Prof. Dr. Christoph Karg
Aalen University Of Applied Sciences
Department Of Computer Science

Information Systems Security

Programming Exercise

October 21, 2015

The goal of this programming exercise is the implementation of the number theoretic algorithms and the RSA cryptosystem on base of the cryptography lecture of the Information Systems Security course.

The requirements to successfully work on the exercise are the JDK of the Java Platform Standard Edition[1] and an IDE such as Eclipse[2] or Netbeans[3].

Note: The symbol ♢ denotes the end of an exercise.

Exercise 1.

a) Download and extract the archive `iss-programming-exercise.zip`[4].

b) Examine the contents of the ZIP archive. Which packages do exist? Which source files are contained in the packages?

c) Create a new project in your favorite IDE and copy the classes into your project. ♢

A mandatory requirement for the implementation of RSA is the ability to handle large integers. Fortunately, Java provides the class `BigInteger` that provides the needed functionality.

An instance of `BigInteger` is declared and initialized as follows:

```
BigInteger a = new BigInteger("79341182948930145");
```

The value of the integer is given as a string containing a decimal number. The length of the string is not restricted.

[1]Web page: http://java.oracle.com
[2]Web page: http://www.eclipse.org
[3]Web page: https://netbeans.org/
[4]The download URL is https://its.informatik.htw-aalen.de/docs/ESIGELEC-ISS/esigelec-iss-code.zip

Since it is not possible to overload operators in Java, big integers cannot be handled like primitive data types. For instance, the code fragment

```
(a + b) / (c - d)
```

is invalid, if the variables are instances of `BigInteger`. Instead of that, `BigInteger` provides methods for various arithmetic operations. The methods do not change the value of the instance, but return a new big integer containing the result of the operation.

The following table contains a selection of common operations and their respective code fragments. `a`, `b`, and `c` are instances of `BigInteger`.

Operation	*Code*
$c = a + b$	`c = a.add(b)`
$c = a - b$	`c = a.subtract(b)`
$c = a * b$	`c = a.multiply(b)`
$c = a/b$	`c = a.divide(b)`
$c = a \bmod b$	`c = a.mod(b)`

The operations may be nested. For example, the statement `(a + b) / (c - d)` results in the following code fragment:

```
a.add(b).divide(c.subtract(d))
```

For convenience, `BigInteger` has static fields `ZERO`, `ONE` and `TEN` for easy instantiating the respective integer constants 0, 1, and 10. For a complete documentation of the `BigInteger`'s methods, we refer to the Java API documentation[5].

$\boxed{\textit{Note:}}$ Use the class `BigIntegerExercise` for the implementation of the following exercises.

Exercise 2. Compute the expression

$$(a * b - 4)/c + ((d * d) - (a - b))$$

where $a = 512$, $b = 102$, $c = 3$, and $d = 761$, using `BigInteger` instances and only *one* (!) Java statement. ◊

In order to work with big integers in Java, it is often necessary to compare two integer values. For this purpose, `BigInteger` implements the `Comparable` interface. This is, `BigInteger` provides a method `compareTo()` which enables the comparison of a `BigInteger` instance with another one.

[5] http://download.oracle.com/javase/8/docs/api/index.html?java/math/BigInteger.html

The next table summarizes the comparison operators and the respective code fragments.

Operator	*Code*
$a < b$	`a.compareTo(b) < 0`
$a \leq b$	`a.compareTo(b) <= 0`
$a = b$	`a.compareTo(b) == 0`
$a \geq b$	`a.compareTo(b) >= 0`
$a > b$	`a.compareTo(b) > 0`
$a \neq b$	`a.compareTo(b) != 0`

Exercise 3. Use `BigInteger` instances to compare $a = 781$ and $b = 12891$ with respect to all comparison operators given in the table above. Generate a console output of the comparisons and the respective results. $\Diamond$

Many cryptosystems rely on the ability to generate pseudo random numbers in a secure manner. Usually, pseudo random number generators (prng) provided by a programming language do not satisfy the security requirements. For instance, the Java class `Random` must not be used to generate cryptographic materials. Fortunately, Java provides with the class `SecureRandom` a cryptographically strong prng. The usage of `SecureRandom` is quite simple. At first, an instance is created as follows:

```
SecureRandom prng = new SecureRandom();
```

Next, a pseudo-random big integer is generated by usage of the appropriate constructor:

```
BigInteger i = new BigInteger(128, prng);
```

The constructor gets the bit length (in this case 128 bit) and the pseudo random number generator as parameters. The result is a 128-bit random integer, which was generated in a secure manner.

