Computeralgebra

Vorlesungsskript

Hubert Grassmann

1. März 1999

Literatur:

Davenport, Siret, Tournier; Computer algebra
Mignotte, Mathematics for Computer Algebra
Cohen, A Course in Computational Algebraic Number Theory
Knuth, The art of computer programming, vol 2

Inhaltsverzeichnis

1 Einleitung

Wenn bei arithmetischen Berechnungen Gleitkomma-Arithmetik verwendet wird, so sind Rundungsfehler und damit Ungenauigkeiten der Ergebnisse unausweichlich, wenngleich sie durch den Einsatz potenter Prozessoren und raffinierter Algorithmen oft in einem erträglichen Rahmen gehalten werden können. Nichtsdestoweniger lassen sich nicht nur Beispiele konstruieren, sondern sie kommen auch in der Praxis vor, in denen die Software überlistet oder die Hardware überfordert wird. Die üblichen Algorithmen zur Lösung linearer Gleichungssysteme oder zur Auswertung rationaler Funktionen können zu Ergebnissen führen, die auch nicht näherungsweise korrekt sind.

Ein Computeralgebra-System ist ein Programm, das zunächst zwei Wünsche des Nutzers erfüllen soll:
1. Es rechnet exakt mit rationalen Zahlen, z.B. ist $(1/7)*7 = 1$, es gilt $1/2-1/3 = 1/6$ usw.
2. Es kann symbolisch rechnen, d.h. man kann mit Formeln manipulieren, z.B. ist $(x+1)^2 = x^2 + 2*x + 1$, hierbei ist x eine Variable, die keinen bestimmten Wert besitzt, mit der eben formal gerechnet wird. (Später kann man in Ausdrücke, in denen Variable vorkommen, Zahlen einsetzen.)

Es gibt viele derartige Systeme, die für Workstations und teilweise auch für Personalcomputer geschaffen worden sind. Solche Systeme wie MATHEMATICA, MAPLE, REDUCE oder DERIVE verlangen vom Nutzer, eine neue Programmiersprache zu erlernen, wenn er sich nicht auf den Dialogbetrieb beschränken will.

In den folgenden Kapiteln soll dargestellt werden, wie die grundlegenden arithmetischen Operationen in verschiedenen Zahlbereichen effektiv implementiert werden können.
Quelltexte aller implementierten Algorithmen können vom Autor bezogen werden, sie sind Bestandteil des CA-Systems SINGULAR, welches an der Humboldt-Universität und an der Universität Kaiserslautern unter DFG-Förderung seit 1991 entwickelt wird.

Wunder darf man von keinem Computeralgebra-System erwarten, da der verfügbare Speicher stets Grenzen setzt.

Zahldarstellung im Computer

Wenn im Computer ein Programm abläuft, so befinden sich im Arbeitsspeicher Informationen, die auf unterschiedliche Weise interpretiert werden: einerseits sind dies „Befehle“, die abzuarbeiten sind, andererseits „Daten“, die zu bearbeiten sind. Der kleinste adressierbare Datenbereich ist ein Byte, das sind acht Bit. Ein Byte kann also 256 verschiedene Werte annehmen. Für viele Zwecke reicht diese Informationsmenge nicht aus, meist faßt man zwei Byte zu einem „Maschinenwort“ zusammen, ein Wort kann also $2^{16} = 65536$ Werte annehmen; ein Doppelwort (4 Byte) reicht bis ca. 4 Milliarden.
Die Bearbeitung der Daten geschieht in sogenannten Registern, diese können je ein Wort aufnehmen, und ein Maschinenbefehl wie „ADD AX, BX“ bewirkt die Addition des Inhalts des Registers BX zum Inhalt von AX. Neben arithmetischen Operationen (Addition, Subtraktion, Multiplikation, Division mit Rest) stehen z.B. bitweise Konjunktion, Shift-Operation usw. zur Verfügung.
Als nutzbarer Zahlbereich stehen also nur die Zahlen zwischen 0 und 65355 oder, wenn man das höchste Bit als Vorzeichenbit interpretiert, zwischen -32768 und 32767 zur Verfügung, das sind die Zahlen, die man als Integer bezeichnet.
Rechenoperationen mit Integerzahlen führen gelegebtlich zu Überläufen: 60 000 + 50 000 = 44 464 (oder einem Abbruch); knapp die Hälfte aller möglichen Additionen sind tatsächlich ausführbar, aber nur ein Bruchteil (0.25 Promille) aller möglichen Multiplikationen (im 2-Byte-Bereich).
Da Integerzahlen, außer bei der Textverarbeitung und ähnlichen Problemen, nicht ausreicht, stellen höhere Programmiersprachen zusätzlich Gleitkommazahlen zur Verfügung, die wie bei Taschenrechnern in der Form 1.23456E-20 dargestellt werden, dabei ist die Mantisse meist auf sechs bis acht Dezimalstellen und der Exponent auf Werte zwischen -40 und 40 beschränkt. Das Ergebnis der Rechnung $(1/7) * 7$ wird dann vielleicht 0,9999997 sein, also kann man

$$\frac{1}{1-(1/7)*7}$$

durchaus berechnen, obwohl es gar keinen Sinn hat. Manchmal kann man durch Verwendung längerer Mantissen („doppelte Genauigkeit“) weiterkommen, aber die Verfälschung von Rechenergebnissen und der Programmabbruch wegen Überschreitung der Bereichsgrenzen (flowting point overflow) sind der Preis, den viele für die Unterstützung der Arbeit durch den Computer zahlen zu müssen glauben. Diese zu bedauernden Mitbürger kennen kein Computeralgebrasystem.
Nutzer von CA-Systemen haben mit einer ganz anderen Sorte von Problemen zu kämpfen: dem Memory-Overflow. Hier hilft oft nur eins: Ein neuer, größerer Rechner, ein besseres Betriebssystem und eine neue Version der verwendeten Programmiersprache müssen beschafft werden.

2 Modulares Rechnen

Ein einfacher Ausweg, dem Problem der Rundungsfehler aus dem Weg zu gehen, besteht darin, nicht mit Gleitkommazahlen, sondern nur mit ganzen Zahlen zu rechnen.

Jedoch ist der im Computer verfügbare Bereich ganzer Zahlen einerseits endlich, was zu Bereichsüberschreitungen führen kann, andererseits ist die Division hier nicht uneingeschränkt ausführbar.
In den „Restklassenkörpern“ $\mathbf{Z}_p$ ($p \leq$ WordSize, p Primzahl) sind beide Schranken aufgehoben.

Sei $m \in \mathbf{Z}$. In der Menge $\mathbf{Z}$ ist durch die Relation $\equiv$ mit

$$a \equiv b \Leftrightarrow (a-b) = k \cdot m \text{ für ein } k \in \mathbf{Z}$$

eine Äquivalenzrelation gegeben. Die Menge aller zu $a \in \mathbf{Z}$ äquivalenten Zahlen bezeichen wir mit $[a]_m$.

Beispiel:

$$m = 6,\ \mathbf{Z}_6 = \{[0]_6, [1]_6, \ldots, [5]_6\}$$

mit

$$[0]_6 = \{0, 6, 12, \ldots\},\ [1]_6 = \{1, 7, 13, \ldots, -5, -11, \ldots\}, \ldots, [5]_6 = \{5, 11, \ldots, -1, -7, \ldots\}$$

Einen Repräsentanten von $a \in [x]_m$ erhält man mit `a:= x mod m` (Pascal) oder `a = mod(x,m)` (Fortran). Fakt: Mit den Operationen

$$[a]_m + [b]_m = [a+b]_m$$

$$[a]_m - [b]_m = [a-b]_m$$

$$[a]_m * [b]_m = [a*b]_m$$

ist die Menge $\mathbf{Z}_m$ ein Ring.

Lemma 2.1 *Wenn p eine Primzahl ist, so ist $\mathbf{Z}_p$ ein Körper.*

Beweis: Es ist zu zeigen, daß zu jedem $i,\ 0 < i < p$, ein j existiert mit $ij \equiv 1(\bmod p)$. Wir betrachten die Zahlen $i, 2i, 3i, \ldots, (p-1)i$ und sehen, daß sie in verschiedenen Äquivalenzklassen mod p liegen: Andererseits wäre $in \equiv im$ für gewisse $n < m$, also $i(m-n) \equiv 0$ oder $p \mid i(m-n)$. Da p eine Primzahl ist, folgt $p \mid i$ (das geht nicht wegen $i < p$), oder $p \mid (m-n)$, dies ist wegen $p > m-n > 0$ auch unmöglich.
Also besitzt jedes von Null verschiedenes Element ein multiplikatives Inverses. □

Wie kann man solche Inversen berechnen? Das Probierverfahren ist höchst uneffektiv. Wir können aber den Euklidischen Algorithmus nutzen:
Es seien zwei natürliche Zahlen a_1, a_2 gegeben, wir führen fortgesetzt Divisionen mit Rest durch:

$$a_1 = q_1 a_2 + a_3$$

$$a_2 = q_2 a_3 + a_4$$

$$\ldots$$

$$a_{n-2} = q_{n-2} a_{n-1} + a_n$$

$$a_{n-1} = q_{n-1} a_n$$

Dabei ist stets $a_i < a_{i-1}$ ind die letzte Division geht auf. Dann ist a_n der größte gemeinsame Teiler von a_1 und a_2.
Wenn diese Gleiungenrückwärts nach a_n aufgelöst werden, ergibt sich eine Darstellung $a_n = a_1u_1 + a_2u_2$ für gewisse $u_1, u_2 \in \mathbf{Z}$. Wenn nun p eine Primzahl ist, so ist $\mathrm{ggT}(p, i) = 1$ für $0 < i < p$, also gilt $iu_1 + pu_2 = 1$, also $iu_1 \equiv 1(\mod p)$ und wir haben das Inverse gefunden.
Wenn $\mathrm{ggT}(a, b) = d$ ist, so kann man die Koeffizienten u, v in der Darstellung $d = ua + vb$ schrittweise bei der ggT-Berechnung ausrechnen:
Wir setzen $r_0 = a$ und $r_1 = b$, dann erhalten wir die nächsten Reste wie folgt (x/y bedeutet hier den ganzzahligen Teil des Quotienten):

$$r_2 = r_0 - (r_0/r_1) \cdot r_1$$

$$r_3 = r_1 - (r_1/r_2) \cdot r_2$$

Wir führen noch Zahlen s_i, t_i ein und setzen

$$\begin{pmatrix} r_0 \\ s_0 \\ t_0 \end{pmatrix} = \begin{pmatrix} r_0 \\ 1 \\ 0 \end{pmatrix}, \quad \begin{pmatrix} r_1 \\ s_1 \\ t_1 \end{pmatrix} = \begin{pmatrix} r_1 \\ 0 \\ 1 \end{pmatrix},$$

$$\begin{pmatrix} r_2 \\ s_2 \\ t_2 \end{pmatrix} = \begin{pmatrix} r_0 \\ s_0 \\ t_0 \end{pmatrix} - (r_0/r_1) \cdot \begin{pmatrix} r_1 \\ s_1 \\ t_1 \end{pmatrix}$$

$$\cdots$$

$$\begin{pmatrix} r_{k+1} \\ s_{k+1} \\ t_{k+1} \end{pmatrix} = \begin{pmatrix} r_{k-1} \\ s_{k-1} \\ t_{k-1} \end{pmatrix} - (r_{k-1}/r_k) \cdot \begin{pmatrix} r_k \\ s_k \\ t_k \end{pmatrix}$$

Nun gilt

$$(\,1 \quad -r_0 \quad r_1\,) \begin{pmatrix} r_2 \\ s_2 \\ t_2 \end{pmatrix} = r_0 - r_0 - (r_0/r_1) \cdot r_1 + r_1 \cdot (r_0/r_1) = 0 = r_2 - r_0s_2 - r_1t_2,$$

also

$$r_2 = r_0s_2 + r_1t_1$$

Wir zeigen nun induktiv

$$r_k = s_k \cdot a + t_k \cdot b.$$

Es gilt

$$(\,1 \quad -a \quad -b\,)\,(\,r_{k+1} \quad s_{k+1} \quad t_{k+1}\,) = r_{k-1} - (r_{k-1}/r_k) \cdot r_k - a \cdot s_{k-1} + a \cdot (r_{k-1}/r_k) \cdot s_k - b \cdot t_{k-1} + b \cdot (r_{k-1}/r_k) \cdot t_k$$

Der 3. und 5. Summand ergibt r_{k-1}, der 4. und 6. Summand ergibt $r_k \cdot (r_{k-1}/r_k)$, die Summe ist also gleich 0.

Beispiel: Wir betrachten 64 und 11:

$$\begin{pmatrix} 64 \\ 1 \\ 0 \end{pmatrix} - 5 \begin{pmatrix} 11 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 9 \\ 1 \\ -5 \end{pmatrix}$$

$$\begin{pmatrix} 11 \\ 0 \\ 1 \end{pmatrix} - 1 \begin{pmatrix} 9 \\ 1 \\ -5 \end{pmatrix} = \begin{pmatrix} 2 \\ -1 \\ 6 \end{pmatrix}$$

$$\begin{pmatrix} 9 \\ 1 \\ -5 \end{pmatrix} - 4 \begin{pmatrix} 2 \\ -1 \\ 6 \end{pmatrix} = \begin{pmatrix} 1 \\ 5 \\ -29 \end{pmatrix} \begin{matrix} \leftarrow & \text{ggT} \\ \leftarrow & s \\ \leftarrow & t \end{matrix}$$

In den Beispielen für Algorithmen, die noch folgen, verwenden wir meist Turbo-Pascal oder MODULA-2 als Beschreibungssprache; das geht im Text etwas durcheinander, aber der Kenner der einen (oder anderen) Sprache wird damit hoffentlich keine Verständnisschwierigkeiten haben.

Die folgende Funktion überprüft, ob die Zahl p eine Primzahl ist:

```
PROCEDURE isprime(p): BOOLEAN;
VAR t,s: LONGCARD;
BEGIN
IF p = 2 THEN RETURN TRUE
ELSIF NOT ODD(p) THEN RETURN FALSE
END;
t:=3; s:= p DIV 2; { √p reicht auch }
WHILE (p DIV t)*t < p) AND (t<=s) DO t:= t+2; {Ersatz für p mod t }
END;
RETURN t>s;
END isprime;
```

Das Rechnen mit Restklassen modulo p kann man zum Beispiel nutzen, um Vermutungen zu widerlegen. Wir wollen überprüfen, ob es eine ganze Zahl x gibt, so daß $x^5+x+1=0$ gilt. Wenn dem so ist, so gilt auch $[x^5+x+1]_p = [x]_p^5+[x]_p+[1]_p = [0]_p$ für jede Primzahl p. Nun überprüfen wir dies Gültigkeit dieser Relation für die endlich vielen Werte $x = 0, 1, \ldots, p-1$. Wenn wir eine Primzahl p finden, so daß $[x^5+x+1]_p \neq [0]_p$ für alle x gilt, so kann es keine ganzzahligen Lösungen geben.

Wenn man eine modulare Arithmetik implementieren will, kann man den Modulo-Operator verwenden, der von höheren Programmiersprachen bereitgestellt wird.

Die Berechnung der Summe, Differenz und des Produkts ist recht einfach und durchsichtig. Zur Berechnung des Quotienten bestimmen wir zunächst mit Hilfe des Euklidischen Algorithmus eine Zahl u, die zum Divisor (modulo p) invers ist. Dies ist schon etwas aufwendiger.

```
var p: integer;

{c = a + b} c:= (a + b) mod p

{c = a - b} c:= (a - b) mod p

{c = a * b} c:= (a * b) mod p

procedure inv(a,b: integer; var u: integer);
{u erfüllt au ≡ 1 mod b}
var z,s,x,y,xx,yy,q,r,v,aa,bb: integer;
begin
aa:= a; bb:= b;
x:=1; yy:= x; xx:=0; y:= xx; r:=1;
while r<>0 do
begin
  q:=a div b; r:=a mod b; z:= yy;
  s:= q*yy; yy:= y-s; y:= z;
  z:= xx; s:= q*xx; xx:= x-s;
  x:= z; a:= b; b:= r;
end;
if x < 0 then
begin
```

```
  z:= x; z:= -z; q:= z div bb;
  r:= z mod bb; r:= 1; q:= r+q;
  z:= q*bb; u:= x+z; z:= q*a; v:= y-z;
end
else
begin
  u:= x; v:= y;
end;
end;

{c = a / b} inv(b, p, u); c:= (a * u) mod p;
```

Es lohnt sich immer, darüber nachzudenken, wie man Rechenzeit sparen kann. Hier wird jedesmal, wenn durch eine Zahl b dividiert werden soll, das Inverse berechnet. Besser ist es, vor Beginn aller Rechnungen die Inversen aller Zahlen $1, 2, \ldots, p-1$ zu bestimmen und in ein Feld zu schreiben:

```
var invers: array[1..1000000] of integer;
for i:= 1 to p-1 do inv(i, p, invers[i]);
```

Nun geht die Division bestimmt schneller:

```
procedure mdiv(a, b: integer; var c: integer);
{c:= a / b}
begin c:= (a * invers[b]) mod p
end;
```

Das ist aber nicht notwendigerweise die bestmögliche Lösung. Wenn der Prozessor deutlich mehr Zeit für eine Multiplikation als für eine Addition benötigt oder wenn die Auswertung des Modulo-Operators eine nennenswerte Zeit benötigt, so sollte man bei der Addition und Subtraktion durch Subtraktion oder Addition der Zahl prime zum Rechenergebnis dafür sorgen, daß das Resultat im Bereich $[0 \ldots p-1]$ liegt. Das klappt nicht so einfach bei der Multiplikation.

Wir werden sehen, daß noch eine ganz andere Möglichkeit für die Implementierung der Rechenoperationen in $\mathbf{Z}_p$ gibt. Dazu unternehmen wir einen kurzen Ausflug in die Gruppentheorie.

Es sei G eine endliche (multiplikativ geschriebene) Gruppe und $x \in G$; die kleinste natürliche Zahl k mit $x^k = 1$ heißt die Ordnung von x, sie wird mit $\mathrm{ord}(x)$ bezeichnet. Die Ordnung von x ist gleich der Anzahl der Elemente der von x erzeugten Untergruppe $< x >$ von G, also nach dem Satz von Lagrange ein Teiler der Ordnung $|\,G\,|$ von G.

Lemma 2.2 *Sei G eine kommutative Gruppe, $x, y \in G$, $ord(x) = a$, $ord(y) = b$, dann gilt:*
1. $ord(xy) \mid kgV(a, b)$,
2. wenn $ggT(a, b) = 1$, so ist $ord(xy) = ab$.

Beweis: 1. Sei $m = \mathrm{kgV}(a, b)$, dann gilt $(xy)^m = x^m y^m = 1$, also $\mathrm{ord}(xy) \mid m$.
2. Sei $ua + vb = 1$, dann ist $(xy)^{ua} = x^{ua} y^{1-vb} = y$. Wir wissen, daß $\mathrm{ord}(xy)$ ein Teiler von ab ist und daß u zu b teilerfremd ist. Falls $\mathrm{ord}(xy) = ac$ für einen echten Teiler c von b gälte, also $(xy)^{ac} = 1$, dann wäre $(xy)^{uac} = 1 = y^c \neq 1$, ein Widerspruch. Folglich gilt $b \mid \mathrm{ord}(xy)$ und analog zeigt man $a \mid \mathrm{ord}(ab)$. Da a und b teilerfremd sind, folgt $ab \mid \mathrm{ord}(xy)$. □

Satz 2.3 *Sei G eine endliche kommutative Gruppe und $y \in G$, dann gibt es ein $x \in G$, so daß ord(x) ein Vielfaches von ord(y) ist.*

Beweis: Sei $x \in G$ ein Element maximaler Ordnung. Wir nehmen an, daß $\mathrm{ord}(y)$ kein Teiler von $\mathrm{ord}(x)$ ist. Dann gibt es eine Primzahl p und eine natürliche Zahl h mit $p^h \mid \mathrm{ord}(y)$, aber p^h teilt nicht $\mathrm{ord}(x)$. Es sei $\mathrm{ord}(x) = p^k a, 0 \leq k < h$ und $\mathrm{ggT}(p, a) = 1$ sowie $\mathrm{ord}(y) = p^h b$. Wir setzen $x^* = x^{p^k}, y^* = y^b$, dann ist $\mathrm{ord}(x^*) = a$ und $\mathrm{ord}(y^*) = p^h$, wegen der Teilerfremdkeit von p und a gilt nach dem vorigen Lemma $\mathrm{ord}(x^* y^*) = ap^h > \mathrm{ord}(x)$ im Widerspruch zur Auswahl von x. □

Für eine natürliche Zahl d bezeichnet man mit $\phi(d)$ die Zahl der zu d teilerfremden Zahlen zwischen 1 und $d - 1$ (ϕ heißt Eulersche Funktion).

Satz 2.4 *Sei $|\,G\,| = n$, dann sind folgende Bedingungen äquivalent:*
1. G ist zyklisch,
2. Wenn $d \mid n$ gilt, so besitzt G höchstens d Elemente x mit ord(x) $|d$.
3. Wenn $d \mid n$ gilt, so besitzt G höchstens $\phi(d)$ Elemente x mit ord$(x) = d$.

Beweis: ($1 \Rightarrow 2$) Sei $G =< x >$ und $d \mid n$, wir setzen $m = n/d$. Dann sind die d-ten Potenzen der folgenden d Elemente gleich 1:

$$1, x^m, x^{2m}, \ldots, x^{(d-1)m},$$

die d-ten Potenzen anderer Elemente sind nicht gleich 1, also haben nur diese d Elemente eine Ordnung, die die Zahl d teilt.
($2 \Rightarrow 3$) Es sei $d \mid n$. Wenn G kein Element der Ordnung d enthält, so ist (3) erfüllt. Sonst sei $\mathrm{ord}(x) = d$, dann sind die Elemente

$$1, x, x^2, \ldots, x^{d-1}$$

alle voneinander verschieden, ihre Ordnung ist ein Teiler von d. Nach Voraussetzung sind dies also die einzigen Elemente, deren Ordnung die Zahl d teilt. Unter diesen haben gerade die x^k genau die Ordnung d, wo $\mathrm{ggT}(k, d) = 1$ ist, deren Anzahl ist $\phi(d)$.
($3 \Rightarrow 1$) Nach dem vorigen Satz gibt es ein Element $x \in G$, dessen Ordnung das kleinste gemeinschaftliche Vielfache der Ordnungen aller Elemente von G ist. Wir nehmen an, daß $\mathrm{ord}(x) < n$ ist. Es sei $H =< x >$, da $H \neq G$ ist, gibt es ein $y \notin H$, es sei $\mathrm{ord}(y) = d$. Nun gilt $d \mid \mathrm{ord}(x)$, also enthält H genau $\phi(d)$ Elemente der Ordnung d. Damit enthielte G aber $\phi(d) + 1$ Elemente der Ordnung d, was der Voraussetzung widerspricht. □

Nun können wir den folgenden schönen Satz beweisen:

Satz 2.5 *Die multiplikative Gruppe $G = \mathbf{Z}_p - \{0\}$ ist zyklisch, d.h. es gibt eine Zahl g, so daß für alle $a = 1, \ldots, p-1$ eine Zahl i existiert, so daß $a \equiv g^i (mod\ p)$.*

Beweis: Es ist $|G| = p-1$, sei $d \mid (p-1)$. Da $\mathbf{Z}_p$ ein Körper ist, hat das Polynom $X^d - 1$ in $\mathbf{Z}_p$ höchstend d Nullstellen, d.h. in G gibt es höchstens d Elemente, deren Ordnung ein Teiler von d ist, also ist G nach dem vorigen Satz eine zyklische Gruppe. □

Die folgende Prozedur sucht nach einer derartigen Zahl g, wobei $p =$ prime eine ungerade Primzahl ist. Für $g = 2, 3, \ldots$ wird g^i berechnet $(i = 1, 2, \ldots, (p-1)/2)$, wenn $g^i \equiv 1$ wird, so ist g kein erzeugendes Element. Wenn aber $g^{(p-1)/2} \equiv -1$ ist, so ist $i = p-1$ die kleinste Zahl mit $g^i \equiv 1$ ist, also ist g ein erzeugendes Element.

```
PROCEDURE generator(VAR g: integer);
VAR y, w, e: integer;
BEGIN
g:= 1;
REPEAT
  INC(g);
  e:= (p-1) DIV 2;
  y:= 1; w:= g;
  LOOP
    IF ODD(e) THEN
      y:= y*w MOD p;
      IF y=1 THEN y:= 0; EXIT;
      END;
    END;
    IF e > 1 THEN
      w:= w*w MOD p;
    END;
    e:=e DIV 2;
    IF e=0 THEN
      EXIT
    END;
  END;
UNTIL y = p - 1;
END generator;
```

Wenn wir ein erzeugendes Element g gefunden haben, schreiben wir für $i = 1, \ldots, p-1$ die Restklassenvertreter von g^i in ein Feld, hier wird $pot[i] := g^i \bmod p$, gleichzeitig merken wir uns die Stelle, wo g^i steht: $place[g^i] := i$. Wir kodieren nun die Zahl $a = g^i$ durch die Zahl i, falls $a \neq 0$ ist, die Zahl 0 kodieren wir durch NULL = 1000000. Dann muß man die Ein- und Ausgabe dieser Kodierung anpassen:

```
PROCEDURE lies(VAR a: integer);
VAR j: INTEGER;
i: integer;
BEGIN read(j);
i:= j MOD p;
IF i=0 THEN a:= NULL ELSE a:= place[i];
END;
```

```
PROCEDURE schreib(a: integer);
BEGIN IF a=NULL THEN write('0') ELSE write(pot[a]);
END;
```

Die Multiplikation/Division läßt sich nun auf die Addition/Subtraktion zurückführen, es gilt ja $g^i * g^j = g^{i+j}$ und $g^i/g^j = g^{i-j}$. Wir setzen $pminus1 = p - 1$.

```
PROCEDURE npMult(a, b:integer;
VAR c: integer);
BEGIN IF (a=NULL) OR (b=NULL) THEN c:= NULL
RETURN END;
c:= a + b;
IF c>pminus1 THEN c:= c - pminus1) END;
END npMult;
```

```
PROCEDURE npDiv(a, b: integer;
VAR c: integer);
BEGIN IF a=NULL THEN c:= NULL; RETURN END;
IF a>b THEN c:= a - b;
ELSE c:= pminus1 - b + a;
END;
END npDiv;
```

Dafür wird die Addition etwas umständlicher:

```
PROCEDURE npAdd(a, b:integer;
VAR c: integer);
VAR h: integer;
BEGIN IF b=NULL THEN c:= a;
ELSIF a=NULL THEN c:= b;
ELSE BEGIN
h:= pot[a] + pot[b];
IF h > p THEN h:= h - p;
ELSIF h = p THEN c:= NULL;
RETURN;
END;
c:= place[h];
END npAdd;
```

Da $g^{(p-1)/2} \equiv -1(modp)$ gilt, erhalten wir durch die folgende Funktion die Zahl $-a$ $(pm1d2 = (p-1)/2)$:

```
PROCEDURE npNeg(a: integer): integer;
BEGIN IF a=NULL THEN RETURN NULL ELSIF a > pm1d2 THEN RETURN a-pm1d2 ELSE
RETURN a+pm1d2 END END npNeg;
```

Damit ist die Subtraktion auf die Addition zurückgeführt.

