Homework2

October 8, 2017

0.1 HOMEWORK 2

0.2 Exercise 2

```
In [2]: n2=651879475121391233955439822679885588484213411517614690751483196566412985
        p21=367341683697145652508155909017012952436901941759760100425264229486655588
        p22=183670841848572826254077954508506476218450970879880050212632114743327794
        q21=177458617971281574862168108912965760083873913115554722250897143958618836
        q22=887293089856407874310840544564828800419369565577773611254485719793091833
        g2=494791384477162096564601220568844169397089238165655673214022878581967639
```

0.2.1 Organise Parameters in a dictionary

```
In [3]: # Put parameters neatly in a dictionary
        parameter_values = [n2,p21,p22,q21,q22,g2]
        parameter_names = ["n2","p21","p22","q21","q22","g2"]
        parameter_dict = dict( zip(parameter_names, parameter_values ) )
```

```
In [4]: for key in parameter_dict:
            print  'Is',  key, 'prime: ',  is_prime(parameter_dict[key])
```

```
Is g2 prime:  False
Is q21 prime:  True
Is q22 prime:  True
Is p21 prime:  True
Is p22 prime:  True
Is n2 prime:  False
```

In your parameter file, you will find six integers n2, p21, p22, q21, q22 and g2 where g2 ∈ Z∗n2 , n2 = p21q21, p21 = 2p22 +1 and q21 = 2q22 +1. Find the order of g2 in Z∗n2 and write it under Q2 in your answer file.

```
In [5]: print "Verify that n2= p21q21: " , n2 == p21 * q21
```

```
Verify that n2= p21q21:  True
```

```
In [6]: # Since n2 = p21 * q21 and (p21 * q21 ) are prime
        # Then the totient function is easy to compute
        # eu = (p21-1)(q21-1)

        multiplicative_group_order = ((p21)-1)*((q21)-1)
```

```
In [7]: power_mod(g2,multiplicative_group_order,n2) # order of group verification
                                                    # this returns one  verifying t
                                                    # element of the group. This is
                                                    # its order
                                                    # should  be a factor of Group
                                                    # ie # Order_group = k * order_
                                                    # Or # order_of_g2 | Order_of_g
```

```
Out[7]: 1
```

```
In [43]: # The order of g2 should be a factor of multiplicative_group_order
         # multiplicative_group_order= (p21-1)*(p22-1) =  (2p22+1-1)(2q22+1 -1) =
         # the factors of multiplicative_group_order are likely 2,4,2p22,2q22,4q22,
         # We pick the one that satisfies the condition g^k = 1
         # where k is an element of the factors
```

```
In [9]: order_check_list = [2,4, p22,q22,2*p22,2*q22,4*q22,4*q22,p22*q22,2*p22*q22,
        order_names = ["2","4","p22","q22","2*p22","2*q22","4*p22","4*q22","p22*q22
        order_dict = dict( zip(order_names, order_check_list ) )
```

```
In [10]: collector =[];
         def order_of_g2_checker(order_dict,g2,n):
             '''
             Input: takes dictionary consisting
             of factors of order of group.
             and n2

             Output: returns the one satisfying
             the condition that g2^k = 1

             '''
             for key in order_dict:
                 check = power_mod(g2,order_dict[key] ,n2)
                 if check == 1:
                     collector.append(key)
                 else:
                     continue
             return collector
```

```
In [11]: collector_ =   order_of_g2_checker(order_dict,g2,n2)
```

```
In [ ]:
```

```
In [15]: print "all possible orders of g3",collector # show all factors equal
```

```
all possible orders of g3 ['4*p22*q22', '2*p22*q22', 'p22*q22']
```

```
In [17]: print "We pick the minimum in the list ", order_dict["p22*q22"] , " ...sin
```

```
We pick the minimum in the list  16296986878034780848885995566997139712105335287940
```

```
In [ ]:
```

