|  |  |
| --- | --- |
| Ex.No.4  10/08/24 | **IMPLEMENTATION OF RSA** |

|  |
| --- |
| **AIM:** |

To simulate the working of RSA in Virtual lab environment and to implement the same in Java/Python

|  |
| --- |
| **THEORY:** |

**One way function:**

A one-way function is a function that is easy to compute in one direction, yet difficult to reverse. In RSA, the multiplication of two large prime numbers p and q to generate n is a one-way function.

**Key generation:**

Key generation is the process of generating keys in cryptography. A key is used to encrypt and decrypt whatever data is being encrypted/decrypted.

ϕ(n)=(p−1)×(q−1)

**RSA Encryption:**

Encryption is performed by raising the plaintext message MMM to the power of e (the public key exponent) and then taking the modulus of n. The ciphertext C is computed as: C=M^e mod n

**RSA Decryption:**

Decryption is the reverse process of encryption. The ciphertext C is raised to the power of d(the private key exponent) and then taken modulus n. The decrypted message M is computed as: M=C^d mod n

|  |
| --- |
| **ALGORITHM:** |

**Key generation:**

1. Select two large prime numbers, p and q.
2. Multiply these numbers to find n = p x q, where n is called the modulus for encryption and decryption.
3. Choose a number e less than n, such that n is relatively prime to (p - 1) x (q -1). It means that e and (p - 1) x (q - 1) have no common factor except 1. Choose "e" such that 1<e < φ (n), e is prime to φ (n) i.e gcd(e,n)==1,
4. Repeat step 3 until the condition is satisfied.
5. To determine the private key, we use the following formula to calculate the d such that:

*D\*e mod phi(n) = 1*

1. The private key is (d, n).
2. A ciphertext message c is decrypted using private key (d, n).
3. To calculate plain text m from the ciphertext c following formula is used to get plain text m.

*M = Cd mod n*

RSA Encryption:

1. Get the Plaintext Input from the user.
2. Use the receiver’s public key to encrypt the message, using the formula

*C =Me mod n*

where, e is a random long integer, n is a product of two large primes and M is the message to be encrypted.

1. Return the Ciphertext.

RSA Decryption:

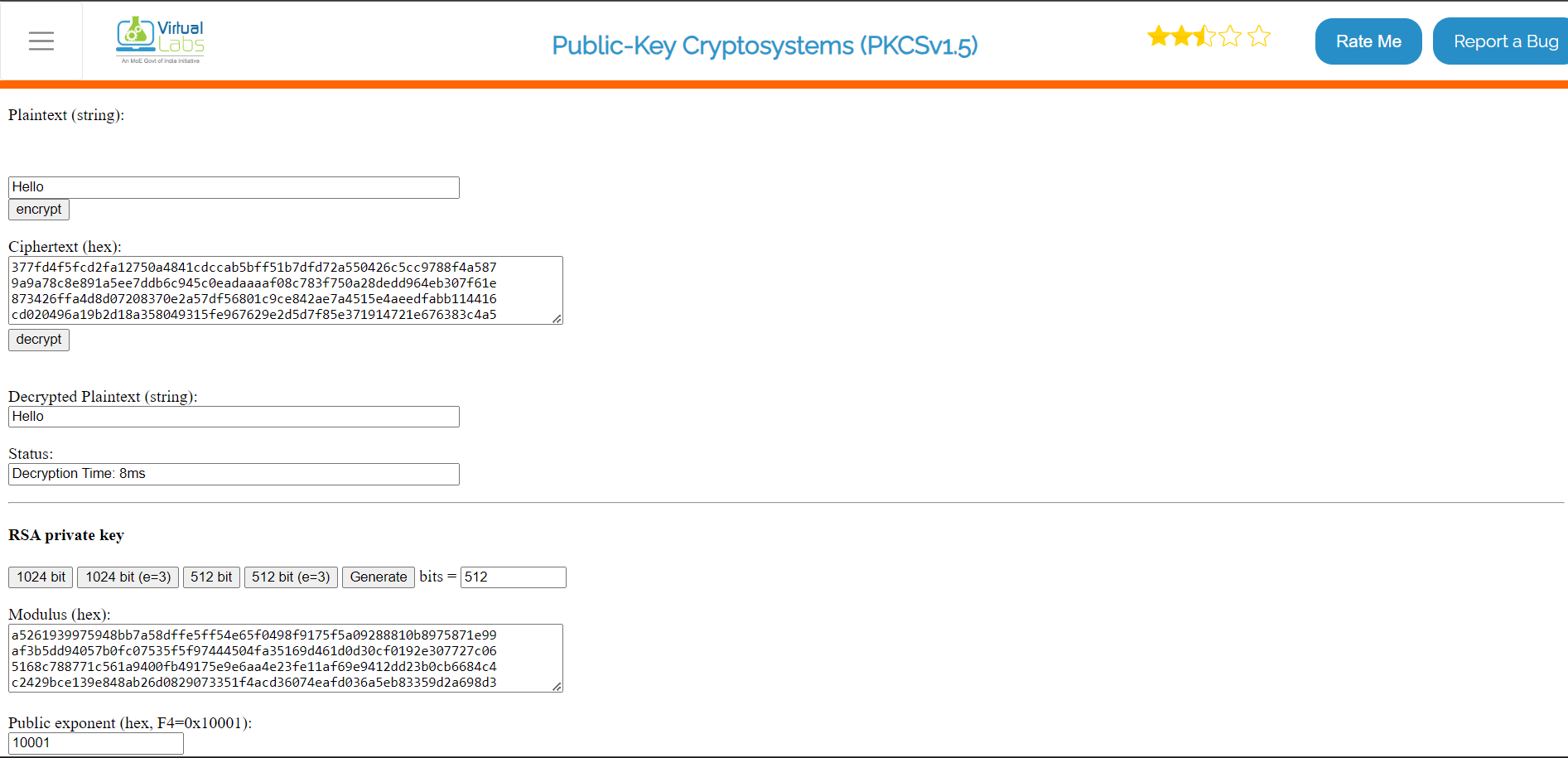
1. Get the Ciphertext from the user.
2. Use the receiver’s secret key to decrypt the message using the formula

