# Grundlegende Algorithmen der Computeralgebra

Hans-Gert Gräbe
Institut für Informatik, Universität Leipzig

1. Juli 1998

## 0 Einleitung

Diese Vorlesung gibt, nach der Vorlesung "Einführung in die Computeralgebra", in der vor allem die Einsatzmöglichkeiten und die dabei zu beachtenden Spezifika von Computeralgebrasystemen (CAS) im Mittelpunkt standen, nunmehr einen tieferen Einblick in die Algorithmen, die für die grundlegenden Funktionalitäten des Rechnens mit exakten Zahlen und Polynomen bzw. rationalen Funktionen benötigt werden.

Neben den Algorithmen selbst soll auch deren Laufzeitverhalten untersucht werden, um auf diese Weise grundlegende Vorstellungen auch 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 fundierte Kenntnisse der linearen Algebra, also über Vektorräume und lineare Gleichungssysteme, an vielen Stellen benötigt. Daneben werden auch Grundkenntnisse der höheren Algebra, wie etwa über endliche Körper und das Rechnen in Restklassenringen, als bekannt vorausgesetzt.

Im Zentrum der Vorlesung stehen *praktikable* Verfahren für die verschiedenen grundlegenden algorithmischen Fragestellungen, die sich im symbolischen Rechnen ergeben, wobei die wichtigsten Ideen für laufzeiteffiziente Verfahren einbezogen werden, ohne allerdings bis in die letzten Details einer getrimmten Implementierung oder eines vielleicht theoretisch interessanten, aber praktisch bedeutungslosen Verfahrens zu verzweigen.

Um ein Gefühl dafür zu bekommen, welcher Effizienzzuwachs sich oft bereits aus einfachen Überlegungen ergibt, wollen wir abschließend die folgende Aufgabe untersuchen:

> *Aufgabe: Gegeben sind $x \in R, n \in \mathbf{N}_+$. Man berechne $x^n \in R$ mit möglichst geringem Aufwand.*

Wir betrachten zwei verschiedene Lösungen.

1. Die klassische Iteration:

```
procedure power1(x:R,n:integer):R;
     p:=1;
     for i:=1 to n do p:=p*x;
     return p;
```

Gehen wir davon aus, daß wir nur Multiplikationen (als die in diesem Algorithmus laufzeitbestimmenden Operationen) zählen, und nehmen an, daß für jede Multiplikation eine konstante Zahl von Prozessortakten benötigt wird (etwa beim Potenzieren von `float`- oder `longint`-Zahlen), so ergibt die Zahl der benötigten Multiplikationen ein Maß für die Laufzeit, das hier also die Größenordnung $O(n)$ hat. Eine zweite Lösung erhalten wir mit dem

2. Schnellen Potenzieren (binäres Potenzieren):

```
procedure power2(x:R,n:integer):R;
      if n=0 then 1 else if n=1 then x
      else if odd(n) then x*power2(x*x,n div 2)
      else power2(x*x,n div 2);
```

Die Anzahl der Multiplikationen $C_2(n)$ erfüllt hier die Rekursionsbeziehung $C_2(n) \leq C_2(\frac{n}{2})+2$, was $C_2(n) = O(\log(n))$ liefert. Zur Berechnung von $2^{100}$ benötigen wir nach dem ersten Verfahren 100 Multiplikationen, nach dem zweiten Verfahren nur 8 ! Da $\log(n)$ die *Bitlänge* des Exponenten $n$ angibt, schließen wir, daß das zweite Verfahren linear in dieser Bitlänge ist, das erste dagegen exponentiell (denn $n = 2^{\log(n)}$).

# 1 Rechnen mit ganzen Zahlen – Die Langzahlarithmetik und deren Komplexität

Zahldarstellung in der Basis $\beta$ (meist eine Zweierpotenz):

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

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

## 1.1 Ein- und Ausgabe

Wir haben dafür drei Datentypen zur Verfügung:

ZAHL = Zahlen zur Basis $\beta$, für die man die volle Arithmetik beherrscht.

DIGIT = Elementare Bestandteile von ZAHL.

STRING (als Sequenz of CHAR) = E/A-Format von ZAHL in einem anderen Zahlsystem, zusammen mit zwei Operationen $val : CHAR \longrightarrow DIGIT$ und $symb : DIGIT \longrightarrow CHAR$

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

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

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

folglich gilt

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

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

Beispiel: $[1A2CF]_{16}$ im 16-er-System ist ins Dezimalsystem zu verwandeln. Wir sehen, daß es einer zusätzlichen Übersetzung `val` der Werte der einzelnen Zeichen in Zahlen bedarf, vgl. auch Unterschied zwischen interner Computerdarstellung derselben Ziffer als `int` und als `char`.

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

Beachte, daß wir bei beiden Rechnungen unsere Fähigkeit, im 10-er-System arithmetische Rechnungen ausführen zu können, genutzt haben, während das "fremde" System nur die Rolle eines EIn- bzw. Ausgabepuffers gespielt hat.

```
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;

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; inc(i);
      return [a_m ... a_0]_c;
```

Benötigte Operationen (unter der Annahme $l(\beta) = l(c) = 1$):
$Dmult : (ZAHL, DIGIT) \longrightarrow ZAHL$ und
$Ddivmod : (ZAHL, DIGIT) \longrightarrow (ZAHL, DIGIT)$
Die benötigen ihrerseits die elementaren Operationen
$Emult : (DIGIT, DIGIT) \longrightarrow DoubleDIGIT$ und
$Edivmod : (DoubleDIGIT, DIGIT) \longrightarrow (DIGIT, DIGIT)$
Komplexität: Es werden $C_{Dmult}(z) = C_{Ddivmod}(z) = l(z)$ Multiplikationen benötigt.
$C_{ZAHL-to-STRING}(z) = C_{STRING-to-ZAHL}(z) \sim \frac{1}{2} l(z)^2$
(jeweils $\sum_{i=0}^{m} (i+1) = \frac{(m+1)(m+2)}{2}$)

## 1.2 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 aufwendigsten 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 $\beta^2$ Werte annehmen, wovon in $\beta$ 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. Beispiel $\beta = 7$. $213\,461 * 61\,534 = 16\,606\,345\,064$

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;
```

Für den Beweis der Korrektheit ist zu zeigen, daß $t$ und $r$ die entsprechenden Bereiche DoubleDigit und Digit nicht verlassen, also immer $t < \beta^2$ und $r < \beta$ gilt. Beweis mit Induktion.

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

> AUFGABE: Untersuche, wie groß die Wahrscheinlichkeit ist, daß bei einer Multiplikation im Zahlsystem zur Basis $\beta$ kein Übertrag auftritt. (Untersuche das für einzelne Werte von $\beta$ und zeige, daß dieser Wert die Ordnung $O(\frac{\log(\beta)}{\beta})$ hat.)

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

Hierbei haben wir nur die (in der Regel etwas teureren) Elementarmultiplikationen und -divisionen gezählt.

> AUFGABE: Zeige $C_{mult}(a,b)) = 4\,l(a)\,l(b)$, wenn alle arithmetischen Elementaroperationen zählen.

### 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

$$a \cdot b = (A_1B_1)\beta^{2l} + (A_1B_2 + A_2B_1)\beta^l + (A_2B_2)$$

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

$$(A_1B_2 + A_2B_1) = (A_1 + B_1)(A_2 + B_2) - A_1B_1 - A_2B_2$$

Komplexität: $C_{Karatsuba}(l) = O(l^\alpha)$ mit $\alpha = \frac{\log 3}{\log 2} \approx 1.58$ für Zahlen der Wortlänge $l \geq l(z)$.

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 (Strassen) haben eine Laufzeit von $O(l \log(l) \log\log(l))$, spielen aber in praktischen Anwendungen keine Rolle.

## 1.3 Division mit Rest

Allgemeines Schema

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

Komplexität: $a = q * b + r$, also $l(q) \leq l(a) - l(b) + 1$.

Wenn korrektes Ziffernraten des Quotienten mit konstantem Aufwand $c$ möglich ist, dann gilt

$$C^*_{divmod}(a, b) = l(q) \cdot (l(b) + c) = O((l(a) - l(b)) \cdot l(b))$$

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

> AUFGABE: Zeige $C_{divmod}(a, b) = l(q) \cdot (2\,l(b) + c)$, wenn alle arithmetischen Elementaroperationen zählen.

Allerdings kann man $q_i$ nicht in konstanter Zeit *korrekt* erraten. Bsp: $20\ldots01 : 10\ldots01$. Dieses Problem tritt allerdings nur auf, wenn $l(b) > 0$, was wir im weiteren voraussetzen wollen.

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.

Beispieldivision: $3\,197\,336 : 235 = 13\,605$ Rest 161

Allgemeines Verfahren: $a = [a_n a_{n-1} \ldots]$, $b = [b_m \ldots]$. 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. $[ac]_\beta = a\beta + c \leq \beta(d + 1)$).

Evtl. notwendige Korrektur `==`

```
      while a ≥ β^i * b do
            a:=a-b;  q_i := q_i + 1;
```

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

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

Für kleine $d$ sind also besonders große Korrekturen zu erwarten.

Beispiel: $100899 = 101 * 999$, $100899 : 101 = (5..9)(4..9)(4..9)$

> AUFGABE: Zeige, daß es zu jeder Basis $\beta$ ein Beispiel gibt, das einen Korrekturaufwand von etwa $\frac{\beta}{2}$ pro Quotientenstelle hat.

Knuths Trick: Finde vorher ein DIGIT $k$, so daß $k \cdot b$ mit einer Ziffer $\geq \frac{\beta}{2}$ beginnt und berechne dann `divmod(a,b)` aus `divmod(ka,kb) = (q,kr)`.

In obigen Beispiel kommt z.B. $k = 5$ in Betracht (das ist $\frac{\beta}{b_m+1}$). Rechne dann $504\,495 : 505$

AUFGABE: Zeige, daß es stets einen Korrekturfaktor $k$ mit der geforderten Eigenschaft gibt, der sich allein aus Kenntnis der ersten Ziffer von $b$ berechnen läßt.

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

AUFGABE: Zeige folgende Aufwandsabschätzung für Knuths Verfahren:

$$C^*_{divmod}(a,b)) = l(q)\,(l(b)+2) + 3\,l(b)$$
$$C_{divmod}(a,b)) = l(q)\,(5\,l(b)+2) + 3\,l(b)$$

Aufwand bleibt jedenfalls $O(l(q) \cdot l(b))$.

## 1.4 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 $a, b \in \mathbf{Z}$ können wir uns im folgenden auf den gcd beschränken.

$\gcd(a,b)$ kann man bekanntlich mit dem *Euklidschen Algorithmus* berechnen:

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

Beispiel:

```
1: Euklid(2134134,581931);
      2134134 = 3 * 581931 + 388341
      581931 = 1 * 388341 + 193590
      388341 = 2 * 193590 + 1161
      193590 = 166 * 1161 + 864
      1161 = 1 * 864 + 297
      864 = 2 * 297 + 270
      297 = 1 * 270 + 27
      270 = 10 * 27 + 0

Der gcd ist also gleich 27
```

**Satz 1** *$g = \gcd(a,b)$ kann für $a, b \in \mathbf{Z}$ 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:D):(D,D,D)
      (u1,v1):=(1,0); (u2,v2):=(0,1);
      while (b≠ 0) do
            (q,r):=divmod(a,b); (u3,v3):=(u1-q*u2,v1-q*v2);
            a:=b; b:=r; (u1,v1):=(u2,v2); (u2,v2):=(u3,v3);
      return (a,u1,v1);
end;
```

Beweis über Schleifeninvariante.
Beispiel:

```
2: ExtendedEuklid(2134134,581931);

   {27,2005,-7353}
```

**Satz 2 (Satz von Lamè, 1845)** *Für $m \geq 1$ seien $a > b > 0$ zwei Zahlen, für die der Euklidsche Algorithmus genau $m$ Schritte benötigt und $a$ sei so klein wie möglich. Dann gilt*

$$a = F_{m+2}, \quad b = F_{m+1},$$

*wobei $F_i$ die Fibonaccifolge ist.*

Binetsche Formel: $F_1 = F_2 = 1,\ F_{m+2} = F_{m+1} + F_m$ für $m > 0$. Dann gilt

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

mit $\alpha = \frac{1+\sqrt{5}}{2} \sim 1.618$.

**Korollar 1** *Der Euklidsche Algorithmus benötigt für die Berechnung des gcd zweier Zahlen $a, b$ höchstens $m \leq l(a) \cdot \frac{\log \beta}{\log \alpha}$ Divisionen mit Rest.*

Beweis: $a \geq F_{m+2} \approx \frac{\alpha^2}{\sqrt{5}} \alpha^m \approx 1.17 \alpha^m > \alpha^m$ und per def. $\log(a) = l(a) \cdot \log(\beta)$.

AUFGABE: Zeigen Sie, daß in der Folge der Reste $r_{i+2} \leq \frac{1}{2} r_i$ gilt und leiten Sie daraus

$$m \leq l(b) \cdot 2 \frac{\log \beta}{\log 2}$$

her. Die Zahl der Divisionen ist also sogar linear in der Bitlänge der *kleineren* der beiden Zahlen.

Im allgemeinen treten entweder viele, aber billige Divisionen oder wenige, aber teure Divisionen auf. 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: Der Euklidsche Algorithmus führt die $m$ Schritte

$$r_{i-1} = q_i\, r_i + r_{i+1} \qquad \text{mit } r_0 = a,\ r_1 = b \text{ und } i = 1, \ldots, m$$

aus. Die Gesamtkosten dieser $m$ Divisionen mit Rest ist 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

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

AUFGABE: Zeige folgende genauere Aufwandsabschätzung:

$$C^*_{gcd}(a,b)) \leq (l(a)+m)\,(l(b)+2)+m\,l(b)$$
$$C_{gcd}(a,b)) \leq (l(a)+m)\,(5\,l(b)+2)+m\,l(b)$$

mit $m \leq c\,l(a)$ und $c = \frac{\log\,\beta}{\log\,\alpha}$ aus dem Satz von Lamé. In beiden Fällen erhalten wir also das Wachstum $O(l(a)\,l(b))$.

**Der binäre gcd-Algorithmus**

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

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 $O(l(a)\cdot l(b))$, allerdings Elementar*additionen*. Ist der schnellste bekannte gcd-Algorithmus, wenn $\beta$ Zweierpotenz und Kofaktoren nicht interessieren.

# 2 Primzahl-Tests und Faktorisierung ganzer Zahlen

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 einen Abschnitt einschieben, in dem die (hoffentlich) aus dem Grundkurs bekannten Sätze über Restklassenringe zusammengefaßt werden.

## 2.1 Der Restklassenring $\mathbf{Z}_n$

Bekanntlich nennt man zwei Zahlen $a, b \in \mathbf{Z}$ *kongruent modulo* $m$ (und schreibt $a \equiv b\ (mod\ m)$), wenn ihre Differenz durch $m$ teilbar ist, also bei Division durch $m$ der Rest 0 bleibt. So gilt $127 \equiv 1\ (mod\ 7)$, aber ebenso $127 \equiv 8\ (mod\ 7)$, 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 $(mod\ 7)$, in der sich die Zahl 1 befindet, besteht etwa aus den Zahlen

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

Die Darstellungen $z \equiv 1\ (mod\ 7),\ 7\,|\,(z-1),\ z = 7k+1,\ z \in [1]_7$ 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 $\mathbf{Z}_m$.

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

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 $(mod\ 7)$ alle Restklassen verschieden von $[0]_7$ prim, $(mod\ 8)$ dagegen nur die Restklassen $[1]_8, [3]_8, [5]_8$ und $[7]_8$ und $(mod\ 6)$ 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 $a \ (mod\ m)$ die *Kürzungsregel*

$$a \cdot x \equiv a \cdot y \ (mod\ m) \ \Rightarrow \ x \equiv y \ (mod\ m).$$

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

Anders formuliert: Die Multiplikationsabbildung

$$\phi_a : \ \mathbf{Z}_m \longrightarrow \mathbf{Z}_m \quad \text{via} \quad x \ (mod\ m) \mapsto a\,x \ (mod\ m)$$

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

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

Der Ring $\mathbf{Z}_m$ 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.

## 2.2 Der Chinesische Restklassensatz

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

**Satz 4 (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*

$$\begin{aligned} x &\equiv x_1 \ (mod\ m_1) \\ &\ldots \\ x &\equiv x_n \ (mod\ m_n) \end{aligned}$$

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

*Anders formuliert, ist die natürliche Abbildung*

$$P : \mathbf{Z}_m \longrightarrow \mathbf{Z}_{m_1} \times \ldots \times \mathbf{Z}_{m_n} \quad \text{via} \quad [x]_m \mapsto ([x]_{m_1}, \ldots, [x]_{m_n})$$

*ein Isomorphismus.*

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

BEWEIS : Injektivität ist trivial, denn $x \equiv 0 \ (mod\ m_i)$ bedeutet $m_i | x$ und wegen der Teilerfremdheit auch $m|x$, also $x \equiv 0 \ (mod\ m)$. Die Surjektivität folgt nun wieder aus der Injektivität und der Gleichmächtigkeit. $\square$

Der angegebene Beweis ist allerdings nicht konstruktiv. Für Anwendungen des Satzes brauchen wir auch eine algorithmische Lösung, die nicht alle Restklassen $(mod\ m)$ 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**

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

zur Berechnung von $x$.

Betrachten wir diese Aufgabe zunächst an einem konkreten Beispiel:

> Gesucht ist eine Restklasse $x \ (mod\ 30)$, so daß $x \equiv 1 \ (mod\ 2)$, $x \equiv 2 \ (mod\ 3)$, $x \equiv 2 \ (mod\ 5)$ gilt.