1 Exercise 1

```
In [126]: m11=2185009518812470266713980977591672001160104495906570899359329
          m12=2829959501808824818968744543479596214902066890103603571190716
          m13=2488773373073389824855058953944254285681363869733847010919619
          m14=2903351574709552337015521886268657163608579148088889492054438
          y11=162935749620444109872969676628557735855110794156738574730650631401980
          y12=180577856212268044905453951265943881014496973360420348707851024424113
          w1=118674347160075300731294
          u1=1292323405965699583332436
```

```
In [127]: # Construct a matrix
          M =matrix(ZZ,[[m11,m12],[m13,m14]])
```

```
In [128]: # Construct y matrix
          Y =matrix(ZZ,[[y11],[y12]])
```

```
In [129]: # compute M inverse
          M_inverse = M.inverse()
```

```
In [130]: # Confirm  M_inverse
          print "Confirm the inverse: \n", M_inverse * M
```

```
Confirm the inverse:
[1 0]
[0 1]
```

```
In [131]: # Compute Q and Q_inverse
          Q =  M_inverse * Y
```

```
In [132]: # Verify Q is correct.
          # Using M* Q should return true

          M * Q == Y
```

```
Out[132]: True

In [133]: Q1a1 = 54296442266677581446281052516352573191115346750662600;
          Q1a2 = 15653053127363916075896277170178676443530012934737100

In [134]: Q1b = gcd(Q1a1,Q1a2)

In [135]: print "the gcd  is : " , Q1b

the gcd  is :  2605731508414536065178300
```

1.0.1 Task 2

```
In [136]: # w1 . z1 ≡ u1 (mod Q1b)
          # We can use the theorems
          # Important theorem :
          # Theorem 1: If ac ≡ bc (mod m) and gcd(c, m) = 1, then a≡b (modm).
          #  Theorem 2: If ac ≡ bc (mod m) and gcd(c, m) = d, then a ≡ b (mod (m/d
          # m/d and c/d are relatively prime are relatively prime from theorem 1

In [137]: ######## Solution exist ########
          # Similar to the question on exercise requirement test.
          # We can compute the gcd of (Q1b, w1 )
          # Then verify that, this   divides u1
          print "gcd of Q1b and w1 ", gcd(Q1b, w1)
          print "gcd of Q1b and w1 divides u1. The remainder of u1 mod 2 is zero :'

gcd of Q1b and w1  2
gcd of Q1b and w1 divides u1. The remainder of u1 mod 2 is zero : 0


In [138]: # Follwoing theorem 1
          ########################
          # compute gcd of (u1,w1)

In [139]: print "greatest common factor of u1 and w1: ", gcd(u1,w1);

greatest common factor of u1 and w1:  2


In [140]: # Using theorem 2 #
          #########################
          # Check if gcd of 2 and Q1b is one

In [141]: print "The gcd of 2 and Q1b is : ", gcd(Q1b,2)

The gcd of 2 and Q1b is :  2
```

```
In [142]: # Then confirm gcd(c/d,m/d) is 1
          print "The gcd of 2/2 and Q1b/2 is : ", gcd(Q1b/2,2/2)
```

```
The gcd of 2/2 and Q1b/2 is :  1
```

```
In [143]: ### Divide out the common factor
          ##  Following theorem 2
          print "Divide out  2 from the common integers following theorem 2 \n";

          u_1 = ZZ(u1/2);
          print "u1 divided by 2, u_1= ",u_1;
          w1_1=ZZ(w1/2);
          print "w1 divided by 2, w_1= ",w1_1;
          Q1b_1 = ZZ(Q1b/2)
          print  "Q1b divided by 2, Q1b_1= " ,Q1b_1
```

```
Divide out  2 from the common integers following theorem 2

u1 divided by 2, u_1=  6461617029828497916662 18
w1 divided by 2, w_1=  593371735800376503656 47
Q1b divided by 2, Q1b_1=  1302865754207268032589150
```

```
In [144]: # Compute gcd of w1_1 and u_1 is : one (theorem 2)
          print "the gcd of w1_1 and u_1 is: ", gcd(w1_1,u_1)
          #  And gcd of 1 and Q1b_1 is one.
          # gcd(1, Q1_b1) = 1, then w1_1≡u1_1 (modQ1b_1).
          # If this returns another value, we can still divide further
```

```
the gcd of w1_1 and u_1 is:  1
```

```
In [145]: # gcd of  Q1b_1,w1_1 is one. Modular inverse computation can thus be carr
          print "gcd of Q1b_1 and w1_1 is :",gcd(Q1b_1,w1_1)
```

```
gcd of Q1b_1 and w1_1 is : 1
```

```
In [146]: # Compute w1_1 inverse modulo Q1b_1
          inverse_w1_1 =  inverse_mod(ZZ(w1_1), ZZ(Q1b_1))

In [147]: # the list of all z1

In [148]: # Multiply w1_1 and u_1 to get
          z1 = mod((inverse_w1_1*u_1),Q1b_1)

In [149]: # To compute all solutions
          # We solze  (z1 +  Q1b_11 * k) mod Q1b
```

