Laboratory 6. **Computational Problems Behind Public-Key Cryptosystems, BigIntegers in Java**

In this laboratory work we pay attention to computational problems that stay at the core of public key cryptosystems, RSA in particular. We exemplify computational problems with the help of the `BigInteger` class from Java. Rather than briefing through the capabilities of this class, we take a problem based approach in which we try to underline the math behind cryptosystems such as the RSA (pointing on issues that potentially cause insecurity). A shortcoming of this laboratory work is that we do not describe the particular algorithms behind these computations, however some of the algorithms are described during the lectures and here we try to fix the notions by playing with numbers.

### 6.1 The Java BigInteger Class

The Java `BigInteger` class allows working with arbitrary precision integers. There is virtually no limit on their size, except for the memory available. However, in public key cryptosystems we usually work with integers that are in the order of several thousands of bits, e.g., 1024-4096 in case of the RSA, so you should imagine this as the practical size that we target. To initialize a `BigInteger` is fairly simple, the constructor of the class can also take strings, for example,

```java
BigInteger two = new BigInteger("2");
```

creates a `BigInteger` with value 2. You can initialize the integer with a value of your choice, e.g.,

```java
BigInteger exponent = new BigInteger("65537");
```

Then operations are simply performed by calling the related methods. For example if you want to compute an exponentiation $2^{65537} \mod 3$ simply call:

```java
BigInteger result = two.modPow(exponent, new BigInteger("3"));
```

In Table 1 we summarize the arithmetic operations and the equivalent Java `BigInteger`'s methods.

<table>
<thead>
<tr>
<th>Arithmetic Operation</th>
<th>Java BigInteger Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>additions and subtractions (+, -)</td>
<td>subtract(BigInteger val)</td>
</tr>
<tr>
<td></td>
<td>add(BigInteger val)</td>
</tr>
</tbody>
</table>
### Table 1. A summary of arithmetic operations and the corresponding methods in Java

<table>
<thead>
<tr>
<th>Multiplications and divisions (*) /</th>
<th>multiply(BigInteger val)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>divide(BigInteger val)</td>
</tr>
<tr>
<td></td>
<td>divideAndRemainder(BigInteger val)</td>
</tr>
<tr>
<td></td>
<td>mod(BigInteger m)</td>
</tr>
<tr>
<td></td>
<td>remainder(BigInteger val)</td>
</tr>
<tr>
<td></td>
<td>compareTo(BigInteger val)</td>
</tr>
<tr>
<td></td>
<td>max(BigInteger val)</td>
</tr>
<tr>
<td></td>
<td>min(BigInteger val)</td>
</tr>
<tr>
<td>Comparisons (&lt;, &gt;)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Exponentiation and modular</td>
<td>modPow(BigInteger exponent, BigInteger m)</td>
</tr>
<tr>
<td>exponentiation, $a^x$</td>
<td>pow(int exponent)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Greatest common divisor (GCD) and</td>
<td>gcd(BigInteger val)</td>
</tr>
<tr>
<td>multiplicative inverse, i.e., $x^{-1}$</td>
<td>modInverse(BigInteger m)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Primality testing</td>
<td>isProbablePrime(int certainty)</td>
</tr>
<tr>
<td></td>
<td>probablePrime(int bitLength, Random rnd)</td>
</tr>
</tbody>
</table>

6.2 Solved Exercises

The private exponent reveals the factorization of the modulus. This is a commonly known property of the RSA. It is also the reason for which a modulus cannot be shared by two distinct entities even if they use distinct public exponents (since the private exponent of each of them can be used to factor the modulus and recover the private exponent of the other). This problem is also referred as the common modulus problem.

Let the following RSA key:

$$K_1 : \{ n = 837210799, e = 7, d = 478341751 \}$$

Show how the modulus can be factored given the private key and find the private exponent for the following key:

$$K_3 : \{ n = 837210799, e = 17, d = ? \}$$
Solution 1. The mathematical relation between the private and public RSA exponents is the following:

\[ d \cdot e \equiv 1 \mod \phi(n) \]

This implies that there exists a number \( k \) such that

\[ d \cdot e = 1 + k \cdot \phi(n) \]

Since

\[ \phi(n) = (p - 1)(q - 1) = p \cdot q - p - q + 1 \]

It follows that

\[ d \cdot e = 1 + k \cdot (p \cdot q - p - q + 1) \]

Rearranging the terms we get

\[ pq + 1 - \frac{d \cdot e - 1}{k} = p + q \]

We know all values from the left side, except for \( k \). However, by closely examining the previous relation \( d \cdot e = 1 + k \cdot (p \cdot q - p - q + 1) \) since on the right side \( p \cdot q \) is much larger than \(-p - q + 1\) we are not far by approximating \( k \) as:

\[ k \approx \left\lfloor \frac{d \cdot e - 1}{p \cdot q} \right\rfloor = \left\lfloor \frac{d \cdot e - 1}{n} \right\rfloor \]

