Primfaktorzerlegung (Faktorisierungs-Algorithmen): Ac 8.7.2024

Die Primfaktorzerlegung hat heutzutage eine große Bedeutung erlangt, weil man damit Verschlüsselungen realisieren kann (**Kryptologie**). Es ist nämlich für große Zahlen mit mehreren 100 Stellen sehr schwer, sie in ihre Primfaktoren zu zerlegen. Auch Supercomputer beißen sich da die Zähne aus. Es wundert daher nicht, dass die Bestimmung der Primfaktorzerlegung einer natürlichen Zahl n deutlich aufwendiger ist als der bloße Primzahltest.

Probedivision („Brute-force-Methode“):

Man prüft für alle Primzahlen (2;3;5;...), ob sie n teilen. Ist dies für eine Primzahl p der Fall, so wird n ersetzt durch n div p (div ist die ganzzahlige Division). Das Ergebnis n: = n div p wird weiter durch p dividiert. Geht die Division auf, so wird wieder n durch n div p ersetzt usw. Andernfalls wird die nächste Primzahl ausprobiert. Dies führt man durch bis zur Quadratwurzel des ursprünglichen n (entsprechend der Vorgehensweise beim Primzahltest) oder bis n=1 ist. Die Primzahleigenschaft p=6i-1 oder p=6i+1 ab i=1 sollte hier aus Geschwindigkeitsgründen ebenfalls Anwendung finden.

Beispiel: n = 12
12 div 2 = 6, also ist **2** ein Primfaktor von 12. Ersetze 12 durch 12 div 2 = 6.
6 div 2 = 3, also ist **2** ein weiterer Primfaktor von 12. Ersetze 6 durch 6 div 2 = 3.
3 div 2 = 1 Rest 1; 2 ist kein weiterer Primfaktor von 12.
Die nächste zu überprüfende Primzahl ist 3:
3 div 3 = 1, also ist **3** ein Primfaktor von 12. Ersetze 3 durch 3 div 3 = 1.
Die Zahl 1 enthält keine weiteren Primfaktoren, daher erfolgt nun der Programmabbruch !
Ergebnis: 12 = 2·2·3

JAVA-Methode (optimiert):

```
public static String primfaktorZerlegung(long n) {
	String s = "";
	long max = (long) Math.sqrt(n);
	while (n % 2 == 0) {
		s = s + "2";
		n = n / 2;
		if (n > 1) s = s + "*";
		else return s;
	}
	while (n % 3 == 0) {
		s = s + "3";
		n = n / 3;
		if (n > 1) s = s + "*";
		else return s;
	}
	while (n % 5 == 0) {
		s = s + "5";
		n = n / 5;
		if (n > 1) s = s + "*";
		else return s;
	}
	long primfaktor = 7, increment = 4, letzteZahl = n;
	boolean letzteZahlprim = true;
	while (n != 1 && primfaktor <= max) {
		if (n % primfaktor == 0) {
			s = s + primfaktor;
			n = n / primfaktor;
			letzteZahlprim = false;
			letzteZahl = n;
			if (n > 1) s = s + "*";
		} else {
			primfaktor = primfaktor + increment;
			increment = 6 - increment;
			letzteZahlprim = true;
		}
	}
	if (letzteZahlprim) s = s + letzteZahl;
	return s;
}
```

Eine sehr viel effizientere Methode zur Primfaktorzerlegung ist das **Abdividieren** mithilfe einer vorher (z.B. durch das "Eratosthenes-Sieb") erzeugten Primzahlliste.