Tests haben ergeben, daß die neue Multiplikation auf MS-DOS-Rechnern etwa dreimal so schnell wie die alte ist, die Addition ist geringfügig langsamer. Auf einem Macintosh-Rechner konnte aber kein Unterschied in der Rechenzeit entdeckt werden. Und auf SPARC-Workstations ist die erste Variante deutlich schneller.

Noch besser wird das Zeitverhalten, wenn man genau umgekehrt vorgeht: Die Restklasse $a = g^i$ wird nicht durch ihren „Logarithmus“ i kodiert, sondern bleibt, wie sie ist. Dann ist die Addition wieder naiv durchzuführen. Wir merken uns aber die Zuordnung $a \leftrightarrow i$ und schreiben $i = L(a)$ und $a = E(i)$. Wir legen eine Logarithmustafel an, die wir für die Multiplikation und Division verwenden:

```
a * b = E(L(a) + L(b));    a / b = E(L(a) - L(b)),
```

wobei `L(a) ± L(b)` ggf. um $p-1$ zu verringern / zu vergrößern ist. Die Inversentabelle ist nun nicht mehr nötig.

Beachten Sie bitte, daß nun beim modularen Multiplizieren die MOD-Operation gar nicht mehr verwendet wird (daher die Beschleunigung).

Als Variante für die Addition bietet sich folgendes an:

$$g^i + g^j = g^j \cdot (g^{i-j} + 1) \text{ für } i > j.$$

Wir merken uns in einer Tabelle die Zahl k mit $g^k = g^i + 1$, dies ist der sog. Jacobilogarithmus $J(i) = k$.

3 Ganze Zahlen

Im nächsten Abschnitt werden wir sehen, wie (innerhalb der durch den Rechner gesetzten Speichergrenzen) eine exakte Arithmetik zu implementieren ist. Als Hilfsmittel dazu ist eine Arithmetik für lange ganze Zahlen nötig, mit der wir uns hier befassen.
Wir beginnen nicht mit natürlichen Zahlen, sondern wollen uns auch das Vorzeichen einer Zahl merken.
Wir stellen eine natürliche Zahl n in einem Stellensystem dar, wie dies seit der Einführung der arabischen Zahlen üblich ist. Dazu wählen wir eine Basiszahl $B > 1$ und können die Zahl n in eindeutiger Weise als

$$\sum_{n=0}^{L} n_i B^i,\ 0 \le n_i < B,$$

darstellen, die Werte $n_0, n_1, \ldots, n_L$ nennen wir die Ziffern der Zahl n, die Zahl L nennen wir die Länge von n.
Wenn wir $B \le 256$ wählen, passen die n_i jeweils in ein Byte, bei $B \le 65536 = 2^{16}$ passen sie in ein Maschinenwort eines PC.
Um alle Informationen über unsere langen Zahlen anzuspeichern, gibt es mehrere Möglichkeiten:
1. Wir legen eine maximale Zahllänge fest:

```
NumSize = 64
```

und legen die Ziffern einer langen Zahl in einem Feld ab:

```
type Digit = byte;
IntArr = array [1..NumSize] of Digit;
IntNumber = record
l: integer;     { Länge }
m: IntArr;     { Ziffernfolge }
n: boolean;     { negativ ? }
end;
```

dies ist die einfachste Methode. Sie hat zwei Nachteile: Bei kurzen Zahlen (l < NumSize) wird Speicherplatz verschenkt, andererseits addieren sich bei Multiplikationen oft die Längen der Faktoren, die vorgegebene Maximalgrenze ist schnell erreicht.
2. Wir legen nur die notwendigen L Ziffern in einer verketteten Liste ab:

```
type IntList = POINTER TO NatNumRec;
IntNumRec = record
  m: Digit;     { Ziffer }
  n: boolean;     { negativ ? }
  f: IntList; { Adresse der nächsten Ziffer }
end;
```

So können wir „wirklich“ beliebig lange Zahlen anspeichern, üblicherweise weist man der letzten gültigen Ziffer im Feld `f` eine leicht zu erkennende Adresse zu, etwa `NIL` (= 0). Wenn man eine neue Zahlvariable belegen will, muß man sich zuerst Speicherplatz für diese Variable holen, dafür gibt es Allockierungsfunktionen (z.B. NEW). Für eine Zahl der Länge L benötigen wir `L * (SizeOf(byte) + SizeOf(ADDRESS))`, auf einem PC also $5L$ Byte. Diese Möglichkeit ist (wegen der gefallenen Schranke) besser als die erste, aber sehr speicherintensiv.
Warum haben wir eigentlich nur ein Byte für eine Ziffer verwendet, man könnte auch ein Wort (= CARDINAL) oder einen noch größeren Speicherbereich (LongWord, LONGCARD) verwenden. Nun, die Rechenoperationen mit langen Zahlen werden später auf Rechenoperationen mit Ziffern zurückgeführt werden, und auf der Maschinenebene gibt es elementare Befehle, die diese Operationen mit Bytes ausführen. Das Produkt von zwei Bytes paßt in ein Doppelbyte, dies läßt sich leicht in zwei Bytes zerlegen. Wir können also als Zifferntyp den größten Typ von Maschinenzahlen wählen, für den es einen doppeltsogroßen Maschinenzahltyp gibt.
Wenn wir also neu

```
type Digit = CARDINAL;
```

festlegen, so sinkt der Adressierungsaufwand.
3. Als Kompromiß zwischen beiden Varianten bietet es sich an, die Ziffern in einem dynamischen Feld zu speichern, dies ist zwar echt begrenzt, aber die Grenze ist doch recht hoch:
Wir beginnen von vorn:

```
const NumSize = 32000;
type Digit = CARDINAL;
IntArr = array[1..NumSize] of Digit;
```

```
IntRec = record
  l: integer;     { Länge }
  n: boolean;     { negativ ? }
  m: IntArr;     { Ziffernfolge }
end;
IntNumber = POINTER TO IntRec;
```

Beachten Sie bitte, daß im Gegensatz zur allerersten Methode das Record-Feld `m`, dessen Größe variabel ist, als letztes in der Definition des Datentyps aufgeführt ist. Der Speicherplatz für solch ein Record wird in dieser Reihenfolge angelegt, wenn `n` hinter `m` käme, würde diese Eintragung an einer Stelle erfolgen, für die gar kein Speicherplatz angefordert wird.
Bevor wir eine Variable N vom Typ IntNumber belegen können, müssen wir hierfür Speicherplatz holen. Hier ist die Verwendung von NEW unangemessen, denn wir wollen ja für eine Zahl der Länge L nur L * SizeOf(Digit)+SizeOf(integer)+SizeOf(boolean) Bytes belegen. Also verwenden wir

```
ALLOCATE(N, L * SizeOf(Digit)+SizeOf(integer)+SizeOf(boolean));
```

Die maximale auf diese Weise darstellbare Zahl hat etwa 20000 Dezimalstellen (etwa 10 A4-Druckseiten), das dürfte (für's erste) reichen.
Wenn wir aus zwei Zahlen X, Y eine neue Zahl Z berechnen wollen, müssen wir den Platzbedarf von Z kennen. Hierfür hat man folgende Abschätzungen:
Falls $Z = X + Y$ oder $Z = X - Y$, so ist die Länge L_Z von Z kleiner oder gleich $\mathrm{Max}(L_X, L_Y) + 1$. Falls $Z = X \cdot Y$ und $X, Y \neq 0$ sind, so ist $L_Z \leq L_X + L_Y$.
Bevor wir mit den arithmetischen Operationen beginnen, wollen wir überlegen, wie die Basis B unseres Zahlsystems zweckmäßigerweise gewählt werden sollte.
Wenn wir für B eine Zehnerpotenz wählen, so sind die Ein- und Ausgabeoperationen sehr leicht zu implementieren. Leider sind diese Operationen diejenigen, die am allerseltensten aufgerufen werden, meist nur am Beginn und am Ende eines Programmlaufs. Wenn wir $B = 32768$ setzen, so bleibt das höchste Bit jeder Ziffer frei, wie im Fall von $B = 10^k$ wird so Speicher verschenkt. Wir haben oben die Festlegung

```
Digit = byte
```

durch

```
Digit = CARDINAL
```

ersetzt. Der Grund für diese Änderung (Adressierungsaufwand) ist inzwischen entfallen, wir verwenden ja dynamische Felder. Jenachdem, für welche Ziffer-Größe man sich entscheidet, wäre B = 256 oder B = 65536 zu wählen. Die internen Darstellungen einer Zahl auf diese oder die andere Weise unterscheiden sich um kein Bit, nur die Länge L ist im ersten Fall doppelt so groß wie im zweiten. Wenn im folgenden also Laufanweisungen der Länge L (oder gar der Länge L^2) auftreten, ist eine kürzere Darstellungslänge eventuell von Vorteil. Da im Fall B = 65536 ständig CARDINALs auf LONGCARDs „gecastet“ und LONGCARDs in CARDINALs zerlegt werden, läßt sich die richtige Wahl nicht rechner- und compilerunabhängig treffen. Man erlebt die größten Überraschungen.

Wir beginnen nun mit der Beschreibung von Implementationen der Rechenoperationen, wir verwenden MODULA-2 als Beschreibungssprache.
Wir definieren zunächst

```
CONST basis = LONGCARD(65536);
```

Die einfachste Operation ist die Addition von Zahlen gleichen Vorzeichens. Die enstprechenden Stellen der Summanden werden als LONGCARDs addiert, der Rest (modulo basis) ist die entsprechende Stelle der Summe, der Quotient (modulo basis) ist der Übertrag, er ist gleich 0 oder 1.

```
PROCEDURE al(a, b: IntNumber;  VAR c: IntNumber);
(* Add longs, signs are equal *)
VAR i, max , min :CARDINAL;
    w: LONGCARD;
    cc: IntNumber;
BEGIN
  IF a^.l > b^.l THEN
    min := b^.l; max := a^.l
  ELSE
    min := a^.l; max := b^.l;
  END;
  newl(c, max + 1);     { allockiert Platz für c }
  clear(c);     { belegt c mit Nullen }
  w:= 0;
  WITH a^ DO
    FOR i:= 1 TO min DO
      w:= w + m[i] + b^.m[i];
      c^.m[i]:= w MOD basis;
      w:= w DIV basis;
    END;     { a oder b ist abgearbeitet }
    IF max  = l THEN    { von a noch ein Rest }
      FOR i:= min +1 TO max DO
        w:= w + m[i];
        c^.m[i]:= w MOD basis;
        w:= w DIV basis;
      END
    ELSE    { oder von b noch ein Rest }
      FOR i:= min +1 TO max DO
        w:= w + b^.m[i];
        c^.m[i]:= w MOD basis;
        w:= w DIV basis;
      END;
    END;
  END;
  IF w>0 THEN    { an letzter Stelle Übertrag ? }
    c^.m[max +1]:= w;
```

```
  ELSE
    newl(cc, c^.l-1);     { sonst c verkürzen }
    cc^.n:= c^.n;
    FOR i:= 1 TO cc^.l DO
      cc^.m[i]:= c^.m[i];
    END;
    lrWegl(c);     { altes c wegwerfen }
    c:= cc;
  END;
  c^.n:= a^.n;
END al;
```

Vielleicht ist es übertrieben, wegen eines Bits (dem Übertrag) gleich von 2-Byte-CARDINALs zu 4-Byte-LONGCARDs überzugehen. Wenn bei einer CARDINAL-Addition ein Überlauf auftritt, wird das „Carry-Flag“ gesetzt und das Ergebnis ist (modulo basis) korrekt. Bei 80i86-Prozessoren ist das Carry-Flag das niedrigste Bit des Flaggenregisters, letzteres kann man sich unter TOPSPEED-Modula mit Hilfe der Funktion SYSTEM.GetFlags als CARDINAL besorgen.
Man kann also gewisse Zeilen der obigen Prozedur durch folgende ersetzen:

```
w:= 0;     { w und wh sind hier CARDINALs ! }
WITH a^ DO FOR i:= 1 TO min DO wh:= w + (m[i]);
w:= ODD(GetFlags());
(* carry *) c^ .m[i]:= wh + (b^.m[i]);
w:= w +ODD(GetFlags());
END;
...
```

Beachten Sie, daß das Carry-Flag nach jeder Addition abgefragt werden muß, daß es aber innerhalb der FOR-Schleife oben höchstens *einmal* gesetzt sein kann.
Leider hatte diese „Optimierung“ auf einem PC keinen Erfolg, die Rechenzeit stieg um 10%. Als Ursache ist zu vermuten, daß für den konkreten Wert basis = 65536 die Operationen `w DIV basis` und `w MOD basis` sehr effektiv vorgenommen werden (es sind ja auch nur Verschiebungen). Vielleicht haben Sie bei einem schlechteren Compiler mit dieser Methode mehr Erfolg.
Wir kommen nun zur Subtraktion, und zwar dem einfachen Fall, daß die Operanden das gleiche Vorzeichen haben und der Subtrahend den größeren Betrag hat. Die einzelnen Stellen werden als LONGINTs subtrahiert und ein Übertrag von der nächsthöheren Stelle des Subtrahenten subtrahiert.

```
PROCEDURE sl(a, b: IntNumber;  VAR c: IntNumber);
(* Subtract longs, signs are equal, a > b *)
VAR i, j, jj: CARDINAL;
    cc: IntNumber;
    lh, li, ln, lb: LONGINT;
BEGIN
  newl(c, a^.l);
```

```
  clear(c);     { c:= 0 }
  WITH a^ DO
    ln := m[1];
    FOR i:= 1 TO b^.l DO    { zunächst a und b }
      li:= ln ;
      ln := m[i+1];
      lb:= b^.m[i];
      lh:= li - lb;
      IF lh < 0 THEN    { Übertrag }
        ln:= ln - 1;
        lh:= lh + basis;
      END;
      jj:= lh;
      c^.m[i]:= jj;
    END;
    FOR i:= b^.l+1 TO l DO    { nun Rest von a }
      li:= ln ;
      ln := m[i+1];
      IF li < 0 THEN
        ln:= ln - 1;
        li:= li + basis;
      END;
      jj:= li;
      c^.m[i]:= jj;
    END;
    j:= l;
    WHILE (j>0) AND (c^.m[j]=0) DO    { wieweit ist c belegt ? }
      j:= j - 1;
    END;
  END;
  IF j=0 THEN
    lrWegl(c);     { c = 0 }
    RETURN;
  END;
  IF j < a^.l THEN
    newl(cc, j);     { c verkürzen }
    clear(cc);
    FOR i:= 1 TO j DO
      cc^.m[i]:= c^.m[i];
    END;
    lrWegl(c);
    c:= cc;
  END;
  c^.n:= a^.n;  { Ergebnis hat Vorzeichen der größeren Zahl }
END sl;
```

Nun kommt der allgemeine Fall der Addition und Subtraktion ganzer Zahlen.

```
PROCEDURE add(a, b:IntNumber;  VAR c: IntNumber);
(* c:= a + b *)
VAR s: CARDINAL;
BEGIN
  IF a^.n = b^.n THEN    { gleiches Vorzeichen }
    al(a,b,c)
  ELSE
    s:= compabs(a, b);     { Vergleiche Absolutbeträge }
    CASE s OF
      2: sl(b, a, c);     { b > a }
      |1: sl(a, b, c);     { a > b }
      |0: c:= NIL;     { a = b }
    END;
  END;
END add;
```

```
PROCEDURE sub(a, b:IntNumber;  VAR c: IntNumber);
(* c:= a - b *)
VAR s: CARDINAL;
BEGIN
  IF a=NIL THEN    { wenn a = 0 }
    lrCopi(b, c);     { c:= b }
    IF c<>NIL THEN
      c^.n:= NOT c^.n    { c:= -c }
    END;
    RETURN;
  END;
  IF b=NIL THEN    { wenn b = 0 }
    lrCopi(a, c);     { c:= a }
    RETURN;
  END;
  IF a^.n <> b^.n THEN    { verschiedene Vorzeichen ? }
    al(a,b,c)    { dann addieren, Vorzeichen von a }
  ELSE
    s:= compabs(a, b);     { sonst Vergleich }
    CASE s OF
      1: sl(a, b, c);     { a > b }
      |2: sl(b, a, c);  c^.n:= NOT b^.n;     { a < b, Vorzeichen ! }
      |0: c:= NIL;     { a = b }
    END;
  END;
END sub;
```

Wir kommen nun zur Multiplikation.
Zunächst führen wir eine Hilfsprozedur an, die ein effektiver Ersatz für folgende Prozeduraufrufe ist:

```
al(a, b, c);
{ hier steckt mindestens eine Allockierung dahinter }
wegl(a); a:= c;
```

Die Variable a wird nur einmal angelegt.

```
PROCEDURE a2((*VAR*) a, b: IntNumber);
(* a:= a + b, signs are equal, a already exists ! *)
VAR i, max, min : INTEGER;
    w: LONGCARD;
BEGIN
  min:= b^.l; max:= a^.l; w:= 0;
  WITH a^ DO
    FOR i:= 1 TO min  DO
      w:= w + m[i] + b^.m[i];
      m[i]:= w MOD basis;
      w:= w DIV basis;
    END;
    FOR i:= min +1 TO max  DO
      w:= w + m[i];
      m[i]:= w MOD basis;
      w:= w DIV basis;
    END;
    IF w>0 THEN
      m[max +1]:= w;
    END;
  END;
END a2;
```

Die Multiplikation wird genauso durchgeführt, wie Sie es in der Schule gelernt haben: Ein Faktor wird mit einer einstelligen Zahl multipliziert und das Ergebnis wird verschoben, dies leistet die Unterprozedur `mh`. Diese Werte werden aufaddiert, dies leistet `a2`. Zu beachten ist, daß die Länge der abzuarbeitenden FOR-Schleife für die Rechenzeit eine wesentliche Bedeutung hat; es wird die kürzere Schleife gewählt.
(Die von nun an auftretenden Vorsilbe „lr“ in den Prozedurnamen wird verwendet, um daran zu erinnern, daß es sich um Operationen mit langen rationalen Zahlen handelt.

```
PROCEDURE lrMl(a, b:IntNumber;  VAR c: IntNumber);
(* mult long, c:= a * b *)
VAR hh: IntNumber;
    la, lb, i, j, hl: CARDINAL; h: LONGCARD;
  PROCEDURE mh(a: IntNumber;  b: LONGCARD;  s: CARDINAL);
  VAR i: CARDINAL;
```

```
  BEGIN
    h:= 0;
    WITH a^ DO
      FOR i:= 1 TO l DO
        h:= h + m[i] * b;
        hh^.m[i+s]:= h MOD basis;
        h:= h DIV basis;
      END;
      hh^.m[l+s+1]:= h;
    END;
    IF h=0 THEN
      hh^.l:= a^.l+s
    ELSE
      hh^.l:= a^.l+s+1;
    END;
  END mh;
BEGIN (* lrMl*)
  la:= a^.l;  lb:= b^.l; hl:= la+lb;
  newl(c, hl); clear(c);
  newl(hh, hl); clear(hh);
  IF la < lb THEN     { kürzere Schleife außen ! }
    FOR i:= 1 TO la-1 DO
      IF a^.m[i]<>0 THEN
        mh(b, a^.m[i], i-1);
        a2(c, hh);
        FOR j:= i TO hh^.l DO
          hh^.m[j]:= 0;
        END;
      END;
    END;
    mh(b, a^.m[la], la-1);
    a2(c, hh);
  ELSE
    FOR i:= 1 TO lb-1 DO
      IF b^.m[i]<>0 THEN
        mh(a, b^.m[i], i-1);
        a2(c, hh);
        FOR j:= i TO hh^.l DO
          hh^.m[j]:= 0;
        END;
      END;
    END;
    mh(a, b^.m[lb], lb-1);
    a2(c, hh);
  END;
```

```
  IF c^.m[hl]=0 THEN
    shrink(c);
  END;
  c^.n:= a^.n <> b^.n;
  hh^.l:= hl;
  lrWegl(hh);
END lrMl;
```

Eine Beschleunigung der Multiplikation kann man von dem folgenden rekursiven Algorithmus von Karatsuba (1962) erwarten:

na = Länge von a; nb = Länge von b
$nah = (na + 1)$ div 2; $nbh = (nb + 1)$ div 2
$nh = \max(nah, nbh)$
Dann ist $a = au * B^{nh} + al$ (al = letzte nh Stellen von a)
analog $b = bu * B^{nh} + bl$
Dann ist

$$a * b = (au * B^{nh} + al) * (bu * B^{nh} + bl) = au * bu * B^{2*nh} + (al * bu + au * bl) * B^{nh} + al * bl$$

Nun ist

$$(al * bu + au * bl) = (au + al) * (bu + bl) - au * bu - al * bl$$

Also ergibt sich folgender Algorithmus:

```
k1:=au+al;
k2:=bu+bl;
k3:=k1*k2;
k4:=au*bu;
k5:=al*bl;
result:=k4*B^(2*nh) + (k3-k4-k5)*B^nh + k5
```

Der Karatsuba-Algorithmus erlaubt es, Zahlen der Länge n in etwa $n^{1.5}$ Zeiteinheiten (anstatt n^2) durchzufuhren. Dies konnte unter Laborbedingungen bestätigt werden: In einem Testprogramm wurden je zwei gleichlange Zahlen multipliziert, bei 40-Byte-Zahlen betrug die Einsparung etwa 10 % , bei 100-Byte Zahlen 20 % . Wurden jedoch eine n-Byte-Zahl mit einer $n/2$-Byte-Zahl multipliziert, so wurden die Rechenzeiten mit wachsendem n beim Karatsuba-Algorithmus immer schlechter.

Nun kommen wir zur Division, das war schon in der Schule etwas schwieriger. Gegeben sind zwei Zahlen

$$a = \sum a_i B^i \text{ und } b = \sum b_i B^i,$$

gesucht sind Zahlen q und r mit

$$a = bq + r,\ 0 \le r < b.$$

Wenn $a < b$ sein sollte, setzen wir $q = 0$ und $r = a$ und sind fertig. Andernfalls muß man die einzelnen Stellen von q nun erraten. Wir bezeichnen wieder mit $L(a)$ die Länge der Zahl a. Als Länge von q ist etwa $L(a) - L(b)$ zu erwarten, wir setzen

$$Q = a_{L(a)} \text{ div } b_{L(B)}$$
$$q = Q \cdot B^{L(a)-L(b)}$$

und prüfen, ob

$$0 \leq r = a - bq < b$$

ist. Wenn nicht, so haben wir zu klein geraten, wir erhöhen Q, oder r ist negativ, dann haben wir zu groß geraten, wir erniedrigen Q und probieren es nochmal. Wenn r endlich im richtigen Bereich liegt, haben wir die erste Stelle von q gefunden, wir ersetzen nun a durch r und berechnen in analoger Weise die nächste Stelle von q.
Genauer: Sei $a = a_0B^{l+1} + a_1B^l + \ldots,\ b = b_1B^l + b_2B^{l-1} + \ldots$, wir suchen das kleinste q mit $a - qb < b$. Wir setzen

$$q^* = \min(\lfloor \frac{a_0B + a_1}{b_1} \rfloor, B - 1).$$

Behauptung: $q^* \geq q$.

Beweis: Der Fall $q^* = B - 1$ ist klar: Wegen $q < B$ ist $q \leq B - 1$.
Sonst haben wir

$$q^* \leq \frac{a_0B + a_1}{b_1} < q^* + 1,$$

also

$$a_0B + a_1 < q^*b_1 + b_1,$$

d.h.

$$a_0B + a_1 \leq q^*b_1 + b_1 - 1$$

und schließlich

$$q^*b_1 \geq a_0B + a_1 - b_1 + 1.$$

Nun ist $q^*b \geq q^*b_1B^l$, also

$$a - q^*b \leq a - q^*b_1B^l \leq a_0B^{l+1} + \ldots + a_{l+1} - a_0B^{l+1} - a_1B^l + b_1B^l - B^l$$
$$= a_2B^{l-1} + \ldots a_{l+1} - B^l + b_1B^l < b_1B^l \leq b.$$

Also ist $q^* \geq q$. □

Das beschriebene „Falschraten“ kann sehr häufig vorkommen und die Rechenzeit erheblich belasten. Seit Erscheinen der Algorithmenbibel von D. Knuth wird allenthalben vorgeschlagen, die Zahlen a und b durch „Erweitern“ so anzupassen, daß $b \geq B/2$ ist, dadurch ist gesichert, daß wie oben der geratene Quotient Q um höchstens 2 vom richtigen Quotienten abweicht. Ich hatte dies erst nachträglich in meine Implementation eingefügt und habe tatsächlich eine Beschleunigung erlebt.
Einen anderen Vorschlag hat W. Pohl (z.Z. Kaiserslautern) gemacht, der deutlich besser ist:
Wir handeln den Fall, daß b eine einstellige Zahl ist, extra ab, das ist auch recht einfach. Nun bilden wir aus den jeweils beiden ersten Stellen von a und b Long-Cardinal-Zahlen und wählen deren Quotienten als Näherung für Q, da liegen wir schon ganz richtig.

Ich will uns den Abdruck der Divisionsprozedur ersparen, die nimmt etwa zwei Druckseiten ein.
Die vorgestellten Prozeduren für die Grundrechenarten sind in Pascal oder MODULA geschrieben. Bei der Multiplikation sehen Sie, daß häufig Wort-Variable auf Longint-Variable „gecastet“ werden, weil das Produkt zweier 16-Bit-Zahlen eben eine 32-Bit-Zahl ist. Es ist vorstellbar, daß man effektiver arbeiten könnte, wenn man die Fähigkeiten des Prozessors besser nutzen würde. Henri Cohen schreibt, daß eine Beschleunigung um den Faktor 8 erreicht werden würde, wenn man in Assemblersprache programmiert. Meine Erfahrungen gehen noch weiter: In einer früheren Version meines CA-Systems, wo die Zahllänge konstant war (64 Byte), hat Sven Suska für die Addition und die Multiplikation Assembler-Routinen geschrieben, allein deren Einsatz brachte eine Beschleunigung um den Faktor 6. Auch für die Division hat er eine Routine geschrieben, die leider den Nachteil hatte, ab und zu den Rechner zum Absturz zu bringen. Wenn sie aber fehlerfrei funktionierte, so erbrachte das nochmals eine Beschleunigung un den Faktor 6. Assembler-Routinen sind allerdings schwer zu transportieren; die erwähnten funktionierten nur im Real-Modus des 286er Prozessors, wo man maximal 500 KByte Speicher zur Verfügung hat. Außerdem waren sie sehr umfangreich (über 18 KByte Quelltext) und für einen Laien wie mich völlig unverständlich.
Cohen schlgt vor, nur einige Grundfunktionen in Assembler zu schreiben. Dazu sollen zwei globale Variable namens `remainder` und `overflow`, die überall bekannt sind, geschaffen werden. Die Zahl-Basis sei `M = 2` ^{16}, `a, b` und `c` seien vorzeichenlose 2-Byte-Zahlen.
Die Grundfunktionen sollen folgendes leisten:

```
c = add(a, b)      : a + b = overflow * M + c
c = addx(a, b)     : a + b + overflow = overflow * M + c
c = sub(a, b)      : a - b = c - overflow * M
c = subx(a, b)     : a - b - overflow = c - overflow * M
c = mul(a, b)      : a * b = remainder * M + c
c = div(a, b)      : remainder * M + a = b * c + remainder
c = shiftl(a, k)   : 2^k * a = remainder * M + c
c = shiftr(a, k)   : a * M / 2^k = c * M + remainder
```

Aus diesen kurzen Funktionen kann man dann die oben angeführten Prozeduren für lange Zahlen zusammensetzen.

