Einführung in die Computeralgebra

Wintersemester 2009/2010

Universität Bayreuth

Michael Stoll

Inhaltsverzeichnis

1. Einführung

Was ist Computeralgebra?

Die Computeralgebra ist Teil eines Gebiets, das man als *Wissenschaftliches Rechnen* bezeichnen kann. Dabei geht es darum, mit Hilfe des Computers mathematische Probleme zu lösen. Die dabei verwendeten Verfahren lassen sich grob einteilen in *numerische Algorithmen* und *symbolische Algorithmen*. Mit den numerischen Verfahren beschäftigt sich die *Numerische Mathematik*, mit den symbolischen zu großen Teilen die *Computeralgebra*. Die wesentlichen Unterschiede sind etwa die folgenden.

Bei den numerischen Verfahren sind die zu Grunde liegenden Objekte *kontinuierlicher* Natur (etwa Vektorfelder), die durch (oft sehr große Anzahlen an) *Gleitkommazahlen* im Computer *näherungsweise* dargestellt werden. Relevante Probleme bei der Konstruktion und der Untersuchung von Algorithmen sind *Konvergenz* (kommen wir bei entsprechend hohem Aufwand der wahren Lösung beliebig nahe?), Kontrolle der *Rundungsfehler*, die bei Rechnungen mit Gleitkommazahlen auftreten, und natürlich die *Effizienz* der Verfahren. Häufig sind *große Mengen* von Daten zu verarbeiten, an denen in gleicher Weise und vielfach hintereinander relativ einfache Berechnungen durchgeführt werden. Man denke etwa an die numerische Lösung eines Systems von partiellen Differentialgleichungen, etwa bei der Wettervorhersage.

Symbolische Verfahren dagegen rechnen *exakt*; die zu Grunde liegenden Objekte sind *algebraischer* und damit *diskreter* Natur, etwa Polynome in mehreren Variablen mit rationalen Zahlen als Koeffizienten. Diese Objekte können sehr *komplex* sein, und diese Komplexität muss durch geeignete Datenstrukturen im Computer abgebildet werden. Die verwendeten Algorithmen sind dementsprechend ebenfalls *komplex*, und das Hauptproblem liegt darin, *effiziente* Algorithmen und Datenstrukturen zu finden. Häufig beruhen diese auf höchst nichttrivialen Resultaten der Algebra und Zahlentheorie. Typische Aufgaben sind etwa die Faktorisierung von ganzen Zahlen (das RSA-Kryptosystem beruht darauf, dass dafür kein wirklich effizientes Verfahren bekannt ist) oder das Auffinden von rationalen Lösungen polynomialer Gleichungssysteme.

Man sollte allerdings auch erwähnen, dass es auch Mischformen gibt. Man kann etwa partielle Differentialgleichungen symbolisch „vorverarbeiten“, um sie in eine für numerische Algorithmen besser geeignete Form zu bringen. Auf der anderen Seite kann es effizienter sein, eine symbolische Rechnung (etwa mit ganzen Zahlen) numerisch, also näherungsweise, durchzuführen, wenn man dadurch das exakte Resultat hinreichend gut approximieren kann.

Als Beispiel betrachten wir folgendes Problem:

$$f(X) = X^6 - a_5X^5 + a_4X^4 - a_3X^3 + a_2X^2 - a_1X + a_0$$

sei ein Polynom mit ganzzahligen Koeffizienten. Wir bezeichnen die Nullstellen von f mit $\alpha_1, \alpha_2, \ldots, \alpha_6$. Wir wollen das Polynom $g(X)$ berechnen, dessen Nullstellen die zehn verschiedenen möglichen Ausdrücke der Form

$$\alpha_{i_1}\alpha_{i_2}\alpha_{i_3} + \alpha_{i_4}\alpha_{i_5}\alpha_{i_6}$$

sind, wo $(i_1, i_2, \ldots, i_6)$ eine Permutation von $(1, 2, \ldots, 6)$ ist. Die Theorie der symmetrischen Polynome sagt uns, dass die Koeffizienten von g Polynome in den a_j

mit ganzzahligen Koeffizienten sind. Explizit gilt

$$\begin{aligned}
g(X) = {} & X^{10} - a_3X^9 + (-9a_0 - a_1a_5 + a_2a_4)X^8 \\
& + (6a_0a_3 + a_0a_4a_5 + a_1a_2 + 2a_1a_3a_5 - a_1a_4^2 - a_2^2a_5)X^7 \\
& + (27a_0^2 + 9a_0a_1a_5 - 9a_0a_2a_4 + a_0a_2a_5^2 + 3a_0a_3^2 - 3a_0a_3a_4a_5 + a_0a_4^3 \\
& \quad + a_1^2a_4 - a_1^2a_5^2 - 3a_1a_2a_3 + a_1a_2a_4a_5 + a_2^3)X^6 \\
& + (-9a_0^2a_3 - 3a_0^2a_4a_5 - 2a_0^2a_5^3 - 3a_0a_1a_2 - 16a_0a_1a_3a_5 + 4a_0a_1a_4^2 \\
& \quad + 2a_0a_1a_4a_5^2 + 4a_0a_2^2a_5 + a_0a_2a_3a_4 + 2a_0a_2a_3a_5^2 - a_0a_2a_4^2a_5 - 2a_1^3 \\
& \quad + 2a_1^2a_2a_5 + 2a_1^2a_3a_4 - a_1^2a_3a_5^2 - a_1a_2^2a_4)X^5 \\
& + (-27a_0^3 - 30a_0^2a_1a_5 + 18a_0^2a_2a_4 - 2a_0^2a_2a_5^2 - 15a_0^2a_3^2 + 16a_0^2a_3a_4a_5 \\
& \quad + a_0^2a_3a_5^3 - 4a_0^2a_4^3 - a_0^2a_4^2a_5^2 - 2a_0a_1^2a_4 + 6a_0a_1^2a_5^2 + 16a_0a_1a_2a_3 \\
& \quad - 3a_0a_1a_2a_5^3 - a_0a_1a_3^2a_5 - 2a_0a_1a_3a_4^2 + a_0a_1a_3a_4a_5^2 - 4a_0a_2^3 - 2a_0a_2^2a_3a_5 \\
& \quad + a_0a_2^2a_4^2 + a_1^3a_3 - 3a_1^3a_4a_5 + a_1^3a_5^3 - a_1^2a_2^2 + a_1^2a_2a_3a_5)X^4 \\
& + (9a_0^3a_5^3 + 39a_0^2a_1a_3a_5 - 14a_0^2a_1a_4a_5^2 + a_0^2a_1a_5^4 - 11a_0^2a_2a_3a_5^2 + 2a_0^2a_2a_4a_5^3 \\
& \quad - 3a_0^2a_3^3 + 3a_0^2a_3^2a_4a_5 - a_0^2a_3^2a_5^3 + 9a_0a_1^3 - 14a_0a_1^2a_2a_5 - 11a_0a_1^2a_3a_4 \\
& \quad + 6a_0a_1^2a_3a_5^2 + 3a_0a_1^2a_4^2a_5 - a_0a_1^2a_4a_5^3 + 3a_0a_1a_2^2a_5^2 + 3a_0a_1a_2a_3^2 \\
& \quad - a_0a_1a_2a_3a_4a_5 + a_1^4a_5 + 2a_1^3a_2a_4 - a_1^3a_2a_5^2 - a_1^3a_3^2)X^3 \\
& + (36a_0^3a_1a_5 - 6a_0^3a_2a_5^2 + 18a_0^3a_3^2 - 24a_0^3a_3a_4a_5 - 4a_0^3a_3a_5^3 + 8a_0^3a_4^2a_5^2 \\
& \quad - a_0^3a_4a_5^4 - 6a_0^2a_1^2a_4 - 7a_0^2a_1^2a_5^2 - 24a_0^2a_1a_2a_3 - 4a_0^2a_1a_2a_4a_5 \\
& \quad + 10a_0^2a_1a_2a_5^3 + 8a_0^2a_1a_3^2a_5 + 8a_0^2a_1a_3a_4^2 - 8a_0^2a_1a_3a_4a_5^2 + a_0^2a_1a_3a_5^4 \\
& \quad + 8a_0^2a_2^2a_3a_5 - 2a_0^2a_2^2a_4a_5^2 - 2a_0^2a_2a_3^2a_4 + a_0^2a_2a_3^2a_5^2 - 4a_0a_1^3a_3 \\
& \quad + 10a_0a_1^3a_4a_5 - 4a_0a_1^3a_5^3 + 8a_0a_1^2a_2^2 - 8a_0a_1^2a_2a_3a_5 - 2a_0a_1^2a_2a_4^2 \\
& \quad + a_0a_1^2a_2a_4a_5^2 + a_0a_1^2a_3^2a_4 - a_1^4a_2 + a_1^4a_3a_5)X^2 \\
& + (-8a_0^4a_5^3 - 24a_0^3a_1a_3a_5 + 16a_0^3a_1a_4a_5^2 - 2a_0^3a_1a_5^4 + 8a_0^3a_2a_3a_5^2 - a_0^3a_2a_5^5 \\
& \quad + 8a_0^3a_3^3 - 8a_0^3a_3^2a_4a_5 + 3a_0^3a_3^2a_5^3 - 8a_0^2a_1^3 + 16a_0^2a_1^2a_2a_5 + 8a_0^2a_1^2a_3a_4 \\
& \quad - 6a_0^2a_1^2a_3a_5^2 - 8a_0^2a_1^2a_4^2a_5 + 3a_0^2a_1^2a_4a_5^3 - 8a_0^2a_1a_2^2a_5^2 - 8a_0^2a_1a_2a_3^2 \\
& \quad + 8a_0^2a_1a_2a_3a_4a_5 - a_0^2a_1a_2a_3a_5^3 - a_0^2a_1a_3^3a_5 - 2a_0a_1^4a_5 + 3a_0a_1^3a_2a_5^2 \\
& \quad + 3a_0a_1^3a_3^2 - a_0a_1^3a_3a_4a_5 - a_1^5a_4)X \\
& + 8a_0^4a_3a_5^3 - 4a_0^4a_4a_5^4 + a_0^4a_5^6 - 4a_0^3a_1a_2a_5^3 - 12a_0^3a_1a_3^2a_5 + 8a_0^3a_1a_3a_4a_5^2 \\
& \quad - 2a_0^3a_1a_3a_5^4 + a_0^3a_2^2a_5^4 - 2a_0^3a_2a_3^2a_5^2 + a_0^3a_3^4 + 8a_0^2a_1^3a_3 - 4a_0^2a_1^3a_4a_5 \\
& \quad + 2a_0^2a_1^3a_5^3 + 8a_0^2a_1^2a_2a_3a_5 - 2a_0^2a_1^2a_2a_4a_5^2 - 2a_0^2a_1^2a_3^2a_4 + a_0^2a_1^2a_3^2a_5^2 \\
& \quad - 4a_0a_1^4a_2 - 2a_0a_1^4a_3a_5 + a_0a_1^4a_4^2 + a_1^6
\end{aligned}$$

Eine rein symbolische Lösung wäre, in diesen Ausdruck die Werte der Koeffizienten des gegebenen Polynoms einzusetzen. Alternativ kann man die Nullstellen α_i als komplexe Zahlen näherungsweise berechnen, daraus die Nullstellen $\beta_1, \ldots, \beta_{10}$ von g bestimmen und dann $g(X)$ näherungsweise als

$$g(X) = (X - \beta_1)(X - \beta_2) \cdots (X - \beta_{10})$$

erhalten. Wenn die Näherung gut genug ist, bekommen wir die wahren Koeffizienten (die ja ganze Zahlen sind) durch Runden. Ein wesentlicher Unterschied zum in der Numerik Üblichen ist hier, dass die numerische Rechnung unter Umständen

mit sehr hoher Genauigkeit (also Anzahl an Nachkommastellen) durchgeführt werden muss, damit das Ergebnis nahe genug an der exakten Lösung liegt.

In dieser Vorlesung werden wir Fragestellungen und Methoden der Computeralgebra an Hand einiger Beispiele kennen lernen. Grundkenntnisse in Algebra (Theorie des euklidischen Rings $\mathbb{Z}$, Polynomringe, endliche Körper) werden vorausgesetzt.

Eine Anmerkung zur Organisation: Für Studierende der Mathematik-Studiengänge (Bachelor oder Master) ist dies eine vierstündige Vorlesung; für Studierende des Lehramts ist sie dreistündig. Dies ist den unterschiedlichen Anzahlen an Leistungspunkten geschuldet, die dafür jeweils vorgesehen sind. In der Praxis bedeutet das, dass einige Teile (insgesamt etwa ein Viertel) dieser Vorlesung für Lehramtsstudierende nicht für die Prüfung relevant sein werden. Welche das sein werden (tendenziell eher gegen Ende des Semesters) wird jeweils vorher angekündigt werden.

Zur Literatur: Ich werde hauptsächlich (aber nicht sklavisch) dem sehr schönen Buch [GG] von von zur Gathen und Gerhard folgen. Leider ist es relativ teuer. Die beiden deutschsprachigen Bücher [Ka] und [Ko], die erschwinglicher sind, sollten aber auch das relevante Material enthalten, wobei sich Anordnung und Stil natürlich unterscheiden.

Für Beispiele und Übungsaufgaben, die am Computer zu bearbeiten sind, werden wir das Computeralgebrasystem MAGMA verwenden, das z.B. im WAP-Pool zur Verfügung steht. Im Unterschied zu den besser bekannten Systemen Maple und Mathematica, die termorientiert arbeiten, basiert MAGMA auf algebraischen Strukturen. Das heißt insbesondere, dass jedes Objekt „weiß“, in welcher Struktur es zu Hause ist. Das ist zum Beispiel wichtig, um die Frage zu beantworten, ob eine gegebene Zahl ein Quadrat ist oder nicht:

```
$ magma
Magma V2.15-14    Sun Oct 18 2009 13:58:14 on linux-j92c [Seed = 3595017827]
Type ? for help.  Type <Ctrl>-D to quit.
> IsSquare(2);
false
> IsSquare(RealField()!2);
true 1.41421356237309504880168872421
> IsSquare(RealField()!-7);
false
> IsSquare(ComplexField()!-7);
true 2.64575131106459059050161575364*$.1
> IsSquare(pAdicField(2)!2);
false
> IsSquare(pAdicField(2)!-7);
true 49333 + O(2^19)
> IsSquare(FiniteField(5)!2);
false
> IsSquare(FiniteField(17)!2);
true 6
```

2. Grundlagen

Bevor wir in die Materie wirklich einsteigen können, müssen wir erst einmal eine Vorstellung davon haben, wie die grundlegenden algebraischen Objekte, nämlich ganze Zahlen und Polynome, im Computer dargestellt werden können, und wie wir die Komplexität bzw. Effizienz unserer Algorithmen messen können.

Computer arbeiten mit *Datenworten,* die aus einer gewissen festen Anzahl an *Bits* bestehen (heutzutage meist 64). Da wir häufig mit sehr großen ganzen Zahlen zu tun haben, reicht ein Wort im Allgemeinen nicht aus, eine ganze Zahl zu speichern. Man wird also ein *Array* von solchen Worten verwenden. Dann muss man zusätzlich noch wissen, wie lang das Array ist, und man muss das Vorzeichen in geeigneter Weise codieren. Mathematisch gesehen, schreiben wir eine ganze Zahl N als

$$N = (-1)^s \sum_{j=0}^{n-1} a_j B^j$$

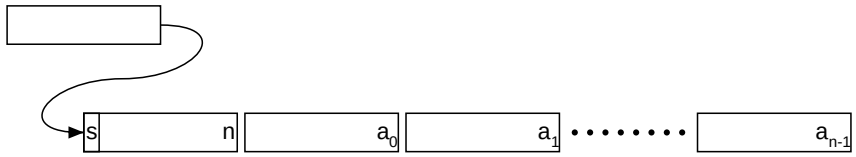
wobei $s \in \{0, 1\}$ und B die der Wortlänge entsprechende Basis ist; wir nehmen im Folgenden $B = 2^{64}$ an. Die „Ziffern“ a_j erfüllen $0 \le a_j < B$. Wenn wir $n < 2^{63}$ annehmen (was realistisch ist, denn größere Zahlen würden jeglichen Speicherplatz sprengen), dann lässt sich N im Computer darstellen als Folge

$$s \cdot 2^{63} + n, a_0, \ldots, a_{n-1}$$

von Worten. Es ist häufig sinnvoll, diese Darstellung zu *normalisieren* (also eindeutig zu machen), indem man fordert, dass $a_{n-1} \neq 0$ ist. Die Zahl 0 kann dann etwa durch das eine Wort 0 dargestellt werden (hat also $s = 0$ und $n = 0$). Die Zahl n ist in diesem Fall die *Wortlänge* $\lambda(N)$ von N; es gilt für $N \neq 0$

$$\lambda(N) = \left\lfloor \frac{\log |N|}{\log B} \right\rfloor + 1 \,.$$

Man verwendet einen *Zeiger* auf das erste Wort der Darstellung von N (also seine *Adresse*), um N im Computer zu repräsentieren:



Polynome in einer Variablen werden analog dargestellt:

$$f(X) = \sum_{j=0}^{n} a_j X^j \,,$$

wobei die *Koeffizienten* a_j aus einem Ring R kommen. Das erste Wort gibt wiederum die Länge ($n + 1$, wenn n der Grad ist) an, es folgen die Koeffizienten $a_0, \ldots, a_n$ als Zeiger auf die entsprechenden Datenstrukturen (etwa für ganze Zahlen wie oben). Man wird verlangen, dass $a_n \neq 0$ ist; das Nullpolynom wird wieder durch das Nullwort dargestellt.

Nachdem die Datenstrukturen geklärt sind, müssen die grundlegenden arithmetischen Operationen implementiert werden. Für ganze Zahlen sind dies etwa Vergleich, Negation, Addition, Multiplikation und Division mit Rest. Für Polynome bleibt vom Vergleich nur der Test auf Gleichheit übrig, und bei der Division mit

Rest nehmen wir an, dass der Divisor Leitkoeffizient 1 hat. (Da die Negation (Vorzeichenwechsel) im wesentlichen trivial ist, ist die Subtraktion als Negation des Subtrahenden plus Addition eingeschlossen.)

Geht man von normierten Darstellungen aus, dann sind zwei ganze Zahlen genau dann gleich, wenn ihre Darstellungen gleich sind (also gleiches Vorzeichen und gleiche Länge, und dann übereinstimmende „Ziffern“). Analog gilt für Polynome, dass sie genau dann gleich sind, wenn sie den selben Grad n haben und ihre Koeffizienten a_j jeweils gleich sind (wobei hier der Algorithmus zum Test der Gleichheit im Koeffizientenring R verwendet wird). Hier ist eine recht ausführliche Version des Vergleichsalgorithmus für ganze Zahlen. Die Syntax ist an MAGMA angelehnt. (Es fehlen die Strichpunkte zum Befehlsabschluss, und die MAGMA-Bezeichnungen eq, ne, gt, lt, ge, le für die Vergleichsoperatoren sind durch die üblichen $=$, $\neq$, $>$, $<$, $\geq$, $\leq$ ersetzt.)

function compare(M, N)
  // Vergleicht $M = (-1)^s \sum_{j=0}^{m-1} a_j B^j$ mit $N = (-1)^t \sum_{j=0}^{n-1} b_j B^j$.
  // Ausgabe: -1 falls $M < N$, 0 falls $M = N$, 1 falls $M > N$.
  if $s \neq t$ then return $(-1)^s$ end if
  // Ab hier haben M und N das selbe Vorzeichen.
  if $m > n$ then return $(-1)^s$ end if
  if $m < n$ then return $-(-1)^s$ end if
  // Ab hier gilt $m = n$.
  for $j = n - 1$ to 0 by -1 do
    if $a_j > b_j$ then return $(-1)^s$ end if
    if $a_j < b_j$ then return $-(-1)^s$ end if
  end for
  // Wenn wir hier ankommen, gilt $M = N$.
  return 0
end function

Der Additionsalgorithmus hat einen ähnlichen Aufbau. Je nach Vorzeichen müssen die Beträge addiert oder subtrahiert werden. Wir verwenden den Additionsbefehl des Prozessors, der für die Eingabeworte u und v und das Ausgabewort w der Relation

$$w + c' \cdot 2^{64} = u + v + c$$

entspricht, wobei $c, c' \in \{0, 1\}$ den Wert des *Carry-Flags* (Übertragsbit) des Prozessors vor und nach der Ausführung bezeichnet. Für die Subtraktion gibt es analog einen Befehl entsprechend der Relation

$$w - c' \cdot 2^{64} = u - v - c\,.$$

Wir verwenden $\leftarrow$ (anstelle von MAGMAs $:=$) für den Zuweisungsoperator.

function add(M, N)
  // Addiert $M = (-1)^s \sum_{j=0}^{m-1} a_j B^j$ und $N = (-1)^t \sum_{j=0}^{n-1} b_j B^j$.
  // Ausgabe: $M + N = (-1)^u \sum_{j=0}^{k-1} d_j B^j$.
  if $s = t$ then
    // M und N haben gleiches Vorzeichen: addieren.
    if $m < n$ then vertausche M und N end if
    // Ab hier ist $m \geq n$.
    $k \leftarrow m$ // Länge der Summe.
    $u \leftarrow s$ // Vorzeichen der Summe.

```
    c ← 0 // Initialisierung Übertrag.
    for j = 0 to n − 1 do
      d_j + c · B ← a_j + b_j + c // Addition mit Übertrag
    end for
    for j = n to m − 1 do
      d_j + c · B ← a_j + 0 + c // Addition mit Übertrag
    end for
    if c = 1 then
      // Ergebnis wird länger.
      d_k ← c
      k ← k + 1
    end if
  else // M und N haben verschiedenes Vorzeichen: subtrahieren.
    ...
  end if
  return (u, k, d_0, d_1, ..., d_{k−1})
end function
```

2.1. **Übung.** Ergänzen Sie im Pseudocode oben den Teil für die Subtraktion.

Achten Sie darauf, jedes Datenwort von M und N möglichst nur einmal „anzufassen“, und stellen Sie sicher, dass das Ergebnis normalisiert ist.

2.2. **Komplexität der Addition.** Was können wir über die Effizienz dieses Additionsalgorithmus sagen? Wir sind hauptsächlich daran interessiert, das Ergebnis *möglichst schnell* zu erhalten. Bei gegebenen Eingabedaten wird sich die Rechenzeit aber je nach Hardware und Implementation stark unterscheiden. Als Maß für die Komplexität besser geeignet ist die Art und Weise wie die Laufzeit von der Größe der Eingabe abhängt. Bei der Addition ist die Größe der Eingabe (gemessen in Worten) $\lambda(M) + \lambda(N)$. Der Prozessorbefehl zur Addition von Worten wird $\max\{\lambda(M), \lambda(N)\}$-mal ausgeführt. Dazu kommt eine konstante Anzahl an Prozessorbefehlen für die Abfragen zu Beginn, die Initialisierung und die Ausgabe. Außerdem braucht man pro Schleifendurchlauf noch eine konstante (oder jedenfalls beschränkte) Zahl von Befehlen (Heraufzählen von j, Vergleich mit dem Endwert, Adressrechnungen zum Zugriff auf a_j, b_j, d_j, ...). Insgesamt werden

$$f(M,N) \le a \max\{\lambda(M), \lambda(N)\} + b \le a\big(\lambda(M) + \lambda(N)\big) + b$$

Prozessorbefehle benötigt, wobei a und b vom verwendeten Prozessor (und dem Compiler etc.) abhängen. Um von solchen Details zu abstrahieren, schreiben wir

$$f(M,N) \ll \lambda(M) + \lambda(N)$$

oder

$$f(M,N) \in O\big(\lambda(M) + \lambda(N)\big)\,.$$

Beide Schreibweisen haben die gleiche Bedeutung, nämlich dass die linke Seite beschränkt ist durch eine Konstante mal die rechte Seite, jedenfalls wenn (hier) $\lambda(M) + \lambda(N)$ hinreichend groß ist.

Im vorliegenden Fall sagen wir auch, die Komplexität des Algorithmus sei *linear* (in der Größe der Eingabe). Es ist klar, dass *jeder* Algorithmus, der zwei ganze Zahlen in der hier beschriebenen Darstellung addiert, *mindestens* lineare Komplexität haben muss: Da jedes Wort der beiden zu addierenden Zahlen das Ergebnis

beeinflussen kann, muss die komplette Eingabe „angefasst“ werden. Es gilt also auch

$$f(M,N) \gg \lambda(M) + \lambda(N)\,.$$

Falls man sowohl obere als auch untere Abschätzungen hat, schreibt man auch

$$f(N) \asymp g(N) :\iff f(N) \gg g(N) \text{ und } f(N) \ll g(N)\,.$$

Für die Addition von Polynomen gilt Entsprechendes. Wir setzen die Definition

$$\sum_{i=0}^{n} a_i X^i + \sum_{i=0}^{n} b_i X^i = \sum_{i=0}^{n} (a_i + b_i) X^i$$

in einen Algorithmus um (dabei sei $a_i = 0$, falls i größer ist als der Grad des Polynoms, entsprechend für b_i). Wir haben also $n+1$ Additionen von Koeffizienten durchzuführen; die Komplexität ist also wiederum *linear*, wenn wir sie in Operationen im Koeffizientenring R messen. Die *Wortkomplexität* kann größer sein; sie hängt vom Typ der Koeffizienten ab. Sind die Koeffizienten ganze Zahlen vom Betrag $\leq M$, dann erhalten wir eine Wortkomplexität $\asymp n \log M$.

2.3. **Multiplikation.** Interessanter ist die *Multiplikation*. Die Schulmethode dafür (für Polynome und Zahlen in Dezimalschreibweise gleichermaßen) sieht so aus:

$(2x^2 - x + 1) \cdot (3x^2 - 2x + 1)$:

1	$2x^2 - x + 1$
$-2x$	$-4x^3 + 2x^2 - 2x$
$3x^2$	$6x^4 - 3x^3 + 3x^2$
	$6x^4 - 7x^3 + 7x^2 - 3x + 1$

$1234 \cdot 567$:

7	8638
60	7404
500	6170
	699678

Als Pseudocode für den etwas übersichtlicheren Fall der Polynome (wo es beim Addieren keinen Übertrag gibt) sieht das so aus:

```
function multiply(p, q)
  // Multipliziert die Polynome p = ∑_{i=0}^{m} a_i X^i und q = ∑_{i=0}^{n} b_i X^i.
  // Ausgabe: p · q = ∑_{i=0}^{m+n} c_i X^i.
  for i = 0 to m + n do c_i ← 0 end for // Initialisierung
  for j = 0 to m do
    for k = 0 to n do
      c_{j+k} ← c_{j+k} + a_j · b_k
    end for
  end for
  return (m + n, c_0, ..., c_{m+n})
end function
```

2.4. **Übung.** Schreiben Sie ein Programm (Pseudocode) für die Schulmethode fürs Multiplizieren zweier ganzer Zahlen. Für die Multiplikation stellt der Prozessor einen Befehl entsprechend der Relation

$$r + s \cdot B = u \cdot v$$

zur Verfügung, wobei u und v die Eingabeworte und r und s die Ausgabeworte sind.

2.5. **Komplexität der Multiplikation.** Wenn wir die Operationen im Koeffizientenring R zählen, die im obigen Multiplikationsalgorithmus ausgeführt werden, kommen wir auf

$$(m+1)(n+1) \text{ Multiplikationen} \quad \text{und} \quad (m+1)(n+1) \text{ Additionen.}$$

Dazu kommen noch $m+n+1$ Operationen, in denen ein Koeffizient auf null gesetzt wird. Die Komplexität der Schulmethode für die Multiplikation zweier Polynome vom Grad n ist also $\asymp n^2$: Das Verfahren hat *quadratische* Komplexität.

Für die Komplexität der Schulmethode für die Multiplikation zweier ganzer Zahlen M und N erhalten wir entsprechend $\asymp \lambda(M) \cdot \lambda(N)$.

Auf den ersten Blick scheint damit zur Multiplikation alles gesagt. Wir werden aber im Verlauf der Vorlesung sehen, dass es durchaus möglich ist, schneller zu multiplizieren.

2.6. **Übung.** Prüfen Sie experimentell, ob MAGMA große ganze Zahlen (oder Polynome großen Grades über einem endlichen Körper) mittels eines Algorithmus quadratischer Komplexität multipliziert, etwa indem Sie die Befehle

```
> N := 10^6;
> a := Random(2^(N-1), 2^N-1);
> time b := a*a;
```

verwenden (für verschiedene Werte von N).

2.7. **Division mit Rest.** Wir erinnern uns:

Ein Integritätsring R heißt *euklidisch* mit Normfunktion $N : R \to \mathbb{Z}_{\geq 0}$, wenn es zu $a, b \in R$ mit $b \neq 0$ stets $q, r \in R$ gibt, so dass

$$a = qb + r \quad \text{und} \quad N(r) < N(b)\,.$$

Im allgemeinen sind der *Quotient* q und der *Rest* r dabei nicht eindeutig bestimmt.

Die wichtigsten Beispiele von euklidischen Ringen sind $R = \mathbb{Z}$ und $R = k[X]$, wo k ein Körper ist. Für $R = \mathbb{Z}$ kann man $N(n) = |n|$ als Normfunktion benutzen, und für $R = k[X]$ setzt man $N(0) = 0$, $N(p) = 1 + \deg p$, wenn $p \neq 0$.

Wir beweisen das für den Polynomring.

2.8. **Satz.** *Sei k ein Körper, und seien $a, b \in k[X]$ mit $b \neq 0$. Dann gibt es eindeutig bestimmte $q, r \in k[X]$ mit*

$$a = qb + r \quad \textit{und} \quad \deg r < \deg b\,.$$

Beweis. Existenz: Wir betrachten b als fest und verwenden Induktion nach dem Grad von a. Für $\deg a < \deg b$ können wir $q = 0$, $r = a$ nehmen. Sei also jetzt $n = \deg a \geq \deg b = m$; wir schreiben

$$a = a_n X^n + a_{n-1} X^{n-1} \cdots + a_1 X + a_0, \quad b = b_m X^m + b_{m-1} X^{m-1} \cdots + b_1 X + b_0$$

mit $b_m \neq 0$. Dann ist

$$\begin{aligned}
\tilde{a} &= a - b_m^{-1} a_n X^{n-m} b \\
&= a_n X^n + a_{n-1} X^{n-1} + \cdots + a_0 - \left(a_n X^n + a_n b_{m-1} b_m^{-1} X^{n-1} + \cdots + a_n b_0 b_m^{-1} X^{n-m}\right) \\
&= (a_{n-1} - a_n b_{m-1} b_m^{-1}) X^{n-1} + \cdots
\end{aligned}$$

ein Polynom mit $\deg \tilde{a} < n = \deg a$. Nach Induktionsvoraussetzung gibt es also $\tilde{q}, r \in k[X]$ mit $\tilde{a} = \tilde{q}b + r$ und $\deg r < \deg b$. Dann gilt aber

$$a = \tilde{a} + b_m^{-1} a_n X^{n-m} b = (\tilde{q} + b_m^{-1} a_n X^{n-m})b + r\,,$$

und die Behauptung gilt mit $q = \tilde{q} + b_m^{-1} a_n X^{n-m}$.

Eindeutigkeit: Gilt $q_1 b + r_1 = q_2 b + r_2$ mit $\deg r_1, \deg r_2 < \deg b$, dann haben wir $(q_1 - q_2)b = r_2 - r_1$. Ist $r_1 \neq r_2$, dann ist der Grad der rechten Seite $< \deg b$, der Grad der linken Seite aber $\geq \deg b$. Also muss $r_1 = r_2$ und damit auch $q_1 = q_2$ sein. □

Aus diesem Beweis ergibt sich ziemlich unmittelbar der Schulalgorithmus zur Polynomdivision:

$$(3X^4 - 2X^2 + 3X - 1) : (X^2 - X + 1) = 3X^2 + 3X - 2 \quad \text{Rest } -2X + 1$$

	$3X^4 - 2X^2 + 3X - 1$
$3X^2$	$-3X^4 + 3X^3 - 3X^2$
	$3X^3 - 5X^2 + 3X - 1$
$3X$	$-3X^3 + 3X^2 - 3X$
	$-2X^2 - 1$
-2	$2X^2 - 2X + 2$
	$-2X + 1$

Für die Formulierung als Pseudocode nehmen wir an, dass b Leitkoeffizient 1 hat (dann funktioniert das Verfahren für beliebige Koeffizientenringe R).

```
function divide(a, b)
  // Dividiert a = Σ_{i=0}^{n} a_i X^i durch b = X^m + Σ_{i=0}^{m-1} b_i X^i.
  // Ausgabe: Quotient q = Σ_{i=0}^{d_q} q_i X^i und Rest r = Σ_{i=0}^{d_r} r_i X^i.
  if n < m then return (q, r) = (0, a) end if
  // Initialisierung
  d_q = n − m
  for i = 0 to n do r_i ← a_i end for
  // Die eigentliche Berechnung
  for j = n − m to 0 by −1 do
    q_j ← r_{m+j}
    // Setze r ← r − q_j X^j b
    for k = 0 to m − 1 do
      r_{j+k} ← r_{j+k} − q_j · b_k
    end for
  end for
  // Normalisiere r
  for i = m − 1 to 0 by −1 do
    if r_i ≠ 0 then
      d_r ← i;
      return (q, r) = ((d_q, q_0, ..., q_{d_q}), (d_r, r_0, ..., r_{d_r}))
    end if
  end for
  // Wenn wir hierher kommen, ist r = 0
  return (q, r) = ((d_q, q_0, ..., q_{d_q}), 0)
end function
```

2.9. **Komplexität der Division.** Wir nehmen an, dass $n \geq m$ ist (sonst ist im wesentlichen nichts zu tun). Dann ist die Anzahl an Operationen in R, die ausgeführt werden,

$$m(n-m+1) \text{ Multiplikationen und Additionen}$$

(wir unterscheiden nicht zwischen Addition und Subtraktion). Für die Division eines Polynoms vom Grad $2n$ durch eines vom Grad n ist der Aufwand also wiederum $\asymp n^2$, also *quadratisch*.

2.10. **Division von ganzen Zahlen.** Für den Ring $\mathbb{Z}$ gilt:

2.11. **Satz.** *Seien $a, b \in \mathbb{Z}$ mit $b \neq 0$. Dann gibt es eindeutig bestimmte $q, r \in \mathbb{Z}$ mit*

$$a = qb + r \quad \textit{und} \quad 0 \leq r < |b| .$$

Beweis. Wir können annehmen, dass b positiv ist.

Existenz: Klar (mit $q = 0, r = a$) für $0 \leq a < b$. Ist $a \geq b$, dann ist $a - b < a$, und mit Induktion gibt es q_1 und r mit $a - b = q_1 b + r$, $0 \leq r < b$. Dann ist $a = qb + r$ mit $q = q_1 + 1$. Ist $a < 0$, dann ist $a + b > a$, und mit Induktion gibt es q_1 und r mit $a + b = q_1 b + r$, $0 \leq r < b$. Dann ist $q = qb + r$ mit $q = q_1 - 1$.

Eindeutigkeit: Aus $q_1 b + r_1 = q_2 b + r_2$ und $0 \leq r_1, r_2 < b$ folgt $(q_1 - q_2)b = r_2 - r_1$, mit rechter Seite $< b$ und durch b teilbarer linker Seite. Es folgt, dass beide Seiten verschwinden. □

Der Algorithmus, der sich aus diesem Beweis ergibt:

```
function divide(a, b)
  // Dividiert a ≥ 0 durch b > 0.
  // Ausgabe: (q, r) mit a = qb + r, 0 ≤ r < b.
  q ← 0; r ← a
  while r ≥ b do
    q ← q + 1; r ← r − b
  end while
  return (q, r)
end function
```

ist sehr ineffizient.

Die Schulmethode für die Division funktioniert ähnlich wie die Polynomdivision: Um 123456 durch 789 zu teilen, rechnen wir

	1 2 3 4 5 6
100	− 7 8 9
	4 4 5 5 6
50	− 3 9 4 5
	5 1 0 6
6	− 4 7 3 4
	3 7 2

und erhalten den Quotienten 156 und den Rest 372. Die Ziffern des Quotienten rät man, indem man die führenden ein bis zwei Ziffern des bisherigen Restes durch die führende Ziffer des Dividenden teilt (hier $12 : 7 = 1$, $44 : 7 = 6$, $51 : 7 = 7$) und dann evtl. korrigiert, falls diese Schätzung zu hoch ausfällt. Mit Hilfe eines

geeigneten Divisionsbefehls des Prozessors, der etwa zu gegebenen Worten a_0, a_1, b mit $b > 0$ und $a_1 < b$ die Worte q und r bestimmt mit $a_0 + a_1 B = qb + r$, $0 \le r < b$, kann man diese Schulmethode implementieren, bei einer Komplexität, die mit der des Polynomdivisionsalgorithmus vergleichbar ist, also $\asymp n^2$ für die Division von a durch b mit $\lambda(a) = 2n$, $\lambda(b) = n$. Allerdings sieht man hier wieder, dass Langzahlarithmetik unangenehm komplizierter sein kann als Polynomarithmetik. Eine wirklich effiziente Umsetzung erfordert einiges an Hirnschmalz!

2.12. **Übung*.** Formulieren Sie die Schulmethode für die Division mit Rest (mit $a \ge 0$ und $b > 0$) als Pseudocode für die von uns gewählte Darstellung ganzer Zahlen.

2.13. **Definition.** Ist R ein euklidischer Ring, und sind $a, b \in R$ mit $b \neq 0$, so schreiben wir $q = a$ quo b, $r = a$ rem b, wenn $q, r \in R$ Quotient und Rest bei Division von a durch b sind. Dabei nehmen wir an, dass q und r durch geeignete Festlegungen eindeutig gemacht werden.

In vielen Computeralgebrasystemen (auch in MAGMA) sind „div“ und „mod“ statt „quo“ und „rem“ gebräuchlich. Wir wollen hier Missverständnisse vermeiden, die durch die verschiedenen anderen möglichen Bedeutungen von „mod“ auftreten können.

3. Der Euklidische Algorithmus

Der Euklidische Algorithmus dient zunächst einmal dazu, größte gemeinsame Teiler zu berechnen. Wir erinnern uns:

3.1. **Definition.** Sei R ein Integritätsring, $a, b \in R$. Ein Element $d \in R$ heißt ein *größter gemeinsamer Teiler* (ggT) von a und b, wenn $d \mid a$ und $d \mid b$, und wenn für jeden weiteren gemeinsamen Teiler d' von a und b gilt $d' \mid d$.

Ein Element $k \in R$ heißt *kleinstes gemeinsames Vielfaches* (kgV) von a und b, wenn $a \mid k$ und $b \mid k$, und wenn für jedes weitere gemeinsame Vielfache k' von a und b gilt $k \mid k'$.

3.2. **Definition.** Sei R ein Integritätsring. Zwei Elemente $a, b \in R$ heißen *assoziiert*, $a \sim b$, wenn $a \mid b$ und $b \mid a$. Gilt $u \sim 1$, so ist u eine *Einheit*. Die Einheiten sind genau die invertierbaren Elemente von R; sie bilden die multiplikative Gruppe $R^\times$.