Die zu untersuchende Zahl n wird hierbei durch die Zahlen der Primzahlliste geteilt, wobei nur solche möglichen Teiler untersucht werden, deren Quadrat n nicht übertreffen.
Außerdem wird noch überprüft, ob die nach dem Abdividieren verbleibende Zahl prim ist !

```
public static String sPrimfaktorenAbdividieren(long n, ArrayList <Integer>
                                                          primZahlenListe) {
   String s = "";
   int teiler;
   int lenListe = primZahlenListe.size();
   int teilerMax = primZahlenListe.get(lenListe-1);
   int  wurzel_n = (int)Math.sqrt(n);
   if (wurzel_n < teilerMax)
      teilerMax = wurzel_n;
   for (int i = 0; i < lenListe; i++) {
      teiler = primZahlenListe.get(i);
      while (n % teiler == 0) {
         s = s + Integer.toString(teiler) + "*";
         n = n / teiler;
      }
      if (n == 1)
         break;
      else if (teiler * teiler > n) {
         s = s + Long.toString(n);
         break;
      }
      else if (teiler >= teilerMax )
         if (istPrimMillerRabin(n))
            s = s + Long.toString(n);
         else
            s = s + " ?" + Long.toString(n) + "? "; // zerlegbare Zahl in ?...? setzen
   }
   if (s.endsWith("*"))
       s = s.substring(0, s.length()-1); // eventuelles * am Ende löschen
   return s;
}
```

Beispiel: primZahlenListe = { 2, 3, 5, 7, 11, 13, 17, 19} , also lenListe = 8 teilerMax = 19

1. Fall: **n = 630** wurzel_n = 25 , daher gilt: wurzel_n > teilerMax

i = 0: teiler = 2 630 mod 2 = 0 , also s = "2·" n = 630 div 2 = 315

i = 1: teiler = 3 315 mod 3 = 0 , also s = "2·3·" n = 315 div 3 = 105

105 mod 3 = 0 , also s = "2·3·3·" n = 105 div 3 = 35

i = 2: teiler = 5 35 mod 5 = 0 , also s = "2·3·3·5·" n = 35 div 5 = 7

i = 3: teiler = 7 7 mod 7 = 0 , also s = "2·3·3·5·7·" n = 7 div 7 = 1

n = 1 , also Abbbruch !

Ergebnis: **s = "2·3·3·5·7"**

2. Fall: **n = 2037** wurzel_n = 45 , daher gilt: wurzel_n > teilerMax

i = 0: teiler = 2 2037 mod 2 ≠ 0 ; 2 ist kein Teiler

i = 1: teiler = 3 2037 mod 3 = 0 also s = "3·" n = 2037 div 3 = 679

i = 2: teiler = 5 679 mod 5 ≠ 0 ; 5 ist kein Teiler

i = 3: teiler = 7 679 mod 7 = 0 , also s = "3·7·" n = 679 div 7 = 97

teiler² > teilerMax , also s = "3·7·97" ; Abbruch

Ergebnis: **s = "3·7·97"**

Eine andere Methode zur Faktorzerlegung ist das

Verfahren von **Fermat** :

Dieses Verfahren wurde 1643 von FERMAT am Beispiel 2027651281 = 44021·46061 beschrieben.

Es ist Grundlage für das wesentlich leistungsfähigere Zerlegungsverfahren **„Quadratisches Sieb“** .

Zugrunde liegende Idee:

Wenn man die zu zerlegende Zahl n als Differenz zweier Quadrate [$n = a^2 – b^2$] darstellen kann, erhält man mithilfe der 3. Binomischen Formel: $n = a^2 – b^2 = (a + b)(a – b)$. Somit hat man n faktorisiert !
Voraussetzung: **n muss ungerade** sein, damit die Zerlegung klappt !

Wie findet man die ganzen Zahlen a und b ?

Zunächst wird $n = a^2 – b^2$ umgestellt zu $b^2 = a^2 – n$.
Man sucht also ganzzahlige a derart, dass $a^2 – n$ ein Quadrat b^2 ergeben (Probiermethode).

Genauer:
Ausgehend von $a_0 = [\sqrt{(n)}]$ berechnet man nacheinander die Quadrate der Zahlen
a_0 , $a_0 + 1$, $a_0 + 2$, … , und subtrahiert davon n, bis man als Ergebnis wieder ein Quadrat hat !

Ohne Beweis: Eine Obergrenze für die $a_0 + k$ ist $(n+9) / 6$.

Algorithmus:

für a von $\lceil \sqrt{n} \rceil$ bis $\frac{n+9}{6}$ wiederhole :
  $z = a^2 - n$
  falls z Quadratzahl ist
    dann sind $a + \sqrt{z}$ und $a - \sqrt{z}$ Teiler von n

Beispiel **1**: n = 561 = 17·33

Die Quadratwurzel aus 561 ist $a_0 = 23, 685 ...$. Wir beginnen also mit 24 ! (Obergrenze = 95)
$24^2 – 561 = 15$ (kein Quadrat)
$25^2 – 561 = 64 = 8^2$ Somit ist eine Zerlegung gefunden, nämlich 561 = (25 - 8)(25 + 8) = 17·33 .
Achtung: Ersichtlich müssen die beiden gefundenen Faktoren nicht zwingend prim sein !

