Государственный Университет Молдовы

Факультет Математики и Информатики

Департамент Информатики

“Криптография и информационная безопасность”

Лабораторная работа №3  
“Алгоритм электронной подписи RSA”

Преподаватель: O.Cerbu

Выполнил: Маруневич Николай, группа I1902

Кишинев 2022

**Class RSA:**

import java.math.BigInteger;  
  
public class RSA {  
  
 public static void main(String[] args) {  
  
 int key\_length = 256;  
 String rawPlaintext = "sign name";  
  
 RSAMathematics RSAMaths = new RSAMathematics();  
 RSAKeysAndValues RSAKeys = new RSAKeysAndValues();  
  
 //Call the generateKeyPair function and pass in the bitlength which was specified as an argument.  
 RSAKeys.generateKeyPair(key\_length);  
  
 System.*out*.println("\nPublic Modulus: " + RSAKeys.getN());  
 System.*out*.println("\nPublic Exponent: " + RSAKeys.getE());  
  
 System.*out*.println("\nPrivate Modulus: " + RSAKeys.getN());  
 System.*out*.println("\nPrivate Exponent: " + RSAKeys.getD());  
  
 System.*out*.println("\nИсходный текст: " + rawPlaintext);  
  
 BigInteger sign = RSAMaths.modularExponentiation(*encodeStringToASCII*(rawPlaintext), RSAKeys.getE(), RSAKeys.getN());  
 System.*out*.println("Подпись: " + sign);  
 BigInteger m1 = RSAMaths.modularExponentiation(sign, RSAKeys.getD(), RSAKeys.getN());  
 System.*out*.println("\nПроверка подлинности подписи - если m = m' то подпись верна" +  
 "\nm = " + *decodeASCIIToString*(m1) +  
 "\nm' = " + rawPlaintext);  
  
 }  
  
 public static BigInteger encodeStringToASCII(String x) {  
 String result = "111";//Ensure that the first 111 padding is in place.  
 for (int i = 0; i < x.length(); i++) {  
 if ((int)x.charAt(i) <= 99) {  
 result = result.concat("0");  
 }  
 result = result.concat(Integer.*toString*(x.charAt(i)));  
 }  
 result = result.concat("111");  
 return new BigInteger(result);  
 }  
  
 public static String decodeASCIIToString(BigInteger x) {  
 String passedString = x.toString();  
 String result = "";  
 String grabbedChar;  
 for (int i = 0; i < passedString.length(); i = i + 3) {  
 try {  
 grabbedChar = passedString.substring(i, i + 3);  
 result = result.concat(String.*valueOf*((char)Integer.*parseInt*(grabbedChar)));  
 } catch (StringIndexOutOfBoundsException e) {  
 System.*out*.println("\n!\n!\n!\n!!!!Decoding error, got an exception when trying to decode ");  
 }  
 }  
 result = result.substring(1, result.length() - 1);  
 return result;  
 }  
  
}

**Class RSAKeysValues:**

import java.math.BigInteger;  
  
  
public class RSAKeysAndValues {  
 private static BigInteger *p*;  
 private static BigInteger *q*;  
 private static BigInteger *n*;  
 public static BigInteger *totient*;  
 private static BigInteger *e*;  
 private static BigInteger *d*;  
  
 RSAMathematics RSAMaths = new RSAMathematics();  
  
 public BigInteger getE() {  
 return *e*;  
 }  
  
 public BigInteger getD() {  
 return *d*;  
 }  
  
 public BigInteger getN() {  
 return *n*;  
 }  
  
 */\*\*generateKeyPair - generates a aPublic and Private keypir, calling on multiple other functions to achieve this including a prime creation method, a  
 \* Fermat primality test and sets static values accordingly.  
 \*/* public void generateKeyPair(int bitLength) {  
 boolean isNotPrimeFlagP;  
 boolean isNotPrimeFlagQ;  
 do{  
 RSAKeysAndValues.*p* = RSAMaths.createPrimeofBitLength(bitLength);  
 RSAKeysAndValues.*q* = RSAMaths.createPrimeofBitLength(bitLength);  
  
 //Print the primes to screen so we know how far in the process we are.  
 System.*out*.println("\nP: "+*p*);  
 System.*out*.println("\nQ: "+*q*);  
  
 isNotPrimeFlagP = RSAMaths.fermatPrimalityTest(*p*, 500);  
 isNotPrimeFlagQ = RSAMaths.fermatPrimalityTest(*q*, 500);  
 } while (!(isNotPrimeFlagP && isNotPrimeFlagQ)); //<-If both are primes, then it's True && True - so the while loop keeps running! We need to NOT flag this,  
 //so we get a false and the while expression breaks and we leave the loop;  
 System.*out*.println("\nIs P prime? - "+ isNotPrimeFlagP);  
 System.*out*.println("IS Q prime? - "+ isNotPrimeFlagQ);  
  
