REPORT OF THE ADVANCE CRYPTOGRAPHY

Malaviya National Institute of Technology, Jaipur

November 2018

REPORT OF THE ADVANCE CRYPTOGRAPHY

Malaviya National Institute
of Technology, Jaipur

Pappu Kumar

November 2018

Report submitted in partial fulfilment for the degree of
M.Tech (Computer Engineering & Information Security)

TABLE OF CONTENTS

1. Caesar Cipher

1.1 Assignment Problem:

Implement Caesar cipher

1.2 Logic/methodology explanation:

The Caesar Cipher technique is one of the earliest and simplest method of encryption technique. It's simply a type of substitution cipher, i.e., each letter of a given text is replaced by a letter some fixed number of positions down the alphabet. (Encryption Phase with shift n)

$$E(n) = (x + n) \bmod 26$$

(Decryption Phase with shift n)

$$D(n) = (x + n) \bmod 26$$

1.3 Code

```
with open('ram.txt','r') as f:
  a = f.read()
  #print(a)

b = []
```

```
c = []

d = []

e = []

k = eval(input("Enter a number: "))

for i in range(len(a)):
  b.append(((int(ord(a[i]) - 96) + k) % 26) + 96)
  c.append(chr(b[i]))

str = ("".join(c))

print("The Encrypted Massage : ",str)

with open('ram.txt','a') as f:
  f.write(str)

#for key in range(25):

for i in range(len(c)):
  d.append(((int(ord(c[i]) - 96) - k) % 26) + 96)
  e.append(chr(d[i]))
     #print(chr(d[i]))

str2 = ("".join(e))

print("The Decrypted Massage : ",str2)


t = []

g = []

for key in range(1,25):
  for j in range(len(c)):
     [f.append(chr(((int(ord(c[j]) - 96) - key) % 26) + 96))]

print(f)

trans = [[chr(((int(ord(c[j]) - 96) - key) % 26) + 96) for j in range(len(c))] for key in range(1,25)]
print(trans)
```

2. Substitution Cipher

2.1 Assignment Problem:

Implement substitution cipher

2.2 Logic/methodology explanation:

Substitution Cipher are the most common form of cipher. The work by replacing each letter of the paintext (and sometimes punctuation marks and spaces) with another letter (or possibly even a random symbol).

A monoalphabetic substitution cipher, also known as simple substitution cipher, relies on a fixed replacement structure. That is substitution is fixed for each letter of the alphabet. Thus if “a” is encrypted to “R” ,then every time we see the letter “a” in the plaintext, we replace it with letter “R” in the ciphertext.

2.3 Code

```
x = input("enter the text : ")


k = x = input("enter the text : ")


k = ['q','w','e','r','t','y','u','i','o','p','a','s','d','f','g','h','j','k','l','z','x','c','v','b','n','m']
```

```
alpha =
['a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u','v','w','x','y','z']

b =[]

t = []

for i in range(len(x)):
  t.append(k[alpha.index(x[i])])
  #b.extend(k[int(ord(x[(i)]) - 97)])

print(t)
for i in range(len(t)):
  b.append(alpha[k.index(t[i])])
  #b.extend(k[int(ord(x[(i)]) - 97)])
print(b)
```

3. Vigenere Cipher:

3.1 Assignment Problem:

Implement Vigenere cipher

3.2 Logic/methodology explanation:

Vigenere Cipher is a method of encrypting alphabetic text. It uses a simple form of polyalphabetic substitution. A polyalphabetic cipher is any cipher based on substitution, using multiple substitution alphabets .The encryption of the original text is done using the Vigenère square or Vigenère table.

3.3 Code

```
a = ['a','b','c','d','e']

k = ['a','y','u','s','h']

b =[]
c= []

for i in range(len(a)):
  b.append((int((ord(a[i]) - 96) + (ord(k[i]) - 96)) % 26) + 96)
  #c.append(chr(b[i]))
```

```
print(b)
'''

#print(b)
#print(c)



#a = ['a','b','c','d','e']
#a = input("Enter the Massage : ")
'''
with open('ram.txt','r') as f:
  a = f.read()
'''
a = 'hello'

k = "ayush"

#k = input("enter the key of text: ")
#k = ['a','y','u','s','h']

b =[]
c= []

for i in range(len(a)):
  b.append((int((ord(a[i]) - 96) + (ord(k[i]) - 96)) % 26) + 96)
  c.append(chr(b[i]))
#print(b)
print(c)

d = []
e = []

for i in range(len(c)):
  d.append((int((ord(c[i]) - 96) - (ord(k[i]) - 96)) % 26) + 96)
  e.append(chr(d[i]))
#print(b)
print(e)
```