Beispiel **2**: n = 1593 = 27·59

Die Quadratwurzel aus 1593 ist $a_0 = 39, 912 ...$. Wir beginnen also mit 40 ! (Obergrenze = 267)
$40^2 – 1593 = 7$ (kein Quadrat)
$41^2 – 1593 = 88$ (kein Quadrat)
$42^2 – 1593 = 171$ (kein Quadrat)
$43^2 – 1593 = 256 = 16^2$ Somit ist eine Zerlegung gefunden, nämlich 1593 = (43 - 16)(43 + 16) = 27·59 .

JAVA-Methode :

```
public static long[] primfaktorZerlegungFermat(long n) {
    // Fermat-Methode; meist recht langsam
    if (n % 2 != 0) { // nur für ungerade n weiterrechnen !
       long grenze = (n+9)/6;
        long a0 = (long) Math.ceil(Math.sqrt(n));
        long aQuadminusn, wurzel;
        for (long a = a0; a <= grenze; a++) {
               aQuadminusn = a * a - n;
               if (istEvtlQuadratisch(aQuadminusn)) {
                     wurzel = (long) Math.sqrt(aQuadminusn);
                     if (wurzel * wurzel == aQuadminusn)
                          return new long[] {a - wurzel, a + wurzel};
               }
        }
    }
    return new long[] {0L, 0L}; // kein Ergebnis
}

public static boolean istEvtlQuadratisch(long testZahl) {
    // Quadratische Zahlen lassen bei Division durch 8 den Rest 0 oder 1 oder 4
    long rest = testZahl % 8;
    return rest == 0 || rest == 1 || rest == 4;
}
```

Nachteil der FERMAT-Methode:

Bei weit voneinander entfernt liegenden Faktoren dauert das Verfahren sehr lange, weil viele Schritte durchzuführen sind !

Eine **Verbesserung** ist die nachfolgend beschriebene **LEHMAN**-Methode !

LEHMAN-Methode zur Primfaktorzerlegung von n :

Die Methode liefert 2 Faktoren, die nicht notwendig prim sind. Vorausgesetzt ist: n **ungerade** !
Zunächst wird eine **Probedivision** bis zur Schranke **k =** $\sqrt[3]{}$**(n)** durchgeführt.
Findet sich ein Teiler t von n , so wird der Algorithmus beendet und die Faktoren sind t und n div t .
Falls aber n in diesem Bereich keine Teiler besitzt, so ist n prim oder das Produkt zweier Primzahlen .
Dann wird folgender Algorithmus von R.S.LEHMAN (vorgestellt 1974) durchgeführt:

für k von 1 bis $\left\lfloor \sqrt[3]{n} \right\rfloor$ wiederhole :

für x von $\left\lceil \sqrt{4 \cdot k \cdot n} \right\rceil$ bis $\left\lfloor \sqrt{4 \cdot k \cdot n} + \frac{\sqrt[6]{n}}{4\sqrt{k}} \right\rfloor$ wiederhole :

$z = x^2 - 4 \cdot k \cdot n$

falls z Quadratzahl ist

dann sind $ggT(x + \sqrt{z}\ , n)$ und $ggT(x - \sqrt{z}\ , n)$ echte Teiler von n

