Assignment-11

Q1. Write a program to implement the RSA signature.

A:

Pseudocode and Explanation –

This code implements the RSA algorithm for digital signature creation and verification. It begins by prompting the user for two prime numbers p and q, which are used to compute the public and private keys. The public key is generated using n = p * q and a random exponent e such that gcd(e, phi(n)) = 1, where phi(n) is the Euler's totient function of n. The private key is derived by calculating the modular inverse of e with respect to phi(n). To sign a message, each character is raised to the power of the private key exponent d modulo n. The signature is then verified by raising each element of the signature to the power of the public key exponent e modulo n and checking if it matches the original message's ASCII values. If all characters are verified successfully, the signature is considered valid, ensuring the integrity and authenticity of the message. The program allows the user to generate key pairs, sign a message, and verify the signature using RSA encryption.

Code –

```
import random
import math

def is_prime(num):
    """Check if a number is prime."""
    if num < 2:
        return False
    for i in range(2, int(math.sqrt(num)) + 1):
        if num % i == 0:
            return False
    return True

def gcd(a, b):
    """Calculate the greatest common divisor of two numbers."""
    while b!= 0:
        a, b = b, a % b
    return a
```

```python
def multiplicative_inverse(e, phi):
    """Calculate the multiplicative inverse of e modulo phi."""
    def extended_gcd(a, b):
        if a == 0:
            return b, 0, 1
        else:
            gcd, x, y = extended_gcd(b % a, a)
            return gcd, y - (b // a) * x, x

    gcd, x, _ = extended_gcd(e, phi)
    if gcd!= 1:
        raise ValueError("e and phi are not coprime")
    return x % phi

def generate_keypair(p, q):
    """Generate a public and private key pair."""
    n = p * q
    phi = (p - 1) * (q - 1)


    e = random.randrange(1, phi)
    while gcd(e, phi)!= 1:
        e = random.randrange(1, phi)


    d = multiplicative_inverse(e, phi)


    return ((e, n), (d, n))

def sign_message(message, private_key):
    """Sign a message using the private key."""
    d, n = private_key
    signature = [pow(ord(char), d, n) for char in message]
    return signature

def verify_signature(signature, message, public_key):
    """Verify a signature using the public key."""
    e, n = public_key
    for i in range(len(message)):
        if pow(signature[i], e, n)!= ord(message[i]):
            return False
    return True


p = int(input("Enter a prime number p: "))
q = int(input("Enter a prime number q: "))
```

```
public_key, private_key = generate_keypair(p, q)


message = input("Enter a message to sign: ")


signature = sign_message(message, private_key)


if verify_signature(signature, message, public_key):
    print("Signature is valid")
else:
    print("Signature is invalid")
```

Output –

```
PS C:\Users\arinr\Desktop\Python\MachineLearning> python -u "c:\Users\arinr\Desktop\Crypto_Lab\Lab_11\RSA_signature.py"
Enter a prime number p: 19
Enter a prime number q: 31
Enter a message to sign: hi
Signature is valid
PS C:\Users\arinr\Desktop\Python\MachineLearning>
```

Q2. Write a program to implement the Elgamal signature.

A:

Pseudocode and Explanation –

This code implements the **ElGamal digital signature scheme**. The user inputs a prime number p, a generator alpha, and other parameters such as the private key a, public key beta, and a random number k. The signing process involves computing two values, r and s, where r is derived from alpha^k % p and s is calculated using the message v and the modular inverse of k. The signature (r, s) is then returned. To verify the signature, the code checks if the calculated values v1 = alpha^v % p and v2 = (beta^r * r^s) % p are equal. If they match, the signature is valid, confirming the authenticity of the message. This implementation ensures both the integrity and authenticity of messages signed using the ElGamal system.*Code –*

```
import random

def is_prime(num):
```

```python
    """Check if a number is prime."""
    if num < 2:
        return False
    for i in range(2, int(num ** 0.5) + 1):
        if num % i == 0:
            return False
    return True

def gcd(a, b):
    """Calculate the greatest common divisor of two numbers."""
    while b!= 0:
        a, b = b, a % b
    return a

def multiplicative_inverse(a, m):
    """Calculate the multiplicative inverse of a modulo m."""
    def extended_gcd(a, b):
        if a == 0:
            return b, 0, 1
        else:
            gcd, x, y = extended_gcd(b % a, a)
            return gcd, y - (b // a) * x, x

    gcd, x, _ = extended_gcd(a, m)
    if gcd!= 1:
        raise ValueError("a and m are not coprime")
    return x % m

def sign_message(p, alpha, a, beta, x, k, v):
    """Sign a message using the ElGamal signature scheme."""
    r = pow(alpha, k, p)
    s = (v - x * r) * multiplicative_inverse(k, p - 1) % (p - 1)
    return (r, s)

def verify_signature(p, alpha, a, beta, r, s, v):
    """Verify a signature using the ElGamal signature scheme."""
    v1 = pow(alpha, v, p)
    v2 = (pow(beta, r, p) * pow(r, s, p)) % p
    return v1 == v2


p = int(input("Enter a prime number p: "))
while not is_prime(p):
    p = int(input("Invalid input. Enter a prime number p: "))

alpha = int(input("Enter the generator alpha: "))
a = int(input("Enter the private key a: "))
beta = int(input("Enter the public key beta: "))
x = int(input("Enter the private key x: "))
```

```
k = int(input("Enter the random number k: "))
v = int(input("Enter the message v: "))


r, s = sign_message(p, alpha, a, beta, x, k, v)
print("Signature: ", (r, s))



gamma = (alpha ** k)%p
for i in range (p):
    if (k * i)% (p-1) ==1 :
        k_inv = i
        break
delta = ((v - a * gamma)* k_inv)%(p-1)

check1 = ((beta ** gamma ) * (gamma ** delta))% p
check2 = (alpha ** v)% p

# Verify the signature
if check1 == check2:
    print("Signature is valid")
else:
    print("Signature is invalid")
```

Output –

```
Enter a prime number p: 467
Enter the generator alpha: 2
Enter the private key a: 127
Enter the public key beta: 132
Enter the private key x: 100
Enter the random number k: 213
Enter the message v: 189
Signature:  (29, 287)
Signature is valid
```