Below, you can find the 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, with an exception that the source code for Question 3 is named as affine_client.py. 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. Because of the fact that the keyspace of the shift cipher is only 26 which is the length of the English alphabet, it is possible to make an exhaustive search and shift the characters one by one depending on the predetermined shift amount while the shift amount would be increased at each iteration.

We can do it by pen and paper easily due to the smallness of the keyspace and the ciphertext “NYVVC” is really short. But instead, we can use the Python code that is named q1.py, provided in my homework folder. (You can reach it via Google Colab, clicking this link as well.)

In the code snippet, I wrote a function for decryption of shift ciphers that take the ciphertext as an argument. Since we know that we will end up with two meaningful plaintexts in the end, at the beginning of the function I created a dictionary for the plaintexts and their shift amounts, so to say, the keys. We know that we are working on mod 26, therefore the maximum shift amount can be 25. In the outer for loop, we’d try each shift amount one by one. At each iteration, we reset the possible plaintext to an empty string and go over the ciphertext while shifting its every character by the shift amount that is determined by the outer for loop. In the end, instead of reading all possible plaintext and choosing the meaningful ones, I used a library called enchant and use that to determine if the possible plaintext of that iteration is an English word or not. If it is in English, I add that word as the plaintext with the corresponding key, the shift amount. In a nutshell, two meaningful plaintexts in English are “WHEEL” with key 9 and “DOLLS” with key 16. You can also see the code and the result in the image below. (In order to use enchant library, there is an installation requirement.)

```
[16]  1 !apt-get install libenchant1c2a
      2 !pip install pyenchant
```

```
 1 import enchant
 2 d = enchant.Dict("en_US")
 3
 4 uppercase ={'A':0, 'B':1, 'C':2, 'D':3, 'E':4, 'F':5, 'G':6, 'H':7, 'I':8,
 5          'J':9, 'K':10, 'L':11, 'M':12, 'N':13, 'O':14, 'P':15, 'Q':16,
 6          'R':17, 'S':18,  'T':19, 'U':20, 'V':21, 'W':22, 'X':23, 'Y':24,
 7          'Z':25}
 8
 9 inv_uppercase = {0:'A', 1:'B', 2:'C', 3:'D', 4:'E', 5:'F', 6:'G', 7:'H',
10                 8:'I', 9:'J', 10:'K', 11:'L', 12:'M', 13:'N', 14:'O', 15:'P',
11                 16:'Q', 17:'R', 18:'S', 19:'T', 20:'U', 21:'V', 22:'W', 23:'X',
12                 24:'Y', 25:'Z'}
13
14 def Shift_Dec(ciphertext):
15   possible_plaintexts = {}
16   for i in range(0,25):
17     plaintext = ""
18     for idx in range(0, len(ciphertext)):
19       plaintext += inv_uppercase[(uppercase[ciphertext[idx]] + i) % 26]
20     if d.check(plaintext):
21       possible_plaintexts[i] = plaintext
22   return possible_plaintexts
23
24 plaintexts_keys = Shift_Dec("NYVVC")
25 print(plaintexts_keys)
```

```
{9: 'WHEEL', 16: 'DOLLS'}
```

Question 2. Since the most frequent letter in the plaintext is provided, we can use that valuable information to obtain a plaintext-ciphertext pair. If we obtain such a pair, then we would get an equation such that we can generate less number of possible alpha-beta pairs to make an exhaustive search much more efficiently. In other words, instead of making an exhaustive search on all of the keyspace which has a size of 26*12 = 312, using that equation, we may have an exhaustive search on a keyspace that has a size of 12 only.

This computation is easy, but I wrote another Python program for this question as well. Again, you can find the implementation as q2.py in my homework folder or you can access it via clicking this link if you use Colab. I have a function to iterate over the ciphertext and count the letters, update the letter_count dictionary meanwhile. For the sake of simplicity, I use the upper() method on the ciphertext, after I reach the result, I use lower() and capitalize() methods to get the plaintext in the provided form. After I count the letters, I found the most frequent letter in the ciphertext, which is G. Then, we have the following equation:

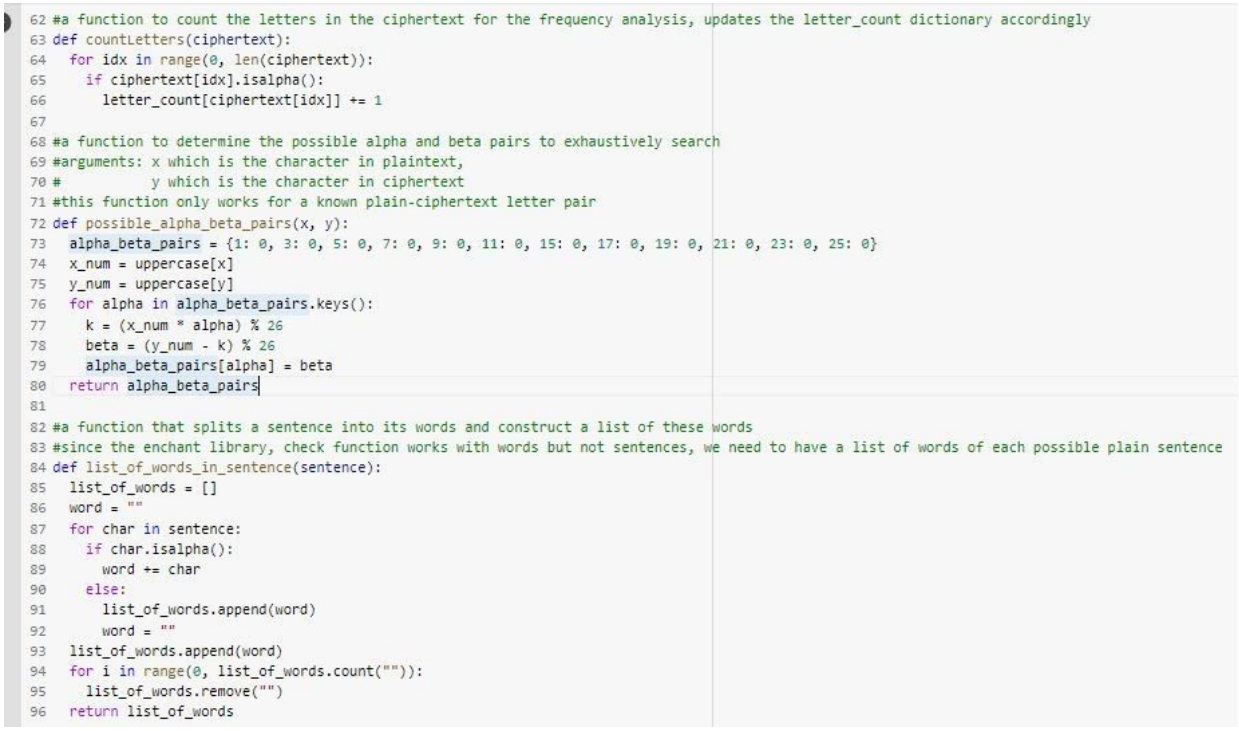
$$6 = 19\ \alpha + \beta \pmod{26}$$

This means that, for each α value, we have only one single β that holds this equation. Using the possible_alpha_beta_pairs function, I compute a dictionary of α keys and β values which has only 12 elements. So, we managed to shrink our keyspace enough to make an exhaustive search efficiently. At this point, we may have another frequency analysis for the second most frequent letters, but the ciphertext is not long enough to provide a relevant frequency analysis and the keyspace is shrunk enough to make the exhaustive search efficient.

Then, all we have to do is iterate over these possible alpha-beta pairs and use the Affine_Dec(...) function which is already provided to have the decrypted text at the end. Having only 12 possible keys means we have only 12 possible decrypted texts as well, but I use the same library that I used in Question 1 to determine if the potential plaintext is in English or not. This library works on words, not sentences; therefore I wrote another function that splits the sentence into its words and checks if that word is in English or not, individually. If a sentence's words are all in the English dictionary, then we found our plaintext and I printed the decrypted text and its corresponding encryption and decryption keys. The sentence is:

"Success is not final, failure is not fatal: it is the courage to continue that counts."
while the encryption key is α = 9, β = 17 and the decryption key is γ = 3, θ = 1.

You can see the code and the result in the image on this page and the next page. The first 60 lines are already provided by the instructor, therefore I cut them off from the screenshots to save some space. I have also installed and imported the enchant library again.



```
#a function to count the letters in the ciphertext for the frequency analysis, updates the letter_count dictionary accordingly
def countLetters(ciphertext):
  for idx in range(0, len(ciphertext)):
    if ciphertext[idx].isalpha():
      letter_count[ciphertext[idx]] += 1

#a function to determine the possible alpha and beta pairs to exhaustively search
#arguments: x which is the character in plaintext,
#           y which is the character in ciphertext
#this function only works for a known plain-ciphertext letter pair
def possible_alpha_beta_pairs(x, y):
  alpha_beta_pairs = {1: 0, 3: 0, 5: 0, 7: 0, 9: 0, 11: 0, 15: 0, 17: 0, 19: 0, 21: 0, 23: 0, 25: 0}
  x_num = uppercase[x]
  y_num = uppercase[y]
  for alpha in alpha_beta_pairs.keys():
    k = (x_num * alpha) % 26
    beta = (y_num - k) % 26
    alpha_beta_pairs[alpha] = beta
  return alpha_beta_pairs

#a function that splits a sentence into its words and construct a list of these words
#since the enchant library, check function works with words but not sentences, we need to have a list of words of each possible plain sentence
def list_of_words_in_sentence(sentence):
  list_of_words = []
  word = ""
  for char in sentence:
    if char.isalpha():
      word += char
    else:
      list_of_words.append(word)
      word = ""
  list_of_words.append(word)
  for i in range(0, list_of_words.count("")):
    list_of_words.remove("")
  return list_of_words
```

```
ciphertext = "Xpjjbxx lx eng klerm, krlmpob lx eng krgrm: lg lx gcb jnportb gn jneglepb gcrg jnpegx."
ciphertext = ciphertext.upper() #to simplify the process of counting
frequent_in_ptext = "T"

countLetters(ciphertext)

#dedicated to find the frequent letter in ciphertext using letter_count dictionary
for dictkey in letter_count:
  if letter_count[dictkey] == max(letter_count.values()):
    frequent_in_ctext = dictkey

alpha_beta_pairs = possible_alpha_beta_pairs(frequent_in_ctext, frequent_in_ptext)

#for each possible alpha-beta pair (total of 12), we compute the encryption and decryption keys
#using the Affine_Dec function provided by the instructor, we decrypt the ciphertext and split the resulting sentence into its words
#in order to make a language check
#if all of its words are in english, then print the plaintext and corresponding encryption and decryption keys, else omit that one
for dictkey in alpha_beta_pairs.keys():
  key.alpha = dictkey
  key.beta = alpha_beta_pairs[dictkey]
  key.gamma = modinv(key.alpha, 26) # you can compute decryption key from encryption key
  key.theta = 26 - (key.gamma * key.beta) % 26

  dtext = Affine_Dec(ciphertext, key)
  list_of_words = list_of_words_in_sentence(dtext)
  in_english = True

  for item in list_of_words:
    if not d.check(item):
      in_english = False
      break

  if in_english:
    print(dtext.lower().capitalize())
    print("Encryption key => alpha =", key.alpha, "and beta =", key.beta)
    print("Decryption key => gamma =", key.gamma, "and theta =", key.theta)
```

```
Success is not final, failure is not fatal: it is the courage to continue that counts.
Encryption key => alpha = 9 and beta = 17
Decryption key => gamma = 3 and theta = 1
```

Question 3. After we fetch the ciphertext and the most frequent letter in the plaintext from the server, we count each letter in the ciphertext to find the most frequent one, because the most frequent letter in the plaintext which is provided by the server is mapped to the most frequent letter in the ciphertext which will give us a linear equation to shrink the keyspace.

Using the equation $18 = 0\ \alpha + \beta \pmod{29}$, we find out that β is always 18 and all we need to do is exhaustively search the possible values of α which have a size of $\Phi(29) = 28$.

