Einführung in die Computeralgebra

Sommersemester 2018

Universität Bayreuth

Michael Stoll

Inhaltsverzeichnis

Bildschirmversion vom 30. Juli 2018, 20:33 Uhr.

1. Einführung

Was ist Computeralgebra?

Die Computeralgebra ist Teil eines Gebiets, das als *Wissenschaftliches Rechnen* bezeichnet wird. 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 in etwa die folgenden.

Bei den numerischen Verfahren sind die zugrunde 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 zum Beispiel 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.

Es gibt allerdings auch Mischformen. 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.

BSP
symbolisch/
numerisch

1.1. **Beispiel.** Wir betrachten folgendes Problem:

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

sei ein Polynom mit ganzzahligen Koeffizienten. Wir bezeichnen die Nullstellen von f in $\mathbb{C}$ mit $\alpha_1, \alpha_2, \alpha_3, \alpha_4$. Wir wollen das Polynom $g(X)$ berechnen, dessen Nullstellen die sechs verschiedenen möglichen Ausdrücke der Form $\alpha_i\alpha_j$ sind mit $1 \le i < j \le 4$. Die Theorie der symmetrischen Polynome sagt uns, dass die Koeffizienten von g Polynome in den a_j mit ganzzahligen Koeffizienten sind. Explizit gilt (wie durch eine symbolische Rechnung nachgewiesen werden kann)

$$g(X) = X^6 - a_2X^5 + (a_1a_3 - a_0)X^4 + (2a_0a_2 - a_0a_3^2 - a_1^2)X^3 + a_0(a_1a_3 - a_0)X^2 - a_0^2a_2X + a_0^3\,.$$

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 $\beta_1, \ldots, \beta_6$ von g bestimmen und dann $g(X)$ näherungsweise als

$$g(X) = (X - \beta_1)(X - \beta_2) \cdots (X - \beta_6)$$

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 diesem Beispiel ist das Einsetzen der Koeffizienten in die explizite Formel für g nicht sehr aufwendig. Ein ähnlich gelagerter Fall tritt auf für f vom Grad 6 und g das Polynom mit den zehn Nullstellen der Form $\alpha_{i_1}\alpha_{i_2}\alpha_{i_3} + \alpha_{i_4}\alpha_{i_5}\alpha_{i_6}$ mit $\{i_1, \ldots, i_6\} = \{1, \ldots, 6\}$, das zu bestimmen für einen Algorithmus aus meinem Forschungsgebiet wichtig ist. Hier hat der explizite Ausdruck für g über 160 Terme, und der Ansatz über eine numerische Näherung gewinnt an Charme. (Man kann übrigens auch mit anderen Vervollständigungen von $\mathbb{Q}$ arbeiten als mit $\mathbb{R}$; diese „p-adischen Zahlen“ (für eine Primzahl p) haben beim numerischen Rechnen gewisse Vorteile.) ♣

Für diese Vorlesung werden Grundkenntnisse in Algebra (Theorie des euklidischen Rings $\mathbb{Z}$, Polynomringe, Chinesischer Restsatz) vorausgesetzt, wie sie in der Vorlesung „Einführung in die Zahlentheorie und algebraische Strukturen“ vermittelt werden. Falls Sie diese Vorlesung (noch) nicht gehört haben, sollten Sie sich die benötigten Kenntnisse in geeigneter Weise aneignen, zum Beispiel indem Sie das Skript zur Vorlesung zu Rate ziehen.

Als Literatur zum Thema ist das sehr schöne Buch [GG] von von zur Gathen und Gerhard zu empfehlen. 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 V2.21-11    Thu Apr 14 2016 11:05:11 on btm2xa  [Seed = 595121159]
Type ? for help.  Type <Ctrl>-D to quit.
> IsSquare(2);
false
> Type(2);
RngIntElt
> IsSquare(RealField()!2);
true 1.41421356237309504880168872421
> Type(RealField()!2);
FldReElt
> 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 in der Computeralgebra häufig mit sehr großen ganzen Zahlen zu tun haben, reicht ein Wort im Allgemeinen nicht aus, um eine ganze Zahl zu speichern. Man wird also ein Array (eine geordnete Liste) 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

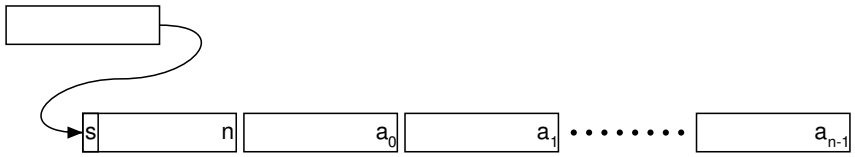
$$s \cdot 2^{63} + n, a_0, \ldots, a_{n-1}$$

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 $N \neq 0$

DEF
Wortlänge

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

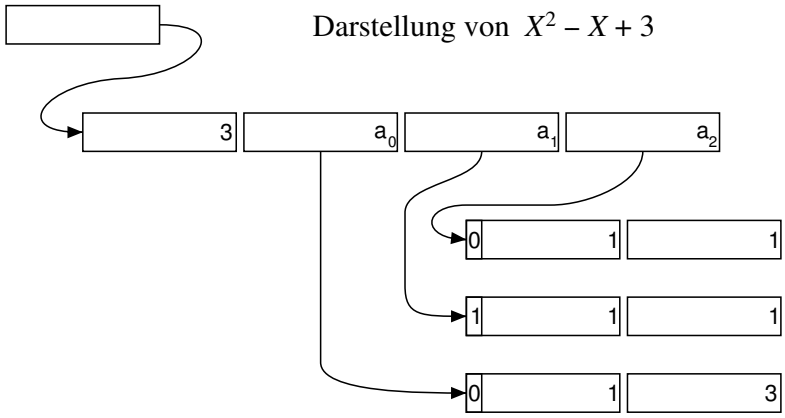
Man verwendet einen *Zeiger* auf das erste Wort der Darstellung von N (also seine Adresse im Speicher), um N im Computer zu repräsentieren:



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. In der folgenden Skizze ist das am Beispiel $X^2 - X + 3$ gezeigt.



Rationale Zahlen werden als Paare von ganzen Zahlen (Zähler und Nenner), d.h. konkret, als zwei Zeiger auf Zähler und Nenner in aufeinanderfolgenden Speicherzellen, dargestellt. Polynome in mehreren Variablen können zum Beispiel als Polynome dargestellt werden, deren Koeffizienten wieder Polynome (in einer Variablen weniger) sind, usw.

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 normalisierten 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 denselben Grad n haben und ihre Koeffizienten a_j jeweils gleich sind (wobei hier der Algorithmus zum Test der Gleichheit im Koeffizientenring R verwendet wird).

Algorithmen in diesem Skript werden in *Pseudocode* formuliert. Er lehnt sich an die Strukturen der meisten Programmiersprachen an und kann in der Regel ohne Schwierigkeiten in eine gegebene Sprache (wie zum Beispiel die Programmiersprache von Magma) übertragen werden. Je nach Abstraktionsgrad können aber auch Anweisungen vorkommen, die keine unmittelbare Entsprechung haben. Die Einrückung dient der Übersichtlichkeit, hat aber sonst keine Bedeutung.

Hier ist eine recht ausführliche Version des Vergleichsalgorithmus für ganze Zahlen, basierend auf der oben eingeführten Datenstruktur. Wir nehmen an, dass der Prozessor Befehle zur Verfügung stellt, mit denen zwei Datenworte (als Elemente von $\mathbb{Z}_{\geq 0}$ interpretiert) verglichen werden können. In einer konkreten Implementation müssen wir auch in der Lage sein, das Vorzeichenbit s bzw. t zu extrahieren.

ALGO
Vergleich in $\mathbb{Z}$

function compare(M, N)
input: $M = (-1)^s \sum_{j=0}^{m-1} a_j B^j$, $N = (-1)^t \sum_{j=0}^{n-1} b_j B^j$
output: -1 falls $M < N$, 0 falls $M = N$, 1 falls $M > N$.

```
if s ≠ t then return (−1)^s end if
// Ab hier haben M und N dasselbe 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

$$w + c' \cdot B = u + v + c$$

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

$$w - c' \cdot B = u - v - c\,.$$

Wir werden im Pseudocode „←“ für den Zuweisungsoperator anstelle des in vielen Programmiersprachen gebräuchlichen „:=“ (oder auch „=“) verwenden.

ALGO
Addition in $\mathbb{Z}$

function add(M, N)
input: $M = (-1)^s \sum_{j=0}^{m-1} a_j B^j$, $N = (-1)^t \sum_{j=0}^{n-1} b_j B^j$.
output: $M + N = (-1)^u \sum_{j=0}^{k-1} d_j B^j$.

```
  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 ≥ n.
    k ← m // Länge der Summe.
    u ← 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
```

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.

Wir präzisieren das und führen noch verwandte Schreibweisen ein.

* 2.1. **Definition.** Sind $f,g\colon \mathbb{Z}_{\ge 0}\to\mathbb{R}$, dann schreiben wir

DEF
O, Ω, Θ
$\ll$, $\gg$, $\asymp$

(1) $f(n)\in O(g(n))$ oder $f(n)\ll g(n)$,
falls es $n_0\in\mathbb{Z}_{\ge 0}$ und $c>0$ gibt, sodass $|f(n)|\le cg(n)$ für alle $n\ge n_0$ gilt;

(2) $f(n)\in\Omega(g(n))$ oder $f(n)\gg g(n)$,
falls es $n_0\in\mathbb{Z}_{\ge 0}$ und $c>0$ gibt, sodass $|f(n)|\ge cg(n)$ für alle $n\ge n_0$ gilt;

(3) $f(n)\in\Theta(g(n))$ oder $f(n)\asymp g(n)$,
falls $f(n)\in O(g(n))$ und $f(n)\in\Omega(g(n))$. ♢

Ist f zum Beispiel ein Polynom vom Grad 3 mit positivem Leitkoeffizienten, dann gilt

$$f(n)\in O(n^3)\,,\ f(n)\in O(n^4)\,,\ f(n)\in\Omega(n^3)\,,\ f(n)\in\Omega(n^2) \text{ und } f(n)\in\Theta(n^3)\,.$$

Man sieht auch häufig die Schreibweise „$f(n)=O(g(n))$“ etc., die allerdings problematisch ist, da die üblichen Regeln für Gleichheit (wie die Transitivität) nicht gelten. Wenn man den führenden Term im Wachstum von f genau angeben will, schreibt man auch z.B. $f(n)\in\frac{2}{3}n^3+O(n^2)$.

Damit können wir die Komplexität der Addition genauer als in $\Theta(\lambda(M)+\lambda(N))$ angeben, denn es werden auch mindestens $\max\{\lambda(M),\lambda(N)\}\ge\frac{1}{2}(\lambda(M)+\lambda(N))$

Rechenschritte ausgeführt. Die Komplexität ist also *linear* in der Länge der Eingabedaten. 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. Insofern ist der angegebene Algorithmus im Wesentlichen optimal.

Für die Addition von Polynomen gilt Entsprechendes. Hier verwendet man meistens die Anzahl der Operationen im Koeffizientenring R als Maß für die Komplexität. 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*. 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 in $O(n \log M)$. Auch wenn die Koeffizienten unterschiedliche Länge haben, ist der Aufwand für die Addition linear in der Eingabelänge; diese Aussage gilt ziemlich allgemein für die Addition auch von komplexeren Objekten.

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) kann man das wie folgt formulieren (wir nehmen an, dass es im Koeffizientenring R keine Nullteiler gibt und dass die Polynome p und q, falls $\neq 0$, normalisiert sind; dann ist das Produkt ebenfalls normalisiert oder 0):

ALGO Multiplikation von Polynomen

function multiply(p, q)
input: $p = \sum_{i=0}^{m} a_i X^i$, $q = \sum_{i=0}^{n} b_i X^i \in R[X]$.
output: $p \cdot q = \sum_{i=0}^{m+n} c_i X^i$.

