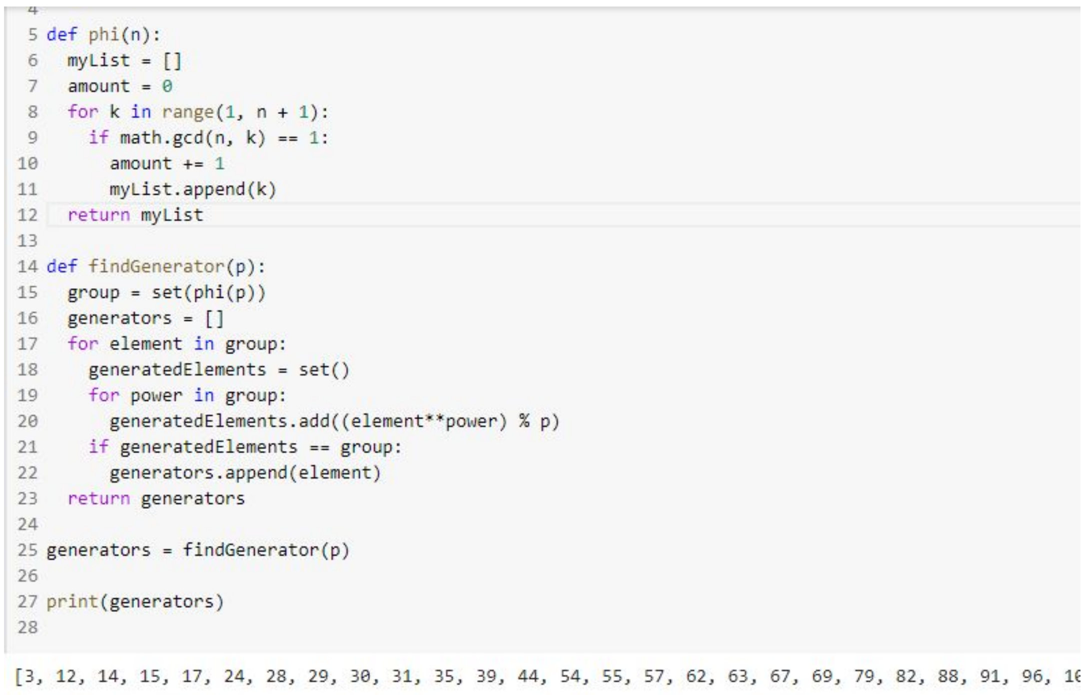
Below, you can find a detailed explanation of each of my solutions. Codes have been provided in 3 different ways to be on the safe side. You can find all the source codes in my homework folder, their names are respective to the question number. You can also access the codes by clicking on the links that 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.

Question 1: (To access the respective Colab file, click here.)

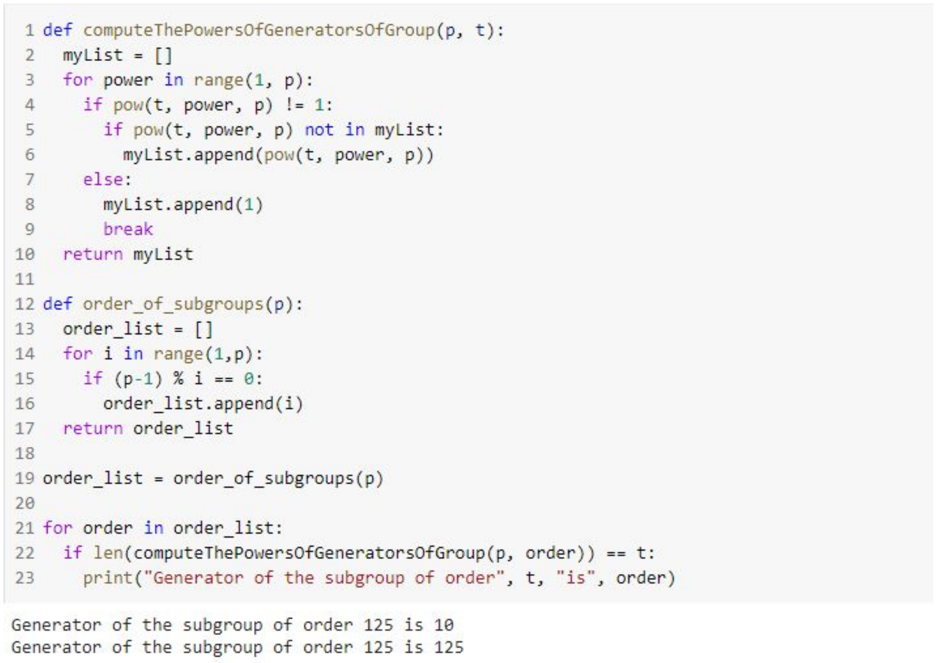
a) The numbers that I get from the server are as follows: p = 751 and t = 125. First, I updated the phi function in such a way that given a number, it returns the list of the numbers which are coprime to that number. Since p is a prime, the list returned from the phi function is the group of Z^*_{751} . Then, implemented the findGenerator function, which again takes a prime number and returns a list of integers that are generators of this group. What the function does is as follows: It iterates over the elements of the group, at each iteration another for loop iterates over the elements again and calculates the $base^{power}$ (mod p), adds it into a set, where the base is the current element of the outer loop, and the power is the current element of the inner loop. After the inner loop terminates, we check if the set that consists of the generated element is equivalent to the group itself: if that is the case, then that base is a generator and it is added into a generators list; else, continue the loop until the termination. I also checked the length of the generator list, which is equal to $\phi(751-1) = 200$, which confirms my findings. I sent the first generator element, which is 3, to the server back.



```python
def phi(n):
  myList = []
  amount = 0
  for k in range(1, n + 1):
    if math.gcd(n, k) == 1:
      amount += 1
      myList.append(k)
  return myList

def findGenerator(p):
  group = set(phi(p))
  generators = []
  for element in group:
    generatedElements = set()
    for power in group:
      generatedElements.add((element**power) % p)
    if generatedElements == group:
      generators.append(element)
  return generators

generators = findGenerator(p)

print(generators)
```

