Zahlen und Primzahlen
Notizen zur Vorlesung
Wintersemester 2003/04

H.-G. Gräbe, Institut für Informatik,
`http://www.informatik.uni-leipzig.de/~graebe`

3. Februar 2004

Inhaltsverzeichnis

0 Einleitung

In der Vorlesung „Einführung in das symbolische Rechnen“ ging es um prinzipielle Fragen und Probleme, deren Betrachtung für den qualifizierten Einsatz von Computeralgebra-Werkzeugen im alltäglichen Gebrauch von Vorteil sind. Diese Vorlesung soll einen tieferen Einblick in die Algorithmen vermitteln, die für die grundlegenden Funktionalitäten des Rechnens mit exakten Zahlen, sowie Primtests und Faktorisierung zum Einsatz kommen.

Im Unterschied zu der bisher angebotenen Vorlesung „Grundlegende Algorithmen der Computeralgebra“ wird es in dieser Vorlesung nur (und damit intensiver) um Primtest- und Faktorisierungsalgorithmen gehen. Das ist durch die enorme Bedeutung dieser Algorithmen für vielfältige Einsatzgebiete, insbesondere im Zusammenhang mit kryptografischen Verfahren, mehr als gerechtfertigt. Ich werde in der Vorlesung auch ein wichtiges neueres theoretisches Ergebnis darstellen: den von drei indischen Mathematikern im August 2002 erbrachten Beweis, dass das Primtestproblem in Polynomialzeit entschieden werden kann.

Das Thema „Polynom-Algorithmen“ wird in einer extra Vorlesung – ebenfalls in größerer Ausführlichkeit als bisher – wahrscheinlich im Sommersemester abgehandelt werden.

Neben den Algorithmen selbst wird auch deren Laufzeitverhalten untersucht, um auf diese Weise grundlegende Vorstellungen über *Komplexitätsfragen* im Zusammenhang mit Anwendungen des symbolischen Rechnens zu erarbeiten. Diese Kenntnisse können in der Vorlesung „Algebraische Komplexitätstheorie“ weiter vertieft werden.

Dieser Kurs stellt deutlich höhere Anforderungen an die mathematische Vorbildung der Teilnehmer als die Vorlesung „Einführung in die Computeralgebra“. Insbesondere werden Grundkenntnisse der höheren Algebra, wie etwa über endliche Körper und das Rechnen in Restklassenringen, als bekannt vorausgesetzt. In der ersten *Übungen zur Vorlesung* können einige dieser Vorkenntnisse repetiert werden. Außerdem liegt ein Studienmaterial zu diesen Fragen im Netz.

Die Vorlesung orientiert sich stark am Buch [3], wobei der Schwerpunkt auf *praktikablen* Verfahren für die verschiedenen grundlegenden algorithmischen Fragestellungen für Zahlen und Primzahlen liegt. Auch die wichtigsten Ideen für laufzeiteffiziente Verfahren werden dargestellt, ohne allerdings bis in die letzten Details einer getrimmten Implementierung oder eines vielleicht theoretisch interessanten, aber praktisch bedeutungslosen Verfahrens zu verzweigen.

Weitere Referenzen sind die Bücher [2, 6, 8].

Die Vorlesung wird von Übungsaufgaben begleitet, die wöchentlich zum Vorlesungstermin abzugeben sind. In etwa 14-täglichem Turnus finden *Besprechungen der Übungsaufgaben* statt, in denen einzelne Fragen aus der Vorlesung und den Übungsaufgaben vertieft werden.

Wenn wenigstens 50 % der Punkte aus den Übungsaufgaben erreicht wurden, wird ein *Übungsschein* (1 SWS) ausgestellt. Eine Note (Modulprüfung) zur Vorlesung wird bei erfolgreicher Teilnahme an der Klausur erteilt.

Details zum Ablauf der Vorlesung finden Sie auf meiner Web-Seite.

1 Primzahlen – grundlegende Eigenschaften

1.1 Teilbarkeit von Zahlen

Teilbarkeit, Primzahleigenschaft

Notation: $\mathbb{N}$ und $\mathbb{P} = \{p_1, p_2, \ldots\}$, p_n die n-te Primzahl

Definition gcd, lcm

Wegen der Beziehung $\operatorname{lcm}(a,b) = \dfrac{a\,b}{\gcd(a,b)}$ zwischen dem kleinsten gemeinsamen Vielfachen

$\mathrm{lcm}(a,b)$ und dem größten gemeinsamen Teiler $\gcd(a,b)$ der Zahlen $a,b \in \mathbb{Z}$ können wir uns im folgenden auf den gcd beschränken.

Den größten gemeinsamen Teiler kann man bekanntlich mit dem *Euklidschen Algorithmus* berechnen.

```
procedure Euklid(a,b:int):int
  while (b ≠ 0) do
    (q,r):= DivMod(a,b);
    a:= b; b:= r;
  return a;
end;
```

Beispiel: `Euklid(2134134,581931)`

$$\begin{aligned}
2134134 &= 3\cdot 581931 + 388341\\
581931 &= 1\cdot 388341 + 193590\\
388341 &= 2\cdot 193590 + 1161\\
193590 &= 166\cdot 1161 \quad + 864\\
1161 &= 1\cdot 864 \quad + 297\\
864 &= 2\cdot 297 \quad + 270\\
297 &= 1\cdot 270 \quad + 27\\
270 &= 10\cdot 27 \quad + 0
\end{aligned}$$

Der gcd ist also gleich 27.

Satz 1
$g = \gcd(a,b)$ kann für $a,b \in \mathbb{Z}$ als ganzzahlige Linearkombination $g = u\cdot a + v\cdot b$ mit geeigneten $u,v \in \mathbf{Z}$ dargestellt werden.

u,v können mit `ExtendedEuklid` berechnet werden:

```
procedure ExtendedEuklid(a,b:int):(int,int,int)
  (u1,v1):=(1,0); (u2,v2):=(0,1);
  while (b ≠ 0) do
    (q,r):=divmod(a,b); u3:= u1 − q · u2; v3:= v1 − q · v2;
    a:=b; b:=r; u1:=u2; v1:=v2; u2:=u3; v2:=v3;
  return (a,u1,v1);
end;
```

Beispiel: `ExtendedEuklid(2134134,581931)`

$$\begin{aligned}
388341 &= 1\cdot 2134134 + (-3)\cdot 581931\\
193590 &= (-1)\cdot 2134134 + 4\cdot 581931\\
1161 &= 3\cdot 2134134 + (-11)\cdot 581931\\
864 &= (-499)\cdot 2134134 + 1830\cdot 581931\\
297 &= 502\cdot 2134134 + (-1841)\cdot 581931\\
270 &= (-1503)\cdot 2134134 + 5512\cdot 581931\\
27 &= 2005\cdot 2134134 + (-7353)\cdot 581931\\
0 &= (-21553)\cdot 2134134 + 79042\cdot 581931
\end{aligned}$$

Erster Satz von Euklid: $p|ab \;\Rightarrow\; p|a$ oder $p|b$. Beweis mit Euklidschem Algorithmus

Satz von der Eindeutigkeit der Primfaktorzerlegung (EPZ). Beweis

Folgerung: (mit Beweis)

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s} = \prod_{p\in\mathbb{P}} \frac{1}{1-p^{-s}} \text{ für } s > 1 \tag{1.1}$$

1.2 Zur Verteilung der Primzahlen

Sieb des Eratostenes

Zweiter Satz von Euklid: Es gibt unendlich viele Primzahlen. Beweis

Formal ergibt sich das auch aus der Formel (1.1) für $s = 1$:

$$\sum_{n=1}^{\infty} \frac{1}{n} = \prod_{p\in\mathbb{P}} \left(\frac{1}{1-\frac{1}{p}}\right)$$

und wegen $\sum_{n=1}^{\infty} \frac{1}{n} = \infty$ weiter

$$\prod_{p\in\mathbb{P}} \left(1-\frac{1}{p}\right) = 0$$

Diese Aussagen sind als Grenzwertaussagen auch in einem mathematisch strengen Sinn richtig. Damit haben wir einen weiteren Beweis, dass die Menge der Primzahlen unendlich ist.

Die Argumentation lässt sogar einen Schluss zu, wie „dicht“ diese Primzahlen liegen: Wegen $\ln(1-x) = -x - \frac{x^2}{2} - \ldots$ gilt

$$-\infty = \ln\left(\prod_{p\in\mathbb{P}} 1-\frac{1}{p}\right) = \sum_{p\in\mathbb{P}} \ln\left(1-\frac{1}{p}\right) \sim -\left(\sum_{p\in\mathbb{P}} \frac{1}{p}\right)$$

Da die Summe der Reziproken der Primzahlen divergiert, existieren damit deutlich mehr Primzahlen als z.B. Quadratzahlen.

Aber wir können sogar eine obere Schranke für die Größe von p_n, der n-ten Primzahl herleiten: Da sich jede Zahl $n = q\cdot m^2$ in das Produkt aus einer Quadratzahl m^2 und einem quadratfreien Teil q zerlegen lässt, ergibt sich

$$\sum_{n\le x} \frac{1}{n} = \sum_{q\le x} \frac{1}{q} \sum_{m\le\sqrt{x/q}} \frac{1}{m^2} < \sum_{q\le x} \frac{1}{q} \sum_{m=1}^{\infty} \frac{1}{m^2} < 2\cdot \sum_{q\le x} \frac{1}{q}$$

wegen

$$\sum_{m=1}^{\infty} \frac{1}{m^2} < 1 + \sum_{m=2}^{\infty} \left(\frac{1}{m-1} - \frac{1}{m}\right) = 2$$

Aus

$$\sum_{q\le x} \frac{1}{q} \le \prod_{p\le x} \left(1+\frac{1}{p}\right) < \exp\left(\sum_{p\le x} \frac{1}{p}\right)$$

und der Integralabschätzung

$$\sum_{n\le x} \frac{1}{n} \sim \ln(x)$$

folgt schließlich

$$\sum_{p\le x} \frac{1}{p} \sim \ln(\ln(x))$$

Dies kann man in die Aussage $p_n \sim n\ln(n)$ umformulieren, d.h. die Wahrscheinlichkeit, dass eine Zahl $n \sim x$ eine Primzahl ist, beträgt etwa $\frac{1}{\ln(x)}$. In der Tat, setzen wir diese Größe in die Reziprokensumme ein und ersetzen wieder durch das Integral, so ergibt sich

$$\int_2^x \frac{d\,t}{t\ln(t)} = \ln(\ln(x)) - \ln(\ln(2))$$

Die Aussage über p_n ist äquivalent zur folgenden Aussage des „Prime Number Theorem" über das Wachstum der *Primzahldichtefunktion* $\pi(x) = |\{p \leq x\ :\ p \in \mathbb{P}\}|$:

$$\pi(x) \sim \frac{x}{\ln(x)}$$

Die Primzahldichtefunktion kann auf folgende Weise auch für sehr große Werte x ausgerechnet werden, ohne dabei alle Primzahlen wirklich zu erzeugen:

Sei $\phi(x,a) := |\{k \leq x\ :\ p_1,\dots,p_a \not| k\}|$ die Anzahl der Zahlen kleiner x, die durch keine der ersten a Primzahlen teilbar sind. Diese Anzahl kann rekursiv über die Formel

$$\phi(x,a) = \phi(x,a-1) - \phi(\lfloor\frac{x}{p_a}\rfloor, a-1),\ \ a > 1$$

berechnet werden. Damit ergibt sich die Primzahldichtefunktion zu

$$\pi(x) = \pi(\sqrt{x}) + \phi(x, \pi(\sqrt{x})) - 1,$$

wobei der erste Summand alle Primzahlen $\leq \sqrt{x}$ und der zweite alle Primzahlen $> \sqrt{x}$ (und die 1) zählt. Rechnet man also etwa alle Primzahlen bis 10 000 mit dem Sieb des Eratostenes aus, so kann man mit dieser Formel $\pi(x)$ für jedes $x \leq 10^8$ bestimmen.

2 Das Rechnen mit ganzen Zahlen
Die Langzahlarithmetik und deren Komplexität

Grundlage des symbolischen Rechnens ist die Möglichkeit, Rechnungen *exakt*, also ohne Rundungsfehler auszuführen. Die Basis für solche Fähigkeiten liegt im exakten Rechnen mit ganzen und gebrochenen Zahlen. Die entsprechenden Verfahren benutzen dazu die Darstellung ganzer Zahlen in einem Positionssystem mit einer fixierten Basis β (meist eine Zweierpotenz):

$$z = \pm \sum_{i=0}^{m} a_i \beta^i =: [\varepsilon, m; a_0, \ldots, a_m]$$

$a_0, \ldots, a_m$ sind dabei *Ziffern* aus dem entsprechenden Positionssystem, d.h. natürliche Zahlen mit der Eigenschaft $0 \le a_i \le \beta - 1$.

Die Zahl $l(z) := m + 1 = \left[\frac{\log z}{\log \beta}\right] + 1$ nennt man die *Wort-* oder *Bitlänge* von z.

Auf der Seite der Zahlen haben wir also die Datentypen DIGIT und ZAHL (als ARRAY of DIGIT, wenn wir mal vom Vorzeichen absehen), für die eine Reihe von Operationen zu definieren (und zu implementieren) sind, zu denen mindestens $+, -, *, /, gcd, lcm$ gehören und die wir weiter unten genauer betrachten wollen.

Außerdem benötigen wir Ein- und Ausgabeprozeduren, die uns die Verbindung zwischen diesem Datentyp ZAHL und dem Datentyp STRING (als ARRAY of CHAR) herstellen. Die Ein- und Ausgabe erfolgt dabei normalerweise nicht im Zahlsystem β, sondern in einem anderen Zahlsystem c. Die Verbindung zwischen beiden Datentypen stellen die Funktionen

$$val : CHAR \longrightarrow DIGIT \quad \text{und} \quad symb : DIGIT \longrightarrow CHAR$$

her, die einzelne Zeichen in DIGITs und umgekehrt verwandeln.

2.1 Ein- und Ausgabe

Die Transformationen, die für die Ein- und Ausgaberoutinen benötigt werden, sind aus dem Grundkurs Informatik gut bekannt. Wir wollen uns hier auf vorzeichenlose ganze Zahlen beschränken. Als STRING sind sie in Form eines Arrays $s = [a_m \ldots a_0]$ von CHAR's gespeichert, der die Positionsdarstellung der Zahl im Zahlsystem mit der Basis c symbolisiert.

Die Umrechnung eines Strings in eine Zahl erfolgt mit dem Hornerschema.

Beispiel: $[1A2CF]_{16}$ im 16-er-System ist ins Dezimalsystem zu verwandeln.

$$\begin{aligned} 1 \cdot 16^4 + 10 \cdot 16^3 + 2 \cdot 16^2 + 12 \cdot 16 + 15 \\ &= (((1 \cdot 16 + 10) \cdot 16 + 2) \cdot 16 + 12) \cdot 16 + 15 \\ &= 107\,215 \end{aligned}$$

```
procedure STRING-to-ZAHL(s=[a_m ... a_0]_c:STRING):ZAHL
  /* Eingabe einer im Positionssystem zur Basis c gegebenen Zahl */
  z:=0;
  for i:=m downto 0 do z:= c·z + val(a_i);
  return z;
```

Die Umrechnung einer Zahl in einen String erfolgt durch fortgesetzte Division mit Rest.

Beispiel: 21357 ist im 6-er-System auszugeben.

$$\begin{array}{ll} 21357 & = 3559 \cdot 6 + 3 \\ 3359 & = 593 \cdot 6 + 1 \\ 593 & = 98 \cdot 6 + 5 \\ 98 & = 16 \cdot 6 + 2 \\ 16 & = 2 \cdot 6 + 4 \end{array}$$

folglich gilt

$$21357 = 2 \cdot 6^5 + 4 \cdot 6^4 + 2 \cdot 6^3 + 5 \cdot 6^2 + 1 \cdot 6 + 3 = [242513]_6$$

```
procedure ZAHL-to-STRING(z:ZAHL):STRING
  /* Ausgabe einer Zahl z im Positionssystem zur Basis c */
  i:=0;
  while z ≠ 0 do
    (q,r):=divmod(z,c);
    a_i:=symb(r); z:=q; i++;
  return [a_m ... a_0]_c;
```

Betrachten wir die Kosten, die mit diesen Umrechnungen verbunden sind. Wir können davon ausgehen, dass $l(\beta) = l(c) = 1$ gilt, d.h. beide Zahlen vom Typ DIGIT sind und somit die auszuführenden Multiplikationen und Divisionen die folgenden Signaturen haben

$$\begin{aligned}&\mathit{Dmult} : (\mathit{ZAHL}, \mathit{DIGIT}) \longrightarrow \mathit{ZAHL}\\ &\mathit{Ddivmod} : (\mathit{ZAHL}, \mathit{DIGIT}) \longrightarrow (\mathit{ZAHL}, \mathit{DIGIT})\end{aligned}$$

Diese benötigen ihrerseits die elementaren Operationen

$$\begin{aligned}&\mathit{Emult} : (\mathit{DIGIT}, \mathit{DIGIT}) \longrightarrow \mathit{DoubleDIGIT}\\ &\mathit{Edivmod} : (\mathit{DoubleDIGIT}, \mathit{DIGIT}) \longrightarrow (\mathit{DIGIT}, \mathit{DIGIT})\end{aligned}$$

aus denen sich jeweils die aktuelle Ziffer sowie der Übertrag ergeben.

Komplexität:

$$\begin{aligned}&C_{Dmult}(z) = C_{Ddivmod}(z) = l(z)\\ &C_{ZAHL-to-STRING}(z) = C_{STRING-to-ZAHL}(z) \sim \tfrac{1}{2}\, l(z)^2\\ &(\text{jeweils } \textstyle\sum_{i=0}^{m} (i+1) = \frac{(m+1)(m+2)}{2})\end{aligned}$$

2.2 Arithmetik ganzer Zahlen

Vergleich zweier Zahlen

Vergleich `comp(a,b:ZAHL):{-1,0,+1}`

„Normalerweise“ in konstanter Zeit ausführbar, nämlich, wenn sich die Zahlen im Vorzeichen oder der Wortlänge unterscheiden.

Am aufwändigsten wird der Vergleich, wenn die beiden Zahlen gleich sind, denn dann müssen wirklich alle Zeichen verglichen werden.

Komplexität:

$$C_{comp}(a,b)) = \begin{cases} \text{worst case:} & \min(l(a), l(b)) + 2 \\ \text{best case:} & 1 \end{cases}$$

Untersuchen wir, wieviel Vergleiche *durchschnittlich* notwendig sind, um zwei (positive) Zahlen a, b derselben Länge m zu vergleichen. Der Durchschnittswert berechnet sich aus der Formel

$$d = \sum_{k=1}^{\infty} p(k) \cdot k = \sum_{k=1}^{\infty} p(\geq k),$$

wobei $p(k)$ die Wahrscheinlichkeit angibt, dass genau k Vergleiche notwendig sind und $p(\geq k)$ die Wahrscheinlichkeit, dass mindestens k Vergleiche benötigt werden. Mindestens k Vergleiche mit

$1 < k \leq m$ werden benötigt, wenn die Zahlen a und b in den ersten $k-1$ Stellen übereinstimmen. Die entsprechende Wahrscheinlichkeit ist also

$$\frac{\beta-1}{(\beta-1)^2}\cdot\frac{\beta}{\beta^2}\cdot\ldots\cdot\frac{\beta}{\beta^2}=\frac{1}{(\beta-1)\beta^{k-2}},$$

denn das Paar $(a_i, b_i), i < m$, kann β^2 Werte annehmen, wovon in β Fällen beide Ziffern gleich sind. Für $i = m$ ist die Ziffer 0 auszuschließen. Folglich gilt (geom. Reihe)

$$d = 1 + \frac{1}{(\beta-1)}\cdot\frac{1}{1-\frac{1}{\beta}} = 1 + \frac{\beta}{(\beta-1)^2} \approx 1$$

Addition und Subtraktion

Addition und Subtraktion laufen wie beim schriftlichen Rechnen üblich ab. Übertrag kann propagieren, bis über die erste Stelle der größeren Zahl hinaus, die Wahrscheinlichkeit ist allerdings gering, da der Übertrag höchstens 1 sein kann, d.h. er auf die Ziffer $\beta - 1$ treffen muss. Für $l(a) > l(b)$ ist die Wahrscheinlichkeit, dass überhaupt ein propagierender Übertrag entsteht, ein wenig größer als $\frac{1}{2}$.
Wir sehen also:

$$l(a \pm b) \leq \max(l(a), l(b)) + 1$$

$$C_{add}(a,b)) = \begin{cases} \text{worst case:} & \max(l(a), l(b)) + 1 \\ \text{best case:} & \min(l(a), l(b)) \\ \text{average case:} & \min(l(a), l(b)) + \frac{1}{2} \end{cases}$$

Multiplikation (klassisches Verfahren)

Hierfür brauchen wir eine Multiplikationstabelle für das „kleine EinmalEins“ im fremden Positionssystem. Man beachte, dass man im Unterschied zum schriftlichen Rechnen in der Schule mit einem Akkumulator c arbeiten muss, um den Übertrag korrekt zu bearbeiten.

```
procedure mult(a,b:ZAHL):ZAHL
  for i:=0 to l(a)+l(b)-1 do c_i := 0;
  for i:=0 to l(a)-1 do
    r:=0;
    for j:=0 to l(b)-1 do
      t:= a_i · b_j + c_{i+j} + r;
      (r,c_{i+j}):= Edivmod(t,β);
    c_{i+l(b)}:= r;                   // evtl. verbliebener Übertrag
  return c;
```

Beweis : Für den Beweis der Korrektheit ist zu zeigen, dass t und r die entsprechenden Bereiche DoubleDIGIT und DIGIT nicht verlassen. Beweis mit Induktion: Ist $a_i, b_j, c_{i+j}, r \leq \beta - 1$ beim Eintritt in die innerste Schleife, so gilt

$$t \leq (\beta-1)^2 + (\beta-1) + (\beta-1) = \beta^2 - 1 < \beta^2.$$

□

Wir erhalten damit für den Berechnungsaufwand folgende Abschätzungen:

Länge: $l(a \cdot b) = l(a) + l(b)$ (oder $l(a) + l(b) - 1$, wenn kein Übertrag stattfindet, was aber eher unwahrscheinlich ist).

Komplexität: $C^*_{mult}(a,b) = 2\,l(a)\,l(b)$.

Hierbei haben wir nur die Elementarmultiplikationen und -divisionen gezählt. Aber auch die Berücksichtigung aller arithmetischen Elementaroperationen führt zum qualitativ gleichen Ergebnis.

Binäres Multiplizieren

Besonders einfach ist die Multiplikation, wenn die beiden Faktoren als Bitfelder zur Basis 2 vorliegen. Dann kommt man allein mit Additionen und Shiftoperationen aus:

```
procedure bin-mult(a,b:ZAHL):ZAHL
  c=0;
  while (a > 0) do
    if odd(a) then c:= c + b;
    a:= a/2; b:= 2b;                 // Shiftoperationen
  return c;
```

Die Komplexität ist jedoch ebenfalls von der Größenordnung $O(l(a)\,l(b))$.

Multiplikation und Quadrieren

Bei der Berechnung von $a^2 = a \cdot b$ mit $b = a$ kann man einige Rechenschritte sparen, da etwa $a_i b_j = a_j b_i$ gilt. Insgesamt kann das klassische Verfahren für die Berechnung eines Quadrats so um den Faktor 2 beschleunigt werden. Für höhere Potenzen sind weitere Geschwindigkeitsvorteile (binäres Potenzieren) bekannt.

Frage: Wie weit kann man das beim Quadrieren treiben?

Antwort: Das beste Quadrierverfahren ist höchsten um den Faktor 2 besser als das beste Multiplikationsverfahren:

$$4x\,y = (x+y)^2 - (x-y)^2$$

Karatsuba-Multiplikation

Idee: Sind a, b beides Zahlen der Länge $2l$, so zerlegen wir sie in

$$a = A_1 \cdot \beta^l + A_2, \quad b = B_1 \cdot \beta^l + B_2$$

und erhalten

$$a \cdot b = (A_1 B_1)\beta^{2l} + (A_1 B_2 + A_2 B_1)\beta^l + (A_2 B_2)$$

Die drei Koeffizienten kann man mit *drei* Multiplikationen l-stelliger Zahlen berechnen wegen

$$(A_1 B_2 + A_2 B_1) = (A_1 + A_2)(B_1 + B_2) - A_1 B_1 - A_2 B_2$$

Komplexität: Bezeichnet $C_{Karatsuba}(l)$ die Laufzeit für die Multiplikation zweier l-stelliger Zahlen mit dem Karatsuba-Verfahren, so gilt

$$C_{Karatsuba}(2\,l) = 3\,C_{Karatsuba}(l),$$

wenn man nur die Multiplikationen berücksichtigt und

$$C_{Karatsuba}(2\,l) = 3\,C_{Karatsuba}(l) + 6\,l,$$

wenn auch die Additionen (zwei l-stellige und zwei $2l$-stellige) berücksichtigt werden. In beiden Fällen erhält man

$$C_{Karatsuba}(l) = O(l^{\alpha}) \text{ mit } \alpha = \frac{\log 3}{\log 2} \approx 1.58$$

In praktischen Anwendungen wird der durch das zusätzliche rekursive Zerlegen notwendige Mehraufwand erst für Zahlen mit mehreren hundert Stellen durch das schnellere Grundverfahren wettgemacht.

Die schnellsten heute theoretisch bekannten Multiplikationsverfahren beruhen auf der schnellen Fourier-Transformation und haben eine Laufzeit von $O(l\ \log(l)\ \log\log(l))$. Wegen des großen implementatorischen Aufwands werden sie nur in speziellen Applikationen eingesetzt, in denen mit mehreren Millionen Stellen zu rechnen ist wie etwa die Weltrekordrechnungen zur Bestimmung möglichst vieler Stellen von π, vgl. [1]. In CAS spielen diese Algorithmen gegenwärtig keine Rolle.

2.3 Division mit Rest

Beispiel: 125... : 2... = (3|4|5|6)...

Allgemeines Schema

```
procedure divmod(a,b:ZAHL):(ZAHL,ZAHL)
  q:= 0; r:= a;
  while r ≥ b do
    Errate die nächste Ziffer q_i des Quotienten
    q:= q + (q_i β^i);
    r:= r − (q_i β^i) · b;
    Evtl. notwendige Korrektur
  return (q,r);
```

Für den Berechnungsaufwand ergeben sich folgende Abschätzungen:

Länge: Wegen $a = q \cdot b + r$ ergibt sich für die Länge des Quotienten $l(q) \leq l(a) - l(b) + 1$ und für den Rest $l(r) \leq l(b)$.

