**Math 4175 – CRYPTOGRAPHY**

AES PROGRAMMING CODE IN PYTHON

I have a lot of print statements in between the methods. I have left them as it as instead of removing them as they might give a better insight on how I have planned my program.

“Final Ciphertext is: ” and “Final Plaintext is: ” are the two main strings that give the output. These two will appear at the end of EncryptCiphertext and DecryptCiphertext methods of my program respectively.

Theses two methods are also my encryption and decryption methods.

I would recommend commenting out one function when running the other to see clear results.

Code:

*'''*

*The method addPolynomial is used to perform addition for 8 bit binary lists in Z mod 2.*

*'''*

def **addPolynomial**(arr1, arr2):

store1 = []

store2 = []

*'Storing the string array as a reversed array'*

for i in reversed(range(len(arr1))):

store1.append(arr1[i])

store2.append(arr2[i])

result = []

*'Adding the two Polynomials'*

for i in range(0, len(store1)):

if store1[i] == store2[i]:

result.append(*'0'*)

else:

result.append(*'1'*)

output = []

*'Reversing the computed arrays to their original form'*

for i in reversed(range(len(result))):

output.append(result[i])

return output

*'''*

*The method multiplies the 8 bit binary arrays. This method does not reduce the computed array and therefore is different from xtime.*

*'''*

def **multiplyPoly**(arr1, arr2):

store1 = [0] \* (8)

store2 = [0] \* (8)

for i in range(0, len(arr1)):

store1[i] = arr1[len(arr1) - i - 1]

store2[i] = arr2[len(arr2) - i - 1]

result = [0]\*((len(arr1) + len(arr2)))

*'Multiplying the two arrays'*

for i in range(0, len(store1)):

for j in range(0, len(store2)):

result[i + j] += (store1[i]\*store2[j])

if(result[i + j] == 2):

result[i + j] = 0

output = []

*'Undo the reverse computed array'*

for i in reversed(range(len(result))):

output.append(result[i])

return output

*'''*

*XTIME FUNCTION HERE:*

*This method is the xtime function and reduces the multiplied polynomial in Z mod 100011011, i.e the AES Spec.*

*It then returns the multiplied polynomial.*

*'''*

def **reducedPoly**(arr1, arr2):

store = [0] \* (len(arr2))

for i in range(0, len(arr2)):

store[i] = arr2[len(arr2) - i - 1]

spec = [*'1'*, *'0'*, *'0'*, *'0'*, *'1'*, *'1'*, *'0'*, *'1'*, *'1'*]

result = arr1

output = [*'0'*]\*(8)

if(store[0] == *'1'*):

output = addPolynomial(output, arr1)

*'Multiplying the two polynomials and then reducing it in AES SPEC.'*

for i in range(1, len(store)):

if(store[i] == *'1'*):

if(result[0] == *'1'*):

result.append(*'0'*)

else:

result.pop(0)

result.append(*'0'*)

if(len(result) == 9):

result = addPolynomial(result, spec)

result.pop(0)

output = addPolynomial(output, result)

else:

if(result[0] == *'1'*):

result.append(*'0'*)

else:

result.pop(0)

result.append(*'0'*)

if(len(result) == 9):

result = addPolynomial(result, spec)

result.pop(0)

return output

*'''*

*128 BYTES KEY EXPANSION*

*The method expands the key to 128 bytes which is later used to encrypt and decrypt the plaintext and cipher text.*

*Takes the initial key as a parameter.*

*'''*

def **KeyExpansion128**(InitialKey):

Ekey = InitialKey

addRd = [*'0'*,*'0'*,*'0'*,*'0'*,*'0'*,*'0'*,*'0'*,*'1'*]

spec = [*'1'*, *'0'*, *'0'*, *'0'*, *'1'*, *'1'*, *'0'*, *'1'*, *'1'*]

while(len(Ekey) < 352):

for i in range(0,4):

temp1 = Ekey[len(Ekey) - 8: len(Ekey)]

if(i == 0):

temp1 = KeyExpansionCore(temp1,addRd)

temp2 = Ekey[len(Ekey) - 32: len(Ekey)]

temp2 = temp2[0: 8]

Ekey = Ekey + XORtemps(temp1, temp2)

if(addRd[0] == *'0'*):

addRd.pop(0)

addRd.append(*'0'*)

else:

addRd.append(*'0'*)

addRd = addPolynomial(spec, addRd)

addRd.pop(0)

print(addRd)

print(Ekey)

return Ekey

*'''*

*192 BYTES KEY EXPANSION*

*The method expands the key to 192 bytes which is later used to encrypt and decrypt the plaintext and cipher text.*

*Takes the initial key as a parameter.*

*'''*

def **KeyExpansion192**(InitialKey):

Ekey = InitialKey

addRd = [*'0'*,*'0'*,*'0'*,*'0'*,*'0'*,*'0'*,*'0'*,*'1'*]

spec = [*'1'*, *'0'*, *'0'*, *'0'*, *'1'*, *'1'*, *'0'*, *'1'*, *'1'*]

while(len(Ekey) < 416):

for i in range(0,6):

temp1 = Ekey[len(Ekey) - 8: len(Ekey)]

print(temp1)

if(i == 0):

temp1 = KeyExpansionCore(temp1,addRd)

temp2 = Ekey[len(Ekey) - 48: len(Ekey)]

temp2 = temp2[0: 8]