 RSAKeysAndValues.*n* = *p*.multiply(*q*);  
  
 //Assign static variables in holding class.  
 RSAKeysAndValues.*totient* = RSAMaths.calculateTotient(*p*,*q*);  
 RSAKeysAndValues.*e* = RSAMaths.generateEValue(RSAKeysAndValues.*totient*);  
 RSAKeysAndValues.*d* = RSAMaths.generateDValue(RSAKeysAndValues.*e*, RSAKeysAndValues.*totient*);  
 }  
}

**Class RSAMathematics:**

import java.math.BigInteger;  
import java.util.Random;  
  
public class RSAMathematics {  
  
 Random rnd = new Random();  
  
 */\*\*This method calls themodInverse function and generates a value for the public key exponent.  
 \*/* public BigInteger generateDValue(BigInteger e, BigInteger totient) {  
 BigInteger result;  
 result = e.modInverse(totient);  
 return result;  
 }  
  
 */\*\*  
 \* This method returns a prime with the bitLength specified as an argument  
 \* when invoked.  
 \*/* public BigInteger createPrimeofBitLength(int bitLength) {  
 return BigInteger.*probablePrime*(bitLength, rnd);  
 }  
  
 */\*\*  
 \* This custom method takes in a BigInteger which is a prime number for testing,  
 \* and an integer value which in essence decides the accuracy of the  
 \* primality test.  
 \*/* public boolean fermatPrimalityTest(BigInteger valueToTest, int howManyTimes) {  
 BigInteger a;  
 for (int repeatCount = 0; repeatCount < howManyTimes; repeatCount++) {  
 //pick a randomly from range 1, n-1  
 do {  
 a = new BigInteger(valueToTest.bitLength(), rnd);  
 } while (a.compareTo(BigInteger.*ONE*) > 0 && a.compareTo(valueToTest) < 0);  
 // old a = a.modPow(valueToTest.subtract(BigInteger.ONE), valueToTest);  
 a = modularExponentiation(a, valueToTest.subtract(BigInteger.*ONE*), valueToTest);  
 //a = a.modPow(valueToTest.subtract(BigInteger.ONE), valueToTest);  
 if (!a.equals(BigInteger.*ONE*)) {  
 return false; //composite  
 }  
 }  
 return true; //prime  
 }  
  
 */\*\*  
 \* Euclidian Algorithm used for computing the greatest common denominator  
 \* Takes two parameters, and returns the greatest common denominator  
 \*/* public BigInteger euclidianAlgorithm(BigInteger a, BigInteger b) {  
 if (b.compareTo(BigInteger.*ZERO*) == 0) { // <-if the value of b, when compared to 0, is zero, then they are matched.  
 return a;  
 } else {  
 return euclidianAlgorithm(b, a.mod(b));  
 }  
  
 }  
  
 */\*\*  
 \* Custom method creates a prime which is smaller and coprime to x. Useful  
 \* when creating the exponent of the public key.  
 \*/* public BigInteger createRelativePrime(BigInteger x) {  
 BigInteger z; //initialise  
 do {  
 z = BigInteger.*probablePrime*(rnd.nextInt(x.bitLength()-1), rnd); // make value z a probableprime of bitLength less than that of X  
 } while (!euclidianAlgorithm(x, z).equals(BigInteger.*ONE*) && !z.equals(BigInteger.*valueOf*(2)) && !fermatPrimalityTest(z,50)); //<- NOT the output, as only false breaks the while loop,  
 // and check that z is not 2, as two is not a usable prime  
 return z;  
 }  
  
 */\*\*  
 \* This method calculates the totient value of two BigInteger's which are  
 \* passed to it, p and q.  
 \*/* public BigInteger calculateTotient(BigInteger p, BigInteger q) {  
 return p.subtract(BigInteger.*ONE*).multiply(q.subtract(BigInteger.*ONE*));  
 }  
  
  
 public BigInteger generateEValue(BigInteger totient) {  
 BigInteger eValue; //create eValue  
 eValue = createRelativePrime(totient);  
 return eValue;  
 }  
  
 public BigInteger modularExponentiation(BigInteger M, BigInteger e, BigInteger n) {  
 BigInteger base = M;  
 BigInteger exponent = e;  
 BigInteger result = BigInteger.*ONE*;  
 BigInteger current\_bit;  
  
 while (exponent.compareTo(BigInteger.*ZERO*) > 0) {  
 current\_bit = exponent.mod(BigInteger.*valueOf*(2));  
 if (current\_bit.equals(BigInteger.*ONE*)) {  
 result = ((result.multiply(base).mod(n)));  
 }  
 exponent = exponent.divide(BigInteger.*valueOf*(2));  
 base = (base.multiply(base)).mod(n);  
 }  
 return result;  
 }  
}

**Результат:**