Aus der Definition oben folgt, dass je zwei größte gemeinsame Teiler von a und b assoziiert sind. Umgekehrt gilt: Ist d ein ggT von a und b, und ist $d \sim d'$, so ist d' ebenfalls ein ggT von a und b. Analoge Aussagen gelten für kleinste gemeinsame Vielfache.

3.3. **Satz.** *In einem euklidischen Ring R haben je zwei Elemente a und b stets größte gemeinsame Teiler.*

Beweis. Der Beweis ergibt sich aus dem klassischen Euklidischen Algorithmus:

```
function gcd(a, b)
  // a, b ∈ R. Ausgabe: Ein ggT von a und b.
  while b ≠ 0 do (a, b) ← (b, a rem b) end while
  return a
end function
```

Sei N eine Normfunktion auf R. Dann wird $N(b)$ bei jedem Durchgang durch die while-Schleife kleiner, also muss b schließlich null werden, und der Algorithmus terminiert.

Es ist klar, dass

$$d \mid a \quad \text{und} \quad d \mid b \iff d \mid b \quad \text{und} \quad d \mid a \operatorname{rem} b$$

gilt. Die Menge der gemeinsamen Teiler von a und b bleibt also stets gleich. Da außerdem klar ist, dass a und 0 den größten gemeinsamen Teiler a haben, folgt, dass obiger Algorithmus tatsächlich einen ggT von a und b liefert. Insbesondere existiert ein ggT. □

3.4. **Bemerkung.** Jeder euklidische Ring ist ein Hauptidealring, und alle Aussagen über ggTs, die für euklidische Ringe richtig sind, stimmen auch für Hauptidealringe. Allerdings haben wir in einem beliebigen Hauptidealring keinen *Algorithmus* zur Verfügung, um ggTs zu berechnen.

3.5. **Beispiel.**

$$\operatorname{ggT}(299, 221) = \operatorname{ggT}(221, 78) = \operatorname{ggT}(78, 65) = \operatorname{ggT}(65, 13) = \operatorname{ggT}(13, 0) = 13\,.$$

3.6. **Lemma.** *Sei R ein euklidischer Ring. Dann gilt für $a, b, c \in R$ (wobei „ggT“ einen beliebigen größten gemeinsamen Teiler bezeichnet):*

(1) $a \mid b \iff \operatorname{ggT}(a, b) \sim a$;
(2) $\operatorname{ggT}(a, 0) \sim \operatorname{ggT}(a, a) \sim a, \quad \operatorname{ggT}(a, 1) \sim 1$;
(3) $\operatorname{ggT}(a, b) \sim \operatorname{ggT}(b, a)$ *(Kommutativität);*
(4) $\operatorname{ggT}\big(\operatorname{ggT}(a, b), c\big) \sim \operatorname{ggT}\big(a, \operatorname{ggT}(b, c)\big)$ *(Assoziativität);*
(5) $\operatorname{ggT}(ac, bc) \sim \operatorname{ggT}(a, b) \cdot c$ *(Distributivität);*
(6) $a \sim b \implies \operatorname{ggT}(a, c) \sim \operatorname{ggT}(b, c)$;
(7) $\operatorname{ggT}(a, b) \sim \operatorname{ggT}(a + bc, b)$.

Beweis. Standard, bzw. Übung. □

Im Hinblick auf die Assoziativität ist es sinnvoll, einfach $\operatorname{ggT}(a_1, a_2, \ldots, a_n)$ zu schreiben anstatt $\operatorname{ggT}\big(\ldots \operatorname{ggT}(a_1, a_2), \ldots, a_n\big)$.

3.7. **Satz.** *Sei R ein euklidischer Ring, $a, b \in R$, und $d \in R$ ein größter gemeinsamer Teiler von a und b. Dann gibt es $s, t \in R$ mit $d = sa + tb$.*

Beweis. Hierfür betrachten wir den *Erweiterten Euklidischen Algorithmus.*

```
function xgcd(a, b)
  // a, b ∈ R.
  // Ausgabe: d, s, t ∈ R mit d = ggT(a, b), d = sa + tb.
  // (Oder auch: ℓ ≥ 0, r_i, s_i, t_i ∈ R für 0 ≤ i ≤ ℓ + 1, q_i ∈ R für 1 ≤ i ≤ ℓ,
  //     mit d = r_ℓ, s = s_ℓ, t = t_ℓ).
  (r_0, s_0, t_0) ← (a, 1, 0)
  (r_1, s_1, t_1) ← (b, 0, 1)
  i ← 1
  while r_i ≠ 0 do
    q_i ← r_{i−1} quo r_i
    (r_{i+1}, s_{i+1}, t_{i+1}) ← (r_{i−1} − q_i r_i, s_{i−1} − q_i s_i, t_{i−1} − q_i t_i)
```

```
    i ← i + 1
  end while
  ℓ ← i − 1
  return (r_ℓ, s_ℓ, t_ℓ)
end function
```

Dass der Algorithmus terminiert, folgt wie in Satz 3.3. Außerdem prüft man nach, dass folgende Aussagen vor und nach jedem Schleifendurchlauf gelten:

$$\operatorname{ggT}(a,b) \sim \operatorname{ggT}(r_{i-1}, r_i)\,, \quad r_i = s_i a + t_i b\,.$$

Am Ende ist $r_{\ell+1} = 0$, $r_\ell \sim \operatorname{ggT}(a,b)$, und die Behauptung ist klar. □

3.8. **Beispiel.** Mit $a = 299$ und $b = 221$ wie oben haben wir:

i	q_i	r_i	s_i	t_i
0		299	1	0
1	1	221	0	1
2	2	78	1	−1
3	1	65	−2	3
4	5	13	3	−4
5		0	−17	23

Es folgt $\operatorname{ggT}(299, 221) = 13 = 3 \cdot 299 - 4 \cdot 221$.

3.9. **Beispiel.** Im Polynomring $\mathbb{Q}[X]$ berechnen wir mit

$$a = 4X^4 - 12X^3 - X^2 + 15X \quad \text{und} \quad b = 12X^2 - 24X - 15$$

folgende Tabelle:

i	q_i	r_i	s_i	t_i
0		$4X^4 - 12X^3 - X^2 + 15X$	1	0
1	$\frac{1}{3}X^2 - \frac{1}{3}X - \frac{1}{3}$	$12X^2 - 24X - 15$	0	1
2	$6X + 3$	$2X - 5$	1	$-\frac{1}{3}X^2 + \frac{1}{3}X + \frac{1}{3}$
3		0	$-6X - 3$	$2X^3 - X^2 - 3X$

Also ist

$$2X - 5 = a + \left(-\tfrac{1}{3}X^2 + \tfrac{1}{3}X + \tfrac{1}{3}\right)b$$

ein ggT von a und b.

3.10. **Komplexität für Polynome.** Wir betrachten hier zunächst den klassischen Algorithmus wie in Satz 3.3, und zwar für Polynome über einem Körper K. Für eine allgemeine Polynomdivision mit Rest eines Polynoms vom Grad n durch eines vom Grad $m \le n$ brauchen wir $2m(n-m+1)$ Operationen in K, plus eine Inversion und $n - m + 1$ weitere Multiplikationen (zur Berechnung von b_m^{-1} und für die richtige Skalierung des Quotienten). Insgesamt also

$$(2m+1)(n-m+1) + 1 = 2m(n-m) + n + m + 2$$

Operationen in K.

Sei $n_i = \deg r_i$ im Algorithmus von Satz 3.7. Dann gilt $\deg q_i = n_{i-1} - n_i$ und $n = \deg a = n_0$, $m = \deg b = n_1$. Der Aufwand ist

$$s(n_0, n_1, \ldots, n_\ell) = \sum_{i=1}^{\ell} \big(2n_i(n_{i-1} - n_i) + n_{i-1} + n_i + 2\big)\,.$$

Wir betrachten zunächst den *Normalfall*, dass

$$n_i = m - i + 1 \quad \text{für } 2 \le i \le \ell = m + 1.$$

Die Grade der sukzessiven Reste r_i werden also immer um eins kleiner. Dann haben wir

$$\begin{aligned} s(n, m, m-1, m-2, \dots, 1) &= \big(2m(n-m) + n + m + 2\big) + \sum_{i=2}^{m+1} (4m - 4i + 7) \\ &= 2m(n-m) + n + m + 2 + m(2m+1) \\ &= 2nm + n + 2m + 2 \le 2(n+1)(m+1) \end{aligned}$$

Operationen in K. Die Berechnung ist also etwa so teuer wie die Multiplikation von a und b. (In beiden Fällen teilen sich die Operationen etwa hälftig in Additionen und Multiplikationen auf. Beim ggT kommen noch $m+1$ Inversionen hinzu.)

Für den allgemeinen Fall zeigen wir, dass s größer wird, wenn wir irgendwo in der Folge $n_1, \dots, n_\ell$ ein zusätzliches Glied einschieben oder am Ende ein Glied anfügen. Es ist

$$\begin{aligned} &s(n_0, \dots, n_{j-1}, k, n_j, \dots, n_\ell) - s(n_0, \dots, n_{j-1}, n_j, \dots, n_\ell) \\ &\qquad = \big(2k(n_{j-1} - k) + n_{j-1} + k + 2\big) + \big(2n_j(k - n_j) + k + n_j + 2\big) \\ &\qquad\qquad - \big(2n_j(n_{j-1} - n_j) + n_{j-1} + n_j + 2\big) \\ &\qquad = 2(n_{j-1} - k)(k - n_j) + 2k + 2 > 0 \end{aligned}$$

und

$$s(n_0, n_1, \dots, n_\ell, k) - s(n_0, n_1, \dots, n_\ell) = 2k(n_\ell - k) + n_\ell + k + 2 > 0\,.$$

Damit ist der Normalfall der teuerste, und die Abschätzung, dass der Aufwand $\le 2(n+1)(m+1)$ ist, gilt allgemein.

Man kann auf ähnliche Weise zeigen, dass zur Berechnung der t_i bzw. der s_i höchstens

$$2(2n - m)m + O(n) \quad \text{bzw.} \quad 2m^2 + O(m)$$

Operationen in K benötigt werden. Für die komplette Berechnung aller Terme im Erweiterten Euklidischen Algorithmus kommt man also mit $6mn + O(n)$ Operationen aus. Benötigt man nur den ggT und $t = t_\ell$, dann reichen $4n^2 + O(n)$ Operationen (wobei man die Berechnung der s_i weglässt).

3.11. **Komplexität für ganze Zahlen.** Eine gute Abschätzung der Anzahl der Schleifendurchläufe (wie sie für Polynome durch $\deg b + 1$ gegeben wird) ist hier nicht so offensichtlich. Eine Möglichkeit ist, sich zu Nutze zu machen, dass

$$r_{i-1} = q_i r_i + r_{i+1} \ge r_i + r_{i+1} > 2r_{i+1}\,.$$

Nach jeweils zwei Schritten hat sich der Rest also mindestens halbiert; damit ist die Anzahl der Schritte höchstens $2\log_2 |b| + O(1) = 128\lambda(b) + O(1)$. Man kann dann — analog wie für Polynome — schließen, dass die Wortkomplexität für den Euklidischen Algorithmus (klassisch oder erweitert) $\ll \lambda(a)\lambda(b)$ ist. Die Größenordnung entspricht wieder der für die (klassische) Multiplikation.

Eine bessere Abschätzung (um einen konstanten Faktor) für die Anzahl der Divisionen bekommt man, indem man sich überlegt, dass der „worst case“ eintritt, wenn alle Quotienten $q_i = 1$ sind. Dann sind a und b aufeinanderfolgende

Fibonacci-Zahlen F_{n+1} und F_n. Aus der Formel

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

und $|(1-\sqrt{5})/2| < 1$ ergibt sich für die Anzahl der Divisionen

$$n \in \frac{\log|b|}{\log\frac{1+\sqrt{5}}{2}} + O(1) \approx 2,078\log|b| + O(1)$$

(der „log“ in [GG] ist zur Basis 2, nicht der natürliche wie hier!), während die vorige Abschätzung

$$n \in 2\frac{\log|b|}{\log 2} + O(1) \approx 2,885\log|b| + O(1)$$

liefert.

4. Normalisierung des ggT

Der ggT ist nur bis auf Assoziiertheit bestimmt. Für Anwendungen ist es aber von Vorteil, eine wohldefinierte Funktion ggT zu haben. Wir müssen also aus den verschiedenen möglichen ggTs einen in geeigneter Weise auswählen.

Dazu brauchen wir eine Funktion $a \mapsto \mathrm{normal}(a)$ (für a im betrachteten euklidischen Ring R) mit folgenden Eigenschaften:

(1) $a \sim \mathrm{normal}(a)$;
(2) $a \sim b \iff \mathrm{normal}(a) = \mathrm{normal}(b)$;
(3) $\mathrm{normal}(ab) = \mathrm{normal}(a)\,\mathrm{normal}(b)$.

Für $a \neq 0$ gilt dann $a = \mathrm{lu}(a)\,\mathrm{normal}(a)$ mit einer Einheit $\mathrm{lu}(a) \in R^\times$ („leading unit“). Wir definieren noch $\mathrm{lu}(0) = 1$; $\mathrm{normal}(0) = 0$ ist klar. Natürlich verlangen wir auch, dass $\mathrm{lu}(a)$ (und damit $\mathrm{normal}(a) = \mathrm{lu}(a)^{-1}a$) berechenbar sind. Ein Element der Form $\mathrm{normal}(a)$ heißt *normalisiert* bzw. die *Normalform* von a.

4.1. **Beispiele.**

(1) Für $R = \mathbb{Z}$ setzt man $\mathrm{normal}(a) = |a|$.
(2) Für $R = K[X]$ setzt man $\mathrm{lu}(f) = \mathrm{lcf}(f)$; normalisiert sind also genau die Polynome mit Leitkoeffizient 1 (und das Nullpolynom).

4.2. **Beispiel.** Man kann in jedem euklidischen (sogar in jedem faktoriellen) Ring eine Normalform definieren, indem man aus jeder Assoziiertheitsklasse von Primelementen eines als normalisiert auswählt; Normalformen sind dann gerade alle Produkte von solchen Primelementen. (Ob sich die Normalform dann effizient bestimmen lässt, ist eine andere Frage.)

Es ist allerdings nicht immer möglich, kompatible Normalformen zu haben, wenn ein Ring in einem anderen enthalten ist. Zum Beispiel ist der Ring $\mathbb{Z}[i]$ der ganzen Gaußschen Zahlen euklidisch (mit Normfunktion $N(a+bi) = a^2+b^2$). Die Assoziierten von $1+i$ sind alle Elemente der Form $\pm1 \pm i$. Eines davon muss normalisiert sein. Nun gilt $(\pm1 \pm i)^4 = -4$, also ist -4 ebenfalls normalisiert. Auf der anderen Seite sind die Assoziierten von 2 in $\mathbb{Z}$ die Elemente ±2, und $(\pm2)^2 = 4$ muss normalisiert sein. Es gibt also keine Normalform auf $\mathbb{Z}[i]$, die eine Normalform auf $\mathbb{Z}$ fortsetzt.

4.3. **Definition.** Im gegebenen euklidischen Ring R sei eine Normalform festgelegt. Dann definieren wir die Funktionen ggT und kgV in der Weise, dass $\mathrm{ggT}(a,b)$ (bzw. $\mathrm{kgV}(a,b)$) der eindeutig bestimmte *normalisierte* größte gemeinsame Teiler (bzw. das eindeutig bestimmte *normalisierte* kleinste gemeinsame Vielfache) von a und b ist.

Man kann dann den (Erweiterten) Euklidischen Algorithmus so anpassen, dass er am Ende das Ergebnis normalisiert (und ggfs. die Koeffizienten s und t anpasst). Meistens ist es aber vorteilhafter, schon im Algorithmus dafür zu sorgen, dass die sukzessiven Reste normalisiert sind. Man erhält dann folgende Version:

```
function xgcd(a, b)
  // a, b ∈ R.
  // Ausgabe: d, s, t ∈ R mit d = ggT(a, b), d = sa + tb.
  // (Oder auch: ℓ ≥ 0, r_i, s_i, t_i ∈ R für 0 ≤ i ≤ ℓ + 1, q_i ∈ R für 1 ≤ i ≤ ℓ,
  //    mit d = r_ℓ, s = s_ℓ, t = t_ℓ).
  ρ_0 ← lu(a)^{-1}; ρ_1 ← lu(b)^{-1}
  (r_0, s_0, t_0) ← (ρ_0 · a, ρ_0, 0)
  (r_1, s_1, t_1) ← (ρ_1 · b, 0, ρ_1)
  i ← 1
  while r_i ≠ 0 do
    q_i ← r_{i-1} quo r_i, r̃ ← r_{i-1} rem r_i // das ist eine Berechnung
    ρ_{i+1} ← lu(r̃)^{-1}
    (r_{i+1}, s_{i+1}, t_{i+1}) ← (ρ_{i+1} · r̃, ρ_{i+1} · (s_{i-1} − q_i s_i), ρ_{i+1} · (t_{i-1} − q_i t_i))
    i ← i + 1
  end while
  ℓ ← i − 1
  return (r_ℓ, s_ℓ, t_ℓ)
end function
```

Wenn nur der ggT benötigt wird, kann man die Teile, die der Berechnung der s_i und t_i dienen, weglassen.

Zum Beispiel stellt sich heraus, dass mit dieser Version des Algorithmus die Koeffizienten der im Verlauf der Rechnung auftretenden Polynome deutlich weniger stark wachsen als mit der Version ohne Normalisierung, wenn man ihn in $\mathbb{Q}[X]$ anwendet.

5. Anwendungen des Euklidischen Algorithmus

5.1. **Quotientenkörper.** Jeder Integritätsring R hat einen Quotientenkörper K. Seine Elemente sind Brüche $\frac{a}{b}$ mit $a, b \in R$, $b \neq 0$. Diese Darstellung ist nicht eindeutig, denn es gilt $\frac{a}{b} = \frac{ac}{bc}$.

Ist R ein euklidischer Ring (oder allgemeiner ein faktorieller Ring), dann gelangen wir zu einer eindeutigen Darstellung, indem wir verlangen, dass

$$\mathrm{ggT}(a,b) = 1 \quad \text{und} \quad \mathrm{lu}(b) = 1$$

gilt (letzteres heißt, dass b normalisiert ist).

Das führt zu folgendem Algorithmus, der aus einer beliebigen Darstellung (a,b) die normalisierte Darstellung berechnet.

function reduce(a, b)
  // $a, b \in R$, $b \neq 0$.
  // Berechnet $a', b' \in R$ mit $\frac{a}{b} = \frac{a'}{b'}$, $\mathrm{ggT}(a', b') = 1$ und $\mathrm{lu}(b') = 1$.
  $g \leftarrow \gcd(a, b)$
  $a' \leftarrow a$ quo g; $b' \leftarrow b$ quo g // Division ohne Rest
  $\rho \leftarrow \mathrm{lu}(b')^{-1}$
  $a' \leftarrow \rho \cdot a'$; $b' \leftarrow \rho \cdot b'$
  return (a', b')
end function

Für die Multiplikation und Addition in K hat man die bekannten Formeln

$$\frac{a}{b} \cdot \frac{c}{d} = \frac{a \cdot c}{b \cdot d} \quad \text{und} \quad \frac{a}{b} + \frac{c}{d} = \frac{a \cdot d + b \cdot c}{b \cdot d}\,.$$

Daraus kann man direkt Algorithmen ableiten. Es ist allerdings besser, nur da größte gemeinsame Teiler zu berechnen, wo tatsächlich möglicherweise etwas gekürzt werden kann. (Wir nehmen natürlich an, dass $\frac{a}{b}$ und $\frac{c}{d}$ in Normalform vorliegen.)

Bei der Multiplikation kann es gemeinsame Teiler von b und c oder von a und d geben. Wir können die Berechnung von $\mathrm{ggT}(ac, bd)$ also ersetzen durch die Berechnung von $\mathrm{ggT}(a, d)$ und $\mathrm{ggT}(b, c)$, was etwas billiger ist:

function multiply($\frac{a}{b}$, $\frac{c}{d}$)
  // $\frac{a}{b}, \frac{c}{d} \in K$ in Normalform.
  // Berechnet $\frac{z}{n} = \frac{a}{b} \cdot \frac{c}{d}$ (in Normalform).
  $g_1 \leftarrow \gcd(a, d)$
  $g_2 \leftarrow \gcd(b, c)$
  $z \leftarrow (a$ quo $g_1) \cdot (c$ quo $g_2)$ // Division ohne Rest
  $n \leftarrow (b$ quo $g_2) \cdot (d$ quo $g_1)$ // Division ohne Rest
  return $\frac{z}{n}$
end function

Bei der Addition genügt es zunächst einmal, die Brüche so zu erweitern, dass im Nenner das kgV von b und d steht (statt des Produkts). Mit

$$b' = b/\,\mathrm{ggT}(b, d) = \mathrm{kgV}(b, d)/d \quad \text{und} \quad d' = d/\,\mathrm{ggT}(b, d) = \mathrm{kgV}(b, d)/b$$

ist dann

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

Außerdem gilt

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

(Übung!). Das liefert folgenden Additionsalgorithmus:

function add($\frac{a}{b}$, $\frac{c}{d}$)
  // $\frac{a}{b}, \frac{c}{d} \in K$ in Normalform.
  // Berechnet $\frac{z}{n} = \frac{a}{b} + \frac{c}{d}$ (in Normalform).
  $g \leftarrow \gcd(b, d)$
  $b' \leftarrow b$ quo g; $d' \leftarrow d$ quo g // Division ohne Rest
  $\tilde{z} \leftarrow a \cdot d' + b' \cdot c$
  $g' \leftarrow \gcd(\tilde{z}, g)$
  $z \leftarrow \tilde{z}$ quo g' // Division ohne Rest
  $n \leftarrow (b' \cdot d)$ quo g' // Division ohne Rest

```
  return z/n
end function
```

Wenn man das tatsächlich implementiert, wird man jeweils prüfen, ob die berechneten ggTs gleich 1 sind, und sich in diesem Fall die Divisionen sparen.

5.2. **Modulare Inversion.** Sei R ein euklidischer Ring, $I \subset R$ ein von null verschiedenes Ideal. Dann ist $I = aR$ mit $0 \neq a \in R$ (denn R ist ein Hauptidealring). Wir haben dann den *Faktorring* oder *Restklassenring* $\bar{R} = R/I$ und den *kanonischen Epimorphismus* $R \to \bar{R}$, $b \mapsto \bar{b}$.

Um in $\bar{R}$ zu rechnen, stellen wir die Elemente von $\bar{R}$ durch geeignete Repräsentanten in R dar. Dabei bietet es sich an, solche Repräsentanten b zu nehmen, die $N(b) < N(a)$ erfüllen (wo N eine Normfunktion auf R ist). Wenn möglich, wird man einen eindeutigen solchen Repräsentanten wählen. Für $R = \mathbb{Z}$ etwa kann man Eindeutigkeit erreichen (wie bei der Division mit Rest), indem man $0 \leq b < |a|$ verlangt. Für $R = K[X]$ genügt die Bedingung $\deg b < \deg a$.

Die Grundrechenoperationen $* = +, -, \cdot$ lassen sich dann via

$$\bar{b} * \bar{c} = \overline{(b * c) \text{ rem } a}$$

implementieren. Je nach Ring ergeben sich noch Vereinfachungen bei der Addition und Subtraktion. Für $R = \mathbb{Z}$ hat man etwa (wir nehmen $a > 0$ an):

```
function add(b̄, c̄)
  // b̄, c̄ ∈ ℤ/aℤ, repräsentiert durch b, c mit 0 ≤ b, c < a
  // Ausgabe: s̄ = b̄ + c̄
  s ← b + c
  if s ≥ a then s ← s − a end if
  return s̄
end function
```

Für den Polynomring $R = K[x]$ kann man einfach die Repräsentanten addieren bzw. subtrahieren. In jedem Fall ist die Komplexität linear in $\lambda(a)$ (Wortkomplexität für $R = \mathbb{Z}$) bzw. in $\deg a$ (Operationen in K für $R = K[X]$).

Bei der Multiplikation hat man (im schlimmsten Fall) ein Element etwa der doppelten Länge von a durch a zu teilen. Die Komplexität der Multiplikation von b und c und der Division von bc durch a ist quadratisch (in $\lambda(a)$ bzw. $\deg a$), wenn wir die Schulmethode verwenden.

Wie sieht es mit der Division aus? Dazu müssen wir erst einmal wissen, welche Elemente in $\bar{R}$ invertierbar sind:

5.3. **Lemma.** *Sei R ein euklidischer Ring, $0 \neq a \in R$, $\bar{R} = R/aR$. Ein Element $\bar{b} \in \bar{R}$ ist genau dann in $\bar{R}$ invertierbar, wenn* $\operatorname{ggT}(a, b) \sim 1$.

Sind $(1, s, t)$ die Ausgabewerte des Erweiterten Euklidischen Algorithmus, angewandt auf a und b, dann gilt $\bar{b}^{-1} = \bar{t}$.

Beweis. $\bar{b}$ ist genau dann invertierbar, wenn es ein $t \in R$ gibt mit $\bar{b}\bar{t} = \bar{1}$, also $bt \equiv 1 \bmod a$. Das wiederum bedeutet, dass es $s \in R$ gibt mit $as + bt = 1$. Daraus folgt $\operatorname{ggT}(a, b) \sim 1$. Umgekehrt folgt aus $\operatorname{ggT}(a, b) \sim 1$, dass es solche $s, t \in R$ gibt. Die zweite Aussage ist dann klar. □

Der Erweiterte Euklidische Algorithmus ermöglicht es uns also, festzustellen, ob ein gegebenes Element $\bar{b} \in \bar{R}$ invertierbar ist, und in diesem Fall das Inverse zu bestimmen. Dazu genügt es, nur die t_i zu berechnen. Für $R = K[X]$ und $\deg a = n$ ist die Komplexität dann $\leq 4n^2 + O(n)$ Operationen in K. Für $R = \mathbb{Z}$ ist die (Wort-)Komplexität $\leq \lambda(a)^2$.

5.4. **Definition.** Ist R ein euklidischer Ring und $0 \neq a \in R$, dann sagen wir a sei *irreduzibel*, wenn a keine Einheit ist und jeder Teiler von a entweder eine Einheit oder zu a assoziiert ist.

Jedes irreduzible Element ist auch *prim:* Sei p irreduzibel $p \mid ab$ und $p \nmid a$. Dann ist $\mathrm{ggT}(a, p) \sim 1$, also $1 = sp + ta$ und damit $b = bsp + bta = p(bs + tu)$ (mit $ab = up$), also gilt $p \mid b$.

Insbesondere gilt also: Ist $a \in R$ irreduzibel, dann ist R/aR ein *Körper*. If Fall $R = \mathbb{Z}$, $a = p$ eine Primzahl, schreiben wir $\mathbb{F}_p$ für den Körper $\mathbb{Z}/p\mathbb{Z}$.

5.5. **Definition.** Sei R ein euklidischer Ring, $a, b \in R$. Wir sagen, a und b seien *teilerfremd* (oder *relativ prim*) und schreiben $a \perp b$, wenn $\mathrm{ggT}(a, b) \sim 1$.

5.6. **Lineare Diophantische Gleichungen.** Der Erweiterte Euklidische Algorithmus verhilft uns auch zur Lösung von Gleichungen der Art

$$ax + by = c$$

über einem euklidischen Ring R. Im Falle $R = \mathbb{Z}$ handelt es sich um eine lineare diophantische Gleichung.

Wir haben folgendes Ergebnis:

5.7. **Lemma.** *Sei R ein euklidischer Ring, $a, b, c \in R$ mit $g = \mathrm{ggT}(a, b) \neq 0$. Die Gleichung $ax + by = c$ ist in R genau dann lösbar, wenn $g \mid c$. In diesem Fall sei $(x, y) = (s, t)$ eine Lösung. Dann sind alle Lösungen gegeben durch*

$$(x, y) = (s + ub', t - ua')\,,$$

wobei $u \in R$ beliebig ist und $a' = a/g$, $b' = b/g$.

Beweis. Gibt es $x, y \in R$ mit $ax + by = c$, dann folgt $g \mid c$. Umgekehrt gibt es $s', t' \in R$ mit $as' + bt' = g$; mit $c' = c/g$ ist dann $(s, t) = (s'c', t'c')$ eine Lösung.

Sei jetzt (s, t) eine Lösung. Es ist klar (wegen $ab' - a'b = 0$), dass dann für jedes $u \in R$ $(s+ub', t-ua')$ wieder eine Lösung ist. Sei nun (x, y) eine beliebige Lösung. Dann gilt $a(x - s) + b(y - t) = 0$, also auch $a'(x - s) + b'(y - t) = 0$. Da a' und b' teilerfremd sind (Übung!), muss a' ein Teiler von $y - t$ und b' ein Teiler von $x - s$ sein (Übung!). Es folgt $(x, y) = (s + ub', t - ua')$ für ein $u \in R$. □

Um eine gegebene Gleichung zu lösen, gehen wir also wie folgt vor:

```
function solve(a,b,c)
  // a, b, c ∈ R, (a, b) ≠ (0, 0).
  // Ausgabe: „keine Lösung“ oder (s, t, v, w),
  //    so dass (s + uv, t + uw) alle Lösungen liefert.
  (g, s, t) ← xgcd(a, b)
  if g | c then
    c' ← c quo g
    (s, t) ← (c' · s, c' · t)
```

```
    return (s, t, b quo g, −a quo g)
  else
    return „keine Lösung“
  end if
end function
```

Man kann statt b/g, $-a/g$ auch die Größen $s_{\ell+1}$, $t_{\ell+1}$ aus dem EEA verwenden, denn $(s_{\ell+1}, t_{\ell+1}) = u(b/g, -a/g)$ mit einer Einheit u (Übung!).

Man kann das Lösungsverfahren auf lineare Gleichungen in mehr als zwei Variablen verallgemeinern, siehe etwa [GG, Thm. 4.11].

6. Modulare Arithmetik

Wir bleiben beim Rechnen in Faktorringen $\bar{R}$. Wir werden nämlich sehen, dass dies in vielen Fällen effizientere Algorithmen ermöglicht.

Wir haben uns schon davon überzeugt, dass wir in $\mathbb{Z}/a\mathbb{Z}$ mit linearem (in $\lambda(a)$) Aufwand addieren und subtrahieren und mit quadratischem Aufwand multiplizieren und invertieren können; analoges gilt für $R = K[X]$.

Wir können aber auch effizient *potenzieren*. Dies geschieht mit der Methode des *sukzessiven Quadrierens.*

6.1. **Lemma.** *In $\mathbb{Z}/a\mathbb{Z}$ können wir $\bar{b}^e$ mit $\ll \lambda(a)^2\lambda(e)$ Wortoperationen berechnen. In $K[X]/aK[X]$ geht es entsprechend mit $\ll (\deg a)^2\lambda(e)$ Operationen in K.*

Beweis. Wir geben einen Algorithmus dafür an:

```
function modpower(a,b,e)
  // a, b ∈ R, e ∈ Z_{≥0}.
  // Ausgabe: b^e rem a.
  if e = 0 then
    return 1
  else
    (e′, d) ← (e quo 2, e rem 2)
    r ← modpower(a, b^2 rem a, e′)
    if d = 1 then r ← (r · b) rem a end if
    return r
  end if
end function
```

Dieser Algorithmus ist korrekt, denn

$$\bar{b}^{2e'} = (\bar{b}^2)^{e'} \quad \text{und} \quad \bar{b}^{2e'+1} = (\bar{b}^2)^{e'} \cdot \bar{b}\,;$$

außerdem wird e bei jedem rekursiven Aufruf kleiner, also erreichen wir schließlich den Fall $e = 0$.

Für die Komplexität überlegen wir uns, dass bei jedem Aufruf ein oder zwei Multiplikationen im Faktorring stattfinden (b^2 rem a und eventuell $(r \cdot b)$ rem a); die Komplexität dafür ist $\ll \lambda(a)^2$ bzw. $\ll (\deg a)^2$. Bei jedem rekursiven Aufruf wird e (mindestens) halbiert, so dass die Rekursionstiefe gerade durch die Anzahl der Bits in der Dualdarstellung von e gegeben ist. Diese ist $\ll \lambda(e)$. □

6.2. **Anwendung: RSA.** Die schnelle modulare Potenzierung ist wesentlich dafür, dass das RSA-Verfahren der Public-Key-Kryptographie praktikabel ist. Dabei wählt man zwei große Primzahlen (mehrere Hundert Dezimalstellen) p und q und setzt $n = pq$. Außerdem wählt man noch eine natürliche Zahl $e > 1$, teilerfremd zu $(p-1)$ und $(q-1)$. Das Paar (n, e) wird als öffentlicher Schlüssel publiziert. Wenn jemand eine Nachricht schicken möchte, codiert er diese als Folge von Zahlen $0 \leq m < n$ und verschlüsselt diese gemäß $m \mapsto m^e$ rem n. Zur Entschlüsselung berechnet man d mit $de \equiv 1 \bmod \mathrm{kgV}(p-1, q-1)$. Das Paar (n, d) ist dann der private Schlüssel, und die Entschlüsselung erfolgt gemäß $c \mapsto c^d$ rem n.

Man kann zwar e so wählen, dass es nicht allzu groß ist, aber der Enstchlüsselungsexponent d wird normalerweise nicht viel kleiner sein als n. Daher ist es wichtig, dass wir c^d rem n effizient berechnen können.

6.3. **Übung.** Vergleichen Sie die Komplexität des naiven Potenzierungsalgorithmus

```
function power(a, e)
  // a ∈ R, e ∈ Z_{≥0}.
  // Ausgabe: a^e.
  r ← 1
  for i = 1 to e do r ← r · a end for
  return r
end function
```

mit der des obigen Algorithmus (ohne die rem a-Operationen und mit a statt b), für $R = K[X]$ (führender Term der Anzahl an Operationen in K), in den beiden Extremfällen $e = 2^k$ und $e = 2^{k+1} - 1$.

6.4. **Übung.** Bestimmen Sie die Komplexität von Addition und Multiplikation im Polynomring $K[X, Y]$ in zwei Variablen über einem Körper K, ausgedrückt im Totalgrad der Operanden (der Totalgrad von $X^m Y^n$ ist $m + n$). Wiederholen Sie dann den Vergleich zwischen dem naiven Potenzierungsalgorithmus und dem Algorithmus, der sukzessives Quadrieren verwendet, für $K[X, Y]$.

6.5. **Übung.** Sei p eine Primzahl. Dann gilt („kleiner Satz von Fermat“) im Körper $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$ für $\bar{a} \neq 0$, dass $\bar{a}^{p-1} = 1$ ist. Insbesondere ist $\bar{a}^{-1} = \bar{a}^{p-2}$. Vergleichen Sie die Komplexität von modpower$(p, a, p-2)$ und der Inversenberechnung mit Hilfe des EEA.

Die Auswertung eines Polynoms ist das selbe wie einen Divisionsrest zu berechnen:

6.6. **Lemma.** *Sei K ein Körper, $\xi \in K$, und $f \in K[X]$. Dann ist*

$$f(\xi) = f \text{ rem } (X - \xi)\,.$$

Beweis. Sei $r = f$ rem $(X - \xi)$ und $q = f$ quo $(X - \xi)$. Dann ist

$$f = q(X - \xi) + r\,,$$

und r ist konstant. Einsetzen von ξ liefert $f(\xi) = r(\xi) = r$. □

Es kann vorteilhaft sein, „modular“ zu rechnen. Im einfachsten Fall haben wir etwas zu berechnen, das sich als Polynom (mit Koeffizienten im Ring R) in den Eingabedaten ausdrücken lässt, und wir haben eine Abschätzung (im Sinne der Normfunktion auf R) für das Ergebnis. Dann wählen wir ein geeignetes Element $a \in R$ (wir haben dabei große Freiheit; z.B. können wir a als irreduizbel wählen), so dass jedes Element von R/aR von höchstens einem Element von R, das der Abschätzung genügt, repräsentiert wird.

Um das Ergebnis zu berechnen, reduzieren wir erst die Eingabedaten modulo a, berechnen dann das Resultat in R/aR, und bestimmen dann das eindeutige dazu passende Element von R.

6.7. **Beispiel.** Eine n-reihige Determinante über einem Körper K lässt sich mit $\ll n^3$ Operationen in K berechnen (Standardalgorithmus via Gauß-Verfahren). Um eine Determinante über $\mathbb{Z}$ zu berechnen, können wir zum Beispiel in $\mathbb{Q}$ arbeiten; die Zwischenergebnisse können dabei aber recht groß werden.

Wir wissen aber, dass

$$\det(a_{ij}) = \sum_{\sigma \in S_n} \varepsilon(\sigma) a_{1,\sigma(1)} a_{2,\sigma(2)} \cdots a_{n,\sigma(n)}$$

ein Polynom (mit ganzzahligen Koeffizienten) in den Matrixeinträgen ist. Gilt $|a_{ij}| \le B$ für alle $1 \le i, j \le n$, dann folgt daraus

$$\bigl|\det(a_{ij})\bigr| \le n!\, B^n\,.$$

(Es gibt etwas bessere Abschätzungen, z.B. $n^{n/2} B^n$.) Wenn wir eine Primzahl $p > 2n!\, B^n$ wählen, dann können wir die Determinante in $\mathbb{F}_p$ berechnen (Aufwand $\ll n^3 \lambda(p)^2 \ll n^5(\lambda(B) + \lambda(n))^2$) und anschließend in $\mathbb{Z}$ rekonstruieren. Allerdings ergibt das noch keinen Vorteil gegenüber der Rechnung in $\mathbb{Q}$ (wie man durch Ausprobieren feststellen kann — dies soll eine Aufforderung sein, es auszuprobieren!). Auf der anderen Seite sieht man so sehr leicht, dass man die Determinante in Polynomzeit berechnen kann (Größe der Eingabe ist $n^2\lambda(B)$), was für die Version über $\mathbb{Q}$ nicht so einfach zu zeigen ist.

In der obigen Überlegung fehlt noch eine Abschätzung des Aufwandes zur Bestimmung von p. Dazu testet man die natürlichen Zahlen ab $2n!B^n + 1$, ob sie prim sind. Nach dem Primzahlsatz braucht man $\ll \log(2n!B^n) \ll n(\lambda(B) + \lambda(n))$ Versuche, bis man eine Primzahl gefunden hat, und der einzelne Test lässt sich in Polynomzeit (polynomial in $n(\lambda(B) + \lambda(n))$) durchführen. Gibt man sich mit „Pseudoprimzahlen“ zufrieden, ist der Test höchstens von kubischer Komplexität, und die Laufzeit vergleichbar mit der Rechnung in $\mathbb{Q}$. Wenn man garantiert eine Primzahl haben möchte, dann wird die Laufzeit von den Primzahltests dominiert.

Meistens lassen sich Berechnungen wie die der Determinante noch beschleunigen, wenn man statt *einer großen* Primzahl *viele kleine* Primzahlen verwendet. Das wichtigste Hilfsmittel ist hier der Chinesische Restsatz.

