RSA

Bei der Behandlung des RSA - Verfahrens stellt sich immer die Frage: Wieviel Mathematik ist notwendig und worauf kann verzichtet werden. Sicher kann man nicht auf einige Kenntnisse über Primzahlen verzichten.

Primzahlen

1. Primzahlen treten nicht regelmäßig auf.
2. Es gibt kein "einfaches" Rechenverfahren zur Bestimmung von Primzahlen.
3. Die Untersuchung der Teilereigenschaft ist prinzipiell der einzige sichere allgemeine Weg, um von irgendeiner beliebigen Zahl zu klären, ob sie prim ist (=eine Primzahl ist) oder nicht.
4. Das Sieb des Eratosthenes ist schnell und einfach zu rechnen, ist für große Zahlen wegen der großen Datenmenge aber zu aufwendig.
5. Es gibt einige Algorithmen, die relativ viele Primzahlen liefern.
6. Es gibt Verfahren, die gute Hinweise darauf liefern, ob eine Zahl eine Primzahl ist oder nicht.
7. Primzahlen haben – insbesondere bei der Rechnung ***modulo*** n – wichtige Eigenschaften.

Man könnte diese Liste sicher noch sinnvoll verlängern.

Restklassen modulo n:

Bei der Zerlegung mit Rest von natürlichen Zahlen erhält man als „Ergebnisse“ zwei Zahlen, einmal den ganzzahligen Quotienten und andererseits den Rest. Zerlegt man z.B. 50 mit der Zahl 6, dann erhält man

$$50 = \mathbf{6} \cdot 8 + 2$$

Die Funktionen zur Berechnung dieser Werte sind in jeder höheren Programmiersprache implementiert, bei Delphi (PASCAL) heißen sie div und mod, bei Scheme quotient und modulo, in Java bringt z.B. die von uns verwendete Klasse BigInteger die Methode `divideAndRemainder(BigInteger val)` mit. Man erhält also..

bei Delphi	50 div 6 → 8	und	50 mod 6 → 2
bei Scheme	(quotient 50 6) → 8	und	(modulo 50 6) → 2

bei JAVA

```
BigInteger result = (new BigInteger("50")).divideAndRemainder(new
                                                    BigInteger("6")
```

liefert `result[0].intValue()` → 8 und `result[1].intValue()` → 2

Durch den Vergleich der Reste entsteht für jeden mögliche Teiler eine Aufteilung der Menge der natürlichen Zahlen, man nennt sie Restklassen modulo n. Es zeigt sich nun, dass diese Restklassen besondere Rechenregeln haben, die man speziell bei der Verschlüsselung nutzen kann.

Beispiele für Berechnungen:

(12 + 15) modulo 17 = 27 modulo 17 = 10

Es gibt ein Additionsinverses modulo 17 zu 12. *Es ist ... ?*

[5 * (12 + 15)] modulo 17 = (60 + 75) modulo 17 = 135 modulo 17 = 16

50 modulo 17 = 16 Zusammenhang ?

700 modulo 17 = 3 , 500 modulo 17 = 7

(700 * 500) modulo 17 = 350.000 modulo 17 = 4 Zusammenhang ?

Betrachte Tabellen der Vielfachen und Potenzen von Zahlen modulo einer Zahl.
Untersuche die Verhältnisse bei anderen modulo – Zahlen.

Aufgabe

Rechnen Sie einige Beispiele durch.
Überlegen Sie sich insbesondere dabei, weshalb sich diese modulo – Rechnungen für Verschlüsselungen eignen könnten !

Die Eulersche Zahl : φ(n)

φ(n) ist die Anzahl der zu n teilerfremden Zahlen unter n.
„*teilerfremd*" heißt dabei: einziger gemeinsamer Teiler ist die 1.
Beispiele:

n = 4, dann ist φ(n) = Anzahl von {1,~~2~~,3} = 2.
n = 7, dann ist φ(n) = Anzahl von {1,~~2~~,~~3~~,~~4~~,~~5~~,~~6~~} = 6.
n = 20, dann ist φ(n) = Anzahl von {1,~~2~~,3,~~4~~,~~5~~,~~6~~,7,~~8~~,9,~~10~~,11,~~12~~,13,~~14~~,~~15~~,~~16~~,17,~~18~~,19} = 8.
Der Fall der 9 zeigt, dass dies nicht nur Primzahlen sind.