Komplexität: Wenn korrektes Ziffernraten des Quotienten mit konstantem Aufwand c möglich ist und evtl. notwendige Korrekturen zunächst unberücksichtigt bleiben, dann gilt

$$C_{divmod}(a,b) = l(q) \cdot (l(b) + c) = O(l(q) \cdot l(b))$$

denn der Hauptaufwand entsteht beim Berechnen der $l(q)$ Zwischenprodukte $(q_i\beta^i) \cdot b$ mit `Dmult`.

Erraten der aktuellen Ziffer

q_i kann man nicht in konstanter Zeit *korrekt* erraten. Bsp.: $20\ldots01 : 10\ldots01$.

Aus komplexitäts*theoretischer* Sicht ist diese Frage irrelevant, denn selbst wenn alle β Ziffern durchprobiert werden, so ist die Laufzeit noch immer $\beta \cdot l(q)\, l(b) = O(l(q)\, l(b))$. Für praktische Zwecke sollte die Näherung von q_i jedoch nicht allzu weit vom wirklichen Ergebnis entfernt sein.

Verwende zum Erraten eines Näherungswerts für q_i `EDivmod` auf den jeweils ersten signifikanten Ziffern von a und b, so dass q_i garantiert zu klein wird und führe dann ggf. Korrektur durch:

$a = [a_n a_{n-1} \ldots]$, $b = [b_m \ldots]$. Berechne den aktuellen Quotienten q als `EDivmod`($[a_n a_{n-1}] = [ac], d+1$) mit $d = b_m$, wobei $a_n = 0$ sein kann (Division muss immer ein DIGIT ergeben!, d.h. $[ac]_\beta = a\beta + c \leq \beta(d+1)$ sein, was durch die evtl. erforderliche Korrekturphase gesichert wird).

Evtl. notwendige Korrektur ==
  `while` $a \geq \beta^i \cdot b$ `do` { `a:=` $a-b$; q_i`++;` }

Wie groß kann die Abweichung werden? Der exakte Wert der Quotientenziffer (d.h. vor dem Abschneiden der Nachkommastellen) liegt im Intervall

$$\frac{a\beta+c}{d+1} \leq q \leq \frac{a\beta+c+1}{d}$$

$$\to\ \Delta = \frac{a\beta+c+1}{d} - \frac{a\beta+c}{d+1} = \frac{a\beta+c+(d+1)}{d(d+1)} \leq \frac{\beta+1}{d}$$

Für kleine d sind also besonders große Korrekturen zu erwarten.

Beispiel: $100899 = 101 \cdot 999$, $100899 : 101 = (5..9)(4..9)(4..9)$
(Zwischenergebnisse sind 999 und 909)

Knuths Trick: Finde vorher ein Skalierungs-DIGIT k, so dass $k \cdot b$ mit einer Ziffer $\geq \left\lfloor \frac{\beta}{2} \right\rfloor$ beginnt und berechne dann `divmod(a,b)` aus `divmod(k a, k b) = (q,k r)`.

In obigen Beispiel kommt z.B. $k = 5$ in Betracht. Rechne dann $504\,495 : 505 = (8..9)(8..9)(7..9)$ ($d = 6$, Zwischenergebnisse sind 4999, 4545)

Oder $k = 9$. Rechne dann $908\,091 : 909 = 9(8..9)(8..9)$ ($d = 10$, Zwischenergebnisse sind 899 und 818)

Damit sind höchstens 3 Korrekturen notwendig (beachte, dass `EDivmod` noch ganzen Teil nimmt).

Rechnung mit $[ace] : [df]$. Differenz analog oben $\Delta \leq \frac{\beta+1}{d\beta+f} < 1$ (fast immer). Damit höchstens eine Korrektur notwendig.

Beispiel: $100899 : 101 = 999$ (mit $[df] + 1 = 11$)

Geht bei Koprozessor und `real`-Arithmetik recht einfach zu implementieren.

Binäre Division mit Rest

Besonders schnell geht es wieder, wenn die Zahlen als Bitfelder gegeben sind.

```
procedure bin-divmod(a,b:ZAHL):(ZAHL,ZAHL)
  s:=0; q:=0;
  while (a ≥ b) { b = 2b; s++; }       // Shiftoperationen, b = b_0 · 2^s
  while (s > 0)
    b = b/2; s--;                      // Shiftoperationen
    if (a ≥ b) { a = a − b; q = q + 2^s; } // Es gilt immer a < 2b
  return (q,a);
```

Aber auch hier ist die Komplexität von der Ordnung $O(l(q)\,l(b))$.

2.4 Die Berechnung von gcd und lcm

Wegen der Beziehung

$$\mathrm{lcm}(a,b) = \frac{a\,b}{\gcd(a,b)}$$

zwischen dem kleinsten gemeinsamen Vielfachen $\mathrm{lcm}(a,b)$ und dem größten gemeinsamen Teiler $\gcd(a,b)$ der Zahlen $a, b \in \mathbb{Z}$ können wir uns im folgenden auf den gcd beschränken.

Komplexität des Euklidschen Algorithmus

Den größten gemeinsamen Teiler kann man bekanntlich als Folge von Divisionen mit Rest mit dem *Euklidschen Algorithmus* berechnen.

```
procedure Euklid(a,b:ZAHL):ZAHL
  while (b ≠ 0) do
    (q,r):= DivMod(a,b);
    a:= b; b:= r;
  return a;
end;
```

Mit $r_0 = a$ und $r_1 = b$ können wir die Folge der Reste so aufschreiben:

$$\begin{aligned} a &= q_1\, b + r_2 \\ b &= q_2\, r_2 + r_3 \\ &\dots \\ r_{i-1} &= q_i\, r_i + r_{i+1} \\ &\dots \\ r_{m-1} &= q_m\, r_m \end{aligned}$$

Dann ist $\gcd(a, b) = r_m$ und es werden zu dessen Berechnung insgesamt m Divisionen mit Rest ausgeführt. Besonders lang wird diese Kette, wenn die Quotienten q_i klein sind.

Fibonacci-Zahlen: $F_1 = F_2 = 1,\ F_{m+2} = F_{m+1} + F_m$ für $m > 0$.

Satz 2 (Satz von Lamè, 1845) *Seien $a > b > 2$ zwei Zahlen, für die der Euklidsche Algorithmus genau m Schritte benötigt. Dann gilt $a \geq F_{m+2},\ b \geq F_{m+1}$.*

Beweis : Sei die Folge der Divisionen mit Rest in der Berechnung von $\gcd(a, b)$ wie oben bezeichnet. Wegen $r_{i-1} \leq r_i + r_{i+1}$ und $r_{m-1} \geq 2 = F_3$ ergibt sich mit Induktion $r_i \geq F_{m-i+2}$ und schließlich die Behauptung. □

Korollar 1 *Der Euklidsche Algorithmus benötigt für die Berechnung des gcd zweier Zahlen a, b (mit $a \geq b$) im Zahlsystem zur Basis β höchstens $2.08\, \ln(\beta)\, l(a)$ Divisionen mit Rest.*

Beweis : Aus der Binetschen Formel

$$F_i = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^i - \left(\frac{1-\sqrt{5}}{2} \right)^i \right) \sim \frac{1}{\sqrt{5}}\, \alpha^i$$

mit $\alpha = \frac{1+\sqrt{5}}{2} \approx 1.618$ ergibt sich

$$a \geq F_{m+2} \approx \frac{\alpha^2}{\sqrt{5}} \alpha^m \approx 1.17\, \alpha^m > \alpha^m,$$

also $l(a) = \log_\beta(a) \geq m \cdot log_\beta(\alpha)$ und folglich

$$m < l(a) \cdot \frac{\ln(\beta)}{\ln(\alpha)}$$

□

Im Allgemeinen treten entweder viele, aber billige Divisionen oder wenige, aber teure Divisionen auf.

Beispiel: `Euklid(2134134,581931)`

$$\begin{aligned}
2134134 &= 3 \cdot 581931 + 388341 \\
581931 &= 1 \cdot 388341 + 193590 \\
388341 &= 2 \cdot 193590 + 1161 \\
193590 &= 166 \cdot 1161 \quad + 864 \\
1161 &= 1 \cdot 864 \quad + 297 \\
864 &= 2 \cdot 297 \quad + 270 \\
297 &= 1 \cdot 270 \quad + 27 \\
270 &= 10 \cdot 27 \quad + 0
\end{aligned}$$

Deshalb ist eine genauere Analyse der Komplexität der gcd-Berechnung notwendig.

Satz 3 *Für das Laufzeitverhalten sowohl von* `Euklid` *als auch* `ExtendedEuklid` *gilt*

$$C_{gcd}(a,b) = O(l(a)\,l(b)).$$

Beweis : Ansatz wie im letzten Beweis. Die Gesamtkosten dieser m Divisionen mit Rest sind von der Größenordnung

$$C = \sum_{i=1}^{m} l(q_i)\,l(r_i) \leq l(r_1) \left(\sum_{i=1}^{m} l(q_i) \right),$$

wobei $l(q_i) \sim l(r_{i-1}) - l(r_i)$ gilt, also insgesamt

$$C \leq l(a)\,l(b).$$

□

Der binäre gcd-Algorithmus

Wenn die Zahlen als Bitfelder gespeichert sind, kann man wieder eine binäre Version des gcd-Algorithmus angeben, die nur mit Shiftoperationen und Additionen auskommt und damit besonders schnell ist.

Bezeichnen wir Divisionen durch 2 als Shift-Operation S und die Berechnung der Differenz mit D (die für Zahlen der Bitlänge $l(z) \leq l$ nur l sehr einfache Elementaroperationen benötigt), so können wir zur Berechnung von $\gcd(a,b)$ wie folgt vorgehen:

Durch Shiften wird zunächst die größte Zweierpotenz gefunden, die in beiden Argumenten enthalten ist. Danach ist eine der verbleibenden Zahlen ungerade und wir können durch Anwenden von S aus der anderen Zahl alle Faktoren 2 heraus dividieren, ohne den gcd zu ändern. Sind beide Zwischenergebnisse ungerade, so bilden wir mit D die Differenz zwischen größerer und kleinerer Zahl. Wegen $\gcd(a,b) = \gcd(a-b,b)$ bleiben wir damit auf der richtigen Spur. Nach endlich vielen Schritten ist eine der beiden Zahlen gleich 0 und die andere folglich der gesuchte gcd.

Beispiel: gcd(2 134 134,581 931)=27

	2134134		581931
S	1067067		581931
D	485136		581931
$4S$	30321		581931
	30321	D	551610
	30321	S	275805
	30321	D	245484
	30321	$2S$	61371
	30321	D	31050
	30321	S	15525
D	14796		15525
$2S$	3699		15525
	3699	D	11826
	3699	S	5913
	3699	D	2214
	3699	S	1107
D	2592		1107
$5S$	81		1107
	81	D	1026
	81	S	513
	81	D	432
	81	$4S$	27
D	54		27
S	27		27

```
procedure bin-gcd(a,b:ZAHL):ZAHL
  s=0;
  while (even(a) and even(b)) { s++; a:= a/2; b:= b/2; }
    // nun ist eine der beiden Zahlen ungerade
  while (even(a)) a:= a/2;
  while (even(b)) b:= b/2;
    // nun sind beiden Zahlen ungerade
  while (a ≠ b)
    if (a < b) (a,b):= (b,a);        // danach ist a > b.
    a:= a − b; while (even(a)) a:= a/2;
  return a · 2^s;
```

Durchschnittliche Kosten: Jeder zweite Schritt ist ein Shift, wo die summarische Binärlänge um mindestens 1 abnimmt. Also haben wir höchstens $(l(a) + l(b))$ Differenzbildungen von Zahlen der maximalen Längen $l(a)$ und $l(b)$, also (average) $l(b)$ Elementaradditionen. Damit höchstens $2\,l(a) \cdot l(b)$ Elementar*additionen*.

Es ist hier zwar nicht offensichtlich, wie das geht, aber man kann diesen Algorihmus auch zu einer erweiterten Version aufbohren, die nicht nur $g = \gcd(a, b)$ berechnet, sondern auch Kofaktoren $u, v \in \mathbb{Z}$ mit $g = a \cdot u + b \cdot v$. Details siehe [3, Alg. 9.4.3].

3 Zahlentheoretische Vorbereitungen

Ein zweites wichtiges Verfahren, um das Rechnen mit langen Zahlen und die durch die Prozessorgröße beschränkten Möglichkeiten eines Computers in Einklang zu bringen, besteht in der Verwendung von Restklassen. Es handelt sich dabei um einen Spezialfall eines generellen Prinzips, des *Rechnens in homomorphen Bildern*, bei dem man versucht, die geforderten Rechnungen zuerst in einem oder mehreren (einfacher handhabbaren) Bildbereichen durchzuführen, um aus der so gewonnenen Information Rückschlüsse zu ziehen und vielleicht sogar das exakte Ergebnis zu extrahieren.

Im Fall der ganzen Zahlen benutzt man dafür deren Reste bei Division durch eine geeignete Zahl, die nahe an der Wortgröße des verwendeten Computers liegt. Die entsprechenden Operationen auf den Resten lassen sich in konstanter Prozessorzeit ausführen und liefern bereits Teilinformationen. So kann man etwa aus der Tatsache, dass ein Rest verschieden von Null ist, bereits schlussfolgern, dass die zu untersuchende Zahl selbst auch verschieden Null ist. Aus der Kenntnis der Reste bei Division durch verschiedene Moduln kann man in vielen Fällen auch die Zahl selbst rekonstruieren, insbesondere, wenn man zusätzlich Informationen über ihre Größe besitzt. Eine auf diesem Prinzip begründete Arithmetik bezeichnet man als *modulare Arithmetik*.

Da grundlegende Kenntnisse des Rechnens mit Resten auch für die weiteren Betrachtungen von Primtest- und Faktorisierungsverfahren wesentlich sind, wollen wir zunächst ein Kapitel zu zahlentheoretischen Grundlagen einschieben, das auf den aus dem Grundkurs bekannten Fakten über das Rechnen in Restklassenringen aufbaut.

3.1 Ein wichtiger Satz über endliche Mengen

Satz 4 *Sei $\phi : M \to M$ eine Abbildung einer endlichen Menge in sich selbst. Dann gilt*

$$\phi \text{ ist injektiv, d.h. } \phi(x_1) = \phi(x_2) \ \Rightarrow x_1 = x_2 \tag{1}$$

genau dann, wenn

$$\phi \text{ ist surjektiv, d.h. } \forall\, y \in M\ \exists\, x \in M\ :\ y = \phi(x) \tag{2}$$

Beweis : Offensichtlich, denn

(1) heißt: jedes $y \in M$ hat *höchstens* ein Urbild,

(2) heißt: jedes $y \in M$ hat *mindestens* ein Urbild

In Wirklichkeit hat wegen der Gleichmächtigkeit in beiden Fällen jedes $y \in M$ *genau* ein Urbild.
□

Dieser Satz ist für unendliche Mengen falsch. So ist z.B. die Abbildung $\phi_1 : \mathbb{N} \to \mathbb{N}$ via $\phi_1(n) = 2\,n$ zwar injektiv, aber nicht surjektiv, die Abbildung $\phi_2 : \mathbb{N} \to \mathbb{N}$ via $\phi_2(n) = n\ div\ 10$ surjektiv, aber nicht injektiv.

3.2 Der Restklassenring $\mathbb{Z}_m$

Bekanntlich nennt man zwei Zahlen $a, b \in \mathbb{Z}$ *kongruent modulo m* (und schreibt $a \equiv b \pmod{m}$), wenn ihre Differenz durch m teilbar ist, also bei Division durch m der Rest 0 bleibt. So gilt $127 \equiv 1 \pmod{7}$, aber ebenso $127 \equiv 8 \pmod{7}$, denn in beiden Fällen ist die Differenz durch 7 teilbar.

Die eingeführte Relation ist eine Äquivalenzrelation, so dass wir die zugehörigen Äquivalenzklassen betrachten können, die als *Restklassen* bezeichnet werden. Die Restklasse $\pmod{7}$, in der sich die Zahl 1 befindet, besteht etwa aus den Zahlen

$$[1]_7 = \{\ldots, -20, -13, -6, 1, 8, 15, \ldots, 127, \ldots\} = \{7k + 1 \mid k \in \mathbb{Z}\}.$$

Die Darstellungen $z \equiv 1 \pmod 7$, $7 \,|\, (z-1)$, $z = 7k+1$, $z \in [1]_7$ und $[z]_7 = [1]_7$ sind also äquivalent zueinander. Wir werden diese unterschiedlichen Schreibweisen im Weiteren frei wechselnd verwenden. Die Menge der Restklassen modulo m bezeichnen wir mit $\mathbb{Z}_m$.

Addition und Multiplikation sind mit der Restklassenbildung verträglich, so dass die Menge $\mathbb{Z}_m$ sogar einen Ring bildet. Im Gegensatz zu den ganzen Zahlen kann dieser Ring aber Nullteiler besitzen. So ist etwa $2, 3 \not\equiv 0 \pmod 6$, dagegen $2 \cdot 3 = 6 \equiv 0 \pmod 6$.

In diesem Zusammenhang spielen die primen Restklassen eine besondere Rolle. Eine Restklasse $[a]_m$ heißt *prim*, wenn ein (und damit jeder) Vertreter dieser Restklasse zu m teilerfremd ist, wenn also $\gcd(a, m) = 1$ gilt. So sind etwa $\pmod 7$ alle Restklassen verschieden von $[0]_7$ prim, $\pmod 8$ dagegen nur die Restklassen $[1]_8, [3]_8, [5]_8$ und $[7]_8$ und $\pmod 6$ gar nur die beiden Restklassen $[1]_6$ und $[5]_6$.

Prime Restklassen haben bzgl. der Multiplikation eine besondere Eigenschaft. Es gilt für eine prime Restklasse $a \pmod m$ die *Kürzungsregel*

$$a \cdot x \equiv a \cdot y \bmod m \;\Rightarrow\; x \equiv y \bmod m.$$

Dies lässt sich sofort aus $m|(a\,x - a\,y) = a\,(x-y)$ und $\gcd(a, m) = 1$ herleiten.

Anders formuliert: Die Multiplikationsabbildung

$$m_a : \; \mathbb{Z}_m \to \mathbb{Z}_m \quad \text{via} \quad (x \bmod m) \mapsto (a\,x \bmod m)$$

ist injektiv und somit, als Abbildung zwischen gleichmächtigen endlichen Mengen, auch surjektiv. Zu einer primen Restklasse $a \in \mathbb{Z}_m$ gibt es also stets ein $x \in \mathbb{Z}_m$, so dass $m_a(x) = a{\cdot}x \equiv 1 \pmod m$ gilt. a ist also zugleich ein *invertierbares Element* des Ringes $\mathbb{Z}_m$. Umgekehrt überzeugt man sich, dass invertierbare Elemente prime Restklassen sein müssen, d.h. die Menge der primen Restklassen mit der Gruppe der im Ring $\mathbb{Z}_m$ invertierbaren Elemente zusammenfällt. Wir bezeichnen deshalb diese Gruppe der primen Restklassen mit $\mathbb{Z}_m^*$.

Da die Menge aller Restklassen $\mathbb{Z}_m$ endlich ist, ist es auch die Menge der primen Restklassen $\mathbb{Z}_m^*$. Ihre Anzahl bezeichnet man mit dem Symbol $\phi(m)$. Die zugehörige Funktion in Abhängigkeit von m bezeichnet man als die *Eulersche ϕ-Funktion*.

Der Ring $\mathbb{Z}_m$ ist genau dann ein Körper, wenn alle von 0 verschiedenen Elemente ein Inverses besitzen, d.h. prime Restklassen sind. Das ist genau dann der Fall, wenn m eine Primzahl ist. Da diese Eigenschaft für endliche Ringe mit der Nullteilerfreiheit zusammenfällt, spielen in modularen Rechnungen Restklassenringe modulo Primzahlen in der Größe eines Computerworts eine besondere Rolle.

3.3 Der Chinesische Restklassensatz

Der folgende Satz gibt nähere Auskunft über Zahlen, die bei Division durch gegebene Moduln vorgegebene Reste lassen.

Satz 5 (Chinesischer Restklassensatz) *Seien $m_1, \ldots, m_n$ paarweise teilerfremde natürliche Zahlen und $m = m_1 \cdot \ldots \cdot m_n$ deren Produkt. Das System von Kongruenzen*

$$\begin{gathered} x \equiv x_1 \bmod m_1 \\ \ldots \\ x \equiv x_n \bmod m_n \end{gathered}$$

hat für jede Wahl von $(x_1, \ldots, x_n)$ genau *eine Restklasse $x \pmod m$ als Lösung.*

Anders formuliert, ist die natürliche Abbildung

$$P : \mathbb{Z}_m \to \mathbb{Z}_{m_1} \times \ldots \times \mathbb{Z}_{m_n} \qquad \text{mit } [x]_m \mapsto ([x]_{m_1}, \ldots, [x]_{m_n})$$

ein Isomorphismus.

Beispiel: $P : \mathbb{Z}_{30} \to \mathbb{Z}_2 \times \mathbb{Z}_3 \times \mathbb{Z}_5$ bildet die Restklasse $[17]_{30}$ auf das Tripel $([1]_2,\ [2]_3,\ [2]_5)$ ab.

Beweis : Injektivität ist trivial, denn $x \equiv 0 \pmod{m_i}$ bedeutet $m_i|x$ und wegen der Teilerfremdheit auch $m|x$, also $x \equiv 0 \pmod{m}$. Die Surjektivität folgt nun wieder aus der Injektivität und der Gleichmächtigkeit der endlichen Mengen auf beiden Seiten des Pfeils. $\square$

Da mit $x \in \mathbb{Z}_m^*$, also $\gcd(x,m) = 1$, auch für jeden Teiler $m_i \,|\, m$ $\gcd(x, m_i) = 1$, also $x \in \mathbb{Z}_{m_i}^*$ folgt, induziert P eine (wie P bijektive) Abbildung

$$P^* : \mathbb{Z}_m^* \to \mathbb{Z}_{m_1}^* \times \ldots \times \mathbb{Z}_{m_n}^*$$

Ist insbesondere $m = p_1^{a_1} \ldots p_k^{a_k}$ die Primfaktorzerlegung von m, so gilt (für $m_i = p_i^{a_i}$)

$$\phi(m) = \phi(p_1^{a_1}) \cdot \ldots \cdot \phi(p_k^{a_k}).$$

Für Primzahlen p hat man

$$\phi(p^a) = p^a - p^{a-1} = p^{a-1}(p-1) = p^a \left(1 - \frac{1}{p}\right).$$

Insbesondere sehen wir an der zweiten Formel, dass für ungerade Primzahlen p die Eulerfunktion $\phi(p^a)$ stets eine gerade Zahl ist.

Zusammengenommen erhält man die Formel

$$\phi(m) = m \prod_{p \in \mathbb{P},\, p|m} \left(1 - \frac{1}{p}\right)$$

Beispiele: $\phi(12) = 4,\ \phi(18) = 6,\ \phi(24) = 8,\ \phi(36) = 12.$

Der angegebene Beweis ist allerdings nicht konstruktiv. Für Anwendungen des Satzes brauchen wir auch eine algorithmische Lösung, die nicht alle Restklassen $\pmod{m}$ prüfen muss (Die Laufzeit eines solchen Verfahrens wäre $O(m)$, also exponentiell in der Bitlänge von m), sondern bei vorgegebenen $(x_1, \ldots, x_n)$ die Lösung x in akzeptabler Laufzeit findet.

Wir suchen also einen **Chinesischen Restklassen-Algorithmus**

$$\texttt{CRA}((x_1, m_1), (x_2, m_2), \ldots, (x_n, m_n)) \ \to\ (x, m)$$

zur Berechnung von x.

Betrachten wir diese Aufgabe zunächst an einem konkreten Beispiel:

> Gesucht ist eine Restklasse $x \pmod{30}$, so dass $x \equiv 1 \pmod{2}$, $x \equiv 2 \pmod{3}$, $x \equiv 2 \pmod{5}$ gilt.

Lösung: $x = 5y + 2$ wegen $x \equiv 2 \pmod{5}$. Da außerdem noch $x = 5y + 2 \equiv 2 \pmod{3}$ gilt, folgt $y \equiv 0 \pmod{3}$, also $y = 3z$ und somit $x = 15z + 2$. Schließlich muss auch $x = 15z + 2 \equiv 1 \pmod{2}$, also $z \equiv 1 \pmod{2}$, d.h. $z = 2u + 1$ und somit $x = 30u + 17$ gelten. Wir erhalten als einzige Lösung $x \equiv 17 \pmod{30}$, also

$$\texttt{CRA}((1, 2), (2, 3), (2, 5)) = (17, 30).$$

Für ein allgemeines Verfahren kann man auf zwei verschiedene Weisen vorgehen.

1) (Lagrange-Verfahren) Bestimme zunächst

$$1 = u_{kj} \cdot m_k + u_{jk} \cdot m_j$$

für alle Paare (j,k) mit $j \neq k$ mit Hilfe von `ExtendedEuklid`. Berechne dann

$$l_k := \prod_{j \neq k} u_{jk} m_j \pmod m$$

Diese Zahlen haben die Eigenschaft, dass

$$l_k \equiv 1 \pmod{m_k} \text{ und } l_k \equiv 0 \pmod{m_j},\ j \neq k$$

gilt. Damit erfüllt

$$x = \sum_k x_k\, l_k \pmod m$$

die gestellten Anforderungen.

Nachteil: Kommt ein weiterer Modul m_{n+1} hinzu, so sind fast alle Rechnungen von neuem auszuführen

Vorteil: Man kann die l_k vorab berechnen, wenn man viele Liftungen bzgl. derselben Moduli zu bestimmen hat.

