```python
#!/usr/bin/env/python3
# Homework Number:  hw04
# Name: Shu Hwai Teoh
# ECN Login: teoh0
# Due Date: Tuesday 2/18/2020 at 4:29PM
import sys
from BitVector import *

AES_modulus = BitVector(bitstring='100011011')
subBytesTable = []        # for encryption
invSubBytesTable = []     # for decryption

def genTables():
    c = BitVector(bitstring='01100011')
    d = BitVector(bitstring='00000101')
    for i in range(0, 256):
        # For the encryption SBox, find the multiplicative inverse x′= x_in^(-1) in
        GF(2^8)
        a = BitVector(intVal = i, size=8).gf_MI(AES_modulus, 8) if i != 0 else
        BitVector(intVal=0)
        # For bit scrambling for the encryption SBox entries:
        # scramble the bits of x′ by XORing x′ with
        # four different circularly rotated versions of itself
        # and with a special constant byte c = 0x63.
        # The four circular rotations are through 4, 5, 6, and 7 bit positions to
        the right.
        a1,a2,a3,a4 = [a.deep_copy() for x in range(4)]
        a ^= (a1 >> 4) ^ (a2 >> 5) ^ (a3 >> 6) ^ (a4 >> 7) ^ c
        subBytesTable.append(int(a))
        # For the decryption Sbox:
        b = BitVector(intVal = i, size=8)
        # For bit scrambling for the decryption SBox entries:
        b1,b2,b3 = [b.deep_copy() for x in range(3)]
        b = (b1 >> 2) ^ (b2 >> 5) ^ (b3 >> 7) ^ d
        check = b.gf_MI(AES_modulus, 8)
        b = check if isinstance(check, BitVector) else 0
        invSubBytesTable.append(int(b))

def gen_key_schedule_256(key_bv):
    # byte_sub_table = gen_subbytes_table()
    #  We need 60 keywords (each keyword consists of 32 bits) in the key schedule for
    #  256 bit AES. The 256-bit AES uses the first four keywords to xor the input
    #  block with.  Subsequently, each of the 14 rounds uses 4 keywords from the key
    #  schedule. We will store all 60 keywords in the following list:
    key_words = [None for i in range(60)]
    round_constant = BitVector(intVal = 0x01, size=8)
    for i in range(8):
        key_words[i] = key_bv[i*32 : i*32 + 32]
    for i in range(8,60):
        if i%8 == 0:
            kwd, round_constant = gee(key_words[i-1], round_constant, subBytesTable)
            key_words[i] = key_words[i-8] ^ kwd
        elif (i - (i//8)*8) < 4:
            key_words[i] = key_words[i-8] ^ key_words[i-1]
        elif (i - (i//8)*8) == 4:
            key_words[i] = BitVector(size = 0)
            for j in range(4):
                key_words[i] += BitVector(intVal =

                                 subBytesTable[key_words[i-1][8*j:8*j+8].intValue()],
                                 size = 8)
            key_words[i] ^= key_words[i-8]
        elif ((i - (i//8)*8) > 4) and ((i - (i//8)*8) < 8):
            key_words[i] = key_words[i-8] ^ key_words[i-1]
        else:
            sys.exit("error in key scheduling algo for i = %d" % i)
    return key_words

def gee(keyword, round_constant, byte_sub_table):
    '''
    This is the g() function for key expension.