4. Affine Cipher:

4.1 Assignment Problem:

Implement Affine cipher

4.2 Logic/methodology explanation:

In the affine cipher the letters of an alphabet of size m are first mapped to the integers in the range 0 … m − 1. It then uses modular arithmetic to transform the integer that each plaintext letter corresponds to into another integer that correspond to a ciphertext letter. The encryption function for a single letter is E(x) = (ax+b) mod m where modulus m is the size of the alphabet and a and b are the key of the cipher. The value a must be chosen such that a and m are coprime. The decryption function is D(x) = aa -1 mod m where a −1 is the modular multiplicative inverse of a modulo m. I.e., it satisfies the equation 1 = aa -1 mod m The multiplicative inverse of a only exists if a and m are coprime. Hence without the restriction on a, decryption might not be possible.

4.3 Code

```
m = input("enter the massage ")

A = eval(input("Enter a number(A): "))

B = eval(input("Enter a number(B): "))
#k = ['a','y','u','s','h']

b =[]
```

```
c= []

for i in range(len(m)):
   b.append((((int(ord(m[i]) - 96))*A + B) % 26) + 96)
   c.append(chr(b[i]))

print(c)

d = []
e = []

number = [1, 3, 5, 7, 9, 11, 15, 17, 19, 21, 23, 25]
inverse = [1, 9, 21, 15, 3, 19, 7, 23, 11, 5, 17, 25]

t = number.index(A)

mi = inverse[t]

for i in range(len(c)):
   d.append((mi*(int((ord(c[i]) - 96) - B)) % 26) + 96)
   e.append(chr(d[i]))
print(e)
```

5. Playfair Cipher:

5.1 Assignment Problem:

Implement playfair cipher

5.2 Logic/methodology explanation:

The Playfair cipher encrypts pairs of letters (digraphs), instead of single letters as is the case with simpler substitution ciphers such as the Caesar Cipher. Frequency analysis is still possible on the Playfair cipher, however it would be against 600 possible pairs of letters instead of 26 different possible letters. For this reason the Playfair cipher is much more secure than older substitution ciphers, and it's use continued up until WWII.
The playfair cipher starts with creating a key table. The key table is a 5×5 grid of letters that will act as the key for encrypting your plaintext. Each of the 25 letters must be unique and one letter of the alphabet (usually Q) is omitted from the table (as there are 25 spots and 26 letters in the alphabet).

5.3 Code

```
key = ['m','o','n','a','r','c','h','y']

alpha = ['a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u','v','w','x','y','z']

main = ['m','o','n','a','r','c','h','y','b','d','e','f','g','i','k','l','p','q','s','t','u','v','w','x','z']

k = 5

matrix = [main[i * k:(i + 1) * k] for i in range((len(main) + k - 1) // k )]
```

```
print (matrix)
plain = 'haesmaiprkpv'
t = []
for i in range(len(plain)):
  t.append(main.index(plain[i]))
print(t)
n = 2
rowdivide = [t[i * n:(i + 1) * n] for i in range((len(t) + n - 1) // n )]
print (rowdivide)
row = []
column = []


for i in range(len(rowdivide)):
  for j in range(len(rowdivide[0])):
    row.append(int(rowdivide[i][j]/k))
    column.append(rowdivide[i][j]%k)


print(row)
print(column)

for i in range(0,len(row),2):
  if(row[i] == row[i+1]):
    print(matrix[row[i]][column[i]+1])
    print(matrix[row[i]+1][column[i]+2])

    #print(row[i])
    #print(column[i]+1)

    #[(row[i]*5 + [column[i+1]])]

    #print(main[int(row[i]*5 + column[i+1])])

  elif(column[i] == column[i+1]):
    print(matrix[row[i]+1][column[i]])
    print(matrix[row[i]+2][column[i]])
    #print('hhhh')
```

```
else:
    #print('ggggg')



    #print(row[i]*5 + column[i])
    #print(row[i+1]*5 + column[i+1])

    #main[row[i]*5 + column[(5-i)]]
    #main[row[i+1]*5 + column[(5-i-1)]]



    #print(matrix[row[i]][column[(5-i)]])
    #print(matrix[row[i]+1][column[(5-(i+1))]])

    print(column[(5-(i+1))])
```

6. Autokey Cipher:

6.1 Assignment Problem:

Implement autokey cipher.