In [150]:
```
solution_collector = [];
flag = True;
first = True
while flag:
    while first==True:

        solution_collector.append(z1) # Append initial solution
        temp = mod((ZZ(z1) + ZZ(Q1b_1) ),Q1b) # compute z1 plus the Q1b_1
        solution_collector.append(temp)
        first = False;  # break out of this loop
    if temp != z1: # check if the returned value is equal to initial valu
                    # we stop computation
        temp = mod((ZZ(temp)+ZZ(Q1b_1)),Q1b) # add temp to new Q1b_1
        solution_collector.append(temp); # append value
        temp = temp; # set temp to new value and continue loop
    else:
        break;

print "solution is : ", solution_collector
```

```
solution is :  [732377014396862977372694, 2035242768604131009961844]
```

2 Exercise 5

In [151]:
```
# Exercise 4
## encrypt a given word using the caesar cipher
## Computation is done modulo 3
## Note use of ord keyword to return ascii values. ASCII valus for "A" to
## To get values between 0 and 65 subtract 65. Letters of alphabet are en
```

In [7]:
```
def str2Int(s):
    '''
    return integer representantion of the letters
    The values should be between 0 and 25

    input: string of characters.
    output: integers representation (0 to 25) of the string input.
    '''
    whiteSpaceNo = 26; # handle whitespace separatelt since it is not conti
    Substract = 97 ; # substract this value from ord to get a value between
    encode =[];
    for x in s:
            if ord(x)==ord(" "):
                encode.append(whiteSpaceNo)
            else:
                new = ord(x)-Substract
                encode.append(new)
```

```
        return encode
```

```
In [8]: str2Int("e e e ") # test the function
```

```
Out[8]: [4, 26, 4, 26, 4, 26]
```

```
In [9]: def Int2str(lst):
            '''
            convert values after encryption to letters.
            Note 65 is added to the character representation.

            input: integer values between 0 and 25. White space is 26.
            output : string.

            '''
            temp = [] # store integer string
            whiteSpaceNo = 26;

            for x in lst:
                if x==whiteSpaceNo:
                    temp.append([chr(int(x)+6)]) # ord white space is 32

                else:
                    temp.append([chr(int(x)+97)])

            my_list = [l[0] for l in temp] # strip square bracket in the list

            return "".join(f for f in my_list), my_list
```

```
In [26]: Int2str([4, 26, 4, 26, 4, 26]) # test the function. NOTE  function returns
                                         # comma separated values and the values wi
                                         # ie a concatenatin of characters
```

```
Out[26]: ('e e e ', ['e', ' ', 'e', ' ', 'e', ' '])
```

```
In [27]: # Encryption of word 4
         word4= 'incorporated'
         # Using the str to Int function
         word4_encryption = str2Int(word4)
         print "Encryption of word4 is:, ", word4_encryption
```

```
Encryption of word4 is:,  [8, 13, 2, 14, 17, 15, 14, 17, 0, 19, 4, 3]
```

```
In [28]: # Decode encoded4= [15, 4, 17, 2, 7, 8, 13, 6]
         encoded4= [15, 4, 17, 2, 7, 8, 13, 6]
         encoded4_decoded = Int2str(encoded4)
         print "Decryption of word4 is:, ", encoded4_decoded
```

```
Decryption of word4 is:,  ('perching', ['p', 'e', 'r', 'c', 'h', 'i', 'n', 'g'])
```

```
In [49]: C4= [19, 13, 26, 3, 26, 14, 5, 7, 10, 14, 7, 14, 19, 7, 11, 6, 26, 14, 10,
         hint4= [-1, 11, 1, -1, -1, 9, 18, 13, 25, -1, 23, -1, -1, 0, 15, 16, -1, -
```

