Q1)

We know our message m is 224 bit number. Thus, if we check the number of bits of maximum valued number with power e, we can see that it is much less than modulo n.

```
Maximum number with 224 bits:  26959946667150639794667015087019630673637144422540572481103610249215
Maximum number of bits when the 17th power is taken:  3808
n bit length:  4096
c bit length:  3809
e bit length:  5
```

3808 bits << 4096 bits.

Therefore, we can say that **c = m^e or c^(1/e) = m** directly from RSA equations.

Firstly, I tried to use second equation which is c^(1/e) = m to find message. However, I could not take the power of cipher. I make some research and find different algorithms which implement this power problem. Then, I realized our maximum 224 bit number's 17^{th} power's length is smaller than c which shows an **inconsistency** with the problem.

The maximum number in 224 bits = 2**224 - 1

However, even its 17^{th} power is not close to cipher bit length. Therefore, I decided use pow(2, 224) check whether it is given correct bit length or not.

Surprisingly, pow(2, 224) is giving the same cipher when it is encoded.

```
upper_bound = pow(2,224)
lower_bound = pow(2,223)

for i in range(upper_bound, lower_bound, -1):
    if pow(i, e,n) == c:
        print("We found it!")
        break

print("Message:", i)
print("Message number of bits: ", i.bit_length())
```

Output:

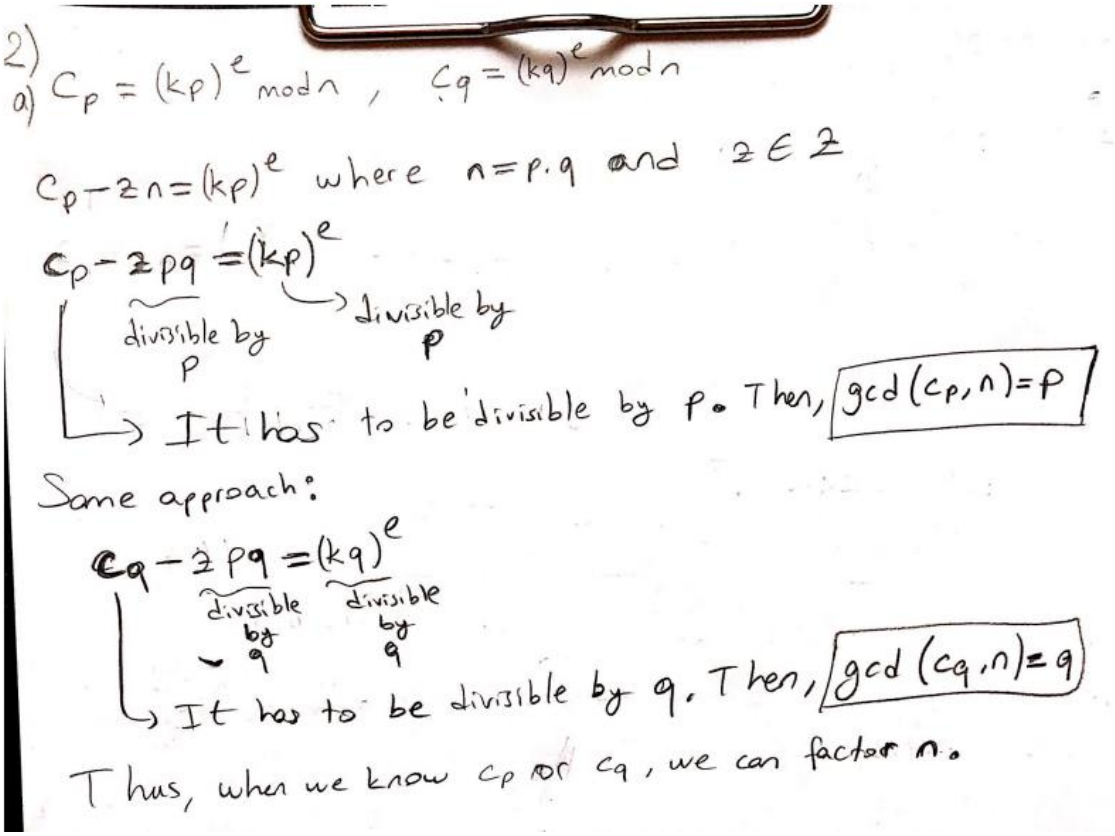
```
We found it!
Message: 26959946667150639794667015087019630673637144422540572481103610249216
Message number of bits:  225
```