6.2 Logic explanation:

The autokey cipher, as used by members of the American Cryptogram Association, starts with a relatively-short keyword, the *primer*, and appends the message to it. If, for example, the keyword is "QUEENLY" and the message is "ATTACK AT DAWN", the key would be "QUEENLYATTACKATDAWN"

1.3 Code

```
m = 'fortificationdefendtheeastwa'

key = 'defendtheeastwallofthecastle'

#key  =  ''

alpha = ['a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u','v','w','x','y','z']

list = ['abcdefghijklmnopqrstuvwxyz','bcdefghijklmnopqrstuvwxyza',
```

```
    'cdefghijklmnopqrstuvwxyzab','defghijklmnopqrstuvwxyzabc',
    'efghijklmnopqrstuvwxyzabcd','fghijklmnopqrstuvwxyzabcde',
    'ghijklmnopqrstuvwxyzabcdef','hijklmnopqrstuvwxyzabcdefg',
    'ijklmnopqrstuvwxyzabcdefgh','jklmnopqrstuvwxyzabcdefghi',
    'klmnopqrstuvwxyzabcdefghij','lmnopqrstuvwxyzabcdefghijk',
    'mnopqrstuvwxyzabcdefghijkl','nopqrstuvwxyzabcdefghijklm',
    'opqrstuvwxyzabcdefghijklmn','pqrstuvwxyzabcdefghijklmno',
    'qrstuvwxyzabcdefghijklmnop','rstuvwxyzabcdefghijklmnopq',
    'stuvwxyzabcdefghijklmnopqr','tuvwxyzabcdefghijklmnopqrs',
    'uvwxyzabcdefghijklmnopqrst','vwxyzabcdefghijklmnopqrstu',
    'wxyzabcdefghijklmnopqrstuv','xyzabcdefghijklmnopqrstuvw',
    'yzabcdefghijklmnopqrstuvwx','zabcdefghijklmnopqrstuvwxy',]

l = []
n = []
o = []

for i in range(len(m)):
   l.append(alpha.index(m[i]))
   n.append(alpha.index(key[i]))
   o.append(list[alpha.index(m[i])][alpha.index(key[i])])



print("massage to encrypt is :",m)

print("key for encrypt is :",key)

print("Encrypted massage is:",o)

#print(n)
#print(l)
'''
s = []
t = []
u = []

for i in range(len(o)):
   s.append(alpha.index(o[i]))
```

```
print(s)

for i in range(len(key)):
  #print("hello")
  print(list[s[i]])
  t.append(list[s[i]].index(key[i]))
  u.append(alpha[t[i]])


#print(list[s[1]])

print(t)

print(u)


'''
```

7. HILL Cipher:

7.1 Assignment Problem:

Implement hill cipher.

7.2. Logic/methodology explanation:

Each letter is represented by a number modulo 26. Often the simple scheme A = 0, B = 1, ..., Z = 25 is used, but this is not an essential feature of the cipher. To encrypt a message, each block of n letters (considered as an n-component vector) is multiplied by an invertible $n \times n$ matrix, against modulus 26. To decrypt the message, each block is multiplied by the inverse of the matrix used for encryption.

The matrix used for encryption is the cipher key, and it should be chosen randomly from the set of invertible $n \times n$ matrices (modulo 26). The cipher can, of course, be adapted to an alphabet with any number of letters; all arithmetic just needs to be done modulo the number of letters instead of modulo 26.