2.0.1 Solution is to Do frequency analysis

```
In [44]: # letterfreq lists the letters a-z, listed in decreasing order of frequenc
         # as given in note.
         letter = ['a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p',
         frequency = ['0.082','0.015','0.028','0.043','0.127','0.022','0.020','0.06
                      '0.075','0.019','0.001','0.060','0.063','0.091','0.028','0.01
         letterfrequency= parameter_dict = dict( zip(letter, frequency ) )

In [45]: letterprops = sorted(letterfrequency.items(),
                                            key=operator.itemgetter(1),reverse=True) # sort

In [46]: letterprops

Out[46]: [('e', '0.127'),
          ('t', '0.091'),
          ('a', '0.082'),
          ('o', '0.075'),
          ('i', '0.070'),
          ('n', '0.067'),
          ('s', '0.063'),
          ('h', '0.061'),
          ('r', '0.060'),
          ('d', '0.043'),
          ('l', '0.040'),
          ('c', '0.028'),
          ('u', '0.028'),
          ('m', '0.024'),
          ('w', '0.023'),
          ('f', '0.022'),
          ('g', '0.020'),
          ('y', '0.020'),
          ('p', '0.019'),
          ('b', '0.015'),
          ('v', '0.010'),
          ('k', '0.008'),
          ('j', '0.002'),
          ('q', '0.001'),
          ('x', '0.001'),
          ('z', '0.001')]

In [47]: print "letter frequency is: "
         letterprops
```

```
letter frequency is:
```

```
Out[47]: [('e', '0.127'),
          ('t', '0.091'),
          ('a', '0.082'),
          ('o', '0.075'),
          ('i', '0.070'),
          ('n', '0.067'),
          ('s', '0.063'),
          ('h', '0.061'),
          ('r', '0.060'),
          ('d', '0.043'),
          ('l', '0.040'),
          ('c', '0.028'),
          ('u', '0.028'),
          ('m', '0.024'),
          ('w', '0.023'),
          ('f', '0.022'),
          ('g', '0.020'),
          ('y', '0.020'),
          ('p', '0.019'),
          ('b', '0.015'),
          ('v', '0.010'),
          ('k', '0.008'),
          ('j', '0.002'),
          ('q', '0.001'),
          ('x', '0.001'),
          ('z', '0.001')]
```

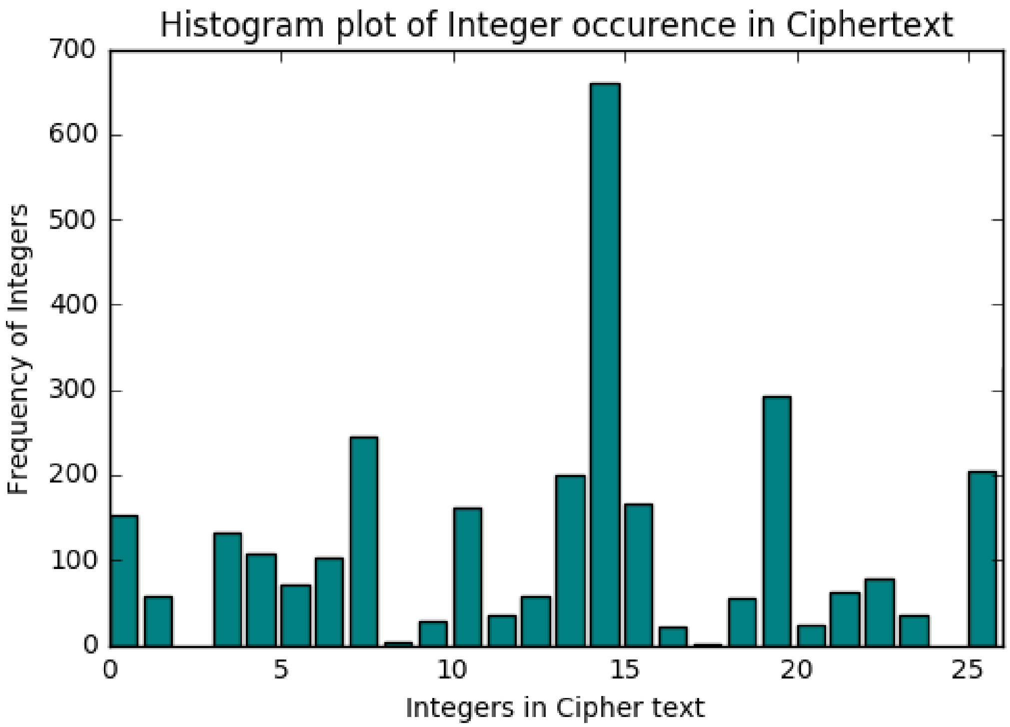
```
In [53]: # Histogram plot gives us hint that some integers are missing
         # this integer must belong to the list occuring integer/ alphabet in the e
         # Inparticular observe 2 and 24 are not represented.----have zero count
         import matplotlib.pyplot as plt

         get_unique_set_element = list(set(C4)) ## gets the unique values in the li

         histo = []

         for i in get_unique_set_element:
             histo.append(C4.count(i)) ## add the number of occurances to the histo

         plt.bar(get_unique_set_element, histo,color="teal")
         plt.title("Histogram plot of Integer occurence in Ciphertext")
         plt.ylabel("Frequency of Integers")
         plt.xlabel("Integers in Cipher text")
         plt.xlim([0,26])
         plt.show()
```




```
In [56]: from collections import Counter # import collection to store count of inte
                                         # we can also use a for loop see below

In [57]: Frequency_of_occurency = Counter(C4) # call counter from collection --- se

In [58]: Frequency_of_occurency # see integers and their frequency in a dictionary

Out[58]: Counter({14: 660, 26: 327, 19: 293, 7: 245, 25: 205, 13: 199, 15: 166, 10:

In [62]: # to verify the output
         # we use default dictionary and use a for loop to count frequencies.
         # This returns values same as that of using the counter from python colle
         # library
         from collections import defaultdict
         Frequency_count_check = defaultdict(int)
         counter_space= 0
         for obj in C4:
             if Integer(obj) < 0:
                 counter_space +=1
             else:
                 Frequency_count_check[obj] += 1

In [63]: Frequency_count_check # this agrees with the

Out[63]: defaultdict(<type 'int'>, {0: 153, 1: 58, 3: 132, 4: 108, 5: 71, 6: 103, 7
```

