Lab - 5

Aim:
Write a Program to find the primitive roots for the Multiplicative Group with respect to Prime Modulus. Using that Implement Elgamal Cryptosystem.

Encryption:
r is a random integer from group $<Z^*_p ,X>$
$C1 = e1^r \bmod p$ $C2 = (e2^r * M) \bmod p$
Cipher Text: C_1, C_2. For message M.

Decryption:
$M = [\, C_2 * (C_1^{\,d})^{-1} \,] \bmod p$

Key Generation:
1. We select large prime number p randomly using Miller Rabin Primality Testing.
2. Select e_1 (Primitive root) of group $<Z^*_p ,X>$.
3. Select d to be a member of the group $G = <Z^*_p ,X>$ such that d belongs to [1,p-2].
4. Now, $e_2 = e_1^{\,d} \bmod p$.
5. Public Key: (e_1, e_2, p).
6. Private Key: d

To find e1 d mod p we are using Multiply and Square method

Code:

```
import random
import time

class EuclidianExtended:
    def __init__(self):
        pass

    def run(self, a, n):
        r1 = n
        r2 = a
        t1 = 0
        t2 = 1
        while r2 > 0:
            q = r1//r2
            r = r1 % r2
            r1 = r2
            r2 = r
```

```
            t = t1 - q*t2
            t1 = t2
            t2 = t
        gcd = r1
        inv = t1
        if(gcd < 0):
            gcd += n
        if(inv < 0):
            inv += n
        if(gcd != 1):
            inv = -1
        return gcd, inv

    def GetGcd(self, a, n):
        gcd, _ = self.run(a, n)
        return gcd

    def GetInv(self, a, n):
        _, inv = self.run(a, n)
        return inv

def PowNMod(base, power, mod):
    res = 1     # Initialize result
    # Update base if it is more
    # than or equal to mod
    base = base % mod
    if (base == 0):
        return 0
    while (power > 0):
        # If power is odd, multiply
        # base with result
        if ((power & 1) == 1):
            res = (res * base) % mod
        # power must be even now
        power = power >> 1      # power = power/2
        base = (base * base) % mod
    return res

import random
class PrimeNumbers:
    def PowNMod(self,base, power, mod):
        res = 1     # Initialize result
        # Update base if it is more
        # than or equal to mod
```

```
    base = base % mod
    if (base == 0):
        return 0
    while (power > 0):
        # If power is odd, multiply
        # base with result
        if ((power & 1) == 1):
            res = (res * base) % mod
        # power must be even now
        power = power >> 1      # power = power/2
        base = (base * base) % mod
    return res

def millerRabinTest(self,n):
    d=n-1
    while (d % 2 == 0):
        d //= 2
    # Miller Rabin Test
    a = 2 + random.randint(1, n - 4)
    #a^d % n
    x = self.PowNMod(a, d, n)

    if (x == 1 or x == n - 1):
        return True
    while (d != n - 1):
        x = (x * x) % n
        d *= 2

        if (x == 1):
            return False
        if (x == n - 1):
            return True
    return False

def isPrime(self,n,accuracyFactor):

    # Corner cases
    if (n <= 1 or n == 4):
        return False
    if (n <= 3):
        return True
    # Iterate given nber of 'k' times
    for _ in range(accuracyFactor):
        if (self.millerRabinTest( n) == False):
```

```
                return False
            return True

        def GetPrime(self,prime_len,accuracyFactor =3):
            #generating random prime numbers
            prime = random.randint(10**(prime_len-1),10**prime_len)
            while not self.isPrime(prime,accuracyFactor):
                prime = random.randint(10**(prime_len-1),10**prime_len)
            print("prime",prime)
            return prime


class PrimitiveRoots:
    def getPrimitiveRoots(self,p):
        EE = EuclidianExtended()
        #phi_p = self.phi(p)
        phi_p = 0
        #empty_set = set()
        sub_group_orders = []
        for a in range(p):
            if(EE.GetGcd(a,p) == 1):
                #calculating co-primes
                phi_p += 1
                temp_list = []
                for i in range(p):
                    temp = PowNMod(a,i,p)
                    if temp not in temp_list:
                        temp_list.append(temp)
                    else:
                        break
                sub_group_orders.append((a,len(temp_list)))
               # print(a)
            else:
                continue
        primitive_roots = [a for (a,o) in sub_group_orders if(o == phi_p) ]
        coprime_list = [a for (a,o) in sub_group_orders]
        return primitive_roots,coprime_list


class ElgamalCriptography:
    def __init__(self,base=0):
        self.base = base
        self.generateKeys()
```

```
def generateKeys(self):
    prime_length = int(input("How many digits of prime required ? : "))
    PN = PrimeNumbers()
    self.p = PN.GetPrime(prime_length,3)
    PR = PrimitiveRoots()
    start_time = time.time()
    self.primitive_roots,self.coprime_list = PR.getPrimitiveRoots(self.p)
    end_time = time.time()
    print("______________")
    print(f"time taken in primitive test : {end_time - start_time}")
    temp_index = random.randint(0,len(self.primitive_roots))
    e_1 = self.primitive_roots[temp_index]
    temp_index = random.randint(0,len(self.coprime_list))
    d = self.coprime_list[temp_index]
    while( d<1 or d>(self.p-1)):
        print("d",d)
        d = self.coprime_list[temp_index]
    e_2 = PowNMod(e_1,d,self.p)
    self.publiC_key = (e_1,e_2,self.p)
    self.private_key = d
    print("keys Generated")

def encrypt(self,plain_text):
    cipherList = []
    base = self.base
    print("Encrypting text")
    for pChar in plain_text:
        M = ord(pChar) - base
        temp_index = random.randint(0,len(self.coprime_list))
        r = self.coprime_list[temp_index]
        C_1 = PowNMod(self.publiC_key[0],r,self.p)
        C_2 = (PowNMod(self.publiC_key[1],r,self.p) * M )%self.p
        cipherList.append((C_1,C_2))
    return cipherList

def decrypt(self,cipher_text):
    EE = EuclidianExtended()
    base=self.base
    plainText = ""
    for cChar in cipher_text:
        C_1,C_2 = cChar
        C_1_inv = EE.GetInv(PowNMod(C_1, self.private_key, self.p),self.p)
        plainChar = (C_2 * C_1_inv) %self.p
        plainText += chr(base+plainChar)
```

```
        return plainText
    def test(self):
        plain_text = "How are you"
        cipher = self.encrypt(plain_text)
        print("stage2")
        decrypted_text = self.decrypt(cipher)
        print(f"plain Text : {plain_text}")
        print(f"Public Key : {self.publiC_key}")
        print(f"private key : {self.private_key}")
        print(f"cipher text : {cipher}")
        print(f"decrypted Text : {decrypted_text}")
if __name__ == "__main__":
    EC = ElgamalCriptography(0)
    EC.test()
```

Output:

```
How many digits of prime required ? : 3
prime 283
------------------------------
Time taken in primitive test : 0.28627872467041016
Keys Generated
Encrypting text

Plain Text : How are you
Public Key : (147, 223, 283)
Private key : 183
cipher text : [(221, 137), (281, 253), (43, 83), (204, 53), (159, 28), (232, 228), (277, 183), (27, 134), (38, 129), (181, 60),
(265, 91)]
Decrypted Text : How are you
```