Wir fahren fort.
Wir wollen den größten gemeinsamen Teiler zweier Zahlen bestimmen, dafür verwenden wir den Euklidischen Algorithmus. Die Prozedur `nlRemainder` liefert nur den Divisionsrest, nicht den Quotienten.

```
PROCEDURE nlGcd(a, b:IntNumber;  VAR z:IntNumber);
VAR h, x, y: IntNumber;
BEGIN
  CASE compabs(a,b) OF    (* Vergleich der Absolutbeträge *)
    2: x:= b;
      nlCopi(a,y);     (* y:= a *)
```

```
    |1: x:= a;
      nlCopi(b,y);
    |0: nlCopi(a, z);
      z^.n:= FALSE;
      RETURN;
  END;
  nlRemainder(x,y,h);
  WHILE h<>NIL DO
    x:= y; y:= h;
    nlRemainder(x,y,h);
    nlWegl(x);
  END;
  y^.n:= FALSE;
  z:= y;
END nlGcd;
```

Eine Variante des Euklidischen Algorithmus liefert für den größten gemeinsamen Teiler g von a und b gleich seine Darstellung als Vielfachsumme: $g = au + bv$.

```
PROCEDURE ggtuv(a,b: IntNumber;  VAR u,v,ggd: IntNumber);
VAR z,s,x,y,xx,yy,q,r,aa,bb: IntNumber;
BEGIN
  IF eq(0, a) OR eq(0, b) THEN
    nlSet1(u); nlSet1(v);
    IF eq(0, a) THEN
      nlCopi(b, ggd); RETURN
    ELSE
      nlCopi(a, ggd); RETURN
    END;
  END;
  nlCopi(a, aa); nlCopi(b, bb);
  nlSet1(x); nlCopi(x, yy);
  xx:= NIL; nlCopi(xx, y);
  WHILE b<>NIL DO
    nlDivmod(a,b,q,r); nlWegl(z);
    nlCopi(yy, z); nlMl(q,yy,s);
    sl(y,s,yy); nlWegl(y);
    nlCopi(z, y); nlWegl(z);
    nlCopi(xx, z); nlMl(q,xx,s);
    sl(x,s,xx); nlWegl(x);
    nlCopi(z, x); nlWegl(a);
    nlCopi(b, a); nlWegl(b);
    nlCopi(r,b); nlWegl(r);
  END;
  IF x^.n THEN
```

```
    nlWegl(z); nlCopi(x, z);
    z^.n:= FALSE; nlDivmod(z,bb,q,r);
    nlSet1(r); al(q,r,q);
    nlMl(q,bb,z); al(x,z,u);
    nlMl(q,aa,z); sl(y,z,v);
  ELSE
    nlCopi(x, u); nlCopi(y, v);
  END;
  nlCopi(a, ggd);
  IF ggd^.n THEN
    ggd^.n:= FALSE;
    IF u<>NIL THEN
      u^.n:= NOT u^.n;
    END;
    IF v<>NIL THEN
      v^.n:= NOT v^.n;
    END;
  END;
END ggtuv;
```

Das ist natürlich noch nicht der Weisheit letzter Schluß. Wenn die Eingabewerte die Größenordnung N haben, so durchläuft der Euklidische Algorithmus etwa $\log(N)$mal eine Schleife, und jedesmal wird eine Langzahldivision durchgeführt. Der folgende Algorithmus von Lehmer verwendet überwiegend Wort-Divisionen. Die Zahlen `a, b` seien gegeben, die Variablen `ah, bh, A, B, C, D, T, q` sind Worte, `t` und `r` sind lange Zahlen, das Zeichen „/“ bezeichnet die ganzzahlige Division.

```
1. Wenn b < M ist, so verwende den alten Algorithmus, und beende.
   Sonst: ah = oberste Stelle von a,
          bh = oberste Stelle von b,
          A:= 1, B:= 0, C:= 0, D:= 1.
2. Wenn bh + C = 0 oder bh + D = 0, so gehe zu 4.
   Sonst: q:= (ah + A)/(bh + C).
   Wenn  q <> (ah + B)/(bh + D) ist, so gehe zu 4.
   (Hier kann ein "Uberlauf auftreten, den man abfangen mu"s.)
3. T:= A - q*C, A:= C, C:= T,
   T:= B - q*D, B:= D, D:= T,
   T:= ah - q*bh, ah:= bh, bh:= T,
   gehe zu 2.
4. Wenn B = 0 ist, so sei t = a mod b, a:= b, b:= t, gehe zu 1.
   (Hier braucht man Langzahlarithmetik, dieser Fall tritt jedoch
   (angeblich) nur mit der Wahrscheinlichkeit 1,4/M = 0,00002 auf.)
   Sonst: t:= A*a, t:= t + B*b, r:= C*a, r:= r + D*b, a:= t, b:= r,
          Gehe zu 1.
```

Eine weitere Variante vermeidet Divisionen (fast) vollstndig, es kommen nur Subtraktionen und Verschiebungen vor:

```
1. r:= a mod b, a:= b, b:= r.
   (Hier wird ein einziges Mal dividiert, dieser Schritt kann auch
   weggelassen werden. Von nun an haengt aber die Zahl der Schritte
   nur noch von der Laenge der kleineren der eingegebenen Zahlen ab.)
2. Wenn b = 0 ist, so ist a das Ergebnis.
   Sonst: k:= 0,
          solange a und b beide gerade sind: k:= k + 1, a:= a/2, b:= b/2.
   (Am Ende haben wir 2^k als genauen Faktor des ggT.)
3. Wenn a gerade ist, so wiederhole a:= a/2, bis a ungerade ist.
   Wenn b gerade ist, so wiederhole b:= b/2, bis b ungerade ist.
4. t:= (a-b)/2
   wenn t = 0 ist, so ist 2^k*a das Ergebnis.
5. Solange t gerade ist, wiederhole t:= t/2.
   Wenn t > 0 ist, so a:= t.
   Sonst: b:= -t.
   Gehe zu 4.
```

Als fünfte Grundrechenart wollen wir die Potenzierung betrachten. Es hat sicher nicht viel Sinn, etwa a^b bilden zu wollen, wobei auch der Exponent b beliebig lang ist, es reicht vielleicht schon $b < 32000$. Wir verwenden einen Trick, der nicht b Multiplikationen erfordert, sondern nur $\log(b)$ Operationen: Wir sehen es am Beispiel

$$a^8 = ((a^2)^2)^2.$$

```
(*Power of long; lu:= x êxp *)

PROCEDURE pl(x:IntNumber;  exp :INTEGER;  VAR lu:IntNumber);
VAR wh, yh, y, w, w1, w2: IntNumber;
BEGIN
  nlSet1(y);
  nlCopi(x, w);
  exp :=ABS(exp);
  WHILE exp >0 DO
    IF ODD(exp) THEN
      nlMl(y,w,yh);
      nlWegl(y);
      y:= yh;
    END;
    nlCopi(w, w1);
    w2:= w;
    IF exp  > 1 THEN
      nlMl(w1,w2,w);
    END;
    nlWegl(w1);
    nlWegl(w2);
    exp:=exp  DIV 2;
```

```
  END;
  nlCopi(y, lu);
END pl;
```

Auch dieser Algorithmus läßt sich verbessern, zwar nicht in der Anzahl der Rechenschritte, aber in der Größe der Operanden: Im folgenden Algorithmus wird im Schritt 3 immer mit *derselben* Zahl `z` multipliziert.
Um g^n zu berechnen, benötigen wir die Zahl e mit $2^e \geq n < 2^{e+1}$, um diese zu bestimmen, müssen die Bits von n durchmustert werden.

```
1. N:= n, z:= g, y:= z, E:= 2^e.
2. Wenn E = 1 ist, so ist y das Ergebnis.
   Sonst: E:= E/2.
3. y:= y * y
   Wenn N >= E ist, so N:= N - E, y:= y * z.
   Gehe zu 2.
```

Man kann die Division vermeiden, wenn man sich die Bits von `n` anschaut:

```
1. N:= n, z:= g, y:= z, f:= e.
2. Wenn f = 0 ist, so ist y das Ergebnis.
   Sonst: f:= f - 1.
3. y:= y * y
   Wenn bit(f, N) = 1 ist, so y:= y * z.
   Gehe zu 2.
```

Zum Abschluß wollen wir uns dem Problem der Ein- und Ausgabe langer ganzer Zahlen widmen, einem Fragenkreis, der in den einschlägigen Computeralgebra-Büchern ausgespart bleibt. Wir geben hier PASCAL-Prozeduren an, mit MODULA ist es nicht ganz so einfach. Es versteht sich, daß die „eingebauten“ Prozeduren zum Lesen von 2-Byte- oder Gleitkommazahlen nicht brauchbar sind. Wir lesen also eine Zeichenkette wie '123444444444444444444' und wandeln diese in eine lange Zahl um, dazu lesen wir (von links) Zeichen für Zeichen, wandeln es in eine Zahl um, multiplizieren mit 10 und addieren die nächste Ziffer:

```
procedure readl(var a: IntNumber);
var s: string;
    h, zehn: IntNumber;
    i: integer;
begin
  readln(s);
  assic(10, zehn);     { zehn := 10 }
  a:= NIL;
  for i:= 1 to Length(s) do
  begin
    mult(a, zehn, b);
    assic(ord(s[i]) - ord('0'), h);
```

```
    { h ist die Zahl, die das Zeichen s[i] darstellt }
    wegl(a);
    add(b, h, a);
    wegl(b);
    wegl(h);
  end;
end;
```

Das Schreiben ist auch nicht schwierig, wir können aber die Ziffern nicht sofort ausgeben, weil wir sie von rechts nach links berechnen müssen:

```
procedure writel(a: IntNumber);
var s: string;
q, r, zehn: IntNumber;
i: integer;
begin
  s:= '';
  while a <> NIL do
  begin
    divmod(a, zehn, q, r);
    a:= q;
    i:= r^.m[1];
    s:= char(i+ord('0')) + s;
    { das Zeichen der Ziffer i wird an s vorn angehängt }
  end;
  write(s);
end;
```

Wir sehen, daß wir bereits zur Ein- und Ausgabe die Rechenoperationen mit langen Zahlen benötigen. Das macht den Test der Arithmetik etwas schwierig, da man sich Zwischenergebnisse, etwa innerhalb der Prozedur `divmod`, nicht mit `writel` ausgeben lassen kann.

4 Rationale Zahlen

Wir stellen rationale Zahlen als Brüche dar:

```
type rat = record
  z, n: IntNumber;
  s: boolean;
end;
```

Zähler z und Nenner n sind lange ganze Zahlen, im Feld s wird angegeben, ob diese Zahl gekürzt wurde oder nicht. Nehmen wir uns zuerst das Kürzen vor, wir berechnen den größten gemeinsamen Teiler und dividieren ihn weg (die Prozedur nlE1 stellt fest, ob die Zahl, auf die sie angewendet wird, gleich 1 ist):

Die Rechenoperationen mit Brüchen realisieren wir, wie wir es gelernt haben. Dabei behandeln wir die Fälle, wo Nenner gleich 1 sind, gesondert. Begründung: Es werden zwar ein paar Tests mehr gemacht, als wenn wir alles über einen Kamm scheren, aber unnötige Rechenoperationen, vor allem aber unnötige Speicherplatzanforderungen und anschließende Freigabe werden vermieden.

```
PROCEDURE nlNormalize(VAR x:rat); (* simplify x *)
VAR zh, nh, r, d: lint;
BEGIN
  IF x.s THEN RETURN
  END;
  IF ((x.n^.l=1) AND (x.n^.m[1]=1)) OR
    ((x.z^.l=1) AND (x.z^.m[1]=1)) THEN
    RETURN
  END;
  nlGcd(x.z,x.n,d);
  IF NOT nlE1(d) THEN
    nlDivmod(x.z,d,zh,r); nlWegl(x.z);
    x.z:= zh;
    nlDivmod(x.n,d,nh,r); nlWegl(x.n);
    x.n:= nh;
  END;
  nlWegl(d);
  x.s:= TRUE;
END nlNormalize;
```

Davenport schlägt vor, Brüche immer zu kürzen, denn wenn

$$\frac{a}{b} \cdot \frac{c}{d} = \frac{p}{q} \text{ ist, so ist } \mathrm{ggT}(p,q) = \mathrm{ggT}(a,d) \cdot \mathrm{ggT}(b,c)$$

und die Zahlen bei der ggT-Berechnung sind rechts kleiner als links. Für die Addition gilt

$$\frac{a}{b} + \frac{c}{d} = \frac{a \cdot \frac{d}{\mathrm{ggT}(b,d)} + c \cdot \frac{b}{\mathrm{ggT}(b,d)}}{\frac{b \cdot d}{\mathrm{ggT}(b,d)}}$$

Man überlegt sich aber schnell, daß man dabei keine Zeit spart.

Addition:

```
PROCEDURE nlAdd(VAR la,li,lu: rat); (* lu:= la + li *)
VAR x,y:lint;
BEGIN
  IF la.z=NIL THEN nlCopy(li, lu); RETURN (* la = 0 ? -> Kopieren *)
  END;
  IF li.z=NIL THEN nlCopy(la, lu); RETURN (* li = 0 ? *)
  END;
  IF nlE1(la.n) THEN (* la ganz ? *)
```

```
    IF nlE1(li.n) THEN (* li ganz ? *)
      add(la.z, li.z, lu.z); (* Summe der Zaehler *)
      nlSet1(lu.n); (* Nenner := 1 *)
      lu.s:= TRUE;
      RETURN
    ELSE (* li nicht ganz *)
      nlMl(la.z, li.n, x);
      add(x, li.z, lu.z); nlWegl(x);
      nlCopi(li.n, lu.n); (* Nenner kopieren *)
      lu.s:= FALSE;
      RETURN;
    END
  ELSE
    IF nlE1(li.n) THEN (* la nicht ganz, li ganz *)
      nlMl(li.z, la.n, y);
      add(la.z, y, lu.z); nlWegl(y);
      nlCopi(la.n, lu.n);
      lu.s:= FALSE;
      RETURN;
    ELSE (* allgemeiner Fall *)
      nlMl(la.z, li.n, x);
      nlMl(li.z, la.n, y);
      add(x, y, lu.z); nlWegl(x); nlWegl(y);
      IF lu.z=NIL THEN (* Ergebnis 0 ? *)
        nlSet1(lu.n); lu.s:= TRUE; RETURN; (* Nenner := 1 *)
      END;
      nlMl(la.n, li.n, lu.n);
      lu.s:= FALSE; (* nicht gekuerzt *)
    END;
  END;
END nlAdd;

PROCEDURE nlMult(VAR la, li, lo: rat); (* lo := la * li *)
BEGIN
  IF (la.z=NIL) OR (li.z=NIL) THEN
    lo.z:=NIL; nlSet1(lo.n); lo.s:= TRUE;
    RETURN;
  END;
  IF nlE1(la.n) AND nlE1(li.n) THEN
    nlMl(la.z, li.z, lo.z);
    nlSet1(lo.n);
    lo.s:= TRUE;
    RETURN
  END;
  nlMl(la.z, li.z, lo.z);
```

```
  nlMl(la.n, li.n, lo.n);
  lo.s:= FALSE;
END nlMult;

PROCEDURE nlDiv(VAR la, li, lo: rat); (* lo := la / li *)
BEGIN
  IF la.z=NIL THEN lo.z:=NIL; nlSet1(lo.n); lo.s:= TRUE; RETURN;
  END;
  nlMl(la.z, li.n, lo.z);
  nlMl(la.n, li.z, lo.n);
  lo.s:= FALSE;
END nlDiv;
```

Bevor wir uns abschließend mit dem Problem des Kürzens beschäftigen wollen, soll der Hintergrund genauer beschrieben werden. Ein Beispiel einer Gröbnerbasenberechnung war im Fall der Charakteristik 0 ein harter Brocken und Anlaß zur genaueren Untersuchung des Verhaltens der implementierten Algorithmen. Die errechnete Gröber-Basis besteht aus drei Polynomen vom Grad < 50, aber während der Rechnung treten Zahlen auf, die bis zu 1000 Byte lang sind (ca. 2500 Dezimalstellen, etwa ein Bildschirm voll). Zuerst stellte sich heraus, daß die Speicherverwaltung des benutzten Systems die Rechenzeit erheblich beeinflußte: Bei TopSpeed-Modula auf DOS-Rechnern konnte man die Speicherverwaltung für lange Zahlen ohne weiteres dem eingebauten Storage-Modul überlassen, auf Mac-Rechnern war dies sehr uneffektiv, hier konnten durch Einsatz einer privaten Speicherverwaltung erst einmal ähnlich Laufzeiten wie auf einem 386er AT (33 MHz) erreicht werden: etwa 90 Minuten. Durch eine weitere Verbesserung des Buchberger-Algorithmus konnten als nächstes Laufzeiten um 25 Minuten erreicht werden, eine deutliche Beschleunigung. Nun wurde noch einmal an der Arithmetik gefeilt. Der Karatsuba-Algorithmus für die Multiplikation erlaubt es, Zahlen der Länge n in etwa $n^{1.5}$ Zeiteinheiten (anstatt n^2) zu multiplizieren. Dies konnte unter Laborbedingungen bestätigt werden: In einem Testprogramm wurden je zwei gleichlange Zahlen multipliziert, bei 40-Byte-Zahlen betrug die Einsparung etwa 10 %, bei 100-Byte Zahlen 20% . Wurden jedoch eine n-Byte-Zahl mit einer $n/2$-Byte-Zahl multipliziert, so wurden die Rechenzeiten mit wachsendem n beim Karatsuba-Algorithmus immer schlechter. Dieser Ansatz wurde verworfen. Dennoch wurde durch Weglassen unnötiger Tests (wird etwa durch Null dividiert? usw.) nochmals eine Verkürzung der Rechenzeit um 20 Sekunden erreicht.
Wir kommen nun zur Frage, wann ein Rechenergebnis gekürzt werden soll.

Die Wahrscheinlichkeit, daß Zähler und Nenner eines Bruchs beide durch die Primzahl p teilbar sind, ist $\frac{1}{p^2}$, für $p = 2,\ 3,\ 5$ also 25 %, 11 %, 4 %, und beim Herauskürzen solch kleiner Zahlen verringert sich die Stellenzahl (zur Basis B) eher selten, so daß keine Beschleunigung der Rechnung eintreten wird.

Um zu Kürzen, muß ein ggT berechnet werden, dies ist eine aufwendige Rechenoperation. Und der Aufwand war vielleicht vergebens, wenn sich herausstellt, daß der gegebene Bruch unkürzbar ist. In einer früheren Implementation wurde der Versuch

des Kürzens unterlassen, wenn nicht Zähler oder Nenner eine gewisse Länge erreicht haben. Diese Schwelle konnte im Dialog durch Belegung einer Variablen verändert werden, die Voreinstellung war 63. Schließlich wurde die Auswirkung dieser Variablen auf die Rechenzeit überprüft. Diese war unerwartet groß. Es ergaben sich folgende Zeiten für exakt dieselben Rechnungen
(Die Schranke wurde in 10er-Schritten erhöht):

Schranke	Zeit (sec)	
1..71	1572..1787	(min bei 31, max bei 51)
81	1539	
91	2967	
101	2063	
111	1495	
121	3008	(Verdopplung !)
131	2662	
141	1621	
151	3687	

Ich vermutete, daß man keinen Rat geben kann, welcher Wert „gut“ ist. Für die Schrankenwerte 11 und 111 wurden die Längen der zu kürzenden Zahlen überprüft. Im ersten Fall hatten die Kandidaten erwartungsgemäß nur kurze gemeinsame Teiler, ab und zu auch gar keinen. Im zweiten Fall war nicht nur das Kürzen (soweit gesehen) stets erfolgreich, sondern die gemeinsamen Faktoren hatten eine erhebliche Länge (60...80% der Länge der Kandidaten). Also: Wenn „krumme“ Zahlen erwartet werden, sollte man die Kürzungs-Schwelle nicht zu klein wählen. Vielleicht ist 63 gar nicht so eine schlechte Wahl?
Aber die Wahrheit lag doch woanders: Sicherlich ist es sinnvoll, solche Zahlen, mit denen häufig operiert wird, in gekürzter Darstellung abzuspeichern. Während man bei einem universellen Programm nicht hervorsehen kann, welche Zahlen der Nutzer weiterzuverwenden gedenkt, war in unserem Fall die Lage übersichtlicher: Es ist zu vermuten, daß die Koeffizienten der Polynome in der Standardbasis häufig benutzt werden. Die Festlegung, daß bei der Einfügung eines Polynoms in die Standardbasis alle Koeffizienten zu kürzen sind, erbrachte einerseits eine bedeutende Rechenzeitersparnis und andererseits die Erkenntnis, daß die Rechenzeit doch nicht, wie oben vermutet, unkontrolliert von der Kürzungsschwelle abhängt. In einer Testserie wurde diese Schwelle schrittweise verdoppelt, hier sind die Ergebnisse:

Schranke	Zeit (sec)
16	1673
32	1480
64	1210
128	788
256	628
512	558
1024	546
2048	541

und dieser Wert blieb auch bei den nächsten drei Verdopplungen stabil. Noch eine Nebenbemerkung: Das obengenannte Beispiel (anfangs 90 Minuten Rechenzeit) war Ausgangspunkt mehrerer prinzipieller Überlegungen. Einerseits wurde der Gröbnerbasenalgorithmus weiter verbessert. Ferner wurde bei den Polynomoperationen alle Nenner heraufmultipliziert, also nur noch ganzzahlig gerechnet. Dies erbrachte eine Halbierung der Rechenzeit. Einige weitere Optimierungen in den arithmetischen Grundoperationen („mache nie etwas Unnötiges“) brachten die Rechenzeit auf nunmehr 15 Sekunden (kein Schreibfehler!).

Ein ähnliches Problem hat man beim Gaußschen Algorithmus: hier ist es sinnvoll, die Einträge der Zeile, die das aktuelle Pivot-Element enthält, zu kürzen, denn mit diesen Zahlen werden viele Operationen durchgeführt.

Der folgende Satz von Dirichlet bringt etwas mehr Licht in die Angelegenheit:

Satz 4.1 *Seien u und v zwei zufällig gewählte natürliche Zahlen, dann ist die Wahrscheinlichkeit, daß u und v teilerfremd sind, gleich* $6/\pi^2 = 0,608$.

Für den Beweis verweisen wir auf P. Naudin, C. Quitté, Algorithmique algébrique, Masson 1992, II-6.3.

5 Polynome und transzendente Körpererweiterungen

Wenn x, y, z Unbestimmte sind, so nennt man einen Ausdruck wie $5x^2+7xy+z^5$ ein Polynom. Die Menge aller Polynome in x, y, z mit Koeffizienten aus einem Körper K wird mit $K[x, y, z]$ bezeichnet. Polynome kann man addieren, subtrahieren, multiplizieren, die Menge $K[x, y, z]$ ist ein Ring.
Wie stellt man Polynome im Rechner dar? Einen Term $x^i y^j z^k$ nennt man ein Monom. Wenn man sich merkt, daß x die 1., y die 2., z die 3. Unbestimmte ist, so ist dieses Monom durch seinen „Exponentenvektor“ $[i, j, k]$ eindeutig bestimmt. Wenn man die Größe der Exponenten beschränkt, so ist die Zahl der möglichen Monome beschränkt, man kann also ein Polynom folgendermaßen abspeichern:

Man numeriert alle möglichen Monome durch (es seien etwa N Stück), dann schafft man sich ein Feld mit N Komponenten und merkt sich zu jedem Monom den entsprechenden Koeffizienten. Wenn ein Monom in einem Polynom gar nicht vorkommt, ist der entsprechende Koeffizient eben gleich Null.
Diese Darstellungsform wird die „dichte“ genannt.

Es ist aber auch eine „operationale“ Darstellung in der Form

$$(x + y)(z + w + u)$$

vorstellbar, die platzsparend ist. Oder auch nicht:

$$(x - y)(x^{n-1} + x^{n-2}y + \ldots + xy^{n-1} + y^n) = x^n - y^n$$

In manchen CA-Systemen werden auch Zahlen nicht unbedingt ausgerechnet, z. B. kann $5+17^3$ als `(plus, 5, (power(17, 3))` gespeichert werden. Dabei wird die eigentliche Rechnung einfach auf später verschoben.

Wir wollen uns nun auf die „dünne“ Darstellung beziehen, die nur die wirklich vorkommenden Monome und ihre Koeffizienten speichert.

```
type monom = array[1..max] of integer;
type polynom = Pointer to polrec;
polrec = Record
  c: rat; (* Koeffizient *)
  e: monom; (* Exponentenvektor *)
  n: polynom;
end;
```

In der Komponente `n` (= next) haben wir die Adresse des nächsten Summanden gespeichert, beim letzten Summanden setzen wir $n = \text{NIL}$.
Vernünftig ist es, die Monome nicht zufällig in so eine Liste einzutragen, man ordnet sie am Besten der Größe nach. Wir brauchen also eine Vergleichsrelation für Monome. Der Grad eines Monoms ist die Summe seiner Exponenten, der Grad eines Polynoms ist das Maximum der Grade seiner Monome. Ein Monom soll größer als ein anderes sein, wenn sein Grad größer ist. Für Monome vom Grad 1 legen wir eine Ordnung willkürlich fest, etwa $x > y > z$. Für Polynome gleichen Grades können wir jetzt die lexikographische Ordnung nehmen:

$$x^3 > x^2y > xy^2 > y^3 > x^2z > y^2z > z^3.$$

Diese Grad-lexikographische Ordnung ist nur eine aus einer Vielzahl von Ordnungsmöglichkeiten. Man fordert aber gewöhnlich, daß für Monome p, q, r mit $p > q$ auch stets $pr > qr$ gilt, dies ist hier erfüllt.