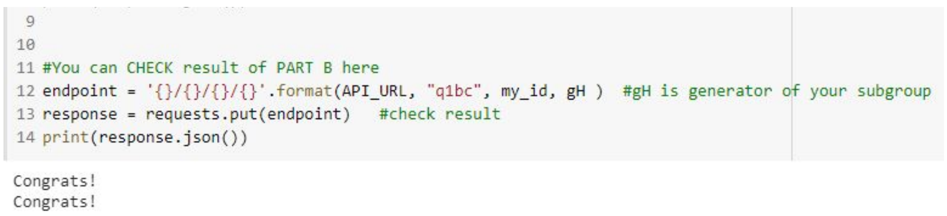
```
[3, 12, 14, 15, 17, 24, 28, 29, 30, 31, 35, 39, 44, 54, 55, 57, 62, 63, 67, 69, 79, 82, 88, 91, 96, 1
```

b) According to the Lagrange Theorem, we know that the order of the group is a multiple of the order of the subgroup. We also know that the powers of the numbers that divide the $\phi(n)$ which is the order of the group form a subgroup. Any subgroup would have an order that divides the order of the group. To find such numbers, I create a function called order_of_subgroups, which takes the prime integer p and calculates the all possible order of subgroups which are possible generators of the subgroups as well, and put them into a list. The return value is the list of integers which divides the order of the group. For each such number, I compute their powers to form the subgroups, using the function computeThePowersOfGeneratorsOfGroup() which takes the current order, which is the generator of the subgroup, and the prime p and iterates until the $order^{num}$ where $num \in Z_{751}^{*}$. We compute until then because we know that these groups are cyclic and repeat themselves after generating number 1. If that is the case, I add 1 as the last generated element and break my loop. At the end of the function, the list I have is the list of the elements of the current subgroup, generated by the numbers that divide the order of the group, which is the generator of the subgroup indeed. Then in the main part back, for each subgroup generator, I call the computeThePowersOfGeneratorsOfGroup() to have the subgroups and check if their order is t which is given by the server. If that is the case, this means I got the generators that generate the subgroups of order t. There are 2 such subgroup generators, one of them is 125 and the other is 10. You can see the answers I got from the server, below:



```python
def computeThePowersOfGeneratorsOfGroup(p, t):
  myList = []
  for power in range(1, p):
    if pow(t, power, p) != 1:
      if pow(t, power, p) not in myList:
        myList.append(pow(t, power, p))
    else:
      myList.append(1)
      break
  return myList

def order_of_subgroups(p):
  order_list = []
  for i in range(1,p):
    if (p-1) % i == 0:
      order_list.append(i)
  return order_list

order_list = order_of_subgroups(p)

for order in order_list:
  if len(computeThePowersOfGeneratorsOfGroup(p, order)) == t:
    print("Generator of the subgroup of order", t, "is", order)
```

```
Generator of the subgroup of order 125 is 10
Generator of the subgroup of order 125 is 125
```

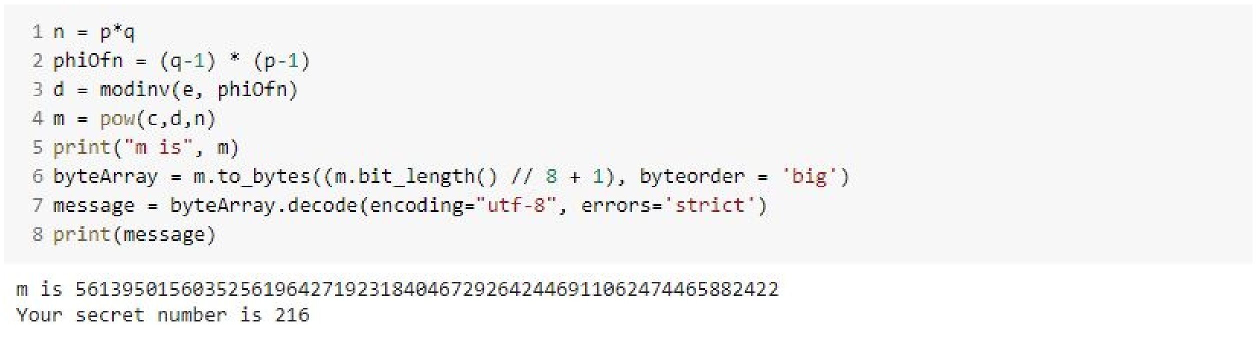


```python


#You can CHECK result of PART B here
endpoint = '{}/{}/{}/{}'.format(API_URL, "q1bc", my_id, gH )  #gH is generator of your subgroup
response = requests.put(endpoint)   #check result
print(response.json())
```

```
Congrats!
Congrats!
```

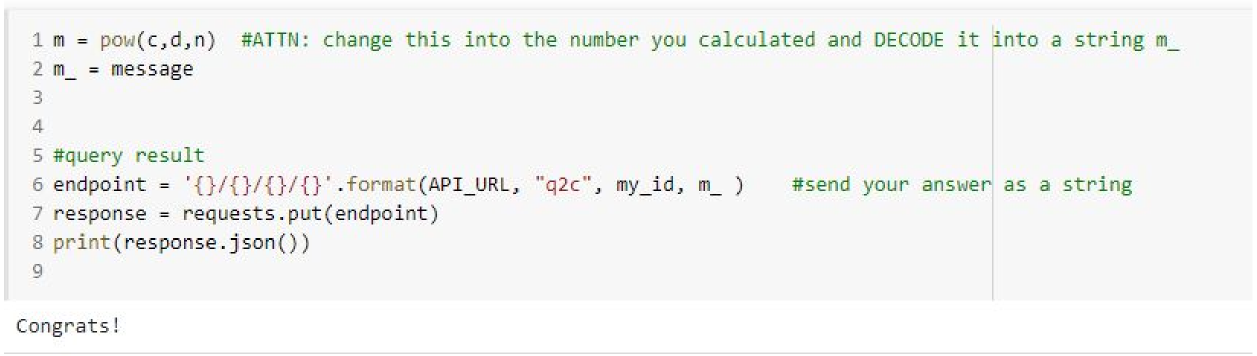
Question 2: (To access the respective Colab file, click here.)