In [73]:
```
# This is the hint
hint4= [-1, 11, 1, -1, -1, 9, 18, 13, 25, -1, 23, -1,
        -1, 0, 15, 16, -1, -1, -1, -1, -1, -1, 5, 17, 12, 2, -1]
```

In [73]:
```
# Observing the the histogram,
# we can verify the integer 2 and 24 did not occur
for x in C4:
    if x == 24 or x == 24: # No number 2 or 24 in the digits. So 2 must be
        print "yes"
```

In [74]:
```
# sort the dicionary of intergers and count
Sorted_dictionary = sorted(Frequency_count_check.items(),
                           key=operator.itemgetter(1),reverse=True) # sort
```

In [ ]:

In [75]:
```
print "cipher text frequency analysis:"
Sorted_dictionary
```

```
cipher text frequency analysis:
```

Out[75]:
```
[(14, 660),
 (26, 327),
 (19, 293),
 (7, 245),
 (25, 205),
 (13, 199),
 (15, 166),
 (10, 162),
 (0, 153),
 (3, 132),
 (4, 108),
 (6, 103),
 (22, 77),
 (5, 71),
 (21, 61),
 (1, 58),
 (12, 57),
 (18, 55),
 (11, 35),
 (23, 34),
 (9, 28),
 (20, 23),
 (16, 21),
 (8, 3),
 (17, 1)]
```

```
In [76]: # Now below is the English word frequency
         print "English words frequency: \n "
         letterprops
```

English words frequency:

```
Out[76]: [('e', '0.127'),
          ('t', '0.091'),
          ('a', '0.082'),
          ('o', '0.075'),
          ('i', '0.070'),
          ('n', '0.067'),
          ('s', '0.063'),
          ('h', '0.061'),
          ('r', '0.060'),
          ('d', '0.043'),
          ('l', '0.040'),
          ('c', '0.028'),
          ('u', '0.028'),
          ('m', '0.024'),
          ('w', '0.023'),
          ('f', '0.022'),
          ('g', '0.020'),
          ('y', '0.020'),
          ('p', '0.019'),
          ('b', '0.015'),
          ('v', '0.010'),
          ('k', '0.008'),
          ('j', '0.002'),
          ('q', '0.001'),
          ('x', '0.001'),
          ('z', '0.001')]
```

```
In [91]: count = 0;
         # since white space is the most freqient we assign it  14
         # and use count to increment the loop each letterprops which has no white
         print " Map the English word frequency to sorted  Frequuency analysis of c
         for key in range(0,len(Sorted_dictionary)):
             if key == 0:
                 print  "Freuency_analysis_comparism( [whitespace] ) = ", Sorted_di

             else:
                 print  "Frequency_analysis_comparism(" , letterprops[count][0]  ,
                 count+=1
```