Exercise 4. Use the `SecureRandom` prng to generate integers with a length of 64, 128, 256, 512, 1024 and 2048 bit. Print the random numbers on the console. $\Diamond$

The next part of this programming exercise focuses on the implementation of the number-theoretic algorithms and the generation of random primes. These mechanisms are the base for a successful implementation of the RSA cryptosystem.

Note: Use the classes `PublicKeyCryptoToolbox` and `PKCTExercise` as starting point for your implementation.

The first algorithm to be implemented is the extended euclidean algorithm (EEA). Given two integers a and b, the extended euclidean algorithm computes the greatest common divisor d of a and b together with the coefficients x and y satisfying the equation

$$\gcd(a, b) = d = ax + by.$$

Since the result of the extended euclidean algorithm consists of the three integer values d, x and y, we use the class `EEAResult` for returning the result of an execution of the algorithm. For convenience, an `EEAResult` instance stores the parameters a and b together with result values d, x and y. Furthermore, the class provides methods to check whether a is relatively prime to b and to compute the multiplicative inverse of $a \bmod b$ in the case that this value does exist.

Note: An integer a is *relatively prime* to the integer b, if $\gcd(a, b) = 1$.

Exercise 5. In the following, we assume that the attributes a, b, d, x and y of an `EEAResult` instance store a correct result of the execution of the extended euclidean algorithm with the parameters a and b.

a) Implement the method `isRelativelyPrime()`. The method shall return `true`, if a is relatively prime to b, this is, if d equals 1.

b) Implement the method `getInverse()`, which computes and returns the multiplicative inverse of a modulo b.

c) Test your implementation with the following values:

$$
\begin{aligned}
a &= 8002109 \\
b &= 7186131 \\
d &= 1 \\
x &= -2996671 \\
y &= 3336940
\end{aligned}
$$

Hint: $a^{-1} \bmod b = 4189460$. ◊

The Java implementation uses the class `EEAResult` to return the result to the calling method.

Exercise 6.

a) Implement the extended euclidean algorithm within the method `extendedEuclideanAlgorithm()` of the class `PublicKeyCryptoToolbox`. Use the class `EEAResult` to return the computed result.

b) Test your implementation by computing the greatest common divisor of $a = 7019544$ and $b = 8135112$ and the coefficients x and y.

Hint: The solution is $d = 3048$, $x = 474$, and $y = -409$.

c) Compute the greatest common divisor of $a = 7186131$ and $b = 8002109$. Is a invertible modulo b? If yes, which is the inverse integer of a? ◊

The next exercise addresses the iterative algorithm for the modular exponentiation $a^b \bmod n$. For a successful implementation the binary representation of the exponent b is needed. This representation can be computed using the methods `bitLength()` and `testBit()` of the class `BigInteger`. The following code fragment illustrates the usage of these methods:

```
BigInteger b = new BigInteger("167");
System.out.print("Binary representation of " + b + ": ");
for (int i=b.bitLength()-1; i>=0; i--) {
    if (b.testBit(i)==true) {
       System.out.print("1");
    } else {
       System.out.print("0");
    }
}
System.out.println("");
```

The above loop computes the binary representation of 167 and prints the result on the console.

Exercise 7.

a) Implement the modular exponentiation inside the method `modExp()` of the class `PublicKeyCryptoToolbox`.

b) Compute the modular exponentiation $a^b \bmod m$, where $a = 17$, $b = 1005$, and $m = 230$.

c) Verify the result of your computation by usage of the built-in method `modPow()` of the class `BigInteger`. $\diamondsuit$

The next step towards the RSA implementation is an algorithm which generates a random number within a given range. This is, given an integer n, the random number shall be chosen uniformly at random from the set $\{1, \dots, n-1\}$. Unfortunately, the class `BigInteger` provides no adequate method for this task.

Therefore, we equip the public key crypto toolbox with this functionality. The algorithm can be derived by the following mechanism. Let $n > 2$ be an integer.

(1) Compute the bit length ℓ of n.

(2) Generate an ℓ-bit random number r

(3) Repeat step (2) until $r \geq 1$ and $r < n$.

