MA 409

(Laboratory Component)

Number Theory and Introduction to Cryptography

(In Python)

Lab Manual

**Number Theory and Cryptography Lab**

**Note:** At least any five from each section

**Number Theory**

1. Implement Euclidean Algorithm.
2. Implement Extended Euclidean Algorithm.
3. Implement Modulo Inverse
4. Implement modular exponentiation (square and multiply)
5. Implement Miller-Rabin Algorithm for testing for primarily
6. Write a simple four-function calculator in GF(24).
7. Write a simple four-function calculator in GF(26).
8. Implement Euclidean Algorithm to find gcd of polynomials (with coefficients in a field).
9. Implement Extended Euclidean Algorithm to find gcd of polynomials (with coefficients in a field).
10. Implement Chinese Remainder Theorem

**Cryptography**

1. To implement a program for encrypting a plain text and decrypting a cipher text using Caesar Cipher (shift cipher) substitution technique
2. To implement a program to encrypt a plain text and decrypt a cipher text using play fair Cipher substitution technique.
3. To develop a program to encrypt and decrypt using the Hill cipher substitution technique
4. To develop a program to implement encryption and decryption using vigenere cipher substitution technique
5. To develop a program for implementing encryption and decryption using rail fence transposition technique.
6. Develop a program to implement RSA algorithm for encryption and decryption.
7. Develop a program to implement Diffie Hellman Key Exchange Algorithm for encryption and Decryption
8. Develop a program to implement Secure Hash Algorithm (SHA-1)
9. To write a program to implement the digital signature scheme in java

**Introduction**

**Practice python:**

1. Write a Python program to display the current date and time.
2. Write a Python program which accepts the radius of a circle from the user and compute the area.
3. Write a Python program which accepts the user's first and last name and print them in reverse order with a space between them.
4. Write a Python program which accepts a sequence of comma-separated numbers from user and generate a list and a tuple with those numbers.   
   Sample data : 3, 5, 7, 23.

**NT Lab-1**

Euclid's algorithm, slow implementation:

def gcd(a, b):

  assert a >= 0 and b >= 0 and a + b > 0

  while a > 0 and b > 0:

    if a >= b:

      a = a - b

    else:

      b = b - a

  return max(a, b)

print(gcd(24, 16))

print(gcd(790933790547, 1849639579327))

# The following call would take too long

#print(gcd(790933790548, 2))

Euclid's algorithm, fast implementation:

def gcd(a, b):

  assert a >= 0 and b >= 0 and a + b > 0

  while a > 0 and b > 0:

    if a >= b:

      a = a % b

    else:

      b = b % a

  return max(a, b)

print(gcd(24, 16))

print(gcd(790933790547, 1849639579327))

print(gcd(790933790548, 2))

**NT Lab-2**

## Extended Euclid's Algorithm: Code

The function extended\_gcd(a,b) returns three values: the greatest common divisor of a and b: d=gcd(a,b); and two numbers x and y such that d = ax + by

def extended\_gcd(a, b):

  assert a >= b and b >= 0 and a + b > 0

  if b == 0:

    d, x, y = a, 1, 0

  else:

    (d, p, q) = extended\_gcd(b, a % b)

    x = q

    y = p - q \* (a // b)

  assert a % d == 0 and b % d == 0

  assert d == a \* x + b \* y

  return (d, x, y)

extended\_gcd(10,6)

extended\_gcd(7,5)

extended\_gcd(391,299)

extended\_gcd(239,201)

**NT Lab-3**

# Iterative Python 3 program to find modular inverse using extended

# Euclid algorithm

# Returns modulo inverse of a with respect to m using extended Euclid

# Algorithm Assumption: a and m are coprimes, i.e., gcd(a, m) = 1

**def** modInverse(a, m) :

    m0 **=** m

    y **=** 0

    x **=** 1

**if** (m **==** 1) :

**return** 0

**while** (a > 1) :

        # q is quotient

        q **=** a **//** m

        t **=** m

        # m is remainder now, process

        # same as Euclid's algo

        m **=** a **%** m

        a **=** t

        t **=** y

        # Update x and y

        y **=** x **-** q **\*** y

        x **=** t

    # Make x positive

**if** (x < 0) :

        x **=** x **+** m0

**return** x

# Driver program to test above function

a **=** 3

m **=** 11

print("Modular multiplicative inverse is",

       modInverse(a, m))

**NT Lab-4**

# Iterative Function to calculate (x^y) in O(log y)

def power(x, y):

    # Initialize result

    res = 1

    while (y > 0):

        # If y is odd, multiply x with result

        if ((y & 1) != 0):

            res = res \* x

        # y must be even now

        y = y >> 1 # y = y/2

        x = x \* x  # Change x to x^2

    return res

--------------------------------------------------------------------------------------------------

# Iterative Python3 program

# to compute modular power

# Iterative Function to calculate

# (x^y)%p in O(log y)

def power(x, y, p) :

    res = 1     # Initialize result

    # Update x if it is more

    # than or equal to p

    x = x % p

    if (x == 0) :

        return 0

    while (y > 0) :

