Muhammet Taha Çakmak 29183

**Q1)** My n = 101 and t = 25

a) Number of elements in the group is equal to n -1, **which is 100**. (All numbers from 1 to 100)

b) To find a generator I used the following python program (Q1.py). Since 101 is a prime, Euler totient function says there should be phi(100) generator for this group. Phi(100) = 40. Let's find all of them by following code:

```
1  residues = []
2  generators = []
3  mult = 1
4  for candidate in range(2,101):
5      for i in range(1,101):
6          mult = (mult * candidate) % 101                    # Multiplication in mod 101
7          if mult not in residues:
8              residues.append(mult)
9          else:
10             # If we find an element for second time for some i, that means we are in a cycle without completing all possible residues
11             # Break the loop and look for the next candidate
12             break
13     if len(residues) == 100:
14         #If the number of unique residues equal to order of the group then the candidate is a generator
15         generators.append(candidate)
16
17     #Before checking the next element, restart the setup by clearing mult and residues list
18     residues.clear()
19     mult = 1
20
21
22 print(sorted(generators))
23 print(len(generators))
```

All the generators can be listed as follows :

[2, 3, 7, 8, 11, 12, 15, 18, 26, 27, 28, 29, 34, 35, 38, 40, 42, 46, 48, 50, 51, 53, 55, 59, 61, 63, 66, 67, 72, 73, 74, 75, 83, 86, 89, 90, 93, 94, 98, 99]

**Since question wants only one of them, I selected 2 as the generator.**

c) Let's find a subgroup in the order of 25 and its possible generators. That's actually a very simple process since we have the previous piece of code. Only thing we need to modify is the if conditional at line 13. This time we will look for some residue class with t = 25 elements.

```
def subgroupGenerator(order):
    residues = []
    generators = []
    mult = 1
    for candidate in range(2,101):
        for i in range(1,101):
            mult = (mult * candidate) % 101                    # Multiplication in mod 101
            if mult not in residues:
                residues.append(mult)
            else:
                # If we find an element for second time for some i, that means we are in a cycle without completing all possible residues
                # Break the loop and look for the next candidate
                break
        if len(residues) == order:
            #If the number of unique residues equal to order of the group then the candidate is a generator
            generators.append(candidate)

        #Before checking the next element, restart the setup by clearing mult and residues list
        residues.clear()
        mult = 1


    print(sorted(generators))
    print(len(generators))


subgroupGenerator(25)
```

Above, I have a slightly modified version of the previous code, we pass the order of the subgroup as parameter and see the result on the terminal. For order t = 25 the following set of generators will give us subgroups with 25 elements:

[5, 16, 19, 24, 25, 31, 37, 52, 54, 56, 58, 68, 71, 78, 79, 80, 81, 88, 92, 97]

**Since the question asks for only one of them, I selected 97 as the generator of the subgroup with 25 elements.**

**Q2)** I used the following code (Q2.py to solve the problem

```
from client import *
from hw01_helper import *
from hw2_helper import *

def binaryLeftRight(base,power,mod):
    binaryInt = bin(power)
    x = 1
    for c in binaryInt[2:]:
        x = (x * x) % mod
        if c == '1':
            x = (x * base) % mod
    return x % mod

p= 16381263243811640233465195523887788805147169859580069932297961503570310535349859890001775447908274539030518348032638619392876202 3
q = 16799131140628182989327790751738092674329777043723781769808884372983741368040712103599372494242432804910022690306691941896357673

e,c = getQ2()
phiN = (p - 1) * (q - 1)

gcd,x,y = egcd(e,phiN)
m1 = binaryLeftRight(c,x,(p * q))
# Convert m1 to binary representation
binary_m1 = bin(m1)[2:]

print(binary_m1)

# checkQ2("I think I have 616 unread e-mails. Is that a lot?")
```

I assumed p and q as prime numbers. By knowing that, phi(n) = phi(p) * phi(q) = (p-1) * (q – 1) I calculated phiN. To decide multiplicative inverse of e in modulo phiN, I used egcd from hw1_helper to get Bezout coefficients. At the end, I calculated the modular exponent by implementing binary left to right algorithm. When I converted the result into binary, I got the following binary:

“01001001001000000111010001101000011010010110111001101011001000000100
1001001000000110100001100001011101100110010100100000001101100011000 1
00110110001000000111010101101110011100100110010101100001011001000010
000001100101001011010110110101100001011010010110110001110011001011 10

0010000001001001011100110010000001110100011010000110000101110100001000000110000100100000011011000110111101110100001111111”

I putted this into an online converter and got the result:

**“I think I have 616 unread e-mails. Is that a lot?”**

**Q3)** When I used Q3.py to decrypt messages I received error messages for first 2 messages but the third one decrypted without any error. As:

**‘An expert is a person who has made all the mistakes that can be made in a very narrow field.'**

Since the encoding being done by the same nonce and key, we can find the right nonce here. First 8 bytes are the nounce which are: **”\xccU\xe0N\x0e\xb6^1”**

Now, we can fix the nonces of first two accordingly,

To fix first one:

- Our nonce is 1U\xe0N\xb6
- We should add **^1** to end of this nonce
- We should delete **‘N’** from xeON
- We should remove ‘**1U’** part and add **”\xccU\xe0N”**
- After these changes we can decryt the message which is:
  **‘The first principle is that you must not fool yourself and you are the easiest person to fool.'**

To fix the second one:

- Nonce of the message two = **“'N\x0e\xb6^\xccU\xe0”**
- The places of them mixed we should put \xccU\xe0 to beginning.
  **“'\xccU\xe0\x0e\xb6^” ->** Now we need to a 1 at the end of the nonce.
- After these two changes we can decode successfully and receiving the message: **'Somewhere, something incredible is waiting to be known.'**

**Q4)** Q4 Solved by the following algorithm which is an implementation of the algorithm in the lecture slides. Resulting output is as follows:

Muhammet Taha Çakmak 29183

```python
def modularEquationSolver(a,b,n):
    gcd,x,_ = egcd(a,n)
    results = []
    if gcd == 1:
        # If a adnd n is co-prime solution is unique which could be found as follows:
        print("There is only 1 solution = ")
        print((b * x) % n)

    else:
        if b % gcd != 0:
            # If gcd is not 1 and does not divide the b there is no solution
            print("No valid answer!")
        else:
            # If gcd is not 1 and divides the b there are n solutions which could be found as follows
            a2 = a // gcd
            n2 = n // gcd
            b2 = b // gcd
            gcd2, x2, _ = egcd(a2,n2)
            result = (b2 * x2) % n2
            print(result)
            while result + n2 < n:
                result = (result + n2)
                print(result)
```

**Answer for part 1 :**

There is only 1 solution =

111563634314800439832213513866100835794512614711477009341 4826

-----------------------------------------------------------------------------------

**Answer for part 2 :**

No valid answer!

-----------------------------------------------------------------------------------

**Answer for part 3 :**

**There are 2 solutions:**

**1 - 18404510856369788270798305143120221499669411911430106143 85900**

**2 - 45730171685793211531469252635686277657592668520678380631 35011**

-----------------------------------------------------------------------------------

**Answer for part 4 :**

**There are 4 solutions:**

1. **12057457679543147764742525934468559057467205133259171935 5582**
2. **16920414540719870513978988950415999365363127960851309070 16143**
3. **32635083313485426251483725307385142824979535408376700946 76704**
4. **48349752086250981988988461664354286284595942855902092823 37265**

Muhammet Taha Çakmak 29183

**Q5)** To find the periods of the polynomials we can simulate corresponding LFSRs by using the given lfsr.py. I just manipulated the example usage part of the code according to the given polynomials **(Q5.py)**.

```
from lfsr import *
import random




import random
length = 256

print ("LFSR: **************")
L = 7                                           # Change the length according to the uncommented part of the code
C = [0]*(L+1)
S = [0]*L

C[0] = C[1] = C[3] = C[5] = C[7] = 1            # 1 + x + x^3 + x^5 + x^7
# C[0] = C[2] = C[5] = C[6] = 1                 # 1 + x^2 + x^5 + x^6
# C[0] = C[1] = C[3] = C[4] = C[5] = 1          # 1 + x + x^3 + x^4 + x^5
for i in range(0,L):                            # for random initial state
    S[i] = random.randint(0, 1)
print ("Initial state: ", S)

keystream = [0]*length
for i in range(0,length):
     keystream[i] = LFSR(C, S)

print ("First period: ", FindPeriod(keystream))
```