6.8. **Chinesischer Restsatz.** *Sei R ein euklidischer Ring, $m_1, \ldots, m_n \in R$ seien paarweise teilerfremd. Dann gibt es für jede Wahl von $a_1, \ldots, a_n \in R$ ein Element $x \in R$ mit $x \equiv a_j \bmod m_j$ für alle $1 \le j \le n$, und x ist modulo $m = m_1 \cdots m_n$ eindeutig bestimmt.*

Man kann das auch so ausdrücken: Der kanonische Ringhomomorphismus

$$R/mR \longrightarrow R/m_1R \times R/m_2R \times \cdots \times R/m_nR$$

ist ein Isomorphismus.

Beweis. Wir geben zwei Varianten eines algorithmischen Beweises für die Existenz. Die erste geht so:

```
function crt((m_1, ..., m_n), (a_1, ..., a_n))
  // m_1, ..., m_n ∈ R paarweise teilerfremd, a_1, ..., a_n ∈ R.
  // Ausgabe: x ∈ R mit x ≡ a_j mod m_j für 1 ≤ j ≤ n.
  m ← m_1 · m_2 · ····· m_n
  x ← 0
  for j = 1 to n do
    // Berechne „Basislösungen“
    (g_j, s_j, t_j) ← xgcd(m_j, m/m_j)
    // Hier könnten wir überprüfen, dass g_j = 1 ist.
    x_j ← t_j · (m/m_j)
    // Es gilt x_j ≡ 0 mod m_i für i ≠ j und x_j ≡ 1 mod m_j.
    x ← x + a_j · x_j
  end for
  return x rem m
end function
```

Es gilt $s_j m_j + t_j(m/m_j) = g_j = 1$, also ist $x_j = t_j(m/m_j) \equiv 1 \bmod m_j$. Dass x_j durch alle m_i mit $i \neq j$ teilbar ist, ist klar. Am Ende ist

$$x = \sum_{i=1}^{n} a_i x_i \equiv a_j \bmod m_j$$

für alle j.

Die zweite Variante verwendet Induktion:

```
function crt2(m_1, m_2, a_1, a_2)
  // m_1, m_2, a_1, a_2 ∈ R, m_1 ⊥ m_2.
  // Ausgabe: x ∈ R mit x ≡ a_1 mod m_1, x ≡ a_2 mod m_2.
  (g, s_1, s_2) ← xgcd(m_1, m_2)
  // g = 1, s_1 m_1 + s_2 m_2 = 1
  return (a_1 · s_2 · m_2 + a_2 · s_1 · m_1) rem (m_1 · m_2)
end function

function crt((m_1, ..., m_n), (a_1, ..., a_n))
  // m_1, ..., m_n ∈ R paarweise teilerfremd, a_1, ..., a_n ∈ R.
  // Ausgabe: x ∈ R mit x ≡ a_j mod m_j für 1 ≤ j ≤ n.
  if n = 0 then return 0 end if
  if n = 1 then return a_1 rem m_1 end if
  m ← m_1 · ····· m_{n-1}
  return crt2(m, m_n, crt((m_1, ..., m_{n-1}), (a_1, ..., a_{n-1})), a_n)
end function
```

Hier ist „crt2“ ein Spezialfall des obigen ersten „crt“-Algorithmus (etwas sparsamer, weil nur einmal der EEA verwendet wird).

Für die erste Version erhält man leicht eine Komplexität $\ll (\deg m)^2$ Operationen in K für Polynome bzw. $\ll \lambda(m)^2$ Wortoperationen für ganze Zahlen.

Der zweite Teil der Behauptung (dass $x \bmod m$ eindeutig bestimmt ist), folgt aus

$$\forall j : m_j \mid x \iff \mathrm{kgV}(m_1, \ldots, m_n) \mid x \iff m \mid x\,,$$

denn $\mathrm{kgV}(m_1, \ldots, m_n) = m_1 \cdots m_n$, wenn die m_j paarweise teilerfremd sind. □

6.9. **Übung.** Vergleichen Sie die Komplexität der beiden Algorithmen für den Chinesischen Restsatz, wenn $m_j \in K[X]$ mit $\deg m_j = d$ für alle j.

Kann man durch bessere Organisation der Rechnung vielleicht noch etwas gewinnen?

6.10. **Interpolation.** Für Polynome $m_j = X - \alpha_j$ (mit paarweise verschiedenen $\alpha_j \in K$) ist die Berechnung von $f \in K[X]$ mit $f \equiv a_j \bmod m_j$ das selbe wie die Werte $a_j(\alpha_j)$ an den Stellen α_j zu *interpolieren*, denn

$$f \equiv a_j \bmod m_j \iff f(\alpha_j) = a_j(\alpha_j)\,.$$

Wir sehen, dass wir das interpolierende Polynom mit $\ll n^2$ Operationen in K berechnen können (wir nehmen an, die a_j sind konstant).

Umgekehrt können wir die Werte $f(\alpha_j)$ ebenfalls mit $\ll n^2$ Operationen in K berechnen. Dazu verwendet man das *Horner-Schema*, das eine Spezialisierung des Divisionsalgorithmus ist:

```
function eval(f, α)
  // f = a_n X^n + ··· + a_1 X + a_0 ∈ K[X], α ∈ K.
  // Ausgabe: f(α) ∈ K.
  if f = 0 then return 0 end if
  r ← a_n
  for j = n − 1 to 0 by −1 do
    r ← α · r + a_j
  end for
  return r
end function
```

Jede Auswertung benötigt n Additionen und Multiplikationen in K.

6.11. **Schnellere Multiplikation?** Man könnte jetzt versuchen, einen schnelleren Multiplikationsalgorithmus für Polynome (zum Beispiel) zu bekommen, indem man die Polynome als Listen von ihren Werten an geeigneten Stellen darstellt. Ein Polynom vom Grad n ist durch $n + 1$ Werte eindeutig bestimmt. Wenn ich zwei Polynome vom Grad n miteinander multiplizieren will, muss ich ihre Werte an $2n + 1$ Stellen kennen. Das liefert folgendes Verfahren zur Multiplikation von a und b in $K[X]$:

(1) Wähle $N = \deg a + \deg b + 1$ verschiedene Elemente $\alpha_j \in K$.
(2) Berechne $u_j = a(\alpha_j)$ und $v_j = b(\alpha_j)$ für $1 \le j \le N$.
(3) Berechne $w_j = u_j \cdot v_j$ für $1 \le j \le N$.
(4) Berechne $a \cdot b$ als das Polynom, das für alle j an α_j den Wert w_j hat.

Die eigentliche Multiplikation ist sehr schnell: Man braucht genau N Multiplikationen in K. Allerdings benötigen der zweite und der vierte Schritt immer noch $\asymp N^2$ Operationen, so dass dieser Ansatz kein schnelleres Verfahren liefert. (Wir werden aber später sehen, dass die Idee ausbaufähig ist.)

Wenn ein komplizierterer Ausdruck berechnet werden soll, kann es sich allerdings lohnen, erst in geeignete Restklassenringe zu gehen und am Ende über den Chinesischen Restsatz wieder zurück in den ursprünglichen Ring zu kommen. Das gilt dann, wenn der mittlere Schritt (Nr. (3) oben) vom Aufwand her die Schritte (2) und (4) dominiert.

6.12. **Schnellere Determinante.** Wir können zum Beispiel die Berechnung der Determinante über $\mathbb{Z}$ beschleunigen. Dazu wählen wir ℓ verschiedene Primzahlen p_j, so dass $P = p_1 \cdots p_\ell > 2n!B^n$ ist (Bezeichnungen wie in 6.7). Wir berechnen die Reduktion mod p_j unserer Matrix für alle j (Kosten: $\ll n^2\lambda(B)\sum_j \lambda(p_j) \ll n^2\lambda(B)\lambda(P)$), dann berechnen wir die Determinanten der reduzierten Matrizen in $\mathbb{F}_{p_j}$ (Kosten: $\ll n^3\sum_j \lambda(p_j)^2 \ll n^3\lambda(P)^2/\ell$, wenn die Primzahlen etwa gleich groß sind), schließlich rekonstruieren wir die Determinante in $\mathbb{Z}$ mit dem Chinesischen Restsatz (Kosten: $\ll \lambda(P)^2$). Mit $\lambda(P) \ll n(\lambda(B) + \lambda(n))$ sind die Gesamtkosten also

$$\ll n^3\lambda(B)\big(\lambda(B) + \lambda(n)\big) + \frac{n^5}{\ell}\big(\lambda(B) + \lambda(n)\big)^2$$

(der letzte Schritt fällt hier nicht ins Gewicht). Je größer wir die Anzahl ℓ der verwendeten Primzahlen wählen können, desto schneller wird die Berechnung. In der Praxis wird man Primzahlen wählen, die knapp in ein Wort passen (also etwa $2^{63} < p < 2^{64}$); davon gibt es nach dem Primzahlsatz genügend, um Ergebnisse zu berechnen, die sich überhaupt im Computer platzmäßig darstellen lassen. Der Vorteil ist, dass man dann mit „einfacher Genauigkeit“ und damit schnell modulo p rechnen kann. Dann ist $\ell \approx \lambda(P) \ll n(\lambda(B) + \lambda(n))$, und die Komplexität ist nur noch

$$\ll n^3\big(n + \lambda(B)\big)\big(\lambda(B) + \lambda(n)\big)\,.$$

Man vergleiche mit $n^5\big(\lambda(B) + \lambda(n)\big)^2$ für den Algorithmus mit einer großen Primzahl (ohne den Aufwand, diese zu bestimmen!).

In der Theorie stimmt das nicht ganz, denn irgendwann gibt es nicht genügend Primzahlen, damit man ℓ wie eben beschrieben wählen kann. Es gilt (das folgt aus dem Primzahlsatz), dass das Produkt der Primzahlen unterhalb von x etwa e^x ist; es gibt etwa $x/\log x$ solche Primzahlen. Der maximal sinnvoll mögliche Wert von ℓ ist also etwa

$$\ell \approx \frac{\log(2n!B^n)}{\log\log(2n!B^n)} \asymp \frac{n\big(\lambda(B) + \lambda(n)\big)}{\log n + \log\big(\lambda(B) + \lambda(n)\big)}\,.$$

Damit bekommen wir eine Komplexität von

$$\ll n^3\big(n(\log n + \log(\lambda(B) + \lambda(n))) + \lambda(B)\big)\big(\lambda(B) + \lambda(n)\big)\,.$$

(Das ist ein bisschen geschummelt, weil dann die Primzahlen nicht alle etwa gleich groß sind; da die größeren Primzahlen aber weit überwiegen, stimmt die Komplexitätsaussage immer noch). Man sollte sich noch überlegen, dass man auch die Liste der ℓ benötigten Primzahlen in vernachlässigbarer Zeit berechnen kann: Man kann (wenn einem nichts besseres einfällt) alle Zahlen bis $\approx n\big(\lambda(B) + \lambda(n)\big)$ testen, ob sie prim sind. Der einzelne Test lässt sich in Polynomzeit in der Größe der getesteten Zahl durchführen, also in $\ll (\log n + \log\log(Bn))^k$ Wortoperationen für ein passendes k. Damit ist der Gesamtaufwand

$$\ll n\big(\lambda(B) + \lambda(n)\big)\big(\log n + \log\log(Bn)\big)^k\,,$$

was deutlich hinter dem Aufwand für den restlichen Algorithmus zurück bleibt.

7. Schnellere Multiplikation

Es gibt eine recht einfache Möglichkeit, schneller als in quadratischer Zeit zu multiplizieren. Der Grundgedanke dabei ist „Divide And Conquer“: Man führt das Problem auf analoge Probleme kleinerer (meistens etwa halber) Größe zurück.

Für die Multiplikation zweier linearer Polynome $aX + b$ und $cX + d$:

$$(aX + b)(cX + d) = acX^2 + (ad + bc)X + bd$$

braucht man im klassischen Algorithmus, wie der der Formel entspricht, *vier* Multiplikationen und eine Addition im Koeffizientenring. Man kann den Koeffizienten von X im Ergebnis aber auch schreiben als

$$ad + bc = (a + b)(c + d) - ac - bd\,.$$

Da man ac und bd sowieso berechnen muss, kommt man mit insegesamt *drei* Multiplikationen aus; der Preis, den man dafür bezahlt, besteht in drei zusätzlichen Additionen/Subtraktionen. Da Additionen aber im allgemeinen deutlich billiger sind als Multiplikationen, kann man damit etwas gewinnen.

7.1. **Die Karatsuba-Multiplikation.** Wir wollen natürlich nicht nur lineare Polynome multiplizieren. Seien jetzt $a, b \in K[X]$ mit $\deg a, \deg b < 2n$. Dann schreiben wir

$$a = a_1X^n + a_0\,, \qquad b = b_1X^n + b_0$$

mit Polynomen a_0, a_1, b_0, b_1 vom Grad $< n$. Wie vorher gilt dann

$$ab = a_1b_1X^{2n} + \big((a_1 + a_0)(b_1 + b_0) - a_1b_1 - a_0b_0\big)X^n + a_0b_0\,.$$

Wenn wir die Teilprodukte mit der klassischen Methode berechnen (zu Kosten von n^2 Multiplikationen und $(n-1)^2$ Additionen), dann bekommen wir hier (M: Multiplikationen, A: Additionen/Subtraktionen in K)

- 3 Multiplikationen von Polynomen vom Grad $< n$: $3n^2$ M, $3(n-1)^2$ A
- 2 Additionen von Polynomen vom Grad $< n$: $2n$ A
- 2 Subtraktionen von Polynomen vom Grad $< 2n$: $4n$ A
- $2n$ Additionen von Koeffizienten beim „Zusammensetzen“: $2n$ A

Insgesamt also $3n^2$ Multiplikationen und $3n^2 + 2n + 3$ Additionen oder Subtraktionen. Die klassische Methode braucht $4n^2$ Multiplikationen und $4n^2 - 4n + 1$ Additionen/Subtraktionen.

Wir können das Ganze natürlich auch rekursiv machen; dann kommt der Geschwindigkeitsvorteil erst richtig zur Geltung:

```
function karatsuba(a, b, k)
  // a, b ∈ K[X], deg a, deg b < 2^k.
  // Ausgabe: a · b.
  if k = 0 then
    return a · b // konstante Polynome
  else
    a_1 X^{2^{k-1}} + a_0 ← a
    b_1 X^{2^{k-1}} + b_0 ← b
    c_2 ← karatsuba(a_1, b_1, k − 1)
    c_0 ← karatsuba(a_0, b_0, k − 1)
    c_1 ← karatsuba(a_1 + a_0, b_1 + b_0, k − 1) − c_2 − c_0
    // c_0, c_1, c_2 sind Polynome vom Grad < 2^k.
    return c_2 X^{2^k} + c_1 X^{2^{k-1}} + c_0
  end if
end function
```

Sei $T(k)$ der Aufwand für einen Aufruf karatsuba$(a,\, b,\, k)$. Dann gilt

$$T(0) = \mathrm{M} \quad \text{und} \quad T(k) = 3T(k-1) + 4 \cdot 2^k \mathrm{A} \text{ für } k \geq 1.$$

Das folgende Lemma sagt uns, wie wir diese Rekursion auflösen können.

7.2. **Lemma.** *Sei $(a_n)_{n\geq 0}$ rekursiv definiert durch*

$$a_0 = \alpha\,, \qquad a_n = \lambda a_{n-1} + \beta\mu^n \quad \textit{für } n \geq 1.$$

Dann gilt

$$a_n = \begin{cases} \alpha\lambda^n + \frac{\beta\mu}{\mu-\lambda}(\mu^n - \lambda^n) & \textit{falls } \lambda \neq \mu, \\ (\alpha + \beta n)\lambda^n & \textit{falls } \lambda = \mu. \end{cases}$$

Beweis. Es ist klar, dass es genau eine Lösung (a_n) gibt. Man prüft in beiden Fällen nach, dass die angegebene Folge eine Lösung ist. □

7.3. **Folgerung.** Aus Lemma 7.2 erhalten wir:

Die Kosten für karatsuba$(a,\, b,\, k)$ betragen 3^k Multiplikationen und $8(3^k - 2^k)$ Additionen oder Subtraktionen in K.

Wenn wir das im Grad $n = 2^k$ ausdrücken, dann sagt das:

Wir können das Produkt zweier Polynome vom Grad $< n = 2^k$ mit einem Aufwand von $n^{\log_2 3}$ Multiplikationen und $8n^{\log_2 3}$ Additionen/Subtraktionen in K berechnen.

Man beachte, dass $\log_2 3 \approx 1,585$ und damit deutlich kleiner als 2 ist.

7.4. **Zur Praxis.** In der Praxis ist es meistens so, dass die klassische Methode für Polynome von kleinem Grad schneller ist. Daher wird man statt „if $k = 0 \ldots$“ eine etwas höhere Schranke wählen (und dann das Resultat klassisch berechnen). Die optimale Wahl muss durch Ausprobieren ermittelt werden.

Meistens ist der Grad nicht von der Form $2^k - 1$. Man geht dann etwa so vor (d ist die Gradschranke für den klassischen Algorithmus):

```
function karatsuba′(a, b)
  // a, b ∈ K[X]. Ausgabe: a · b.
  n ← deg a + 1; m ← deg b + 1
  if n < m then (a, b, n, m) ← (b, a, m, n) end if
  // Jetzt ist n ≥ m.
  if m < d then
    return multiply(a, b) // Klassische Methode
  else
    k ← ⌈n/2⌉
    a_1 X^k + a_0 ← a
    b_1 X^k + b_0 ← b
    if b_1 = 0 then
      c_1 ← karatsuba′(a_1, b_0)
      c_0 ← karatsuba′(a_0, b_0)
      return c_1 X^k + c_0
    else
      c_2 ← karatsuba′(a_1, b_1)
      c_0 ← karatsuba′(a_0, b_0)
      c_1 ← karatsuba′(a_1 + a_0, b_1 + b_0) − c_2 − c_0
```

```
      return c_2 X^{2k} + c_1 X^k + c_0
    end if // b_1 = 0
  end if // n < d
end function
```

Die Alternative wäre, karatsuba(a, b, k) zu verwenden mit dem kleinsten k, das $2^k > \deg a, \deg b$ erfüllt. Dabei verschenkt man aber zu viel (im schlimmsten Fall einen Faktor ≈ 3).

7.5. **Übung.** Zeigen Sie, dass die Komplexität von karatsuba′ für $n = \deg a \geq m = \deg b$ in der Form $\ll nm^{\log_2 3 - 1}$ beschränkt werden kann.

7.6. **Übung.** Implementieren Sie die klassische und die Karatsuba-Multiplikation für Polynome. Vergleichen Sie die Laufzeiten für Polynome verschieden hohen Grades über einem endlichen Körper. Finden Sie den optimalen Wert des Parameters d.

7.7. **Übung.** Finden Sie einen Karatsuba-analogen Algorithmus, der die Polynome in drei statt zwei Stücke zerlegt. Ist er asymptotisch schneller?

Wie sieht es aus, wenn K eine primitive dritte Einheitswurzel ω enthält?

7.8. **Karatsuba für ganze Zahlen.** Für ganze Zahlen (statt Polynome) funktioniert der Algorithmus im wesentlichen genauso: Sei $\lambda(a), \lambda(b) \leq 2N$ und schreibe

$$a = a_1 B^N + a_0 \quad \text{und} \quad b = b_1 B^N + b_0$$

(mit $B = 2^{64}$ wie üblich). Dann ist

$$a \cdot b = (a_1 b_1) B^{2N} + \big((a_1 + a_0)(b_1 + b_0) - a_1 b_1 - a_0 b_0\big) B^N + (a_0 b_0)\,.$$

Das einzige Problem ist (wie meistens), dass bei der Berechnung des Teilprodukts $(a_1 + a_0)(b_1 + b_0)$ die Faktoren $\geq B^N$ sein können, was für die Rekursion nicht so schön ist. Es gilt aber stets $a_1 + a_0, b_1 + b_0 < 2B^N$; es gibt also höchstens 1 Bit „Überlauf“, das man am besten separat verarztet.

An der Komplexität ändert sich nichts: Mit dem Karatsuba-Algorithmus kann man zwei Zahlen $a, b \in \mathbb{Z}$ mit $\lambda(a), \lambda(b) \leq n$ mit $\ll n^{\log_2 3}$ Wortoperationen multiplizieren.

7.9. **Geht es noch besser?** Da wir gesehen haben, dass die Schulmethode für die Multiplikation noch nicht der Weisheit letzter Schluss ist, kann man sich fragen, ob mit der Karatsuba-Methode schon das Ende der Fahnenstange erreicht ist. Man kann etwa untersuchen, ob ähnliche Ansätze, bei denen (zum Beispiel) das Polynom in mehr als zwei Teile aufgeteilt wird, zu einer besseren asymptotischen Komplexität führen. Die Antwort ist „Ja“.

7.10. **Satz.** *Sei $\varepsilon > 0$ und K ein unendlicher Körper. Dann gibt es einen Algorithmus, der zwei Polynome vom Grad $< n$ über K in $\ll n^{1+\varepsilon}$ Operationen in K multipliziert.*

Beweis. Sei $m \geq 2$, so dass $\log(2 + \frac{1}{m}) < \varepsilon \log m$. Wähle $2m + 1$ verschiedene Elemente $\alpha_1, \ldots, \alpha_{2m+1}$ in K (hier braucht man, dass K groß genug ist). Sei V_{2m+1} der K-Vektorraum der Polynome vom Grad $\leq 2m$. Dann ist die K-lineare Abbildung

$$\phi : V_{2m+1} \longrightarrow K^{2m+1}\,, \qquad f \longmapsto \big(f(\alpha_1), \ldots, f(\alpha_{2m+1})\big)$$

ein Isomorphismus. Sei M die Matrix von ϕ^{-1} bezüglich der Standardbasis auf K^{2m+1} und der Basis $1, X, \ldots, X^{2m}$ auf V_{2m+1}. Sei weiter A die Matrix der Einschränkung von ϕ auf V_{m+1} (Polynome vom Grad $\leq m$).

Für Polynome

$$a = a_m X^m + \cdots + a_1 X + a_0 \quad \text{und} \quad b = b_m X^m + \cdots + b_1 X + b_0$$

und deren Produkt

$$c = a \cdot b = c_{2m} X^{2m} + \cdots + c_1 X + c_0$$

gilt dann, wenn wir $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$ für ihre Koeffizientenvektoren schreiben,

$$\mathbf{c} = M\big((A\mathbf{a}) \bullet (A\mathbf{b})\big)\,,$$

wobei $\bullet$ die komponentenweise Multiplikation bezeichnet. Diese Berechnung kostet $2m+1$ Multiplikationen für die komponentenweise Multiplikation, dazu noch etliche Multiplikationen mit festen Elementen von K (den Einträgen der Matrizen M und A) und Additionen.

Wir wenden das nun auf Polynome höheren Grades an. Wir schreiben

$$\begin{aligned} a &= a_m X^{mN} + a_{m-1} X^{(m-1)N} + \cdots + a_1 X^N + a_0 \quad \text{und} \\ b &= b_m X^{mN} + b_{m-1} X^{(m-1)N} + \cdots + b_1 X^N + b_0 \end{aligned}$$

mit Polynomen a_j, b_j vom Grad $< N$, und analog

$$c = a \cdot b = c_{2m} X^{2mN} + \cdots + c_1 X^N + c_0\,.$$

Dann gilt nach wie vor

$$\mathbf{c} = M\big((A\mathbf{a}) \bullet (A\mathbf{b})\big)\,,$$

wobei $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$ jetzt Vektoren von *Polynomen* vom Grad $< N$ sind. Der Aufwand für die Multiplikation mit den Matrizen M und A ist linear in N. Dazu kommen $(2m+1)$ Multiplikationen von Polynomen vom Grad $< N$. Wenn $T_m(k)$ den Gesamtaufwand für die rekursive Multiplikation von Polynomen vom Grad $< m^k$ bezeichnet, haben wir

$$T(0) \in O(1) \quad \text{und} \quad T(k) \in (2m+1)T(k-1) + O(m^{k-1}) \text{ für } k \geq 1.$$

Es ergibt sich, dass $T(k) \ll (2m+1)^k$ ist. Für Polynome vom Grad $< n$ ergibt sich (durch Aufrunden des Grades oder durch ungefähre Teilung in m Teile) eine Komplexität

$$\ll n^{\log_m(2m+1)} = n^{1+\log_m(2+1/m)} \ll n^{1+\varepsilon}\,.$$

□

In der Praxis sind derartige Ansätze aber zu kompliziert, um gegenüber der Karatsuba-Methode einen Vorteil zu ergeben. Um etwas besseres zu bekommen, muss man das Auswerten und Interpolieren beschleunigen. Dafür macht man sich zu Nutze, dass man bei der Wahl der Stützstellen freie Hand hat. Die wesentliche Idee ist, dafür geeignete Einheitswurzeln zu verwenden. Die zugehörige Theorie wird im folgenden Abschnitt bereitgestellt.

8. Die Diskrete Fourier-Transformation

Die Fourier-Transformation in der Analysis ist die Abbildung

$$f \longmapsto \Big(\xi \mapsto \int_{-\infty}^{\infty} e^{ix\xi} f(x)\, dx\Big)$$

(für geeignete Klassen von Funktionen $f : \mathbb{R} \to \mathbb{C}$). Die *Diskrete Fourier-Transformation* ist eine diskrete Version davon. Sie hat die gleichen schönen Eigenschaften, aber den Vorteil, dass man sich keine Gedanken über Konvergenz machen muss.

8.1. **Definition.** Sei R ein Ring und $n \in \mathbb{Z}_{\geq 1}$. Ein Element $\omega \in R$ heißt *n-te Einheitswurzel*, wenn $\omega^n = 1$ ist. ω ist eine *primitive n-te Einheitswurzel*, wenn zusätzlich $n \cdot 1_R$ in R invertierbar ist, und wenn für keinen Primteiler p von n das Element $\omega^{n/p} - 1$ ein Nullteiler in R ist. (Insbesondere muss $\omega^{n/p} \neq 1$ gelten.)

8.2. **Beispiel.** Ein endlicher Körper $\mathbb{F}_q$ enthält primitive n-te Einheitswurzeln genau für $n \mid q - 1$.

Wir beweisen die wichtigste Eigenschaft von primitiven n-ten Einheitswurzeln:

8.3. **Lemma.** *Sei R ein Ring, seien $1 \leq \ell < n$, und sei $\omega \in R$ eine primitive n-te Einheitswurzel. Dann gilt*

(1) *$\omega^\ell - 1$ ist kein Nullteiler in R;*
(2) *$\sum_{j=0}^{n-1} \omega^{j\ell} = 0$.*

Beweis. Sei $g = \mathrm{ggT}(\ell, n) < n$. Dann gibt es einen Primteiler p von n mit $g \mid \frac{n}{p}$. Da $\omega^g - 1$ ein Teiler von $\omega^{n/p} - 1$ ist, ist $\omega^g - 1$ kein Nullteiler.

Jetzt schreiben wir $g = u\ell + vn$ mit $u, v \in \mathbb{Z}$. Dann ist

$$\omega^g - 1 = \omega^{u\ell}\omega^{vn} - 1 = \omega^{u\ell} - 1\,,$$

also ist $\omega^\ell - 1$ ein Teiler von $\omega^g - 1$ und damit ebenfalls kein Nullteiler.

Schließlich ist

$$(\omega^\ell - 1) \sum_{j=0}^{n-1} \omega^{j\ell} = \omega^{n\ell} - 1 = 0\,.$$

Da $\omega^\ell - 1$ kein Nullteiler ist, muss die Summe verschwinden. □

Sei im Folgenden R ein Ring, $n \geq 1$ und $\omega \in R$ eine primitive n-te Einheitswurzel. Wir identifizieren ein Polynom

$$a = a_{n-1}X^{n-1} + \cdots + a_1 X + a_0 \in R[X]$$

mit dem Vektor $(a_0, a_1, \ldots, a_{n-1}) \in R^n$.

8.4. **Definition.** Die R-lineare Abbildung

$$\mathrm{DFT}_\omega : R^n \longrightarrow R^n\,, \qquad a \longmapsto \big(a(1), a(\omega), a(\omega^2), \ldots, a(\omega^{n-1})\big)$$

heißt *Diskrete Fourier-Transformation (DFT)* (bezüglich ω).

8.5. **Definition.** Für zwei Polynome $a, b \in R^n$ definieren wir die *Faltung* als das Polynom

$$c = a * b = a *_n b = \sum_{j=0}^{n-1} c_j X^j \quad \text{mit} \quad c_j = \sum_{k+\ell \equiv j \bmod n} a_k b_\ell \,.$$

Wenn wir R^n mit $R[X]/(X^n - 1)R[x]$ identifizieren, dann ist die Faltung genau die Multiplikation in diesem Restklassenring.

Der Zusammenhang zwischen den beiden Definitionen wird aus dem folgenden Lemma deutlich.

8.6. **Lemma.** *Seien $a, b \in R^n$, $\omega \in R$ eine primitive n-te Einheitswurzel. Dann gilt*

$$\mathrm{DFT}_\omega(a * b) = \mathrm{DFT}_\omega(a) \bullet \mathrm{DFT}_\omega(b) \,.$$

(Dabei steht $\bullet$ wie oben für die komponentenweise Multiplikation.)

Beweis. Gilt $f \equiv g \bmod (X^n - 1)$, dann ist $f(\omega^j) = g(\omega^j)$, denn $(X^n - 1)(\omega^j) = \omega^{nj} - 1 = 0$. Damit gilt für die j-te Komponente des Vektors $\mathrm{DFT}_\omega(a * b)$:

$$\big(\mathrm{DFT}_\omega(a*b)\big)_j = (a*b)(\omega^j) = (a\cdot b)(\omega^j) = a(\omega^j)\cdot b(\omega^j) = \big(\mathrm{DFT}_\omega(a)\big)_j \cdot \big(\mathrm{DFT}_\omega(b)\big)_j \,.$$

□

Die lineare Abbildung DFT_ω ist sogar invertierbar.

8.7. **Satz.** *Sei R ein Ring, $n \geq 1$, $\omega \in R$ eine primitive n-te Einheitswurzel. Dann ist $\omega^{-1} = \omega^{n-1}$ ebenfalls eine primitive n-te Einheitswurzel, und es gilt für $a \in R^n$:*

$$\mathrm{DFT}_\omega\big(\mathrm{DFT}_{\omega^{-1}}(a)\big) = na \,.$$

Insbesondere ist DFT_ω invertierbar, und es gilt $\mathrm{DFT}_\omega^{-1} = n^{-1}\,\mathrm{DFT}_{\omega^{-1}}$.

Beweis. Es genügt, die Behauptung auf der Standardbasis nachzuprüfen. Es ist dann zu zeigen, dass $\sum_{j=0}^{n-1} \omega^{kj}\omega^{-mj} = n\delta_{k,m}$ ist. Nun gilt aber

$$\sum_{j=0}^{n-1} \omega^{kj}\omega^{-mj} = \sum_{j=0}^{n-1} \omega^{(k-m)j} = 0 \quad \text{für } k \neq m$$

(denn dann ist $k - m \not\equiv 0 \bmod n$) und

$$\sum_{j=0}^{n-1} \omega^{kj}\omega^{-kj} = \sum_{j=0}^{n-1} 1 = n \,.$$

Dass ω^{-1} eine primitive n-te Einheitswurzel ist, folgt aus Lemma 8.3. □

Es folgt, dass DFT_ω ein Ringisomorphimus zwischen $R[X]/(X^n - 1)R[X]$ und R^n (mit komponentenweisen Operationen) ist.

Um zwei Polynome a und b mit $\deg(ab) < n$ zu multiplizieren, können wir also so vorgehen: Wir wählen $N \geq n$ und eine primitive N-te Einheitswurzel ω. Dann ist

$$a \cdot b = N^{-1}\,\mathrm{DFT}_{\omega^{-1}}\big(\mathrm{DFT}_\omega(a) \bullet \mathrm{DFT}_\omega(b)\big) \,,$$

wobei wir die üblichen Identifikationen von R^N mit den Polynomen vom Grad $< N$ und mit dem Restklassenring $R[X]/(X^N-1)R[X]$ vorgenommen haben. Der Aufwand besteht in N Multiplikationen und den drei DFT-Berechnungen (plus Multiplikation mit N^{-1}). Die Komplexität wird dabei von den DFT-Berechnungen dominiert (alles andere ist linear in N).

8.8. **FFT — Fast Fourier Transform.** Wir lernen jetzt eine Möglichkeit kennen, DFT_ω sehr schnell zu berechnen, wenn $n = 2^k$ eine Potenz von 2 ist. Die Grundidee ist wiederum „Divide And Conquer“.

Sei zunächst nur vorausgesetzt, dass $n = 2m$ gerade ist, und sei $a \in R[X]$ ein Polynom vom Grad $< n$. Wir schreiben

$$a = a_1 X^m + a_0$$

mit Polynomen a_0, a_1 vom Grad $< m$. Dann gilt (unter Beachtung von $\omega^m = -1$: Es ist $(\omega^m+1)(\omega^m-1) = \omega^n - 1 = 0$, und $\omega^m - 1$ ist kein Nullteiler)

$$a(\omega^{2j}) = a_1(\omega^{2j})(\omega^m)^{2j} + a_0(\omega^{2j}) = (a_0 + a_1)(\omega^{2j})$$

und

$$a(\omega^{2j+1}) = a_1(\omega^{2j+1})(\omega^m)^{2j+1} + a_0(\omega^{2j+1}) = (a_0 - a_1)(\omega \cdot \omega^{2j})\,.$$

Wir setzen $r_0(X) = a_0(X) + a_1(X)$ und $r_1(X) = (a_0 - a_1)(\omega X)$. ω^2 ist eine primitive m-te Einheitswurzel. Damit ist die Berechnung von $\mathrm{DFT}_\omega(a)$ äquivalent zur Berechnung von $\mathrm{DFT}_{\omega^2}(r_0)$ und von $\mathrm{DFT}_{\omega^2}(r_1)$. Die Berechnung von r_0 und r_1 aus a erfordert dabei $n = 2m$ Additionen bzw. Subtraktionen und m Multiplikationen mit Potenzen von ω.

Wir erhalten folgenden Algorithmus. Wir nehmen dabei an, dass die Potenzen ω^j vorberechnet sind (das kostet einmalig 2^k Multiplikationen in R).

```
function fft(a, k, ω)
  // a = (a_0, ..., a_{2^k-1}) ∈ R^{2^k}, ω ∈ R primitive 2^k-te Einheitswurzel.
  // Ausgabe: DFT_ω(a) ∈ R^{2^k}.
  if k = 0 then
    return (a_0)
  else
    m ← 2^{k-1}
    for j = 0 to m − 1 do
      b_j ← a_j + a_{m+j}
      c_j ← (a_j − a_{m+j}) · ω^j
    end for
    (b_0, ..., b_{m-1}) ← fft((b_0, ..., b_{m-1}), k − 1, ω^2)
    (c_0, ..., c_{m-1}) ← fft((c_0, ..., c_{m-1}), k − 1, ω^2)
    return (b_0, c_0, b_1, c_1, ..., b_{m-1}, c_{m-1})
  end if
end function
```

Sei $T(k)$ der Aufwand für einen Aufruf $\mathrm{fft}(a, k, \omega)$. Dann gilt

$$T(k) = 2T(k-1) + 2^k\mathrm{A} + 2^{k-1}\mathrm{M} \quad \text{für } k \geq 1.$$

Dabei steht A wieder für Additionen und Subtraktionen und M für Multiplikationen in R (hier sind das nur Multiplikationen mit Potenzen von ω). Lemma 7.2 sagt uns dann, dass gilt

$$T(k) \in \big(\mathrm{A} + \tfrac{1}{2}\mathrm{M}\big)k2^k + O(2^k)\,.$$

8.9. **Folgerung.** *Sei R ein Ring, $\omega \in R$ eine primitive n-te Einheitswurzel mit $n = 2^k$. Dann kann man die Faltung zweier Polynome in $R[X]$ vom Grad $< n$ mit $\ll n \log n$ Operationen in R berechnen.*

Beweis. Wir verwenden folgenden Algorithmus:

```
function convolution(a, b, k, ω)
  // a, b ∈ R[X], deg a, deg b < 2^k.
  // Ausgabe: a *_{2^k} b ∈ R[X].
  // Wir identifizieren Polynome und ihre Koeffizientenvektoren.
  â ← fft(a, k, ω)
  b̂ ← fft(b, k, ω)
  n ← 2^k
  for j = 0 to n − 1 do
    ĉ_j ← â_j · b̂_j
  end for
  c ← fft(ĉ, k, ω^{−1})
  d ← n^{−1} // (in R)
  return d · c
end function
```

Der Aufwand dafür besteht in drei FFTs ($\ll n \log n$), der punktweisen Multiplikation von $\hat{a}$ und $\hat{b}$ ($\ll n$) und schließlich der Multiplikation des Ergebnisses mit d ($\ll n$). Die Komplexität der FFTs dominiert den Gesamtaufwand, der damit ebenfalls $\ll n \log n$ ist. □

8.10. **Satz.** *Sei R ein Ring, in dem es primitive 2^k-te Einheitswurzeln gibt für jedes k. Dann kann man zwei Polynome $a, b \in R[X]$ mit $\deg ab < n$ mit einem Aufwand von $\ll n \log n$ Operationen in R multiplizieren.*

Beweis. Wähle k mit $2^k \geq n$. Dann ist $a *_{2^k} b = a \cdot b$, und wir rufen einfach convolution(a, b, k) auf. Man beachte, dass (mit dem minimalen k) $2^k < 2n \ll n$ gilt. □

8.11. **„Three Primes"-FFT für die Multiplikation ganzer Zahlen.** Wir können Satz 8.10 benutzen, um einen schnellen Multiplikationsalgorithmus für ganze Zahlen zu konstruieren. Dieser wird zwar nur für ganze Zahlen beschränkter Länge funktionieren, aber die Beschränkung liegt weit jenseits dessen, was mit dem Computer überhaupt verarbeitbar ist.

Seien $a, b \in \mathbb{Z}_{>0}$ mit $\lambda(a), \lambda(b) \leq n$, und $B = 2^{64}$. Wir schreiben

$$a = \tilde{a}(B) \quad \text{und} \quad b = \tilde{b}(B)$$

mit Polynomen

$$\tilde{a} = a_{n-1}X^{n-1} + \cdots + a_1 X + a_0 \quad \text{und} \quad \tilde{b} = b_{n-1}X^{n-1} + \cdots + b_1 X + b_0\,,$$

wobei die Koeffizienten a_j, b_j Wortlänge haben, also in $[0, B-1]$ liegen. Für die Koeffizienten c_j des Produktes $\tilde{a} \cdot \tilde{b}$ gilt dann $c_j \leq n(B-1)^2$. Wir können drei Primzahlen p, q, r wählen mit $p, q, r \equiv 1 \bmod 2^{57}$ und $2^{63} < p, q, r < 2^{64}$, zum Beispiel $p = 95 \cdot 2^{57} + 1$, $q = 108 \cdot 2^{57} + 1$ und $r = 123 \cdot 2^{57} + 1$. Wir setzen voraus, dass $n(B-1)^2 < pqr$ (das gilt für $n \leq 2^{63}$; für eine solche Zahl wäre der Speicherplatzbedarf $2^{63} \cdot 8 = 2^{66}$ Byte — ein Gigabyte sind nur 2^{30} Byte!). Dann lässt sich c_j nach dem Chinesischen Restsatz aus c_j rem p, c_j rem q und c_j rem r

bestimmen. In den endlichen Körpern $\mathbb{F}_p$, $\mathbb{F}_q$, $\mathbb{F}_r$ gibt es jeweils primitive 2^{57}-te Einheitswurzeln, etwa $\omega_p = 55$, $\omega_q = 64$, $\omega_r = 493$. Wir können also mittels FFT Polynome vom Grad $< 2^{56}$ über diesen Körpern multiplizieren. Dies reduziert die Schranke für n auf 2^{56} (was immer noch weit ausreicht).