First I got the numbers from the server. Then I pass the number e to the modinv() function which is provided to us. Since p and q are large numbers and primes according to the convention, their multiplication is a larger number which is the composite number n. Calculating the Totient function of this very big number n would take a long time, therefore we use the formula $\phi(n) = (p-1) * (q-1)$. Using these two values, phi(n) and the inverse of number e, we calculate the actual exponent d. Then I used the pow() function which takes 3 parameters: the base, the exponent, and the modulus. That's how I calculate the value m. As the last step, I convert the integer m into a byte array using the builtin to_byte function and decode that byte array using utf-8 which is Unicode into a string. The message I got is "`Your secret number is 216`". You can see the code and the response I got from the server, below:



```
n = p*q
phiOfn = (q-1) * (p-1)
d = modinv(e, phiOfn)
m = pow(c,d,n)
print("m is", m)
byteArray = m.to_bytes((m.bit_length() // 8 + 1), byteorder = 'big')
message = byteArray.decode(encoding="utf-8", errors='strict')
print(message)
```

```
m is 561395015603525619642719231840467292642446911062474465882422
Your secret number is 216
```



```
m = pow(c,d,n)  #ATTN: change this into the number you calculated and DECODE it into a string m_
m_ = message


#query result
endpoint = '{}/{}/{}/{}'.format(API_URL, "q2c", my_id, m_ )    #send your answer as a string
response = requests.put(endpoint)
print(response.json())

```

```
Congrats!
```

Question 3: (To access the respective Colab file, click here.)

We know that there is exactly one solution to $x \equiv b.a^{-1}(mod\ n)$ if gcd(a,n) = 1. If gcd(a,n) $\neq$ 1, then if b is not divisible by gcd(a,n) then there is no solution, else there is exactly gcd(a,n) solution, which can be obtained by $\frac{a}{d}\widehat{x} \equiv \frac{b}{d} mod \frac{n}{d}$, where d = gcd(a,n). Other gcd(a,n)-1 solutions are obtained by incrementing the solution by n/d.

I implemented a function called find_solutions, which takes a,b, and n as parameters. If the gcd(a,n) is equal to 1, then calculates the one and only solution. Else, we compute the gcd(a,n) to divide all numbers in the equation by gcd(a,n), if gcd(a,n) divides b. And calculate the solutions as I already explained.

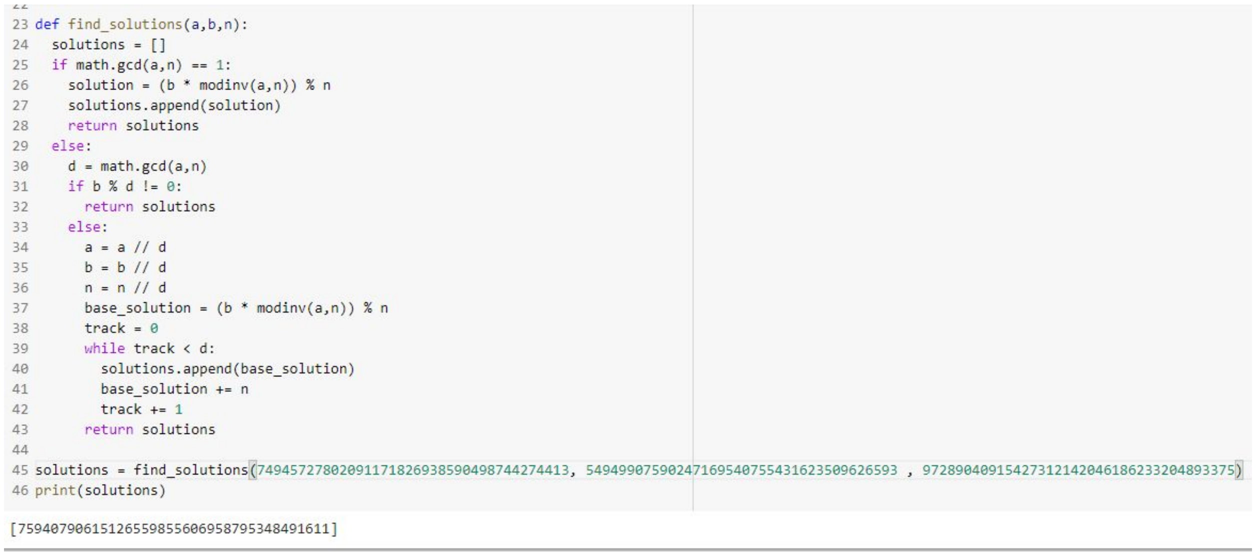
3.a. No solutions, because gcd(a,n) = 5 does not divide b.

3.b. There are exactly 5 solutions because gcd(a,n) = 5. Solutions are as follows:

```
[11368713749418004372789821825215865218,
30826521932503466801199059071856843893,
50284330115588929229608296318497822568,
69742138298674391658017533565138801243,
89199946481759854086426770811779779918]
```

3.c. There is exactly 1 solution because gcd(a,n) = 1. Solution is:

```
[7594079061512655985560695879534849161l]
```



```
def find_solutions(a,b,n):
  solutions = []
  if math.gcd(a,n) == 1:
    solution = (b * modinv(a,n)) % n
    solutions.append(solution)
    return solutions
  else:
    d = math.gcd(a,n)
    if b % d != 0:
      return solutions
    else:
      a = a // d
      b = b // d
      n = n // d
      base_solution = (b * modinv(a,n)) % n
      track = 0
      while track < d:
        solutions.append(base_solution)
        base_solution += n
        track += 1
      return solutions

solutions = find_solutions(74945727802091171826938590498744274413, 54949907590247169540755431623509626593 , 97289040915427312142046186233204893375)
print(solutions)
```

```
[7594079061512655985560695879534849161l]
```