Beispiel: Betrachten wir dieselbe Liftungsaufgabe wie oben. Für die drei Moduln $2, 3, 5$ erhalten wir als Kofaktoren

$$1 = -2 + 3 = -2 \cdot 2 + 5 = 2 \cdot 3 - 5$$

und somit

$$l_1 = 3 \cdot 5,\ l_2 = (-2)(-5) = 10,\ l_3 = (-2 \cdot 2)(2 \cdot 3) = -24.$$

Offensichtlich erfüllen diese drei Zahlen die obige Bedingung $l_k \equiv 1 \pmod{m_k}$ und $l_k \equiv 0 \pmod{m_j}$ für $j \neq k$. Als Lösung des Liftungsproblems $\texttt{CRA}((1,2),(2,3),(2,5))$ ergibt sich dann

$$x = l_1 + 2l_2 + 2l_3 = -13 \equiv 17 \pmod{30}.$$

2) (Newton-Verfahren) Die Grundidee besteht darin, ein Verfahren

$$\texttt{CRA2}((x_1,m_1),(x_2,m_2)) \ \rightarrow\ (x,m)$$

zum Liften für zwei Argumente anzugeben und das allgemeine Liftungsproblem darauf rekursiv zurückzuführen:

$$\texttt{CRA}((x_1,m_1),(x_2,m_2),\ldots,(x_n,m_n)) = \texttt{CRA}(\texttt{CRA2}((x_1,m_1),(x_2,m_2)),(x_3,m_3),\ldots,(x_n,m_n)))$$

Vorbetrachtungen:

$$\begin{aligned} x \equiv x_1 \pmod{m_1} &\ \Rightarrow\ x = x_1 + c \cdot m_1 \\ x \equiv x_2 \pmod{m_2} &\ \Rightarrow\ c \cdot m_1 \equiv x_2 - x_1 \pmod{m_2} \end{aligned}$$

Wir brauchen also die inverse Restklasse $m_1^{-1} \pmod{m_2}$, um x zu berechnen. Diese ergibt sich als Nebenprodukt des Erweiterten Euklidschen Algorithmus (MuPAD):

```
invmod:=proc(x,m) local u;
begin
  x:=x mod m;
  u:=ExtendedEuklid(x,m); //* d.h. 1=u*x+v*m */
  return(u[2]);
end_proc;
```

Allerdings bestimmt `1/a mod m` in MuPAD diese inverse Restklasse bereits, so dass sich `CRA2` und `CRA` wie folgt ergeben:

```
CRA2:=proc(a,b) local c;
begin
  c:=(b[1]-a[1]) * modp(1/a[2],b[2]) mod b[2];
  [a[1]+c*a[2],a[2]*b[2]];
end_proc;

CRA:=proc()
begin
  if args(0)<2 then args()
  elif args()=2 then CRA2(args())
  else CRA(CRA2(args(1..2)),args(3..args(0)))
  end_if;
end_proc;
```

Beispiele:

1) `u:=CRA2([5,13],[2,11])`:

Wegen $1 = 6 \cdot 2 - 11$, also $13^{-1} \equiv 2^{-1} \equiv 6 \pmod{11}$ ergibt sich $c = (2-5) \cdot 6 \equiv 4 \pmod{11}$ und $x \equiv 5 + 4 \cdot 13 = 57 \pmod{143}$.

2) Bestimmen Sie `CRA`((x, x^2) : $x \in \{2, 3, 5, 7, 11, 13\}$).

Antwort mit MuPAD und obigen Funktionen:

```
u:=map([2,3,5,7,11,13],x->[x,x^2]);
v:=CRA(op(u));
```

$$v := [127357230, 901800900]$$

Probe:

```
map(u,x->v[1] mod x[2]);
```

$$[2, 3, 5, 7, 11, 13]$$

Kostenbetrachtungen: Typische Einsatzsituation ist der Fall, dass alle Moduln die Größe eines Computerworts, also die Wortlänge 1 haben und daraus eine ganze Zahl der Wortlänge n zu konstruieren ist.

Verfahren 1: Es sind zu berechnen:

$O(n^2)$ `ExtendedEuklid`, jedes mit konstantem Aufwand
n Produkte l_k, jedes mit Aufwand $O(n^2)$
n Summanden und $n-1$ Additionen, jede mit Aufwand $O(n)$

zusammen also $O(n^3)$, wobei der Aufwand nach dem preprocessing nur $O(n^2)$ ist.

Verfahren 2: Wegen der rekursiven Natur von `CRA` wollen wir bei der Analyse von `CRA2` $l(m_1) = k,\ l(m_2) = 1$ annehmen.

Reduktionen von Zahlen der Länge k modulo m_2 mit Aufwand $O(k)$
`ExtendedEuklid` mit Aufwand $O(1)$
$x = x_1 + c \cdot m_1$ mit Aufwand $O(k)$

zusammen also $C_{CRA2} = O(k)$ und über alle Durchläufe $C_{CRA} = O(n^2)$.

Ein ähnliches Ergebnis kann man auch beim Lagrangeansatz erreichen, wenn man die l_k etwas cleverer berechnet.

3.4 Die Gruppe der primen Restklassen

Da Produkt- und Inversenbildung nicht aus der Menge herausführen, bilden die primen Restklassen $\mathbb{Z}_m^*$ eine Gruppe. Diese Gruppe enthält genau $\phi(m)$ Elemente. Allgemeine Gruppeneigenschaften spezifizieren zu interessanten zahlentheoretischen Sätzen.

So bezeichnet man etwa für ein Element a einer Gruppe G die Mächtigkeit der von a erzeugten Untergruppe $\langle a\rangle \subset G$ als die *Ordnung* $ord(a)$ von a. Im Falle endlicher Gruppen ist diese Ordnung immer endlich und es gilt

$$ord(a) = \min\{n > 0 \ :\ a^n = 1\}.$$

Weiter gilt

$$a^n = 1 \ \Leftrightarrow\ ord(a)\,|\,n$$

Dies ist eine unmittelbare Folgerung aus dem Satz von Lagrange.

Satz 6 (Satz von Lagrange) *Ist H eine Untergruppe von G, so ist $|H|$ ein Teiler von $|G|$.*

Insbesondere ist also die Gruppenordnung $|G|$ durch die Ordnung $ord(a)$ jedes Elements teilbar und es gilt stets $a^{|G|} = 1$. Für die Gruppe der primen Restklassen bedeutet das:

Korollar 2 (Satz von Euler) *Ist $(a,m) = 1$, so gilt $a^{\phi(m)} \equiv 1 \pmod m$.*

Ein Spezialfall dieses Satzes ist der

Korollar 3 (Kleiner Satz von Fermat)
Ist m eine Primzahl und $1 < a < m$, so gilt $a^{m-1} \equiv 1 \pmod m$.

P induziert einen Gruppenisomorphismus

$$P^* : \mathbb{Z}_m^* \longrightarrow \mathbb{Z}_{m_1}^* \times \ldots \times \mathbb{Z}_{m_n}^*$$

Auf der Basis dieses Isomorphismus P^* kann man auch den Satz von Euler verfeinern: Da mit $x^{\phi(m_i)} \equiv 1 \pmod{m_i}$ auch für jedes Vielfache c von m_i die Beziehung $x^c \equiv 1 \pmod{m_i}$ gilt, erhalten wir für $c = \mathrm{lcm}(\phi(m_1), \ldots, \phi(m_n))$, dass $x^c \equiv 1 \pmod{m_i}$ für alle $i = 1, \ldots, n$, also nach dem Chinesischen Restklassensatz sogar $x^c \equiv 1 \pmod m$ gilt.

Satz 7 (Satz von Carmichael) *Ist $m = p_1^{a_1} \ldots p_k^{a_k}$ die Primfaktorzerlegung von m, so gilt*

$$a^{\psi(m)} \equiv 1 \pmod m$$

mit

$$\psi(m) = \mathrm{lcm}(p_1^{a_1-1}(p_1 - 1), \ldots, p_k^{a_k-1}(p_k - 1))$$

Die Zahl $\psi(m)$ bezeichnet man auch als den Carmichael-Exponenten *von m.*

Beispiele:

$$\begin{aligned} 12 = 2^2 \cdot 3 \quad &\Rightarrow \phi(12) = 4,\ \psi(12) = 2 \\ 24 = 2^3 \cdot 3 \quad &\Rightarrow \phi(24) = 8,\ \psi(24) = 4, \\ 561 = 3 \cdot 11 \cdot 17 &\Rightarrow \phi(561) = 2 \cdot 10 \cdot 16 = 320,\ \psi(561) = \mathrm{lcm}(2, 10, 16) = 80 \end{aligned}$$

Dieses Ergebnis ist zugleich ziemlich optimal. Für eine Gruppe G bezeichnet man das Maximum $\max(ord(g)\ :\ g \in G)$ als Exponente $exp(G)$ der Gruppe. Nach dem Satz von Lagrange gilt stets $exp(G)\,|\,|G|$ und Gleichheit tritt genau für zyklische Gruppen ein.

Satz 8 *Für $p > 2$ ist $G = \mathbb{Z}_{p^n}^*$ zyklisch, d.h. $exp(G) = |G|$.*

Für $n = 1, 2$ ist $G = \mathbb{Z}_{2^n}^*$ ebenfalls zyklisch, für $n > 2$ gilt dagegen $exp(G) = \frac{|G|}{2}$.
$\mathbb{Z}_m^*$ ist damit genau für $m = 2,\ 4,\ p^a,\ 2p^a$ zyklisch, wobei p eine ungerade Primzahl ist.

(o. Bew.) Insbesondere gilt $exp(Z_{24}^*) = 2 < \psi(24) = 4$.

Korollar 4 *Für ungerades m ist $\psi(m)$ die Exponente der Gruppe der primen Restklassen $\mathbb{Z}_m^*$.*

3.5 Endliche Körper

Rechnen in endlichen Körpern

$K = Z_p[x]/(f(x))$ ist ein endliche Körper gdw $f(x)$ ist irreduzibel in $Z_p[x]$.
Rechnen in solchen endlichen Körpern: $f = x^d - r(x), deg(r) < d$, beschreibt eine algebraische Ersetzungsregel $x^d \Rightarrow r(x)$. Dann ist

$$K \longrightarrow \mathbb{Z}_p^d \qquad \text{mit } \sum_{i=0}^{d-1} a_i x^i \mapsto (a_{d-1}, \dots, a_0)$$

ein $\mathbb{Z}_p$-Vektorraumisomorphismus und jedes Element aus K kann als d-Vektor mit Einträgen aus $\mathbb{Z}_p$ dargestellt werden. K enthält also genau p^d Elemente. Addition erfolgt komponentenweise, Multiplikation wie bei klassischen Polynomen mit nachfolgender Anwendung der Ersetzungsregel für Potenzen $x^k, k \geq d$.

Beispiel: $p = 2, f = x^3+x+1$. Der Körper ist $K = \mathbb{Z}_2(\alpha)$, wobei α ein algebraisches Element mit dem charakteristischen Polynom $\alpha^3 + \alpha + 1 = 0$ ist, welches der algebraischen Ersetzungsrelation $\alpha^3 \mapsto \alpha + 1$ über $\mathbb{Z}_2$ entspricht.

Die $2^3 = 8$ Elemente dieses Körpers lassen sich als $a_2\,\alpha^2 + a_1\,\alpha + a_0$ mit $a_i \in \mathbb{Z}_2$ oder aber als Bitvektoren (a_2, a_2, a_0) darstellen. Außerdem ist K^* zyklisch, denn α erzeugt diese Gruppe:

$$\begin{aligned}
0 &= (000)\\
1 &= (001) = \alpha^0\\
\alpha &= (010) = \alpha^1\\
\alpha + 1 &= (011) = \alpha^3\\
\alpha^2 &= (100) = \alpha^2\\
\alpha^2 + 1 &= (101) = \alpha^6\\
\alpha^2 + \alpha &= (110) = \alpha^4\\
\alpha^2 + \alpha + 1 &= (111) = \alpha^5
\end{aligned}$$

Für $p = 2$ kann das Produkt durch einfache Bitoperationen berechnet werden.

Der Struktursatz

Satz 9 (Struktursatz über endliche Körper) *Sei K ein endlicher Körper. Dann gilt*

1. *Die Charakteristik $char(K) = p$ ist eine Primzahl und K damit eine endliche Erweiterung des Körpers $\mathbb{Z}_p$.*
2. *Es existiert eine Zahl $d > 0$, so dass K genau $q = p^d$ Elemente hat.*
3. *Die Elemente von K sind genau die Nullstellen des Polynoms $x^q - x$. Dieses Polynom kann über K also in Linearfaktoren $x^q - x = \prod_{a \in K} (x - a)$ zerlegt werden.*
4. *Die Gruppe K^* ist zyklisch. Ein erzeugendes Element dieser Gruppe bezeichnet man auch als* primitive Wurzel *von K.*

   Ein Element $a \in K^*$ ist genau dann eine primitive Wurzel, wenn $a^{\frac{q-1}{d}} \neq 1$ für alle Primteiler $d|(q-1)$ gilt.

Für jedes Paar (p, d) gibt es damit bis auf Isomorphie genau einen Körper mit p^d Elementen. Diesen bezeichnet man als Galois-Körper *$GF(p^d)$.*

5. *Die Unterkörper von* $K = GF(p^d)$ *stehen in 1-1-Korrespondenz zu den Teilern* $e|d$.
   Zu jedem solchen Teiler gibt es genau einen zu $GF(p^e)$ *isomorphen Unterkörper*
   $$K_e = \{a \in K \,:\, a^{p^e} = a\}$$
   und dies sind alle Unterkörper von K.
   Der Verband der Unterkörper (bzgl. Inklusion) ist isomorph zum Teilerverband von d.

Beweis :
4. : Ist $d|(q-1)$, so ist $(x^d-1)|(x^{q-1}-1)$ und hat damit in K genau d Nullstellen. Ist $\psi(d)$ die Zahl der Elemente in K^*, die genau die Ordnung d haben, so gilt $d = \sum_{c|d}\psi(c)$. Vergleich mit der Eulerschen ϕ-Funktion, für die ebenfalls $d = \sum_{c|d}\phi(c)$ gilt, liefert $\psi(d) = \phi(d)$. Insbesondere ist also $\psi(q-1) > 0$.

Bemerkung : Es gibt also genau $\phi(q-1)$ primitive Wurzeln.

5. : Ist $K_e \subset K$ ein zu $GF(p^e)$ isomorpher Unterkörper, so gilt $a^{p^d} = a^{p^e} = a$ für $a \in K_e$ und $e > 0$ ist der kleinste Exponent, für den diese Beziehung für *alle* $a \in K_e$ gilt.

Ist nun $d = e \cdot q + r$ mit $e > r \geq 0$, so gilt $p^d = (p^e)^q \cdot p^r$ und
$$a = a^{p^d} = \left(\left(\dots\left(a^{p^e}\right)^{\cdots}\right)^{p^e}\right)^{p^r} = a^{p^r}$$
für alle $a \in K_e$. Also muss $r = 0$ sein. Umgekehrt prüft man leicht, dass die angegebene Menge K_e ein Teilkörper von K ist. $\square$

Für endliche Körper K mit $char(K) = p$ bezeichnet man die Abbildung
$$F : K \to K \text{ mit } F(a) = a^p$$
als die *Frobeniusabbildung*. Diese ist wegen $(a+b)^p = a^p + b^p$ für $a, b \in K$ ein Ring- und wegen $a^p = a$ für $a \in \mathbb{Z}_p$ sogar ein $\mathbb{Z}_p$-Algebramorphismus.

Kreisteilungspolynome und endliche Körper

Das definierende Polynom $f(x)$ ist ein irreduzibler Teiler von $x^{q-1} - 1$.

In $\mathbb{Z}[x]$ gilt: Zu jedem $r > 0$ hat $x^r - 1$ genau ein irreduzibles Polynom $\Phi_r(x)$ vom Grad $\phi(r)$ als Teiler. Ist $\zeta \in \mathbb{C}$ eine primitive r-te Einheitswurzel, so ergibt sich dieses Polynom als
$$\Phi_r(x) = \prod_{c \in \mathbb{Z}_r^*} (x - \zeta^c).$$
Dieses Polynom bezeichnet man auch als *r-tes Kreisteilungspolynom*. Es kann aus der Formel
$$(x^r - 1) = \prod_{c|r} \Phi_c(x)$$
berechnet werden.

Beispiel: Wegen $(x^{15}-1) = \Phi_{15}(x)\Phi_5(x)\Phi_3(x)\Phi_1(x)$ gilt
$$\Phi_{15}(x) = \frac{(x^{15}-1)(x-1)}{(x^5-1)(x^3-1)} = x^8 - x^7 + x^5 - x^4 + x^3 - x + 1$$
Alle Faktorzerlegungen in $\mathbb{Z}[x]$ induzieren Faktorzerlegungen in $\mathbb{Z}_p[x]$. Allerdings bleiben irreduzible über $\mathbb{Z}$ Polynome nicht unbedingt irreduzibel bei Reduktion $\pmod p$:
$$x^8 - x^7 + x^5 - x^4 + x^3 - x + 1 \equiv (x^4 + x + 1)(x^4 + x^3 + 1) \pmod 2$$
Weiteres Beispiel ($p = 2, r = 7$):
$$\Phi_7(x) = (x^6 + \ldots + 1) \equiv (x^3 + x^2 + 1)(x^3 + x + 1) \pmod 2.$$

Satz 10 *Genauer gilt (mit $(r,p)=1$):*

$$\Phi_r(x) \equiv h_1(x) \cdot \ldots \cdot h_s(x) \pmod{p}$$

mit Polynomen $h_i(x)$ vom Grad $deg\, h_i = d = ord(p \in \mathbb{Z}_r^*)$ und

$$h_i(x) = \prod_{k=0}^{d-1} (x - \zeta^{c_i\, p^k})$$

für geeignete $c_i \in \mathbb{Z}_p^*$ ist die Zerlegung von $\Phi_r \in \mathbb{Z}_p[x]$ in irreduzible Faktoren.

Beweis : Die c_i sind Vertreter von Nebenklassen der von p erzeugten Untergruppe in $\mathbb{Z}_r^*$. $h_i(x)$ sind unter der Frobeniusabbildung invariant, also die Koeffizienten aus $\mathbb{Z}_p$. Andererseits gilt: Hat $f(x) \in \mathbb{Z}_p[x]$ eine Nullstelle $a \in K$, so ist auch $F(a) = a^p$ eine Nullstelle von f. Damit muss ein Faktor $h \in \mathbb{Z}_p[x]$ mit der Nullstelle ζ^c auch alle $\zeta^{c\,p^k}$ als Nullstelle und damit eines der h_i als Faktor haben. $\square$

Beispiel: $K = GF(2^6 = 64)$. Das charakteristische Polynom eines erzeugenden Elements α ist ein über $\mathbb{Z}_2$ irreduzibler Faktor von

$$\Phi_{63}(x) = 1 - x^3 + x^9 - x^{12} + x^{18} - x^{24} + x^{27} - x^{33} + x^{36},$$

also eines der sechs Polynome $f_1, \ldots, f_6$, die MuPAD beim Faktorisieren berechnet:

```
f:=poly(1-x^3+x^9-x^12+x^18-x^24+x^27-x^33+x^36, Dom::IntegerMod(2));
expr(factor(f));
```

$$\left(x + x^6 + 1\right) \left(x^5 + x^6 + 1\right) \left(x + x^2 + x^5 + x^6 + 1\right)$$
$$\left(x + x^3 + x^4 + x^6 + 1\right) \left(x + x^4 + x^5 + x^6 + 1\right) \left(x^2 + x^3 + x^5 + x^6 + 1\right)$$

Die sechs möglichen Körper $Z_2[x]/(f_i(x))$ sind allerdings alle zueinander isomorph. Ist etwa α eine Nullstelle von $x^6 + x + 1$, so ist $\alpha + 1$ eine Nullstelle von $1 + x + x^2 + x^4 + x^6$.

```
f1:=1 + x + x^6;
expr(poly(subs(f1,x=x+1), Dom::IntegerMod(2)));
```

Nehmen wir $a \in K$ mit dem charakteristischen Polynom $a^6 + a + 1$ als Erzeugende des Körpers, so können wir Elemente des Körpers als Polynome aus $\mathbb{Z}_2[a]$ auffassen, die $\pmod{a^6 + a + 1}$ zu reduzieren sind.

```
K:= u -> expr(poly(divide(u,a^6+a+1,Rem),[a],Dom::IntegerMod(2)));
```

$a \in K$ erzeugt auch in diesem Fall K^*:

```
{K(a^i) $ i=0..62}; nops(%);
```

Die beiden Unterkörper sind $GF(2^2)$

```
[0,1,K(a^21), K(a^42)];
```

$$\left[0, 1, a + a^3 + a^4 + a^5 + 1, a + a^3 + a^4 + a^5\right]$$

und $GF(2^3)$

```
s:=[0,1,K(a^(9*i))$i=1..6];
```

In beiden Fällen überzeugt man sich leicht, dass diese Mengen nicht nur multiplikativ, sondern auch additiv abgeschlossen sind. Die zweite kann als Menge von Bitvektoren dargestellt werden:

```
map(s,u->revert([coeff(u,a,i)$i=0..5]));
```

$$[0,0,0,0,0,0] \quad [0,0,0,0,0,1] \quad [0,1,1,0,0,0] \quad [0,0,1,1,1,1]$$
$$[0,0,1,1,1,0] \quad [0,1,0,1,1,0] \quad [0,1,1,0,0,1] \quad [0,1,0,1,1,1]$$

4 Primzahl-Testverfahren

4.1 Primtest durch Probedivision

Ist eine große ganze Zahl gegeben, so ist es in vielen Fällen einfach zu erkennen, dass es sich um eine zusammengesetzte Zahl handelt. So sind z.B. 50 % aller Zahlen gerade. Macht man eine Probedivision durch die 4 Primzahlen < 10, so kann man bereits 77 % aller Zahlen als zusammengesetzt erkennen. Übrig bleibt eine Grauzone von möglichen Kandidaten von Primzahlen, für die ein aufwändigeres Verfahren herangezogen werden muss, um die Primzahleigenschaft auch wirklich zu beweisen.

Ein erstes solches Verfahren ist die **Methode der Probedivision**. Die folgenden Prozeduren sind in MuPAD geschrieben, sehen aber in jedem anderen CAS ähnlich aus.

```
primeTestByTrialDivision:=proc(m:DOM_INT) local z;
begin
  if (m<3) then bool(m=2)
  else
    z:=2;
    while z*z<=m do
      if m mod z = 0 then return(FALSE) end_if;
      z:=z+1;
    end_while;
    TRUE;
  end_if;
end_proc;
```

oder in etwas effizienterer Form (Aufwand auf $\frac{1}{3}$ reduziert)

```
primeTestByTrialDivision1:=proc(m:DOM_INT) local z;
begin
  if (m<2) then FALSE
  elif (m mod 2 = 0) then bool(m=2);
  elif (m mod 3 = 0) then bool(m=3);
  else
    z:=5;
    while z*z<=m do
      if m mod z = 0 then return(FALSE) end_if;
      if m mod (z+2) = 0 then return(FALSE) end_if;
      z:=z+6;
    end_while;
    TRUE;
  end_if;
end_proc;
```

`bool` erzwingt dabei die boolesche Auswertung des zurückzugebenden Ausdrucks, da sonst standardmäßig angenommen wird, dass es sich um eine Gleichung handelt, der man in einem symbolischen Kontext nur dann einen Wahrheitswert zuordnen kann, wenn in ihr keine (ungebundenen) Symbole auftreten.

Der Aufwand für dieses Verfahren ist am größten, wenn m eine Primzahl ist, d.h. wirklich alle Tests bis zum Ende durchgeführt werden müssen. Die Laufzeit ist dabei von der Größenordnung $O(\sqrt{m})$, also exponentiell in der Bitlänge $l(m)$ der zu untersuchenden Zahl.

Dieses Verfahren liefert uns allerdings für eine zusammengesetzte Zahl neben der Antwort auch einen Faktor, so dass eine rekursive Anwendung nicht nur die Primzahleigenschaft prüfen kann, sondern für Faktorisierungen geeignet ist. Experimente mit CAS zeigen dagegen, dass Faktorisieren offensichtlich um Größenordnungen schwieriger ist als der reine Primzahltest (siehe etwa [CA-Praktikum], 2. Tutorium).

Gleichwohl wird in allen CAS der Test mit Probedivision für eine Liste von kleinen Primzahlen als Vortest angewendet, um die aufwändigeren Verfahren nur für solche Zahlen anzuwenden die „nicht offensichtlich“ zusammengesetzt sind. In der folgenden MuPAD-Prozedur ist `smallPrimes` eine Liste aller „kleinen“ Primzahlen:

```
smallPrimesTest:=proc(m:DOM_INT) local i,smallPrimes;
begin
  smallPrimes:=[2, 3, 5, 7, 11, 13, 17, 19, 23, 29];
  if (m<2) then FALSE
  else
    for i in smallPrimes do
      if (m mod i = 0) then return(bool(m=i)) end_if;
     end_for;
    FAIL;
  end_if
end_proc;
```

In dieser Prozedurdefinition wird MuPADs dreiwertige Logik ausgenutzt, die neben den (sicheren) Wahrheitswerten `true` und `false` auch noch die Möglichkeit `FAIL` erlaubt, weil oftmals symbolische Ausdrücke auftreten, die erst nach einer Belegung der Variablen einen definierten Wahrheitswert annehmen.

4.2 Das Sieb des Eratostenes

Damit findet man auf einen Ritt *alle* Primzahlen $\leq N$.

Die folgende MuPAD-Implementierung legt dazu in der globalen Variablen B ein entsprechendes Feld an.

```
Eratostenes:=proc(N) local p,k;
begin
  for p from 2 to N do B[p]:=TRUE end_for;
  p:=2; while (p^2 <= N) do
    if B[p] then // p Primzahl, filtere aus ab B[p^2]
      k:=p; while (p*k <= N) do B[p*k]:=FALSE; k:=k+1 end_while;
    end_if;
    p:=p+1;
  end_while:
end_proc;
```

Kosten:

$$\sum_{p^2 \leq N} \frac{N}{p} = N \cdot \ln(\ln(N)) + O(N)$$

In dem Intervall sind $\pi(N) \sim \frac{N}{\ln(N)}$ Primzahlen, so dass die *Kosten pro Primzahl* nur $O(\ln(N)) = O(l(N))$ sind. Aber das Verfahren ist natürlich sehr speicherintensiv.

Will man sich auf ein Segment der Länge M beschränken und mit einer (bekannten) Liste von Primzahlen bis $\sqrt{N}$ sieben, so sind die Kosten

$$M \cdot \ln(\ln(N)) + \pi(\sqrt{N}) + O(M).$$

Für $M \ll N$ dominiert die Größe des zweiten Terms die Wachstumsordnung. Ein ungelöstes Problem ist z.B. die Frage, ob es ein schnelles Verfahren gibt, das alle Primzahlen im Intervall $N \leq x \leq N + N^{1/4}$ findet.