7.3 Code:

```
import numpy as np



alpha = ['a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u','v','w','x','y','z']


m = input('entre the masss:')


key = 'gybnqkurp'


l = []
n = []


for i in range(len(key)):
  l.append(alpha.index(key[i]))
print(l)



b = np.array(l)
b.resize(3, 3) # new_shape parameter doesn't have to be a tuple
print(b)
```

```
for j in range(len(m)):
  n.append(alpha.index(m[j]))
print(n)



multiply = np.matmul(b, n)

print(multiply)

t = []
l = []

for i in range(len(multiply)):
  l.append(multiply[i]%26)
  t.append(alpha[multiply[i]%26])

print(l)
print(t)

inverse_of_b = np.linalg.inv(b)

print(inverse_of_b)

multiply = np.matmul(inverse_of_b, l)

print(multiply)
```

8. RC4:

8.1 Assignment Problem:

Implement rc4.

8.2 Logic/methodology explanation:

RC4 generates a pseudorandom stream of bits (a keystream). As with any stream cipher, these can be used for encryption by combining it with the plaintext using bitwise exclusive-or; decryption is performed the same way (since exclusive-or with given data is an involution). This is similar to the one-time pad except that generated *pseudorandom bits*, rather than a prepared stream, are used.

To generate the keystream, the cipher makes use of a secret internal state which consists of two parts:

The permutation is initialized with a variable length key, typically between 40 and 2048 bits, using the *key-scheduling* algorithm (KSA). Once this has been completed, the stream of bits is generated using the *pseudo-random generation algorithm* (PRGA).

7.3 Code

```
'''
RC4
Created By Punnet Kumar

'''
def swap(x, y):
  temp = x
  x = y
  y = temp

plain_text = input("enter the plain text : ")

p = []

for i in range(len(plain_text)):
  p.append((ord(plain_text[i])- 96))
```

```
print(p)



#p = [1,2,2,2]

k = [1,2,3,6]

s = [i for i in range(256)]

#print(s)

t = [k[i%len(k)] for i in range(len(s))]

#print(t)

ss = []


j = 0
for i in range(256):
  j = (j + s[i] + t[i])% 256


  temp = s[i]
  s[i] = s[j]
  s[j] = temp
  #print(j)


#print(s)


i = 0
j = 0
for i in range(len(p)):
  i = (i + 1)% 256
  j = (j + s[i])% 256
```

```
    swap(s[i], s[j])
    t = (s[i] + s[j])% 256
    k = s[t]
    #print(k)
    ss.append(k)

print(ss)

cipher = []
ccc = []

for i in range(len(p)):
    cipher.append((ss[i] ^ p[i]))
    ccc.append(chr(cipher[i]))



#print(ccc)
#ct=''
#ctext=ct.join(ccc)
#print('\n\n')
print('encrypted text is:',ccc)

decry = []
for i in range(len(p)):
    decry.append(chr(p[i]+96))

ct=''
ctext=ct.join(decry)
print('decrypted text is:',ctext)
#print(decry)




'''
for char in m:
```

```
    byte = ord(char)
    cipher1 = byte ^ Byte
    check.append(chr(cipher1))
    t2=''
    n=t2.join(check)
    print('reversed checked  text is:',n)

'''



'''
j = 0
    for i in range(256):
        if len(key) > 0:
            ko=key[i % len(key)]
            km= int(ko)
            j = (j + k[i] + km) % 256
            k[i], k[j] = k[j], k[i]
        else:
            j = (j + k[i]) % 256
'''#c = eval(input("c = "))
```

9. LFSR:

9.1. Assignment Problem:

- On each call to lfsr_calculate, you will shift the contents of the register 1 bit to the right.
- This shift is neither a logical shift or an arithmetic shift. On the left side, you will shift in a single bit equal to the Exclusive Or (XOR) On each call to lfsr_calculate, you will shift the contents of the register 1 bit to the right.

  This shift is neither a logical shift or an arithmetic shift. On the left side, you will shift in a single bit equal to the Exclusive Or (XOR) of the bits originally in position 11, 13, 14, and 16.The curved head-light shaped object is an XOR, which takes two inputs (a, b) and outputs a^b.If you implemented lfsr_calculate() correctly, it should output all 65535 positive 16-bit integers before cycling back to the starting number.of the bits originally in position 11, 13, 14, and 16.

- The curved head-light shaped object is an XOR, which takes two inputs (a, b) and outputs a^b.

- If you implemented lfsr_calculate() correctly, it should output all 65535 positive 16-bit integers before cycling back to the starting number.