        # If y is odd, multiply

        # x with result

        if ((y & 1) == 1) :

            res = (res \* x) % p

        # y must be even now

        y = y >> 1      # y = y/2

        x = (x \* x) % p

    return res

# Driver Code

x = 2; y = 5; p = 13

print("Power is ", power(x, y, p))

NT-5 Miller-Rabin

First Method

# Python3 program Miller-Rabin primality test

import random

# Utility function to do modular exponentiation. It returns (x^y) % p

def power(x, y, p):

    # Initialize result

    res = 1;

    # Update x if it is more than or equal to p

    x = x % p;

    while (y > 0):

        # If y is odd, multiply x with result

        if (y & 1):

            res = (res \* x) % p;

        # y must be even now

        y = y>>1; # y = y/2

        x = (x \* x) % p;

    return res;

# This function is called for all k trials.

# It returns false if n is composite and returns false if n is

# probably prime. d is an odd number such that d\*2<sup>r</sup> = n-1

# for some r >= 1

def miillerTest(d, n):

    # Pick a random number in [2..n-2]

    # Corner cases make sure that n > 4

    a = 2 + random.randint(1, n - 4);

    # Compute a^d % n

    x = power(a, d, n);

    if (x == 1 or x == n - 1):

        return True;

    # Keep squaring x while one of the following doesn't happen

    # (i) d does not reach n-1

    # (ii) (x^2) % n is not 1

    # (iii) (x^2) % n is not n-1

    while (d != n - 1):

        x = (x \* x) % n;

        d \*= 2;

        if (x == 1):

            return False;

        if (x == n - 1):

            return True;

    # Return composite

    return False;

# It returns false if n is composite and returns true if n

# is probably prime. k is an input parameter that determines

# accuracy level. Higher value of k indicates more accuracy.

def isPrime( n, k):

    # Corner cases

    if (n <= 1 or n == 4):

        return False;

    if (n <= 3):

        return True;

    # Find r such that n =

    # 2^d \* r + 1 for some r >= 1

    d = n - 1;

    while (d % 2 == 0):

        d //= 2;

    # Iterate given nber of 'k' times

    for i in range(k):

        if (miillerTest(d, n) == False):

            return False;

    return True;

# Driver Code

# Number of iterations

k = 4;

print("All primes smaller than 100: ");

for n in range(1,100):

    if (isPrime(n, k)):

        print(n , end=" ");

Second Method

**import** **random**  
   
**def** is\_Prime**(**n**)**:  
 """  
 Miller-Rabin primality test.  
   
 A return value of False means n is certainly not prime. A return value of  
 True means n is very likely a prime.  
 """  
 **if** n!=**int(**n**)**:  
 **return** **False**  
 n=**int(**n**)**  
 *#Miller-Rabin test for prime*  
 **if** n==**0** **or** n==**1** **or** n==**4** **or** n==**6** **or** n==**8** **or** n==**9**:  
 **return** **False**  
   
 **if** n==**2** **or** n==**3** **or** n==**5** **or** n==**7**:  
 **return** **True**  
 s = **0**  
 d = n-**1**  
 **while** d%**2**==**0**:  
 d>>=**1**  
 s+=**1**  
 **assert(2**\*\*s \* d == n-**1)**  
   
 **def** trial\_composite**(**a**)**:  
 **if** **pow(**a, d, n**)** == **1**:  
 **return** **False**  
 **for** i **in** **range(**s**)**:  
 **if** **pow(**a, **2**\*\*i \* d, n**)** == n-**1**:  
 **return** **False**  
 **return** **True**   
   
 **for** i **in** **range(8)**:*#number of trials*   
 a = **random**.**randrange(2**, n**)**  
 **if** trial\_composite**(**a**)**:  
 **return** **False**  
   
 **return** **True**

NT Lab 6

Write a simple four-function calculator in GF(24).

NT Lab 7

Write a simple four-function calculator in GF(26).

NT Lab 8

Implement Euclidean Algorithm to find gcd of polynomials (with coefficients in a field).

NT Lab 9

Implement Extended Euclidean Algorithm to find gcd of polynomials (with coefficients in a field).

NT Lab 10

Implement Chinese Remainder Theorem

# Python 2.x program to combine modular equations

# using Chinese Remainder Theorem

# function that implements Extended euclidean

# algorithm

def extended\_euclidean(a, b):

    if a == 0:

        return (b, 0, 1)

    else:

        g, y, x = extended\_euclidean(b % a, a)

        return (g, x - (b // a) \* y, y)

# modular inverse driver function

def modinv(a, m):

    g, x, y = extended\_euclidean(a, m)

    return x % m

# function implementing Chinese remainder theorem

# list m contains all the modulii

# list x contains the remainders of the equations

def crt(m, x):

    # We run this loop while the list of

    # remainders has length greater than 1

    while True:

        # temp1 will contain the new value

        # of A. which is calculated according

        # to the equation m1' \* m1 \* x0 + m0'

        # \* m0 \* x1

        temp1 = modinv(m[1],m[0]) \* x[0] \* m[1] + \

                modinv(m[0],m[1]) \* x[1] \* m[0]

        # temp2 contains the value of the modulus

        # in the new equation, which will be the

        # product of the modulii of the two

        # equations that we are combining

        temp2 = m[0] \* m[1]

        # we then remove the first two elements

        # from the list of remainders, and replace

        # it with the remainder value, which will

        # be temp1 % temp2

        x.remove(x[0])

        x.remove(x[0])

        x = [temp1 % temp2] + x

        # we then remove the first two values from

        # the list of modulii as we no longer require

        # them and simply replace them with the new

        # modulii that  we calculated

        m.remove(m[0])

        m.remove(m[0])

        m = [temp2] + m

        # once the list has only one element left,

        # we can break as it will only  contain

        # the value of our final remainder

        if len(x) == 1:

            break

    # returns the remainder of the final equation

    return x[0]

# driver segment

m = [4, 7, 9, 11]

x = [3, 4, 1, 0]

print crt(m, x)

CRYP Lab – 1

To implement a program for encrypting a plain text and decrypting a cipher text using Caesar Cipher (shift cipher) substitution technique

key = 'abcdefghijklmnopqrstuvwxyz'

def enc\_caesar(n, plaintext):

result = ''

for l in plaintext.lower():

try:

i = (key.index(l) + n) % 26

result += key[i]

except ValueError:

result += l

return result.lower()

plaintext = 'We hold these truths to be self-evident, that all men are created equal.'

ciphertext = enc\_caesar(3, plaintext)

print (ciphertext)

key = 'abcdefghijklmnopqrstuvwxyz'

def dec\_caesar(n, ciphertext):

result = ''

for l in ciphertext:

try:

i = (key.index(l) - n) % 26

result += key[i]

except ValueError:

result += l

return result

ciphertext = 'zh krog wkhvh wuxwkv wr eh vhoi-hylghqw, wkdw doo phq duh fuhdwhg htxdo.'

plaintext = dec\_caesar(3, ciphertext)

print (plaintext)

CRYP Lab – 2

To implement a program to encrypt a plain text and decrypt a cipher text using play fair Cipher substitution technique.

def convertPlainTextToDiagraphs (plainText):

# append X if Two letters are being repeated

for s in range(0,len(plainText)+1,2):

if s<len(plainText)-1:

if plainText[s]==plainText[s+1]:

plainText=plainText[:s+1]+'X'+plainText[s+1:]

# append X if the total letters are odd, to make plaintext even

if len(plainText)%2 != 0:

plainText = plainText[:]+'X'

return plainText

def generateKeyMatrix (key):

# Intially Create 5X5 matrix with all values as 0

# [

# [0, 0, 0, 0, 0],

# [0, 0, 0, 0, 0],

# [0, 0, 0, 0, 0],

# [0, 0, 0, 0, 0],

# [0, 0, 0, 0, 0]

# ]

matrix\_5x5 = [[0 for i in range (5)] for j in range(5)]

simpleKeyArr = []

"""

Generate SimpleKeyArray with key from user Input

with following below condition:

1. Character Should not be repeated again

2. Replacing J as I (as per rule of playfair cipher)

"""

for c in key:

if c not in simpleKeyArr:

if c == 'J':

simpleKeyArr.append('I')

else:

simpleKeyArr.append(c)

"""

Fill the remaining SimpleKeyArray with rest of unused letters from english alphabets

"""

is\_I\_exist = "I" in simpleKeyArr

# A-Z's ASCII Value lies between 65 to 90 but as range's second parameter excludes that value we will use 65 to 91

for i in range(65,91):

if chr(i) not in simpleKeyArr:

# I = 73

# J = 74

# We want I in simpleKeyArr not J

if i==73 and not is\_I\_exist:

simpleKeyArr.append("I")

is\_I\_exist = True

elif i==73 or i==74 and is\_I\_exist:

pass

else:

simpleKeyArr.append(chr(i))

"""

Now map simpleKeyArr to matrix\_5x5

"""

index = 0

for i in range(0,5):

for j in range(0,5):

matrix\_5x5[i][j] = simpleKeyArr[index]

index+=1

return matrix\_5x5

def indexLocator (char,cipherKeyMatrix):

indexOfChar = []

# convert the character value from J to I

if char=="J":

char = "I"

for i,j in enumerate(cipherKeyMatrix):

# enumerate will return object like this:

# [

# (0, ['K', 'A', 'R', 'E', 'N']),

# (1, ['D', 'B', 'C', 'F', 'G']),

# (2, ['H', 'I', 'L', 'M', 'O']),

# (3, ['P', 'Q', 'S', 'T', 'U']),

# (4, ['V', 'W', 'X', 'Y', 'Z'])

# ]

# i,j will map to tupels of above array

# j refers to inside matrix => ['K', 'A', 'R', 'E', 'N'],

for k,l in enumerate(j):