Map the English word frequency to sorted Frequuency analysis of cipher text:

```
Freuency_analysis_comparism( [whitespace] ) =  14
Frequency_analysis_comparism( e  ) =  26
Frequency_analysis_comparism( t  ) =  19
Frequency_analysis_comparism( a  ) =  7
Frequency_analysis_comparism( o  ) =  25
Frequency_analysis_comparism( i  ) =  13
Frequency_analysis_comparism( n  ) =  15
Frequency_analysis_comparism( s  ) =  10
Frequency_analysis_comparism( h  ) =  0
Frequency_analysis_comparism( r  ) =  3
Frequency_analysis_comparism( d  ) =  4
Frequency_analysis_comparism( l  ) =  6
Frequency_analysis_comparism( c  ) =  22
Frequency_analysis_comparism( u  ) =  5
Frequency_analysis_comparism( m  ) =  21
Frequency_analysis_comparism( w  ) =  1
Frequency_analysis_comparism( f  ) =  12
Frequency_analysis_comparism( g  ) =  18
Frequency_analysis_comparism( y  ) =  11
Frequency_analysis_comparism( p  ) =  23
Frequency_analysis_comparism( b  ) =  9
Frequency_analysis_comparism( v  ) =  20
Frequency_analysis_comparism( k  ) =  16
Frequency_analysis_comparism( j  ) =  8
Frequency_analysis_comparism( q  ) =  17
```

```
In [92]: # Using the frequecy mapping above and the hint:
         # We have that the most frequent string is the whitespace
         # Thus white space is the seperation between the words.
         # Therefore, we read the cipher text at intervals of 14-(which is white sp
         # This will show that 26 maps to e
         # and that t also maps to 19
         # and finally 7 also maps to a,.
         # The others we detect reading the text
         # There is 2 missing in the whole cipher text and it corresponds to z in t
         # This is not suprising since it is less frequent
         # In addition j and q mappings also seem to be correct directly from frequ
```

```
In [78]: QValues = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,2
         Qkeys = [7,11,1,4,26,9,18,13,25,8,23,6,21,0,15,16,24,3,10,19,22,20,5,17,12
```

```
In [79]: Q3b = parameter_dict = dict( zip(QValues, Qkeys ) )
```

```
In [80]: for i in Q3b:
             print "Q3b(", i,") = ", Q3b[i]
```

```
Q3b( 0 ) =  7
Q3b( 1 ) =  11
```

```
Q3b( 2 ) =  1
Q3b( 3 ) =  4
Q3b( 4 ) =  26
Q3b( 5 ) =  9
Q3b( 6 ) =  18
Q3b( 7 ) =  13
Q3b( 8 ) =  25
Q3b( 9 ) =  8
Q3b( 10 ) =  23
Q3b( 11 ) =  6
Q3b( 12 ) =  21
Q3b( 13 ) =  0
Q3b( 14 ) =  15
Q3b( 15 ) =  16
Q3b( 16 ) =  24
Q3b( 17 ) =  3
Q3b( 18 ) =  10
Q3b( 19 ) =  19
Q3b( 20 ) =  22
Q3b( 21 ) =  20
Q3b( 22 ) =  5
Q3b( 23 ) =  17
Q3b( 24 ) =  12
Q3b( 25 ) =  2
Q3b( 26 ) =  14
```

In [ ]:

In [84]:
```
# This decryptor function will decode the cipher text wince we now know th
decryptor = parameter_dict = dict( zip(Qkeys,QValues ) )
decrypted_text = []
for i in C4:
    if i in QValues:
          #if i != 19:
          #print i
          temp = decryptor[i]
          decrypted_text.append(temp)
          #print decryptor[i]
          #break
```

In [89]:
```
decoded_cipher_text= Int2str(decrypted_text)
```

In [90]:
```
decoded_cipher_text[0]
```

Out[90]:
```
'there was a table set out under a tree in front of the house and the marc
```

In [147]:
```
# This is from Alice in Wonderland;
```

In [10]:

3 Exercise 3

In [154]:
```
n3=568830357877119847097460449419085423536863856861868992248085501685701
g3=203698518359409027124265214238059481017110873774291553498692977466663
X3=248679706080276757382170068920822764513877101364898013653555411947642
Y3=289985203603444913685169959893808797852583434797311705353716284232897
```

In [155]:
```
 # Alice chooses random a in {Z}_p^* and sends A=aP (this means  P+ ...+
 # Bob chooses  b in {Z}_p^* and sends B=bP.
 # Alice computes K=aB=(ab)P
 # Bob computes K=bA=(ba)P=(ab)P
```

In [156]:
```
# If we transpose the Diffie-Hellman in the additive group (Zp,+),
# the intractibility of the discrete logarithm problem is no longer satis
```

In [157]:
```
# The exponentation is transposed to a multiplication,
# while the discrete logarithm operation becomes
# equivalent to a division (i.e., to a multiplication with an inverse ele
```

In [158]:
```
# As computing the inverse (with respect to the multiplication) of an ele
# in Zp is an easy task with the Ex- tended Euclid Algorithm,
```

In [159]:
```
# Solution#####
# we do not know a and b
# But we know bg = x_3 mod n3
# Which implies that computing the inverse: bg * g^-1 = x^3 * g^-1 mod n
```

In [160]:
```
print "gcd of g3 and n3 is: ",gcd(g3, n3)
print "gcd of X3 and n3 is: ", gcd(X3, n3)
print "gcd of X3 and g3 is: ", gcd(X3, n3)
```

```
gcd of g3 and n3 is:  1
gcd of X3 and n3 is:  1
gcd of X3 and g3 is:  1
```