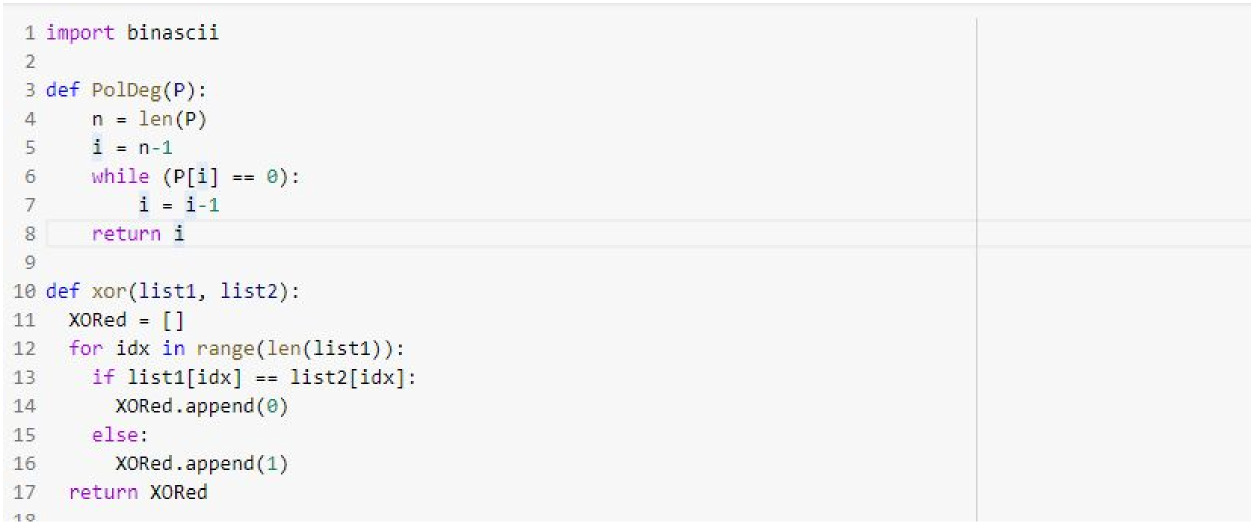
Question 4: (To access the respective Colab file, click here.)

For this question, one approach may be generating a random initial state, using the initial state and the given polynomial function calling the lfsr() function and let it generate a keystream. Then, we can use the findPeriod() function to calculate their periods. The periods must be equal to the 2^L -1 where L is the degree of the polynomial if the polynomial generates the maximum period sequences. But my approach is different than that.

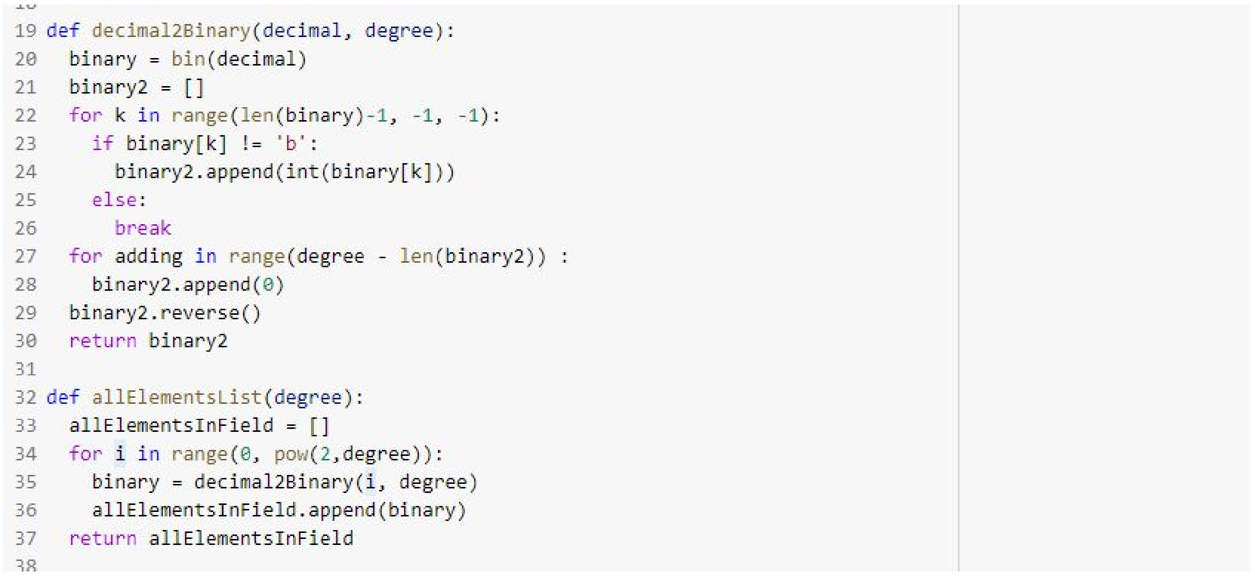
For a polynomial to generate the maximum period sequence, it should be primitive polynomial. Since we can observe these polynomials in GF(2^n) binary fields, I decide on the primitivity of the polynomial on that. For a polynomial of degree n, the polynomial should be able to generate the polynomials of degree < n, which corresponds to binary sequences on their own. At each step, I represent the polynomial as a byte list, the most significant bit is the element in index 0. I use the PolDeg() function to have the degree. I used my generator function to generate the polynomials of degree < n. This function has a loop that iterates 2^degree times unless a break exists unless we generate the polynomial of f(x) = 1 very soon. At each step, depending on the polynomial we have, we make a polynomial multiplication and reduce the polynomial if necessary. The code may seem complicated but is exactly equivalent to what we have in the slides. If necessary, I can make a demo on that. You can also find the screenshots of my code and output below.