    '''
    rotated_word = keyword.deep_copy()
    rotated_word << 8
    newword = BitVector(size = 0)
    for i in range(4):
        newword += BitVector(intVal = byte_sub_table[rotated_word[8*i:8*i+8].intValue()], size = 8)
    newword[:8] ^= round_constant
    round_constant = round_constant.gf_multiply_modular(BitVector(intVal = 0x02), AES_modulus, 8)
    return newword, round_constant

def keyEncryptExpend():
    # read key string from key.txt and turn it into a bitVector
    with open(sys.argv[3], "r") as f:
        key = f.read().strip()
    key_bv = BitVector(textstring=key)
    key_words = gen_key_schedule_256(key_bv)
    key_schedule = []
    #Each 32-bit word of the key schedule is shown as a sequence of 4 one-byte integers
    for word_index,word in enumerate(key_words):
        keyword_in_ints = []
        for i in range(4):
            keyword_in_ints.append(word[i*8:i*8+8].intValue())
        # if word_index % 4 == 0: print("\n")
        # print("word %d:  %s" % (word_index, str(keyword_in_ints)))
        key_schedule.append(keyword_in_ints)
    num_rounds = 14
    round_keys = [None for i in range(num_rounds+1)]
    # de_round_key = [None for i in range(num_rounds+1)]
    for i in range(num_rounds+1):
        round_keys[i] = (key_words[i*4] + key_words[i*4+1] + key_words[i*4+2] +
                                    key_words[i*4+3])#.get_bitvector_in_hex()
        # de_round_key[num_rounds-i] = key_words[i*4+3] + key_words[i*4+2] + key_words[i*4+1] + key_words[i*4]
    return round_keys #, de_round_key #list of 32-bit bitVector (each round key has 4 words)

def AES_Encrypt(fileName, round_keys):
    FILEIN = open(fileName)
    input_bv = BitVector(textstring=FILEIN.read())
    # create empty bit vector to store output
    output_bv = BitVector(size=0)
    # loop through all the input and extract 64 bit at a time
    for j in range(0, input_bv.length(), 128):
        if input_bv.length() < j+128:
            # padding the last byte with 0s
            bv = input_bv[j:] + BitVector(bitlist=[0] * (j+128-input_bv.length()))
        else:
            bv = input_bv[j:j+128]
        # add round key
        bv = bv ^ round_keys[0]
        if j==0: print(bv.get_hex_string_from_bitvector())
        # 13 round
        for i in range(1,14):
            # substitute bytes
            bv = subBytes(bv)
            if i==1 and j==0: print(bv.get_hex_string_from_bitvector())
            bv = shiftRows(bv)
            if i==1 and j==0: print(bv.get_hex_string_from_bitvector())
            bv = mixColumns(bv)
            if i==1 and j==0: print(bv.get_hex_string_from_bitvector())
            # add round key
            bv = bv ^ round_keys[i]
            if i==1 and j==0: print(bv.get_hex_string_from_bitvector())
        #last round
        bv = subBytes(bv)
        bv = shiftRows(bv)
        bv = bv ^ round_keys[-1]
        output_bv += bv
    return output_bv # return the bit vector of the encrypted text for the whole content
```

```
def AES_Decrypt(fileName, round_keys):
    FILEIN = open(fileName)
    input_bv = BitVector(hexstring=FILEIN.read())
    # create empty bit vector to store output
    output_bv = BitVector(size=0)
    # loop through all the input and extract 64 bit at a time
    for j in range(0, input_bv.length(), 128):
        if input_bv.length() < j+128:
            # padding the last byte with 0s
            bv = input_bv[j:] + BitVector(bitlist=[0] * (j+128-input_bv.length()))
        else:
            bv = input_bv[j:j+128]
        # add round key
        bv = bv ^ round_keys[0]
        # 13 rounds
        for i in range(1,14):
            bv = InvShiftRows(bv)
            bv = InvSubBytes(bv)
            bv = bv ^ round_keys[i]
            bv = InvMixColumns(bv)
        #last round
        bv = InvShiftRows(bv)
        bv = InvSubBytes(bv)
        bv = bv ^ round_keys[-1]
        output_bv += bv
    return output_bv # return the bit vector of the encrypted text for the whole
    content

def subBytes(bv):
    c = BitVector(bitstring='01100011')
    bv_out = BitVector(size=0)
    for i in range(0, bv.length(), 8):
        # extract 1 byte at a time,
        # bv_out += subBytesTable[int(bv[i:i+4]) *10 + int(bv[i+4:i+8])]
        a = bv[i:i+8].gf_MI(AES_modulus, 8) if int(bv[i:i+8]) != 0 else
        BitVector(intVal=0)
        # For bit scrambling for the encryption SBox entries:
        # scramble the bits of x' by XORing x' with
        # four different circularly rotated versions of itself
        # and with a special constant byte c = 0x63.
        # The four circular rotations are through 4, 5, 6, and 7 bit positions to
        the right.
        a1,a2,a3,a4 = [a.deep_copy() for x in range(4)]
        a ^= (a1 >> 4) ^ (a2 >> 5) ^ (a3 >> 6) ^ (a4 >> 7) ^ c
        bv_out += a
    return bv_out
def shiftRows(bv):
    #(i) not shifting the first row of the state array;
    #(ii) circularly shifting the second row by one byte to the left;
    #(iii) circularly shifting the third row by two bytes to the left;
    #(iv) circularly shifting the last row by three bytes to the left.
