Primzahlen, Primzahltests, Primfaktorzerlegung

Ac 2013-2022
14.03.2022

Eine natürliche Zahl heißt **Primzahl**, wenn sie durch genau 2 Zahlen teilbar ist, nämlich durch 1 und durch sich selbst. Demnach ist 1 keine Primzahl, wohl aber die 2 !
Ist n Primzahl, so sagt man auch „**n ist prim**“.

Es gibt **unendlich viele Primzahlen**, was sich indirekt beweisen lässt.
Man nimmt an, dass es k Primzahlen $p_1, p_2, p_3, \dots, p_k$ gibt, also endlich viele.
Die Zahl $n = p_1 \cdot p_2 \cdot p_3 \cdot \dots \cdot p_k + 1$ ist jedoch größer als alle vorher bekannten Primzahlen und durch keine davon teilbar, weil immer der Rest 1 bleibt. Also muss entweder n eine noch größere Primzahl sein als die vorher als bekannt angenommenen oder n ist durch mindestens eine größere als die k bekannten Primzahlen teilbar. Die Annahme war also falsch ! Es muss daher unendlich viele Primzahlen geben.

Die ersten Primzahlen sind:

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 109 113 127
131 137 139 149 151 157 163 167 173 179 181 191 193 197 199 211 223 227 229 233 239 241 251
257 263 269 271 277 281 283 293 307 311 313 317 331 337 347 349 353 359 367 373 379 383 389
397 401 409 419 421 431 433 439 443 449 457 461 463 467 479 487 491 499 503 509 521 523
541 547 557 563 569 571 577 587 593 599 601 607 613 617 619 631 641 643 647 653 659 661 673
677 683 691 701 709 719 727 733 739 743 751 757 761 769 773 787 797 809 811 821 823 827 829
839 853 857 859 863 877 881 883 887 907 911 919 929 937 941 947 953 967 971 977 983 991 997
1009 1013 1019 1021 1031 1033 1039 1049 1051 1061 1063 1069 1087 1091 1093 1097 1103 1109 1117
1123 1129 1151 1153 1163 1171 1181 1187 1193 1201 1213 1217 1223 1229 1231 1237 1249 1259 1277
1279 1283 1289 1291 1297 1301 1303 1307 1319 1321 1327 1361 1367 1373 1381 1399 1409 1423 1427
1429 1433 1439 1447 1451 1453 1459 1471 1481 1483 1487 1489 1493 1499 1511 1523 1531 1543 1549
1553 1559 1567 1571 1579 1583 1597 1601 1607 1609 1613 1619 1621 1627 1637 1657 1663 1667 1669
1693 1697 1699 1709 1721 1723 1733 1741 1747 1753 1759 1777 1783 1787 1789 1801 1811 1823 1831
1847 1861 1867 1871 1873 1877 1879 1889 1901 1907 1913 1931 1933 1949 1951 1973 1979 1987 1993
1997 1999 2003 2011 2017 2027 2029 2039 2053 2063 2069 2081 2083 2087 2089 2099 2111 2113 2129
2131 2137 2141 2143 2153 2161 2179 2203 2207 2213 2221 2237 2239 2243 2251 2267 2269 2273 2281
2287 2293 2297 2309 2311 2333 2339 2341 2347 2351 2357 2371 2377 2381 2383 2389 2393 2399 2411
2417 2423 2437 2441 2447 2459 2467 2473 2477 2503 2521 2531 2539 2543 2549 2551 2557 2579 2591
2593 2609 2617 2621 2633 2647 2657 2659 2663 2671 2677 2683 2687 2689 2693 2699 2707 2711 2713
2719 2729 2731 2741 2749 2753 2767 2777 2789 2791 2797 2801 2803 2819 2833 2837 2843 2851 2857
2861 2879 2887 2897 2903 2909 2917 2927 2939 2953 2957 2963 2969 2971 2999 3001 3011 3019 3023
3037 3041 3049 3061 3067 3079 3083 3089 3109 3119 3121 3137 3163 3167 3169 3181 3187 3191 3203
3209 3217 3221 3229 3251 3253 3257 3259 3271 3299 3301 3307 3313 3319 3323 3329 3331 3343 3347
3359 3361 3371 3373 3389 3391 3407 3413 3433 3449 3457 3461 3463 3467 3469 3491 3499 3511 3517
3527 3529 3533 3539 3541 3547 3557 3559 3571 3581 3583 3593 3607 3613 3617 3623 3631 3637 3643
3659 3671 3673 3677 3691 3697 3701 3709 3719 3727 3733 3739 3761 3767 3769 3779 3793 3797 3803
3821 3823 3833 3847 3851 3853 3863 3877 3881 3889 3907 3911 3917 3919 3923 3929 3931 3943 3947
3967 3989 4001 4003 4007 4013 4019 4021 4027 4049 4051 4057 4073 4079 4091 4093 4099 4111 4127
4129 4133 4139 4153 4157 4159 4177 4201 4211 4217 4219 4229 4231 4241 4243 4253 4259 4261 4271
4273 4283 4289 4297 4327 4337 4339 4349 4357 4363 4373 4391 4397 4409 4421 4423 4441 4447 4451
4457 4463 4481 4483 4493 4507 4513 4517 4519 4523 4547 4549 4561 4567 4583 4591 4597 4603 4621
4637 4639 4643 4649 4651 4657 4663 4673 4679 4691 4703 4721 4723 4729 4733 4751 4759 4783 4787
4789 4793 4799 4801 4813 4817 4831 4861 4871 4877 4889 4903 4909 4919 4931 4933 4937 4943 4951
4957 4967 4969 4973 4987 4993 4999
```

Ist eine natürliche Zahl **nicht prim**, so nennt man sie **zerlegbar** bzw.
zusammengesetzt (engl.: composite) .

Jede zusammengesetzte Zahl n lässt sich als Produkt von Primzahlen eindeutig darstellen, es gilt also
$n = p_1^{t1} \cdot p_2^{t2} \cdot p_3^{t3} \cdot \dots \cdot p_k^{tk}$ ($t_i \in \mathbb{N}$) Beispiel: $420 = 2^2 \cdot 3 \cdot 5 \cdot 7$

Es gibt eine Reihe von Algorithmen, mit denen eine natürliche Zahl n auf „**Primalität“** geprüft werden kann (Primzahltests ; s.u.).

Auch für eine eventuell existierende Primfaktorzerlegung von n gibt es Algorithmen (s.u.)

Primzahlerzeugung:

Der bekannteste Primzahlerzeuger ist das aus der Antike bekannte **„Sieb des Eratosthenes“** (auch als „**sieve**“ bekannt; ca. 275 -194 v.Chr.) , mit dem man eine Tabelle aller Primzahlen von 2 bis zu einem vorgegebenen n bestimmt:

Will man die Primzahlen im Bereich von 2 bis n bestimmen, so legt man zunächst eine Tabelle mit allen natürlichen Zahlen von 2 bis n an. Jetzt streicht man aus dieser Tabelle zunächst alle Vielfachen von 2, anschließend alle Vielfachen von 3. Zu diesem Zeitpunkt ist die 4 bereits gestrichen, es müssen also keine Vielfachen davon beachtet werden. Als nächstes sind die Vielfachen der 5 dran, dann die von 7, usw. .
Hat man die Quadratwurzel von n überschritten, so findet man keine Vielfachen mehr, denn wenn t ein Teiler von n ist, dann ist es auch n / t . Man ist fertig.
Die nicht gestrichenen Zahlen sind die Primzahlen.

Nachteil: Für große n ergibt sich ein hoher Rechen- und Speicheraufwand.

JAVA-Methode:

```
public static ArrayList <Integer> eratosthenesListe(int nmax) {
   // Sieb des Eratosthenes; ArrayList wird erstellt von 2 bis zu <= nmax
   // nmax <= Integer.MAX_VALUE = 2^31-1 = 2.147.483.647   beachten !!
   nmax++;
   final int maxprim = (int)Math.sqrt(nmax)+2;
   boolean[] boolFeld = new boolean[nmax]; // alle Zahlen von 0 bis nmax = false
   for (int i = 0; i < nmax; i++)
      boolFeld[i] = i % 2 == 1; // alle ungeraden Zahlen = true

   // der eigentliche Algorithmus; boolsches Feld aufbauen:
   for (int prim = 3; prim < maxprim; prim += 2) // nur die ungeraden Zahlen
      // (die geraden Zahlen wurden schon gestrichen)
      if (boolFeld[prim]) { // noch nicht gestrichen, d.h. prim
         for (int i = prim; i <= nmax / prim; i++) {
            final int zahl = i * prim;
            if (zahl < nmax) // Überlauf verhindern
               boolFeld[zahl] = false; // zahl = i*prim streichen
         }
      }

   // ArrayListe aufbauen
   ArrayList <Integer> primZahlenListe = new ArrayList <Integer> () ;
   // Primzahl 2 am Anfang einfügen, da 2 nicht im boolFeld !
   primZahlenListe.add(2);
   for (int i = 3; i < nmax; i++)
      if (boolFeld[i])
         primZahlenListe.add(i);
   return primZahlenListe;
}
```

Hinweis: Bei genügend großem Speicher und schnellem Rechner kann man durchaus **bis n = 10^8** rechnen. Somit hätte man alle Primzahlen von 2 bis 100 Millionen ermittelt !

Weiter unten werden noch Verfahren zur Erzeugung großer Primzahlen vorgestellt !

Primzahltests:

Die einfachste (sehr langsame) Methode ist die sog. „Holzhammermethode“ bzw. brute-force-Methode namens **Probedivision** (trial division) :

Die zu testende Zahl n wird durch t = 2; 3; 4; 5; 6; …; n-1 geteilt .
Geht die Division bei einem der Teiler t auf (Rest=0), so ist n keine Primzahl und man kann die Untersuchung beenden. Andernfalls ist n Primzahl.

Verbesserung1:
Es genügt, ab 3 lediglich ungerade Teiler zu betrachten, denn wenn 2 schon kein Teiler war, können 4; 6; 8; … auch keine Teiler sein.

Verbesserung2:
Es genügt, lediglich bis zur Quadratwurzel von n zu teilen, denn falls t ein Teiler von n ist, so ist auch n / t ein Teiler von n.
Beispiel: p = 1000 10 teilt 1000 und 100 teilt 1000 ; es genügt also, bis 31 zu teilen .

Verbesserung 3:
Jede Primzahl größer als 3 ist von der Form 6i - 1 oder 6i + 1 (ab i = 1).
Daher braucht man lediglich t = 2 , t = 3 , t = 5 zu testen und
dann den Teiler t abwechselnd um 2 bzw. 4 erhöhen.

Hierzu folgende JAVA-Methode:

```
public static boolean istPrimTrialdivision(long n) {
	if (n < 7)
		return (n == 2 || n == 3 || n == 5);
	if (n % 2 == 0 || n % 3 == 0 || n % 5 == 0)
		return false;
	long max = (long) Math.sqrt(n), teiler = 7, increment = 4;
	while (teiler <= max) {
		if (n % teiler == 0)
			return false;
		else {
			teiler += increment;
			increment = 6 - increment;
		}
	}
	return true;
}
```

Probabilistische Primzahltests:

Dies sind Tests, bei denen Wahrscheinlichkeiten eine Rolle spielen.

Erkennt ein solcher Test eine Zahl als zusammengesetzt, so ist sie mit Sicherheit zusammengesetzt.
Erkennt der Test eine Zahl als Primzahl, so irrt er mit einer gewissen Wahrscheinlichkeit p .

Primzahltest von **Fermat**:

Grundlage ist der „kleine Satz von Fermat“:

Wenn n eine Primzahl ist , dann gilt für alle a aus der Menge { 1 ; 2; … ; n-1 } : $a^{n-1} \bmod n = 1$

Dies bedeutet: a^{n-1} lässt bei Division durch n den ganzzahligen Rest 1 .

Beispiel: n = 7: $1^6 \bmod 7 = 1 \bmod 7 = 1$ $2^6 \bmod 7 = 64 \bmod 7 = 1$ $3^6 \bmod 7 = 729 \bmod 7 = 1$
$4^6 \bmod 7 = 4096 \bmod 7 = 1$ $5^6 \bmod 7 = 15625 \bmod 7 = 1$ $6^6 \bmod 7 = 46656 \bmod 7 = 1$

Der Satz ist nur von links nach rechts gültig (wenn... dann...) und daher nicht umkehrbar , d.h. :
Ist das Ergebnis von $a^{n-1} \bmod n$ ungleich 1, war n auf alle Fälle keine Primzahl.
Ist das Ergebnis von $a^{n-1} \bmod n$ gleich 1, könnte n eine Primzahl sein. Dies ist aber nicht sicher !

Gegenbeispiel 1: **n=42** :
Für a=2 gilt: $2^{41} = 2199023255552$; $2199023255552 \bmod 42 = 32$; also 42 nicht prim.

Gegenbeispiel 2: **n=341** (im folgenden wird mit dem unten erläuterten PowerMod gerechnet) :
Für a=2 gilt: $2^{340} \bmod 341 = 1$.
Für a = 3 gilt aber: $3^{340} \bmod 341 = 56$; also 341 nicht prim.
Man nennt die Zahl 341 **Fermatsche Pseudoprimzahl** zur Basis 2 und
die Zahl 2 einen **Lügner** für die Primalität von 341 .
Anmerkung: Unter anderem sind auch 561 und 645 Fermatsche Pseudoprimzahlen zur Basis 2.

Es gibt sogar Zahlen, die für alle Basen a aus { 1; 2; … ; n-1 } Pseudoprimzahlen sind. Das Ergebnis von $a^{n-1} \bmod n$ ist bei diesen Zahlen immer 1, obwohl es sich **nicht** um eine Primzahl handelt.
Man nennt diese Zahlen **Carmichael-Zahlen** (R.D.Carmichael entdeckte sie 1910).
Das kleinste Beispiel für Carmichael-Zahlen ist die Zahl 561.

Bei den Carmichael-Zahlen versagt der Primzahltest von Fermat völlig !!

Beispielrechnungen für n=561 (Carmichael-Zahl):
Für a=2: $2^{560} \bmod 561 = 1$ Für a=3: $3^{560} \bmod 561 = 1$ Für a=4: $4^{560} \bmod 561 = 1$
Für a=5: $5^{560} \bmod 561 = 1$ Für a=500: $500^{560} \bmod 561 = 1$ Für a=560: $4^{560} \bmod 561 = 1$
Jedoch ist 561 keine (!) Primzahl, denn $561 = 3 \cdot 11 \cdot 17$

Seit 1992 weiß man, dass es unendlich viele Carmichael-Zahlen gibt, sie sind aber dünn gesät.

Die ersten Carmichael-Zahlen sind :
561 1105 1729 2465 2821 6601 8911 10585
Weitere Beispiele von Carmichael-Zahlen:
410041 56052361 118901521 ...

Es lässt sich beweisen, dass jede Carmichael-Zahl mindestens drei verschiedene Primfaktoren enthalten muss und quadratfrei ist !

PowerMod :

Die Berechnung von von $\mathbf{a^{n-1}\ mod\ n}$ kann auch ohne riesige Zwischenergebnisse vorgenommen werden, und zwar mit der Modulo-Division **PowerMod** . Diese basiert auf der folgenden Identität:

$$a^n \bmod p = \begin{cases} (a^{n/2} \bmod p)^2 \bmod p \ ; \ n \ gerade \\ (a^{n-1} \bmod p) \cdot a \bmod p \ ; \ n \ ungerade \end{cases}$$

Beispiel: $3^9 \bmod 17 = (3^8 \bmod 17)\cdot 3 \bmod 17 = ((3^4 \bmod 17)^2 \bmod 17)\cdot 3 \bmod 17 =$
$(((3^2 \bmod 17)^2 \bmod 17)^2 \bmod 17)\cdot 3 \bmod 17 =$
$(((9 \bmod 17)^2 \bmod 17)^2 \bmod 17)\cdot 3 \bmod 17 =$
$((9^2 \bmod 17)^2 \bmod 17)\cdot 3 \bmod 17 =$
$((81 \bmod 17)^2 \bmod 17)\cdot 3 \bmod 17 =$
$(13^2 \bmod 17)\cdot 3 \bmod 17 =$
$(169 \bmod 17)\cdot 3 \bmod 17 =$
$16\cdot 3 \bmod 17 =$
$48 \bmod 17 = 14$

Wie man sieht, kommen keine riesigen Zahlen vor.

Ein effizienter Algorithmus ist der folgende:

Algorithmus **PowerMod** (basis, expo, m) :

Berechnet basis^expo mod m.

```
tmp = 1
solange expo > 0
    falls expo ungerade
        dann tmp = tmp· basis mod m
    expo = expo div 2
    basis = basis² mod m
ergebnis = tmp
```

Test des Algorithmus für obiges Beispiel: $3^9 \bmod 17$:

basis=3 expo=9 m=17
tmp=1
tmp=1*3 mod 17 = 3
expo = 9 div 2 = 4
basis = 3^2 mod 17 = 9
expo = 4 div 2 = 2
basis = 9^2 mod 17 = 13
expo = 2 div 2 = 1
basis = 13^2 mod 17 = 16
tmp=3*16 mod 17 = 14
expo = 1 div 2 = 0
basis = 16^2 mod 17 = 1
ergebnis=14

Primzahltest von **Miller** und **Rabin**

Grundlage ist auch hier der bereits oben erwähnte „kleine Satz von Fermat“. Jedoch umgeht der Test von Miller und Rabin das Problem mit den Carmichael-Zahlen. Entscheidend ist folgender Satz:

Ist n eine Primzahl und $n-1 = 2^s \cdot d$ mit ungeradem d , dann gilt für alle a < n mit ggT(a, n) = 1 :

Entweder $a^d \bmod n = 1$ oder $a^{2^r \cdot d} \bmod n = (n-1)$ für ein r aus der Menge { 0; 1; … ; s-1 }

Außerdem: Wenn $a^d \bmod n \neq 1 \wedge a^d \bmod n \neq n-1 \wedge a^{2^r \cdot d} \bmod n = 1; r > 0$ dann ist n zusammengesetzt (nicht prim) !

Der Primzahl-Test besteht nun darin, zu der zu prüfenden Zahl n eine teilerfremde Zahl a zu finden, die keine der ersten beiden Aussagen oder aber die 3. erfüllt. In diesem Fall wäre n keine Primzahl .
Da mehr als ¾ der Zahlen a aus der Menge {2; 3; … ; n-2} die Eigenschaft (beide Aussagen nicht erfüllt) besitzen, ist die Wahrscheinlichkeit für die Nicht-Primalität (Zerlegbarkeit) von n bei einem einmaligen Miller-Rabin-Test größer als ¾ .

Umgekehrt ist die Wahrscheinlichkeit dafür, die Primalität von n zu bestätigen, kleiner als ¼ .
Jargon: „**Man findet keinen Zeugen (engl: witness) gegen die Primalität von n**“ .
Durch Wiederholung des Tests mit anderem a-Wert kann man die Wahrscheinlichkeit verkleinern.
Bei 10-facher Durchführung ist die Wahrscheinlichkeit, nicht prim zu sein, kleiner als $0{,}25^{10} = 9{,}5 \cdot 10^{-7}$
Hat man dann immer noch keinen Zeugen gegen die Primalität von n gefunden, so ist n mit hoher Wahrscheinlichkeit prim !

Zu prüfen ist für jedes ausgewählte a:

- Berechne $x = a^d \bmod n$; falls x = 1 oder x = n-1, dann untersuche ein neues a (n evtl. prim)
- Andernfalls berechne $\mathbf{x = a^{z \cdot d} \bmod n}$ mit $z = 2^r$ und r aus { 1; 2; ... ; s-1 }
  - Falls x = 1, dann ist n auf jeden Fall zusammengesetzt (nicht prim)
  - Falls x = n-1, dann untersuche ein neues a (n evtl. prim)
- Falls x ≠ n-1, dann ist n zusammengesetzt, sonst n evtl. prim

Beispiel 1: **n = 97**. Dann ist $96 = 2^5 \cdot 3$. Also d = 3 und s = 5 .

Wähle **a = 2** < n und berechne erst $a^d \bmod n$, also $2^3 \bmod 97 = 8$.
Berechne nun $a^{2^r \cdot d} \bmod n$ für r = 1; 2; 3; 4 , also
$2^6 \bmod 97 = 64$ $2^{12} \bmod 97 = 22$ $2^{24} \bmod 97 = 96 = 97-1$
Daher spricht a = 2 nicht gegen die Primalität von 97.

Wähle **a = 37** < n und berechne erst $a^d \bmod n$, also $37^3 \bmod 97 = 19$.
Berechne nun $a^{2^r \cdot d} \bmod n$ für r = 1; 2; 3; 4.
$37^6 \bmod 97 = 70$ $37^{12} \bmod 97 = 50$ $37^{24} \bmod 97 = 75$ $37^{48} \bmod 97 = 96 = 97-1$!!
Daher spricht auch a = 37 nicht gegen die Primalität von 97.

Wähle **a = 96** < n und berechne erst $a^d \bmod n$, also $96^3 \bmod 97 = 96 = 97-1$!!
Daher spricht auch a = 96 nicht gegen die Primalität von 97.

Inzwischen liegt die Wahrscheinlichkeit für die Zerlegbarkeit von 97 bei $0{,}25^3 \approx 1{,}5\%$.
Für n = 97 findet man auch bei weiterer Suche kein a, das gegen die Primalität von 97 spricht .

Beispiel 2: **n = 561**. Dann ist $560 = 2^4 \cdot 35$. Also d = 35 und s = 4 .

Wähle **a = 2** < n und berechne erst $a^d \bmod n$, also $2^{35} \bmod 561 = 1$.
Dieses Ergebnis spricht nicht gegen die Primalität von 561.

Wähle **a = 3** < n und berechne erst $a^d \bmod n$, also $3^{35} \bmod 561 = 78$.

Berechne nun $a^{2^r \cdot d} \bmod n$ für r = 1; 2; 3 .
$3^{70} \bmod 561 = 474$ $3^{140} \bmod 561 = 276$ $3^{280} \bmod 561 = 441$.
Da keines der Ergebnisse 560 ist, spricht a = 3 gegen die Primalität von 561.
Daher ist 561 zusammengesetzt !

Beispiel 3: **n = 1729**. Dann ist $1728 = 2^6 \cdot 27$. Also d = 27 und s = 6 .

Wähle **a = 2** < n und berechne erst $a^d \bmod n$, also $2^{27} \bmod 1729 = 645$.

Berechne nun $a^{2^r \cdot d} \bmod n$ für r = 1; 2; 3; 4; 5 .
$2^{54} \bmod 1729 = 1065$ $2^{108} \bmod 1729 = 1$.
Dieses Ergebnis spricht gegen die Primalität von 1729.
Daher ist 1729 zusammengesetzt !

Aus "Wikipedia":

Es genügt, a aus der folgenden Zahlenmenge auszuwählen: $a \in \{2,3, \dots , \min(n-1, 2\cdot(\ln n)^2)\}$

Außerdem ist bekannt, dass für kleine n eine viel kleinere Anzahl ausreicht, um auf Primalität zu testen:

n < 1.373.653 : es genügt, a= 2 und 3 zu testen,
n < 9.080.191 : es genügt, a = 31 und 73 zu testen,
n < 4.759.123.141 : es genügt, a = 2, 7, und 61 zu testen,
n < 2.152.302.898.747 : es genügt, a = 2, 3, 5, 7, und 11 zu testen,
n < 3.474.749.660.383 : es genügt, a = 2, 3, 5, 7, 11, und 13 zu testen,
n < 341.550.071.728.321 : es genügt, a = 2, 3, 5, 7, 11, 13, und 17 zu testen.
n < 3.825.123.056.546.413.051 : es genügt, a = 2, 3, 5, 7, 11, 13, 17, 19, und 23 zu testen.
n < 318.665.857.834.031.151.167.461 : es genügt, a = 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37 zu testen.

Dabei dürfen nur solche n getestet werden, die größer sind als das jeweils größte a !!

In der folgenden JAVA-Methode für den Datentyp **long** wollen wir jedoch lediglich auf obige Menge mit

$2(\ln n)^2$ zurückgreifen.

Man beachte aber, dass beim Datentyp **long** die Gefahr eines Überlaufs besteht, wenn Rechenoperationen wie Potenzierung vorgenommen werden.

Beim Quadrieren einer natürlichen Zahl a darf a^2 die Zahl

Long.MAX_VALUE = 9223372036854775807 = 2^{63}-1 nicht überschreiten .

Daher muss gelten: a < 3037000500
Die zu untersuchende Zahl n darf beim Datentyp **long** somit höchstens 3037000500 betragen !

Das beim Miller-Rabin-Test notwendige Potenzieren (bei $a^d \bmod n$) birgt noch eine andere Gefahr, nämlich dass Potenzen extrem groß werden. Zur Vermeidung dieses Problems gibt es die so genannte Methode powerMod() , die bereits weiter oben erläutert wurde und die als höchste Potenz das Quadrat verwendet.
Die entsprechende Java-Methode ist unten angegeben:

```java
public static long powerMod(long basis, long expo, long m) { // berechnet basis^expo mod m
   long erg = 1;
   while (expo > 0) {
      if (expo % 2 == 1)
         erg = (erg * basis) % m;
      expo = expo / 2;
      basis = (basis * basis) % m;
   }
   return erg;
}

public static long zufZahl(long von, long bis) { // Hilfsmethode
   Random rand = new Random();
   long zuf = Math.abs(rand.nextLong()) % (bis - von + 1);  // zuf in [0..bis-von]
   return zuf + von;  // Ergebnis in [von..bis]
}

public static boolean istPrimMillerRabin(long n) { // testet probabilistisch, ob n prim ist
   if (n == 2) return true;
   if (n < 2 || n % 2 == 0) return false;
   // Erzeuge ungerades d mit:   2^s * d = n-1
   long d = n - 1; // d zunächst gerade
   int s = 0;
   while (d % 2 == 0) {
      d = d / 2;
      s = s + 1;
   }
   long k = 20; // k ist die Anzahl der Durchläufe ; typisch k = 20
   long a, x;
   long aMin = 2;
   long aMax = Math.min(n-1, (long) (Math.log(n)*Math.log(n)*2.0));
   for (long i = 1; i <= k; i++) {
      a = zufZahl(aMin, aMax);  // a = random(2, min(n-1, 2*(ln n)²))
      // Berechnung von x = a^d mod n
      x = powerMod(a, d, n);
      if (x == 1 || x == n - 1)
         continue; // evtl. n prim; nächsten Durchlauf starten
      // Berechnung von x = a^(2^r*d) mod n
      for (int r = 1; r < s; r++) {
         x = (x * x) % n; // evtl. hier "Long-Überlauf" ?
         if (x == 1)
            return false; // n ist zusammengesetzt
         if (x == n - 1)
            break; // evtl. n prim; nächsten Durchlauf starten
      }
      if (x != n - 1)
         return false; // n ist zusammengesetzt
   } // Ende for long i
   return true; // sehr wahrscheinlich ist n prim
}

public static void main(String[] args) {
   String sEingabe = JOptionPane.showInputDialog("Miller-Rabin: LongZahl n = ", 982609483L);
   long n = Long.parseLong(sEingabe);
   if (n > 3037000500)
      System.out.println("Zahl " + n + " außerhalb des gültigen Bereichs !");
   else {
      if (istPrimMillerRabin(n))
         System.out.println(n + " ist vermutlich Primzahl ");
      else
          System.out.println(n + " ist keine Primzahl.");
   }
}
```

Anmerkung: In der BigInteger-Klasse von Java ist ein Miller-Rabin-Primzahltest bereits eingebaut , nämlich die Methode "isProbablePrime()" .

Primfaktorzerlegung:

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“ ; siehe oben):

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) + "? ";
   }
   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 $a_0 = 23,685\ldots$. 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 $a_0 = 39,912\ldots$. 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 $\lfloor \sqrt[3]{n} \rfloor$ wiederhole :

für x von $\lceil \sqrt{4 \cdot k \cdot n} \rceil$ 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 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 .

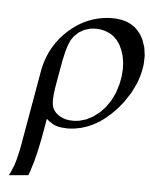
Für das obige Beispiel mit n = 143 läuft das Verfahren folgendermaßen ab:

