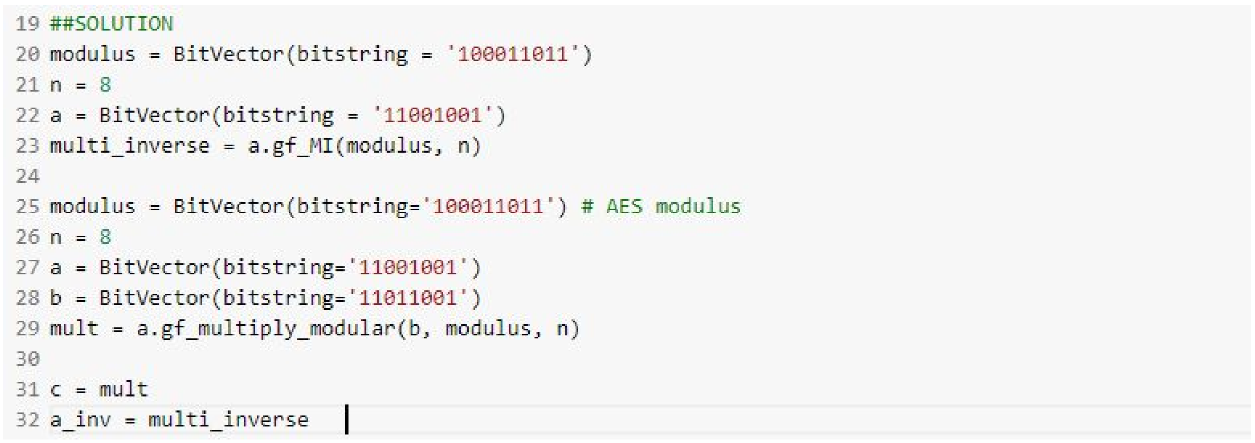
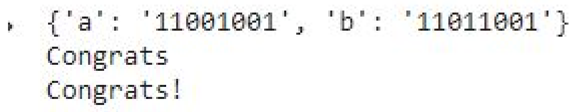
*Below, you can find a detailed explanation of each of my solutions. Codes have been provided in 3 different ways to be on the safe side. You can find all the source codes in my homework folder, their names are respective to the question number. You can also access the codes by clicking on the links that I provided at each question, this will direct you to the Colab file of the solution. In case you want to take a look at the solutions while reading the explanations and not to switch between different windows, I put the screenshots of the codes and the outputs as well.* ***Since there are 3 explanation based questions, I prefer to have a .pdf file rather than a .txt file, I hope this is not a problem at all.***

**Question 1:** (Access the Colab file)

I used the gf_MI() to find the inverse of a polynomial in GF(2^8) and gf_multiply_modular() function to multiply 2 polynomials. I have uploaded the .py file inside the zipped folder, the name is q1.py. The answer I got from the server is “Congrats” twice.



```
##SOLUTION
modulus = BitVector(bitstring = '100011011')
n = 8
a = BitVector(bitstring = '11001001')
multi_inverse = a.gf_MI(modulus, n)

modulus = BitVector(bitstring='100011011') # AES modulus
n = 8
a = BitVector(bitstring='11001001')
b = BitVector(bitstring='11011001')
mult = a.gf_multiply_modular(b, modulus, n)

c = mult
a_inv = multi_inverse
```