falls kein Teiler gefunden wurde, dann ist n prim

JAVA-Methode (verwendet das Abdividierverfahren, so dass für „Lehman“ eine ungerade Zahl bleibt):

```
public static String sPrimfaktorZerlegungLehman(long n) {
   // Lehman-Verfahren:  Erweiterung der Fermat-Zerlegung
   int grenze = (int)(Math.cbrt(n));  // 3.Wurzel von n
   ArrayList <Integer> primZahlenListe = eratosthenesListe(grenze);
   System.out.println("Probedivision:");
   String s = sPrimfaktorenAbdividieren(n, primZahlenListe);
   if (s.indexOf('*') == -1) {  // keine Zerlegung bei der Probedivision gefunden
      System.out.println("Probedivision findet keinen Teiler !");
      s = ""; // Voraussetzung für LEHMAN-Verfahren erfüllt !
   }
   else {
      System.out.print("Abdividiert: ");
      return s;  // Lösung durch Probedivision
   }

   System.out.println("Starte LEHMAN-Algorithmus:");
   int xStart, xEnde;
   long z = 0, wurzelz = 0, x = 0;
   double nHoch1sechstel = Math.cbrt(Math.sqrt(n));

   for (int k = 1; k <= grenze; k++) {
      double wurzel4kn = Math.sqrt(4*k*n);
      xStart = (int) Math.ceil(wurzel4kn);
      xEnde = (int) (wurzel4kn + nHoch1sechstel / 4.0 / Math.sqrt(k));
      System.out.println("xStart = " + xStart + "   xEnde = " + xEnde);
      for (x = xStart; x <= xEnde; x++) {
         z = x*x - 4*k*n;
         if (istEvtlQuadratisch(z)) {
            wurzelz = (long)Math.rint(Math.sqrt(z));
            if (wurzelz * wurzelz == z) {
               System.out.println("k = " + k + " x = " + x + " z = " + z +
                                                  "  wurzel(z) = " + wurzelz);
               long erg = ggT(x - wurzelz, n);
               s = s + Long.toString(erg);
               erg = ggT(x + wurzelz, n);
               return s + "·" + Long.toString(erg);
            } // if
         } // if
      } // for x
   } // for k
   return s;
}
```

Beispiel **1**: n = 1147 (= 31·37) ;
die Probedivision läuft hier bis zu dem Teiler 10 , ohne Ergebnis !

für k = 1 gilt:
x = 68 z = 68² - 4588 = 36 z ist Quadratzahl von 6
dann sind ggT(68 + √(36), 1147) = 37 und ggT(68 - √(36), 1147) = 31 echte Teiler von 1147 .
Also ist 1147 = 37·31 eine Zerlegung

Beispiel **2**: n = 2027651281 (= 44021·37) ;
die Probedivision läuft hier bis zu dem Teiler 1265 , ohne Ergebnis !

für k = 1 gilt: x = 90059 bis x = 90067
x = 90059 z = 90059² - 2027651281 = 6082972200 z ist keine Quadratzahl
auch für die anderen x-Werte findet man keine Quadratzahlen

für k = 2 gilt: x = 127363 bis x = 127368
bei allen diesen x-Werten findet man keine Quadratzahlen
...

Man muss warten bis zu k = 462 und x = 1935743:
z = 1394761 ist Quadratzahl von 1181
Somit sind ggT(1935743 + 1181, 1147) = 46061 und ggT(1935743 - 1181, 1147) = 44021 echte Teiler von 2027651281 . Also ist 2027651281 = 46061· 44021 eine Zerlegung

Eine bessere (schnellere) Methode zur Faktorzerlegung ist die

Rho-Methode von Pollard :

Ein Beispiel: n = 143 = 11·13

Die Rho-Methode wendet eine Funktion f der Form $f(x) = (x^2 + c) \bmod n$ an, die eine Pseudozufallszahl liefert. Durch fortwährende Anwendung der Funktion f auf ihre Ergebnisse $f(x_{n+1}) = f(f(x_n))$ wird eine ganze Folge von Zufallszahlen generiert, die wegen der Modulo-Division eine Periode haben muss.

Für n = 143 und z.B. x0 = 5 c = 37 ergibt sich eine Periodenlänge von 6 (Vorperiodenlänge = 3):
xk = 5 62 20 **8 101 85 112 140 46** 8 101 85 112 140 46 8 101 85 112 140

Wendet man die gleiche Funktion f auf einen der Primfaktoren, etwa p = 11 an, so erhält man:
ak = 5 7 9 **8 2** 8 2 8 2 8 2 8 2 8 2 8
Die Periodenlänge beträgt hier **2** (bei gleicher Vorperiodenlänge 3).