```
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 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 **Brent** 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
```

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)
- Pollard (p-1)
- ECM (= Elliptische Kurven Methode)
- CFF (= Kettenbruch-Algorithmus ; „Continued Fraction Factorization“)
- 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“)

Wie erzeugt man **große** Primzahlen ?

Es gibt eine Reihe von Formeln, welche zu Primzahlen führen können, d.h. es muss geprüft werden, ob es sich tatsächlich um eine Primzahl handelt !

Zunächst wird eine Methode vorgestellt, die auf **bereits bekannte** Primzahlen zurückgreift :

Kennt man die ersten k Primzahlen p_i (i = 1 bis k), so lässt sich eine neue Zahl n auf folgende Weise erzeugen:

$n = p_1 \cdot p_2 \cdot p_3 \cdot \ldots \cdot p_k + 1$

n ist entweder **prim** oder
n besitzt **mindestens zwei Primfaktoren**, die von den bekannten k Primzahlen verschieden sind.

Man findet also auf jeden Fall mindestens eine Primzahl oberhalb von p_k !

Beispiele:

$n = 2+1 = 3$ prim
$n = 2 \cdot 3+1 = 7$ prim
$n = 2 \cdot 3 \cdot 5+1 = 31$ prim
$n = 2 \cdot 3 \cdot 5 \cdot 7+1 = 211$ prim
$n = 2 \cdot 3 \cdot 5 \cdot 7 \cdot 11+1 = 2311$ prim
$n = 2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13+1 = 30031 = 59 \cdot 509$ alle Faktoren prim
$n = 2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17+1 = 510511 = 19 \cdot 97 \cdot 277$ alle Faktoren prim
$n = 2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19+1 = 9699691 = 347 \cdot 27953$ alle Faktoren prim
$n = 2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23+1 = 223092871 = 317 \cdot 703763$ alle Faktoren prim
$n = 2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29+1 = 6469693231 = 331 \cdot 571 \cdot 34231$ alle Faktoren prim
$n = 2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31+1 = 200560490131$ prim
$n = 2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31 \cdot 37+1 = 7420738134811 = 181 \cdot 60611 \cdot 676421$ alle Faktoren prim
$n = 2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31 \cdot 37 \cdot 41+1 = 304250263527211 = 61 \cdot 450451 \cdot 11072701$ alle Faktoren prim
$n = 2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31 \cdot 37 \cdot 41 \cdot 43+1 = 13082761331670031 = 167 \cdot 78339888213593$
alle Faktoren prim
$n = 2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31 \cdot 37 \cdot 41 \cdot 43 \cdot 47+1 = 614889782588491411 = 953 \cdot 46727 \cdot 13808181181$
alle Faktoren prim
$n = 2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31 \cdot 37 \cdot 41 \cdot 43 \cdot 47 \cdot 53+1 = 32589158477190044731 =$
$73 \cdot 139 \cdot 173 \cdot 18564761860301$ alle Faktoren prim

Die auf diese Weise erzeugte Zahl n ist prim für k = 2, 3, 5, 7, 11, 31, 379, 1019, 1021, ...

Weitere Formeln:

n! - 1 ist prim für n = 3, 4, 6, 7, 12, 14, 30, 32, 33, 38, 94, 166, ...

n! + 1 ist prim für n = 1, 2, 3, 11, 27, 37, 41, 73, 77, 116, 154, …

kgV(1, … , n) + 1 liefert die Primzahlen 2, 3, 7, 13, 61, 421, 2521, 232792561, …

Drei sehr bekannte Generatoren gehen zurück auf die
bedeutenden Mathematiker Leonhard **Euler** (Schweiz) und Pierre **de Fermat** (Frankreich)
sowie auf den franz. Mönch Marin **Mersenne** :

Einige **„Euler-Zahlen“** ; das sind Zahlen der Form $E_n = n^2 + n + 41\,;\quad n \in \mathbb{N}$

Die ersten 40 Euler-Zahlen E_n sind Primzahlen.

$0^2 + 0 + 41 = 41$ (prim)
$1^2 + 1 + 41 = 43$ (prim)
$2^2 + 2 + 41 = 47$ (prim)
$3^2 + 3 + 41 = 53$ (prim)
...
...
$38^2 + 38 + 41 = 1523$ (prim)
$39^2 + 39 + 41 = 1601$ (prim)

$40^2 + 40 + 41 = 1681 = 41 \cdot 41$
$41^2 + 41 + 41 = 1763 = 41 \cdot 43$
...
$100^2 + 100 + 41 = 10141 =$ (prim)
...
$100000^2 + 100000 + 41 = 10000100041 = 163 \cdot 199 \cdot 308293$
...
$100000000^2 + 100000000 + 41 = 10000000100000041 =$ (prim)

Eine weitere von Euler angegebene Formel ist **$n^2 + n + 17$** ,
die 16 aufeinander folgende Primzahlen (n = 0 bis n = 15) erzeugt !
17, 19, 23, 29, 37, 47, 59, 73, 89, 107, 127, 149, 173, 199, 227, 257,
$289 = 17^2$, … erst die Zahl 289 (n = 16) ist zerlegbar !

Eine ähnliche (mithilfe von Computern gefundene) Formel ist: **$n^2 - 79n + 1601$** ,
die 80 aufeinander folgende Primzahlen (n = 0 bis n = 79) erzeugt !
1601, 1523, 1447, 1373, 1301, 1231, 1163, 1097, 1033, 971, 911, 853, 797, 743,
691, 641, 593, 547, 503, 461, 421, 383, 347, 313, 281, 251, 223, 197, 173, 151,
131, 113, 97, 83, 71, 61, 53, 47, 43, 41, 41, 43, 47, 53, 61, 71, 83, 97, 113,
131, 151, 173, 197, 223, 251, 281, 313, 347, 383, 421, 461, 503, 547, 593, 641,
691, 743, 797, 853, 911, 971, 1033, 1097, 1163, 1231, 1301, 1373, 1447, 1523,
1601, $1681=41^2$, ... erst die Zahl 1681 (n = 80) ist zerlegbar !

Auch **$n^2 - 9n + 61$** liefert sehr viele Primzahlen; unter den ersten 1000 Zahlen finden sich mehr als 50% Primzahlen !

Allgemeine Formel: **$n^2 + a \cdot n + b$**

Einiges über Fermat-Zahlen, das sind Zahlen der Form $F_n = 2^{2^n} + 1;\ n \in \mathbb{N}$

Pierre de Fermat (1601-1665) vermutete, dass diese Zahlen immer Primzahlen seien, jedoch zeigte der schweizer Mathematiker Leonhard Euler bereits 1732 , dass **F5 zerlegbar** ist.

$F_0 = 2^{2^0} + 1 = 2^1 + 1 = 3$ (prim)

$F_1 = 2^{2^1} + 1 = 2^2 + 1 = 5$ (prim)

$F_2 = 2^{2^2} + 1 = 2^4 + 1 = 17$ (prim)

$F_3 = 2^{2^3} + 1 = 2^8 + 1 = 257$ (prim)

$F_4 = 2^{2^4} + 1 = 2^{16} + 1 = 65537$ (prim ; größte bekannte Fermatsche Primzahl !)

$F_5 = 2^{2^5} + 1 = 2^{32} + 1 = 4294967297 = 641 \cdot 6700417$ von Leonhard Euler (1732) zerlegt !

$F_6 = 2^{2^6} + 1 = 2^{64} + 1 = 18446744073709551617 = 274177 \cdot 67280421310721$ von Landry (1880) zerlegt !

$F_7 = 2^{2^7} + 1 = 2^{128} + 1 = 340282366920938463463374607431768211457 =$

$59649589127497217 \cdot 5704689200685129054721$ von Morrison & Brillhart (1970) zerlegt !

$F_8 = 2^{2^8} + 1 = 2^{256} + 1 = 115792089237316195423570985008687907853269984665640564039457584007913129639937 = 1238926361552897 \cdot$

$93461639715357977769163558199606896584051237541638188580280321$

von Brent & Pollard (1980) zerlegt !

$F_9 = 2^{2^9} + 1 = 2^{512} + 1 = 13407807929942597099574024998205846127479365820592393377723561443721764030073546976801874298166903427690031858186486050853753882811946569946433649006084097 = 2424833 \cdot 7455602825647884208337395736200454918783366342657 \cdot$

$741640062627530801524787141901937474059940781097519023905821316144415759504$

$7050080928187116939407 37$ von Lenstra & Manasse (1990) zerlegt !

Für Fermatzahlen gilt folgende Rekursionsformel: $F_n = \prod_{k=1}^{n-1} F_k + 2$

Beispiele: $F_3 = 257$ und $F_0 \cdot F_1 \cdot F_2 + 2 = 3 \cdot 5 \cdot 17 + 2 = 255 + 2 = 257$

$F_4 = 65537$ und $F_0 \cdot F_1 \cdot F_2 \cdot F_3 + 2 = 3 \cdot 5 \cdot 17 \cdot 257 + 2 = 65535 + 2 = 65537$

Interessant ist: Jede Fermat-Zahl F_n mit $n > 1$ endet mit der dezimalen Ziffer 7 .
Die beiden letzten Ziffern sind dabei 17, 37, 57 oder 97 .

Bisher unbewiesene Vermutung: $F_k = 2^{2^k} + 1;\quad k \in \mathbb{N}$ ist für $k > 4$ stets zerlegbar !

Weitere vollständige Zerlegungen (F10, F11) siehe Tabelle weiter unten !

Für F_k mit $k > 11$ gibt es derzeit (02/2022) noch keine vollständigen Zerlegungen !!
Bekannt ist jedoch, dass F_5 bis F_{32} zerlegbar sind !

Die Primzahleigenschaft von F_n lässt sich u.a. mit dem Pépin-Test überprüfen:

Dieser Test gilt ausschließlich für Fermatzahlen !

Pépin-Test : F_n mit n > 0 ist genau dann Primzahl, wenn gilt: $3^{((F_n - 1) / 2)} \bmod F_n = -1$

Die Formel lässt sich vereinfachen, wenn man für F_n den Term $2^{(2^n)}+1$ einsetzt. Dann ist nämlich $F_n - 1 = 2^{(2^n)}$ und $(F_n - 1) / 2 = 2^{(2^n - 1)}$. Fn lässt sich dann schreiben als $2^{(2^n - 1)} * 2 + 1$.

Algorithmus Pépin :

```
Fnm1h = 2^(2^n - 1)
  Fn = Fnm1h * 2 + 1
  Erg = 3^Fnm1h mod Fn
  Falls Erg = -1 oder Erg = Fn - 1
        dann ist Fn prim
        sonst ist Fn zusammengesetzt
```

Beispiele:

n = 2: Fnm1h = 8 $F_2 = 17$ Erg = $3^8 \bmod 17 = 16 = F_2 - 1 \Rightarrow F_2$ ist Primzahl !

n = 3: Fnm1h = 128 $F_3 = 257$ Erg = $3^{128} \bmod 257 = 256 = F_3 - 1 \Rightarrow F_3$ ist Primzahl !

n = 5: *Fnm1h* = 2147483648 $F_5 = 4294967297$ $3^{2147483648} \bmod 4294967297 = 10324303 \neq -1$

$\Rightarrow F_5$ ist zusammengesetzt !

Da die Zahlen sehr schnell riesengroß werden, sollte man unbedingt mit PowerMod rechnen.

Die Anzahl der Ziffern von F_n lässt sich übrigens schätzen durch $\lfloor 2^n \cdot \lg(2) \rfloor + 1$

Nach dieser Formel hat z.B. F_{20} 315653 Ziffern; in Wirklichkeit sind es 315686 ! .

Java-Methode (Pépin) :

```
public static boolean istPrimPepin(int n) {
   // Pépin-Test für Fermat-Primzahlen.
   // Testet, ob 3^(2^(2^n-1)) mod F(n) = -1  ( bzw. = F(n) - 1 )
   int zweiHochNm1 = (1 << n) - 1;
   BigInteger Fnm1h =  BigInteger.ONE.shiftLeft(zweiHochNm1);
   BigInteger Fn = Fnm1h.shiftLeft(1).add(BigInteger.ONE);
   BigInteger Erg = BigInteger.valueOf(3).modPow(Fnm1h, Fn);
   if (Erg.equals(Fn.subtract(BigInteger.ONE)) || Erg.equals(BigInteger.ONE.negate()))
         return true;
   return false;
}
```

Der Nachweis der Zerlegbarkeit von F_{15} dauert bei meinem Computer mit dem Pépin-Test 19s und mit dem Miller-Rabin-Test 38s .

Bei F_{17} dauert der Pépin-Test bereits 1156s (ca. 19 min) !

Einiges über Mersenne-Zahlen: $M_n = 2^n - 1\,;\ n \in \mathbb{N}^*$

Z.B. $M_1 = 2^1 - 1 = 1$ $\quad M_2 = 2^2 - 1 = 3$ $\quad$ usw.

Mersenne-Zahlen haben die Binärdarstellung 111 ... 111 , wie man sich leicht verdeutlichen kann.
Mersenne-Zahlen M_n können nur dann Primzahlen sein, wenn auch n prim ist, aber nicht jede Mersenne-Zahl mit n = prim ist eine Primzahl (z.B. ist M_{11} keine Primzahl).
Bisher (2022) sind erst **51 Mersenne-Primzahlen** bekannt !

Mit dem **"Lucas-Lehmer-Test"** können Mersenne-Zahlen auf Primalität getestet werden .
Dieser Test dient zum Testen der Primalität von ungeraden Mersenne-Zahlen ab M_3 .
Voraussetzung: In der Darstellung von $M_n = 2^n - 1$ sei n ungerade und prim.
Die Folge s(k) sei definiert durch $s(1) = 4$; $s(k+1) = s(k)^2 - 2$.
Dann gilt: $M_n = 2^n - 1$ ist genau dann prim, wenn $s(n-1)$ durch M_n teilbar ist .
Andere Formulierung: $s(1) = 4$ und $s(k+1) = (s(k)^2 - 2) \bmod M_n \Rightarrow M_n$ prim $\Leftrightarrow s(n-1) = 0$

Beispiel 1: $M_7 = 2^7 - 1 = 127$. $\quad n = 7$
$s(1) = 4$
$s(2) = (4^2 - 2) \bmod 127 = 14 \bmod 127 = 14$
$s(3) = (14^2 - 2) \bmod 127 = 194 \bmod 127 = 67$
$s(4) = (67^2 - 2) \bmod 127 = 4487 \bmod 127 = 42$
$s(5) = (42^2 - 2) \bmod 127 = 1762 \bmod 127 = 111$
$s(6) = (111^2 - 2) \bmod 127 = 12319 \bmod 127 = 0$
Wegen $s(n-1) = s(6) = 0$ ist M_7 eine Primzahl !

Beispiel 2: $M_{11} = 2^{11} - 1 = 2047$. $\quad n = 11$
$s(1) = 4$
$s(2) = (4^2 - 2) \bmod 2047 = 14 \bmod 2047 = 14$
$s(3) = (14^2 - 2) \bmod 2047 = 194 \bmod 2047 = 194$
$s(4) = (194^2 - 2) \bmod 2047 = 37634 \bmod 2047 = 788$
$s(5) = (788^2 - 2) \bmod 2047 = 620942 \bmod 2047 = 701$
$s(6) = (701^2 - 2) \bmod 2047 = 491399 \bmod 2047 = 119$
$s(7) = (119^2 - 2) \bmod 2047 = 14159 \bmod 2047 = 1877$
$s(8) = (1877^2 - 2) \bmod 2047 = 3523127 \bmod 2047 = 240$
$s(9) = (240^2 - 2) \bmod 2047 = 57598 \bmod 2047 = 282$
$s(10) = (282^2 - 2) \bmod 2047 = 79522 \bmod 2047 = 1736$
Wegen $s(n-1) = s(10) = 1736 \neq 0$ ist M_{11} keine Primzahl !

Java-Methode "Lucas-Lehmer-Test":