Bei der Eulerschen Zahl gibt es interessante Spezialfälle:

1. n ist Primzahl: Dann ist φ(n) = n - 1
2. n ist das Produkt von zwei Primzahlen p und q, also $n = p \cdot q$.
   Dann ist $\varphi(n) = (p-1) \cdot (q-1)$

(Einfach einzusehen : Alle Vielfachen von p fallen heraus und zusätzlich die von q fallen heraus.)
Beispiel: Das Produkt von 5 und 7, also 35 :

~~1~~ 2 3 4 ~~5~~ 6 ~~7~~ 8 9 ~~10~~ 11 12 13 ~~14~~ ~~15~~ 16 17 18 19 ~~20~~
~~21~~ 22 23 24 ~~25~~ 26 27 ~~28~~ 29 30 31 32 33 34

Die Anzahl der oben nicht weggestrichenen Ziffern ist : (5 – 1) * (7 – 1) = 4 * 6 = 24

In Bezug auf unsere modulo - Rechnungen bei der Verschlüsselung ergibt sich eine Folgerung im Satz von Euler:

Satz von Euler

Sind zwei natürliche Zahlen m und n teilerfremd, dann gilt

$$m^{\varphi(n)} \text{ modulo } n = 1$$

Greifen wir zurück auf die beiden o.a. Spezialfälle:

1. m^{n-1} modulo n = 1 für jede Primzahl n!
2. $m^{(p-1)\cdot(q-1)} \text{ modulo } (p \cdot q) = 1$ für jedes Produkt p ▪ q von zwei Primzahlen

RSA

Die Beschreibung des RSA - Verfahrens selbst ist nun eigentlich recht einfach. Zunächst der Grundgedanke: Warum ist es so wichtig, dass das Ergebnis in der Zeile 2 den Wert 1 hat? Ganz einfach: Nimmt man noch einmal mit m mal, so erhält man wegen $1 \cdot m = m$ sicher wieder den ursprünglichen Wert, genau das, was man beim Entschlüsseln erzielen will.

Betrachtet man in der oben angegebenen Gleichung das m als die zu verschlüsselnde Botschaft, dann wird sie in der Menge der Zahlen {1; ... ; n} durch die zu übermittelnde Codezahl m' ersetzt, indem man eine Schlüsselzahl e als Exponent verwendet und m damit potenziert. Das geschieht natürlich alles modulo n. Kann man nun für die Entschlüsselung eine Zahl d angeben, die mit e multipliziert gerade eine Zahl ergibt, die um 1 größer als ein Vielfaches (hier k) von φ(n) ist, dann ergibt

$$m^{d*e} \text{ modulo } n = m^{k*\varphi(n)+1} \text{ modulo } n = (1^k * m) \text{ modulo } n = m$$

wieder den ursprünglichen Wert.

d und e

Die eigentliche Frage ist daher : Woher bekommen wir p, q, d und e und welche Anforderungen haben außer p und q (Primzahlen) die Zahlen e und d zu erfüllen?

Die Anforderung an e ist relativ einfach: e muss teilerfremd zu φ(n) sein. (Überlegen Sie einmal, weshalb!) Eine Primzahl größer als φ(n) erfüllt diese Bedingung sicher.

Eine der wichtigen Teilaufgaben, die in unserem Verfahren zu leisten sind, wird also die Bereitstellung der benötigten (sehr großen) Primzahlen sein.
Glücklicherweise ist das Problem nicht ganz so schwer, wie das Problem, eine vorgegebene Zahl in ihre Primfaktoren zu zerlegen. Zunächst einmal können wir den kleinen Fermatschen Satz einsetzen. Er beruht auf dem ersten o.a. Spezialfall des Satzes von Euler, den ich hier noch einmal in spezieller Form angebe:

$$\text{p Primzahl und } m < n \Rightarrow m^{p-1} \bmod n = 1$$

Leider ist er eine *wenn ... dann ...* – Aussage und nicht umkehrbar; man kann also ***nicht*** sagen *„wenn* $m^{p-1} \bmod n = 1$ *ist, dann muss p eine Primzahl sein“*.

