BigInteger – nicht nur für RSA

Der Datentyp BigInteger ist eine für RSA wesentliche Erweiterung der elementaren Datentypen. Erzeugt man BigInteger – Objekt, kann man unter BlueJ erkennen, dass er offensichtlich intern durch ein int[], also ein Array von elementaren int - Zahlen realisiert ist. Prinzipiell sollte das allerdings belanglos sein, wenn man mit einem BigInteger – Objekt arbeitet: Im Sinne der Kapselung sollte der intern verwendete eigentlich nicht einmal erkennbar sein.
Der Datentyp BigInteger bringt viele Methoden für die RSA – Verschlüsselung mit, so dass sie nicht erst von uns geschrieben werden müssen. Darunter ist auch die eigentliche Verschlüsselungsfunktion **`message.modPow(e, n)`**, die wir als „schnelle Potenz modulo“ bezeichnet haben und bei der wir uns überzeugen konnten, warum sie so heißt.
Weiterhin gibt es Funktionen zum Generieren von Primzahlen, wobei die Dokumentation leider unklar lässt, in welchem Sinne mit **`BigInteger.probablePrime(bitLength, random)`** generierte Zahlen Primzahlen sind. Da es seit 2003 einen Algorithmus mit polynomialer Laufzeit geben soll, mit dem man eine Zahl daraufhin testen kann, ob sie Primzahl ist oder nicht, fragt man sich natürlich, ob er hier genutzt wird.
Für das Generieren des Schlüssels ist außerdem **`e.modInverse(phi)`** wichtig, da wir gesehen haben, dass die Bestimmung des modularen Inversen nicht elementar ist.
Interessant ist, wie man eine Funktion wie modPow selbst realisieren kann und warum sie so viel schneller als die einfache Potenzfunktion ist.

```
// einfache Standardberechnung der Potenz.
public BigInteger doofePotenzModulo(BigInteger basis,
                                    BigInteger hochzahl, BigInteger modulo){
      potenz=BigInteger.ONE;
      for (BigInteger i=hochzahl;
            !i.equals(BigInteger.ZERO);
            i=i.subtract(BigInteger.ONE)) {
                  potenz=(potenz.multiply(basis)).remainder(modulo);
      }
      return potenz;
}
```

In der vorliegenden Version wird von der vorgegebenen Hochzahl einfach herunter gezählt und dabei der aktuelle Wert von **`potenz`** (hier sinnvoller so benannt als die Benennung mit **`basis`**, wie im Kurs erarbeitet) jeweils mit **`basis`** multipliziert. Diese Berechnung ist selbst bei großem Wert von **`basis`** und **`modulo`** schnell, ist aber sehr empfindlich gegen eine Erhöhung von **`hochzahl`**. Wie gravierend das ist, wird allerdings nicht deutlich, wenn man **`hochzahl`** verdoppelt, sondern erst dann, wenn man die Zahl der Stellen der Hochzahl vergrößert: Mit jeder weiteren dezimalen Stelle der Hochzahl wächst der Berechnungsaufwand mit einem Faktor 10 an, er hängt von der Stellenzahl der Hochzahl daher exponentiell ab! Wie man bei Laufzeituntersuchungen feststellen kann, führt dies bei längeren Zahlen dazu, dass keine Berechnung mehr zu einem Ergebnis kommt.

Logarithmisch ist besser als exponentiell

Bei der schnellen Variante geht man grundsätzlich anders vor. Keine Rolle – wenn überhaupt eine nachteilige – spielt die Tatsache, dass wir diese Methode rekursiv arbeiten lassen. Die rekursive Variante erleichtert allerdings in einem großen Maße die Lesbarkeit.
Der Kern der Verbesserung ist aber ein anderer:
Bei jedem Berechnungsschritt wird zunächst geprüft, ob **`hochzahl`** gerade ist. Wenn das

zutrifft, wird mit dem Quadrat von **basis** und der Hälfte der **hochzahl** weiter gearbeitet. Hat man eine ungerade Zahl, wird zusätzlich das Ergebnis dieser neu definierten Berechnung noch einmal mit der aktuellen **basis** multipliziert.
Die Methode sieht in der optimierten Variante dann so aus:

```
// optimierte schnelle Potenz modulo
public BigInteger schnellePotenzModulo(BigInteger basis,
                                       BigInteger hochzahl,
                                       BigInteger modulo){
      if (hochzahl.equals(BigInteger.ZERO)) return BigInteger.ONE;
      else if (hochzahl.testBit(0))
      return
            basis.multiply(
                  optimierteSchnellePotenzModulo(
                        basis.multiply(basis).remainder(modulo),
                        hochzahl.shiftRight(1),
                        modulo)).remainder(modulo) ;
      else return
            optimierteSchnellePotenzModulo(
                  basis.multiply(basis).remainder(modulo),
                  hochzahl.shiftRight(1),
                  modulo).remainder(modulo) ;
}
```

letztes bit 1 ?