```
if p = 0 or q = 0 then
  return 0 ∈ R[X] // Sonderfall: Produkt ist null
end if
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
```

Wie sieht es mit der Komplexität aus? 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 $\asymp \lambda(M) \cdot \lambda(N)$. Dabei stützt man sich auf einen Befehl des Prozessors, der aus Worten a und b Worte c_0 und c_1 berechnet mit $a \cdot b = c_0 + c_1 \cdot B$.

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.

Als letzte der Grundrechenarten bleibt noch die Division mit Rest.

Wir erinnern uns:

2.2. **Definition.** Ein Integritätsbereich R ist ein *euklidischer Ring* mit Normfunktion $N\colon R \to \mathbb{Z}_{\geq 0}$, wenn es zu $a, b \in R$ mit $b \neq 0$ stets $q, r \in R$ gibt, sodass

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

DEF euklidischer Ring

(Ein *Integritätsbereich* ist ein kommutativer Ring (in dem also $a \cdot b = b \cdot a$ gilt) ohne Nullteiler (aus $a \cdot b = 0$ folgt $a = 0$ oder $b = 0$).)

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

DEF Quotient Rest

Die wichtigsten Beispiele von euklidischen Ringen sind $R = \mathbb{Z}$ und der Polynomring $R = k[X]$ für einen Körper k. Für $R = \mathbb{Z}$ kann man $N(n) = |n|$ als Normfunktion benutzen, und für $R = k[X]$ setzt man $N(0) = 0$, $N(p) = 1 + \deg(p)$ (oder $N(p) = 2^{\deg(p)}$; das hat den Vorteil, dass wie für $\mathbb{Z}$ dann $N(pq) = N(p)N(q)$ gilt), wenn $p \neq 0$ ist.

Wir beweisen das für den Polynomring.

2.3. **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*

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

SATZ $k[X]$ ist euklidisch

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

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

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 der Leitkoeffizient von b eine Einheit in R ist (dann funktioniert das Verfahren für beliebige Koeffizientenringe R).

ALGO Polynomdivision

function divide(a, b)
input: $a = \sum_{i=0}^{n} a_i X^i$, $b = \sum_{i=0}^{m} b_i X^i \in R[X]$ mit $b_m \in R^\times$
output: (q, r) mit $q = \sum_{i=0}^{d_q} q_i X^i$, $r = \sum_{i=0}^{d_r} r_i X^i$, $a = qb + r$ und $d_r < m$

```
if n < m then return (0, a) end if
// Initialisierung
d_q = n − m
for i = 0 to n do r_i ← a_i end for
// Das Inverse des Leitkoeffizienten von b berechnen
u ← b_m^{-1}
// Die eigentliche Berechnung:
for j = n − m to 0 by −1 do
    q_j ← u · r_{m+j} // nächster Koeffizient von q
    // 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 // r_{j+m} wird nicht mehr benötigt
```

```
    end for
  end for
  // Normalisiere r
  for i = m − 1 to 0 by −1 do
    if r_i ≠ 0 then
      d_r ← i
      return ((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 ((d_q; q_0, ..., q_{d_q}), 0)
end function
```

Wie sieht es mit der Komplexität der Division aus?

Wir nehmen an, dass $n \geq m$ ist (sonst ist im Wesentlichen nichts zu tun). Dann ist die Anzahl an Operationen in R

eine Inversion und $(2m+1)(n-m+1)$ Multiplikationen und Additionen

(wir unterscheiden nicht zwischen Addition und Subtraktion). Für die Division eines Polynoms vom Grad $2n$ durch eines vom Grad n ergibt sich also ein Aufwand von $(2n+1)(n+1)+1 \asymp n^2$ Operationen; das ist wiederum *quadratisch*.

Für den Ring $\mathbb{Z}$ gilt:

SATZ $\mathbb{Z}$ ist euklidisch

2.4. **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:

ALGO schlechter Divisions-algorithmus

```
function divide(a, b)
input:  a ∈ ℤ_{≥0}, b ∈ ℤ_{>0}
output: 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 effizientere 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 (und korrekte) Umsetzung erfordert einiges an Hirnschmalz!

2.5. **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. ♢

DEF
quo, rem

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:

DEF ggT, kgV

3.1. **Definition.** Seien R ein Integritätsbereich und $a, b \in R$. 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 $k \mid k'$. ♢

Die Worte „größter“ und „kleinstes“ beziehen sich hier auf die Teilbarkeitsrelation, d.h., b wird als mindestens so groß wie a betrachtet, wenn b ein Vielfaches von a ist.

DEF assoziierte Elemente

3.2. **Definition.** Sei R ein Integritätsbereich. Zwei Elemente $a, b \in R$ heißen *assoziiert*, $a \sim b$, wenn $a \mid b$ und $b \mid a$ gilt. ♢

Aus der Reflexivität und der Transitivität der Teilbarkeitsrelation folgt leicht, dass Assoziiertheit eine Äquivalenzrelation ist.

Gilt $u \sim 1$, so ist u eine Einheit in R (also invertierbar). Allgemeiner gilt:

$$a \sim b \iff \exists u \in R^\times : b = u \cdot a\,.$$

(Die Assoziiertheitsklassen sind also gerade die Bahnen der natürlichen Operation von $R^\times$ auf R durch Multiplikation.)

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. Wenn es also größte gemeinsame Teiler von a und b gibt, dann bilden sie genau eine Assoziiertheitsklasse.

* **SATZ** Existenz des ggT

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:

ALGO Euklidischer Algorithmus

---

function gcd(a, b)
input: $a, b \in R$.
output: Ein ggT von a und b.

  while $b \neq 0$ **do** $(a, b) \leftarrow (b, a \text{ rem } b)$ **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

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

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. **Beispiel.** **BSP** ggT in $\mathbb{Z}$

$$\mathrm{ggT}(299, 221) \sim \mathrm{ggT}(221, 78) \sim \mathrm{ggT}(78, 65) \sim \mathrm{ggT}(65, 13) \sim \mathrm{ggT}(13, 0) \sim 13\,.$$

♣

In der „Einführung in die Zahlentheorie und algebraische Strukturen“ haben Sie gelernt, dass jeder euklidische Ring ein Hauptidealring ist. Im Wesentlichen gelten alle Aussagen über größte gemeinsame Teiler, die für euklidische Ringe richtig sind, allgemeiner auch in Hauptidealringen. Allerdings haben wir in einem beliebigen Hauptidealring keinen *Algorithmus* zur Verfügung, um ggTs zu berechnen.

3.5. **Satz.** *Seien R ein euklidischer Ring, $a, b \in R$; $d \in R$ sei ein größter gemeinsamer Teiler von a und b. Dann gibt es $s, t \in R$ mit $d = sa + tb$.* **SATZ** ggT als Linearkombination

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

ALGO EEA

---

function xgcd(a, b)
input: $a, b \in R$.
output: $(d, s, t) \in R^3$ mit $d \sim \mathrm{ggT}(a, b)$, $d = sa + tb$.

```
(r_0, s_0, t_0) ← (a, 1, 0)
(r_1, s_1, t_1) ← (b, 0, 1)
i ← 0
while r_{i+1} ≠ 0 do
    i ← i + 1
    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)
end while
return (r_i, s_i, t_i)
end function
```

Dass der Algorithmus terminiert, folgt wie in Satz 3.3. Außerdem prüft man nach, dass folgende Aussagen für alle relevanten i gelten:

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

Am Ende ist $r_{i+1} = 0$, $r_i \sim \mathrm{ggT}(a, b)$, und die Behauptung ist klar. ❑

Je nach Anwendung kann es auch sinnvoll sein, die vollständige Folge von Daten r_i, q_i, s_i, t_i zurückzugeben oder auch nur d und (z.B.) t zu berechnen.

3.6. **Beispiele.** **BSP** EEA

(1) 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 $\mathrm{ggT}(299, 221) \sim 13 = 3 \cdot 299 - 4 \cdot 221$.

(2) 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. ♣

Wir wollen jetzt die Komplexität des Euklidischen Algorithmus untersuchen. Wir betrachten zunächst den Algorithmus wie in Satz 3.3 für Polynome über einem Körper K. Wie wir im vorigen Abschnitt gesehen haben, brauchen wir für eine allgemeine Polynomdivision mit Rest eines Polynoms vom Grad n durch eines vom Grad $m \leq n$ insgesamt

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

Operationen in K.

Sei $n_i = \deg(r_i)$ im Algorithmus von Satz 3.5 (im Algorithmus von Satz 3.3 hatten wir die sukzessiven Reste nicht separat bezeichnet). Dann gilt $\deg(q_i) = n_{i-1} - n_i$ und $n = \deg(a) = n_0$, $m = \deg(b) = n_1$. Wir nehmen an, dass $n \geq m$ gilt. Der Aufwand in Operationen in K ist dann gegeben durch

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

Dabei ist ℓ die Anzahl der durchgeführten Divisionen. Wir betrachten zunächst den *Normalfall*, dass

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

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

$$\begin{aligned} s(n, m, m-1, m-2, \dots, 1, 0) &= \big(2m(n-m) + n + m + 2\big) + \sum_{i=2}^{m+1} (4m - 4i + 7) \\ &= 2m(n-m) + n + m + 2 + m(2m+1) \\ &= 2nm + n + 2m + 2 \le 2(n+1)(m+1) \end{aligned}$$

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 (mit $n_{j-1} > k > n_j$)

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

und (mit $n_\ell > k \ge 0$)

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

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 $2(3n-m)m + O(n) \le 4n^2 + O(n)$ Operationen (wobei man die Berechnung der s_i weglässt).