Andererseits kann man mit hoher Sicherheit davon ausgehen, eine Primzahl zu haben, wenn man mehrfach zufällige Zahlen m_i unter n auswählt und sie jeweils dem Fermattest unterzieht, also untersucht, ob man bei der o.a. Rechnung 1 erhält oder nicht. In den seltenen Fällen, bei denen dabei ein Teiler übersehen wird, wird er sicher kein kleiner Teiler sein und daher nur bei sehr wenigen Zahlen des Zahlenraumes, den man verschlüsseln will, zu Schwierigkeiten führen.
Aufgabe: Von welcher Schwierigkeit ist hier die Rede?

d zu e finden

Beschäftigen wir uns nun mit der Bestimmung des ***modularen Inversen*** d zu e. Nach der oben angeführten Gleichung ist das also die Zahl d zu e mit :

$$d * e = k * \varphi(n) + 1$$

oder anders formuliert ist das Produkt um 1 größer als ein Vielfaches von phi(n) und damit :

$$(d * e) \text{ modulo } \varphi(n) = 1$$

Solche Zahlen lassen sich natürlich bei kleinen Zahlen mit Durchprobieren aller Zahlen finden, bei größeren Zahlen ist das sicher kein angemessenes Verfahren mehr. Es gibt zudem ein besseres:
Erweitert man den ***Euklidischen Algorithmus*** zur Bestimmung des ggT von zwei Zahlen auf das Verfahren von Euklid-Bachet, bei dem die Faktoren der Darstellung der einzelnen Teilwerte aus den ursprünglichen beiden Zahlen mit verwaltet werden, dann stellt man fest, dass man mit seiner Hilfe auf einfache Art zu zwei natürlichen Zahlen eine Summendarstellung zu ihrem ggT finden kann:

$$ggT(a,b) = x * a + y * b \qquad [x,y \epsilon Z]$$

Man kann nun zeigen, dass es genau dann eine Zahl b mit a* b mod n = 1 gibt, wenn der ggT von a und b = 1 ist.
Damit können wir in die o.a. Gleichung einsetzen :

$$1 = ggT(e, \varphi(n)) = x * e + y * \varphi(n)$$

und y ist nun das oben benutzte k und der Faktor x vor dem e ist das von uns gesuchte d.

Man übergebe also einer Funktion für das Verfahren von Euklid-Bachet die beiden Parameter e und phi(n), dann erhält man das d.

Damit kennen wir das ganze...

Verfahren der Schlüsselerzeugung :

1. Wähle zwei Primzahlen p und q.
2. Berechne deren Produkt p*q = n , also den modulo - Wert bei Ver- und Entschlüsselung.
3. Berechne das zugehörige φ(n) = (p-1)*(q-1).
4. Suche eine zu φ(n) teilerfremde Zahl e als Exponent der Verschlüsselung.
5. Unterscheide zwischen dem Paar e und n als öffentlichem Schlüssel und dem Paar d und n als dem geheimen Schlüssel (oder umgekehrt).
6. Vernichte alle nicht benötigten Zahlen und Unterlagen, vor allem aber p und q!

Warum ist RSA so sicher ?

Dies liegt in erster Linie daran, dass es kein (bekanntes?) Verfahren gibt, eine sehr große Zahl in ihre Primfaktoren zu zerlegen. Dadurch ist es nicht möglich, die Schlüssel erneut zu erzeugen, ohne p und q zu kennen!
Das Verfahren wirkt also wie eine Falltür: Hat man aus den Primzahlen und φ(n), die nur demjenigen bekannt sind, der die Schlüssel erzeugt, erst einmal n, e und d berechnet, lassen sich rückwärts die Zahlen p, q und φ(n) nicht mehr bestimmen.

Wie geht RSA-Verschlüsselung ?