Thus,

Message:
26959946667150639794667015087019630673637144422540572481103610249216

Q2)

Part A:



Part B:

Since we showed if we know c_p or c_q values, we can factorize n to find p and q values. I followed below simple algorithm steps:

```
n = 180169674772689055638858699242372583186946943366168296635599516671542541535056690723662381774732634405580004929915363725372132
e = 65537

cp = 986782016050004825543163892795029444288885745280201735645350541631009180800660514585614546374307414257312119957188330368713468
cq = 140485344653232305453253541357793353734644094796221306042212463008486867889142210276208096483328329570105315597285881500662659

cm = 531014076826388911268676698266348523272414732277954807833668971962310073834447607178348718953370311072128032122169756314494284

p = gcd(cp, n)
q = gcd(cq, n)

phi_n = (p-1)*(q-1)
d = modinv(e, phi_n)
print("p:", p)
print("q:", q)
print("d:", d)


m_ = pow(cm,d,n)
print("\n")
print(m_.to_bytes((m_.bit_length() + 7) // 8, 'big'))

print("\nReversed?")
print(m_.to_bytes((m_.bit_length() + 7) // 8, 'big')[::-1])
```

I found following outputs:

```
p:
119930273251555701891443865610384158624238297876761336393716178246299551162786800677714755334278289178015102897504042232297016291471934789813961048339187695237790282000408328937732148978236744988829455805317131463975046055458079676401458858459875408623620213779402719981577151373623133961315903667304164470941
q:
150228686959447034457865122762757898898341135174753580226519729936414879885646803625058402873359317276522400897007595862841592765556143833685472829649537615563178755409770180422620394773942014602865741340086949030611073332977452549653998605016494779410550789215602534590415959938476493315233644641381526246916d:
627048859711348500857031611306479736623750374786598818676684830822468808439981698691733717731949380984310287691905264386836897927116543352271911499622098663002165458852104186898919520914468110102469334502418230206079465976560764687131158163736739224960949981759580151298265064440774195718641469007127046775968880323737117620526206129010986014116866479180142393672486971569452480084307311742499786399572315234690340004568837417249334320300768391623313332610885384990892440724823140286308251307193933872489731790106741503140826680310693415258261819029591756347428159623897338604435136612195714199254595413051363862367 3


b".rotcaf namuh eht tnuocca otni nekat ton evah llits uoy eveileb etiuq t'nac I tub ,smis lautca gnihctaw era ew won dna ,snoitalumis retupmoc eht tuoba draeh lla ev'eW ?won suoires teg ew naC"

Reversed?
b"Can we get serious now? We've all heard about the computer simulations, and now we are watching actual sims, but I can't quite believe you still have not taken into account the human factor."
```

Answer: “Can we get serious now? We've all heard about the computer simulations, and now we are watching actual sims, but I can't quite believe you still have not taken into account the human factor.”

Q3)

We know that we can use correlation attack in Geffe generator where LFSR1 and LFSR3 correlated with the output sequence.

Firstly, I found all possible states for the LFSR1 after defining my connection polynomials as following way: (Code is taken from stackoverflow with simple modifications)

```
22  def find_all_states(n, i):
23      all_states = list(map(list, itertools.product([0, 1], repeat=n-1)))
24      return all_states
25
```

Then, I count the number of coincidences with output sequence (z) for each possible initial states with the following algorithm:

```
26  def coincidence_count(n, temp,C,z):
27      count_list = []
28      all_states = find_all_states(n,temp)
29      for k in all_states:
30          key = []
31          count = 0
32          # print("State:", k)
33          # print("Polynom:", C)
34          for i in range(len(z)):
35              key.append(LFSR(C,k))
36              # print("State: ", k)
37              # print("Key:", key)
38          for m in range(len(z)):
39              if key[m] == z[m]:
40                  count += 1
41
42          count_list.append(count)
43      return count_list
```

The initial state which gives the most coincidence count is more likely to be the initial state of the LFSR1. I found following result which is interesting:

```
Length of z:  90
Number of possible states for LFSR1 = 128
Maximum coincidences for LFSR1:  90
Corresponding state:  [0, 1, 1, 1, 1, 1, 0]
```

We can say that our output sequence can be determined by just looking to LFSR1 with above initial state.