Im Fall von ganzen Zahlen ist eine gute Abschätzung der Anzahl der Schleifendurchläufe (wie sie für Polynome durch $\deg(b) + 1$ gegeben wird) nicht so offensichtlich. Hier hilft die Beobachtung, dass Folgendes gilt:

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

Nach jeweils zwei Schritten hat sich der Rest also mehr als 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.

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 eindeutig bestimmten 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 $\mathrm{normal}(a)$ berechenbar sind. Ein Element der Form $\mathrm{normal}(a)$ heißt *normalisiert* bzw. die *Normalform* von a.

DEF normalisiert Normalform

3.7. **Beispiele.**

(1) Für $a \in \mathbb{Z}$ setzt man $\mathrm{normal}(a) = |a|$.

(2) Für $0 \neq f \in K[X]$ setzt man $\mathrm{lu}(f) = \mathrm{lcf}(f)$; normalisiert sind also genau die „normierten“ Polynome mit Leitkoeffizient 1 (und das Nullpolynom). ♣

BSP Normalformen

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.

3.8. **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. ♢

DEF Funktionen ggT, kgV

Man kann dann den (Erweiterten) Euklidischen Algorithmus so anpassen, dass er am Ende das Ergebnis normalisiert (und ggf. 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:

ALGO
EEA
normalisiert

---

function xgcd(a, b)

input: $a, b \in R$.

output: $d, s, t \in R$ mit $d = \mathrm{ggT}(a, b)$, $d = sa + tb$.

$\rho_0 \leftarrow \mathrm{lu}(a)^{-1}$; $\rho_1 \leftarrow \mathrm{lu}(b)^{-1}$
$(r_0, s_0, t_0) \leftarrow (\rho_0 \cdot a, \rho_0, 0)$ // erste „Reste“ normalisieren
$(r_1, s_1, t_1) \leftarrow (\rho_1 \cdot b, 0, \rho_1)$ // Relation $r_i = s_i a + t_i b$ beachten
$i \leftarrow 0$
while $r_{i+1} \neq 0$ **do**
  $i \leftarrow i + 1$
  $q_i \leftarrow r_{i-1}$ quo r_i; $\tilde{r} \leftarrow r_{i-1}$ rem r_i // das ist *eine* Berechnung
  $\rho_{i+1} \leftarrow \mathrm{lu}(\tilde{r})^{-1}$
  $(r_{i+1}, s_{i+1}, t_{i+1}) \leftarrow \big(\rho_{i+1} \cdot \tilde{r}, \rho_{i+1} \cdot (s_{i-1} - q_i s_i), \rho_{i+1} \cdot (t_{i-1} - q_i t_i)\big)$
end while
return (r_i, s_i, t_i)
end function

Wenn nur der ggT benötigt wird, kann man die Teile, die der Berechnung der s_i und t_i dienen, natürlich 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 in $\mathbb{Q}[X]$ rechnet. Daher ist diese Variante schneller, obwohl sie mehr Operationen in $\mathbb{Q}$ ausführen muss.

Zum Abschluss dieses Abschnitts wollen wir uns noch einige Anwendungen des Euklidischen Algorithmus ansehen.

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 in R normalisiert ist). Wir nennen diese Darstellung wieder *normalisiert* bzw. die *Normalform* von a/b.

DEF
normalisiert
Normalform

Welche Art von Normalisierung jeweils gemeint ist, hängt vom Kontext ab. In einem Körper gilt $\mathrm{normal}(a) = 0$ für $a = 0$ und $\mathrm{normal}(a) = 1$ für $a \neq 0$; dort ist die Normalisierung von Assoziierten also nicht relevant.

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

ALGO
gekürzter
Bruch

function reduce(a, b)

input: $a, b \in R$, $b \neq 0$.

output: $a', b' \in R$ mit $\frac{a}{b} = \frac{a'}{b'}$, $\operatorname{ggT}(a', b') = 1$ und $\operatorname{lu}(b') = 1$.

$g \leftarrow \gcd(a, b)$
$a' \leftarrow a$ quo g; $b' \leftarrow b$ quo g // Division ohne Rest
$\rho \leftarrow \operatorname{lu}(b')^{-1}$
$a' \leftarrow \rho \cdot a'$; $b' \leftarrow \rho \cdot b'$
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 $\operatorname{ggT}(ac, bd)$ also ersetzen durch die Berechnung von $\operatorname{ggT}(a, d)$ und $\operatorname{ggT}(b, c)$, was im Allgemeinen etwas billiger ist:

ALGO Mult. von Brüchen

---

function multiply($\frac{a}{b}$, $\frac{c}{d}$)

input: $\frac{a}{b}, \frac{c}{d} \in K$ in Normalform.

output: $\frac{z}{n} = \frac{a}{b} \cdot \frac{c}{d}$ (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

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

ist dann

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

Außerdem gilt

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

Das liefert folgenden Additionsalgorithmus:

ALGO Add. von Brüchen

---

function add($\frac{a}{b}$, $\frac{c}{d}$)

input: $\frac{a}{b}, \frac{c}{d} \in K$ in Normalform.
output: $\frac{z}{n} = \frac{a}{b} + \frac{c}{d}$ (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 $\frac{z}{n}$
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.

Restklassenringe. Seien R ein euklidischer Ring und $I \subset R$ ein von null verschiedenes Ideal. Wir haben dann den *Faktorring* oder *Restklassenring* $\bar{R} = R/I$ und den *kanonischen Epimorphismus* $R \to \bar{R}$, $b \mapsto \bar{b}$. ($\bar{b}$ ist die Restklasse von b.) Da R ein Hauptidealring ist, ist $I = Ra = \langle a \rangle$ mit $0 \neq a \in R$.

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 (wobei 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 $* \in \{+, -, \cdot\}$ eines Rings 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 und schreiben n statt a):

ALGO
Addition in $\mathbb{Z}/n\mathbb{Z}$

---

function add($\bar{b}$, $\bar{c}$)
input: $\bar{b}, \bar{c} \in \mathbb{Z}/n\mathbb{Z}$, repräsentiert durch b, c mit $0 \leq b, c < n$.
output: $\bar{b} + \bar{c} = \bar{s}$ mit $0 \leq s < n$.

$s \leftarrow b + c$
if $s \geq n$ **then** $s \leftarrow s - n$ **end if**
return $\bar{s}$
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:

LEMMA Einheiten in $\bar{R}$

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

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. ❑

DEF relativ prim

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

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 $\ll \lambda(a)^2$.

DEF irreduzibel

3.11. **Definition.** Ist R ein Integritätsbereich 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. ♢

In einem Hauptidealring (also insbesondere in einem euklidischen Ring) sind die irreduziblen Elemente genau die *Primelemente*, also die Elemente $p \neq 0$, die keine Einheiten sind und für die die Implikation „$p \mid ab \Rightarrow p \mid a$ oder $p \mid b$“ gilt. Das von einem irreduziblen Element p erzeugte Ideal ist (wiederum in einem Hauptidealring) ein maximales Ideal; damit ist R/Rp ein Körper (und umgekehrt: Ist R/Ra ein Körper, dann ist a irreduzibel).

Im Fall $R = \mathbb{Z}$ und p eine Primzahl schreiben wir $\mathbb{F}_p$ für den Körper $\mathbb{Z}/p\mathbb{Z}$. Die vier Grundrechenoperationen in $\mathbb{F}_p$ haben dann also eine Wortkomplexität von $\ll \lambda(p)$ für Addition und Subtraktion und von $\ll \lambda(p)^2$ für Multiplikation und Division.

Im Fall $R = K[X]$ und $a \in K[X]$ irreduzibel erhalten wir als Restklassenringe R/Ra die endlichen Körpererweiterungen von K. Zum Beispiel ist

$$\mathbb{Q}(i) \cong \mathbb{Q}[X]/\langle X^2 + 1\rangle \qquad \text{oder} \qquad \mathbb{Q}(\sqrt[3]{2}) \cong \mathbb{Q}[X]/\langle X^3 - 2\rangle\,.$$

4. 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; die analogen Aussagen gelten für $R = K[X]$ bezüglich Operationen in K und ausgedrückt durch den Grad von a.

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

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

LEMMA effizientes Potenzieren

Beweis. Wir geben einen Algorithmus dafür an:

ALGO modulare Potenz

---

function modpower(a,b,e)

input: $a, b \in R$ mit $a \neq 0$, $a \notin R^\times$, $e \in \mathbb{Z}_{\geq 0}$.

output: b^e rem a.

```
  if e = 0 then return 1 end if
  if e = 1 then return b rem a end if // spart letztes Quadrieren
  (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 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, sodass die Rekursionstiefe gerade durch die Anzahl der Bits in der Dualdarstellung von e gegeben ist. Diese ist $\ll \lambda(e)$. ❑

4.2. **Beispiel.** 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.

BSP RSA

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

Die Berechnung von d aus n und e ist im Wesentlichen äquivalent dazu, n in die Primfaktoren p und q zu zerlegen. Dafür ist bisher kein effizienter (also mit Komplexität $\ll \lambda(n)^k$ für festes k) Algorithmus bekannt. Der asymptotisch schnellste bekannte Algorithmus (das „Zahlkörpersieb“) hat eine „subexponentielle“ Komplexität von $e^{O(1)(\log n)^{1/3}(\log\log n)^{2/3}}$. ♣

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 irreduzibel wählen), sodass jedes Element von R/aR von höchstens einem Element von R repräsentiert wird, das der Abschätzung genügt.

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

4.3. **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. **BSP** Determinante

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 $|a_{ij}| \le m$ für alle $1 \le i,j \le n$, dann ist (Hadamardsche Ungleichung)

$$\bigl|\det(a_{ij})\bigr| \le n^{n/2}\, m^n =: M\,.$$

Wenn wir eine Primzahl $p > 2M$ wählen, dann können wir die Determinante in $\mathbb{F}_p$ berechnen (Aufwand $\ll n^3\lambda(p)^2 \ll n^5(\lambda(m)+\lambda(n))^2$) und anschließend in $\mathbb{Z}$ rekonstruieren. Allerdings ergibt das noch keinen Vorteil gegenüber der Rechnung in $\mathbb{Q}$ (wie man durch Ausprobieren feststellen kann). Auf der anderen Seite sieht man so sehr leicht, dass man die Determinante in Polynomzeit berechnen kann (die Größe der Eingabe ist $n^2\lambda(m)$), 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 ungerade natürliche Zahlen $2M \le N \le 4M$, ob sie prim sind. Nach dem Primzahlsatz braucht man $\ll \log(2M) \ll n(\lambda(m)+\lambda(n))$ Versuche, bis man eine Primzahl gefunden hat, und der einzelne Test lässt sich in Polynomzeit (polynomial in $n(\lambda(m)+\lambda(n))$) durchführen. Gibt man sich mit „Pseudoprimzahlen“ zufrieden, ist der Test höchstens von kubischer Komplexität und man erhält insgesamt eine Laufzeit $\ll n^4(\lambda(m)+\lambda(n))^4$. 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.

SATZ Chinesischer Restsatz

4.4. **Satz.** *Sei R ein euklidischer Ring; $m_1, \dots, m_n \in R$ seien paarweise teilerfremd. Dann gibt es für jede Wahl von $a_1, \dots, a_n \in R$ 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

$$R/Rm \longrightarrow R/Rm_1 \times R/Rm_2 \times \cdots \times R/Rm_n$$

ist ein Isomorphismus.

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

ALGO CRS

---

function crt$\big((m_1, \dots, m_n), (a_1, \dots, a_n)\big)$

input: $m_1, \dots, m_n \in R$ paarweise teilerfremd, $a_1, \dots, a_n \in R$.

output: $x \in R$ mit $x \equiv a_j \bmod m_j$ für $1 \le j \le n$.