Da der Integer-Bereich für die einzelnen Exponenten nie ausgeschöpft werden wird, kann man das Feld `monom` natürlich packen, wenn man sich auf eine Beschränkung der einzelnen Maximalgrade einläßt.

Nun können wir Rechenoperationen für Polynome implementieren:

Addition:

Seien zwei Polynome p, q gegeben, in beiden seien die Monome der Größe nach geordnet. Die einfachste, wenn auch zeitaufwendige Möglichkeit wäre es, das eine Polynom an das andere anzuhängen und das Ergebnis zu sortieren. Besser ist es, sortierte Listen auch sortiert zu verarbeiten:

Wir schauen uns die jeweils größten Monome an. Es gibt drei Möglichkeiten:

1. $p\hat{}.e > q\hat{}.e$, dann übernehmen wir $p\hat{}.e$ nebst seinem Koeffizienten in $p+q$ und ersetzen p durch $p\hat{}.n$.

2. $p\hat{}.e < q\hat{}.e$, dann übernehmen wir $q\hat{}.e$ nebst seinem Koeffizienten in $p+q$ und ersetzen q durch $q\hat{}$.n.

3. $p\hat{}.e = q\hat{}.e$, dann bilden wir $a = p\hat{}.c + q\hat{}.c$, wenn $a \neq 0$ ist, so übernehmen wir $p\hat{}.e$ mit dem Koeffizienten a in $p+q$ und ersetzen p durch $p\hat{}.n$ und q durch $q\hat{}.n$.

Dies wiederholen wir, bis p und q abgearbeitet sind, also beide auf NIL zeigen.

Die Subtraktion ist analog zu behandeln.

Damit können wir die Eingabe von Polynomen organisieren:

```
p:= nil;
repeat
  liesmonom(m);
  p:= p + m;
until m = nil;
```

Bei der Multiplikation bekandeln wir zuerst den Spezialfall, wo das Polynom q nur einen Summanden besitzt. Hier ist einfach jeder Summand von p mit $q\hat{}.c$ zu multiplizieren, zu den Exponentenvektoren von p ist $q\hat{}.e$ zu addieren. Da eventuelle sehr oft mit derselben Zahl $q\hat{}.c$ zu multiplizieren ist, ist es ratsam, diese Zahl am Anfang zu kürzen. Den allgemeinen Fall führen wir auf den Spezialfall zurück: Für jeden Summanden m von q bilden wir $m \cdot p$ und addieren all diese Polynome.

Der Quotientenkörper $K(x,y,z)$ des Polynomrings $K[x,y,z]$ besteht aus allen Brüchen $\frac{f(x,y,z)}{g(x,y,z)}$ mit $g(x,y,z) \neq 0$. Rechenoperationen mit derartigen „rationalen Funktionen“ kann man in Analogie zum Rechnen mit rationalen Zahlen implementieren. Versuchen Sie aber bitte nicht, zum Zwecke des Kürzens eines Bruchs den größten gemeinsamen Teiler zweier Polynome in mehreren Veränderlichen zu berechnen. Hierfür gibt es keinen Euklidischen Algorithmus.

Im Falle von Polynomen in nur einer Veränderlichen sind die Dinge einfacher:

Satz 5.1 *(Division mit Rest): Seinen $f(x), g(x) \in \mathbf{Q}[x]$ gegeben, dann gibt es eindeutig bestimmte Polynome $q(x), r(x)$, so daß $f(x) = q(x) \cdot g(x) + r(x)$ gilt, wobei der Grad von $r(x)$ kleiner als der Grad von $g(x)$ ist.*

Beweis: Wenn $\deg(g) > \deg(f)$ ist, so setzen wir $q = 0$ und $r = f$. Weiterhin sei $\deg(f) >= \deg(g)$. Wir führen die Induktion über $n = \deg(f)$.
Ohne Beschränkung der Allgemeinheit können wir annehmen, daß die höchsten Koeffizienten von f und g gleich 1 sind (solche Polynome heißen „normiert“).
Sei also $n = 1$, d.h. $f(z) = z + a$. Dann ist $g(z) = z + b$ oder $g(z) = 1$, im ersten Fall wählen wir $q(z) = 1, r(z) = a - b$ und im zweiten Fall $q(z) = f(z),\ r(z) = 0$.
Wir setzen nun voraus, daß den Satz für Polynome von einem Grad, der kleiner als n ist, bewiesen ist. Sei also

$$f(z) = z^n + a_1 z^{n-1} + \ldots,\ g(z) = z^m + b_1 z^{m-1} + \ldots,$$

dann setzen wir $q_1(z) = z^{n-m}$, es ist

$$q_1(z)g(z) = x^n + b_1 x^{n-1} + \ldots$$

und das Polynom

$$f_1(z) = f(z) - q_1(z)g(z) = (a_1 - b_1)z^{n-1} + \ldots$$

hat einen Grad, der kleiner als n ist, also gibt es Polynome $q_2(z)$ und $r(z)$ mit $f = q_2 g + r$ und wir wissen, daß $r = 0$ oder $\deg(r) < \deg(g)$ gilt. Dann haben wir mit

$$f = f_1 + q_1 g = (q_2 + q_1)g + r$$

die gewünschte Zerlegung gefunden.
Wir zeigen noch die Einzigkeit von $q(x)$ und $r(x)$. Sei etwa noch

$$f(x) = s(x) \cdot g(x) + t(x), \deg(t) < \deg(g),$$

dann ist

$$(q(x) - s(x)) \cdot g(x) = t(x) - r(x).$$

Wenn $q(x) \neq s(x)$ wäre, stünde links ein Polynom, dessen Grad mindetens gleich $\deg(g)$ ist, und rechts ein Polynom, dessen Grad kleiner als $\deg(g)$ ist.

Folgerung 5.2 *Sei $a \in \mathbf{Q}$ eine Nullstelle von $f(x)$, dann ist das lineare Polynom $x - a$ ein Teiler von $f(x)$.*

Beweis: Wir teilen $f(x)$ durch $x - a$:

$$f(x) = q(x) \cdot (x - a) + r,$$

der Grad von r ist kleiner als 1, also ist r eine Konstante. Wir setzen $x = a$ ein und finden $f(a) = r$, also ist $r = 0$. Das heißt, $f(x)$ ist durch $x - a$ teilbar.

Lemma 5.3 *Ein Polynom $f(x)$ besitzt genau dann eine mehrfache Nullstelle r, wenn $f'(r) = 0$ ist.*

Beweis: Sei $f(x) = (x-r)^k \cdot g(x), k > 1$. Dann ist

$$f'(x) = k \cdot (x-r)^{k-1} \cdot g(x) + (x-1)^k \cdot g'(x),$$

also ist r eine gemeinsame Nullstelle von f unf f'.

Folgerung 5.4 *Sei $g = ggT(f, f')$; dann besitzt f/g dieselben Nullstellen wie f, aber jede Nullstelle ist einfach.*

Wir wollen uns nun mit einem klassischen Verfahren der näherungsweisen Nullstellenberechnung befassen.
Das einfachste Näherungsverfahren ist die Newton-Interpolation: Wenn für eine reelle Zahl a der Wert von $f(a)$ beinahe Null ist, so ist $a - f(a)/f'(a)$ eine bessere Näherung. Eine Voraussetzung dafür ist aber, daß f'' in der Nähe des Startpunkts der Iteration sein Vorzeichen nicht ändert. Man muß also erst einmal in die Nähe einer Nullstelle geraten.
Als erstes kann man ein Intervall angeben, in dem alle Nullstellen liegen müssen.

Satz 5.5 (Cauchysche Ungleichung)
Sei $f(x) = \sum a_i x^{n-i} \in \mathbf{C}[x]$, dann gilt für jede Nullstelle z von $f(x)$ die Ungleichung

$$\mid z \mid < 1 + \frac{\max(\mid a_1 \mid, \ldots, \mid a_n \mid)}{\mid a_0 \mid}.$$

Beweis: Sei $h = \max(\mid a_1 \mid, \ldots, \mid a_n \mid)$ und $f(z) = 0$, dann ist

$$a_0 z^n = -a_1 z^{n-1} - \ldots - a_n,$$

$$\mid a_0 \mid\mid z^n \mid \leq h \cdot (\mid z \mid^{n-1} + \ldots + 1) < \frac{h \cdot \mid z \mid^n}{\mid z \mid - 1},$$

also $\mid a_o \mid \cdot (\mid z \mid - 1) < h$.

Als nächstes behandeln wir ein Verfahren, das es gestattet, die Anzahl der Nullstellen eines Polynoms in einem gegebenen Intervall zu berechnen. Durch fortgesetzte Intervallteilung kann man jede einzelne Nullstelle dann beliebig genau einschachteln.

Der Sturmsche Lehrsatz
Sei $f(x) \in \mathbf{Q}[x]$ und $a \in \mathbf{Q}$. Wie oben haben wir

$$f(x) = (x-a)q(x) + f(a),$$

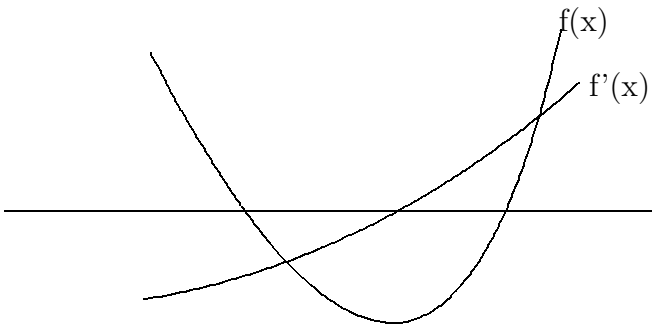
also

$$q(x) = \frac{f(x) - f(a)}{x - a}.$$

Wenn a eine einfache Nullstelle von $f(x)$ ist, so ist

$$q(a) = f'(a) \neq 0$$

und dies gilt in einer Umgebung von a. Es gibt zwei Fälle:



1) $f'(a) > 0$, dann ist $f(x)$ bei a monoton wachsend; wenn x von links durch a hindurch läuft, ist $f(x)$ zuerst negativ, dann positiv.
2) $f'(a) < 0$, dann ist $f(x)$ bei a monoton fallend; wenn x von links durch a hindurch läuft, ist $f(x)$ zuerst positiv, dann negativ.
Beiden Fällen ist folgendes gemeinsam: Wenn x wachsend durch a läuft, gehen die Funktionen $f(x)$ und $f'(x)$ von verschiedenen zu gleichen Vorzeichen über.
Wir konstruieren nun eine „Sturmsche Kette“, wir setzen $f_1(x) = f'(x)$ und dividieren fortlaufend mit Rest:

$$\begin{aligned} f &= q_1 \cdot f_1 - f_2 \\ f_1 &= q_2 \cdot f_2 - f_3 \\ &\dots \\ f_{i-1} &= q_i \cdot f_i - f_{i+1} \\ &\dots \\ f_m &= c \text{ konstant.} \end{aligned}$$

Beachten Sie das Minuszeichen.
Wenn $f(x)$ nur einfache Nullstellen besitzt, ist $\mathrm{ggT}(f, f_1) = 1$, also $f_m \neq 0$.
Nun gilt: Für keine Zahl $r \in \mathbf{R}$ gibt es ein i mit $f_i(r) = 0 = f_{i+1}(r)$, denn sonst wäre $f(r) = 0 = f'(r)$, also r eine mehrfache Nullstelle. Sei r eine relle Zahl, sei $w(r)$ die Zahl der Indizes i mit $\mathrm{sgn}(f_i(r)) \neq \mathrm{sgn}(f_{i+1}(r))$, dies ist die Zahl der Vorzeichenwechsel in der Sturmschen Kette $f(r), f_1(r), \dots, f_m(r)$.

Satz 5.6 *Die Zahl der im Intervall (a, b) gelegenen Nullstellen von $f(x)$ ist gleich $w(a) - w(b)$.*

Beweis: Wenn r die x-Achse entlangläuft, so kann sich $w(r)$ nur dann ändern, wenn für ein i galt $f_i(r) = 0$. Dann ist aber

$$f_{i-1}(r) = -f_{i+1}(r).$$

Schauen wir uns die Werte von f_{i-1}, f_i und f_{i+1} in der Nähe von r an:

	f_{i-1}	f_i	f_{i+1}
$r - \epsilon$	$+t$	p	$-t$
r	$+t$	0	$-t$
$r + \epsilon$	$+t$	$-p$	$-t$

Dabei sei $p, t \in \{\pm 1\}$; egal, welchen Wert p hat, p oder $-p$ stimmt mit einem ihrer Nachbarn überein. Beim Durchgang durch r findet also hier gar keine Änderung der Wechselzahl statt. Eine Änderung der Wechselzahl sich kann also nur beim Durchgang durch eine Nullstelle von f ereignen, und oben haben wir gesehen, das dort ein Vorzeichenwechsel zwischen f und f' verlorengeht.
Schauen wir uns das in einem Beispiel an:

$$\begin{aligned} f(x) &= x^5 - 4x - 2 \\ f_1(x) &= 5x^4 - 4 \\ x^5 - 4x - 2 &= (5x^4 - 4) \cdot x/5 - 16/5x + 2, \\ \text{wir setzen } f_2(x) &= 8x + 5 \\ f_3(x) &> 0 \end{aligned}$$

x	f	f_1	f_2	f_3	$w(x)$
-2	-	+	-	+	3
-1	+	+	-	+	2
0	-	-	+	+	1
1	-	+	+	+	1
2	+	+	+	+	0

Also liegen zwischen -2 und -1, zwischen -1 und 0 sowie zwischen 1 und 2 je eine Nullstelle.

Die folgende Konstruktion erlaubt es festzustellen, ob zwei Polynome gemeinsame Nullstellen besitzen.

Resultante und Diskriminante

Seien zwei Polynome

$$f(x) = a_0 x^m + a_1 x^{m-1} + \ldots + a_m$$
$$g(x) = b_0 x^n + b_1 x^{n-1} + \ldots + b_n$$

gegeben.

Satz 5.7 *Die Polynome $f(x)$ und $g(x)$ haben genau dann einen nicht konstanten größten gemeinsamen Teiler, wenn es von Null verschiedene Polynome*

$$h(x) = c_0 x^{m-1} + c_1 x^{m-2} + \ldots + c_{m-1}$$

$$k(x) = d_0 x^{n-1} + d_1 x^{n-2} + \ldots + d_{n-1}$$

so daß

$$k(x) \cdot f(x) = h(x) \cdot g(x).$$

(Es ist $\deg(h) < \deg(f)$ *und* $\deg(k) < \deg(g)$*).*

Beweis: Sei $k \cdot f = h \cdot g$, dann können nicht alle Teiler von f in h aufgehen, da der Grad von h zu klein ist, also muß f einen gemeinsamen Teiler mit g besitzen.

Sei umgekehrt $t(x)$ ein gemeinsamer Teiler von $f(x)$ und $g(x)$. Dann gibt es Polynome $h(x)$ und $k(x)$ mit $f = t \cdot h, g = t \cdot k$, also

$$k \cdot f = k \cdot t \cdot h = g \cdot h.$$

□

Wenn wir die Gleichung $k \cdot f = h \cdot g$ ausmultiplizieren und die Koeffizienten gleicher Potenzen von x vergleichen, erhalten wir

$$\begin{aligned} d_0a_0 &= c_0b_0 \\ d_0a_1 + d_1a_0 &= c_0b_1 + c_1b_0 \\ d_0a_2 + d_1a_1 + d_2a_0 &= c_0b_2 + c_1b_1 + c_2b_0 \\ &\dots \\ d_{n-2}a_m + d_{n-1}a_{m-1} &= c_{m-2}b_n + c_{m-1}b_{n-1} \\ d_{n-1}a_m &= c_{m-1}b_n \end{aligned}$$

Wir fassen dies als lineares Gleichungssystem für die Unbekannten d_i und $-c_i$ auf, eine von Null verschiedene Lösung existiert genau dann, wenn die Determinante der Koeffizientenmatrix verschwindet. Die Determinante der transponierten dieser Matrix wird als Resultante von $f(x)$ und $g(x)$ bezeichnet:

$$\mathrm{Res}(f,g) = \begin{vmatrix} a_0 & a_1 & \dots & a_m & & & & \\ & a_0 & a_1 & \dots & a_m & & & \\ & & \dots & & & & & \\ & & & & a_0 & a_1 & \dots & a_m \\ b_0 & b_1 & \dots & b_n & & & & \\ & b_0 & b_1 & \dots & b_n & & & \\ & & & \dots & & & & \\ & & & & b_0 & b_1 & \dots & b_n \end{vmatrix}$$

(Die Matrix enthält n a-Zeilen und m b-Zeilen.)

Folgerung 5.8 *Die Polynome $f(x)$ und $g(x)$ besitzen genau dann gemeinsame Nullstellen, wenn $Res(f,g) = 0$ ist.*

Beispiel: $f(x) = x^2 - 1,\ g(x) = x^2 + 2x + 1$

$$\begin{vmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \\ 1 & 2 & 1 & 0 \\ 0 & 1 & 2 & 1 \end{vmatrix} = 0$$

Sei $f(x) = \prod(x - x_i),\ g(x) = \prod(x - y_j)$ normierte Polynome, dann sind die Koeffizienten von f bzw. von g die elementarsymmetrischen Funktionen der Nullstellen x_i bzw. y_j, also ist auch die Resultante ein Polynom in den x_i und y_j. Da die Resultante stets

verschwindet, wenn ein x_i gleich einem y_j ist, ist sie durch alle Differenzen $(x_i - y_j)$ teilbar, und wenn wir uns Grad und höchsten Koeffizienten genauer anschauen, finden wir

$$\mathrm{Res}(f,g) = \prod (x_i - y_j).$$

Die Resultante von f und f' wird als Diskriminante von f bezeichnet, sie verschwindet genau dann, wenn f mehrfache Nullstellen besitzt.

Rationale Funktionen in einer Variablen und Potenzreihen

Sei $r(x) = z(x)/n(x)$ eine rationale Funktion. Mittels der Taylor-Entwicklung kann man sie in eine Potenzreihe umformen:

$$r(x) = r(0) + r'(0) \cdot x + r''(0) \cdot x^2/2 + \ldots + r^{(n)}(0) \cdot x^n/n! + \ldots$$

Es kann also sein, daß eine Potenzreihe (die wir nicht im Computer speichern können, weil sie nichts Endliches ist) eigentlich eine rationale Funktion ist, die wir sehr wohl im Rechner behandeln können. Wie kann man nun feststellen, ob dies der Fall ist? Also

$$\sum_{i=1}^{\infty} a_i x^i = \frac{\sum_{i=1}^{m} b_i x^i}{\sum_{i=1}^{n} c_i x^i}$$

Wenn wir den Nenner hochmultiplizieren und wieder einen Koeffizientenvergleich machen, erhalten wir

$$b_k = \sum_{i+j=k} a_i c_j,$$

und dies ist für $k > m$ gleich Null. Wir schreiben das ein bißchen anders:

$$a_k = h_1 a_{k-1} + \ldots + h_n a_{k-n} \qquad \text{für } k > m.$$

Die Koeffizienten einer rationalen Potenzreihe genügen also einer Rekursionsformel.

Wir kommen nun zum interessanten Begriff der Padé-Approximation einer Potenzreihe $f(x)$.
In der Literatur hat sich eine altertümliche Bezeichnungsweise erhalten: L und M seien vorgegebene natürliche Zahlen, gesucht sind Polynome p_L, q_M mit $\deg(p_L) \leq L$, $\deg(q_M) \leq M$ und

$$f(x) - p_L(x)/q_M(x) = o(x^{L+M+1}),$$

weiterhin sei $q_M(x)$ normiert und p_L und q_M teilerfremd. Die rationale Funktion $p_L(x)/q_M(x)$ wird mit dem Symbol $[L/M]$ bezeichnet, dies ist die Padé-Approximation von $f(x)$.
Man kann versuchen, einen Ansatz mit unbestimmten Koeffizienten zu machen. Sei

$$p_L(x) = p_0 + p_1 x + \ldots + p_L x^L,$$

$$q_M(x) = 1 + q_1 x + \ldots + q_M x^M,$$

dann erhalten wir für die Koeffizienten das folgende inhomogene Gleichungssystem:

$$\begin{aligned}
a_0 &= p_0 \\
a_1 + a_0 q_1 &= p_1 \\
a_2 + a_1 q_1 + a_0 q_2 &= p_2 \\
&\dots \\
a_L + a_{L-1} q_1 + \dots + a_0 q_L &= p_L \\
a_{L+1} + a_L q_1 + \dots + a_0 q_{L+1} &= 0 \qquad (q_j = 0 \text{ für } j > M) \\
&\dots \\
a_{L+M} + a_{L+M-1} q_1 + \dots + a_L q_M &= 0
\end{aligned}$$

Dieses Gleichungssystem muß nicht lösbar sein, z.B. ist die [1/1]-Approximation von $1 + x^2 + \dots$ durch die Gleichungen

$$1 = p_0$$

$$q_1 = p_1$$

$$1 = 0$$

bestimmt. Falls aber eine $[L/M]$-Approximation existiert, so ist sie eindeutig bestimmt, wie man leicht sieht.

6 Algebraische Körpererweiterungen

Eine (komplexe) Zahl α heißt algebraische Zahl, wenn es ein Polynom $f(x) \in \mathbf{Q}[x]$ gibt, so daß $f(\alpha) = 0$ ist. Das normierte Polynom $m(x)$ minimalen Grades mit $m(\alpha) = 0$ heißt das Minimalpolynom von α.

Satz 6.1 *Das Minimalpolynom $m(x)$ einer algebraischen Zahl α ist irreduzibel, d.h. es gibt keine Polynome $f(X),\ g(x) \in \mathbf{Q}[x]$ mit $m(x) = f(x) \cdot g(x)$.*

Beweis: Wenn $m(x) = f(x) \cdot g(x)$ gilt, so gilt auch $m(\alpha) = f(\alpha) \cdot g(\alpha)$, also muß wegen $m(\alpha) = 0$ auch $f(\alpha) = 0$ oder $g(\alpha) = 0$ gelten. Dies widerspricht der Minimalität des Grades von $m(x)$.

Von nun an wollen wir mit griechischen Buchstaben stets algebraische Zahlen bezeichnen.
Der kleinste Körper, die sowohl alle rationalen Zahlen als auch die Zahl α enthält, besteht aus allen Brüchen der Form $\frac{f(\alpha)}{g(\alpha)}$ mit $g(\alpha) \neq 0$, er wird mit $\mathbf{Q}(\alpha)$ bezeichnet. Solch ein Körper wird als einfache Erweiterung des Körpers $\mathbf{Q}$ bezeichnet. Wir werden sehen, daß wir seine Elemente etwas einfacher darstellen können, so daß sie auch ein Computer versteht. Denken Sie an $\alpha = \sqrt{2}$.

Zunächst überlegen wir uns, daß es ein Polynom $u(x)$ gibt, so das $u(\alpha) = \dfrac{1}{g(\alpha)}$ gilt, d.h. jedes Element von $\mathbf{Q}(\alpha)$ ist ein Polynom in α, nicht ein Bruch zweier Polynome.

In der Tat: Da das Minimalpolynom $m(x)$ von α irreduzibel ist, gibt es nur zwei Möglichkeiten: Entweder ist $m(x)$ ein Teiler von $g(x)$, dann wäre aber $g(\alpha) = 0$, oder der größte gemeinsame Teiler von $g(x)$ und $m(x)$ ist gleich 1. Dann gibt es aber Polynome $u(x),\ v(x)$ mit $g(x) \cdot u(x) + m(x) \cdot v(x) = 1$, nun setzen wir $x = \alpha$ und sehen $g(\alpha) \cdot u(\alpha) = 1$, wie gewünscht.
Weiter: Der Grad von $m(x)$ sei gleich n. Dann gibt es für jedes Polynom $f(x)$ ein Polynom $r(x)$ von einem Grade, der kleiner als n ist, mit $f(\alpha) = r(\alpha)$. Wenn der Grad von $f(x)$ größer oder gleich n ist, so dividieren wir $f(x)$ durch $m(x)$:

$$f(x) = q(x) \cdot m(x) + r(x),\ \deg(r) < n.$$

Wir setzen wieder $x = \alpha$ und erhalten $f(\alpha) = r(\alpha)$.

Folgerung 6.2 $\mathbf{Q}(\alpha)$ *ist ein* $\mathbf{Q}$*-Vektorraum der Dimension* n*.*

Beweis: Die Menge $\{1, \alpha, \alpha^2, \ldots, \alpha^{n-1}\}$ bildet eine $\mathbf{Q}$-Basis von $\mathbf{Q}(\alpha)$.

Da das Minimalpolynom $m(x)$ irreduzibel ist, ist das von $m(x)$ erzeugte Ideal $(m(x))$ des Polynomrings $\mathbf{Q}[x]$ ein maximales Ideal, *al*so ist der Faktorring $\mathbf{Q}[x]/(m(x))$ ein Körper. Durch die Zuordnung

$$x^i \bmod m(x) \to \alpha^i$$

wird ein Isomorphismus $\mathbf{Q}[x]/(m(x)) \to \mathbf{Q}(\alpha)$ gegeben. Also können wir die Elemente $\beta \in \mathbf{Q}(\alpha)$ folgendermaßen im Rechner darstellen: $\beta = \sum b_i x^i$ ist ein Polynom vom Grade $\leq n - 1$, wir rechnen mit diesen Elementen modulo $m(x)$. Wenn der Grad eines Produkts zweier Polynome die Zahl $n - 1$ übersteigt, ersetzen wir es durch seine Restklasse mod $m(x)$. Wie wir oben gesehen haben, ist das Inverse eines Polynoms mit Hilfe des Euklidischen Algorithmus berechenbar.

Es wäre aber auch zu überlegen, ob man die zugegebenermaßen zeitaufwendige Inversenberechnung nicht so lange wie möglich vor sich herschieben sollte. Wir fügen einfach jedem Element von $\mathbf{Q}(\alpha)$ einen Nenner hinzu, der anfangs gleich 1 gesetzt wird. Nun wird wie mit gewöhnlichen Brüchen gerechnet, bei einer Division wird einfach mit dem reziproken Bruch multipliziert, und Zähler und Nenner werden stets modulo $m(x)$ reduziert. Erst, wenn eine Ausgabe notwendig ist, werden die Nenner bereinigt: der Zähler wird mit dem Inversen des Nenners multipliziert.
Damit haben wir die Arithmetik in endlichen algebraischen Erweiterungskörpern von $\mathbf{Q}$ im Griff.
Schwieriger wird es, wenn wir zwei beliebige algebraische Zahlen α, β addieren oder multiplizieren wollen. Beide Zahlen sind als Nullstellen ihrer jeweiligen Minimalpolynome zu verstehen, wir müßten also das Minimalpolynom von $\alpha + \beta$ bestimmen. Alle drei Zahlen liegen im Körper $\mathbf{Q}(\alpha, \beta)$, der folgende Satz sagt aus, daß dies ebenfalls eine einfache Erweiterung von $\mathbf{Q}$ ist.