Verfahren kann ebenso wie die Probedivision aufgebohrt werden zu einem Verfahren, das zu jeder Zahl im betrachteten Intervall einen Primteiler liefert:

```
... while (p*k <= N) do B[p*k]:=p; k:=k+1 end_while;
```

4.3 Der Fermat-Test

Ein weiteres Verfahren, mit dem zusammengesetzte Zahlen sicher erkannt werden können, das nicht auf Faktorzerlegungen beruht, ist der **Fermat-Test**. Dieser Test beruht auf der folgenden Umkehrung des Kleinen Satzes von Fermat:

> *Gilt für eine Zahl a mit $1 < a < m$*
>
> $$a^{m-1} \not\equiv 1 \pmod{m},$$
>
> *so kann m keine Primzahl sein.*

Eine Realisierung in MuPAD hätte etwa folgende Gestalt:

```
FermatTest:=proc(m:DOM_INT,a:DOM_INT) local D;
begin
  D:=Dom::IntegerMod(m);
  iszero(D(a)^(m-1) - 1)
end_proc;
```

In Maple hätten wir `evalb(a&^(m-1)-1 mod m = 0)` mit dem *inerten* Potenzoperator `&^` schreiben müssen, da sonst zuerst a^{n-1} als ganze Zahl berechnet und dann erst $\pmod{n}$ reduziert würde. In MuPAD wird das modulare Potenzieren automatisch eingesetzt, weil `D(a)` als Element eines entsprechenden Domains spezifiziert ist. Beide CAS binden dynamisch die entsprechende Methode während der Ausführung der Rechnungen. In Mathematica muss das modulare Potenzieren als `PowerMod[a,m-1,m]` explizit im Code spezifiziert werden.

Mit binärem Potenzieren sind die Kosten dieses Verfahrens von der Größenordnung $O(l(m)^3)$ ($\phi(m) \sim m$, $\log(m)$ Multiplikationen von Zahlen der Länge $l(m)$ mit anschließender Restreduktion), also polynomial in der Bitlänge der zu untersuchenden Zahl.

Der Fermat-Test ist allerdings nur ein notwendiges Kriterium. Genauer gesagt können wir aus $a^{m-1} \not\equiv 1 \pmod{m}$ mit Sicherheit schließen, dass m eine zusammengesetzte Zahl ist. Ansonsten können wir den Test mit einer anderen Basis a wiederholen, weil vielleicht zufällig $ord(a) \,|\, m - 1$ galt. Wir bezeichnen deshalb eine Zahl m, die den Fermat-Test mit der Basis a besteht, als *Pseudoprimzahl zur Basis a*.

Auf dieser Grundlage können wir einen **Las-Vegas-Test** versuchen:

> *Mache den Fermat-Test für c verschiedene (zufällig gewählte) Basen $a_1, \ldots, a_c$.*
>
> *Ist einmal $a_i^{m-1} \not\equiv 1 \pmod{m}$, so ist m garantiert eine zusammengesetzte Zahl. Die Basis a bezeichnet man in diesem Fall auch als* **Fermat-Zeugen** *(witness) dafür, dass m zusammengesetzt ist.*
>
> *Ist stets $a_i^{m-1} \equiv 1 \pmod{m}$, so ist m wahrscheinlich (hoffentlich mit großer Wahrscheinlichkeit) eine Primzahl.*

Eine MuPAD-Implementierung hätte etwa folgende Gestalt:

```
LasVegas:=proc(m:DOM_INT,c:DOM_INT) local a,i,r;
begin
  r:=random(m); // r ist Zufallszahlengenerator
```

```
  for i from 1 to c do
    a:=r();
    if igcd(a,m)<>1 then return(FALSE) end_if;
    if not FermatTest(m,a) then return(FALSE) end_if;
  end_for;
  TRUE;
end_proc;
```

`r:=random(m)` gibt dabei keine Zufallszahl, sondern eine (nullstellige) Funktionsdefinition zurück (ähnlich in MAPLE oder Java), `r()` ruft diese auf. Ist a zufällig keine prime Restklasse, dann ist m zusammengesetzt und der berechnete *gcd* sogar ein echter Teiler von m. Dieser (sehr selten auftretende) Fall wird gesondert abgefangen.

Was können wir über die Wahrscheinlichkeit im unsicheren Zweig dieses Tests aussagen?

Satz 11 *Die Menge*

$$P_m := \{a \in \mathbb{Z}_m^* \;:\; a^{m-1} \equiv 1 \pmod{m}\}$$

der Restklassen $\pmod m$*, für die der Fermat-Test kein Ergebnis liefert, ist eine Untergruppe der Gruppe der primen Restklassen* $\mathbb{Z}_m^*$*.*

Beweis mit Untergruppenkriterium.

Nach dem Satz von Lagrange ist somit $|P_m|$ ein Teiler von $\phi(m) = |\mathbb{Z}_m^*|$. Ist m also zusammengesetzt und $P_m \neq \mathbb{Z}_m^*$, dann erkennt der Fermat-Test für eine zufällig gewählte Basis in wenigstens der Hälfte der Fälle, dass m zusammengesetzt ist.

In diesem Fall ist die Wahrscheinlichkeit, dass im unsicheren Zweig des Las-Vegas-Tests m keine Primzahl ist, höchstens 2^{-c}, also bei genügend vielen Tests verschwindend klein.

4.4 Carmichael-Zahlen

Ist m eine zusammengesetzte Zahl und $P_m = \mathbb{Z}_m^*$, so kann der Fermat-Test m prinzipiell nicht von Primzahlen unterscheiden. Gibt es solche Zahlen ?

Antwort: Ja, z.B. die Zahl $561 = 3 \cdot 11 \cdot 17$. Dann ist $\psi(m) = lcm(2, 10, 16) = 80$ und somit für $a \in \mathbb{Z}_{561}^*$ stets $a^{560} = 1$

Zusammengesetzte Zahlen m, für welche $a^{m-1} \equiv 1 \bmod m$ für alle $a \in \mathbb{Z}_m^*$ gilt, nennt man *Carmichael-Zahlen.*

Satz 12 *Die ungerade zusammengesetzte Zahl* m *ist genau dann Carmichael-Zahl, wenn* $\psi(m)$ *ein Teiler von* $m - 1$ *ist. Solche Zahlen kann der Fermat-Test nicht von Primzahlen unterscheiden.*

Weitere Carmichael-Zahlen sind z.B. $1105 = 5 \cdot 13 \cdot 17$ und $1729 = 7 \cdot 13 \cdot 19$. In den Übungsaufgaben finden Sie ein Verfahren, mit dem weitere Carmichaelzahlen konstruiert werden können.

Carmichaelzahlen sind unter den Primzahlen recht selten. So gibt es unter den Zahlen $< 10^{15}$ nur etwa 10^5 solcher Zahlen. Andererseits gibt es unendlich viele solche Zahlen und Alford, Granville und Pomerance (1994) haben sogar gezeigt, dass für die Anzahl der Carmichaelzahlen kleiner x $C(x) \gtrsim x^{2/7}$ gilt, d.h. ihre Anzahl wächst exponentiell mit der Bitlänge von x.

4.5 Der Rabin-Miller- oder strenge Pseudoprimzahl-Test

Ein Primzahltest ohne solche systematischen Ausnahmen beruht auf der folgenden Verfeinerung des Fermat-Tests: Für eine Primzahl m und eine Basis $1 < a < m$ muss $a^{m-1} \equiv 1 \pmod{m}$ gelten. Ist $m - 1 = 2^t \cdot q$ die Zerlegung des Exponenten in eine Zweierpotenz und einen ungeraden Anteil q, so ergibt die Restklasse $b := a^q \pmod{m}$ nach t-maligem Quadrieren den Rest 1. Bezeichnet u

das Element in der Folge $\{b, b^2, b^4, b^8, \dots, b^{2^t}\}$, wo erstmals $u^2 \equiv 1 \pmod m$ gilt, so muss, wenn m eine Primzahl ist, $u \equiv -1$ $(mod\ m)$ gelten. Ist dagegen m keine Primzahl, so hat die Kongruenz $u^2 \equiv 1 \pmod m$ noch andere Lösungen.

In der Tat, ist $m = p \cdot q$ für zwei teilerfremde Faktoren p, q, so gibt es nach dem chinesischen Restklassensatz eine Restklasse a_1 $(mod\ m)$ mit $a_1 \equiv 1 \pmod p$ und $a_1 \equiv -1 \pmod q$ und eine weitere Restklasse $a_2 \pmod m$ mit $a_2 \equiv -1 \pmod p$ und $a_2 \equiv 1 \pmod q$. Für beide gilt $a_i^2 \equiv 1 \pmod p$ und $a_i^2 \equiv 1 \pmod q$, also auch $a_i^2 \equiv 1 \pmod m$, aber $a_i \not\equiv \pm 1 \pmod m$.

Wählt man a zufällig aus, so ist die Wahrscheinlichkeit, dass u bei zusammengesetztem m auf eine solche Restklasse trifft, d.h. $u \not\equiv -1 \pmod m$, aber $u^2 \equiv 1 \pmod m$ gilt, groß. Verhält sich m unter diesem verfeinerten Test bzgl. einer Basis a wie eine Primzahl, so bezeichnet man m auch als *strenge Pseudoprimzahl zur Basis a*.

Beispiel: Die Carmichaelzahl $m = 561$ passiert den Fermat-Test zur Basis $a = 13$. Der verfeinerte Test erkennt sie bzgl. derselben Basis als zusammengesetze Zahl: $m - 1 = 2^4 \cdot 35$, also $q = 35$

```
b:=Dom::IntegerMod(561)(13)^35;
```

Es gilt $b^2 \equiv 67 \mod 561, b^4 \equiv 1 \mod 561$.

Dieser Test wurde von Artjuhov (1966/67) vorgeschlagen und eine Dekade später von J. Selfridge wiederentdeckt und popularisiert. Eine genauere Analyse (Monier, Rabin 1980) zeigt, dass diese Wahrscheinlichkeit sogar wenigstens $\frac{3}{4}$ beträgt, d.h. ein Las-Vegas-Test auf dieser Basis mit c Durchläufen nur mit der Wahrscheinlichkeit 4^{-c} fehlerhaft antwortet. Diese Idee realisiert der folgende **Rabin-Miller-Test**

```
primeTestRabinMiller:=proc(m:DOM_INT,c:DOM_INT) local a,q,b,t,i,j,r,D;
begin
  D:=Dom::IntegerMod(m);
  q:=m-1; t:=0; r:=random(m);
  while q mod 2 = 0 do q:=q/2; t:=t+1 end_while;
        /* nun ist  m-1 = 2^t * q */
  for i from 1 to c do
    a:=r(); b:=D(a)^q;
    if iszero(b-1) or iszero(b+1) then next end_if;
    /* keine Information, wenn b = 1 oder b = -1 (mod m) */
    for j from 1 to (t-1) do
      /* nun ist  b <> 1 oder -1 (m) */
      b:=b*b;
        if iszero(b-1) then return(FALSE) end_if;
        if iszero(b+1) then break end_if;   /* keine Information */
    end_for;
    if iszero(b+1) then next;         /* keine Information */
    else return(FALSE) end_if;
  end_for;
  TRUE;
end_proc;
```

Die Hauptschleife (Iterationsvariable i) führt den eigentlichen Test für c verschiedene zufällig gewählte Basen a aus. In jedem dieser Tests wird zunächst $b \equiv a^q \pmod m$ berechnet. Ist bereits $b \equiv \pm 1 \pmod m$, so kann für diese Basis keine Aussage getroffen werden. Ansonsten quadrieren wir b:

> Erhalten wir den Rest 1, so war $b \not\equiv \pm 1 \pmod m$, aber $b^2 \equiv 1 \pmod m$, also ist m nicht prim.
>
> Erhalten wir den Rest -1, so wird im nächsten Schritt $b^2 \equiv 1$ $(mod\ m)$, woraus wir

jedoch kein Kapital schlagen können. Aus der gewählten Basis a kann keine Aussage getroffen werden.

Anderenfalls quadrieren wir noch einmal.

Das Quadrieren ist nach $(t-1)$ Schritten abzubrechen, denn $b^{2^t} \equiv a^{m-1} \pmod{m}$. Da nach Verlassen der inneren Schleife stets $b \not\equiv 1 \pmod{m}$ gilt, kommen folgende Fälle in Frage:

$b^2 \not\equiv 1 \pmod{m}$, also m nicht prim nach dem Fermat-Test.

$b \not\equiv -1 \pmod{m}$, aber $b^2 \equiv 1 \pmod{m}$, also ist m nicht prim.

$b \equiv -1 \pmod{m}$. In diesem Fall kann aus der gewählten Basis a keine Aussage getroffen werden.

Wird $FALSE$ zurückgegeben, so bezeichnet man die zugehörige Basis a als **Rabin-Miller-Zeugen** dafür, dass m zusammengesetzt ist.

Kleinste Rabin-Miller-Zeugen und deterministische Primzahltests mit polynomialer Laufzeit

Ist m eine ungerade zusammengesetzte Zahl, so sind wenigstens $\frac{3}{4}$ der $a \in \mathbb{Z}_m^*$ Rabin-Miller-Zeugen für m und für viele m wird bereits $a = 2$ erfolgreich sein. Dazu mehr in zwei Übungsaufgaben.

Bezeichnen wir mit $W(m)$ den kleinsten Rabin-Miller-Zeugen für m.

Satz 13 *Es gibt unendlich viele ungerade zusammengesetzte Zahlen m mit $W(m) \geq 3$.*

Beweis in einer Übungsaufgabe.

Gäbe es eine Schranke $W(m) < B$ mit einer Konstanten B, so ließe sich der Rabin-Miller-Test zu einem deterministischen Primzahltest mit polynomialer Laufzeit erweitern; man müsste den Test nur mit allen $a, 1 < a < B$ ausführen. [Alford, Granville, Pomerance 1994] beweisen:

Satz 14 *Es gibt keine Konstante $B = O(1)$ mit $W(m) < B$.*

Genauer: Es gibt eine Serie von zusammengesetzten Zahlen m, für die

$$W(m) > (\ln m)^{1/(3 \ln \ln m)}$$

gilt.

Für einen deterministischen Primzahltest mit polynomialer Laufzeit würde allerdings bereits die Abschätzung $W(m) = O(\ln(m)^k)$ ausreichen. Eine solche Abschätzung bewies [Bach 1985] unter der Voraussetzung der Gültigkeit einer der großen zahlentheoretischen Vermutungen.

Satz 15 *Gilt die Erweiterte Riemannsche Vermutung, so kann man $W(m) < 2\ln(m)^2$ für alle ungeraden zusammengesetzten Zahlen m beweisen.*

4.6 Quadratische Reste und der Solovay-Strassen-Test

Eine andere Verfeinerung des Fermat-Tests beruht auf der Verwendung von quadratischen Resten. Im folgenden sei m stets eine ungerade Zahl.

Definition: $a \in \mathbb{Z}_m^*$ heißt *quadratischer Rest*, wenn es eine Restklasse $x \in \mathbb{Z}_m^*$ mit $x^2 = a$ gibt. Anderenfalls heißt a *quadratischer Nichtrest*.

$$Q := \{a \in \mathbb{Z}_m^* \ : \ \exists x \in \mathbb{Z}_m^* \ (a = x^2)\} \qquad NQ := \mathbb{Z}_m^* \setminus Q$$

Lemma 1 *Es gilt:*
1) $a, b \in Q \Rightarrow a \cdot b \in Q$.
2) $a \in Q, b \in NQ \Rightarrow a \cdot b \in NQ$.
3) $|Q| \leq \frac{\phi(m)}{2},\ |NQ| \geq \frac{\phi(m)}{2}$.

Beweis : 1) Ist $a_1 = x_1^2, a_2 = x_2^2$, so ist $a_1 a_2 = (x_1 x_2)^2$.
2) Ist $a_1 = x_1^2$ und wäre $a_1 a_2 = x^2$, so wäre $a_2 = (x\,x_1^{-1})^2$.
3) $q\ :\ \mathbb{Z}_m^* \to\ \mathbb{Z}_m^*$ via $x \mapsto x^2$ ist (mindestens) 2-1-Abbildung. $\Box$
Beispiele: $m = 7, m = 11, m = 8$.
Definition: Für eine Primzahl m und $a \in \mathbb{Z}_m^*$ definieren wir das *Legendre-Symbol*

$$\left(\frac{a}{m}\right) = \begin{cases} +1, \text{ wenn } a \text{ quadratischer Rest ist} \\ -1 \text{ sonst} \end{cases}$$

Satz 16 *Ist $m > 2$ eine Primzahl, so gilt*
1. Es gibt genau $\frac{m-1}{2}$ quadratische Reste und $\frac{m-1}{2}$ quadratische Nichtreste.
2. $\kappa :\ \mathbb{Z}_m^* \to \{+1, -1\}$ via $a \mapsto \left(\frac{a}{m}\right)$ ist ein Gruppenhomomorphismus.
3.

$$a^{\frac{m-1}{2}} \equiv \left(\frac{a}{m}\right) \pmod m.$$

Beweis : 1) Ist m eine Primzahl, so folgt aus $x^2 \equiv y^2 \pmod m$ bzw. $m \,|\, x^2 - y^2 = (x+y)(x-y)$, dass $m \,|\, x+y$ oder $m \,|\, x-y$ bzw. $x \equiv \pm y (mod\ m)$ gilt. q ist also *exakt* eine 2-1-Abbildung.
2) Abzählen zeigt, dass dann auch das Produkt zweier Nichtreste ein quadratischer Rest sein muss.
3) Sei $x = a^{\frac{m-1}{2}} \pmod m$. Wegen $x^2 \equiv 1$ gilt wie in 1) $x \equiv \pm 1$. Da $\mathbb{Z}_m^*$ zyklisch ist, gibt es eine Restklasse a_0 der Ordnung $m-1$, für die also $a_0^{\frac{m-1}{2}} \not\equiv 1$ gilt. Dann gilt aber $(a \cdot a_0)^{\frac{m-1}{2}} \equiv -1 \pmod m$ für alle quadratischen Reste $a \in \mathbb{Z}_m^*$. Da Q und $Q_1 := \{a \cdot a_0\ :\ a \in Q\}$ je $\frac{m-1}{2}$ Elemente enthalten, ist gerade $Q_1 = NQ$. $\Box$

Die Definition des Legendre-Symbols kann man auf beliebige ungerade Zahlen $m = p_1^{a_1} \ldots p_k^{a_k}$ erweitern, indem man

$$\left(\frac{a}{m}\right) := \prod_i \left(\frac{a}{p_i}\right)^{a_i}$$

setzt. Diese Erweiterung bezeichnet man als *Jacobi-Symbol.*

$\kappa :\ \mathbb{Z}_m^* \to \{+1, -1\}$ ist offensichtlich auch in diesem Fall ein Gruppenhomomorphismus, allerdings kann man am Vorzeichen nicht mehr ablesen, ob es sich um einen quadratischen Rest handelt. So ist z.B.

$$\left(\frac{2}{15}\right) = \left(\frac{2}{3}\right)\left(\frac{2}{5}\right) = (-1)(-1) = +1,$$

aber $2 \pmod{15}$ kein quadratischer Rest, weil ja bereits $2\ (mod\ 3)$ kein quadratischer Rest ist.

Ist also $m > 2$ eine Primzahl, so gilt stets

$$a^{\frac{m-1}{2}} \cdot \left(\frac{a}{m}\right) \equiv 1 \pmod m.$$

Ist andererseits die ungerade Zahl m keine Primzahl, so ist in der Regel $a^{m-1} \not\equiv 1 \pmod m$ und so erst recht $a^{\frac{m-1}{2}} \not\equiv \pm 1 \pmod m$. Es stellt sich heraus, dass sich selbst für Carmichaelzahlen m stets eine prime Restklasse a finden lässt, für die $a^{\frac{m-1}{2}} \cdot \left(\frac{a}{m}\right) \not\equiv 1\ (mod\ m)$ gilt. (o.Bew.)

Da $f :\ a \mapsto a^{\frac{m-1}{2}} \cdot \left(\frac{a}{m}\right)$ ein Gruppenhomomorphismus ist, gilt wieder $|ker\ f| \leq \frac{\phi(m)}{2}$, d.h. aus der Existenz einer Restklasse a mit $f(a) \neq 1$ folgt, dass wenigstens die Hälfte der Restklassen diese Eigenschaft hat.

Zur Berechnung des Jacobi-Symbols: Hier spielt das von Gauß entdeckte quadratische Reziprozitätsgesetz (3) unten) eine zentrale Rolle.

1) $\left(\frac{a\,b}{m}\right) = \left(\frac{a}{m}\right)\left(\frac{b}{m}\right)$ für $a, b \in \mathbb{Z}_m^*$.

2) $\left(\frac{1}{m}\right) = 1 \quad \left(\frac{2}{m}\right) = (-1)^{\frac{m^2-1}{8}}$.

3) $\left(\frac{p}{q}\right) = (-1)^{\frac{p-1}{2}\frac{q-1}{2}}\left(\frac{q}{p}\right)$ für ungerade teilerfremde Zahlen p, q.

Beispiel: $12^2 = 144 \equiv 43\ (101)$ ist quadratischer Rest. Wir erhalten nacheinander

$$\left(\frac{43}{101}\right) = + \left(\frac{101}{43}\right) = \left(\frac{15}{43}\right) = - \left(\frac{43}{15}\right) = - \left(\frac{13}{15}\right) = - \left(\frac{15}{13}\right) = - \left(\frac{2}{13}\right) = -(-1)^{\frac{13^2-1}{8}} = +1$$

Der Aufwand für die Berechnung des Werts eines Jacobisymbols für $a \in \mathbb{Z}_m^*$ ist wie der Euklidsche Algorithmus von der Größenordnung $O(l(m)^2)$, also polynomial in der Bitlänge der zu untersuchenden Zahl m. MuPAD stellt eine solche Funktion in seinem Paket `numtheory` zur Verfügung.

Zusammenfassend erhalten wir damit den folgenden **Solovay-Strassen-Test**:

```
primeTestSolovayStrassen:=proc(m:DOM_INT,c:DOM_INT)
  local i,a,r,D;
begin
  D:=Dom::IntegerMod(m);
  r:=random(1..m-1);
  for i from 1 to c do
    a:=r();
    if igcd(a,m) <> 1 then return(FALSE) end_if;
    if not iszero(D(a)^((m-1)/2)*numlib::jacobi(a,m) - 1)
      then return(FALSE)
    end_if;
  end_for;
  TRUE;
end_proc;
```

Im Gegensatz zum Fermat-Test ist dieser Algorithmus ein echter Las-Vegas-Algorithmus, d.h. im Ergebnis des Tests von c zufällig gewählten Restklassen a_i können wir hier folgende Aussagen treffen:

> Gibt es ein a, für welches $a^{\frac{m-1}{2}} \cdot \left(\frac{a}{m}\right) \neq 1$ gilt, so ist m garantiert zusammengesetzt.
>
> Gilt $a^{\frac{m-1}{2}} \cdot \left(\frac{a}{m}\right) = 1$ für alle a_i, so ist die Wahrscheinlichkeit, dass m zusammengesetzt ist, kleiner als 2^{-c}.

4.7 Primzahl-Zertifikate

Allen bisherigen Tests haftet der Makel an, dass Primzahlen zwar mit hoher Wahrscheinlichkeit korrekt erkannt werden, aber nicht mit Sicherheit bekannt ist, ob es sich wirklich um Primzahlen handelt. Die vorgestellten Algorithmen sind für praktische Belange, d.h. in Bereichen, in denen sie noch mit vertretbarem Zeitaufwand angewendet werden können, ausreichend und wurden auch in der Form in den verschiedenen CAS implementiert.

Aus dem Axiom-Handbuch:

> `prime?(n)` returns true if n is prime and false if not. The algorithm used is Rabin's probabilistic primality test. If `prime? n` returns false, n is proven composite. If `prime? n` returns true, `prime?` may be in error however, the probability of error is very low and

> is zero below $25 \cdot 10^9$ (due to a result of Pomerance et al), below 10^{12} and 10^{13} due to results of Pinch, and below 341550071728321 due to a result of Jaeschke. Specifically, this implementation does at least 10 pseudo prime tests and so the probability of error is $< 4^{-10}$...

Aus dem Maple-Handbuch:

- The function `isprime` is a probabilistic primality testing routine.
- It returns false if n is shown to be composite within one strong pseudo-primality test and one Lucas test and returns true otherwise. If `isprime` returns true, n is „very probably" prime – see [4], vol. 2, section 4.5.4, Algorithm P for a reference and [7]. No counter example is known and it has been conjectured that such a counter example must be hundreds of digits long.

Aus dem MuPAD-Handbuch:

- `isprime` ist ein schneller stochastischer Primzahltest. Die Funktion gibt `TRUE` zurück, wenn die ganze Zahl n eine Primzahl oder eine starke Pseudo-Primzahl für zehn zufällig gewählte Basen ist, sonst wird `FALSE` zurückgegeben.
- Wenn n positiv ist und `isprime` `FALSE` zurückgibt, dann ist n mit Sicherheit keine Primzahl. Wenn n positiv ist und `isprime` `TRUE` zurückgibt, dann ist n mit großer Wahrscheinlichkeit eine Primzahl.
- Die Funktion `numlib::proveprime` stellt einen Primzahltest zur Verfügung, der stets eine korrekte Antwort liefert, aber im Allgemeinen viel langsamer ist als `isprime`.

Aus dem Mathematica-Handbuch:

- `PrimeQ` first tests for divisibility using small primes, then uses the Miller?Rabin strong pseudoprime test base 2 and base 3, and then uses a Lucas test.
- As of 1997, this procedure is known to be correct only for $n < 10^{16}$, and it is conceivable that for larger n it could claim a composite number to be prime.
- The package `NumberTheory‘PrimeQ‘` contains a much slower algorithm which has been proved correct for all n. It can return an explicit certificate of primality.

Möchte man für gewisse Anwendungen sichergehen, dass es sich bei der untersuchten Zahl garantiert um eine Primzahl handelt, können einige CAS ein *Zertifikat* für die Primzahleigenschaft erstellen. Ein solches Zertifikat kann etwa darin bestehen, zu der Primzahl m ein Erzeugendes a der zyklischen Gruppe $\mathbb{Z}_m^*$ anzugeben zusammen mit einem Beweis, dass dieses Element die behauptete Eigenschaft besitzt.

Satz 17 *Die ungerade Zahl m ist genau dann eine Primzahl, wenn $\mathbb{Z}_m^*$ eine zyklische Gruppe der Ordnung $m-1$ ist, d.h. wenn es ein Element $a \in \mathbb{Z}_m^*$ gibt, so dass $ord(a) = m-1$ gilt.*

Satz 18 (Satz von Lucas-Lehmer, 1876) *Die ungerade Zahl m ist genau dann eine Primzahl, wenn es ein Element $a \in \mathbb{Z}_m^*$ gibt mit*

$$\begin{aligned} a^{m-1} &\equiv 1 \pmod{m} \quad \textit{und} \\ a^{(m-1)/p} &\not\equiv 1 \pmod{m} \quad \textit{für alle Primteiler } p \textit{ der Zahl } m-1. \end{aligned}$$

Die letzte Bedingung sichert, dass $ord(a)$ durch $m-1$, aber keinen Teiler von $m-1$ teilbar ist.