Aus der letztgenannten Tatsache folgert man, dass $x_{k+2} - x_k$ den Faktor p = 11 enthält.
In der Tat findet man $x_5 - x_3 = 85 - 8 = 77 = 7 \cdot 11$. Entsprechend $x_6 - x_4 = 112 - 101 = 11 = 1 \cdot 11$.
Für jedes dieser Beispiele gilt folglich auch $ggT(x_{k+2} - x_k , n) = 11$.
Für x_k-Paare mit anderem Index-Unterschied gilt übrigens immer $ggT(x_k - x_i , n) = 1$.

Aus der x_k-Folge findet man die Paare mit $ggT(x_k - x_i , n) \neq 1$ mit einem Trick (Floyd):
Man berechnet neben der x_k-Folge noch eine zweite Folge x_{2k} , d.h. es wird nur jeder zweite Wert der Folge x_k berechnet. Somit ist die zweite Folge schneller als die erste, und man muss nur immer wieder $ggT(x_{2k} - x_k , n)$ berechnen. Irgendwann ist $x_{2k} = x_{k+pL}$ (pL=Periodenlänge der p-Folge) und der ggT ist gleich einem der gesuchten Primfaktoren. Dabei muss nicht unbedingt der kleinste der Primfaktoren als erster gefunden werde, ja es muss nicht einmal ein Primfaktor sein, sondern es kann auch nur irgend ein Teiler von n sein .

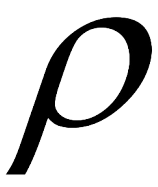
Für das obige Beispiel mit n = 143 läuft das Verfahren folgendermaßen ab:

```
xk  =  5 62  20   8 101  85 112 140 46 8 101
x2k =  5 20 101 112  46 101
```

Es gilt ggT(112-8,143) = ggT(104,143) = 13 .
Also hat man hier den größeren der beiden Primfaktoren gefunden !

Anmerkung:

Das periodische Verhalten der Folge gab der Rho-Methode ihren Namen, da man sich die Periode wie einen Kreis vorstellen kann und die Folgenglieder am Anfang wie einen Stängel, der in den Kreis hineinführt. Graphisch sieht das aus wie der griechische Buchstabe ρ.



Algorithmus (Pollard-Rho; ermittelt **einen** Teiler):

Gegeben seien:

- Die zu zerlegende (ungerade) Zahl n
- Eine Funktion g(x), z.B. $g(x) = (x^2 + c) \bmod n$; c = -2; c = 0 sollten vermieden werden !!
- Ein Startwert x_o für x .

```
x = x0
y = x
divisor = 1
solange divisor = 1 wiederhole
      x = g(x)
      y = g(g(y))
      divisor = ggT(| x – y | , n)
ende solange
ausgabe(divisor)
```

Hinweise:

1) Falls der divisor = n, dann wiederhole den Algorithmus mit anderem c bzw. x_o .

2) Zur Zerlegung von Fermatzahlen $F_k = 2^{(2^k)} + 1$ empfiehlt **Richard P. Brent** die Verwendung von $g(x) = (x^{(2^{(k+2)})} + 1) \bmod F_k$ und $x_0 = 3$.

JAVA-Methode (vorausgesetzt: n sei ungerade) :

```
public static long rho(long N) {
    long divisor;
    Random rand = new Random();
    long c = Math.abs(rand.nextLong()) % N;  // zufzahl in [0..N-1]
    long x = Math.abs(rand.nextLong()) % N;
    long y = x;
    do {
        x = (x * x + c) % N;
        y = ( y * y + c) % N;
        y = ( y * y + c) % N;
        divisor = ggT(x - y, N);
    } while (divisor == 1);
    return divisor;
}

public static long ggT(long a, long b) {
    long tmp;
    a = Math.abs(a);
    b = Math.abs(b); // da ggT immer > 0 !!
    while (b > 0) {
        tmp = a % b;
        a = b;
        b = tmp;
    }
    return a;
}
```

Pollard-Rho: Beispielrechnung für n = 341:

```
c = 22  x = 252  xquadr = 252
x = (252*252+22)%341 = 63526%341 = 100
xquadr = 100     xquadr = (100*100+22)%341 = 10022%341 = 133
divisor = ggT(100-133, 341) = ggT(-33, 341) = 11 = gesuchter Faktor !
Primfaktorzerlegung von 341 = 11*31
```