Das führt auf folgenden Algorithmus. Wir berechnen den Vektor

$$(u, v, w) \leftarrow \big(\mathrm{crt}((p,q,r),(1,0,0)), \mathrm{crt}((p,q,r),(0,1,0)), \mathrm{crt}((p,q,r),(0,0,1))\big)$$

mit $0 \le u, v, w < pqr$ ein für allemal; dann gilt für $x = (au + bv + cw)$ rem:

$$0 \le x < pqr\,, \quad \text{sowie} \quad x \equiv a \bmod p\,, \quad x \equiv b \bmod q \quad \text{und} \quad x \equiv c \bmod r\,.$$

```
function fast_multiply(a, b)
  // a, b ∈ Z_{>0} mit λ(a), λ(b) < 2^56.
  // Ausgabe: a · b.
  n ← max{λ(a), λ(b)}
  ã ← a_{n−1}X^{n−1} + ··· + a_0 wie oben
  b̃ ← b_{n−1}X^{n−1} + ··· + b_0 wie oben
  ã_p ← ã rem p; ã_q ← ã rem q; ã_r ← ã rem r
  b̃_p ← b̃ rem p; b̃_q ← b̃ rem q; b̃_r ← b̃ rem r
  k ← ⌈log_2 n⌉ + 1
  c̃_p ← convolution(ã_p, b̃_p, k, ω_p^{2^{57−k}})
  c̃_q ← convolution(ã_q, b̃_q, k, ω_q^{2^{57−k}})
  c̃_r ← convolution(ã_r, b̃_r, k, ω_r^{2^{57−k}})
  c̃ ← u · c̃_p + v · c̃_q + w · c̃_r
  c̃ ← c̃ rem pqr
  return c̃(2^64)
end function
```

Was kostet das? Die Reduktion von $\tilde{a}$ und $\tilde{b}$ modulo p, q, r kostet $\ll n$ Wortoperationen (pro Koeffizient ein Vergleich und eventuell eine Subtraktion, denn die Koeffizienten sind $< 2p$). Die Multiplikation als Faltung mittels FFT kostet jeweils $\ll k2^k \ll n \log n$ Wortoperationen (Operationen in $\mathbb{F}_p$ usw. sind in wenigen Wortoperationen zu erledigen, da $p, q, r < B$ sind). Die Rekonstruktion von $\tilde{c}$ kostet wieder einen Aufwand von $\ll n$ Wortoperationen, und das gleiche gilt für die Auswertung am Ende. Die Komplexität ist also

$$\ll n \log n \quad \text{Wortoperationen}\,.$$

Da wir n beschränkt haben, hat diese Aussage eigentlich keinen Sinn. In der Praxis ist dieser Algorithmus aber tatsächlich schneller etwa als Karatsuba für Zahlen einer Größe, die gelgentlich vorkommen kann.

8.12. **Schnelle Multiplikation in $R[X]$ für beliebige Ringe R.** Ist R ein beliebiger Ring (also zum Beispiel einer, der keine primitiven 2^k-ten Einheitswurzeln enthält), kann man wie folgt vorgehen. Wir nehmen erst einmal an, dass 2 in R invertierbar ist. Dann kann man sich eine künstliche primitive 2^k-te Einheitswurzel schaffen, indem in $R[y]$ modulo $y^{2^{k-1}} + 1$ rechnet. Wenn man das geschickt genug macht, verliert man nicht allzu viel, und man erhält einen Algorithmus, der Polynome vom Grad $< n$ in $\ll n \log n \log\log n$ Operationen in R multipliziert.

Wenn 2 nicht invertierbar ist, kann man immer noch, indem man die Multiplikation mit n^{-1} am Ende des Algorithmus im Beweis von Folgerung 8.9 weglässt, ein

Vielfaches $2^\kappa ab$ des Produktes von a und b berechnen. Analog gibt es eine „3-adische FFT“, mit deren Hilfe man $3^\lambda ab$ mit vergleichbarem Aufwand berechnen kann. Da 2^κ und 3^λ teilerfremd sind, kann man (schnell) $u, v \in \mathbb{Z}$ berechnen mit $u \cdot 2^\kappa + v \cdot 3^\lambda = 1$. Dann ist $ab = u \cdot 2^\kappa ab + v \cdot 3^\lambda ab$. Es folgt:

8.13. **Satz.** *Sei R ein beliebiger (kommutativer) Ring (mit 1). Es gibt einen Algorithmus, der das Produkt zweier Polynome in $R[X]$ vom Grad $< n$ mit*

$$\ll n \log n \log\log n$$

Operationen in R berechnet.

Wer genau wissen möchte, wie das geht, kann es in [GG, § 8] nachlesen.

Eine Variante (eigentlich die ursprüngliche Version) des oben angedeuteten Ansatzes führt auf das folgende berühmte Ergebnis:

8.14. **Satz (Schönhage und Strassen).** *Es gibt einen Algorithmus, der zwei ganze Zahlen a und b mit $\lambda(a), \lambda(b) \le n$ in*

$$\ll n \log n \log\log n$$

Wortoperationen berechnet.

Dieses Resultat ist hauptsächlich von theoretischem Interesse. Der Faktor $\log\log n$ wächst so langsam („The function $\log\log x$ tends to infinity, but has never been observed to do so“), dass er für praktische Zwecke als konstant anzusehen ist. Wir haben das oben bei dem „Three Primes“-Algorithmus gesehen.

8.15. **Schnelle Multiplikation in $\mathbb{Z}[X]$ und in $R[X, Y]$.**

Wir führen die „Soft-O“-Notation ein:

$$f(n) \in \tilde{O}(g(n))$$

soll bedeuten $f(n) \ll g(n)\big(\log g(n)\big)^k$ für eine Konstante k. Wir vernachlässigen also logarithmische Faktoren. Die obigen Sätze lassen sich dann so ausdrücken, dass man in $R[X]$ bzw. $\mathbb{Z}$ in $\tilde{O}(n)$ (Wort-)Operationen multiplizieren kann.

Analoge Aussagen gelten für Polynome über $\mathbb{Z}$ und für Polynome in mehreren Variablen. Um das zu sehen, kann man die *Kronecker-Substitution* verwenden. Seien zunächst $a, b \in R[X, Y]$ mit $\deg_X a, \deg_X b < m$ und $\deg_Y a, \deg_Y b < n$. Wenn wir $R[X, Y]$ als $R[X][Y]$ betrachten, dann gilt für die Koeffizienten von $c = a \cdot b$, dass ihr Grad (in X) kleiner ist als $2m - 1$. Wir können daher c aus

$$c(X, X^{2m-1}) = a(X, X^{2m-1}) \cdot b(X, X^{2m-1})$$

eindeutig rekonstruieren. Das Produkt auf der rechten Seite hat Faktoren vom Grad $< (2m-1)(n-1)+m \ll mn$ und kann in $\tilde{O}(mn)$ Operationen in R berechnet werden. Das Konvertieren zwischen $a(X, Y)$ und $a(X, X^{2m-1})$ usw. bedeutet dabei nur ein Umgruppieren der internen Darstellung; gerechnet wird dabei nicht. Formal kann man den Ansatz so interpretieren, dass man im Restklassenring

$$R[X, Y]/(Y - X^{2m-1})R[X, Y]$$

rechnet.

Seien jetzt $a, b \in \mathbb{Z}[X]$ mit $\deg a, \deg b < n$ und mit Koeffizienten, deren Länge $\le \ell$ ist. Dann ist die Länge der Koeffizienten c_j des Produkts $c = a \cdot b$ beschränkt

durch $2\ell + O(\log n)$. Mit $B = 2^{64}$ sei m so groß, dass $B^m > 2\max_j |c_j|$ ist; dann ist $m \ll \ell + \log n$. Analog wie eben gilt, dass wir c eindeutig (und „billig“) aus

$$c(B^m) = a(B^m) \cdot b(B^m)$$

rekonstruieren können. Die ganzen Zahlen im Produkt rechts haben Längen von höchstens $mn \ll n(\ell + \log n)$, also haben wir eine Komplexität von $\tilde{O}(\ell n)$. Hier rechnen wir in $\mathbb{Z}[X]/(X - B^m)\mathbb{Z}[X]$.

In der Praxis wird man eher modulo hinreichend vieler (FFT-)geeigneter Primzahlen reduzieren, dann multiplizieren, und das Ergebnis mit dem Chinesischen Restsatz zusammenbauen.

Als Gesamtergebnis lässt sich festhalten:

8.16. **Satz.** *Sei $R = \mathbb{Z}$, $\mathbb{Z}/N\mathbb{Z}$ oder $\mathbb{Q}$, und sei $m \geq 0$. Dann lassen sich Addition, Subtraktion und Multiplikation im Ring $R[X_1, \ldots, X_m]$ in $\tilde{O}$(Eingabelänge) Wortoperationen durchführen.*

Dabei gehen wir davon aus, dass die interne Darstellung der Polynome „dicht“ (*dense*) ist, d.h., die Eingabelänge ist von der Größenordnung $\prod_i \deg_{X_i} f$ mal Länge des größten Koeffizienten. Für „dünn“ (*sparse*) dargestellte Polynome gilt der Satz nicht.

8.17. **Übung.** Betrachten Sie folgende Darstellung von Polynomen in einer Variablen über $\mathbb{F}_2$: Das Polynom $f = \sum_j a_j X^j \in \mathbb{F}_2[X]$ wird dargestellt durch die aufsteigende Folge der Exponenten j mit $a_j = 1$ (d.h., $a_j \neq 0$). Sei $\ell(f)$ die Länge von f in dieser Darstellung, also die Anzahl der von null verschiedenen Terme in f. Zeigen Sie, dass die Komplexität der Multiplikation zweier Polynome $f, g \in \mathbb{F}_2[X]$ mindestens von der Ordnung $\ell(f)\ell(g)$ ist.

Wir müssen auch noch zeigen, dass man in $\mathbb{Z}/N\mathbb{Z}$ schnell rechnen kann. Dazu brauchen wir eine schnelle Division mit Rest.

(Für das schnelle Rechnen in $\mathbb{Q}$ brauchen wir auch einen schnellen ggT. Das wird später besprochen.)

9. Newton-Iteration

Die Newton-Iteration zur Approximation von Nullstellen ist aus der Analysis bekannt:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}\,.$$

Liegt der Startwert x_0 genügend nahe an einer Nullstelle der (differenzierbaren) Funktion f, dann konvergiert (x_n) gegen diese Nullstelle, und die Konvergenz ist sehr schnell („quadratisch“), wenn es eine einfache Nullstelle ist.

Das selbe Prinzip funktioniert auch in anderen metrischen Räumen als $\mathbb{R}$. Besonders übersichtlich wird es, wenn die Metrik *nicht-archimedisch* ist, also die verschärfte („ultrametrische“) Dreiecksungleichung erfüllt:

$$d(x, z) \leq \max\{d(x, y), d(y, z)\}$$

Ein Beispiel für so einen Raum ist $\mathbb{Q}$ mit der p-adischen Metrik für eine Primzahl p:

$$d(x, y) = |x - y|_p \quad \text{mit} \quad |x|_p = \begin{cases} 0 & \text{für } x = 0, \\ p^{-v_p(x)} & \text{für } x \neq 0. \end{cases}$$

Hier ist $v_p(x)$ definiert als die ganze Zahl n, so dass $x = p^n \frac{a}{b}$ ist mit $p \nmid a, b$.

Analog definiert man $v_X(f)$ für ein Polynom, eine rationale Funktion oder eine Potenzreihe f; mit $d(f,g) = 2^{-v_X(f-g)}$ erhält man wieder einen ultrametrischen Raum.

Allgemeiner definiert man den Begriff einer Bewertung auf einem Ring oder einem Körper.

9.1. **Definition.** Sei R ein Integritätsring. Eine Funktion $v : R \setminus \{0\} \to \mathbb{Z}$ heißt *(diskrete) Bewertung* auf R, wenn gilt:

(1) $v(xy) = v(x) + v(y)$ für alle $x, y \in R \setminus \{0\}$ und
(2) $v(x+y) \geq \min\{v(x), v(y)\}$ für alle $x, y \in R \setminus \{0\}$ mit $x + y \neq 0$.

Die Bewertung heißt *trivial*, wenn $v(R \setminus \{0\}) = \{0\}$ und *normiert*, wenn $1 \in v(R \setminus \{0\})$.

Ein Integritätsring R zusammen mit einer normierten diskreten Bewertung, der die Bedingung $v(x) = 0 \Longrightarrow x \in R^\times$ erfüllt, heißt *diskreter Bewertungsring*.

Die p-adische Bewertung v_p auf $\mathbb{Q}$ und die X-adische Bewertung v_X auf $K(X)$ sind normierte diskrete Bewertungen.

Man setzt gerne $v(0) = \infty$; dann gelten die beiden Eigenschaften in der Definition für *alle* $x, y \in K$ (mit den üblichen Rechenregeln $n + \infty = \infty$ und $n < \infty$ für $n \in \mathbb{Z}$).

Eigenschaft (2) hat folgende Ergänzung:

$$v(x+y) = \min\{v(x), v(y)\} \quad \text{falls } v(x) \neq v(y).$$

Sei etwa $v(x) < v(y)$. Wir nehmen an, $v(x+y) > v(x)$. Wegen $v(-y) = v(y)$ (s.u.) würde dann folgen: $v(x) \geq \min\{v(-y), v(x+y)\} = \min\{v(y), v(x+y)\} > v(x)$, ein Widerspruch.

Sei $a > 1$ und v eine Bewertung auf dem Integritätsring R. Dann definiert

$$d(x,y) = a^{-v(x-y)} \quad \text{für } x \neq y, \qquad d(x,x) = 0$$

eine Metrik auf R, die die verschärfte Dreiecksungleichung erfüllt.

9.2. **Lemma.** *Sei K ein Körper mit normierter diskreter Bewertung v. Dann ist*

$$R = \{\alpha \in K \mid v(\alpha) \geq 0\}$$

ein diskreter Bewertungsring mit Einheitengruppe $R^\times = \{\alpha \in K \mid v(\alpha) = 0\}$. Das Ideal $M = \{\alpha \in K \mid v(\alpha) > 0\}$ ist das einzige maximale Ideal in R.

R heißt auch der *Bewertungsring* von K.

Beweis. Aus den Eigenschaften einer Bewertung folgt, dass $v(1) = 0$, und dass R unter Addition und Multiplikation abgeschlossen ist. Außerdem ist $0 = v(1) = v((-1)^2) = 2v(-1)$, also ist $-1 \in R$; damit ist R auch unter Negation abgeschlossen und damit ein Unterring von K ($0 \in R$ ist klar). Für $\alpha \in K^\times$ gilt $v(\alpha^{-1}) = -v(\alpha)$; es folgt

$$\alpha \in R^\times \iff \alpha \in R \wedge \alpha^{-1} \in R \iff v(\alpha) \geq 0 \wedge v(\alpha) \leq 0 \iff v(\alpha) = 0\,.$$

Es ist klar, dass M ein Ideal in R ist; wegen $M = R \setminus R^\times$ ist es auch das einzige (denn kein echtes Ideal kann eine Einheit enthalten; dann ist es aber in M enthalten). □

Umgekehrt gilt: Ist (R, v) ein diskreter Bewertungsring, dann lässt sich die Bewertung v von R auf den Quotientenkörper K von R fortsetzen (mittels $v(a/b) = v(a) - v(b)$), und K wird dadurch ein diskret bewerteter Körper mit Bewertungsring R.

Sei nun (R, v) ein diskreter Bewertungsring; $f \in R[X]$ sei ein Polynom. Sei weiter $x_0 \in R$ mit $v(f(x_0)) > 0$ und $v(f'(x_0)) = 0$. (Dabei ist f' die formal mit Hilfe der üblichen Ableitungsregeln definierte Ableitung des Polynoms f.) Wir setzen wie aus der Analysis bekannt

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}\,.$$

9.3. **Lemma.** *Es gilt $f'(x_n) \in R^\times$, damit $x_{n+1} \in R$, sowie $v(f(x_{n+1})) \geq 2v(f(x_n))$ und $v(x_{n+1} - x_n) = v(f(x_n))$.*

Beweis. Der Beweis geschieht durch Induktion. Die Behauptung $f'(x_n) \in R^\times$ stimmt für $n = 0$ nach Voraussetzung. Wir nehmen an, sie gelte für n. Wir schreiben

$$f(X) = f(x_n) + (X - x_n)f'(x_n) + (X - x_n)^2 g_n(X)$$

mit einem Polynom $g_n \in R[X]$. Wenn wir x_{n+1} einsetzen, erhalten wir

$$f(x_{n+1}) = f(x_n) + (x_{n+1} - x_n)f'(x_n) + (x_{n+1} - x_n)^2 g_n(x_{n+1}) = (x_{n+1} - x_n)^2 g_n(x_{n+1})$$

nach Definition von x_{n+1}. Die Aussage $v(x_{n+1} - x_n) = v(f(x_n))$ folgt aus der Definition mit $v(f'(x_n)) = 0$. Damit gilt

$$v(f(x_{n+1})) \geq 2v(x_{n+1} - x_n) = 2v(f(x_n))\,.$$

Außerdem ist

$$f'(X) = f'(x_n) + (X - x_n)h_n(X)$$

mit einem Poynom $h_n \in R[X]$; damit ist $f'(x_{n+1}) = f'(x_n) + (x_{n+1} - x_n)h_n(x_{n+1})$; wegen $v(f'(x_n)) = 0$ und $v(x_{n+1} - x_n) > 0$ folgt $v(f'(x_{n+1})) = 0$. □

Im diskret bewerteten Körper $K(X)$ (mit der X-adischen Bewertung v_X) besteht der Bewertungsring R aus den rationalen Funktionen, die in 0 definiert sind, und das maximale Ideal aus den rationalen Funtktionen, die in 0 eine Nullstelle haben. Allgemein gilt für zwei Elemente $a, b \in R$:

$$v(a - b) \geq n \iff a \equiv b \bmod M^n\,.$$

In $K(X)$ gilt, dass eine rationale Funktion in R modulo M^n kongruent ist zum Anfangsstück der Länge n ihrer Taylorreihe in 0 (die man ganz formal definieren kann); damit ist R/M^n isomorph zu $K[X]/X^nK[X]$, und wir können uns auf das Rechnen mit Polynomen beschränken.

Wir werden die Newton-Iteration jetzt benutzen, um das Inverse eines Polynoms $a \in K[X]$ modulo X^n zu berechnen; dabei nehmen wir an, dass $a(0) = 1$ ist. (Wir brauchen $a(0) \neq 0$, damit a in $K[X]/X^nK[X]$ invertierbar ist; dann können wir a entsprechend skalieren.)

Wir wollen die Gleichung

$$f(Y) = 1/Y - a = 0$$

lösen. Der entsprechende Newton-Iterationsschritt sieht dann so aus:

$$b_{n+1} = b_n - \frac{f(b_n)}{f'(b_n)} = b_n - \frac{b_n^{-1} - a}{-b_n^{-2}} = b_n + b_n(1 - ab_n) = 2b_n - ab_n^2\,.$$

Hier ist f kein Polynom, deswegen können wir Lemma 9.3 nicht anwenden. Man sieht aber direkt, dass

$$1 - ab_{n+1} = (1 - ab_n)^2 \,,$$

so dass aus $ab_n \equiv 1 \bmod X^k$ folgt, dass $ab_{n+1} \equiv 1 \bmod X^{2k}$. Mit $b_0 = 1$ (dann gilt $ab_0 \equiv 1 \bmod X$) kann man die Iteration starten. Wir erhalten folgenden Algorithmus.

```
function invert(a, n)
  // a ∈ K[X] mit a(0) = 1, n ≥ 1.
  // Ausgabe: b ∈ K[X] mit deg b < n und ab ≡ 1 mod X^n.
  k ← 1
  b ← 1 ∈ K[X]
  while k < n do
    k ← min{2k, n}
    b ← (2b − a · b^2) rem X^k
  end while
  return b
end function
```

Wir wissen, dass $b_{j+1} \equiv b_j \bmod X^{2^j}$ ist. Das bedeutet, dass in $b \leftarrow (2b - a \cdot b^2) \operatorname{rem} X^k$ nur die obere Hälfte der rechten Seite berechnet werden muss, die dann die obere Hälfte von b wird (und gleich der oberen Hälfte von $-a \cdot b^2$ ist).

Die Kosten dafür sind für jedes $k = 1, 2, 4, 8, \ldots$ zwei Multiplikationen von Polynomen der Länge $\le \min\{2k, n\}$ (man beachte, dass man nur mit den Anfangsstücken der Polynome rechnen muss), dazu kommt noch linearer Aufwand. Wenn wir die schnelle Multiplikation verwenden, dann ist der Aufwand beschränkt durch

$$\ll \sum_{j=1}^{\lceil \log_2 n \rceil} 2^j j \log j \ll 2^{\lceil \log_2 n \rceil} \lceil \log_2 n \rceil \log \lceil \log_2 n \rceil \ll n \log n \log \log n \,,$$

also von der gleichen Größenordnung wie die Multiplikation. Erlaubt K direkt die FFT, dann fällt der Faktor $\log \log n$ noch weg. In jedem Fall haben wir Komplexität $\tilde{O}(n)$.

9.4. **Schnelle Division mit Rest.** Wir könne diese schnelle Approximation des Inversen benutzen, um auch die Division mit Rest in $R[X]$ zu beschleunigen. Dazu führen wir zunächt eine Operation ein, die ein Polynom „auf den Kopf stellt“:

9.5. **Definition.** Sei $a \in R[X]$, $a = \sum_{j \ge 0} a_j X^j$, und $n \ge 0$. Dann setzen wir

$$\operatorname{rev}_n(a) = \sum_{j=0}^{n} a_{n-j} X^j \,.$$

Ist $\deg a \le n$, dann ist $\operatorname{rev}_n(a) = X^n a(X^{-1})$.

Seien nun $a, b \in R[X]$ mit $\deg a = n \ge m = \deg b$ und $b_m = 1$. Dann sind Quotient q und Rest r mit $a = qb + r$ und $\deg r < m$ für jeden Koeffizientenring R eindeutig bestimmt. Aus der Gleichung folgt

$$X^n a\Big(\frac{1}{X}\Big) = X^{n-m} q\Big(\frac{1}{X}\Big) \cdot X^m b\Big(\frac{1}{X}\Big) + X^{n-m+1} \cdot X^{m-1} r\Big(\frac{1}{X}\Big) \,,$$

also

$$\operatorname{rev}_n(a) = \operatorname{rev}_{n-m}(q) \cdot \operatorname{rev}_m(b) + X^{n-m+1} \operatorname{rev}_{m-1}(r) \,.$$

Insbesondere haben wir

$$\mathrm{rev}_n(a) \equiv \mathrm{rev}_{n-m}(q) \cdot \mathrm{rev}_m(b) \bmod X^{n-m+1}\,.$$

Diese Kongruenz bestimmt $\mathrm{rev}_{n-m}(q)$ (und damit auch q) eindeutig. Um den Quotienten zu berechnen, bestimmen wir das Inverse von $\mathrm{rev}_m(b)$ mod X^{n-m+1} (beachte: $\mathrm{rev}_m(b)(0) = 1$) und multiplizieren mit a. Wir erhalten folgenden Algorithmus.

```
function quotrem(a, b)
  // a, b ∈ R[X], b mit Leitkoeffizient 1.
  // Ausgabe (q, r) ∈ R[X] × R[X] mit a = qb + r und deg r < m.
  n ← deg a
  m ← deg b
  if n < m then return (0, a) end if
  ã ← rev_n(a)
  b̃ ← rev_m(b)
  q̃ ← invert(b̃, n − m + 1)
  q̃ ← (q̃ · ã) rem X^{n−m+1}
  q ← rev_{n−m}(q̃)
  r ← a − q · b
  return (q, r)
end function
```

Der Aufwand hierfür besteht in einer Inversion mod X^{n-m+1}, zwei Multiplikationen der Länge $n-m+1$ bzw. n, sowie linearem (in n) Aufwand für das Umsortieren der Koeffizienten und die Subtraktion am Ende. Insgesamt ist der Aufwand nur um einen konstanten Faktor teurer als eine Multiplikation. Genauer brauchen wir von dem Produkt $q \cdot b$ nur den unteren Teil (mod X^m), denn wir wissen, dass $\deg r < m$ ist. Das reduziert den Aufwand auf

$$\begin{aligned}&\ll (n-m+1)\log(n-m+1)\log\log(n-m+1) + m\log m\log\log m\\ &\ll n\log n\log\log n \in \tilde{O}(n)\,.\end{aligned}$$

Als Folgerung erhalten wir:

9.6. **Satz.** *Sei R ein beliebiger kommutativer Ring mit 1, $a \in R[X]$ mit Leitkoeffizient 1 und Grad n. Dann lassen sich die Ringoperationen im Restklassenring $R[X]/aR[X]$ mit einem Aufwand von $\ll n\log n\log\log n$ (oder $\tilde{O}(n)$) Operationen in R durchführen.*

Beweis. Wir repräsentieren die Element von $R[X]/aR[X]$ durch den eindeutig bestimmten Repräsentanten der Restklasse vom Grad $< n$. Addition und Subtraktion finden dann mit diesen Repräsentanten statt und benötigen je n Operationen in R. Für die Multiplikation benutzen wir $(b \cdot c)$ rem a mit der schnellen Multiplikation und Division in $R[X]$; der Aufwand dafür ist wie angegeben. □

Zusammen mit dem entsprechenden Ergebnis für Restklassenringe $\mathbb{Z}/N\mathbb{Z}$ folgt, dass man die Ringoperationen im endlichen Körper $\mathbb{F}_{p^n}$ mit einem Aufwand von $\tilde{O}(\lambda(p^n))$ Wortoperationen durchführen kann, denn $\mathbb{F}_{p^n} = \mathbb{F}_p[X]/a\mathbb{F}_p[X]$ für ein (beliebiges) irreduzibles Polynom $a \in \mathbb{F}_p[X]$ vom Grad n.

9.7. **Schnelle Division mit Rest in $\mathbb{Z}$.** Der oben für $R[X]$ verwendete Ansatz, die Polynome umzudrehen, lässt sich in $\mathbb{Z}$ nicht nutzen (weil die Überträge die Symmetrie zerstören). Statt dessen verwendet man Newton-Iteration in $\mathbb{R}$, um eine hinreichend gute Approximation von $1/b$ zu bestimmen. Man skaliert dabei mit einer geeigneten Potenz von $B = 2^{64}$, um nicht mit (Dual-)Brüchen rechnen zu müssen. Die Details sind etwas verwickelt, aber am Ende bekommt man einen Divisionsalgorithmus, der wiederum nicht teurer ist als eine Multiplikation (bis auf einen konstanten Faktor).

Es folgt:

9.8. **Satz.** *Sei $N \in \mathbb{Z}_{>0}$. Dann lassen sich die Ringoperationen in $\mathbb{Z}/N\mathbb{Z}$ mit einem Aufwand von $\ll n \log n \log\log n$ (oder $\tilde{O}(n)$) Wortoperationen durchführen, wo $n = \lambda(N)$ ist.*

Der Beweis ist völlig analog zu Satz 9.6.

9.9. **Inverse modulo p^n.** Man kann die Berechnung des Inversen mod X^n verallgemeinern auf die Berechnung von Inversen modulo p^n, wenn ein Inverses mod p bekannt ist. Sei dafür R ein beliebiger Ring und a rem b ein Element c von R mit $c \equiv a \bmod b$.

```
function invnewton(a, b, p, n)
  // a, b, p ∈ R mit ab ≡ 1 mod p, n ∈ Z_{>0}.
  // Ausgabe: c ∈ R mit ac ≡ 1 mod p^n.
  k ← 1
  c ← b
  while k < n do
    // hier gilt ac ≡ 1 mod p^k
    k ← min{2k, n}
    c ← (2c − ac^2) rem p^k
  end while
  return c
end function
```

Der Beweis dafür, dass das funktioniert, ist analog zum Beweis im Fall $R = K[X]$, $p = X$. Der Aufwand ist $\tilde{O}(n \deg p)$ Operationen im Koeffizientenring, wenn R ein Polynomring ist und $\tilde{O}(n\lambda(p))$ für $R = \mathbb{Z}$ (unter der Annahme, dass $\deg a < n \deg p$ bzw. $|a| < p^n$; anderenfalls fallen noch Kosten an für das Reduzieren von a mod p^n).

Als Korollar erhalten wir:

Seien R ein kommutativer Ring mit 1, $p \in R$, $n \in \mathbb{Z}_{>0}$. Dann ist $a \in R$ genau dann modulo p^n invertierbar, wenn a modulo p invertierbar ist.

Die eine Richtung ist trivial ($ab \equiv 1 \bmod p^n \implies ab \equiv 1 \bmod p$), die andere wird durch den obigen Algorithmus geliefert.

9.10. **Berechnung der p-adischen Darstellung.** Wir können die schnelle Division mit Rest dazu verwenden, ein gegebenes Ringelement a in der Form $a = a_0 + a_1 p + \cdots + a_n p^n$ darzustellen. Wir formulieren dies erst einmal für Polynome. Sei R ein kommutativer Ring mit 1.

```
function gentaylor(a, p)
  // a, p ∈ R[X], deg p > 0, lcf p = 1.
```

```
  // Ausgabe: (a_0, a_1, ..., a_n) mit a = a_0 + a_1 p + ... a_n p^n, deg a_j < deg p.
  if deg a < deg p then
    return (a)
  else
    n ← ⌊(deg a)/(deg p)⌋ // n ≥ 1
    k ← ⌈n/2⌉
    P ← p^k // durch sukzessives Quadrieren
    (q, r) ← quotrem(a, P)
    (a_0, ..., a_{k-1}) ← gentaylor(r, p)
    (a_k, ..., a_n) ← gentaylor(q, p)
    return (a_0, ..., a_{k-1}, a_k, ..., a_n).
  end if
end function
```

Der Name „gentaylor“ bezieht sich darauf, dass dies eine Verallgemeinerung der Taylorentwicklung im Punkt x_0 liefert (die bekommt man mit $p = X - x_0$).

Wie teuer ist das? Die Analyse ist einfacher, wenn $n = 2^m - 1$ ist, denn dann sind die auftretenden Werte von k stets von der Form 2^l, und das Polynom wird in zwei gleich große Teile geteilt (nach dem bewährten Prinzip).

Die Berechnung von p^k durch sukzessives Quadrieren und mit schneller Multiplikation kostet eine Multiplikation der Länge $k \deg p$ (für das letzte Quadrieren) plus die Kosten für die Berechnung von $p^{k/2}$. Für $k = 2^l$ ist das

$$\begin{aligned}
&\ll \sum_{j=1}^{l} (2^j \deg p) \log(2^j \deg p) \log\log(2^j \deg p) \\
&\ll (\deg p) \sum_{j=1}^{l} 2^j \log(2^l \deg p) \log\log(2^l \deg p) \\
&\ll k \deg p \log(k \deg p) \log\log(k \deg p) \in \tilde{O}(k \deg p)\,.
\end{aligned}$$

Für beliebiges k ist der Aufwand höchstens um $O(k \deg p)$ größer, durch die zusätzlichen Multiplikationen der Form $p \cdot p^l$.

Die Berechnung von Quotient und Rest mit der schnellen Methode kostet ebenfalls

$$\ll 2^l \deg p \log(2^l \deg p) \log\log(2^l \deg p) \quad \text{Operationen in } R.$$

Für den Gesamtaufwand beachten wir, dass wir 2^t Aufrufe mit $n = 2^{m-t} - 1$, also $k = 2^{m-t-1}$, generieren. Das liefert die Abschätzung

$$\begin{aligned}
&\ll \sum_{t=0}^{m-1} 2^t \cdot 2^{m-t-1} \deg p \log(2^{m-t-1} \deg p) \log\log(2^{m-t-1} \deg p) \\
&\ll 2^m (\deg p) m \log(2^m \deg p) \log\log(2^m \deg p) \\
&\ll n (\deg p) \log n \log(n \deg p) \log\log(n \deg p) \in \tilde{O}(n \deg p) \in \tilde{O}(\deg a)\,.
\end{aligned}$$

Man verliert also einen Faktor $\log n$ gegenüber den Kosten einer schnellen Multiplikation.

Analog erhält man einen Algorithmus, der eine positive ganze Zahl a in der Basis $b \geq 2$ darstellt, mit Aufwand $\tilde{O}(\lambda(a))$. Das ist zum Beispiel dann nützlich, wenn man die intern binär dargestellten Zahlen in Dezimalschreibweise ausgeben möchte.

9.11. **Newton-Iteration ohne Division.** In der üblichen Iterationsvorschrift

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

muss durch $f'(x_n)$ geteilt werden. Wir können das umgehen, indem wir das Inverse von $f'(x_n)$ modulo einer geeigneten Potenz des betrachteten Ideals ebenfalls durch Newton-Iteration mitberechnen. Das führt auf folgende Variante des Algorithmus.

```
function newton(f, a, b, p, n)
  // f ∈ R[X], a, b, p ∈ R, n ∈ Z_{>0} mit f(a) ≡ 0 mod p und b · f'(a) ≡ 1 mod p.
  // Ausgabe: c ∈ R mit f(c) ≡ 0 mod p^n und c ≡ a mod p.
  k ← 1, P ← p // P = p^k
  c ← a; d ← b
  while 2k < n do
    k ← 2k; P ← P^2
    c ← (c − f(c) · d) rem P
    d ← (2d − f'(c) · d^2) rem P
  end while
  c ← (c − f(c) · d) rem p^n
  return c
end function
```

Zum Beweis der Korrektheit zeigt man, dass zu Beginn jedes Durchlaufs durch die while -Schleife folgende Aussagen gelten:

$$c \equiv a \bmod p\,, \quad P = p^k\,, \quad f(c) \equiv 0 \bmod P\,, \quad d \cdot f'(c) \equiv 1 \bmod P\,.$$

Zu Beginn ist das klar auf Grund der gemachten Voraussetzungen. In der Zuweisung

$$c \leftarrow \big(c - f(c) \cdot d\big) \text{ rem } P$$

gilt dann $f(c_{\text{alt}}) \equiv 0 \bmod P_{\text{alt}}$ und $d_{\text{alt}} \equiv f'(c_{\text{alt}})^{-1} \bmod P_{\text{alt}}$, also ist (wegen $P_{\text{neu}} = P_{\text{alt}}^2$)

$$f(c_{\text{alt}}) \cdot d_{\text{alt}} \equiv f(c_{\text{alt}}) \cdot f'(c_{\text{alt}})^{-1} \bmod P_{\text{neu}}\,.$$

Damit gilt für das neue c:

$$c_{\text{neu}} \equiv c_{\text{alt}} - \frac{f(c_{\text{alt}})}{f'(c_{\text{alt}})} \bmod P_{\text{neu}}$$

und damit nach Lemma 9.3, dass $f(c_{\text{neu}}) \equiv 0 \bmod P_{\text{neu}}$. Außerdem ist $c_{\text{neu}} \equiv c_{\text{alt}} \bmod P_{\text{alt}}$ (das zeigt auch, dass $c_{\text{neu}} \equiv a \bmod p$ ist); das impliziert

$$f'(c_{\text{neu}}) \equiv f'(c_{\text{alt}}) \bmod P_{\text{alt}}\,, \quad \text{also} \quad d_{\text{alt}} \cdot f'(c_{\text{neu}}) \equiv 1 \bmod P_{\text{alt}}\,.$$

Wie vorher sieht man dann

$$d_{\text{neu}} \cdot f'(c_{\text{neu}}) \equiv 1 \bmod P_{\text{neu}}\,.$$

Damit ist gezeigt, dass die obigen Aussagen am Ende des Schleifenrumpfs wieder erfüllt sind. Analog gilt nach der letzten Zuweisung dann

$$c \equiv a \bmod p \quad \text{und} \quad f(c) \equiv 0 \bmod p^n$$

wie gewünscht.

Die Kosten für eine Iteration $c \leftarrow \big(c - f(c) \cdot d\big)$ rem P (und analog für d) setzen sich zusammen aus den Kosten für die Berechnung von $f(c)$ rem P, plus einer weiteren Multiplikation mod P und einer Subtraktion. Um $f(c)$ rem P zu berechnen, können wir das Horner-Schema (siehe Abschnitt 6.10) verwenden. Das erfordert $\deg f$ Multiplikationen mod P und $\deg f$ Additionen. Der Aufwand für

einen Iterationsschritt ist also $\ll (\deg f)$ mal der Aufwand für eine Multiplikation mod P. Die Kosten für den letzten Schritt dominieren die Kosten für die vorherigen Schritte. Damit ist der Aufwand

$$\ll (\deg f) M(p^n)\,,$$

wo $M(p^n)$ für die Kosten für eine Multiplikation von Elementen der Größe von p^n steht.

9.12. **Berechnung von n-ten Wurzeln in $\mathbb{Z}$.** Ein typischer Anwendungsfall der Newton-Iteration ist das Berechnen exakter n-ter Wurzeln aus ganzen Zahlen. Dabei wollen wir 2-adisch rechnen, weil das optimal zu der internen Darstellung der Zahlen passt. Es soll also die positive ganzzahlige Nullstelle von $f(X) = X^n - a$ berechnet werden, falls sie existiert; dabei sei $a \in \mathbb{Z}_{>0}$. Die Ableitung ist $f'(X) = nX^{n-1}$. Ist $b \in \mathbb{Z}$ mit $b^n \equiv a \bmod 2$, dann ist $f'(b) \equiv na \bmod 2$. Damit wir den Algorithmus 9.11 anwenden können, muss $na \bmod 2$ invertierbar sein. Deshalb nehmen wir erst einmal an, dass n ungerade ist. Den geraden Anteil von a können wir abspalten und dann annehmen, dass auch a ungerade ist. Wir erhalten folgenden Algorithmus.

```
function nthroot(a, n)
  // a ∈ Z_{>0}, n ∈ Z_{>0} ungerade.
  // Ausgabe: false, wenn b^n = a in Z nicht lösbar ist;
  //          (true, b) mit b ∈ Z_{>0} und b^n = a sonst.
  k ← v_2(a)
  if k rem n ≠ 0 then return false end if
  a ← a/2^k // jetzt ist a ungerade
  m ← 1 + ⌊(log_2 a)/n⌋
  b ← newton(X^n − a, 1, 1, 2, m)
  if b^n = a then
    return (true, b · 2^{k quo n})
  else
    return false
  end if
end function
```

Zum Beweis der Korrektheit brauchen wir noch ein Lemma:

9.13. **Lemma.** *Seien $f \in R[X]$, $a, a', p \in R$ und $n \in \mathbb{Z}_{>0}$ mit $a' \equiv a \bmod p$ und $f(a) \equiv f(a') \equiv 0 \bmod p^n$. Wenn $f'(a)$ in R/pR invertierbar ist, dann gilt $a' \equiv a \bmod p^n$.*

Das sagt also, dass es zu jeder Nullstelle $\alpha \bmod p$ von f genau eine Nullstelle $a \bmod p^n$ von f gibt mit $a \equiv \alpha \bmod p$.