In our case, starting from the already known key we get:

\[ k \approx \left\lfloor \frac{7 \cdot 478341751 - 1}{837210799} \right\rfloor = 4 \]

It follows that:

\[ p + q = \frac{4 \cdot (837210799 + 1) + 1 - 7 \cdot 478341751}{4} = 112736 \]

This implies that \( p \) and \( q \) can be extracted as roots of the equation \( x^2 - Sx + P = 0 \) where \( S = 112736 \) and \( P = 837210799 \). By elementary calculations, we get:

\[ \Delta = 112736^2 - 4 \cdot 837210799 = 9360562500 \]

The roots follow as:

\[ x_1 = \frac{112736 + \sqrt{9360562500}}{2} = 104743 \quad \text{and} \]

\[ x_2 = \frac{112736 - \sqrt{9360562500}}{2} = 104743 \]
These are the factors of the modulus. Finding the second private exponent is now trivial as:

\[ d = e^{-1} \mod (p-1)(q-1) \Rightarrow d = 17^{-1} \mod 837098064 = 246205313 \]

**Solution 2.** The private exponent always decrypts a message encrypted with the public one, since:

\[ \forall x \in \mathbb{Z}_n, x = (x^e)^d \mod n \]

Given the values from the first key we always have:

\[ x = (x^2)^{478341751} \mod 837210799 \]

By multiplying with \( x^{-1} \) and rearranging we get:

\[ x^{7 \cdot 478341751 - 1} = 1 \mod 837210799 \]

Dividing by 2 we get:

\[ \left( \frac{x^{7 \cdot 478341751 - 1}}{2} \right)^2 = 1 \mod 837210799 \]

This means that the right quantity is a square root of 1. To eliminate the two trivial roots of 1, i.e., +1 and -1, we continuously divide the exponent until we get a non-trivial root. For example, let us fix \( x = 10 \) and compute:

\[
x^{7 \cdot 478341751 - 1} = 1, x^{4 \cdot 478341751 - 1} = 1, x^{8 \cdot 478341751 - 1} = 1, x^{16 \cdot 478341751 - 1} = 562155682
\]

It is easy to note that when dividing the exponent with 16 the result is no longer 1. For this final result we have:

\[ \text{cmmdc}(562155682 - 1, n) = 7993 = p \]

\[ \text{cmmdc}(562155682 + 1, n) = 104743 = q \]

In this way we have successfully extracted the factors of \( n \). The mathematical explanation is that we have:

\[
\left( x^{7 \cdot 478341751 - 1} \right)^2 \equiv 1 \mod n \iff \left( x^{16 \cdot 7 \cdot 478341751 - 1} - 1 \right) \left( x^{16 \cdot 7 \cdot 478341751 - 1} + 1 \right) \equiv 0 \mod n
\]
and since \( x^{16} \neq \pm 1 \) it means that the two factors contain the prime numbers that divide \( n \).

**Small encryption exponents.** While small exponents are preferred for encryption because they result in faster operation, small exponents are known to cause insecurity. The .NET framework has the default exponent set to 65537, this should be secure, but it may be tempting to use even smaller exponents. Consider the following two 1536 and 2048 bit modules taken from the RSA challenge website [http://www.rsasecurity.com/rsalabs/node.asp?id=2093](http://www.rsasecurity.com/rsalabs/node.asp?id=2093)

\[
\begin{align*}
n_1 &= 18476997032117414743068356202001644030185493386634101714717857749106516967111612498593376 \\
&\quad 843054357448586561606154457179405222971773252466096064694607124962372044220226975675687378 \\
&\quad 42756238950876467844093285157496578843415088475528298187264513398636493190808467199043 \\
&\quad 18743812833635027954702826532978029349161558118810498499083195450098489397752272570525785 \\
&\quad 91944993870073697556884369338127796130892303925695253261620823676490316036551371447913 \\
&\quad 932347169566988069 \\
\end{align*}
\]

\[
\begin{align*}
n_2 &= 25195908475678934940271832400483985714292821262040320277771378360436620207059555624018 \\
&\quad 52588078440694192906412495510821892985591491761845082808498739280728777673597 \\
&\quad 14183472702618699375014971824691165077613379850995700097304597488084284017974291006424586 \\
&\quad 9181719511874612151517645632282216869987549182422433637259085141865462043576798423387184 \\
&\quad 774447920739934236588482428119816381501056748104516603773060562016196762561338441463083 \\
&\quad 390441495263443219011146575444541784240209246165157233507787077498171257724679629263863563 \\
&\quad 73289912154831438167899885044044536402352738195137863564931212010397122822120720357 \\
\end{align*}
\]

