**Experiment No. 5**

**Aim:** To implement RSA algorithm.

**Theory:**

The **RSA algorithm** is an asymmetric cryptography algorithm; this means that it uses a *public* key and a *private* key (i.e. two different, mathematically linked keys). As their names suggest, a public key is shared publicly, while a private key is secret and must not be shared with anyone.

The RSA algorithm is named after those who invented it in 1978: Ron Rivest, Adi Shamir, and Leonard Adleman.

The following illustration highlights how asymmetric cryptography works:

**How it works**

The RSA algorithm ensures that the keys, in the above illustration, are as secure as possible. The following steps highlight how it works:

**1. Generating the keys**

1. Select two large prime numbers,  and . The prime numbers need to be large so that they will be difficult for someone to figure out.
2. Calculate .
3. Calculate the ***totient*** function;
4. Select an integer , such that  is ***co-prime*** to  and 1 < e < . The pair of numbers  makes up the public key.

**Note:** Two integers are co-prime if the only positive integer that divides them is 1.

Calculate

 can be found using the ***Extended Euclidean algorithm***. The pair  makes up the private key.

**2. Encryption**

Given a plaintext , represented as a number, the cipher text  is calculated as:

.

**3. Decryption**

Using the private key , the plaintext can be found using:

**Example:**

Assuming receiver selects

Considering, e = 1901 which is indeed coprime to

If sender wants to send the Plain text using public key it will encrypted as follows:

This Cipher text will be received at receiver side and it will use its own private key to decrypt it:

**Implementation:**

import math

import random

#implementation of RSA key generation algorithm and encryption, decryption using same key

def gcd(a,b):

if not b:

return a

return gcd(b,a%b)

class RSA:

def \_\_init\_\_(self):

self.primes = []

self.generate\_totient()

self.generate\_keys()

def generate\_random\_prime(self):

self.primes.append(2)

for i in range(3,200):

if not i%2:

continue

Flag = True

for j in range(2,i):

if not i%j:

Flag = False

break

if Flag:

self.primes.append(i)

def select\_primes(self):

self.generate\_random\_prime()

p = random.sample(self.primes, 1)[0]

q = p

while(q == p):

q = random.sample(self.primes, 1)[0]

self.p = p

self.q = q

def generate\_totient(self):

self.select\_primes()

self.n = self.p \* self.q

self.totient\_n = (self.p-1)\*(self.q-1)

def multiplicative\_inverse(self):

a,m,x,y = self.e, self.totient\_n,1,0

while (a > 1):

q = a // m

t = m

m = a % m

a = t

t = y

y = x - q \* y

x = t

self.e\_inverse = x

def generate\_keys(self):

ls = [x for x in range(2,self.totient\_n)]

e = None

ls = random.sample(ls,len(ls))

for x in ls:

k = gcd(self.totient\_n,x)

if k == 1:

e = x

break

self.e = e

self.multiplicative\_inverse()

self.generate\_private\_key()

def generate\_private\_key(self):

self.d = self.e\_inverse % self.totient\_n

def fast\_expo(self,txt,key):

return (txt\*\*key)%self.n

def encryption(self,plain\_txt):

return self.fast\_expo(plain\_txt,self.e)

def decryption(self, cipher\_txt):

return self.fast\_expo(cipher\_txt, self.d)

g = RSA()

print(f"p = {g.p}, q = {g.q}, n = {g.n}, totient\_n = {g.totient\_n} \

public\_key = {g.e} public\_key\_inverse = {g.e\_inverse} private\_key = {g.d}\n")

encrypted = g.encryption(int(input(f"Enter value in [1,{g.n}] to encrypt it: ")))

print(f"Encrypted value = {encrypted}\n")

decrypted = g.decryption(int(input(f"Enter value for n = {g.n} to decrypt it: ")))

print(f"Decrypted value = {decrypted}")

**Output:**