```
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/m_j, m_j)
    // Hier könnten wir überprüfen, dass g_j = 1 ist. t_j wird nicht gebraucht.
    x_j ← s_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 $s_j(m/m_j) + t_j m_j = g_j = 1$, also ist $x_j = s_j(m/m_j) \equiv 1 \bmod m_j$. Dass x_j durch alle m_i mit $i \ne j$ 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 Rekursion:

ALGO CRS

---

function crt2(m_1, m_2, a_1, a_2)

input: $m_1, m_2, a_1, a_2 \in R$, $m_1 \perp m_2$.

output: $x \in R$ mit $x \equiv a_1 \bmod m_1$, $x \equiv a_2 \bmod m_2$.

```
(g, s_1, s_2) ← xgcd(m_1, m_2)
// g = 1, s_1 m_1 + s_2 m_2 = 1
```

return $(a_1 \cdot s_2 \cdot m_2 + a_2 \cdot s_1 \cdot m_1) \operatorname{rem} (m_1 \cdot m_2)$
end function

function $\operatorname{crt}\big((m_1, \dots, m_n), (a_1, \dots, a_n)\big)$

input: $m_1, \dots, m_n \in R$ paarweise teilerfremd, $a_1, \dots, a_n \in R$.
output: $x \in R$ mit $x \equiv a_j \bmod m_j$ für $1 \le j \le n$.

if $n = 0$ **then return** 0 **end if**
if $n = 1$ **then return** $a_1 \operatorname{rem} m_1$ **end if**
$m \leftarrow m_1 \cdots m_{n-1}$
return $\operatorname{crt2}(m, m_n, \operatorname{crt}((m_1, \dots, m_{n-1}), (a_1, \dots, a_{n-1})), a_n)$
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 (Übung).

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

$$\begin{aligned} \forall j\colon x \equiv x' \bmod m_j &\iff \forall j\colon m_j \mid x - x' \\ &\iff \operatorname{kgV}(m_1, \dots, m_n) \mid x - x' \iff m_1 \cdots m_n \mid x - x' \\ &\iff x \equiv x' \bmod m\,, \end{aligned}$$

denn $\operatorname{kgV}(m_1, \dots, m_n) = m_1 \cdots m_n$, wenn die m_j paarweise teilerfremd sind. ❑

Eine Anwendung des Chinesischen Restsatzes ist die *Interpolation* von vorgegebenen Werten an vorgegebenen Stellen durch ein Polynom. Als Vorüberlegung zeigen wir, dass man die Auswertung eines Polynoms als die Berechnung eines Divisionsrestes interpretieren kann.

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

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

LEMMA
$f(\xi)$ als Divisionsrest

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

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

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

Für Polynome $m_j = X - \alpha_j$ (mit paarweise verschiedenen $\alpha_j \in K$) ist die Berechnung von $f \in K[X]$ mit $f \equiv a_j \bmod m_j$ also dasselbe 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 (vom Grad $< n$, dann ist es eindeutig bestimmt) mit $\ll n^2$ Operationen in K berechnen können (wir nehmen an, dass die a_j konstant sind).

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:

ALGO
Auswertung

function eval(f, α)
input: $f = a_n X^n + \cdots + a_1 X + a_0 \in K[X]$, $\alpha \in K$.
output: $f(\alpha) \in K$.

```
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.

Schnellere Multiplikation in $K[X]$? Man könnte jetzt versuchen, einen schnelleren Multiplikationsalgorithmus für Polynome zu bekommen, indem man die Polynome statt als Array von Koeffizienten durch die Liste ihrer Werte an geeigneten Stellen darstellt. Ein Polynom vom Grad n ist durch $n+1$ Werte eindeutig bestimmt. Wenn wir zwei Polynome vom Grad m und n miteinander multiplizieren wollen, müssen wir ihre Werte an $m+n+1$ 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, sodass 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.

Schnellere Determinante über $\mathbb{Z}$. Wir können zum Beispiel die Berechnung der Determinante über $\mathbb{Z}$ beschleunigen. Dazu wählen wir ℓ verschiedene Primzahlen p_j, sodass $P = p_1 \cdots p_\ell > 2M$ ist (Bezeichnungen wie in 4.3). Wir berechnen die Reduktion mod p_j unserer Matrix für alle j (Kosten: $\ll n^2\lambda(m)\sum_j \lambda(p_j) \ll n^2\lambda(m)\lambda(P)$), 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 $\lambda(P) \asymp n\lambda(nm)$ sind die Gesamtkosten also

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

(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 $\ell \approx \lambda(P) \asymp n\lambda(nm)$, und die Komplexität ist nur noch

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

Man vergleiche mit $n^5\lambda(nm)^2$ 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

$$\ell \approx \frac{\log M}{\log\log M} \asymp \frac{n\lambda(nm)}{\log(n\lambda(nm))}\,.$$

Damit bekommen wir eine Komplexität von

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

(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 $\approx n\lambda(nm)$ testen, ob sie prim sind. Der einzelne Test lässt sich in Polynomzeit in der Größe der getesteten Zahl durchführen, also in $\ll (\log(n\lambda(mn)))^k$ Wortoperationen für ein passendes k. Damit ist der Gesamtaufwand

$$\ll n\lambda(nm)\big(\log(n\lambda(mn))\big)^k\,,$$

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

5. Schnellere Multiplikation

Es gibt eine recht einfache Möglichkeit, schneller als in quadratischer Zeit zu multiplizieren. Der Grundgedanke dabei ist „Divide And Conquer“ („Teile und herrsche“): 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 er 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 insgesamt *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.

Die Karatsuba-Multiplikation.

Wir wollen natürlich nicht nur lineare Polynome multiplizieren. Seien also 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^2 Additionen), dann bekommen wir hier als Aufwand (M: Multiplikationen, A: Additionen/Subtraktionen in K)

• 3 Multiplikationen von Polynomen vom Grad $< n$:	$3n^2\mathsf{M} + 3n^2\mathsf{A}$
• 2 Additionen von Polynomen vom Grad $< n$:	$2n\,\mathsf{A}$
• 2 Subtraktionen von Polynomen vom Grad $< 2n$:	$4n\,\mathsf{A}$
• $2n$ Additionen von Koeffizienten beim „Zusammensetzen“:	$2n\,\mathsf{A}$

Insgesamt also $3n^2$ Multiplikationen und $3n^2 + 8n$ Additionen oder Subtraktionen. Die klassische Methode braucht $4n^2$ Multiplikationen und $4n^2$ Additionen/Subtraktionen. Damit ist die klassische Methode bereits für kleines n aufwendiger.

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

ALGO Karatsuba-Mult. I

function karatsuba(a, b, k)

input: $a, b \in K[X]$, $\deg(a), \deg(b) < 2^k$.

output: $a \cdot b$.

```
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
```

$c_2 \leftarrow \text{karatsuba}(a_1, b_1, k-1)$
$c_0 \leftarrow \text{karatsuba}(a_0, b_0, k-1)$
$c_1 \leftarrow \text{karatsuba}(a_1 + a_0, b_1 + b_0, k-1) - c_2 - c_0$
// 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

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

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

LEMMA Rekursion

5.1. **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

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

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.

Wie kommt man darauf? Die homogene lineare Rekursion $a_n = \lambda a_{n-1}$ hat offensichtlich als Lösungen genau die Folgen $(\gamma\lambda^n)_{n\geq 0}$ mit γ beliebig. Eine spezielle Lösung der inhomogenen Rekursion $a_n = \lambda a_{n-1} + \beta\mu^n$ erhält man mit dem Ansatz $a_n = \gamma'\mu^n$; das liefert $\frac{\beta\mu}{\lambda-\mu}(\lambda^n - \mu^n)$ und funktioniert, wenn $\lambda \neq \mu$ ist. Berücksichtigung des Anfangswerts $a_0 = \alpha$ ergibt dann die angegebene Lösung. Für den Fall $\lambda = \mu$ kann man den Grenzwert

$$\lim_{\mu\to\lambda} \mu \frac{\lambda^n - \mu^n}{\lambda - \mu} = n\lambda^n$$

verwenden. ❑

Aus Lemma 5.1 erhalten wir:

* **FOLG** Komplexität der Karatsuba-Mult.

5.2. **Folgerung.** *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.

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):

ALGO
Karatsuba-Mult. II

function karatsuba′$(a,\ b)$
input: $a, b \in K[X]$.
output: $a \cdot b$.

```
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 // m < 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).

Karatsuba für ganze Zahlen.

Für ganze Zahlen (anstelle von Polynomen) funktioniert der Algorithmus im Wesentlichen genauso: Sei $\lambda(a), \lambda(b) \le 2N$ 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 $\ge B^N$ 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 behandelt.

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

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“.

SATZ
Mult. mit Komplexität $\ll n^{1+\varepsilon}$

5.3. **Satz.** *Sei $\varepsilon > 0$ und sei 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 $m \geq 1$ so groß, dass $\log(2 - \frac{1}{m+1}) < \varepsilon \log(m+1)$ ist. 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

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

ein Isomorphismus. Sei M die Matrix der inversen Abbildung 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

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

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 $< N$.