The first polynomial $P_1(x) = x^7 + x^3 + x^2 + 1$ does not generate maximum period sequences because it is not primitive.

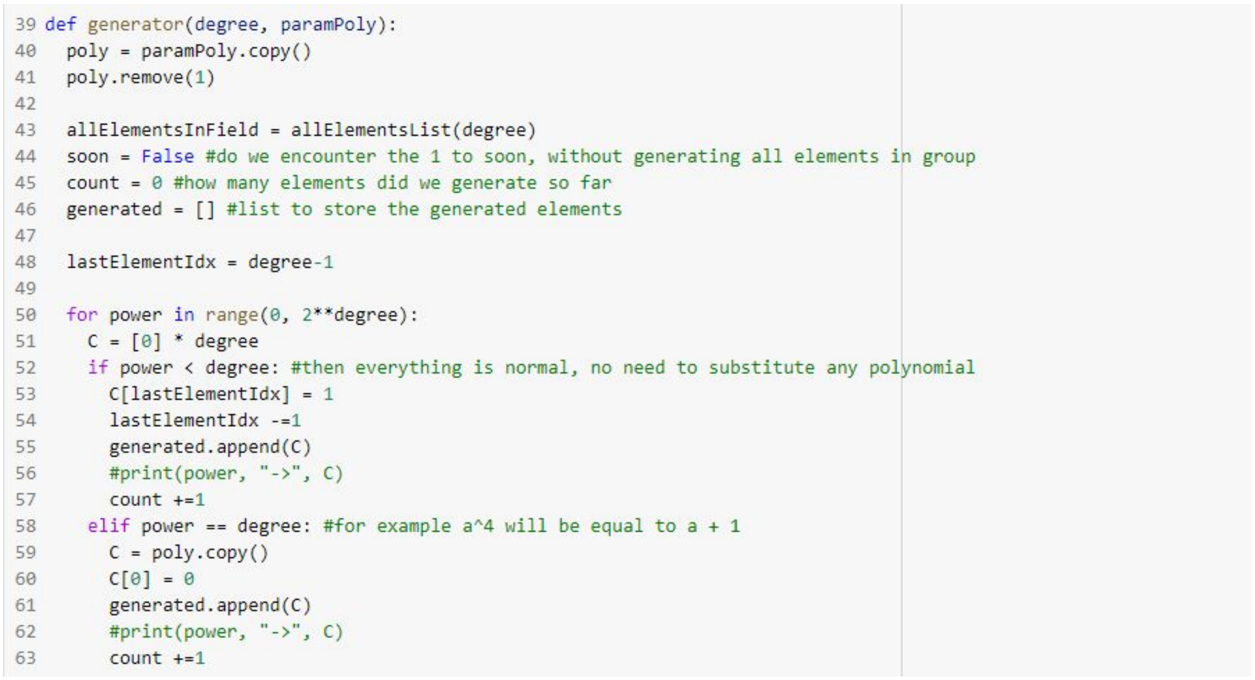
The second polynomial $P_2(x) = x^7 + x + 1$ generates the maximum period sequences because it is primitive.



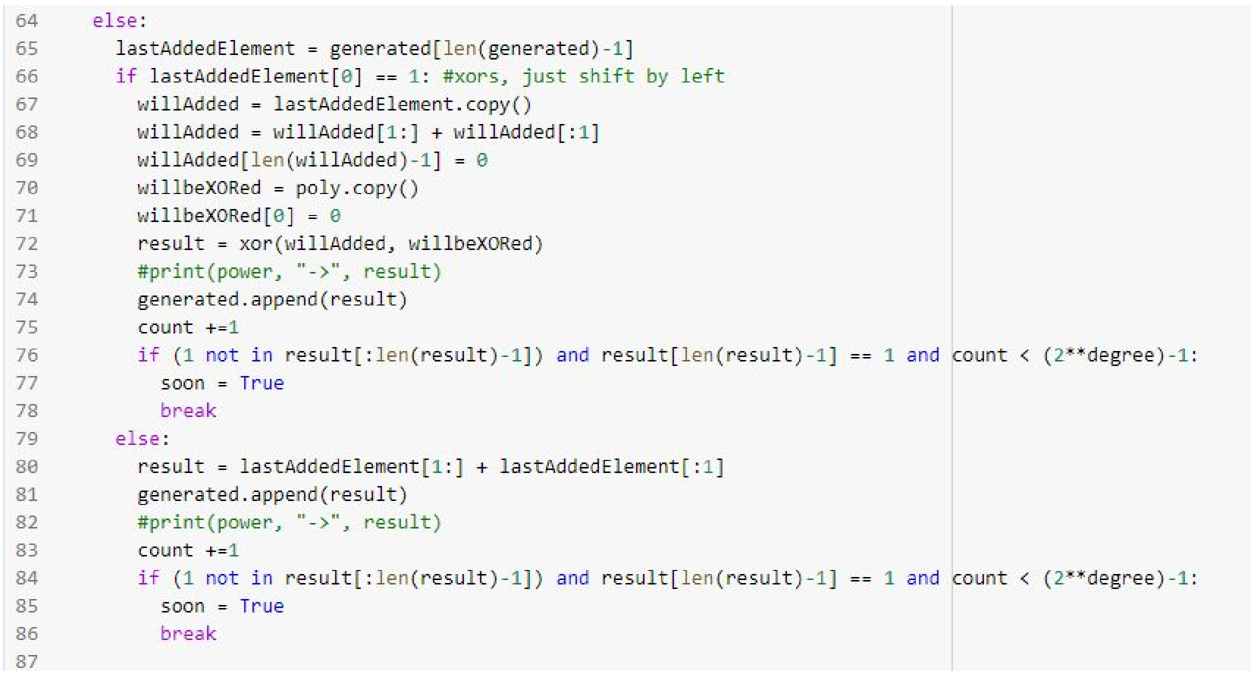
```python
import binascii

def PolDeg(P):
    n = len(P)
    i = n-1
    while (P[i] == 0):
        i = i-1
    return i

def xor(list1, list2):
  XORed = []
  for idx in range(len(list1)):
    if list1[idx] == list2[idx]:
      XORed.append(0)
    else:
      XORed.append(1)
  return XORed