Satz 6.3 (Satz vom primitiven Element) *Seien* α, β *algebraische Zahlen, dann gibt es eine algebraische Zahl* δ *mit* $\mathbf{Q}(\alpha, \beta) = \mathbf{Q}(\delta)$*, d.h. es gibt Polynome* $p(x, y),\ q(x),\ r(x)$ *mit* $\delta = p(\alpha, \beta),\ \alpha = q(\delta),\ \beta = r(\delta)$*.*

Beweis: Sei $f(x)$ das Minimalpolynom von α, seine Nullstellen seien $\alpha = \alpha_1, \alpha_2, \ldots, \alpha_n$. Sei $g(x)$ das Minimalpolynom von β, seine Nullstellen seien $\beta = \beta_1, \beta_2, \ldots, \beta_m$. Dann gibt es eine rationale Zahl c mit

$$\alpha_i + c\beta_k \neq \alpha_1 + c\beta_1 \text{ für alle } i, k,$$

denn jede der Gleichungen

$$\alpha_i + x\beta_k = \alpha_1 + x\beta_1$$

hat höchstens eine Lösung $x \in \mathbf{Q}$. Wir setzen $\delta = \alpha + c\beta \in \mathbf{Q}(\alpha, \beta)$, also ist $\mathbf{Q}(\delta) \subseteq \mathbf{Q}(\alpha, \beta)$.
Wir betrachten nun $g(x)$ und $f(\delta - cx)$ als Polynome in $\mathbf{Q}(\delta)[x]$. Es gilt $g(\beta) = 0$ und $f(\delta - c\beta) = f(\alpha) = 0$ sowie

$$\delta - c\beta_k = \alpha_1 + c\beta - c\beta_k \neq \alpha_i \text{ für alle } k,$$

also $f(\delta - c\beta_k) \neq 0$ für $k > 1$. Also ist β die einzige gemeinsame Nullstelle der Polynome $g(x)$ und $f(\delta - cx)$, demnach ist ihr größter gemeinsamer Teiler, der im Polynomring $\mathbf{Q}(\delta)[x]$ zu berechnen ist, gleich $x - \beta$. Das heißt aber, daß β im Koeffizientenkörper $\mathbf{Q}(\delta)$ liegt, damit ist auch $\alpha = \delta - c\beta \in \mathbf{Q}(\delta)$, also $\mathbf{Q}(\delta) = \mathbf{Q}(\alpha, \beta)$.

Beispiel: $\alpha = \sqrt{2},\ \beta = \sqrt{3},\ \delta = \sqrt{2} + \sqrt{3}$, das Minimalpolynom von δ ist $x^4 - 10x^2 + 1$, es gilt $\sqrt{2} = (\delta^3 - 9\delta)/2,\ \sqrt{3} = (11\delta - \delta^3)/2$.

Man kann auch direkt ein Polynom angeben, daß als Nullstelle die Zahl $\alpha + \beta$ besitzt: Wir wählen eine neue Unbestimmte y und betrachten das Polynom $f(y-x)$ als Polynom in x, die Resultante

$$r(y) = \operatorname{Res}(f(y - x), g(x))$$

verschwindet genau dann, wenn $f(y-x)$ und $g(x)$ eine gemeinsame Nullstelle besitzen, dies ist für $y = \alpha + \beta$ der Fall, die Nullstelle ist gleich $x = \beta$.
In unserem Beipiel sieht das folgendermaßen aus:

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

$$r(y) = \begin{vmatrix} 1 & -2y & y^2 - 2 & 0 \\ 0 & 1 & -2y & y^2 - 2 \\ 1 & 0 & -3 & 0 \\ 0 & 1 & 0 & -3 \end{vmatrix} = y^4 - 10y + 1.$$

Folgerung 6.4 *Die Menge der algebraischen Zahlen ist ein Körper.*

7 Gleichungen dritten und vierten Grades

Nun sollen explizite Formeln für die Nullstellen von Polynomen mit reellen Koeffizienten entwickelt werde, soweit dies möglich ist.

Sei

$$f(y) = y^3 + ay^2 + by + c$$

ein Polynom dritten Grades, dessen Nullstellen wir suchen. Wenn wir $y = x+m$ setzen, so hat der Koeffizient von x^2 in $f(y)$ den Wert $3m + a$, er verschwindet für $m = -\frac{a}{3}$. Somit können wir annehmen, daß die Gleichung in der Form

$$x^3 + px + q = 0$$

zu lösen ist. Wir leiten zunächst die nach Geronimo Cardano (1501-1576) benannten und von Niccolo Targaglia (1500-1557) gefundenen Cardanische Formel her.

Wir machen den Ansatz $x = u + v$ und erhalten durch Einsetzen von x^3 in die obige Gleichung

$$u^3 + v^3 + 3uv(u + v) + p(u + v) + q = 0.$$

Wir suchen solche Werte u, v, daß

$$u^3 + v^3 + q = 0$$

und

$$3uv + p = 0$$

gilt, dann wäre die Gleichung gelöst. Dies ist der Fall, wenn

$$u^3 + v^3 = -q$$

und

$$u^3v^3 = -\frac{p^3}{27}$$

gelten. Nach der Formel von Viètá heißt das, daß die Werte u^3 und v^3 die Lösungen der Gleichung

$$z^2 + qz - \frac{p^3}{27} = 0$$

sind. Für quadratische Gleichungen kennen wir schon eine Lösungsformel, also gilt

$$u^3 = -\frac{q}{2} + \sqrt{\frac{q^2}{4} + \frac{p^3}{27}}$$

$$v^3 = -\frac{q}{2} - \sqrt{\frac{q^2}{4} + \frac{p^3}{27}}$$

Sei $\epsilon \in \mathbf{C}$ eine primitive dritte Einheitswurzel, dann haben u^3 und v^3 drei verschiedene dritte Wurzeln, die sich jeweils um einen Faktor ϵ unterscheiden, dies ergäbe neun mögliche Werte für x. Beachten wir aber (s.o.), daß uv reell sein soll, so erhalten wir als folgende drei Lösungen der gegebenen Gleichung:

$$x_1 = \sqrt[3]{-\frac{q}{2} + \sqrt{\frac{q^2}{4} + \frac{p^3}{27}}} + \sqrt[3]{-\frac{q}{2} - \sqrt{\frac{q^2}{4} + \frac{p^3}{27}}}$$

$$x_2 = \epsilon \sqrt[3]{-\frac{q}{2} + \sqrt{\frac{q^2}{4} + \frac{p^3}{27}}} + \epsilon^2 \sqrt[3]{-\frac{q}{2} - \sqrt{\frac{q^2}{4} + \frac{p^3}{27}}}$$

$$x_3 = \epsilon^2 \sqrt[3]{-\frac{q}{2} + \sqrt{\frac{q^2}{4} + \frac{p^3}{27}}} + \epsilon \sqrt[3]{-\frac{q}{2} - \sqrt{\frac{q^2}{4} + \frac{p^3}{27}}}$$

Dies also sind die Cardanischen Formeln. Sie haben allerdings ihre Tücken, wie den Mathematikern des 16. Jahrhunderts schmerzlich bewußt wurde. Betrachten wir ein Beispiel:

$$x^3 - 981x - 11340 = 0,$$

wir erhalten

$$x_1 = \sqrt[3]{-5670 + \sqrt{32148900 - 34965783}} + \dots$$

hier wären also dritte Wurzeln aus (echt) komplexen Zahlen zu ziehen. Die „alten Herren“ haben vielleicht weniger die imaginären Terme gestört, seit jeher haben Mathematiker mit Dingen operiert, die sie nicht verstanden haben. Aber: die drei Lösungen dieser Gleichung sind reell, wie wir noch sehen werden, sie lassen sich aber nicht durch „normale“ Rechenoperationen (Addition, Multiplikation, Wurzelziehen) aus den Koeffizienten der Gleichung berechnen. Eine dritte Wurzel aus einer beliebigen komplexen Zahl durch „Radikale“ darstellen zu können, würde bedeuten, daß man einen Winkel mit Zirkel und Lineal in drei gleiche Teile teilen könnte; die Galois-Theorie lehrt, daß dies unmöglich ist. Um diesem Problem wenigstens einen Namen zu geben, nannte man diesen Fall „casus irreducibilis“.
Mit algebraischen Methoden ist dieses Problem prinzipiell nicht lösbar, hier zeigen sich auch Grenzen der Computeralgebra.
Jedoch hat Viètá um 1600 eine „transzendente“ Lösung gefunden: Wir haben

$$x^3 + px + q = 0$$

und die Zahl p ist notwendigerweise negativ. Sei

$$u^3 = -\frac{q}{2} + i\sqrt{-\frac{q^2}{4} - \frac{p^3}{27}} = r(\cos\alpha + i\sin\alpha),$$

dann errechnet man $r = \sqrt{-\frac{p^3}{27}}$ und $\cos\alpha = -\frac{q}{2r}$ also

$$\begin{aligned} x_1 &= r^{1/3}\,(\cos\frac{\alpha}{3} + i\sin\frac{\alpha}{3}) + r^{1/3}\,(\cos\frac{\alpha}{3} - i\sin\frac{\alpha}{3}) \\ &= 2\sqrt{-\frac{p}{3}}\,\cos\frac{\alpha}{3} \\ x_2 &= 2\sqrt{\frac{-p}{3}}\,\cos\frac{\alpha-\pi}{3} \\ x_3 &= 2\sqrt{\frac{-p}{3}}\,\cos\frac{\alpha+\pi}{3} \end{aligned}$$

Im oben angeführten Beispiel gilt

$$r \approx 5913,1872$$

$$\cos\alpha \approx 0,9588$$
$$\alpha \approx 16,5$$
$$\cos(5,5^\circ) \approx 0,9954$$
$$x_1 \approx 35,99$$
$$x_2 \approx -21$$
$$x_3 \approx -15$$

Wer nachrechnen möchte, wird sehen, daß die Lösungen ganzzahlig sind, sich aber nicht „einfacher“ aus den Koeffizienten berechnen lassen.

Als nächstes betrachten wir Gleichungen 4. Grades, diese möge bereits in die Form

$$x^4 + px^2 + qx + r = 0$$

gebracht worden sein.
Wir machen wieder einen Ansatz $x = u+v+w$ und führen folgende Rechnungen durch:

$$x^2 = u^2 + v^2 + w^2 + 2(uv + uw + vw),$$
$$x^2 - u^2 + v^2 + w^2 = 2(uv + uw + vw),$$
$$x^4 - 2x^2(u^2 + v^2 + w^2) + (u^2 + v^2 + w^2)^2$$
$$= 4(u^2v^2 + u^2w^2 + v^2w^2) + 8(u^2vw + v^2uw + w^2uv)^2$$
$$= 4(u^2v^2 + u^2w^2 + v^2w^2) + 8\text{uvwx},$$

nun setzen wir x^4 in die obige Gleichung ein:
$2x^2(u^2+v^2+w^2) - (u^2+v^2+w^2)^2 + 4(u^2v^2+u^2w^2+v^2w^2) + 8(u^2vw+v^2uw+w^2uv)^2 + 4(u^2v^2 + u^2w^2 + v^2w^2) + 8uvwx + px^2 + qx + r = 0.$
Nun wählen wir u, v, w so, daß die Koeffizienten von x^2, x und 1 Null werden:

$$u^2 + v^2 + w^2 = -\frac{p}{2},$$
$$8uvw = -q,$$

daraus folgt

$$u^2v^2w^2 = \frac{q^2}{64},$$

und

$$u^2v^2 + u^2w^2 + v^2w^2 = \frac{p^2}{16} - \frac{r}{4},$$

d.h. die Zahlen $u^2,\ v^2,\ w^2$ sind die Nullstellen des Polynoms

$$x^3 + \frac{p}{2}x^2 + (\frac{p^2}{64} - \frac{r}{4})x - \frac{q^2}{64}.$$

Diese seien gleich $z_1,\ z_2,\ z_3$, dann erhalten wir

$$u = \pm\sqrt{z_1},\ v = \pm\sqrt{x_2}, w = \pm\sqrt{x_3},$$

wegen $8uvw = -q$ legen die Vorzeichen von u und v bereits das von w fest, es gibt also vier Lösungen.

Man könnte versuchen, nach demselben Verfahren Gleichungen fünften Grades zu lösen, ich habe es nicht probiert, weil ich weiß, das es nicht geht, dies ist ein Resultat der Galois-Theorie. Hier bei dem konkreten Ansatz $x = t+u+v+w$ wird man wahrscheinlich wieder auf eine Gleichung für Terme, die aus t, u, v, w gebildet werden, geführt, und diese wird mindestens den Grad 5 haben.

8 Matrizen

Matrizen sind „zweidimensionale Gebilde“. Wenn wir in einer höheren Programmiersprache einen Typ
`matrix = array[1..m, 1..n] of number`
erklären, so werden bei einer Belegung einer Variablen dieses Typs die einzelnen Einträge (hoffentlich) zeilenweise in den (eindimensionalen) Speicher geschrieben. Dabei muß der Compiler wissen, wieviel Speicherplatz er für eine Variable vom Typ `matrix` bereitzustellen hat. Deshalb müssen die Zeilen- und Spaltenanzahlen (m und n) Konstanten sein, die zur Compilierungszeit feststehen; man kann also eine Matrix nicht bei Bedarf vergrößern.
Dies ist lästig. Stellen Sie sich vor, Sie wollen mit demselben Programm Übungsaufgaben mit 2×2-Matrizen, aber auch ein aufwendiges Problem, wo 20-reihige Matrizen vorkommen, bearbeiten. Dann müßte man sich für maximal mögliche Zeilenzahl festlegen, bei „kleinen“ Anwendungen würde, wie immer, Speicherplatz verschenkt.
In unserem Modell besteht eine rationale Zahl aus zwei Pointern, dies nimmt 8 Byte ein, eine 20×20-Matrix belegt also bereits 3 Kilobyte, ohne daß irgendetwas in ihr steht; der Platzbedarf des Inhalts kommt schließlich hinzu. Wenn Sie nun 100 Matrizen verwalten wollen, wird es schon eng.
Wie schon bei der Darstellung langer ganzer Zahlen könnte man versuchen, dynamische Arrays zu verwenden.
Bei einer $m \times m$-Matrix a steht der Inhalt von $a[i, j]$ an der Speicheradresse
(Startadresse von a) $* (i - 1) * n + j$,
wenn wir (provisorisch) als Matrix-Typ
`matrix = pointer to array[1..1] of number`
vereinbaren würden, so konnten wir mit
`getmem(a, m*n)`
genau den benötigten Speicherplatz reservieren und auf a_{ij} durch $a[(i - 1) * n + j]$ zugreifen. Der wie üblich faule Programmierer schreibt hierfür zwei Prozeduren:

```
procedure getmat(a: matrix; i, j: integer; var z: number);
begin z:= a[(i-1)*n + j] end;

procedure putmat(var a: matrix; i, j: integer; z: number);
begin a[(i-1)*n + j]:= z end;
```

denn eine Anweisung `putmat(a,i,j,zehn);` ist übersichtlicher als `a[(i-1)*n + j]:= zehn;`

Dieser Versuchung zu widerstehen ist schwer, bedenken Sie jedoch, daß ein Prozeduraufruf etwas Zeit kostet und daß ein guter Compiler bei den Anweisungen `z:= a[(i-1)*n + j]` und `z:= a[i,j]` im zweiten Fall wahrscheinlich effektiveren Code erzeugt.
Es folgt ein Vorschlag, wie man mit dem Speicherplatz sparsam, wenn auch nicht knausrig, umgehen und dennoch die Übersicht über die Matrixelemente behalten kann:

```
type zeile = array[1..matsize] of number;
zeilp = pointer to zeile;
matrix = array[1..1] of zeilp;
```

Wenn man nun eine $m \times n$-Matrix a einrichten will, so wären folgende Anweisungen nötig:

```
for i:= 1 to m do
    getmem(a[i]^, sizeof(zeile));
```

und auf a_{ij} greift man über `a[i]^[j]` zu. Das erfordert etwas mehr Schreibarbeit, aber man sieht doch, wohin man greift.
Natürlich kann man auch noch den Zeilen-Typ dynamisieren, dann ist noch ein Zirkumflex (`^`) mehr zu tippen.

Wie die elementaren Rechenoperationen mit Matrizen zu implementieren sind, braucht an dieser Stelle sicher nicht erläutert zu werden. Wir kommen also zu den weniger elementaren Operationen.

Zuerst wollen wir uns ein Verfahren ansehen, mit dem man zwei 2×2-Matrizen nicht mit 8, sondern mit 7 Multiplikationen berechnen kann. Dieses Verfahren von Strassen (1969) gestattet es, durch Zerteilung von $n \times n$-Matrizen in Blöcke der halben Größe die für die Multiplikation notwendige Zeit von n^3 auf $n^{2.7}$ zu verringern. Es sei aber bemerkt, daß dieses Verfahren von keinem der gängigen Computeralgebrasysteme genutzt wird.

Wir betrachten das Produkt zweier 2×2-Matrizen:

$$\begin{pmatrix} a_1 & a_2 \\ a_2 & a_4 \end{pmatrix} \begin{pmatrix} b_1 & b_2 \\ b_3 & b_4 \end{pmatrix} = \begin{pmatrix} c_1 & c_2 \\ c_3 & c_4 \end{pmatrix}$$

Wir fassen a und b als Elemente des R^4 auf, dann sind die c_i Bilinearformen auf R^4, zu denen folgende Darstellungsmatrizen gehören:

$$c_1 : \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, \; c_2 : \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, \; c_3 : \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}, \; c_4 : \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Zu einer Bilinearform, die das Produkt zweier Linearformen $r_1a_1 + r_2a_2 + r_3a_4 + r_4a_4$ und $s_1b_1 + s_2b_2 + s_3b_3 + s_4 - b_4$ ist, gehört die Matrix

$$\begin{pmatrix} r_1s_1 & r_2s_1 & r_3s_1 & r_4s_1 \\ r_1s_2 & r_2s_2 & r_3s_2 & r_4s_2 \\ r_1s_3 & r_2s_3 & r_3s_3 & r_4s_3 \\ r_1s_4 & r_2s_4 & r_3s_4 & r_4s_4 \end{pmatrix},$$

diese hat den Rang 1. Das Problem besteht also darin, die obigen c-Matrizen als Summen von Matrizen vom Rang 1 darzustellen. Dies gelingt mit folgenden Matrizen:

$$m_1 = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & -1 \end{pmatrix},\ m_2 = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix},\ m_3 = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ -1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix},$$

$$m_4 = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix},\ m_5 = \begin{pmatrix} 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix},\ m_6 = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 \end{pmatrix},$$

$$m_7 = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

Es gilt $m_1 = (a_2 - a_4)(b_3 + b_4),\ m_2 = (a_1 + a_4)(b_1 + b_4),\ m_3 = (a_1 - a_3)(b_1 + b_2),\ m_4 = (a_1 + a_2)b_4,\ m_5 = a_1(b_2 - b_4),\ m_6 = a_4(b_3 - b_1),\ m_7 = (a_3 + a_4)b_1$.
Schließlich gilt

$$c_1 = m_1 + m_2 - m_4 + m_6$$

$$c_2 = m_4 + m_5$$

$$c_3 = m_6 + m_7$$

$$c_4 = m_2 - m_3 + m_5 - m_7$$

Die gesparte Multiplikation ist durch 18 (statt 4) Additionen erkauft worden.

Strassens Publikation von 1969 ist zwei Druckseiten lang, sie enthält keinen Hinweis, wie er diese Resultate erhalten hat. Es hat sich in den Folgejahren ein Wettlauf entwickelt, wie schnell man (theoretisch) die Matrixmultiplikation ausführen könnte, der Weltrekord lag in den 80er Jahren bei $n^{2.49}$.

Im ersten Semester haben Sie gelernt, daß man eine Matrix, die zu einem linearen homogenen Gleichungssystem $Ax = 0$ gehört, mit Hilfe von Zeilenoperationen (Gaußscher Algorithmus) in eine Gestalt wie die folgende bringen können:

$$\begin{bmatrix} 0 & \dots & 1 & a_{1,k_1+1} & \dots & a_{1,k_2-1} & 0 & a_{1,k_2+1} & \dots & a_{1,k_r-1} & 0 & & \dots & b_1 \\ 0 & \dots & 0 & & \dots & 0 & 1 & a_{2,k_2+1} & \dots & a_{2,k_r-1} & 0 & a_{2,k_r+1} & \dots & b_2 \\ & \dots & & & & & & & & & & & & \\ 0 & \dots & 0 & & \dots & & & & & & 1 & a_{r,k_r+1} & \dots & b_r \\ & \dots & & & & & & & & & & & & \\ 0 & & & & & & \dots & & & & & & 0 & b_{r+1} \\ & \dots & & & & & & & & & & & & \\ 0 & & & & & & \dots & & & & & & 0 & b_m \end{bmatrix}$$

Es ist nicht schwierig, den Gaußschen Algorithmus zu implementieren. Tun Sie es (vertauschen Sie keine Spalten, merken Sie sich die Nummer der i-ten ausgezeichneten

Spalte in $k[i]$)! Dabei können Sie solche Tricks wie Pivotisierung und Equikalibrierung, die Sie in der numerischen Mathematik kennengelernt haben, beiseitelassen, bei exakter Arithmetik gibt es keine numerische Instabiltät.

```
procedure GAUSS(var a:nummatrix; var l:zeil; var rk: integer; m,n:byte);
          (* Auf die Matrix a wird der Gausssche Algorithmus angewandt,
            es werden keine Spalten vertauscht.
            Die in der 'Zeile' l ausgegebenen, von 0 verschiedenen Zahlen
            sind die Numern der ausgezeichneten Spalten in der reduzierten
            Form von a; rk = Rang(a). *)
label exitr, exit0;
var i,j,k,p:integer; h,x,y:Zahl;
begin
l[0]:=0;
rk:=0;
for k:=1 to m do
begin
  if l[k-1]=n then goto exitr;
  for j:=l[k-1]+1 to n do
    for i:=k to m do
      if not zero(a[i]^[j]) then goto exit0
      else if (j=n) and (i=m) then goto exitr;
  exit0: l[k]:=j;
  inc(rk);
  if i>k then
    for p:=j to n do
      xchg(a[i]^[p],a[k]^[p]);
  nlCopy(a[k]^[j], x);
  kurz(x);
  nlDelete(a[k]^[j]);
  assig(1,a[k]^[j]);
  for p:=j+1 to n do
  begin
    nlDiv(a[k]^[p],x, h);
    kurz(h);
    nlDelete(a[k]^[p]);
    a[k]^[p]:= h;
  end;
  nlDelete(x);
  for i:=1 to m do
    if i<>k then
      if not zero(a[i]^[j]) then
      begin
        nlCopy(a[i]^[j], x);
        kurz(x);
        altsign(x);
```

```
        nlDelete(a[i]^[j]);
        assig(0,a[i]^[j]);
        for p:=j+1 to n do
        begin
          nlMult(a[k]^[p],x,y);
          plus(y,a[i]^[p], h);
          nlDelete(y);
          nlDelete(a[i]^[p]);
          a[i]^[p]:= h;
        end;
        nlDelete(x);
      end;
end;
exitr:
for i:=rk+1 to matsize do l[i]:= 0;
l[0]:= n;
end;
```

Die oben angegebene Form nennen wir die reduzierte Form der Matrix A. Die folgende Prozedur berechnet eine Matrix, deren Spalten die eine Basis des Lösungsraums des Gleichungssystems $Ax = 0$ bilden:

```
procedure nullraum(var ein, aus: nummatrix; l: zeil; m,n,r: integer);
          (* Die Matrix ein soll in reduzierter Form vorliegen, in l sollen
            die Nummern der ausgezeichneten Spalten dieser Matrix stehen,
            r = Rang(ein)  (alles wie bei der Ausgabe von Gauss).
            aus ist eine n x (n-r) - Matrix, deren Spalten eine Basis des
            Loesungsraums des zu ein gehoerigen homogenen Gleichungssystems
            bilden. *)
var i,j,k,kk: integer;
begin
  newmat(aus,n,n-r);
  k:=1;
  kk:=1;
  for j:=1 to n do
    if (k<=r) and (j=l[k]) then inc(k)
    else
    begin
      assig(-1, aus[j]^[kk]);
      for i:=1 to r do
        nlCopy(ein[i]^[j], aus[l[i]]^[kk]);
      inc(kk);
    end;
end;

procedure loesung(var ein, aus: nummatrix; l: zeil; m,n,r: integer;
```

```
          var ok: boolean);
          (* Die Matrix ein soll in reduzierter Form vorliegen, in l sollen
            die Nummern der ausgezeichneten Spalten dieser Matrix stehen,
            r = Rang(ein)  (alles wie bei der Ausgabe von Gauss).
            aus ist eine (n-1) x (n-r) - Matrix, deren erste n-r-1 Spalten
            eine Basis des Loesungsraums des zu ein gehoerigen homogenen
            Gleichungssystems bilden, die letzte Spalte ist eine spezielle
            Loesung des inhomogenen Systems.
            ok: Gleichungssystem ist lsbar *)
var i,j,k,kk: integer; mz: zahl;
begin
  i:= 1;
  while (i<n) and zero(ein[r]^[i]) do inc(i);
  ok:= i<n;
  if not ok then exit;
  assig(1,mz);
  newmat(aus,n-1,n-r);
  k:=1;
  kk:=1;
  for j:=1 to n-1 do
    if (k<=r) and (j=l[k]) then inc(k)
    else
    begin
      assig(1, aus[j]^[kk]);
      for i:=1 to r do
      begin
        nlCopy(ein[i]^[j], aus[l[i]]^[kk]);
        altsign(aus[l[i]]^[kk]);
      end;
      inc(kk);
    end;
  for i:= 1 to r do
    nlCopy(ein[i]^[n], aus[l[i]]^[n-r]);
end;
```

Wenn wir die Prozedur Gauss, die die Matrix in ihre reduzierte Form überführt, durch eine ähnliche ersetzen, die ohne Divisionen auskommt, so löst die angegebene Prozedur auch lineare Gleichungen, deren Koeffizienten in einem Polynomring $\mathbf{Q}[x, y, z, \ldots]$ liegen.

Nun können wir mit den Unterräumen des Vektorraums $\mathbf{R}^n$ operieren: Wir beschreiben einen Unterraum $U \subset \mathbf{R}^n$ durch eine Matrix M_U, deren Spalten ein Erzeugendensystem von U bilden. Der Gaußsche Algorithmus kann genutzt werden, um linear abhängige Elemente zu entfernen; wir können also voraussetzen, daß die Spalten von M_U eine BAsis von U bilden.