Wir bezeichnen mit $T_m(k)$ den Gesamtaufwand für die rekursive Multiplikation von zwei Polynomen vom Grad $< (m+1)^k$. Dann ist

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

Mit Lemma 5.1 ergibt sich, dass $T_m(k) \ll (2m+1)^k$ ist. Für Polynome vom Grad $< n$ ergibt sich (durch Aufrunden des Grades oder durch ungefähre Teilung in $m+1$ Teile) eine Komplexität

$$\ll n^{\log_{m+1}(2m+1)} = n^{1+\log_{m+1}(2-1/(m+1))} \ll n^{1+\varepsilon}\,. \qquad \text{❑}$$

In der Praxis sind derartige Ansätze aber zu kompliziert, um gegenüber der Karatsuba-Methode (die eine Variante des Falles $m = 1$ ist) einen Vorteil zu ergeben. Um etwas Besseres zu bekommen, muss man das Auswerten und Interpolieren beschleunigen. Dafür macht man sich zunutze, dass man bei der Wahl der Stützstellen freie Hand hat. Die wesentliche Idee ist, dafür geeignete Einheitswurzeln zu verwenden. Das werden wir im folgenden Abschnitt näher untersuchen.

6. Die Diskrete Fourier-Transformation

Die Fourier-Transformation in der Analysis ist die Abbildung

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

(für geeignete Klassen von Funktionen $f\colon \mathbb{R} \to \mathbb{C}$). 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.

DEF (primitive) n-te Einheitswurzel

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

Aus diesen Bedingungen folgt, dass $n \cdot 1_R$ in R invertierbar ist (siehe unten).

BSP primitive Einheitsw.

6.2. **Beispiele.** Ist R ein Körper, dann ist $\omega \in R^\times$ genau dann eine primitive n-te Einheitswurzel, wenn ω Ordnung n in der multiplikativen Gruppe $R^\times$ hat.

Ein endlicher Körper $\mathbb{F}_q$ enthält also primitive n-te Einheitswurzeln genau für $n \mid q-1$, denn die Gruppe $\mathbb{F}_q^\times$ ist zyklisch von der Ordnung $q-1$.

$\mathbb{Q}$ und $\mathbb{R}$ enthalten die primitiven ersten und zweiten Einheitswurzeln 1 und -1.

$\mathbb{C}$ enthält primitive n-te Einheitswurzeln für alle n. Zum Beispiel ist $\zeta_n = e^{2\pi i/n}$ eine primitive n-te Einheitswurzel.

Der Ring $\mathbb{Z}$ enthält nach unserer Definition nur die erste primitive Einheitswurzel 1 (denn 2 ist nicht invertierbar in $\mathbb{Z}$). ♣

Wir beweisen die wichtigsten Eigenschaften von primitiven n-ten Einheitswurzeln:

LEMMA Eigensch. prim. n-te Einheitsw.

6.3. **Lemma.** *Sei R ein Ring und sei $\omega \in R$ eine primitive n-te Einheitswurzel. Sei weiter $\ell \in \mathbb{Z}$ mit $n \nmid \ell$. Dann gilt:*

(1) $\omega^\ell - 1 \in R^\times$.

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

(3) $n \cdot 1_R \in R^\times$.

Beweis.

(1) Für $k, m \in \mathbb{Z}$ gilt, dass $\omega^k - 1$ ein Teiler von $\omega^{km} - 1$ in R ist. Für $m \geq 0$ folgt das daraus, dass $X^m - 1$ als Polynom durch $X - 1$ teilbar ist. Für $m < 0$ verwenden wir $\omega^{km} - 1 = -\omega^{km}(\omega^{k(-m)} - 1)$.
Sei $g = \mathrm{ggT}(\ell, n) < n$. Dann gibt es einen Primteiler p von n mit $g \mid \frac{n}{p}$. Da $\omega^g - 1$ dann ein Teiler von $\omega^{n/p} - 1 \in R^\times$ ist, ist $\omega^g - 1$ ebenfalls invertierbar. 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 invertierbar.

(2) Es ist

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

Da $\omega^\ell - 1$ nach Teil (1) invertierbar ist, muss die Summe verschwinden.

(3) Es sei $\Phi_m \in \mathbb{Z}[X]$ das m-te Kreisteilungspolynom; seine komplexen Nullstellen sind gerade die primitiven m-ten Einheitswurzeln in $\mathbb{C}$. Man zeigt in der Algebra, dass Φ_m irreduzibel ist. Dann ist $X^n - 1 = \prod_{d|n}\Phi_d$ die Zerlegung von $X^n - 1$ in irreduzible Faktoren in $\mathbb{Z}[X]$. Für $d \mid n$, $d < n$, ist $\Phi_d(\omega)$ ein Teiler von $\omega^d - 1$, was nach Teil (1) invertierbar ist; damit ist $\Phi_d(\omega)$ ebenfalls invertierbar. Aus

$$0 = \omega^n - 1 = \prod_{d|n}\Phi_d(\omega)$$

folgt somit $\Phi_n(\omega) = 0$. Sei $\zeta \in \mathbb{C}$ eine primitive n-te Einheitswurzel. Obige Relation liefert dann einen Ringhomomorphismus

$$\phi\colon \mathbb{Z}[\zeta] \xrightarrow{\cong} \mathbb{Z}[X]/\langle\Phi_n\rangle \longrightarrow R\,,$$

der ζ auf ω abbildet. (Die erste Abbildung ist die Inverse von $\mathbb{Z}[X]/\langle\Phi_n\rangle \to \mathbb{Z}[\zeta]$, $\bar{X} \mapsto \zeta$, die nach dem Homomorphiesatz für Ringe ein Isomorphismus ist.) In $\mathbb{Z}[\zeta]$ gilt

$$n = \frac{X^n - 1}{X - 1}\Big|_{X=1} = \prod_{j=1}^{n-1}(1 - \zeta^j)\,.$$

Anwenden von ϕ ergibt

$$n \cdot 1_R = \phi(n) = \prod_{j=1}^{n-1}(1_R - \omega^j) \in R^\times\,,$$

da alle Faktoren nach Teil (1) invertierbar sind. ❑

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

$$a = a_{n-1}X^{n-1} + \cdots + a_1X + a_0 \in R[X]$$

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

* 6.4. **Definition.** Die R-lineare Abbildung **DEF** DFT

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

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

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

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

Wenn wir R^n mit $R[X]/\langle X^n - 1\rangle$ identifizieren, dann ist die Faltung genau die Multiplikation in diesem Restklassenring (denn $a *_n b \equiv ab \bmod X^n - 1$).

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

LEMMA Faltung und DFT

∗ 6.6. **Lemma.** *Seien $a, b \in R^n$ und $\omega \in R$ eine primitive n-te Einheitswurzel. Dann gilt*

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

Dabei steht $\bullet$ für die komponentenweise Multiplikation zweier Vektoren in R^n.

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 $\mathrm{DFT}_\omega(a*b)$:

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

denn $a * b \equiv a \cdot b \bmod X^n - 1$. ❑

Die lineare Abbildung DFT_ω ist sogar invertierbar.

SATZ DFT ist invertierbar

∗ 6.7. **Satz.** *Seien 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. Dass ω^{-1} eine primitive n-te Einheitswurzel ist, folgt aus Lemma 6.3, (1).

Es genügt, die zweite 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 nach Lemma 6.3, (2)

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

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

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

Es folgt, dass DFT_ω ein Ringisomorphismus zwischen $R[X]/\langle X^n - 1\rangle$ und R^n (mit komponentenweisen Operationen) ist. Dieser Isomorphismus kann als Spezialfall des Chinesischen Restsatzes interpretiert werden: Für $0 \leq j < k < n$ sind $X - \omega^j$ und $X - \omega^k$ in $R[X]$ relativ prim, denn $\omega^k - \omega^j = \omega^j(\omega^{k-j} - 1)$ ist invertierbar, und

$$\frac{1}{\omega^k - \omega^j}(X - \omega^j) - \frac{1}{\omega^k - \omega^j}(X - \omega^k) = 1\,.$$

Der Chinesische Restsatz liefert daher den Isomorphismus

$$\frac{R[X]}{\langle X^n - 1\rangle} = \frac{R[X]}{\langle \prod_{j=0}^{n-1}(X - \omega^j)\rangle} \longrightarrow \prod_{j=0}^{n-1} \frac{R[X]}{\langle X - \omega^j\rangle} \cong R^n\,,$$

wobei der letzte Isomorphismus das Produkt der durch Auswerten in ω^j induzierten Isomorphismen $R[X]/\langle X - \omega^j\rangle \to R$ 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 $R[X]/\langle X^N - 1\rangle$ vorgenommen haben. Der Aufwand

besteht in N Multiplikationen in R und N Multiplikation mit N^{-1} sowie den drei DFT-Berechnungen. Die Komplexität wird dabei von den DFT-Berechnungen dominiert (alles andere ist linear in N).

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. Dann gilt $\omega^m = -1$, denn

$$(\omega^m + 1)(\omega^m - 1) = \omega^n - 1 = 0$$

und $\omega^m - 1$ ist invertierbar.

Sei $a \in R[X]$ ein Polynom vom Grad $< n = 2m$. Wir schreiben

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

mit Polynomen a_0, a_1 vom Grad $< m$, ähnlich wie wir das schon bei der Karatsuba-Multiplikation getan haben. Dann gilt

$$\begin{aligned} a(\omega^{2j}) &= a_1(\omega^{2j})(\omega^m)^{2j} + a_0(\omega^{2j}) = (a_0 + a_1)(\omega^{2j}) \qquad \text{und} \\ a(\omega^{2j+1}) &= a_1(\omega^{2j+1})(\omega^m)^{2j+1} + a_0(\omega^{2j+1}) = (a_0 - a_1)(\omega \cdot \omega^{2j})\,. \end{aligned}$$

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).

ALGO
FFT

---

function $\mathrm{fft}(a,\, k,\, \omega)$

input: $a = (a_0, \ldots, a_{2^k-1}) \in R^{2^k}$, $\omega \in R$ primitive 2^k-te Einheitswurzel.

output: $\mathrm{DFT}_\omega(a) \in R^{2^k}$.

```
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

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

Dabei steht A wieder für Additionen und Subtraktionen und M für Multiplikationen in R (hier sind das nur Multiplikationen mit Potenzen von ω). Lemma 5.1 sagt uns dann, dass Folgendes gilt:

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

6.8. **Folgerung.** *Seien R ein Ring und $\omega \in R$ 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.*

FOLG Komplexität Faltung

Beweis. Wir verwenden folgenden Algorithmus:

ALGO Faltung mit FFT

---

function convolution(a, b, k, ω)

input: $a, b \in R[X]$, $\deg(a), \deg(b) < 2^k$, $\omega \in R$ primitive 2^kte Einheitswurzel.

output: $a *_{2^k} b \in R[X]$.

