Disha Meghnani
Student No.: 2763978

Network Security and Cryptography, Exercises 1 (unassessed)

Decrypt the following ciphertext encrypted with the columnar transposition cipher (you need to find the key by brute-force search):
AVUEVLETSEISBNACBOOLEOBTILBDLCOBOOE [1 point]

Ciphertext: AVUEVLETSEISBNACBOOLEOBTILBDLCOBOOE
Total No. of characters = 35
Factors of 35: 1, 5, 7, 35
Key is one of the factors, but 1 and 35 can not be the key because 1 as key will give 35 rows and 35 will give 1 row in both the case they do not make a sensible sentence

With 5 as a key:
35/5 = 7
That is 7 rows:

A	T	A	O	L
V	S	C	B	C
U	E	B	T	O
E	I	O	I	B
V	S	O	L	O
L	B	L	B	O
E	N	E	D	E

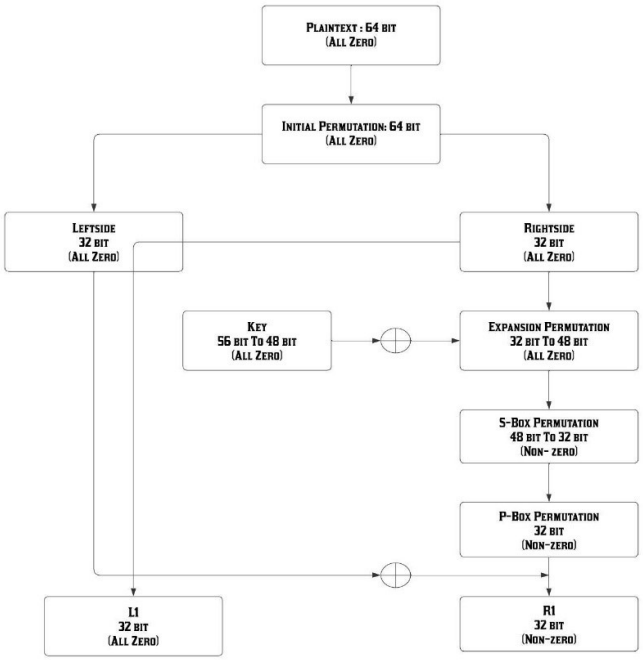
This makes a gibberish sentence, hence cannot be the key.

With 7 as a key:
35/7 = 5
That is 5 rows:

A	L	I	C	E	L	O
V	E	S	B	O	B	B
U	T	B	O	B	D	O
E	S	N	O	T	L	O
V	E	A	L	I	C	E

Sentence: ALICE LOVES BOB BUT BOB DOES NOT LOVE ALICE
Hence the key is 7

What is the output of the first round of the DES algorithm when the plaintext and the key are both all zeros? [1 point]



S-box Permutation:

	S1	S2	S3	S4	S5	S6	S7	S8
Input (6 bit)	000000	000000	000000	000000	000000	000000	000000	000000
Output (Decimal)	14	15	10	7	2	12	4	13
Output (Binary) (4 bit)	1110	1111	1010	0111	0010	1100	0100	1101

P-Box Permutation

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

11011000110110001101101110111100 XOR 00000000000000000000000000000000:

11011000110110001101101110111100

Output of Round 1:
00000000000000000000000000000000110110001101100011011011101111100

Remember that it is desirable for good block ciphers that a change in one input bit affects many output bits, a property that is called *diffusion* or *avalanche effect*. We will try to get a feeling for the avalanche property of DES. Let *x* be all zeros (0x0000000000000000) and *y* be all zeros except 1 in the 13th bit (0x000*8*000000000000). Let the key be all zeros. After just one round, how many bits in the block are different when *x* is the input, compared to when *y* is the input? What about after two rounds? Three? Four? (For this exercise, you might like to search for an implementation of DES on the web, and download it and modify it to output the answers.) [2 points]

