CODE

*"""  
Homework Number: 4  
Name: Sneha Mahapatra  
ECN Login: mahapat0  
Due Date: 02/18/2020  
"""*# !/usr/bin/env python  
# !/usr/bin/env python -W ignore:tostring:DeprecationWarning  
#python AES.py -e message.txt key.txt encrypted.txt  
#python AES.py -d encrypted.txt key.txt decrypted.txt  
from sys import \*  
from BitVector import \*  
AES\_modulus = BitVector(bitstring='100011011')  
  
SUBBYTESTABLE = [] # for encryption  
INVSUBBYTESTABLE = [] # for decryption  
KEY\_SCHEDULE = []  
SIZE = 256  
SIZEPLAIN = 128  
  
  
def encrypt():  
 #Read Key  
 READKEY = open(sys.argv[3], 'r')  
 key = READKEY.read()  
 READKEY.close()  
  
 # Create Key Schedule  
 #print("encrypt")  
 KEY\_SCHEDULE = keyInit(key)  
 first\_words = KEY\_SCHEDULE[0]  
  
 #Turn File into BitVector  
 bv = BitVector(filename=sys.argv[2])  
  
 #Open File for Encrypted Text  
 encryptedText = open(sys.argv[4], "w")  
  
 while(bv.more\_to\_read):  
 #Get 128 Bit for Plain Text  
 bitvec = bv.read\_bits\_from\_file(SIZEPLAIN)  
  
 #If block is < 128, pad zeroes  
 if (len(str(bitvec)) % SIZEPLAIN != 0):  
 x = bitvec.length() % SIZE  
 bitvec.pad\_from\_right(SIZEPLAIN-x)  
 # First Add Round Key  
 firstwordsHex = BitVector(hexstring = first\_words)  
 bitvec = bitvec ^ firstwordsHex  
 numRounds = 1  
 while(numRounds < 14):  
 #Turn Bit vector into State Array  
 statearray = matrixArray(bitvec)  
  
 # SubBytes  
 statearray = substitution(statearray, 'E')  
  
 # ShiftRows  
 statearray = shiftRows(statearray, 'E')  
 statearray = intToBitVector(statearray)  
  
 # Mix Columns --> Not on last round  
 statearray = mixColumns(statearray, 'E')  
 p = bitToHex(statearray)  
  
 # Add Round Keys  
 next\_word = KEY\_SCHEDULE[numRounds]  
 statearray = roundKeys(p, next\_word)  
 #print(statearray.get\_bitvector\_in\_hex())  
 bitvec = statearray#BitVector(hexstring= statearray)  
 if (len(str(bitvec)) % SIZEPLAIN != 0):  
 x = bitvec.length() % SIZE  
 bitvec.pad\_from\_left(SIZEPLAIN - x)  
 numRounds += 1  
 if(numRounds == 14):  
 #Turn Bit vector into State Array  
 statearray = matrixArray(bitvec)  
  
 # SubBytes  
 statearray = substitution(statearray, 'E')  
  
 # ShiftRows  
 statearray = shiftRows(statearray, 'E')  
 statearray = intToBitVector(statearray)  
  
 hexFinal = ""  
 for x in statearray:  
 for y in x:  
 hexFinal += y.get\_bitvector\_in\_hex()  
 #Add RoundKey  
 next\_word = KEY\_SCHEDULE[numRounds]  
 hexFinal = roundKeys(hexFinal, next\_word)  
 #print(hexFinal.get\_bitvector\_in\_hex())  
 encryptedText.write(hexFinal.get\_bitvector\_in\_hex())  
 encryptedText.close()  
 pass  
  
  
def decrypt():  
 #print("decrypt")  
 # Read Key  
 READKEY = open(sys.argv[3], 'r')  
 key = READKEY.read()  
 #printKeySchedule(key)  
 READKEY.close()  
  
 # Create Key Schedule  
 KEY\_SCHEDULE = keyInit(key)  
 KEY\_SCHEDULE = KEY\_SCHEDULE[::-1]  
 #printKeySchedule(KEY\_SCHEDULE)  
 # Turn File into BitVector  
 HEXFILE = open(sys.argv[2], 'r')  
 hexString = HEXFILE.read()  
 bv = BitVector(hexstring = hexString)  
 BITVECTORTEXT = open("bitVector.txt", "wb")  
 bv.write\_to\_file(BITVECTORTEXT)  
 BITVECTORTEXT.close()  
 bv = BitVector(filename="bitVector.txt")  
 # Open File for Encrypted Text  
 decryptedText = open(sys.argv[4], "w")  
  
 while (bv.more\_to\_read):  
 # Get 128 Bit for Plain Text  
 bitvec = bv.read\_bits\_from\_file(SIZEPLAIN)  
 # If block is < 128, pad zeroes  
 if (len(str(bitvec)) % SIZEPLAIN != 0):  
 x = bitvec.length() % SIZEPLAIN  
 bitvec.pad\_from\_right(SIZEPLAIN - x)  
 #print("before round key")  
 #print(bitvec.get\_bitvector\_in\_hex())  
  