By this exercise we show that even if the factorization of these numbers is unknown (these challenge numbers were not yet factored, so it is impossible for us to know their factorization), one can still recover encrypted values in certain situations if the exponents are small. Consider that one fixes an encryption exponent \( e = 2 \) (this is in fact known as the Rabin cryptosystem and is a secure cryptosystem when correctly used, see the lecture material for more details) and that one encrypts the same message \( m \) once with each modulus, i.e., \( c_1 = m^2 \mod n_1, c_2 = m^2 \mod n_2 \). Given the result of the encryptions below, you are requested to find the encrypted message:

\[
\begin{align*}
c_1 &= 172082497552251785753946730914655180603828422705148960933919792910306562922397291446654035 \\
&\quad 1368594626690514052214759764494443164349805757586202347941324566382604120964935386258122 \\
&\quad 499988036157176134095979071800119001744747405240965750082014086617138982108989997849347323 \\
&\quad 51564883260736757498753677321490105289244104109064444335973488450882364503785143383799248 \\
\end{align*}
\]
Solution. The mistake comes from the fact that the small encryption exponent allows one to recover the message by squaring the output composed via the Chinese Remaindering Theorem (CRT). We show how this can be done in what follows. CRT implies that the following result holds:

\[
\begin{align*}
2^{11} & \equiv c_1 \mod n_1 \\
2^{22} & \equiv c_2 \mod n_2
\end{align*}
\]

But message \( m \) was encrypted with the first modulus, this means it cannot exceed 1536 bits. Therefore the square of the message has at most \( 2 \times 1536 = 3072 \) bits. CRT allows one to retrieve a solution modulo \( n_1 \cdot n_2 \), moreover, this solution is unique. Since the two modules have 1536 and 2058 bits respectively it means that this solution is unique for up to \( 1536+2048 = 3584 \) bits and thus message \( m \) can be fully recovered as square root of the value retrieved via CRT. We show how this can be done by using the CRT solution offered by Gauss. First, we compute the modular inverses:

\[
\begin{align*}
n_1^{-1} \mod n_2 &= 43109875894216565950520012736069966019932054377912959230612193555358489326217220 \\
&\quad + 47996479172395205182484709925813450868235923610986766116713929723063714071159178 \\
&\quad + 93215317798609151299752560828411461201437032915540744391593323344251931239955 \\
&\quad + 7745758659436899226059650980952083423254419082478210313631854686339459392688075 \\
&\quad + 63205737617621886415267112093028170757381011542972482815196722455369893470428219 \\
&\quad + 467027579753828654250472908499342412098242978632588981001473499765226608485132147 \\
&\quad + 822588060662093776567407298712411515994875794907548056460946369535345877671301941 \\
&\quad + 75604485063787798604617845708610302486546965847917376536
\end{align*}
\]

Then we use Gauss’s solution for the CRT and compute:

\[
m^2 = (c_1 n_1 n_2^{-1} \mod n_1 + c_2 n_1 n_2^{-1} \mod n_2) \cdot n_1 n_2 = 4024840927931778156259470333147159100344874869225366381022540697914175856029447329 \\
+ 171360980482486692513635802024046789117355557994243268711309574801864624906335017
\]
Balanced vs. unbalanced RSA. The RSA version in which the two factors \( p \) and \( q \) have the same size is also referred as balanced RSA. An unbalanced version of the RSA was also proposed, it benefits from a large modulus (harder to factor, thus increased security) but still fast for decryption if this is performed via the smaller factor. Unbalanced RSA assumes the use of a small \( p \) (e.g., several hundred bits) and a larger \( q \) (e.g., several thousand bits). Only messages smaller than \( p \) are encrypted and then decryption is performed modulo \( p \) (this can be done only by the owner of the private key who is in possession of \( p \)). For correct encryption, a bound \( l \) on the size of the plaintext is made public (this does not make the scheme unsafe, it is simply the bitlength of \( p \) which does not make factorization trivial). We give a small numerical example:

**Key generation:**

\[
p = 541, q = 104729, e = 7, l = 200
\]

\[
\Rightarrow n = 56658389, \phi(n) = (p - 1)(q - 1) = 56553120,
\]

\[
d = e^{-1} \mod (p - 1) = 463
\]

**Encryption:**

\[
m = 300 \Rightarrow c = 300^7 \mod 56658389 = 18157376
\]

**Decryption:**

\[
m = 18157376^{463} \mod 541 = 300
\]