For this question, I couldn't find a proper library that checks whether a given text is in Turkish or not. I print 28 possible plaintexts, and the meaningful one is of course obvious with $\alpha = 21$ and $\beta = 18$. In the next code cell, I use the right key pair to decrypt the text. I also sent the resulting decrypted text to the server, but in case of any error, the decrypted text is:

"KOMŞU ÇİFTLİKLERDEN BİR TEMSİLCİLER KURULU BİR DENETLEME GEZİSİ İÇİN ÇAĞRILMIŞTI TÜM ÇİFTLİĞİ GEZEN ÇİFTÇİLER GÖRDÜKLERİ HER ŞEYE ÖZELLİKLE DE YEL DEĞİRMENİNE HAYRAN KALDIKLARINI BELİRTTİLER HAYVANLAR ŞALGAM TARLASINDAKİ AYRIKOTLARINI YOLMAKTAYDILAR KENDİLERİNİ TÜMÜYLE İŞLERİNE VERMİŞLERDİ DAHA ÇOK DOMUZLARDAN MI YOKSA ÇİFTLİĞE KONUK GELEN İNSANLARDAN MI KORKMAK GEREKTİĞİNİ KESTİREMEDİKLERİNDEN BAŞLARINI BİLE KALDIRMIYORLARDI AKŞAMLEYİN ÇİFTLİK EVİNDEN KAHKAHALAR VE ŞARKILAR YÜKSELDİ BİRBİRİNE KARIŞAN SESLERİ DUYAN HAYVANLAR BİRDEN KULAK KESİLDİLER

İLK KEZ EŞİT KOŞULLARDA BİR ARAYA GELEN HAYVANLARLA İNSANLAR ORADA NE YAPIYORLARDI ACABA HEP BİRLİKTE HİÇ SES ÇIKARMAMAYA ÇALIŞARAK ÇİFTLİK EVİNİN BAHÇESİNE YAKLAŞTILAR BAHÇE KAPISININ ÖNÜNE GELDİKLERİNDE ÜRKEREK DURAKSADILARSA DA CLOVERIN ÖNE DÜŞMESİYLE İÇERİ GİRİP PARMAKLARININ UCUNA BASARAK EVE YÖNELDİLER”

I made small changes in my code for the previous questions and converted the functions to their suitable versions for the Turkish language. I also copied and pasted the ciphertext, because this is the trial code for me to work on. You can access the Colab via this link, or see the whole program in the affine_client.py document in my homework folder. Below, you can find the screenshots of code that I write for the trial on Google Colab. Then I used a text editor to edit the affine_client.py, then used the terminal to execute and communicate with the server. You can also see the response from the server below:

```
71
72 ctext = "BYLNA TOZĞŞOBŞGÇMGF IOÇ ĞGLUOŞCOŞGÇ BAÇAŞA IOÇ MGFGĞŞGLG RGVOUO OTOF TÖJÇÜŞLÜNĞÜ ĞSL T
73 countLetters(ctext)
74
75 letter = "A"
76
77 #dedicated to find the frequent letter in ciphertext using letter_count dictionary
78 for dictkey in letter_count:
79   if letter_count[dictkey] == max(letter_count.values()):
80     frequent_in_ctext = dictkey
81
82 alpha_beta_pairs = possible_alpha_beta_pairs(letter, frequent_in_ctext)
83
84 #for each possible alpha-beta pair (total of 29), we compute the encryption and decryption keys
85 #using the Affine_Dec function provided by the instructor, we decrypt the ciphertext and split
86 #in order to make a language check
87 #if all of its words are in english, then print the plaintext and corresponding encryption and
88 for dictkey in alpha_beta_pairs.keys():
89   key.alpha = dictkey
90   key.beta = alpha_beta_pairs[dictkey]
91   key.gamma = modinv(key.alpha, 29) # you can compute decryption key from encryption key
92   key.theta = 29 - (key.gamma * key.beta) % 29
93
94   dtext = Affine_Dec(ctext, key)
95   print(key.alpha, key.beta, "->", dtext)
96   print()
97
```

```
10 18 -> GYOTD MVBZJVGJÜKRÜŞ EVK ZÜOÖVJIVJÜK GDKDJD EVK RÜŞÜZJÜOÜ FÜUVÖV VMVŞ MAİKSJOSTZS ZHO MVBZJVİV FÜUÜŞ MVBZMVJÜK FÇKRHGJÜKV NÜK TÜPÜ ÇUÜJJVGJÜ RÜ PÜJ RÜİ\
11 18 -> HLVKB İSŞGÇSHÇZÜEZR TSÜ GZVPSÇOSÇZÜ HBÜBÇB TSÜ EZRZGÇZVZ NZFSPS SİSR İAIÜYÇVYKGY GUV İSŞGÇSIS NZFZR İSŞGİSÇZÜ NĞÜEUHÇZÜS DZÜ KZJZ ĞFZÇÇSHÇZ EZ JZÇ EZIS
12 18 -> BĞPUK YJÜDIJBINFGNZ HJF DNPMJIÖJINF BKFKIK HJF GNZNDINPN ENRJMJ JYJZ YALFÇIPÇUDÇ DŞP YJÜDIJLJ ENRNZ YJÜDYJINF EOFGŞBINFJ TNF UNİN ORNIIJBIN GN İNI GNL:
13 18 -> STŞİJ NRÇVGRSGOICOĞ MRI VOŞÜRGBRGOI SJIJGJ MRI COĞOVGOŞO ÖOLRÜR RNRĞ NADIEGŞEİVE VYŞ NRÇVGRDR ÖOLOĞ NRÇVNRGOI ÖHICYSGOIR POI İOZO HLOGGRSGO CO ZOG CODF
14 18 -> EİĞDG PCHRSCESŞBFŞU NCB RŞĞOCSÇCSŞB EGBGSG NCB FŞUŞRSŞĞŞ ÜŞKCOC CPCU PAJBMSĞMDRM RTĞ PCHRSCJC ÜŞKŞU PCHRPCSŞB ÜYBFTESŞBC ZŞB DŞVŞ YKŞSSCESŞ FŞ VŞS FŞJ(
15 18 -> UÖSÜŞ IYRHĞYUĞGZTGE KYZ HGSJYĞVYĞGZ UŞZŞĞŞ KYZ TGEGHĞGSG DGNYJY YIYE IAOZLĞSLÜHL HFS IYRHĞYOY DGNGE IYRHIYĞGZ DCZTFUĞGZY BGZ ÜGÇG CNGĞĞYUĞG TG ÇGĞ TGO\
16 18 -> ĞFGÖO KHVÇŞHĞŞJPYJS LHP ÇJGDHŞZHŞJP ĞOPOŞO LHP YJSJÇŞJGJ İJMHDH HKHS KAÜPUŞGUÖÇU ÇCG KHVÇŞHÜH İJMJS KHVÇKHŞJP İRPYCĞŞJPH IJP ÖJBJ RMJŞŞHĞŞJ YJ BJŞ YJÜH
17 18 -> ZSIEN CODÜPOZPKTŞKB ROT ÜKILOPİOPKT ZNTNPN ROT ŞKBKÜPKIK UKHOLO OCOB CAMTVPIVEÜV ÜGI CODÜPOMO UKHKB CODÜCOPKT UJTŞGZPKTO FKT EKÖK JHKPPOZPK ŞK ÖKP ŞKM
18 18 -> RMÇNZ ÖĞGŞVĞRVBDUBH FĞD ŞBÇIĞVJĞVBD RZDZVZ FĞD UBHBŞVBÇB KBTĞIĞ ĞÖĞH ÖAPDCVÇCNŞC ŞEÇ ÖĞGŞVĞPĞ KBTBH ÖĞGŞÖĞVBD KSDUERVBDĞ ÜBD NBOB STBVVĞRVB UB OBV UBP(
19 18 -> SCJFÜ LÇZBOÇŞODNHDG UÇN BDJİÇOPÇODN ŞÜNÜOÜ UÇN HDGDBODJD TDEÇİÇ ÇLÇG LAÖNĞOJĞFBĞ BRJ LÇZBOÇÖÇ TDEDG LÇZBLÇODN TVNHRŞODNÇ KDN FDID VEDOOÇŞOD HD IDO HDÖ(
20 18 -> ÖZTVC ŞKMLFKÖFYSIVİ OKS LYTHKFEKFYS ÖCSCFC OKS IYİYLFYTY ÇYJKHK KŞKİ ŞARSÜFTÜVLÜ LPT ŞKMLFKRK ÇYJYİ ŞKMLŞKFYS ÇNSIPÖFYSK ĞYS VYUY NJYFFKÖFY IY UYF IYR
21 18 -> KOMŞU ÇİFTLİKLERDEN BİR TEMSİLCİLER KURULU BİR DENETLEME GEZİSİ İÇİN ÇAĞRILMIŞTI TÜM ÇİFTLİĞİ GEZEN ÇİFTÇİLER GÖRDÜKLERİ HER ŞEYE ÖZELLİKLE DE YEL DEĞİ
22 18 -> PGKSM RÜİÖNÜPNLYOLI VÜY ÖLKUÜNTÜNLY PMYMNM VÜY OLILÖNLKL ĞLÇÜUÜ ÜRÜI RAEYZNKZSÖZ ÖJK RÜİÖNÜEÜ ĞLÇLI RÜİÖRÜNLY ĞDYOJPNLYÜ CLY SLFL DÇLNNÜPNL OL FLN OLE(
23 18 -> YKRIÇ DEĞSHEYHVOMVC İEO SVRZEHŞEHVO YÇOÇHÇ İEO MVCVSHVRV PVÖEZE EDEC DABOTHRTIST SLR DEĞSHEBE PVÖVC DEĞSDEHVO PUOMLYHVOE JVO IVGV UÖVHHEYHV MV GVH MVBE
```

```
#encryption is done with alpha = 21 and beta = 18
key.alpha = 21
key.beta = alpha_beta_pairs[dictkey]
key.gamma = modinv(key.alpha, 29) # you can compute decryption key from encryption key
key.theta = 29 - (key.gamma * key.beta) % 29

print(Affine_Dec(ctext, key))
print("Encryption key => alpha =", key.alpha, "and beta =", key.beta)
print("Decryption key => gamma =", key.gamma, "and theta =", key.theta)
```

```
KOMŞU ÇİFTLİKLERDEN BİR TEMSİLCİLER KURULU BİR DENETLEME GEZİSİ İÇİN ÇAĞRILMIŞTI TÜM ÇİFTLİĞİ GEZEN ÇİFTÇİLER GÖRDÜ
Encryption key => alpha = 21 and beta = 18
Decryption key => gamma = 18 and theta = 24
```

```
C:\Users\pc\Downloads>py affine_client.py
Congrats!
```