```



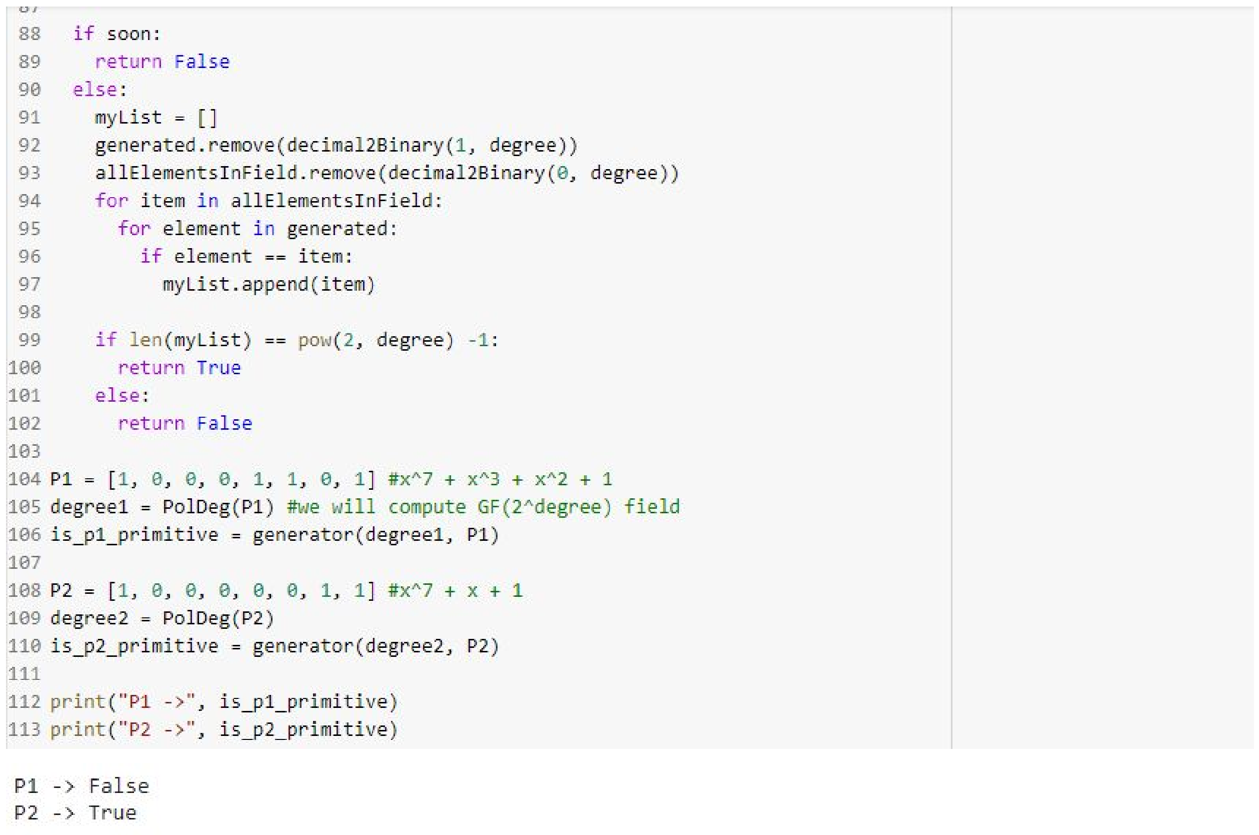
```python
def decimal2Binary(decimal, degree):
  binary = bin(decimal)
  binary2 = []
  for k in range(len(binary)-1, -1, -1):
    if binary[k] != 'b':
      binary2.append(int(binary[k]))
    else:
      break
  for adding in range(degree - len(binary2)) :
    binary2.append(0)
  binary2.reverse()
  return binary2

def allElementsList(degree):
  allElementsInField = []
  for i in range(0, pow(2,degree)):
    binary = decimal2Binary(i, degree)
    allElementsInField.append(binary)
  return allElementsInField

```



```python
def generator(degree, paramPoly):
  poly = paramPoly.copy()
  poly.remove(1)

  allElementsInField = allElementsList(degree)
  soon = False #do we encounter the 1 to soon, without generating all elements in group
  count = 0 #how many elements did we generate so far
  generated = [] #list to store the generated elements

  lastElementIdx = degree-1

  for power in range(0, 2**degree):
    C = [0] * degree
    if power < degree: #then everything is normal, no need to substitute any polynomial
      C[lastElementIdx] = 1
      lastElementIdx -=1
      generated.append(C)
      #print(power, "->", C)
      count +=1
    elif power == degree: #for example a^4 will be equal to a + 1
      C = poly.copy()
      C[0] = 0
      generated.append(C)
      #print(power, "->", C)
      count +=1
```



```python
    else:
      lastAddedElement = generated[len(generated)-1]
      if lastAddedElement[0] == 1: #xors, just shift by left
        willAdded = lastAddedElement.copy()
        willAdded = willAdded[1:] + willAdded[:1]
        willAdded[len(willAdded)-1] = 0
        willbeXORed = poly.copy()
        willbeXORed[0] = 0
        result = xor(willAdded, willbeXORed)
        #print(power, "->", result)
        generated.append(result)
        count +=1
        if (1 not in result[:len(result)-1]) and result[len(result)-1] == 1 and count < (2**degree)-1:
          soon = True
          break
      else:
        result = lastAddedElement[1:] + lastAddedElement[:1]
        generated.append(result)
        #print(power, "->", result)
        count +=1
        if (1 not in result[:len(result)-1]) and result[len(result)-1] == 1 and count < (2**degree)-1:
          soon = True
          break