Pollard-Rho: Beispielrechnung für n = 1073741823:

```
c= 394261293  x= 682806759
x= 473509515  xquadr= 240021111       divisor= 279  ( = 3 * 3 * 31 )
c= 76  x= 234
x= 148  xquadr= 218                   divisor= 1
x= 218  xquadr= 239                   divisor= 3
c= 13684341  x= 132608568
x= 276515673  xquadr= 225268035       divisor= 3
c= 70884621  x= 58719382
x= 33848693  xquadr= 23944366         divisor= 1
x= 23944366  xquadr= 2618003          divisor= 1
x= 55615084  xquadr= 119069118        divisor= 7
c= 13583729  x= 2535019
x= 7146956  xquadr= 4344211           divisor= 11
c= 413404  x= 635240
x= 120753  xquadr= 193492             divisor= 1
x= 193492  xquadr= 421148             divisor= 1
x= 1149475  xquadr= 808968            divisor= 1
x= 421148  xquadr= 901630             divisor= 151
c= 10084  x= 5584
x= 7961  xquadr= 5408                 divisor= 1
x= 5408  xquadr= 7934                 divisor= 1
x= 2437  xquadr= 1449                 divisor= 1
x= 7934  xquadr= 781                  divisor= 1
x= 7205  xquadr= 9195                 divisor= 1
x= 1449  xquadr= 6988                 divisor= 1
x= 6180  xquadr= 6986                 divisor= 31
Primfaktorzerlegung von 1073741823 = 3*3*7*11*31*151*331
```

Eine Verbesserung des Pollard-Rho-Verfahrens stellt das Verfahren von **R.P.Brent** (s.o.) dar !!

Weitere Beispiele für Pollard-Rho und Brent-Rho (zerlegt mit eigenem Java-Programm) :

wechselnde Rechenzeiten wegen probabilistischer Eigenschaften dieser Verfahren !!

```
856633869006220875991877574358267 63 = 48616941244147 · 1762006919983579059529
rho: 10 sec     brent: 4 sec

944985070106979751360865000279913963061930000000000000000000000000000000000000000000000000000000000000
0000000000000944985070106979751360865000279913963061 93 =
73·137·401·1201·1601·1676321·9485933·100100051·100100051·994205803·1000000007·596484808
1· 1296944190290577505513857711845642744990757009476567578215372915271968 01
rho: 1,1 sec     brent: 0,3 sec
```

Eine weitere auf **Pollard** zurückzuführende Methode ist die

p-1 - Methode :

Die p-1 – Methode ist nur dann effektiv, wenn n wenigstens einen Primfaktor p hat, für den p-1 nur kleine Faktoren hat ; daher auch der Name .

Beispiel: n = 6994241 = 2081·3361 .
2080 = 2·2·2·2·2·5·13 und sogar 3360 = 2·2·2·2·2·3·5·7 (nur kleine Faktoren !)

Algorithmus (Pollard p-1 ; ermittelt **einen** Teiler):

Gegeben seien die zu zerlegende (**ungerade**) Zahl n und eine Grenze B .

```
Bmax = 900000   B0 = 5
a = 2    B = B0

wiederhole
    k = kgV(2,3, … ,B)          // kleinstes gemeinsames Vielfaches
    x = a^k mod n
    d = ggT(x-1, n)

    falls  1 < d < n:   return d              // Teiler gefunden
    falls  d = 1:       erhöhe B              //  B = B * 3
    falls  d = n:       erhöhe a um 1  und  setze B=B0

solange B < Bmax

return 0                        // keinen Teiler gefunden
```

Beispiel 1: n = 1001 Setze B = 5 , dann gilt: k = kgV(2, .. ,5) = 60

B	**k**	**a**	**x = a^k mod n**	**d = ggT(x-1,n)**	
5	60	2	1	1001	
5	60	3	1	1001	
5	60	4	1	1001	
5	60	5	1	1001	
5	60	6	1	1001	
5	60	7	287	**143**	**Teiler gefunden**

Beispiel 2: n = 91 Setze B = 5 , dann gilt: k = kgV(2, .. ,5) = 60

B	**k**	**a**	**x = a^k mod n**	**d = ggT(x-1,n)**	
5	60	2	1	91	
5	60	3	1	91	
5	60	4	1	91	
5	60	5	1	91	
5	60	6	1	91	
5	60	7	14	**13**	**Teiler gefunden**