Beweis. Wie vorher schreiben wir $f(X) = f(a) + (X-a)f'(a) + (X-a)^2 g(X)$ mit $g \in R[X]$. Wir setzen $X = a'$ und erhalten

$$0 \equiv f(a') = f(a) + (a'-a)f'(a) + (a'-a)^2 g(a') \equiv (a'-a)\big(f'(a) + (a'-a)g(a')\big) \bmod p^n\,.$$

Es gilt

$$f'(a) + (a'-a)g(a') \equiv f'(a) \bmod p\,,$$

also ist $f'(a) + (a'-a)g(a') \bmod p$ und damit auch mod p^n invertierbar. Sei $b \in R$ mit $b\big(f'(a) + (a'-a)g(a')\big) \equiv 1 \bmod p^n$. Wir multiplizieren mit b und bekommen $a' - a \equiv 0 \bmod p^n$, wie gewünscht. □

Nun zum Beweis der Korrektheit von „nthroot“. Wir schreiben zunächst $a = 2^k a'$ mit a' ungerade. Wenn $a = b^n$, dann muss k ein Vielfaches von n sein. Wenn also $k \operatorname{rem} n \neq 0$ ist, dann kann a keine $n$te Potenz sein. Wenn k ein Vielfaches von n ist, dann ist a genau dann eine $n$te Potenz, wenn a' eine ist; gilt $(b')^n = a'$, dann ist $b = b' 2^{k/n}$ eine $n$te Wurzel von a.

Der Aufruf $\operatorname{newton}(X^n - a, 1, 1, 2, 1 + \lfloor (\log_2 a)/n \rfloor)$ berechnet die nach Lemma 9.13 eindeutige Nullstelle mod 2^m (mit $m = 1 + \lfloor (\log_2 a)/n \rfloor$) von $f(X) = X^n - a$. Die Voraussetzungen sind erfüllt, denn $f(1) = 1 - a \equiv 0 \bmod 2$ und $1 \cdot f'(1) = n \equiv 1 \bmod 2$. (Beachte, dass f nur die eine Nullstelle 1 mod 2 besitzt.) Gilt $b^n = a$, dann ist offensichtlich b eine $n$te Wurzel von a. Gilt umgekehrt, dass a eine $n$te Wurzel c in $\mathbb{Z}_{>0}$ hat, dann gilt $c \equiv b \bmod 2^m$ und

$$c^n = a = 2^{\log_2 a} \implies 0 < c = 2^{(\log_2 a)/n} < 2^m .$$

Da auch $0 < b < 2^m$, folgt $b = c$, also $b^n = a$, und wir erhalten das richtige Ergebnis.

Die Kosten für den Aufruf von „newton“ sind

$$\ll nM(2^m) \ll nM(a^{1/n}) \ll M(a) \in \tilde{O}(\lambda(a)) \text{ Wortoperationen} .$$

Dazu kommt die Berechnung von b^n, mit vergleichbaren Kosten.

Man kann noch etwas Rechenzeit sparen (ohne allerdings eine bessere asymptotische Komplexität zu erreichen), wenn man eine spezielle Version von „newton“ für $f = X^n - a$ verwendet, in der man jeweils c^{n-1} mitberechnet und das dann für die Auswertung $f(c) = c \cdot c^{n-1} - a$ und $f'(c) = nc^{n-1}$ verwendet.

9.14. **Berechnung von Quadratwurzeln in $\mathbb{Z}$.** Für Quadratwurzeln funktioniert der obige Ansatz nicht direkt, weil die Ableitung von $X^2 - a$ modulo 2 niemals invertierbar ist. Wir können aber wie vorher annehmen, dass a ungerade ist. Dann kann a höchstens dann ein Quadrat sein, wenn $a \equiv 1 \bmod 8$ ist, denn

$$(4k \pm 1)^2 = 16k^2 \pm 8k + 1 \equiv 1 \bmod 8 .$$

Wenn a Quadratzahl ist, dann ist eine der beiden Quadratwurzeln $\equiv 1 \bmod 4$. Wir schreiben also $a = 8A + 1$ und suchen nach einer Nullstellen von $(4X + 1)^2 - (8A + 1) = 8(2X^2 + X - A)$. Mit $f(X) = 2X^2 + X - A$ gilt $f'(X) = 4X + 1$, und das ist stets ungerade. Das liefert folgenden Algorithmus:

```
function sqrt(a)
  // a ∈ Z_{>0}.
  // Ausgabe: false, wenn a kein Quadrat ist;
  //          (true, b) mit b ∈ Z_{>0} und b^2 = a sonst.
  k ← v_2(a)
  if k rem 2 ≠ 0 then return false end if
  a ← a/2^k
  if a rem 8 ≠ 1 then return false end if
  m ← ⌊(log_2 a)/2⌋ − 1
  A ← a quo 8
  if m ≤ 0 then return (true, (2A + 1)2^{k/2}) end if // a = 1 oder 9
  B ← newton(2X^2 + X − A, A rem 2, 1, 2, m)
  b ← 4B + 1
  if b^2 = a then return (true, b · 2^{k/2}) end if
  b ← 2^{m+2} − b
  if b^2 = a then return (true, b · 2^{k/2}) end if
```

```
  return false
end function
```

Wie vorher ist klar, dass a ein Quadrat ist, wenn das Ergebnis true ist. Ist umgekehrt a ein Quadrat und a ungerade, dann gibt es ein eindeutiges $c = 4C + 1$ mit $c^2 = a$. Der Aufruf von „newton“ liefert die eindeutige Nullstelle von $2X^2 + X - A$ mod 2^m, also gilt $C \equiv B \bmod 2^m$ und damit $c \equiv b \bmod 2^{m+2}$. In jedem Fall gilt

$$c^2 = a = 2^{\log_2 a} \implies |c| = 2^{(\log_2 a)/2} < 2^{m+2}\,.$$

Ist c positiv, dann folgt $c = b$, und die erste Abfrage „$b^2 = a$?“ ist erfüllt. Ist c negativ, dann folgt $c = b - 2^{m+2}$, also ist die zweite Abfrage „$b^2 = a$?“ erfüllt. Wir sehen, dass der Algorithmus die Quadratwurzel berechnet, wenn sie existiert. Die Kosten sind wie vorher $\in \tilde{O}(\lambda(a))$ Wortoperationen.

9.15. **Übung.** Schreiben Sie eine (effiziente) Funktion, die zu einer positiven ganzen Zahl a das maximale $n \geq 1$ berechnet, so dass a eine n-te Potenz ist; als zweiter Wert soll die positive n-te Wurzel von a bestimmt werden. Schätzen Sie die Komplexität Ihrer Implementation ab.

10. Resultanten und modulare ggT-Berechnung

Bevor wir uns schnellen Verfahren zur Berechnung von ggTs zuwenden (die man zum Beispiel für schnelle Berechnungen mit rationalen Zahlen braucht), wollen wir uns erst einmal der Frage zuwenden, wie man zum Beispiel ggTs von Polynomen in $\mathbb{Z}[X]$ (oder $F[X, Y]$) vernünftig ausrechnen kann. Das Problem, das man dabei in den Griff bekommen möchte, ist, dass die Koeffizienten der Polynome, die im Verlauf des Euklidischen Algorithmus auftreten, relativ stark wachsen können, obwohl das Endergebnis vergleichsweise klein ist.

Zunächst einmal wollen wir abschätzen, wie groß ein Divisionsrest in $\mathbb{Q}[X]$ werden kann. Dazu brauchen wir ein Maß für die „Länge“ der Koeffizienten. Wir setzen

$$\lambda(r/s) = \max\{\lambda(r), \lambda(s)\} \quad \text{für } r \in \mathbb{Z},\ s \in \mathbb{Z}_{>0} \text{ teilerfremd.}$$

Ist $a = \sum_{j=0}^{n} a_j X^j \in \mathbb{Q}[X]$, dann sei c der kleinste gemeinsame Nenner der Koeffizienten a_j: $a_j = b_j/c$. Wir behandeln a als in der Form

$$a = (b_0 + b_1 X + \cdots + b_n X^n)/c$$

dargestellt und definieren

$$\lambda(a) = \max\{\lambda(c), \lambda(b_0), \ldots, \lambda(b_n)\}\,.$$

Ist $a \in \mathbb{Z}[X]$, dann gilt

$$\lambda(a) = \max\{\lambda(a_0), \ldots, \lambda(a_n)\}\,.$$

Wir können a in maximal $(n + 1)\lambda(a)$ (falls $a \in \mathbb{Z}[X]$) bzw. $(n + 2)\lambda(a)$ (falls $a \in \mathbb{Q}[X]$) Datenworten unterbringen.

Für $a, b \in \mathbb{Z}[X]$, $c, d \in \mathbb{Q}$ haben wir dann

$$\begin{aligned}
\lambda(c + d) &\leq \lambda(c) + \lambda(d) + 1 \\
\lambda(cd) &\leq \lambda(c) + \lambda(d) \\
\lambda(c/d) &\leq \lambda(c) + \lambda(d) \qquad (\text{wenn } d \neq 0)
\end{aligned}$$

$$\begin{aligned}
\lambda(a + b) &\leq \max\{\lambda(a), \lambda(b)\} + 1 \\
\lambda(ab) &\leq \lambda(a) + \lambda(b) + \lambda(\min\{\deg a, \deg b\} + 1)
\end{aligned}$$

Uns interessiert, wie groß der Rest bei einer Division wird. Seien also $a, b \in \mathbb{Q}[X]$, jeweils mit Leitkoeffizient 1 und $n = \deg a = \deg b + 1$:

$$a = X^n + (a_{n-1}X^{n-1} + \cdots + a_0)/c$$
$$b = X^{n-1} + (b_{n-2}X^{n-2} + \cdots + b_0)/d$$

mit $a_j, b_j, c, d \in \mathbb{Z}$. Sei weiter $a = qb + \rho^{-1}r$ mit $\rho \in \mathbb{Q}^\times$, $r \in \mathbb{Q}[X]$, $\deg r < n-1$, $\mathrm{lcf}(r) = 1$. Dann ist

$$q = X + \frac{a_{n-1}}{c} - \frac{b_{n-2}}{d} = X + \frac{a_{n-1}d - b_{n-2}c}{cd} = \frac{cd\,X + (a_{n-1}d - b_{n-2}c)}{cd}$$

und

$$\rho^{-1}r = a - qb = \frac{1}{cd^2}\sum_{j=0}^{n-2}\big(a_j d^2 - b_{j-1}cd - b_j(a_{n-1}d - b_{n-2}c)\big)X^j\,.$$

Wir sehen, dass

$$\lambda(q) \le \lambda(a) + \lambda(b) + 1 \quad \text{und} \quad \lambda(\rho^{-1}r) \le \lambda(a) + 2\lambda(b) + 1 \le 3\max\{\lambda(a), \lambda(b)\} + 1\,.$$

Für $\lambda(r)$ gilt die selbe Abschätzung wie für $\lambda(\rho^{-1}r)$. Es ist tatsächlich so (wie man ausprobieren kann — Übung!), dass der Divisionrest bei Division von zwei zufälligen Polynomen mit Koeffizientengröße ℓ normalerweise Koeffizienten der Größe nahe an 3ℓ hat. Das sind natürlich erst einmal schlechte Nachrichten, denn es lässt befürchten, dass die Größe der Koeffizienten der sukzessiven Reste im Euklidischen Algorithmus exponentiell wächst. Wir werden allerdings sehen, dass das nicht stimmt und die Länge der Koeffizienten sich polynomial beschränken lässt.

Am Ende ist die Beschränkung der Koeffizienten in den Zwischenergebnissen allerdings gar nicht so wichtig. Wenn wir gute Schranken für die Endergebnisse haben (also für den ggT und für die Koeffizienten der Linearkombination, die den ggT darstellt), dann können wir hoffen, mit geeigneten modularen Algorithmen in jedem Fall effizienter hinzukommen.

Da wir rationale Zahlen nicht modulo p reduzieren können (das geht nur, wenn p den Nenner nicht teilt), müssen wir in $\mathbb{Z}[X]$ rechnen. Das ist aber kein euklidischer Ring, also müssen wir uns kurz an einige Tatsachen aus der Algebra erinnern. Ein Integritätsring heißt *faktoriell*, wenn in ihm der Satz von der eindeutigen Primfaktorzerlegung gilt. Jeder Hauptidealring (und damit jeder euklidische Ring) ist faktoriell; die Umkehrung gilt nicht. In faktoriellen Ringen gibt es größte gemeinsame Teiler (die bis auf assoziierte eindeutig bestimmt sind).

10.1. **Satz (Gauß).** *Ist R ein faktorieller Ring, so ist auch $R[X]$ faktoriell.*

10.2. **Folgerung.** *Sei $R = \mathbb{Z}$ oder ein Körper. Dann ist $R[X_1, \ldots, X_n]$ ein faktorieller Ring.*

Der Satz beruht auf dem Lemma von Gauß. Dazu brauchen wir noch ein paar Begriffe.

10.3. **Definition.** Sei R ein faktorieller Ring. Ist

$$0 \neq a = a_0 + a_1X + \cdots + a_nX^n \in R[X]\,,$$

so heißt $\mathrm{cont}(a) = \mathrm{ggT}(a_0, a_1, \ldots, a_n)$ der *Inhalt* von a. Gilt $\mathrm{cont}(a) = 1$, so heißt a *primitiv*. Wir können stets schreiben $a = \mathrm{cont}(a)\,\mathrm{pp}(a)$ mit einem primitiven Polynom $\mathrm{pp}(a) \in R[X]$; $\mathrm{pp}(a)$ heißt der *primitive Anteil* von a.

Ist K der Quotientenkörper von R und $0 \neq a \in K[X]$, dann gibt es ein bis auf Multiplikation mit Einheiten von R eindeutig bestimmtes Element $c \in K^\times$, so dass $c^{-1}a \in R[X]$ ein primitives Polynom ist. Wir setzen dann $\mathrm{cont}(a) = c$ und $\mathrm{pp}(a) = c^{-1}a$.

Für das Nullpolynom definieren wir $\mathrm{cont}(0) = 0$ und $\mathrm{pp}(0) = 1$.

10.4. **Lemma von Gauß.** *Sei R ein faktorieller Ring. Für Polynome $a, b \in R[X]$ gilt dann* $\mathrm{cont}(ab) = \mathrm{cont}(a)\,\mathrm{cont}(b)$.

Es genügt, diese Aussage für primitive Polynome zu beweisen: Das Produkt zweier primitiver Polynome ist wieder primitiv.

Da $R[X]$ wieder faktoriell ist, gibt es größte gemeinsame Teiler in $R[X]$. Wir nehmen an, dass wir eine multiplikative Normalform $\mathrm{normal}(\cdot)$ auf R definiert haben als $\mathrm{normal}(r) = \mathrm{lu}(r)^{-1}r$; dann setzen wir

$$\mathrm{lu}(a) = \mathrm{lu}(\mathrm{lcf}(a)) \quad \text{und} \quad \mathrm{normal}(a) = \mathrm{lu}(a)^{-1}a$$

für Polynome $a \in R[X]$. Wie früher haben wir dann eine eindeutig definierte *Funktion* ggT auf $R[X]$.

Aus dem Lemma von Gauß folgt dann für $a, b \in R[X]$

$$c := \mathrm{ggT}(a, b) = \mathrm{ggT}\big(\mathrm{cont}(a), \mathrm{cont}(b)\big)\,\mathrm{ggT}\big(\mathrm{pp}(a), \mathrm{pp}(b)\big)$$

mit $\mathrm{cont}(c) = \mathrm{ggT}(\mathrm{cont}(a), \mathrm{cont}(b))$ und $\mathrm{pp}(c) = \mathrm{ggT}(\mathrm{pp}(a), \mathrm{pp}(b))$. Insbesondere ist

$$\mathrm{lcf}(c)^{-1}c = \mathrm{ggT}(a, b) \quad \text{in } K[X].$$

(Beachte, dass in $K[X]$ Polynome auf Leitkoeffizient 1 normiert werden.)

Wir können also wie folgt größte gemeinsame Teiler in $R[X]$ berechnen:

```
function gcd_{R[X]}(a, b)
  // a, b ∈ R[X].
  // Ausgabe: ggT(a, b) ∈ R[X].
  c ← cont(a) ∈ R
  if c = 0 then return normal(b) end if
  d ← cont(b) ∈ R
  if d = 0 then return normal(a) end if
  ã ← a/c ∈ R[X] // ã = pp(a)
  b̃ ← b/d ∈ R[X] // b̃ = pp(b)
  g̃ ← gcd_{K[X]}(ã, b̃) ∈ K[X] // K Quotientenkörper von R, ggT in K[X]
  g ← pp(gcd_R(lcf(ã), lcf(b̃)) · g) ∈ R[X]
  return gcd_R(c, d) · g
end function
```

Es ist nur zu überlegen, dass $\mathrm{gcd}_R(\mathrm{lcf}(\tilde a), \mathrm{lcf}(\tilde b)) \cdot g$ in $R[X]$ ist (also Koeffizienten in R hat). Sei dazu $h = \mathrm{ggT}_{R[X]}(\tilde a, \tilde b)$ der ggT in $R[X]$. Dann gilt $h \in R[X]$ und $h = \mathrm{lcf}(h)\tilde g$. Außerdem teilt h sowohl $\tilde a$ als auch $\tilde b$; damit teilt $\mathrm{lcf}(h)$ beide Leitkoeffizienten und somit $\mathrm{ggT}_R(\mathrm{lcf}(\tilde a), \mathrm{lcf}(\tilde b))$. Damit ist $g = rh$ mit $r \in R$ und somit in $R[X]$.

Für $R = \mathbb{Z}$ oder $R = F[Y]$ ist diese Methode aber nicht unbedingt optimal, da (wie wir gesehen haben) die Größe der Zwischenergebnisse stark zunehmen kann. Wir würden daher gerne modular rechnen. Dazu müssen wir den Zusammenhang zwischen der Reduktion mod p des ggT zweier Polynome und dem ggT der Reduktion der Polynome studieren. Das wichtigste Hilfsmittel dafür ist die *Resultante*.

10.5. **Definition.** Sei R ein Integritätsring, und seien $a, b \in R[X]$ mit $\deg a \le n$, $\deg b \le m$. Wir schreiben

$$a = a_n X^n + \cdots + a_1 X + a_0 \quad \text{und} \quad b = b_m X^m + \cdots + b_1 X + b_0$$

und defineren die *Sylvestermatrix* von a und b als die $\big((m+n)\times(m+n)\big)$-Matrix

$$\mathrm{Syl}(a,b) = \mathrm{Syl}_{n,m}(a,b) = \begin{pmatrix}
a_n & & & & b_m & & & & \\
a_{n-1} & a_n & & & b_{m-1} & b_m & & & \\
\vdots & a_{n-1} & \ddots & & \vdots & b_{m-1} & \ddots & & \\
\vdots & \vdots & \ddots & a_n & b_1 & \vdots & \ddots & \ddots & \\
a_1 & \vdots & & a_{n-1} & b_0 & b_1 & & \ddots & b_m \\
a_0 & a_1 & & \vdots & & b_0 & \ddots & & b_{m-1} \\
 & a_0 & \ddots & \vdots & & & \ddots & \ddots & \vdots \\
 & & \ddots & a_1 & & & & \ddots & b_1 \\
 & & & a_0 & & & & & b_0
\end{pmatrix}$$

(mit Nullen ober- und unterhalb der angegebenen Einträge). Die *Resultante* von a und b ist

$$\mathrm{Res}_{n,m}(a,b) = \det \mathrm{Syl}_{m,n}(a,b) \in R\,.$$

Wir schreiben auch $\mathrm{Res}(a,b) = \mathrm{Res}_{\deg a, \deg b}(a,b)$. Dabei sei $\mathrm{Res}(0,b) = 0$, außer b ist konstant und $\neq 0$, dann ist $\mathrm{Res}(0,b) = 1$; analog für $\mathrm{Res}(a,0)$.

Wenn wir mit $R[X]_{<n}$ den R-Modul der Polynome vom Grad $< n$ bezeichnen, dann ist $\mathrm{Syl}_{m,n}(a,b)$ die Matrix der R-linearen Abbildung

$$\varphi_{a,b} : R[X]_{<m} \times R[X]_{<n} \longrightarrow R[X]_{<m+n}\,, \qquad (s,t) \longmapsto sa + tb$$

bezüglich der Standardbasen $(X^{m-1},0),\ldots,(X,0),(1,0),(0,X^{n-1}),\ldots,(0,X),(0,1)$ und $X^{m+n-1},\ldots,X,1$.

Der Zusammenhang mit dem größten gemeinsamen Teiler wird durch das folgende Resultat hergestellt.

10.6. **Proposition.** *Seien R ein Integritätsring mit Quotientenkörper K und $a, b \in R[X]$ mit $\deg a \le n$ und $\deg b \le m$ und $n, m \ge 0$, $n + m \ge 1$. Dann sind die folgenden Aussagen äquivalent:*

(1) $\deg a < n$ *und* $\deg b < m$, *oder* $\mathrm{ggT}_{K[X]}(a,b)$ *ist nicht konstant;*
(2) $\mathrm{Res}_{n,m}(a,b) = 0$*;*
(3) *es gibt* $(0,0) \neq (s,t) \in R[X]_{<m} \times R[X]_{<n}$ *mit* $sa + tb = 0$.

Beweis. Gilt $\deg a < n$ und $\deg b < m$, dann ist die erste Zeile (die gibt es wegen $n+m \ge 1$) der Sylvestermatrix $\mathrm{Syl}_{n,m}(a,b)$ null, also ist auch $\mathrm{Res}_{n,m}(a,b) = 0$. In diesem Fall gilt $b\cdot a + (-a)\cdot b = 0$ (oder, falls $a = b = 0$, $1\cdot a + 0\cdot b = 0\cdot a + 1\cdot b = 0$). Wir können also im Folgenden voraussetzen, dass (nach eventuellem Vertauschen von a und b) $\deg a = n$ (insbesondere $a \neq 0$) gilt.

Sei $c = \mathrm{ggT}_{K[X]}(a,b)$. Ist c nicht konstant, dann gilt mit $s = b/c$, $t = -a/c$, dass $\deg s < \deg b \le m$, $\deg t < \deg a = n$ und $sa + tb = 0$. Da $a \neq 0$, ist auch $t \neq 0$. Ist c konstant, dann gilt $\mathrm{ggT}_{K[X]}(a,b) = 1$ und damit (in $K[X]$)

$$sa + tb = 0 \implies a \mid tb \implies a \mid t \implies \deg t \ge n \text{ oder } t = 0\,.$$

Ist $\deg t < n$, dann muss $t = 0$ sein und damit auch $s = 0$. Das liefert einen Widerspruch zu Aussage (3). Damit ist die Äquivalenz von (1) und (3) gezeigt.

Um die Äquivalenz von (2) und (3) einzusehen, beachten wir, dass über K gilt:

$$\begin{aligned}\mathrm{Res}_{n,m}(a,b)=0 &\iff \ker\varphi_{a,b}\neq 0\\ &\iff \exists(0,0)\neq(s,t)\in K[X]_{<m}\times K[X]_{<n} : sa+tb=0\,.\end{aligned}$$

Die letzte Aussage ist äquivalent zu (3), da wir s und t mit einem gemeinsamen Nenner der Koeffizienten multiplizieren können. □

10.7. **Bemerkung.** Der etwas unschöne Sonderfall $\deg a < n$, $\deg b < m$ lässt sich vermeiden, wenn man a, und b als *homogene* Polynome in zwei Variablen vom Grad n bzw m auffasst (d.h. $a = a_nX^n + \cdots + a_1XY^{n-1} + a_0Y^n$ und analog für b). Dann verschwindet die Resultante genau dann, wenn a und b einen nicht konstanten gemeinsamen Faktor haben. Im Fall $a_n = b_m = 0$ ist Y ein gemeinsamer Faktor.

Hier ist noch eine wichtige Eigenschaft der Resultante:

10.8. **Lemma.** *Seien a und b wie in Proposition 10.6. Dann gibt es Polynome $(0,0)\neq(s,t)\in R[X]_{<m}\times R[X]_{<n}$ mit*

$$sa+tb=\mathrm{Res}_{n,m}(a,b)\,.$$

Beweis. Im Fall $\mathrm{Res}_{n,m}(a,b)=0$ ist das in Prop. 10.6 enthalten. Anderenfalls folgt die Aussage aus der Existenz der adjungierten Matrix $\widetilde{\mathrm{Syl}}_{n,m}(a,b)$ mit

$$\mathrm{Syl}_{n,m}(a,b)\cdot\widetilde{\mathrm{Syl}}_{n,m}(a,b)=\mathrm{Res}_{n,m}(a,b)I_{m+n}\,.$$

Die Koeffizienten von s und t ergeben sich aus der letzten Spalte dieser Matrix. □

10.9. **Folgerung.** *Sei K ein Körper, $a,b\in K[X]$ mit $\mathrm{ggT}_{K[X]}(a,b)=1$ und $\deg a+\deg b\geq 1$. Dann gibt es genau ein Paar $(s,t)\in K[X]_{<\deg b}\times K[X]_{<\deg a}$ mit $sa+tb=1$; dies sind die Polynome, die vom Erweiterten Euklidischen Algorithmus berechnet werden.*

Beweis. Ist $\mathrm{ggT}_{K[X]}(a,b)=1$, dann ist $\mathrm{Res}(a,b)\neq 0$, und die K-lineare Abbildung $\varphi_{a,b}$ ist ein Isomorphismus. Daraus folgt die Existenz und Eindeutigkeit von (s,t).

Der EEA liefert Polynome s und t mit $sa+tb=1$. Man zeigt leicht durch Induktion, dass (siehe den Algorithmus im Beweis von Satz 3.7)

$$\deg s_{i+1}+\deg r_i\leq\deg b \quad\text{und}\quad \deg t_{i+1}+\deg r_i\leq\deg a\,.$$

Gilt $s=s_\ell$ und $t=t_\ell$, so folgt $\deg r_{\ell-1}>\deg r_\ell=0$ (denn r_ℓ ist konstant), also gilt $\deg s<\deg b$, $\deg t<\deg a$. Damit greift die Eindeutigkeit der Lösung mit diesen Gradbeschränkungen. □

10.10. **Beispiele.** Man kann obige Resultate wie folgt interpretieren: $\operatorname{Res}(a,b) = 0$ genau dann, wenn a und b in einem Erweiterungskörper von K (dem Quotientenkörper des Koeffizientenrings R) eine gemeinsame Nullstelle haben.

Für Polynome $a = X^2 + pX + q$ und $b = X^2 + rX + s$ gilt zum Beispiel, dass a und b genau dann eine gemeinsame Nullstelle haben, wenn

$$\operatorname{Res}(a,b) = \begin{vmatrix} 1 & 0 & 1 & 0 \\ p & 1 & r & 1 \\ q & p & s & r \\ 0 & q & 0 & s \end{vmatrix} = p^2 s - pqr - prs + q^2 + qr^2 - 2qs + s^2$$

verschwindet.

Das Polynom $a = X^2 + pX + q$ hat genau dann eine doppelte Nullstelle, wenn

$$\operatorname{Res}(a,a') = \begin{vmatrix} 1 & 2 & 0 \\ p & p & 2 \\ q & 0 & p \end{vmatrix} = 4q - p^2$$

verschwindet.

Der Nutzen der Resultante zeigt sich recht schön an folgendem Ergebnis.

10.11. **Folgerung.** *Seien $a, b \in \mathbb{Z}[X]$ mit $\operatorname{ggT}_{\mathbb{Q}[X]}(a,b) = 1$. Dann gibt es nur endlich viele verschiedene Primzahlen p, so dass p die Leitkoeffizienten von a und b teilt oder a und b in einer Erweiterung von $\mathbb{F}_p$ eine gemeinsame Nullstelle haben.*

Beweis. Sei $n = \deg a$ und $m = \deg b$. Wir können annehmen, dass $m + n \geq 1$ ist. Aus der Voraussetzung folgt

$$r := \operatorname{Res}(a,b) = \operatorname{Res}_{n,m}(a,b) \in \mathbb{Z} \setminus \{0\}\,.$$

Dann gibt es nur endlich viele Primzahlen, die r teilen. Sei p eine Primzahl, die r nicht teilt; wir bezeichnen Reduktion mod p mit einem Querstrich. Nach Definition der Resultante ist r ein Polynom mit ganzzahligen Koeffizienten in den Koeffizienten von a und b. Also ist

$$\operatorname{Res}_{n,m}(\bar{a},\bar{b}) = \overline{\operatorname{Res}(a,b)} = \bar{r} \neq 0 \text{ in } \mathbb{F}_p.$$

Das bedeutet nach Proposition 10.6, dass $\deg \bar{a} = n$ oder $\deg \bar{b} = m$, und dass $\operatorname{ggT}_{\mathbb{F}_p[X]}(\bar{a},\bar{b}) = 1$. Letzteres heißt, dass $\bar{a}$ und $\bar{b}$ in keiner Körpererweiterung von $\mathbb{F}_p$ eine gemeinsame Nullstelle haben. Die erste Aussage ist gleich bedeutend damit, dass p nicht sowohl den Leitkoeffizienten von a als auch den Leitkoeffizienten von b teilt. □

Wenn wir es mit Polynomen in mehreren Variablen zu tun haben, dann schreiben wir $\operatorname{Res}_X(a,b)$, wenn wir die Resultante bezüglich der Variablen X bilden (also a und b also Polynome in X auffassen). Entsprechend schreiben wir $\deg_X a$ für den Grad von a als Polynom in X. Dann gilt die folgende Abschätzung.

10.12. **Lemma.** *Sei F ein Körper, und seien $a, b \in F[X,Y] \setminus \{0\}$. Dann gilt*

$$\begin{aligned} \deg_Y \operatorname{Res}_X(a,b) &\leq \deg_X b \deg_Y a + \deg_X a \deg_Y b \\ &\leq (\deg_X a + \deg_X b) \max\{\deg_Y a, \deg_Y b\}\,. \end{aligned}$$

Beweis. Sei $n = \deg_X a$, $m = \deg_X b$, und sei $s_{ij} \in F[Y]$ der Eintrag der Sylvestermatrix $\mathrm{Syl}_{n,m}(a,b)$ in Zeile i und Spalte j. Dann gilt

$$\begin{aligned} \deg_Y s_{ij} &\le \deg_Y a \quad \text{für } 1 \le j \le m \quad \text{und} \\ \deg_Y s_{ij} &\le \deg_Y b \quad \text{für } m+1 \le j \le n+m. \end{aligned}$$

Jeder Term in dem Ausdruck für $\mathrm{Res}_X(a,b) = \det \mathrm{Syl}_{n,m}(a,b)$ als Polynom in den s_{ij} ist ein Produkt von Einträgen, das aus jeder Spalte genau einen Eintrag enthält. Jeder Term hat also Grad $\le m \deg_Y a + n \deg_Y b$; das gilt dann auch für die Resultante. □

Für Polynome mit ganzzahligen Koeffizienten gibt es eine analoge Abschätzung. Dafür führen wir analog wie in der linearen Algebra folgende *Normen* ein.

10.13. **Definition.** Sei $a = a_0 + a_1 X + \cdots + a_n X^n \in \mathbb{Z}[X]$ (oder $\mathbb{R}[X]$ oder $\mathbb{C}[X]$). Dann setzen wir

$$\begin{aligned} \|a\|_\infty &= \max\{|a_0|, |a_1|, \ldots, |a_n|\} \\ \|a\|_2 &= \sqrt{|a_0|^2 + |a_1|^2 + \cdots + |a_n|^2} \\ \|a\|_1 &= |a_0| + |a_1| + \cdots + |a_n| \end{aligned}$$

Es gelten die aus der linearen Algebra bekannten Abschätzungen

$$\|a\|_\infty \le \|a\|_2 \le \|a\|_1 \le (n+1)\|a\|_\infty \quad \text{und} \quad \|a\|_2 \le \sqrt{n+1}\|a\|_\infty .$$

10.14. **Lemma.** *Seien $a, b \in \mathbb{Z}[X] \setminus \{0\}$ mit $\deg a = n$, $\deg b = m$. Dann gilt*

$$|\operatorname{Res}(a,b)| \le \|a\|_2^m \|b\|_2^n \le (n+1)^{m/2}(m+1)^{n/2}\|a\|_\infty^m \|b\|_\infty^n .$$

Beweis. Die erste Ungleichung folgt direkt aus der Hadamard-Ungleichung

$$|\det(v_1, v_2, \ldots, v_n)| \le \|v_1\|_2 \|v_2\|_2 \cdots \|v_n\|_2$$

(für Spaltenvektoren v_j), wenn man sie auf die Sylvestermatrix von a und b anwendet. Die zweite Ungleichung ist klar. □

Jetzt können wir den Zusammenhang zwischen $\overline{\mathrm{ggT}(a,b)}$ und $\mathrm{ggT}(\bar a, \bar b)$ aufklären (der Querstrich bezeichne dabei Reduktion mod p).

10.15. **Satz.** *Sei R ein euklidischer Ring, $p \in R$ ein Primelement, und $a, b \in R[X] \setminus \{0\}$. Dann ist $F = R/pR$ ein Körper, und wir bezeichnen die kanonischen Homomorphismen $R \to F$ bzw. $R[X] \to F[X]$ mit einem Quaerstrich. Wir setzen $g = \mathrm{ggT}_{R[X]}(a,b)$, $d = \deg g$, $\tilde d = \deg \mathrm{ggT}_{F[X]}(\bar a, \bar b)$, $l = \mathrm{lcf}(g)$ und $m = \mathrm{ggT}_R(\mathrm{lcf}(a), \mathrm{lcf}(b))$. Wenn $p \nmid m$, dann gilt*

(1) $l \mid m$,
(2) $\tilde d \ge d$,
(3) $\tilde d = d \iff \bar g = \bar l \, \mathrm{ggT}_{F[X]}(\bar a, \bar b) \iff p \nmid \operatorname{Res}(a/g, b/g)$.

Beweis. Aussage (1) ist klar. Seien $u = a/g$, $v = b/g$, dann gilt

$$\bar u \bar g = \bar a \quad \text{und} \quad \bar v \bar g = \bar b ,$$

Also ist $\bar g$ ein Teiler von $\operatorname{ggT}_{F[X]}(\bar a, \bar b)$. Da $l \mid m$, aber $p \nmid m$, teilt p den Leitkoeffizienten l von g nicht; es folgt $\deg \bar g = \deg g = d$. Das impliziert (2) und die erste Äquivalenz in (3). Die zweite Äquivalenz in (3) ergibt sich aus

$$\begin{aligned}\bar g = \bar l \operatorname{ggT}_{F[X]}(\bar a, \bar b) &\iff \operatorname{ggT}_{F[X]}(\bar u, \bar v) = 1 \\ &\iff \operatorname{Res}_{\deg u, \deg v}(\bar u, \bar v) \neq 0 \iff p \nmid \operatorname{Res}(u, v)\,.\end{aligned}$$

Man beachte dabei, dass p nicht beide Leitkoeffizienten von u und v teilt, und benutze Proposition 10.6. □

Wenn wir den ggT modular berechnen wollen, müssen wir also ein Primelement p so wählen, dass

(1) p nicht beide Leitkoeffizienten teilt,
(2) p nicht die Resultante $\operatorname{Res}(a/g, b/g)$ teilt,
(3) p groß genug ist, dass die Koeffizienten von g rekonstruiert werden können.

Wir betrachten zunächst den Fall $R = F[Y]$, d.h., die Berechnung von größten gemeinsamen Teilern von Polynomen a und b in zwei Variablen über einem Körper F. Wir können annehmen, dass a und b als Polynome in X primitiv sind, also keine nicht konstanten Teiler haben, die Polynome nur in Y sind.

Die Koeffizienten sind dann Polynome in Y, und es ist klar, dass der Grad in Y des ggT beschränkt ist durch $\min\{\deg_Y a, \deg_Y b\}$. Das Primelement p ist ein irreduzibles Polynom in $F[Y]$; für die Rekonstruierbarkeit der Koeffizienten genügt es also, wenn

$$\deg_Y p > \min\{\deg_Y a, \deg_Y b\}$$

ist. Das sichert auch schon, dass p nicht beide Leitkoeffizienten teilt. Der Grad der Resultante $\operatorname{Res}_X(a/g, b/g)$ ist nach Lemma 10.12 beschränkt durch

$$\deg_X a \deg_Y b + \deg_X b \deg_Y a\,.$$

Wenn alle Grade etwa von der Größe n sind, dann müsste $\deg p \gg n^2$ sein, um sicherzustellen, dass p die Resultante nicht teilt. Das würde keinen besonders effizienten Algorithmus liefern. Auf der anderen Seite ist es für ein gegebenes irreduzibles Polynom p sehr unwahrscheinlich, dass es die Resultante teilt, auch wenn sein Grad nur von der Größenordnung n ist. Man kann also hoffen, das richtige Ergebnis zu bekommen, wenn man ein zufälliges p wählt. Wenn man schnell testen kann, ob das Ergebnis stimmt, dann kann man, falls nötig, die Rechnung mit einem anderen p wiederholen. Das führt auf einen *probabilistischen Algorithmus*. Das ist ein Algorithmus, dessen Ablauf von Zufallsexperimenten abhängt, der aber (wenn er terminiert) das richtige Ergebnis liefert. Lediglich die Laufzeit ist dann durch eine Zufallsvariable gegeben.

Wir wählen also ein irreduzibles $p \in F[Y]$ zufällig mit einem noch zu bestimmenden Grad. Dann ist $K = F[Y]/pF[Y]$ eine Körpererweiterung von F, in der wir (effizient, mit Komplexität $\tilde O(\deg p)$ Operationen in F[1]) rechnen können. Wir berechnen $\tilde g = \operatorname{ggT}_{K[X]}(\bar a, \bar b)$. Sei $m = \operatorname{ggT}_R(\operatorname{lcf}_X(a), \operatorname{lcf}_X(b))$. Wenn $\deg_X \tilde g = \deg_X g$ ist (mit $g = \operatorname{ggT}_{F[X,Y]}(a, b)$), dann ist nach Satz 10.15 $\bar m \tilde g$ die Reduktion mod p von sg mit einem $s \in F[Y]$. Wir können sg bestimmen als das eindeutige Polynom g_1 mit $\deg_Y g_1 \le \deg_Y m + \deg_Y g$, dessen Reduktion $\bar m \tilde g$ ist, sofern

$$\deg_Y p \ge \deg_Y m + \min\{\deg_Y a, \deg_Y b\} + 1$$

ist. Es ist dann $g = \operatorname{pp}_X(g_1)$ der ggT von a und b in $F[X, Y]$.