In [161]:
```
inverse_g3 = inverse_mod(g3, n3)
b = mod((inverse_g3*X3),n3)
print "the inverse of g3 mod n3 is \n:", inverse_g3
print  "\n this is our b \n", b
```

```
the inverse of g3 mod n3 is
: 4963428056654161989910881599687196161473324506105624946759302918148925225983860

 this is our b
34017958190506706324838256085325820265105352860857142346292737049368607101457374012
```

In [162]:
```
# TO check, denote gb as product of g and b
gb = mod((g3*b), n3)
print "is gb equal to X3:", gb == X3
```

```
is gb equal to X3: True
```

```
In [163]: # Similarly to compute a
          # But we know ag = Y_3 mod n3
          # Which implies that computing the inverse: bg * g^-1 = Y^3 * g^-1 mod n3
```

```
In [164]: print "gcd of g3 and n3 is: ",gcd(g3, n3)
          print "gcd of X3 and n3 is: ", gcd(X3, n3)
          print "gcd of Y3 and g3 is: ", gcd(X3, n3)
```

```
gcd of g3 and n3 is:  1
gcd of X3 and n3 is:  1
gcd of Y3 and g3 is:  1
```

```
In [165]: a = mod((inverse_g3*Y3),n3)
          print  "\n This is our a:  \n", a
```

```
 This is our a:
156557508707931622209860944864231081049060688229999436142828632598044120008216614
```

```
In [166]: # To check
          # TO check, denote gb as product of g and b
          ga = mod((a*g3), n3)
          print "is gb equal to X3", ga == Y3
```

```
is gb equal to X3 True
```

```
In [167]: # To compute the final Compute the output Q3 of the
          # DH key exchange between Alice and Bob and write it in your
          # Final output is : b(Y3) = b(ag3) and a(X3) = a (bg3)
```

```
In [168]: # Both Alice and Bob should have same final key
          # We compute for each separately  and compare. To confirm
          # equality

          Q3a = mod((b*Y3), n3)
          Q3b = mod((a*X3),n3)
```

```
In [169]: # Check the output is same
          print "the output of both Alice and Bob is same: ", Q3a == Q3b
```

```
the output of both Alice and Bob is same:  True
```

```
In [170]: Q3 = Q3a
```

```
In [171]: print "The Q3 is: \n", Q3

The Q3 is:
48392708167681394736021577917471700183429743894786023673753202450964184935043951381
```

4 Exercise 5

```
In [172]: C5= [15, 1, 19, 19, 23, 14, 17, 3, 26, 9, 19, 26, 19, 2, 15, 20, 17, 7, 5
          # WE observe that the plain text contains 26 at different intervals
          # Since we made a good guess that the message contains words from dictio
          # and spaces. we can reasonable assume the space correspond to 26.
          # In particular since we also known that in any message the encoding of
          # space is either 26 or 0 (26 + 1 = 0 or 26 + 0 = 26) and no zero in the
          # We can first work with the assumption that26 is space

In [173]: f = open("dictionary.txt","r") # open the dictionary
          #ct = f.readlines()

In [3]: # read the dictionary
        dictionary=[]
        for line in f:
            line=line.rstrip("\n")
            dictionary.append(line)

In [4]: dictionary[1:10] # see the result of what was read

Out[4]: ['aardvark',
         'aardvarks',
         'abaci',
         'aback',
         'abacus',
         'abacuses',
         'abaft',
         'abalone',
         'abalones']

In [10]: # encode each word from the dictionary
         dictionary_encoded =[];
         for x in dictionary:
             temp = str2Int(x)
             dictionary_encoded.append(temp)

In [64]: dictionary_encoded[1]

Out[64]: [0, 0, 17, 3, 21, 0, 17, 10]

In [11]: # Now combine the encoding and the dictionary text into a dictionary
         from collections import OrderedDict
```

```
dictionary_of_encodings = OrderedDict( zip(dictionary, dictionary_encoded
```

```
In [12]: # This is our cipher text
         # we consider the ciphertext small subset at a time
         # In particular we consider all integers using 26
         # as our delimiter
         # so that we get something like this
         # [15, 1, 19, 19, 23, 14, 17, 3]  + 26 +
         # [9, 19,] + 26 +
         # [19, 2, 15, 20, 17, 7, 5, 19]
         # we try to decode each of this subset
         C5= [15, 1, 19, 19, 23, 14, 17, 3, 26, 9, 19, 26, 19, 2, 15, 20, 17, 7, 5,
```

```
In [152]: dictionary_of_encodings.items()[0:5] # test the new word-encoding pair in
```

```
Out[152]: [('a', [0]),
           ('aardvark', [0, 0, 17, 3, 21, 0, 17, 10]),
           ('aardvarks', [0, 0, 17, 3, 21, 0, 17, 10, 18]),
           ('abaci', [0, 1, 0, 2, 8]),
           ('aback', [0, 1, 0, 2, 10])]
```

```
In [14]: dictionary_of_encodings['s'] # we can return the encoding for s
```

```
Out[14]: [18]
```

```
In [ ]: C5= [15, 1, 19, 19, 23, 14, 17, 3, 26, 9, 19, 26, 19, 2, 15, 20, 17, 7, 5,
```

```
In [241]: # Break C5 into sublist using 26 as  space as explained above
          C51 = [15, 1, 19, 19, 23, 14, 17, 3]
          C52 = [9, 19]
          C53 = [19, 2, 15, 20, 17, 7, 5, 19]
```

```
In [242]: import itertools

          def key_combination_generator(K,N):
              '''
              Function returns all
              combinations of 1 and 0
              of length 2^(k*N)
              input : integer K and N
              output: 2^(k*N) of {1 , 0}* each of length (k*N)

              '''
              collector = [];
              [collector.append(i) for i in itertools.product([0, 1], repeat = K*N)
              return collector
```