 # First Add Round Key  
 first\_words = KEY\_SCHEDULE[0]  
 firstwordsHex = BitVector(hexstring=first\_words)  
 bitvec = bitvec ^ firstwordsHex  
 numRounds = 1  
 while (numRounds < 14):  
 # Turn Bit vector into State Array  
 if(numRounds == 1):  
 statearray = matrixArray(bitvec)  
 #print("stateArray")  
 #print(statearray)  
  
 # ShiftRows - WORKING  
 statearray = InverseshiftRows(statearray)  
 #print("Shift Row")  
 #print(statearray)  
 statearray = hextobit(statearray)  
  
 # SubBytes - WORKING  
 statearray = substitution(statearray, 'D')  
 #print("subbytes")  
 statearray = [list(x) for x in zip(statearray[0], statearray[1], statearray[2], statearray[3])]  
 #print(statearray)  
 statearray = intToHex(statearray)  
 #print(statearray)  
  
 # Add Round Keys - WORKING  
 NW = KEY\_SCHEDULE[numRounds]  
 #print("round key")  
 #print(NW)  
 hexFinalL = ""  
 if(numRounds == 1):  
 for i in range(4):  
 for j in range(4):  
 hexFinalL += statearray[j][i]  
 else:  
 for i in range(4):  
 for j in range(4):  
 if(len(str(statearray[j][i])) == 1):  
 statearray[j][i] = '0' + statearray[j][i]  
 #print(statearray[j][i])  
 hexFinalL += statearray[j][i]  
 #print(hexFinalL)  
 hexFinalL = BitVector(hexstring=hexFinalL)  
 next\_word = BitVector(hexstring=NW)  
 statearray = hexFinalL ^ next\_word  
 #print("inverse round key")  
 #print(statearray.get\_bitvector\_in\_hex())  
  
 # Mix Columns --> Not on last round  
 statearray = matrixArray(statearray)  
 statearray = hextobit(statearray)  
 statearray = mixColumns(statearray, 'D')  
 #print("Mix Columns")  
 #print(statearray[0][0].get\_bitvector\_in\_hex())  
 #print(statearray[3][3].get\_bitvector\_in\_hex())  
 #statearray = [list(x) for x in zip(statearray[0], statearray[1], statearray[2], statearray[3])]  
 #Put into bitvector and start process again  
 statearray = bitToHex(statearray)  
 numRounds += 1  
  
 if (numRounds == 14):  
 #statearray = matrixArray(bitvec)  
 #print("stateArray")  
 #print(statearray)  
  
 # ShiftRows  
 statearray = InverseshiftRows(statearray)  
 #print("Shift Row")  
 #print(statearray)  
 statearray = hextobit(statearray)  
  
 # SubBytes  
 #statearray = BitVector(hexstring= "b7af2f6ffcaf7c7bf2cafefcad63f09c")  
 #statearray = matrixArray(statearray)  
 #statearray = hextobit(statearray)  
 statearray = substitution(statearray, 'D')  
 #print("subbytes")  
 statearray = intToHex(statearray)  
 #print(statearray)  
  
 # Add Round Keys  
 NW = KEY\_SCHEDULE[numRounds]  
 #print("round key")  
 #print(NW)  
 hexFinalL = ""  
 for i in range(4):  
 for j in range(4):  
 if (len(str(statearray[i][j])) == 1):  
 statearray[i][j] = '0' + statearray[i][j]  
 #print(statearray[i][j])  
 hexFinalL += statearray[i][j]  
 #print(hexFinalL)  
 hexFinalL = BitVector(hexstring=hexFinalL)  
 next\_word = BitVector(hexstring=NW)  
 hexFinal = hexFinalL ^ next\_word  
 #Send text to file  
 #print(hexFinal.get\_bitvector\_in\_hex())  
 #print(hexFinal.get\_bitvector\_in\_ascii())  
 hexFinal = hexFinal.get\_text\_from\_bitvector()  
 if(ord(hexFinal[len(hexFinal)-1]) == 0):  
 hexFinal = hexFinal[:-1]  
 decryptedText.write(hexFinal)  
  
 decryptedText.close()  
  
def intToBitVector(matrix):  
 for i in range(4):  
 for j in range(4):  
 matrix[i][j] = BitVector(intVal = matrix[i][j], size = 8)  
 return matrix  
  
def hextobit(matrix):  
 for i in range(4):  
 for j in range(4):  
 matrix[i][j] = BitVector(hexstring = matrix[i][j])  
 return matrix  
  