```
public static boolean istPrimLucasLehmer(int n) {
   // Lucas-Lehmer-Test für Mersennesche Primzahlen.
   // Testet für ungerade(!) Zahlen n >= 3 (und n=2), ob 2^n-1 eine Primzahl ist
   if (n == 2 || n == 3) return true;
   if (n % 2 == 0 || n < 3) return false;
      // berechne Mersennezahl 2^n-1
   BigInteger mersenneBig = BigInteger.ONE.shiftLeft(n).subtract(BigInteger.ONE);
   BigInteger sBig = new BigInteger("4");
   BigInteger zweiBig = new BigInteger("2");
    for (int k = 2; k < n; k++)
        sBig = sBig.multiply(sBig).subtract(zweiBig).mod(mersenneBig);
    return sBig.equals(BigInteger.ZERO);
}
```

Im folgenden werden **die bisher bekannten 51 Mersenne-Primzahlen** aufgeführt.

1. $2^{2}-1$ = 3
2. $2^{3}-1$ = 7
3. $2^{5}-1$ = 31
4. $2^{7}-1$ = 127
5. $2^{13}-1$ = 8191
6. $2^{17}-1$ = 131071
7. $2^{19}-1$ = 524287
8. $2^{31}-1$ = 2147483647 (Nachweis durch Euler) 10 Stellen
9. $2^{61}-1$ = 2305843009213693951 19 Stellen
10. $2^{89}-1$ = 618970019642690137449562111
11. $2^{107}-1$ = 162259276829213363391578010288127
12. $2^{127}-1$ = 170141183460469231731687303715884105727 (Lucas) 39 Stellen
13. $2^{521}-1$ = 6864797660130609714981900799081393217269435300143305409394463459185543183397656052122559640661454554977296311391480858037121987999716643812574028291115057151 157 Stellen
14. $2^{607}-1$ = 531137992816767098689588206552468627329593117727031923199444138200403559860852242739162502265229285668889329486246501015346579337652707239409519978766587351943831270835393219031728127 183 Stellen
15. $2^{1279}-1$ = 104079321946643990819252403273640855386152622472667048053191123504036080596733602980122394417323241848424216139542810077913835662483234649081399066056773207629241295093892203457731833496615835504729594205476898112116936771475484788669625013844382602917323488853111608285384165850282556046662248311890918801847068222203140521026698435488732958028878050869736186900714720710555703168729087 386 Stellen
16. $2^{2203}-1$ = 147597991521418023508489862273738173631206614533316977514777121647857029787807894937740733704938928938274850753149648047728126483876025919181446336533026954049696120111343015690239609398909022625932693502528140961498349938822283144859860183431853623092377264139020949023183644689960821079548296376309423663094541083279376990539998245718632294472963641889062337217172374210563644036821845964963294853869690587265048691443463745750728044182367681351785209934866084717257940842231667809767022401199028017047489448742692474210882353680848507250224051945258754287534997655857267022963396259752126374778977855015526465226099888699140135404838098656812504194976866977710076644 Stellen
17. $2^{2281}-1$ = 446087557183758429571151706402101809886208632412859901111991219963404685792820473369112545269003989026153245931124316702395758705693679364790903497461147071065254193353938124978226307947312410798874869040070279328428810311754844108094878252494866760969586998128982645877596028979171536962503068429617331702184750324583009171832104916050157628886606372145501702225925125224076829605427173573964812995250569412480720738476855293681666712844831190877620606786663862190240118570736831901886479225810414714078935386562497968178729127629594924411960961386713946279899275006954917139758796061223803393537381034666494402951052059047968693255388647930440925104186817009640171764133172418132836351 687 Stellen
18. $2^{3217}-1$ =

259117086013202627776246767922441530941818887553125427303974923161874019266586362086201209516800483406550695241733194177441689509238807017410377709597512042313066624082916353517952311186154862265604547691127595848775610568757931191017711408826252153849035830401185072116424747461823031471398340229288074545677907941037288235820705892351068433882986888616658650280927692080339605869308790500409503709875902119018371991620994002568935113136548829739112656797303241986517250116412703509705427773477972349821676443446668383119322540099648994051790241624056519054483690809616061625743042361721863339415852426431208737266591962061753535748892894599629195183082621860853400937932839420261866586142503251450773096274235376822938649407127700846077124211823080804139298087057504713825264571448379371125032081826126566649084251699453951887789613650248405739378594599444335231188280123660406262468609212150349937584782292237144339628858485938215738821232393687046160677362909315071 969 Stellen

19. $2^{4253}-1$ = 1907970075... 1281 Stellen
20. $2^{4423}-1$ = 2855425422... 1332 Stellen
21. $2^{9689}-1$ = 4782202788... 2917 Stellen
22. $2^{9941}-1$ = 3460882824... 2993 Stellen
23. $2^{11213}-1$ = 2814112013... 3376 Stellen
24. $2^{19937}-1$ = 4315424797... 6002 Stellen
25. $2^{21701}-1$ = 4486791661... 6533 Stellen
26. $2^{23209}-1$ = 4028741157... 6987 Stellen
27. $2^{44497}-1$ = 8545098243... 13395 Stellen
28. $2^{86243}-1$ = 5369279955... 25962 Stellen
29. $2^{110503}-1$ = 5219283133... 33265 Stellen
30. $2^{132049}-1$ = 5127402762... 39751 Stellen
31. $2^{216091}-1$ = 7460931030... 65050 Stellen
32. $2^{756839}-1$ = 1741359068... 227832 Stellen
33. $2^{859433}-1$ = 1294981256... 258716 Stellen
34. $2^{1257787}-1$ = 4122457736... 378632 Stellen
35. $2^{1398269}-1$ = 8147175644... 420921 Stellen
36. $2^{2976221}-1$ = 6233400762... 895932 Stellen
37. $2^{3021377}-1$ = 1274116830... 909526 Stellen
38. $2^{6972593}-1$ = 4370757441... 2098960 Stellen
39. $2^{13466917}-1$ = 9249477380... 4053946 Stellen
40. $2^{20996011}-1$ = 1259768954... 6320430 Stellen
41. $2^{24036583}-1$ = 7235733 Stellen
42. $2^{25964951}-1$ = 7816230 Stellen
43. $2^{30402457}-1$ = 9152052 Stellen
44. $2^{32582657}-1$ = 9808358 Stellen
45. $2^{37156667}-1$ = 11185272 Stellen
46. $2^{42643801}-1$ = 12837064 Stellen ; 2009 gefunden
47. $2^{43112609}-1$ = 12978189 Stellen ; 2008 gefunden
48. $2^{57885161}-1$ = 17425170 Stellen ; 2013 gefunden
49. $2^{74207281}-1$ = 22338618 Stellen ; 2016 gefunden
50. $2^{77232917}-1$ = 23249425 Stellen ; 12-2017 gefunden (Jon Pace)
51. $2^{82589933}-1$ = 1488944457... 24862048 Stellen ; 2018 gefunden (Patrick Laroche)

größte bisher (02/2022) bekannte Primzahl (Stand: 12/2018 !)

Anmerkungen:

1) Die Berechnung der 43. Mersenneprimzahl dauert auf meinem Rechner bereits **12s**.
Die Berechnung der 51. Mersenneprimzahl sogar **52s** !

2) Zur Überprüfung der Primalität der 27. Mersenne-Primzahl mit dem **Lucas-Lehmer-Test** , programmiert mit Java17, benötigt mein Rechner **29s** (Ryzen 9 5900X ; Windows 11).
Bei der 28. Mersenne-Primzahl sind es bereits **166s** .
Der **Miller-Rabin-Test** dauert noch sehr viel länger (27. Mersenne-Primzahl: **174s**) !

RepUnit - Zahlen R_n

RepUnits sind Zahlen, die ausschließlich die Ziffer 1 (dezimal) enthalten !
R_n kann nur dann prim sein, wenn n prim ist (notwendig, nicht hinreichend !)
Gemessen an den zerlegbaren RepUnit-Zahlen gibt es nur wenige prime (s.u.),
man vermutet dennoch, dass es unendlich viele prime RepUnit-Zahlen gibt.

Beispiele:

R2 = 11 prim;
R3 = 111 = 3·37
R7 = 1111111 = 239·4649
R11 = 11111111111 = 21649·513239
R17 = 11111111111111111 = 2071723·5363222357
R19 = 1111111111111111111 prim;
R23 = 11111111111111111111111 prim;

R53 = 11111111111111111111111111111111111111111111111111111 =
107·1659431·1325815267337711173·4719885879949142566020007l yafu: 0,3s msi: 0s

R71 = 11111111111111111111111111111111111111111111111111111111111111111111111 =
241573142393627673576957439049·459948113478868463102217288952230343018 39 yafu: 14s msi: 16s

R97 = 1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
111111111 = 12004721·10939984685537053754033926684207011910766229658034803 9·
84603573139691923376721153789909716 9 yafu: 425 s msi: 843 s

R114 = 1111...11 = 3· 7· 11· 13· 37· 21319· 1458973· 10749631· 909090909090909091·
1111111111111111111· 39311230223051293779765 19· 753201806271328462547977919407
yafu: 1 s msi: 22 s

R131 = 1111...11 = 80173·109517·141811693·446790173·7370364319027·15594845538029429933·
731772397003105767769 3·131758351065116151205213·18022206228783402545124708 1
yafu: 1,2s msi: 855s

R200 = 1111...11 = 11·41·73·101·137·251·271·401·1201·1601·3541·5051·9091·21401·25601·27961·
60101·1676321·7019801·5964848081·182521213001·14103673319201·78875943472201·1680588011350901·
12969441902905775055138577118456427449907570094765675782153729152719680 1
yafu: 0,8s msi: 336s brent: 138s

R431 = 1111...11 = 3449· 42362129· 7567197163·
10049662295682655156769587841862238326712165785450544362338167983483011083703916156849256124357671
42097226396821567424584912627196816645970883845618740958081203197221906694509297877794482680071 4
27651194893957926742116010296174904421933279054622701855491680458598498382162071236343774999377 27
62119223771166832044277782443582309364525727387690981501489126090503307810449887713171707038823 89
8180962558635376339757 (zerlegbar; 410 Zeichen) **bisher noch nicht zerlegt !**

Des weiteren sind auch **R317** und **R1031** prim !

Wahrscheinlich prim sind:
R49081 (09-1999; Dubner), **R86453** (10-2000; Baxter), **R109297** (03-2007; Bourdelais-Dubner), **R270343** (07-2007; Voznyy-Budnyy), **R5794777** (04-2021; Batalov-Proper), **R8177207** (05-2021; Batalov-Proper).

Repunit-Zahlen haben die Form $R_n = \frac{10^n - 1}{9}$, da $9 \cdot R_n = 10^n - 1$ gilt !

Beispiel: R_5 (= 11111) = $(10^5 - 1) / 9 = 99999 / 9 = 11111$

Verallgemeinerte („generalized“) RepUnits zur Basis b :

"Generalized" (verallgemeinerte) Repunit-Zahlen zur Basis b haben die Form $R_n(b) = \frac{b^n - 1}{b - 1}$

Für **b = 10** ergibt sich die normale obige Formel für **Repunits**.
Für **b = 2** erhalten wir $2^n - 1$, also die Definition der **Mersenne-Zahlen** !

Betrachten wir einige Beispiele für b = 12: **$R_n(12) = (12^n - 1) / 11$** :