Ganz einfach: Verwandle die Botschaft in Zahlen, berechne mit deinem Schlüsselpaar den Potenz - modulo - Wert und übermittle das Ergebnis. Der Empfänger arbeitet mit seinem Schlüsselpaar entsprechend.

Warum verschlüsselt man nicht immer mit RSA?

Das Verfahren ist leider nicht das schnellste.
So kommen wir zum nächsten : IDEA !

RSA und JAVA

Natürlich bietet ein so riesengroßes Paket wie JAVA für ein Standardverschlüsselungsverfahren wie RSA spezielle tools an. Für uns interessant ist jedoch ein Zwischenschritt, bei dem wir uns auf die Möglichkeiten beschränken, die uns die Klasse BigInteger bietet – und das ist schon fast alles, was wir benötigen!
Aus der Hilfe:
`public class BigInteger extends Number mplements Comparable<BigInteger>`
`Immutable arbitrary-precision integers` [nicht veränderbar, beliebige Genauikeit]...
`BigInteger provides analogues to all of Java's primitive integer operators, and all relevant methods from java.lang.Math. Additionally, BigInteger provides operations for modular arithmetic, GCD calculation, primality testing, prime generation, ...` [wie primitive Typen, zusätzlich modulo, ggt, Primzahl – Tests und Erzeugung]
`... All of the details in the Spec concerning overflow are ignored, as BigIntegers are made as large as necessary to accommodate the results of an operation.`
...
`Modular arithmetic operations are provided to compute residues, perform exponentiation, and compute multiplicative inverses. These methods always return a non-negative result, between 0 and (modulus - 1), inclusive.`
...
Als Konstruktor werden wir in der Regel den mit **`BigInteger(String val)`** verwenden, es gibt aber auch interessante andere.
Wichtig sind einige – keinesfalls selbstverständliche – Methoden. Neben den Standardmethoden für arithmetische Operationen add, subtract, multiply, divide, pow (Potenzieren), remainder, mod, gcd, abs, negate, equals und compareTo gibt es z.B.:
`divideAndRemainder(BigInteger val)`, die ein Array von zwei BigInteger – Zahlen zurückliefert, das an der Position 0 den Quotienten enthält und die Position 1 den Rest.
`isProbablePrime(int certainty)`, die true bzw. false zurückliefert im Sinne einer Wahrscheinlichkeitsuntersuchung[s.o.], ob das Objekt Primzahl ist.
`modPow(BigInteger exponent, BigInteger m)` potenziert [optimiert] modulo m.
`modInverse(BigInteger m)` berechnet das modulare Multiplikationsinverse [s.o.]
`probablePrime(int bitLength, Random rnd)` und **`nextProbablePrime()`** sind weitere interessante Methoden.
Beachten Sie: Alle – auch die algebraischen – Methoden werden wirklich Objekt – orientiert verwendet: Die Addition zweier BigInteger – Zahlen sieht also z.B. So aus

```
BigInteger a = new  BigInteger("12");
BigInteger b = new  BigInteger("23");
BigInteger resultat = a.add(b);
```

Eine Lösung in Java

```
import java.math.BigInteger;
import java.util.Random;

/**
 * Die Klasse RSA.
 *
 * @author (claus albowski)
 * @version (1.0)
 */
public class RSA
{
    Random random;
    int bitLength;
    BigInteger p,q,n,phi,d,e,message,code;

    /**
     * Konstruktor für Objekte der Klasse RSA
     * der auch eine Initialisierung einschließlich e und d
     * durchführt.
     */
    public RSA()
    {
        random=new Random();
        bitLength=200;
        definiereWerte();
        setzeEundD();
    }

    /**
     * gibt eine zufällige Pseudoprimzahl mit der ueergebenen Bit-Länge
zurück.
     */
    public BigInteger gibZufallsPrimzahl(int bitLength){
        return BigInteger.probablePrime(bitLength, random);
    }

    /**
     * wählt zunächst zwei (Pseudo-)Primzahlen p und q, berechnet n und
phi.
     */
    public void definiereWerte(){
        p=gibZufallsPrimzahl(bitLength);
        q=gibZufallsPrimzahl(bitLength);
        n=p.multiply(q);
        phi=(p.subtract(BigInteger.ONE)).multiply(q.subtract(BigInteger.O
NE));
    }

    /**
     * setzt e und dazu d.
     */
```

```
    public void setzeEundD(){
        e=gibZufallsPrimzahl(2*bitLength+1).mod(phi);
        d=e.modInverse(phi);
    }

    /**
     * verschluesseleRSA
     */
    public String verschluesseleRSA(){
        code=message.modPow(e, n);
        return code.toString();
    }

    /**
     * setzt die Botschaft.
     */
    public void setzeBotschaft(String botschaft){
        message=new BigInteger(botschaft);
    }

    /**
     * setzt den Code.
     */
    public void setzeCode(String codeString){
        code=new BigInteger(codeString);
    }

    /**
     * entschluesseleRSA
     */
    public String entschluesseleRSA(){
        message=code.modPow(d, n);
        return message.toString();
    }
}
```