Code :

```
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 = "0000000000000000"
key = "0000000000000000"
```

```
# 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 1:

Encryption
After initial permutation 0000000000000000
Round 1 00000000 D8D8DBBC 000000000000
Round 2 D8D8DBBC E73AED4F 000000000000
Round 3 E73AED4F 5BFA6BA6 000000000000
Round 4 5BFA6BA6 5E45E257 000000000000 …

Output 2:

Encryption
After initial permutation 0000000000000200
Round 1 00000200 D8F8DBB4 000000000000
Round 2 D8F8DBB4 E528E26B 000000000000
Round 3 E528E26B 87D70236 000000000000
Round 4 87D70236 3D14F53C 000000000000 …

Hence,
After 1 round 8 bits are different
After 2 round 16 bits are different
After 3 round 48 bits are different
After 4 round 56 bits are different

Consider AES with 128-bit keys. Assume that the principal key k is all-zeros. Then the initial round key (k_0) is also all-zeros. What is the first round subkey (k_1) and the second round subkey (k_2)? (Again, for this exercise and the following one, you might like to use a computer for help.) [2 points]

Code:

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad

def xor_bytes(a, b):
    return bytes(x ^ y for x, y in zip(a, b))

def rot_word(word):
    return word[1:] + word[:1]

def sub_word(word):
    s_box = [
        # 0    1   2    3    4    5    6    7    8    9    A    B    C    D    E    F
        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,
        0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16
    ]
    return bytes([s_box[b] for b in word])

def key_expansion(key):
    RCON = [
        0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1B, 0x36, 0x6C, 0xD8, 0xAB, 0x4D, 0x9A
    ]

    expanded_keys = [key]

    for i in range(1, 11):
        temp = expanded_keys[-1][-4:]
        temp = sub_word(rot_word(temp))
        temp = xor_bytes(temp, bytes([RCON[i-1], 0, 0, 0]))

        new_key = xor_bytes(expanded_keys[-1][:4], temp)
        for j in range(1, 4):
            new_key += xor_bytes(new_key[-4:], expanded_keys[-1][j * 4:(j + 1) * 4])

        expanded_keys.append(new_key)

    return expanded_keys[1], expanded_keys[2]

# Initial key is all-zeros for 128-bit AES key
initial_key = bytes(16)

k1, k2 = key_expansion(initial_key)
print(f"k1: {k1.hex()}")
print(f"k2: {k2.hex()}")
```

Output:
k1: 62636363626363636263636362636363
k2: 9b9898c9f9fbfbaa9b9898c9f9fbfbaa

Initial key is 128 bit which is all zeros.
Initial sub key (k0) is 128 bit which is all zeros.

k0 = [w0, w1, w2, w3] = [0x00000000, 0x00000000, 0x00000000, 0x00000000]

k1 = [w4, w5, w6, w7]

w' for k1:
w3 = 0x00000000
Rotword(w3) = 0x00000000
Subword(w3) = 0x63636363
XOR with Rcon[1]
Rcon[1] = 01000000
0x63636363 XOR 0x01000000
w' = 0x62636363

w4 = w' XOR w3 = 0x62636363
w5 = w4 XOR w1 = 0x62636363
w6 = w5 XOR w2 = 0x62636363
w7 = w6 XOR w3 = 0x62636363

k1 = [w4, w5, w6, w7] = [0x62636363, 0x62636363, 0x62636363, 0x62636363]

k2 = [w8, w9, w10, w11]

w' for k2:
w7 = 0x62636363
Rotword(w3) = 0x63636362
Subword(w3) = 0xfbfbfbaa
XOR with Rcon[2]
Rcon[1] = 02000000
0xfbfbfbaa XOR 0x02000000
w' = 0xf9fbfbaa

w8 = w' XOR w7 = 0x9b9898c9
w9 = w4 XOR w5 = 0xf9fbfbaa
w10 = w5 XOR w6 = 0x9b9898c9
w11 = w6 XOR w7 = 0xf9fbfbaa

k1 = [w8, w9, w10, w11] = [0x9b9898c9, 0xf9fbfbaa, 0x9b9898c9, 0xf9fbfbaa]

Again, using AES-128, assume that the principal key is all-zeros, and that the plaintext is also all-zeros. What is the output of the first round, and what is the output of the second round? [2 points]