Zur Summe $U + V$ zweier Unterräume $U, V \subset \mathbf{R}^n$ gehört dann die Matrix (M_U, M_V), die durch Hintereinanderschreiben beider Matrizen entsteht.

Wir überlegen, wie man aus zwei Basen $\{x_1, \ldots, x_k\}$ von U und $\{y_1, \ldots, y_l\}$ von V eine Basis des Durchschnitts $U \cap V$ bestimmen kann:

Die Vektoren seien so numeriert, daß

$$x, \ldots, x_k, y_1, \ldots, y_s$$

eine Basis von $U + V$ bilden. Die restlichen Vektoren $y_{s+1}, \ldots, y_l$ haben dann eine Darstellung

$$y_i = a_{i1}x_1 + \ldots + a_{ik}x_k + b_{i1}y_1 + \ldots + b_{is}y_s,\ i = s+1, \ldots l.$$

Dann sind die $l - s$ Vektoren

$$z_{i-s} = -b_{i1}y_1 - \ldots - b_{is}y_s + y_i$$

linear unabhängig und liegen in V, aßerdem ist

$$z_{i-s} = a_{i1}x_1 + \ldots + a_{ik}x_k,$$

also liegen sie auch in V und da sie die richtige Anzahl haben, bilden sie eine Basis von $U \cap V$.

Eine wichtige Operation mit Matrizen ist die Determinantenberechnung. Hierzu eignet sich, falls die Komponenten in einem Körper liegen, ebenfalls am Besten eine Variante des Gaußschen Algorithmus. Dieses Verfahren ist nicht anwendbar, wenn Divisionen nicht gestattet sind. Bleibt dann nur der Laplacesche Entwicklungssatz mit seinen $n!$ Rechenoperationen?
Wir werden bald ein originelles Determinantenberechnungsverfahren kennenlernen, das mit n^4 Rechenoperationen und fast ohne Divisionen auskommt.

Wir wollen uns dazu genauer mit dem Zusammenhang zwischen den Koeffizienten und den Nullstellen eines Polynoms befassen. Sei

$$f(z) = z^n + b_1 z^{n-1} + \ldots + b_{n-1}z + b_n = (z - z_1) \ldots (z - z_n)$$

ein Polynom mit den Koeffizienten b_i und den Nullstellen z_i. Wir wissen, daß $f(z)$ durch die Linearfaktoren $z - z_i$ teilbar ist. Wie sehen die Koeffizienten von $f(z)/(z - z_i)$ aus?

Satz 8.1

$$\frac{\sum_{i=0}^{n} b_{n-i}z^i}{z - z_1} = \sum_{i=0}^{n-1} z^{n-1-i} \sum_{j=0}^{i} b_j z_1^{i-j}.$$

Den Beweis wollen wir weglassen, multiplizieren Sie bitte selbst $\sum\ z^{n-1-i} \sum b_j z_1^{i-j}$ und $z - z_1$ miteinander und stellen Sie fest, ob Sie $f(z)$ herausbekommen. □

Abkürzung: Sei $f(z)$ ein Polynom vom Grade n mit den Nullstellen $z_1, \ldots, z_n$. Wir setzen

$$s_0 = n,$$

$$s_1 = z_1 + \ldots + z_n$$
$$s_2 = z_1^2 + \ldots + z_n^2,$$
$$\ldots$$
$$s_i = z_1^i + \ldots + z_n^i.$$

Die Zahl s_i heißt die i-te Potenzsumme der x_j.

Wir stellen nun eine Beziehung zwischen den Potenzsummen der Wurzeln und den Koeffizienten des Polynoms auf, dies sind die sogenannten Newtonschen Formeln.

Satz 8.2

$$b_i = -\frac{1}{i}\sum_{j=0}^{i-1} b_j s_{i-j}$$

Beweis: Es ist $f(z) = \prod(z - z_i)$. Wir betrachten die Ableitung $f'(z)$ von $f(z)$:

$$f'(z) = \sum_k \prod_{i\neq k}(z - z_i) = \sum f(z)/(z - z_k)$$

$$= \sum_{k=1}^{n}\sum_{i=0}^{n-1} z^{n-i-1}\sum_{j=0}^{i} b_j z_k^{i-j}$$

$$= \sum_{i=0}^{n-1}\sum_{j=0}^{i} z^{n-i-1} b_j \sum_k z_k^{i-j}$$

$$= \sum_{i=0}^{n-1}\sum_{j=0}^{i} z^{n-i-1} b_j s_{i-j}.$$

Andererseits gilt

$$f'(z) = \sum z^{n-i-1}(n-i)b_i$$

und durch Koeffizientenvergleich erhalten wir

$$(n-i)b_i = \sum_{j=0}^{i} b_j s_{i-1} = \sum_{j=0}^{i-1} b_j s_{i-j} + n b_i,$$

und daraus ergibt sich die Behauptung. □

Wir kehren nun doch nach diesem Seitensprung wieder zu unseren lieben Matrizen zurück. Aber wir wenden das Gelernte an:

Lemma 8.3 *Seien $z_1, \ldots z_n$ die Eigenwerte der Matrix A, dann ist $s_i = Sp(A^i)$.*

Beweis: A^i hat die Eigenwerte $z_1^i, \ldots, z_n^i$ und die Spur einer Matrix ist die Summe ihrer Eigenwerte. □

Nun können wir die Newtonschen Formeln verwenden, um die Koeffizienten des charakteristischen Polynoms einer Matrix zu bestimmen, ohne irgendwelche Determinanten ausrechnen zu müssen.

Folgerung 8.4 *Sei A eine Matrix und $c_A(z) = \sum b_i z^{n-i}$ ihr charakteristisches Polynom. Dann ist*

$$b_i = -1/i \sum_{j=0}^{i-1} b_j Sp(A^{i-j}). \quad \square$$

Umgekehrt wissen wir, daß $(-1)^n b_n$, der konstante Term des charakteristischen Polynoms der Matrix A, gleich der Determinante von A ist. Zu ihrer Berechnung brauchen wir also nur die Spuren der Matrizen $A, A^2, \ldots A^n$ zu berechnen und die obige Formel anzuwenden.
Der Leser möge sich aber keinen Illusionen hingeben, das Problem liegt hier weniger in der Rechenzeit, sondern im Speicherbedarf. Wenn zum Beispiel A eine „generische“ Matrix ist, deren n^2 Einträge unabhängige Variable sind, so hat schon für $n = 6$ der Term $Sp(A^6)$ so viele Summanden, daß er mehr als 1/2 Megabyte Speicher verbraucht.

Der Algorithmus von Bareiss

Ganzzahlige Matrizen können ohne Erzeugung von Brüchen in eine Dreiecksform überführt werden. Seien

$$m_{ij}^{(0)} = m_{ij}$$

die Komponenten der Eingabematrix, wir setzen

$$m_{00}^{(-1)} = 1$$

und für $k < i, j \leq n$ setzen wir

$$m_{ij}^{(k)} = \frac{1}{m_{k-1,k-1}^{k-2}} \cdot (m_{kk}^{(k-1)} \cdot m_{ij}^{k-1} - m_{ik}^{(k-1)} \cdot m_{kj}^{(k-1)}),$$

dies ist eine ganze Zahl. (Beim Gaußschen Algorithmus steht $m_{ij} - m_{ik} \cdot m_{kj}/m_{kk}$ auf der echten Seite.)

Wir weisen die Teilbarkeit nach:

$$\begin{pmatrix} a & b & c \\ f & g & h \\ k & l & m \end{pmatrix} \to \begin{pmatrix} a & b & c \\ 0 & ag - bf & ah - cf \\ 0 & al - bk & am - ck \end{pmatrix}$$

und im nächsten Schritt steht an der Stelle (3,3) der Term

$$(ag - bf)(am - ck) - (al - bk)(ah - cf),$$

dieser ist durch a teilbar; analog gilt dies für größere Matrizen.

Die Smithsche Normalform

Die folgenden Operationen können mit Matrizen über beliebigen Euklidischen Ringen, insbesondere über $\mathbf{Z}$ und $\mathbf{Q}$[x] durchgeführt werden.

Sei eine (rechteckige) Matrix (a_{ij} gegeben. Wir wenden Zeilen- und Spaltenoperationen an:

1. Durch Vertauschungen bringen wir an die Stelle (1,1) ein Element minimalen Grades.

2. Wir dividieren mit Rest:
$$a_{21} = a_{11} \cdot q + r.$$

3. Wir subtrahieren das q-fache der ersten Zeile von der zweiten, danach steht an der Stelle (2,1) gerade r.

   Wenn $r = 0$ ist, so bearbeiten wir die nächste Zeile.

   Wenn $r \neq 0$ ist, so hat es einen kleineren Grad als a_{11}, wir gehen zurück zu 1.

4. Wenn wir unterhalb von (1,1) Nullen erzeugt haben (das ist nach endlich vielen Schritten gelungen), so bearbeiten wir die erste Zeile und bringen dort Nullen hin. In jedem Schritt verkleinert sich evtl. der Grad von a_{11}.

5. Wenn nun a_{11} alle restlichen Einträge teilt, so ist es gut. Wenn aber a_{ij} kein Vielfaches von a_{11} ist, so addieren wir die i-te Zeile zur ersten und gehen zurück zu 1.

6. Nach endlich vielen Schritten haben wir es erreicht, daß a_{11} den Rest der Matrix teilt. Nun behandeln wir die Restmatrix analog.

Am Ende haben wir eine Diagonalmatrix, die sogenannte Shmithsche Normalform der Matrix A:
$$\begin{pmatrix} d_1 & & & \\ & d_2 & & \\ & & \ddots & \\ & & & d_n \end{pmatrix} \text{ mit } d_1 \mid d_2 \mid \ldots \mid d_n.$$

Diese Diagonalelemente heißen die Invariantenteiler der Matrix A. Zwei Matrizen A, B aus $M_{mn}(R)$ sind genau dann äquivalent (d.h. $XAY = B$ mit *invertierbaren* Matrizen $X \in M_{mm}, Y \in M_{nn}$), wenn ihre Invariantenteiler übereinstimmen.

Bei den betrachteten Operationen hat sich die Determinante der Matrix nicht geändert, sie kann also ohne Divisionen berechnet werden. Wenn wir auf diese Weise das charkteristische Polynom $\det(A - xE)$ einer Matrix A berechnen, erhalten wir eventuell schon eine Faktorisierung des Polynoms.

9 Primzahltest und Faktorisierung ganzer Zahlen

Eine Zahl p ist genau dann eine Primzahl, wenn sie keinen Primteiler $\leq \sqrt{p}$ besitzt. Um also große Zahlen auf Primalität zu untersuchen, sind Tabellen kleiner Primzahlen nützlich.
Wenn wir zum Beispiel alle Primzahlen zwischen 100 und 200 suchen, so müssen wir aus diesen Zahlen alle Vielfachen von 2, 3, 5, 7, 11 und 13 wegstreichen, was ganz einfach ist; es bleiben 21 Zahlen übrig.

Lehmer hat 1909 eine Tabelle aller Primzahlen unter 10 Millionen als Buch veröffentlich, und in einem zweiten Buch sind die kleinsten Primteiler aller Zahlen unter 10 Millionen enthalten, die nicht durch 2, 3, 5 oder 7 teilbar sind. Damit ist eine vollständige Faktorisierung dieser Zahlen möglich. Man sollte aber bedenken, daß ein Computer eine solche Zahl schneller zerlegen kann, als man in einer Tabelle nachschlagen kann.

Für ein Sieb-Programm braucht man nicht viel Speicherplatz: man repräsentiert jede Zahl durch ein Bit (die i-te Zahl durch das i-te Bit), setzt alle Bits auf 0, geht für alle relevanten Primzahlen p in $p$er Schritten über die Bits und setzt Einsen. Die restlichen Null-Bits repräsentieren Primzahlen.

Wenn aus irgendeinem Grund alle Primzahlen nacheinander durchforstet werden müssen, so sollte man $(p_i - p_{i-1})/2$ tabellieren, hierfür reichen 6 Bit für $p < 10^6$, also braucht man insgesamt 59 KByte, die Primzahldifferenzen für $p < 10^9$ passen in 8 Bit, dafür braucht man insgesamt 51 MByte, und für $p < 10^{12}$ benötigt man 12 Bit (72 GByte).

Die Anzahl $\pi(x)$ der Primzahlen unterhalb einer gegebenen Schranke x berechnet sich nach Lagrange (1830) wie folgt (alle Brüche sind als ihre ganzen Teile zu verstehen):

$$1 + \pi(x) = \pi(\sqrt{(}x)) + x - \sum_{p_i \le \sqrt{(}x)} \frac{x}{p_i} + \sum_{p_i < p_j \le \sqrt{(}x)} \frac{x}{p_i p_j} - \cdots$$

Nach dem Ein- und Ausschlußprinzip wird die Zahl der Vielfachen kleiner Primzahlen weggenommen, die der Vielfachen zweier kleiner Primzahlen hinzugefügt usw. Lagrange berechte so die Zahl der Primzahlen unter 10^6 zu 78.526; eine damals aktuelle Primzahltabelle enthielt 78.492 Einträge, der richtige Wert ist 78.489.
Meisel hat 1885 $\pi(10^9) = 50.847.478$ berechnet, dies sind 56 Zahlen zu wenig, wie von Lehmer erst 1958 bemerkt wurde.

Ein Primzahltest ist eine Bedingung $P(n) \in \{0,\ 1\}$ mit $P(n) = 1$ genau dann, wenn n eine Primzahl ist. Demgegenüber ist ein Kompositionstest eine Bedingung K, so daß aus $K(n) = 1$ folgt, daß n zusammengesetzt ist, d.h. wenn n eine Primzahl ist, so gilt stets $K(n) = 0$, es ist aber auch $K(xy) = 0$ möglich.

Der kleine Satz von Fermat besagt: Ist p eine Primzahl und $\mathrm{ggT}(a, p) = 1$, so gilt

$$a^{p-1} \equiv 1 \ (mod\ p).$$

Wenn also

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

gilt, so ist N zusammengesetzt.

Allerdings kann dieser Test versagen: $341 = 11 \cdot 31$, aber $2^{340} \equiv 1 \ (mod\ 341)$, jedoch wird die Zahl 341 entlarvt, wenn wir $a = 3$ wählen: $3^{340} \equiv 56 \ (mod\ 341)$. Man sagt: 341 ist eine Fermatsche Pseudoprimzahl zur Basis 2.
Von Lehmer (1936) stammt eine Liste aller Pseudoprimzahlen zur Basis 2 zwischen 10^7 (dem Ende damaliger Primzahllisten) und 10^8, die keinen Faktor ≤ 313 haben. Pomerance, Selfrigde und Wagstaff veröffentlichten 1980 eine Liste von 1770 Zahlen

unter $25 \cdot 10^9$, die pseudoprim zu allen Basen 2, 3, 5, 7 sind. In diesem Bereich genügen also *vier* Fermat-Tests als Primzahltest. Man bedenke, daß zur Berechnung von $a^n,\ n \approx 10^9 = 2^{30}$ jeweils nur 60 Multiplikationen und MOD-Operationen nötig sind.

Leider gibt es Zahlen, die pseudoprim zu jeder Basis sind, diese heißen Carmichael-Zahlen; die kleinste ist $561 = 3 \cdot 11 \cdot 17$, es gibt etwa 100000 unter 10^{15}.

Der Fermat-Test kann durch weitere Tests ergänzt werden, für die wir im folgenden die theoretischen Grundlagen legen.

Definition: Es sei $ggT(a,n) = 1$, dann heißt a ein qudratischer Rest von n, wenn $x^2 \equiv a\ (mod\ n)$ für ein x gilt, anderenfalls heißt a quadratischer Nichtrest. Wenn p eine ungerade Primzahl ist, so ist das folgende Legendre-Symbol definiert:

$$\left(\frac{a}{p}\right) = \begin{cases} 1, & a \text{ quadratischer Rest } mod\ p \\ -1, & a \text{ Nichtrest} \end{cases}$$

Lemma 9.1 *1. Die Menge der quadratischen Reste von n ist eine Untergruppe von* $(\mathbf{Z}/n\mathbf{Z})^*$.
2. Wenn $(\mathbf{Z}/n\mathbf{Z})^*$ *zyklisch ist (z.B. wenn n eine Primzahl ist), dann gibt es gleichviele Reste wir Nichtreste und das Produkt zweier Nichtreste ist ein Rest.*

Beweis: 1. Sei $x^2 \equiv a,\ y^2 \equiv b$, dann ist $(xy)^2 \equiv ab$.
2. Wenn $(\mathbf{Z}/n\mathbf{Z})^* = \langle g \rangle$ ist, dann sind $1,\ g^2,\ g^4, \ldots$ quadratische Reste und $g,\ g^3, \ldots$ sind quadratische Nichtreste. Schließlich ist das Produkt zweier ungerader Potenzen eine gerade Potenz. □

Folgerung 9.2 $\left(\frac{a}{p}\right)\left(\frac{b}{p}\right) = \left(\frac{ab}{p}\right)$.

Satz 9.3 (Euler) *Wenn* $ggT(a,p) = 1, p \neq 2$ *ist, so gilt* $\left(\frac{a}{p}\right) \equiv a^{(p-1)/2}\ (mod\ p)$.

Beweis: Sei $\mathbf{Z}/p\mathbf{Z} = \langle g \rangle$. Es sei $g^s \equiv -1$. Dann ist $g^{2s} = 1$, also ist $2s$ ein Vielfaches von $p-1$, denn $ord(g) = p-1$. Also sind für s nur die Werte $0,\ (p-1)/2,\ p-1$ möglich. Im ersten und dritten Fall ist aber $g^s = +1$, also folgt $g^{(p-1)/2} \equiv -1$. Sei nun $a = g^t$, dann ist $a^{(p-1)/2} = g^{t(p-1)/2} \equiv (-1)^t$, also ist a genau dann quadratischer Rest, wenn t gerade ist, und genau dann gilt $a^{(p-1)/2} \equiv 1$. □

Ohne Beweis geben wir das folgende „quadratische Reziprozitätsgesetz“ an:

Satz 9.4 *Seien* p, q *ungerade Primzahlen, dann gilt*

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

Wir können dies anwenden, um das Legendre-Symbol schnell zu berechnen:
4567 ist eine Primzahl.
$\left(\frac{123}{4567}\right) = \left(\frac{3}{4567}\right)\left(\frac{41}{4567}\right) = \left(\frac{4567}{3}\right)(-1)^{2283}\left(\frac{4567}{41}\right)(-1)^{20\cdot 2283} = \left(\frac{1}{3}\right)(-1)\left(\frac{16}{41}\right) = -1.$
Das folgende Jacobi-Symbol verallgemeinert das Legendre-Symbol:

Definition: Sei $n = \prod p_i^{a_i}$, wir setzen $\left(\frac{a}{n}\right) = \prod \left(\frac{a}{p_i}\right)^{a_i}$.

Satz 9.5 *Wenn a, b, n ungerade sind und $ggT(a,n) = ggT(b,n) = 1$ ist, so gilt* $\left(\frac{a}{n}\right)\left(\frac{b}{n}\right) = \left(\frac{ab}{n}\right)$. □

Wenn also a ein quadratischer Rest ist, so gilt $\left(\frac{a}{n}\right) = 1$, aber die Umkehrung gilt nicht.

Wir kommen zurück zum Primzahltest.

Wenn N ungerade und $ggT(a,N) = 1$, aber $a^{\frac{N-1}{2}} \not\equiv \pm 1 \ (mod\ N)$ ist, so ist N zusammengesetzt, denn nach dem Eulerschen Satz ist dieser Term im Primzahlfall gleich dem Legendre-Symbol.
Falls aber $a^{\frac{N-1}{2}} \equiv \pm 1$ ist, so berechne man das Jacobi-Symbol $\left(\frac{a}{N}\right)$. Wenn nun $a^{\frac{N-1}{2}} \not\equiv \pm\left(\frac{a}{N}\right)$ ist, so ist N zusammengesetzt, denn wenn N eine Primzahl wäre, so wären Jacobi- und Legendre-Symbole gleich und man wendet den Satz von Euler an.

Eine Zahl N mit $a^{\frac{N-1}{2}} \equiv \pm\left(\frac{a}{N}\right)$ heißt Euler-pseudoprim zur Basis a.
Einige Carmichael-Zahlen werden durch Euler entlarvt, z.B. 1729, denn $11^{864} \equiv 1$ $(mod\ 1729)$, aber $\left(\frac{11}{1729}\right) = -1$.

Pinch (1993) hat festgestellt, daß so bekannte Computeralgebrasysteme wie Mathematica, Maple V und Axiom einige bekannte Carmichael-Zahlen als Primzahlen passieren lassen.

Der folgende Satz von Lucas und Lehmer kehrt den Satz von Fermat um:

Satz 9.6 *Sei $N - 1 = \prod q_i^{a_i}$ die Primzahlzerlegung. Wenn ein a existiert, so daß*

$$a^{(N-1)/q_i} \not\equiv 1 \ (mod\ N) \text{ für alle } i,$$

aber

$$a^{N-1} \equiv 1 \ (mod\ N),$$

dann ist N eine Primzahl.

Beweis: Wenn N prim ist, so existiert in $\mathbf{Z}/N\mathbf{Z}^*$ ein Element der Ordnung $N-1$, und sonst nicht, da dann $\varphi(N) < N$ ist. Nach Voraussetzung ist die Ordnung von a ein Teiler von $N-1$. Jeder echte Teiler von $N-1$ ist ein Teiler von $(N-1)/q_i$, diese sind aber nicht gleich der Ordnung von A, also ist die Ordnung von A gleich $N-1$. □

Für die Auswahl dieser Zahl a sollte man keine mit $\left(\frac{a}{N}\right) = 1$ nehmen, denn diese können keine Erzeugenden sein.

Schließlich erwähnen wir den Lucas-Primzahltest für die Mersenne-Zahlen $M_n = 2^n - 1$. Diese Zahl ist genau dann prim, wenn für die rekursive Folge

$$v_0 = 4,\ v_i \equiv v_{i-1}^2 (\ mod\ M_n)$$

gilt:

$$V_{n-2} \equiv 0 (\ mod\ M_n)$$

gilt.

Wir wollen uns nun mit der Faktorisierung von Zahlen beschäftigen. Als Beispiel für die Schwierigkeiten, die hier zu erwarten sind, beginnen wir mit den Fermatzahlen

$$F_n = 2^{2^n} + 1,$$

diese Zahlen sind für kleine Werte von n Primzahlen: 3, 5, 17, 129; $F_4 = 65537$ ist die größte bekannte Primzahl unter den Fermatzahlen. Pépin hat gezeigt: F_n ist genau dann eine Primzahl, wenn

$$3^{\frac{F_n-1}{2}} \equiv -1 \ (mod\ F_n)$$

ist. Die kleinste Fermatzahl, wo die Zerlegbarkeit unklar ist, ist $F_{24} = 10^{5\,\text{Mio}}$; die Zahl F_{14} ist seit 1963 als zusammengesetzt bekannt, aber man kennt keinen Faktor.

Die Feststellung, ob eine Zahl eine Primzahl ist, ist relativ schnell zu machen. Faktorisierungen sind schwierig und langwierig. Deshalb sollte einen Faktorisierungsalgorithmus nur auf eine Zahl anwenden, von der man weiß, daß sie zerlegbar ist.

Ein erstes Beispiel der Zerlegung einer wirklich großen Zahl wurde durch Cole (1903) gegeben:

$$2^{67} - 1 \approx 10^9 \cdot 10^{13}.$$

1. Als erstes sollte man Probedivisionen durch kleine Primzahlen machen, die als Tabelle vorliegen oder on-line erzeugt werden.
Besser ist es, nicht nur durch Primzahlen zu teilen, sondern durch alle Zahlen der Form $6k \pm 1$ (sowie durch 2 und 3), dann braucht man in keiner Tabelle nachzuschlagen. Die Zahlen dieser Form erzeugt man sich, indem man $z = 5$ und $d = 2$ als Anfangswerte wählt und dann $z := z + d,\ d := 6 - d$ iteriert.
Die Grenze der Probedivision liegt bei $z > \sqrt{N}$, also wenn $z > N/z$ ist, man braucht also keine Wurzel zu berechnen, währede der Quotient sowieso berechnet werden muß. Man achte darauf, daß jeder gefundene Faktor sofort wegdividiert werden muß und daß Primfaktoren auch mehrfach auftreten können.

2. Wir fassen Produkte von Primzahlen blockweise zusammen:

$$p_0 = \prod_{2 \le p \le 97} P \approx 10^{37}$$

$$p_1 = \prod_{101 \le p \le 199} P \approx 10^{46}$$

$$\cdots$$

$$p_9 = \prod_{907 \le p \le 997} P \approx 10^{42}$$

Der mit dem Euklidischen Algorithmus berechenbare $ggT(N, p_i)$ teilt N.

3. Fermatsche Methode

Wir versuchen, die Zahl $N = a \cdot b$ in der Form $N = x^2 - y^2 = (x+y)(x-y)$ darzustellen. Dabei ist $x > \sqrt{N}$, wir beginnen also mit $m = \lfloor \sqrt{N} \rfloor + 1$, setzen $z = m^2 - N$ und überprüfen, ob dies eine Quadratzahl ist. Wenn nicht, so erhöhen wir m um 1, d.h. die nächste zu testende Zahl ist $z + 2m + 1$.

Es sei nebenbei bemerkt, daß man Quadratwurzeln wie folgt berechnen kann (eine Zahl ist dann eine Quadratzahl, wenn sie gleich dem Quadrat ihrer Wurzel ist): zunächst gibt es ein Iterationsverfahren

$$x_n = \frac{x_{n-1} + \frac{a}{x_{n-1}}}{2},$$

oder man verwendet die binomische Formel

$$(1+x)^{\frac{1}{2}} = 1 + \binom{\frac{1}{2}}{1}x + \binom{\frac{1}{2}}{2}x^2 + \cdots.$$

Schließlich kann man es einer Zahl schnell ansehen, daß sie keine Quadratzahl ist, denn die letzten beiden Dezimalstellen einer Quadratzahl haben die Form

$$uu, g1, g4, 25, u6, g9,$$

wobei u eine ungerade und g eine gerade Zahl bedeutet.