```
R1(12) = 1
R2(12) = 13          prim !
R3(12) = 157         prim !
R4(12) = 1885 = 5· 13· 29
R5(12) = 22621     prim !
R6(12) = 271453 = 7· 13· 19· 157
...
R19(12) = 29043636306420266077    prim !
...
R50(12) = 827312559091104524847962297758415120447922963932416 93 =
13· 1951· 19141· 22621· 60601· 73951· 303551· 438472201· 12629757106815551
brent: 0,3 s
...
R97(12) = 435700623537534460534556100566797400050569661118420894078389027832099599815
930778113305073283279681915 81    prim !
...
R100(12) = 752890677472859547803712941779425907287384374412631304967456240093868108 48
192838457979116663350729035052125 =
5· 5· 5· 13· 29· 101· 1201· 1951· 19141· 22621· 60601· 73951· 303551· 85403261·
438472201· 700936801· 2334798291701· 14807687049800501· 12629757106815551   brent: 14s
...
R107(12) = 269774342001974285278694638169555674601473577261535176943833382434453840404
476293460132377629078395944401301624943 7 =
126047 · 8368183883 · 25576252654191267599857410066384594796702097194341694171659889 7
073179438991591162215340597323148093 7       brent: 0,9s
...
R109(12) = 388475052482842970801320278964160171426121951256610654799120070705613530 18
244586258259062378587289015993787433991894 1   prim !
```

Desweiteren sind prim: $R_{317}(12)$ $R_{353}(12)$ $R_{701}(12)$ $R_{9739}(12)$ * ...

Die mit Sternchen versehene ist eine „Quasiprimzahl“ , deren Primalität noch nicht nachgewiesen werden konnte !

Def.: m heißt quasiprim zur Basis b, wenn $b^{m-1} \bmod m = 1$ gilt

Beispiel: 341 (= 11 · 31) ist quasiprim zur Basis 2, denn $2^{340} \bmod 341 = 1$

Anmerkung: Im Internet existieren Tabellen zur Zerlegung von Repunit-Zahlen (auch generalisierten) !

Links: https://stdkmd.net/nrr/repunit/
https://homes.cerias.purdue.edu/~ssw/cun/xtend/crombie
https://homes.cerias.purdue.edu/~ssw/cun/index.html

Die folgenden Faktorisierungen wurden durchgeführt auf einem **AMD Ryzen 9 5900X**

Programme zur Faktorisierung mit Ermittelung der Berechnungszeit

1) CrypTool2
2) Msieve 1.53
3) Yafu-x64 2.08 (Konsolenprogramm; C++; Ben Buhrow) ; Aufruf z.B.: yafu-x64 factor(1001)
4) alpertron programs (Dario Alpern; verwendet Elliptische Kurven - läuft im Browser; C++)
5) wxMaxima(CAS; GUI) ; Aufruf z.B. factor(576);
6) PARI/GP 2.13.3 (Konsolenprogramm von Karim Belabas)) ; Aufruf z.B. factor(1001);
7) Tilman Neumann: JavaMathLibrary ; Aufruf: PSIQS_U (bzw. PSIQS bzw. SIQS)
8) jLangZahlToolDecGUI www.k-achilles.de/java.html (jar executable GUI von Ac)

Yafu: meint yafu-x64 , V2.08 . yafu/s meint Verwendung von siqs(..)
msi: meint msieve 1.53
ct2: meint CrypTool2 (Quadratisches Sieb; 12 Prozessorkerne ; 24 Threads)
max: meint wxMaxima
pari: meint PARI/GP
Til: meint Tilman Neumann ; Til/P für PSIQS Til/S für SIQS
rho: bzw. brent bzw. fmt(Fermat) bzw. Leh(Lehman): meint jLangzahlToolDecGUI (mein Java-Programm).

Die hinter ct2 in Klammern angegebene Zahl ist diejenige für msi (msieve).

Einige zerlegbare Mersennezahlen:

$2^8-1 = 255 = 3\cdot5\cdot17$

$2^9-1 = 511 = 7\cdot73$

$2^{10}-1 = 1023 = 3\cdot11\cdot31$

$2^{11}-1 = 2047 = 23\cdot89$

$2^{22}-1 = 4194303 = 3\cdot23\cdot89\cdot683$

$2^{23}-1 = 8388607 = 47\cdot178481$

$2^{29}-1 = 536870911 = 233\cdot1103\cdot2089$ 9 Stellen

$2^{41}-1 = 2199023255551 = 13367\cdot164511353$ 13 Stellen

$2^{50}-1 = 1125899906842623 = 3\cdot11\cdot31\cdot251\cdot601\cdot1801\cdot4051$ 16 Stellen

$2^{63}-1 = 9223372036854775807 = 7\cdot7\cdot73\cdot127\cdot337\cdot92737\cdot649657$

$2^{70}-1 = 1180591620717411303423 = 3\cdot11\cdot31\cdot43\cdot71\cdot127\cdot281\cdot86171\cdot122921$ 22 Stellen

$2^{97}-1 = 158456325028528675187087900671 = 11447\cdot13842607235828485645766393$

$2^{137}-1 = 174224571863520493293247799005065324265471 =$
$32032215596496435569\cdot543904218360020429015$ 42 Stellen pari: 0,05s

$2^{147}-1 = 178405961588244985132285746181186892047843327 =$
$7^3\cdot127\cdot337\cdot4432676798593\cdot274167236252872553506872$ 45 Stellen brent: 0,2s

$2^{149}-1 = 713623846352979940529142984724747568191373311 =$
$86656268566282183151\cdot 8235109336690846723986161$ 45 Stellen pari: 0,1s

$2^{157}-1 = 182687704666362864775460604089535377456991567871 =$
$852133201\cdot60726444167\cdot1654058017289\cdot2134387368610417$ 48 Stellen brent: 1s

$2^{257}-1 = 231584178474632390847141970017375815706539969331281128078915168015826259279871 =$
$535006138814359\cdot1155685395246619182673033\cdot3745505985018109365817766300963131813 93$ 78 Stellen
yafu: 0,9s msi: 4s pari: 5s ct2: 13s

$2^{331}-1 = 437450144956602384874500445423524273070633886178642487285154121281990599839875184644702635404610764 7 = 16937389168607\cdot 865118802936559\cdot$
$29854262498019746361376721533356942800568646883582125372179668262555191 9$ 100 Stellen brent: 13s

Einige ausgesuchte Zahlen und ihre Faktorisierung :

digits	Zahl n =	Zeit / s
9	138909787 = 11783·11789	
10	8616460799 = 89681·96079 („**JEVON**s" Zahl)	
10	4294967297 = 641·6700417 (**Fermat F5 = 2^32+1**) Euler; 1732	
10	2^{32} + 2768 = 4294970064 = 2·2·2·2·3·3·7·11·19·19·29·37	
11	17543877193 = 52579·333667	
12	241873694197 = 15619·15485863	
13	4156338464051 = 2038717·2038703	
14	18128486155487 = 4257749·4257763	
14	31415926535897 = 163·44777·4304347	
15	341550071728319 = 4185971·81593989	
15	239812014798221 = 15485863·15485867	
15	191427922719503 = 13835747·13835749	
16	1001001000000007 = 64513·15516268039	
17	68442389367182561 = 15485863·4419669047	
17	10000000000000331 = 31·322580645161301	
18	352523661753384896 = 2^6·83·2603401·25491133	
18	914561491635920053 = 323769703·2824728451	Leh: 0,65
18	965521343020615207 = 982609429·982609483	Fmt: 0,04
19	4160426670634335197 = 2039712397·2039712401	Leh: 0,25
19	8115469396070122309 = 3013189513·2693315293	Fmt: 0,4
20	18446744073709551617 = 274177·67280421310721 (**Fermat F6 = 2^64+1**) zerlegt von Clausen 1855; unveröffentlicht sowie Landry/Lasseur 1880	brent: 0,04
21	109890109889010989011 = 599144041·183411838171	
21	415758465533848642967 = 11110693267·37419669101	
24	662835905978993515936337 = 724170440549·915303730813	brent: 0,7
25	5018557517866741394046901 = 1481124532001·3388342714901	brent: 0,7
26	41615325283752522089299429 = 16927^5·29947	
26	33333332222222222218888889 = 29999999·1111111111111111111	
28	5313531353135313531353135313 = 3·7·11·23·29·239·281·4649·909091·121499449	
32	79798272259984334113870248395353 = 1303·6763·10627·16927·29947·157543·10670053	
33	100000000000000000000000000000001 = 19841·976193·6187457·834427406578561	
34	1240819002867598361280210027487189 = 538119463428139·2305843009213693951	brent: 13
37	231000000000000000000000000000000000 = 2^{34}·3·5^{34}·7·11	
39	340282366920938463463374607431768211457 = *zerlegt von Morrison & Brillhart; 1970* 59649589127497217·5704689200685129054721 (**Fermat F7 = 2^128+1**)	brent:124(34) pari: 0,02
42	281728347264723747372617263727162736372615 = 3·5·11^2·11673043·26741327·4972638385453668937186 61	Probe-Div: 0,3
45	100000000000000000000000000000000000000000001 = 3·137·617·1620583484601296758492708265640210 6953	Probe-Div: 0,3
46	1427247692705959880439315947500961989719490561 = 2305843009213693951· 6189700196426901374495621 11 = **M61**·**M89**	brent: 276
49	85779122932654454031623614621629972200431028761 99 = 12764787846358441471· 6719980305597139683616666935769	pari: 0,3 yafu: 0,4
55	675062234896414305195630546932696211776378888978198 5387 = 7·97·997·9973·99991·999983·9999991·99999989·999999937·9999999967 dies ist das Produkt der jeweils größten ein-, zwei-, drei- bis zehnstelligen Primzahlen	brent: 0,3
59	(5^{91}-1)/78124 = 5169945003752470999575098886703017633412033367157592781 2501 = 305175781· 96806477905685890863555 59· 17499733663152976533452519	yafu: 0,6 ct2: 0 (0) pari: 0,2 Til: 1,6

60	10043362776618689222137263060906276685840468102970909235609 7 = 6189700196426901374495621 11 · 16225927682921336339157801028812 7 = **M89 · M107**	yafu: 1,9 ct2: 0 Til: 0,5
61	(2^214+1)/4285 = 61442410541748650348843656758267463429927126838126669343238 61 = 843589 · 8174912477117 · 23528569104401 · 37866809061660057264219253397	brent: 6 Til: 17
68	19319597769065583697597909403141892104992276797595838342261048259737 = 25424590664090308146038848774534 73 · 75987841945288753799392097264437 69	Yafu: 9,9 ct2: 2 (11) pari: 13 Til: 1,5
71	27606985387162255149739023449107931668458716142620601169954803000803329 = 16225927682921336339157801028812 7 · 17014118346046923173168730371588410572 7 = **M107 · M127**	yafu: 15,5 ct2: 2 (19) pari: 22 Til: 2,3
75	51308301164631635148188579465248381026592967341847801763983519195343090270 7 = 13698630136986301369863013698630136997 · 37455059850181093658177663009631318139 3	yafu: 39 ct2: 9 (53) pari: 70 Til: 4,9
78	11579208923731619542357098500868790785326998466564056403945758400791312963993 7 = 1238926361552897 · 93461639715357977769163558199606896584051237541638188580280321 (**Fermat F8 = 2^256+1**) zerlegt von Brent & Pollard; 1980 [brent: 103s (7)]	ct2: 10 (45) yafu: 0,7 pari: 1 Til: 5,7
80	25389739913858345524208216151645759882986445317021182277812631082013053557679073 = 57841295759118288267473253993921326751 37 · 43895524089907382652596083010742568073 29	yafu: 67 ct2: 16(96) pari: 202 Til: 9,8
84	23986136625956052485865148735412958823329151363927624346327130843836030789585592184 3 = 489756435648946347624324859824637632984391 · 489756435648946347624324859824637632984373 **Progr.„Fermat" löst in 0,03s !!**	yafu: 0,01 ct2: 33(264) pari: 697 Til: 20
85	(11^93+1)/5823582370884 = 12143097918088867233227584561865518009470962182715685846053337560098991375703209766 23 = 237843323473654847623 · 365852473845513195122 3 · 139550866104193032581962716205911186751428 7	yafu: 3,9 ct2: 36 (233) pari: 11 Til: 3,2
87	(5^128+1)/514 = 57173849748165734823382127296801832578728869492805813302954849527160372541572333309900 9 = 23653200983830003298459393 · 24171717725330873572798545219226642215966994254472458802413313	ct2: 68 (399) Til: 39 yafu: 4,7 pari: 80
88	38353230902979262626847647698929219318455512236590939056458280995140151560618248 17478743 = 7432339208719 · 341117531003194129 · 15127682224137352558644030052641058393243747785206318539 93	Yafu: 0,9 ct2: 84 (41) pari: 7 Til: 54
94	17375855758549154624366762941835019305510457496880844836981879745085135987501018 73166735739991 = 23096923021977474330660193223430446887477 43 · 75230175647273284335652420066607907427136274861453 7	yafu/s: 10 pari: 10852 ct2: 328(1826) Til: 169
100	43442698883654206465611893379827125408623206550910149411977322744011293292242275 19682504576978654093 = 58219095811886820515178767227081790299538366398541 · 74619329410444640929930650594192092128852582596673	yafu/s: 4567 ct2: 1548 msi: 8201 pari: - Til:
109	(5^160+1)/1282 = 53371900607145248472073115100684817548883806654203780552126052885 57426303375236909669637698215227603168457793 = 75068993 · 241931001601 · 46957667265666758402894952584920394200961 · 62582663240692632675871452234418855417095109446 41	yafu: 708 ct2: 1885 pari: 4410 Til: 807
121	73281095113106734398239989296282702447648950464623884587479737832729774008066025 61983902737712247132163867744658163132181 = 2756163353 · 598990818061 · 4527716228491 · 248158049830971629 · 33637310674071348724927955857253537 · 11744593722752035313978951707661039 9	yafu: 18 pari: 35 ct2: 2d (476) Til:
134	(10^142+1)/440729761 = 22689640874966916518260721676111180519983083239073569166117 64731721849843491735517266327744088967933345440676968488179766920709491184 1 = 3806238494887148 09 · 77169265188335087786895085049 41 · 93611382287513950329431625811490669 · 82519882659061966708762483486719446639288430446081 (1991 faktorisiert)	yafu/s: > 1d msi: > 1d ct2: 24 Tage ! pari: 1789 Til:
137	(2^484+1)/1254743089 = 39807333662911990505576361597868630214848364295425530264663 21613189291518872533907371447387108570999155129110094863887018959911630851195 3 = 33186913 · 1251287137 · 2931542417 · 38608979869428210686559330362638245355335498797441 · 84694409197705740057696939084347325062258739942360856026657 29	yafu:Absturz nach 30s ct2: 11326 msi: 51156 pari: Abbruch Til: 7575