Question 4. After the encoding, in order to differentiate the trigrams (the encryption must be done in such a way that decryption is possible) we should first determine the number of possible permutations of trigrams on alphabet of size 31, such that each encoded number must be unique. First letter of the trigram could be any letter among 31 letters, the second letter of the trigram could be any letter among 31 letters, and the third letter of the trigram could be any letter among 31 letters as well. Therefore the modulus should be 31*31*31 which is 31^3 = 29,791.

After encoding the trigrams and determining the modulus, we have the equation for the affine cipher:

$$y = x*\alpha + \beta \pmod{29,791}$$

Here, x is the number for encoded trigram in the plaintext, and y is the number for the encoded trigram in the ciphertext. Since the encryption must be invertible such that we must have an access to x using y as well; α should be coprime to modulus value which is 29,791. To calculate the values that α might get, we'd use the phi function.

Φ(29,791) is the number of integers that are coprime to 29,791.
$\Phi(29,791) = \Phi(31^3) = 31^3 - 31^2 = 28,830$.

β has always an inverse, therefore the number of possible values of β is 29791.
The size of the keyspace is the multiplication of the number of possible values of α and the number of possible values of β which is equal to 29,791 * 28,830 = 858,874,530.

Question 5. In the previous question, we determined the modulus and understood the mechanism of the encoding. Using the hints provided by the question, if the length of the plaintext is 3k+1 for some integer k and plaintext is ending with a dot, then to perform the encoding, XX should be padded to the dot at the end to make the length of the plaintext multiple of 3. In other words, we observe that the last trigram in the plaintext is ".XX" and we also see

that this last trigram is encrypted to “XFD” in the ciphertext. This means that we have an equation to end up with possible alpha-beta pairs again! The equation is as follows:

(Encoded “.XX”) * α + β (mod 29,791) = (Encoded “XFD”) (mod 29,791)

.XX → 27 * 31 * 31 + 23 * 31 + 23 = 26,683
XFD → 23 * 31 * 31 + 5 * 31 + 3 = 22,261

26,683 * α + β (mod 29,791) = 22,261 (mod 29,791)

We know that we have 28,830 possible α values, we calculated this using the phi function. I modified the phi function that is given in the hw01.helper.py, and created a dictionary that has keys of possible α values, the β values for a particular α is initially 0.
Then I used the possible_alpha_beta_pairs function which takes the encoded version of “.XX”, “XFD” and the alpha-beta pairs dictionary. To encode the trigrams, I used my encoder function. Inside the possible_alpha_beta_pairs function, I iterate over the keys of the dictionary and calculated a β value for each key α.Then I go back to the main, for each α,β pair, I calculate the respective γ and θ values to use in the decryption process. The ciphertext is decrypted with each possible key. During decryption, I first grouped the ciphertext into trigrams, encoded each trigram, decrypted using γ and θ values and decoded again to have the plaintext trigram. Since for this question, we have 28,830 possible keys and 28,830 possible decrypted texts; printing them all and finding the meaningful decryption would take an eternity. That’s why I again used the enchant library (I installed it beforehand). I make a list of words of each decrypted text and check whether the words are in English or not. At the end:

The encryption key: α = 129 and β = 6119.
The decryption key: γ = 26096 and θ = 28127.
The decrypted text is:

```
TO LIVE IS THE RAREST THING IN THE WORLD. MOST PEOPLE EXIST, THAT IS ALL.
```

You can find the code that I have already explained, in my homework folder. The name of the program is q5.py and you can also access it via clicking the link.