Show how a CCA2 (Chosen Ciphertext Attack) attack can be mounted such that the adversary can recover the private key. Use the previous numbers to illustrate the attack.
Solution. The CCA2 attack assume that the adversary has unlimited access to the decryption machine, i.e., the machine accepts to decrypt messages at his choice. The adversary can cheat and encrypt a message that is larger than the bound \( l \), e.g.,

\[
c = 1000^7 \mod 56658389 = 27641532
\]

The decryption machine performs decryption according to the rules and answers with:

\[
m = 27641532^{463} \mod 541 = 459
\]

Now the adversary can use this response to factor the modulus as:

\[
\gcd(1000 - 459, n) = 541
\]

Thus, the adversary can factor the modulus and completely break the cryptosystem. The mathematical fact behind this attack is trivial. Since \( x \equiv (x^e)^d \mod p \) but \( x \neq (x^e)^d \mod p \) (note that \( p \) ) it follows \( x - (x^e)^d = k \cdot p \) and thus \( x - (x^e)^d \neq 0 \) which implies \( \gcd(x - (x^e)^d, n) = p \) and thus the modulus can be factored.

6.3 Further Exercises

1. Given the RSA encryption below with the corresponding modulus and exponent, find the encrypted message assuming that encryption was performed without padding.

\[
n = 8716664131891073309298060436223878083629567867863418669374287834553659623916739172495744915952229207084297741464557132198229086365652604590297378403184129
\]

\[
e = 3
\]

\[
c = 1375865583010982618632308529423371271821438577980922927124130396877925863587827122886875024570556859122064458153631
\]

2. Given the RSA key-pair below find the factorization of the modulus.

\[
n = 5076313634899413540120536350051034312987619378778911504647420938544746517711031490115528420427319479274407389058253897498557110913160302801741874277608327,
\]

\[
e = 3
\]

\[
d = 3384209089932942360080357566700689541991746252519274336431613959029831011807259226655786125050887727921274719751986104162037800807641522348207376583379547
\]
3. The following fact is considered an interesting property of the RSA, although we do not know the sum of the two factors of the modulus, i.e., $p + q$, we can compute the value of $x^{p+q} \mod n$. Figure out how this is possible and compute this value for the numbers below.

\[ n = 107006465856808584852050373529985247886583743870981513899285988324995549891628785723362749860665786676359278833959921943627412052904161935201780928478603, \]
\[ x = 7133764390453923899013669156866568319243891625806543425995239922166636999277387253194048505767340924598064169304136210581809906511216168762318630818311867 \]

4. Factor the following integer, knowing that it is the power of a prime number. What is the expected number of steps to factor an integer of this form?

\[ n = 14121216559045592723913725470284552915893297299545955512586695122770931673525642809374899750759599902194861123590215515956690880367223678270178015326064870241064451357668006100271472321177891238940152788700404344528460044850936426758850098076585795411392720202615259916568029436599814044031229151775310358906532071125841544313301394408906580430629631327415853437044184526066718512746557009387552200433014081763141603486989053788826143369939787183615667314218625753419259203124994887398592902895704663282917257084748597189183168763622960749 \]

5. To speed-up verification time for multiple RSA signatures, rather than verifying each signature independently, one can check the following equality: \( (\prod_{i=1}^{k} s_i)^e = \prod_{i=1}^{k} h(m_i) \) (this is called batch verification). This method is fast as it requires a single modular exponentiation, in contrast to \( k \) exponentiations (and indeed modular exponentiation is the most expensive computational step in verifying signatures). However, there is a problem with this method: show that given multiple signatures \( \{ s_1, s_2, ..., s_k \} \) corresponding to a set of messages \( \{ m_1, m, ..., m_k \} \) one can produce a fake set of signatures that passes the batch verification test but no signature will hold for any of the messages in particular.

6. Prove the equivalence between the following computational problems: RSA-Key, computing Euler-Phi and Integer Factorization.

**Note.** Since there is no method in the Java.BigInteger class for computing integer square roots, you may recycle the naive code below.
private static BigInteger NaiveSquareRootSearch(BigInteger a, BigInteger left, BigInteger right)
{
    // fix root as the arithmetic mean of left and right
    BigInteger root = left.add(right).shiftRight(1);
    // if the root is not between [root, root+1],
    // is not an integer and root is our best integer approximation
    if(!((root.pow(2).compareTo(a) == -1)&&(root.add(BigInteger.ONE).pow(2).compareTo(a) == 1))){
        if (root.pow(2).compareTo(a) == -1) root = NaiveSquareRootSearch(a, root, right);
        if (root.pow(2).compareTo(a) == 1) root = NaiveSquareRootSearch(a, left, root);
    }
    return root;
}

public static BigInteger SquareRoot(BigInteger a)
{
    return NaiveSquareRootSearch(a, BigInteger.ZERO, a);
}