# again enumerate will return object that look like this in first iteration:

# [(0,'K'),(1,'A'),(2,'R'),(3,'E'),(4,'N')]

# k,l will map to tupels of above array

if char == l:

indexOfChar.append(i) #add 1st dimension of 5X5 matrix => i.e., indexOfChar = [i]

indexOfChar.append(k) #add 2nd dimension of 5X5 matrix => i.e., indexOfChar = [i,k]

return indexOfChar

# Now with the help of indexOfChar = [i,k] we can pretty much locate every element,

# inside our 5X5 matrix like this => cipherKeyMatrix[i][k]

def encryption (plainText,key):

cipherText = []

# 1. Generate Key Matrix

keyMatrix = generateKeyMatrix(key)

# 2. Encrypt According to Rules of playfair cipher

i = 0

while i < len(plainText):

# 2.1 calculate two grouped characters indexes from keyMatrix

n1 = indexLocator(plainText[i],keyMatrix)

n2 = indexLocator(plainText[i+1],keyMatrix)

# 2.2 if same column then look in below row so

# format is [row,col]

# now to see below row => increase the row in both item

# (n1[0]+1,n1[1]) => (3+1,1) => (4,1)

# (n2[0]+1,n2[1]) => (4+1,1) => (5,1)

# but in our matrix we have 0 to 4 indexes only

# so to make value bound under 0 to 4 we will do %5

# i.e.,

# (n1[0]+1 % 5,n1[1])

# (n2[0]+1 % 5,n2[1])

if n1[1] == n2[1]:

i1 = (n1[0] + 1) % 5

j1 = n1[1]

i2 = (n2[0] + 1) % 5

j2 = n2[1]

cipherText.append(keyMatrix[i1][j1])

cipherText.append(keyMatrix[i2][j2])

cipherText.append(", ")

# same row

elif n1[0]==n2[0]:

i1= n1[0]

j1= (n1[1] + 1) % 5

i2= n2[0]

j2= (n2[1] + 1) % 5

cipherText.append(keyMatrix[i1][j1])

cipherText.append(keyMatrix[i2][j2])

cipherText.append(", ")

# if making rectangle then

# [4,3] [1,2] => [4,2] [3,1]

# exchange columns of both value

else:

i1 = n1[0]

j1 = n1[1]

i2 = n2[0]

j2 = n2[1]

cipherText.append(keyMatrix[i1][j2])

cipherText.append(keyMatrix[i2][j1])

cipherText.append(", ")

i += 2

return cipherText

def main():

#Getting user inputs Key (to make the 5x5 char matrix) and Plain Text (Message that is to be encripted)

key = input("Enter key: ").replace(" ","").upper()

plainText =input("Plain Text: ").replace(" ","").upper()

convertedPlainText = convertPlainTextToDiagraphs(plainText)

cipherText = " ".join(encryption(convertedPlainText,key))

print(cipherText)

if \_\_name\_\_ == "\_\_main\_\_":

main()

CRYP Lab – 3

To develop a program to encrypt and decrypt using the Hill cipher substitution technique