You can also find the screenshots of the code and the result below:

```
inv_alphabet = dict([(value, key) for key, value in alphabet.items()])

# This is method to compute Euler's function
# The method here is based on "counting", which is not good for large numbers in
def phi(n, alpha_beta_pairs):
  amount = 0
  for k in range(1, n + 1):
    if math.gcd(n, k) == 1:
      alpha_beta_pairs[k] = 0
      amount += 1
  return alpha_beta_pairs

def decoder(big_number):
  triagram = ""
  last_num = big_number % 31
  big_number -= last_num
  middle_num = (big_number % (31*31)) // 31
  big_number -= middle_num
  first_num = (big_number % (31*31*31)) // (31*31)
  triagram += inv_alphabet[first_num]
  triagram += inv_alphabet[middle_num]
  triagram += inv_alphabet[last_num]
  return triagram


def encoder(triagram):
  encoded = 0
  for idx in range(len(triagram)):
    if idx == 0:
      encoded += 31 * 31 * alphabet[triagram[idx]]
    elif idx == 1:
      encoded += 31 * alphabet[triagram[idx]]
    else:
      encoded += alphabet[triagram[idx]]
  return encoded
```

```
def triagram_seperator(text):
  triagram_list = []
  lenght_text = len(text)
  triagram = ""
  count = 0
  for idx in range(lenght_text):
    if count < 3:
      triagram += text[idx]
      count += 1
    if count == 3:
      triagram_list.append(triagram)
      count = 0
      triagram = ""
  return triagram_list


def Affine_Dec(ctext, key):
    clen = len(ctext)
    ptext = ''
    triagram_list = triagram_seperator(ctext)
    for idx in range(0, len(triagram_list)):
      current_triagram = triagram_list[idx]
      encoded_num = encoder(current_triagram)
      encoded_plaintext_bignumber = (encoded_num * key.gamma + key.theta) % (31*31*31)
      encoded_plaintext_triagram = decoder(encoded_plaintext_bignumber)
      ptext += encoded_plaintext_triagram
    return ptext


class key(object):
    alpha=0
    beta=0
    gamma=0
    theta=0

```

```
#a function to determine the possible alpha and beta pairs to exhaustively search
#arguments: x which is the character in plaintext,
#           y which is the character in ciphertext
#this function only works for a known plain-ciphertext letter pair
def possible_alpha_beta_pairs(x_num, y_num, alpha_beta_pairs):
  for alpha in alpha_beta_pairs.keys():
    k = (x_num * alpha) % (31*31*31)
    beta = (y_num - k) % (31*31*31)
    alpha_beta_pairs[alpha] = beta
  return alpha_beta_pairs

#a function that splits a sentence into its words and construct a list of these words
#since the enchant library, check function works with words but not sentences, we need to have a
def list_of_words_in_sentence(sentence):
  list_of_words = []
  word = ""
  for char in sentence:
    if char.isalpha():
      word += char
    else:
      list_of_words.append(word)
      word = ""
  list_of_words.append(word)
  for i in range(0, list_of_words.count("")):
    list_of_words.remove("")
  return list_of_words

ciphertext = "IDSEOYLTVVDO?PSAUEKZO?LQIILQMP?LQNP!YSFNGSDBJZRZYTZTPS?EVYF,?LQ ,SAXSWTFXFD"

alpha_beta_pairs = {}
alpha_beta_pairs = phi(31*31*31, alpha_beta_pairs)
alpha_beta_pairs = possible_alpha_beta_pairs(encoder(".XX"), encoder("XFD"), alpha_beta_pairs)
```

```
#for each possible alpha-beta pair (total of 28830), we compute the encryption and decryption keys
#using the Affine_Dec function provided by the instructor, we decrypt the ciphertext and split the
#in order to make a language check
#if all of its words are in english, then print the plaintext and corresponding encryption and dec
for dictkey in alpha_beta_pairs.keys():
  key.alpha = dictkey
  key.beta = alpha_beta_pairs[dictkey]
  key.gamma = modinv(key.alpha, 31*31*31) # you can compute decryption key from encryption key
  key.theta = (31*31*31) - (key.gamma * key.beta) % (31*31*31)



  dtext = Affine_Dec(ciphertext, key)


  list_of_words = list_of_words_in_sentence(dtext)

  in_english = True

  for item in list_of_words:
    if not d.check(item):
      in_english = False
      break

  if in_english:
    print(dtext)
    print("Encryption key => alpha =", key.alpha, "and beta =", key.beta)
    print("Decryption key => gamma =", key.gamma, "and theta =", key.theta)
```

```
TO LIVE IS THE RAREST THING IN THE WORLD. MOST PEOPLE EXIST, THAT IS ALL.XX
Encryption key => alpha = 129 and beta = 6119
Decryption key => gamma = 26096 and theta = 28127
```

Question 6. The given ciphertext is too short to apply any frequency analysis using language statistics. But the hint says to us that the message is from the instructor to us. Therefore, there may be the name of the instructor at the end of the message. Since after the last comma, there are 2 words each consisting of 5 letters, this might correspond to ERKAY SAVAS. Now, we know that there is a possibility that ERKAY SAVAS is mapped EXEPY LABUH, but we don't know the length of the key yet. To find out, we should realize a pattern during this mapping.

To find the key length, and also the key, I wrote a Python program (you can access it from my homework folder, its name is q6.py, and also you can click this link to see via Google Colab) which calculates the shift amounts between ERKAY SAVAS and EXEPY LABUH. The program only calculates the shift amounts. By looking at the result, we realize that there is a pattern in the result which is [0, 6, 20, 15, 0, 19, 0, 6, 20, 15]. Then, we see that the key length is 6 since after 6 numbers, the same numbers start repeating. The numbers are {0, 6, 20, 15, 0, 19}. But the order may be different, the key may start from anywhere. There are 6 possible keys, one starting with 0, other starting with 6, other starting with 20, other starting with 15, other starting with 0, and the last one starting with 19. We might try all 6 possibilities, to be precise, try out these ones, one by one:

[0, 6, 20, 15, 0, 19]
[6, 20, 15, 0, 19, 0]
[20, 15, 0, 19, 0, 6]
[15, 0, 19, 0, 6, 20]
[0, 19, 0, 6, 20, 15]
[19, 0, 6, 20, 15, 0]

But instead, we have an observation. The first letter of the plaintext is encrypted with the first letter in the key vector during the encryption. If we count the alphabetical characters in the ciphertext, we found the number 51. This means that we encrypt 51 letters.
(51 % key length) = 51 % 6 = 3. This 3 means that the last 3 items in the difference list that we computed using ERKAY SAVAS, is the point where the key starts over.
Therefore, we have [~~0,~~ 6, 20, 15, 0, 19, 0, ~~6, 20, 15~~]. We know that the key starts at number 6. Therefore the key is [6, 20, 15, 0, 19, 0]. After we find the key vector, using the inverse dictionary, we find the actual key which is "GUPATA".