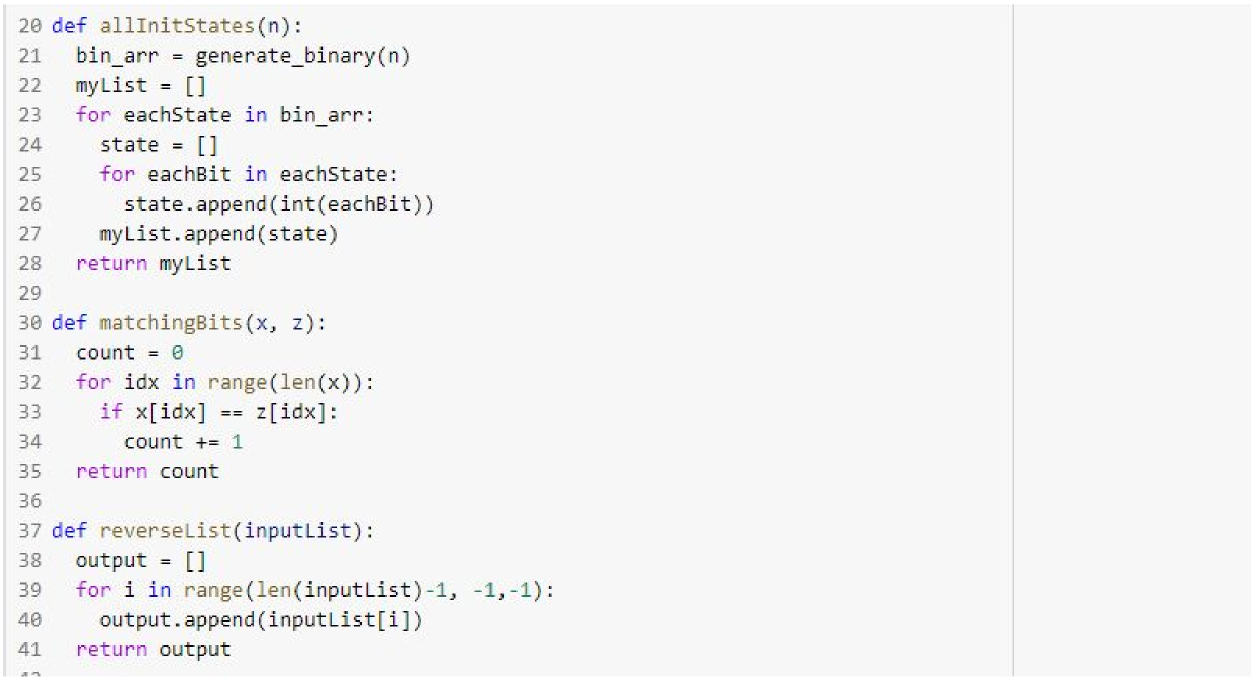
```
{'a': '11001001', 'b': '11011001'}
Congrats
Congrats!
```

**Question 2:** (Access the Colab file)

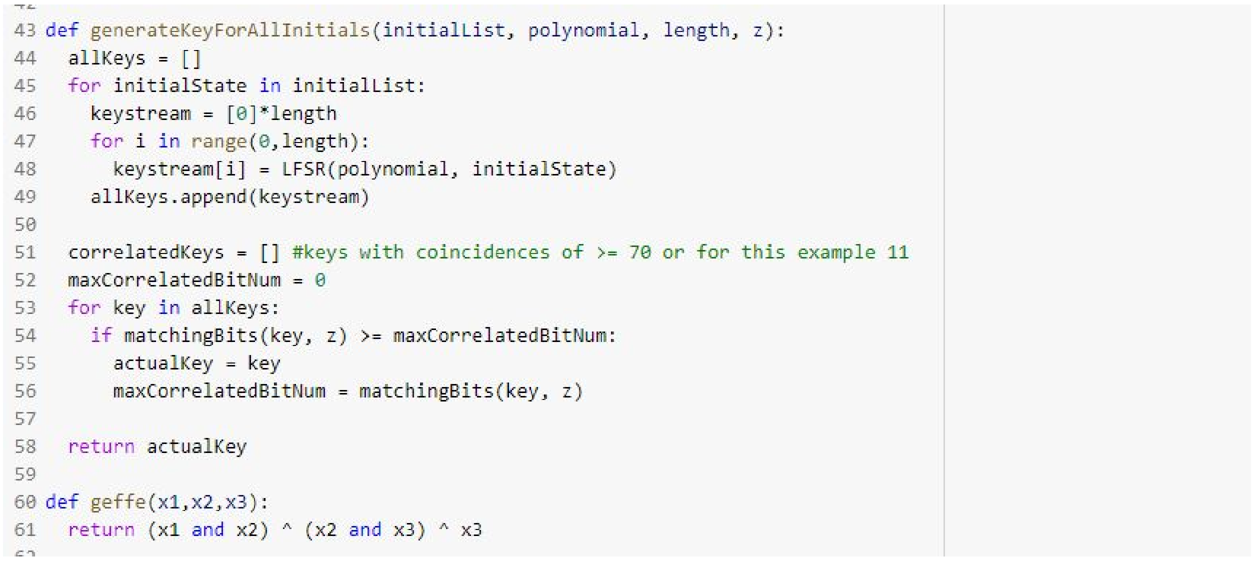
In the Geffe generator, we know that x1 and x3 are correlated with the output sequence z, where x1 and x3 are the keystreams generated by the LFSR1 and LFSR3 respectively. So the first step is to make an exhaustive search on their initial states. The length of the LFSR1 is 14, then the number of bits in the initial state is 14. I created a list of 14-bit sequences that is likely to be the initial state of the LFSR1, inside the function allInitStates(). The length of this list is 2^14. Then, inside a function called generateKeyForAllInitials(), I compute all possible keys that correspond to each element of the initial state list using the provided lfsr() function. For each possible keystream, I compute the coincidences with the z sequence. The keystream where the number of matching bits with the sequence z is maximum is the most likely one to be the keystream of LFSR1. The maximum matching bits for the first LFSR is 80, which is acceptable since 110*0.75 = 82.5. Then I repeat this process for LFSR3 as well since this is also correlated. For the most probable keystream of LFSR3, the number of matching bits is 87, which is pretty satisfying. Since we find the initial states of the LFSR1 and LFSR3, what we can do is generating all possible initial states of LFSR2 (there are 2^17 possible initial states) for each of these initial states, compute the keystream, and put it to the geffe generator with the keystreams that we get from the LFSR1 and LFSR3. If the combined sequence is exactly equal to z, given in the question, we find the keystream for LFSR2 as well.