**Results are as follows:**

**Period of polynomial 1:** 127 (Maximum Period)

**Period of polynomial 2:** 21

**Period of polynomial 3:** 31 (Maximum Period)

**Q6)** To decide this algorithm we need to apply BM algorithm. Luckly, we have it in lfsr.py. When we run the following small script we obtain the following numbers. (Q6.py)

```python
from lfsr import *

x1 = [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0,
1, 0, 1, 0, 1, 1, 0]

x2 = [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0,
1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0]


x3 = [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1,
1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0,
0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1]

L1, _ = BM(x1)
L2, _ = BM(x2)
L3, _ = BM(x3)

print(L1,L2,L3)
print(len(x1),len(x2),len(x3))
```

**Smallest lfsr to produce the streams accordingly as follows: 36 43 31**

**Length of the streams accordingly: 75 80 90**

We know that expected length of lfsr must be n/2 + 2/9. When we imply the formula to lengths of stream we get the expected lengths as follows: 37.7 40.2 45.2. If these numbers are close to the actual length of the smallest lfsr, we can say that it is random, however if it is far small from it is not random.

- **We can deduce that stream 1 and 2 seemed to us random but third one is not. Since 37.7 and 40.2 are very close to 36 and 43 but 31 is far away from the 45.2**

**Q7)**

- First, we need to find the correct format of the instructor name. (After a few billion of hours and mistake I found the correct combination but here let's assume we hit the correct combination at the beginning). Let's say our message is from Erkay Savas and he signed the message by the string "Erkay Savas".
- Since his name contains 11 character the last 77 byte of the plaintext must contain his name. (11 x 7 =77)
- When we XOR the ctext's last 77 character by "Erkay Savas" in binary. We receive the last 77 character of the key stream.
- Now, we can use this part of the keystream to decide the LFSR. By using BM algorithm, we see that smallest LFSR corresponds to a $27^{th}$ degree polynomial which is:

$$x^{27} + x^{25} + x^{24} + x + 1$$

- Now, we know that every 27 bit is used to produce another bit in keystream. However, At this point we need to think reversely.
- Let's say we have 27 bit part of the stream (Let's call it calculation block, CB). We can set the following algebra:
  - The $27^{th}$ bit (let's call it ans) is a result of the combination of first 26 bits with an unknown bit (let's say c)
  - Our LFSR, XOR the first $24^{th}$ $25^{th}$ and $27^{th}$ bits of it to produce the ans
  - C corresponds to $27^{th}$ bit, so we can set the following equation:

$$\mathbf{CB[0] \oplus CB[23] \oplus CB[24] \oplus C = ans}$$

  - At this point we can calculate left hand side's first 3 XOR. Let's call this part as the mid (short hand for middle part)
  - The calculation becomes:

$$\mathbf{mid \oplus C = ans}$$

  - Since we know the mid and ans values all the time we can easly produce the C at every round. Which is basically:

$$\mathbf{C = mid \oplus ans}$$

  - We found the previous byte. We have to repeat this until the keystream's length is equal to the ctext.

- After obtained the full keystream, we can xor it by ctext and can obtain the answer, which is:
  **Dear Student,**
  **Outstanding job on tackling this challenging problem!**
  **Congratulations!**
  **Best, Erkay Savas**

**The implementation can be found in Q7.py**

```python
def findLeftMost(arr):
    arr.insert(0,'c')
    ans = arr[27]
    calcBlock = arr[:27]
    calcBlock = calcBlock[::-1]
    mid = calcBlock[0] ^ calcBlock[23] ^ calcBlock[24]
    arr[0] = mid ^ ans


binary = ASCII2bin("Erkay Savas")
lastPart = ctext[-len(binary):]

keyStream = []

for i in range(0,77):
    #XOR the known plaintext and the final part to obtain the last part of the key stream
    keyStream.append(binary[i] ^ lastPart[i])

L,C = BM(keyStream)

while len(keyStream) != len(ctext):
    #Untill the key space size becomes equal to length of ctext we will produce a new byte according to the algorithm
    findLeftMost(keyStream)

resultText = []
for i in range(0,len(keyStream)):
    #After key stream is completed. We will use is to decrypt by doing XOR
    resultText.append( keyStream[i] ^ctext [i])


print(bin2ASCII(resultText))
```