4. Legendre

Wir suchen $x \not\equiv \pm y$ mit $x^2 \equiv y^2 \ (mod\ N)$. Dann ist

$$x^2 - y^2 \equiv 0 \equiv (x+y)(x-y) \ (mod\ N),$$

also $(x+y)(x-y) = tN$, aber N teilt dieser Faktoren. Also sind die Primteiler von N auch in $x \pm y$ vorhanden, man berechne also $ggT(N, x-y)$.
Aber wie soll man solche x, y finden?
Wir suchen für „kleine“ Primzahlen p_i folgende Kongruenzen zu lösen:

$$x_k^2 \equiv (-1)^{e_{ok}} p_1^{e_{1k}} \cdots p_m^{e_{mk}} \ (mod\ N).$$

Wenn wir genügend viele gefunden haben, so sind die Vektoren $(e_{ok}, \ldots, e_{mk})$ modulo 2 linear abhängig, also gibt es a_k mit

$$\sum_{k=1}^{n} a_k(e_{ok}, \ldots, e_{mk}) \equiv (0, \ldots, 0),$$

also

$$\sum_{k=1}^{n} a_k(e_{ok}, \ldots, e_{mk}) = 2(v_0, \ldots, v_m).$$

Wir setzen dann $x = \prod x_k^{a_k}$, $y = (-1)^{v_o} p_1^{v_1} \cdots p_m^{v_m}$, es gilt

$$\begin{aligned} x^2 = \prod (x_k^2)^{a_k} &= (\prod (-1)^{e_{ok}} p_1^{e_{1k}} \cdots p_m^{e_{mk}})^{a_k} \\ &= (-1)^{\sum a_k e_{0k}} p_1^{\sum a_k e_{1k}} \cdots p_m^{\sum a_k e_{mk}} \\ &= (-1)^{2v_0} p_1^{2v_1} \cdots p_m^{2v_m} \\ &= y^2. \end{aligned}$$

Moderne Faktorisierungsmethoden wie die von Pollard, auf die wir aber hier nicht eingehen, haben ihre Leistungsgrenze bei 120 bis 150 Dezimalstellen.

Daß eine Zahl N durchschnittlich $\ln\ln N$ verschiedene Primfaktoren hat, wollen wir einmal hinnehmen. Wie groß aber sind diese?
Wir ordnen die Primteiler von N der Größe nach:

$$N = P_s(N)P_{s-1}(N)\cdots P_2(N)P(N),$$

wobei $P(N)$ der größte sei. Dann hat $N/P(N)$ noch $s-1$ verschiedene Primteiler, also

$$\begin{aligned} s-1 \approx \ln\ln\frac{n}{\ln N} &= \ln(\ln N - \ln P(N)) \\ &= \ln\ln N + \ln(1-\frac{\ln P(N)}{\ln N}) \\ &= s + \ln(1-\frac{\ln P(N)}{\ln N}), \end{aligned}$$

also $\ln(1-\frac{\ln P(N)}{\ln N}) \approx -1$, d.h. $1-\frac{\ln P(N)}{\ln N} \approx \frac{1}{e}$, also $\ln P(N) \approx (1-\frac{1}{e})\ln N = 0,632\ln N$ und somit

$$P(N) \approx N^{0,632}.$$

Primzahlen und Krytographie

Bei der Krytographie geht es darum, zunächst einer Zeichenkette eine Zahl T zuzuordnen, die mittels einer Verschlüsselungsfunktion f zu einem Code $C = f(t)$ chiffriert wird. Der Code C wird an den Empfänger gesandt, der über die Inverse f^{-1} der Verschlüsselungsfunktion verfügen muß, um den Originaltext lesen zu können.
Die Funktionen f und f^{-1} sind im einfachsten Fall Tabellen. Besser sind Funktionen, die von einem Parameter abhängen (einem Schlüssel), diesen Schlüssel sollte man häufig wechseln.

Ein erfahrener Verschlüßler geht davon aus, daß dem unberechtigten Mithörer seiner Nachricht der Verschlüßlungalgorithmus bekannt geworden ist, (hoffentlich) nicht aber der jeweilige Schlüssel.
Ein einfacher Schlüssel, der schon von Caesar benutzt wurde, besteht im Verschieben der Buchstaben um eine konstante Zahl: $C = T + k$. Durch Rivest, Shamir, Adleman ist nun Codierungsverfahren mit öffentlichem Schlüssel entwickelt worden, das RSA-Verfahren. Dabei ist

$$C = T^k \ (mod\ N), \quad \text{also } T = \sqrt[k]{N}\ (mod\ N),$$

die erste Operation ist schnell durchgeführt, die zweite ist schwierig, falls N eine zusammengesetzte Zahl ist.

Wir verfahren wie folgt: Wir suchen Zahlen N, k, so daß ein k' mit $T = C^{k'}\ (mod\ N)$ existiert, d.h. $a^{kk'} \equiv a\ (mod\ N)$ für alle a, und es soll schwer sein, eine Zahl s mit $a^{ks} \equiv a\ (mod\ N)$ zu bestimmen.

Für die Eulersche Funktion φ gilt $a^{\varphi(N)} \equiv 1 \ (mod\ N)$, falls $ggT(a, N) = 1$ ist. Die Carmichael-Funktion $\lambda(N)$ ist ein Teiler von $\varphi(N)$, und es gilt

$$a^{\lambda(N)} \equiv 1 \ (mod\ N),$$

für *alle* a, falls N ein Produkt verschiedener Primzahlen ist. Wenn $p \neq q$ Primzahlen sind, dann gilt

$$\lambda(pq) = \frac{(p-1)(q-1)}{ggT(p-1, q-1)} = kgV(p-1, q-1).$$

Dann gilt $a^{kk'} \equiv a \ (mod\ N)$ genau dann, wenn $kk' \equiv 1 \ (mod \lambda(n)$. Wir setzen also $N = pq$ und suchen ein m mit

$$m\lambda(N) + 1 = k \cdot k'$$

indem wir am Besten k vorgeben und die Kongruenz $m\lambda(N) + 1 \equiv 0 \ (mod\ k)$ lösen. Wenn $\lambda(N)$ nicht bekannt ist, kann ein Unbefugter die Zahl k' nicht bestimmen.

Wir bestimmen dann $C = f(T) = T^k \ (mod\ N)$ und versenden diesen Code. Dabei sollte k nicht zu klein sein, damit $T^k > N$ wird und tatsächlich eine $(mod\ N)$-Reduktion vorgenommen wird.
Der Empfänger berechnet

$$C^{k'} \equiv T^{kk'} \equiv T^{m\lambda(N)+1} \equiv T \ (mod\ N).$$

Die Schlüssel N und k kann man als Zeitungsinserat veröffentlichen. Der unbefugte Entschlüßler muß die Zahl $\lambda(N)$ berechnen, dazu muß er die Zahl N zerlegen, und das schafft er nicht in einer vernünftigen Zeit, wenn p und q groß sind.

10 Faktorisierung von Polynomen

Gegeben ist ein Polynom $f(x) \in \mathbf{Q}[x]$ mit rationalen Koeffizienten. Das Polynom $f(x)$ ist entweder irreduzibel, oder es gibt Polynome $g(x),\ h(x) \in \mathbf{Q}[x]$, so daß $f(x) = g(x)h(x)$ gilt. Solche Teiler von $f(x)$ wollen wir berechnen.
Als erstes wollen wir eine Schranke für die Koeffizienten eines Faktors eines Polynoms angeben:

Satz 10.1 *Sei $f(x) \in \mathbf{C}[x],\ \deg(f) = n$, die Beträge der Nullstellen von $f(x)$ seien durch die Zahl b beschränkt, dann haben die Koeffizienten eines Teilers $p(x)$ von $f(x)$ höchstens den Betrag $\max_k \left|\binom{r}{k} b^k\right|$, wobei $r = \deg(p)$.*

Beweis: Sei $p(x) = \prod(x + z_i) = \sum p_i x^{r-i}$, dann gilt nach Viètá

$$p_1 = z_1 + \ldots + z_r,$$

$$p_2 = z_1 z_2 + \ldots + z_{r-1} z_r,$$

$$\ldots$$

$$p_r = z_1 \dots z_r,$$
$$\text{also } \mid p_i \mid \leq \binom{r}{i} b^i. \quad \square$$

Bleiben wir bei Polynomen mit rationalen Koeffizienten. Wenn wir etwa vorhandene Nenner der Koeffizienten von $f(x)$ „hochmultiplizieren“, erhalten wir ein Polynom mit ganzzahligen Koeffizienten. Wir überlegen schnell, wie die Zerlegungen in $\mathbf{Q}[x]$ und in $\mathbf{Z}[x]$ zusammenhängen:
Sei $f(x) = \sum a_i x^{n-i} \in \mathbf{Z}[x]$, der größte gemeinsame Teiler der Koeffizienten heißt der Inhalt von f: $\text{cont}(f) = \text{ggT}(a_0, \dots, a_n)$.
Ein Polynom $f(x)$ heißt primitiv, wenn $\text{cont}(f) = 1$ ist. Es gilt das

Lemma 10.2 (Lemma von Gauß) *: Wenn $f(x)$ und $g(x)$ primitive Polynome sind, so ist auch $h(x) = f(x)g(x)$ primitiv.*

Beweis: Andernfalls gibt es eine Primzahl p, die alle Koeffizienten von $h(x)$ teilt. Seien $f_p(x),\ g_p(x),\ h_p(x)$ die Polynome aus $\mathbf{Z}/(p)[x]$, deren Koeffizienten die Restklassen der Koeffizienten von $f,\ g,\ h$ sind. Dann gilt

$$f_p(x)g_p(x) = h_p(x) = 0 \text{ in } \mathbf{Z}/(p)[x],$$

ein Widerspruch, da es in einem Polynomring über einem Körper keine Nullteiler gibt. $\square$
Nun beweisen wir den

Satz 10.3 (Satz von Gauß) *Wenn sich ein Polynom $f(x) \in \mathbf{Z}[x]$ in ein Produkt zweier Polynome mit rationalen Koeffizienten zerlegen läßt, so läßt es sich auch in ein Produkt von Polynomen mit ganzzahligen Koeffizienten zerlegen.*

Beweis: Jedes Polynom $f(x) \in \mathbf{Q}[x]$ kann als

$$f(x) = \frac{a}{b} f^*(x)$$

mit $a, b \in \mathbf{Z}$ und einem primitiven Polynom $f^*(x)$ dargestellt werden. Sei also

$$f(x) = g(x)h(x) = \frac{a}{b} f^*(x)g^*(x)$$

mit primitiven f^*, g^* (die Brüche haben wir schon zusammengefaßt). Dann ist das Polynom $f^*(x)g^*(x)$ primitiv, auf der linken Seite stehen nur ganzzahlige Koeffizienten, also muß $\frac{a}{b}$ eine ganze Zahl sein. $\square$

Wir beschränken uns also auf die Faktorisierung ganzzahliger Polynome. Kronecker hat gezeigt, daß dieses Problem algorithmisch lösbar ist: Sei $\deg(f) = n$, dann kann ein Teiler von f höchstens den Grad $m = n/2$ haben. Seien $m+1$ verschiedene Zahlen $a_0, \dots a_m$ gewählt, wir berechnen $f(a_0), \dots, f(a_m)$. Wenn es einen Teiler $g(x)$ von $f(x)$ gibt, so gilt $g(a_i) \mid f(a_i)$ für alle i. Sei F_i die Menge aller Teiler von $f(a_i)$. Nun wählen wir für $k = 1, \dots, m$ $(k+1)$-tupel $(b_{i_1}, \dots, b_{i_{k+1}})$ aus, wobei $b_{i_j} \in F_{i_j}$ gilt und bilden das (eindeutig bestimmte) Polynom $g(x)$ vom Grade k mit $g(a_{i_j}) = b_{i_j}$. Wenn es einen

Teiler $g(x)$ gibt, so ist es eines von diesen. Wir müssen also alle k-elementigen Mengen, deren Elemente in verschiedenen F_i liegen, durchforsten, das sind ungefähr 2^m Stück. Das Polynom $g(x)$ mit $g(a_{i_j}) = b_{i_j}$ erhalten wir wie folgt: Sei

$$L_i(x) = \frac{(x_1 - a_0)(x_1 - a_2)\dots(x - a_{i-1})(x - a_{i+1})\dots(x - a_k)}{(a_i - a_0)(a_i - a_2)\dots(a_i - a_{i-1})(a_i - a_{i+1})\dots(a_i - a_k)},$$

dann gilt $L_i(a_j) = \delta_{ij}$, also leistet $g(x) = \sum b_i L_i(x)$ das Verlangte.

Betrachten wir ein Beispiel: $f(x) = x^4 + 4$, $m = 2$, wir wählen $a_0 = 0$, $a_1 = 1$, $a_2 = -1$ und erhalten $f(0) = 4$, $f(1) = f(-1) = 5$, also die Teilermengen $F_0 = \{\pm 1, \pm 2, \pm 4\}$, $F_1 = F_2 = \{\pm 1, \pm 5\}$. Probieren wir $b_0 = 1$, $b_1 = -1$, $b_2 = 5$, das Polynom, daß diese Werte annimmt, ist gleich $x^2 - 3x + 1$, dies ist kein Teiler von $f(x)$. Von den insgesamt 96 Möglichkeiten ist $b_0 = 2, b_1 = 5, b_2 = 1$ eine günstige, sie führt zum Polynom $x^2 + 2x + 2$, dies ist ein Teiler von $f(x)$.

Als ein Verfahren, daß die Problemgröße reduziert, behandeln wir zunächst die quadradfreie Zerlegung von Polynomen.
Gegeben sei ein Polynom $f(x)$, gesucht ist eine Zerlegung $f = \prod_{ige1} g_i^i$, wobei g_i alle ireduziblen Faktoren von f enthält, die i-mal in f vorkommen. Um diese Zerlegung herzuleiten, zerlegen wir f über $\mathbf{C}$ in Linearfaktoren:

$$f(x) = \prod (x - a_i)^{n_i}$$

. Die Ableitung von f hat dann die Form

$$f' = \sum_i (n_i (x - a_i)^{n_i - 1} \prod_{j \neq i} (x - a_j)^{n_j}.$$

Demnach hat $\mathrm{ggT}(f, f')$ die Faktoren $(x - a_j)^{m_j}$, wobei $m_j = n_j$ oder $m_j = n_{j-1}$ ist. Also teilt $(x - a_j)$ alle Summanden von f' außer einem und es ist $\mathrm{ggT}(f, f') = \prod (x - a_i)^{n_i - 1}$. Das Polynom $g(x) = \frac{f}{\mathrm{ggT}(f, f')} = \prod (x - a_i)$ hat also dieselben Nullstellen wie f, aber jede nur als einfache, und $\mathrm{ggT}(g, \mathrm{ggT}(f, f')) = \prod_{n_i > 1} (x - a_i)$ hat alle mehrfachen Nullstellen von f als einfache Nullstellen. Somit besitzt

$$\frac{g}{\mathrm{ggT}(g, \mathrm{ggT}(f, f'))} = \prod_{n_i = 1} (x - a_i) = g_1$$

genau die einfachen Faktoren von f.
Wenn wir dasselbe Verfahren auf $f_1 = \mathrm{ggT}(f, f')$ anwenden, erhalten wir die einfachen Faktoren von f_1, also die zweifachen Faktoren von f, usw.
Wir bemerken schließlich, daß die g_i teilerfremd sind.

Wir wollen nun versuchen, mit weniger Versuchen als Kronecker auszukommen. Wir werden Polynome faktorisieren, indem wir folgende Schritte unternehmen:
1. Mehrfache Nullstellen von $f(x)$ werden entfernt (s. o.).
2. Wir suchen eine Primzahl p, so daß $f_p(x)$ denselben Grad wie $f(x)$ hat (p darf den höchsten Koeffizienten von $f(x)$ nicht teilen) und daß $f_p(x)$ weiter quadratfrei ist (p darf die Diskriminante von $f(x)$ nicht teilen. Wir zerlegen nun $f_p = g_p h_p$ im Ring $\mathbf{Z}/(p)[x]$.

3. Wir „liften“ die mod$-p$-Zerlegung zu einer mod$-p^k$-Zerlegung für hinreichend großes $k : f(x) \equiv \prod h_i(x) (\text{mod } p^k)$.
4. Jeder Faktor $g(x)$ von $f(x)$ ist das Produkt einiger der h_i (vielleicht selbst eines der h_i). Wir überprüfen, ob $h_i \mid f$, ob $h_i h_j \mid f, \ldots$
Die „durschnittliche“ Anzahl der Faktoren eines Polynoms vom Grade n ist $\log(n)$, also sind etwa $2^{\log(n)} = n$ Probedivisionen nötig. Jedoch entspricht nicht jeder Zerlegung modulo p einer Zerlegung über $\mathbf{Z}$:

$$x^2 + 2 \equiv (x+1)(x+2) \text{ (mod 3)}$$

Wir beschäftigen uns noch einmal mit endlichen Körpern.
Ein endlicher Körper K hat die Charakteristik p, besitzt also $\mathbf{F}_p = Z/pZ$ als Unterkörper. K ist als $\mathbf{F}_p$-Vektorraum endlichdimensional, hat also $q = p^n$ Elemente.
Die multiplikative Gruppe von K hat dann $q-1$ Elemente, also gilt $a^q = a$ für alle $a \in K$, d.h. K ist als Zerfällungsköprer des Polynoms $x^q - x$ bis auf Isomorphie eindeutig bestimmt; er wird mit $\mathbf{F}_q$ bezeichnet.
Umgekehrt: Sei $\mathbf{F}_q$ ein Unterkörper eines Körpers L, dann ist die Menge

$$E = \{b \in L \mid b^q = b\}$$

ein Unterkörper von L, wie wir gleich sehen: Seien $x, y \in E$, dann ist $(xy)^q = x^q y^q = xy \in E$, wenn $x \neq 0$ ist, so ist $x^{q-1} = 1$, also $x^{q-2} = x^{-1}$ und $(x^{-1})^q = (x^q)^{q-2} = x^{q-2} = x^{-1} \in E$. Weiter folgt aus $(x+y)^p = x^p + y^p$ durch Induktion $(x+y)^q = x^q + y^q = x + y \in E$ und schließlich $(-x)^q = (-1)^q x^q = -x$, da entweder q ungerade oder $p = 2$ ist.

Der Körper E besteht aus den Nullstellen des Polynoms $f(x) = x^q - x$ in L, das Polynom f hat nur einfache Nullstellen, weil $f'(x) = qx^{q-1} - 1 = -1 \neq 0$ ist, es gibt also genau q Stück, also ist $E = \mathbf{F}_q$.

Wir halten fest:

$$\mathbf{F}_q = \{b \in L \mid b^q = b\}.$$

Außerdem gilt $x^q - x = \prod_{a \in \mathbf{F}_q} (x - a)$.

Sei $K = \mathbf{F}_q$ und $f(x) = f_1(x) \cdots f_k(x) \in K[x]$ ein quadratfreies Polynom mit den irreduziblen Faktoren f_i. Wir wollen die Anzahl k der Faktoren bestimmen.
Wir betrachten die Faktoralgebra $A = K[x]/(f(x))$, nach dem chinesischen Restsatz gilt

$$A = \oplus_{i=1}^k K[x]/(f_i(x)) = \oplus K_i,$$

wobei die K_i Körper mit q^{d_i} Elementen sind ($d_i = \deg(f_i)$. Wir betrachten die Funktion $Q : A \longrightarrow A$, $Q(a) = a^q - a$. Da alle Körper K_i die Charakteristik p haben, ist Q eine K-lineare Abbildung. Die K_i sind Q-invariant, wir setzen $Q_i = Q \mid_{K_i}$ und wissen, daß $\text{Ker}Q_i = \mathbf{F}_q = K$ ist. Also ist $\text{Ker}Q = K^k$, d.h. die Zahl k der irreduziblen Faktoren von $f(x)$ ist gleich $\dim_K(\text{Ker}Q)$.

Wir betrachten ein Beispiel:

Sei $q = p = 7$, $f(x) = x^4 - 3x^3 - 3x^2 - 3x + 1$. Dann hat A die Dimension 4; wir berechnen die Bilder der kanonischen Basis.

$$Q(1) = 1^7 - 1 = 0$$

$$Q(x) = x^7 - x = -x^3 + 2x^2 - 2x + 1$$
$$Q(x^2) = x^{14} - x^2 = -2x^3 - 2x^2 + 3x + 2$$
$$Q(x^3) = x^{21} - x^3 = -3x^3 - x^2 + x + 3$$

Die Darstellungsmatrix vom Q ist dann

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 0 & -2 & 3 & 1 \\ 0 & 2 & -3 & -1 \\ 0 & -1 & -2 & -3 \end{pmatrix},$$

sie hat den Rang 1, also hat f drei irreduzible Faktoren. Es ist $\mathrm{ggT}(f, f') = x - 1$, dies ist also ein doppelter Linearfaktor von f und wir erhalten $f(x) = (x-1)^2(x-3)(x+2)$.

Im Folgenden werden wir ein Polynom $a(x) = \sum a_i x^{n-1} \in \mathbf{Z}/(p)[x]$ mit dem Zeilenvektor $a = (a_0, \ldots, a_n)$ seiner Koeffizienten identifizieren.
Sei $f(x) \in \mathbf{Z}/(p)[x]$ ein gegebenes Polynom. Wir konstruieren die Matrix

$$Q = (r_{ij})$$

folgendermaßen: Die Zeilen von Q seien die Koeffizienten von x^{ip} modulo $f(x)$ für $i = 0, \ldots, n-1$.

Satz 10.4

$$aQ \equiv a^p \ (mod\ f(x)).$$

Beweis: Es ist

$$a(x)^p \equiv a(x^p) = \sum a_i x^{pi} \equiv \sum a_i \sum r_{ik} x^k \ (\mathrm{mod}\ f(x))$$

$$= \sum (\sum a_i r_{ik}) x^k = aQ. \quad \square$$

Nun können wir $f(x)$ modulo p zerlegen:

Satz 10.5 (Berlekamp) *Sei $h(x) \in \mathbf{Z}/(p)[x]$ ein Polynom mit $h^p - h \equiv 0 (mod\ f)$, dann gilt $f = \prod_{i=0}^{p-1} ggT(f, h-i)$. Die Zahl der irreduziblen Teiler von $f(x)$ in $\mathbf{Z}/(p)[x]$ ist gleich der Dimension des Nullraums der Matrix $Q - E$.*

Beweis: Es gilt $h^p - h = h(x^p - x) = h(\prod(x-i)) = \prod(h-i)$ (siehe oben). Wenn nun f ein Teiler von $h^p - h$ ist, so folgt $f \mid \prod \mathrm{ggT}(f, h-i)$. Andererseits gilt $\mathrm{ggT}(f, h-i) \mid f$ und für $i \neq j$ sind $h-i$ und $h-j$ teilerfremd, also gilt die behauptete Gleichheit.
Weiter gilt $f \mid (h^p - h)$ genau dann, wenn $hQ = h^p = h \ (\mathrm{mod}\ f)$, also wenn $h(Q-E) = 0$.

Sei $f = f_1 \dots f_r$ ein Teiler von $\prod(h-i)$, die f_i seien die paarweise verschiedenen irreduziblen Faktoren von f. Dann gilt $f_i \mid (h - s_i)$ für gewisse $s_i \in \{0, \dots, p-1\}$. Zu gegebenen $s_1, \dots, s_r$ gibt es nach dem chinesischen Restsatz ein modulo f eindeutig bestimmtes h mit $h \equiv s_i \pmod{f_i}$, also gibt es zu jedem der p^r r-tupel $(s_1, \dots, s_r)$ eine Lösung h, der Lösungsraum des Gleichungssystems $h(Q-E) = 0$ hat also die Dimension r. □

Beispiel:

Wir wählen $p = 2$ und betrachten $f(x) = x^2 + x$. Dann ist $Q = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ dann hat $Q-E) = 0$ einen 2-dimensionalen Nullraum. Wenn wir die Lösung $h(x) = 1$ verwenden, erhalten wir $\mathrm{ggT}(f, h-i) = 1$, also keinen nichtkonstanten Teiler. Die Lösung $h(x) = x$ liefert mit $i = 0,\ 1$ die beiden Teiler $\mathrm{ggT}(f, x-i)$.

Den nächsten Schritt, das Liften einer Zerlegung modulo eine Primzahlpotenz, machen wir mit Hilfe des Henselschen Lemmas:

Lemma 10.6 (Lemma von Hensel) *Seien $f(x),\ g_0(x),\ h_0(x) \in \mathbf{Z}[x]$ Polynome, so daß*

$$f(c) \equiv g_0(x)h_0(x) \ (mod\ p),$$

wobei g_0, h_0 teilerfremd sind, dann gibt es Polynome $g_k(x),\ h_k(x)$, so daß

$$f(x) \equiv g_k(x)h_k(x) \ (mod\ p^{k+1}),$$

$$g_k(x) \equiv g_0(x) \qquad (mod\ p),$$
$$h_k(x) \equiv h_0(x) \qquad (mod\ p).$$

Beweis: Wir führen die Induktion über k. Seien g_k und h_k schon konstruiert. Nach Voraussetzung ist

$$f(x) - g_k(x)h_k(x) = p^{k+1} \cdot w \tag{1}$$

ein Polynom, dessen Koeffizienten durch p^{k+1} teilbar ist. Wegen der Teilerfremdheit von g_0 und h_0 gibt es Polynome $l(x), m(x)$ mit

$$l(x)g_0(x) + m(x)h_0(x) \equiv 1 \qquad (\mathrm{mod}\ p), \tag{2}$$

wir setzen

$$g_{k+1}(x) = g_k(x) + p^{k+1}u(x),$$
$$h_{k+1}(x) = h_k(x) + p^{k+1}v(x),$$

wobei die Polynome u und v noch zu bestimmen sind. Dann gilt

$$g_{k+1}h_{k+1} - f = g_kh_k - f + p^{k+1}(g_kv + h_ku) + p^{2k+2}uv,$$

wegen (1) folgt

$$g_{k+1}h_{k+1} - f \equiv p^{k+1}(g_kv + h_ku - w) \qquad (\mathrm{mod}\ p^{k+2}),$$

die rechte Seite ist durch p^{k+2} teilbar, wenn

$$g_k v + h_k u \equiv w \qquad (mod\ p)$$

wäre. Wir multiplizieren (2) mit w:

$$wlg_0 + wmh_0 \equiv w \qquad (\text{mod } p),$$

führen eine Restdivision durch:

$$wm = qg_0 + u$$

und setzen dies in die letzte Gleichung ein:

$$(wl + qh_0)g_0 + uh_0 \equiv w \ (\text{mod } p).$$

Wenn wir den Klammerausdruck mit v benennen und berücksichtigen, daß $g_0 \equiv g_k (\text{mod } p)$ und $h_0 \equiv h_k (\text{mod } p)$ gilt, erhalten wir das Gewünschte. □

Beispiel: Sei $f(x) = 5x^2 + 57x + 70$; wir wählen $p = 2$ und erhalten modulo 2 das schon oben betrachtete Polynom $x^2 + x$. Die Hensel-Liftung liefert bei der Eingabe von x und $x - 1$ schrittweise:

$$t = 1 : \ x + 2, \ x - 1$$

$$t = 2 : \ x + 2, \ x - 1$$

$$t = 3 : x + 2, \ -3x - 1$$

$$t = 4 : \ x - 6, \ 5x + 7$$

$$t = 5 : \ x + 10, \ 5x + 7,$$

dies ist die Zerlegung von $f(x)$ über $\mathbf{Z}$.

