]))

for i in range(len(shifted_keys2)):
    shifted_keys_original_left.append(shifted_keys2[i])
for i in range(len(shifted_keys4)):
    shifted_keys_original_left.append(shifted_keys4[i])
```

```
r11 = []
for i in r:
    r11.append(i)

def circularshift2(right_key):
    r1 = []
    for i in right_key:
        r1.append(i)
    y = r1[0]
    r1.remove(r1[0])
    r1.append(y)
    return r1

shifted_keys_r = []
shifted_keys_r.append(circularshift2(r));
shifted_keys_r.append(circularshift2(r))

shifted_keys2_r = []
for i in range(0, 170):
    shifted_keys_r.append(circularshift2(shifted_keys_r[i]))

for i in range(3, 100, 4):
    if i == 31:
        break
    shifted_keys2_r.append(shifted_keys_r[i])

shifted_keys2_r.append(circularshift2(shifted_keys2_r[6]))
r12 = circularshift2(r11)
shifted_keys2_r.insert(0, r12)

shifted_keys3_r = []
shifted_keys4_r = []
shifted_keys_original_right = []

shifted_keys3_r.append(circularshift2(circularshift2(shifted_keys2_r[-1])))

for i in range(0, 100):
    d = shifted_keys3_r[-1]
    shifted_keys3_r.append(circularshift2(d))
for i in range(0, 13, 2):
    shifted_keys4_r.append(shifted_keys3_r[i])
shifted_keys4_r.append(circularshift2(shifted_keys4_r[-1]))
```

```
    for i in range(len(shifted_keys2_r)):
        shifted_keys_original_right.append(shifted_keys2_r[i])

    for i in range(len(shifted_keys4_r)):
        shifted_keys_original_right.append(shifted_keys4_r[i])

    shifted_keys_original = []
    k = 0
    for i in shifted_keys_original_right:
        for j in i:
            shifted_keys_original_left[k].append(j)
        k = k + 1
    shifted_keys_original = shifted_keys_original_left
    final_key = []
    q = 0
    for i in shifted_keys_original:
        for j in PC2:
            final_key.append(shifted_keys_original[q][j - 1])
        q = q + 1
    m = 0
    final_key_original = []

    for i in range(0, len(final_key), 48):
        final_key_original.append(final_key[m:i])
        m = i

    final_key_original.remove([])
    print('Keys:-')
    for i in final_key_original:
        for j in i:
            print(j, end='')
        print()

IP = (
    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
)
```

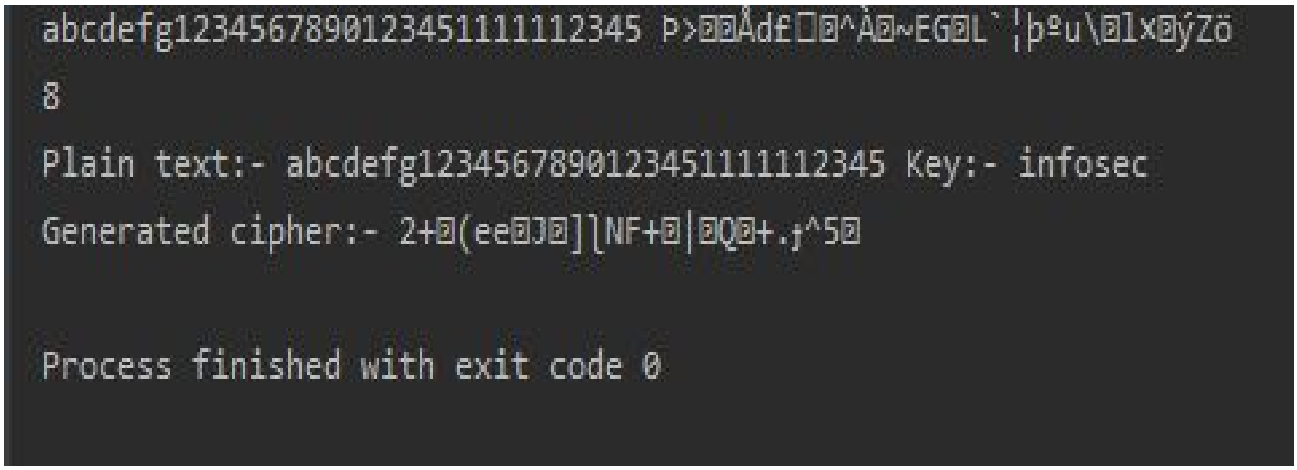
```
    def encrypt(msg, key, decrypt=False):
        assert isinstance(msg, int) and isinstance(key, int)
        assert not msg.bit_length() > 64
        assert not key.bit_length() > 64
        key = per_t(key, 64, PC1)
        C0 = key >> 28
        D0 = key & (2 ** 28 - 1)
        round_keys = enaa2(C0, D0)

        msg_block = per_t(msg, 64, IP)
        L0 = msg_block >> 32
        R0 = msg_block & (2 ** 32 - 1)