|  |
| --- |
|  |
|  |  |
|  | import numpy as np |
|  | def encrypt(msg): |
|  | # Replace spaces with nothing |
|  | msg = msg.replace(" ", "") |
|  | # Ask for keyword and get encryption matrix |
|  | C = make\_key() |
|  | # Append zero if the messsage isn't divisble by 2 |
|  | len\_check = len(msg) % 2 == 0 |
|  | if not len\_check: |
|  | msg += "0" |
|  | # Populate message matrix |
|  | P = create\_matrix\_of\_integers\_from\_string(msg) |
|  | # Calculate length of the message |
|  | msg\_len = int(len(msg) / 2) |
|  | # Calculate P \* C |
|  | encrypted\_msg = "" |
|  | for i in range(msg\_len): |
|  | # Dot product |
|  | row\_0 = P[0][i] \* C[0][0] + P[1][i] \* C[0][1] |
|  | # Modulate and add 65 to get back to the A-Z range in ascii |
|  | integer = int(row\_0 % 26 + 65) |
|  | # Change back to chr type and add to text |
|  | encrypted\_msg += chr(integer) |
|  | # Repeat for the second column |
|  | row\_1 = P[0][i] \* C[1][0] + P[1][i] \* C[1][1] |
|  | integer = int(row\_1 % 26 + 65) |
|  | encrypted\_msg += chr(integer) |
|  | return encrypted\_msg |
|  |  |
|  | def decrypt(encrypted\_msg): |
|  | # Ask for keyword and get encryption matrix |
|  | C = make\_key() |
|  | # Inverse matrix |
|  | determinant = C[0][0] \* C[1][1] - C[0][1] \* C[1][0] |
|  | determinant = determinant % 26 |
|  | multiplicative\_inverse = find\_multiplicative\_inverse(determinant) |
|  | C\_inverse = C |
|  | # Swap a <-> d |
|  | C\_inverse[0][0], C\_inverse[1][1] = C\_inverse[1, 1], C\_inverse[0, 0] |
|  | # Replace |
|  | C[0][1] \*= -1 |
|  | C[1][0] \*= -1 |
|  | for row in range(2): |
|  | for column in range(2): |
|  | C\_inverse[row][column] \*= multiplicative\_inverse |
|  | C\_inverse[row][column] = C\_inverse[row][column] % 26 |
|  |  |
|  | P = create\_matrix\_of\_integers\_from\_string(encrypted\_msg) |
|  | msg\_len = int(len(encrypted\_msg) / 2) |
|  | decrypted\_msg = "" |
|  | for i in range(msg\_len): |
|  | # Dot product |
|  | column\_0 = P[0][i] \* C\_inverse[0][0] + P[1][i] \* C\_inverse[0][1] |
|  | # Modulate and add 65 to get back to the A-Z range in ascii |
|  | integer = int(column\_0 % 26 + 65) |
|  | # Change back to chr type and add to text |
|  | decrypted\_msg += chr(integer) |
|  | # Repeat for the second column |
|  | column\_1 = P[0][i] \* C\_inverse[1][0] + P[1][i] \* C\_inverse[1][1] |
|  | integer = int(column\_1 % 26 + 65) |
|  | decrypted\_msg += chr(integer) |
|  | if decrypted\_msg[-1] == "0": |
|  | decrypted\_msg = decrypted\_msg[:-1] |
|  | return decrypted\_msg |
|  |  |
|  | def find\_multiplicative\_inverse(determinant): |
|  | multiplicative\_inverse = -1 |
|  | for i in range(26): |
|  | inverse = determinant \* i |
|  | if inverse % 26 == 1: |
|  | multiplicative\_inverse = i |
|  | break |
|  | return multiplicative\_inverse |
|  |  |
|  |  |
|  | def make\_key(): |
|  | # Make sure cipher determinant is relatively prime to 26 and only a/A - z/Z are given |
|  | determinant = 0 |
|  | C = None |
|  | while True: |
|  | cipher = input("Input 4 letter cipher: ") |
|  | C = create\_matrix\_of\_integers\_from\_string(cipher) |
|  | determinant = C[0][0] \* C[1][1] - C[0][1] \* C[1][0] |
|  | determinant = determinant % 26 |
|  | inverse\_element = find\_multiplicative\_inverse(determinant) |
|  | if inverse\_element == -1: |
|  | print("Determinant is not relatively prime to 26, uninvertible key") |
|  | elif np.amax(C) > 26 and np.amin(C) < 0: |
|  | print("Only a-z characters are accepted") |
|  | print(np.amax(C), np.amin(C)) |
|  | else: |
|  | break |
|  | return C |
|  |  |
|  | def create\_matrix\_of\_integers\_from\_string(string): |
|  | # Map string to a list of integers a/A <-> 0, b/B <-> 1 ... z/Z <-> 25 |
|  | integers = [chr\_to\_int(c) for c in string] |
|  | length = len(integers) |
|  | M = np.zeros((2, int(length / 2)), dtype=np.int32) |
|  | iterator = 0 |
|  | for column in range(int(length / 2)): |
|  | for row in range(2): |
|  | M[row][column] = integers[iterator] |
|  | iterator += 1 |
|  | return M |
|  |  |
|  | def chr\_to\_int(char): |
|  | # Uppercase the char to get into range 65-90 in ascii table |
|  | char = char.upper() |
|  | # Cast chr to int and subtract 65 to get 0-25 |
|  | integer = ord(char) - 65 |
|  | return integer |
|  |  |
|  | if \_\_name\_\_ == "\_\_main\_\_": |
|  | msg = input("Message: ") |
|  | encrypted\_msg = encrypt(msg) |
|  | print(encrypted\_msg) |
|  | decrypted\_msg = decrypt(encrypted\_msg) |
|  | print(decrypted\_msg) |

CRYP Lab – 4

To develop a program to implement encryption and decryption using vigenere cipher substitution technique

**Plaintext: We hold these truths to be self-evident, that all men are created equal.**

**Ciphertext: zi jzlu tamgr wvwehj th js fhph-pvzdxvh, gkev llc mxv oeh gtpakew mehdp.**

def key\_vigenere(key):

keyArray = []

for i in range(0,len(key)):

keyElement = ord(key[i]) – 65

keyArray.append(keyElement)

return keyArray

secretKey = 'DECLARATION'

key = key\_vigenere(secretKey)

print(key)

def shiftEnc(c, n):

return chr(((ord(c) - ord('A') + n) % 26) + ord('a'))

def enc\_vigenere(plainttext, key):

secret = "".join([shiftEnc(plainttext[i], key[i % len(key)]) for i in range(len(plainttext))])

return secret

secretKey = 'DECLARATION'

key = key\_vigenere(secretKey)

plaintext = 'ALL MEN ARE CREATED EQUAL'

ciphertext = enc\_vigenere(plaintext, key)

print(ciphertext)

def shiftDec(c, n):

c = c.upper()

return chr(((ord(c) - ord('A') - n) % 26) + ord('a'))

def dec\_vigenere(ciphertext, key):

plain = "".join([shiftDec(ciphertext[i], key[i % len(key)]) for i in range(len(ciphertext))])

return plain

secretKey = 'DECLARATION'

key = key\_vigenere(secretKey)

decoded = dec\_vigenere(ciphertext, key)

CRYP Lab – 5

To develop a program for implementing encryption and decryption using rail fence transposition technique.