Ekey = Ekey + XORtemps(temp1, temp2)

if(addRd[0] == *'0'*):

addRd.pop(0)

addRd.append(*'0'*)

else:

addRd.append(*'0'*)

addRd = addPolynomial(spec, addRd)

addRd.pop(0)

print(addRd)

print(Ekey)

return Ekey

*'''*

*256 BYTES KEY EXPANSION*

*The method expands the key to 256 bytes which is later used to encrypt and decrypt the plaintext and cipher text.*

*Takes the initial key as a parameter.*

*'''*

def **KeyExpansion256**(InitialKey):

Ekey = InitialKey

addRd = [*'0'*,*'0'*,*'0'*,*'0'*,*'0'*,*'0'*,*'0'*,*'1'*]

spec = [*'1'*, *'0'*, *'0'*, *'0'*, *'1'*, *'1'*, *'0'*, *'1'*, *'1'*]

while(len(Ekey) < 480):

for i in range(0,8):

temp1 = Ekey[len(Ekey) - 8: len(Ekey)]

print(temp1)

if(i == 0):

temp1 = KeyExpansionCore(temp1,addRd)

if(i == 4):

temp1 = lookUp(temp1)

temp2 = Ekey[len(Ekey) - 64: len(Ekey)]

temp2 = temp2[0: 8]

Ekey = Ekey + XORtemps(temp1, temp2)

if(addRd[0] == *'0'*):

addRd.pop(0)

addRd.append(*'0'*)

else:

addRd.append(*'0'*)

addRd = addPolynomial(spec, addRd)

addRd.pop(0)

print(addRd)

print(Ekey)

return Ekey

*'''*

*Key Expansion Core*

*Performs the steps as outlined in the core method of Key Expansion.*

*'''*

def **KeyExpansionCore**(string, rd):

*'Rotate bytes'*

byte1 = string[0:2]

byte2 = string[2:4]

byte3 = string[4:6]

byte4 = string[6: len(string)]

temp = byte1

byte1 = byte2

byte2 = byte3

byte3 = byte4

byte4 = temp

*'Subbytes'*

b1 = SubBytes(int(byte1[0:1], 16), int(byte1[1:len(byte1)], 16))

b2 = SubBytes(int(byte2[0:1], 16), int(byte2[1:len(byte1)], 16))

b3 = SubBytes(int(byte3[0:1], 16), int(byte3[1:len(byte1)], 16))

b4 = SubBytes(int(byte4[0:1], 16), int(byte4[1:len(byte1)], 16))

first = list(bin(b1)[2:10].zfill(8))

output1 = hex(int(*""*.join(addPolynomial(first, rd)), 2))

output2 = hex(b2)

output3 = hex(b3)

output4 = hex(b4)

concat = str(output1)[len(output1) - 2: len(output1)] + str(output2)[len(output2) - 2: len(output2)] + str(output3)[len(output3) - 2: len(output3)] + str(output4)[len(output4) - 2: len(output4)]

if *'x'* in concat:

concat = concat.replace(*'x'*, *'0'*)

print(concat)

return concat

*'''*

*Method that substitutes a 8 bit string by looking up the values from the aes table.*

*'''*

def **lookUp**(string):

byte1 = string[0:2]

byte2 = string[2:4]

byte3 = string[4:6]

byte4 = string[6: len(string)]

b1 = SubBytes(int(byte1[0:1], 16), int(byte1[1:len(byte1)], 16))

b2 = SubBytes(int(byte2[0:1], 16), int(byte2[1:len(byte1)], 16))

b3 = SubBytes(int(byte3[0:1], 16), int(byte3[1:len(byte1)], 16))

b4 = SubBytes(int(byte4[0:1], 16), int(byte4[1:len(byte1)], 16))

output1 = hex(b1)

output2 = hex(b2)

output3 = hex(b3)

output4 = hex(b4)

concat = str(output1)[len(output1) - 2: len(output1)] + str(output2)[len(output2) - 2: len(output2)] + str(output3)[len(output3) - 2: len(output3)] + str(output4)[len(output4) - 2: len(output4)]

if *'x'* in concat:

concat = concat.replace(*'x'*, *'0'*)

print(concat)

return concat

*'''*

*Method that reverses the action performed by the look up method.*

*Useful in decryption of ciphertext.*

*'''*

def **invLookUp**(string):

byte1 = string[0:2]

byte2 = string[2:4]

byte3 = string[4:6]

byte4 = string[6: len(string)]

b1 = invSubBytes(int(byte1[0:1], 16), int(byte1[1:len(byte1)], 16))

b2 = invSubBytes(int(byte2[0:1], 16), int(byte2[1:len(byte1)], 16))

b3 = invSubBytes(int(byte3[0:1], 16), int(byte3[1:len(byte1)], 16))