Betrachten wir als Beispiel die Primzahl $m = 55499821019$. Für einen *Beweis* der Primzahleigenschaft prüfen wir die Voraussetzungen des Satzes für $a = 2$. Mit MuPAD erhalten wir

```
m:=55499821019;
Z_m:=Dom::IntegerMod(m);
u:=numlib::primedivisors(m-1);
```

$$\{2, 17, 1447, 1128091\}$$

```
map(u, p->expr(Z_m(2)^((m-1)/p)));
```

$$[55499821018, 44871471456, 31660914932, 44713683701]$$

```
Z_m(2)^(m-1);
```

$$1\ mod\ 55499821019$$

Die Voraussetzungen des Satzes sind also erfüllt, so dass 2 die Gruppe $\mathbb{Z}_m^*$ erzeugt.

Oftmals ist allerdings für eine konkrete Restklasse a die Beziehung $a^{\frac{m-1}{p}} \not\equiv 1 \pmod{m}$ nur für einige der Primfaktoren von $m-1$ erfüllt und es ist schwierig, eine Restklasse zu finden, die für *alle* Primteiler passt.

Beispiel:

```
m:=nextprime(2*10^10); /* = 20000000089 */
Z_m:=Dom::IntegerMod(m);
u:=numlib::primedivisors(m-1);
```

$$[2, 3, 67, 1381979]$$

```
for a in [2,3,5,7,11,13,17,23,29] do
print(map(u, p->expr(Z_m(a)^((m-1)/p))))
end_for;
```

$$[1, 1, 3108157330, -2457953819]$$
$$[1, 1, -4865156001, -7521929036]$$
$$[1, 5697767833, 6300564838, -3927279952]$$
$$[-1, 1, 1, -855778974]$$
$$[1, 1, 8042780317, -7269258111]$$
$$[1, -5697767834, 8735956541, -5244188757]$$
$$[1, 1, -2293750453, 9516320400]$$
$$[1, 5697767833, -8450150690, -4573270973]$$
$$[-1, -5697767834, 8111724708, -1296001895]$$

Erst $a = 29$ hat die geforderte Eigenschaft, dass $a^{\frac{m-1}{p}} \neq 1$ für *alle* Primteiler p von m gilt. Es stellt sich – mit Blick auf den Chinesischen Restklassensatz nicht verwunderlich – heraus, dass es genügt, für jeden Primteiler *seine* Restklasse a zu finden, womit die Rechnungen in diesem Beispiel bereits für $a = 7$ beendet werden können.

Satz 19 *Seien $\{p_1, \ldots, p_k\}$ die (verschiedenen) Primfaktoren von $m-1$. Dann gilt*

$$\exists\, a \in \mathbb{Z}_m^*\ \forall i\ a^{\frac{m-1}{p_i}} \not\equiv 1 \pmod{m}$$

genau dann, wenn

$$\forall i\ \exists\, a_i \in \mathbb{Z}_m^*\ a_i^{\frac{m-1}{p_i}} \not\equiv 1 \pmod{m},$$

d.h. es gibt eine gemeinsame Basis für alle Primteiler von $m-1$, wenn es für jeden Primteiler einzeln eine passende Basis gibt.

Beweis als Übungsaufgabe.

Zur Bestimmung eines Primzahlzertifikats für die primzahlverdächtige Zahl m reicht es also aus, für jeden Teiler p der Zahl $m-1$ in einer Liste von kleinen Zahlen eine solche Zahl a_p zu finden, dass $a_p^{\frac{m-1}{p}} \not\equiv 1 \pmod{m}$ gilt:

```
certifyPrime:=proc(m) local u,v,p,a,D;
begin
  if not isprime(m) then error(expr2text(m)." ist nicht prim") end_if;
  D:=Dom::IntegerMod(m);
  u:=null(); v:=numlib::primedivisors(m-1);
  for p in v do
    for a in [2,3,5,7,11,13,17,19,23] do
       if not iszero(D(a)^((m-1)/p)-1) then
         u:=u,[p,a]; break;
       end_if;
    end_for;
  end_for;
  if nops([u])<>nops(v) then
    error("Kein Zertifikat für ".expr2text(m)." gefunden");
  end_if;
  [u];
end_proc;
```

Auf obiges Beispiel angewendet erhalten wir damit folgendes Zertifikat:

```
certifyPrime(m);
```

$$[[2, 7], [3, 5], [67, 2], [1381979, 2]]$$

Der erste Eintrag gibt dabei jeweils den Primfaktor von $m-1$, der zweite die für diesen Primfaktor p geeignete Basis a_p an. Da bei der Faktorisierung von $m-1$ auch Pseudoprimzahltests verwendet werden, müssen auch die Faktoren von $m-1$ zertifiziert werden. Dies ist etwa mit folgender rekursiven Prozedur möglich:

```
certifyPrime2:=proc(m) local u,v;
begin
  u:=certifyPrime(m);
  print(Unquoted, "Zertifikat für ".expr2text(m)." :");
  print(Unquoted, expr2text(u));
  v:=select(map(u,x->x[1]),x->x>1000);
  map(v,x->certifyPrime2(x));
  null();
end;

certifyPrime2(m);
```

```
Zertifikat für 20000000089 :

[[2, 7], [3, 5], [67, 2], [1381979, 2]]

Zertifikat für 1381979 :

[[2, 2], [13, 2], [23, 3], [2311, 2]]
```

```
Zertifikat für 2311 :

[[2, 3], [3, 2], [5, 2], [7, 2], [11, 2]]
```

Dieses Kriterium geht also davon aus, dass eine Faktorzerlegung der Zahl $m-1$ berechnet werden kann. Für Zahlen in der Nachbarschaft einer Primzahl m ist das erstaunlicherweise oft möglich.

Besonders einfach ist ein solcher Nachweis für Primzahlen spezieller Form. Für die *Fermatzahlen* $F_k := 2^{2^k}+1$ etwa ist F_k-1 nur durch 2 teilbar. Sie sind deshalb interessant, weil sie die einzigen Zahlen der Form 2^a+1 sind, die prim sein können. Fermat behauptete in einem Brief an Mersenne, dass alle Zahlen F_k prim sind. Er konnte dies für die ersten 5 Fermatzahlen $3, 5, 17, 257$ und 65537 nachweisen, vermerkte allerdings, dass er die Frage für die nächste Fermatzahl $F_5 = 4294967297$ nicht entscheiden könne. Dies wäre allerdings mit dem Fermat-Test für die Basis $a = 3$ (vom Rechenaufwand abgesehen) gar nicht so schwierig gewesen:

```
m:=2^(2^5)+1;
Z_m:=Dom::IntegerMod(m);
Z_m(3)^(m-1);
```

$$3029026160 \pmod{4294967297}$$

Diese Basis kann man generell für den Fermat-Test von F_k verwenden und zeigen, dass auch die nächsten Fermatzahlen (deren Stellenzahl sich allerdings jeweils verdoppelt) zusammengesetzt sind. Auf dieser Basis kann man auch Primzahlzertifikate erzeugen:

Satz 20 *Test von Pepin, 1877 Eine Fermatzahl $F_k, k>1$ ist genau dann prim, wenn $3^{(F_k-1)/2} \equiv -1 \pmod{F_k}$ gilt.*

Beweis : Die eine Richtung folgt sofort aus dem Satz von Lucas-Lehmer. Ist umgekehrt F_k eine Primzahl, so gilt $2^{2^k} \equiv 1 \pmod 3$ und damit $F_k \equiv 2 \pmod 3$. F_k ist also ein quadratischer Nichtrest $\pmod 3$ und $\left(\frac{F_k}{3}\right) = -1$. Nach dem quadratischen Reziprozitätsgesetz ergibt sich

$$3^{(F_k-1)/2} \equiv \left(\frac{3}{F_k}\right) = (-1)^{(F_k-1)/2}\left(\frac{F_k}{3}\right) = -1 \pmod{F_k}.$$

□

```
PepinTest:=proc(n:DOM_INT) local m,D;
begin
  m:=2^(2^n)+1;
  D:=Dom::IntegerMod(m);
  iszero(D(3)^((m-1)/2) + 1)
end_proc;
```

Mit diesem einfachen Test konnte bewiesen werden, dass die Zahlen F_k mit $5 \le k \le 23$ zusammengesetzt sind. Die nächsten beiden Zahlen, für welche diese Frage offen ist, sind F_{24} und F_{31}. F_{24} hat 5 050 446 und F_{31} hat 646 456 994 Stellen.

Primzahlen dieser Gestalt spielen eine große Rolle in der Frage der Konstruierbarkeit regelmäßiger n-Ecke mit Zirkel und Lineal. So konnte Gauss die Konstruierbarkeit des regelmäßigen 17-Ecks nachweisen, weil die Primzahl 17 in dieser Reihe auftritt, vgl. etwa [5].

Die Faktorisierung $F_5 = 641 \cdot 6700417$ fand erstmals Euler, allerdings scheiterte er bereits an der nächsten Zahl $F_6 = 18446744073709551617$, deren Faktorisierung

$$F_6 = 67280421310721 \cdot 274177$$

erst im Jahre 1880 entdeckt wurde. Ein modernes CAS berechnet diese Zerlegung heute im Bruchteil einer Sekunde, kommt aber bei der nächsten Fermatzahl

$$F_7 = 340282366920938463463374607431768211457$$

bereits in Schwierigkeiten. Man geht heute davon aus, dass es außer den bereits gefundenen keine weiteren primen Fermatzahlen gibt. Für einen Beweis dieser Vermutung gibt es jedoch nicht einmal ansatzweise Ideen, vgl. [7].

4.8 Der AKS-Primzahltest – ein Primtestverfahren in Polynomialzeit

Moderne Primtestverfahren verwenden auch andere Gruppen als $\mathbb{Z}_m^*$ zum Test, insbesondere solche, die mit elliptischen Kurven verbunden sind. Bis vor Kurzem kannte man noch keinen sicheren Primzahltest mit garantiert polynomialem Laufzeitverhalten.

Anfang August 2002 verbreitete sich in der Primzahltest-Gemeinde die Nachricht wie ein Lauffeuer, dass einige bis dahin unbekannte Inder einen deterministischen Primzahltest mit polynomialer Laufzeit entdeckt hätten. Die Anerkennung und Beweisglättung durch führende Experten folgte im Laufe einer Woche, so dass damit eines der großen Probleme der Komplexitätstheorie eine Lösung gefunden hat.

Die Entdecker dieses Beweises sind Manindra **A**grawal, Professor am Indian Institute of Technology in Kanpur seit 1996 sowie Neeraj **K**ayal und Nitin **S**axena, zwei Studenten und Mitglieder der indischen Mannschaft bei der Internationalen Mathematik-Olympiade 1997.

Besonders erstaunlich ist die Tatsache, dass – etwa im Gegensatz zum Beweis des „großen Fermat“ – der Beweis nur relativ einfache algebraische Argumente verwendet und gut auch von Mathematikern mit „durchschnittlichen“ Kenntnissen der Zahlentheorie nachvollzogen werden kann.

Der AKS-Test nutzt Rechnungen in endlichen Körpern $GF(p^k)$ mit $k > 1$ und geht von folgender Charakterisierung von Primzahlen aus:

Satz 21 *Sei $n \in \mathbb{N}, a \in \mathbb{Z}_n^*$. Dann gilt die Gleichung*

$$(x+a)^n = x^n + a \pmod{n}$$

genau dann, wenn n eine Primzahl ist.

Dieser Satz verallgemeinert den kleinen Satz von Fermat ($x = 0$).

Beweis : Es gilt $\binom{n}{k} \equiv 0 \pmod{n}$, wenn n eine Primzahl und $0 < k < n$ ist. Beweis der Umkehrung in einer Übungsaufgabe. $\square$

Ist wieder $m = l(n) \sim \log(n)$ die Bitlänge der zu untersuchenden Zahl n, so enthält die linke Seite in expandierter Form etwa $n \sim 2^m$ Terme, ist also bereits als Datenstruktur exponentiell. Wir führen deshalb eine weitere Reduktion $\pmod{f(x)}$ mit einem (monischen) Polynom $f(x) \in \mathbb{Z}_n[x]$ vom Grad $\deg(f(x)) = r$ aus, rechnen also in $R = \mathbb{Z}_n[x]/(f(x))$.

Für primes n und irreduzibles $f(x)$ ist das gerade der endliche Körper $GF(n^r)$.

Wir hatten bereits gesehen, dass das Rechnen in R so einfach wie Rechnen mit $\mathbb{Z}_n$-Vektoren der Länge r ist, weil $f(x)$ einer algebraischen Ersetzungsregel für x^r entspricht.

Besonders einfach wird die Rechnung für $f(x) = x^r - 1$. Es gilt

$$n \text{ ist prim } \Rightarrow (x+a)^n \equiv x^n + a \mod (x^r - 1, n).$$

Gefragt sind Werte (r, a), für welche die Umkehrung dieser Aussage richtig ist.

Satz 22 (Der Satz von [AKS], 14.08.2002) *Für $n \in \mathbb{N}$ seien r, q so gewählt, dass* $\gcd(n, r) = 1$*, $q \mid r - 1$ und $n^{(r-1)/q} \not\equiv 1 \pmod{r}$ gilt.*

Sei weiter S eine genügend große Menge von Restklassen aus $\mathbb{Z}_n$ mit $\gcd(n, a - a') = 1$ für alle $a, a' \in S$. Genügend groß bedeutet dabei ($s = |S|$)

$$\binom{q+s-1}{s} \geq n^{2\lfloor\sqrt{r}\rfloor}\,.$$

Gilt dann

$$(x+a)^n \equiv x^n + a \mod (x^r - 1, n)$$

für alle $a \in S$, so ist n eine Primzahlpotenz.

Der Beweis dieses Satzes erfolgt weiter unten. Wir diskutieren zunächst seine Konsequenzen.

Eine Wahl für die Parameter ist z.B. (Existenz von entsprechenden q und r vorausgesetzt)

$$q \geq 4\sqrt{r}\cdot m, \qquad S = \{1, \ldots, s\} \text{ mit } s = 2\sqrt{r}\cdot m, \qquad m = \log_2(n)\,,$$

denn es gilt

$$\binom{q+s-1}{s} \geq \left(\frac{q}{s}\right)^s \geq 2^{2\sqrt{r}\cdot m} = n^{2\sqrt{r}}$$

Damit ergibt sich der folgende **Primtest-Algorithmus**

1. Wenn n echte Primzahlpotenz $\Rightarrow$ `return false`
2. Wähle geeignete (q, r, s)
3. Für $a \in \{1, \ldots, s\}$ prüfe
   (a) Ist $\gcd(a, n) > 1$? $\Rightarrow$ `return false`
   (b) Ist $(x+a)^n \not\equiv x^n + a \mod (x^r - 1, n)$? $\Rightarrow$ `return false`
4. `return true`

Kosten

Schritt 1 kann mit Newton-Iteration in polynomialer Laufzeit erledigt werden: $n = a^k$ bedeutet $a = n^{1/k}$. Für fixiertes k kann $n^{1/k}$ näherungsweise ausgerechnet werden und für die nächstgelegenen ganzen Zahlen a die Beziehung $n = a^k$ geprüft werden. Die Zahl der in Frage kommenden Exponenten ist durch $k < \log_2(n) = m$ höchstens linear in m.

Die größten Kosten verursacht Schritt 3b. Diese sind bei schneller FFT-Arithmetik bis auf logarithmische Faktoren wie das Rechnen mit $\mathbb{Z}_n$-Vektoren der Länge r, also $\tilde{O}(r\,s\,m^2)$

Die **Gesamtkosten** betragen bei obiger Wahl von s also gerade $\tilde{O}(r^{3/2}\,m^3)$.

Der Algorithmus hat also polynomiale Laufzeit, wenn wir zeigen können, dass es geeignete $r \in m^{O(1)}$ gibt. Es zeigt sich, dass dazu maximal $O(m^6)$ Zahlen getestet werden müssen.

Zur Wahl von q und r

Gesucht sind Primzahlen r, für die $r - 1$ einen „großen“ Faktor q besitzt.

Dazu wird ein Ergebnis der analytischen Zahlentheorie über die Dichte von Primzahlen r mit großem Faktor von $r - 1$ verwendet. Diese sind für große x genauso so häufig wie Primzahlen. Genauer, für

$$P(x) = \left\{ r \leq x \;:\; \exists\, q \ (q, r \text{ prim}) \wedge (q \mid r-1) \wedge (q > x^{2/3}) \right\}$$

gilt

$$|P(x)| \gtrsim \pi(x) \sim \frac{x}{\log(x)}.$$

Für den größten Primfaktor q eines solchen $r \in P(x)$ gilt also $k_0 = \frac{r-1}{q} < x^{1/3}$.

Wir müssen für [AKS] solche r ausschließen, für welche $n^{k_0} \equiv 1 \pmod r$ ist. Dazu fordern wir stärker, dass $n^k - 1$ für kein $k < x^{1/3}$ durch r teilbar ist.

$n^k - 1$ hat bei festem k höchstens $O(k \cdot m)$ Primteiler. Hier ist wieder $m = l(n) \sim \log(n)$ die Bitlänge von n. Also sind insgesamt höchstens

$$O\left(\sum_{k=1}^{x^{1/3}} k\, m\right) = O(x^{2/3}\, m)$$

Werte von r zu vermeiden. Es reicht also, x so groß zu wählen, dass $x^{2/3}\, m \gtrsim \dfrac{x}{m}$, also $x \gtrsim m^6$ gilt, um ein r mit der geforderten zusätzlichen Teilbarkeitseigenschaft zu finden.

Der Beweis des Satzes von [AKS]

Sei p ein Primfaktor von n mit $p^{(r-1)/q} \not\equiv 1 \pmod r$. Wegen $p^{(r-1)} \equiv 1 \pmod r$ ist $d = ord(p \in \mathbb{Z}_r^*)$ ein Vielfaches von q.

Nach Voraussetzung gilt $(x+a)^n = x^n + a$ in $R = \mathbb{Z}_p[x]/(x^r - 1)$ für alle $a \in S$.

Die Substitution $x \mapsto x^t$ zeigt, dass dann auch

$$(x^t + a)^n = x^{n\,t} + a \quad \text{in } \mathbb{Z}_p[x]/(x^{r\,t} - 1)$$

und wegen $(x^r - 1) \mid (x^{r\,t} - 1)$ auch in R gilt. Mit Induktion nach i ergibt sich

$$(x+a)^t = x^t + a \quad \text{in } R \text{ für alle } t \in \{n^i, i \geq 0\}.$$

In $\mathbb{Z}_p[x]$ gilt generell $f(x^p) = f(x)^p$: Für $f(x) = \sum a_i x^i$ folgt

$$\left(\sum a_i x^i\right)^p = \sum a_i^p x^{i\,p} = \sum a_i x^{i\,p}$$

wegen $a^p = a$ in $\mathbb{Z}_p$. Damit gilt analog

$$(x+a)^t = x^t + a \quad \text{in } R \text{ für alle } t \in \{n^i p^j, i, j \geq 0\}.$$

Betrachten wir nun die $n^i p^j$ mit $0 \leq i, j \leq \lfloor\sqrt{r}\rfloor$ und nehmen an, dass verschiedene Paare (i, j) aus diesem Bereich auch verschiedene Zahlen liefern. Es gibt wenigstens $r + 1$ solche Paare und für jedes von ihnen gilt $n^i p^j < n^{i+j} \leq n^{2\lfloor\sqrt{r}\rfloor}$.

Nach dem Schubfachprinzip gibt es also $t = n^{i_1} p^{j_1} \neq u = n^{i_2} p^{j_2}$ mit $|t - u| < n^{2\lfloor\sqrt{r}\rfloor}$, $t \equiv u \pmod r$ und folglich $x^t = x^u$ in R. Damit gilt auch $(x+a)^t = (x+a)^u$ in R für alle $a \in S$.

R ist kein Körper, da $x^r - 1 \in \mathbb{Z}_p[x]$ nicht irreduzibel ist. Aus dem Satz über die Faktorzerlegung der Kreisteilungspolynome über $\mathbb{Z}_p$ (Abschnitt 3.5) wissen wir

$$x^r - 1 = (x-1) \cdot h_1(x) \cdot \ldots \cdot h_s(x) \pmod p,$$

wobei $h_i(x) \in \mathbb{Z}_p[x]$ irreduzible Faktoren sind, die alle denselben Grad $d = ord(p \in \mathbb{Z}_r^*)$ haben, und $s = \frac{r-1}{d}$ gilt. d ist dasselbe wie oben, so dass $d \geq q \geq 2$ gilt.

Ist $h(x)$ einer dieser irreduziblen Faktoren, so können wir R durch den Körper $K = R/(h(x)) = \mathbb{Z}_p[x]/(h(x))$ ersetzen. Auch in K gilt $(x+a)^t = (x+a)^u$ für alle $a \in S$. Schließlich gilt aus Gradgründen $(x+a) \neq 0$ in K für jedes a.

Betrachten wir nun die Gruppe $G \subset K^*$, die von $\{x + a \,:\, a \in S\}$ erzeugt wird. Dann gilt $g^t = g^u$ für alle $g \in G$.

G hat wenigstens $\binom{q+s-1}{s} \geq n^{2\lfloor\sqrt{r}\rfloor} > |t - u|$ Elemente, denn die Produkte $\prod_{a \in S} (x+a)^{e_a}$ mit $\sum_{a \in S} e_a < q$ sind paarweise verschieden – sie sind verschieden in $\mathbb{Z}_p[x]$ und haben alle einen Grad

$< q \leq d = \deg(h(x))$ und die Zahl der Terme in s Variablen vom Grad $< q$ ist gerade gleich $\binom{q+s-1}{s}$.

Folglich hat das Polynom $Y^{|t-u|} - 1 \in K[Y]$ mehr als $|t-u|$ Nullstellen in K. Dieser Widerspruch zeigt: die Annahme, dass alle $n^i p^j$ mit $0 < i, j < \lfloor\sqrt{r}\rfloor$ paarweise verschieden sind, war falsch.

Also gibt es Paare $(i_1, j_1) \neq (i_2, j_2)$ mit $n^{i_1} p^{j_1} = n^{i_2} p^{j_2}$, womit auch $n = p^k$ mit $k = \dfrac{j_2 - j_1}{i_2 - i_1}$ gilt.
$\square$

5 Faktorisierungs-Algorithmen

Faktorisierungsverfahren arbeiten meist so, dass sie zuerst einen Primtest anwenden, dann von einer als zusammengesetzt erkannten Zahl m einen echten Faktor n bestimmen und schließlich rekursiv n und m/n faktorisieren.

```
FactorA:=proc(m:DOM_INT,splitFactorFunction) local n;
begin
  if isprime(m) then return(m) end_if;
  n:=splitFactorFunction(m);
  FactorA(n,splitFactorFunction), FactorA(m/n,splitFactorFunction)
end_proc;
```

5.1 Faktorisierung durch Probedivision

Unser erstes Primtestverfahren, `primeTestByTrialDivision(m)`, fand zusammen mit der Erkenntnis, dass m zusammengesetzt ist, auch einen Faktor dieser Zahl. Eine entsprechende Faktorabspaltung, die für zusammengesetzte Zahlen einen echten Faktor und für Primzahlen die Zahl selbst zurückgibt, hätte dann folgende Gestalt

```
splitTrialFactor:=proc(m:DOM_INT) local z;
begin
  if (m<3) then return(m) end_if;
  z:=2;
  while z*z<=m do
    if m mod z = 0 then return(z) end_if;
    z:=z+1;
  end_while;
  m;
end_proc;
```

Das Laufzeitverhalten hängt von der Größe des entdeckten Faktors ab, d.h. vom kleinsten Primfaktor r der Zahl m, und ist von der Größenordnung $O(r \cdot l(m)^2)$, also im Fall zweier etwa gleichgroßer Faktoren schlimmstenfalls $O(\sqrt{m} \cdot l(m)^2)$.

Die zugehörige Faktorisierungsfunktion ist dann

```
trialFactor:=proc(m:DOM_INT) begin [FactorA(m,splitTrialFactor)] end;
```

wobei es natürlich günstiger ist, die Suche nach weiteren Faktoren an der Stelle fortzusetzen, wo der letzte Faktor gefunden wurde und nicht die Faktorisierung mit den beiden Faktoren n und m/n neu zu starten. Schließlich ist n nach Konstruktion sowieso prim und m/n durch keinen Primfaktor $< n$ teilbar.

Dieses Verfahren ist jedoch nur für Zahlen geeignet, die aus kleinen und möglicherweise einem einzelnen großen Primfaktor bestehen. Zahlen, deren Faktorzerlegung mehrere zehnstellige Primteiler enthält, lassen sich selbst auf modernen Computern nicht auf diese Weise zerlegen.

5.2 `smallPrimeFactors` und CAS-Implementierungen

Andererseits gibt es kein anderes Verfahren, das so effizient kleine Faktoren zu finden vermag. Deshalb werden in praktischen Implementierungen in einem Preprocessing kleine Teiler durch ein solches Probedivisionsverfahren `smallPrimeFactors` vorab herausdividiert. Zusammengesetzte Zahlen, die aus vielen solchen kleinen Teilern und einem großen primen „Rest“ bestehen, können auf diese Weise vollständig faktorisiert werden. In der folgenden MuPAD-Implementierung sind das die Primfaktoren < 30.

```
smallPrimeFactors:=proc(m:DOM_INT) local m0,u,i,smallPrimes;
begin
  u:=null(); m0:=m;
  smallPrimes:=[2, 3, 5, 7, 11, 13, 17, 19, 23, 29];
  for i in smallPrimes do
    if m0=1 then break end_if;
      while (m0 mod i = 0) do u:=u,i; m0:=m0/i end_while;
  end_for;
  [m0,u];
  /* erster Listeneintrag ist der noch nicht zerlegte Rest */
end_proc;
```

Für die Zahl $m = 10^{11} + 1$ ergibt sich

```
smallPrimeFactors(10^11+1);
```

$$[35932447, 11, 11, 23]$$

wobei $35932447 = 4093 \cdot 8779$ der nicht weiter zerlegte Rest ist. MuPAD verwendet dazu eine beim Systemstart berechnete Primzahltabelle bis zu einer mit

```
ifactor(PrimeLimit);
```

ermittelbaren Grenze von etwa 10^6. In den großen CAS kann man die Faktorisierung auf diese „einfachen" Faktoren begrenzen, etwa in MuPAD mit

```
ifactor(10^43+1,UsePrimeTab);
```

$$11 \cdot 909090909090909090909090909090909090909189$$

Maple liefert analog

```
ifactor(10^43+1,easy);
```

```
(11) _c35_1 (57009401)
```

wobei _c35_1 für einen 35-stelligen zusammengesetzten Faktor steht, Mathematica wie MuPAD nur

```
FactorInteger[10^43+1,FactorComplete->False]
```

$$\{\{11, 1\}, \{909090909090909090909090909090909090909189, 1\}\}$$

Die vollständige Faktorzerlegung dieser Zahl lautet

$$10^{43} + 1 = 11 \cdot 7306116556571817748755241 \cdot 2182600451 \cdot 57009401$$

und wurde sowohl von Maple 8 (2.13 s.) als auch MuPAD 2.5.0 (9.04 s.) und Mathematica 4.1 (29.18 s.) gefunden.