Optimiert ist sie dadurch, dass nach Möglichkeit schnelle Methoden von BigInteger zur Berechnung der Zwischenwerte verwendet werden. Zum Halbieren verwenden wir die Methode **hochzahl.shiftRight(1)**, bei der nur in der Zahl alle bits um eine Stelle nach rechts verschoben werden. Dabei brauchen wir nicht darauf zu achten, dass in der letzten Stelle möglicherweise eine 1 steht. Das wird vorher abgeprüft und durch das zusätzliche Multiplizieren mit der **basis** berücksichtigt.
Dass – und warum – diese Methode für größere Zahlen sehr viel schneller sein muss als die vorher berachtete, wird bei der Erläuterung dieser Funktion deutlich:
Die Zahl der Berechnungsschritte hängt nicht mehr exponentiell von der Hochzahl ab, sondern davon, wie oft diese **shiftRight** – Methode aufgerufen wird, also für jedes bit der vorgegebenen **hochzahl** genau einmal! Damit können wir auch für sehr große Hochzahlen den Berechnungsaufwand schnell abschätzen. Eine Zahl mit 300 dezimalen Stellen beispielsweise – sie ist also etwa 10^{300} groß – braucht wegen $2^{10} = 1024 \approx 10^3$ für 3 dezimale Stellen ziemlich genau 10 duale Stellen (bit), bei 300 dezimalen also hundert mal so viele: $100 \cdot 10 = 1000$ bit hat also die Zahl und das bedeutet, dass die Funktion 1000-mal ausgeführt wird! Dies liegt immense Größenordnungen unter der Zahl von 10^{300} für die Standardfunktion.
Der Berechnungsaufwand hängt logarithmisch (Zweier-Logarithmus) von der Stellenzahl ab und das führt hier dazu, dass diese Berechnung auch für große Hochzahlen noch (relativ) schnell ist. Wir müssen davon ausgehen, dass in JAVA die Methode **modPow** genau so realisiert wurde.

Rekursion und Iteration

Ein interessanter Vergleich ergibt sich bei der vergleichenden Betrachtung einer iterativen mit einer rekursiven Lösung (derselben) optimierten Berechnung.
Bei der iterativen Version verwenden wir zusätzlich einen Akkumulator für die Zwischenberechnungen. Für die Basis und die Hochzahl können wir die übergebenen Parameter verwenden. Sie werden dabei zwar verändert, das betrifft aber nur ihren internen Wert

innerhalb dieser Methode.

```
/**
 * iterative Version.
 */
public BigInteger iterativePotMod(BigInteger basis,
                                  BigInteger hochzahl,
                                  BigInteger modulo) {
      BigInteger akku=BigInteger.ONE;
      while (hochzahl.compareTo(BigInteger.ZERO)>0){
            if (hochzahl.testBit(0))
                  akku=akku.multiply(basis).remainder(modulo);
            basis=basis.multiply(basis).remainder(modulo);
            hochzahl=hochzahl.shiftRight(1);
      }
      return akku;
}
```

Es empfiehlt sich, einmal beide Versionen mit einfachen Zahlen durchzuspielen, um den unterschiedlichen Datenfluss bei trotzdem gleichen Ergebnissen zu verstehen.
Jeweils eine Tabelle der lokalen Datenwerte hilft beim Verständnis.

rekursiv:

basis	hochzahl	modulo	ergebnis	zus. Faktor?
26	21	1001	741	26
676	10		221	1
520	5		221	520
130	2		884	1
884	1		884	884

iterativ:

basis	hochzahl	modulo	akku	zus. Faktor?
26	21	1001	26	26
676	10		26	1
520	5		507	520
130	2		507	1
884	1		741	884