b4 = invSubBytes(int(byte4[0:1], 16), int(byte4[1:len(byte1)], 16))

output1 = hex(b1)

output2 = hex(b2)

output3 = hex(b3)

output4 = hex(b4)

concat = str(output1)[len(output1) - 2: len(output1)] + str(output2)[len(output2) - 2: len(output2)] + str(output3)[len(output3) - 2: len(output3)] + str(output4)[len(output4) - 2: len(output4)]

if *'x'* in concat:

concat = concat.replace(*'x'*, *'0'*)

print(concat)

return concat

*'''*

*IMPORTANT METHOD:*

*The method performs XOR operation on 2, 32 bytes string.*

*This method is useful for performing addRound step*

*'''*

def **XORtemps**(temp1, temp2):

output = *""*

*'Performs the XOR operation between 2, 32 byte strings'*

for i in range(0, 4):

b1 = list(bin(int(temp1[2\*i:2\*i + 2], 16))[2:].zfill(8))

b2 = list(bin(int(temp2[2\*i:2\*i + 2], 16))[2:].zfill(8))

byte1 = addPolynomial(b1, b2)

res1 = hex(int(*""*.join(byte1), 2))

res1 = str(res1)[len(res1) - 2:]

if *'x'* in res1:

res1 = res1.replace(*'x'*, *'0'*)

output = output + res1

print(output)

return output

*'''*

*AES LOOKUP TABLE*

*Table to lookup values for subbytes function. Returns the integer value of the looked up byte.*

*'''*

def **SubBytes**(b1, b2):

aes\_sbox = [

[int(*'63'*, 16), int(*'7c'*, 16), int(*'77'*, 16), int(*'7b'*, 16), int(*'f2'*, 16), int(*'6b'*, 16), int(*'6f'*, 16), int(*'c5'*, 16), int(

*'30'*, 16), int(*'01'*, 16), int(*'67'*, 16), int(*'2b'*, 16), int(*'fe'*, 16), int(*'d7'*, 16), int(*'ab'*, 16), int(*'76'*, 16)],

[int(*'ca'*, 16), int(*'82'*, 16), int(*'c9'*, 16), int(*'7d'*, 16), int(*'fa'*, 16), int(*'59'*, 16), int(*'47'*, 16), int(*'f0'*, 16), int(

*'ad'*, 16), int(*'d4'*, 16), int(*'a2'*, 16), int(*'af'*, 16), int(*'9c'*, 16), int(*'a4'*, 16), int(*'72'*, 16), int(*'c0'*, 16)],

[int(*'b7'*, 16), int(*'fd'*, 16), int(*'93'*, 16), int(*'26'*, 16), int(*'36'*, 16), int(*'3f'*, 16), int(*'f7'*, 16), int(*'cc'*, 16), int(

*'34'*, 16), int(*'a5'*, 16), int(*'e5'*, 16), int(*'f1'*, 16), int(*'71'*, 16), int(*'d8'*, 16), int(*'31'*, 16), int(*'15'*, 16)],

[int(*'04'*, 16), int(*'c7'*, 16), int(*'23'*, 16), int(*'c3'*, 16), int(*'18'*, 16), int(*'96'*, 16), int(*'05'*, 16), int(*'9a'*, 16), int(

*'07'*, 16), int(*'12'*, 16), int(*'80'*, 16), int(*'e2'*, 16), int(*'eb'*, 16), int(*'27'*, 16), int(*'b2'*, 16), int(*'75'*, 16)],

[int(*'09'*, 16), int(*'83'*, 16), int(*'2c'*, 16), int(*'1a'*, 16), int(*'1b'*, 16), int(*'6e'*, 16), int(*'5a'*, 16), int(*'a0'*, 16), int(

*'52'*, 16), int(*'3b'*, 16), int(*'d6'*, 16), int(*'b3'*, 16), int(*'29'*, 16), int(*'e3'*, 16), int(*'2f'*, 16), int(*'84'*, 16)],

[int(*'53'*, 16), int(*'d1'*, 16), int(*'00'*, 16), int(*'ed'*, 16), int(*'20'*, 16), int(*'fc'*, 16), int(*'b1'*, 16), int(*'5b'*, 16), int(

*'6a'*, 16), int(*'cb'*, 16), int(*'be'*, 16), int(*'39'*, 16), int(*'4a'*, 16), int(*'4c'*, 16), int(*'58'*, 16), int(*'cf'*, 16)],

[int(*'d0'*, 16), int(*'ef'*, 16), int(*'aa'*, 16), int(*'fb'*, 16), int(*'43'*, 16), int(*'4d'*, 16), int(*'33'*, 16), int(*'85'*, 16), int(

*'45'*, 16), int(*'f9'*, 16), int(*'02'*, 16), int(*'7f'*, 16), int(*'50'*, 16), int(*'3c'*, 16), int(*'9f'*, 16), int(*'a8'*, 16)],

[int(*'51'*, 16), int(*'a3'*, 16), int(*'40'*, 16), int(*'8f'*, 16), int(*'92'*, 16), int(*'9d'*, 16), int(*'38'*, 16), int(*'f5'*, 16), int(