Then, I did the same operation for the LFSR3:

```
Number of possible states for LFSR3 = 2048
Maximum coincidences for LFSR3:  62
Corresponding state:  [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
```

Then, I combined these two LFSR1 and LFSR3 with these initial states to find LFSR2. To do this, I used my Geffe generator to match with output sequence (z).

```
45  def geffe(s1, s2, s3):
46      z_test = (s1 and s2) ^ (s2 and s3) ^ s3
47      return z_test
```

```
94
95  for k in state_list2:
96      final = k.copy()
97      key2 = []
98      for i in range(len(z)):
99          key2.append(LFSR(C2,k))
100     z2 = []
101     for m in range(len(z)):
102         z2.append(geffe(key1[m], key2[m], key3[m]))
103     if z2 == z:
104         print("WOW", final)
```

Following output for LFSR2 initial state:

```
Number of possible states for LFSR2 = 8192
WOW [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

Then, I wanted to prove these initial states are giving our output sequence (z) with the following algorithm:

```
112 result = []
113 for i in range(len(z)):
114     result.append(geffe(key1[i], key2[i], key3[i]))
115     if result == z:
116         print("Proved!")
117
118 print("LFSR1 initial state: ", S1)
119 print("LFSR2 initial state: ", S2)
120 print("LFSR3 initial state: ", S3)
```

```
Proved!
LFSR1 initial state:  [1, 1, 1, 0, 1, 1, 0]
LFSR2 initial state:  [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
LFSR3 initial state:  [0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0]
```

Thus,

LFSR1 initial state: [1, 1, 1, 0, 1, 1, 0]

LFSR2 initial state: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

LFSR3 initial state: [0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0]

Q4)

Simplest thing to do is finding factors of n. Since n is not too large, I used simple algorithm to find factors of n. However, if the n was larger, I would use stronger methods such as Quadratic Sieve method.

```
for i in range(1,n,2):
    if n%i == 0:
        print("Found a factor!", i)
        if isprime(i) == True:
            if isprime(int(n/i)) == True:
                    p = i
                    q = int(n/i)
            break

print("Our p: ", p)
print("Our q: ", q)

phi_n = (p-1) * (q-1)
d = modinv(e, phi_n)

m_ = pow(c, d, n)
print(m_.to_bytes((m_.bit_length() + 7) // 8, 'big').decode()[::-1])
```

After finding my p and q values, I did normal RSA decryption and found the following message.

```
Our p:  13882590739
Our q:  17110666501
Corresponding message: Bingo
```

Message: Bingo

Q5)

For this question, I just simply used the BitVector library functions to find results:

```
39  ##SOLUTION
40  a_vec  = BitVector(bitstring = a)
41  b_vec  = BitVector(bitstring = b)
42  poly = BitVector(bitstring = '100011011')
43  check_mult(a_vec.gf_multiply_modular(b_vec, poly, 8))
44  check_inv(a_vec.gf_MI(poly, 8))
45
```

I got the following outputs:

```
{'a': '01010001', 'b': '00111010'}
Congrats
Congrats
```

Q6)

I tried to follow rainbow attack steps which we have seen in the lecture. Firstly, I checked whether Case 1 applies or not where we only make one reduction with given digest.

After Case 1 failure, I started to repeatedly hash and reduce to see any correspondence in rainbow table for the last column. When I found the correspondence, I recorded the initial block of the password. Then, I started to make chain from this initial state to find same digest value with our digest to find our password:

NORMALLY, we just have to regenerate as many blocks as we created. HOWEVER, It does not work for this case. I could not find any reason for that. Therefore, my code just running from initial state to until finding the same digest value (It does not stop until finding ☹). I know this is not correct method, but I would rather to show some correct passwords than finding all wrong passwords with our lecture methods.

Normal Algorithm (did not work): After finding Case 2 correspondence in the rainbow table's second column. Start from its initial block and create as many we produced earlier – 1 to find corresponding password.

Thus, I did not create as many as produced. I let it run all the way:

```
for org_digest in digestt:
    seq = 0
    temp = Reduction(org_digest%pwd_space, Alphabet, pwd_len, seq)
    for i in range(len(Rainbow_Table)):
        if temp == Rainbow_Table[i][1]:
            print("YES CASE 1 APPLY")
            initial_pwd = Rainbow_Table[i][0]
            seq2 = 0
            print("Corresponding Initial Block: ", initial_pwd)
            for k in range(t-1):
                hash = SHA3_256.new(initial_pwd.encode('utf-8')) # hash it
                temp_digest = int.from_bytes(hash.digest(), byteorder='big') # convert the hash into an integer
                initial_pwd = Reduction(temp_digest%pwd_space, Alphabet, pwd_len, seq2)
                seq2 += 1
            hash = SHA3_256.new(initial_pwd.encode('utf-8')) # hash it
            temp_digest = int.from_bytes(hash.digest(), byteorder='big') # convert the hash into an integer
            print("PROVING!")
            if org_digest == temp_digest:
                print("Corresponding Password: ", initial_pwd)
    seq = 0
    pwd = Reduction(org_digest%pwd_space, Alphabet, pwd_len, seq)
    myflag = True
    while myflag:
        myflag2 = True
        hash = SHA3_256.new(pwd.encode('utf-8'))
        digest = int.from_bytes(hash.digest(), byteorder='big')
        seq += 1
        pwd = Reduction(digest%pwd_space, Alphabet, pwd_len, seq)
        for i in range(len(Rainbow_Table)):
            if pwd == Rainbow_Table[i][1]:
                initial_pwd = Rainbow_Table[i][0]
                print("YES CASE 2 APPLY")
                print("Corresponsing Initial block: ",initial_pwd)
                myflag = False
                for seq in range(pwd_space):
                    hash = SHA3_256.new(initial_pwd.encode('utf-8'))
                    digest = int.from_bytes(hash.digest(), byteorder='big')
                    if digest == org_digest:
                        print("Corresponding Password: ", initial_pwd)
                        break
                    initial_pwd = Reduction(digest%pwd_space, Alphabet, pwd_len, seq)
                break
```

However, this method is taking too long to find passwords even though I used two different computers. I found some of the passwords as following, the rest is not available because of my limited time.

OUTPUTS:

Digest 0:

```
YES CASE 2 APPLY
Corresponsing Initial block:  GNMCYA
Corresponding Password:  WZIOJF
```

Digest 1:

One of my pc is stucked here 😃. Actually, it is not stucked, it is taking too long time!

Digest 2:

```
In [3]: runcell(0, '/home/ozgur/Downloads/temp.py')
YES CASE 2 APPLY
Corresponsing Initial block:  MOAAML
Corresponding Password:  PGNXND
```

Digest 3:

```
In [4]: runcell(0, '/home/ozgur/Downloads/temp.py')
YES CASE 2 APPLY
Corresponsing Initial block:  ZYBDRM
Corresponding Password:  FHJOLG
```

Digest 4:

```
In [5]: runcell(0, '/home/ozgur/Downloads/temp.py')
YES CASE 2 APPLY
Corresponsing Initial block:  JKJAFR
Corresponding Password:  LKHWUE
```

Digest 5:

```
In [1]: runcell(0, '/home/ozgur/Downloads/temp.py')
YES CASE 2 APPLY
Corresponsing Initial block:  KPVBXT
Corresponding Password:  THSBIG
```

Digest 6:

```
In [2]: runcell(0, '/home/ozgur/Downloads/temp.py')
YES CASE 2 APPLY
Corresponsing Initial block:  CQSRMQ
Corresponding Password:  HHNKTB
```

Digest 7:

The other pc is also stucked in here 😃.

CODES

Question 1)

```
import math
from sympy import *
import math
import random
import warnings



n = 548500902145867973227715390953418318013359802187256802338524882275579547934760125329409722630825850640481397494578199
c = 210002027846652282364048950525727883422548762581439698149825328219549364036763241490186660161103079324168573047741947
e = 17


max224bit = pow(2,224)-1
print("Maximum number with 224 bits: ", max224bit)
print("Maximum number of bits when the 17th power is taken: ", pow(max224bit,e).bit_length())
print("n bit length: ",n.bit_length())
print("c bit length: ",c.bit_length())
print("e bit length: ",e.bit_length())


upper_bound = pow(2,224)
lower_bound = pow(2,223)
for i in range(upper_bound, lower_bound, -1):
    if pow(i, e,n) == c:
        print("We found it!")
        break

print("Message:", i)
print("Message number of bits: ", i.bit_length())
```

Question 2)