[1] für die Division in K müssen wir das noch zeigen!

Für das Folgende setzen wir voraus, dass

$$\deg_Y p \geq \deg_Y m + \max\{\deg_Y a, \deg_Y b\} + 1$$

ist. Dann gilt $\deg_Y h < \deg_Y p$ für jeden Teiler h von ma oder mb. Wir können g_1 wie oben in jedem Fall berechnen. Wenn g_1 ein Teiler von ma und von mb ist, dann ist $g = \mathrm{pp}_X(g_1)$ der ggT von a und b in $F[X,Y]$ (denn $\mathrm{pp}_X(g_1)$ ist dann ein gemeinsamer Teiler, und $\deg_X g_1 = \deg_X \tilde{g} \geq \deg_X g$). Um das zu testen, seien $\tilde{u} = \bar{a}/\tilde{g}$ und $\tilde{v} = \bar{b}/\tilde{g}$. Dann gibt es eindeutig bestimmte Polynome $u, v \in F[X,Y]$ mit $\deg_Y u, \deg_Y v < \deg_Y p$ und $\bar{u} = \tilde{u}$, $\bar{v} = \tilde{v}$. Es gilt dann $\overline{ug_1} = \tilde{u}\bar{m}\tilde{g} = \overline{ma}$ und analog $\overline{vg_1} = \overline{mb}$. Ist $ma = hg_1$, dann gilt $\bar{h} = \tilde{u}$ und $\deg_Y h < \deg p$, also $h = u$. Gilt $\deg_Y(ug_1) = \deg_Y(ma)$, dann folgt wegen

$$\overline{ug_1} = \overline{ma} \quad \text{und} \quad \deg_Y p > \deg_Y(ug_1) = \deg_Y(ma)\,,$$

dass $ug_1 = ma$, also ist g_1 ein Teiler von ma. Wir sehen:

$$g_1 \mid ma \iff \deg_Y(ug_1) = \deg_Y(ma)\,,$$

und analog

$$g_1 \mid mb \iff \deg_Y(vg_1) = \deg_Y(mb)\,.$$

10.16. **Modularer ggT in $F[X,Y]$ mit „großem“ Primelement.** Das führt nun auf folgenden Algorithmus.

```
function gcd_{F[X,Y]}(a, b)
  // a, b ∈ F[X,Y] primitiv als Polynome in X.
  // Ausgabe: ggT_{F[X,Y]}(a, b).
  m ← gcd_{F[Y]}(lcf_X(a), lcf_X(b))
  d_u ← deg_Y m + deg_Y a
  d_v ← deg_Y m + deg_Y b
  repeat
    p ← zufälliges irreduzibles Polynom in F[Y] mit deg p = max{d_u, d_v} + 1
    K ← F[Y]/pF[Y]
    ā ← a mod p ∈ K[X]
    b̄ ← b mod p ∈ K[X]
    m̄ ← m mod p ∈ K
    g̃ ← gcd_{K[X]}(ā, b̄)
    u ← lift(ā/g̃)
    v ← lift(b̄/g̃)
    g_1 ← lift(m̄ · g̃)
  until deg_Y u + deg_Y g_1 = d_u and deg_Y v + deg_Y g_1 = d_v
  return pp_X(g_1)
end function
```

Hier sei $\mathrm{lift}(\tilde{h})$ für $\tilde{h} \in K[X]$ das eindeutig bestimmte Polynom $h \in F[X,Y]$ mit $\deg_Y h < \deg_Y p$ und $\bar{h} = \tilde{h}$. Algorithmisch passiert dabei nichts, es wird lediglich die interne Darstellung anders interpretiert.

Dass dieser Algorithmus das korrekte Ergebnis liefert, haben wir bereits diskutiert. Was sind die Kosten? Dazu nehmen wir an, dass $\deg_X a, \deg_X b \leq n$ und $\deg_Y a, \deg_Y b \leq k$. Die Berechnung von m erfordert eine ggT-Berechnung in $F[Y]$ von Polynomen vom Grad $\leq k$; der Aufwand dafür ist $\ll k^2$ Operationen in F. Wie man ein zufälliges irreduzibles Polynom vom Grad d findet, werden wir später diskutieren. Für $F = \mathbb{F}_q$ zum Beispiel ist das mit einem durchschnittlichen Aufwand von $\tilde{O}(d^2 \log q)$ Operationen in $\mathbb{F}_q$ (also mit $\tilde{O}(d^2 (\log q)^2)$ Wortoperationen)

möglich. Hier ist $d \le 2k+1$, also gelten die Abschätzungen mit k an Stelle von d. Bei der Reduktion mod p passiert nichts, da die Koeffizienten von a und b Grad $< d = \deg p$ haben; das Gleiche gilt für m. Die Berechnung von $\tilde{g}$ mit dem Euklidischen Algorithmus in $K[X]$ braucht $\ll n^2$ Operationen in K, also $\tilde{O}(n^2k)$ Operationen in F. Die Kosten für $\bar{a}/\tilde{g}$ und $\bar{b}/\tilde{g}$ sind $\tilde{O}(nk)$, ebenso für $\bar{m}\cdot\tilde{g}$. Der Aufwand für einen Schleifendurchlauf ist also

$$\tilde{O}(n^2k) + \text{Aufwand zur Bestimmung von } p\,.$$

Die Schritte vor und nach der repeat-Schleife brauchen zusammen nicht mehr als $\tilde{O}(nk^2)$ Operationen in F.

Wir müssen noch überlegen, wie wahrscheinlich es ist, dass wir ein „gutes“ p erwischen. Dazu sei $n = \max\{\deg_X a, \deg_X b\}$ und $k = \max\{\deg_Y a, \deg_Y b\}$. Dann ist $d = \deg_Y p \ge k+1$. Sei $g = \mathrm{ggT}_{F[X,Y]}(a,b)$ und $r = \mathrm{Res}_X(a/g, b/g)$. Dann ist p „gut“, wenn p kein Teiler von r ist. Der Grad von r ist höchstens $2nk$ nach Lemma 10.12. Es gibt also höchstens $2nk/(k+1) < 2n$ verschiedene irreduzible Polynome p vom Grad d, die r teilen. Wenn das weniger als die Hälfte aller (normierten) irreduziblen Polynome vom Grad d ist, oder genauer, wenn wir ein „schlechtes“ p mit Wahrscheinlichkeit $< 1/2$ wählen, dann ist der Erwartungswert für die Anzahl der Schleifendurchläufe kleiner als 2, und die erwartete Komplexität ist $\tilde{O}(nk(n+k))$ (plus der Aufwand für das Auffinden eines irreduziblen Polynoms).

Ist $F = \mathbb{F}_q$ ein endlicher Körper, dann gibt es etwa q^d/d irreduzible Polynome vom Grad d, so dass die obige Bedingung fast immer erfüllt ist. Wenn d sehr klein ist, muss man eventuell Polynome p von etwas größerem Grad verwenden.

10.17. **Eine Schranke für die Koeffizienten von Teilern in $\mathbb{Z}[X]$.** Bevor wir einen analogen Algorithmus für $\mathbb{Z}[X]$ formulieren können, müssen wir eine brauchbare Schranke für die Größe der Koeffizienten der Teiler eines gegebenen Polynoms $a \in \mathbb{Z}[X]$ beweisen: Die zu verwendende Primzahl p muss ja ausreichend groß sein, damit wir die Koeffizienten des ggT aus ihren Bildern mod p rekonstruieren können.

Für Polynome $a \in F[Y][X]$ gilt, dass die Koeffizienten der Teiler von a nicht größer (im Sinne von $\deg_Y$) sein können als die Koeffizienten von a selbst. Das gilt in $\mathbb{Z}[X]$ nicht: Die irreduziblen Faktoren der Polynome X^n-1 sind gerade die Kreisteilungspolynome, und es ist bekannt, dass diese beliebig große Koeffizienten haben können. Wie üblich ist die Situation über $\mathbb{Z}$ also ein wenig komplizierter.

Wir erinnern uns an die 2-Norm für Polynome in $\mathbb{C}[X]$ aus Definition 10.13. Wir brauchen noch eine weitere Größe.

10.18. **Definition.** Sei $a = a_0 + a_1X + \cdots + a_nX^n \in \mathbb{C}[X]$ mit $a_n \ne 0$. Wir schreiben

$$a = a_n(X-x_1)(X-x_2)\cdots(X-x_n) \quad \text{mit } x_j \in \mathbb{C}.$$

Das *Mahler-Maß* von a ist

$$M(a) = |a_n| \prod_{j=1}^{n} \max\{1, |x_j|\}\,.$$

Aus der Definition ist klar, dass

$$M(a) \ge |\operatorname{lcf}(a)| \quad \text{und} \quad M(ab) = M(a)M(b)\,.$$

Diese Multiplikativität wird für uns wichtig sein.

Erst einmal brauchen wir ein Lemma.

10.19. **Lemma.** *Sei $a \in \mathbb{C}[X]$ und $z \in \mathbb{C}$. Dann gilt*

$$\|(X - z)a\|_2 = \|(\bar{z}X - 1)a\|_2 \,.$$

Beweis. Sei $a = a_0 + a_1 X + \cdots + a_n X^n$. Auf der einen Seite ist

$$\begin{aligned}
\|(X - z)a\|_2^2 &= |-za_0|^2 + |a_0 - za_1|^2 + |a_1 - za_2|^2 + \cdots + |a_{n-1} - za_n|^2 + |a_n|^2 \\
&= |z|^2|a_0|^2 + |a_0|^2 - \bar{z}a_0\bar{a}_1 - z\bar{a}_0 a_1 + |z|^2|a_1|^2 \\
&\qquad + \cdots + |a_{n-1}|^2 - \bar{z}a_{n-1}\bar{a}_n - z\bar{a}_{n-1} + |z|^2|a_n|^2 + |a_n|^2 \\
&= (|z|^2 + 1)\|a\|_2^2 - 2\operatorname{Re}\sum_{j=0}^{n-1} z\bar{a}_j a_{j+1} \,.
\end{aligned}$$

Auf der anderen Seite haben wir

$$\begin{aligned}
\|(\bar{z}X - 1)a\|_2^2 &= |-a_0|^2 + |\bar{z}a_0 - a_1|^2 + |\bar{z}a_1 - a_2|^2 + \cdots + |\bar{z}a_{n-1} - a_n|^2 + |\bar{z}a_n|^2 \\
&= |a_0|^2 + |z|^2|a_0|^2 - \bar{z}a_0\bar{a}_1 - z\bar{a}_0 a_1 + |a_1|^2 \\
&\qquad + \cdots + |z|^2|a_{n-1}|^2 - \bar{z}a_{n-1}\bar{a}_n - z\bar{a}_{n-1} + |a_n|^2 + |z|^2|a_n|^2 \\
&= (|z|^2 + 1)\|a\|_2^2 - 2\operatorname{Re}\sum_{j=0}^{n-1} z\bar{a}_j a_{j+1} \,.
\end{aligned}$$

Wir sehen, dass wir beide Male das selbe Ergebnis bekommen. □

Jetzt setzen wir $\|a\|_2$ und $M(a)$ miteinander in Beziehung.

10.20. **Lemma (Landausche Ungleichung).** *Sei $a \in \mathbb{C}[X]$. Dann gilt*

$$M(a) \le \|a\|_2 \,.$$

Beweis. Sei wieder $a = a_n \prod_{j=1}^{n}(X - x_j)$. Dabei gelte $|x_1|, \ldots, |x_m| > 1$ und $|x_{m+1}|, \ldots, |x_n| \le 1$. Sei weiter

$$b = a_n \prod_{j=1}^{m}(\bar{x}_j X - 1) \prod_{j=m+1}^{n} (X - x_j) \,.$$

Nach Lemma 10.20 gilt dann $\|b\|_2 = \|a\|_2$. Außerdem ist

$$M(a)^2 = |a_n|^2|x_1|^2 \cdots |x_m|^2 = |a_n\bar{x}_1 \cdots \bar{x}_m|^2 = |\operatorname{lcf}(b)|^2 \le \|b\|_2^2 = \|a\|_2^2 \,.$$

□

10.21. **Bemerkung.** Man kann zeigen:

$$\|a\|_2^2 = \int_0^1 |a(e^{2\pi it})|^2 \, dt$$

und

$$M(a) = \exp\Big(\int_0^1 \log|a(e^{2\pi it})| \, dt\Big), \quad \text{also} \quad M(a)^2 = \exp\Big(\int_0^1 \log|a(e^{2\pi it})|^2 \, dt\Big).$$

Die Landausche Ungleichung ist dann ein Spezialfall der (stetigen Version der) Ungleichung zwischen arithmetischem und geometrischem Mittel (die wiederum aus der Konvexität nach oben der Logarithmusfunktion folgt).

10.22. **Lemma.** *Sei $a \in \mathbb{C}[X]$ mit $\deg a = n$. Dann gilt*

$$\|a\|_1 \le 2^n M(a).$$

Beweis. Wir schreiben wieder

$$a = a_0 + a_1 X + \cdots + a_n X^n = a_n \prod_{j=1}^{n} (X - x_j).$$

Dann ist

$$a_j = \pm a_n \sum_{S \subset \{1,\dots,n\}, \#S = n-j} \prod_{i \in S} x_i .$$

Weiterhin ist

$$\Big|a_n \prod_{i \in S} x_i\Big| \le |a_n| \prod_{i=1}^{n} \max\{1, |x_i|\} = M(a),$$

also haben wir

$$|a_j| \le \binom{n}{n-j} M(a) \implies \|a\|_1 = \sum_{j=0}^{n} |a_j| \le 2^n M(a).$$

□

Damit ergibt sich jetzt die erwünschte Abschätzung.

10.23. **Satz (Mignotte).** *Seien $a, b, c \in \mathbb{Z}[X]$ mit $\deg c = n \ge 1$ und so dass $ab \mid c$. Sei weiter $k = \deg a$ und $m = \deg b$. Dann gelten folgende Abschätzungen:*

(1) $\|a\|_\infty \|b\|_\infty \le \|a\|_2 \|b\|_2 \le \|a\|_1 \|b\|_1 \le 2^{k+m} \|c\|_2 \le \sqrt{1+n}\, 2^{k+m} \|c\|_\infty$;

(2) $\|a\|_\infty \le \|a\|_2 \le 2^k \|c\|_2 \le 2^k \|c\|_1$ *und* $\|a\|_\infty \le \|a\|_2 \le \sqrt{1+n}\, 2^k \|c\|_\infty$.

Beweis. (1) Die ersten beiden und die letzte Ungleichung sind klar. Die verbleibende folgt aus

$$\|a\|_1 \|b\|_1 \le 2^k M(a)\, 2^m M(b) = 2^{k+m} M(ab) \le 2^{k+m} M(c) \le 2^{k+m} \|c\|_2 .$$

Dabei haben wir erst Lemma 10.22 und dann Lemma 10.20 benutzt. Die Ungleichung $M(ab) \le M(c)$ folgt aus

$$M(c) = M(ab) M\Big(\frac{c}{ab}\Big) \quad \text{und} \quad M\Big(\frac{c}{ab}\Big) \ge \Big|\operatorname{lcf}\Big(\frac{c}{ab}\Big)\Big| \ge 1,$$

da $c/(ab)$ Koeffizienten in $\mathbb{Z}$ hat.

(2) folgt aus (1) mit $b = 1$. □

Ist also $a \in \mathbb{Z}[X]$ mit $\deg a = n$, so gilt für jeden Teiler $b \in \mathbb{Z}[X]$ von a, dass

$$\|b\|_\infty \leq 2^n \sqrt{n+1}\, \|a\|_\infty .$$

10.24. **Modularer ggT in $\mathbb{Z}[X]$ mit großer Primzahl.** Analog zum Algorithmus für $F[X,Y]$ haben wir dann folgenden Algorithmus für $\mathbb{Z}[X]$:

```
function ggT_{Z[X]}(a, b)
  // a, b ∈ Z[X] primitiv.
  // Ausgabe: ggT_{Z[X]}(a, b).
  n ← max{deg a, deg b}
  A ← max{‖a‖_∞, ‖b‖_∞}
  m ← ggT_Z(lcf(a), lcf(b))
  B ← 2^n √(n+1) A m ∈ R
  repeat
    p ← zufällige Primzahl mit 2B < p ≤ 4B
    ā ← a mod p ∈ F_p[X]; b̄ ← b mod p ∈ F_p[X]
    m̄ ← m mod p ∈ F_p
    g̃ ← ggT_{F_p[X]}(ā, b̄)
    u ← lift(ā/g̃); v ← lift(b̄/g̃)
    g_1 ← lift(m̄ · g̃)
  until ‖u‖_1 ‖g_1‖_1 ≤ B and ‖v‖_1 ‖g_1‖_1 ≤ B
  return pp(g_1)
end function
```

lift(h) für $h \in \mathbb{F}_p[X]$ sei dabei das Polynom $h_1 \in \mathbb{Z}[X]$ mit $h_1 \bmod p = h$ und $\|h_1\|_\infty < p/2$ (p sei ungerade).

Wie in der Diskussion von Algorithmus 10.16 gilt

$$u \cdot g_1 \equiv m \cdot a \bmod p \quad \text{und} \quad v \cdot g_1 \equiv m \cdot b \bmod p\,.$$

Außerdem haben wir

$$\|u \cdot g_1\|_\infty \leq \|u \cdot g_1\|_1 \leq \|u\|_1 \, \|g_1\|_1$$

und analog für v. Ist die Abbruchbedingung erfüllt, dann folgt wegen

$$\|ma\|_\infty, \|mb\|_\infty \leq mA \leq B \quad \text{und} \quad B < p/2\,,$$

dass

$$u \cdot g_1 = ma \quad \text{und} \quad v \cdot g_1 = mb\,,$$

also $g_1 \mid m \operatorname{ggT}_{\mathbb{Z}[X]}(a,b)$. Da andererseits $\deg g_1 \geq \deg \operatorname{ggT}_{\mathbb{Z}[X]}(a,b)$ (denn p teilt die Leitkoeffizienten von a und b nicht), folgt $\operatorname{pp}(g_1) = \operatorname{ggT}_{\mathbb{Z}[X]}(a,b) =: g$, und das Ergebnis ist korrekt.

Umgekehrt gilt auch wieder, dass $p \nmid \operatorname{Res}(a/g, b/g)$ impliziert, dass die Abbruchbedingung erfüllt ist: Wir haben nach Satz 10.23 die Abschätzung

$$\|g\|_\infty \leq 2^n \sqrt{n+1} \|a\|_\infty \leq 2^n \sqrt{n+1}\, A\,,$$

und mit $s = m/\operatorname{lcf}(g) \in \mathbb{Z}_{>0}$ dann auch

$$\|sg\|_\infty \leq m \|g\|_\infty \leq 2^n \sqrt{n+1}\, A\, m = B\,.$$

Außerdem gilt $sg \bmod p = \bar{m}\tilde{g}$ (Satz 10.15). Daraus folgt

$$sg = \operatorname{lift}(\bar{m}\tilde{g}) = g_1\,.$$

Es gibt dann Polynome $u_1, v_1 \in \mathbb{Z}[X]$ mit $u_1 g_1 = ma$, $v_1 g_1 = mb$; wir haben die Abschätzungen (wieder nach Satz 10.23)

$$\|u_1\|_\infty, \|v_1\|_\infty \le 2^n \sqrt{n+1}\, A\, m = B$$

und die Relationen

$$u_1 \bmod p = \bar{a}/\tilde{g}\,, \qquad v_1 \bmod p = \bar{b}/\tilde{g}\,.$$

Es folgt $u_1 = \operatorname{lift}(\bar{a}/\tilde{g}) = u$; ebenso $v_1 = v$. Nach Satz 10.23 gilt dann auch

$$\|u\|_1 \cdot \|g_1\|_1 \le 2^n \sqrt{n+1}\, \|ma\|_\infty \le B$$

und ebenso $\|v\|_1 \cdot \|g_1\|_1 \le B$; die Abbruchbedingung ist also erfüllt.

Mit wie vielen Durchläufen durch die Schleife müssen wir rechnen? Für die Resultante $r = \operatorname{Res}(a/g, b/g)$ gilt nach Lemma 10.14 und Satz 10.23 mit $d = \deg g$:

$$\begin{aligned} |r| &\le \|a/g\|_2^{n-d} \|b/g\|_2^{n-d} \\ &\le \left(2^{n-d}\sqrt{n+1}\|a\|_\infty\right)^{n-d} \left(2^{n-d}\sqrt{n+1}\|b\|_\infty\right)^{n-d} \\ &\le 2^{2(n-d)^2} (n+1)^n A^{2(n-d)} \le 2^{2n^2} (n+1)^n A^{2n} \le B^{2n}\,. \end{aligned}$$

Es gibt also höchstens $2n$ Primzahlen $p > B$, die r teilen. Andererseits gibt es nach dem Primzahlsatz

$$\gg \frac{B}{\log B} \gg \frac{2^n}{n}$$

Primzahlen zwischen $2B$ und $4B$. Außer wenn B (und damit n und A) sehr klein ist, gibt es demnach sehr viel mehr als $2n$ solche Primzahlen, und es ist sehr unwahrscheinlich, dass wir eine erwischen, die r teilt. Der Erwartungswert für die Anzahl der Schleifendurchläufe liegt also nahe bei 1.

Für die Komplexität gilt wieder, dass die ggT-Berechnung in $\mathbb{F}_p[X]$ und die Berechnung von $\operatorname{pp}(g_1)$ die dominierenden Schritte sind (abgesehen von der Bestimmung einer geeigneten Primzahl p). Die Kosten dafür sind

$$\in \tilde{O}(n^2 \log B) + \tilde{O}(n(\log B)^2) \in \tilde{O}\big(n(n^2 + (\log A)^2)\big) \quad \text{Wortoperationen.}$$

Für das Auffinden einer „Pseudoprimzahl“ braucht man $\ll (\log B)^4$ Wortoperationen (und noch mehr, wenn man garantiert eine Primzahl möchte). Im Endeffekt dominiert also der Aufwand für die Bestimmung von p die Laufzeit. Wir werden bald sehen, wie man dieses Problem umgehen kann.

10.25. **Modularer ggT mit kleinen Primelementen.** Wie wir bereits früher bei der Determinante gesehen haben, ist es meistens effizienter, modulo vieler kleiner Primelemente zu rechnen statt modulo einem großem. Wenn wir den ggT in $F[X,Y]$ berechnen wollen, dann sind die kleinsten Primelemente die irreduziblen Polynome $p = Y - \alpha$ vom Grad 1. Das Rechnen modulo p entspricht dem Einsetzen von $Y = \alpha$; die Rekostruktion mit Hilfe des Chinesischen Restsatzes entspricht der Interpolation. Wenn der Körper F genügend viele Elemente enthält, dann reicht es aus, nur diese einfachsten Primelemente zu verwenden. Eine erste Version des Algorithmus ist dann wie folgt.

function $\gcd_{F[X,Y]}(a, b)$
  // $a, b \in F[X,Y]$ primitiv bezüglich X.
  // Ausgabe: $\operatorname{ggT}_{F[X,Y]}(a, b)$.
  $m \leftarrow \gcd_{F[Y]}(\operatorname{lcf}_X(a), \operatorname{lcf}_X(b))$
  $d_u \leftarrow \deg_Y m + \deg_Y a$
  $d_v \leftarrow \deg_Y m + \deg_Y b$

$l \leftarrow \max\{d_u, d_v\} + 1$
repeat
  repeat
    $S \leftarrow$ zufällige Teilmenge von F mit $\#S \geq 2l$
    $S \leftarrow \{s \in S \mid m(s) \neq 0\}$
    for $s \in S$ do
      $a_s \leftarrow a(X, s)$
      $b_s \leftarrow b(X, s)$
      $\tilde{g}_s \leftarrow \gcd_{F[X]}(a_s, b_s)$
      $u_s \leftarrow a_s/\tilde{g}_s$
      $v_s \leftarrow b_s/\tilde{g}_s$
    end for
    $e \leftarrow \min\{\deg \tilde{g}_s \mid s \in S\}$
    $S \leftarrow \{s \in S \mid \deg \tilde{g}_s = e\}$
  until $\#S \geq l$
  $S \leftarrow$ Teilmenge von S mit l Elementen
  $u \leftarrow \text{interpolate}(S, (u_s)_{s\in S})$
  $v \leftarrow \text{interpolate}(S, (v_s)_{s\in S})$
  $g_1 \leftarrow \text{interpolate}(S, (m(s)\tilde{g}_s)_{s\in S})$
until $\deg_Y u + \deg_Y g_1 = d_u$ and $\deg_Y v + \deg_Y g_1 = d_v$
return $\text{pp}_X(g_1)$
end function

Dabei sei $\text{interpolate}(S, (h_s)_{s\in S})$ das Polynom $h \in F[X, Y]$ mit $\deg_Y h < \#S$ und $h(X, s) = h_s$ für alle $s \in S$.

Die Korrektheit folgt wie vorher: Wenn die Abbruchbedingung der äußeren repeat-Schleife erfüllt ist, dann gilt

$$ug_1 \equiv ma \bmod \prod_{s\in S}(Y - s) \quad \text{und} \quad \deg_Y(ug_1) = \deg_Y(ma) < l = \#S\,,$$

also folgt $ug_1 = ma$ und ebenso $vg_1 = mb$. Weil für jedes $s \in S$ gilt, dass $\deg \tilde{g}_s \geq \deg_X g$ (für $g = \text{ggT}_{F[X,Y]}(a, b)$) ist, gilt wieder

$$\text{pp}(g_1) \mid g \quad \text{und} \quad \deg_X g_1 \geq \deg_X g\,,$$

also ist $\text{pp}(g_1) = g$, denn beide Polynome sind primitiv und normiert.

Wie vorher gilt mit $r = \text{Res}_X(a/g, b/g)$, dass für $s \in F$ mit $m(s) \neq 0$ und $r(s) \neq 0$ die Relation $g(X, s) = \text{lcf}(g(X, s))\tilde{g}_s$ gilt. Damit der Algorithmus terminiert, brauchen wir also l Elemente s mit $m(s)r(s) \neq 0$. Sei $n = \max\{\deg_X a, \deg_X b\}$ und $k = \max\{\deg_Y a, \deg_Y b\}$. Da nach Lemma 10.12

$$\deg_Y(mr) \leq \deg_Y m + n(\deg_Y a + \deg_Y b) \leq (2n + 1)k\,,$$

gibt es höchstens $(2n+1)k$ „schlechte“ Wahlen von s. Wenn also $\#F \geq (4n+2)k$ ist, dann ist die Wahrscheinlichkeit, dass unsere ursprünglich gewählte Menge S bereits l „gute“ Elemente enthält, mindestens $1/2$, und der Erwartungswert für die Anzahl der Schleifendurchläufe ist höchstens 2. Ist der Körper F zu klein, kann man ihn entweder geeignet erweitern, oder man rechnet moldulo Polynomen von höherem Grad (was mathematisch auf das Gleiche hinausläuft).

Wie sieht es mit der Komplexität aus? In der innersten Schleife haben wir die Berechnung von a_s und b_s (jeweils $\ll kn$ Operationen in F), die Berechnung von $\tilde{g}_s$ ($\ll n^2$) und die Quotienten u_s und v_s (jeweils $\tilde{O}(n)$). Insgesamt ergibt das

$\ll kn(k+n)$ für die innere repeat-Schleife. Die Berechnung von u, v und g_1 erfordert $\ll n$ Interpolationen aus $\ll k$ Werten; das lässt sich in $\ll k^2 n$ Operationen in F erledigen. Insgesamt erhalten wir eine Komplexitätsschranke von

$$\ll kn(k+n) \quad \text{Operationen in } F.$$

Zum Vergleich: Der Algorithmus 10.16 mit einem großen Primelement hatte eine Komplexität von

$$\tilde{O}(kn(k+n)) + \text{Aufwand für das Auffinden von } p\,.$$

Das sieht nicht viel langsamer aus; allerdings kommen logarithmische Faktoren dazu, die im $\tilde{O}$ versteckt sind. Außerdem sparen wir uns den Aufwand, große irreduzible Polynome finden zu müssen. Und wir werden noch sehen, dass wir die Auswertungen, Interpolationen und ggT-Berechnungen in $F[X]$ noch beschleunigen können, so dass sich die Komplexität auf $\tilde{O}(kn)$ reduziert.

In der Praxis wird man S schrittweise vergrößern (und jeweils als „schlecht“ erkannte Elemente hinauswerfen), bis die Abbruchbedingung erfüllt ist. Das könnte etwa so aussehen:

```
function gcd_{F[X,Y]}(a, b)
  // a, b ∈ F[X,Y] primitiv bezüglich X.
  // Ausgabe: ggT_{F[X,Y]}(a, b).
  m ← gcd_{F[Y]}(lcf_X(a), lcf_X(b))
  d_u ← deg_Y m + deg_Y a;   d_v ← deg_Y m + deg_Y b;   l ← max{d_u, d_v} + 1
  e ← min{deg_X a, deg_X b} // Schranke für deg_X g.
  S ← ∅ ⊂ F
  while true do
    if e = 0 then return 1 ∈ F[X,Y] end if // ggT ist konstant.
    s ← zufälliges Element von F
    m_s ← m(s)
    if m_s ≠ 0 then
      a_s ← a(X, s);   b_s ← b(X, s);   g̃_s ← gcd_{F[X]}(a_s, b_s)
      if deg g̃_s < e then
        // Neue Schranke für deg_X g; alle bisherigen s sind „schlecht“.
        e ← deg g̃_s;   S ← {s}
      else
        if deg g̃_s = e then
          // Möglicherweise neues „gutes“ s.
          S ← S ∪ {s}
        end if
      end if
    end if
    if #S ≥ l then
      for s ∈ S do   u_s ← a_s/g̃_s;   v_s ← b_s/g̃_s   end for
      u ← interpolate(S, (u_s)_{s∈S});   v ← interpolate(S, (v_s)_{s∈S})
      g_1 ← interpolate(S, (m_s g̃_s)_{s∈S})
      if deg_Y u + deg_Y g_1 = d_u and deg_Y v + deg_Y g_1 = d_v then
        return pp_X(g_1)
      else
        // Wenn die Abbruchbedingung nicht erfüllt ist, muss deg_X g < e sein.
```

```
        e ← e − 1;    S ← ∅
      end if
    end if
  end while
end function
```

10.26. **Modularer ggT in $\mathbb{Z}[X]$ mit kleinen Primzahlen.** Für den ggT in $\mathbb{Z}[X]$ hat man einen analogen Algorithmus. Wir schreiben $\mathrm{CRS}(P, (h_p)_{p\in P})$ für das Polynom $h \in \mathbb{Z}[X]$ mit $\|h\|_\infty < (\prod_{p\in P} p)/2$ und $h \mod p = h_p$ für alle $p \in P$. Hierbei is P eine endliche Menge von ungeraden Primzahlen, und $h_p \in \mathbb{F}_p[X]$ ist ein Polynom. Die zweite Variante des Algorithmus sieht dann zum Beispiel so aus:

```
function gcd_{Z[X]}(a, b)
  // a, b ∈ Z[X] primitiv.
  // Ausgabe: ggT_{Z[X]}(a, b).
  n ← max{deg a, deg b};    A ← max{‖a‖_∞, ‖b‖_∞};    m ← gcd_Z(lcf(a), lcf(b))
  e ← min{deg a, deg b} // Schranke für deg g.
  P ← ∅ ⊂ Z
  R ← ⌈log_2((n + 1)^n A^{2n} m)⌉ // Schranke für die Anzahl der Primteiler von rm.
  U ← ⌈4R log(2R)⌉ // Höchstens jede zweite Primzahl < U ist „schlecht“.
  while true do
    if e = 0 then return 1 ∈ Z[X] end if // ggT ist konstant.
    p ← zufällige Primzahl < U
    m_p ← m mod p
    if m_p ≠ 0 then
      a_p ← a mod p;    b_p ← b mod p
      g̃_p ← gcd_{F_p[X]}(a_p, b_p)
      u_p ← a_p/g̃_p;    v_p ← b_p/g̃_p
      if deg g̃_p < e then
        // Neue Schranke für deg g; alle bisherigen p sind „schlecht“.
        e ← deg g̃_p;    P ← {p}
      else
        if deg g̃_p = e then
          // Möglicherweise neues „gutes“ p.
          P ← P ∪ {p}
        end if
      end if
    end if
    u ← CRS(P, (u_p)_{p∈P});    v ← CRS(P, (v_p)_{p∈P});    g_1 ← CRS(P, (m_p g̃_p)_{p∈P})
    if ug_1 = ma_1 and vg_1 = mb_1 then
      return pp(g_1)
    end if
  end while
end function
```

In der Praxis werden die Primzahlen $< U$ in ein Datenwort passen. Man kann dann zufällige Primzahlen wählen, die etwas weniger als Wortlänge haben.

Wenn man für die Operationen in $\mathbb{F}_p$ eine Komplexität von $\tilde{O}(\log p)$ ansetzt, dann erhält man hier für die Gesamtkomplexität eine Abschätzung von

$$\tilde{O}(n \log B(n + \log B)) = \tilde{O}\big(n(n^2 + (\log A)^2)\big)$$

(mit $B = 2^n\sqrt{n+1}\,A\,m$ wie vorher). Auch diese lässt sich noch verbessern. In jedem Fall spart man sich das Suchen nach großen Primzahlen, das die Laufzeit der Variante mit großem p dominiert.

11. Schneller Chinesischer Restsatz

„Divide-and-Conquer“-Methoden wie wir sie für die schnelle Multiplikation verwenden können, sind auch anwendbar auf die Berechnung von simultanen Resten modulo vieler Elemente m_j und auf die Rekonstruktion eines Elements aus seinen Resten nach dem Chinesischen Restsatz. Damit lassen sich zum Beispiel die modularen ggT-Algorithmen aus dem vorigen Abschnitt noch weiter beschleunigen.

11.1. **Schnelle Auswertung an vielen Punkten.** Wir betrachten zunächst den Spezialfall $R[X]$ und $m_j = X - a_j$ mit $a_j \in R$, $j = 1, \dots, n$. Sei $f \in R[X]$ mit $\deg f < n$. Dann ist das erste Problem, möglichst schnell den Vektor der Werte $(f(a_1), \dots, f(a_n)) = (f \text{ rem } m_1, \dots, f \text{ rem } m_n)$ zu berechnen. Wenn man nur einen Wert von f bestimmen muss, dann braucht man dazu linearen Aufwand (in $\deg f$), denn man muss jeden Koeffizienten von f betrachten. Wir haben im Zusammenhang mit der FFT gesehen, dass man die Werte von f an $n = 2^k$ speziellen Stellen (den n-ten Einheitswurzeln) mit einem Aufwand $\ll n \log n$ berechnen kann, wenn $\deg f < n$ ist. Wenn wir schnelle Multiplikation und Division von Polynomen zur Verfügung haben, dann dauert es nicht wesentlich länger, $n \approx \deg f$ beliebige Werte von f zu bestimmen.

Der Einfachheit halber nehmen wir jetzt an, dass $n = 2^k$ eine Zweierpotenz ist. Wir setzen

$$M_{k-1,0} = \prod_{j=1}^{2^{k-1}} m_j \qquad \text{und} \qquad M_{k-1,1} = \prod_{j=1}^{2^{k-1}} m_{2^{k-1}+j}\,.$$

Mit $f_0 = f \text{ rem } M_{k-1,0}$ und $f_1 = f \text{ rem } M_{k-1,1}$ gilt dann

$$\big(f(a_1), \dots, f(a_n)\big) = \big(f_0(a_1), \dots, f_0(a_{2^{k-1}}), f_1(a_{2^{k-1}+1}), \dots, f_1(a_n)\big)\,.$$

(Falls $k = 0$, dann ist natürlich $f(a_1) = f$, denn dann ist f konstant, und wir brauchen diese Aufteilung nicht vorzunehmen.) Damit ist das ursprüngliche Problem auf zwei gleichartige Probleme der halben Größe zurückgeführt. Der Aufwand dafür beträgt zwei Divisionen eines Polynoms vom Grad $< n$ durch ein Polynom vom Grad $n/2$ mit Leitkoeffizient 1; das lässt sich in $\tilde{O}(n)$ Operationen in R erledigen. Insgesamt ergibt sich eine Komplexität von

$$\sum_{\kappa=1}^{k} 2^\kappa \tilde{O}(2^{k-\kappa}) \in k\tilde{O}(2^k) \in \tilde{O}(n) \quad \text{Operationen in } R.$$

Dabei ist aber der Aufwand zur Berechnung der $M_{k-1,i}$, und allgemeiner von

$$M_{\kappa,i} = \prod_{j=1}^{2^\kappa} m_{i2^\kappa+j}\,,$$

die in den rekursiven Aufrufen benötigt werden, noch nicht berücksichtigt. Diese Polynome lassen sich bequem nach der Formel

$$M_{0,i} = m_{i+1} = X - a_{i+1}\,, \qquad M_{\kappa+1,i} = M_{\kappa,2i} M_{\kappa,2i+1}$$

rekursiv berechnen. Der Aufwand dafür ist analog wir eben, nur dass wir statt Divisionen hier Multiplikationen benötigen.

Etwas genauer sehen wir:

11.2. **Satz.** *Sei R ein Ring, $a_1, \dots, a_n \in R$ mit $n = 2^k$, und $f \in R[X]$ mit $\deg f < n$. Seien weiter $M(n)$ und $D(n)$ obere Schranken für die Komplexität einer Multiplikation von Polynomen vom Grad $\le n$ bzw. einer Division eines Polynom vom Grad $\le 2n$ durch ein Polynom vom Grad n, ausgedrückt in Operationen in R. Dann lassen sich die Werte $f(a_j) \in R$ für $j = 1, \dots, n$ mit einem Aufwand von $\ll (M(n) + D(n)) \log n$ Operationen in R berechnen.*

Der Satz bleibt gültig, wenn wir die Voraussetzung $n = 2^k$ weglassen. In diesem Fall teilt man die Auswertungspunkte jeweils in zwei etwa gleich große Mengen auf.

Wir haben uns schon überlegt, dass $D(n) \ll M(n) \in \tilde{O}(n)$ gilt. Damit lässt sich der Aufwand (wie oben) etwas gröber als $\tilde{O}(n)$ beschreiben.

11.3. **Schnelle Interpolation.** Jetzt wollen wir das umgekehrte Problem betrachten. Sei dazu wieder R ein Ring und $a_1, \dots, a_n \in R$, so dass $a_i - a_j \in R^\times$ ist für alle $1 \le i < j \le n$. Seien weiter $b_1, \dots, b_n \in R$ gegeben. Wir möchten das Polynom $f \in R[X]$ berechnen mit $\deg f < n$ und $f(a_j) = b_j$ für alle $j = 1, \dots, n$.