9.2 Logic/methodology explanation:

In computing, a **linear-feedback shift register** (**LFSR**) is a shift register whose input bit is a linear function of its previous state.The most commonly used linear function of single bits is exclusive-or (XOR). Thus, an LFSR is most often a shift register whose input bit is driven by the XOR of some bits of the overall shift register value.
The initial value of the LFSR is called the seed, and because the operation of the register is deterministic, the stream of values produced by the register is completely determined by its current (or previous) state. Likewise, because the register has a finite number of possible states, it must eventually enter a repeating cycle. However, an LFSR with a well-chosen feedback function can produce a sequence of bits that appears random and has a very long cycle.

9.3 Code:

```
#n = input("enter the n :")


sid = input("enter the seed from lfsr with space betwwen each bit:")
```

```
se = sid.split()

print(se)

print(len(se))

print(len(se)-1)

print(se[len(se)-1])

#print(n)

#print(seed)

#[int(x) for x in "1 2 3 4 5".split()]

#print(seed[0])

#tag_postion = 1

#se1 = int(se[tag_postion])

#se2 = int(se[tag_postion - 1])

#print(len(seed))

#test = (tag_postion ^ se)

#print(test)

def leftshift(seed):
  tag_postion = 1
  se1 = int(seed[tag_postion])
  se2 = int(seed[tag_postion - 1])
  for i in range(len(seed)):
     if(i==(len(seed)-1)):
        seed[(len(seed)-1)]= (se1 ^ se2)
        #print(seed[(len(seed)-1)])
```

```
        else:
            seed[i] = seed[i+1]
            #print(seed[i])
    return(seed)


#while()
seprint = leftshift(se)

#tets  = ['0','1', '1', '1', '0']
#print(seprint)

test = se ke sath comparising

print(cmp(seprint,tets))

comparising two list

and while(!= list are not equar):
        test = seprint
        left(seprint)
#c = a << 2;       # 240 = 1111 0000
#print "Line 5 - Value of c is ", c
#a = 4
#c = a << 2
#d = a >> 2

#print(c)
#print(d)
```

10. RSA:

10.1 Assignment Problem:

Implement 256-bit RSA algorithm for encryption and decryption. Perform the encryption on the message " Attack at dawn".
Regarding key generation, note the following:

1. use e = 65537.

2. Use the *PrimeGenerator* function to generate values of p and q which must satisfy the following:
(a) p and q should not be equal.
(b) (p -1) and (q - 1) should be co-prime to e

3. Compute d. You may use the *multiplicative inverse* function

4. Use the *Mod-exponentiation* function discussed in the lecture to compute general modular exponentiation.

5. Perform the *encryption* and *decryption*

6. Breaking RSA encryption for small values of e, like 3:
In this scenario, a sender A sends the same message M to 3 different receivers using their respective public keys. All of the public keys have the same value of e, but different values of n. An attacker can intercept the three cipher texts and use the Chinese Remainder Theorem to calculate the value of M^3 mod N, where N is the product of the values of n. The attacker can then solve the cube-root to get the plaintext message M. Write a script that does the following:
1. Generates three sets of public and private keys with e = 3
2. Encrypts the given plaintext with each of the three public keys
3. Takes the three encrypted files and the public keys, and outputs the decrypted file

10.2 Logic/methodology explanation:

The RSA algorithm involves four steps: key generation, key distribution, encryption and decryption.
A basic principle behind RSA is the observation that it is practical to find three very large positive integers *e*, *d* and *n* such that with modular exponentiation for all integer *m* (with $0 \leq m < n$):
{\displaystyle (m^{e})^{d}\equiv m{\pmod {n}}} $(m^e)^d = m \pmod n$
and that even knowing *e* and *n* or even *m* it can be extremely difficult to find *d*.
In addition, for some operations it is convenient that the order of the two exponentiations can be changed and that this relation also implies:
{\displaystyle (m^{d})^{e}\equiv m{\pmod {n}}} $(m^d)^e = m \pmod n$
RSA involves a *public key* and a *private key*. The public key can be known by everyone, and it is used for encrypting messages. The intention is that messages encrypted with the

public key can only be decrypted in a reasonable amount of time by using the private key. The public key is represented by the integers *n* and *e*; and, the private key, by the integer *d*(although *n* is also used during the decryption process