Given a plain-text message and a numeric key, cipher/de-cipher the given text using Rail Fence algorithm.   
The rail fence cipher (also called a zigzag cipher) is a form of transposition cipher. It derives its name from the way in which it is encoded.   
**Examples:** 

**Encryption**

Input : "GeeksforGeeks "

Key = 3

Output : GsGsekfrek eoe

**Decryption**

Input : GsGsekfrek eoe

Key = 3

Output : "GeeksforGeeks "

**Encryption**

Input : "defend the east wall"

Key = 3

Output : dnhaweedtees alf tl

**Decryption**

Input : dnhaweedtees alf tl

Key = 3

Output : defend the east wall

**Encryption**

Input : "attack at once"

Key = 2

Output : atc toctaka ne

**Decryption**

Input : "atc toctaka ne"

Key = 2

Output : attack at once

# function to encrypt a message

def encryptRailFence(text, key):

    # create the matrix to cipher

    # plain text key = rows ,

    # length(text) = columns

    # filling the rail matrix

    # to distinguish filled

    # spaces from blank ones

    rail = [['\n' for i in range(len(text))]

                  for j in range(key)]

    # to find the direction

    dir\_down = False

    row, col = 0, 0

    for i in range(len(text)):

        # check the direction of flow

        # reverse the direction if we've just

        # filled the top or bottom rail

        if (row == 0) or (row == key - 1):

            dir\_down = not dir\_down

        # fill the corresponding alphabet

        rail[row][col] = text[i]

        col += 1

        # find the next row using

        # direction flag

        if dir\_down:

            row += 1

        else:

            row -= 1

    # now we can construct the cipher

    # using the rail matrix

    result = []

    for i in range(key):

        for j in range(len(text)):

            if rail[i][j] != '\n':

                result.append(rail[i][j])

    return("" . join(result))

# This function receives cipher-text

# and key and returns the original

# text after decryption

def decryptRailFence(cipher, key):

    # create the matrix to cipher

    # plain text key = rows ,

    # length(text) = columns

    # filling the rail matrix to

    # distinguish filled spaces

    # from blank ones

    rail = [['\n' for i in range(len(cipher))]

                  for j in range(key)]

    # to find the direction

    dir\_down = None

    row, col = 0, 0

    # mark the places with '\*'

    for i in range(len(cipher)):

        if row == 0:

            dir\_down = True

        if row == key - 1:

            dir\_down = False

        # place the marker

        rail[row][col] = '\*'

        col += 1

        # find the next row

        # using direction flag

        if dir\_down:

            row += 1

        else:

            row -= 1

    # now we can construct the

    # fill the rail matrix

    index = 0

    for i in range(key):

        for j in range(len(cipher)):

            if ((rail[i][j] == '\*') and

               (index < len(cipher))):

                rail[i][j] = cipher[index]

                index += 1

    # now read the matrix in

    # zig-zag manner to construct

    # the resultant text

    result = []

    row, col = 0, 0

    for i in range(len(cipher)):

        # check the direction of flow

        if row == 0:

            dir\_down = True

        if row == key-1:

            dir\_down = False

        # place the marker

        if (rail[row][col] != '\*'):

            result.append(rail[row][col])

            col += 1

        # find the next row using

        # direction flag

        if dir\_down:

            row += 1

        else:

            row -= 1

    return("".join(result))

# Driver code

if \_\_name\_\_ == "\_\_main\_\_":

    print(encryptRailFence("attack at once", 2))

    print(encryptRailFence("GeeksforGeeks ", 3))

    print(encryptRailFence("defend the east wall", 3))

    # Now decryption of the

    # same cipher-text

    print(decryptRailFence("GsGsekfrek eoe", 3))

    print(decryptRailFence("atc toctaka ne", 2))

    print(decryptRailFence("dnhaweedtees alf tl", 3))

CRYP Lab – 6

Develop a program to implement RSA algorithm for encryption and decryption.