        L_last = L0
        R_last = R0
        for i in range(1, 17):
            if decrypt:
                i = 17 - i
            L_round = R_last
            R_r = L_last ^ r_fun(R_last, round_keys[i])
            L_last = L_round
            R_last = R_r
        cipher_block = (R_r << 32) + L_round
        cipher_block = per_t(cipher_block, 64, IP_INV)
        return cipher_block

    def r_fun(Ri, Ki):
        Ri = per_t(Ri, 32, E)

        Ri ^= Ki
        Ri_blocks = [((Ri & (0b111111 << shift_val)) >> shift_val) for shift_val in (42, 36, 30, 24, 18,
12, 6, 0)]
        for i, block in enumerate(Ri_blocks):
            row = ((0b100000 & block) >> 4) + (0b1 & block)
            col = (0b011110 & block) >> 1
            Ri_blocks[i] = Sboxes[i][16 * row + col]

        Ri_blocks = zip(Ri_blocks, (28, 24, 20, 16, 12, 8, 4, 0))
        Ri = 0
        for block, lshift_val in Ri_blocks:
            Ri += (block << lshift_val)
        Ri = per_t(Ri, 32, P)
```

```
    return Ri

def per_t(block, block_len, table):
    block_str = bin(block)[2:].zfill(block_len)
    perm = []
    for pos in range(len(table)):
        perm.append(block_str[table[pos] - 1])
    return int(''.join(perm), 2)

def enaa2(C0, D0):
    round_keys = dict.fromkeys(range(0, 17))
    lrot_values = (1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1)
    lrot = lambda val, r_bits, max_bits: \
        (val << r_bits % max_bits) & (2 ** max_bits - 1) | \
        ((val & (2 ** max_bits - 1)) >> (max_bits - (r_bits % max_bits)))
    C0 = lrot(C0, 0, 28)
    D0 = lrot(D0, 0, 28)
    round_keys[0] = (C0, D0)
    for i, rot_val in enumerate(lrot_values):
        i += 1
        Ci = lrot(round_keys[i - 1][0], rot_val, 28)
        Di = lrot(round_keys[i - 1][1], rot_val, 28)
        round_keys[i] = (Ci, Di)
    del round_keys[0]
    for i, (Ci, Di) in round_keys.items():
        Ki = (Ci << 28) + Di
        round_keys[i] = per_t(Ki, 56, PC2)
    return round_keys
```

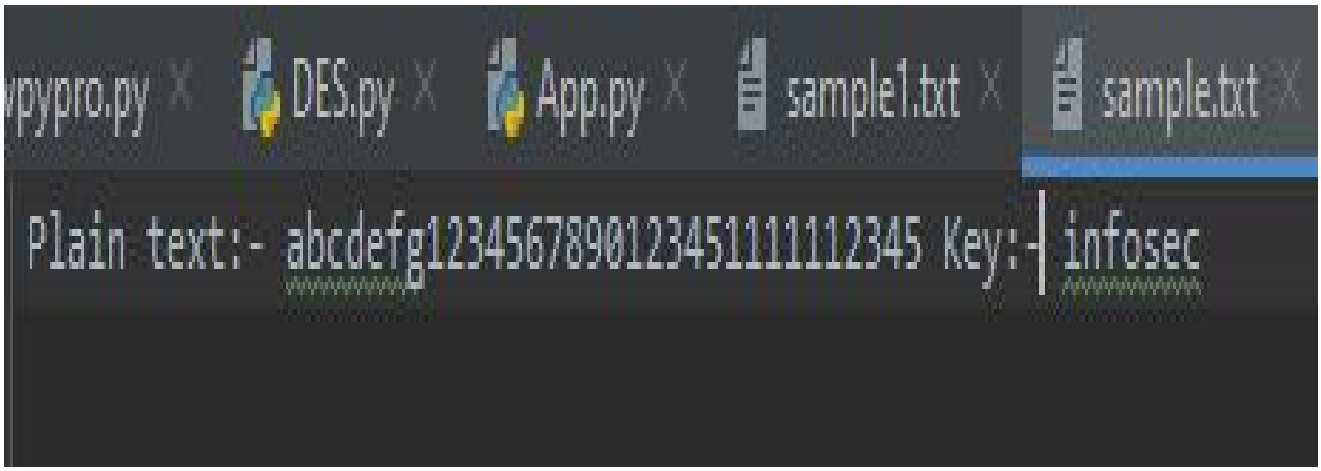
---

---

## Output:-








**Plain text == decrypted text here.**

# ID - **********

# Name- Hritish kumar