Eine entsprechende Rahmenprozedur spaltet kleine Primfaktoren ab, untersucht, ob der verbleibende Rest bereits prim ist und wendet ggf. rekursiv eine „harte" Faktorsuche an.

```
FactorB:=proc(m:DOM_INT,splitFactorFunction) local u;
begin
  u:=smallPrimeFactors(m); m:=u[1];
  if m = 1 then u[2..nops(u)]
  elif isprime(m) then u
  else [op(u,2..nops(u)), FactorA(m,splitFactorFunction)]
  end_if;
end_proc;
```

5.3 Faktorisierungsverfahren – das globale Bild

Die Laufzeit der Faktorisierung von m durch Probedivision hat schlimmstenfalls die Größenordnung $O(\sqrt{m} \cdot l(m)^2)$. Wegen $\sqrt{m} = 2^{\log_2(m)/2} \in 2^{O(l(m))}$ handelt es sich also um einen Algorithmus mit exponentieller Laufzeit. Alle klassischen Faktorisierungsalgorithmen gehören zu dieser Laufzeitklasse $2^{O(l(m))}$. Die Laufzeitabschätzung hat damit die Form $O(m^\alpha \cdot l(m)^k)$, in welcher der genaue Wert des Exponenten α wichtiger ist als die Exponenten k. Wir schreiben deshalb auch $\tilde{O}(m^\alpha)$, wenn polylogarithmische Faktoren $l(m)^k$ nicht mit berücksichtigt werden.

$\alpha = 0$ entspricht einem (bisher nicht bekannten) Verfahren mit polynomialer Laufzeit, $\alpha > 0$ rein exponentieller Laufzeit. Alle klassischen Faktorisierungsverfahren gehören zur letzteren Kategorie und unterscheiden sich nur in der Größe von α. Für `trialFactor` gilt $\alpha = \frac{1}{2}$. Moderne Faktorisierungsverfahren erreichen subexponentielle Laufzeit. Dies bedeutet, dass deren Komplexität von der Größenordnung $\tilde{O}(m^{\alpha_m})$ mit $\alpha_m \to 0$ ist.

Um Zahlen m zu faktorisieren, die in mehrere „große“ Primfaktoren aufspalten, wie das etwa für die meisten der Fermatzahlen $F_n = 2^{2^n} + 1, n \geq 5$ der Fall ist, sind andere Verfahren als die Probedivision erforderlich. Solche Verfahren bestehen im Kern aus einer Routine, welche in der Lage ist, einen nicht trivialen Faktor $n \mid m$ zu finden, und die rekursiv oder kombiniert mit anderen Verfahren eingesetzt wird, um die vollständige Primfaktorzerlegung zu bestimmen.

Der rekursive Aufruf der Faktorisierung für n und m/n, der ja alle bis dahin gesammelte Information über m „vergisst“, ist für komplizierte Faktorisierungsprobleme gerechtfertigt, da auf Grund des exponentiellen bzw. subexponentiellen Charakters der Algorithmen die komplette Faktorisierung der „kleineren“ Faktoren oft nicht mehr zeitkritisch ist. Außerdem sind die meisten `splitFactor`-Algorithmen auf die Zahl m zugeschnitten, so dass Zwischenergebnisse nicht so einfach weiter zu verwenden sind.

Je nach eingesetztem `splitFactor`-Algorithmus unterscheidet man zwischen Faktorisierungsverfahren erster und zweiter Art. *Faktorisierungsverfahren der ersten Art* produzieren (mit großer Wahrscheinlichkeit) kleinere Faktoren, so dass ihre (durchschnittliche) Laufzeit von der Größe des kleinsten Primfaktors r der Zahl m abhängt. Zu diesen Verfahren gehören die Pollardsche Rho-Methode, mit der Brent und Pollard 1981 spektakulär die Fermatzahl

$$F_8 = 1238926361552897 \cdot 93461639715357977769163558199606896584051237541638188580280321$$

faktorisieren konnten (MuPAD heute: 350 s.), Pollards $(p-1)$-Methode sowie die auf elliptischen Kurven basierenden Verfahren, mit denen die Faktorzerlegungen von F_{10} und F_{11} entdeckt wurden. Sie sind für Aufgaben sinnvoll einsetzbar, in denen Faktoren bis zu 40 Stellen abzuspalten sind und finden sehr effektiv Faktoren bis zu 20 Stellen. Sie sollten deshalb immer – nach dem Abspalten kleiner Faktoren – zuerst versucht werden.

Für Faktorisierungen bis zu 100-stelliger Zahlen m, bei denen die bisher beschriebenen Methoden versagen, stehen Verfahren der zweiten Art zur Verfügung. Sie werden nach dem Zahlentheortiker Maurice Kraitchik (1882 - 1950), der ein solches Verfahren erstmal etwa 1920 vorschlug, auch als *Verfahren der Kraitchik-Familie* bezeichnet. Dieses Verfahren – das quadratische Sieb – wurde allerdings erst um 1980 implementiert.

Verfahren der zweiten Art beruhen alle darauf, ein Paar (x, y) mit $x^2 \equiv y^2 \pmod{m}$ zu finden, womit $\gcd(m, x-y)$ ein echter Teiler von m ist. Da die Zahlen x, y etwa die Größe von m haben, werden eher große Faktoren entdeckt und die Laufzeit des Verfahrens hängt nicht von von der Größe des kleinsten Primfaktors r, sondern nur von der Größe von m ab. All diese Verfahren sind kompliziert und wegen der Tatsache, dass sie das Vorhandensein kleiner Primfaktoren nicht honorieren, für die Faktorisierung von Zahlen ohne „kleine“ Primfaktoren besonders gut geeignet. So faktorisierten Brillhard und Morrison 1975 mit der Kettenbruchmethode erstmals die 39-stellige Zahl

$$F_7 = 59649589127497217 \cdot 5704689200685129054721$$

(MuPAD heute: 50 s.). Mit verschiedenen Siebmethoden wurde die Faktorzerlegung der 155-stelligen Zahl F_9 gefunden, die selbst für Maple-Verhältnisse keine „kleinen“ Faktoren enthält.

Die meisten Faktorisierungsrekorde entstanden im Zusammenhang mit dem Cunningham-Projekt, alle Zahlen der Form $b^n \pm 1$ mit $2 \leq b \leq 12$ zu faktorisieren. Mehr zu dem Projekt unter `http://www.cerias.purdue.edu/homes/ssw/cun`.

5.4 Die Fermat-Methode

Nachdem wir mit `splitTrialFactor` ein deterministisches Verfahren der ersten Art kennen gelernt haben, soll nun ein deterministisches Verfahren der zweiten Art beschrieben werden, das in praktischen Implementierungen zwar keine große Rolle spielt, aber dessen Ansatz für fortgeschrittenere Verfahren wichtig ist. Es geht auf Pierre Fermat zurück.

Seine Idee ist die Folgende: Für eine ungerade zusammengesetzte Zahl $m = a \cdot b$ sind $x = \frac{a+b}{2}, y = \frac{a-b}{2}$ ganze Zahlen und

$$m = a \cdot b = (x + y)(x - y) = x^2 - y^2.$$

Können wir umgekehrt $m = x^2 - y^2$ als Differenz zweier Quadrate schreiben, so haben wir eine Faktorzerlegung von m gefunden.

Wir suchen eine solche Darstellung, indem wir von $x = \lceil\sqrt{m}\rceil$ starten und jeweils prüfen, ob $s = x^2 - m$ ein genaues Quadrat ist. Dabei wird die MuPAD-Funktion `isqrt` verwendet, die `isqrt(s)` $= \lfloor\sqrt{s}\rfloor$ mit dem Newtonverfahren berechnet wie in den Übungsaufgaben beschrieben.

```
splitFermatFactor:=proc(m:Type::Odd) local x,y,s;
begin
  x:=isqrt(m);
  repeat x:=x+1; s:=x^2-m; y:=isqrt(s);
  until (y^2=s) end_repeat;
  x-y;
end_proc;
```

Der schlechteste Fall tritt für $m = 3p$ ein, wo $x = (m + 9)/6$ und $y = (m - 9)/6$ gilt. Die repeat-Schleife wird also nach höchstens $O(m)$ Durchläufen verlassen und wir haben im schlechtesten Fall

$$C_{\text{Fermat}}(m) = \tilde{O}(m^1).$$

Wir können den mehrfachen Aufruf von `isqrt` vermeiden, wenn wir von $x = \lfloor\sqrt{m}\rfloor + 1, y = 0$ ausgehen, nur die Änderungen von $r = x^2 - y^2 - m$ protokollieren und jeweils y bzw. x inkrementieren, je nachdem ob $r > 0$ oder $r < 0$ gilt. Für $r = 0$ haben wir eine Zerlegung $m = x^2 - y^2$ gefunden. Da die ganze Zeit $x > y$ gilt, folgen auf eine Erhöhung von x stets mehrere Erhöhungen von y. Zusammengenommen sieht eine entsprechende Implementierung wie folgt aus:

```
splitFermatFactor:=proc(m:Type::Odd) local r,s,u,v;
begin
  s:=isqrt(m)+1;
  if (m=(s-1)^2) then return (s-1) end_if;
  r:=s^2-m; u:=2*s+1; v:=1;
  while(r<>0) do
    while(r>0) do r:=r-v; v:=v+2 end_while;
    if (r<0) then r:=r+u; u:=u+2 end_if;
  end_while;
  (u-v)/2;
end_proc;

FermatFactor:= proc(m:DOM_INT)
  begin FactorB(m,splitFermatFactor) end_proc;
```

`FermatFactor` funktioniert dann besonders gut, wenn m in zwei etwa gleich große Faktoren zerfällt. Das können wir durch Skalierung versuchen zu verbessern. So findet `splitFermatFactor` für $m = 2581 = 29 \cdot 89$ den Faktor 29 erst nach 10 Durchläufen, während für $3m = 7743$ der Faktor $3 \cdot 29 = 87$ bereits nach zwei Durchläufen gefunden ist.

Da `trialFactor` mit etwa gleich großen Faktoren besondere Probleme hat, sollte eine Mischung beider Verfahren, welche die jeweiligen Stärken kombiniert, zu Laufzeitvorteilen führen. Das folgende **Verfahren von Lehman** wendet `trialFactor` für $d \leq m^{1/3}$ an und geht dann zur Suche nach Quadratzerlegungen von $k\,m = x^2 - y^2$ für verschiedene $k,\ 1 \leq k \leq m^{1/3}$, und x in der Nähe von $2\sqrt{k\,m}$ über.

```
splitLehmanFactor:=proc(m) local d,k,x,y;
begin
  for 2 ≤ d ≤ ⌊m^(1/3)⌋ do
    if d|m then return d
  for 1 ≤ k ≤ ⌈m^(1/3)⌉ do
    for ⌈2√(k m)⌉ ≤ x ≤ ⌊2√(k m) + m^(1/6)/(4√k)⌋ do
      if y = √(x^2 − 4 k m) ∈ ℕ then return gcd(x+y, m)
end;
```

Die zweite innere Schleife wird etwa

$$\sum_{k=1}^{m^{1/3}} \frac{m^{1/6}}{4\sqrt{k}} \sim \frac{1}{2} m^{1/3}$$

mal durchlaufen, wobei immer `isqrt` aufgerufen wird, dessen Laufzeit polynomial in der Bitlänge $l(m)$ ist. Da auch die erste Schleife in ähnlicher Zeit abgearbeitet werden kann, ergibt sich eine Gesamtlaufzeit dieses Verfahrens von der Größenordnung

$$C_{\mathrm{LehmanFactor}}(m) = \tilde{O}(m^{1/3}),\ \text{also } \alpha = \frac{1}{3}$$

im Gegensatz zur `trialFactor` ($\alpha = \frac{1}{2}$) und `FermatFactor` ($\alpha = 1$). Wir haben damit das erste Faktorisierungs-Verfahren kennengelernt, das im Mittel schneller als die Probedivision ist.

Zum Beweis der Korrektheit des Verfahrens siehe [3, 5.1.2].

5.5 Die Pollardsche Rho-Methode

1975 schlug J. Pollard eine neuen Zugang zu Faktorisierungsalgorithmen vor, die Eigenschaften von (deterministischen) Zahlenfolgen verwendet, die sich fast wie Zufallsfolgen verhalten.

Die grundlegende Idee dieses Verfahrens benutzt Fixpunkteigenschaften von Abbildungen der endlichen Menge $S = \mathbb{Z}_r = \{0, 1, \ldots, r-1\}$ auf sich selbst. Eine solche Abbildung $f : S \to S$ ist durch die Werte $f(0), \ldots, f(r-1)$ eindeutig bestimmt, die andererseits frei gewählt werden können. Es gibt also r^r solche Funktionen.

Betrachten wir nun eine Folge $\mathbf{x} = (x_0, x_1, x_2, \ldots)$, deren Glieder der Bedingung $x_{i+1} = f(x_i)$ genügen. Wir wollen eine solche Folge als *Pollard-Sequenz* bezeichnen. Sie ist durch den Startwert x_0 und die Übergangsfunktion f eindeutig bestimmt.

Wegen der Endlichkeit der auftretenden Reste gibt es in jeder solchen Pollard-Sequenz ein Indexpaar $k > j$, so dass $x_k = x_j$ gilt, d.h. sich nach endlich vielen Schritten Folgenglieder wiederholen. Nach der Bildungsvorschrift gilt dann auch $x_{k+i} = x_{j+i}$ für $i \geq 0$, d.h. die Folge ist in Wirklichkeit sogar periodisch, evtl. mit einer Vorperiode.

Beispiel:

```
gen:=proc(x,f,n) local l,y,i;
begin l:=x; y:=x;
  for i from 1 to n do y:=f(y); l:=l,y end_for;
  [l]
end_proc;
```

erzeugt aus einem Startwert x und einer Funktion f die ersten 15 Elemente der entsprechenden Pollard-Sequenz. Aufrufe mit verschiedenen Startwerten und Moduln sowie der Funktion $f(x) = x^2 + 1$ liefern:

```
gen(2,x -> x^2+1 mod 13,15);
             [2, 5, 0, 1, 2, 5, 0, 1, 2, 5, 0, 1, 2, 5, 0, 1]

gen(4,x -> x^2+1 mod 13,10);
                     [4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4]

gen(3,x -> x^2+1 mod 13,5);
                          [3, 10, 10, 10, 10, 10]

gen(7,x -> x^2+1 mod 37,15);
          [7, 13, 22, 4, 17, 31, 0, 1, 2, 5, 26, 11, 11, 11, 11, 11]
```

Wir sehen in allen Beispielen, dass unterschiedliche Startwerte dabei zu unterschiedlichen Vorperiodenlängen und auch zu unterschiedlichen Perioden führen können.

Die Funktion $f : \mathbb{Z}_r \to \mathbb{Z}_r$ können wir uns stets, wie im Beispiel $f(x) = x^2 + 1$, als von einer Funktion $f : \mathbb{Z} \to \mathbb{Z}$ induziert vorstellen, so dass wir Pollardsequenzen (f, x_0) bzgl. verschiedener Moduln r vergleichen können. Wir rechnen dazu im Folgenden mit Resten statt Restklassen und schreiben $x_{i+1} \equiv f(x_i) \pmod{r}$.

Ist r ein Teiler von m, so folgt dann aus $x_j \equiv x_k \pmod{m}$ bereits $x_j \equiv x_k \pmod{r}$, d.h. Vorperiodenlänge und Periodenlänge sind für einen Teiler r nicht größer als für die Zahl m selbst.

Ist insbesondere r ein echter Teiler der zu faktorisierenden Zahl m und

$$K(f, x_0)^{(r)} = \min\{k > 0 : \exists j < k \; x_k \equiv x_j \pmod{r}\}$$

der Index, an dem sich das „Rho schließt“, so können wir erwarten, dass „im Durchschnitt“ $K(f, x_0)^{(r)} < K(f, x_0)^{(m)}$ gilt, d.h. für geeignete Folgenglieder

$$x_j \equiv x_k \pmod{r}, \quad \text{aber} \quad x_j \not\equiv x_k \pmod{m} \tag{P}$$

gilt. Dann ist aber $x_j - x_k$ durch r, nicht aber durch m teilbar und somit $\gcd(x_j - x_k, m)$ ebenfalls ein echter Teiler von m. Selbst wenn wir den Teiler r nicht kennen und alle Reste erst einmal nur $\pmod{m}$ bestimmen, können wir $\gcd(x_j - x_k, m)$ für alle Paare $j < k$ ausrechnen und hoffen, dabei auf einen echten Faktor von m zu stoßen.

Beispiel: Betrachten wir wieder die Übergangsfunktion $f(x) = x^2 + 1$ und $m = 91$ und die Pollard-Sequenz bis zum Index 10 zum Startwert 7:

```
o:=gen(7,x -> x^2+1 mod 91,10);
                    [7, 50, 44, 26, 40, 54, 5, 26, 40, 54, 5]
```

Dann bestimmen wir die Menge aller verschiedenen $\gcd(x - y, m)$ für alle Paare (x, y) aus o:

```
{igcd(o[i]-o[j],91) $ j=1..i-1 $i=1..nops(o)};
```

$$\{1, 7, 13, 91\}$$

Wir sehen, dass in diesem Beispiel unter den gcd tatsächlich Teiler der zu faktorisierenden Zahl vorkommen. Die Pollard-Sequenz ist $\pmod{91}$ periodisch mit der Periodenlänge 4, während dieselbe Folge $\pmod{7}$ periodisch mit der Periodenlänge 1 ist:

```
gen(7,x -> x^2+1 mod 91,15);
        [7, 50, 44, 26, 40, 54, 5, 26, 40, 54, 5, 26, 40, 54, 5, 26]

gen(7,x -> x^2+1 mod 7,15);
        [7, 1, 2, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5]
```

Was kann man über den Durchschnittswert $K^{(r)}$ von $K(f, x_0)^{(r)}$, gemittelt über alle f und x_0 sagen? Ist k der kleinste Wert, ab dem sich Funktionswerte wiederholen, so können wir für eine Funktion f, die gut genug ausgewählt wurde, davon ausgehen, dass die Werte in der Vorperiode $x_0, \ldots, x_{k-1}$ und damit auch die verschiedenen Reste, die in der Pollard-Sequenz überhaupt auftreten, „zufällig“ verteilt sind.

Lemma 2 *Sei S eine r-elementige Menge. Dann ist der Anteil der Paare $(f, x_0) \in (S \to S, S)$, für die das Anfangsstück $\{x_0, \ldots, x_l\}$ der zugehörigen Pollard-Sequenz aus paarweise verschiedenen Zahlen besteht, kleiner als $e^{-\frac{l^2}{2r}}$.*

Beweis : Die Anzahl der Paare (f, x_0) ist gleich r^{r+1}, da eine Funktion f durch ihre Wertetabelle eindeutig bestimmt ist und $f(x)$ für jedes der r Argumente aus S ein beliebiges der r Elemente aus S als Funktionswert annehmen kann.

Für welche Paare sind nun $\{x_0, \ldots, x_l\}$ paarweise verschieden? x_0 kann beliebig gewählt werden, d.h. es stehen r verschiedene Möglichkeiten zur Verfügung. Wegen $x_1 = f(x_0) \neq x_0$ kann $f(x_0)$ einen Wert nicht annehmen. Es stehen noch $r-1$ Werte zur Verfügung. Analog darf $x_2 = f(x_1)$ die Werte x_0, x_1 nicht annehmen, d.h. für diesen Funktionswert stehen noch $r-2$ Elemente aus S zur Auswahl usw. Für $x_l = f(x_{l-1})$ können wir noch unter $r-l$ Elementen wählen. Alle anderen Funktionswerte von f haben keinen Einfluss auf die betrachtete Folge, können also beliebig aus den r möglichen Elementen gewählt werden.

Die gesuchte Wahrscheinlichkeit als der Quotient zwischen der Zahl der günstigen und der Zahl der möglichen Auswahlen ergibt sich damit zu

$$p := \frac{r(r-1)\cdot \ldots \cdot (r-l) r^{r-l}}{r^{r+1}} = \prod_{j=0}^{l} \left(1 - \frac{j}{r}\right).$$

Wegen $\log(1-x) < -x$ für $0 < x < 1$ erhalten wir schließlich

$$\log(p) < \sum_{j=0}^{l} -\frac{j}{r} < -\frac{l^2}{2r},$$

woraus die Behauptung folgt. $\square$

Satz 23 *Für den Durchschnittswert $K^{(r)}$ gilt $K^{(r)} \leq \sqrt{\frac{\pi\, r}{2}}$.*

Beweis : Für den Durchschnittswert

$$K^{(r)} = \sum_d d \cdot p(\{(f, x_0)\ :\ K(f, x_0; r) = d\}) = \sum_d p(\{(f, x_0)\ :\ K(f, x_0; r) \geq d\})$$

(vgl. die Komplexitätsberechnung von `comp`) erhalten wir mit obigem Lemma

$$K^{(r)} < \sum_{d \geq 0} e^{-\frac{d^2}{2r}} \approx \int_0^\infty e^{-\frac{x^2}{2r}}\, dx = \sqrt{\frac{\pi\, r}{2}}$$

$\square$

Wir werden also die Pollard-Sequenz Element für Element berechnen und jedes neu berechnete Element sofort mit seinen Vorgängern vergleichen, in der Hoffnung, dass wir spätestens nach $k \approx O(\sqrt{r})$ Gliedern einen nicht trivialen Teiler entdecken.

Dieses Vorgehen ist leider nicht besser als `trialFactor`: Das Berechnen von l Folgengliedern verursacht einen Aufwand von $k \cdot O(l(m)^2)$, die Berechnung der gcd's aller $O(k^2)$ paarweisen Differenzen mit m einen Aufwand von $k^2 \cdot O(l(m)^2)$ und somit für $k \approx \sqrt{r}$ einen Gesamtaufwand in der Größenordnung $\tilde{O}(k^2) = \tilde{O}(r)$.

Um nicht alle Paare (x_i, x_j) zu untersuchen, wenden wir den folgenden **Trick von Floyd** an: Ist in der Pollardsequenz für $j < k$ das erste Mal $x_k \equiv x_j \pmod{r}$ und $l = k-j$ die Periodenlänge, so gilt $x_m \equiv x_{m+l} \pmod{r}$ für $m \geq j$ und damit für jedes genügend große Vielfache von l, insbesondere für $m = l \cdot \lceil j/l \rceil \geq j$, auch $x_m \equiv x_{2m} \pmod{r}$. Wegen

$$m = l \cdot \lceil j/l \rceil < l\left(\frac{j}{l} + 1\right) = j + l = k$$

können wir also mit gleichem Erfolg die Paare $(x_i, x_{2i}),\ i \leq k$, untersuchen.

Eine Implementierung dieser Idee in MuPAD kann wie folgt ausgeführt werden:

```
pollardRhoEngine:=proc(m,f,x0) local u,v,g;
begin
  u:=x0; v:=x0;      // u=v=x_1
  repeat
    u:=f(u) mod m;   // u=x_i
    v:=f(v) mod m;
    v:=f(v) mod m;   // v=x_2i
    g:=igcd(u-v,m);
  until g>1 end_repeat;
  if (g=m) then FAILED else g end_if;
end_proc;
```

In der folgenden Implementierung wird die Pollardsche Rho-Methode für $f(x) = x^2 + 1$ und verschiedene Startwerte angewendet. Alternativ können auch andere einfach zu berechnende Funktionen wie $x^2 - 1, x^2 + 3$ oder $x^2 + 5$ eingesetzt werden.

```
splitPollardRhoFactor:=proc(m:DOM_INT) local a,g,r;
begin
  r:=random(m);
  for i from 1 to 100 do /* Zahl der Versuche */
    a:=r();
    g:=igcd(a,m); if g>1 then return(g) end_if;
    g:=pollardRhoEngine(m,x->x^2+1,a);
    if g<>FAILED then return(g)
    end_if;
  end_for;
  error("Faktorisierung von ".m." fehlgeschlagen");
end_proc;

pollardRhoFactor:=proc(m:DOM_INT)
   begin FactorB(m,splitPollardRhoFactor) end_proc;
```

Ein Vergleich der Laufzeiten von `trialFactor` und `pollardFactor` für die ersten Mersennezahlen $M_p = 2^p - 1$, für die p prim, aber M_p nicht prim ist, zeigt den Gewinn eindrucksvoll.

```
MersenneNonPrimes:=select([$10..100],p->isprime(p) and not isprime(2^p-1));
```

```
timecheck:=proc(n) local t,t1,t2,f;
begin // Achtung: Für p=67 wird es für trialFactor problematisch.
  t1:=FAILED;
  traperror((t:=time(); trialFactor(n); t1:=time()-t),2);
  t:=time(); f:=pollardRhoFactor(n); t2:=time()-t;
  print(Unquoted,"".n." == ".expr2text(f));
  t1,t2;
end_proc;

map(MersenneNonPrimes,p->[p,timecheck(2^p-1)]):
```

In der folgenden Tabelle sind die Laufzeiten für beide Verfahren und $M_p = 2^p - 1$ unter MuPAD 2.5 zusammengestellt (t_1 = `trialFactor`, t_2 = `pollardRhoFactor`, Zeitangaben in ms., $*$ bedeutet mehr als 10 s.).

p	11	23	29	37	41	43	47	53	59	67	71	73	79	83	97
t_1	0	1	13	2	125	102	68	757	1720	$*$	$*$	$*$	$*$	4	113
t_2	0	1	5	2	14	4	5	32	52	836	130	200	422	3	6

Trotz der beeindruckenden Laufzeitunterschiede zwischen beiden Verfahren ist allerdings auch die Pollardsche Rho-Methode im schlechtesten Fall von exponentieller Laufzeit, denn wenn m in zwei etwa gleichgroße Faktoren zerfällt, dann gilt $r \sim O(\sqrt{m})$ und damit

$$C_{\text{Pollard-Rho}}(m) \sim \tilde{O}(m^\alpha) \text{ mit } \alpha = \frac{1}{4}\,.$$

5.6 Das quadratische Sieb

Die nächste Faktorisierungsmethode gehört zu den Faktorisierungsverfahren der zweiten Art und ist eine Verfeinerung der Fermat-Methode. Die Idee soll zunächst an einem Beispiel demonstriert werden.