```



```
  if soon:
    return False
  else:
    myList = []
    generated.remove(decimal2Binary(1, degree))
    allElementsInField.remove(decimal2Binary(0, degree))
    for item in allElementsInField:
      for element in generated:
        if element == item:
          myList.append(item)

    if len(myList) == pow(2, degree) -1:
      return True
    else:
      return False

P1 = [1, 0, 0, 0, 1, 1, 0, 1] #x^7 + x^3 + x^2 + 1
degree1 = PolDeg(P1) #we will compute GF(2^degree) field
is_p1_primitive = generator(degree1, P1)

P2 = [1, 0, 0, 0, 0, 0, 1, 1] #x^7 + x + 1
degree2 = PolDeg(P2)
is_p2_primitive = generator(degree2, P2)

print("P1 ->", is_p1_primitive)
print("P2 ->", is_p2_primitive)
```

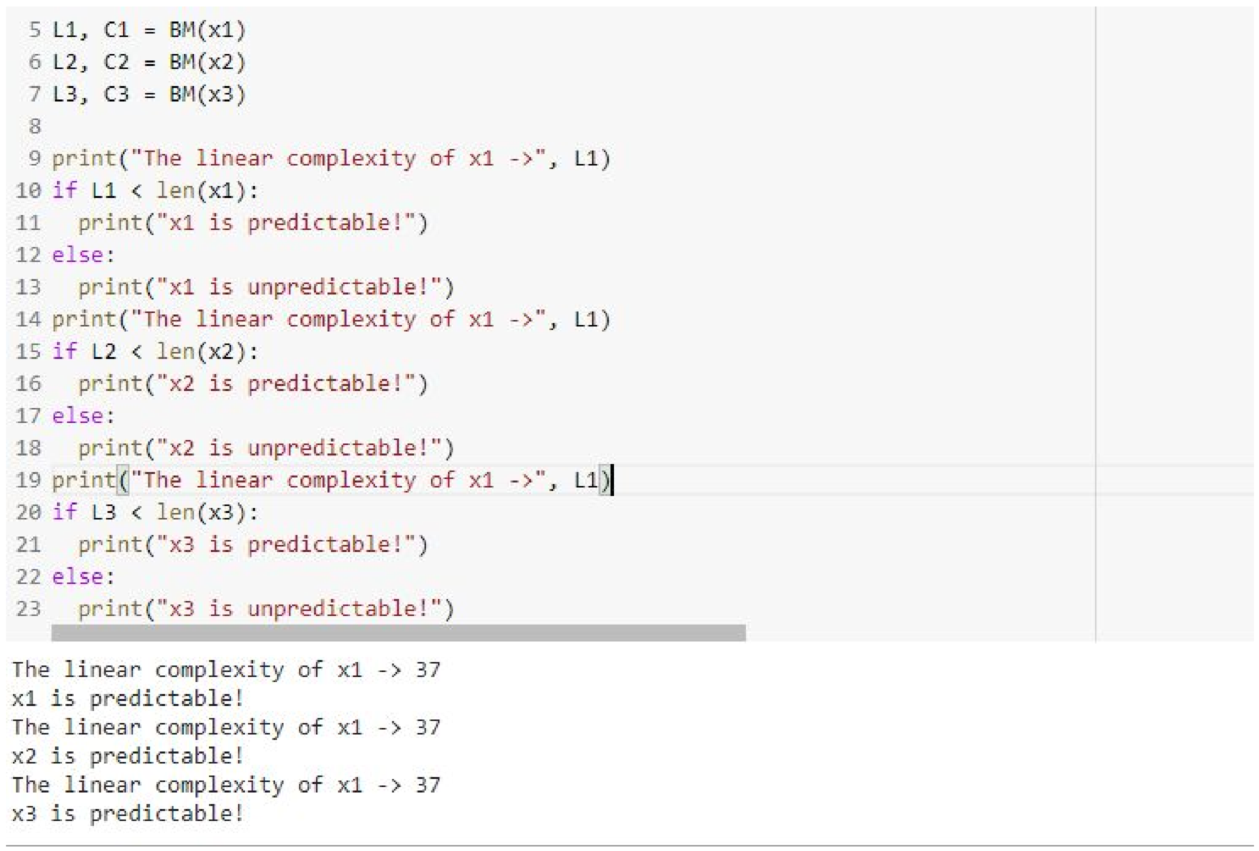
```
P1 -> False
P2 -> True
```

Question 5: (To access the respective Colab file, click here.)

We know that the expected linear complexity of a sequence should be

$$\approx n/2 \quad + 2/9 \,.$$

We also know that when n is large, constant 2/9 is effectless. Then, we can say that if the linear complexity of the sequences, x1 - x2 - x3, are smaller than < n/2, then they are predictable. To measure the linear complexities, I used BM() algorithm provided. I passed each of these sequences as a parameter to BM() function and their linear complexities are equal to 37, all of them. But the expected linear complexity should be n/2 where n is the length of the sequence. The lengths of the sequences are equal and 150, meaning that any linear complexity < 75 will make the sequence predictable. As a result, all of these sequences are indeed predictable sequences.



```
L1, C1 = BM(x1)
L2, C2 = BM(x2)
L3, C3 = BM(x3)

print("The linear complexity of x1 ->", L1)
if L1 < len(x1):
  print("x1 is predictable!")
else:
  print("x1 is unpredictable!")
print("The linear complexity of x1 ->", L1)
if L2 < len(x2):
  print("x2 is predictable!")
else:
  print("x2 is unpredictable!")
print("The linear complexity of x1 ->", L1)
if L3 < len(x3):
  print("x3 is predictable!")
else:
  print("x3 is unpredictable!")
```