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

$$l_k := \prod_{j \neq k} u_{jk} m_j \ (mod\ m)$$

Diese Zahlen haben die Eigenschaft, daß

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

gilt. Damit erfüllt

$$x = \sum_k x_k \, l_k \ (mod\ m)$$

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 $l_k \equiv 1 \ (mod\ m_k)$ und $l_k \equiv 0 \ (mod\ m_j)$ für $j \neq k$. Als Lösung des Liftungsproblems $\texttt{CRA}((1, 2), (2, 3), (2, 5))$ ergibt sich dann

$$x = l_1 + 2l_2 + 2l_3 = -13 \equiv 17 \ (mod\ 30).$$

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

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

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

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

Vorbetrachtungen:

$$r \equiv r_2 \ (mod\ m_2) \ => r = r_2 + c \cdot m_2$$
$$r \equiv r_1 (mod\ m_1) \ => c \cdot m_2 \equiv r_1 - r_2 \ (mod\ m_1)$$

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

```
procedure inv(x,m)
      x:=x (mod m);
      (u,v,1):=ExtendedEuklid(x,m); /* d.h. 1 = u·x + v·m */
      return u;
```

Dann ergibt sich `CRA2` als

```
procedure CRA2((r1,m1),(r2,m2))
      c:=(r1 − r2)*inv(m2,m1) (mod m1);
      return r = r2 + c·m2;
```

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` $l(m_1) = 1,\ l(m_2) = k$ annehmen.

Reduktionen von Zahlen der Länge $k$ modulo $m_1$ mit Aufwand $O(k)$
`ExtendedEuklid` mit Aufwand $O(1)$
$r = r_2 + c * m_2$ 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.

AUFGABE: Berechnen Sie `CRA((2,11),(5,13),(1,17),(3,19 ),(7,23))`.

## 2.3 Die Gruppe der primen Restklassen

Da Produkt- und Inversenbildung nicht aus der Menge herausführen, bilden die primen Restklassen $\mathbf{Z}_m^*$ 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

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

**Satz 5 (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 $a^{\phi(m)} \equiv 1 \ (mod \ m)$.*

Ein Spezialfall dieses Satzes:

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

Da mit $x \in \mathbf{Z}_m^*$, also $gcd(x, m) = 1$, auch für jeden Teiler $m_i \,|\, m$ $gcd(x, m_i) = 1$, also $x \in \mathbf{Z}_{m_i}^*$ folgt, induziert $P$ einen Gruppenisomorphismus

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

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.

AUFGABE: Untersuchen Sie, für welche Zahlen $\phi(m)$ ungerade ist.

Zusammengenommen erhält man die Formel

$$\phi(m) = m \prod_{\substack{p|m \\ p \text{ prim}}} \left(1 - \frac{1}{p}\right)$$

Beispiele: $\phi(12) = 4$, $\phi(18) = 6$, $\phi(24) = 8$, $\phi(36) = 12$ und jeweils Demonstration, daß Eulersatz gilt.

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

**Satz 6 (Satz von Carmichael)** *Ist $m = p_1^{a_1} \ldots p_k^{a_k}$ die Primfaktorzerlegung von $m$, so gilt*

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

*mit*

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

Beispiele: $\psi(12) = 2,\ \psi(18) = 6,\ \psi(24) = 4,\ \psi(36) = 6$ und jeweils Demonstration, daß Satz gilt. ($Z_{24}^*$ hat Exponente 2).

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

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 $exp(G) \mid |G|$ und Gleichheit tritt genau für zyklische Gruppen ein.

AUFGABE: Zeigen Sie:

1) Das Gruppenelement $x = (x_1, \ldots, x_n) \in \mathbf{Z}_{m_1}^* \times \ldots \times \mathbf{Z}_{m_n}^*$ hat die Ordnung $ord(x) = lcm(ord(x_1), \ldots, ord(x_n))$.
2) In einer abelschen Gruppe $G$ gibt es zu vorgegebenen $a, b \in G$ stets ein $c \in G$ so daß $ord(c) = lcm(ord(a), ord(b))$ gilt. (Beachten Sie, daß die "einfache" Lösung $c = a \cdot b$ z.B. für $b = a^{-1}$ nicht funktioniert !)
3) Folgern Sie daraus, daß für alle $a \in G$ $ord(a) \mid exp(G)$ gilt, d.h.

$$exp(G) = lcm(ord(a) \ : \ a \in G)$$

ist.

**Satz 7** *Für $p > 2$ ist $G = \mathbf{Z}_{p^n}^*$ zyklisch, d.h. $exp(G) = |G|$.*
*Für $n = 1, 2$ ist $G = \mathbf{Z}_{2^n}^*$ ebenfalls zyklisch, für $n > 2$ gilt dagegen $exp(G) = \frac{|G|}{2}$.*
*$\mathbf{Z}_m^*$ ist damit genau für $m = 2,\ 4,\ p^a,\ 2p^a$ zyklisch, wobei $p$ eine ungerade Primzahl ist.*

(o. Bew.)

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

## 2.4 Primzahl-Testverfahren

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 aufwendigeres 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 **Maple** geschrieben, sehen aber in jedem anderen CAS ähnlich aus.

```
primeTestByTrialDivision:=proc(m::integer)
  local z;
  if (m<3) then evalb(m=2)
  else
    for z from 2 while z*z<=m do
      if m mod z = 0 then RETURN(false) fi;
    od;
    true;
  fi;
end;
```

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

```
primeTestByTrialDivision1:=proc(m::integer)
  local z;
  if (m<2) then false
  elif (m mod 2 = 0) then evalb(m=2);
  elif (m mod 3 = 0) then evalb(m=3);
  else
    for z from 5 by 6 while z*z<=m do
      if m mod z = 0 then RETURN(false) fi;
      if m mod (z+2) = 0 then RETURN(false) fi;
    od;
    true;
  fi;
end;
```

`evalb` 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 aufwendigeren Verfahren nur für solche Zahlen anzuwenden die "nicht offensichtlich" keine Primzahlen sind. In der folgenden Maple-Prozedur ist `smallPrimes` eine Liste aller "kleinen" Primzahlen:

```
smallPrimesTest:=proc(m::integer)
  local i,smallPrimes;
  smallPrimes:=[3, 5, 7, 11, 13, 17, 19, 23, 29];
  if (m<2) then false
  elif type(m,even) then evalb(m=2)
  else
     for i in smallPrimes do
       if (m mod i = 0) then RETURN(evalb(m=i)) fi;
     od;
     FAIL;
  fi
end;
```

In dieser Prozedurdefinition wird Maples 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.

> AUFGABE: Untersuchen Sie die Wirksamkeit dieses Verfahrens sowohl theoretisch als auch praktisch. Zeigen Sie insbesondere, daß in einem genügend großen Intervall nur etwa 16 % der Zahlen den Test nicht bestehen.

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 $1 < a < m$*

$$a^{m-1} \not\equiv 1 \ (mod\ m),$$

*so kann $m$ keine Primzahl sein.*

Eine Realisierung in Maple hätte etwa folgende Gestalt:

```
FermatTest:=proc(m::integer,a::integer)
  evalb(a &^ (m-1) -1 mod m = 0)
end;
```

Hierbei wird der *inerte* Potenzoperator `&^` verwendet, da sonst zuerst $a^{n-1}$ als ganze Zahl berechnet und dann erst $(mod\ n)$ reduziert würde, während in unserem Fall das Potenzieren direkt im Ring $\mathbf{Z}_n$ ausgeführt wird.

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 $a^{m-1} \not\equiv 1 \ (mod\ m)$ 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.

Auf dieser Basis könnte man einen **Las-Vegas-Test** versuchen:

*Mache den Fermat-Test für $c$ verschiedene (zufällig gewählte) Basen $a_1, \ldots, a_c$.*

*Ist einmal $a_i^{m-1} \not\equiv 1 \ (mod\ m)$, so ist $m$ garantiert eine zusammengesetzte Zahl.*

*Ist stets $a_i^{m-1} \equiv 1 \ (mod\ m)$, so ist $m$ wahrscheinlich (hoffentlich mit großer Wahrscheinlichkeit) eine Primzahl.*

Eine Maple-Implementierung hätte etwa folgende Gestalt:

```
LasVegas1:=proc(m::integer,c::integer)
  local i;
  for i from 1 to c do
    if not FermatTest(m,rand(m)()) then RETURN(false) fi;
    od;
  true;
end;
```

`rand(m)` gibt dabei gemäß der Maple-Dokumentation keine Zufallszahl, sondern eine (nullstellige) Funktionsdefinition zurück, `rand(m)()` ruft diese auf.

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

**Satz 8** *Die Menge*

$$P_m := \{a \in \mathbf{Z}_m^* \ : \ a^{m-1} \equiv 1 \ (mod\ m)\}$$

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

Beweis mit Untergruppenkriterium.

Nach dem Satz von Lagrange ist somit $|P_m|$ ein Teiler von $\phi(m) = |\mathbf{Z}_m^*|$. Ist $m$ also zusammengesetzt und $P_m \neq \mathbf{Z}_m^*$, 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.

Ist andererseits $m$ eine zusammengesetzte Zahl und $P_m = \mathbf{Z}_m^*$, 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 $a \in \mathbf{Z}_{561}^*$ stets $a^{560} = 1$

Zusammengesetzte Zahlen dieser Art nennt man *Carmichael-Zahlen.*

**Satz 9** *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 1105=5*13*17 und 1729=7*13*19.

AUFGABE: a) Zeigen Sie, daß Carmichael-Zahlen $m$ stets quadratfrei sind und immer wenigstens 3 Primfaktoren haben. Führen Sie dazu die Annahmen $m = p^a \cdot q$ und $m = p \cdot q$ jeweils zum Widerspruch.

b) Zeigen Sie, daß $N = (6t+1)(12t+1)(18t+1)$ eine Carmichaelzahl ist, wenn $6t+1, 12t+1$ und $18t+1$ Primzahlen sind.

Carmichaelzahlen sind unter den Primzahlen also 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 sich die Anzahl $C(x)$ der Carmichaelzahlen kleiner $x$ durch $C(x) \geq x^{\frac{2}{7}}$ nach unten hin abschätzen läßt, d.h. ihre Zahl exponentiell mit der Bitlänge von $x$ wächst.

## 2.5 Quadratische Reste und der Solovay-Strassen-Test

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

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

**Lemma 1** *1) Das Produkt zweier quadratischer Reste ist wieder ein quadratischer Rest.*

*2) Das Produkt eines quadratischen Rests mit einem quadratischen Nichtrest ist ein quadratischer Nichtrest.*

*3) Es gibt höchstens $\frac{\phi(m)}{2}$ quadratische Reste und damit mindestens $\frac{\phi(m)}{2}$ quadratische Nichtreste.*

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) $\mathbf{Z}_m^* \longrightarrow \mathbf{Z}_m^*$ via $x \mapsto x^2$ ist (mindestens) 2-1-Abbildung. □

Beispiele: $m = 7, m = 11, m = 8$.

DEFINITION: Für eine Primzahl $m$ und $a \in \mathbf{Z}_m^*$ 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 10** *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. $\kappa : \mathbf{Z}_m^* \to \{+1, -1\}$ 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) Aus $x^2 \equiv y^2 \pmod{m}$ folgt, daß $m$ ein Teiler von $x^2 - y^2 = (x+y)(x-y)$ ist. Ist $m$ eine Primzahl, so gilt $m | x+y$ oder $m | x-y$. Damit gibt es im Fall eines quadratischen Rests $a$ genau zwei Restklassen $\pm x$, für die $x^2 \equiv a$ gilt.

2) Abzählen zeigt, daß dann auch das Produkt zweier Nichtreste ein quadratischer Rest sein muß.

3) Wegen $a^{m-1} \equiv 1 \pmod{m}$ und da $m$ prim ist kann $a^{\frac{m-1}{2}}$ nur die Reste $+1$ oder $-1$ annehmen. Weiterhin ist $\mathbf{Z}_m^*$ eine zyklische Gruppe. Nehmen wir an, daß sie vom Element $\sigma$ erzeugt wird, so

sind genau die $\sigma^{2k}$, $k = 1, \ldots, \frac{m-1}{2}$ die quadratischen Reste und die $\sigma^{2k-1}$, $k = 1, \ldots, \frac{m-1}{2}$, die quadratischen Nichtreste. $\square$

Die Definition des Legendresymbols kann man erweitern zur Definition des *Jacobi-Symbols*, indem man für die ungerade Zahl $m = p_1^{a_1} \ldots p_k^{a_k}$

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

setzt.

$\kappa : \mathbf{Z}_m^* \to \{+1, -1\}$ 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 ($mod$ 15) kein quadratischer Rest, weil ja bereits 2 ($mod$ 3) kein quadratischer Rest ist.

> AUFGABE: Zeigen Sie, daß für eine quadratfreie ungerade Zahl $m = p_1 \cdot \ldots \cdot p_k$ die Restklasse $a \in \mathbf{Z}_m^*$ genau dann ein quadratischer Rest ist, wenn für alle $i = 1, \ldots, k$ stets $(\frac{a}{p_i}) = +1$ gilt. (Hinweis: Chinesischer Restklassensatz)

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

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

Ist andererseits die ungerade Zahl $m$ keine Primzahl, so ist in der Regel $a^{m-1} \not\equiv 1 \ (mod\ m)$ und so erst recht $a^{\frac{m-1}{2}} \not\equiv \pm 1 \ (mod\ m)$. 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 eine zentrale Rolle.

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

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

3) $\left(\frac{p}{q}\right) = (-1)^{\frac{p-1}{2}\frac{q-1}{2}} \left(\frac{q}{p}\right)$

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

$$\left(\tfrac{43}{101}\right) = + \left(\tfrac{101}{43}\right) = \left(\tfrac{15}{43}\right)$$
$$\left(\tfrac{3}{43}\right) = - \left(\tfrac{43}{3}\right) = -1$$
$$\left(\tfrac{5}{43}\right) = - \left(\tfrac{43}{5}\right) = - \left(\tfrac{3}{5}\right) = \left(\tfrac{2}{3}\right) = -1$$

oder

$$\left(\tfrac{15}{43}\right) = - \left(\tfrac{43}{15}\right) = - \left(\tfrac{13}{15}\right) = - \left(\tfrac{2}{13}\right) = -(-1)^{\frac{12 \cdot 14}{8}} = +1$$

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

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

```
primeTestSolovayStrassen:=proc(m::integer,c::integer)
  local a,i;
  for i from 1 to c do
      a:=rand(m)();
      if igcd(a,m) <> 1 then RETURN(false) fi;
      if (a &^ ((m-1)/2) * numtheory[jacobi](a,m) mod m <> 1)
            then RETURN(false) fi;
      od;
  true;
end;
```

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 $a^{\frac{m-1}{2}} * (\frac{a}{m}) \neq 1$ gilt, so ist $m$ garantiert zusammengesetzt.
>
> Gilt $a^{\frac{m-1}{2}} * (\frac{a}{m}) = 1$ für alle $a_i$, so ist die Wahrscheinlichkeit, daß $m$ zusammengesetzt ist, kleiner als $2^{-c}$.

## 2.6 Der Rabin-Miller-Test

Ein weiterer Primzahltest beruht auf der folgenden Verfeinerung des Fermat-Tests: Für eine Primzahl $m$ und eine zufällig gewählte Zahl $1 < a < m$ muß $a^{m-1} \equiv 1 \ (mod\ m)$ gelten. Ist $m-1 = 2^t \cdot q$ die Zerlegung des Exponenten in eine Zweierpotenz und einen ungeraden Anteil $q$, so ergibt die Restklasse $b := a^q \ (mod\ m)$ nach $t$-maligem Quadrieren den Rest 1. Bezeichnet $u$ das Element in der Folge $\{b, b^2, b^4, b^8, \ldots, b^{2^t}\}$, wo erstmals $u^2 \equiv 1 \ (mod\ m)$ gilt, so muß, wenn $m$ eine Primzahl ist, $u \equiv -1 \ (mod\ m)$ gelten. Ist dagegen $m$ keine Primzahl, so hat die Kongruenz $u^2 \equiv 1 \ (mod\ m)$ 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 $a_1 \equiv 1 \ (mod\ p)$ und $a_1 \equiv -1 \ (mod\ q)$ und eine weitere Restklasse $a_2 \ (mod\ m)$ mit $a_2 \equiv -1 \ (mod\ p)$ und $a_2 \equiv 1 \ (mod\ q)$. Für beide gilt $a_i^2 \equiv 1 \ (mod\ p)$ und $a_i^2 \equiv 1 \ (mod\ q)$, also auch $a_i^2 \equiv 1 \ (mod\ m)$, aber $a_i \not\equiv \pm 1 \ (mod\ m)$.

Wählt man $a$ zufällig aus, so ist die Wahrscheinlichkeit, daß $u$ bei zusammengesetzten $m$ auf eine solche Restklasse trifft, d.h. $u \not\equiv -1 \ (mod\ m)$, aber $u^2 \equiv 1 \ (mod\ m)$ gilt, groß. Eine genauere Analyse 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.

Auf der beschriebenen Verfeinerung beruht der folgende **Rabin-Miller-Test**

```
primeTestRabinMiller:=proc(m::integer,c::integer)
  local a,q,b,t,i,j;
  q:=m-1; t:=0;
  while type(q,even) do q:=q/2; t:=t+1 od;
        # nun ist  m-1 = 2^t *q
  for i from 1 to c do
      a:=rand(m)(); b:=mods(a &^ q,m);
      if (b=1)or(b=-1) then next fi; # keine Information
      for j from 1 to (t-1) do
          # nun ist  b <> 1 oder -1 (m)
        b:=mods(b*b,m);
        if (b = 1) then RETURN(false) fi;
        if (b = -1) then break fi;   # keine Information
      od;
      if (b = -1) then next;         # keine Information
      else RETURN(false)
```

```
      fi;
  od;
  true;
end;
```

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

> Erhalten wir den Rest 1, so war $b \not\equiv \pm 1 \ (mod\ m)$, aber $b^2 \equiv 1 \ (mod\ m)$, 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 $b^{2^t} \equiv a^{m-1} \ (mod\ m)$. Da nach Verlassen der inneren Schleife stets $b \not\equiv 1 \ (mod\ m)$ gilt, kommen folgende Fälle in Frage:

> $b^2 \not\equiv 1 \ (mod\ m)$, also $m$ nicht prim nach dem Fermat-Test.
>
> $b \not\equiv -1 \ (mod\ m)$, aber $b^2 \equiv 1 \ (mod\ m)$, also ist $m$ nicht prim.
>
> $b \equiv -1 \ (mod\ m)$. In diesem Fall kann aus der gewählten Basis $a$ keine Aussage getroffen werden.

