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. ***Yet, if you will test my code using Google Colab, then you should upload the respective helper file for that question since Google Colab resets the uploaded files when the runtime is reset.*** *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: (You can access the respective Google Colab file through this link)

Since we can send as many ciphertexts as we want and get the corresponding plaintext messages for them, we can make a known ciphertext attack. We know that for the specific ciphertext c that we get from the user, the plaintext is m. In the cryptographic protocol, we will send the ciphertext to the other party. If we modify an already known ciphertext with a constant random value, then we get the information of the new ciphertext that we get, say c_, and also the m_ that came back from the server.
fujıeo

I choose a random value r, say 22, and raise it to the power of e because of the fact that we raise the message to the power of e during the encryption process. So instead of regular encryption that we send the message through raising it to the power of e in mod N, I also raised that random value to the power of e. Then, this is the modified ciphertext that we know the relationship with the original one. So, I sent that c_ to the server and got the respective m_. For the decryption, we must raise the ciphertext to the power of d. So what the server-side did is the following operation: $(r^e * m^e)^d \ mod\ N$. We know that $e * d \equiv 1\ mod\ \Phi(n).$ Therefore, since e*d will be in the exponent, $(r^{ed} * m^{ed}) \ \ mod\ N = r * m\ mod N$. So the m_ that we received from the server is equal to $r * m\ mod N$. All we have to do is divide the m_ by r to get the original message.

```
###### Query Oracle it will return corresponding plaintext
endpoint = '{}/{}/{}/{}'.format(API_URL, "RSA_Oracle_query", my_id, c_)
response = requests.get(endpoint)
if response.ok: m_ = (response.json()['m_'])
else: print(response)

inv_r = modinv(r, N)
message = m_ * inv_r
message = message % N
print("message: ",message)

byte_array = message.to_bytes(message.bit_length() // 8 + 1, byteorder= "big")
messagetext = byte_array.decode("utf-8")
print(messagetext)

res = 86616
###Send your answer to the server.
endpoint = '{}/{}/{}/{}'.format(API_URL, "RSA_Oracle_checker", my_id, res)
response = requests.put(endpoint)
print(response.json())

message:  609582572245086581955173486556226059889437853770629680858438291721089382181432413886171267314993869702443318
Bravo: you find it. Your secret code is 86616
Nope, Try again
```

Since the message that I find out is "`Bravo: you find it. Your secret code is 86616`", I tried to send the secret code 86616 to the server. Therefore, I got the "Nope, try again" response from the server in my first trial.

```
#-----------solution------------------
r = 22 #this is a random number, but does not really matter what it is. I choose it here :)
r_to_pow_e = pow(r, e, N)
willsent = r_to_pow_e * c
c_ = willsent % N

###### Query Oracle it will return corresponding plaintext
endpoint = '{}/{}/{}/{}'.format(API_URL, "RSA_Oracle_query", my_id, c_)
response = requests.get(endpoint)
if response.ok: m_ = (response.json()['m_'])
else: print(response)

inv_r = modinv(r, N)
message = m_ * inv_r
message = message % N
print("message: ",message)

byte_array = message.to_bytes(message.bit_length() // 8 + 1, byteorder= "big")
messagetext = byte_array.decode("utf-8")
print(messagetext)

res = byte_array
###Send your answer to the server.
endpoint = '{}/{}/{}/{}'.format(API_URL, "RSA_Oracle_checker", my_id, res)
response = requests.put(endpoint)
print(response.json())
```

When I sent the byte array, I got the congrats answer :)

Question 2: (You can access the respective Google Colab file through this link)

Since in the OAEP RSA, there is a randomizer R; we get different ciphertexts even we encrypt the same plaintext. But here, the R-value is much smaller than it supposed to be, this means that we can make an exhaustive search on R. Also, the pin is a 4 digit integer that we can make an exhaustive search on. This means that we can create 2 loops in a nested fashion, one would be iterate over the possible pins and the other one would be iterate over the possible R values. For each R and pin pair, I called the RSA_OAEP_Enc function to check if the current encryption is equal to the ciphertext that we get from the server. If this is the case, then we found the R-value and also the pin, which is pin = 6436 and R = 236 for my case.

```
18
19 ########
20 k0 = 8
21
22 for pin in range(0, 10000):
23   for r in range(2**(k0-1), 2**k0):
24     currEncryption = RSA_OAEP_Enc(pin, e, N, r)
25     if currEncryption == c:
26       print("pin is", pin)
27       print("r is", r)
28       PIN_ = pin
29       break
30
31 #########
32 # Client sends PIN_ to server
33 endpoint = '{}/{}/{}/{}'.format(API_URL, "RSA_OAEP", my_id, PIN_)
34 response = requests.put(endpoint)
35 print(response.json())
```

```
pin is 6436
r is 236
Congrats
```

Question 3: (You can access the respective Google Colab file through this link)