*M = Cd mod n*

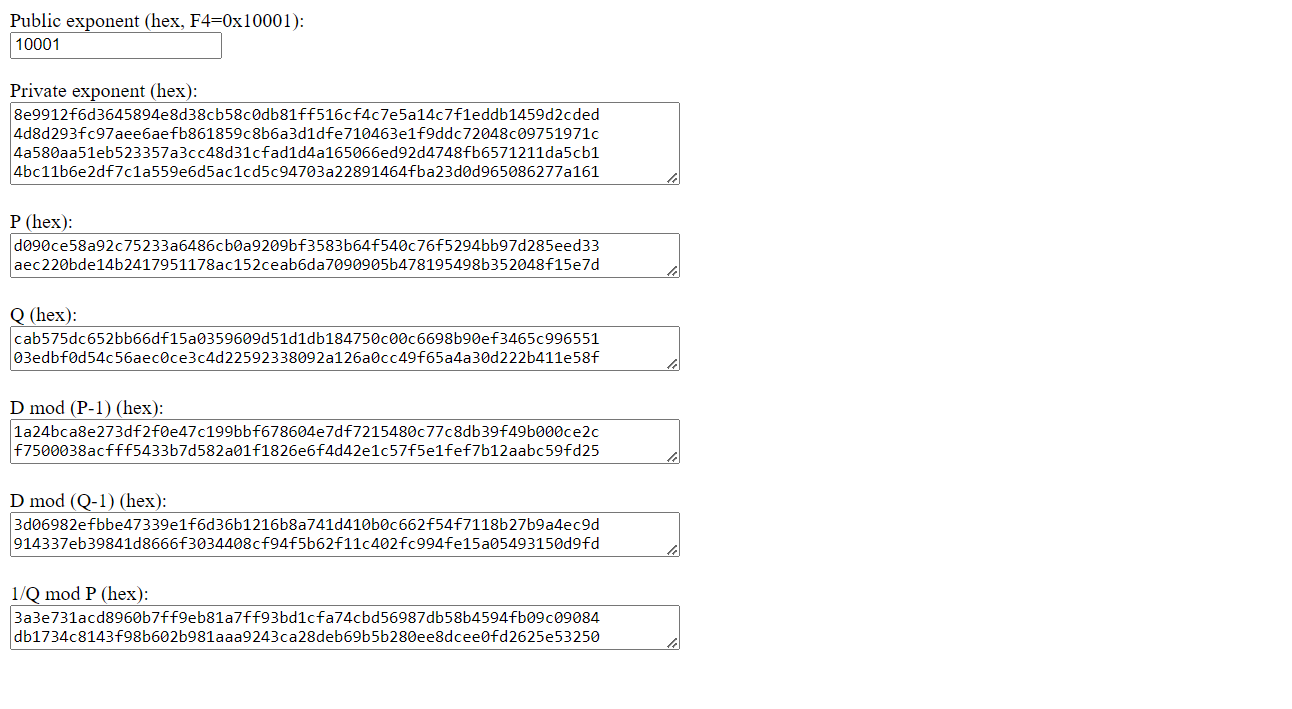
1. Where d is the inverse of e with respect to φ (n) n is a product of two large primes and C is the message to be decrypted.