def bitToInt(matrix):  
 for i in range(4):  
 for j in range(4):  
 matrix[i][j] = int(matrix[i][j])  
 return matrix  
  
def bitToHex(matrix):  
 returnMatrix = [[0 for x in range(4)] for x in range(4)]  
 for i in range(4):  
 for j in range(4):  
 returnMatrix[i][j] = matrix[i][j].get\_bitvector\_in\_hex()  
 return returnMatrix  
  
def intToHex(matrix):  
 for i in range(4):  
 for j in range(4):  
 matrix[i][j] = format(matrix[i][j], 'x')  
 if(len(str(matrix[i][j])) == 1):  
 matrix[i][j] = '0' + matrix[i][j]  
 return matrix  
  
def matrixArray(bitvec):  
 bitvecHex = bitvec.get\_bitvector\_in\_hex()  
 n = 2  
 bitvecHex = [bitvecHex[i:i + n] for i in range(0, len(bitvecHex), n)]  
 statearray = [[0 for x in range(4)] for x in range(4)]  
 index = 0  
 for i in range(4):  
 for j in range(4):  
 statearray[i][j] = bitvecHex[index]  
 index += 1  
 return statearray  
  
def substitution(hexVector, charX):  
 if(charX == 'E'):  
 for i in range(4):  
 for j in range(4):  
 hexVector[i][j] = SUBBYTESTABLE[int(hexVector[i][j], 16)]  
 else:  
 for i in range(4):  
 for j in range(4):  
 hexVector[i][j] = INVSUBBYTESTABLE[hexVector[i][j].int\_val()]  
 return hexVector  
  
def shiftRows(vector, charX):  
 shift = 1  
 vector = [list(x) for x in zip(vector[0], vector[1], vector[2], vector[3])]  
 while(shift < 4):  
 if(charX == 'D'):  
 vector[shift] = rotateElemList(vector[shift], -shift)  
 else:  
 vector[shift] = rotateElemList(vector[shift], shift)  
 shift+=1  
 vector = [list(x) for x in zip(vector[0], vector[1], vector[2], vector[3])]  
 return vector  
  
def InverseshiftRows(vector):  
 shift = 1  
 vector = [list(x) for x in zip(vector[0], vector[1], vector[2], vector[3])]  
 while(shift < 4):  
 vector[shift] = rotateElemList(vector[shift], -shift)  
 shift+=1  
 vector = [list(x) for x in zip(vector[0], vector[1], vector[2], vector[3])]  
 return vector  
  
def rotateElemList(listX, shift):  
 return listX[shift:] + listX[:shift]  
  
def mixColumns(matrix, charX):  
 MIXCOLUMNSE = [[2, 3, 1, 1], [1, 2, 3, 1], [1, 1, 2, 3], [3, 1, 1, 2]]  
 MIXCOLUMNSD = [[0x0E, 0x0B, 0x0D, 0x09],[0x09, 0x0E, 0x0B, 0x0D],[0x0D, 0x09,0x0E ,0x0B],[0x0B, 0x0D, 0x09, 0x0E]]  
 if(charX == 'E'):  
 mixColumns = intToBitVector(MIXCOLUMNSE)  
 else:  
 mixColumns = intToBitVector(MIXCOLUMNSD)  
 matrix = [list(x) for x in zip(matrix[0], matrix[1], matrix[2], matrix[3])]  
 endMatrix = [[0 for x in range(4)] for x in range(4)]  
 for i in range(4):  
 for j in range(4):  
 int0 = mixColumns[i][0]  
 int1 = mixColumns[i][1]  
 int2 = mixColumns[i][2]  
 int3 = mixColumns[i][3]  
 bitvec0 = int0.gf\_multiply\_modular(matrix[0][j], AES\_modulus, 8)  
 bitvec1 = int1.gf\_multiply\_modular(matrix[1][j], AES\_modulus, 8)  
 bitvec2 = int2.gf\_multiply\_modular(matrix[2][j], AES\_modulus, 8)  
 bitvec3 = int3.gf\_multiply\_modular(matrix[3][j], AES\_modulus, 8)  
 bitvec0 ^= bitvec1  
 bitvec0 ^= bitvec2  
 bitvec0 ^= bitvec3  
 endMatrix[i][j] = bitvec0  
 endMatrix = [list(x) for x in zip(endMatrix[0], endMatrix[1], endMatrix[2], endMatrix[3])]  
 return endMatrix  
  