Multivariate Polynome

Wir wollen nur kurz und ohne Beweise ein Verfahren vorstellen, wie E. Kaltofen (1982) die Faktorisierung von Polynomen in zwei Variablen durchführte.
Sei $f(x, y) \in \mathbf{Z}[x, y]$ ein Polynom, so das 1) $f(x, 0)$ quadratfrei ist und 2) $f(x, y) = \sum b_i(y) x^{n-i}$ mit $b_0(y) = 1$ gilt. Weiter sei $h(x)$ ein irreduzibler Faktor von $f(x, 0)$. Wir setzen

$$d = 2 \cdot \deg_x f \cdot \deg_y f,$$

im Körper

$$F = \mathbf{Q}[t]/(h(t))$$

ist $(t \bmod h)$ eine Nullstelle von $h(x)$, wir bestimmen ein Element $b \in F[y]$ mit

$$f(b, y) \equiv 0 \qquad (\text{mod } y^{d+1})$$

mit Hilfe einer (abgeänderten) Newton-Iteration wie folgt: wir setzen

$$a_0 = t \bmod h, \qquad f_x = \frac{\partial f}{\partial x}, \qquad s = 1/f_x(a_0),$$

und für $k = 1, \ldots, d$

$$a_k = a_{k-1} - s \cdot f(a_{k-1}, y) \text{ modulo } y^{k+1}$$

(d.h. höhere Potenzen von y werden weggelassen), und schließlich

$$b = a_d.$$

Nun suchen wir die kleinste Zahl i, $\deg h \leq i < \deg_x f$, so daß $u_0, \ldots, u_{i-1} \in \mathbf{Q}[y]$ mit $\deg_y(u_j) \leq \deg_y f$ existieren, so daß

$$b^i + \sum u_j b^j \equiv 0 \qquad (\text{mod } y^{d+1})$$

gilt. Dann ist

$$g(x, y) = x^i + \sum u_j x^j \in \mathbf{Q}[x, y]$$

ein Faktor von $f(x, y)$.

11 Gröbner-Basen

Der Ausgangspunkt ist die Untersuchung von Lösungen polynomialer Gleichungen in mehreren Unbekannten:

$$f_1(x_1, \ldots, x_n) = 0$$

$$\ldots$$

$$f_m(x_1, \ldots, x_n) = 0.$$

Die Lösungsmenge dieses Gleichungssystems ändert sich nicht, wenn wir mit diesen Gleichungen Operationen (z.B. Addition eines Vielfachen einer Gleichung zu einer anderen) durchführen, die rückgängig gemacht werden können, also wenn wir wenn wir ein anderes Erzeugendensystem des von den Polynomen $f_1, \ldots, f_m$ erzeugten Ideals in $\mathbf{Q}[x_1, \ldots, x_n]$ betrachten.
In der Menge S der Monome $x_1^{k_1} \ldots x_n^{k_n}$ führen wir eine Ordnung ein, die folgende Forderungen erfüllt:

1. für alle $a, b, c \in S$ gilt: wenn $a < b$, so gilt auch $ac < bc$,
2. für $b \neq 1$ gilt $a < ab$.

Ein Beispiel für eine derartige Ordnung ist die „Grad-lexikographische“ Ordnung: Wenn $\deg(a) < \deg(b)$, so setzen wir $a < b$ fest; wenn $\deg(a) = \deg(b)$ gilt, so setzen wir $a < b$, falls

$$a = x_{i_1} \ldots x_{i_k} x_p \ldots$$

$$b = x_{i_1} \ldots x_{i_k} x_q \ldots$$

und $p < q$ gilt.
Wenn dann $f \in \mathbf{Q}[x_1, \ldots, x_n]$ ist, so nennt man das größte in f vorkommende Monom das Leitmonom $LM(f)$, den zugehörigen Koefizienten nennt man des Leitkoeffizienten $lc(f)$.

Sei nun $F = \{f_1, \ldots, f_m\} \subseteq \mathbf{Q}[x_1, \ldots, x_n]$ eine Menge von Polynomen und $f \in \mathbf{Q}[x_1, \ldots, x_n]$, dann nennen wir ein Polynom g eine F-Reduzierte von f, wenn es Polynome $h_1, \ldots, h_m$ gibt, so daß

$$g = f - \sum h_i f_i$$

gilt und entweder $g = 0$ oder $LM(g) < LM(f)$ ist. In dieser Situation schreiben wir kurz

$$f \longrightarrow_F g.$$

Wenn sich eine Reduzierte weiter reduzieren läßt, so kann man dies solange wiederholen, bis ein Polynom g_* entsteht, dessen Leitmonom durch kein Leitmonom der f_i teilbar ist, das also nicht F-reduziert werden kann. Ein derartiges Polynom nennen wir eine F-Normalform von f. Solche Normalformen müssen nicht eindeutig bestimmt sein, z.B. hat für $F = \{x^4 - 1, x^3 - 5\}$ das Polynom x^5 die beiden Normalformen x und $5x^2$.

Definition: Ein Erzeugendensystem F eines Ideals $I \subseteq \mathbf{Q}[x_1, \ldots, x_n]$ heißt Gröbner-Basis von I, wenn für jedes $f \in I$ das Nullpolynom die einzige F-Normalform von f ist.

Beispiel (Davenport):
Wir suchen die Lösungen des folgenden Gleichungssystems

$$g_1 = x^3yz - xz^2 = 0,$$

$$g_2 = xy^2z - xyz = 0,$$

$$g_3 = x^2y^2 - z = 0,$$

wir setzen $x > y > z$, eine Gröbner-Basis des von g_1, g_2, g_3 erzeugten Ideals besteht aus g_2, g_3 und den folgenden Polynomen:

$$g_4 = x^2yz - z^2,$$

$$g_5 = yz^2 - z^2,$$

$$g_6 = x^2z^2 - z^3,$$

wie wir später sehen werden. Wenn $z = 0$ ist, so erhalten wir aus der dritten Gleichung, daß $x = 0$ oder $y = 0$ gelten muß. Wenn $z \neq 0$ ist, so folgt aus der fünften Gleichung $y = 1$ und aus der dritten $x^2 = z$. Die Menge aller Lösungen besteht aus zwei sich schneidenden Geraden und einer Parabel.

Sei $I \subseteq \mathbf{Q}[x_1, \ldots, x_n]$ ein Ideal und $LM(I) \subseteq S$ die Menge aller Leitmonome von Elementen aus I. Sei $m \in LM(I)$ und $b \in S$, dann gibt es ein $f \in I$ mit $m = LM(f)$ und weil I ein Ideal ist, gilt $bf \in I$, wegen der Monotonie der Ordnung ist dessen Leitmonom gleich bm, also gilt $bm \in LM(I)$, wir sagen: $LM(I)$ ist ein Ideal der Halbgruppe S.

Satz 11.1 *F ist genau dann eine Gröbner-Basis des Ideals I, wenn $LM(F)$ das Halbgruppenideal $LM(I)$ erzeugt.*

Beweis: Sei $LM(F) \cdot S = LM(I)$ und $0 \neq g \in I$, dann ist $LM(g) = LM(f) \cdot b$ für ein geeignetes $b \in S$ und ein $f \in F$, also läßt sich g reduzieren:

$$h = g - bf,$$

nun ist $h = 0$ oder $LM(h) < LM(g)$. Da es keine unendliche absteigende Folge von Monomen gibt, erhalten wir nach endlich vielen Schritten das Nullpolynom als Normalform.
Sei umgekehrt $LM(F) \cdot S \neq LM(I)$, dann gibt es ein $0 \neq g \in I$, das sich mittels keines Polynoms $f \in F$ reduzieren läßt, also bereits seine Normalform darstellt. □

Definition: Seien $f, g \in \mathbf{Q}[x_1, \ldots, x_n]$ normierte Polynome, das kleinste gemeinschaftliche Vielfache ihrer Leitmonome sei gleich

$$h = p \cdot LM(f) = q \cdot LM(g),$$

das folgende Polynom $s(f, g)$ heißt das S-Polynom von f, g:

$$s(f, g) = pf - qg.$$

Der folgende Algorithmus (Buchberger 1970) berechnet eine Gröbner-Basis des von F erzeugten Ideals:
Wir bilden die Menge aller Paare

$$P = \{(f_i, f_j) \mid f_i, f_j \in F, f_i \neq f_j\}.$$

Solange die Menge P nicht leer ist, bilden wir für jedes Paar $(f_i, f_j) \in P$ das S-Polynom $s(f_i, f_j)$, entfernen es aus P und prüfen, ob dessen F-Normalform h_{ij} von Null verschieden ist. Wenn dies der Fall ist, so fügen wir h_{ij} zu F hinzu und beginnen von vorn. Als Resultat erhalten wir ein Erzeugendensystem des von F erzeugten Ideals, wo die Normalform jedes S-Polynoms Null ist.

Satz 11.2 *Das Resultat des obigen Algorithmus ist eine Gröbner-Basis von $I = (F)$.*

Beweis: Wir zeigen, daß $LM(I)$ von $LM(F)$ erzeugt wird. Sei $g \in I, g = \sum c_i p_i f_i$ mit $c_i \in \mathbf{Q}, f_i \in F$ und $p_i \in$ S. Weiter gelte

$$LM(p_1 f_1) = \ldots = LM(p_r r_r) > LM(p_i f_i) \text{ für alle } i > r.$$

Wenn $r = 1$ ist, so ist $LM(g) = LM(p_1 f_1) = p_1 LM(f_1) \in S \cdot LM(F)$ wie gewünscht. Wenn $r > 1$ ist, so werden wir diese Zahl schrittweise verkleinern: Wir setzen

$$l_i = LM(f_i),\ f_i = l_i + h_i,$$

dann gilt

$$p_1 l_1 = p_2 l_2.$$

Dann tritt einer der folgenden Fälle auf: l_1 teilt p_2 oder l_1 und l_2 haben einen echten gemeinsamen Teiler. Sei zunächst l_1 ein Teiler von $p_2, l_1 w = p_2$, dann gilt

$$p_1 = w l_2$$

und

$$p_2h_2 = l_1wh_2 < l_1wl_2 = p_2l_2 = p_1l_1,$$

also ist jeder Summand in p_2h_2 auf der rechten Seite von

$$p_1f_1 - p_2f_2 = p_1h_1 - p_2h_2$$

kleiner als p_1l_1 und wir können g mit weniger als r gleichen Anfangstermen darstellen:

$$g = (c_1 + c_2)p_2f_2 + c_1p_1h_1 - c_2p_2h_2 + \sum_{i>2} c_ip_if_i.$$

Im zweiten Fall haben wir

$$wl_1 = yl_2,\ p_1 = tw,$$

dann läßt sich nach Voraussetzung das S-Polynom

$$s = s(f_1, f_2) = wf_1 - yf_2 = \sum r_kf_k$$

durch Polynome $f_k \in F$ erzeugen, für die gilt

$$LM(r_kf_k) \leq LM(s) < wl_1 \leq p_1l_1,$$

also hat

$$g = c_1t\sum r_kf_k + c_1tp_2f_2 + \sum_{i>2} c_ip_if_i$$

weniger als r gleiche Anfangssummanden. □

Wir wollen eine Gröbner-Basis für die obigen Polynome berechnen: Wir setzen

$$g_4 = s(g_2,g_3) = -x^2yz + z^2,$$

können $g_1 = xg_4$ weglassen, das S-Polynom

$$s(g_2, g_4) = -x^2yz + z^2$$

kann mittel g_4 reduziert werden, wir erhalten

$$g_5 = yz^2 - z^2.$$

Folgende Eigenschaften der Lösungsmenge eines Gleichungssystems $F = 0$ kann man an der Gröbnerbasis G von $I = (F)$ ablesen:

- $F = 0$ und $G = 0$ haben dieselbe Lösungsmenge.
- $F = 0$ hat keine Lösung, wenn G ein konstantes Polynom enthält.
- $F = 0$ hat nur endlich viele Lösungen, wenn folgendes gilt: Bezüglich der lexikographischen Ordnung gibt es für jede Unbekannte x_i ein $f_i \in G$ mit $LM(f_i) = x_i^{n_i}$.

Beweis: Wir haben $f_1 = x_1^{n_1}$ + niedrigere Terme, in f_1 kommen keine x_i mit $i > 1$ vor, sonst wäre $x_1^{n_1}$ nicht das Leitmonom. Also ist f_1 ein Polynom nur in der Variablen x_1, hat also nur endlich viele Nullstellen. Weiter ist $f_2 = x_2^{n_2}$ + niedrigere Terme, in f_2 kommen also nur die Variablen x_1 und x_2 vor also hat $f_2 = 0$ für jede Nullstelle von f_1 auch nur endlich viele Lösungen.

Wir betrachten zwei einfache Spezialfälle:

Es seien $f_1(x), \ldots, f_m(x)$ Polynome in einer Variablen. Wenn

$$f_i = q \cdot f_j + r$$

ist, so ist das S-Polynom von f_i und f_j gerade gleich r. Wenn also $f = ggT(f_1, \ldots, f_m)$ ist, so ist $\{f\}$ eine Gröbnerbasis des von den f_i erzeugten Ideals.

Es seinen $f_i(x_1, \ldots, x_n)$ Polynome vom Grad 1. Wenn wir ihre Koeffizienten zeilenweise in eine Matrix schreiben, so entspricht der Bildung eines S-Polynoms gerade eine elementare Zeilenoperation.

Wir betrachten noch einige Anwendungen:

1. Es sei $S =< x_1, \ldots, x_n \mid f_i = g_j >$ eine endlich erzeugte kommutative Halbgruppe und $f, g \in S$. Wir fragen, ob aus den gegebenen Gleichungen die Gleichung $f = g$ folgt. Dazu bestimmen wir eine Gröbnerbasis des Ideals $(f_i - g_j)$ und vergleichen die Normalformen von f und g.

2. Rechnen in Faktorringen $K[x_i, \ldots, x_n]/I$: Wir bestimmen eine Gröbnerbasis des Ideals I. Dann haben wir eine Bijektion: Restklasse $f + I \mapsto NF(f)$ und $(f + I) + (g + I) \mapsto NF(f + g)$ sowie $(f + I) \cdot (g + I) \mapsto NF(f \cdot g)$.

3. Wir suchen in I ein univariates Polynom $f(x_i)$: Wir prüfen für $U_k = \{1, x_i, x_i^2, \ldots, x_i^k\}$, ob $\{NF(u) \mid u \in U_k\}$ linear abhängig ist; in diesem Fall existiert solch ein f und hat den Grad $k - 1$.

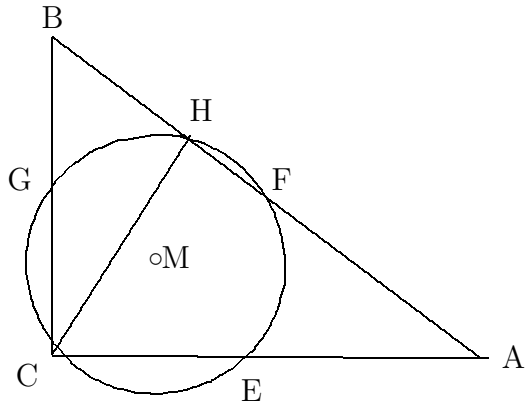
4. Sei F eine Gröbnerbasis des Ideals I; dann heißen die Variablen $x_1, \ldots, x_k$ unabhängig modulo F, wenn I kein Polynom enthält, das nur von $x_1, \ldots, x_k$ abhängt. Wenn $x_1 < x_2 < \ldots <$ Rest ist, so ist dies genau dann der Fall, wenn $F \cap K[x_1, \ldots, x_k] = \{0\}$ ist. Die Dimension der Nullstellenmenge von I ist dann $\geq k$.

   Beispiele:

$\{x - 1, y - 1\}$	0-dim.
$\{x^2 + y^2 - 1, xy\} \to \{y^3 - y\}$	0-dim.
$\{x^2 + x + 1, y2 + xy\}$	0-dim.
$\{x^2 + y^2 - 1\}$	1-dim.
$\{xy\}$	1-dim.
$\{x^2 + y^2 + z^2 - 1\}$	2-dim.

5. Es ist $f_1(x) = \ldots = f_k(x) = 0 \Rightarrow g(x) = 0$ genau dann, wenn $1 \in GB(F \cup \{y \cdot g - 1\})$.

6. **Appoloniuskreis** Wir zeigen: Im rechtwinkligen Dreieck liegen der Höhenfußpunkt und die Seitenmittelpunkte auf einem Kreis.



$A = (y_1, 0),\ B = (0, y_2),\ C = (0, 0),$
$E = (y_3, 0),\ F = (y_4, y_5),\ G = (0, y_6),\ H = (y_9, y_{10}),\ M = (y_7, y_8)$

$f_1 = 2y_3 - y_1 = 0$	E ist Mittelpunkt von CA
$f_2 = 2y_4 - y_1 = 0$ $f_3 = 2y_5 - y_2 = 0$	F ist Mittelpunkt von AB
$f_4 = 2y_6 - y_2 = 0$	G ist Mittelpunkt von BC
$f_5 = (y_7 - y_3)^2 + y_8^2 - (y_7 - y_4)2 - (y_8 - y_5)^2 = 0$	$EM = FM$
$f_6 = (y_7 - y_3)^2 + y_8^2 - (y_8 - y_6)2 - y_7^2 = 0$	$EM = GM$
$f_7 = (y_9 - y_1)y_2 + y_1 y_{10} = 0$	$H \in AB$
$f_8 = y_8 - y_1 y_9 + y_2 y_1 0 = 0$	$CH \perp AB$

Behauptung:

$$f = (y_7 - y_3)^2 + y_8^2 - (y_7 - y_9)^2 - (y_8 - y_{10})^2 = 0 \quad EM = HM$$

Es ist $1 \in GB(f_1, \ldots, f_8, y \cdot f - 1)$ zu prüfen.

12 Diskrete Fourier-Transformation

Wir benötigen hier einige Hilfsmittel aus der Algebrentheorie, die hier ohne Beweise zusammengestellt werden sollen.

Sei K ein Körper, A ein K-Vektorraum und gleichzeitig ein Ring; vermöge der Abbildung $K \longrightarrow A$ mit $k \mapsto k \cdot 1$ wird K in A eingebettet, wir identifizieren K mit $K \cdot 1$. Es gelte

$$k \cdot a = a \cdot k \text{ für alle } k \in K, a \in A.$$

Dann heißt A eine K-Algebra.

Sei M ein linker A-Modul, wegen $K \subseteq A$ operiert auch K auf M, d.h. M ist auch ein K-Vektorraum.

Wir vereinbaren, daß alle in diesem Abschnitt betrachteten Vektorräume endlichdimensional sind.
Sei A ein Ring und M ein linker A-Modul, M heißt einfach, wenn M keine echten Untermoduln besitzt. Ein Linksideal $L \subseteq A$ heißt minimal, wenn $\{0\}$ das einzige echt in L enthaltene Linksideal ist. Ein Linksideal L heißt maximal, wenn A das einzige L echt enthaltende Linksideal ist.

Satz 12.1 *1. Jedes minimale Linksideal ist ein einfacher A-Modul.*
2. Für jedes maximale Linksideal $L \subseteq A$ ist A/L ein einfacher A-Modul.
3. Jeder einfache A-Modul ist isomorph zu A/L für ein geeignetes maximales Linksideal $L \subseteq A$.

Eine K-Algebra A heißt einfache Algebra, wenn A genau zwei Ideale besitzt, nämlich $\{0\}$ und M.

Satz 12.2 (Wedderburn) *Jede einfache* $\mathbf{C}$*-Algebra A ist isomorph zu einer Matrixalgebra $M_{nn}(C)$.*

Sei A eine beliebige K-Algebra, ein A-Modul M heißt halbeinfach, wenn $M = M_1 \oplus \ldots \oplus M_k$ eine direkte Summe einfacher A-Moduln ist.

Zu direkten Zerlegungen $M = M_1 \oplus \ldots M_n$ gehören Endomorphismen $p_1, \ldots, p_n \in End(M)$, für die $p_i \circ p_j = p_i \delta_{ij}$ galt (man nennt solche Elemente „orthogonale Idempotente“) und es galt $M_i = p_i(M)$.
Wenn wir nun eine Zerlegung einer Algebra $A = L_1 \oplus \ldots \oplus L_n$ in Linksideale vornehmen, so entspricht dem die Existenz orthogonaler Idempotenter e_i in $End_R(R) = R$ und es gilt $L_i = Ae_i$.

Eine K-Algebra A heißt halbeinfach, wenn $A = L_1 \oplus \ldots \oplus L_n$ eine direkte Summe minimaler Linksideale ist.

Satz 12.3 *Sei $A = \oplus L_i$ eine halbeinfache Algebra, die L_i seien minimale Linksideale und $L_1 \cong \ldots \cong L_r$ und $L_1 \ncong L_i$ für $i > r$. Dann ist $I = L_1 \oplus \ldots \oplus L_r$ ein minimales zweiseitiges Ideal von A.*

Folgerung 12.4 *Sei A eine halbeinfache Algebra, dann ist A eine direkte Summe minimaler Ideale: $A = I_1 \oplus \ldots \oplus I_s$, jedes I_i ist eine direkte Summe paarweise isomorpher minimaler Linksideale.*

Satz 12.5 *Sei $A = I_1 \oplus \ldots \oplus I_s$ mit minimalen Idealen I_i. Dann gilt $I_i \cdot I_j = \{0\}$ für $i \neq j$ und jedes I_i ist eine einfache Algebra.*

Sei nun $G = \{g_1, \ldots, g_n\}$ eine endliche Gruppe. Mit

$$KG = \{\sum r_i g_i \mid r_i \in K\}$$

bezeichnen wir die Menge aller formaler Linearkombinationen der Elemente der Gruppe G, dies ist ein n-dimensionaler Vektorraum. Wir führen eine Multiplikation ein, die durch die Multiplikation in G induziert wird:

$$(\sum r_i g_i)(\sum s_j g_j) = \sum r_i s_j (g_i g_j),$$

diese erfüllt offenbar das Assoziativgesetz, die Körperelemente kommutieren mit allen Elementen und das Einselement von G ist das neutrale Element. Die Menge KG ist also eine K-Algebra, sie heißt die Gruppenalgebra der Gruppe G.

Satz 12.6 (Maschke) *Wenn die Zahl $\mid G \mid$ in K invertierbar ist, so ist KG eine halbeinfache Algebra.*

Die Gruppe G sei kommutativ, dann ist KG kommutativ. Wir betrachten den Spezialfall $K = \mathbf{C}$. Nach dem Satz von Maschke ist $\mathbf{C}G$ eine halbeinfache Algebra, wir haben gesehen, das halbeinfache Algebren sich als eine direkte Summe einfacher Algebren darstellen lassen:

$$\mathbf{C}G = A_1 \oplus \ldots \oplus A_m.$$

Nach dem Satz von Wedderburn ist jede einfache $\mathbf{C}$-Algebra isomorph zu einer Matrixalgebra:

$$A_i \cong M_{n_i n_i}(\mathbf{C}).$$

Für $n_i > 1$ ist diese Algebra nichtkommutativ, also müssen alle $n_i = 1$ sein, also $A_i \cong \mathbf{C}$. Insgesamt erhalten wir

$$\mathbf{C}G \cong \mathbf{C} \times \ldots \times \mathbf{C}.$$

Es sei $C_n = \{1, g, g^2, \ldots, g^{n-1}\}$ mit $g^n = 1$ die zyklische Gruppe mit n Elementen. Dann ist $\mathbf{C}C_n$ isomorph zur Faktoralgebra $\mathbf{C}[x]/(x^n - 1)$ des Polynomrings $\mathbf{C}[x]$. Die Multiplikation in ist relativ komplex, wenn das Produkt zweier Polynome zu berechnen ist, so sind etwa n^2 Multiplikationen von Körperelementen durchzuführen. Nach den obigen Resultaten ist aber

$$\mathbf{C}C_n \cong \mathbf{C} \times \ \ldots \times \mathbf{C}$$

und die Multiplikation in dieser Algebra geschieht komponentenweise, für eine Multiplikation von Algebra-Elementen benötigt man also nur n Mutiplikationen von Körperelementen. Es wäre schön, wenn wir den obigen Isomorphismus explizit kennen würden.

Dies ist möglich. Wir bestimmen nämlich die Idempotenten e_i mit

$$A_i = \mathbf{C}C_n e_i.$$

Lemma 12.7 *Sei G eine kommutative Gruppe und $f : G \longrightarrow \mathbf{C}\backslash\{0\}$ ein Homomorphismus von multiplikativen Gruppen, dann ist*

$$\frac{1}{\mid G \mid} \sum f(g_i) g_i \in \mathbf{C}G$$

idempotent.

Beweis:

$$\frac{1}{\mid G \mid}\sum f(g_i)g_i \cdot \frac{1}{\mid G \mid}\sum f(g_j)g_j = \frac{1}{\mid G \mid\mid G \mid}\sum f(g_ig_j)g_ig_j$$
$$= \frac{1}{\mid G \mid\mid G \mid}\sum f(g_i)g_i \cdot \mid G \mid = \frac{1}{\mid G \mid}\sum f(g_i)g_i.$$

Wenn speziell $G = C_n =< g >$ ist, so ist jeder Homomorphismus $f : C_n \longrightarrow \mathbf{C}\backslash\{0\}$ durch $f(g) \in \mathbf{C}$ bestimmt, wegen $g^n = 1$ muß $f(g)^n = 1$ sein, d.h. $f(g)$ ist eine n-te Einheitwurzel. Sei also ω eine primitive n-te Einheitwurzel, dann gibt es die folgenden n Homomorphismen $f_i : C_n \longrightarrow \mathbf{C}\backslash\{0\}$ mit $f_i(g) = \omega^i$. Also haben wir n Idempotente in der Gruppenalgebra:

$$e_i = \frac{1}{n}(1 + \omega^i g + \omega^{2i}g^2 + \omega^{(n-1)i}g^{n-1}), i = 0, \ldots, n-1.$$

Also hat der Isomorphismus

$$F^{-1} : \oplus \mathbf{C}C_n e_i \to \mathbf{C}C_n$$

bezüglich der Basen $\{e_1, \ldots, e_n\}$ und $\{1, g, \ldots, g^{n-1}\}$ die Darstellungsmatrix

$$\frac{1}{n}\begin{pmatrix} 1 & 1 & \cdots & 1 \\ 1 & \omega & \cdots & \omega^{n-1} \\ & & \cdots & \\ 1 & \omega^{n-1} & \cdots & \omega^{(n-1)(n-1)} \end{pmatrix}$$

der (eigentlich interessante) inverse Isomorphismus F hat die zu dieser inverse Darstellungsmatrix, diese hat die Gestalt

$$\begin{pmatrix} 1 & 1 & \cdots & 1 \\ 1 & \zeta & \cdots & \zeta^{n-1} \\ & & \cdots & \\ 1 & \zeta^{n-1} & \cdots & \zeta^{(n-1)(n-1)} \end{pmatrix}$$

mit $\zeta = \omega^{-1}$.