Beispiel: $m = 2183$. Es gilt $453^2 \equiv 7 \pmod{m}, 1014^2 \equiv 3 \pmod{m}, 209^2 \equiv 21 \pmod{m}$. Keiner der drei Reste liefert ein vollständiges Quadrat, aber aus den Faktorzerlegungen können wir $x = 453 \cdot 1051 \cdot 209 \equiv 687 \pmod{m}$ und $y = 3 \cdot 7$ kombinieren, so dass $x^2 \equiv y^2 \pmod{m}$ gilt.

Generell interessieren wir uns nur für solche x, für welche $z \equiv x^2 \pmod{m}$ einfach zu faktorisieren ist. Aus den so gewonnenen Faktorisierungen versuchen wir, durch Produktbildung ein vollständiges Quadrat zusammenzustellen.

Die Faktorisierung wird dabei bzgl. einer vorab berechneten Liste $B = (p_1, \ldots, p_h)$ von Primzahlen, der *Faktorbasis*, ausgeführt und alle Zahlen, die sich nicht vollständig in Faktoren aus der Faktorbasis zerlegen lassen, werden nicht weiter betrachtet. Aus Effizienzgründen wird dabei mit dem symmetrischen Restesystem $z \in \{-\frac{m-1}{2}, \ldots, \frac{m-1}{2}\}$ gearbeitet, so dass bei der Faktorzerlegung auch das Vorzeichen zu berücksichtigen ist.

Mit der folgenden Routine wird für eine Zahl $z \in \mathbb{Z}$ das Vorzeichen sowie die Exponenten der Faktorzerlegung extrahiert, wenn eine solche nur Faktoren aus B enthält. Derartige Zahlen werden auch als *B-Zahlen* bezeichnet.

```
getExponents:=proc(z,FactorBase) local i,p,l;
begin
  if z<0 then l:=1; z:=-z else l:=0 end_if;
  for p in FactorBase do
    i:=0;
    while (z mod p=0) do i:=i+1; z:=z/p end_while;
    l:=l,i;
```

```
  end_for;
  if z <> 1 then FAILED else [l] end_if;
end_proc;
```

Untersuchen wir die Zahl $m = 394663$, indem wir für eine Reihe von x in der Nähe von $\sqrt{m}$ die Faktorzerlegung von $z \equiv x^2 \pmod{m}$ bzgl. des symmetrischen Restesystems zu finden. In der Nähe von $\sqrt{m}$ gilt $z = x^2 - m$.

```
B:=select([i$i=1..50],isprime);
/* Exponentenvektoren verschiedener x^2-m erzeugen */
m0:=isqrt(m);
l:=[i$i=m0-50..m0+50]; /* x-Liste */
l:=select(map(l,x->[x,getExponents(x^2-m,B)]), x->x[2]<>FAILED);
```

Die Liste l enthält Paare (x_i, v_i), wobei v_i der Exponentenvektor der Zerlegung von $z_i = x_i^2 - m$ ist. In die Liste sind nur solche Werte x_i aufgenommen, für die z_i eine B-Zahl ist.

```
[[587, [1, 1, 2, 0, 0, 2, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0]],
   [601, [1, 1, 2, 0, 0, 1, 2, 0, 0, 0, 0, 0, 0, 0, 0]],
   [605, [1, 1, 2, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0]],
   [609, [1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1]],
   [623, [1, 1, 3, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0]],
   [628, [1, 0, 2, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0]],
   [632, [0, 0, 2, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0]],
   [634, [0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0]],
   [642, [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0]],
   [653, [0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0]],
   [656, [0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1]]]
```

Aus der Zerlegung $632^2 - m = 3^2 \cdot 23^2$ können wir sofort $x = 632, y = 46$ und $\gcd(632-46, m) = 563$ als nicht trivialen Teiler von m ablesen. Aber auch aus den Zerlegungen

$$\begin{aligned} 601^2 - m &= -2 \cdot 3^2 \cdot 11 \cdot 13^2 \\ 605^2 - m &= -2 \cdot 3^2 \cdot 37 \cdot 43 \\ 642^2 - m &= 11 \cdot 37 \cdot 43 \end{aligned}$$

können wir $x = 601 \cdot 605 \cdot 642 \equiv 188577 \pmod{m}$, $y = -2 \cdot 3^2 \cdot 11 \cdot 13 \cdot 37 \cdot 43 \equiv 148604 \pmod{m}$ und $\gcd(188577 - 148604, m) = 563$ als nicht trivialen Teiler von m ablesen.

Jede solche Kombination entspricht einer ganzzahligen Linearkombination der Exponentenvektoren v_i der einzelnen x-Werte, in der alle Einträge gerade sind. Um solche Kombinationen zu finden können wir die nicht trivialen Lösungen eines homogenen linearen Gleichungssystems über $\mathbb{Z}_2$ bestimmen. Dazu stellen wir aus den Exponentenvektoren die Koeffizientenmatrix zusammen und berechnen für letztere eine Basis N des Nullraums.

```
M:=Dom::Matrix(Dom::IntegerMod(2))(map(l,x->x[2]));
N:=linalg::nullspace(linalg::transpose(M));
```

In unserem Beispiel ist dieser Nullraum dreidimensional und aus jedem Vektor $n \in N$ dieser Basis des Nullraums können wir über die Liste l und die Faktorbasis B Paare $(x, y) \in \mathbb{Z}_m^2$ mit $x^2 \equiv y^2 \pmod{m}$ konstruieren und $\gcd(x - y, m)$ als möglichen nicht trivialen Faktor berechnen. Dazu werden die x_i sowie die Exponentenvektoren v_i für jeden Eintrag $n_i = 1$ kumuliert. Der kumulierte Exponentenvektor enthält nur gerade Einträge, so dass wir durch 2 teilen können, was den Exponentenvektor von b ergibt. Aus letzterem und der Faktorbasis kann schließlich b selbst berechnet werden.

```
qsTest:=proc(m,FactorBase,l,n) local a,b,i,x,y;
begin
  x:=1; y:=[0$i=0..nops(FactorBase)];
  for i from 1 to nops(l) do
    if not iszero(n[i]) then
      x:=x*l[i][1]; y:=zip(y,l[i][2],_plus)
    end_if
  end_for;
  y:=map(y,i->i/2);
  y:=_mult(op(zip([-1,op(FactorBase)],y,(a,b)->a^b)));
  igcd(x-y,m)
end_proc;
```

Wenden wir diese Funktion auf unsere Nullraumbasis N an, so sehen wir, dass wir in allen drei Fällen einen nicht trivialen Teiler von m finden.

```
map(N,n->qsTest(m,B,l,n));
```

$$[563, 701, 701]$$

Hier ist noch ein komplexeres Beispiel mit einer größeren Zahl m:

```
m:=774419;
B:=select([i$i=1..50],isprime);
m0:=isqrt(m);
l:=[i$i=m0-100..m0+100]: /* x-Liste */
l:=select(map(l,x->[x,getExponents(mods(x^2,m),B)]), x->x[2]<>FAILED);
```

Die Faktorbasis B enthält (mit Vorzeichenfeld) 15 Elemente, die Liste l nur 9, so dass eigentlich mit nicht trivialen Lösungen nicht zu rechnen ist. Aber die Primfaktoren $3, 11, 29, 41, 43$ kommen in keiner Zerlegung eines der $x_i^2 - m$ vor.

```
M:=Dom::Matrix(Dom::IntegerMod(2))(map(l,x->x[2]));
N:=linalg::nullspace(linalg::transpose(M));

map(N,n->qsTest(m,B,l,n));
```

Der Nullraum ist eindimensional liefert den nicht trivialen Teiler 16477 von m.

Primfaktoren, die in Zerlegungen von $x^2 - m$ nicht auftreten können, lassen sich systematisch finden. Ist nämlich $p \,|\, x^2 - m$ ein Primteiler, so gilt $m \equiv x^2 \pmod{p}$ und m muss ein quadratischer Rest $\pmod{p}$ sein. Bei der Aufstellung der Faktorbasis können wir also alle Faktoren p außer Betracht lassen, deren Jacobisymbol $\left(\frac{m}{p}\right) = -1$ ist.

```
factorBase:=proc(m,len) local x;
begin select([$2..len],x->isprime(x) and numlib::jacobi(m,x)=1) end_proc;
```

Damit verringert sich die Zahl der Primzahlen in der Faktorbasis etwa um den Faktor 2, was auf die folgenden (groben) Laufzeitaussagen keinen Einfluss hat, jedoch praktisch wichtig ist.

Die Umsetzung der einen oder anderen Variante dieser Idee geht bis auf die Arbeiten von Brillhart und Morrison (1975) zurück, die mit der Kettenbruchmethode erstmals einen Faktorisierungsalgorithmus mit subexponentieller Laufzeit vorschlugen. Die folgende Variante wurde 1982 von C. Pomerance vorgeschlagen: Wähle eine Faktorbasis B und suche im Bereich um $\sqrt{m}$ so lange Werte x_i, bis (entweder $\gcd(x_i, m) > 1$ ist oder) $|B|+2$ B-Zahlen $z_i = x_i^2 - m$ gefunden sind. Dann hat das lineare Gleichungssystem M mehr Variablen als Gleichungen und so garantiert nicht triviale Lösungen. Die Wahrscheinlichkeit, dass für ein so gefundenes Paar (x, y) noch $x \not\equiv \pm y \pmod{m}$ gilt, ist $\frac{2}{2^t} \leq \frac{1}{2}$, wenn m in t Faktoren zerfällt.

```
getQSFactor:=proc(m,len) local B,g,n,c,r0,x1,x2,v,l,M,N;
begin

/* (1) Aufstellen der Faktorbasis */
  B:=factorBase(m,len);

/* (2) Aufbau der x-Liste */
  n:=nops(B)+2; c:=0; l:=null(); r0:=isqrt(m);
  while (n>0) do
    c:=c+1; x1:=r0+c; x2:=r0-c;
    g:=igcd(x1*x2,m); if g<>1 then return(g) end_if;
    v:=getExponents(x1^2-m,B);
    if v<>FAILED then l:=l,[x1,v]; n:=n-1; end_if;
    v:=getExponents(x2^2-m,B);
    if v<>FAILED then l:=l,[x2,v]; n:=n-1; end_if;
  end_while;
  l:=[l];
  print(Unquoted,
    "x-Liste aus ".nops(l)." Elementen nach ".c." Versuchen aufgebaut");

/* (3) Nullraum der Exponentenmatrix (mod 2) bestimmen */
  M:=Dom::Matrix(Dom::IntegerMod(2))(map(l,x->x[2]));
  N:=linalg::nullspace(linalg::transpose(M));
  print(Unquoted,"Nullraum hat Dimension ".nops(N));

/* (4) Auswertung */
  for x in N do
    n:=qsTest(m,B,l,x); if (1<n) and (n<m) then return(n) end_if;
  end_for;
  error("Kein echter Teiler von ".m." gefunden");
end_proc;
```

Für kleine Faktorbasen wird der Anteil der B-Zahlen im Schritt (2) gering sein, für große Faktorbasen sind dagegen die Rechnungen in einem Durchlauf der Schleife (2) teuer. Die folgende Effizienzanalyse gibt uns den Wert b für ein Trade-off zwischen beiden Effekten.

Ist $B = \{p \in \mathbb{P}, p \leq b\}$, $h = |B| \sim \frac{b}{\ln(b)}$ die Anzahl der Elemente in der Faktorbasis und $n = \ln(m) \sim l(m)$, so erhalten wir folgende Kosten für die einzelnen Schritte von `getQSFactor`:

- $b \cdot \ln(\ln(b)) = \tilde{O}(b)$ für die Berechnung der Faktorbasis mit dem Sieb des Eratostenes im Schritt (1),
- $h\,O(n^2)$ für einen Durchlauf der Schleife (2), also den Gesamtaufwand $O(k\,h^2\,n^2) = \tilde{O}(k\,h^2)$, wenn k die durchschnittlich erforderliche Zahl von Durchläufen bezeichnet, bis eine B-Zahl gefunden wurde,
- $O(h^3)$ für die Bestimmung einer Basis des Nullraums in (3) und
- $O(h\,n^2)$ für die Untersuchung eines der Nullvektoren $n \in N$ (die im Allgemeinen bereits einen nicht trivialen Teiler von m liefert).

Die Gesamtkosten sind also von der Größenordnung $\tilde{O}(b^3 + k\,b^2)$ und wir wollen nun ein gutes Trade-off zwischen b und k bestimmen.

Dazu müssen wir zunächst eine Abschätzung für k finden. Wir wollen davon ausgehen, dass die B-Zahlen $x^2 - m$ für $x \in [1 \ldots (m-1)]$ einigermaßen gleichverteilt sind, was so nicht stimmt, denn in der Nähe von $x = \sqrt{m}$ ist $x^2 - m$ betragsmäßig klein und eher mit B-Zahlen zu rechnen. Aber das wirkt sich eher günstig auf die Laufzeit von `getQSFactor` gegenüber unserer Annahme aus.

Wir beschränken uns auf die Betrachtung des Falls, dass $m = q_1 \cdot \ldots \cdot q_t$ in paarweise verschiedene Primfaktoren zerfällt. Sei p_h der größte Primfaktor aus B und $r \in \mathbb{N}$ so gewählt, dass $p_h^{2r} \leq m$ gilt. Sei S die Menge der in (2) „nützlichen" Zahlen, also

$$S = \left\{x \in \mathbb{N} : 1 \leq x < m \text{ und } x^2 \pmod{m} \text{ ist eine B-Zahl}\right\}.$$

Wegen $k = \frac{\phi(m)}{|S|} < \frac{m}{|S|}$ wollen wir $|S|$ nach unten abschätzen.

Für $a \in \mathbb{Z}_m^*$ sei $\chi_i(a) = \left(\frac{a}{q_i}\right) \in \{+1, -1\}$ und $\chi(a) = (\chi_i(a))_{i=1,\ldots,t} \in \{+1, -1\}^t = G$. $\chi_i(a)$ gibt an, ob $a \pmod{q_i}$ quadratischer Rest ist oder nicht, und $\chi(a)$ fasst diese Informationen in einem Vektor zur *QR-Signatur* von a zusammen.

$\chi : \mathbb{Z}_m^* \to G$ ist ein Gruppenhomomorphismus und nach dem Chinesischen Restklassensatz besteht $Q = Ker(\chi)$ genau aus den quadratischen Resten in $\mathbb{Z}_m^*$. Aus demselben Grund gibt es zu jedem $a \in Q$ genau 2^t Restklassen $b \in \mathbb{Z}_m^*$ mit $b^2 \equiv a \pmod{m}$.

Sei nun für $s \leq 2r$

$$B_s = \left\{a = \prod_{p \in B} p^{e_p} : \sum_{p \in B} e_p = s\right\}$$

die Menge aller B-Zahlen, die in genau s Faktoren zerfallen. Wegen $\gcd(p, m) = 1$ für $p \in B$ und $p_h^{2r} < m$ ist $B_s \subseteq \mathbb{Z}_m^*$.

Betrachten wir weiter für $g \in G$ die Menge $U_g = B_r \cap \chi^{-1}(g)$ der Elemente aus B_r mit vorgegebener QR-Signatur und die Multiplikationsabbildung $\mu : \mathbb{Z}_m^* \times \mathbb{Z}_m^* \to \mathbb{Z}_m^*$ via $\mu(b, c) = b \cdot c \pmod{m}$. Da $g^2 = e$ für alle Elemente $g \in G$ gilt, bildet μ die Mengen $U_g \times U_g$ in $B_{2r} \cap Q$ ab. Sei also

$$V = \mu\left(\bigcup_{g \in G} (U_g \times U_g)\right) \subseteq B_{2r} \cap Q\,.$$

Jedes $a \in B_{2r} \cap Q$ hat genau 2^t Quadratwurzeln in $\mathbb{Z}_m^*$ und diese liegen alle in S. Damit gilt

$$|S| \geq 2^t\ |B_{2r} \cap Q| \geq 2^t\,|V|.$$

Zur Abschätzung von $|V|$ untersuchen wir, wie viele $(b, c) \in \bigcup_{g \in G} U_g \times U_g$ auf dasselbe $a \in V$ abgebildet werden. Wegen $b\,c \equiv a \pmod{m}$ und $b\,c < m, a < m$ gilt sogar $b\,c = a$. Die gesuchte Anzahl ist gleich der Zahl der möglichen Zerlegungen der $2r$ Primfaktoren von a in zwei Gruppen von jeweils r Primfaktoren, also höchstens $\binom{2r}{r}$:

$$\binom{2r}{r}|V| \geq \sum_{g \in G} |U_g|^2.$$

Die Cauchy-Schwarzsche Ungleichung ergibt mit $|G| = 2^t$

$$2^t \left(\sum_{g \in G} |U_g|^2\right) = (1^2 + \cdots + 1^2)\left(\sum_{g \in G} |U_g|^2\right) \geq \left(\sum_{g \in G} 1 \cdot |U_g|\right)^2 = |B_r|^2$$

Mit $|B_r| = \binom{h+r-1}{r} \geq \frac{h^r}{r!}$ ergibt sich schließlich

$$|S| \geq 2^t\,|V| \geq \left(\frac{h^r}{r!}\right)^2 \frac{(r!)^2}{(2r)!} = \frac{h^{2r}}{(2r)!}$$

und damit für die durchschnittliche Anzahl k der Durchläufe von (2) pro B-Zahl

$$k \leq \frac{m}{|S|} \leq \frac{m\,(2r)!}{h^{2r}}\,.$$

Fixieren wir nun r, so sichert die Wahl $b = m^{1/(2r)}$, dass $p_h^{2r} \leq b^{2r} = m$ gilt. Mit $h \geq \frac{b}{\ln(b)}$ nach dem Primzahlverteilungssatz, $n = \ln(m) = 2r\,\ln(b)$ und $(2r)! < (2r)^{2r}$ ergibt sich

$$k \leq m\left(\frac{2r\,\ln(b)}{b}\right)^{2r} = m\left(\frac{n}{b}\right)^{2r} = n^{2r}\,.$$

Wir wählen nun r so, dass b und k etwa die gleiche Größenordnung haben. Aus $\ln(b) = \frac{n}{2r} = \ln(n^{2r}) = 2r\,\ln(n)$ erhalten wir $r = \frac{1}{2}\sqrt{\frac{n}{\ln(n)}}$ und damit $b = n^{2r} = e^{\sqrt{n\ln(n)}}$. Eine genauere Analyse zeigt, dass der Wert $b = e^{\frac{1}{2}\sqrt{n\ln(n)}}$ noch angemessener ist.

Dies begründet zugleich, warum $x^2 - m$ stets der kleinste symmetrische Rest $\pmod{m}$ ist: Es werden nur $h \cdot k \sim e^{2\sqrt{n\ln(n)}} \ll \sqrt{m} = e^{\frac{1}{2}n}$ solche Werte überhaupt durchlaufen.

Damit bekommen wir folgenden QS-Faktorisierungsalgorithmus

```
splitQSFactor:=proc(m)
begin getQSFactor(m,floor(exp(sqrt(ln(m)*ln(ln(m))))/2)) end_proc;

QSFactor:=proc(m:DOM_INT) begin FactorB(m,splitQSFactor) end_proc;
```

Mit $n = \ln(m)$ und $b = e^{\sqrt{n\ln(n)}}$ erhalten wir als Laufzeit für diese Variante des quadratischen Siebs

$$C_{\text{QSFactor}}(m) \in \tilde{O}\left(e^{3\sqrt{n\,\ln(n)}}\right)\,,$$

also bereits subexponentielle Laufzeit. Allerdings kommen diese Vorteile für kleine Zahlen von etwa 20 Stellen noch nicht zum Tragen. Der Flaschenhals der MuPAD-Implementierung ist die Bestimmung einer Basis des Nullraums N. Da die Matrix M dünn besetzt ist, können hierfür spezielle Verfahren (Wiedemann-Algorithmus) angewendet werden, der nur eine Laufzeit $O(h^2)$ hat. Außerdem ist eine spezielle Implementierung, welche die Laufzeitvorteile der Rechnungen über $\mathbb{Z}_2$ ausnutzt, angezeigt.

5.7 Pollards $(p-1)$-Methode

Dieser bereits früher von Pollard vorgeschlagene Algorithmus beruht wieder auf dem kleinen Satz von Fermat. Ist p ein Primteiler von m und $a \in \mathbb{Z}_p^*$, so gilt $a^{p-1} \equiv 1 \pmod{p}$ und somit $p \mid \gcd(a^{p-1} - 1, m)$. Ist q ein anderer Primteiler von m, so ist es sehr unwahrscheinlich, dass $a^{p-1} \equiv 1 \pmod{q}$ gilt (anderenfalls müsste $p-1$ ein Vielfaches der Ordnung von $a \in \mathbb{Z}_q^*$ sein). Dann ist aber $\gcd(a^{p-1}-1, m)$ ein echter Teiler von m.

Dasselbe Argument funktioniert nicht nur für $p-1$, sondern für jedes Vielfache der Ordnung $o = \operatorname{ord}(a \in \mathbb{Z}_p^*)$: $p \mid \gcd(a^{c\,o} - 1, m)$ und $\gcd(a^{c\,o} - 1, m)$ ist mit großer Wahrscheinlichkeit ein echter Teiler von m. Allerdings kennen wir weder p noch o.

Der Kern dieser Idee ist die folgende Beobachtung: Ist $\pi : \mathbb{Z}_m^* \to \mathbb{Z}_p^*$ die natürliche Abbildung, so ist für $z \in \mathbb{Z}_m^*$ die Ordnung $o = \operatorname{ord}(\pi(z) \in \mathbb{Z}_p^*)$ ein (im Allgemeinen echter) Teiler der Ordnung $\operatorname{ord}(z \in \mathbb{Z}_m^*)$. Also wird man beim Scannen von Elementen z^k auf solche stoßen, für welche $\pi(z^k) = 1$, aber $z^k \neq 1$ ist. Aus einem solchen Element lassen sich Rückschlüsse auf Faktoren von m gewinnen.

Diese Idee lässt sich auf andere Gruppen G übertragen, für welche es „modulare" Varianten G_m und eine Abbildung $\pi : G_m \to G_p$ gibt und ist z.B. die Grundlage für die Faktorisierungsalgorithmen auf der Basis elliptischer Kurven.

Praktisch bestimmen wir $\gcd(a^{k!}-1,m)$ für wachsende k, wobei wir natürlich $a^{k!}-1 \pmod m$ berechnen. Ist k wenigstens so groß wie der größte Primteiler von o, dann ist $k!$ ein Vielfaches von o und $p \mid \gcd(a^{k!}-1,m)$.

```
splitPollardpm1Factor:=proc(m:DOM_INT) local D,a,g,r,i,l;
begin
  D:=Dom::IntegerMod(m);
  r:=random(m);
  for l from 1 to 100 do /* maximal 100 Versuche */
    a:=D(r());
    for i from 1 to 10^5 do /* Exponent < 10^5 ! */
      a:=a^i; g:=igcd(expr(a)-1,m);
      if g>1 then
        if g=m then break; /* gcd kein echter Teiler von m */
        else return(g)
        end_if;
      end_if;
    end_for;
  end_for;
  error("Faktorisierung von ".m." fehlgeschlagen");
end_proc;
```

Ein Resultat der Zahlentheorie besagt, dass der größte Primteiler einer Zahl o im Durchschnitt die Größe o^α mit $\alpha = 1 - 1/e \sim 0.632$ hat. Ist also r wieder der kleinste Primfaktor von m, so wird im Durchschnitt nach höchstens $k \sim r^{0.6}$ Durchläufen die innere Schleife mit einem nicht trivialen gcd verlassen. Die Pollardsche $(p-1)$-Methode ist also ein Faktorisierungsverfahren erster Art, das mit einer Laufzeit von $\tilde{O}(r^{0.6})$ nur knapp schlechter als die Pollarsche Rho-Methode ist.

Das wird auch durch praktische Experimente bestätigt. In der folgenden Tabelle sind wieder die Laufzeiten für beide Verfahren und $M_p = 2^p - 1$ unter MuPAD 2.5 zusammengestellt (t_1 = `pollardpm1Factor`, t_2 = `pollardRhoFactor`, Zeitangaben in ms.).

p	11	23	29	37	41	43	47	53	59	67	71	73	79	83	97
t_1	0	3	108	5	19	18	12	213	9	364	1181	111	8972	13	16
t_2	0	1	5	2	14	4	5	32	52	836	130	200	422	3	6

5.8 Faktorisierung ganzer Zahlen in den großen CAS

Zum Faktorisieren sehr großer Zahlen sind nicht nur gute Faktorisierungsalgorithmen erforderlich, sondern auch eine leistungsfähige Langzahlarithmetik und sehr hohe Rechenleistungen, die nur in verteilten Anwendungen zur Verfügung stehen. Mit den großen Faktorisierungsprojekten wie etwa dem Cunningham-Projekt oder GIMPS, dem „Great Internet Mersenne Prime Search“, haben wir es also – für das symbolische Rechnen nicht ungewöhnlich – mit Anwendungen zu tun, mit denen die Leistungsfähigkeit nicht nur moderner Rechentechnik, sondern auch moderner Informatikkonzepte aus verschiedenen Gebieten einem nicht trivialen Test unterzogen werden können.

Für Standard-Anwendungen reicht es dagegen meist aus, auf Faktorisierungsverfahren wie etwa das Pollardsche Rho-Verfahren oder Verfahren der ersten Art mit subexponentieller Laufzeit zurückzugreifen. Dabei wird vor allem die von Brillhard und Morrison 1975 vorgeschlagene Kettenbruchmethode eingesetzt, welche die Idee des quadratischen Siebs mit der Eigenschaft von periodischen Kettenbrüchen verbindet, besonders gute rationale Näherungen für Quadratwurzeln zu liefern. In praktischen Anwendungen lohnt es sich, in kniffligen Fällen auch verschiedene Verfahren zu probieren, da sich die Laufzeiten für einzelne Beispiele sehr unterscheiden können.

In den gängigen CAS stehen standardmäßig gut getunte klassische Verfahren zur Verfügung. Modernere Verfahren sind oft über spezielle Pakete zugänglich, die von einschlägigen Experten für

Forschungszwecke erstellt wurden und der Allgemeinheit über die entsprechenden Verteiler zur Verfügung stehen. So lesen wir etwa in der MAPLE-Dokumentation von `ifactor`:

```
Calling Sequence:
   ifactor(n)
   ifactor(n, method)
```

- If a second parameter is specified, the named method will be used when the front-end code fails to achieve the factorization. By default, the Morrison-Brillhart algorithm is used as the base method. Currently accepted names are:

  'squfof' - D. Shanks' undocumented square-free factorization;
  'pollard' - J.M. Pollard's rho method;
  'lenstra' - Lenstra's elliptic curve method; and
  'easy' - which does no further work.