def InversemixColumns(matrix):  
 MIXCOLUMNSD = [[0x0E, 0x0B, 0x0D, 0x09],[0x09, 0x0E, 0x0B, 0x0D],[0x0D, 0x09,0x0E ,0x0B],[0x0B, 0x0D, 0x09, 0x0E]]  
 mixColumns = intToBitVector(MIXCOLUMNSD)  
 endMatrix = BitVector(size = 0)  
 for i in range(4):  
 for j in range(4):  
 int0 = mixColumns[i][0]  
 int1 = mixColumns[i][1]  
 int2 = mixColumns[i][2]  
 int3 = mixColumns[i][3]  
 bitvec0 = int0.gf\_multiply\_modular(matrix[0][j], AES\_modulus, 8)  
 bitvec1 = int1.gf\_multiply\_modular(matrix[1][j], AES\_modulus, 8)  
 bitvec2 = int2.gf\_multiply\_modular(matrix[2][j], AES\_modulus, 8)  
 bitvec3 = int3.gf\_multiply\_modular(matrix[3][j], AES\_modulus, 8)  
 endMatrix += (bitvec0 ^ bitvec1 ^ bitvec2 ^ bitvec3)  
 return endMatrix  
  
def roundKeys(p, next\_words):  
 hexFinal = ""  
 for x in p:  
 for y in x:  
 hexFinal += y  
 hexFinal = BitVector(hexstring=hexFinal)  
 if (len(str(hexFinal)) % SIZEPLAIN != 0):  
 x = hexFinal.length() % SIZEPLAIN  
 hexFinal.pad\_from\_left(SIZEPLAIN - x)  
 next\_word = BitVector(hexstring=next\_words)  
 hexFinal = hexFinal ^ next\_word  
 return hexFinal  
  
def InverseroundKeys(p, next\_words):  
 hexFinal = ""  
 for x in p:  
 for y in x:  
 hexFinal += y  
 print(hexFinal)  
 hexFinal = BitVector(hexstring=hexFinal)  
 if (len(str(hexFinal)) % SIZEPLAIN != 0):  
 x = hexFinal.length() % SIZEPLAIN  
 hexFinal.pad\_from\_left(SIZEPLAIN - x)  
 next\_word = BitVector(hexstring=next\_words)  
 hexFinal = hexFinal ^ next\_word  
 return hexFinal  
  
def printKeySchedule(key\_schedule):  
 index = 0  
 for word in key\_schedule:  
 print("word " + str(index) + " " + str(word))  
 if((index+1) % 4 == 0):  
 print("\n")  
 index+=1  
  
def keyInit(key):  
 key\_words = []  
 key = key.strip()  
 key += '0' \* (SIZE // 8 - len(key)) if len(key) < SIZE // 8 else key[:SIZE // 8]  
 key\_bv = BitVector(textstring=key)  
 key\_words = gen\_key\_schedule\_256(key\_bv)  
  
 key\_schedule = []  
  
 enumkey\_word = enumerate(key\_words)  
 for word\_index, word in enumkey\_word:  
 keyword\_in\_ints = []  
 for i in range(4):  
 keyword\_in\_ints.append(word[i \* 8:i \* 8 + 8].intValue())  
 #print("word %d: %s" % (word\_index, str(keyword\_in\_ints)))  
 key\_schedule.append(keyword\_in\_ints)  
 #print(key\_schedule)  
 num\_rounds = 14  
 round\_keys = [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()  
 return round\_keys  
  
def gee(keyword, round\_constant, byte\_sub\_table):  
 *'''  
 This is the g() function you see in Figure 4 of Lecture 8.  
 '''* 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 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, byte\_sub\_table)  
 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 =  
 byte\_sub\_table[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 gen\_subbytes\_table():  
 subBytesTable = []  
 c = BitVector(bitstring='01100011')  
 for i in range(0, 256):  
 a = BitVector(intVal = i, size=8).gf\_MI(AES\_modulus, 8) if i != 0 else BitVector(intVal=0)  
 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))  
 return subBytesTable  
  
def genTables():  
 c = BitVector(bitstring='01100011')  
 d = BitVector(bitstring='00000101')  
 for i in range(0, 256):  
 # For the encryption SBox  
 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:  
 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 main():  
 charInput = sys.argv[1]  
  
 # Generate Table for Encryption and Decryption  
 genTables()  
 #print(SUBBYTESTABLE)  
 if (charInput == '-e'):  
 encrypt()  
 pass  
 elif (charInput == '-d'):  
 decrypt()  
 else:  
 print("Either -e or -d")  
 pass  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

Theory Problems

**1A.** 3x^4 + 5x^2 + 10 + 3x^4 + 5x^2 + 9x + 10   
  
**6x^4 + 10x^2 + 9x + 9**

**1B.**

25x^5 + 10x^4 + 35x^3 + 15x^4 + 6x^3 + 21x^2 + 15x^3 + 6x^2 + 21x + 10x^2 + 4x + 14

**3x^5 + 3x^4 + x^3 + 4x^2 + 3x + 3**