Since at the beginning of my main program, I reversed the sequence z and proceed the algorithm in that way but I was not supposed to do so, I also reversed the keystreams that I got from these functions to correct the flow of the program. Then the initial states are the reversed versions of the first 14, 17, and 11 bits of the keystreams respectively. Below you can see the initial states:

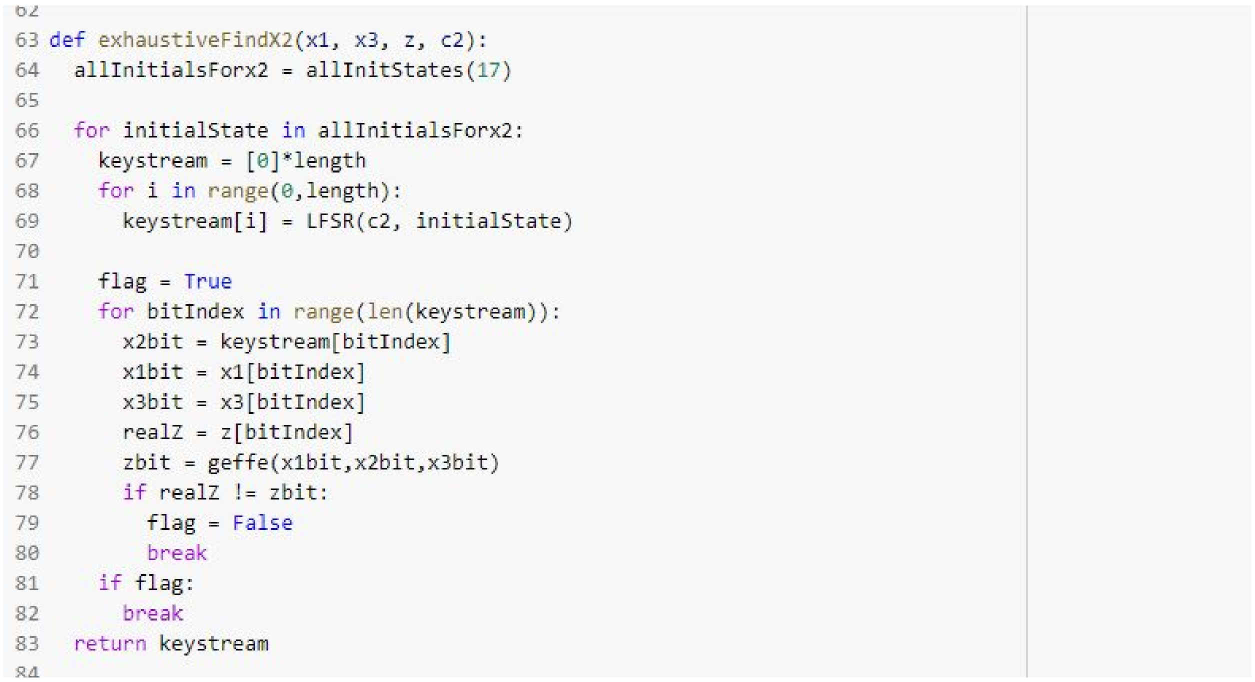
```
Initial state for LFSR1:
[1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1]

Initial state for LFSR2:
[0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0]

Initial state for LFSR3:
[1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0]
```