- If the 'easy' option is chosen, the result of the ifactor call will be a product of the factors that were easy to compute, and a name `_c.m._.n` indicating an m-digit composite number that was not factored where the n is an integer which preserves (but does not imply) the uniqueness of this composite.

- The Pollard base method accepts an additional optional integer: `ifactor(n,pollard,k)`, which increases the efficiency of the method when one of the factors is of the form $k*m+1$.

`ifactor` in MUPAD benutzt die elliptische Kurvenmethode, eines der Verfahren, das wie Pollards $(p-1)$-Methode funktioniert, aber eine andere Gruppe verwendet.

Im MATHEMATICA-Handbuch heißt es

> `FactorInteger` switches between removing small primes by trial division and using the Pollard $(p-1)$, Pollard rho and continued fraction algorithm.

Im Paket `NumberTheory‘FactorIntegerECM‘` wird die elliptische Kurvenmethode als zusätzliche Möglichkeit angeboten, um besonders hartnäckige zusammengesetzte Zahlen vielleicht doch noch zu faktorisieren.

5.9 Mersennezahlen und der Lucas-Test

Besonders viel Bewegung in der „Primzahl-Szene“ war in den letzten Jahren bei den Mersennezahlen $M_q = 2^q - 1,\ q \in \mathbb{P}$ zu verzeichnen. Mersenne behauptete im Jahr 1644, dass M_q für $q \in \{2, 3, 5, 7, 13, 17, 19, 31, 67, 127, 257\}$ prim sei und für keinen anderen Exponenten $q \leq 257$. Mit einem CAS wie etwa MUPAD kann diese Behauptung heute leicht überprüft werden:

```
select([$1..1000],x->isprime(x) and isprime (2^x-1));
```

$$[2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127, 521, 607]$$

Mersennes Liste enthält also 4 Fehler, wobei $q = 67$ gewöhnlich großzügig als Schreibfehler gewertet wird.

Damit sind zugleich die ersten 14 Mersenneschen Primzahlen gefunden. Je größer q wird, desto seltener treten solche Zahlen auf. So gibt es im Bereich $128 \leq q \leq 1000$ nur noch zwei Stück. Diese wurden erst 1952 von R. M. Robinson entdeckt. Zugleich wird der Rechenaufwand immer größer, da die Stellenzahl von M_q linear mit q wächst, so dass die Suche nach weiteren Mersenneschen Primzahlen eine große Herausforderung an Hard- und Software sowie die Organisation der entsprechenden Rechnungen darstellt.

R.M.Robinson fand im Jahr 1952 auch die Mersenneschen Primzahlen Nummer 15 bis 17: M_{1279} (386 Stellen), M_{2203} (664 Stellen) und M_{2281} (687 Stellen). Weitere Mersennesche Primzahlen wurden in den 80er Jahren vor allem von D. Slowinski und seinen Mitstreitern gefunden, so etwa die 30. Mersennesche Primzahl $M_{132\,049}$ (39 751 Stellen). Allerdings wurde diese Zahl erst später als Nummer 30 identifiziert, denn man hatte die Primzahl $M_{110\,503}$ übersehen.

In den 90er Jahren wurde, mit der zunehmenden Leistunsgfähigkeit der Rechentechnik und insbesondere der Möglichkeit, stabile verteilte Rechnungen in lose gekoppelten Netzwerken zu organisieren, eine ganze Reihe neuer Mersennescher Primzahlen entdeckt. Während [6] in der 3. Auflage seines Buchs (1996) noch die 33. Mersennesche Primzahl $M_{859\,433}$ mit 258 716 Stellen als größte bekannte Primzahl nennt, wurde im letzten Jahr (12. Dezember 2003) bereits $M_{20\,996\,011}$ mit 6 320 430 Stellen als die 40. Mersennesche Primzahl identifiziert.

Es bedarf einer ausgefeilten Arithmetik und guter Algorithmen, um mit solchen Riesenzahlen zu rechnen. Die letzten Mersenneschen Primzahlen wurden allen im Rahmen eines großen Projekts zum verteilten Rechnen, dem GIMPS-Projekt (Great Internet Mersenne Primes Search, siehe `http://www.mersenne.org`) gefunden. Mehr dazu auch auf den Webseiten von Chris Caldwell unter `http://www.utm.edu/research/primes`.

Trotz Einsatz einer schnellen Arithmetik auf der Basis der FFT, in der Multiplikationen und Divisionen von n-stelligen Zahlen in der Laufzeit $O(n\log(n)\log(\log(n)))$ (statt $O(n^2)$ für das klassische Verfahren) ausgeführt werden, lohnt es, bei solchen Zahlenriesen vor einem der „schnellen“ Primzahltests, etwa dem Fermat-Test, Teilbarkeitstests durch kleine Primfaktoren zu versuchen. Schließlich ist die Laufzeit des Fermat-Tests für n-stellige Zahlen (klassiche Arithmetik) von der Größenordnung $O(n^3)$, während eine Teilbarkeitsuntersuchung durch eine k-stellige Zahl in der Laufzeit $O(k\,n)$ ausgeführt werden kann.

Deshalb werden bei der Untersuchung, ob $m = 2^q - 1$ eine Primzahl ist, zunächst Teilbarkeitsuntersuchungen mit einer Variante des Siebs des Eratostenes sowie einer auf den Spezialfall zugeschnittenen Variante der Pollardschen $(p-1)$-Methode eingesetzt, um auf diese Weise Mersennezahlen M_q mit „kleinen“ Primteilern zu identifizieren und auszusortieren. Da für Teiler $r \mid M_q$ stets $r \equiv 1 \pmod q$ (Übungsaufgabe 1 d) und sogar $r \equiv 1 \pmod{2q}$ gilt, kann man sich bei diesen Teilbarkeitsuntersuchungen zudem auf ausgewählte Primteiler beschränken.

Danach wird ein harter Primzahltest angewendet, der zugleich Beweiskraft hat, d.h. ein Primzahlzertifikat erstellt. Allerdings ist dafür das weiter oben vorgestellte Verfahren wenig geeignet, da für die Erstellung des Zertifikats die Faktorzerlegung von $m - 1 = 2^q - 2$ gefunden werden muss. Da statt dessen die Faktorzerlegung von $m + 1 = 2^q$ offensichtlich ist, wird ein Primzahltest samt Zertifikat eingesetzt, der statt $\mathbb{Z}_m^*$ eine Gruppe G_m verwendet, die genau für prime m zyklisch der Ordnung $m + 1$ ist.

In diesem Abschnitt wollen wir uns näher mit diesem Ansatz befassen, der auf Lucas-Folgen und Rechnungen in quadratischen Erweiterungen des Körpers $\mathbb{Q}$ aufsetzt. Er führt auf das folgende Kriterium, mit dem Mersennesche Primzahlen sicher identifiziert werden können:

Satz 24 (Lucas-Lehmer-Test, 1878, 1930/35) *Die Mersennezahl $m = 2^q - 1$ (q prim) ist genau dann eine Primzahl, wenn $l_{m-1} \equiv 0 \pmod m$ gilt, wobei l_k die durch die Rekursionsbeziehung $l_k = l_{k-1}^2 - 2$ und den Startwert $l_1 = 4$ definierte Zahlenfolge ist.*

Nullstellen quadratischer Polynome und Lucas-Folgen

Sei $f = x^2 - P\,x + Q \in \mathbb{Z}[x]$ ein Polynom, $\Delta = P^2 - 4\,Q$ dessen Diskriminante und

$$\alpha_{1,2} = \frac{P \pm \sqrt{\Delta}}{2}$$

die beiden Nullstellen von f. Dann gilt

$$\alpha_1 + \alpha_2 = P, \qquad \alpha_1 \cdot \alpha_2 = Q, \qquad \alpha_1 - \alpha_2 = \sqrt{\Delta}\,.$$

Eng verbunden mit f ist die zweistufige lineare Rekursionsbeziehung $z_{i+1} = P \cdot z_i - Q \cdot z_{i-1}$. Sie definiert die Folge $\{z_i\}_{i \geq 0}$ eindeutig, wenn die beiden Startwerte z_0 und z_1 vorgegeben werden. Eine solche Folge wird als *Lucas-Folge* zu f bezeichnet. Ist $\Delta \neq 0$ und damit $\alpha_1 \neq \alpha_2$, so lassen sich die Folgenwerte z_n durch eine geschlossene Formel

$$z_n = C_1 \cdot \alpha_1^n + C_2 \cdot \alpha_2^n, \quad n \geq 0$$

mit geeigneten Werten C_1, C_2 darstellen.

Beispiel: Die Fibonaccifolge $\mathbf{F} = \{0, 1, 1, 2, 3, 5, 8, \ldots\}$ ergibt sich für $f = x^2 - x - 1$, also $\Delta = 5$, und es gilt die Binetsche Formel

$$F_n = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right)$$

Die Werte der Folge $\{z_i\}_{i \geq 0}$ lassen sich über die Matrixbeziehung

$$\begin{pmatrix} z_{n+1} \\ z_n \end{pmatrix} = \begin{pmatrix} P & -Q \\ 1 & 0 \end{pmatrix} \begin{pmatrix} z_n \\ z_{n-1} \end{pmatrix},$$

also

$$\begin{pmatrix} z_{n+1} \\ z_n \end{pmatrix} = \begin{pmatrix} P & -Q \\ 1 & 0 \end{pmatrix}^n \begin{pmatrix} z_1 \\ z_0 \end{pmatrix}$$

effizient berechnen.

Satz 25 *Für ein einzelnes Folgenglied z_n kann dessen Rest $z_n \pmod m$ in der Zeit $O(m^2 \log(n))$ (klassische Multiplikation) berechnet werden.*

Die folgende MuPAD-Prozedur berechnet $z_n \pmod m$

```
LucasFolge:=proc(P,Q,m,n,a1,a0) local u,v,M;
begin
  M:=Dom::Matrix(Dom::IntegerMod(m))([[P,-Q],[1,0]]);
  u:=Dom::Matrix(Dom::IntegerMod(m))([[a1],[a0]]);
  v:=M^n*u;
  expr(v[2]);
end_proc;
```

Umgekehrt kann man auch die Potenzen α_i^n der Nullstellen von f über solche Lucas-Folgen berechnen, denn für

$$\alpha_{1,2}^n = \frac{V_n \pm U_n \sqrt{\Delta}}{2} \qquad \text{mit } U_n = \frac{1}{\Delta}\left(\alpha_1^n - \alpha_2^n\right) \text{ und } V_n = \alpha_1^n + \alpha_2^n$$

gelten die Beziehungen

$$\begin{aligned} U_{n+1} &= P\,U_n - Q\,U_{n-1}, \quad U_0 = 0, U_1 = 1 \\ V_{n+1} &= P\,V_n - Q\,V_{n-1}, \quad V_0 = 2, V_1 = P \end{aligned}$$

Daraus folgt sofort $U_n, V_n \in \mathbb{Z}$. In Anwendungen spielen diese beiden Folgen $\{U_n\}$ und $\{V_n\}$ eine wichtige Rolle.

```
LucasU:=proc(P,Q,m,n) begin LucasFolge(P,Q,m,n,1,0) end_proc;
LucasV:=proc(P,Q,m,n) begin LucasFolge(P,Q,m,n,P,2) end_proc;
```

Lucas-Folgen und ganze Elemente in $\mathbb{Q}[\sqrt{D}]$

In diesem Abschnitt betrachten wir quadratische Körper-Erweiterungen von $\mathbb{Q}$

$$K = \mathbb{Q}[\sqrt{D}] = \left\{ a + b\sqrt{D} \,:\, a, b \in \mathbb{Q} \right\}.$$

Jede Zahl $z = a + b\sqrt{D} \in K$ ist Nullstelle eines quadratischen Polynoms $f = x^2 - p\,x + q \in \mathbb{Q}[x]$. Die andere Nullstelle dieses Polynoms ist die zu z *konjugierte Zahl* $\overline{z} = a - b\sqrt{D}$, so dass $f = (x - z)(x - \overline{z})$ und damit $p = z + \overline{z} = 2a$ und $q = z \cdot \overline{z} = a^2 - b^2 D = N(z)$ gilt. Die Abbildung $N : K \to \mathbb{Q}$ bezeichnet man auch als die *Normabbildung*. Die Normabbildung ist eine multiplikative Abbildung, d.h. es gilt $N(z_1 \cdot z_2) = N(z_1) \cdot N(z_2)$ und $N(1) = 1$.

Lucas-Folgen stehen in engem Zusammenhang zum Rechnen im Ring $\mathcal{O}$ der ganzen Elemente von K. $z \in K$ heißt *ganz*, wenn es ein Polynom $f = x^2 - Px + Q \in \mathbb{Z}[x]$ mit Nullstelle z gibt. Ist D quadratfrei, lässt sich $\mathcal{O}$ wie folgt beschreiben:

$$\mathcal{O} = \begin{cases} \left\{ \frac{v+u\sqrt{D}}{2} \,:\, u, v \in \mathbb{Z}, u + v \text{ gerade} \right\} & \text{wenn } D \equiv 1 \pmod 4 \\ \\ \left\{ v + u\sqrt{D} \,:\, u, v \in \mathbb{Z} \right\} & \text{wenn } D \equiv -1 \pmod 4 \end{cases}$$

Sei $z = \frac{v+u\sqrt{D}}{2} \in \mathcal{O}$ und $f = x^2 - P\,x + Q$ das (eindeutig bestimmte) monische Polynom mit $f(z) = 0$. Dann sind $P = v, Q = N(z)$ und $\Delta = P^2 - 4\,Q = u^2 D$ ganzzahlig und die Folgen (U_k) und (V_k) in der Darstellung

$$z^k = \frac{V_k + U_k\sqrt{D}}{2}$$

sind Lucas-Folgen und damit selbst ganzzahlige Folgen. Wir schreiben auch $U_k(z)$ und $V_k(z)$, wenn die Beziehung zu z hervorgehoben werden soll.

Für ungerades n ist der Nenner 2 invertierbar und wir können auf $\mathcal{O}$ die Kongruenzrelation

$$z_1 = a_1 + b_1\sqrt{D} \equiv z_2 = a_2 + b_2\sqrt{D} \pmod n \Leftrightarrow a_1 \equiv a_2 \pmod n \text{ und } b_1 \equiv b_2 \pmod n$$

und den Restklassenring

$$\mathcal{O}_n = \left\{ v + u\sqrt{D} \,:\, u, v \in \mathbb{Z}_n \right\}$$

definieren[1]. Viele Eigenschaften von $\mathbb{Z}_n$ übertragen sich auf diesen Ring, so dass es nicht verwundert, wenn er in Zahlalgorithmen eine wichtige Rolle spielt.

Bestimmen wir zunächst in Analogie zu $\mathbb{Z}_n^*$ die Gruppe $\mathcal{O}_n^*$ der invertierbaren Elemente dieses Rings. Die Norm N induziert eine Normabbildung $N : \mathcal{O}_n \to \mathbb{Z}_n$.

Lemma 3 *Es gilt* $\mathcal{O}_n^* = N^{-1}(\mathbb{Z}_n^*)$.

Beweis : Ist $z = v + u\sqrt{D} \in \mathcal{O}_n$ invertierbar und $z' = v' + u'\sqrt{D}$ eine inverse Restklasse, so gilt $N(z \cdot z') = N(z) \cdot N(z') \equiv 1 \pmod n$, also $N(z) \in \mathbb{Z}_n^*$. Ist umgekehrt $N(z)$ invertierbar $\pmod n$, so rechnet man leicht nach, dass $z' = N(z)^{-1} \cdot \overline{z}$ zu z invers ist. □

Primzahltests und die Gruppe G_n

$\mathbb{Z}_n$ ist als Unterring der Elemente $z = v + u\sqrt{D}$ mit $u = 0$ in $\mathcal{O}_n$ enthalten und damit $\mathbb{Z}_n^*$ auch eine Untergruppe von $\mathcal{O}_n^*$ und wir können die Faktorgruppe $G_n = \mathcal{O}_n^*/\mathbb{Z}_n^*$ betrachten.

[1] Wir betrachten $\mathcal{O}_n$ als Menge formaler Paare (v, u), so dass stets $|\mathcal{O}_n| = n^2$ gilt, unabhängig davon, ob $D \pmod n$ ein quadratischer Rest ist oder nicht.

Ist n prim und $D \pmod n$ kein quadratischer Rest, also $\left(\frac{D}{n}\right) = -1$, so ist $\mathcal{O}_n = K_n = Z_n[\sqrt{D}]$ ein zu $GF(n^2)$ isomorpher Körper und $\mathcal{O}_n^*$ besteht aus den n^2-1 Elementen von K_n, welche verschieden von Null sind. G_n ist dann eine zyklische Gruppe (wie $\mathcal{O}_n^*$) der Ordnung $\frac{n^2-1}{n-1} = n+1$.

Ist dagegen $\left(\frac{D}{n}\right) = 1$, also $D \pmod n$ ein quadratischer Rest, so gilt $\sqrt{D} \in \mathbb{Z}_n$ und damit $\mathbb{Z}_n[\sqrt{D}] = \mathbb{Z}_n$. Die Struktur von $\mathcal{O}_n$ und damit auch von G_n ist in diesem Fall nicht so offensichtlich.

Wir zeigen nun, dass die Existenz eines Elements $z \in G_n$ der Ordnung $n+1$ im Fall $\left(\frac{D}{n}\right) = -1$ auch hinreichend für die Primzahleigenschaft von n ist.

Lemma 4 *Ist $n = p^e$ eine Primzahlpotenz, so gilt*

$$|\mathcal{O}_n^*| = \begin{cases} p^{2e} - p^{2e-2} & \textit{wenn } \left(\frac{D}{p}\right) = -1 \\ p^{2e} - 2\,p^{e-1}(p-1) - p^{2e-2} & \textit{wenn } \left(\frac{D}{p}\right) = 1 \end{cases}$$

und damit

$$|G_n| = p^{e-1}(1+p) - \left(1 + \left(\frac{D}{p}\right)\right)$$

Beweis : Offensichtlich ist

$$|\mathcal{O}_n^*| = p^{2e} - |\{z \in \mathcal{O}_n \,:\, p \,|\, N(z)\}|$$

Zur zweiten Menge gehören in beiden Fällen die $z = v + u\sqrt{D}$ mit $u \equiv v \equiv 0 \pmod p$. Im ersten Fall wurden damit schon alles gezählt. Im zweiten sind auch noch die z, für welche $v \equiv \pm u \cdot a \pmod p$ mit $a^2 \equiv D \pmod p$ und $u \in \mathbb{Z}_n^*$ gilt, zu zählen.

Die Aussage für $|G_n|$ ergibt sich nach Division durch $|\mathbb{Z}_n^*| = p^{e-1}(p-1)$. □

Satz 26 *Sei $n \in \mathbb{N}$ mit $\gcd(2\,D, n) = 1$ und $\left(\frac{D}{n}\right) = -1$.*
n ist genau dann eine Primzahl, wenn G_n eine zyklische Gruppe der Ordnung $n+1$ ist.

Beweis : Weiter oben hatten wir bereits gezeigt, dass für eine Primzahl n die Gruppe G_n zyklisch der Ordnung $n+1$ ist.

Ist n eine reine Primzahlpotenz $n = p^e$ mit $e \geq 2$ und (folglich) $\left(\frac{D}{p}\right) = -1$, so ist $n+1 = p^e+1$ kein Teiler von $|G_n| = p^{e-1}(1+p)$ und G_n kann somit kein Element der Ordnung $n+1$ enthalten.

Ist $n = \prod_{i=1}^t p_i^{e_i}$ zusammengesetzt, so gilt zunächst $G_n \cong \prod_i G_{p_i^{e_i}}$, denn nach dem Chinesischen Restklassensatz gibt es Isomorphismen

$$\mathcal{O}_n^* \xrightarrow{\sim} \prod \mathcal{O}_{p_i^{e_i}}^* \text{ und } \mathbb{Z}_n^* \xrightarrow{\sim} \prod \mathbb{Z}_{p_i^{e_i}}^*.$$

Ähnlich wie im Satz von Carmichael ergibt sich damit für die Exponente

$$exp(G_n) \,|\, \operatorname{lcm}\left(\left|G_{p_i^{e_i}}\right|, i = 1, \dots, t\right).$$

Da alle $\left|G_{p_i^{e_i}}\right|$ gerade sind, unterscheiden sich lcm und Produkt dieser Zahlen wenigstens um einen Faktor 2^{t-1}, was für $t \geq 2$ die Abschätzung

$$exp(G_n) \leq \frac{2}{2^t}\, n \prod_i \left(1 + \frac{1}{p_i}\right) \leq 2\,n\left(\frac{2}{3}\right)^t < n$$

liefert. G_n enthält also auch in diesem Fall kein Element der Ordnung $n+1$. □

In die Sprache der Lucas-Folgen übertragen lauten diese Aussagen

Satz 27 (Lucas-Test und Lucas-Zertifikat)

(1) Ist n prim, so gilt $n \mid U_{n+1}$ für jede Lucas-U-Folge U_i mit Diskriminante Δ, für welche $\gcd(2\Delta, n) = 1$ und $\left(\frac{\Delta}{n}\right) = -1$ erfüllt ist.

(2) Existiert eine Lucas-U-Folge U_i mit Diskriminante Δ, für welche $\gcd(2\,\Delta, n) = 1$ und $\left(\frac{\Delta}{n}\right) = -1$ erfüllt ist und für welche neben $n \mid U_{n+1}$ weiter $U_{\frac{n+1}{p}} \not\equiv 0 \pmod n$ für alle Primteiler $p \mid (n+1)$ gilt, so ist n prim.

Bemerkung: Ist $\left(\frac{\Delta}{n}\right) = +1$ und n prim, so gilt $n \mid U_{n-1}$.
Die erste Eigenschaft kann wieder als Basis für einen LasVegas-Primzahltest verwendet werden, wobei sich durch die Wahl von $[P, Q]$ Adjustierungsmöglichkeiten ergeben:

```
LucasTest:=proc(P,Q,m) local d,e;
begin
  d:=P^2-4*Q;
  e:=numlib::jacobi(d,m);
  if not iszero(LucasU(P,Q,m,m-e) mod m) then return(FALSE) end_if;
  FAIL;
end_proc;
```

Auch hier ist m garantiert zusammengesetzt, wenn für ein Paar $[P, Q]$ $U_{m-e} \not\equiv 0 \pmod m$ gilt. Dieses Paar bezeichnet man auch als **Lucas-Zeugen** für m. Lässt sich trotz intensiver Suche kein solcher Lucas-Zeuge finden, so ist m mit hoher Wahrscheinlichkeit prim.

Die zweite Aussage ist die Basis für folgendes Kriterium, mit dem entschieden werden kann, ob für primes q die Mersennezahl $m = 2^q - 1$ eine Primzahl ist.

Satz 28 *Sei $z = 1 + \sqrt{3}$ und $V_n = V_n(z)$ die zugehörige Lucas-V-Folge. Die Mersennezahl $m = 2^q - 1,\ q \geq 3$, ist genau dann prim, wenn m ein Teiler von $V_{\frac{m+1}{2}}$ ist.*

Beweis : Es gilt wie oben gezeigt

$$z^n = \frac{V_n + U_n\sqrt{12}}{2} \tag{1}$$

für die Lucas-Folgen U_n und V_n mit den Parametern $P = 2,\ Q = -2,\ \Delta = P^2 - 4\,Q = 12$.
Aus $m \equiv (-1)^q - 1 \equiv 1 \pmod 3$ und dem quadratischen Reziprozitätsgesetz folgt

$$\left(\frac{3}{m}\right) = -\left(\frac{m}{3}\right) = -1,$$

so dass wir uns im Fall $\left(\frac{\Delta}{m}\right) = -1$ befinden.
Für Lucas-Folgen gelten folgende Formeln, die sich leicht aus (1) herleiten lassen[2]:

$$\begin{aligned} U_{2n} &= U_n\,V_n && (2)\\ 2\,V_{2n} &= V_n^2 + \Delta\,U_n^2 && (2')\\ 4\,Q^n &= V_n^2 - \Delta\,U_n^2 && (3)\\ V_n^2 &= V_{2n} + 2\,Q^n && (4) \end{aligned}$$

Ist nun m prim, so folgt aus dem Lucas-Kriterium $U_{m+1} \equiv 0 \pmod m$, $U_{\frac{m+1}{2}} \not\equiv 0 \pmod m$ und damit wegen (2) auch $V_{\frac{m+1}{2}} \equiv 0 \pmod m$.

[2] (2) und (2') ergeben sich aus $z^{2n} = (z^n)^2$, (3) aus $N(z^n) = N(z)^n$ und (4) aus (2') und (3).

Gilt umgekehrt $V_{\frac{m+1}{2}} \equiv 0 \pmod m$, so folgt $U_{m+1} \equiv 0 \pmod m$ wieder aus (2), und aus (3) ergibt sich

$$12\, U^2_{\frac{m+1}{2}} + 4\,(-2)^{\frac{m+1}{2}} \equiv 0 \pmod m,$$

woraus sofort $U_{\frac{m+1}{2}} \not\equiv 0 \pmod m$ folgt. $\square$

Der Zusammenhang zum Kriterium von Lucas-Lehmer ergibt sich aus der Beziehung

$$l_k = V_{2^k}/2^{2^{k-1}},$$

welche sich durch vollständige Induktion leicht verifizieren lässt: Es gilt wegen $V_2 = P^2 - 2Q = 8$

$$l_1 = 4 = \frac{V_2}{2}$$

und

$$l_{k+1} = l_k^2 - 2 \overset{IV}{=} \frac{V^2_{2^k}}{2^{2^k}} - 2 \overset{(4)}{=} \frac{V_{2^{k+1}} + 2\,(-2)^{2^k}}{2^{2^k}} - 2 = \frac{V_{2^{k+1}}}{2^{2^k}}\,.$$

Literatur

[1] J. Arndt and C. Haenel. π - *Algorithmen, Computer, Arithmetik*. Springer, Berlin, 2 edition, 2000.

[2] D. Bressoud and S. Wagon. *Computational Number Theory*. Key College Publishing and Springer, New York, 2000.

[3] R. Crandall and C. Pomerance. *Prime Numbers – A Computational Perspective*. Springer, New York, 2001.

[4] D.E. Knuth. *The art of computer programming*. Addison Wesley, 1991.

[5] E. Kunz. *Algebra*. Vieweg, 1991.

[6] P. Ribenboim. *The new book of prime number records*. Springer, New York, 1996.

[7] H. Riesel. *Prime numbers and computer methods for factorization*. Birkhäuser, 1994.

[8] G. Tenenbaum. *The Prime Numbers and Their Distribution*, volume 6 of *AMS Student Mathematical Library*. Amer. Math. Soc., 2001.