Code:

```
import numpy as np

# AES S-box used for SubWord operation in key expansion
sbox = np.array([
    [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],
    [0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16]
])

# Rcon used in key expansion
rcon = [0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1B, 0x36]

# Helper functions for key expansion
def sub_word(word):
    # Applies S-box to a word (4 bytes)
    return [sbox[byte >> 4][byte & 0x0F] for byte in word]

def rot_word(word):
    # Rotates a word (4 bytes)
    return word[1:] + word[:1]

def xor_words(word1, word2):
    return [b1 ^ b2 for b1, b2 in zip(word1, word2)]

# Key expansion function to generate round keys
def key_expansion(key):
    key_schedule = [list(key[i:i + 4]) for i in range(0, len(key), 4)]

    for i in range(4, 44):
        temp = key_schedule[i - 1]
        if i % 4 == 0:
            temp = xor_words(sub_word(rot_word(temp)), [rcon[(i // 4) - 1], 0x00, 0x00, 0x00])
        key_schedule.append(xor_words(key_schedule[i - 4], temp))

    return key_schedule

# Now, the full AES encryption process including rounds

# Helper functions for AES encryption steps

# ShiftRows step for AES (modifies state matrix row-wise)
def shift_rows(state):
```

```
    state = np.reshape(state, (4, 4))  # 4x4 matrix
    state = state.tolist()  # Convert to list for manipulation
    # Perform the row shifts
    state[1] = state[1][1:] + state[1][:1]  # 1-byte shift to the left
    state[2] = state[2][2:] + state[2][:2]  # 2-byte shift to the left
    state[3] = state[3][3:] + state[3][:3]  # 3-byte shift to the left
    return np.reshape(state, (16,))  # Flatten the state back into a single array

# AddRoundKey step for AES (XOR with round key)
def add_round_key(state, round_key):
    return [s ^ rk for s, rk in zip(state, round_key)]

# SubBytes step for AES (substitute bytes using the S-box)
def sub_bytes(state):
    return [sbox[byte >> 4][byte & 0x0F] for byte in state]

# MixColumns step for AES (Galois field multiplication)
def mix_columns(state):
    state = np.reshape(state, (4, 4)).T  # Transpose to make it column-major
    for i in range(4):
        a = state[i, :]
        # Galois field multiplication and column mixing
        state[i, :] = [
            gf_mult(a[0], 2) ^ gf_mult(a[1], 3) ^ a[2] ^ a[3],
            a[0] ^ gf_mult(a[1], 2) ^ gf_mult(a[2], 3) ^ a[3],
            a[0] ^ a[1] ^ gf_mult(a[2], 2) ^ gf_mult(a[3], 3),
            gf_mult(a[0], 3) ^ a[1] ^ a[2] ^ gf_mult(a[3], 2)
        ]
    return np.reshape(state.T, (16,))

# Helper function for Galois field multiplication (used in MixColumns)
def gf_mult(x, y):
    result = 0
    for i in range(8):
        if y & 1:
            result ^= x
        carry = x & 0x80
        x <<= 1
        if carry:
            x ^= 0x1b
        y >>= 1
    return result & 0xFF

# AES encryption round function
def aes_round(state, round_key, mix_columns_step=True):
    state = sub_bytes(state)        # SubBytes
    state = shift_rows(state)       # ShiftRows
    if mix_columns_step:
        state = mix_columns(state)   # MixColumns (not in the last round)
    state = add_round_key(state, round_key)  # AddRoundKey
    return state

# Initial all-zero plaintext and key
plaintext = [0x00] * 16
initial_key = [0x00] * 16

# Generate the key schedule (44 words, or 11 round keys)
key_schedule = key_expansion(initial_key)

# Initial AddRoundKey (round 0)
state = add_round_key(plaintext, initial_key)
```

```
# First round
round_1_key = [byte for word in key_schedule[4:8] for byte in word]
state = aes_round(state, round_1_key)

# Store the output after the first round
first_round_output = state

# Second round
round_2_key = [byte for word in key_schedule[8:12] for byte in word]
state = aes_round(state, round_2_key, mix_columns_step=True)

# Store the output after the second round
second_round_output = state

# Formatting the output as hexadecimal strings
first_round_output_hex = ''.join([f'{byte:02x}' for byte in first_round_output])
second_round_output_hex = ''.join([f'{byte:02x}' for byte in second_round_output])

# Output the results
print("First round output:", first_round_output_hex)
print("Second round output:", second_round_output_hex)
```

Output:

First round output: 01000000010000000100000001000000
Second round output: c6e4e48b8587b9f7e7dac5b5bba687d6

Programming exercise.
Let MY24SHA a hash function which outputs the first 24 bits (6 nibbles) of SHA-1. For example, SHA-1 of "mark" is
f1b5a91d4d6ad523f2610114591c007e75d15084
so, the MY24SHA of "mark" is f1b5a9.
Find any collision for MY24SHA. Note: you should find two strings such that the unix command
echo -n *str* | sha1sum - | cut -c1-6
produces the same answer when *str* is replaced by each string. To enable me to verify your answer, please make sure the two strings are typable on a regular keyboard!
Hint: **You should not write the code for SHA-1; you should use an existing library. [2 points]**

Code:

```
import hashlib

def my24sha(input_string):
    sha1_hash = hashlib.sha1(input_string.encode()).hexdigest()
    return sha1_hash[:6]

collision_found = False
attempts = 0

while not collision_found:
    attempts += 1
    str1 = 'str' + str(attempts)
    str2 = 'str' + str(attempts + 1)
    if my24sha(str1) == my24sha(str2):
        collision_found = True

print("Collision found!")
print("String 1: ", str1)
print("String 2: ", str2)
print("MY24SHA of both strings: ", my24sha(str1))
```

Output:

Collision found!
String 1: str6067189
String 2: str6067190
MY24SHA of both strings: 096b70