|  |
| --- |
| import random |
|  | from hashlib import sha256 |
|  |  |
|  |  |
|  | def coprime(a, b): |
|  | while b != 0: |
|  | a, b = b, a % b |
|  | return a |
|  |  |
|  |  |
|  | def extended\_gcd(aa, bb): |
|  | lastremainder, remainder = abs(aa), abs(bb) |
|  | x, lastx, y, lasty = 0, 1, 1, 0 |
|  | while remainder: |
|  | lastremainder, (quotient, remainder) = remainder, divmod(lastremainder, remainder) |
|  | x, lastx = lastx - quotient\*x, x |
|  | y, lasty = lasty - quotient\*y, y |
|  | return lastremainder, lastx \* (-1 if aa < 0 else 1), lasty \* (-1 if bb < 0 else 1) |
|  |  |
|  | #Euclid's extended algorithm for finding the multiplicative inverse of two numbers |
|  | def modinv(a, m): |
|  | g, x, y = extended\_gcd(a, m) |
|  | if g != 1: |
|  | raise Exception('Modular inverse does not exist') |
|  | return x % m |
|  |  |
|  |  |
|  | def is\_prime(num): |
|  | if num == 2: |
|  | return True |
|  | if num < 2 or num % 2 == 0: |
|  | return False |
|  | for n in range(3, int(num\*\*0.5)+2, 2): |
|  | if num % n == 0: |
|  | return False |
|  | return True |
|  |  |
|  |  |
|  | def generate\_keypair(p, q): |
|  | if not (is\_prime(p) and is\_prime(q)): |
|  | raise ValueError('Both numbers must be prime.') |
|  | elif p == q: |
|  | raise ValueError('p and q cannot be equal') |
|  |  |
|  | n = p \* q |
|  |  |
|  | #Phi is the totient of n |
|  | phi = (p-1) \* (q-1) |
|  |  |
|  | #Choose an integer e such that e and phi(n) are coprime |
|  | e = random.randrange(1, phi) |
|  |  |
|  | #Use Euclid's Algorithm to verify that e and phi(n) are comprime |
|  | g = coprime(e, phi) |
|  |  |
|  | while g != 1: |
|  | e = random.randrange(1, phi) |
|  | g = coprime(e, phi) |
|  |  |
|  | #Use Extended Euclid's Algorithm to generate the private key |
|  | d = modinv(e, phi) |
|  |  |
|  | #Return public and private keypair |
|  | #Public key is (e, n) and private key is (d, n) |
|  | return ((e, n), (d, n)) |
|  |  |
|  |  |
|  | def encrypt(privatek, plaintext): |
|  | #Unpack the key into it's components |
|  | key, n = privatek |
|  |  |
|  | #Convert each letter in the plaintext to numbers based on the character using a^b mod m |
|  |  |
|  | numberRepr = [ord(char) for char in plaintext] |
|  | print("Number representation before encryption: ", numberRepr) |
|  | cipher = [pow(ord(char),key,n) for char in plaintext] |
|  |  |
|  | #Return the array of bytes |
|  | return cipher |
|  |  |
|  |  |
|  | def decrypt(publick, ciphertext): |
|  | #Unpack the key into its components |
|  | key, n = publick |
|  |  |
|  | #Generate the plaintext based on the ciphertext and key using a^b mod m |
|  | numberRepr = [pow(char, key, n) for char in ciphertext] |
|  | plain = [chr(pow(char, key, n)) for char in ciphertext] |
|  |  |
|  | print("Decrypted number representation is: ", numberRepr) |
|  |  |
|  | #Return the array of bytes as a string |
|  | return ''.join(plain) |
|  |  |
|  |  |
|  | def hashFunction(message): |
|  | hashed = sha256(message.encode("UTF-8")).hexdigest() |
|  | return hashed |
|  |  |
|  |  |
|  | def verify(receivedHashed, message): |
|  | ourHashed = hashFunction(message) |
|  | if receivedHashed == ourHashed: |
|  | print("Verification successful: ", ) |
|  | print(receivedHashed, " = ", ourHashed) |
|  | else: |
|  |  |
|  | print("Verification failed") |
|  | print(receivedHashed, " != ", ourHashed) |
|  |  |
|  |  |
|  | def main(): |
|  | p = int(input("Enter a prime number (17, 19, 23, etc): ")) |
|  | q = int(input("Enter another prime number (Not one you entered above): ")) |
|  | #p = 17 |
|  | #q=23 |
|  |  |
|  |  |
|  | print("Generating your public/private keypairs now . . .") |
|  | public, private = generate\_keypair(p, q) |
|  |  |
|  | print("Your public key is ", public ," and your private key is ", private) |
|  | message = input("Enter a message to encrypt with your private key: ") |
|  | print("") |
|  |  |
|  | hashed = hashFunction(message) |
|  |  |
|  | print("Encrypting message with private key ", private ," . . .") |
|  | encrypted\_msg = encrypt(private, hashed) |
|  | print("Your encrypted hashed message is: ") |
|  | print(''.join(map(lambda x: str(x), encrypted\_msg))) |
|  | #print(encrypted\_msg) |
|  |  |
|  | print("") |
|  | print("Decrypting message with public key ", public ," . . .") |
|  |  |
|  | decrypted\_msg = decrypt(public, encrypted\_msg) |
|  | print("Your decrypted message is:") |
|  | print(decrypted\_msg) |
|  |  |
|  | print("") |
|  | print("Verification process . . .") |
|  | verify(decrypted\_msg, message) |
|  |  |
|  | main() |