Since the numbers are small, we can directly compute the session key k using the equality of $g^k mod\ p\ ? = r$ where r is given in the question. We also know that $t = h^k * m\ mod\ p$. So $m = t * (h^k)^{-1}\ mod\ p$. When we convert the encoded message of type integer to bytes and then decode it to the string, the message is:
"`Why is Monday so far from Friday, and Friday so close to Monday?`"

```
import math
from ElGamal import egcd, modinv

#------Given in the question
q = 2022967828283532245360658374440322019407596246123901055008730902181 1
p = 1251504390939480375345041102264985427372153725101110774882681116845968062835139115448704132059500673623933219249223694396652305374441
g = 2256483143741433163413007675067934542893022968337437312283381964942344365449719628255630752397325376452002398784394008507857025386941
h = 1265126138933377994348793193477342224736956600354964713945582205290651827674605003741290400826146165752452701594226002213036650208861
r = 3813677439444837990381281624769265484071989883494833765363155214071727573627590213038823018054653614040833306533736593789523636716081
t = 1019203324011337764086016919505431598172751432732900879044413072910705693004729954775515077563623725236797903281542668544832920731811

for k in range(q):
  possibleR = pow(g,k,p)
  if possibleR == r:
    print("k: ", k)
    break

h_to_power_k = pow(h, k, p)
inv_h_to_k = modinv(h_to_power_k, p)
messageint = (t * inv_h_to_k) % p

byte_array = messageint.to_bytes(messageint.bit_length() // 8 + 1, byteorder= "big")
messagetext = byte_array.decode("utf-8")
print(messagetext)
```

```
k:  31659
Why is Monday so far from Friday, and Friday so close to Monday?
```

Question 4: (You can access the respective Google Colab file through this link)

When I look at the values, I realized that the R values are the same! This means that the session key is the same. If the session key same, then we can use the following formula to find the secret key:

$$a = (s_i h_j - s_j h_i)(r(s_j - s_i))^{-1} \bmod q$$

In the implementation of the DSA, SHAKE128 is used for hashing. Therefore, I used SHAKE128 in order to hash the messages.

In the end, I found out that the `secret key is:`
`168874198460519327134644531443752111733505626315532547031556139226 71`

```python
from DSA import modinv, egcd
from Crypto.Hash import SHA3_256
from Crypto.Hash import SHAKE128

#-----given in the question--------
s1 = 241287483677536823019495765940525844957957956834050121761817762978O
s2 = 343379365128270720539597367095485301128970178274104846189598795161
g = 138430796393513409202731847145908844004328470930587709707751330796280153434746389859495142244692313165O
p = 218441021121222374840584849902232225278169817028282791714981430365827162714854740283805426968621937208S
r = 616457299314826827854431524615879496606124345660308142738979269878O
q = 184628707979587343584605403158023119637449999545068079815084986350 91

message1_byte = b"He who laugh last didn't get the joke"
message2_byte = b"Ask me no questions, and I'll tell you no lies"

shake1 = SHAKE128.new(message1_byte)
h1 = int.from_bytes(shake1.read(q.bit_length()//8), byteorder='big')

shake2 = SHAKE128.new(message2_byte)
h2 = int.from_bytes(shake2.read(q.bit_length()//8), byteorder='big')

secondPart = r * (s1 - s2)
secondPart_inv = modinv(secondPart, q)
firstPart = (s2*h1 - s1*h2) % q
a = (firstPart * secondPart_inv) % q
print(" secret key is:", a)
```

```
secret key is: 168874198460519327134644531443752111733505626315532547031556139226 71
```

Bonus: (You can access the respective Google Colab file through this link)

Since the hint says that the encryptor ran out of random numbers for the second message and r values are not the same, then the only option is that the session keys are not the same but related to each other with a relation $k_2 = x * k_1$.

Since we don't know the x, I created a for loop that iterates over possible x values, and for each x value I computed the respective private key a which can be calculated through the following formula:

$$a = (s_i h_j - s_j h_i x)(s_j r_i x - s_i r_j)^{-1} \bmod q$$

For each possible private key, I check if $g^a mod\, p = \beta$. If this equality holds, then we find the private key.

The private key is
`38468022319344408234287699540778097655787016963246135508538566407 2`

X is `127.`

```
shake1 = SHAKE128.new(m1_byte)
h1 = int.from_bytes(shake1.read(q.bit_length()//8), byteorder='big')

shake2 = SHAKE128.new(m2_byte)
h2 = int.from_bytes(shake2.read(q.bit_length()//8), byteorder='big')

for x in range(200): #j=2, i=1
  secondPart = ((s2*r1*x)%q) - ((s1*r2)%q)
  secondInv = modinv(secondPart, q)
  firstPart = s1*h2 - s2*h1*x
  a = (firstPart * secondInv) % q
  if beta == pow(g, a, p):
    print("x:", x)
    print("a:", a)
    break


```

```
x: 127
a: 38468022319344408234287699540778097655787016963246135508538566407 2
```