```
// 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. ❑

* 6.9. **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.*

SATZ schnelle Mult. in $R[X]$

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, ω) auf mit einer primitiven 2^k-ten Einheitswurzel $\omega \in R$. Man beachte, dass (mit dem minimalen k) $2^k < 2n \ll n$ gilt. ❑

„Three Primes"-FFT für die Multiplikation ganzer Zahlen.

Wir können Satz 6.9 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 $\lambda(a), \lambda(b) \le n$, und $B = 2^{64}$. Wir schreiben

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

mit Polynomen

$$\tilde{a} = a_{n-1}X^{n-1} + \cdots + a_1X + a_0 \quad \text{und} \quad \tilde{b} = b_{n-1}X^{n-1} + \cdots + b_1X + b_0\,,$$

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 $c_j \le n(B-1)^2$. 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 $n(B-1)^2 < pqr$ ist (das gilt für $n \le 2^{63}$; 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 pqr:

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

ALGO
Schnelle Mult. in $\mathbb{Z}$

---

function fast_multiply(a, b)

input: $a, b \in \mathbb{Z}_{>0}$ mit $\lambda(a), \lambda(b) < 2^{56}$.

output: $a \cdot b$.

$n \leftarrow \max\{\lambda(a), \lambda(b)\}$
$\tilde{a} \leftarrow a_{n-1}X^{n-1} + \cdots + a_0$ wie oben
$\tilde{b} \leftarrow b_{n-1}X^{n-1} + \cdots + b_0$ wie oben
$\tilde{a}_p \leftarrow \tilde{a}$ rem p; $\tilde{a}_q \leftarrow \tilde{a}$ rem q; $\tilde{a}_r \leftarrow \tilde{a}$ rem r // koeffizientenweise
$\tilde{b}_p \leftarrow \tilde{b}$ rem p; $\tilde{b}_q \leftarrow \tilde{b}$ rem q; $\tilde{b}_r \leftarrow \tilde{b}$ rem r// koeffizientenweise
$k \leftarrow \lceil \log_2 n \rceil + 1$
$\tilde{c}_p \leftarrow \mathrm{convolution}(\tilde{a}_p, \tilde{b}_p, k, \omega_p^{2^{57-k}})$ // Berechnung in $\mathbb{F}_p$
$\tilde{c}_q \leftarrow \mathrm{convolution}(\tilde{a}_q, \tilde{b}_q, k, \omega_q^{2^{57-k}})$ // Berechnung in $\mathbb{F}_q$
$\tilde{c}_r \leftarrow \mathrm{convolution}(\tilde{a}_r, \tilde{b}_r, k, \omega_r^{2^{57-k}})$ // Berechnung in $\mathbb{F}_r$
$\tilde{c} \leftarrow u \cdot \tilde{c}_p + v \cdot \tilde{c}_q + w \cdot \tilde{c}_r$
$\tilde{c} \leftarrow \tilde{c}$ rem pqr
return $\tilde{c}(2^{64})$

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 gelegentlich vorkommen kann.

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 man in $R[y]$ modulo $y^{2^{k-1}} + 1$ rechnet. Wenn man das geschickt genug macht, hat man nicht allzu viel zusätzlichen Aufwand, 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 6.8 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:

SATZ schnelle Mult. in alg. $R[X]$

6.10. **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:

* **SATZ** Satz von Schönhage und Strassen

6.11. **Satz.** *Es gibt einen Algorithmus, der das Produkt zweier ganzer Zahlen a und b der Länge $\lambda(a), \lambda(b) \le n$ 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.

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

Um die Notation zu vereinfachen und uns auf das Wesentliche zu konzentrieren, führen wir folgende „Soft-O“-Schreibweise ein.

∗ 6.12. **Definition.** Seien $f, g\colon \mathbb{Z}_{>0} \to \mathbb{R}_{>0}$ zwei Funktionen. Wir schreiben

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

wenn $g(n) \to \infty$ für $n \to \infty$ und es Konstanten $C > 0$ und k gibt, sodass $f(n) \le Cg(n)(\log g(n))^k$ gilt für alle hinreichend großen n.

Im Fall $g(n) = n$, also $f(n) \in \tilde{O}(n)$, sagen wir auch, f wachse *quasi-linear*. ♢

DEF
Soft-O
quasi-linear

Die obigen Sätze lassen sich dann so ausdrücken, dass man in $R[X]$ bzw. $\mathbb{Z}$ in $\tilde{O}(n)$ (Wort-)Operationen multiplizieren kann: die Komplexität ist quasi-linear.

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

$$R[X, Y]/\langle Y - X^{2m-1}\rangle$$

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 $\mathbb{Z}[X]/\langle X - B^m\rangle$.

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:

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

SATZ
quasi-lineare
Multiplikation

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.

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.)

7. 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 (zweimal stetig differenzierbaren) Funktion f, dann konvergiert (x_n) gegen diese Nullstelle, und die Konvergenz ist sehr schnell („quadratisch"), wenn es eine einfache Nullstelle ist.

Wir werden das jetzt analog anwenden; dabei wird „zwei Elemente sind nahe beieinander" als „ihre Differenz ist in einer hohen Potenz eines (gegebenen) Ideals enthalten" verstanden. (Man kann daraus tatsächlich eine Metrik machen.)

Wir betrachten also einen (kommutativen) Ring R und ein Ideal $I \subset R$. Ist $a \in R$, dann gilt

$$Ra + I = R \iff \bar a \in (R/I)^\times\,,$$

wobei $\bar a \in R/I$ die Restklasse von a bezeichnet. Denn die Gleichung ist äquivalent zu $1 \in Ra + I$ und damit zu $\exists b \in R\colon ba \equiv 1 \bmod I$. Daher sagen wir in diesem Fall auch, a sei *modulo I invertierbar*. Das lässt sich dann auf Potenzen von I ausdehnen, wie das folgende Lemma zeigt.

DEF mod I invertierbar

7.1. **Lemma.** *Seien R ein Ring, $I \subset R$ ein Ideal und $a \in R$ mit $Ra + I = R$. Dann gilt auch $Ra + I^n = R$ für alle $n \geq 1$.*

LEMMA Invertierbarkeit mod I^n

Beweis. Induktion über n; der Induktionsanfang $n = 1$ ist gegeben. Wir nehmen an, dass $Ra + I^n = R$ ist; es gibt also $b \in R$ und $c \in I^n$ mit $ba + c = 1$. Dann ist auch

$$1 = (ba + c)^2 = (b^2a + 2bc)a + c^2 \in Ra + I^{2n} \subset Ra + I^{n+1}\,. \qquad \text{❑}$$

Mit dieser Vorarbeit können wir jetzt das Newton-Verfahren in unserem Kontext beschreiben. Für ein Polynom $f \in R[X]$ bezeichne f' die formal mithilfe der üblichen Ableitungsregeln definierte Ableitung.

7.2. **Lemma.** *Seien R und I wie oben. Sei $f \in R[X]$ ein Polynom. Sei weiter $x_0 \in R$ mit $f(x_0) \in I$ und $f'(x_0)$ mod I invertierbar. Wir können eine Folge (x_n) von Elementen von R wie folgt definieren:*
Es ist $f'(x_n)$ mod I (und damit auch mod I^{2^n}) invertierbar. Sei $a_n \in R$ mit $a_n f'(x_n) \equiv 1 \bmod I^{2^n}$; wir setzen

LEMMA Newton-Verfahren

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

Dann gilt $f(x_n) \in I^{2^n}$ und $x_{n+1} - x_n \in I^{2^n}$.

Beweis. Induktion über n. Die Behauptungen „$f'(x_n)$ ist mod I invertierbar" und „$f(x_n) \in I^{2^n}$" stimmen für $n = 0$ nach Voraussetzung. Wir nehmen an, sie gelten für n und zeigen sie für $n + 1$, zusammen mit $x_{n+1} - x_n \in I^{2^n}$.

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

$$\begin{aligned} f(x_{n+1}) &= f(x_n) + (x_{n+1} - x_n)f'(x_n) + (x_{n+1} - x_n)^2 g_n(x_{n+1}) \\ &= f(x_n)(1 - a_n f'(x_n)) + (x_{n+1} - x_n)^2 g_n(x_{n+1}) \\ &\equiv (x_{n+1} - x_n)^2 g_n(x_{n+1}) \bmod I^{2^{n+1}} \end{aligned}$$

nach Definition von x_{n+1} und weil $f(x_n)$ und $1 - a_n f'(x_n)$ beide in I^{2^n} liegen. Die Aussage $x_{n+1} - x_n \in I^{2^n}$ folgt mit $f(x_n) \in I^{2^n}$ aus der Definition von x_{n+1}. Damit gilt

$$f(x_{n+1}) \in (I^{2^n})^2 = I^{2^{n+1}} .$$

Außerdem ist

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

mit einem Polynom $h_n \in R[X]$; damit ist $f'(x_{n+1}) = f'(x_n) + (x_{n+1} - x_n)h_n(x_{n+1})$. Da $f'(x_n)$ mod I invertierbar und $x_{n+1} - x_n \in I$ ist, folgt $f'(x_{n+1}) \equiv f'(x_n) \bmod I$. Da $f'(x_n)$ mod I invertierbar ist, gilt das dann auch für $f'(x_{n+1})$. ❑

Das folgende Ergebnis zeigt, dass x_n mod I^{2^n} eindeutig bestimmt ist:

LEMMA
Eindeutigkeit bei Newton

7.3. **Lemma.** *Seien R und I wie oben. Seien weiter $f \in R[X]$, $a, a' \in R$ und $n \in \mathbb{Z}_{>0}$ mit $a' \equiv a \bmod I$ und $f(a) \equiv f(a') \equiv 0 \bmod I^n$. Wenn $f'(a)$ mod I invertierbar ist, dann gilt $a' \equiv a \bmod I^n$.*

Das sagt also, dass es zu jeder Nullstelle $\alpha \in R/I$ von f mit $f'(\alpha) \in (R/I)^\times$ genau eine Nullstelle a mod I^n von f gibt mit $a + I = \alpha$.