Nach der Langrangeschen Interpolationsformel ist

$$f = \sum_{j=1}^{n} b_j \frac{\prod_{i \ne j}(X - a_i)}{\prod_{i \ne j}(a_j - a_i)} = \sum_{j=1}^{n} \frac{b_j}{s_j} \frac{m}{m_j}\,,$$

wenn

$$m = \prod_{j=1}^{n} m_j = \prod_{j=1}^{n} (X - a_j) \quad \text{und} \quad s_j = \prod_{i \ne j} (a_j - a_i)$$

ist. Die Elemente $s_j \in R^\times$ lassen sich wie folgt bestimmen:

$$s_j = m'(a_j) \qquad (\text{denn } m' = \sum_{j=1}^{n} \prod_{i \ne j} (X - a_i);$$

alle bis auf einen Summanden verschwinden, wenn man a_j einsetzt). Für diese Berechnung lässt sich also der in 11.1 beschriebene Algorithmus einsetzen.

Wir brauchen noch ein schnelles Verfahren zur Berechnung von Linearkombinationen der Art

$$\sum_{j=1}^{n} c_j \frac{m}{X - a_j}\,.$$

Wir nehmen wieder an, dass $n = 2^k$ ist; die Bezeichnungen $M_{\kappa,i}$ behalten wir bei und ergänzen sie durch $M_{k,0} = m = M_{k-1,0} M_{k-1,1}$. Dann gilt für $k \ge 1$:

$$\sum_{j=1}^{n} c_j \frac{m}{X - a_j} = M_{k_1,1} \sum_{j=1}^{2^{k-1}} c_j \frac{M_{k-1,0}}{X - a_j} + M_{k-1,0} \sum_{j=1}^{2^{k-1}} c_{2^{k-1}+j} \frac{M_{k-1,1}}{X - a_{2^{k-1}+j}}\,.$$

Wir führen das Problem wieder auf zwei gleichartige Probleme halber Größe zurück und müssen dafür zwei Multiplikationen von Polynomen vom Grad $\le n/2$

(und eine Addition) durchführen. Für diesen Schritt ergibt sich wie oben eine Komplexität (für alle rekursiven Aufrufe zusammen) von $\ll M(n)\log n$ Operationen in R. Dazu kommt die Berechnung der $M_{\kappa,i}$, die vergleichbare Komplexität hat und die Berechnung der s_j in $\ll (M(n)+D(n))\log n$ Operationen in R, sowie n Inversionen und Multiplikationen in R (zur Berechnung von $c_j = b_j/s_j$) und $O(n\log n)$ Additionen. Insgesamt erhalten wir einen Algorithmus, der die gleiche Art von Komplexität aufweist wie der Auswertungsalgorithmus in 11.1. Wie vorher gilt die Komplexitätsaussage auch, wenn n keine Zweierpotenz ist.

11.4. **Verallgemeinerung.** Das Verfahren von 11.1 lässt sich analog anwenden in jedem Ring R (mit geeigneter Division mit Rest) und mit beliebigen m_j. Insbesondere haben wir:

11.5. **Satz.** *Sei K ein Körper, und seien $m_1,\dots,m_n \in K[X]\setminus K$. Setze $m = m_1\cdots m_n$. Sei weiter $f\in K[X]$ mit $\deg f < \deg m$. Dann können wir die Reste*

$$f \text{ rem } m_1, f \text{ rem } m_2, \dots, f \text{ rem } m_n$$

mit einem Aufwand von $\tilde{O}(\deg m)$ Operationen in K berechnen.

11.6. **Satz.** *Seien $m_1,\dots,m_n \in \mathbb{Z}_{>1}$, $m = m_1\cdots m_n$, und $a\in\mathbb{Z}$, $0\le a < m$. Dann können wir*

$$a \text{ rem } m_1, a \text{ rem } m_2, \dots, a \text{ rem } m_n$$

mit einem Aufwand von $\tilde{O}(\lambda(m))$ Wortoperationen berechnen.

Etwas interessanter ist der allgemeine Chinesische Restsatz. Sei R ein euklidischer Ring, und seien $m_1,\dots,m_n \in R\setminus\{0\}$ paarweise teilerfremd. Wir schreiben wieder $m = m_1\cdots m_n$. Sei s_j ein Inverses von m/m_j modulo m_j. Für $b_1,\dots,b_n\in R$ gilt dann

$$\Big(\sum_{i=1}^{n}(b_is_i \text{ rem } m_i)\frac{m}{m_i}\Big) \text{ rem } m_j = \Big(b_js_j\frac{m}{m_j}\Big) \text{ rem } m_j = b_j \text{ rem } m_j\,,$$

denn $s_j(m/m_j)\equiv 1 \bmod m_j$. Damit ist

$$x = \sum_{i=1}^{n}(b_is_i \text{ rem } m_i)\frac{m}{m_i}$$

eine Lösung des Systems $x\equiv b_j \bmod m_j$ $(1\le j\le n)$ von Kongruenzen. Das verallgemeinert die Lagrange-Interpolationsformel.

Ist $R = K[X]$, dann gilt $\deg x < \deg m$, und x ist die kanonische Lösung. Für $R=\mathbb{Z}$ gilt $0\le x < nm$; hier muss man also im allgemeinen noch x rem m berechnen, was aber auch schnell geht.

Zur Berechnung der s_j kann man wie folgt vorgehen: Wir berechnen erst

$$r_j = m \text{ rem } m_j^2 \qquad \text{für alle } 1\le j\le n$$

mit dem Algorithmus, der den Sätzen 11.5 und 11.6 zu Grunde liegt. Dann gilt für $t_j = r_j/m_j$ (die Division geht auf) $t_j = (m/m_j)$ rem m_j, und wir können s_j als Inverses von t_j mod m_j mit dem Erweiterten Euklidischen Algorithmus berechnen. Dafür gibt es ebenfalls einen schnellen Algorithmus, dessen Aufwand $\tilde{O}(\deg m_j)$ Operationen in K (für $R=K[X]$) bzw. $\tilde{O}(\lambda(m_j))$ Wortoperationen (für $R=\mathbb{Z}$) ist. Insgesamt ist der Aufwand für diese Vorberechnung in der gleichen Größenordnung $\tilde{O}(\deg m)$ bzw. $\tilde{O}(\lambda(m))$ wie die anderen Teile der Berechnung.

Wenn wir die s_j und die Teilprodukte $M_{\kappa,i}$ berechnet haben, können wir die Linearkombination x wie oben mit einem Algorithmus analog zu 11.3 berechnen. Wir erhalten analog zu 11.4 folgende Resultate.

11.7. **Satz.** *Sei K ein Körper, und seien $m_1, \ldots, m_n \in K[X] \setminus K$ paarweise teilerfremd. Seien weiter $b_1, \ldots, b_n \in K[X]$ mit $\deg b_j < \deg m_j$. Setze $m = m_1 \cdots m_n$. Dann können wir das Polynom $f \in K[X]$ mit $f \operatorname{rem} m_j = b_j$ für $1 \leq j \leq n$ und $\deg f < \deg m$ in $\tilde{O}(\deg m)$ Operationen in K berechnen.*

11.8. **Satz.** *Seien $m_1, \ldots, m_n \in \mathbb{Z}_{>1}$ paarweise teilerfremd. Setze $m = m_1 \cdots m_n$. Seien weiter $b_1, \ldots, b_n \in \mathbb{Z}$ mit $0 \leq b_j < m_j$. Dann können wir die ganze Zahl $a \in \mathbb{Z}$ mit $a \operatorname{rem} m_j = b_j$ für $1 \leq j \leq n$ und $0 \leq a < m$ in $\tilde{O}(\lambda(m))$ Wortoperationen berechnen.*

Wir illustrieren den Algorithmus hinter Satz 11.8 mit einem einfachen Beispiel. Seien

$$(m_1, \ldots, m_4) = (11, 13, 17, 19)$$

und

$$(b_1, \ldots, b_4) = (2, 3, 5, 7)\,.$$

Wir berechnen zunächst die Teilprodukte

$$M_{1,0} = 11 \cdot 13 = 143\,, \quad M_{1,1} = 17 \cdot 19 = 323\,, \qquad M_{2,0} = m = 143 \cdot 323 = 46\,189\,.$$

Wir berechnen $r_j = m \operatorname{rem} m_j^2$:

$$\begin{aligned} m \operatorname{rem} M_{1,0}^2 = 5291\,, &\quad m \operatorname{rem} M_{1,1}^2 = 46\,189 \\ r_1 = 5291 \operatorname{rem} m_1^2 = 88\,, &\quad r_2 = 5291 \operatorname{rem} m_2^2 = 52\,, \\ r_3 = 46\,189 \operatorname{rem} m_3^2 = 238\,, &\quad r_4 = 46\,189 \operatorname{rem} m_4^2 = 342\,. \end{aligned}$$

Dann haben wir

$$t_1 = \frac{r_1}{m_1} = 8\,, \quad t_2 = \frac{r_2}{m_2} = 4\,, \quad t_3 = \frac{r_3}{m_3} = 14\,, \quad t_4 = \frac{r_4}{m_4} = 18\,.$$

Als Inverse mod m_j erhalten wir

$$s_1 = 7\,, \quad s_2 = 10\,, \quad s_3 = 11\,, \quad s_4 = 18\,.$$

Die Koeffizienten der Linearkombination von m/m_j sind

$$\begin{aligned} c_1 = (b_1 s_1) \operatorname{rem} m_1 = 3\,, &\quad c_2 = (b_2 s_2) \operatorname{rem} m_2 = 4\,, \\ c_3 = (b_3 s_3) \operatorname{rem} m_3 = 4\,, &\quad c_4 = (b_4 s_4) \operatorname{rem} m_4 = 12\,. \end{aligned}$$

Die Berechnung der Linearkombination beginnt mit $c_1, \ldots, c_4$ und schreitet dann fort:

$$m_2 c_1 + m_1 c_2 = 83\,, \quad m_4 c_3 + m_3 c_4 = 280\,, \quad x = M_{1,1} \cdot 83 + M_{1,0} \cdot 280 = 66\,849\,,$$

und schließlich

$$x \operatorname{rem} m = 20\,660\,.$$

11.9. **Schneller ggT.** Auch für die ggT-Berechnung gibt es schnelle Algorithmen, die auf einem „Divide and Conquer“-Ansatz beruhen. Die zu Grunde liegende Idee dabei ist, dass der Anfang der Folge $q_1, q_2, \ldots$ der sukzessiven Quotienten im Euklidischen Algorithmus, angewandt auf a und b, nur von den „Anfängen“ (d.h., den höherwertigen Teilen) von a und b abhängt. Wenn

$$r_0 = a, r_1 = b, r_2 = r_0 - q_1 r_0, r_3 = r_1 - q_2 r_2, \ldots, r_\ell = 0$$

die Folge der sukzessiven Reste ist, dann lassen sich r_k und r_{k+1} aus a und b und $q_1, \ldots, q_k$ berechnen:

$$\begin{pmatrix} r_k \\ r_{k+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & -q_k \end{pmatrix} \begin{pmatrix} r_{k-1} \\ r_k \end{pmatrix} = Q_k \begin{pmatrix} r_{k-1} \\ r_k \end{pmatrix} = \cdots = Q_k Q_{k-1} \cdots Q_1 \begin{pmatrix} a \\ b \end{pmatrix} = R_k \begin{pmatrix} a \\ b \end{pmatrix}$$

mit

$$Q_k = \begin{pmatrix} 0 & 1 \\ 1 & -q_k \end{pmatrix} \quad \text{und} \quad R_k = Q_k Q_{k-1} \cdots Q_1 \,.$$

Der Algorithmus sieht dann etwa so aus:

(1) Ist a oder b „klein“, dann berechne $\mathrm{ggT}(a, b)$ und $R_{\ell-1}$ direkt.
(2) Rufe den Algorithmus rekursiv auf mit ausreichend langen Anfangsstücken von a und b und berechne R_k für geeignetes k (z.B. $k \approx \ell/2$).
(3) Berechne
$$\begin{pmatrix} r_k \\ r_{k+1} \end{pmatrix} = R_k \begin{pmatrix} a \\ b \end{pmatrix} \,.$$
(4) Falls $r_{k+1} = 0$, gib r_k und R_k zurück.
(5) Berechne q_{k+1}, r_{k+2}, Q_{k+1}, R_{k+1}.
(6) Rufe den Algorithmus rekursiv auf mit $(a, b) \leftarrow (r_{k+1}, r_{k+2})$ und berechne $g = \mathrm{ggT}(r_{k+1}, r_{k+2})$ und $R'_{k+1} = Q_{\ell-1} \cdots Q_{k+3} Q_{k+2}$.
(7) Gib g und $R_{\ell-1} = R'_{k+1} R_{k+1}$ zurück.

Siehe [GG, § 11] für eine detaillierte Beschreibung dieses Ansatzes, wenn a und b Polynome über einem Körper sind.

Man erhält folgende Aussagen:

11.10. **Satz.** *Sei K ein Körper, $a, b \in K[X]$. Dann können wir die Resultate des Erweiterten Euklidischen Algorithmus, angewandt auf a und b, mit einem Aufwand von $\tilde{O}(\max\{\deg a, \deg b\})$ Operationen in K berechnen.*

Die Resultate des EEA sind dabei (g, u, v) mit $g = \mathrm{ggT}(a, b)$ und $ua + vb = g$.

11.11. **Satz.** *Seien $a, b \in \mathbb{Z}$. Dann können wir die Resultate des EEA, angewandt auf a und b, mit einem Aufwand von $\tilde{O}(\max\{\lambda(a), \lambda(b)\})$ Wortoperationen berechnen.*

Als Folgerungen ergeben sich unmittelbar:

11.12. **Folgerung.** *Sei K ein Körper $f \in K[X]$ irreduzibel. Dann lassen sich die Körperoperationen im Körper $L = K[X]/fK[X]$ mit einem Aufwand von $\tilde{O}(\deg f)$ Operationen in K durchführen.*

11.13. **Folgerung.** *Sei p eine Primzahl, Dann lassen sich die Körperoperationen im Körper $\mathbb{F}_p$ mit einem Aufwand von $\tilde{O}(\lambda(p))$ Wortoperationen durchführen.*

Beide Aussagen gelten allgemeiner auch wenn f nicht irreduzibel, bzw. p nicht prim ist, wenn man die Division ersetzt durch „teste, ob a invertierbar ist, und falls ja, berechne das Inverse“.

11.14. **Übung.** Benutzen Sie die hier formulierten Aussagen, um die Komplexitätsabschätzung für die modularen ggT-Algorithmen in $K[X,Y]$ und $\mathbb{Z}[X]$ zu verbessern.

12. Faktorisierung von Polynomen über endlichen Körpern

Unser nächstes Thema wird die Faktorisierung sein: Ist R ein faktorieller Ring und $a \in R \setminus \{0\}$, so lässt sich a in eindeutiger Weise schreiben in der Form

$$a = u \prod_p p^{e_p},$$

wo $u \in R^\times$ eine Einheit ist, p alle normierten Primelemente von R durchläuft und $e_p \in \mathbb{Z}_{\geq 0}$ ist für alle p und $e_p = 0$ für alle bis auf endlich viele p. Das Faktorisierungsproblem besteht darin, zu gegebenem a die Einheit u und die Paare (p, e_p) zu finden, für die $e_p > 0$ ist.

Die faktoriellen Ringe, die wir kennen, haben die Form

$$R = \mathbb{Z}[X_1, \ldots, X_n] \quad \text{oder} \quad R = K[X_1, \ldots, X_n]$$

mit einem Körper K, dabei ist $n \geq 0$. Im Gegensatz zu anderen Fragestellungen (wie zum Beispiel der Berechnung von größten gemeinsamen Teilern) hängen die zu verwendenden Algorithmen stark vom Koeffizientenring ab. Für viele andere Fälle grundlegend ist der Fall $R = \mathbb{F}_q[X]$ eines Polynomrings (in einer Variablen) über einem endlichen Körper $\mathbb{F}_q$.

12.1. **Wiederholung: Endliche Körper.** Wir erinnern uns an einige wichtige Tatsachen über endliche Körper.

12.1.1. *Ist K ein endlicher Körper, dann gibt es eine Primzahl p und eine positive ganze Zahl e, so dass $\#K = p^e$. Dann ist K eine Körpererweiterung vom Grad e des „Primkörpers“ $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$.*

Zum Beweis betrachtet man den kanonischen Ringhomomorphismus $\varphi : \mathbb{Z} \to K$. Sei Bild ist ein endlicher Integritätsring, also gilt $\ker \varphi = p\mathbb{Z}$ für eine Primzahl p, und K enthält (ein isomorphes Bild von) $\mathbb{F}_p$. Dann ist K ein endlich-dimensionaler Vektorraum über $\mathbb{F}_p$; sei $\dim_{\mathbb{F}_p} K = e$. Die Behauptung folgt.

12.1.2. *Sei K ein endlicher Körper mit $\#K = q$ und L/K eine Körpererweiterung. Dann gilt $K = \{x \in L \mid x^q = x\}$.*

Die multiplikative Gruppe $K^\times$ hat Ordnung $q-1$. Daher gilt für jedes $x \in K^\times$, dass $x^{q-1} = 1$, also auch $x^q = x$ ist. Letzteres gilt natürlich auch für $x = 0$. Das Polynom $X^q - X \in L[X]$ hat höchstens q Nullstellen in L. Wir kennen aber bereits q Nullstellen, nämlich die Elemente von K. Daher sind die Elemente von K genau die Nullstellen von $X^q - X$.

Damit folgt:

12.1.3. $X^q - X = \prod_{\alpha \in K} (X - \alpha)$.

12.1.4. *Sei L/K eine Körpererweiterung, und $\#K = q$. Dann ist die Abbildung $\phi : L \to L$, $x \mapsto x^q$, ein K-linearer Ringhomomorphismus.*

Dass $\phi(xy) = (xy)^q = x^q y^q = \phi(x)\phi(y)$ gilt, ist klar. Nach 12.1.2 gilt $a^q = a$ für $a \in K$, also folgt $\phi(a) = a$ (und insbesondere $\phi(1) = 1$). Sei $q = p^e$. Dann gilt auch $(x+y)^p = x^p + y^p$ für $x, y \in L$, denn die inneren Binomialkoeffizienten sind alle durch p teilbar, verschwinden also in einem Körper der Charakteristik p. Durch Induktion folgt dann $\phi(x+y) = (x+y)^q = x^q + y^q = \phi(x) + \phi(y)$.

Ist die Körpererweiterung endlich, dann ist ϕ ein Automorphismus von L/K (denn ϕ ist offensichtlich injektiv: $\ker \phi = 0$, also wegen der endlichen Dimension des K-Vektorraums L auch surjektiv); ϕ heißt der *Frobeniusautomorphismus* von L/K.

12.1.5. *Sei p prim, $e \geq 1$ und $q = p^e$. Dann gibt es bis auf Isomorphie genau einen Körper $\mathbb{F}_q$ mit q Elementen.*

Sei K ein algebraischer Abschluss von $\mathbb{F}_p$. Das Polynom $X^q - X$ ist separabel (denn seine Ableitung ist -1, hat also keine gemeinsamen Nullstellen mit $X^q - X$; beachte $q \cdot 1_{\mathbb{F}_p} = 0$), also hat die Menge

$$\mathbb{F}_q = \{\alpha \in K \mid \alpha^q = \alpha\}$$

genau q Elemente. Nach 12.1.4 ist $\mathbb{F}_q$ die Menge der Fixpunkte eines $\mathbb{F}_p$-linearen Ringhomomorphismus, also ist $\mathbb{F}_q$ eine $\mathbb{F}_p$-Algebra. Da $\mathbb{F}_q$ endlich und nullteilerfrei ist, muss $\mathbb{F}_q$ ein Körper sein. Das zeigt die Existenz.

Zur Eindeutigkeit: Sei K wie eben. Jeder Körper F mit q Elementen ist eine endliche Körpererweiterung von $\mathbb{F}_p$, also lässt sich F in K einbetten: F ist isomorph zu einem Unterkörper von K mit q Elementen. Aus 12.1.2 folgt aber, dass es genau einen solchen Unterkörper gibt, nämlich $\mathbb{F}_q$. Also ist jeder Körper mit q Elementen zu $\mathbb{F}_q$ isomorph.

Aussage 12.1.3 hat eine Verallgemeinerung:

12.1.6. *$X^{q^n} - X \in \mathbb{F}_q[X]$ ist das Produkt aller normierten irreduziblen Polynome in $\mathbb{F}_q[X]$, deren Grad ein Teiler von n ist.*

Der Körper $\mathbb{F}_{q^n}$ ist eine Körpererweiterung vom Grad n von $\mathbb{F}_q$. Nach 12.1.3 gilt in $\mathbb{F}_{q^n}[X]$:

$$X^{q^n} - X = \prod_{\alpha \in \mathbb{F}_{q^n}} (X - \alpha)\,.$$

Sei $\alpha \in \mathbb{F}_{q^n}$ und M_α das Minimalpolynom von α über $\mathbb{F}_q$. Da $\mathbb{F}_q \subset \mathbb{F}_q(\alpha) \subset \mathbb{F}_{q^n}$, gilt

$$\deg M_\alpha = \dim_{\mathbb{F}_q} \mathbb{F}_q(\alpha) \mid n\,.$$

Das Minimalpolynom M_α teilt jedes Polynom in $\mathbb{F}_q[X]$, das α als Nullstelle hat; da verschiedene Minimalpolynome paarweise teilerfremd sind, folgt

$$\prod_{M \in \{M_\alpha \mid \alpha \in \mathbb{F}_q\}} M \mid X^{q^n} - X\,.$$

Da die linke Seite aber alle $\alpha \in \mathbb{F}_{q^n}$ als Nullstellen hat (und beide Polynome normiert sind), gilt Gleichheit. Sei nun f irgend ein normiertes irreduzibles Polynom, dessen Grad n teilt, und sei $\beta \in K$ eine Nullstelle von f. Die Körpererweiterung $\mathbb{F}_q(\beta)$ muss dann (als Körper mit $q^{\deg f}$ Elementen) gerade $\mathbb{F}_{q^{\deg f}}$ sein, ist also in $\mathbb{F}_{q^n}$ enthalten. Es folgt $\beta \in \mathbb{F}_{q^n}$, und $f = M_\beta$.

12.1.7. *Sei K ein endlicher Körper. Dann ist die multiplikative Gruppe $K^\times$ zyklisch.*

Das folgt aus einem bekannten Satz der Algebra: Jede endliche Untergruppe der multiplikativen Gruppe eines Körpers ist zyklisch. Hier ist sogar die ganze multiplikative Gruppe endlich.

12.1.8. *Sei K ein endlicher Körper und $k \in \mathbb{Z}_{>1}$ mit $\#K \equiv 1 \bmod k$. Dann gilt*

$$\{a^k \mid a \in K^\times\} = \{a \in K \mid a^{(\#K-1)/k} = 1\},$$

und diese Menge hat genau $(\#K-1)/k$ Elemente.

Sei $\#K = q$. Nach 12.1.7 ist $K^\times$ zyklisch; außerdem gilt $\#K^\times = q-1$, also ist k ein Teiler der Ordnung von $K^\times$. Daher hat $K^\times$ genau eine Untergruppe der Ordnung $(q-1)/k$, die durch die Menge auf der rechten Seite gegeben ist. Die linke Seite ist wegen $a^{q-1} = 1$ in der rechten Seite enthalten und hat mindestens $(q-1)/k$ Elemente (denn die Abbildung $a \mapsto a^k$ hat Fasern der Größe $\leq k$), also müssen wir Gleichheit haben.

Jetzt können wir uns der Faktorisierung von Polynomen über $K = \mathbb{F}_q$ zuwenden. Sei also $f \in \mathbb{F}_q[X]$; wir können annehmen, dass f Leitkoeffizient 1 hat, und wir werden zunächst einmal voraussetzen, dass f quadratfrei ist (dass also in der Faktorisierung von f keine Faktoren mehrfach auftreten).

12.2. **Trennung der irreduziblen Faktoren nach ihrem Grad.** Der erste Schritt besteht darin, die irreduziblen Faktoren von f nach ihrem Grad zu trennen. Zum Beispiel gilt nach Aussage 12.1.3, dass

$$\mathrm{ggT}(f, X^q - X) = \prod_{\alpha \in \mathbb{F}_q, f(\alpha)=0} (X - \alpha)$$

das Produkt der irreduziblen Faktoren von f vom Grad 1 ist. Die Berechnung von $\mathrm{ggT}(f, X^q - X)$ erfolgt dabei am besten als $\mathrm{ggT}(f, (X^q \text{ rem } f) - X)$, wobei wir X^q rem f durch sukzessives Quadrieren im Ring $\mathbb{F}_q[X]/f\mathbb{F}_q[X]$ effizient berechnen können.

Zur Isolation der Faktoren mit höherem Grad verwenden wir entsprechend Aussage 12.1.6. Das führt auf folgenden Algorithmus zur „Distinct Degree Factorization“.

```
function ddf(f)
  // f ∈ F_q[X] quadratfrei mit lcf(f) = 1.
  // Ausgabe: (u_1, u_2, ..., u_d) mit u_j ∈ F_q[X] Produkt von normierten
  //    irreduziblen Polynomen vom Grad j, u_d ≠ 1 und f = u_1 ··· u_d.
  u ← () // leere Liste
  a ← X // a ≡ X^{q^j} mod f
  while deg f > 0 do
    a ← a^q rem f // sukzessives Quadrieren in F_q[X]/fF_q[X]
    h ← gcd(f, a − X)
    u ← append(u, h) // Liste verlängern
    f ← f/h
  end while
  return u
end function
```

Zum Beweis der Korrektheit zeigt man, dass im j-ten Schleifendurchlauf das Polynom f nur noch irreduzible Faktoren vom Grad $\geq j$ enthält (das ist klar für $j = 1$). Dann ist $h = \mathrm{ggT}(f, X^{q^j} - X)$ das Produkt der in f enthaltenen irreduziblen Faktoren, deren Grad j teilt. Das sind dann aber gerade die irreduziblen Faktoren vom Grad j. Damit hat $u_j = h$ den korrekten Wert, und der neue Wert von f hat nur noch irreduzible Faktoren vom Grad $> j$.

Zur Komplexität: Im „schlimmsten“ Fall ist f irreduzibel, dann muss die Schleife $n = \deg f$ Mal durchlaufen werden, und die Variable f hat immer den gleichen Wert. Die Berechnung von a^q rem f braucht $\tilde{O}(n \log q)$ Operationen in $\mathbb{F}_q$, die Berechnung von h und die Division f/h höchstens $\tilde{O}(n)$ Operationen in $\mathbb{F}_q$. Insgesamt haben wir also eine Komplexität von $\tilde{O}(n^2 \log q)$ Operationen in $\mathbb{F}_q$ oder $\tilde{O}(n^2 (\log q)^2)$ Wortoperationen (denn Operationen in $\mathbb{F}_q$ können mit $\tilde{O}(\log q)$ Wortoperationen ausgeführt werden). Da die Größe der Eingabe f von der Größenordnung $n \log q$ ist, haben wir hier einen Algorithmus von im wesentlichen quadratischer Komplexität.

Man kann das Programm etwas schneller beenden, wenn $\deg f < 2j$ ist, denn dann muss f irreduzibel sein, und man kann $u_j = \cdots = u_{\deg f - 1} = 0$ und $u_{\deg f} = f$ setzen. Im besten Fall spart man sich so die Hälfte der Arbeit.

12.3. **Übung.** Sei $n = \deg f$ fixiert. Was ist, für $q \to \infty$, die Wahrscheinlichkeit p_n dafür, dass obiger Algorithmus schon die vollständige Faktorisierung von f liefert? (D.h., dass alle irreduziblen Faktoren verschiedenen Grad haben.) Dabei sei angenommen, dass f zufällig und gleichverteilt aus den q^n normierten Polynomen in $\mathbb{F}_q[X]$ vom Grad n ausgewählt wird.

Zum Beispiel gilt $p_1 = 1$, $p_2 = \frac{1}{2}$ (ein Polynom vom Grad 2 ist für große q etwa mit Wahrscheinlichkeit $\frac{1}{2}$ irreduzibel), $p_3 = \frac{5}{6}$ ($\frac{1}{3}$ für irreduzibel, $\frac{1}{2}$ für irreduzible Faktoren mit den Graden 1 und 2).

12.4. **Bestimmung der Nullstellen in $\mathbb{F}_q$.** Um die Faktorisierung zu vervollständigen, müssen wir Produkte der Form $f = h_1 h_2 \cdots h_k$ faktorisieren, wobei die Polynome h_j paarweise verschiedene normierte irreduzible Polynome des selben Grades d sind. Es gilt dann $k = (\deg f)/d$; wir wissen also, wie viele Faktoren es sind. Gilt $d = \deg f$, dann ist $k = 1$, und f ist irreduzibel.

Wir betrachten erst den Fall $d = 1$. Dann ist f ein Produkt verschiedener Polynome der Form $X - \alpha$, und es geht darum, die Nullstellen von f in $\mathbb{F}_q$ zu finden. Eine Möglichkeit besteht darin, alle $\alpha \in \mathbb{F}_q$ durchzuprobieren. Mit den effizienten Methoden zur Auswertung in mehreren Punkten geht das in $\tilde{O}(q)$ Operationen in $\mathbb{F}_q$. Das ist vertretbar, wenn q vergleichsweise klein ist, wird aber für große q untolerierbar langsam — der Aufwand wächst exponentiell mit der Eingabegröße.

Um zu einer effizienteren Methode zu kommen, erinnern wir uns an das Prinzip von „Divide and Conquer“: Wir sollten versuchen, das Problem in zwei gleichartige Probleme der halben Größe zu zerlegen. Wir wollen also $f = f_1 f_2$ faktorisieren, wobei die Faktoren f_1 und f_2 etwa den gleichen Grad $\approx (\deg f)/2$ haben. Dazu teilen wir $\mathbb{F}_q$ in zwei etwa gleich große Teilmengen S_1 und S_2 auf und berechnen

$$f_1 = \mathrm{ggT}\Big(f, \prod_{\alpha \in S_1} (X - \alpha)\Big) \quad \text{und} \quad f_2 = \mathrm{ggT}\Big(f, \prod_{\alpha \in S_2} (X - \alpha)\Big).$$

Wir können natürlich Pech haben, und (fast) alle Nullstellen von f liegen in einer der beiden Teilmengen. Aber wenn wir die Aufteilung in hinreichend zufälliger

Weise vornehmen, dann ist die Wahrscheinlichkeit dafür sehr klein. Das führt dann in natürlicher Weise auf einen probabilistischen Algorithmus. Es ist ein offenes Problem, ob es einen *deterministischen* Algorithmus gibt, der in Polynomzeit (polynomial in $\deg f$ und $\log q$) wenigstens eine Nullstelle von f bestimmt.

Das praktische Problem ist, wie man f_1 und f_2 effizient berechnen kann. Dazu erinnern wir uns an Eigenschaft 12.1.8 von endlichen Körpern: Wenn q ungerade ist, dann sind genau die Hälfte der Elemente von $\mathbb{F}_q^\times$ Quadrate, und zwar genau die $a \in \mathbb{F}_q^\times$ mit $a^{(q-1)/2} = 1$. Das Polynom $X^{(q-1)/2} - 1$ hat also genau $(q-1)/2$ verschiedene Nullstellen in $\mathbb{F}_q$. Dasselbe gilt natürlich für $(X-a)^{(q-1)/2} - 1$. Das liefert folgenden Algorithmus:

```
function zeros(f)
  // f ∈ F_q[X] normiert mit f | X^q − X, q ungerade.
  // Ausgabe: Z ⊂ F_q, die Menge der Nullstellen von f.
  if deg f = 0 then return ∅ end if
  if deg f = 1 then return {−f(0)} end if
  a ← zufälliges Element von F_q
  h ← (X − a)^{(q−1)/2} rem f // sukzessives Quadrieren
  f_1 ← gcd(f, h − 1)
  f_2 ← f/f_1
  return zeros(f_1) ∪ zeros(f_2)
end function
```

Die Komplexität ist $\tilde{O}(n \log q)$ Operationen in $\mathbb{F}_q$ mal die Rekursionstiefe (mit $n = \deg f$ wie oben), denn der Aufwand für einen Durchlauf ist $\tilde{O}(n \log q)$, analog zum vorigen Algorithmus. Man kann erwarten, dass die Rekursionstiefe $\ll \log n$ ist, denn das gilt, wenn f_1 und f_2 etwa den gleichen Grad haben. Etwas genauer kann man so argumentieren: Seien $\alpha, \beta \in \mathbb{F}_q$ zwei verschiedene Nullstellen von f. Dann werden α und β mit Wahrscheinlichkeit nahe bei $\frac{1}{2}$ in einem Durchlauf getrennt. Der Erwartungswert der Anzahl der Paare von Nullstellen, die nach k rekursiven Aufrufen noch nicht getrennt sind, ist also etwa

$$\binom{n}{2} 2^{-k},$$

und das ist sehr klein, sobald $k \gg \log n$ ist. Insgesamt ist die erwartete Komplexität

$$\tilde{O}(n \log n \log q) \in \tilde{O}(n \log q) \quad \text{Operationen in } \mathbb{F}_q.$$

Wenn $q = 2^m$ ist, $\mathbb{F}_q$ also Charakteristik 2 hat, kann man statt dessen verwenden, dass die Funktion

$$\mathrm{Tr}_{\mathbb{F}_q/\mathbb{F}_2} : x \longmapsto x + x^2 + x^4 + \ldots + x^{q/2}$$

für genau die Hälfte der Elemente $x \in \mathbb{F}_q$ den Wert 0 und für die andere Hälfte den Wert 1 annimmt. (Tr ist die *Spur* der Körpererweiterung $\mathbb{F}_q/\mathbb{F}_2$, eine $\mathbb{F}_2$-lineare Abbildung $\mathbb{F}_q \to \mathbb{F}_2$, die surjektiv ist, weil $\mathbb{F}_q/\mathbb{F}_2$ separabel ist.) Man kann zeigen, dass man jede Untergruppe vom Index 2 der additiven Grupp $\mathbb{F}_q$ erhält als Kern von $x \mapsto \mathrm{Tr}_{\mathbb{F}_q/\mathbb{F}_2}(ax)$ mit einem $a \in \mathbb{F}_q^\times$. Man hat dann also die folgende Modifikation:

```
function zeros2(f)
  // f ∈ F_q[X] normiert mit f | X^q − X, q = 2^m.
  // Ausgabe: Z ⊂ F_q, die Menge der Nullstellen von f.
  if deg f = 0 then return ∅ end if
```

```
  if deg f = 1 then return {f(0)} end if
  a ← zufälliges Element von F_q^×
  g ← aX; h ← g
  for i = 1 to m − 1 do
    g ← g^2 rem f ; h ← h + g
  end for
  f_1 ← gcd(f, h)
  f_2 ← f/f_1
  return zeros2(f_1) ∪ zeros2(f_2)
end function
```

12.5. **Trennung irreduzibler Faktoren gleichen Grades.** Die Idee, die wir zur Nullstellenbestimmung verwendet haben, lässt sich verallgemeinern. Wir nehmen an, $f \in \mathbb{F}_q[X]$ sei ein Polynom mit Leitkoeffizient 1, das Produkt von k verschiedenen irreduziblen Polynomen vom Grad d ist. Dann ist $n = \deg f = kd$. Wie vorher nehmen wir an, dass q ungerade ist. Wir schreiben

$$f = h_1 \cdots h_k$$

mit irreduziblen normierten Polynomen $h_j \in \mathbb{F}_q[X]$ vom Grad d. Nach dem Chinesischen Restsatz gilt dann, dass

$$\begin{aligned} \varphi : \mathbb{F}_q[X]/f\mathbb{F}_q[X] &\longrightarrow \prod_{j=1}^{k} \mathbb{F}_q[X]/h_j\mathbb{F}_q[X] \cong (\mathbb{F}_{q^d})^k \\ a &\longmapsto (a \bmod h_1, \ldots, a \bmod h_j) \end{aligned}$$

ein Isomorphismus von Ringen ist. Wählen wir a zufällig und gleichverteilt in $\mathbb{F}_q[X]/f\mathbb{F}_q[X]$, dann ist $\varphi(a)$ ein zufälliges Element in $(\mathbb{F}_{q^d})^k$. Wenn $a \perp f$, dann ist $\varphi(a)$ ein zufälliges Element von $(\mathbb{F}_{q^d}^\times)^n$, und $b = \varphi(a^{(q^d-1)/2}) = \varphi(a)^{(q^d-1)/2}$ ist ein zufälliges Element von $\{\pm 1\}^k$. Wir schreiben $b = (b_1, \ldots, b_k)$ mit $b_j \in \{\pm 1\}$. Es gilt dann

$$g := \mathrm{ggT}(f, a^{(q^d-1)/2} - 1) = \prod_{j:b_j=1} h_j \,.$$

Mit einer Wahrscheinlichkeit von $2^{1-k} \le 1/2$ (für $k \ge 2$) ist $b \notin \{\pm 1\}$ (d.h., die b_j sind nicht alle gleich); dann ist $g \ne 1$ und $g \ne f$, so dass $f = g \cdot (f/g)$ eine nichttriviale Faktorisierung ist. Wir wenden dann die selbe Idee rekursiv auf g und auf f/g an und erhalten folgenden Algorithmus für die „Equal Degree Factorization“.

```
function edf(f, d)
  // f ∈ F_q[X], q ungerade, f normiert und Produkt von verschiedenen
  //    irreduziblen Polynomen vom Grad d.
  // Ausgabe: (h_1, ..., h_k) mit h_j irreduzibel, f = h_1 ··· h_k.
  if deg f = 0 then return () end if // f konstant
  if deg f = d then return (f) end if // f irreduzibel
  // ab hier ist k ≥ 2.
  a ← zufälliges Polynom in F_q[X] vom Grad < deg f
  g ← gcd(f, a) // Test ob a ⊥ f
  if g ≠ 1 then return edf(g, d) cat edf(f/g, d) end if
  g ← gcd(f, a^{(q^d−1)/2} rem f − 1)
  return edf(g, d) cat edf(f/g, d)
end function
```

Hier ist

$$(a_1, \ldots, a_m) \text{ cat } (b_1, \ldots, b_n) = (a_1, \ldots, a_m, b_1, \ldots, b_n)\,.$$

Die Kosten für einen Durchlauf betragen ($n = \deg f$ wie oben)

- $\tilde{O}(n \log q)$ für die Wahl von a
- $\tilde{O}(n)$ für $\mathrm{ggT}(f, a)$
- $\tilde{O}(n\, d \log q)$ für $a^{(q^d-1)/2}$ rem f durch sukzessives Quadrieren
- $\tilde{O}(n)$ für den zweiten ggT

Operationen in $\mathbb{F}_q$. Der dritte Schritt ist dominant; wir haben also

$$\tilde{O}(nd \log q) \quad \text{Operationen in } \mathbb{F}_q.$$

Wenn wir die rekursiven Aufrufe als Binärbaum darstellen, dann ist die Summe der Werte von n aller Aufrufe auf der selben Ebene des Baumes immer höchstens gleich dem ursprünglichen n (denn $\deg g + \deg(f/g) = \deg f$). Der Gesamtaufwand ist also höchstens

$$\tilde{O}(rnd \log q) \quad \text{Operationen in } \mathbb{F}_q,$$

wobei r die maximale Rekursionstiefe bezeichnet. Wir müssen also den Erwartungswert von r betrachten.