*'bc'*, 16), int(*'b6'*, 16), int(*'da'*, 16), int(*'21'*, 16), int(*'10'*, 16), int(*'ff'*, 16), int(*'f3'*, 16), int(*'d2'*, 16)],

[int(*'cd'*, 16), int(*'0c'*, 16), int(*'13'*, 16), int(*'ec'*, 16), int(*'5f'*, 16), int(*'97'*, 16), int(*'44'*, 16), int(*'17'*, 16), int(

*'c4'*, 16), int(*'a7'*, 16), int(*'7e'*, 16), int(*'3d'*, 16), int(*'64'*, 16), int(*'5d'*, 16), int(*'19'*, 16), int(*'73'*, 16)],

[int(*'60'*, 16), int(*'81'*, 16), int(*'4f'*, 16), int(*'dc'*, 16), int(*'22'*, 16), int(*'2a'*, 16), int(*'90'*, 16), int(*'88'*, 16), int(

*'46'*, 16), int(*'ee'*, 16), int(*'b8'*, 16), int(*'14'*, 16), int(*'de'*, 16), int(*'5e'*, 16), int(*'0b'*, 16), int(*'db'*, 16)],

[int(*'e0'*, 16), int(*'32'*, 16), int(*'3a'*, 16), int(*'0a'*, 16), int(*'49'*, 16), int(*'06'*, 16), int(*'24'*, 16), int(*'5c'*, 16), int(

*'c2'*, 16), int(*'d3'*, 16), int(*'ac'*, 16), int(*'62'*, 16), int(*'91'*, 16), int(*'95'*, 16), int(*'e4'*, 16), int(*'79'*, 16)],

[int(*'e7'*, 16), int(*'c8'*, 16), int(*'37'*, 16), int(*'6d'*, 16), int(*'8d'*, 16), int(*'d5'*, 16), int(*'4e'*, 16), int(*'a9'*, 16), int(

*'6c'*, 16), int(*'56'*, 16), int(*'f4'*, 16), int(*'ea'*, 16), int(*'65'*, 16), int(*'7a'*, 16), int(*'ae'*, 16), int(*'08'*, 16)],

[int(*'ba'*, 16), int(*'78'*, 16), int(*'25'*, 16), int(*'2e'*, 16), int(*'1c'*, 16), int(*'a6'*, 16), int(*'b4'*, 16), int(*'c6'*, 16), int(

*'e8'*, 16), int(*'dd'*, 16), int(*'74'*, 16), int(*'1f'*, 16), int(*'4b'*, 16), int(*'bd'*, 16), int(*'8b'*, 16), int(*'8a'*, 16)],

[int(*'70'*, 16), int(*'3e'*, 16), int(*'b5'*, 16), int(*'66'*, 16), int(*'48'*, 16), int(*'03'*, 16), int(*'f6'*, 16), int(*'0e'*, 16), int(

*'61'*, 16), int(*'35'*, 16), int(*'57'*, 16), int(*'b9'*, 16), int(*'86'*, 16), int(*'c1'*, 16), int(*'1d'*, 16), int(*'9e'*, 16)],

[int(*'e1'*, 16), int(*'f8'*, 16), int(*'98'*, 16), int(*'11'*, 16), int(*'69'*, 16), int(*'d9'*, 16), int(*'8e'*, 16), int(*'94'*, 16), int(

*'9b'*, 16), int(*'1e'*, 16), int(*'87'*, 16), int(*'e9'*, 16), int(*'ce'*, 16), int(*'55'*, 16), int(*'28'*, 16), int(*'df'*, 16)],

[int(*'8c'*, 16), int(*'a1'*, 16), int(*'89'*, 16), int(*'0d'*, 16), int(*'bf'*, 16), int(*'e6'*, 16), int(*'42'*, 16), int(*'68'*, 16), int(

*'41'*, 16), int(*'99'*, 16), int(*'2d'*, 16), int(*'0f'*, 16), int(*'b0'*, 16), int(*'54'*, 16), int(*'bb'*, 16), int(*'16'*, 16)]]

res = aes\_sbox[b1][b2]

return res

*'''*

*REVERSE LOOKUP TABLE FOR DECRYPTION*

*'''*

def **invSubBytes**(b1, b2):

inv\_aes\_sbox = [

[int(*'52'*, 16), int(*'09'*, 16), int(*'6a'*, 16), int(*'d5'*, 16), int(*'30'*, 16), int(*'36'*, 16), int(*'a5'*, 16), int(*'38'*, 16), int(

*'bf'*, 16), int(*'40'*, 16), int(*'a3'*, 16), int(*'9e'*, 16), int(*'81'*, 16), int(*'f3'*, 16), int(*'d7'*, 16), int(*'fb'*, 16)],

[int(*'7c'*, 16), int(*'e3'*, 16), int(*'39'*, 16), int(*'82'*, 16), int(*'9b'*, 16), int(*'2f'*, 16), int(*'ff'*, 16), int(*'87'*, 16), int(

