Primzahlen(1): Erzeugung und Primalitäts-Test

Ac 2013-2024
14.07.2024

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
$\mathbf{n = p_1^{t1} \cdot p_2^{t2} \cdot p_3^{t3} \cdot \dots \cdot p_k^{tk}}$ **($t_i \in \mathbb{N}$)** 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.)

Die Anzahl π der Primzahlen in einem bestimmten Bereich [0 ; x] ist der folgenden Tabelle zu entnehmen:

x	π(x)
10^2	25
10^3	168
10^4	1229
10^5	9592
10^6	78498
10^7	664579
10^8	5761455
$3 \cdot 10^8$	16252325
10^9	50847534
10^{10}	455052511
10^{11}	4118054813
10^{12}	37607912018
10^{13}	346065536839
10^{14}	3204941750802
10^{15}	29844570422669

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 += 2)
      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} mod n = 1

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

Beispiel: n = 7: 1^6 mod 7 = 1 mod 7 = 1 2^6 mod 7 = 64 mod 7 = 1 3^6 mod 7 = 729 mod 7 = 1
4^6 mod 7 = 4096 mod 7 = 1 5^6 mod 7 = 15625 mod 7 = 1 6^6 mod 7 = 46656 mod 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} mod n ungleich 1, war n auf alle Fälle keine Primzahl.
Ist das Ergebnis von a^{n-1} mod 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 mod 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} mod 341 = 1 .
Für a = 3 gilt aber: 3^{340} mod 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} mod 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} mod 561 = 1 Für a=3: 3^{560} mod 561 = 1 Für a=4: 4^{560} mod 561 = 1
Für a=5: 5^{560} mod 561 = 1 Für a=500: 500^{560} mod 561 = 1 Für a=560: 4^{560} mod 561 = 1
Jedoch ist 561 keine (!) Primzahl, denn 561 = 3·11·17

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 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:

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

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: $a \in \{2,3, \ldots , \min(n-1, 2\cdot(\ln n)^2)\}$

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:

```
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()" .