## 2.7 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 $25 * 10^9$ (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 [2], Section 4.5.4, Algorithm P for a reference and [5]. No counter example is known and it has been conjectured that such a counter example must be hundreds of digits long.

Allerdings möchte man für gewisse Anwendungen sichergehen, daß es sich bei der untersuchten Zahl garantiert um eine Primzahl handelt, also ein *Zertifikat* für die Primzahleigenschaft haben.

Ein solches Zertifikat kann etwa erstellt werden, wenn man zu der vermuteten Primzahl $m$ ein Erzeugendes $a$ der zyklischen Gruppe $\mathbf{Z}_m^*$ angibt (zusammen mit einem Beweis, daß dieses Element die behauptete Eigenschaft besitzt).

**Satz 11** *Die ungerade Zahl $m$ ist genau dann eine Primzahl, wenn $\mathbf{Z}_m^*$ eine zyklische Gruppe der Ordnung $m-1$ ist, d.h. wenn es ein Element $a \in \mathbf{Z}_m^*$ gibt, so daß $ord(a) = m-1$ gilt.*

*Ein Element $a \in \mathbf{Z}_m^*$ hat genau dann die Ordnung $ord(a) = m-1$, wenn $a^{m-1} = 1$, aber für alle Primteiler $p$ der Zahl $m-1$ die Beziehung $a^{\frac{m-1}{p}} \neq 1$ gilt.*

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 Maple erhalten wir

```
m:=55499821019;
with(numtheory):
u:=factorset(m-1);
                          u := {2, 17, 1447, 1128091}

for x in u do print (2 &^ ((m-1)/x) mod m) od;

                                  55499821018

                                  44871471456

                                  31660914932

                                  44713683701
2 &^ (m-1) mod m;
                                        1
```

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

Oftmals ist allerdings für eine konkrete Restklasse $a$ die Beziehung $a^{\frac{m-1}{p}} \not\equiv 1 \ (mod\ m)$ nur für einige der Primfaktoren von $m-1$ erfüllt und es ist schwierig, eine Restklasse zu finden, die für *alle* Primteiler paßt. Es stellt sich, mit Blick auf den Chinesischen Restklassensatz nicht verwunderlich, heraus, daß es genügt, für jeden Primteiler *seine* Restklasse $a$ zu finden.

> AUFGABE: Seien $\{p_1, \ldots, p_k\}$ die (verschiedenen) Primfaktoren von $m-1$. Zeigen Sie folgenden Zusammenhang:
>
> $$\exists\, a \in \mathbf{Z}_m^*\ \forall i\ a^{\frac{m-1}{p_i}} \not\equiv 1 \ (mod\ m)$$
>
> genau dann, wenn
>
> $$\forall i\ \exists\, a_i \in \mathbf{Z}_m^*\ a_i^{\frac{m-1}{p_i}} \not\equiv 1 \ (mod\ m),$$
>
> d.h. es gibt eine gemeinsame Basis für alle Primteiler von $m-1$, wenn es für jeden Primteiler einzeln eine passende Basis gibt.

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 $a_p^{\frac{m-1}{p}} \not\equiv 1 \ (mod\ m)$ gilt:

```
certifyPrime:=proc(m)
   local u,v,x,a;
   if not isprime(m) then print(` `.m.` is not prime`); RETURN(NULL) fi;
   u:=[]; v:=numtheory[factorset](m-1);
   for x in v do
     for a in [2,3,5,7,11,13,17,19,23] do
```

```
      if (a &^ ((m-1)/x) mod m) <> 1 then u:=[op(u),[x,a]]; break; fi;
    od;
  od;
  if nops(u)<>nops(v) then
    print(` `.m.` not certified`); RETURN(NULL);
  fi;
  u;
end;
```

Auf eine konkrete Primzahl angewendet erhalten wir damit folgendes Zertifikat:

```
m:=nextprime(22^20);
                        m := 705429498686404044207947783

certifyPrime(m);

[[2, 5], [7, 2], [31, 2], [1571, 2], [317, 2], [4655972662649089, 2], [701, 2] ]
```

Der erste Eintrag gibt dabei jeweils den Primfaktor von $m-1$, der zweite die für diesen Primfaktor $p$ geeignete Basis $a_p$ an.

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 der Form $m = 2^a+1$, da hier die Zahl $m-1$ nur durch 2 teilbar ist. Allerdings stellt es sich heraus, daß es nicht viele solche Primzahlen gibt.

> AUFGABE: Zeigen Sie, daß $m = 2^a+1$ nur dann eine Primzahl sein kann, wenn $a$ keinen ungeraden Faktor enthält, d.h. selbst Zweierpotenz ist.

Zahlen der Form $F_n := 2^{2^n}+1$ bezeichnet man als *Fermatzahlen*, da sie erstmals von Fermat untersucht wurden, der in einem Brief an Mersenne die Behauptung aufstellte, daß alle Zahlen $F_n$ 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:

```
3 &^ (4294967297-1) mod 4294967297;
                                    3029026160
```

Diese Basis kann man generell für den Fermattest von $F_n$ verwenden und zeigen, daß auch die nächsten Fermatzahlen (deren Stellenzahl sich allerdings jeweils verdoppelt) zusammengesetzt sind.

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 [3].

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. [5].

Neben den Fermatzahlen spielen auch Primzahlen der Form $m = 2^a - 1$ eine wichtige Rolle. Die größten heute bekannten Primzahlen haben diese Gestalt. Da $m-1$ keine so einfache Faktorisierung wie für Fermatzahlen besitzt, ist hier die Erstellung eines Primzahlzertifikats schwieriger, jedoch hat $m-1 = 2\,(2^{a-1}-1)$ wieder einen Faktor derselben Gestalt, was für eine teilweise Faktorisierung oft sehr hilfreich ist.

AUFGABE: Zeigen Sie folgende Eigenschaften:

1) Ist $b$ ein Teiler von $a$, so ist auch $2^b - 1$ ein Teiler von $2^a - 1$.

2) $2^a - 1$ ist höchstens dann eine Primzahl, wenn $a$ selbst prim ist.

3) Untersuchen Sie, welche der Zahlen $2^a - 1$ für $a < 100$ wirklich Primzahlen sind.

Primzahlen der Form $M_p = 2^p - 1$ bezeichnet man als *Mersennesche Primzahlen*.

Moderne Primtestverfahren verwenden auch andere Gruppen als $\mathbf{Z}_m^*$ zum Test, insbesondere solche, die mit elliptischen Kurven verbunden sind. Allerdings kennt man bis heute noch keinen sicheren Primzahltest mit garantiert polynomialem Laufzeitverhalten.

## 2.8 Faktorisierung langer Integer-Zahlen

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.

```
OurFactor:=proc(m::integer,splitFactorFunction)
  local n;
  if isprime(m) then m
  else
    n:=splitFactorFunction(m);
    OurFactor(n,splitFactorFunction), OurFactor(m/n,splitFactorFunction)
  fi;
end;
```

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::integer)
  local z;
  if (m<3) then m
  else
    for z from 2 while z*z<=m do
        if m mod z = 0 then RETURN(z) fi;
    od;
    m;
  fi;
end;
```

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

Die zugehörige Faktorisierungsfunktion ist dann

```
trialFactor:=proc(m::integer) OurFactor(m,splitTrialFactor) end;
```

Für Anwendungen ist dieses Verfahren jedoch nur für kleine Zahlen geeignet. Allerdings werden in praktischen Implementierungen in einem Preprocessing kleine Teiler durch ein solches Probedivisionsverfahren vorab herausdividiert, da zusammengesetzte Zahlen oftmals aus vielen solchen kleinen Teilern und einem großen primen "Rest" bestehen. Für "harte" Faktorisierungsprobleme, wenn die zu untersuchende Zahl aus mehreren "großen" Primfaktoren besteht, müssen allerdings andere Zerlegungsverfahren herangezogen werden.

AUFGABE: Modifizieren Sie `OurFactor` entsprechend.

Wir wollen an dieser Stelle stellvertretend für andere klassische Verfahren ein solches Zerlegungsverfahren, die **Pollardsche Rho-Methode**, kennenlernen.

Die grundlegende Idee dieses Verfahrens benutzt Fixpunkteigenschaften von Abbildungen der endlichen Menge $S = \mathbf{Z}_r = \{0, 1, \ldots, r-1\}$ 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 (Maple):

```
gen:=proc(x,f) local l,y,i; l:=x; y:=x;
    for i from 1 to 15 do y:=f(y); l:=l,y od;
    [l] end;
```

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);
               [2, 5, 0, 1, 2, 5, 0, 1, 2, 5, 0, 1, 2, 5, 0, 1]

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

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

gen(7,x -> x^2+1 mod 37);
          [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.

AUFGABE: Analysieren Sie für beide Fälle mit einem geeigneten CAS, in welche Vorperioden und Perioden die Menge $\mathbf{Z}_r$ jeweils zerfällt.

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. Diese Werte extrahiert die Prozedur

```
pollard1:=proc(f,x0) local u,x;
    x:=x0; u:=[];
    while not member(x,u) do u:=[op(u),x]; x:=f(x)  od;
    u;
end;
```

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

> AUFGABE: Zeigen Sie, daß die Periodenlänge $(mod\ r)$ sogar ein Teiler der Periodenlänge $(mod\ m)$ ist.

Ist insbesondere $r$ ein echter Teiler der zu faktorisierenden Zahl $m$, so können wir erwarten, daß "im Durchschnitt" die Periodenlänge $(mod\ r)$ echt kleiner als die Periodenlänge $(mod\ m)$ ist, d.h. für geeignete Folgenglieder

$$x_j \equiv x_k \ (mod\ r), \quad \text{aber} \quad x_j \not\equiv x_k \ (mod\ m) \quad (*)$$

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 $(mod\ m)$ bestimmen, können wir $gcd(x_j - x_k, m)$ für alle Paare $j < k$ ausrechnen und hoffen, dabei auf einen echten Faktor von $m$ u stoßen.

Beispiel: Betrachten wir wieder die Übergangsfunktion $f(x) = x^2+1$ und $m = 91$ und erzeugen zunächst alle verschiedenen Zahlen aus der Pollard-Sequenz zum Startwert 7:

```
o:=pollard1(x -> x^2+1 mod 91,7);
                          o := [7, 50, 44, 26, 40, 54, 5]
```

Dann bestimmen wir $gcd(x - y, m)$ für alle Paare $(x, y)$ aus $o$:

```
seq(seq(igcd(op(i,o)-op(j,o),91), i=1..(j-1)), j=1..nops(o));
        1, 1, 1, 1, 1, 1, 1, 1, 1, 7, 1, 1, 1, 7, 7, 1, 1, 13, 7, 7, 7
```

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

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

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

Bezeichnen wir für die Funktion $f$ und den Startwert $x_0$ mit

$$K(f, x_0; r) := \min(k \ : \ \exists\, j < k \ x_j \equiv x_k \ (mod\ r)\,)$$

den Index, mit dem die Vorperiode $(mod\ r)$ endet, so ist unsere Hoffnung also, daß für $r < m$ im Durchschnitt

$$K(f, x_0; r) < K(f, x_0; m)$$

gilt, also $(*)$ beim systematischen Durchmustern der Folge auftritt.

Was kann man über den Durchschnittswert $K^{(r)}$ von $K(f, x_0; r)$, gemittelt über alle $f$ und $x_0$ sagen?

**Satz 12** *Sei $S$ eine $r$-elementige Menge, $f : S \to S$ eine Abbildung und $x_0 \in S$.*

*Dann ist der Teil der Paare $(f, x_0)$, für die das Anfangsstück $\{x_0, \ldots, x_l\}$ der zugehörigen Pollard-Sequenz aus paarweise verschiedenen Zahlen besteht, kleiner als $e^{-\frac{l^2}{2r}}$.*

*Für den Durchschnittswert $K^{(r)}$ gilt dann*

$$K^{(r)} \leq \sqrt{\frac{\pi\, r}{2}}.$$

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.

Für den Durchschnittswert

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

(vgl. die Komplexitätsberechnung von `comp`) erhalten wir schließlich

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

□

Wir werden also nicht, wie oben, zuerst die ganze Pollard-Sequenz berechnen (Aufwand $O(m)$), sondern statt dessen jedes neuberechnete Element sofort mit seinen Vorgängern vergleichen, in der Hoffnung, daß wir spätestens nach $l \approx O(\sqrt{r})$ Gliedern einen nichttrivialen Teiler entdecken.

Dieses Vorgehen ist leider nicht besser als `Trialfactor`: Das Berechnen von $l$ Folgengliedern verursacht einen Aufwand von $l \cdot O(l(m)^2)$, die Berechnung der gcd's aller $O(l^2)$ paarweisen Differenzen mit $m$ einen Aufwand von $l^2 \cdot O(l(m)^2)$ und somit einen Gesamtaufwand in der Größenordnung

$$O(l^2 \cdot l(m)^2) \sim O(r \cdot l(m)^2).$$

Dabei ist die gcd-Berechnung der aufwendige Teil, so daß die Frage entsteht, ob man dort sparen kann. Das scheint möglich, denn auf Grund der polynomialen Bildungsvorschrift $f$ ist $x_i - x_k$ ein Teiler von $x_{i+1} - x_{k+1} = f(x_i) - f(x_k)$. Haben also $x_i - x_k$ und $m$ einen nichttrivialen gemeinsamen Teiler, so auch $x_{i+1} - x_{k+1}$ und $m$. Wir wollen deshalb jedes neuberechnete Folgenglied $x$ nicht mit allen vorhergehenden vergleichen, sondern nur mit einem einzigen. Als Referenzelement $y$ merken wir uns das jeweils zuletzt berechnete und behalten es zuerst für 2, das nächste für 4, dann für 8 usw. Schritte bei. Weiterhin wollen wir aus Effizienzgründen zuerst das Produkt $p$ (natürlich $(mod\ m)$) aller Differenzen $(x - y)$ mit demselben Referenzelement berechnen und erst dann $gcd(p, m)$. Obwohl $gcd$ und Produkt in derselben Komplexitätsklasse $O(l(m)^2)$ liegen, ist dieses Vorgehen natürlich schneller. Unser Algorithmus hat dann das folgende Aussehen:

```
pollard2:=proc(m,f,x0)
  local x,y,k,h,g,p; x:=x0; h:=0;
  do
    p:=1; y:=x; # nun ist y=x_j mit j=2^h-1
    for k from 2^h to (2^(h+1)-1) do
      x:=f(x) mod m; p:=p*(x-y) mod m;
      if p=0 then RETURN(0) fi; # Methode mit f gescheitert
      g:=igcd(p,m);
      if g>1 then RETURN(g) fi;
    od;
    h:=h+1;
  od;
end;
```

Da jedes Folgenglied nur noch mit einem einzigen anderen Folgenglied verglichen wird, sinkt die Komplexität des Algorithmus auf $O(l_1 \cdot l(m)^2)$, wobei $l_1$ der Index des ersten Folgenglieds ist, das einen nichttrivialen gcd erzeugt. Dieser Index

$$l_1 = min\{k \,:\, gcd(x_k - y, m) > 1 \; mit \; y = [\log_2(k)]\}$$

wird natürlich größer sein als der Index

$$l_0 = min\{k \,:\, \exists\, j < k \; gcd(x_k - x_j, m) > 1\},$$

da ja einfach weniger Differenzen untersucht werden.

Wie hängen nun $l_0$ und $l_1$ zusammen? Betrachten wir dazu die Menge

$$I = \{(k, j) \,:\, gcd(x_k - x_j, m) > 1\}.$$

Wir hatten gesehen, daß

$$(k, j) \in I \;\Rightarrow\; (k+1, j+1) \in I \text{ und allgemein } (k+i, j+i) \in I$$

gilt. Für $k = l_0$, $(k, j) \in I$ und $h := [\log_2(k)]$, also $2^{h+1} - 1 \geq k \geq 2^h$ gilt dann

$$(k + 2^{h+1} - 1 - j, 2^{h+1} - 1) \in I,$$

also $l_1 < k + 2 \cdot 2^h \leq 3\,k$.

Folglich ist $l_1 \leq 3\,l_0 \leq 3\sqrt{\frac{\pi\, r}{2}}$, die Komplexität dieses Algorithmus als nur noch von der Größenordnung

$$O(\sqrt{r}\, l(m)^2).$$

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:

| $p$ | $M_p$ | Faktoren | `trialFactor` | `pollardFactor` |
|---|---|---|---|---|
| 11 | 2047 | [23, 89] | 0 | 0 |
| 23 | 8388607 | [47, 178481] | 0 | .010 |
| 29 | 536870911 | [233, 1103, 2089] | .080 | .010 |
| 37 | 137438953471 | [223, 616318177] | .010 | .010 |
| 41 | 2199023255551 | [13367, 164511353] | .860 | .020 |
| 43 | 8796093022207 | [431, 9719, 2099863] | .710 | .070 |
| 47 | 140737488355327 | [2351, 4513, 13264529] | .480 | .060 |
| 53 | 9007199254740991 | [6361, 69431, 20394401] | 5.650 | .130 |
| 59 | 576460752303423487 | [179951, 3203431780337] | 13.610 | .280 |

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

$$C_{pollardFactor} \sim O(\sqrt[4]{m}\, l(m)^2) = O(l(m)^2 \cdot 2^{\frac{l(m)}{4}}).$$

Das beste heute bekannte klassische Faktorisierungsverfahren, die *Kettenbruchmethode von Morrison und Brillhart*, hat bereits subexponentielle Laufzeit, d.h. eine Laufzeit $O(2^{l\,\alpha(l)})$ mit $\alpha(l) \to 0$, wenn $l = l(m) \to \infty$. Genauer gilt

$$C_{MB} = O(2^{C \cdot \sqrt{l(m)\ \log(l(m))}})$$

Von dieser Güte sind auch die schnellsten heute bekannten Faktorisierungsverfahren, die auf elliptischen Kurven beruhen. Sie unterscheiden sich nur in der Größe des Faktors $C$ im Exponenten. In praktischen Anwendungen lohnt es sich, verschiedene Verfahren alternativ zu probieren, da sich die Laufzeiten auch für einzelne Beispiele sehr unterscheiden.

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 Mapledokumentation:

```
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 indicating
  an m-digit composite number that was not factored.