Using the key, we find the plaintext, which is:
"DEAR STUDENT, SIMPLICITY IS THE KEY TO BRILLIANCE, ERKAY SAVAS"

You can find the screenshot of the code, that can be accessed via the homework folder or the provided link.

```python
uppercase ={'A':0, 'B':1, 'C':2, 'D':3, 'E':4, 'F':5, 'G':6, 'H':7, 'I':8,
         'J':9, 'K':10, 'L':11, 'M':12, 'N':13, 'O':14, 'P':15, 'Q':16,
         'R':17, 'S':18,  'T':19, 'U':20, 'V':21, 'W':22, 'X':23, 'Y':24,
         'Z':25}

inv_uppercase = {0:'A', 1:'B', 2:'C', 3:'D', 4:'E', 5:'F', 6:'G', 7:'H',
                 8:'I', 9:'J', 10:'K', 11:'L', 12:'M', 13:'N', 14:'O', 15:'P',
                 16:'Q', 17:'R', 18:'S', 19:'T', 20:'U', 21:'V', 22:'W', 23:'X',
                 24:'Y', 25:'Z'}


ciphertext = "JYPR LTAXTNM, SOGELBCONN IL TNY ZER TU VGIELOUCCX, EXEPY LABUH"
subciphertext = "EXEPY LABUH"
subplaintext = "ERKAY SAVAS"

diff= []
for i in range(len(subciphertext)):
  if subciphertext[i].isalpha():
    cipher_num = uppercase[subciphertext[i]]
    plain_num = uppercase[subplaintext[i]]
    diff.append((cipher_num - plain_num)%26)

print(diff)
```

```
[0, 6, 20, 15, 0, 19, 0, 6, 20, 15]
```

```python
keyVector = [6,20,15,0,19,0]
key = ""
for i in keyVector:
  key += inv_uppercase[i]
print("Key is:", key)

plaintext = ""
count = 0
for char in ciphertext:
  if char.isalpha():
    plaintext += inv_uppercase[(uppercase[char] - keyVector[count % 6]) % 26]
    count += 1
  else:
    plaintext += char

print("Plaintext is:", plaintext)

```

```
Key is: GUPATA
Plaintext is: DEAR STUDENT, SIMPLICITY IS THE KEY TO BRILLIANCE, ERKAY SAVAS
```

Bonus: In this question, we don't have any hint provided, but the text is long enough to apply language statistics using frequency analysis first. After I did the frequency analysis, I made my decision on my own about the key because this is really difficult to code since I realize that I start exhaustive search while coding, which is not my desire for this long text to decrypt.

Before applying frequency analysis, what we should do is shift the text characters to the right for some amount of shift, then count the coincidences. Since we only encrypted the alphabetic characters, the first thing to do is omit non-alphabetic characters such as spaces and punctuation marks. Then, as stated in the lecture, since the key is used in a cyclic fashion, key length of more that 13 is not usually used. Therefore, to shift the text I formed a loop that iterates 13 times where the shift amount at each iteration is x = 1,2,3,..,13. I stored each shift amount and corresponding number of coincidence pairs in a dictionary called shift_coincidence_dict. The amount of shifts where we observe the maximum number of coincidences is the most probable key length, which is computed as 7 by my program. Then I constructed 7 sub ciphertexts such that in the first sub ciphertext I concatenated 1'st, 8'th, 15'th letters and so on. To do that, I use a function called subciphertext_constructor which returns a list of sub ciphertexts at the end. After that, for each ciphertext we should find the most frequent letters, to make a decision on the encryption and the key on our own. As a result, I have the repetition times of each letter in each sub ciphertext which is down below, relatively most frequent ones are marked as bold: (numbers in the 2'nd row in the table show the upper letter's repetition times in corresponding sub ciphertext, while percentages in the 4'th row show the 3'rd row letter's frequency percentage in English language).

1'st sub ciphertext:

{'A': 4, 'B': 1, 'C': 0, 'D': 3, 'E': 0, 'F': 2, 'G': 0, 'H': 5, 'I': 0, 'J': 3, 'K': 2, **'L': 17**, 'M': 0, 'N': 2, **'O': 8**, 'P': 4, 'Q': 0, 'R': 0, 'S': 5, 'T': 0, **'U': 7**, **'V': 10**, 'W': 2, 'X': 0, 'Y': 5, 'Z': 7}

L	V	O	U	...
17	10	8	7	...
E	O	H	N	...
12.7%	7.5%	6.1%	6.7%	...

When k1 is equal to 7, most frequent letters in the sub ciphertext have been mapped by the relatively frequent letters in English. There is no inconsistency, therefore:

k1 = L (11) - E (4) = 7
= V (21) - O (14) = 7
= O (14) - H (7) = 7
= U(20) - N (13) = 7

k1 → 7 (H)

2’nd sub ciphertext:

{'A': 2, **'B': 10**, 'C': 4, 'D': 2, 'E': 2, 'F': 0, 'G': 0, 'H': 0, 'I': 7, 'J': 1, 'K': 0, 'L': 2, **'M': 8**, 'N': 3, 'O': 1, 'P': 4, 'Q': 7, 'R': 0, 'S': 1, 'T': 4, 'U': 1, 'V': 7, **'W': 11**, 'X': 1, 'Y': 0, **'Z': 9**}

W	B	Z	M	...
11	10	9	8	...
O	T	R	E	...
7.5%	9.1%	6.0%	12.7%	...

Since W is the most frequent letter in the 2’nd sub ciphertext, I first map E to W. But then, some inconsistencies occurred since then J has to be mapped to B but J has a frequency of 0.02%, which is significantly less frequent.

Then I tried the second most frequent letter in English, T to map W, but there were still inconsistencies because then Y is mapped to B but the frequency of Y is 2.0%.

For A maps W, the same thing happened again because F is mapped to B then, but frequency of B is still small, which is 2.2%.