In [243]:
```
C51_keys_collector = key_combination_generator(1,len(C51))
C52_keys_collector = key_combination_generator(1,len(C52))
C53_keys_collector = key_combination_generator(1,len(C53))
```

In [ ]:

In [265]:
```
def all_possible_decryption(possible_keys,cipher_text):
    '''
    Input : decryption keys in the set {0,1}
    Output: List. All possible decryptions
            List. All possible keys
    '''
    collector = [];
    for item in possible_keys:
        temp = list(item) # store an instance of key here
        templist =[] # empty list. to store values after subtration of ke
        for i,j in  map(None,cipher_text,temp):
            #print i,j
            temp2 = i-j # decryption step. Subtract the each element of k
            templist.append(temp2) # put in templist. And empty temp list
            #print
            #print templist
        collector.append(templist)
    return collector
```

In [274]:
```
C51_all_decryption = all_possible_decryption(C51_keys_collector,C51)
C52_all_decryption = all_possible_decryption(C52_keys_collector,C52)
C53_all_decrption =all_possible_decryption(C53_keys_collector,C53)
```

In [275]:
```
# Use lambda mapping and any to see common element from the
# lsit of decryption  and list words in dictionary
```

In [276]:
```
#Use simple lambda expression to check if a common list element exist bet
# the list of dictionary encodings and all possibe encryption
# As we can see there is a common element between dictionary encoding
# and the list of all possible encryption
print any(map(lambda x: x in dictionary_encoded,C51_all_decryption))
print any(map(lambda x: x in dictionary_encoded,C52_all_decryption))
print any(map(lambda x: x in dictionary_encoded,C53_all_decrption))
```

```
True
True
True
```

In [310]:
```
# Since true was return above
# We use the function below
```

```
        # to retrieve all element corresponding to true from list

        all_decryption = [C51_all_deccryption,C52_all_decryption,C53_all_decrptic
        def decryption(index):
            '''
            Input: integer i. Signifying index in all_decryption
            Output : List. The ecncoding for the element
            '''
            common_element = []
            for i in dictionary_encoded:
                for j in all_decryption[index]:
                    if j ==i:
                        common_element.append(i)
            common_element = common_element[0]
            return common_element
```

```
In [311]: index1 = 0; # index of C51_all_deccryption in all_decryption
          index2 =1; # index of C52_all_deccryption in all_decryption
          index3 = 2 ; # index of C52_all_deccryption in all_decryption
          C51_encoding = decryption(index1)
          C52_encoding = decryption(index2)
          C53_encoding =decryption(index3)
```

```
In [312]: # Next we get the word for the encoding from dictionary
```

```
In [313]: def phrase(index):
              temp = ''
              for k, v in dictionary_of_encodings.items():
                  if index == v :
                   temp = k
              return temp + " "
```

```
In [314]: phrase1 = phrase(C51_encoding)
          phrase2 = phrase(C52_encoding)
          phrase3 = phrase(C53_encoding)
```

```
In [315]: print "The decrypted phrase phrase is"
          phrase1 + " " + phrase2 +" " + phrase3
```

```
The decrypted phrase phrase is
```

```
Out[315]: 'password  is  scourges '
```

```
In [ ]:
```