*'34'*, 16), int(*'8e'*, 16), int(*'43'*, 16), int(*'44'*, 16), int(*'c4'*, 16), int(*'de'*, 16), int(*'e9'*, 16), int(*'cb'*, 16)],

[int(*'54'*, 16), int(*'7b'*, 16), int(*'94'*, 16), int(*'32'*, 16), int(*'a6'*, 16), int(*'c2'*, 16), int(*'23'*, 16), int(*'3d'*, 16), int(

*'ee'*, 16), int(*'4c'*, 16), int(*'95'*, 16), int(*'0b'*, 16), int(*'42'*, 16), int(*'fa'*, 16), int(*'c3'*, 16), int(*'4e'*, 16)],

[int(*'08'*, 16), int(*'2e'*, 16), int(*'a1'*, 16), int(*'66'*, 16), int(*'28'*, 16), int(*'d9'*, 16), int(*'24'*, 16), int(*'b2'*, 16), int(

*'76'*, 16), int(*'5b'*, 16), int(*'a2'*, 16), int(*'49'*, 16), int(*'6d'*, 16), int(*'8b'*, 16), int(*'d1'*, 16), int(*'25'*, 16)],

[int(*'72'*, 16), int(*'f8'*, 16), int(*'f6'*, 16), int(*'64'*, 16), int(*'86'*, 16), int(*'68'*, 16), int(*'98'*, 16), int(*'16'*, 16), int(

*'d4'*, 16), int(*'a4'*, 16), int(*'5c'*, 16), int(*'cc'*, 16), int(*'5d'*, 16), int(*'65'*, 16), int(*'b6'*, 16), int(*'92'*, 16)],

[int(*'6c'*, 16), int(*'70'*, 16), int(*'48'*, 16), int(*'50'*, 16), int(*'fd'*, 16), int(*'ed'*, 16), int(*'b9'*, 16), int(*'da'*, 16), int(

*'5e'*, 16), int(*'15'*, 16), int(*'46'*, 16), int(*'57'*, 16), int(*'a7'*, 16), int(*'8d'*, 16), int(*'9d'*, 16), int(*'84'*, 16)],

[int(*'90'*, 16), int(*'d8'*, 16), int(*'ab'*, 16), int(*'00'*, 16), int(*'8c'*, 16), int(*'bc'*, 16), int(*'d3'*, 16), int(*'0a'*, 16), int(

*'f7'*, 16), int(*'e4'*, 16), int(*'58'*, 16), int(*'05'*, 16), int(*'b8'*, 16), int(*'b3'*, 16), int(*'45'*, 16), int(*'06'*, 16)],

[int(*'d0'*, 16), int(*'2c'*, 16), int(*'1e'*, 16), int(*'8f'*, 16), int(*'ca'*, 16), int(*'3f'*, 16), int(*'0f'*, 16), int(*'02'*, 16), int(

*'c1'*, 16), int(*'af'*, 16), int(*'bd'*, 16), int(*'03'*, 16), int(*'01'*, 16), int(*'13'*, 16), int(*'8a'*, 16), int(*'6b'*, 16)],

[int(*'3a'*, 16), int(*'91'*, 16), int(*'11'*, 16), int(*'41'*, 16), int(*'4f'*, 16), int(*'67'*, 16), int(*'dc'*, 16), int(*'ea'*, 16), int(

*'97'*, 16), int(*'f2'*, 16), int(*'cf'*, 16), int(*'ce'*, 16), int(*'f0'*, 16), int(*'b4'*, 16), int(*'e6'*, 16), int(*'73'*, 16)],

[int(*'96'*, 16), int(*'ac'*, 16), int(*'74'*, 16), int(*'22'*, 16), int(*'e7'*, 16), int(*'ad'*, 16), int(*'35'*, 16), int(*'85'*, 16), int(

*'e2'*, 16), int(*'f9'*, 16), int(*'37'*, 16), int(*'e8'*, 16), int(*'1c'*, 16), int(*'75'*, 16), int(*'df'*, 16), int(*'6e'*, 16)],

[int(*'47'*, 16), int(*'f1'*, 16), int(*'1a'*, 16), int(*'71'*, 16), int(*'1d'*, 16), int(*'29'*, 16), int(*'c5'*, 16), int(*'89'*, 16), int(

*'6f'*, 16), int(*'b7'*, 16), int(*'62'*, 16), int(*'0e'*, 16), int(*'aa'*, 16), int(*'18'*, 16), int(*'be'*, 16), int(*'1b'*, 16)],

[int(*'fc'*, 16), int(*'56'*, 16), int(*'3e'*, 16), int(*'4b'*, 16), int(*'c6'*, 16), int(*'d2'*, 16), int(*'79'*, 16), int(*'20'*, 16), int(

*'9a'*, 16), int(*'db'*, 16), int(*'c0'*, 16), int(*'fe'*, 16), int(*'78'*, 16), int(*'cd'*, 16), int(*'5a'*, 16), int(*'f4'*, 16)],