(4) Return r as result

The task of next exercise is the implementation of the above mechanism.

Exercise 8. The class `PublicKeyCryptoToolbox` has a `SecureRandom` attribute `prng` which can be used for the generation of integers.

a) Implement the method `randomInteger(int bit_length)` which returns a random number with `bit_length` bits.

   Hint: Use a combination of `BigInteger` and the attribute `prng` to generate the random integer.

b) Implement the method `randomInteger(BigInteger n)` which returns a random number chosen uniformly at random from $\{1, \ldots, n-1\}$.

c) Generate 20 random numbers in the set $\{1, \ldots, 102030405060708090\}$. ◊

The ability of generating random prime numbers is a mandatory requirement for the RSA cryptosystem. Until now, we are able to generate random integers in a secure fashion. The missing link is the ability to check whether a given integer is a prime number. The method of choice is the Miller Rabin primality test The Miller Rabin test is part of the class `PublicKeyCryptoToolbox`. Its implementation is the task of the next exercise.

Exercise 9.

a) Implement the method `witness(BigInteger a, BigInteger n)` which checks whether `a` is a witness that `n` is not a prime.

b) Implement the Miller Rabin test within the method `millerRabinTest(BigInteger n, int s)`. Use the method `randomInteger()` to generate the random numbers.

c) Use your implementation with $s = 100$ to check the correctness of the following table:

Integer	*Prime?*
343232674978653231166402657365997144371953839307928119227511	yes
667984267564412673929015509827448340743034959781814076053617	yes
902857742149935096180418505174605673479122931367283811478172	no
408025803078911998315951562970145017384911797981108589419277	no
1040747016400791716218800060097121047453800566864795676123313	yes
341920262248211364330159957004187372102128507551704555404569	no
880723572255844606588685481136407927962444382553394348261623	yes
1130242628975018265380102543215055338361897468448588898970126	no

d) Implement the method `randomPrime(int bit_length, int s)` of the class `PublicKeyCryptoToolbox`, which returns a random prime with `bit_length` bits. The error rate shall be 2^{-s}.

e) Generate a random prime with 128, 256, 512, and 1024 bits. The error rate shall be 2^{-100}. ◊

Now we are able to work on the last part of this programming exercise: the RSA cryptosystem.

Note: The classes `RSAExercise`, `RSAEncryptor`, and `RSADecryptor` are the framework for the next exercises.

Before working on the RSA cryptosystem itself, we implement a method to generate the RSA parameters, this is two different primes p and q, and the private key d and its public counterpart e.

Exercise 10. The generation of the RSA parameters is done within the method `rsaParamsExercise()` of the class `RSAExercise`.

Important: The result of the parameter generation is a console output which will be used in the next exercise in a "cut and paste" manner. Please use the pre-defined variables and do not modify the prepared source code.

a) Generate two different primes p and q each with a length of 256 bit. The error rate shall be 2^{-50}.

b) Compute n and $\varphi(n)$ from p and q.

c) Generate a random public key e.

d) Compute the private key d from e. ◊

The implementation of the RSA cryptosystem is split in two parts. The class `RSAEncryptor` includes the encryption part and the class `RSADecryptor` handles the decryption part.

Note: Use the generated parameters of exercise 10 in the following exercises 11 and 12.

Exercise 11. The class `RSAEncryptor` contains the code for RSA encryption.

a) Analyze the methods and attributes of the class `RSAEncryptor`.

b) Implement the method `encrypt(BigInteger x)` which performs the RSA encryption of the integer `x`.

c) Extend the method `rsaExercise()` with the parameters, you generated in exercise 9.

d) Choose an appropriate plain text x and encrypt this plaintext with RSA.

e) Use the method `encrypt(String x)` to encrypt a string of your choice. ◊

Exercise 12. The class `RSADecryptor` contains the code for RSA decryption.

a) Analyze the methods and attributes of the class `RSADecryptor`.

b) Implement the method `decrypt(BigInteger y)` which performs the RSA decryption of the integer `y`. Use Garner's formula to compute the plaintext.

c) Extend the method `rsaExercise()` such that you can decrypt the ciphertext which you computed in exercise 11d.

d) Use the method `decrypt(Vector<BigInteger> v)` to decrypt the string of exercise 11e. ◊

Exercise 13. As a final test run the method `finalTest()` of the class `RSAExercise` and analyze the console output. ◊