|  |
| --- |
| **Screen Shots of simulation in Virtual labs** |

**Encryption and Decryption**

****

**Key Generation:**

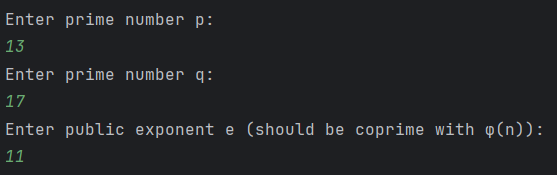
****

|  |
| --- |
| **Coding** |

import java.util.Scanner;  
import java.math.BigInteger;  
public class RSA {  
 public int eulerTotient(int n) {  
 int result = n;  
 for (int p = 2; p \* p <= n; p++) {  
 if (n % p == 0) {  
 while (n % p == 0) {  
 n /= p;  
 }  
 result -= result / p;  
 }  
 }  
 if (n > 1) {  
 result -= result / n;  
 }  
 return result;  
 }  
 public BigInteger modInverse(BigInteger e, BigInteger phi) {  
 return e.modInverse(phi);  
 }  
 public BigInteger encrypt(BigInteger M, BigInteger e, BigInteger n) {  
 return M.modPow(e, n);  
 }  
 public BigInteger decrypt(BigInteger C, BigInteger d, BigInteger n) {  
 return C.modPow(d, n);  
 }  
 public static void main(String[] args) {  
 Scanner sc = new Scanner(System.*in*);  
 RSA rsa = new RSA();  
 System.*out*.println("Enter prime number p:");  
 int p = sc.nextInt();  
 System.*out*.println("Enter prime number q:");  
 int q = sc.nextInt();  
 int n = p \* q;  
 int phi = rsa.eulerTotient(p) \* rsa.eulerTotient(q);  
 BigInteger phiBig = BigInteger.*valueOf*(phi);  
 System.*out*.println("Enter public exponent e (should be coprime with φ(n)):");  
 int e = sc.nextInt();  
 BigInteger eBig = BigInteger.*valueOf*(e);  
 BigInteger d = rsa.modInverse(eBig, phiBig);  
 System.*out*.println("Public Key: (e = " + e + ", n = " + n + ")");  
 System.*out*.println("Private Key: (d = " + d + ", n = " + n + ")");  
 System.*out*.println("Enter the message M (as an integer):");  
 BigInteger M = sc.nextBigInteger();  
 BigInteger C = rsa.encrypt(M, eBig, BigInteger.*valueOf*(n));  
 System.*out*.println("Encrypted message C = " + C);  
 BigInteger decryptedMessage = rsa.decrypt(C, d, BigInteger.*valueOf*(n));  
 System.*out*.println("Decrypted message M = " + decryptedMessage);  
 }  
}

|  |
| --- |
| **SCREEN SHOTS:** |

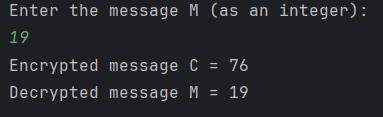
**User Defined Values**

****

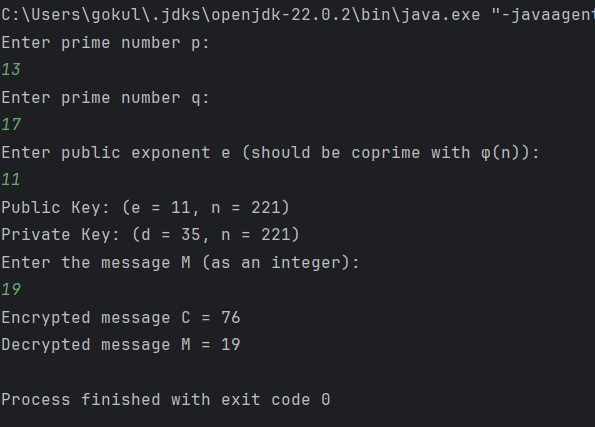
**Generating Keys:**

****

**Encrypting and Decrypting Messages:**

****

**Final Output:**

****

|  |
| --- |
| **RESULT:** |

Thus, the stimulation of RSA in Virtual lab environment and to implement the same in Python has been done successfully.

|  |
| --- |
| **Evaluation** |

|  |  |  |
| --- | --- | --- |
| Parameter | Max Marks | Marks Obtained |
| Uniqueness of the Code | 40 |  |
| Completion of experiment on time | 5 |  |
| Documentation | 20 |  |
| Simulation in Vlabs | 10 |  |
| Total | 75 |  |
| Signature of the faculty with Date |  |  |