[int(*'1f'*, 16), int(*'dd'*, 16), int(*'a8'*, 16), int(*'33'*, 16), int(*'88'*, 16), int(*'07'*, 16), int(*'c7'*, 16), int(*'31'*, 16), int(

*'b1'*, 16), int(*'12'*, 16), int(*'10'*, 16), int(*'59'*, 16), int(*'27'*, 16), int(*'80'*, 16), int(*'ec'*, 16), int(*'5f'*, 16)],

[int(*'60'*, 16), int(*'51'*, 16), int(*'7f'*, 16), int(*'a9'*, 16), int(*'19'*, 16), int(*'b5'*, 16), int(*'4a'*, 16), int(*'0d'*, 16), int(

*'2d'*, 16), int(*'e5'*, 16), int(*'7a'*, 16), int(*'9f'*, 16), int(*'93'*, 16), int(*'c9'*, 16), int(*'9c'*, 16), int(*'ef'*, 16)],

[int(*'a0'*, 16), int(*'e0'*, 16), int(*'3b'*, 16), int(*'4d'*, 16), int(*'ae'*, 16), int(*'2a'*, 16), int(*'f5'*, 16), int(*'b0'*, 16), int(

*'c8'*, 16), int(*'eb'*, 16), int(*'bb'*, 16), int(*'3c'*, 16), int(*'83'*, 16), int(*'53'*, 16), int(*'99'*, 16), int(*'61'*, 16)],

[int(*'17'*, 16), int(*'2b'*, 16), int(*'04'*, 16), int(*'7e'*, 16), int(*'ba'*, 16), int(*'77'*, 16), int(*'d6'*, 16), int(*'26'*, 16), int(

*'e1'*, 16), int(*'69'*, 16), int(*'14'*, 16), int(*'63'*, 16), int(*'55'*, 16), int(*'21'*, 16), int(*'0c'*, 16), int(*'7d'*, 16)]]

res = inv\_aes\_sbox[b1][b2]

return res

*'''*

*ENCRYPTION METHOD:*

*Method that encrypts the given plaintext in ECB or CBC format.*

*Returns the Ciphertext once encrypted.*

*'''*

def **EncryptCiphertext**(plaintext, key, CBC):

Ciphertext = *""*

*'Encrypts first block of plaintext'*

Ciphertext = Ciphertext + Encryption(plaintext[0: 0 + 32], key)

*'Loop to encrypt remaining block of plaintexts'*

for i in range (32, len(plaintext), 32):

if(CBC):

str = *""*

j = 0

k = i

while(j < 4):

str = str + XORtemps(plaintext[k: k + 8], Ciphertext[k - 32 :k - 24])

k = k + 8

j = j + 1;

Ciphertext = Ciphertext + Encryption(str, key)

else:

Ciphertext = Ciphertext + Encryption(plaintext[i: i + 32], key)

print(*"Final Ciphertext is: "* + Ciphertext)

return Ciphertext

*'''*

*Encryption Helper Function that takes in plaintext string (32 byte) and performs the rounds for encryption depending on the key size.*

*'''*

def **Encryption**(input, key):

i = 0

EncryptionString = *""*

if len(key) == 32:

ExpandedKey = KeyExpansion128(key)

while (i < 320):

if(i == 0):

oneRoundArr = oneRoundProccess(input,ExpandedKey[i: i + 32])

EncryptionString = MixColoumns(oneRoundArr)

else:

oneRoundArr = oneRoundProccess(EncryptionString, ExpandedKey[i: i + 32])

if(i != 288):

EncryptionString = MixColoumns(oneRoundArr)

else:

EncryptionString = ConvertArrayToString(oneRoundArr)

i = i + 32

EncryptionString = AddRound(EncryptionString, ExpandedKey[i: i + 32])

print(*"Final Encryption String: "* + EncryptionString)

return EncryptionString

elif len(key) == 48:

ExpandedKey = KeyExpansion192(key)

while (i < 384):

if(i == 0):

oneRoundArr = oneRoundProccess(input,ExpandedKey[i: i + 32])

EncryptionString = MixColoumns(oneRoundArr)

else:

oneRoundArr = oneRoundProccess(EncryptionString, ExpandedKey[i: i + 32])

if(i != 352):

EncryptionString = MixColoumns(oneRoundArr)

else:

EncryptionString = ConvertArrayToString(oneRoundArr)

i = i + 32

EncryptionString = AddRound(EncryptionString, ExpandedKey[i: i + 32])

print(*"Final Encryption String: "* + EncryptionString)

return EncryptionString

elif len(key) == 64:

ExpandedKey = KeyExpansion256(key)

while (i < 448):

if(i == 0):

oneRoundArr = oneRoundProccess(input,ExpandedKey[i: i + 32])

EncryptionString = MixColoumns(oneRoundArr)

else:

oneRoundArr = oneRoundProccess(EncryptionString, ExpandedKey[i: i + 32])

if(i != 416):

EncryptionString = MixColoumns(oneRoundArr)

else:

EncryptionString = ConvertArrayToString(oneRoundArr)

i = i + 32

EncryptionString = AddRound(EncryptionString, ExpandedKey[i: i + 32])

print(*"Final Encryption String: "* + EncryptionString)

return EncryptionString

else:

print(*"Invalid Key Length"*)