CRYP Lab – 7

Develop a program to implement Diffie Hellman Key Exchange Algorithm for encryption and Decryption

def power(a, b, p):

if (b == 1):

return a;

else:

return pow(a,b,p)

def main():

P = 0; G = 0; x = 0; a = x;

y = 0; b = 0;

ka = 0; kb = 0;

# Both the users will be agreed upon the public keys G and P

P = 23; # A prime number P is taken

print("The value of P:", P);

G = 9; # A primitive root for P, G is taken

print("The value of G:", G);

# Alice will choose the private key a

a = 4; # a is the chosen private key

print("The private key a for Alice:", a);

x = power(G, a, P); # gets the generated key

# Bob will choose the private key b

b = 3; # b is the chosen private key

print("The private key b for Bob:", b);

y = power(G, b, P); # gets the generated key

# Generating the secret key after the exchange of keys

ka = power(y, a, P); # Secret key for Alice

kb = power(x, b, P); # Secret key for Bob

print("Secret key for the Alice is:", ka);

print("Secret Key for the Bob is:", kb);

if \_\_name\_\_ == '\_\_main\_\_':

main()

CRYP Lab – 8

Develop a program to implement Secure Hash Algorithm (SHA-1)

***Input****: hello world****Output****: 2aae6c35c94fcfb415dbe95f408b9ce91ee846ed*

***Input****: GeeksForGeeks****Output****: addf120b430021c36c232c99ef8d926aea2acd6b*

**import** hashlib

str1**=**"Krish Naik1"

*### Process of Hashing*

*### First step is to encode*

*### Then Apply hashing Algorithms*

hashedval **=** hashlib**.**sha256(str1**.**encode())

print(hashedval)

*###convert this value to hexadecimal*

hashedval**.**hexdigest()

Code -2

import hashlib

h = hashlib.sha1()

print h.hexdigest()

#OUTPUT

#da39a3ee5e6b4b0d3255bfef95601890afd80709

h = hashlib.sha1("abc")

print h.hexdigest()

#OUTPUT

#a9993e364706816aba3e25717850c26c9cd0d89d

h = hashlib.new("sha1","abc")

print h.hexdigest()

CRYP Lab – 9

To write a program to implement the digital signature scheme

RSA algorithm is an asymmetric cryptography algorithm. Asymmetric actually means that it works on two different keys i.e. Public Key and Private Key. As the name describes that the Public Key is given to everyone and the Private key is kept private.

### Algorithm

**RSA Key Generation:**

* Choose two large prime numbers p and q
* Calculate n=p\*q
* Select public key e such that it is not a factor of (p-1)\*(q-1)
* Select private key d such that the following equation is true (d\*e)mod(p-1)(q-1)=1 or d is inverse of E in modulo (p-1)\*(q-1)

**RSA Digital Signature Scheme:** In RSA, d is private; e and n are public.

* Alice creates her digital signature using S=M^d mod n where M is the message
* Alice sends Message M and Signature S to Bob
* Bob computes M1=S^e mod n
* If M1=M then Bob accepts the data sent by Alice.

Below is the implementation.

# Function to find gcd

# of two numbers

def euclid(m, n):

    if n == 0:

        return m

    else:

        r = m % n

        return euclid(n, r)

# Program to find

# Multiplicative inverse

def exteuclid(a, b):

    r1 = a

    r2 = b

    s1 = int(1)

    s2 = int(0)

    t1 = int(0)

    t2 = int(1)

    while r2 > 0:

        q = r1//r2

        r = r1-q \* r2

        r1 = r2

        r2 = r

        s = s1-q \* s2

        s1 = s2

        s2 = s

        t = t1-q \* t2

        t1 = t2

        t2 = t

    if t1 < 0:

        t1 = t1 % a

    return (r1, t1)

# Enter two large prime

# numbers p and q

p = 823

q = 953

n = p \* q

Pn = (p-1)\*(q-1)

# Generate encryption key

# in range 1<e<Pn

key = []

for i in range(2, Pn):

    gcd = euclid(Pn, i)

    if gcd == 1:

        key.append(i)

# Select an encryption key

# from the above list

e = int(313)

# Obtain inverse of

# encryption key in Z\_Pn

r, d = exteuclid(Pn, e)

if r == 1:

    d = int(d)

    print("decryption key is: ", d)

else:

    print("Multiplicative inverse for\

    the given encryption key does not \

    exist. Choose a different encryption key ")

# Enter the message to be sent

M = 19070

# Signature is created by Alice

S = (M\*\*d) % n

# Alice sends M and S both to Bob

# Bob generates message M1 using the

# signature S, Alice's public key e

# and product n.

M1 = (S\*\*e) % n

# If M = M1 only then Bob accepts

# the message sent by Alice.

if M == M1:

    print("As M = M1, Accept the\

    message sent by Alice")

else:

    print("As M not equal to M1,\

    Do not accept the message\

    sent by Alice ")