RSA Beispielrechnung

Ein Beispiel mit sehr kleinen Zahlen:
Wir wählen als Primzahlen die beiden Zahlen 13 und 17.
Aus **p = 13** und **q = 17** erhält man als Produkt **n = p * q = 221**
und **φ(n) = (p-1)*(q-1) = 192.**

Nun suchen wir eine Primzahl e > φ(n), entscheiden uns für 197, allerdings verwenden wir diese Zahl modulo φ(n), benutzen also die 5 [=197-192].
Als Multiplikationsinverses finden wir 77 (vielleicht durch Probieren, sonst s.o.). Wegen der etwas einfacheren Rechnung tauschen wir e und d aus, so dass wir die Beispielbotschaft **m = 3** zur Chiffrierung mit 77 potenzieren und zur Dechiffrierung dann mit 5 arbeiten können. Die Rechnung mit den Nebenrechnungen:

$$
\begin{aligned}
& 3^{77} \ mod\, 221 \\
&= [3 \cdot 3^{76}] \ mod\, 221 \\
&= [3 \cdot 9^{38}] \ mod\, 221 \\
&= [3 \cdot 81^{19}] \ mod\, 221 && 6561 = 221 \cdot 29 + 152 \\
&= [3 \cdot 81 \cdot 81^{18}] \ mod\, 221 && also\ Ersetzen\ von\ 6561\ durch\ 152 \\
&= [3 \cdot 81 \cdot 6561^{9}] \ mod\, 221
\end{aligned}
$$

Die Rechnung kann auch bei der $3 \cdot 81 = 243$ schon durch modulo 221 zu 22 vereinfacht werden, so dass es nun weiter gehen kann mit:

$$
\begin{aligned}
& [22 \cdot 152 \cdot 152^{8}] modulo\, 221 && 22 \cdot 152 = 3344 = 221 * 15 + 29\ und\ 152 * 152 = 23104 = 221 * 104 + 120 \\
&= [29 \cdot 120^{4}] modulo\, 221 && 120 * 120 = 14400 = 221 * 65 + 35 \\
&= [29 \cdot 35^{2}] modulo\, 221 && 35 * 35 = 1225 = 221 * 5 + 120 \\
&= [29 \cdot 120] modulo\, 221 \\
&= 3480\ modulo\, 221 && 3480 = 221 * 15 + 165 \\
&= 165
\end{aligned}
$$

Unsere Codezahl ist also **m' = 165**.

Zum Test führen wir auch die Decodierung durch.

$$
\begin{aligned}
& [165^{5}] modulo\, 221 \\
&= [165 \cdot 165^{4}] modulo\, 221 && 165 * 165 = 27225 = 221 * 123 + 42 \\
&= [165 \cdot 42^{2}] modulo\, 221 && 42 * 42 = 1764 = 221 * 7 + 217 \\
&= [165 \cdot 217] modulo\, 221 && 165 * 217 = 35805 = 221 * 162 + 3 \\
&= 3
\end{aligned}
$$

Wie gewünscht, erhalten wir nun die ursprüngliche Botschaft **3** zurück.