*'''*

*Computes the cipher text for one round encryption.*

*'''*

def **oneRoundProccess**(input, key):

if(len(input) < len(key)):

if(len(input) % 2 == 0):

while(len(input) != len(key)):

input = input + *"0"*

else:

print(*"Odd Length of Input"*)

input = input[0:len(input)-1] + *"0"* + input[len(input) - 1: len(input)]

print(*"Adding 0 in between : "* + input)

while(len(input) != len(key)):

input = input + *"0"*

print(*"Input is "* + input)

str = *""*

i = 0

while(i < len(key)):

str = str + XORtemps(input[i: i + 8], key[i:i + 8])

i = i + 8

print(*"First Step (Add Round) is"* + str)

j = 0

t1 = *""*

while(j < len(key)):

t1 = t1 + lookUp(str[j: j + 8])

j = j + 8

print(*"SubBytes: "* + t1)

arr = ConvertStringToMatrix(t1)

arr = ShiftRows(arr)

print(arr)

return arr

*'''*

*Helper method that converts string to matrix.*

*'''*

def **ConvertStringToMatrix**(t1):

rows, cols = (4, 4)

arr = [[*"0"* for i in range(cols)] for j in range(rows)]

k = 0

while(k < 4):

i = 0

while(i < 4):

arr[i][k] = t1[i\*2 : (i\*2 + 2)]

i = i + 1

t1 = t1[8: len(t1)]

k = k + 1

print(arr)

return arr

def **AddRound**(temp, key):

i = 0

res = *""*

while(i < len(key)):

res = res + XORtemps(temp[i: i + 8], key[i:i + 8])

i = i + 8

return res

print(res)

*'''*

*Helper methods to convert array to string*

*'''*

def **ConvertArrayToString**(arr):

arrString = *""*

for i in range (0 , 4):

for j in range (0, 4):

arrString = arrString + arr[j][i]

print(arrString)

return arrString

def **shiftLeft**(word, n):

return word[n:] + word[:n]

def **shiftRight**(word, n):

return word[len(word) - n: len(word)] + word[:len(word) - n]

*'''*

*Shift rows to left*

*'''*

def **invShiftRows**(temp):

temp[0] = shiftRight(temp[0], 0)

temp[1] = shiftRight(temp[1], 1)

temp[2] = shiftRight(temp[2], 2)

temp[3] = shiftRight(temp[3], 3)

return temp

*'''*

*Shift Rows to right.*

*'''*

def **ShiftRows**(temp):

temp[0] = shiftLeft(temp[0], 0)

temp[1] = shiftLeft(temp[1], 1)

temp[2] = shiftLeft(temp[2], 2)

temp[3] = shiftLeft(temp[3], 3)

return temp

*'''*

*Multiplies the matrix with the plain text to encrypt it*

*'''*

def **MixColoumns**(temp):

mc = [[*'02'*, *'03'*, *'01'*, *'01'*], [*'01'*, *'02'*, *'03'*, *'01'*], [*'01'*, *'01'*, *'02'*, *'03'*], [*'03'*, *'01'*, *'01'*, *'02'*]]

print(*"temp is: "*)

print(temp)

output = *""*

for k in range (0, 4):

for j in range (0, 4):

resByte = [*'0'*,*'0'*,*'0'*,*'0'*,*'0'*,*'0'*,*'0'*,*'0'*]

for i in range(0, 4):

print(*"Spec value: "* + mc[j][i])

print(*"Temp Value: "* + temp[i][k])

b1 = list(bin(int(mc[j][i], 16))[2:].zfill(8))

b2 = list(bin(int(temp[i][k], 16))[2:].zfill(8))

print(b1)

print(b2)

newByte = reducedPoly(b1, b2)

resByte = addPolynomial(newByte, resByte)

print(*"Byte 1: "*)

print(resByte)

res2 = hex(int(*""*.join(resByte), 2))

res2 = str(res2)[len(res2) - 2:]

print(*"Res2 is: "*)

if *'x'* in res2:

res2 = res2.replace(*'x'*, *'0'*)

print(res2)

print(*"Byte 1: "*)

print(resByte)

res1 = hex(int(*""*.join(resByte), 2))

res1 = str(res1)[len(res1) - 2:]

print(*"Encryption bytes: "*)

if *'x'* in res1:

res1 = res1.replace(*'x'*, *'0'*)

output = output + res1

print(output)

print(output)

return output

*'''*

*Inverses the Mix columns method.*

*'''*

def **invMixColoumns**(temp):

mc = [[*'0e'*, *'0b'*, *'0d'*, *'09'*], [*'09'*, *'0e'*, *'0b'*, *'0d'*], [*'0d'*, *'09'*, *'0e'*, *'0b'*], [*'0b'*, *'0d'*, *'09'*, *'0e'*]]

output = *""*

for k in range (0, 4):

for j in range (0, 4):

resByte = [*'0'*,*'0'*,*'0'*,*'0'*,*'0'*,*'0'*,*'0'*,*'0'*]