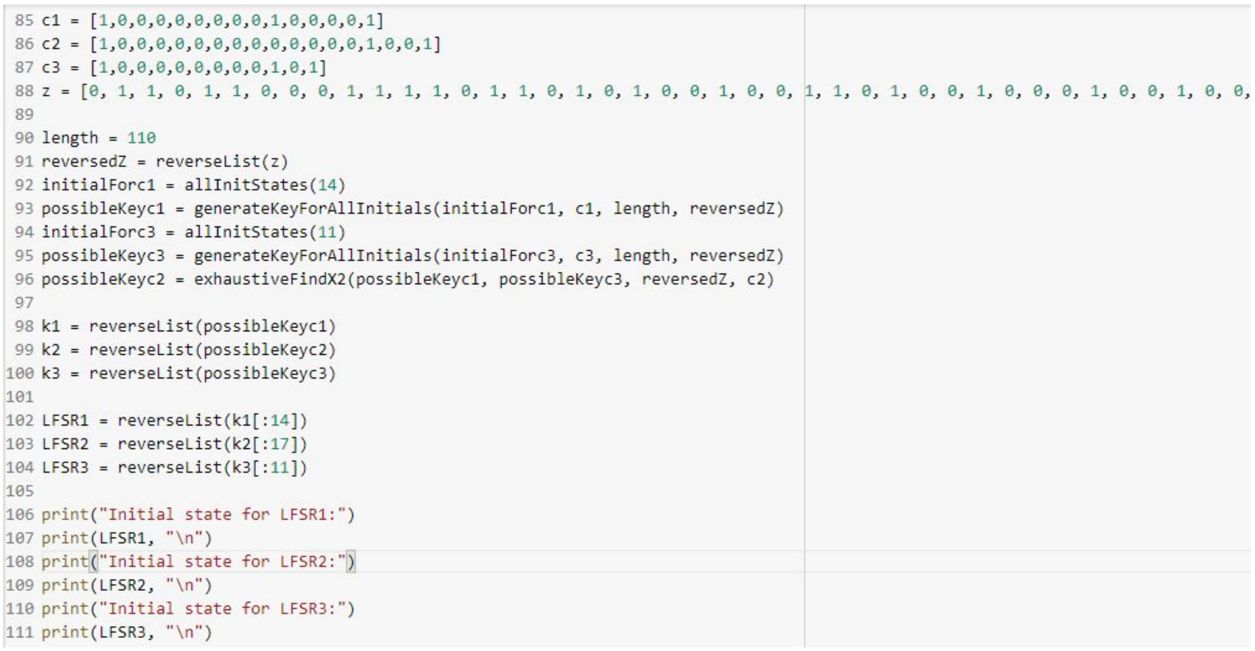
```python
def allInitStates(n):
  bin_arr = generate_binary(n)
  myList = []
  for eachState in bin_arr:
    state = []
    for eachBit in eachState:
      state.append(int(eachBit))
    myList.append(state)
  return myList

def matchingBits(x, z):
  count = 0
  for idx in range(len(x)):
    if x[idx] == z[idx]:
      count += 1
  return count

def reverseList(inputList):
  output = []
  for i in range(len(inputList)-1, -1,-1):
    output.append(inputList[i])
  return output
```