- 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.
```

Betrachten wir zum Schluß eine komplexe Faktorisierungsaufgabe aus [2]:

```
u:=2^107-2^54+1;
                    u := 162259276829213345377179500806145

pollardFactor(u);
                     5, 857, 37866809061660057264219253397
```

Wir erhalten diese drei Faktoren, wobei beim dritten, den wir mit $u_1$ bezeichnen wollen, noch ein gewisser Zweifel bleibt, ob es sich wirklich um eine Primzahl handelt. Wir können diese Zweifel zerstreuen, indem wir ein Primzahlzertifikat erzeugen, das, wie beschrieben, mit einer Faktorisoerung von $u_1 - 1$ verbunden ist.

```
certifyPrime(37866809061660057264219253397);
  [[2, 2], [19, 3], [107, 2], [353, 3], [143675413657196977, 3], [91813, 3]]
```

Wiederum bleibt ein Rest Zweifel, ob $u_2 = 143675413657196977$ eine Primzahl ist, die wir auf demselben Wege zerstreuen können:

```
certifyPrime(143675413657196977);
            [[2, 5], [3, 2], [1103, 2], [1653701519, 2], [547, 2]]
```

Und für ganz große Skeptiker auch noch dies:

```
certifyPrime(1653701519);
            [[2, 7], [7, 2], [19, 2], [23, 2], [137, 2], [1973, 2]]
```

# 3 Polynome, ihre Darstellung und Arithmetik

Polynome in einer Variablen über einem Grundbereich $A$. Dichte und dünne Darstellung

Dichte Darstellung ist an die Ringstruktur von $R = A[x]$ schlechter angepaßt als die dünne Darstellung.

**Komplexität der dichten Darstellung:**

Zur Beschreibung findet nicht mehr nur ein Parameter wie bei den langen Zahlen, sondern 2 Parameter Verwendung: $(t, d)$ mit $t$ als Schranke für die Bitlänge der Koeffizienten und $d$ als Gradschranke. Speziell wollen wir zwei wichtige Fälle gleich mit betrachten: Polynome, bei denen die Operationen auf den Koeffizienten in konstanter Zeit ablaufen (float, modular) und Polynome mit ganzzahligen Koeffizienten.

Dann gilt für das entsprechende Element aus $f \in R = A[x]$:

Seine Bitlänge ist $L(f) = O(t\,d)$

Addition zweier solcher Polynome
Komplexität : $C_R^+(f,g) \sim C_A^+(t(f), t(g)) \cdot max(d(f), d(g))$

Multiplikation zweier solcher Polynome
Komplexität : $C_R^*(f,g) \sim C_A^*(t(f), t(g)) \cdot d(f)\,d(g)$

Diese Komplexität entspricht dem klassischen Multiplikationsverfahren. Man kann auf offensichtliche Weise etwa die Idee der Karatsubamultiplikation auf Polynome übertragen und erhält dann eine Komplexität

$$C_R^*(f,g) \sim C_A^*(t(f), t(g))\,d^\alpha$$

wobei $d$ eine Gradgrenze für $f$ und $g$ ist und $\alpha = \log(3)/\log(2)$ gilt.

**Komplexität der dünnen Darstellung:**

Beschreibung durch $(t, l)$ mit $t$ als Schranke für die Bitlänge der Koeffizienten und $l$ als Anzahl der Terme. Dann gilt für das entsprechende Element aus $A[x]$:

Bitlänge $O(t\,l)$

Addition zweier solcher Polynome
Komplexität : worst case $C_R^+(t,l) \sim C_A^+(t)\,l$
average, wenn $l \ll d$: Nur Merge-Kosten, also $2\,l$ Exponentenvergleiche.
average, wenn $l \sim d$, wie worst case.

Anzahl der Terme: Im Fall $l \ll d$ eher $2l$, im Fall $l \sim d$ eher $l$.

Zeigt die Schwierigkeit, die Komplexität dünner Polynome einzuschätzen.

**Polynome in mehreren Variablen** $R = A[x_1, \ldots, x_m]$

Rekursive und distributive Darstellung. $d_i(f)$ und $t(f)$ für $f \in R$.

Beispiel aus der Vorlesung ECA:

$$h_3(x_1, \ldots, x_4) = x_1x_2x_3 + x_1x_2x_4 + x_1x_3x_4 + x_2x_3x_4 + {x_1}^3 + {x_2}^3 + {x_3}^3 + {x_4}^3 + x_1{x_2}^2 + {x_1}^2x_2 + x_1{x_3}^2 + {x_1}^2x_3 + x_1{x_4}^2 + x_2{x_3}^2 + {x_1}^2x_4 + {x_2}^2x_3 + x_2{x_4}^2 + {x_2}^2x_4 + x_3{x_4}^2 + {x_3}^2x_4$$

Distributive Darstellung geht von einer *Termordnung* aus.

Komplexität wollen wir nur in einem rekursiven Ansatz betrachten. Ausgangsparameter ist der Polynomring $R = A[x_1, \ldots, x_m]$, wobei wir die Komplexität eines Polynoms $f \in R$ durch ein Tupel $(t, d_1, \ldots, d_m)$ charakterisieren wollen, in dem $t$ für die Koeffizientenkomplexität (als z.B. deren Bitlänge) steht und $d_i = d_i(f)$ gilt.
Wir erhalten:

Bitlänge $L(f) = O(t\, d_1\, \ldots\, d_m)$

Addition zweier solcher Polynome

$$C_R^+(f,g) \sim C_A^+(t) \cdot \prod_{i=1}^{m} max(d_i(f), d_i(g))$$

Multiplikation zweier solcher Polynome

$$C_R^*(f,g) \sim C_A^*(t) \cdot \prod_{i=1}^{m} (d_i(f)\, d_i(g))$$

Für Grundbereiche mit konstanten Kosten für die Arithmetik und Gardschranken $(d_1, \ldots, d_m)$ erhalten wir daraus

$$C_R^+(f,g) \sim d_1 \cdot \ldots \cdot d_m$$
$$C_R^*(f,g) \sim (d_1 \cdot \ldots \cdot d_m)^2,$$

für den Grundbereich $\mathbf{Z}$ und Koeffizienten mit einer Bitlänge kleiner als $t$ dagegen

$$C_R^+(f,g) \sim t \cdot d_1 \cdot \ldots \cdot d_m$$
$$C_R^*(f,g) \sim t^2 \cdot (d_1 \cdot \ldots \cdot d_m)^2.$$

Wir können auf dieser Basis die auch durch vielfältige Erfahrungen mit konkreten Rechnungen bestätigte Regel formulieren, daß

> der Aufwandszuwachs beim Übergang von einem Koeffizientenbereich mit konstanten Arithmetikkosten (`float` oder modular) zum Koeffizientenbereich $\mathbf{Z}$ mit dem Aufwandszuwachs bei der Vergrößerung der Anzahl der Variablen um 1 gleichgesetzt werden kann.

# 4 Lineare Algebra

In diesem Abschnitt wollen wir kurz das Lösen linearer Gleichungssysteme mit Einträgen aus einem Grundbereich $R$ betrachten, wobei der Schwerpunkt auf dem Einfluß liegt, den verschiedene Kostenmodelle verursachen.

Ein solches $n$-reihiges (oBdA quadratisches) Gleichungssystem $A \cdot \mathbf{x} = \mathbf{b}$ kann man mit verschiedenen Verfahren lösen: Gaußalgorithmus, Cramersche Regel, Berechnen der zu $A$ inversen Matrix.

Betrachten wir zunächst Gleichungssysteme über einem arithmetischen Grundbereich mit Einheitskostenarithmetik und vergleichen diese drei Herangehensweisen:

## 4.1 Lineare Algebra unter Einheitskostenarithmetik

Ein zentrales Problem, das bei den einzelnen Verfahren immer wieder auftritt, ist die Berechnung verschiedener Determinanten. Wir wollen zuerst untersuchen, mit welchem Aufwand dieses Teilproblem gelöst werden kann.

### Die Determinante

1) Die Determinante kann man mit der Definition berechnen. Ergibt $n!$ Summanden, jeder ein Produkt aus $n$ Faktoren, also Komplexität $exp(O(n \log(n)))$ (Stirlingformel für $n!$).

2) Dasselbe gilt in etwa, wenn man die Determinante nach der Leibnizregel durch (rekursives) Entwicklen nach einer Zeile oder Spalte bestimmt.

> AUFGABE: Zeigen Sie, daß beim Berechnen einer $n$-reihigen Determinante mit der Leibnizregel in schlechtesten Fall etwa $e \cdot n!$ Multiplikationen auszuführen sind.

Die Determinante kann man aber auch als Nebenprodukt des Gaußalgorithmus gewinnen. Es stellt sich heraus, daß dies der effizienteste Weg ist, der Teil hier also teurer als das Ganze zu sein scheint.

### Der Gaußalgorithmus

Im Gaußalgorithmus wird die Matrix $A$ Schritt für Schritt mit Hilfe von Pivotelementen auf Dreiecksform gebracht. Eine Mapleprozedur für Schritt $i$ in diesem Verfahren hat etwa folgende Gestalt:

```
rstep:=proc(B,i) local n,j,k,A; A:=copy(B); n:=matsize(A);
    for k from i+1 to n do A[i,k]:=A[i,k]/A[i,i] od;
    A[i,i]:=1;
    for j from i+1 to n do for k from i+1 to n do
        A[j,k]:=A[j,k]-A[i,k]*A[j,i] od od;
    for j from i+1 to n do A[j,i]:=0 od;
    eval(A)
end;
```

Abzählen zeigt, daß dabei $(n-i)(n-i+1)$ Multiplikationen oder Divisionen auszuführen sind.

**Satz 13** *Der Gaußalgorithmus benötigt auf einer n-reihigen Matrix $O(n^3)$ Multiplikationen oder Divisionen, um die Matrix zu triangulieren.*

Der Gaußalgorithmus ist also bei Einheitskostenarithmetik ein billigeres Verfahren zur Determinantenberechnung und damit das gängige Verfahren zum Lösen linearer Gleichungssysteme.

### Inversenbestimmung einer Matrix

Die Inverse einer Matrix kann über die Adjunktenformel bestimmt werden. Dazu sind $n^2+1$ Determinanten von je $(n-1)$-reihigen Matrizen zu berechnen. Gesamtaufwand, wenn man die Determinanten mit dem Gaußalgorithmus bestimmt, ist also $O(n^5)$.

Statt dessen kann man das Triangulierungsverfahren, das beim Gaußalgorithmus Anwendung findet, modifizieren, und die Ausgangskonfiguration $(A|E)$ mit der Einheitsmatrix auf der rechten Seite zur Konfiguration $(E|A^{-1})$ umformen. Dazu braucht man dann auch nur noch $O(n^3)$ Multiplikationen oder Divisionen. Obige $n^2$ Adjunkten hängen nämlich alle irgendwie miteinander zusammen.

> AUFGABE: Führen Sie die entsprechenden Rechnungen zur Komplexitätsabschätzung genau aus.

Das auf dieser Basis entwickelte Verfahren ist also etwa genauso gut wie der Gaußalgorithmus selbst.

**Satz 14** *Über einem Grundbereich mit Einheitskostenarithmetik können die wichtigsten Fragestellungen, die mit einer quadratischen n-reihigen Matrix A verbunden werden können (insbesondere Berechnung der Determinante und der Inversen) mit $O(n^3)$ Multiplikationen oder Divisionen bewältigt werden.*

## 4.2 Der Gaußalgorithmus über den rationalen Zahlen

Experimente mit zufälligen Matrizen und obiger Prozedur `rstep` zeigen ein relativ unangenehmes Koeffizientenwachstum, d.h. der Rechenaufwand wird zum Ende des Algorithmus immer größer. Was kann man über dieses Wachstum aussagen?

Kostenabschätzung, wenn sich bei den rationalen Operationen nichts wegkürzt: In jedem Schritt verdoppelt sich die Bitlänge der entsprechenden Zahlen. Gesamtaufwand also von der Größe

$$\sum_{k=1}^{n-1} (n-k)^2 (2^{k-1} l)^2 = (3\,2^n - n^2 - 2\,n - 3) l^2,$$

also exponentiell in der Anzahl der Zeilen der Matrix.

Diese Aussage war unter der Annahme getroffen worden, daß sich keine gemeinsamen Faktoren unterwegs herauskürzen lassen. Schauen wir auf das Wachstum in realen Beispielen, so vermuten wir allerdings, daß dies geschehen kann. Schauen wir uns die Situation einmal in einer generischen Matrix an. Wir nehmen dazu eine generische 4-reihige Matrix und führen die Schritte des Gaußalgorithmus auf dieser aus. Wir sehen, daß nach entsprechender Simplifikation der Grad von Zähler und Nenner der entsprechenden rationalen Ausdrücke offensichtlich nur linear wächst.

Da es schwierig ist, den zusätzlichen Aufwand für die gcd-Berechnung, der sich ja auch bei der Länge der Zwischenergebnisse benerkbar macht, abzuschätzen, wollen wir zunächst folgende *nennerfreie Version des Gaußalgorithmus* studieren:

```
nstep:=proc(B,i) local n,j,k,A; A:=copy(B); n:=matsize(A);
    for j from i+1 to n do for k from i+1 to n do
        A[j,k]:=A[j,k]*A[i,i]-A[i,k]*A[j,i] od od;
    for j from i+1 to n do A[j,i]:=0 od;
    eval(A)
end;
```

Betrachten wir wieder Beispiele mit zufälligen Matrizen, so erkennen wir hier deutlich das exponentielle Wachstum.

Untersuchen wir eine generische $n$-reihige Matrix $A$ und zerlegen die Einträge in Faktoren, so erkennen wir, daß dort systematisch Faktoren enthalten sind. Wir bezeichnen dazu mit $M_k(i,j)$ die Teilmatrix aus $A$, die aus den Elementen mit den Zeilennummern $1, 2, \ldots, k, i$ und den Spaltennummern $1, 2, \ldots, k, j$ besteht und mit $D_k(i,j)$ deren Determinante.

Die Struktur unserer Beispielrechnungen läßt folgenden Satz vermuten:

**Satz 15** *Es gilt*

$$D_{k-1}(k,k) \cdot D_{k-1}(m,n) - D_{k-1}(m,k) \cdot D_{k-1}(k,n) = D_{k-2}(k-1,k-1) \cdot D_k(m,n).$$

Beweis mit Evidenz, siehe `bsp3`.

Damit können wir in jedem Schritt des nennerfreien Gaußalgorithmus das im vorletzten Schritt verwendete Pivotelement wieder herausdividieren. Wir erhalten damit den folgenden *Bareiss-Algorithmus*:

```
bareiss:=proc(B) local n,p,i,j,k,A;
  A:=copy(B); n:=matsize(A);
  for i from 1 to n-1 do
      if i<2 then p:=1 else p:=A[i-1,i-1] fi;
      for j from i+1 to n do for k from i+1 to n do
          A[j,k]:=normal((A[j,k]*A[i,i]-A[i,k]*A[j,i])/p);
      od od;
      for j from i+1 to n do A[j,i]:=0 od;
  od;
  eval(A)
end;
```

Auf einer generischen Matrix als Eingabe enthält die Dreiecksform des Bareissverfahrens an der Stelle $(i,j)$ den Eintrag $D_i(i,j)$. Insbesondere steht für eine quadratische Matrix $A$ an der Stelle $(n,n)$ von `bareiss(A)` die Determinante von $A$. Dieses Verfahren kann auch auf Matrizen mit Einträgen aus einem Polynomring $R = k[x_1,\ldots,x_m]$ angewendet werden, da dort ebenfalls eine exakte Division ausgeführt werden kann.

Für ganzzahlige Matrizen wächst die Bitlänge der Einträge der Matrizen, die in den verschiedenen Schritten im Bareissalgorithmus entstehen, nur noch linear: Haben die Einträge in der Ausgangsmatrix die Länge $l$, so wird im Schritt $i$ zweimal das Produkt zweier Zahlen der Länge $i \cdot l$ berechnet und danach ein Faktor der Länge $(i-1)\,l$ abgespalten. Das Ergebnis hat dann die Bitlänge $(i+1)\,l$. Im Schritt $i$ sind insgesamt $(n-i)^2$ solcher Operationen auszuführen, was sich zu einem Gesamtaufwand von $(n-i)^2(3i^2-1)l^2$ summiert. Für den Bareissalgorithmus erhalten wir damit eine Komplexität

$$\sum_{i=1}^{n-1} (n-i)^2(3i^2-1)l^2 = O(n^5\,l^2).$$

**Satz 16** *Der Aufwand, eine $n$-reihige ganzzahlige Matrix, deren Einträge eine Bitlänge der Größenordnung $l$ haben, mit dem Bareissverfahren zu triangulieren, ist von der Größenordnung $O(n^5\,l^2)$.*

> AUFGABE: Zeigen Sie, daß der Aufwand für $n$-reihige Matrix mit Einträgen aus $R = k[x_1,\ldots,x_m]$ vom Grad $(d_1,\ldots,d_m)$ von der Größenordnung $O(n^{2m+3}\,(d_1\cdot\ldots\cdot d_m)^2)$ ist. (Hinweis: Zeigen Sie, daß die Polynome im Schritt $i$ des Bareissverfahrens den Gradvektor $(i\,d_1,\ldots,i\,d_m)$ haben)

## 4.3 Ein modulares Determinantenverfahren

Statt die obigen recht komplizierten Rechnungen über den ganzen Zahlen auszuführen, könnten wir diese auch zunächst über geeigneten Restklassenkörpern ausführen und dann versuchen, aus den Resten, die sich über verschiedenen solchen Bereichen ergeben, mit Hilfe des Chinesischen Restklassensatzes auf das korrekte Ergebnis über den ganzen Zahlen zu schließen.

Wir wollen dies hier nur für die Determinante der $n$-reihigen Matrix $A$ betrachten, da wir gesehen hatten, daß sich im Ergebnis des Bareiss-Algorithmus eine Dreiecksform ergibt, deren $O(n^2)$ Einträge gerade die Determinanten gewisser Teilmatrizen von $A$ sind.

Wir nutzen dabei aus, daß $det(A \bmod p) = det(A) \bmod p$ gilt, d.h. sich der Rest der Determinante bei Division durch $p$ auch als Determinante $det(A)$ im Restklassenring $\mathbf{Z}_p$ bestimmen läßt. Ist dabei $p$ eine Primzahl der Wortlänge 1, d.h. die entsprechenden Arithmetikkosten Einheitskosten, so beträgt der Aufwand zur Berechnung dieser Determinante wie oben gezeigt $O(n^3)$. Ist $M$ eine Schranke für $|det(A)|$, also $-M < det(A) < M$, so genügt es nach dem CRT, diese Rechnungen für Primzahlen $p_1,\ldots,p_N$ auszuführen, für die $p_1 \cdot \ldots \cdot p_N \geq 2M$ gilt, und dann die vom Betrag her kleinste Restklasse des Liftungsproblems zu finden. Das folgende Programm bestimmt die Determinante bis zu einer Schranke $M = 3234846615$ korrekt:

```
moddet:=proc(A) local primes,u,i;
  primes:=[2,3,5,7,11,13,17,19,23,29];
  u:=[seq([Det(A) mod op(i,primes),op(i,primes)],i=1..nops(primes))];
  CRA(u);
end;
```

Für Anwendungen ist die Größe der Determinante vorab allerdings nicht bekannt, so daß auch nicht klar ist, wie viele Primzahlen benötigt werden. Eine grobe Abschätzung der Determinante hilft aber weiter:

**Satz 17 (Schranke von Hadamard)** *Ist $A = ||a_{ij}||$ eine $n$-reihige Matrix, so gilt für die Determinante*

$$|det(A)| \leq \prod_{i=1}^{n} \sqrt{\sum_{j=1}^{n} a_{ij}^2}.$$

*Ist insbesondere $A$ eine ganzzahlige Matrix mit Einträgen der Bitlänge $l$, so gilt für die Bitlänge der Determinante*

$$l(det(A)) \leq n\,(\log(n) + l).$$

Der Beweis folgt etwa aus der Interpretation der Determinante als das Volumen des $n$-dimensionalen Parallelepipeds, das von den Zeilenvektoren von $A$ aufgespannt wird und welches nicht größer als das Produkt der Längen dieser Vektoren ist.

Für die modulare Version der Determinantenberechnung erhalten wir damit

**Satz 18** *Der Aufwand, die Determinante einer $n$-reihigen ganzzahligen Matrix, deren Einträge eine Bitlänge der Größenordnung $l$ haben, mit dem modularen Verfahren zu bestimmen, ist von der Größenordnung $O(n^4\ \log(n) + n^4\, l + n^2\, l^2)$.*

In der Tat, es sind $N = O(n\,(\log(n) + l))$ Determinanten-Reste zu bestimmen, jeder mit dem Aufwand $O(n^3)$ und dann diese $N$ Reste modulo verschiedener Primzahlen mit `CRA` zu liften, was einen Aufwand von $O(N^2)$ verursacht.

> AUFGABE: Leiten Sie daraus eine Schranke für eine modulare Version des Bareiss-Algorithms her.

# 5 Polynomiale gcd-Berechnung

Die Berechnung polynomialer größter gemeinsamer Teiler (gcd) ist eine der zentralen Aufgaben jedes Computeralgebrasystems. Hätte man keine solche Simplifikationsmöglichkeit zur Verfügung, so würde sich etwa bei der Addition rationaler Funktionen nach der Formel

$$\frac{a(x)}{b(x)} + \frac{c(x)}{d(x)} = \frac{a(x)d(x) + b(x)c(x)}{b(x)d(x)}$$

in jedem Schritt Zähler- und Nennergrad verdoppeln.

## 5.1 Allgemeine Teilbarkeitstheorie

Bevor wir uns der gcd-Berechnung von Polynomen widmen, seien die wichtigsten Begriffe und Eigenschaften zur Teilbarkeit noch einmal zusammengestellt. Wir fixieren dazu einen Integritätsbereich $D$.

Für $a, b \in D$ ist $a$ ein Teiler von $b$, wenn ein $c \in D$ mit $b = a \cdot c$ existiert. Wir schreiben $a|b$.

Gilt $a|b$ und $b|a$, so unterscheiden sich die beiden Elemente um einen in $D$ invertierbaren Faktor $c \in D^*$. Solche Elemente heißen *zueinander assoziiert*. Wir schreiben $a \sim b$.

Aus Sicht der Teilbarkeit kann man solche Elemente nicht unterscheiden. Ist z.B. $D = k[x]$ ein univariater Polynomring, also $D^* = k$ die Menge der konstanten Polynome, so sind Teilbarkeitsfragen nur bis auf einen solchen konstanten Faktor entscheidbar.

Ein Element $g \in D$ bezeichnet man als *größten gemeinsamen Teiler* (gcd) der Elemente $a, b \in D$, wenn

1. $g|a,\ g|b$ und
2. $d|a,\ d|b\ \Rightarrow\ d|g$ für alle $d \in D$

gilt. Ein gcd muß nicht existieren und auch nicht eindeutig sein. Allerdings sind zwei gcd $g_1$ und $g_2$ von vorgegebenen Elementen zueinander assoziiert, denn aus der zweiten Bedingung folgt $g_1|g_2$ und auch $g_2|g_1$.

> AUFGABE: Definieren Sie analog den Begriff des kleinsten gemeinsamen Vielfachen $lcm(a, b)$, zeigen Sie, daß ein solches lcm eindeutig bis auf assoziierte Elemente bestimmt ist und daß $\frac{a\,b}{gcd(a,b)}$ ein solches lcm ist.

## 5.2 Univariate Polynome über einem Integritätsbereich

In der rekursiven Darstellung hatten wir multivariate Polynome als univariate Polynome betrachtet, deren Koeffizienten selbst wieder Polynome sind, allerdings eine Variable weniger enthalten. Wir wollen diese Situation jetzt unter dem Aspekt von Teilbarkeitsfragen näher untersuchen. Sei also $R$ ein Integritätsbereich, $D = R[x]$ der entsprechende univariate Polynomring und schließlich $K = Q(R)$ der Quotientenkörper von $R$. Wir wollen im weiteren untersuchen, wie Teilbarkeitsfragen in $R$, in $R[x]$ und in $K[x]$ zusammenhängen, wobei wir davon ausgehen, daß in $R$ gcd existieren und auch effektiv berechnet werden können.

$K[x]$ ist als univariater Ring über einem Körper ein Euklidscher Ring, in dem man ähnlich wie über den ganzen Zahlen, eine Division mit Rest und darauf aufbauend den Euklidschen Algorithmus zur gcd-Berechnung zur Verfügung hat.

Für $f = a_0 + a_1 x + \ldots + a_n x^n \in R[x]$ bezeichnet man

$a := gcd(a_0, \ldots, a_n)$ den *Inhalt* $cont(f)$ von $f$,
$pp(f) := \frac{a_0}{a} + \frac{a_1}{a}x + \ldots + \frac{a_n}{a}x^n$ den *primitiven Teil* von $f$.
$f$ heißt schließlich *primitiv*, wenn $cont(f) \sim 1$ gilt.

Ist $f(x) \in K[x]$ ein Polynom mit rationalen Koeffizienten, so können wir durch Multiplikation mit einem geeigneten Nenner $r \in R$ erreichen, daß $r \cdot f(x) \in R[x]$ nennerfrei ist und damit auch den primitiven Teil $pp(f) := pp(r \cdot f)$ von $f$ definieren. Ebenso erhalten wir einen Inhalt $cont(f) := \frac{cont(r \cdot f)}{r} \in K$.

> AUFGABE: Zeigen Sie, daß $cont(f)$ und $pp(f)$ stets eindeutig bis auf eine Einheit $c \in R^*$ bestimmt sind und daß obige Defintion für $f(x) \in K[x]$ nicht vom gewählten Nenner $r$ abhängt.

Eine zentrale Rolle in der Teilbarkeit univariater Polynome spielt das folgende

**Lemma 2 (Gauß-Lemma)** *Über einem faktoriellen Ring $R$ ist das Produkt zweier primitiver Polynome wieder ein primitives Polynom.*

> AUFGABE: Beweisen Sie die Folgerungen:
>
> 1. $\forall\ f, g \in R[x]\ cont(fg) \sim cont(f)cont(g)$,
> 2. $\forall\ f, g \in R[x]\ pp(fg) \sim pp(f)pp(g)$,
> 3. $\forall\ f, g \in R[x]\ g|f\ \Rightarrow\ cont(g)|cont(f),\ pp(g)|pp(f)$,
> 4. $\forall\ f \in R[x], r \in R\ cont(rf) \sim r \cdot cont(f),\ pp(rf) \sim pp(f)$,
> 5. $\forall\ f, g \in K[x]\ g|f$ in $K[x]\ \Rightarrow\ pp(g)|pp(f)$ in $R[x]$

**Satz 19** *Die gcd-Berechnung von $f, g \in R[x]$ kann man auf die in $R$ und in $K[x]$ zurückführen. Es gilt*

$$gcd_{R[x]}(f,g) = gcd_R(cont(f), cont(g)) \cdot pp(gcd_{K[x]}(f,g)).$$

BEWEIS : Wir haben nur zu zeigen, daß die rechte Seite ein gcd von $f$ und $g$ ist. Diese rechte Seite besteht aus den Faktoren $r = gcd_R(cont(f), cont(g)) \in R$ und $h = pp(gcd_{K[x]}(f,g)) \in R[x]$.

$r|cont(f)$ und $h = pp(h)|pp(f)$, also ist $r\,h$ ein Teiler von $f \sim cont(f)\,pp(f)$ und analog von $g$.

Ist umgekehrt $d \in R[x]$ ein gemeinsamer Teiler von $f$ und $g$, so gilt $cont(d)|cont(f)$, $cont(g)$ und folglich $cont(d)|gcd(cont(f), cont(g))$. Wegen $d|f, g$ gilt $d|h$ wenigstens in $K[x]$ und damit wieder $pp(d)|pp(h) = h$ in R[x]. □

Damit spielt die gcd-Berechnung von univariaten Polynomen über einem Körper und dabei der Euklidsche Algorithmus eine zentrale Rolle. Es stellt sich allerdings heraus, daß der Euklidsche Algorithmus in seiner einfachen Form für die direkte Anwendung in komplexeren Aufgabestellungen nicht schnell genug ist. In den letzten 25 Jahren wurden eine Reihe von Verbesserungen gefunden, die im folgenden diskutiert werden sollen.

## 5.3 Der Euklidsche Algorithmus für Polynome

Für univariate Polynome über einem Körper $k$ kann man den Euklidschen Algorithmus als eine Folge von Divisionen mit Rest betrachten. Eine solche Division mit Rest für Polynome $f, g$ geht wie beim schriftlichen Dividieren von $r := f$ aus und führt in jedem Schritt die Ersetzung

$$r \;\mapsto\; r - \frac{lc(r)}{lc(g)}\, g\, x^{d(r)-d(g)}$$

aus, wodurch der Leitterm von $r$ weggehoben wird.

```
divrem := proc(f,g,x) local c,q,r,d;
  r:=f; q:=0;
  d:=degree(r,x)-degree(g,x);
  while r<>0 and d>=0 do
      c:=lcoeff(r,x)/lcoeff(g,x)*x^d;
      q:=q+c; r:=normal(r-c*g);
      d:=degree(r,x)-degree(g,x);
  od;
  [q,collect(r,x)];
end;

euklid := proc(a,b,x) local f,g,r;
  f:=a; g:=b;
  while g<>0 do
      r:=op(2,divrem(f,g,x)); f:=g; g:=r;
  od;
  f;
end;
```

Beispiel: $k = \mathbf{Z}_{23}$. Wir betrachten die Polynome

$$f := x^8 + x^6 - 3x^4 - 3x^3 + 8x^2 + 2x - 5 \qquad g := 3x^6 + 5x^4 - 4x^2 - 9x + 21$$

und erhalten nacheinander

$$\begin{aligned}
&3x^6 + 5x^4 - 4x^2 - 9x + 21\\
&2x^4 - 5x^2 - 8\\
&-x^2 - 9x + 2\\
&-8 + 10x\\
&-4
\end{aligned}$$

Also sind $f$ und $g$ relativ prim im Polynomring $\mathbf{Z}_{23}[x]$.

Über einem solchen Körper mit Einheitskostenarithmetik können wir die Komplexität durch den Grad $d(f)$ und $d(g)$ der eingehenden Polynome messen und erhalten genau wie bei den entsprechenden Algorithmen für ganze Zahlen

$C_{divrem} = O(d(g)\,(d(f) - d(g))$ für die Komplexität der Division mit Rest und

$C_{gcd} = O(d(f) \cdot d(g))$ für die Komplexität des Euklidschen Algorithmus.

Führt man dieselben Rechnungen im Ring $\mathbf{Q}[x]$ aus, so erhält man

$$3\,x^6 + 5\,x^4 - 4\,x^2 - 9\,x + 21$$
$$-\frac{5\,x^4}{9} + \frac{x^2}{9} - \frac{1}{3}$$
$$-\frac{117\,x^2}{25} - 9\,x + \frac{441}{25}$$
$$-\frac{102500}{6591} + \frac{233150\,x}{19773}$$
$$-\frac{1288744821}{543589225}$$

Wiederum erhalten wir als Ergebnis, daß $f$ und $g$ relativ prim sind, stellen aber ein unverhältnismäßiges Wachstum der an der Rechnung beteiligten Koeffizienten fest. In jedem Schritt wurden außerdem eine Reihe von gcd-Berechnungen im Bereich der ganzen Zahlen ausgeführt, denn in den Ergebnissen liegen stets gekürzte Brüche als Koeffizienten vor.

Betrachten wir nun Polynome in mehreren Veränderlichen.
Beispiel: $R = \mathbf{Q}[x, y]$,

$$f := (y+1)\,x^3 + (y-1)\,x + y, \qquad g := (y+2)\,x^3 + (y+3)\,x^2 + 1$$

Der Euklidsche Algorithmus liefert in diesem Fall

$$(y+2)\,x^3 + (y+3)\,x^2 + 1$$
$$-\frac{y^2+4\,y+3}{y+2}x^2 - \frac{-y+2-y^2}{y+2}x - \frac{-y-y^2+1}{y+2}$$
$$\frac{-30\,y-16+11\,y^2+37\,y^3+19\,y^4+3\,y^5}{(y^2+4\,y+3)^2}x + \frac{4+14\,y+2\,y^5+13\,y^4+32\,y^2+31\,y^3}{(y^2+4\,y+3)^2}$$
$$-\frac{\left(y^2+4\,y+3\right)^2\left(y^6+16\,y^5+54\,y^4+49\,y^3-9\,y^2+4\,y+12\right)}{(3\,y^4+13\,y^3+11\,y^2-11\,y-8)(-30\,y-16+11\,y^2+37\,y^3+19\,y^4+3\,y^5)}$$

Auch hierbei sind rekursiv gcd-Berechnungen erfolgt. Wir wollen den dabei entstehenden Aufwand unter plausiblen Annahmen abschätzen. Wir betrachten dazu die gcd-Berechnung durch rekursive Anwendung des Euklidschen Algorithmus im Ring $R = k(x_1, \ldots, x_{n-1})[x_n]$ (über einem Körper $k$ mit Einheitskostenarithmetik) auf Polynome vom Grad $< d$ in $x_n$. Bezeichne $C(d, n)$ die gesuchte Komplexität (also insbesondere $C(d, 0) = O(d^2)$). Sei $b(d, k)$ eine Schranke für den Grad der rationalen Funktionen aus $k(x_1, \ldots, x_{n-1})$, die als Zähler und Nenner der Koeffizienten im $k$-ten Schritt auftreten. Da der im Schritt $k$ erzeugte Rest einen Grad $d_n(r_k) < d - k$ besitzt, fallen also gcd-Berechnungen für etwa $d - k$ Koeffizienten (deren Zähler und Nenner Polynome in $n - 1$ Variablen vom Grad $b(d, k)$ sind) an. Wir erhalten so die Rekursionsformel

$$C(d, n) := \sum_{k=1}^{d} (d - k) * C(b(d, k, n), n - 1).$$

Eine sehr optimistische Annahme, die selbst in obigem Beispiel nicht erfüllt ist, geht davon aus, daß Zähler- und Nennergrad in jedem Schritt nur um einen konstanten Betrag, etwa 1, zunehmen.

Setzen wir entsprechend $b(d,k) = d + k$, so erhalten wir

$$C(d,2) = \frac{11}{12}\,d^4 + \frac{d^3}{2} - \frac{5}{12}\,d^2$$
$$C(d,3) = \frac{209}{120}\,d^6 + \frac{133}{120}\,d^5 - \frac{19}{12}\,d^4 + \ldots$$
$$C(d,4) = \frac{51623}{6720}\,d^8 + \frac{323}{80}\,d^7 - \frac{399}{32}\,d^6 + \ldots$$

und allgemein $C(d,n) = O(d^{2n})$. Die Anzahl der Variablen geht also bereits hier exponentiell ein.

Eine weniger optimistische Annahme geht von linearem Wachstum der Grade aus, was noch immer bedeutet, daß sich aus Zähler und Nenner in späteren Schritten bedeutende Teile gegenüber einem nicht simplifizierenden Verfahren wegkürzen müßten. Setzen wir $b(d,k) = d * k$, so erhalten wir

$$C(d,2) = \frac{d^6}{12} - \frac{d^4}{12}$$
$$C(d,3) = \frac{d^{14}}{672} - \frac{d^{12}}{144} + \frac{11}{1440}\,d^{10} + \frac{d^8}{504} - \frac{d^6}{240}$$
$$C(d,4) = \frac{d^{30}}{161280} - \frac{d^{28}}{8064} + \frac{127}{116480}\,d^{26} + \ldots$$

und allgemein $C(d,n) = O(d^{c(n)})$ mit $c(n) = 2 \cdot c(n-1) + 2$, also $C(d,n) = O(d^{2^n})$. Selbst in diesem noch relativ guten Szenario geht die Anzahl der Variablen bereits doppelt exponentiell ein.

AUFGABE: Finden Sie eine explizite Formel für $c(n)$.

## 5.4 Polynomiale Restsequenzen

Um zu effektiveren Verfahren zu gelangen, wollen wir versuchen, die Rechnungen ähnlich wie beim Bareiss-Algorithmus nennerfrei auszuführen.

Seien dazu $f(x), g(x) \in R[x]$. Betrachten wir die Division mit Rest $divrem(f,g)$. In der klassischen Version über $K$ entsteht als Nenner eine Potenz von $lc(g)$, weil in jedem Schleifendurchlauf durch diesen Leitkoeffizienten geteilt wird. Um Divisionen zu vermeiden, modifizieren wir dazu den Ersetzungsschritt in der Division mit Rest zu

$$r \;\mapsto\; lc(g)\,r - lc(r)\,g\,x^{d(r)-d(g)}$$

Im Ergebnis erhalten wir eine Darstellung

$$lc(g)^l f(x) = q(x) \cdot g(x) + r(x),$$

wobei $l$ die Anzahl der Ersetzungsschritte angibt. Eine entsprechende Mapleprozedur hat folgendes Aussehen:

```
pdivrem1 := proc(f,g,x) local c,q,r,d;
  r:=f; q:=0;
  d:=degree(r,x)-degree(g,x);
  while r<>0 and d>=0 do
      c:=lcoeff(r,x)*x^d;
      q:=lcoeff(g,x)*q+c; r:=normal(lcoeff(g,x)*r-c*g);
      d:=degree(r,x)-degree(g,x);
  od;
  [q,collect(r,x)];
end;
```

Da in jedem Ersetzungsschritt der Grad von $r$ um wenigstens 1 sinkt, gilt

$$l \le d := deg(f) - deg(g) + 1,$$

so daß wir im weiteren für unsere theoretischen Untersuchungen $l$ durch $d$ ersetzen wollen, was durch eine entsprechende Nachjustierung in obiger Prozedur erfolgen kann:

```
pdivrem := proc(f,g,x) local c,q,r,d,n;
  r:=f; q:=0;
  d:=degree(r,x)-degree(g,x); n:=d+1;
  while r<>0 and d>=0 do
      c:=lcoeff(r,x)*x^d;
      q:=lcoeff(g,x)*q+c; r:=normal(lcoeff(g,x)*r-c*g);
      d:=degree(r,x)-degree(g,x); n:=n-1;
  od;
  [lcoeff(g,x)^n*q,collect(lcoeff(g,x)^n*r,x)];
end;
```

DEFINITION: Ist $R$ ein Integritätsbereich und $f(x), g(x) \in R[x], deg\ f \geq deg\ g$, so gibt es eindeutig bestimmte Polynome $q(x), r(x) \in R[x]$ mit $r = 0$ oder $deg\ r < deg\ g$ und

$$lc(g)^{deg(f)-deg(g)+1} f(x) = q(x) \cdot g(x) + r(x).$$

$q$ nennt man den *Pseudoquotienten* $pquot(f, g)$ und $r$ nennt man den *Pseudorest* $prem(f, g)$ der Polynome $f$ und $g$.

Über $K[x]$ sind der übliche Rest und der Pseudorest zueinander assoziierte Polynome. Also kann man bei der Berechnung von $gcd_{K[x]}(f, g)$ mit Hilfe des Euklidschen Algorithmus die Restbildung durch Pseudorestbildung ersetzen. Die Folge der dabei entstehenden Reste bezeichnet man als die *Euklidsche Polynomiale Restsequenz*. Eine Maple-Implementierung sieht so aus:

```
eprs := proc(a,b,x) local f,g,r;
  f:=a; g:=b;
  while g<>0 do
     r:=op(2,pdivrem(f,g,x));
     f:=g; g:=r;
  od;
  f;
end;
```

AUFGABE: Führen Sie einige Beispielrechnungen mit dieser Implementierung aus, etwa mit den Polynomen

$$f_1 := x^8 + x^6 - 3\,x^4 - 3\,x^3 + 8\,x^2 + 2\,x - 5, g_1 := 3\,x^6 + 5\,x^4 - 4\,x^2 - 9\,x + 21$$
$$f_2 := 3\,x^6 + x^3 + x + 5, \quad g_2 := 5\,x^5 - 3\,x^4 + 1$$
$$f_3 := (y+1)\,x^3 + (y-1)\,x + y, \qquad g_3 := (y+2)\,x^3 + (y+3)\,x^2 + 1$$

Das Ergebnis ist im Hinblick auf Vermeidung von Rechenaufwand auf den ersten Blick allerdings nicht sehr ermutigend, denn die entstehenden Koeffizienten sind deutlich größer als die, die bei der rationalen Version auftreten. Eine genauere Analyse der in den Restsequenzen auftretenden Koeffizienten zeigt wie bereits im Fall linearer Gleichungssysteme, daß sich diese gewöhnlich aus vielen Faktoren zusammensetzen. Es entsteht also die Frage, ob man Faktoren finden kann, die in allen Koeffizienten eines Restpolynoms gemeinsam vorkommen. Diese könnte man dann ausdividieren, bevor die Rechnung fortgesetzt wird.

DEFINITION: Seien $f(x), g(x) \in R[x]$ zwei Polynome mit $deg\ f \geq deg\ g$. Als *Polynomiale Restsequenz* (PRS) bezeichnet man eine Folge von Polynomen

$$R_0 = f,\ R_1 = g,\ R_2,\ R_3,\ \ldots,\ R_k \neq 0$$

mit

(1) $deg\ R_1 > deg\ R_2 > \ \ldots, > deg\ R_k$.

(2) $\alpha_i \cdot R_{i-1}(x) = Q_i(x) \cdot R_i(x) + \beta_i \cdot R_{i+1}(x)$
mit geeigneten $\alpha_i, \beta_i \in R$ und $Q_i(x) \in R[x]$.

(3) $prem(R_{k-1}, R_k) = 0$

Wie oben gilt dann stets $pp(R_k) = pp(gcd_{K[x]}(f,g))$.

Setzen wir $\delta_i := deg(R_{i-1}) - deg(R_i)$, so kann man wie bei der Pseudodivision $\alpha_i = lc(R_i)^{\delta_i+1}$ nehmen.

Zur Komplexität dieser Berechnungen: Seien $f, g \in \mathbf{Z}[x]$ zwei Polynome, deren Koeffizienten eine durchschnittliche Länge $l$ haben. Aus der Berechnungsvorschrift folgt

$$l_{i+1} = l(\alpha_i) + l_{i-1} - l(\beta_i) = (\delta_i + 1)l_i + l_{i-1} - l(\beta_i)$$

mit $l_0 = l_1 = l, l_2 = (\delta_1 + 1)l$, wenn $l_i := l(lc(R_i))$ die Länge des Leitkoeffizienten des Rests $R_i$ bezeichnet.

AUFGABE: Zeigen Sie, daß eine analoge Beziehung für $f, g \in k[x_1, \ldots, x_n] = R[x]$ mit $R = k[x_1, \ldots, x_{n-1}]$ gilt, wenn $k$ ein Körper mit Einheitskostenarithmetik und $l(\alpha) := (d_1(\alpha), \ldots, d_{n-1}(\alpha))$ der Gradvektor von $\alpha \in R$ ist.

1) $\alpha_i = lc(R_i)^{\delta_i+1}, \beta_i = 1$ liefert die oben beschriebene *Euklidsche PRS*. Für den allgemeinen Fall $\delta_i = 1$ (d.h. der Grad in der Restsequenz fällt in jedem Schritt um genau 1; solche PRS heißen *normal*) ergibt sich deren Koeffizientenwachstum damit aus der Rekursion

$$l_{i+1} = 2l_i + l_{i-1} \quad \text{mit} \quad l_1 = l, l_2 = 2l$$

Wir erhalten $2\,l_i \leq l_{i+1} \leq 3\,l_i$, also exponentielles Koeffizientenwachstum.

AUFGABE: Finden Sie das genaue Wachstum $l_i = O(?)$

2) Man könnte auch in jedem Schritt $R_{i+1}$ durch seinen primitiven Teil ersetzen, also $\beta_i = cont(R_{i+1})$ ausdividieren. Das wäre der größtmögliche Faktor:

```
pprs := proc(a,b,x) local f,g,r;
  f:=a; g:=b;
  while g<>0 do
     r:=primpart(op(2,pdivrem(f,g,x)),x);
     f:=g; g:=r;
  od;
  f;
end;
```

Für den Rechenaufwand gilt allerdings das oben Gesagte, d.h. bei selbst weitgehenden Annahmen über ein langsames Koeffizientenwachstum ist die Laufzeit dieses Verfahrens ebenfalls exponentiell in der Koeffizientengröße und doppelt exponentiell in der Anzahl der Variablen, obwohl nicht so große Zahlen wie bei der Euklidschen PRS entstehen.

Es ergibt sich also die Frage, ob man ähnlich wie beim Bareiss-Algorithmus Faktoren, die man in früheren Schritten hineingesteckt hat, in späteren Schritten wieder herausdividieren kann, ohne in jedem Fall erst eine aufwendige gcd-Berechnung auf den Koeffizienten zu starten. Dies ist in der Tat möglich:

**Satz 20 (Silvester 1853, Collins 1967)**

$$\alpha_i = lc(R_i)^{\delta_i+1}, \qquad \beta_1 = 1, \beta_i = \alpha_{i-1} \text{ für } i > 1$$

*liefert eine PRS, die* reduzierte Polynomiale Rest-Sequenz.

Für einen Beweis dieses Satzes sei auf [1, ch. 7] bzw. [4] verwiesen. Von seiner Evidenz kann man sich an Hand der folgenden Maple-Implementierung überzeugen:

```
redprs := proc(a,b,x) local f,g,r,beta;
  f:=a; g:=b; beta:=1;
  while g<>0 do
     r:=normal(op(2,pdivrem(f,g,x))/beta);
     print(collect(f,x));
     beta:=lcoeff(g,x)^(degree(f,x)-degree(g,x)+1);
     f:=g; g:=r;
  od;
  f;
end;
```

Für die oben angegebenen Beispiele gehen die im Laufe der Rechnungen auszuführenden Divisionen jeweils auf.

> AUFGABE: Überzeugen Sie sich von der Richtigkeit des Satzes an Hand weiterer selbstgewählter Beispiele.

Für normale Restsequenzen, d.h. wenn $\delta_i = 1$ für alle $i > 1$ gilt, erhalten wir für das Koeffizientenwachstum

$$l_{i+1} = 2l_i + l_{i-1} - 2l_{i-1} = 2l_i - l_{i-1} \quad \text{mit} \quad l_1 = l, l_2 = 2l.$$

> AUFGABE: Zeigen Sie, daß in diesem Fall $l_i = i\,l$ gilt, d.h. daß das Koeffizientenwachstum linear ist.

Liegt dagegen eine Restsequenz mit größeren Gradsprüngen vor, so kann das Koeffizientenwachstum noch immer exponentiell sein.

> AUFGABE: Leiten Sie für den Fall, daß alle $\delta_i = 2$ sind, eine Formel für $l_i$ her und überzeugen Sie sich, daß diese exponentiell in $i$ ist.

Ein noch besseres Verhalten legt die *Subresultanten-PRS* an den Tag:

**Satz 21 (Collins 1967, Brown/Traub 1971)** $\alpha_i = lc(R_i)^{\delta_i+1}$ *und*

$$\begin{aligned} h_1 &= 1, \quad h_i = lc(R_{i-1})^{\delta_{i-1}} \cdot h_{i-1}^{1-\delta_{i-1}} \quad (i > 1), \\ \beta_1 &= 1, \quad \beta_i = lc(R_{i-1}) \cdot h_i^{\delta_i} \quad (i > 1) \end{aligned}$$

*definiert eine Polynomiale Restsequenz, die* Subresultanten-PRS.

Für einen Beweis dieses Satzes sei ebenfalls auf [1, ch. 7] bzw. [4] verwiesen. Mit folgender Maple-Implementierung kann man sich überzeugen, daß wiederum alle Divisionen aufgehen:

```
sprs := proc(a,b,x) local f,g,r,lc,h,delta;
  f:=a; g:=b; lc:=1; h:=1;
  print(collect(f,x)); print(collect(g,x));
  while g<>0 do
     delta:=degree(f,x)-degree(g,x);
     r:=normal(op(2,pdivrem(f,g,x))/(lc*h^delta));
     print(collect(r,x));
     lc:=lcoeff(g,x);
     h:=lc^delta*h^(1-delta);
     f:=g; g:=r;
  od;
  f;
end;
```

AUFGABE: Überzeugen Sie sich an verschiedenen selbstgewählten Beispielen, daß bei der angegebenen Wahl der Faktorreihe $\beta_i$ die Division immer aufgeht. Untersuchen Sie den Unterschied zwischen der Subresultanten-PRS und der reduzierten PRS, wenn die PRS nicht normal ist.

Bemerkung: Für normale Restsequenzen fällt die Subresultanten-PRS mit der reduzierten PRS zusammen. Im Gegensatz zu letzterer wachsen aber die Koeffizienten auch bei beliebigen Gradsprüngen nur linear.

AUFGABE: Zeigen Sie, daß beim Subresultantenalgorithmus das Koeffizientenwachstum immer linear ist, indem Sie aus den Rekursionsbeziehungen $l_i = (\delta_1 + \ldots + \delta_{i-1} + 1)\, l$ und $l(h_i) = l_i - l$ herleiten.

Unsere bisherigen Komplexitätsuntersuchungen hatten wir auf den Ring $\mathbf{Z}[x]$ konzentriert, wo die Länge der Koeffizienten bei Rechnungen einer einfachen Gesetzmäßigkeit folgt. Untersuchen wir deshalb zum Schluß die Komplexität der Rechnungen im Ring $R = k[x_1, \ldots, x_n]$ über einem Körper $k$ mit Einheitskostenarithmetik. Hier wird die Größe der Koeffizienten $l = (d_1, \ldots, d_{n-1})$ durch einen Gradvektor beschrieben, ihre Bitlänge dagegen durch das Produkt $L(l) := \prod_{i=1}^{n-1} d_i$. Letztere Größe geht auch in alle Komplexitätsformeln ein, wie wir im Kapitel 4 gesehen hatten. Wir wollen deshalb die Komplexität der Berechnung einer normalen PRS mit Hilfe des Algorithmus `redprs` untersuchen, der in diesem Fall mit `sprs` zusammenfällt.

Wir nehmen also an, daß $\delta_i = 1$, $\alpha_i = lc(R_i)^2$ und $l_i = i\, l$ für alle $i > 1$ gilt (die Koeffizientengröße $l_i$ im Polynom $R_i$ also linear wächst). Damit gilt $d_n(R_i) = d - i$ mit $d = d_n(f) = d_n(g) + 1$ und wir erhalten $R_{i+1}$ aus

$$\alpha_i \cdot R_{i-1} = Q_i \cdot R_i + \beta_i \cdot R_{i+1}$$

Die Berechnung erfolgt in drei Etappen:

Zuerst wird $R_{i-1}$ konditioniert. $\alpha_i \cdot R_{i-1}$ zu berechnen bedeutet, die $d-i+2$ Summanden von $R_{i-1}$ der Größe $l_{i-1}$ jeweils mit $\alpha_i$ der Größe $l(\alpha_i) = 2\, l_i$ zu multiplizieren.
Kosten: $(d - i + 2)\, L(l_{i-1})\, L(2\, l_i)$
Koeffizientengröße: $c_i = 2\, l_i + l_{i-1}$

Dann wird $Q_i$ berechnet. In einer normalen PRS ist jedes $Q_i$ vom Grad 1. Es zu berechnen ist so teuer wie das Multiplizieren $Q_i \cdot R_i$.
Kosten: $2\,(d - i + 1)\, L(c_i - l_i)\, L(l_i)$

Schließlich wird $\beta_i$ aus dem Rest koeffizientenweise ausdividiert.
Kosten: $(d - i)\, L(2\, l_{i-1})\, L(l_{i+1})$

Für die Gesamtkosten im Schritt $i$ erhalten wir also die Formel

$$s_i = (d - i + 2)\, L(l_{i-1})\, L(2\, l_i) + 2\,(d - i + 1)\, L(l_i + l_{i-1})\, L(l_i) + (d - i)\, L(2\, l_{i-1})\, L(l_{i+1})$$

$$= (d - i + 2)\, L((i-1)l)\, L(2il) + 2\,(d - i + 1)\, L((2i-1)\, l) L(i\, l) + (d - i)\, L(2\,(i-1)\, l)\, L((i+1)\, l)$$

und schließlich wegen $L(i\, l) = i^{n-1} L(l)$

$$s_i \sim 2^{n+1} (d - i + 1)\, i^{2n-2}\, L(l)^2$$

Summieren wir den Aufwand der einzelnen Schritte, und berücksichtigen, daß $L(f) = d_1 \cdot \ldots \cdot d_n = L(l) \cdot d$ gilt, so erhalten wir

$$\sum_{i=1}^{d-1} s_i = O(d^{2n-2} L(f)^2).$$

**Satz 22** *Sei $R[x_n]$ der univariate Polynomring über $R = k[x_1, \ldots, x_{n-1}]$ und einem Grundkörper $k$ mit Einheitskostenarithmetik und $f, g \in R[x_n]$ zwei Polynome mit durch $(d_1, \ldots, d_n)$ beschränkten Gradvektoren. Zur Berechnung der Subresultanten-PRS werden dann $O((d_1 \cdots d_n)^2 (d_n)^{2n-2})$ Elementarmultiplikationen benötigt.*

AUFGABE: Zeigen Sie, daß die entsprechende Komplexität über $R = \mathbf{Z}$ für Polynome mit Koeffizientenlängen, die durch $t(f), t(g) \leq b$ beschränkt sind, von der Größenordnung $O(b^2 d^4)$ und über $R = \mathbf{Z}[x_1, \ldots, x_{n-1}]$ von der Größenordnung $O((b \cdot d_1 \cdots d_n)^2 (d_n)^{2n})$ ist.

Dies bestätigt die oben beschriebene Heuristik, daß die Berücksichtigung der Koeffizientenlänge der Hinzunahme einer zusätzlichen Variablen entspricht, ein weiteres Mal.

Dies ist allerdings noch nicht der Gesamtaufwand für die Berechnung des gcd, da nach der Formel

$$gcd_{R[x]}(f, g) = gcd_R(cont(f), cont(g)) \cdot pp(gcd_{K[x]}(f, g))$$

noch die Berechnung sowohl des Inhalts-gcd als auch des primitiven Teils aussteht. Dies führt auf die Berechnung von maximal $3d_n$ gcd's in $n-1$ Variablen. Bezeichnet $C_{gcd}(d_1, \ldots, d_n)$ die Komplexität der gcd-Berechnung von Polynomen $f, g \in k[x_1, \ldots, x_n]$, deren Gradvektor durch $(d_1, \ldots, d_n)$ beschränkt ist, so erhalten wir für den Gesamtaufwand der gcd-Berechnung

$$C_{gcd}(d_1, \ldots, d_n) \leq 3d_n \cdot C_{gcd}(d_1, \ldots, d_{n-1}) + O((d_1 \cdots d_n)^2 (d_n)^{2n-2})$$

AUFGABE: Zeigen Sie, daß $C_{gcd}(d_1, \ldots, d_n) = O(\max_i\{(d_1 \cdots d_n)^2 (d_i)^{2i-2}\})$ gilt.

**Satz 23** *Der Gesamtaufwand für die gcd-Berechnung von $f, g \in k[x_1, \ldots, x_n]$ über einem Körper $k$ mit Einheitskostenarithmetik mit durch $(d_1, \ldots, d_n)$ beschränkten Gradvektoren über die Subresultanten-PRS ist von der Größenordnung*

$$O((d_1 \cdots d_n)^2 \cdot \max\{d_i^{2i-2},\ i = 1, \ldots, n\}).$$

Insbesondere bestätigt diese Schranke, daß man die Variablen so anordnen sollte, daß $d_1 \geq d_2 \geq \ldots \geq d_n$ gilt.

## 5.5 Modulare gcd-Berechnung

Die bisherigen Verfahren zur gcd-Berechnung zeichnen sich durch außerordentliches Wachstum der intermediären Daten aus. Dieser für viele symbolische Rechnungen typische Effekt läßt sich manchmal dadurch umgehen, daß man die entsprechenden symbolischen Rechnungen statt mit Unbestimmten mit konkreten Parameterwerten ausführt und aus diesem Ergebnis z.B. durch Interpolationstechniken versucht, das entsprechende korrekte symbolische Ergebnis zu gewinnen.

Bei der Berechnung größter gemeinsamer Teiler trat Koeffizientenwachstum besonders hinderlich für Polynome mit ganzzahligen Koeffizienten bzw. für multivariate Polynome auf. Entsprechende Rechnungen wollen wir deshalb im ersten Fall mit modularen Koeffizienten, d.h. modulo einer Primzahl, und im zweiten Fall nach Substitution konkreter Werte für $x_1, \ldots, x_{n-1}$ ausführen.

Betrachten wir zunächst einige Beispiele:

1) Das folgende Beispiel führte bei der Berechnung polynomialer Restsequenzen über $\mathbf{Z}$ zu langen Koeffizienten:

$$f = 3 \cdot x^6 + x^3 + x + 5, \quad g = 5 \cdot x^5 - 3 \cdot x^4 + 1$$

Berechnet man dieselben Sequenzen über modularen Bereichen, so erhält man

mod 2:

$$\begin{aligned} r_0 &= f = x^6 + x^3 + x + 1, \\ r_1 &= g = x^5 + x^4 + 1, \\ r_2 &= x^4 + x, \\ r_3 &= 1 \end{aligned}$$

mod 3:

$$\begin{aligned} r_0 &= f = x^3 + x + 2, \\ r_1 &= g = 2 \cdot x^5 + 1, \\ r_2 &= x^3 + x + 2, \\ r_3 &= x^2 + x + 1, \\ r_4 &= 1 \end{aligned}$$

mod 7:

$$\begin{aligned} r_0 &= f = 3 \cdot x^6 + x^3 + x + 5, \\ r_1 &= g = 5 \cdot x^5 + 4 \cdot x^4 + 1, \\ r_2 &= x^4 + 3 \cdot x^3 + 4 \cdot x + 3, \\ r_3 &= x^3 + 3 \cdot x^2 + 3 \cdot x + 4, \\ r_4 &= x^2 + 6, \\ r_5 &= x, \\ r_6 &= 1 \end{aligned}$$

Stets ist das Ergebnis dasselbe wie über den ganzen Zahlen, wenn auch unterschiedlich lange Restsequenzen zu bilden waren: $f$ und $g$ sind auch im Ring $\mathbf{Z}_p[x]$ zueinander teilerfremd.

2)

$$\begin{aligned} f &:= x^8 + x^6 - 3 \cdot x^4 - 3 \cdot x^3 + 8 \cdot x^2 + 2 \cdot x - 5, \\ g &:= 3 \cdot x^6 + 5 \cdot x^4 - 4 \cdot x^2 - 9 \cdot x + 21 \end{aligned}$$

Wir erhalten folgende größten gemeinsamen Teiler für die verschiedenen modularen Bereiche:

$$\begin{array}{lcl} \text{gcd mod } 2 & : & x^2 + x + 1 \\ \text{gcd mod } 3 & : & 1 \\ \text{gcd mod } 5 & : & 1 \\ \text{gcd mod } 7 & : & x + 3 \\ \text{gcd mod } 11 & : & 1 \\ \text{gcd mod } 13 & : & 1 \end{array}$$

3) Untersuchen wir nun ein Beispiel, das einen nichttrivialen gemeinsamen Teiler $5x + 1$ im Ring $\mathbf{Z}[x]$ hat:

$$\begin{aligned} f &= 5 \cdot x^9 + x^8 + 5 \cdot x^7 + x^6 - 15 \cdot x^5 - 18 \cdot x^4 + 37 \cdot x^3 + 18 \cdot x^2 - 23 \cdot x - 5, \\ g &= 15 \cdot x^7 + 3 \cdot x^6 + 25 \cdot x^5 + 5 \cdot x^4 - 20 \cdot x^3 - 49 \cdot x^2 + 96 \cdot x + 21 \end{aligned}$$

Wir erhalten in diesem Fall über den verschiedenen modularen Bereichen

$$\begin{array}{lcl} \text{gcd mod } 2 & : & x^3 + 1 \\ \text{gcd mod } 3 & : & x + 2 \\ \text{gcd mod } 5 & : & 1 \\ \text{gcd mod } 7 & : & x^2 + 6 \cdot x + 2 \\ \text{gcd mod } 11 & : & x + 9 \\ \text{gcd mod } 13 & : & x + 8 \end{array}$$

d.h. ein meist wenigstens vom Grad her korrektes Ergebnis.

Wir wollen nun zunächst die zu untersuchende Situation allgemein mathematisch beschreiben. Seien $R$ und $R'$ zwei Integritätsbereiche, die durch einen Homomorphismus $\phi : R \to R'$ verbunden sind. Ein solcher Homomorphismus induziert durch seine Wirkung auf den Koeffizienten einen Ringhomomorphismus $\phi : R[x] \to R'[x]$, den wir mit demselben Symbol bezeichnen wollen.

**Satz 24** *Sei $\phi : R[x] \to R'[x]$ wie eben beschrieben und $f(x), g(x) \in R[x]$. Ist $\phi(lc(gcd(f,g))) \neq 0$, so gilt*

$$deg\ gcd(\phi(f), \phi(g)) \geq deg\ gcd(f,g).$$

BEWEIS : Wegen der Operationstreue von $\phi$ gilt $d \,|\, f \quad \Rightarrow \quad \phi(d) \,|\, \phi(f)$. Für $h = gcd(f,g)$ als gemeinsamen Teiler von $f$ und $g$ gilt also $\phi(h) \,|\, gcd(\phi(f), \phi(g))$ und folglich

$$deg\ \phi(h) \leq deg\ gcd(\phi(f), \phi(g)).$$

Aus der Voraussetzung folgt aber $deg\ \phi(h) = deg\ h$ $\quad \square$

In der Voraussetzung kommt der Leitkoeffizient des zu berechnenden $gcd(f,g)$ vor. Berücksichtigt man $lc(gcd(f,g)) \,|\, gcd(lc(f), lc(g))$, so kann man diese Bedingung durch die schärfere, aber

überprüfbare Bedingung $\phi(gcd(lc(f), lc(g))) \neq 0$ oder gar $\phi(lc(f) \cdot lc(g)) \neq 0$ ersetzen. Beispiel 3) zeigt, daß im Fall $\phi(gcd(lc(f), lc(g))) = 0$ der Grad des gcd in der Tat kleiner als erwartet sein kann. Immer gibt es aber nur endlich viele Primzahlen, für die man den Satz deshalb nicht anwenden kann und diese Primzahlen kann man aus den Eingabegrößen sofort direkt ablesen.

**Korollar 5** *Sind unter obigen Voraussetzungen $\phi(f)$ und $\phi(g)$ teilerfremd, so auch $f$ und $g$.*

Das erlaubt es uns in vielen Fällen, schnell zu beweisen, daß zwei Polynome zueinander teilerfremd sind. Betrachten wir etwa die beiden Polynome

$$f(x,t) = \sum_{i=0}^{7} (t+i) \cdot x^i,$$
$$g(x,t) = \sum_{i=0}^{6} (t-i^2) \cdot x^i$$

im bivariaten Polynomring $\mathbf{Z}[t,x] = \mathbf{Z}[t][x]$ und den Ringhomomorphismus $\phi_{t=1} : \mathbf{Z}[t] \to \mathbf{Z}$, der durch die Zuordnung $t \mapsto 1$ induziert wird. Wir erhalten

$$\phi(f) = \sum_{i=0}^{7} (i+1) \cdot x^i = 8 \cdot x^7 + 7 \cdot x^6 + 6 \cdot x^5 + 5 \cdot x^4 + 4 \cdot x^3 + 3 \cdot x^2 + 2 \cdot x + 1,$$
$$\phi(g) = \sum_{i=0}^{6} (1-i^2) \cdot x^i = -35 \cdot x^6 - 24 \cdot x^5 - 15 \cdot x^4 - 8 \cdot x^3 - 3 \cdot x^2 + 1$$

Da die Voraussetzung des Satzes wiederum erfüllt ist, können wir aus der Teilerfremdheit von $\phi(f), \phi(g)$ auf die Teilerfremdheit der beiden Ausgangspolynome schließen.

In unseren Beispielen hatten wir andererseits gesehen, daß in der Tat

$$deg\ gcd(\phi(f), \phi(g)) > deg\ gcd(f,g)$$

eintreten kann. Mehr noch ist in obigen Beispielen keine direkte Verbindung zwischen den Eingangsdaten und dem Eintreten eines solchen Gradsprungs zu erkennen. Morphismen $\phi$ mit dieser Eigenschaft nennen wir *unglücklich*.

### 5.5.1 Der Übergang von ganzzahligen zu modularen Koeffizienten

Zur Berechnung des gcd von Polynomen mit ganzzahligen Koeffizienten betrachten wir den natürlichen Homomorphismus $\phi : \mathbf{Z} \longrightarrow \mathbf{Z}_p$ des Übergangs zu Restklassen modulo einer Primzahl $p$. Damit werden die Rechnungen auf solche mit Polynomen mit modularen Koeffizienten zurückgeführt. Da letzterer Koeffizientenbereich ein Körper (mit Einheitskostenarithmetik) ist, kann man den Euklidschen Algorithmus direkt anwenden. Für $f \in \mathbf{Z}[x]$ wollen wir dabei $\phi_p(f)$ kurz durch $f_p$ bezeichnen.

**Satz 25** *In dieser Situation gibt es nur endlich viele unglückliche Primzahlen.*

BEWEIS : Betrachten wir eine PRS zur gcd-Berechnung über $\mathbf{Z}[x]$. Wenn $p \in \mathbf{Z}$ keinen der Leitkoeffizienten $lc(R_i)$ teilt, dann liefert diese Sequenz eine Restfolge auch über $\mathbf{Z}_p[x]$. Damit ist in diesem Fall $deg\ gcd(\phi(f), \phi(g)) = deg\ gcd(f,g)$. □

Verwendet man Primzahlen $p$ etwa in der Wortgröße des entsprechenden Computers, so ist die Wahrscheinlichkeit, daß man auf eine solche Primzahl trifft, außerordentlich gering.

Man kann nun versuchen, den gcd über $\mathbf{Z}$ aus den Werten des gcd über $\mathbf{Z}_p$ für verschiedene $p$ zu rekonstruieren.

Beispiel:

1) Betrachten wir die beiden Polynome

$$f := x^4 + 25x^3 + 145x^2 - 171x - 360, \quad g := x^5 + 14x^4 + 15x^3 - x^2 - 14x - 15$$

und berechnen für $p \in \{2,3,5,7,11,13,17\}$ nacheinander $h_p := gcd(f,g)$. Für die ersten beiden Primzahlen erhalten wir $h_2 = x^3 + x^2 + x + 1$ und $h_3 = x^3 + x^2 + x$, so daß die Vermutung nahe liegt, daß dies die Reduktionen *mod* 2 bzw. *mod* 3 des ganzzahligen gcd sind, dieser also Grad 3

hat. Da $f$ und $g$ beide monisch sind, hat dieser gcd ebenfalls den Leitkoeffizienten 1, so daß wir mit dem chinesischen Restklassensatz versuchen können, die beiden Polynome zu liften, indem wir die jeweils betragsmäßig kleinsten ganzen Zahlen mit den entsprechenden Resten finden. Dies wäre eines der Polynome $gcd = x^3 + x^2 + x \pm 3$, die beide aber weder Teiler von $f$ noch Teiler von $g$ sind. Also ist die Näherung entweder noch nicht gut genug oder alle bisherigen Primzahlen sind unglücklich. Jedenfalls können wir bereits jetzt schließen, daß $deg\ gcd(f,g) \leq 3$ gilt.

Für $p = 5$ erhalten wir $h_5 = x^4 + 4x$, d.h. es handelt sich auf alle Fälle um eine unglückliche Primzahl, so daß wir mit dem gefundenen Ergebnis nichts anfangen können.

Für $p = 7$ erhalten wir $h_7 = x^2 + 1$. Damit gilt $deg\ gcd(f,g) \leq 2$ und auch $p = 2, 3$ waren unglückliche Primzahlen und die entsprechenden Ergebnisse irrelevant für das Liften. Für die weiteren Primzahlen erhalten wir entsprechend $h_{11} = x^2 + 3x + 4, h_{13} = x^3 + 5x^2 + 6x + 8$ (womit auch $p = 13$ unglücklich ist) und $h_{17} = x^2 + 14x + 15$. Wir vermuten nunmehr $deg\ gcd(f,g) = 2$ und erhalten nach Liften der Ergebnisse für $p = 7, 11, 17$ als Kandidaten $gcd = x^2 + 14x + 15$. Probedivision zeigt, daß das tatsächlich ein gemeinsamer Teiler von $f$ und $g$ ist. Da andererseits der Grad des gcd nicht größer als 2 sein kann, haben wir das korrekte Ergebnis, wenigstens in $\mathbf{Q}[x]$, gefunden.

AUFGABE: Untersuchen Sie dieselbe Problemstellung für die Polynome

$$f := 2x^6 - 13x^5 + 20x^4 + 12x^3 - 20x^2 - 15x - 18;$$
$$g := 2x^6 + x^5 - 14x^4 - 11x^3 + 22x^2 + 28x + 8;$$

Hauptproblem beim Liften von modularen Kandidaten gleichen Grades zu einem ganzzahligen gcd-Kandidaten ist die Tatsache, daß der gcd nur bis auf assoziierte Elemente eindeutig bestimmt ist und die entsprechende Normierung über $\mathbf{Z}$ und über $\mathbf{Z}_p$ auf unterschiedliche Weise erfolgt. Während letztere üblicherweise auf den Leitkoeffizienten 1 normiert werden, kann ein solcher Leitkoeffizient über $\mathbf{Z}$ nicht garantiert werden. Dort ist nur $lc(gcd(f,g)) \mid l := gcd(lc(f), lc(g))$ bekannt. Wir berechnen deshalb nicht den normierten (primitiven) $gcd(f,g)$, sondern einen solchen mit Leitkoeffizient $l$. Dieser ist in $\mathbf{Q}[x]$ zu dem gesuchten assoziiert. Da wir den Inhalt des gcd sowieso getrennt berechnen müssen, entsteht dadurch auch kein wesentlicher Zusatzaufwand. Die vorgenommene Normierung ermöglicht nun das Liften der entsprechenden Polynome.

Insgesamt erhalten wir den folgenden Algorithmus:

```
modularGCD(f,g);
      local
            M = kumulierter Modul
            p = aktuelle Primzahl
            (h,M) = Lift der modularen Ergebnisse
            d = Grad des gcd
            c = gcd(lc(f),lc(g))
      c := gcd(lc(f),lc(g)), M:=1
      d := deg f + deg g + 1 (* garantiert zu groß *)
      repeat
            Wähle Primzahl p > c,  (p, M) = 1
            h_p := c · gcd(f_p, g_p)
            if (deg h_p < d) then (* alle bisherigen p unlucky *)
                  d:=deg h_p
                  M:=p
                  (h, p) := Lift(h_p, p) mit lc(h) = c
            else if deg h_p > d then continue (* p unlucky *)
            else
                  (h, p · M) := Lift((h, M), (h_p, p)) mit lc(h) = c
                  if h|f, h|g then return gcd(cont(f),cont(g)) * pp(h)
                  else M:=p * M
      forever
```

AUFGABE: Führen Sie die entsprechenden Berechnungen für

$$f = 12x^3 + 6x^2 + 14x + 7,\ g = 30x^3 + 15x^2 + 2x + 1$$

aus.

Zur Komplexität dieses Algorithmus: Um die Zahl der modularen Ergebnisse zu bestimmen, die notwendig sind, um nach dem Liften ein korrektes Ergebnis zu erhalten, benötigen wir eine Schranke für die Länge der Koeffizienten im gcd. Man beachte zunächst, daß im gcd längere Koeffizienten als in den Ausgangspolynomen auftreten können.

Beispiel (von Davenport und Trager):
$f := (x+1)^2 \cdot (x-1) = x^3 + x^2 - x - 1$ und $g := (x+1)^2 \cdot (x^2 - x + 1) = x^4 + x^3 + x + 1$ haben den gcd $x^2 + 2\,x + 1$.

Ist $b$ eine Schranke für die Bitlänge der Koeffizienten von $f, g$, so sind in der Subresultanten-PRS die Bitlängen alle auftretenden Koeffizienten der Ordnung $O(d\,b)$, wobei $d$ eine Schranke für den Grad von $f$ und $g$ ist. Folglich gilt diese Schranke erst recht für den gcd.

Da eine Zahl der Bitlänge $n$ höchstens $O(n)$ Primfaktoren haben kann, ist $O(d\,b)$ zugleich eine Schranke für die Anzahl der unglücklichen Primzahlen, die im Laufe des Algorithmus besucht werden können. Höchstens weitere $O(d\,b)$ Schleifendurchläufe sind notwendig, um zum korrekten Ergebnis zu liften. In jedem der insgesamt $O(d\,b)$ Durchläufe sind dann

1. $\phi(f)$ und $\phi(g)$ zu berechnen (Kosten $O(d\,b)$ der modularen Reduktion von $d$ Koeffizienten der Bitlänge $b$),

2. der modulare gcd zu bestimmen (Kosten $O(d^2)$) und

3. die Koeffizienten der $O(db)$ Ergebnisse zu liften (Kosten $O(d^2\,b^2)$).

Die Gesamtkosten dieses Verfahrens sind also maximal von der Ordnung $O(d^3b^3)$.

Dieselben Überlegungen, die wir oben für univariate Polynome ausgearbeitet haben, gelten auch für multivariate Polynome. $\phi_p$ induziert entsprechend einen Homomorphismus

$$\phi_p : \ S := \mathbf{Z}[x_1, \ldots, x_n] \longrightarrow S_p := \mathbf{Z}_p[x_1, \ldots, x_n].$$

Wir fixieren dazu auf dem Monoid der Terme $T = T(x_1, \ldots, x_n)$ eine (noethersche) Termordnung $<$ und definieren Leitkoeffizient $lc(f) \in \mathbf{Z}$ und Leitterm $lt(f) \in T(X)$ für $f \in S$ wie gewohnt. Auch in diesem Fall gilt

$$gcd_{\mathbf{Z}[x_1,\ldots,x_n]}(f,g) = gcd_{\mathbf{Z}}(cont(f), cont(g)) \cdot pp(gcd_{\mathbf{Q}[x_1,\ldots,x_n]}(f,g)),$$

wobei *cont* und *pp* hier multivariat zu verstehen sind.

Sind $f = c_1 X^{A_1} + < kleinere\ Terme >$ und $g = c_2 X^{A_2} + < kleinere\ Terme >$ zwei Polynome aus $S$, so gilt wiederum

**Satz 26** *Wenn $p \not| c := gcd(c_1, c_2)$, so ist $lt(gcd(f_p, g_p)) \geq lt(gcd(f,g))$ bzgl. der (fixierten) Termordnung und die beiden sind nur für endlich viele Primzahlen verschieden.*

AUFGABE: Beweisen Sie den ersten Teil ($lt(gcd(f_p, g_p)) \geq lt(gcd(f,g))$) dieses Satzes.

Die folgende modifizierte Version des oben beschriebenen Algorithmus erlaubt dann die gcd-Berechnung in $S$, wenn man einen Algorithmus für die gcd-Berechnung in $S_p$ zur Verfügung hat.

```
modularMGCD(f,g);
      Input: f,g ∈ S = Z[x_1,...,x_n]
      Output: gcd(f,g)
      local
            M = kumulierter Modul
            p = aktuelle Primzahl
            (h,M) = Lift der modularen Ergebnisse
            d = Leitterm des gcd
            c = integerGCD(lc(f),lc(g))
      c := integerGCD(lc(f),lc(g)), M:=1
      d := lt(f^2) (* garantiert zu groß *)
      repeat
            Wähle Primzahl p > c, (p,M) = 1
            h_p := c · gcd(f_p,g_p) (* normierter S_p-gcd *)
            if (lt(h_p) < d) then (* alle bisherigen p unlucky *)
                  d:=lt(h_p)
                  M:=p
                  (h,p) := Lift(h_p,p) mit lc(h) = c
            else if lt(h_p) > d then continue (* p unlucky *)
            else
                  (h, p·M) := Lift((h,M),(h_p,p)) mit lc(h) = c
                  if h|f, h|g then return integerGCD(cont(f),cont(g)) * pp(h)
                  else M:=p * M
      forever
```

Ist $B$ eine Schranke für die Bitlänge der Koeffizienten von $gcd(f,g)$ über $\mathbf{Z}$, $(d_1, \ldots, d_n)$ eine Schranke für den Gradvektor von $f$ und $g$ und vernachlässigen wir den Aufwand, der durch unglückliche Primzahlen entsteht, so erhalten wir eine ähnliche Abschätzung wie oben. Die Schleife wird höchstens $O(B)$ mal durchlaufen, bis die notwendige Liftgenauigkeit garantiert erreicht wird. $d$ aus der Formel für $\mathbf{Z}[x]$ ist durch die Anzahl der Terme in $f$ und $g$, also $L := max(L(f), L(g)) = O(d_1 \cdot \ldots \cdot d_n)$ zu ersetzen. Bezeichnet schließlich $C(d_1, \ldots, d_n)$ den Aufwand zur Berechnung von $gcd(f_p, g_p)$, erhalten wir insgesamt

**Satz 27** *Der Aufwand zur Berechnung von* `modularMGCD(f,g)` *unter obigen Annahmen ist von der Ordnung*

$$B \cdot O(B \cdot d_1 \cdot \ldots \cdot d_n + C(d_1, \ldots, d_n))$$

AUFGABE: Überprüfen Sie diese Beziehung.

### 5.5.2 Die Reduktion der Variablenzahl

Im vorigen Abschnitt ist die Frage offengeblieben, wie man den gcd multivariater Polynome über einem Körper $k = \mathbf{Z}_p$ berechnen kann. Dies erfolgt durch eine andere Anwendung modularer Techniken, nämlich den Übergang von multivariaten zu univariaten Polynomen. Wir wollen auch diese Aufgabe zunächst an einem Beispiel studieren.

Beispiel 1:
Wir betrachten die Polynome

$$f(x,y) = x^3\,y + 2\,x^2\,y^2 + x^2 + x\,y^3 + 2\,x\,y + y^2,$$
$$g(x,y) = x^3\,y - 2\,x^2\,y^2 + x^2 + x\,y^3 - 2\,x\,y + y^2$$

im Ring $S = k[x,y] = k[y][x]$ über dem Körper $k = \mathbf{Z}_{181}$ und wollen deren gcd $h(x,y)$ daselbst berechnen. Wir betrachten dazu den Morphismus

$$\phi_a : R = k[y] \longrightarrow R' = k \qquad \text{via } y \mapsto a$$

für verschiedene Werte von $a \in k$ und bezeichnen ähnlich wie im vorigen Abschnitt $\phi_a(f)$ kurz durch $f_a$.

Für verschiedene Werte von $a$ erhalten wir die Polynome

| $a$ | $f(x,a)$ | $g(x,a)$ | $h_a(x)$ |
|---|---|---|---|
| 1 | $x^3 + 3\,x^2 + 3\,x + 1$ | $x^3 + 180\,x^2 + 180\,x + 1$ | $x + 1$ |
| 2 | $2\,(x^3 + 95\,x^2 + 6\,x + 2)$ | $2\,(x^3 + 87\,x^2 + 2\,x + 2)$ | $x + 91$ |
| 3 | $3\,(x^3 + 127\,x^2 + 11\,x + 3)$ | $3\,(x^3 + 115\,x^2 + 7\,x + 3)$ | $x + 121$ |
| 4 | $4\,(x^3 + 144\,x^2 + 18\,x + 4)$ | $4\,(x^3 + 128\,x^2 + 14\,x + 4)$ | $x + 136$ |

Nach unserem Satz dürfen wir nur die Ersetzung $y = 0$ nicht vornehmen, da $lc(f) = lc(g) = y$ in $R[x]$ gilt. Alle Morphismen liefern Polynome vom gleichen Grad, so daß wir annehmen können, daß alle gutartig sind und der richtige gcd $h \in R[x]$ ebenfalls den Grad 1 in $x$ hat, so daß wir versuchen können, diesen durch modulare Rekonstruktion zu finden. Genau wie oben haben wir aber dabei das Normierungsproblem zu lösen. Wir suchen deshalb $h$ in der Form $h = y \cdot x + c_0(y)$, weil $gcd(lc(f), lc(g)) = y$. Skalieren wir die Ergebnisse in unserer Tabelle entsprechend, so erhalten wir für $c_0(y)$ als Bedingung, daß $c_0(1) = c_0(2) = c_0(3) = c_0(4) = 1$ gelten muß, so daß wir vermuten können, daß $h = y\,x + 1$ gilt. Probedivision in $R[x]$ durch $f$ und $g$ zeigt, daß dies in der Tat das korrekte Ergebnis ist.

AUFGABE: Berechnen Sie analog $gcd(x^3 + x^2\,y - x\,y^2 - y^3, x^3 - x^2\,y - x\,y^2 + y^3)$ in $k[x,y]$.

Im allgemeinen haben wir beim Liften einen größeren Aufwand zu treiben, wie das folgende Beispiel zeigt.

Beispiel 2:
Wir betrachten wieder in $k[x,y]$ mit $k = \mathbf{Z}_{181}$ die Polynome

$$f(x,y) = x^3\,y + 62\,x^2\,y^2 + 122\,x^2\,y - x^2 + 61\,x\,y^3 + 123\,x\,y^2 + 120\,x\,y + 59\,x + y^3 - y,$$
$$g(x,y) = x^3\,y + x^2\,y^2 - x^2 - x\,y^2 - y^3 + y$$

Die entsprechenden modularen Reduktionen ergeben in diesem Fall

| $a$ | $f(x,a)$ | $g(x,a)$ | $h_a(x)$ |
|---|---|---|---|
| 1 | $x^3+2\,x^2+x$ | $x^3-x$ | $x^2+x$ |
| 2 | $2\,(x^3+155\,x^2+6\,x+3)$ | $2\,(x^3+92\,x^2+179\,x+178)$ | $x+92$ |
| 3 | $3\,(x^3+6\,x^2+32\,x+8)$ | $3\,(x^3+63\,x^2+178\,x+173)$ | $x+63$ |
| 4 | $4\,(x^3+53\,x^2+19\,x+15)$ | $4\,(x^3+49\,x^2+177\,x+166)$ | $x^2+51\,x+98$ |

Die Morphismen $\phi_1$ und $\phi_4$ liefern einen gcd von zu hohem Grad, sind also unglücklich. Wir vermuten deshalb, daß der richtige gcd $h(x,y) \in R[x]$ wiederum vom Grad 1 in $x$ ist und suchen ihn in der Form $h = y\,x + c_0(y)$ mit $c_0(2) = 3$, $c_0(3) = 8$. $c_0(y)$ ist also ein nicht-konstantes Polynom in $y$ und als Absolutglied von $h \in R[x]$ ein Teiler von $abs(f) \sim abs(g) = y^3 - y$, also $deg\ c_0(y) \leq 3$. Ein solches Polynom kann man aber durch Interpolation aus seinen Werten an vier verschiedenen Stellen bestimmen. $\phi_5$ und $\phi_7$ liefern zusätzlich zu unseren bereits bekannten Werten $c_0(5) = 24$, $c_0(7) = 48$, was zusammen $c_0(y) = y^2 - 1$ und $h = y\,x + y^2 - 1$ ergibt. Die Probedivision zeigt, daß das wirklich ein gemeinsamer Teiler der beiden Polynome $f$ und $g$ und damit auch ihr gcd ist.

Aufgabe ([6, S. 95]): Berechnen Sie analog für

$$f = x^3\,y\,(2\,y-1) + x^2\,y^3 + x\,(2\,y^4 - y^3 - 6\,y + 3) + y^2\,(y^3 - 3),$$
$$g = x^3\,(2\,y-1) + x^2\,y\,(-y+1) + x\,(-y^3 + 4\,y - 2) + 2\,y^2$$

$gcd(f,g)$ in $k[x,y]$.

Beschreiben wir nun den allgemeinen Fall. Wir wollen dies gleich für multivariate Polynome ausführen. Sei $\phi_a : R = k[x_0] \to R' = k$ wie oben ein durch $x_0 \mapsto a \in k$ induzierter *Evaluationshomomorphismus* und

$$\phi_a : R[x_1,\ldots,x_n] \longrightarrow k[x_1,\ldots,x_n]$$

der entsprechende Morphismus multivariater Polynomringe, der durch Koeffizientenreduktion entsteht. Ist $f = \sum c_\alpha(x_0)\mathbf{x}^\alpha$ mit $\mathbf{x}^\alpha = x_1^{\alpha_1} \cdot \ldots \cdot x_n^{\alpha_n}$, so erhalten wir $f_a = \sum c_\alpha(a)\mathbf{x}^\alpha$.

Der folgende Algorithmus reduziert nach obigem Schema die gcd-Berechnung im Ring $S = k[x_0,\ldots,x_n]$ auf die in $S' = k[x_1,\ldots,x_n]$. Wir fixieren dazu wie im vorigen Abschnitt eine (noethersche) Termordnung auf $T(x_1,\ldots,x_n)$.

```
evalMGCD(f,g);
      Input: f,g ∈ S
      Output: gcd(f,g)
      local
            a = aktueller Evaluationspunkt für x_0
            M = Liste der modularen Ergebnisse
            d = Leitterm des gcd
            c = gcd(lc(f),lc(g))
      c(x_0) := gcd_{k[x_0]}(lc(f),lc(g))
      d := lt(f^2)  (* garantiert zu groß *)
      repeat
            Wähle a ∈ k, so daß c(a) ≠ 0
            h_a := c(a) · gcd_{S'}(f_a,g_a)  (* normierter S'-gcd *)
            if (lt(h_a) < d) then (* alle bisherigen a unlucky *)
                  d:=lt h_a
                  M:={(h_a,a)}
            else if lt(h_a) > d then continue (* a unlucky *)
            else
                  M:=cons((h_a,a),M)
                  h := Lift(M) mit lc(h) = c(x_0)
```

```
            if h|f, h|g then return gcd_{k[x_0]}(cont(f),cont(g)) * pp_{k[x_0]}(h)
forever
```

AUFGABE: Zeigen Sie, daß `evalMGCD(f,g)` die angegebene Spezifikation erfüllt.

Untersuchen wir nun die Komplexität dieses Algorithmus. Sei dazu der Gradvektor von $f$ und $g$ durch $d = (d_0, \ldots, d_n)$ beschränkt und $C(d_0, \ldots, d_n)$ die zu bestimmende Komplexität, wobei wir wiederum den Aufwand, der eventuell durch schlechte Evaluationspunkte entsteht, d.h. wo $lt(h_a) > lt(h)$ ist, nicht berücksichtigen wollen. Im Gegensatz zum ganzzahligen Liften haben in dieser Situation gemeinsame Teiler von $f, g$ einen $x_0$-Grad, der $d_0$ nicht übersteigt. $2d_0$ ist also eine Gradschranke für die Koeffizienten in $h \in R[x_1, \ldots, x_n]$, da ja $h$ nicht der normierte gcd ist, sondern auf den Leitkoeffizienten $c = c(x_0)$ getrimmt wurde. Wir benötigen also höchstens $O(d_0)$ Schleifendurchläufe, bis wir das korrekte Ergebnis durch Liften gefunden haben.

Die Liste $M$ enthält $O(d_0)$ Polynome aus $k[x_1, \ldots, x_n]$, deren Gradvektor durch $(d_1, \ldots, d_n)$ beschränkt ist. Die Anzahl der Terme jedes dieser Polynome wie auch von $f$ und $g$ ist damit durch $L = L(d_1, \ldots, d_n) = O(d_1 \cdot \ldots \cdot d_n)$ beschränkt. Der Aufwand $C_M(d_0, \ldots, d_n)$ zum Liften dieser Liste ergibt sich aus dem Interpolationsaufwand $C_I(d_0)$, der in $k[x_0]$ notwendig ist, um ein Polynom vom Grad $O(d_0)$ aus seinen Werten an einer entsprechenden Anzahl von Stützstellen zu rekonstruieren, als $C_M = L \cdot C_I$, da diese Interpolation für jeden der Koeffizienten von $h$ auszuführen ist.

AUFGABE: Zeigen Sie, daß die Lagrange-Interpolationsformel $C_I(d) = O(d^3)$ ergibt.

Damit entsteht bis zum return-Befehl der Schleife ein Aufwand von

$$O(d_0) \cdot (C(d_1, \ldots, d_n) + C_M(d_0, \ldots, d_n))$$

und danach zum Adjustieren der Koeffizienten nochmals ein Aufwand von $L \cdot O(d_0^2)$. Verwenden wir obige (nicht optimale) Abschätzung für $C_I$, so erhalten wir für den Gesamtaufwand die folgende Rekursionsformel:

$$C(d_0, \ldots, d_n) = O\left(d_0\, C(d_1, \ldots, d_n) + (d_1 \ldots d_n) \cdot d_0^4\right)$$

AUFGABE: Zeigen Sie, daß unter diesen Voraussetzungen

$$C(d_0, \ldots, d_n) = (d_0^3 + \ldots + d_n^3) \cdot O(d_0 \cdot \ldots \cdot d_n) = O(d_0^3 + \ldots + d_n^3) \cdot L(d_0, \ldots, d_n)$$

gilt.

# Literatur


[1] K.O. Geddes, S.R. Czapor, and G. Labahn. *Algorithms for Computer Algebra.* Kluwer, Boston, 1992.

[2] D.E. Knuth. *The art of computer programming*, volume 2. Addison Wesley, 1981.

[3] E. Kunz. *Algebra.* Vieweg, 1991.

[4] R. Loos. Generalized polynomial remainder sequences. page ??

[5] H. Riesel. *Prime numbers and computer methods for factorization.* Birkhäuser, 1994.

[6] F. Winkler. *Polynomial algorithms in computer algebra.* Texts and Monographs in Symbolic Computation. Springer, Wien, 1996.