155	13407807929942597099574024998205846127479365820592393377723561443721764030073546	yafu: > 2d

	976801874298166903427690031858186486050853753882811946569946433649006084097 = 2424833· 74556028256478842083373957362004549187833663426 57· 7416400626275308015247871419019374740599407810975190239058213161444157595047050080928187116939407 37 (**Fermat F9 = 2^512+1**) zerlegt von Lenstra & Manasse; 1990	ct2: 3968d pari: Til:
162	$(12^{151}-1)/11$ = 82219620528659701952660120743076100427390924357073396551677033937335320743050235802427303275633200540806689460669679221954509396712733084562446289606063026821231 7 = 16537237851564688924261407041648853990657743· 4971786780032337881876339900596001648747659834953921156974700575915322824191116704320092701688428573103024883134912641 9 (1993 faktorisiert)	yafu:Absturz ct2: pari: Abbruch nach 20 h
211	1451227665928429038078135931392086641805560994157364861845488884120406459417175207376752494760562665707898465687588427397594196091371554080090795137543260169489098207940366177919426111059426911414577902662170527 = 47·523·599·2417·1523·8191·32497·11491·4639· 196717· 48397· 100673· 178481· 993683· 489553· 1126847· 2299159· 9316273· 9341359· 17094767· 7504421· 14718679249· 13444476836590589479· 51441563151591093599· 2602424497125099161599259277368903526464607082594529975972217 Anmerkung zu * : Pari sowie ct2 und Til benötigen die von kleinen Faktoren bereinigte 98-stellige Zahl 532625402624252052802620263859734619694804385495449365900704994145164583371556186403831320165180 19	yafu: 2,8 msi: 4888 ct2: 908 * pari: 16 * Til: 404 *
284	$(2^{953}+1)/5721$ = 133080345593536956502815591514201902951792557813251173402649045965698587041798119079728400108727853784649003072990526516663651048019400755530310305457861571124083389321639478318240769417087998824954110861836031436725517800720411572301421894329402716046524575072980542451628711517762 33 = 425796183929· 16247002794788943855987796558425843 77· 3802306738549441324432139091271828121·1280648868301666714448025761291158720600277· 33884958374667213943683932046721815228158303686049930480849258405552811 77· 1165882340667125990314837655838327081813101225814639260043952099413134433416292453613 9 (2002 faktorisiert)	yafu: einige Tage ct2: Absturz msi: Abbruch pari: / Til: Abbruch
309	1797693134862315907729305190789024733617976978942306572734300811577326758055009631327084773224075360211201138798713933576587897688144166224928474306394741243777678934248654852763022196012460941194530829520850057688381506823424628814739131105408272371633505106845862982399472459384797163048353563296242241372 17 = 45592577· 6487031809· 4659775785220018543264560743076778192897· 1304398744054881897274847687965099039466085308416118921868952957768324162514718635741402279775731048958987839288429238448311490329137987290886016179460941194490105959067101305319061710183544916096191939124885381160807122996723228062178207531270144245 77 (**Fermat F10 = 2^1024+1**) zerlegt von Brillhart 1962 ; Brent 1995	yafu: 2601 Til: Abbruch pari: / msi, c2t: Eingabe > 10^274
313	$2^{1039}-1$ = 589068086431683676644738724917747624711938696459815017753575689937658432079465555993259138490065014034006389161562581754376322314451080388584562460719428810761069833174599222153387113189363201210623862217392146903328852155899782370013718480620182690736866953411252382072659135491210334387684495620912657652829388 7 = 5080711· 5585366661993629126074920465831594496864652701848863764801005234631985328837475 3·207581819464423827645704813703594695162939708007395209881208387037927290903246793823431438841448348825340533447691122230281583276965253760914101891052419938993341097116243589620659721674811617490048036597355734092532054255236 8 (**Mersenne M1039**) zerlegt von Thorsten Kleinjung; 2007	yafu/s: Ausstieg pari:
617	3231700607131100730071487668866995196044410266971548403213034542752465513886789089319720141152291346368871796092189801949411955915049092109508815238644828312063087736730099609175019775038965210679605763838406756827679221864261975616183809433847617047058164585203630504288757589154106580860755239912393038552191433338966834242068497478656456949485617603532632205807780565933102619270846031415025859286417711672594360371846185735759835115230164590440369761323328723122712568471082020972515710172693132346967854258065669793504599726835299863821552516638943733554360213543322960464531847860495214819355585361105959623065 7 = 319489· 974849· 167988556341760475137· 3560841906445833920513· 1734624471791475554302589708643097783774218447236640846493470190613635791928791088575910383304088371779838108684515464219407129783061341898642808260145427587085892438736855639731189488693991585455066111474202161325570172605641393943669457932209686651089596854827053880726458285541519364019124649311825460928798157330577955733585049822792800909428725675915189121186227517143192297881009792510360354969172799126635273587832366471931547770914277453770382945849189175903251109393813224860442985739716507110592444621775425407069130470346646436034913824417233065988341 77 (**Fermat F11 = 2^2048+1**) zerlegt von Brent & Morain; 1988	yafu: 15 pari: 241 Til: Abbruch

Semiprimzahlen

Eine **Semiprimzahl** ist eine natürliche Zahl, die zerlegbar ist in ein Produkt aus **genau 2** Primzahlen .
Es handelt sich also hier nicht um eine Primzahl !
Die ersten Semiprimzahlen sind 4 (=2·2) 6 (=2·3) 9 (=3·3) 10 (=2·5) 14 (=2·7) 15 (=3·5) 21 (=3·7)

Beispiel 1: 80313965845625220873194079672 1
besteht aus 30 Ziffern (100 Bits) und lässt sich zerlegen in
878529705721751· 914186114852471
Yafu: 0,17s brent(Ac): 8s Lehman(Ac): 0,3 s

Beispiel 2: 713623846352979940529142984724747568191373311
besteht aus 45 Ziffern (149 Bits) und lässt sich zerlegen in
86656268566282183151· 8235109336690846723986161
Yafu: 0,26s pari: 0,1s

Beispiel 3: 38!+1 = 523022617466601111760007224100074291200000001 =
45 Ziffern (149 Bits)
37280713718589679646221· 14029308060317546154181
Yafu: 0,3 s pari: 0,1s

Beispiel 4: 10^{48} + 19 = 1000000000000000000000000000000000000000000000019 =
49 Ziffern (160 Bits)
17507530776122167975943559 13· 571182774308676717563
Yafu: 0,4s pari: 0,2s

Beispiel 5: 85397342226735670654637567334028930584606539118838378 3 =
54 Ziffern (180 Bits)
27182818284590452353603 53 · 31415926535897932384626433831 1
Yafu: 1s Til/P: 0,5s pari: 0,6s

Beispiel 6: 71641520761751435455133616475667090434063332228247871795429
besteht aus 59 Ziffern (196 Bits) und lässt sich zerlegen in
35744050410138836561078538901 7· 20042921812081555426974363543 7
Yafu: 2s Til/P: 0,5s pari: 2,5s

Beispiel 7: 10^{63} + 19 = 1000000000000000000000000000000000000000000000000000000000000019
= 64 Ziffern (210 Bits)
44741590990368441481914582608543· 22350568628980377780095228766733
Yafu: 3,5s Til/P: 0,7s

Beispiel 8: 729346944528564617209248390517758983860666588441034039195491780030381328027527 9
besteht aus 79 Ziffern (262 Bits) und lässt sich zerlegen in
84818438291948899360848100931373480897 7· 85989197539586788824000429721336460377 27
Yafu: 65s Til/P: 11s pari: 216s

Beispiel 9: 16368054432676235348632643877118481207371456933092253539382430638081302047541099874017838 57 =
91 Ziffern (300 Bits)
13485329506402520485674964872111548148784559687 71·
12137674815365144903021465040591855759822 67
Yafu: 7s Til/P: 89s

Beispiel 10:
11288798737163099824081460333619591342348211143669600740142907237723834164788215269828
1999652360869 =
99 Ziffern (326 Bits)
36190234934785044449782293780001398745689734619 37·
31192941293433332564432124322469839822238973411637
Yafu/s: 3510s Til/P: 513s

Beispiel 11: $(11^{104}+1) / (2 \cdot 17 \cdot 6304673) =$ 941234360735926294697117213629451435752898137898308254134753221194264012130159069863408961146891168 1 =

100 Ziffern (333 Bits) - wurde 1988 faktorisiert mit MPQS

867592223134283908122180770958507080489 77 ·

1084881048536374706129613998429729484098346115257905772167 53

pari: 20635s ct2: 798s msi: 10s yafu: 2671s Til/P: 416s

Beispiel 12: 28810398274578959718816270531375307346387908251661274960666743202415714464947623866204429538207354 53 =

100 Ziffern (331 Bits)

618162834186865969389336374155487198277265679 ·

466064872898356637396439537520952929159659540064606830 7

yafu/s: 4279s Til/P: 632s

Beispiel 13: 18305793363806612289463999598616113379051784851055493866944917116280421806056361920816522436937410941183836997361687856 17 =

121 Ziffern (400 Bits)

785506617513464525087105385677215644061830019071786760495463 ·

233044419431550225562286848781148674808128493437968665268915 9

yafu/s: Ausstieg Til/P: pari:

Beispiel 14: $(6^{353}-1) / 5$ = 973691505184416442565958983076531038101774699445446034442467673403970145084942466298465294694187891794816051886144204066226423206167081784681898063663685509304513573706979052346135130666317823161124260153050164931265319361687960957823878998047485678787428763591656991956664 3 =

274 Ziffern (911 Bits)

135095261330112651830775049635590807381121031111382732318390846759744072165636542920143351738198057636666351316191686483 ·

720744381111301937643935864029025391613890867099707817049849566271785734074845094811610876273732867041786794660514517682420730722427836886613902736846235 21

yafu/s: Til/P:

RSA – Zahlen

RSA-Zahlen sind Semiprimzahlen (s. oben !), welche in der sogenannten Factoring Challenge of RSA Security (ein mit Preisgeldern ausgelobter Wettbewerb) gelistet waren ;
Bezeichnungen: RSA-100 etc. bis RSA-500 .

Diese Zahlen sind schwer zu faktorisieren, wenn sie genügend groß sind (etwa 100 Ziffern und mehr !) .
Daher werden sie in der **Kryptographie** verwendet.
Benannt wurden sie nach den Wissenschaftlern **R**IVEST, **S**HAMIR und **A**DLEMAN.

Berühmtes Beispiel für eine RSA-Zahl aus „Spektrum der Wissenschaft“:

Die **129-stellige** Zahl „**RSA-129**“ =
114381625757888867669235779976146612010218296721242362562561842935706935245733897830597123563958705058989075147599290026879543541 (426 Bits)

ist Produkt zweier Primzahlen. Wie lauten diese Faktoren?
Diese Frage stellte **Martin Gardner** (1914 - 2010) den Lesern des Scientific American im August 1977 in seiner Kolumne "Mathematical Recreations".
Im Gegensatz zu den Rätseln, die Gardner sonst aufzugeben pflegte, musste dieses ungewöhnlich lange auf eine Lösung warten. Erst mehr als 16 Jahre später, **im April 1994**, präsentierten Paul Leyland von der Universität Oxford, Michael Graff von der Universität von Iowa in Iowa City und Derek Atkins vom Massachusetts Institute of Technology in Cambridge die 64- bzw. 65-stelligen Primfaktoren

349052951084765094914784961990389813341776463849338784399082057 sowie
327691329932667095499619881908344614131776429679929425397982885 33

Beispiel 2: RSA-100 =
152260502792253336053561837813263742971806811496138068865790849458012296325895289765400035069200 6139 = (100 Ziffern; 330 Bits) ; wurde 1991 faktorisiert mit MPQS
37975227936943673922808872755445627854565536638199 ·
40094690950920881030683735292761468389214899724061
yafu: 4195s ct2: 1287s Til/P: 3521s pari: 28856s

Beispiel 3: RSA-120 =
227010481295437363334259960947493668895875336466084780038173258247009162675779735389791151574049166747880487470296548479 = (120 Ziffern; 397 Bits) ; wurde 1993 faktorisiert
33274145556934980157511463037491414880636424032401714634068 83 ·
69334266711083018119732540189970064136196586312733668067 3013

Beispiel 4: RSA-140 =
212902463182587575474978820162715174978067039632772162782333832153819499840564959113665738530219183167831073879953172308895692308734419364 71 =
(140 Ziffern; 463 Bits) ; wurde 1999 faktorisiert
33987174230284385545301236276138758356339864959695974234909293027714 79 ·
62642001874012850961516549482644422193020371786235090191116606539460 49

Beispiel 5: RSA-170 =
260626236841398449215298792666744321970859253804864064161647851918599996285420693614502839319145146186835121981648059198820530572229741164780650958098323773365107115457 59 =
(170 Ziffern; 563 Bits) ; wurde 2009 faktorisiert
35864207304285014867998045872685204232914596810599781611402318606339484508580405939 63 ·
72670290641070190788637977639239462641361378038569966703137089360022815822495874944 93

Weitere Beispiele für RSA-Zahlen

RSA 200