But when I map O which is the 4’th most frequent letter in English to W, the result is consistent.

k2 = W (22) - O (14) = 8
= B (1) - T (19) = 8
= Z (25) - R (17) = 8
= M(12) - E (4) = 8

k2 → 8 (I)

3’rd sub ciphertext:

{**'A': 8**, 'B': 1, 'C': 0, 'D': 6, 'E': 5, **'F': 10**, 'G': 6, 'H': 0, 'I': 2, 'J': 0, 'K': 6, 'L': 0, 'M': 1, 'N': 0, 'O': 0, 'P': 1, **'Q': 14**, 'R': 1, 'S': 3, 'T': 4, 'U': 4, 'V': 0, 'W': 1, 'X': 5, 'Y': 5, 'Z': 4}

Q	F	A	...
14	10	8	...
E	T	O	...
12.7%	9.1%	7.5%	...

When k3 is equal to 12, most frequent letters in the sub ciphertext have been mapped by the relatively frequent letters in English. There is no inconsistency, therefore:

k3 = Q (16) - E (4) = 12
= F (5) - T (19) = 12
= A (0) - O (14) = 12

k3 → 12 (M)

4’th sub ciphertext:

{'A': 5, **'B': 8**, 'C': 1, 'D': 2, 'E': 3, 'F': 0, **'G': 7**, 'H': 0, 'I': 4, 'J': 1, 'K': 2, 'L': 5, **'M': 10**, 'N': 4, 'O': 1, 'P': 6, 'Q': 4, 'R': 0, 'S': 2, 'T': 0, 'U': 2, 'V': 4, 'W': 5, 'X': 1, 'Y': 0, **'Z': 10**}

M	Z	B	...
10	10	8	...
E	R	T	...
12.7%	6.0%	9.1%	...

When k4 is equal to 8, most frequent letters in the sub ciphertext have been mapped by the relatively frequent letters in English. There is no inconsistency, therefore:

k4 = M (12) - E (4) = 8
= Z (25) - R (17) = 8
= B (1) - T (19) = 8

k4 → 8 (I)

5’th sub ciphertext:

{'A': 4, 'B': 7, 'C': 0, 'D': 0, 'E': 5, 'F': 2, 'G': 6, 'H': 7, 'I': 0, 'J': 0, 'K': 3, 'L': 6, **'M': 7**, 'N': 5, 'O': 3, 'P': 2, 'Q': 0, 'R': 4, 'S': 0, **'T': 7**, 'U': 2, 'V': 2, 'W': 3, **'X': 11**, 'Y': 0, 'Z': 1}

X	T	M	...
11	7	7	...
E	A	T	...
12.7%	8.2%	9.1%	...

When k5 is equal to 19, most frequent letters in the sub ciphertext have been mapped by the relatively frequent letters in English. There is no inconsistency, therefore:

k5 = X (23) - E (4) = 19
= T (19) - A (0) = 19
= M (12) - T (19) = 19

k5 → 19 (T)

6’th sub ciphertext:

{'A': 6, 'B': 0, 'C': 0, 'D': 5, 'E': 1, **'F': 7**, 'G': 10, 'H': 0, 'I': 0, 'J': 7, 'K': 5, **'L': 9**, 'M': 2, 'N': 0, 'O': 5, 'P': 0, 'Q': 4, 'R': 0, 'S': 4, 'T': 0, 'U': 0, 'V': 3, **'W': 12**, 'X': 0, 'Y': 1, 'Z': 6}

W	L	F	...
12	9	7	…
E	T	N	...
12.7%	9.1%	6.7%	...

When k6 is equal to 18, most frequent letters in the sub ciphertext have been mapped by the relatively frequent letters in English. There is no inconsistency, therefore:

k6 = W (22) - E (4) = 18
= L (11) - T (19) = 18
= F (5) - N (13) = 18

k6 → 18 (S)

7’th sub ciphertext:

{'A': 0, **'B': 7**, 'C': 7, 'D': 0, 'E': 0, 'F': 4, 'G': 2, 'H': 5, 'I': 5, 'J': 1, 'K': 0, **'L': 6**, **'M': 8**, **'N': 8**, 'O': 3, 'P': 0, 'Q': 3, 'R': 0, 'S': 4, 'T': 0, 'U': 4, 'V': 0, 'W': 2, **'X': 8**, **'Y': 8**, 'Z': 1}

Y	X	M	N	B	L	...
8	8	8	8	7	6	...
E	D	S	T	H	R	...
12.7%	4.3%	6.3%	6.7%	6.1%	6.0%	...

When k7 is equal to 20, most frequent letters in the sub ciphertext have been mapped by the relatively frequent letters in English. There is no inconsistency, therefore:

k7 = Y (24) - E (4) = 20
= X (23) - D (3) = 20
= M (12) - S (18) = 20
= N (13) - T (19) = 20
= B (1) - H (7) = 20
= L (11) - R (17) = 20

k7 → 20 (U)

Therefore, the key is “HIMITSU” and the key vector is [7, 8, 12, 8, 19, 18, 20]. Now, since we got the key, we can decrypt the ciphertext. To do the decryption, I wrote a function called decipher, takes the ciphertext and key vector as input, then returns the decrypted text. You can access both parts of my algorithm, finding the key length part and decryption part via this link, or bonus.py file in my homework folder. If you want to take a look at the code without switching between this document and the code editor, I also added the screenshots of my code and the results below. The decrypted text is:

```
“You said: I’ll go to another country, go to another shore,
find another city better than this one.
Whatever I try to do is fated to turn out wrong
and my heart lies buried like something dead.
How long can I let my mind moulder in this place
Wherever I turn, wherever I look,
I see the black ruins of my life, here,
where I’ve spent so many years, wasted them, destroyed them totally.
You won’t find a new country, won’t find another shore.
This city will always pursue you.
```