```
    Unless b==0, the result will have the same sign as b (so that when
    b is divided by it, the result comes out positive).
    """
    while b:
        a, b = b, a%b
    return a

def egcd(a, b):
    x,y, u,v = 0,1, 1,0
    while a != 0:
        q, r = b//a, b%a
        m, n = x-u*q, y-v*q
        b,a, x,y, u,v = a,r, u,v, m,n
    gcd = b
    return gcd, x, y

def modinv(a, m):
    if a < 0:
        a = m+a
    gcd, x, y = egcd(a, m)
    if gcd != 1:
        return None  # modular inverse does not exist
    else:
        return x % m

n = 180169674772689055638858699242372583186946943366168296635599516671542541535056
e = 65537

cp = 986782016050004825543163892795029444288885745280201735645350541631009180800660
cq = 140485344653232305453253541357793353734644094796221306042212463008486867889142
cm = 531014076826388911268676698266348523272414732277954807833668971962310073834447

p = gcd(cp, n)
q = gcd(cq, n)

phi_n = (p-1)*(q-1)
d = modinv(e, phi_n)
print("p:", p)
print("q:", q)
print("d:", d)


m_ = pow(cm,d,n)
print("\n")
print(m_.to_bytes((m_.bit_length() + 7) // 8, 'big'))

print("\nReversed?")
print(m_.to_bytes((m_.bit_length() + 7) // 8, 'big')[::-1])
```

Question 3)

```python
# -*- coding: utf-8 -*-
"""
Created on Thu Nov 25 20:03:13 2021

@author: user
"""
import copy
import itertools

def LFSR(C, S):
    L = len(S)
    fb = 0
    out = S[L-1]
    for i in range(0,L):
        fb = fb^(S[i]&C[i+1])
    for i in range(L-1,0,-1):
        S[i] = S[i-1]

    S[0] = fb
    return out

def find_all_states(n, i):
    all_states = list(map(list, itertools.product([0, 1], repeat=n-1)))
    return all_states

def coincidence_count(n, temp,C,z):
    count_list = []
    all_states = find_all_states(n,temp)
    for k in all_states:
        key = []
        count = 0
        # print("State:", k)
        # print("Polynom:", C)
        for i in range(len(z)):
            key.append(LFSR(C,k))
            # print("State: ", k)
            # print("Key:", key)
        for m in range(len(z)):
            if key[m] == z[m]:
                count += 1

        count_list.append(count)
    return count_list

def geffe(s1, s2, s3):
    z_test = (s1 and s2) ^ (s2 and s3) ^ s3
    return z_test

z = [0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1,

```

```
C1 = [0] * 8
for i in [0,5,7]:
    C1[i] = 1

C2 = [0] * 14
for i in [0,3,7, 13]:
    C2[i] = 1

C3 = [0] * 12
for i in [0,2,11]:
    C3[i] = 1


print("Length of z: ",len(z))
state_list1 = find_all_states(8,1)
print("Number of possible states for LFSR{} = {}".format(1,len(state_list1)))
count_list1 = coincidence_count(8, 1,C1,z)

print("Maximum coincidences for LFSR1: ", max(count_list1))
print("Corresponding state: ", state_list1[count_list1.index(max(count_list1))])

state_list3 = find_all_states(12,3)
print("Number of possible states for LFSR{} = {}".format(3,len(state_list3)))
count_list3 = coincidence_count(12, 3,C3,z)

print("Maximum coincidences for LFSR3: ", max(count_list3))
print("Corresponding state: ", state_list3[count_list3.index(max(count_list3))])

state_list2 = find_all_states(14,2)
print("Number of possible states for LFSR{} = {}".format(2,len(state_list2)))
count_list2 = coincidence_count(14, 2,C2,z)

S1 = [0, 1, 1, 1, 1, 1, 0]
key1 = []
for i in range(len(z)):
    key1.append(LFSR(C1,S1))

S3 = [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
key3 = []
for i in range(len(z)):
    key3.append(LFSR(C3,S3))
```