Wenn wir die (für q und/oder d groß sehr seltenen) Fälle außer Acht lassen, in denen a und f nicht teilerfremd sind, dann können wir das Verfahren wie folgt interpretieren: In jedem Durchgang wird die jeweils aktuelle Menge von irreduziblen Faktoren in zwei disjunkte Teilmengen aufgeteilt; dabei ist jede mögliche Aufteilung gleich wahrscheinlich. Sei $E(k)$ der Erwartungswert der Rekursionstiefe bei k Faktoren, dann gilt

$$E(0) = E(1) = 0\,, \qquad E(k) = 1 + 2^{-k} \sum_{l=0}^{k} \binom{k}{l} \max\{E(l), E(k-l)\} \quad \text{für } k \geq 2.$$

Es ist klar, dass $E(k)$ monoton steigt. Damit erhalten wir die übersichtlichere Rekursion

$$E(k) = 1 + 2^{1-k} \sum_{l=0}^{\lfloor k/2 \rfloor} \binom{k}{l} E(k-l) \quad \Big[+ 2^{-k} \binom{k}{k/2} E(k/2) \Big]\,;$$

der letzte Summand ist nur vorhanden, wenn k gerade ist. Wir erhalten zum Beispiel

$$E(2) = 1 + \frac{1}{2} \cdot E(2) + \frac{1}{4} \cdot E(1) = 1 + \frac{E(2)}{2}\,,$$

also $E(2) = 2$, dann

$$E(3) = 1 + \frac{1}{4}\big(E(3) + 3E(2)\big) = \frac{5}{2} + \frac{E(3)}{4}\,,$$

also $E(3) = \frac{10}{3}$ usw.

Etwas einfacher wird es, wenn wir alle Auswahlen in einer Ebene des Baumes zusammenfassen. Dann haben wir eine zufällige Folge $(v_m)_{m \geq 1}$ von Elementen in $\{\pm 1\}^k$, und die maximale Rekursionstiefe ist der kleinste Index r, so dass es für jedes Paar $1 \leq i < j \leq k$ stets ein v_m mit $m \leq r$ gibt, so dass die Einträge an den Positionen i und j in v_m verschieden sind. Für ein gegebenes Paar (i, j) ist die

Wahrscheinlichkeit, dass dies nicht der Fall ist, 2^{-r}. Die Wahrscheinlichkeit, dass es ein noch nicht getrenntes Paar gibt, ist demnach

$$p_r \le \binom{k}{2} 2^{-r} < k^2 2^{-r-1}\,.$$

Für den Erwartungswert ergibt sich dann

$$\begin{aligned} E(k) &= \sum_{r=1}^{\infty} r(p_{r-1} - p_r) = \sum_{r=0}^{\infty} p_r \\ &\le \sum_{r=0}^{\lceil 2\log_2 k\rceil - 2} 1 + k^2 \sum_{r=\lceil 2\log_2 k\rceil - 1}^{\infty} 2^{-r-1} \\ &= \lceil 2\log_2 k\rceil - 1 + k^2 2^{-\lceil 2\log_2 k\rceil + 1} \le \lceil 2\log_2 k\rceil + 1 \ll \log k\,. \end{aligned}$$

(Man bekommt auch die etwas bessere Schranke

$$E(k) \le \left\lceil \log_2 \binom{k}{2} \right\rceil + 2\,,$$

etwa $E(2) \le 2$, $E(3) \le 2 + \log_2 3 \approx 3{,}585$ usw., die aber asymptotisch keinen Gewinn bringt.)

Insgesamt sehen wir:

Der Erwartungswert für den Aufwand des obigen Algorithmus ist

$$\in \tilde{O}(nd \log q \log k) \in \tilde{O}(nd \log q) \quad \textit{Operationen in } \mathbb{F}_q.$$

Bisher haben wir immer noch vorausgesetzt, dass das zu faktorisierende Polynom quadratfrei ist. Eine einfache Möglichkeit, sich von dieser Einschränkung zu befreien, besteht darin, jeweils die Vielfachheit jedes gefundenen irreduziblen Faktors gleich festzustellen. Es gilt auch für nicht quadratfreies f ohne irreduzible Faktoren vom Grad $< d$, dass $\mathrm{ggT}(f, X^{q^d} - X)$ das Produkt der verschiedenen irreduziblen Faktoren vom Grad d ist, die f teilen. Per „Equal Degree Factorization“ können wir diese irreduziblen Faktoren finden und dann in der richtigen Vielfachheit abdividieren. Das ergibt folgenden Algorithmus.

```
function factor(f)
  // f ∈ F_q[X], q ungerade, lcf(f) = 1.
  // Ausgabe: ((h_1, e_1), ..., (h_k, e_k)) mit f = ∏_j h_j^{e_j},
  //    h_j normiert und irreduzibel, e_j ≥ 1.
  u ← () // leere Liste
  d ← 0
  a ← X // a ≡ X^{q^d} mod f
  while deg f > 0 do
    d ← d + 1
    if deg f < 2d then return append(u, (f, 1)) end if // f ist irreduzibel
    a ← a^q rem f // sukzessives Quadrieren in F_q[X]/fF_q[X]
    g ← gcd(f, a − X)
    if deg g > 0 then
      (h_1, ..., h_m) ← edf(g, d) // equal degree factorization
      f ← f/g // Abdividieren von h_1 ··· h_m
      for j = 1 to m do
```

```
            e ← 1
            while f rem h_j = 0 do
              e ← e + 1
              f ← f/h_j
            end while
            // Jetzt ist f nicht mehr durch h_j teilbar
            u ← append(u, (h_j, e))
          end for
        end if
      end while
      return u
    end function
```

Der Aufwand für die Division durch h_j fällt nicht ins Gewicht. Die Gesamtkomplexität für das Faktorisieren eines Polynoms vom Grad n über $\mathbb{F}_q$ ist damit im Erwartungswert von der Größenordnung $\tilde{O}(n^2 \log q)$ Operationen in $\mathbb{F}_q$ oder $\tilde{O}(n^2 (\log q)^2)$ Wortoperationen.

12.6. **Quadratfreie Faktorisierung.** Alternativ kann man zuerst f in quadratfreie Bestandteile zerlegen. Dabei bestimmt man paarweise teilerfremde quadratfreie Polynome $h_1, h_2, \ldots, h_m$, so dass $f = h_1 h_2^2 \cdots h_m^m$. Hat f diese Form, dann gilt in Charakteristik 0, dass

$$\mathrm{ggT}(f, f') = h_2 h_3^2 \cdots h_m^{m-1} \quad \text{und} \quad \frac{f}{\mathrm{ggT}(f, f')} = h_1 h_2 \cdots h_m \,;$$

damit lassen sich die h_j dann iterativ bestimmen:

$$f' = h_2 h_3^2 \cdots h_m^{m-1} (h_1' h_2 \cdots h_m + 2 h_1 h_2' h_3 \cdots h_m + \ldots + m h_1 \cdots h_{m_1} h_m') \,,$$

und der zweite Faktor, er sei mit h bezeichnet, ist wegen

$$\mathrm{ggT}(h, h_j) = \mathrm{ggT}(j h_1 \cdots h_{j-1} h_j' h_{j+1} \cdots h_m, h_j) = \mathrm{ggT}(j h_j', h_j) = \mathrm{ggT}(j, h_j)$$

(beachte $h_j \perp h_i$ für $i \neq j$ und $h_j \perp h_j'$) teilerfremd zu f.

In Charakteristik p gilt wegen $p = 0$ abweichend, dass

$$\mathrm{ggT}(f, f') = h_2 h_3^2 \cdots h_m^{m-1} \cdot \prod_{p \mid k} h_k \,,$$

also

$$\frac{f}{\mathrm{ggT}(f, f')} = \prod_{p \nmid k} h_k \,.$$

Man erreicht dann irgendwann den Punkt, dass $f' = 0$ ist. Das bedeutet $f = g(X^p)$ für ein Polynom g. Ist der Grundkörper endlich (oder wenigstens perfekt), dann folgt $g(X^p) = h(X)^p$, wobei die Koeffizienten von h die (nach Voraussetzung existierenden) p-ten Wurzeln der Koeffizienten von g sind. Man kann dann mit h weiter machen. Eine genaue Beschreibung findet sich in [GG].

12.7. **Effizientere Berechnung der q-ten Potenzen.** Während der Faktorisierung von f müssen wir die iterierten q-ten Potenzen von X im Restklassenring $\mathbb{F}_q[X]/f\mathbb{F}_q[X]$ berechnen. Bisher haben wir dafür die allgemein anwendbare Methode des sukzessiven Quadrierens benutzt. Wir können diese Berechnungen effizienter machen, wenn wir die speziellen Eigenschaften endlicher Körper ausnutzen.

Dafür erinnern wir uns daran, dass die Abbildung $x \mapsto x^q$ einen Endomorphismus auf jeder $\mathbb{F}_q$-Algebra definiert. Insbesondere ist

$$\phi : \mathbb{F}_q[X]/f\mathbb{F}_q[X] \longrightarrow \mathbb{F}_q[X]/f\mathbb{F}_q[X]\,, \qquad a \longmapsto a^q$$

eine $\mathbb{F}_q$-lineare Abbildung. Wenn wir eine $\mathbb{F}_q$-Basis von $\mathbb{F}_q[X]/f\mathbb{F}_q[X]$ wählen, etwa (die Bilder von) $1, X, X^2, \ldots, X^{n-1}$ (mit $n = \deg f$), dann können wir ϕ durch eine $n \times n$-Matrix M über $\mathbb{F}_q$ darstellen. Die Zuweisung $a \leftarrow a^q \bmod f$ in den obigen Algorithmen kann dann ersetzt werden durch $a \leftarrow M \cdot a$ (wenn wir a mit seinem Koeffizientenvektor identifizieren). Die Kosten dafür betragen $\ll n^2$ Operationen in $\mathbb{F}_q$. Das ist zu vergleichen mit den Kosten von $\tilde{O}(n \log q)$ Operationen in $\mathbb{F}_q$ für das sukzessive Quadrieren. Es wird sich also vor allem dann lohnen, die Variante mit der Matrix M zu benutzen, wenn n gegenüber $\log q$ nicht zu groß ist.

Es geht aber noch etwas besser. Wenn $g \in \mathbb{F}_q[X]$ ein Polynom ist, dann gilt $g(X)^q = g(X^q)$. Das führt zu folgendem Algorithmus zur Berechnung der Potenzen $X^{q^j} \bmod f$.

```
function itfrob(f, m)
  // f ∈ F_q[X], m ≥ 0.
  // Ausgabe: (X^q rem f, X^{q^2} rem f, ..., X^{q^d} rem f) mit d ≥ m.
  u ← (X^q rem f) // Liste für das Ergebnis
  // X^q rem f durch sukzessives Quadrieren
  while length(u) < m do
    h ← last(u) ∈ F_q[X] // letzter Eintrag in u
    u ← u cat lift(evalmult(h, u mod f))
  end while
  return u
end function
```

evalmult$(h, u \bmod f)$ ist hier die schnelle Auswertung von h in mehreren Punkten wie in 11.1; dabei sei $(u_1, \ldots, u_k) \bmod f = (u_1 \bmod f, \ldots, u_k \bmod f)$ ein Tupel von Elementen von $\mathbb{F}_q[X]/f\mathbb{F}_q[X]$. lift wandelt die Liste von Elementen des Restklassenrings wieder in eine Liste von Polynomen um. Im Computer passiert dabei nichts, nur die Interpretation ändert sich.

Der Algorithmus ist korrekt: Sei zu Beginn eines Durchlaufs durch die while-Schleife

$$u = \left(X^q \text{ rem } f, X^{q^2} \text{ rem } f, \ldots, X^{q^{2^k}} \text{ rem } f\right).$$

Dann wird $h = X^{q^{2^k}} \text{ rem } f$, und wir werten h aus an den Stellen

$$X^q \bmod f, X^{q^2} \bmod f, \ldots, X^{q^{2^k}} \bmod f \in \mathbb{F}_q[X]/f\mathbb{F}_q[X]\,.$$

Wegen

$$h(X^{q^l}) \bmod f = h(X)^{q^l} \bmod f = X^{q^{2^k+l}} \bmod f$$

ergibt evalmult dann die Liste

$$\left(X^{q^{2^k+1}} \bmod f, X^{q^{2^k+2}} \bmod f, \ldots, X^{q^{2^{k+1}}} \bmod f\right),$$

und lift wandelt dies in die kanonischen Repräsentanten der Restklassen um. Am Ende der Schleife hat u wieder die Form wie zu Beginn, mit einem um 1 erhöhten Wert von k.

Sei wie üblich $n = \deg f$. Der Aufwand beträgt $\tilde{O}(n \log q)$ Operationen in $\mathbb{F}_q$ für die Berechnung von $X^q \text{ rem } f$ durch sukzessives Quadrieren. Die schnelle Auswertung eines Polynoms vom Grad l in k Punkten hat Komplexität $\tilde{O}(l+k)$ Operationen im

Restklassenring, also $\tilde{O}((l+k)n)$ Operationen in $\mathbb{F}_q$. In unserem Fall ist $l \leq n-1$, und k ist der Reihe nach $1, 2, 4, \ldots, 2^s$ mit $s = \lceil \log_2 m \rceil - 1$. Die Summe dieser Werte von k ist $2^{s+1} - 1 < 2m$, die Anzahl ist $s + 1 \ll \log m$. Insgesamt ist der Aufwand für die while-Schleife also beschränkt durch $\tilde{O}((n+m)n)$ Operationen in $\mathbb{F}_q$. Gilt (wie meist in den Anwendungen) $m \ll n$, dann reduziert sich das auf $\tilde{O}(n^2)$, und der Gesamtaufwand ist

$$\tilde{O}\big(n(n + \log q)\big) \quad \text{Operationen in } \mathbb{F}_q$$

oder

$$\tilde{O}\big(n^2 \log q + n(\log q)^2\big) \quad \text{Wortoperationen.}$$

Die Komplexität der „Distinct Degree Factorization“ reduziert sich dann ebenfalls auf diese Größenordnung (gegenüber vorher $n^2(\log q)^2$).

Die Berechnung der $(q^d - 1)/2$-ten Potenz in der „Equal Degree Factorization“ lässt sich beschleunigen, indem man

$$\frac{q^d - 1}{2} = \big(1 + q + q^2 + \cdots + q^{d-1}\big)\,\frac{q-1}{2}$$

verwendet. Wenn man die Werte von X^{q^j} rem f für $j < d$ schon berechnet hat, dann kann man $a, a^q, \ldots, a^{q^{d-1}}$ im Restklassenring durch Auswerten in

$$X \bmod f, X^q \bmod f, \ldots, X^{q^{d-1}} \bmod f$$

bestimmen (ähnlich wie im Algorithmus itfrob oben), dann das Produkt bilden und schließlich noch davon die $(q-1)/2$-te Potenz durch sukzessives Quadrieren berechnen. Die Komplexität reduziert sich analog. Insgesamt erhält man das folgende Resultat.

12.8. **Satz.** *Die vollständige Faktorisierung eines normierten Polynoms $f \in \mathbb{F}_q[X]$ vom Grad n kann mit einem Aufwand von*

$$\tilde{O}\big(n^2 \log q + n(\log q)^2\big) \quad \textit{Wortoperationen}$$

bestimmt werden.

13. Faktorisierung von primitiven Polynomen über $\mathbb{Z}$

Jetzt wollen wir uns der Faktorisierung von Polynomen in $\mathbb{Z}[X]$ zuwenden. Wir können erst einmal jedes (von null verschiedene) Polynom $f \in \mathbb{Z}[X]$ faktorisieren als

$$f = \operatorname{cont}(f)\operatorname{pp}(f)$$

mit $\operatorname{cont}(f) \in \mathbb{Z}$ und einem *primitiven* Polynom $\operatorname{pp}(f) \in \mathbb{Z}[X]$. Die vollständige Faktorisierung von f ergibt sich dann aus der Faktorisierung von $\operatorname{cont}(f)$ (in Primzahlen) und der von $\operatorname{pp}(f)$; dabei sind die Faktoren von $\operatorname{pp}(f)$ wieder primitiv. Aus dem Lemma von Gauß folgt, dass die Faktorisierung von primitiven Polynomen in $\mathbb{Z}[X]$ äquivalent ist zur Faktorisierung von (oBdA normierten) Polynomen in $\mathbb{Q}[X]$.

Die Faktorisierung von ganzen Zahlen ist eine eigene Geschichte; deshalb setzen wir hier voraus, dass das zu faktorisierende Polynom f primitiv ist. Mit dem Algorithmus aus Abschnitt 12.6 können wir f in ein Produkt von Potenzen quadratfreier Polynome zerlegen. Also können wir auch voraussetzen, dass f quadratfrei ist. Die Aufgabe lautet also, ein gegebenes primitives und quadratfreies Polynom zu faktorisieren.

13.1. **Große Primzahl.** Eine Möglichkeit besteht darin, f modulo einer geeigneten Primzahl p zu reduzieren, und die Faktorisierung von f aus der von $f \bmod p$ zu rekonstruieren. Wir werden diesen Ansatz im Folgenden genauer untersuchen.

Für unsere Zwecke ist es von Vorteil, wenn $f \bmod p$ auch quadratfrei ist (und den gleichen Grad n wie f hat). Ein Polynom ist genau dann quadratfrei, wenn es keine gemeinsame Nullstelle mit seiner Ableitung hat. Nach der Theorie der Resultante ist das äquivalent zu

$$\operatorname{disc} f := \operatorname{Res}(f, f') \neq 0\,.$$

Daraus (und aus $\operatorname{Res}(\bar f, \bar f') = \overline{\operatorname{Res}(f, f')}$, wo $\bar f = f \bmod p$) folgt, dass p genau dann die gewünschte Eigenschaft hat, wenn

$$p \nmid \operatorname{lcf}(f) \quad \text{und} \quad p \nmid \operatorname{disc} f\,.$$

Da $\operatorname{disc} f \neq 0$ nach Voraussetzung, gibt es nur endlich viele „schlechte“ Primzahlen. $\operatorname{disc} f$ heißt die *Diskriminante* von f (und wird häufig auch als $\operatorname{disc} f = \operatorname{Res}(f, f')/\operatorname{lcf}(f)$ definiert, was für uns hier aber keinen Unterschied macht).

Außerdem müssen wir in der Lage sein, die Faktoren von f in $\mathbb{Z}[X]$ aus geeigneten Teilern von $\bar f$ in $\mathbb{F}_p[X]$ zu rekonstruieren. Dafür ist nötig, dass p größer ist als das Doppelte des Absolutbetrags jedes Koeffizienten eines Teilers von f. Nach Satz 10.23 ist

$$B = 2^n \sqrt{n+1}\, \|f\|_\infty$$

eine obere Schranke für die Beträge der Koeffizienten; wir sollten also $p > 2B$ wählen (aber nicht viel größer, damit die Rechnung nicht ineffizient wird).

Wir wählen also eine zufällige Primzahl $2B < p < 4B$, setzen $\bar f = f \bmod p$ und testen, ob $\operatorname{ggT}(\bar f, \bar f') = 1$ ist (p kann den Leitkoeffizienten von f nicht teilen, da $|\operatorname{lcf}(f)| \leq \|f\|_\infty < B$ ist). Ist das nicht der Fall, dann ist $\bar f$ nicht quadratfrei, und wir wählen eine neue Primzahl p.

Dann faktorisieren wir $\bar f/\operatorname{lcf}(\bar f)$. Wir bekommen eine Anzahl von (verschiedenen) normierten irreduziblen Faktoren $h_1, \dots, h_m$. Der Einfachheit halber nehmen wir für den Moment einmal an, dass $\operatorname{lcf}(f) = 1$ ist. Dann sind die irreduziblen Faktoren von f ebenfalls normiert, und für jeden solchen Faktor g gilt

$$g \bmod p = \prod_{j \in J_g} h_j$$

für eine geeignete Teilmenge $J_g \subset \{1, 2, \dots, m\}$. Wir berechnen also

$$g_J = \operatorname{lift}\Big(\prod_{j \in J} h_j\Big)$$

für Teilmengen J von $\{1, \dots, m\}$ und testen, ob g_J ein Teiler von f ist. (Dabei sei $\operatorname{lift}(h)$ wie früher das Polynom H mit $\|H\|_\infty < p/2$ und $H \bmod p = h$.) Am einfachsten geht das, indem wir auch $g_{J'}$ berechnen mit $J' = \{1, \dots, m\} \setminus J$ und testen, ob $g_J g_{J'} = f$ ist. Dazu reicht es aus, die Ungleichung

$$\|g_J\|_1 \, \|g_{J'}\|_1 \leq B$$

nachzuprüfen, denn nach Satz 10.23 muss das erfüllt sein, wenn $g_J g_{J'} = f$ ist; umgekehrt ist

$$\|g_J g_{J'}\|_\infty \leq \|g_J g_{J'}\|_1 \leq \|g_J\|_1 \, \|g_{J'}\|_1 \leq B$$

und $g_J g_{J'} \equiv f \bmod p$, also folgt Gleichheit.

Ist f nicht normiert und $\operatorname{lcf}(f) = l$, dann setzen wir statt dessen

$$g_J = \operatorname{lift}\bigl(l \prod_{j \in J} h_j\bigr)\,;$$

in diesem Fall gilt

$$lf \equiv g_J g_{J'} \bmod p\,.$$

Alles geht genauso durch wie vorher, wenn wir B durch $|l| \cdot B$ ersetzen.

Wir erhalten also folgenden Algorithmus.

```
function factor(f)
  // f ∈ ℤ[X] primitiv und quadratfrei, lcf(f) > 0.
  // Ausgabe: (g_1, ..., g_k) mit g_j ∈ ℤ[X] primitiv und irreduzibel, f = g_1 ··· g_k.
  n ← deg f
  if n = 0 then return () end if // f = 1
  if n = 1 then return (f) end if // noch ein trivialer Fall
  l ← lcf(f)
  B ← √(n+1) 2^n l ‖f‖_∞ ∈ ℝ
  // Wahl einer geeigneten Prinzahl
  repeat
    p ← zufällige Primzahl mit 2B < p < 4B
    f̄ ← f mod p ∈ 𝔽_p[X]; l̄ ← l mod p
  until gcd(f̄, f̄') = 1
  (h_1, ..., h_m) ← factor(f̄/l̄)
  return combine(f, B, p, (h_1, ..., h_m), 1)
end function

function combine(f, B, p, (h_1, ..., h_m), d)
  // f, B, p, (h_1, ..., h_m) wie oben,
  //    jeder irreduzible Faktor von f zerlegt sich mod p in ≥ d Faktoren.
  // Ausgabe: wie oben.
  if 2d > m then return (f) end if // dann muss f irreduzibel sein
  l̄ ← lcf(f) mod p ∈ 𝔽_p
  for J ⊂ {1, 2, ..., m}, #J = d do
    g_1 ← 1 ∈ 𝔽_p[X]; g_2 ← 1 ∈ 𝔽_p[X]
    for j = 1 to m do
      if j ∈ J then g_1 ← g_1 · h_j else g_2 ← g_2 · h_j end if
    end for
    G_1 ← lift(l̄ g_1); G_2 ← lift(l̄ g_2)
    if ‖G_1‖_1 · ‖G_2‖_1 ≤ B then
      // Faktor gefunden
      return (pp(G_1)) cat combine(pp(G_2), B, p, (h_j)_{j∉J}, d)
    end if
  end for
  return combine(f, B, p, (h_1, ..., h_m), d + 1)
end function
```

Zur Komplexität: Ist $\|f\|_\infty \le A$, dann gilt $\log B \ll n + \log A$. Eine zufällige (Pseudo-)Primzahl im relevanten Intervall lässt sich durch einen probabilistischen Algorithmus in erwarteten $\tilde{O}\bigl((\log B)^3\bigr)$ Wortoperationen finden. Eine Abschätzung der Diskriminante zeigt, dass die Wahrscheinlichkeit, dass p „gut“

ist, mindestens $1/2$ beträgt (Übung). Die erwartete Anzahl an Versuchen, bis ein gutes p gefunden ist, ist also höchstens 2. (Die Berechnung des ggT in $\mathbb{F}_p[X]$ ist mit $\tilde{O}(n \log B)$ Wortoperationen hier vernachlässigbar.)

Der Aufwand für die Faktorisierung mod p ist $\tilde{O}\big(n(\log B)^2\big)$.

Für eine Teilmenge J können die Multiplikationen der h_j mit einem Aufwand von $\tilde{O}(mn \log B)$ Wortoperationen durchgeführt werden (mit einem hierarchischen Ansatz wie in Abschnitt 11 kann man das zu $\tilde{O}(n \log B)$ beschleunigen); die weiteren Berechnungen sind vernachlässigbar. Im schlimmsten Fall ist f irreduzibel, dann müssen alle Teilmengen J mit $\#J \le m/2$ durchprobiert werden. Davon gibt es mindestens 2^{m-1}, und m kann im ungünstigsten Fall gleich n sein.

Das Verfahren zum Kombinieren der Faktoren hat also bei ungünstigen Eingangsdaten *exponentielle* Komplexität.

Für ein gegebenes quadratfreies Polynom $f \in \mathbb{Z}[X]$ gilt stets, dass es unendlich viele verschiedene Primzahlen p gibt, so dass $f \bmod p$ in Linearfaktoren zerfällt. Das ist eine Folgerung aus dem Satz von Chebotarev; genauer ergibt sich, dass die Menge dieser Primzahlen die Dichte $1/[K : \mathbb{Q}]$ hat, wo K der Zerfällungskörper von f ist. Man kann also durch eine unglückliche Wahl der Primzahl p im Algorithmus stets den Fall exponentieller Komplexität bekommen.

Es ergibt sich dann die Frage, ob man wenigstens bei passender Wahl von p erreichen kann, dass man polynomiale Komplexität bekommt. Diese Hoffnung erfüllt sich leider nicht: Es gibt Polynome, die sich *immer* schlecht verhalten.

13.2. **Swinnerton-Dyer-Polynome.** Seien $p_1, p_2, \dots, p_m$ paarweise verschiedene Primzahlen (es genügt anzunehmen, dass kein Produkt von verschiedenen der p_j ein Quadrat ist). Wir betrachten das Polynom

$$\mathrm{SD}_{p_1,\dots,p_m}(X) = \prod_{\varepsilon_1,\dots,\varepsilon_m \in \{\pm 1\}} \Big(X - \sum_{j=1}^{m} \varepsilon_j \sqrt{p_j}\Big)\,.$$

Zum Beispiel ist

$$\begin{aligned}
\mathrm{SD}_{2,3}(X) &= (X - \sqrt{2} - \sqrt{3})(X - \sqrt{2} + \sqrt{3})(X + \sqrt{2} - \sqrt{3})(X + \sqrt{2} + \sqrt{3}) \\
&= \big((X - \sqrt{2})^2 - 3\big)\big((X + \sqrt{2})^2 - 3\big) \\
&= (X^2 - 2\sqrt{2}X - 1)(X^2 + 2\sqrt{2}X - 1) \\
&= (X^2 - 1)^2 - 8X^2 = X^4 - 10X^2 + 1\,.
\end{aligned}$$

Allgemeiner gilt

$$\mathrm{SD}_{p_1,\dots,p_m,p_{m+1}}(X) = \mathrm{SD}_{p_1,\dots,p_m}(X - \sqrt{p_{m+1}})\,\mathrm{SD}_{p_1,\dots,p_m}(X + \sqrt{p_{m+1}})\,.$$

Also etwa

$$\begin{aligned}
\mathrm{SD}_{2,3,5}(X) &= \big((X - \sqrt{5})^4 - 10(X - \sqrt{5})^2 + 1\big)\big((X + \sqrt{5})^4 - 10(X + \sqrt{5})^2 + 1\big) \\
&= (X^4 + 20X^2 - 24)^2 - 5(4X^3)^2 \\
&= X^8 - 40X^6 + 352X^4 - 960X^2 + 576\,.
\end{aligned}$$

(Die MAGMA-Funktion `SwinnertonDyerPolynomial`(n) berechnet $\mathrm{SD}_{2,3,5,\dots,p_n}$, wo p_n die n-te Primzahl ist.)

Für diese Polynome gilt:

(1) $\mathrm{SD}_{p_1,\dots,p_m} \in \mathbb{Z}[X]$;

(2) $\mathrm{SD}_{p_1,\dots,p_m}$ ist irreduzibel;

(3) Für jede Primzahl p zerfällt $\mathrm{SD}_{p_1,\dots,p_m} \bmod p$ in $\mathbb{F}_p[X]$ in Faktoren vom Grad höchstens 2.

Sei dazu $K = \mathbb{Q}(\sqrt{p_1}, \sqrt{p_2}, \dots, \sqrt{p_m})$. Dann ist $K/\mathbb{Q}$ eine Galoiserweiterung, und die Elemente der Galoisgruppe bilden $\sqrt{p_j}$ auf $\pm\sqrt{p_j}$ ab. Die Nullstellen des Polynoms werden also nur vertauscht; es folgt $\mathrm{SD}_{p_1,\dots,p_m} \in \mathbb{Q}[X]$. Auf der anderen Seite sind die Nullstellen alle ganz-algebraisch, also folgt, dass die Koeffizienten sogar ganzzahlig sind.

Für die Irreduzibilität überlegt man sich, dass $[K : \mathbb{Q}] = 2^m$ ist (denn p_m ist kein Quadrat in $\mathbb{Q}(\sqrt{p_1}, \dots, \sqrt{p_{m-1}})$). Es folgt, dass es zu jeder Wahl von Vorzeichen ε_j ein Element σ der Galoisgruppe von $K/\mathbb{Q}$ gibt mit $\sigma(\sqrt{p_j}) = \varepsilon_j\sqrt{p_j}$ für alle j. Die Galoisgruppe operiert dann transitiv auf den Nullstellen; damit ist $\mathrm{SD}_{p_1,\dots,p_m}$ das Minimalpolynom etwa von $\sum_j \sqrt{p_j}$ und insbesondere irreduzibel.

Sei nun p irgend eine Primzahl. Dann wird jedes Element von $\mathbb{F}_p$ in $\mathbb{F}_{p^2}$ ein Quadrat (klar für $\mathbb{F}_2$, und für p ungerade ist $\mathbb{F}_{p^2} = \mathbb{F}_p(\sqrt{d})$ für jedes Nichtquadrat $d \in \mathbb{F}_p$). Es folgt, dass $\mathrm{SD}_{p_1,\dots,p_m} \bmod p$ in $\mathbb{F}_{p^2}[X]$ in Linearfaktoren zerfällt. Der Frobenius-Automorphismus von $\mathbb{F}_{p^2}/\mathbb{F}_p$ kann einige davon paarweise vertauschen; das Produkt zweier solcher Faktoren ist dann in $\mathbb{F}_p[X]$. Faktoren, die vom Frobenius festgelassen werden, sind bereits in $\mathbb{F}_p[X]$. Damit kann $\mathrm{SD}_{p_1,\dots,p_m} \bmod p$ in $\mathbb{F}_p[X]$ höchstens irreduzible Faktoren vom Grad 2 haben.

Ist also $f = \mathrm{SD}_{p_1,\dots,p_m}$ (mit $n = 2^m$), dann haben wir mod p (wenn $p \nmid \mathrm{disc}(f)$) mindestens $2^{m-1} = n/2$ Faktoren, und der Algorithmus muss alle Teilmengen der Größe $\leq n/4$ durchprobieren, bis feststeht, dass f irreduzibel ist. Das sind mehr als $2^{n/2-1}$, also haben wir hier in jedem Fall exponentielle Komplexität.

13.3. **Verhalten für zufällige Polynome.** Ist $f \in \mathbb{Z}[X]$ ein zufällig gewähltes (quadratfreies, primitives) Polynom vom Grad n, etwa mit Koeffizienten vom Betrag $\leq A$ mit nicht zu kleinem A, dann ist f mit sehr hoher Wahrscheinlichkeit irreduzibel und hat Galoisgruppe S_n. In diesem Fall ist der Erwartungswert der Anzahl der irreduziblen Faktoren von f mod p gleich der durchschnittlichen Anzahl von Zykeln einer Permutation in S_n. Man kann sich überlegen (Übung), dass dieser Erwartungswert $\in \log n + O(1)$ ist. (Die Standardabweichung ist etwa $\sqrt{\log n}$, also relativ klein.) Die Anzahl der zu betrachtenden Teilmengen von Faktoren ist dann etwa $\frac{1}{2}2^{\log n+O(1)} \ll n^{\log 2}$. Damit ist die erwartete Laufzeit doch wieder polynomial.

Auf der anderen Seite sind viele der Polynome, die man faktorisieren möchte, gerade *nicht* zufällig. Im allgemeinen wird sich der Algorithmus umso schlechter verhalten, je mehr Elemente der Galoisgruppe als Permutationen der Nullstellen in viele Zykel zerfallen. Das bedeutet im wesentlichen, dass die Galoisgruppe einen kleinen Exponenten hat. Bei den Swinnerton-Dyer-Polynomen ist der Exponent zum Beispiel 2.

13.4. **Kleine Primzahlen und Hensel-Lift.** In jedem Fall erscheint es günstig, wenn man mehrere Primzahlen ausprobieren kann, um eine zu finden, die möglichst wenige Faktoren liefert. Allerdings ist in unserem bisherigen Algorithmus der Aufwand für das Finden einer geeigneten Primzahl sehr groß. Wir würden daher gerne mit kleinen Primzahlen arbeiten. Eine Möglichkeit besteht darin, mit einer Faktorisierung mod p zu beginnen und sie zu einer Faktorisierung mod p^e „hochzuheben“, wo $p^e > 2B$ ist. Die Bedingung $p^e > 2B$ garantiert wieder, dass wir die

Faktoren in $\mathbb{Z}[X]$ rekonstruieren können. Das folgende Lemma zeigt, dass dieser Ansatz funktioniert.

13.5. **Lemma (Hensel).** *Sei $f \in \mathbb{Z}[X]$ und p eine Primzahl mit $p \nmid \operatorname{lcf}(f) = l$. Seien weiter $\bar{g}, \bar{h} \in \mathbb{F}_p[X]$ mit $\operatorname{lcf}(\bar{g}) = \operatorname{lcf}(\bar{h}) = 1$, $\operatorname{Res}(\bar{g}, \bar{h}) \neq 0$ und $(l \bmod p)\bar{g}\bar{h} = f \bmod p$. Dann gibt es für jedes $e \geq 1$ eindeutig bestimmte Polynome $g_e, h_e \in \mathbb{Z}/p^e\mathbb{Z}[X]$ mit $\operatorname{lcf}(g_e) = \operatorname{lcf}(h_e) = 1$, $g_e \bmod p = \bar{g}$, $h_e \bmod p = \bar{h}$ und $(l \bmod p^e)g_e h_e = f \bmod p^e$.*

Beweis. Induktion nach e. Für $e = 1$ ist die Behauptung trivial. Wir nehmen an, die Aussage sei wahr für ein e und zeigen sie für $e + 1$.

Da g_e, h_e nach Annahme eindeutig bestimmt sind, muss gelten

$$g_{e+1} \bmod p^e = g_e \quad \text{und} \quad h_{e+1} \bmod p^e = h_e\,.$$

Seien $\tilde{g}_e, \tilde{h}_e \in \mathbb{Z}/p^{e+1}\mathbb{Z}[X]$ mit $\operatorname{lcf}(\tilde{g}_e) = \operatorname{lcf}(\tilde{h}_e) = 1$ und so dass $\tilde{g}_e \bmod p^e = g_e$, $\tilde{h}_e \bmod p^e = h_e$. (Die Leitkoeffizienten sind eindeutig bestimmt.) Wir können dann ansetzen:

$$g_{e+1} = \tilde{g}_e + p^e u\,, \quad h_{e+1} = \tilde{h}_e + p^e v$$

mit $u, v \in \mathbb{F}_p[X]$, $\deg u < \deg \bar{g}$ und $\deg v < \deg \bar{h}$. Nach Voraussetzung gilt

$$(l \bmod p^{e+1})\tilde{g}_e\tilde{h}_e = (f \bmod p^{e+1}) + p^e w$$

mit $w \in \mathbb{F}_p[X]$, $\deg w < \deg f = \deg \bar{g} + \deg \bar{h}$. Die Bedingung

$$(l \bmod p^{e+1})g_{e+1}h_{e+1} = f \bmod p^{e+1}$$

bedeutet dann

$$p^e(w + ul\tilde{h}_e + vl\tilde{g}_e) \equiv 0 \bmod p^{e+1}$$

oder äquivalent

$$w + u(l \bmod p)\bar{h} + v(l \bmod p)\bar{g} = 0 \in \mathbb{F}_p[X]\,.$$

Da $\operatorname{Res}(\bar{g}, \bar{h}) \neq 0$ die Determinante der Koeffizientenmatrix dieses linearen Gleichungssystems ist, gibt es eine eindeutige Lösung $u \in \mathbb{F}_p[X]_{<\deg \bar{g}}$, $v \in \mathbb{F}_p[X]_{<\deg \bar{h}}$. Das zeigt die Aussage für $e + 1$. □

Die Berechnung der Faktoren g_e und h_e kann also wie folgt geschehen.

```
function hensel(f, p, ḡ, h̄, e)
  // f, p, ḡ, h̄ wie oben, e ≥ 1.
  // Ausgabe: g_e, h_e wie oben.
  g_1 ← ḡ; h_1 ← h̄
  l ← lcf(f); l̄ ← l mod p
  for j = 1 to e − 1 do
    w ← (l lift(g_j) lift(h_j) − f)/p^j mod p
    (u, v) ← Lösung von u l̄h̄ + v l̄ḡ = −w
    g_{j+1} ← lift(g_j) + p^j u
    h_{j+1} ← lift(h_j) + p^j v
  end for
  return (g_e, h_e).
end function
```

Da man mehrmals ein lineares Gleichungssystem mit der selben Koeffizientenmatrix, aber wechselnder rechter Seite lösen muss, ist es sinnvoll, die Inverse dieser Matrix zu berechnen (Komplexität $O(n^3)$ Operationen in $\mathbb{F}_p$, wenn $n = \deg f$).

Dann reduziert sich der Aufwand für die Lösung des Gleichungssystems auf $O(n^2)$ Operationen in $\mathbb{F}_p$ pro Durchlauf.

Literatur

[GG] Joachim von zur Gathen und Jürgen Gerhard: *Modern computer algebra,* 2nd edition, Cambridge University Press, 2003.

[Ka] Michael Kaplan: *Computeralgebra,* Springer-Verlag Berlin Heidelberg, 2005
Online: http://dx.doi.org/10.1007/b137968

[Ko] Wolfram Koepf: *Computeralgebra, eine algorithmisch orientierte Einführung,* Springer-Verlag Berlin Heidelberg, 2006
Online: http://dx.doi.org/10.1007/3-540-29895-9