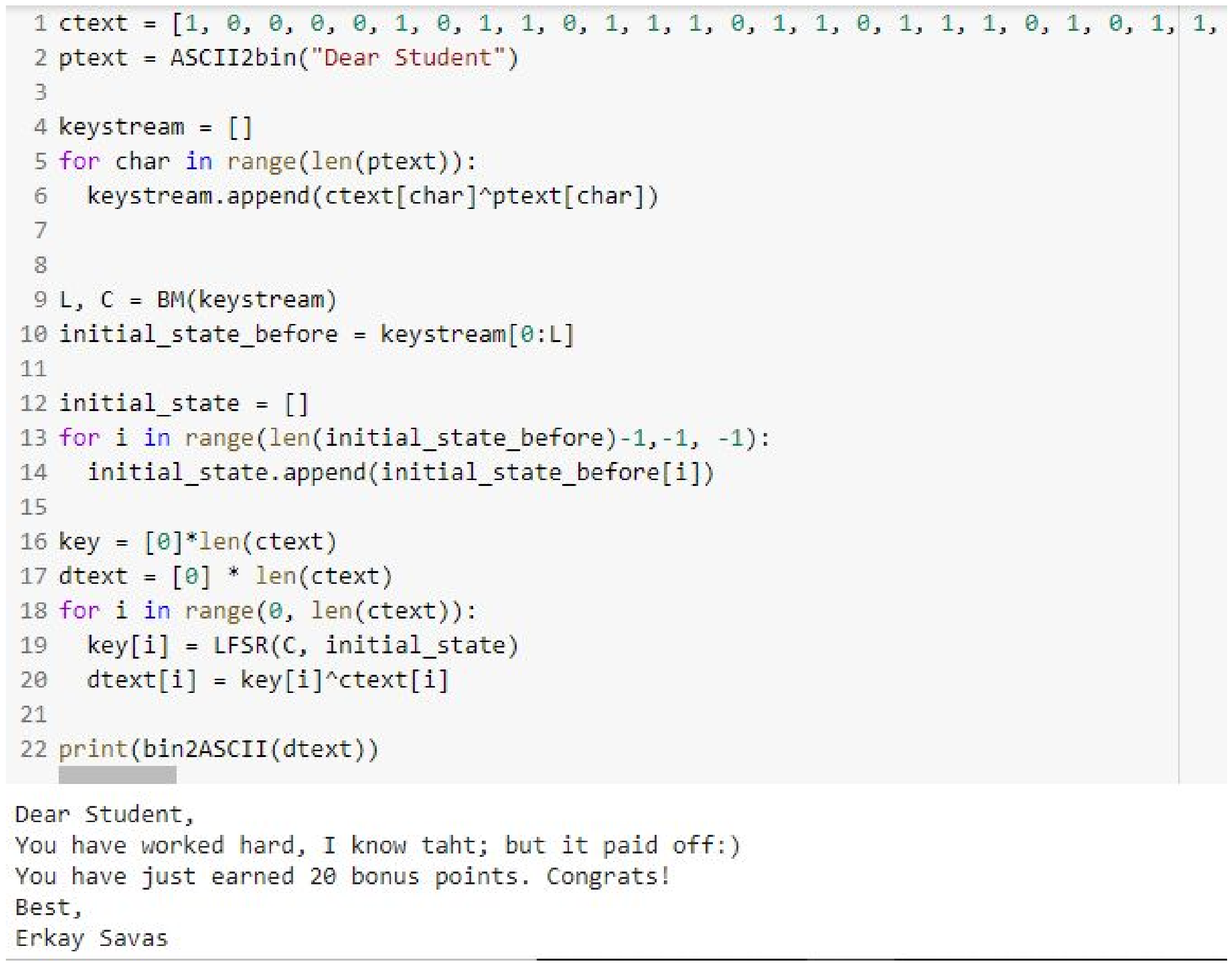
```
The linear complexity of x1 -> 37
x1 is predictable!
The linear complexity of x1 -> 37
x2 is predictable!
The linear complexity of x1 -> 37
x3 is predictable!
```

Bonus: (To access the respective Colab file, click here.)

I first encode the “Dear Student” part into a bit array, which consists of 84 bits. Then I xor’ed this encoded plaintext part with the first 84 bit of the ctext provided. What I obtain at the end is the 84-bit part of the keystream. Then, I passed this 84-bit keystream sequence to the BM algorithm hoping that it is long enough :) My aim was to get the shortest LFSR that can generate this sequence and use this as the connection polynomial. The value returned from the BM algorithm saying that the linear complexity is 27, which shows that the 84-bit key sequence is indeed enough. Since we found the connection polynomial, the only thing left is to find the initial state of the LFSR, after that, we can use the LFSR() function to compute the whole keystream and make a simple xor operation to get the decrypted encoded plaintext. I observed that the initial state is the first 27 bits of the key sequence because these are the bits that have not been affected by the feedback bits and any xors related to the connection polynomial and current state. They are just shifted and leave the flip-flops as they are without an effect of any logical operator. But the difference is that the first bit coming out of the LFSR is indeed stored in the last flip-flop of the LFSR. That’s why after I sliced the 27 bit part, I reversed it to get the initial state. Lastly, I passed the connection polynomial and the initial state to LFSR() function, to get the whole keystream and xor’ed the keystream with the ciphertext to get the decipher text, but encoded. Then I used the bin2ASCII() function and convert the bit array into a string, which is

“Dear Student,

You have worked hard, I know taht; but it paid off:)
You have just earned 20 bonus points. Congrats!
Best,
Erkay Savas”



```
ctext = [1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1,
ptext = ASCII2bin("Dear Student")

keystream = []
for char in range(len(ptext)):
  keystream.append(ctext[char]^ptext[char])


L, C = BM(keystream)
initial_state_before = keystream[0:L]

initial_state = []
for i in range(len(initial_state_before)-1,-1, -1):
  initial_state.append(initial_state_before[i])

key = [0]*len(ctext)
dtext = [0] * len(ctext)
for i in range(0, len(ctext)):
  key[i] = LFSR(C, initial_state)
  dtext[i] = key[i]^ctext[i]

print(bin2ASCII(dtext))
```

```
Dear Student,
You have worked hard, I know taht; but it paid off:)
You have just earned 20 bonus points. Congrats!
Best,
Erkay Savas
```