```
RSA-200 = 27997833911221327870829467638722601621070446786955428537560009929326128400107609345671052955360856061822351910951365788637105954482006576775098580557613579098734950144178863178946295187237869221823983 =
```
(200 Ziffern; 663 Bits)
```
35324619344027701212726049781984643686711974001976250236493034687761212536794232000585479565280883490·
79258699544783330333470858414800596877379758573642199607343303414557678728181521353814093047401854 67
```
zerlegt 05-2005 von Gruppe Prof. Jens Franke

RSA 260

```
RSA-260 = 22112825529529666435281085255026230927612089502470015394413748319128822941402001986512729726569746599085900330031400051170742204560859276357953757185954298838958709229238491006703034124620545784566413664540684214361293017694020846391065875914794251435144458199
```
Bis heute noch **nicht** zerlegt !

RSA 270

```
RSA-270 = 23310853034440754452763765691068052414561981248030544904294861196849591824513578286788836931857711641821391926857265831491306067262691135402760979316634162669394659619642774427388660187689631346870405906674690312391074827760654864915192081269930976658751473545659499320 7
```
Bis heute noch **nicht** zerlegt !

RSA 400

```
RSA-400 = 2014096878945207511726700485783442547915321782072704356103039129009966793396141985086509455102260403208695558793091390340438867513766123418942845301603261911930567685648626153212566300102683464717478365971313989431406854640516317519403149294308737302321684840956395183222117468443578509847947119995373645360710979599471328761075043464682551112058642299370598078702810603300890715874500584758146849481
```
Bis heute noch **nicht** zerlegt !

RSA 576

Die Primfaktorzerlegung dieser 174-stelligen Zahl wurde im Dezember 2003 von Jens Franke und Thorsten Kleinjung vom Mathematischen Institut in Bonn und dem Institut für Experimentelle Mathematik in Essen gefunden. Das Preisgeld lag bei 10.000 US$.

```
RSA-576 = 188198812920607963838697239461650439807163563379417382700763356422988859715234665485319060606504743045317388011303396716199692321205734031879550656996221305168759307650257059 =
3980750864240649373971255005503864911990643623425267084063851895759463889572617685833317 ·
4727721461074353025362230719730482246329146953020971164598521711305207112563635903975 27
```

Anmerkung zur (anscheinend verwirrenden) Bezeichnung der RSA-Zahlen:

Die ersten erzeugten RSA-Nummern von RSA-100 bis RSA-500 wurden entsprechend ihrer Anzahl von Dezimalstellen gekennzeichnet.
Später, beginnend mit RSA-576, werden stattdessen Binärziffern gezählt.
Eine Ausnahme bildet RSA-617, das vor der Änderung des Nummerierungsschemas erstellt wurde !

RSA 640

Die Faktoren dieser 193-stelligen Zahl wurden im November 2005 von F. Bahr, M. Boehm, J. Franke, T. Kleinjung gefunden, die zuvor schon RSA 200 faktorisiert hatten. Das Preisgeld lag bei 20.000 US$.

```
RSA-640 =
3107418240490043721350750035888567930037346022842727545720161948823206440518081504556346829671723286782437916272838033415471073108501919548529007337724822783525742386454014691736602477652346609
```

wird zerlegt in

```
163473364580925384844313388386509085984178367003309231218111085238933310010450815121211816751157 9 ·
190087128166482211312685157393541397547189678996851549366663853908802710380210449895719126146557 1
```

RSA 768

Die Faktorisierung dieser 232-stelligen Zahl wurde am 12. Dezember 2009 von Thorsten Kleinjung et al. vollendet.[1] Der RSA Factoring Challenge war zu dieser Zeit schon beendet, sodass kein Preisgeld ausgezahlt wurde.

```
RSA-768 =
123018668453011775513049495838496272077285356959533479219732245215172640050726365751874520219978646938995647494277406384592519255732630345373154826850791702612214291346 3
429214311602221240479274737794080665351419597459856902143413 =

334780716989568987860441698482126908177047949837137685689124313889828837938780022876147116525317430877378144679994 89 ·
367460436667995904282446337996279526322791581643430876426760322838157396665112792333734171433968102700927987363089 17
```