Beispiel 3: n = 6994241 Setze B = 5 , dann gilt: k = kgV(2, .. ,5) = 60

B	**k**	**a**	**x = a^k mod n**	**d = ggT(x-1,n)**	
5	60	2	3598787	1	
15	360360	2	5256605	**3361**	**Teiler gefunden**

Beispiel 4: n = 2199023255551 Setze B = 5 , dann gilt: k = kgV(2, .. ,15) = 360360

B	**k**	**a**	**x = a^k mod n**	**d = ggT(x-1,n)**
5	60	2	524288	1

15	360360	2	2048	1
45	9419588158802421600	2	1	2199023255551 (= n)
5	60	3	622942850755	1
15	360360	3	373912776357	1
45	9419588158802421600	3	1386971536374	1
135	174934204792066…	3	935012120683	1
405	239268433282037…	3	1892390451106	**13367 = Teiler gefunden**

Java-Implementierung mit BigIntegers:

```
public static BigInteger pollardPMinus1(BigInteger nBig) { // ermittelt 1 Faktor
   BigInteger aBig, kBig, dBig, xBig;
   int B0 = 5, Bmax = 900_000;
   aBig = BigInteger.TWO;    // Basis festlegen
   int B = B0;

   do {
      kBig = kgVbisZu(B); // Exponent k festlegen
      xBig = aBig.modPow(kBig, nBig);  // a^k mod n
      dBig = nBig.gcd(xBig.subtract(BigInteger.ONE)); // ggT von a^k-1 und n

      System.out.println("B = " + B + "   a = " + aBig.toString()
      + "   x = " + xBig.toString() + "   d = " + dBig.toString());

      if (dBig.compareTo(BigInteger.ONE) > 0 && dBig.compareTo(nBig) < 0) {
         System.out.println("Erfolg !");
         return dBig; // Teiler gefunden
      }

      if (dBig.equals(BigInteger.ONE))
            B *= 3;
      else { // d = n
         aBig = aBig.add(BigInteger.ONE);
         B = B0;
      }

   } while (B < Bmax);

   return BigInteger.ZERO;   // keinen Teiler gefunden
}


public static BigInteger kgVbisZu(int k) {
   BigInteger[] feldBig = new BigInteger[k];
   for (int i = 1; i <= k; i++) {
      feldBig[i-1] = BigInteger.valueOf(i);
   }
   return ggT_kgvBig(feldBig)[1];  // [1] = kgV ;  [0] = ggT
}


public static BigInteger[] ggT_kgvBig(BigInteger... values) {
   BigInteger ggTBig = values[0].gcd(values[1]);
   BigInteger kgVBig = values[0].multiply(values[1]).divide(ggTBig);

   for (int i = 2; i < values.length; i++) {
         ggTBig = ggTBig.gcd(values[i]);
         kgVBig = kgVBig.multiply(values[i]).divide(kgVBig.gcd(values[i]));
   }
   return new BigInteger[] {ggTBig, kgVBig};
}
```

Es gibt noch eine Reihe **weiterer Verfahren**, die hier nicht näher erläutert werden sollen.

Die **gängigsten Verfahren zur Faktorisierung** von natürlichen Zahlen seien hier aufgelistet:

- Probedivision
- Fermat
- Lehman
- PollardRho (Verbesserung: Brent-Pollard-Rho)
- Pollard (p-1)
- ECM (= Elliptische Kurven Methode; Erweiterung von p-1 !)
- CFF (= Kettenbruch-Algorithmus ; „Continued Fraction Factorization“; kaum noch benutzt)
- QS (= Quadratisches Sieb ; „Quadratic Sieve“)
  – MPQS (= Multipolynomiales Quadratisches Sieb)
  – SIQS (= Selbstinitialisierendes Quadratisches Sieb ; „Self Initializing Quadratic Sieve“)
- NFS (= Zahlkörper-Sieb ; „Number Field Sieve“)
  – GNFS (= Allgemeines Zahlkörper-Sieb ; „General Number Field Sieve“)
  – SNFS (= Spezielles Zahlkörper-Sieb ; „Special Number Field Sieve“)