    #[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
    # -> [0,5,10,15,4,9,14,3,8,13,2,7,12,1,6,11]
    bv_out = BitVector(size=0)
    for i in range(4):
        a = 4*i
        for j in range(4):
            b = a + 5*j
            if b <= 15:
                bv_out += bv[b*8:b*8+8]
            else:
                bv_out += bv[(b-15-1)*8:(b-15-1)*8+8]
    return bv_out
def mixColumns(bv):
    #  Each byte in a column is replaced by two times that byte,
    # plus three times the next byte, plus the byte that comes next,
    # plus the byte that follows.
    bv_out = BitVector(size=0)
    one = BitVector(intVal = 1, size = 8)
    two = BitVector(intVal = 2, size = 8)
    three = BitVector(intVal = 3, size = 8)
```

```python
    m = [[two,three,one,one],[one,two, three, one],[one, one, two, three], [three,
    one, one, two]]
    for i in range(4):
        for j in range(4):
            a = m[j][0].gf_multiply_modular(bv[8*i*4:8*i*4+8], AES_modulus, 8)
            b = m[j][1].gf_multiply_modular(bv[8*(i*4+1):8*(i*4+1)+8], AES_modulus, 8)
            c = m[j][2].gf_multiply_modular(bv[8*(i*4+2):8*(i*4+2)+8], AES_modulus, 8)
            d = m[j][3].gf_multiply_modular(bv[8*(i*4+3):8*(i*4+3)+8], AES_modulus, 8)
            bv_out += (a^b^c^d)
    return bv_out
def InvShiftRows(bv):
    # The first row is left unchanged,
    # the second row is shifted to the right by one byte,
    # the third row to the right by two bytes,
    # and the last row to the right by three bytes
    #[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
    # -> [0,13,10,7, 4,1,14,11, 8,5,2,15, 12,9,6,3]
    bv_out = BitVector(size=0)
    for i in range(4):
        a = 4*i
        for j in range(4):
            b = a - 3*j
            if b >= 0:
                bv_out += bv[b*8:b*8+8]
            else:
                bv_out += bv[(b+15+1)*8:(b+15+1)*8+8]
    return bv_out
def InvSubBytes(bv):
    d = BitVector(bitstring='00000101')
    bv_out = BitVector(size=0)
    for i in range(0, bv.length(), 8):
        # bv_out += invSubBytesTable[int(bv[i:i+4])][int(bv[i+4:i+8])]
                # For the decryption Sbox:
        b = bv[i:i+8]
        # For bit scrambling for the decryption SBox entries:
        b1,b2,b3 = [b.deep_copy() for x in range(3)]
        b = (b1 >> 2) ^ (b2 >> 5) ^ (b3 >> 7) ^ d
        check = b.gf_MI(AES_modulus, 8)
        b = check if isinstance(check, BitVector) else BitVector(intVal=0, size=8)
        bv_out += b
    return bv_out
def InvMixColumns(bv):
    #  Each byte in a column is replaced by two times that byte,
    # plus three times the next byte, plus the byte that comes next,
    # plus the byte that follows.
    bv_out = BitVector(size=0)
    oe = BitVector(hexstring = "0E")
    ob = BitVector(hexstring = "0B")
    od = BitVector(hexstring = "0D")
    o9 = BitVector(hexstring = "09")
    m = [[oe,ob,od,o9],[o9,oe,ob,od],[od,o9,oe,ob],[ob,od,o9,oe]]
    for i in range(4):
        for j in range(4):
            a = m[j][0].gf_multiply_modular(bv[8*i*4:8*i*4+8], AES_modulus, 8)
            b = m[j][1].gf_multiply_modular(bv[8*(i*4+1):8*(i*4+1)+8], AES_modulus, 8)
            c = m[j][2].gf_multiply_modular(bv[8*(i*4+2):8*(i*4+2)+8], AES_modulus, 8)
            d = m[j][3].gf_multiply_modular(bv[8*(i*4+3):8*(i*4+3)+8], AES_modulus, 8)
            bv_out += (a^b^c^d)
    return bv_out

if __name__ == "__main__":
    genTables()
    # read key from file, encrypt and expend is as 60 round keys (each 4 words)
    round_keys = keyEncryptExpend()
    # encrypt the message.txt with AES
    # python AES.py -e message.txt key.txt encrypted.txt
    # python AES.py -d encrypted.txt key.txt decrypted.txt
    if sys.argv[1] == "-e":
        # perform AES encryption on the plain text
        encryptedText = AES_Encrypt(sys.argv[2], round_keys)
        # transform the ciphertext into the hex string and write out to the file
        with open(sys.argv[4], 'w') as f:
            f.write(encryptedText.get_hex_string_from_bitvector())
    # decrypt the message.txt with DES
    elif sys.argv[1] == "-d":
        # perform AES decryption on the encrypted.txt with round keys in the
        inversed order
        decryptedText = AES_Decrypt(sys.argv[2], round_keys[::-1])
        with open(sys.argv[4], "wb") as f:
            decryptedText.write_to_file(f)

```