You'll walk the same streets, grow old
in the same neighborhoods, turn gray in these same houses.
You'll always end up in this city. Don't hope for things elsewhere:
there's no ship for you, there's no road.
Now that you've wasted your life here, in this small corner,
you've destroyed it everywhere in the world."

```
20
21 def countLetters(ciphertext):
22   letter_count = {'A':0, 'B':0, 'C':0, 'D':0, 'E':0, 'F':0, 'G':0, 'H':0, 'I':0,
23         'J':0, 'K':0, 'L':0, 'M':0, 'N':0, 'O':0, 'P':0, 'Q':0,
24         'R':0, 'S':0,  'T':0, 'U':0, 'V':0, 'W':0, 'X':0, 'Y':0, 'Z':0}
25   ciphertext = ciphertext.upper()
26   for idx in range(0, len(ciphertext)):
27     if ciphertext[idx].isalpha():
28       letter_count[ciphertext[idx]] += 1
29   return letter_count
30
31 def subciphertext_constructor(ciphertext, groupingNum):
32   subciphertext_list = []
33   for i in range(groupingNum):
34     subciphertext_list.append("")
35   idx = 0
36   counter = 0
37   while idx < len(ciphertext):
38     if ciphertext[idx].isalpha():
39       subciphertextpos = counter % groupingNum
40       subciphertext_list[subciphertextpos] += ciphertext[idx]
41       counter +=1
42     idx +=1
43   return subciphertext_list
```

```
45 def omit_nonalphabetic(ciphertext):
46   char_list = []
47   for idx in range(len(ciphertext)):
48     if ciphertext[idx].isalpha():
49       char_list.append(ciphertext[idx])
50   return char_list
51
52 def shift_coincidence(ciphertext):
53   shift_coincidence_dict = {}
54   for shift in range(1,14):
55     coincidence = 0
56     for idx in range(len(ciphertext) - shift):
57       if ciphertext[idx] == ciphertext[idx + shift]:
58         coincidence += 1
59     shift_coincidence_dict[shift] = coincidence
60
61   return shift_coincidence_dict
62
63 ciphertext = "Fwg atax: P'tx oh li hvabawl jwgvmjs, nw fw tfiapqz lziym,\nrqgv uuwf
64 char_list = omit_nonalphabetic(ciphertext)
65
66 shift_coincidence_dict = shift_coincidence(char_list)
67
68 for shift_amount in shift_coincidence_dict:
69   if shift_coincidence_dict[shift_amount] == max(shift_coincidence_dict.values()):
70     possibleKeyLength = shift_amount
```

```
subciphertext_list = subciphertext_constructor(ciphertext, possibleKeyLength)

letterCounts_forSubCiphertexts = []
frequent_lettersDict = {}

#this loop creates a list of dictionaries, where we have the letters as keys
#and repetition times as values in the dictionary. We have 7 such dictionaries,
#which is the keylength
count = 1
for eachList in subciphertext_list:
  letter_countList = countLetters(eachList)
  letterCounts_forSubCiphertexts.append(letter_countList)
  print(letter_countList)
  frequent_list = []
  for dictkey in letter_countList:
    if letter_countList[dictkey] == max(letter_countList.values()):
      frequent_in_ctext = dictkey
      frequent_list.append(frequent_in_ctext)
    frequent_lettersDict[count] = frequent_list
  count +=1

print(frequent_lettersDict)
```

```
{'A': 4, 'B': 1, 'C': 0, 'D': 3, 'E': 0, 'F': 2, 'G': 0, 'H': 5, 'I': 0, 'J': 3, 'K': 2, 'L': 17, 'M': 0, 'N': 2, 'O': 8, 'P': 4, 'Q': 0, 'R': 0
{'A': 2, 'B': 10, 'C': 4, 'D': 2, 'E': 2, 'F': 0, 'G': 0, 'H': 0, 'I': 7, 'J': 1, 'K': 0, 'L': 2, 'M': 8, 'N': 3, 'O': 1, 'P': 4, 'Q': 7, 'R': 0
{'A': 8, 'B': 1, 'C': 0, 'D': 6, 'E': 5, 'F': 10, 'G': 6, 'H': 0, 'I': 2, 'J': 0, 'K': 6, 'L': 0, 'M': 1, 'N': 0, 'O': 0, 'P': 1, 'Q': 14, 'R':
{'A': 5, 'B': 8, 'C': 1, 'D': 2, 'E': 3, 'F': 0, 'G': 7, 'H': 0, 'I': 4, 'J': 1, 'K': 2, 'L': 5, 'M': 10, 'N': 4, 'O': 1, 'P': 6, 'Q': 4, 'R': 0
{'A': 4, 'B': 7, 'C': 0, 'D': 0, 'E': 5, 'F': 2, 'G': 6, 'H': 7, 'I': 0, 'J': 0, 'K': 3, 'L': 6, 'M': 7, 'N': 5, 'O': 3, 'P': 2, 'Q': 0, 'R': 4,
{'A': 6, 'B': 0, 'C': 0, 'D': 5, 'E': 1, 'F': 7, 'G': 10, 'H': 0, 'I': 0, 'J': 7, 'K': 5, 'L': 9, 'M': 2, 'N': 0, 'O': 5, 'P': 0, 'Q': 4, 'R': 0
{'A': 0, 'B': 7, 'C': 7, 'D': 0, 'E': 0, 'F': 4, 'G': 2, 'H': 5, 'I': 5, 'J': 1, 'K': 0, 'L': 6, 'M': 8, 'N': 8, 'O': 3, 'P': 0, 'Q': 3, 'R': 0,
{1: ['L'], 2: ['W'], 3: ['Q'], 4: ['M', 'Z'], 5: ['X'], 6: ['W'], 7: ['M', 'N', 'X', 'Y']}
```

```
def decipher(ciphertext, keyVector):
  plaintext = ""
  counter = 0
  for i in range(len(ciphertext)):
    if ciphertext[i].isalpha():
      shift_amount = keyVector[counter % len(keyVector)]
      if ciphertext[i] in uppercase:
        plaintext += inv_uppercase[(uppercase[ciphertext[i]] - shift_amount) % 26]
      else:
        plaintext += inv_lowercase[(lowercase[ciphertext[i]] - shift_amount) % 26]
      counter += 1
    else:
      plaintext += ciphertext[i]
  return plaintext


key = "HIMITSU"
keyVector = [7,8,12,8,19,18,20]
ciphertext = "Fwg atax: P’tx oh li hvabawl jwgvmjs, nw fw tfiapqz lziym,\nrqgv uuwfp

decryptedText = decipher(ciphertext, keyVector)
print(decryptedText)
```

Output:

```
You said: I'll go to another country, go to another shore,
find another city better than this one.
Whatever I try to do is fated to turn out wrong
and my heart lies buried like something dead.
How long can I let my mind moulder in this place
Wherever I turn, wherever I look,
I see the black ruins of my life, here,
where I've spent so many years, wasted them, destroyed them totally.
You won't find a new country, won't find another shore.
This city will always pursue you.
You'll walk the same streets, grow old
in the same neighborhoods, turn gray in these same houses.
You'll always end up in this city. Don't hope for things elsewhere:
there's no ship for you, there's no road.
Now that you've wasted your life here, in this small corner,
you've destroyed it everywhere in the world.
```