for i in range(0, 4):

b1 = list(bin(int(mc[j][i], 16))[2:].zfill(8))

b2 = list(bin(int(temp[i][k], 16))[2:].zfill(8))

newByte = reducedPoly(b1, b2)

resByte = addPolynomial(newByte, resByte)

res2 = hex(int(*""*.join(resByte), 2))

res2 = str(res2)[len(res2) - 2:]

if *'x'* in res2:

res2 = res2.replace(*'x'*, *'0'*)

print(res2)

res1 = hex(int(*""*.join(resByte), 2))

res1 = str(res1)[len(res1) - 2:]

if *'x'* in res1:

res1 = res1.replace(*'x'*, *'0'*)

output = output + res1

print(*"Encryption bytes: "*)

print(output)

return output

*'''*

*DECRYPTION METHOD:*

*Decrypts the ciphertext to give the correct plaintext using ciphertext, key and boolean value for CBC OR EBC.*

*'''*

def **DecryptCiphertext**(cipherText, key, CBC):

plaintext = *""*

plaintext = plaintext + Decryption(cipherText[0: 0 + 32], key)

for i in range (32, len(cipherText), 32):

plaintext = plaintext + Decryption(cipherText[i: i + 32], key)

if(CBC):

str = *""*

j = 0

k = i

while(j < 4):

str = str + XORtemps(plaintext[k: k + 8], cipherText[k - 32 :k - 24])

k = k + 8

j = j + 1;

plaintext = plaintext.replace(plaintext[i:i+32], str)

print(*"Final PlainText is: "* + plaintext)

return plaintext

*'''*

*Decryption helper that takes in ciphertext and key and returns plaintext*

*'''*

def **Decryption**(cipherText, key):

print(*"Recieved CipherText is: "* + cipherText)

originalText = *""*

if len(key) == 32:

i = 352

ExpandedKey = KeyExpansion128(key)

originalText = AddRound(cipherText, ExpandedKey[i - 32: i])

i = i - 32

originalText = invOneRoundProcess(originalText, ExpandedKey[i - 32: i])

print(*"Original Text: "* + originalText)

i = i - 32

while (i > 0):

arr = ConvertStringToMatrix(originalText)

originalText = invMixColoumns(arr)

originalText = invOneRoundProcess(originalText, ExpandedKey[i- 32: i])

i = i - 32

print(*"At the end of loop: "* + originalText)

print(*"Final Plaintext : "* + originalText)

return originalText

elif len(key) == 48:

i = 416

ExpandedKey = KeyExpansion192(key)

originalText = AddRound(cipherText, ExpandedKey[i - 32: i])

i = i - 32

originalText = invOneRoundProcess(originalText, ExpandedKey[i - 32: i])

print(*"Original Text: "* + originalText)

i = i - 32

while (i > 0):

arr = ConvertStringToMatrix(originalText)

originalText = invMixColoumns(arr)

originalText = invOneRoundProcess(originalText, ExpandedKey[i- 32: i])

i = i - 32

print(*"At the end of loop: "* + originalText)

print(*"Final Plaintext : "* + originalText)

return originalText

elif len(key) == 64:

i = 480

ExpandedKey = KeyExpansion256(key)

originalText = AddRound(cipherText, ExpandedKey[i - 32: i])

i = i - 32

originalText = invOneRoundProcess(originalText, ExpandedKey[i - 32: i])

print(*"Original Text: "* + originalText)

i = i - 32

while (i > 0):

arr = ConvertStringToMatrix(originalText)

originalText = invMixColoumns(arr)

originalText = invOneRoundProcess(originalText, ExpandedKey[i- 32: i])

i = i - 32

print(*"At the end of loop: "* + originalText)

print(*"Final Plaintext : "* + originalText)

return originalText

else:

print(*"Invalid Key Length"*)

*'''*

*Inverses the one round for Decryption*

*'''*

def **invOneRoundProcess**(input, key):

str = *""*

t1 = *""*

arr = ConvertStringToMatrix(input)

arr = invShiftRows(arr)

print(*"Shifted array is: "*)

print(arr)

t1 = ConvertArrayToString(arr)

print(*"t1 is: "* + t1)

j = 0

output = *""*

while(j < len(key)):

output = output + invLookUp(t1[j: j + 8])

j = j + 8

print(*"SubBytes: "* + output)

i = 0

while(i < len(key)):

str = str + XORtemps(output[i: i + 8], key[i:i + 8])

i = i + 8

return str

DecryptCiphertext(*"a5360648c5a07b8b0d32526666d6956740ff173728e3873e0f369e0eccdaf8b5707e16aa4879b76e81719c449e710b8f003140671445d240e4223fa7d10f834774496b0c743721f6e7cb222b5a69a41aa37370002db9a29e7301013960c91068"*, *"4e0e01285b1ff23909b11b5de4ea01c11acf4a713a66f782"*, True)

EncryptCiphertext(*"1526154061b689e0f00a5c2ff1ec19e4"*, *"30190dcc24585301f5bfc5b666c84775"*, False)