10.3 Code:

```
import random

lower = eval(input("Enter the first Range"))

upper = eval(input("Enter the second Range"))

#260 - 600

primelist = []

def prime(a,b):
   for num in range(a,b + 1):
  # prime numbers are greater than 1
      if num > 1:
         for i in range(2,num):
            if (num % i) == 0:
               break
         else:
            #print(num)
            #primelist.extend(num)
            primelist.append(num)

prime(lower,upper)

print(primelist)

length_of_list = (len(primelist) - 1)

p = primelist[(random.randint(0, length_of_list))]

q = primelist[(random.randint(0, length_of_list))]

print("value of P : ",p)
```

```
print("value of Q : ",q)

n = p * q

fi_of_n = (p-1) * (q-1)


print("value of N : ",n)

print("value of Fi of N : ",fi_of_n)


def extened(r1, r2):
    t1 = 0
    t2 = 1
    while(r1!=1 and r2!=0):
        q = int(r1/r2)
        r = r1%r2
        t = t1-q*t2
        r1 = r2
        r2 = r
        t1 = t2
        t2 = t

    while(t1<0):
        t1 = t1+fi_of_n
    return(t1)
    #print("Value of d : ",t1)
    #d = t1

e = 65537

print("value of E : ",e)

d = extened(fi_of_n,e)

print("value of D : ",d)

###==================================###
```

```
        ###Encryption###
###=================================###
m = input("enter the massage : ")
mi = []
cipher = []
#hexenc = []

for i in range(len(m)):
  mi.append(ord(m[i]))
  cipher.append((pow(mi[i],e))%n)
  #hexenc.append

print(mi)
print(cipher)




###=================================###
        ###Decryption###
###=================================###
decrypt = []
plaintext = []
hextext = []
for i in range(len(cipher)):
  #mi.append(ord(m[i]))
  decrypt.append((pow(cipher[i],d))%n)
  plaintext.append(chr(decrypt[i]))
  hextext.append(hex(decrypt[i]))
print(decrypt)
print("Decrypted massage in hex : ",hextext)
str1 = ''.join(plaintext)
print("Decrypted Massage is : ", str1)
```

11. RABIN MILLER:

11.1 Assignment Problem:

Implement rabin miller.

11.2 Logic explanation:

bool isPrime(int n, int k)

1) Handle base cases for n < 3

2) If n is even, return false.

3) Find an odd number d such that n-1 can be written as $d*2^r$.

Note that since n is odd, (n-1) must be even and r must be greater than 0.

4) Do following k times

if (millerTest(n, d) == false)

return false

5) Return true.

11.3 Code

```
import random


def millerTest(n,d):
    a = random.randint(2,n-2)
    #print(a)
    x = pow(a,d) % n
    if(x==1 or x == (n-1)):
        return True



n = eval(input("enter the n : "))
```

```
#print(n)
k = eval(input("enter the k : "))




if(n%2==0):
  print(n," is not prime")


# 1) Compute d and r such that d*2r = n-1
else:
  r = 0
  m = (n-1)
  while(m%2==0):
    m = m/2
    r += 1
  d=m
  print("d = ",d)
  print("r = ",r)


  for i in range(k):
    if(millerTest(n,d)==False):
      print("not prime")
    else:
      print("maybe prime")
```

12. A power B mod c

code:

```
import math
import numpy as np

#a = eval(input("a = "))

#b = eval(input("b = "))

#c = eval(input("c = "))

a,b,c = 5,117,19

#a=117
bin = [int(x) for x in bin(b)[2:]]

#bin = [1,0,0,0,0,0,0,0,0]
print(bin)

sim = bin[::-1]
```

```
print(sim)

l = [a % c]

index = len(sim)

for i in range(1,index):
  #l.append(i)
  l.append((l[i-1] * l[i-1]) % c)


print(l)

temp = 1

fi = []

'''
for i in range(index,0,-1):
  if(bin[i-1] == 1):
     #temp = temp * l[i-1]
     fi.append(l[i-1])

print(fi)
'''

for i in range(index):
  if(sim[i] == 1):
     temp = temp * l[i]
     #fi.append(l[i])

print(temp)


for i in range(0,len(fi),1):
  temp = temp * fi[i]
```

```
print(temp)
print(temp%c)
```