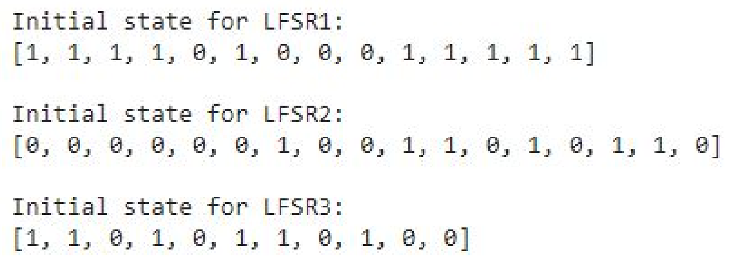
```python
def generateKeyForAllInitials(initialList, polynomial, length, z):
  allKeys = []
  for initialState in initialList:
    keystream = [0]*length
    for i in range(0,length):
      keystream[i] = LFSR(polynomial, initialState)
    allKeys.append(keystream)

  correlatedKeys = [] #keys with coincidences of >= 70 or for this example 11
  maxCorrelatedBitNum = 0
  for key in allKeys:
    if matchingBits(key, z) >= maxCorrelatedBitNum:
      actualKey = key
      maxCorrelatedBitNum = matchingBits(key, z)

  return actualKey

def geffe(x1,x2,x3):
  return (x1 and x2) ^ (x2 and x3) ^ x3
```



```python
def exhaustiveFindX2(x1, x3, z, c2):
  allInitialsForx2 = allInitStates(17)

  for initialState in allInitialsForx2:
    keystream = [0]*length
    for i in range(0,length):
      keystream[i] = LFSR(c2, initialState)

    flag = True
    for bitIndex in range(len(keystream)):
      x2bit = keystream[bitIndex]
      x1bit = x1[bitIndex]
      x3bit = x3[bitIndex]
      realZ = z[bitIndex]
      zbit = geffe(x1bit,x2bit,x3bit)
      if realZ != zbit:
        flag = False
        break
    if flag:
      break
  return keystream
```



```
c1 = [1,0,0,0,0,0,0,0,0,1,0,0,0,0,1]
c2 = [1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,1]
c3 = [1,0,0,0,0,0,0,0,0,1,0,1]
z = [0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0,

length = 110
reversedZ = reverseList(z)
initialForc1 = allInitStates(14)
possibleKeyc1 = generateKeyForAllInitials(initialForc1, c1, length, reversedZ)
initialForc3 = allInitStates(11)
possibleKeyc3 = generateKeyForAllInitials(initialForc3, c3, length, reversedZ)
possibleKeyc2 = exhaustiveFindX2(possibleKeyc1, possibleKeyc3, reversedZ, c2)

k1 = reverseList(possibleKeyc1)
k2 = reverseList(possibleKeyc2)
k3 = reverseList(possibleKeyc3)

LFSR1 = reverseList(k1[:14])
LFSR2 = reverseList(k2[:17])
LFSR3 = reverseList(k3[:11])

print("Initial state for LFSR1:")
print(LFSR1, "\n")
print("Initial state for LFSR2:")
print(LFSR2, "\n")
print("Initial state for LFSR3:")
print(LFSR3, "\n")
```



```
Initial state for LFSR1:
[1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1]

Initial state for LFSR2:
[0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0]

Initial state for LFSR3:
[1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0]
```

**Question 3:**
**Part a:**

Linear complexity:

$$L = 79 * 85 + 79 * 97 + 85 * 97 + 79 * 85 * 97 = 672448$$

Period:

$$T = (2^{79} - 1) * (2^{85} - 1) + (2^{79} - 1) * (2^{97} - 1) + (2^{85} - 1) * (2^{97} - 1) + (2^{79} - 1) * (2^{85} - 1) * (2^{97} - 1)$$
$$= 3.70534686 * 10^{78}$$

**Part b:**

Nonlinearity degree: The maximum of the order of the terms in the combining function is 3, therefore the nonlinearity degree is 3.

Balance: For x1 = 00001111, x2 = 00110011 and x3 = 01010101 -> the sequence generated by this combining function is z = 00010110.
The number of 0's is not equal to the number of 1's. Therefore it is not balanced.

Correlation: When I observe the matching bits of the sequences, I realize that there is 5 matching bits out of 8 for each of x1,x2, and x3. To be uncorrelated, for an attacker to guess the next bit should be 50%, but since the number of matching bits is not 4 out of 8 but 5, the probability of guessing is 62.5%. Therefore, all of the sequences x1,x2, and x3 are correlated to output sequence z.

**Question 4:**

When we eliminate ShiftRow and MixColumn layers, we have just 2 layers to encrypt: Key addition and Byte Substitution layers. In the Byte Substitution layer, we can use a lookup table of 256 entries because each byte in the matrix will have 16 different possible mappings. If we use a chosen-plaintext attack, where we know 256 plaintext-ciphertext pairs; we can break the AES. This is how we do: We already know which byte maps to which byte when it is at a certain position in the matrix using the lookup table in byte substitution. But we also have a Key Addition layer that we XOR the plaintext with the round key. Since we have the chance to choose the plaintext, if we choose 256 plaintexts which have the same bits internally, e.g [0,0,0,0,0,0,0] and [1,1,1,1,1,1,1,1] …. and [255,255,255,255,255,255,255,255]; then we understand the exact encryption from plaintext to ciphertext mapping as in the idea that we use in the Byte Substitution layer lookup table. In this way, for each possible value of a byte, for each possible different position in the matrix, we have a certain ciphertext byte in our newly created lookup table. Using that table, we can decrypt any given plaintext.

**Question 5:**

Let's say that the ciphertext block $C_i$ is corrupted. Then $P_i$ **is corrupted** because we decrypt the $C_i$ first and XOR with $C_{i-1}$, and $C_i$ is corrupted! Then, for $P_{i+1}$ we use the $C_i$ itself to XOR with $D_k(C_{i+1})$. So, $P_{i+1}$ **is also corrupted.** For the decryption of $P_{i+2}$, we use $C_{i+1}$ and $C_{i+2}$ which both are **not** corrupted. Therefore, after 2 blocks; CBC is resynchronized.