```
S2 = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
key2 = []
for i in range(len(z)):
    key2.append(LFSR(C2,S2))
result = []
for i in range(len(z)):
    result.append(geffe(key1[i], key2[i], key3[i]))
    if result == z:
        print("Proved!")
print("LFSR1 initial state: ", S1)
print("LFSR2 initial state: ", S2)
print("LFSR3 initial state: ", S3)
```

Question 4)

```
n = 237540380304900134239
c = 226131284405640469226
e = pow(2,16) + 1


for i in range(1,n,2):
    if n%i == 0:
        print("Found a factor!", i)
        if isprime(i) == True:
            if isprime(int(n/i)) == True:
                        p = i
                        q = int(n/i)
                break

print("Our p: ", p)
print("Our q: ", q)

phi_n = (p-1) * (q-1)
d = modinv(e, phi_n)

m_ = pow(c, d, n)
print("Corresponding message:", m_.to_bytes((m_.bit_length() + 7) // 8, 'big').decode()[::-1])
```

Question 5)

```
import random
import requests
from BitVector import *

API_URL = 'http://cryptlygos.pythonanywhere.com'
my_id = 25331

def get_poly():
  endpoint = '{}/{}/{}'.format(API_URL, "poly", my_id )
  response = requests.get(endpoint)
  a = 0
  b = 0
  if response.ok:
    res = response.json()
    print(res)
    return res['a'], res['b']
  else:
    print(response.json())

def check_mult(c):
  #check result of part a
  endpoint = '{}/{}/{}/{}'.format(API_URL, "mult", my_id, c)
  response = requests.put(endpoint)
  print(response.json())

def check_inv(a_inv):
  #check result of part b
  response = requests.put('{}/{}/{}/{}'.format(API_URL, "inv", my_id, a_inv))
  print(response.json())

a, b = get_poly()
##SOLUTION
a_vec  = BitVector(bitstring = a)
b_vec  = BitVector(bitstring = b)
poly = BitVector(bitstring = '100011011')
check_mult(a_vec.gf_multiply_modular(b_vec, poly, 8))
check_inv(a_vec.gf_MI(poly, 8))
```

Question 6)

```
###
for org_digest in digestt:
    seq = 0
    temp = Reduction(org_digest%pwd_space, Alphabet, pwd_len, seq)
    for i in range(len(Rainbow_Table)):
        if temp == Rainbow_Table[i][1]:
            print("YES CASE 1 APPLY")
            initial_pwd = Rainbow_Table[i][0]
            seq2 = 0
            print("Corresponding Initial Block: ", initial_pwd)
            for k in range(t-1):
                hash = SHA3_256.new(initial_pwd.encode('utf-8')) # hash it
                temp_digest = int.from_bytes(hash.digest(), byteorder='big') # convert the hash into an integer
                initial_pwd = Reduction(temp_digest%pwd_space, Alphabet, pwd_len, seq2)
                seq2 += 1
            hash = SHA3_256.new(initial_pwd.encode('utf-8')) # hash it
            temp_digest = int.from_bytes(hash.digest(), byteorder='big') # convert the hash into an integer
            print("PROVING!")
            if org_digest == temp_digest:
                print("Corresponding Password: ", initial_pwd)

    seq = 0
    pwd = Reduction(org_digest%pwd_space, Alphabet, pwd_len, seq)
    myflag = True
    while myflag:
        myflag2 = True
        hash = SHA3_256.new(pwd.encode('utf-8'))
        digest = int.from_bytes(hash.digest(), byteorder='big')
        seq += 1
        pwd = Reduction(digest%pwd_space, Alphabet, pwd_len, seq)
        for i in range(len(Rainbow_Table)):
            if pwd == Rainbow_Table[i][1]:
                initial_pwd = Rainbow_Table[i][0]
                print("YES CASE 2 APPLY")
                print("Corresponsing Initial block: ",initial_pwd)
                myflag = False
                for seq in range(pwd_space):
                    hash = SHA3_256.new(initial_pwd.encode('utf-8'))
                    digest = int.from_bytes(hash.digest(), byteorder='big')
                    if digest == org_digest:
                        print("Corresponding Password: ", initial_pwd)
                        break
                    initial_pwd = Reduction(digest%pwd_space, Alphabet, pwd_len, seq)
                break
```