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

$$\begin{aligned} 0 \equiv f(a') &= f(a) + (a' - a)f'(a) + (a' - a)^2 g(a') \\ &\equiv (a' - a)\big(f'(a) + (a' - a)g(a')\big) \bmod I^n . \end{aligned}$$

Es gilt

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

also ist $f'(a) + (a' - a)g(a')$ mod I und damit auch mod I^n invertierbar. Dann muss aber $a' \equiv a \bmod I^n$ sein. ❑

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 $K[X]/\langle X^n \rangle$ invertierbar ist; dann können wir a entsprechend skalieren. Hier ist $R = K[X]$ und $I = \langle X \rangle$.)

Wir wollen also die Gleichung

$$f(Y) = \frac{1}{Y} - a = 0$$

lösen. Der entsprechende Newton-Iterationsschritt sähe 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 7.2 nicht anwenden. Man sieht aber direkt, dass gilt

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

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

Wir erhalten folgenden Algorithmus.

ALGO Inverse mod X^n

function invert(a, n)
input: $a \in K[X]$ mit $a(0) = 1$, $n \geq 1$.
output: $b \in K[X]$ mit $\deg(b) < n$ und $ab \equiv 1 \bmod X^n$.

```
k ← 1
b ← 1 ∈ K[X]
while k < n do
  k ← min{2k, n}
  b ← (2b − a · b²) 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 $\leq \min\{2k, n\}$ (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 $\tilde{O}(n)$.

Schnelle Division mit Rest.

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

DEF rev_n

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

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

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

Seien nun $a, b \in R[X]$ mit $\deg(a) = n \geq m = \deg(b)$ 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 $a = qb + r$ 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, denn es ist ja $\deg(q) = n - m < n - m + 1$. Um $\mathrm{rev}_{n-m}(q)$ zu berechnen, bestimmen wir das Inverse von $\mathrm{rev}_m(b) \bmod X^{n-m+1}$ (beachte: $\mathrm{rev}_m(b)(0) = 1$) und multiplizieren mit $\mathrm{rev}_n(a)$; dann ist $q = \mathrm{rev}_{n-m}(\mathrm{rev}_{n-m}(q))$. Wir erhalten folgenden Algorithmus.

ALGO schnelle Division mit Rest

---

function quotrem(a, b)

input: $a, b \in R[X]$, b mit Leitkoeffizient 1.

output: $(q, r) \in R[X] \times R[X]$ mit $a = qb + r$ und $\deg(r) < \deg(b)$.

$n \leftarrow \deg(a)$
$m \leftarrow \deg(b)$
if $n < m$ **then return** $(0, a)$ **end if**
$\tilde{a} \leftarrow \mathrm{rev}_n(a)$
$\tilde{b} \leftarrow \mathrm{rev}_m(b)$
$\tilde{q} \leftarrow (\tilde{a} \cdot \mathrm{invert}(\tilde{b}, n - m + 1)) \bmod X^{n-m+1}$
$q \leftarrow \mathrm{rev}_{n-m}(\tilde{q})$
$r \leftarrow a - q \cdot b$
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:

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

SATZ Rechnen in $R[X]/\langle a\rangle$

Beweis. Wir repräsentieren die Elemente von $R[X]/\langle a\rangle$ 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 $(b \cdot c) \bmod a$ mit der schnellen Multiplikation und Division in $R[X]$; der Aufwand dafür ist wie angegeben. ❑

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). Stattdessen 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:

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

SATZ Rechnen in $\mathbb{Z}/N\mathbb{Z}$

Der Beweis ist völlig analog zu Satz 7.5.

* 7.7. **Folgerung.** *Die Ringoperationen im endlichen Körper $\mathbb{F}_{p^n}$ können mit einem Aufwand von $\tilde{O}(\lambda(p^n))$ Wortoperationen durchgeführt werden.*

FOLG Rechnen in endlichen Körpern

Beweis. Es ist $\mathbb{F}_{p^n} \cong \mathbb{F}_p[X]/\langle a\rangle$ für ein (beliebiges) irreduzibles Polynom $a \in \mathbb{F}_p[X]$ vom Grad n. Die Aussage folgt damit aus Satz 7.6 und Satz 7.5. ❑

Um auch Inverse in $\mathbb{F}_{p^e}$ schnell berechnen zu können, brauchen wir noch einen schnellen ggT.

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. p ist dabei ein beliebiges Element von R (das sinnvollerweise kein Nullteiler und keine Einheit ist). Sei dafür R ein beliebiger Ring; mit a rem b bezeichnen wir ein Element c von R mit $c \equiv a \bmod b$.

ALGO Inverse mod p^n

---

function invnewton(a, b, p, n)

input: $a, b, p \in R$ mit $ab \equiv 1 \bmod p$, $n \in \mathbb{Z}_{>0}$.

output: $c \in R$ mit $ac \equiv 1 \bmod p^n$.

```
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 $R = \mathbb{Z}$ (jeweils mit der üblichen Bedeutung von rem und 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).

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.

ALGO „Taylorentwicklung“

function gentaylor(a, p)
input: $a, p \in R[X]$, $\deg(p) > 0$, $\mathrm{lcf}(p) = 1$.
output: $(a_0, a_1, \ldots, a_n)$
mit $a = a_0 + a_1 p + \ldots a_n p^n$, $\deg(a_j) < \deg(p)$, $n = 0$ oder $a_n \neq 0$.

```
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) // schnelle Division mit Rest
  (a_0, ..., a_l) ← gentaylor(r, p) // mit l ≤ k − 1
  (a_k, ..., a_n) ← gentaylor(q, p)
  return (a_0, ..., a_l, 0, ..., 0, a_k, ..., a_n). // mit k − 1 − l Nullen
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 durch die zusätzlichen Multiplikationen der Form $p^m \cdot p^{2^e}$ höchstens um $\tilde{O}(k\deg(p))$ größer.

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.

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.

ALGO Newton ohne Division

---

function newton(f, a, b, p, n)

input: $f \in R[X]$, $a, b, p \in R$, $n \in \mathbb{Z}_{>0}$
mit $f(a) \equiv 0 \bmod p$ und $b \cdot f'(a) \equiv 1 \bmod p$.

output: $c \in R$ mit $f(c) \equiv 0 \bmod p^n$ und $c \equiv a \bmod p$.

```
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 aufgrund der gemachten Voraussetzungen. In der Zuweisung

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

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 7.2, 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

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

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 $c \leftarrow \big(c - f(c) \cdot d\big) \operatorname{rem} P$ (und analog für d) setzen sich zusammen aus den Kosten für die Berechnung von $f(c) \operatorname{rem} P$, plus einer weiteren Multiplikation mod P und einer Subtraktion. Um $f(c) \operatorname{rem} P$ zu berechnen, können wir das Horner-Schema (Algorithmus „Auswertung“ auf Seite 26) 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 die letzte Iteration dominieren die Kosten für die vorherigen Iterationen: Wir schreiben $\mathsf{M}(P)$ für die Kosten einer Multiplikation von Elementen der Größe von P. Dann gilt (für alle bisher besprochenen Multiplikationsalgorithmen) $\mathsf{M}(P^n) \geq n\,\mathsf{M}(P)$, und es folgt, dass

$$\sum_{j=0}^{m-1} \mathsf{M}(p^{2^j}) \leq \sum_{j=0}^{m-1} 2^{j-m}\,\mathsf{M}(p^{2^m}) < \mathsf{M}(p^{2^m}) \ll \mathsf{M}(p^n)$$

ist. Hier ist m die Anzahl der Iterationen der **while**-Schleife. Damit ist der Aufwand insgesamt

$$\ll \deg(f)\,\mathsf{M}(p^n) .$$

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 „Newton ohne Division“ auf Seite 48 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.

ALGO
$n$te Wurzel

function nthroot(a, n)

input: $a \in \mathbb{Z}_{>0}$, $n \in \mathbb{Z}_{>0}$ ungerade.
output: (**false**, 0), wenn $b^n = a$ in $\mathbb{Z}$ nicht lösbar ist;
(**true**, b) mit $b \in \mathbb{Z}_{>0}$ und $b^n = a$ sonst.

$k \leftarrow v_2(a)$ // $a = 2^k a'$ mit $2 \nmid a'$
if $k \operatorname{rem} n \neq 0$ **then return** (**false**, 0) **end if**
$a \leftarrow a/2^k$ // jetzt ist a ungerade
$m \leftarrow 1 + \lfloor (\log_2 a)/n \rfloor$ // Genauigkeit für newton
$b \leftarrow \text{newton}(X^n - a, 1, 1, 2, m)$
if $b^n = a$ **then**
  return (**true**, $b \cdot 2^{k \operatorname{quo} n}$)
else
  return (**false**, 0)
end if
end function

Wir beweisen die Korrektheit von „nthroot“: Wir schreiben zunächst $a = 2^k a'$ mit a' ungerade. Wenn $a = b^n$ ist, 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 $\text{newton}(X^n - a, 1, 1, 2, 1 + \lfloor (\log_2 a)/n \rfloor)$ berechnet die nach Lemma 7.3 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

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

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.

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. In diesem Fall 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 Nullstelle 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:

ALGO Quadratwurzel

function sqrt(a)
input: $a \in \mathbb{Z}_{>0}$.
output: (**false**, 0), wenn a kein Quadrat ist;
(**true**, b) mit $b \in \mathbb{Z}_{>0}$ und $b^2 = a$ sonst.

```
k ← v_2(a)
if k rem 2 ≠ 0 then return (false, 0) end if
a ← a/2^k
if a rem 8 ≠ 1 then return (false, 0) end if
m ← ⌊(log_2 a)/2⌋ − 1 // Genauigkeit für newton
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) // hier ist m ≥ 1
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, 0)
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.

8. Schnelle Algorithmen für verschiedene Berechnungen

„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.

Schnelle Auswertung an vielen Stellen.

Wir betrachten zunächst wie üblich den Ring $R[X]$ und $m_j = X - a_j$ mit $a_j \in R$, $j = 1, \dots, n$. Sei $f \in R[X]$ mit $\deg(f) < n$. Das erste Problem, das wir behandeln wollen, besteht darin, 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 $k = 0$ ist, 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 wie eben, nur dass wir statt Divisionen hier Multiplikationen benötigen.

Diese Analyse 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.

Etwas genauer sehen wir:

* 8.1. **Satz.** *Seien R ein Ring, $a_1, \dots, a_n \in R$ und sei $f \in R[X]$ mit $\deg(f) < n$. Seien weiter $\mathsf{M}(n)$ und $\mathsf{D}(n)$ obere Schranken für die Komplexität einer Multiplikation von Polynomen vom Grad $< n$ bzw. einer Division eines Polynoms vom Grad $< 2n$ 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 $\ll (\mathsf{M}(n) + \mathsf{D}(n)) \log n$ Operationen in R berechnen.*

SATZ simultane Auswertung

Wir haben uns schon überlegt, dass $\mathsf{D}(n) \ll \mathsf{M}(n) \in \tilde{O}(n)$ gilt. Damit lässt sich der Aufwand (wie oben) etwas gröber als $\tilde{O}(n)$ beschreiben.

Schnelle Interpolation.

Jetzt wollen wir das umgekehrte Problem betrachten. Sei dazu wieder R ein Ring und seien $a_1, \dots, a_n \in R$, sodass $a_i - a_j \in R^\times$ 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 Lagrangeschen Interpolationsformel ist

$$f = \sum_{j=1}^{n} b_j \frac{\prod_{i \neq j}(X - a_i)}{\prod_{i \neq j}(a_j - a_i)} = \sum_{j=1}^{n} \frac{b_j}{s_j} \frac{m}{m_j},$$

wenn

$$m = \prod_{j=1}^{n} m_j = \prod_{j=1}^{n} (X - a_j) \quad \text{und} \quad s_j = \prod_{i \neq j} (a_j - a_i)$$

ist. Die Elemente $s_j \in R^\times$ lassen sich wie folgt bestimmen:

$$s_j = m'(a_j) \qquad (\text{denn } m' = \sum_{j=1}^{n} \prod_{i \neq j} (X - a_i);$$

alle bis auf einen Summanden verschwinden, wenn man a_j einsetzt). Für diese Berechnung lässt sich also der vor Satz 8.1 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 $\ll \mathsf{M}(n) \log n$ Operationen in R. Dazu kommt die Berechnung der $M_{\kappa,i}$, die vergleichbare Komplexität hat und die Berechnung der s_j in $\ll (\mathsf{M}(n) + \mathsf{D}(n)) \log n$ 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 8.1. Wie vorher gilt die Komplexitätsaussage auch, wenn n keine Zweierpotenz ist.

Verallgemeinerung.

Das Verfahren von Satz 8.1 lässt sich analog anwenden in jedem Ring R (mit geeigneter Division mit Rest) und mit beliebigen m_j. Insbesondere haben wir:

SATZ simultane Reste in $K[X]$

8.2. **Satz.** *Sei K ein Körper und seien $m_1, \dots, m_n \in K[X] \setminus K$. Wir setzen $m = m_1 \cdots m_n$. 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.

SATZ simultane Reste in $\mathbb{Z}$

8.3. **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 $m_1, \dots, m_n \in R \setminus (\{0\} \cup R^\times)$ paarweise teilerfremd. Wir schreiben wieder $m = m_1 \cdots m_n$. Sei s_j ein Inverses von m/m_j modulo m_j. (Vorsicht: Dieses s_j ist, was vorher $1/s_j$ war.) Für $b_1, \dots, b_n \in R$ gilt dann

$$\Big(\sum_{i=1}^{n} (b_i s_i \text{ rem } m_i)\frac{m}{m_i}\Big) \text{ rem } m_j = \Big(b_j s_j \frac{m}{m_j}\Big) \text{ rem } m_j = b_j \text{ rem } m_j\,,$$

denn $s_j(m/m_j) \equiv 1 \bmod m_j$. Damit ist

$$x = \sum_{i=1}^{n} (b_i s_i \text{ rem } m_i)\frac{m}{m_i}$$

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 8.2 und 8.3 zugrunde 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; siehe die Sätze 8.7 und 8.8 später in diesem Abschnitt. 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. (Man beachte, dass der Aufwand für die Berechnung aller m/m_j mittels Division $\gg n\deg(m)$ Operationen in K bzw. $\gg n\lambda(m)$ Wortoperationen wäre, was den angestrebten Gewinn in der Komplexität zunichte machen würde. Daher der „Umweg“ über die r_j.)

Wenn wir die s_j und die Teilprodukte $M_{\kappa,i}$ berechnet haben, können wir die Linearkombination x wie oben mit einem Algorithmus in derselben Weise wie bei der

schnellen Interpolation berechnen. Wir erhalten analog zu den Sätzen 8.2 und 8.3 folgende Resultate:

SATZ schneller Chin. Restsatz in $K[X]$

8.4. **Satz.** *Sei K ein Körper und seien $m_1, \dots, m_n \in K[X] \setminus K$ paarweise teilerfremd; wir setzen $m = m_1 \cdots m_n$. Seien weiter $b_1, \dots, b_n \in K[X]$ gegeben mit $\deg(b_j) < \deg(m_j)$. Dann können wir das Polynom $f \in K[X]$ mit $f \operatorname{rem} m_j = b_j$ für $1 \le j \le n$ und $\deg(f) < \deg(m)$ in $\tilde{O}(\deg(m))$ Operationen in K berechnen.*

SATZ schneller Chin. Restsatz in $\mathbb{Z}$

8.5. **Satz.** *Seien $m_1, \dots, m_n \in \mathbb{Z}_{>1}$ paarweise teilerfremd. Setze $m = m_1 \cdots m_n$. Seien weiter $b_1, \dots, b_n \in \mathbb{Z}$ mit $0 \le b_j < m_j$. Dann können wir die ganze Zahl $a \in \mathbb{Z}$ mit $a \operatorname{rem} m_j = b_j$ für $1 \le j \le n$ und $0 \le a < m$ in $\tilde{O}(\lambda(m))$ Wortoperationen berechnen.*

BSP schneller CRS in $\mathbb{Z}$

8.6. **Beispiel.** Wir illustrieren den Algorithmus hinter Satz 8.5 mit einem einfachen Beispiel. Seien

$$(m_1, \dots, m_4) = (11, 13, 17, 19)$$

und

$$(b_1, \dots, b_4) = (2, 3, 5, 7)\,.$$

Wir berechnen zunächst die Teilprodukte

$$M_{1,0} = 11 \cdot 13 = 143\,, \quad M_{1,1} = 17 \cdot 19 = 323\,, \quad M_{2,0} = m = 143 \cdot 323 = 46\,189\,.$$

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 $c_1, \dots, c_4$ 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

$$x \operatorname{rem} m = 20\,660\,.$$

♣

Schneller ggT.

Auch für die ggT-Berechnung gibt es schnelle Algorithmen, die auf einem „Divide and Conquer"-Ansatz beruhen. Die zugrunde 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

$$r_0 = a\,,\ r_1 = b\,,\ r_2 = r_0 - q_1 r_1\,,\ r_3 = r_1 - q_2 r_2\,,\ \ldots\,,\ r_\ell = 0$$

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:

$$\begin{pmatrix} r_k \\ r_{k+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & -q_k \end{pmatrix} \begin{pmatrix} r_{k-1} \\ r_k \end{pmatrix} = Q_k \begin{pmatrix} r_{k-1} \\ r_k \end{pmatrix} = \ldots = Q_k Q_{k-1} \cdots Q_1 \begin{pmatrix} a \\ b \end{pmatrix} = R_k \begin{pmatrix} a \\ b \end{pmatrix}$$

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 wie folgt aus. Dabei setzen wir $R_k = R_{\ell-1}$ für $k \geq \ell$.

ALGO
schneller
ggT

---

function fast_gcd(a, b, k)

input: $a, b \in R$, $k \in \mathbb{Z}_{\geq 0}$.

output: $M \in \mathrm{Mat}(2, R)$ mit $M = R_k$ wie oben.

```
  if k = 0 then return ( 1 0 ; 0 1 ) end if
  if a oder b „klein" then
    berechne M = R_k direkt; return M
  end if
  a' ← ausreichend langes Anfangsstück von a
  b' ← ausreichend langes Anfangsstück von b
  M' ← fast_gcd(a', b', ⌊k/2⌋)    // berechnet M' = R_⌊k/2⌋
  ( r ; r' ) ← M' ( a ; b )    // (r, r') = (r_⌊k/2⌋, r_⌊k/2⌋+1)
  if r' = 0 then return M' end if    // r = ggT(a, b), ⌊k/2⌋ ≥ ℓ − 1
  (q, r'') ← quotrem(r, r')    // ein Schritt im Euklidischen Algorithmus
  M'' ← ( 0 1 ; 1 −q ) · M'    // M'' = R_⌊k/2⌋+1
  return fast_gcd(r', r'', k − 1 − ⌊k/2⌋) · M''
end function
```

Wenn $k \geq \ell - 1$ ist, dann ist $M \left(\begin{smallmatrix} a \\ b \end{smallmatrix}\right) = \left(\begin{smallmatrix} \mathrm{ggT}(a,b) \\ 0 \end{smallmatrix}\right)$. Wenn der zweite Eintrag dieses Vektors $\neq 0$ ist, dann kann man fast_gcd mit den Einträgen dieses Vektors und einem geeigneten Wert von k aufrufen (ähnlich wie in der Rekursion oben). In der Praxis wird man $k = \infty$ erlauben mit der Bedeutung, dass tatsächlich $R_{\ell-1}$ und damit der ggT berechnet werden soll. Dann muss man im ersten rekursiven Aufruf $\lfloor k/2 \rfloor$ durch eine geeignete Schätzung von $\ell/2$ ersetzen.

Siehe [GG, § 11] für eine detaillierte Beschreibung dieses Ansatzes, wenn a und b Polynome über einem Körper sind.

Wir beachten noch, dass die Koeffizienten s_n und t_n im Erweiterten Euklidischen Algorithmus dieselbe Rekursion erfüllen wie die Reste r_n, aber mit Startwerten

$s_0 = 1$, $s_1 = 0$ und $t_0 = 0$, $t_1 = 1$. Es folgt

$$\begin{pmatrix} s_k \\ s_{k+1} \end{pmatrix} = R_k \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \text{und} \quad \begin{pmatrix} t_k \\ t_{k+1} \end{pmatrix} = R_k \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

Insbesondere ist

$$R_k = \begin{pmatrix} s_k & t_k \\ s_{k+1} & t_{k+1} \end{pmatrix}.$$

Es gilt stets $r_n = s_n a + t_n b$, also sind $s = s_{\ell-1}$ und $t = t_{\ell-1}$ die Koeffizienten der Linearkombination, die $g = \mathrm{ggT}(a, b)$ in der Form $sa + tb$ darstellt. Das bedeutet, dass die erste Zeile von $R_{\ell-1}$ gerade aus s und t besteht. Wir bezeichnen das Tripel (g, s, t) als „die Resultate des EEA“.

Man erhält folgende Aussagen:

SATZ schneller EEA in $K[X]$

8.7. **Satz.** *Sei K ein Körper und seien $a, b \in K[X]$. 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.*

SATZ schneller EEA in $\mathbb{Z}$

8.8. **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:

FOLG Rechnen in $K[X]/\langle f \rangle$

∗ 8.9. **Folgerung.** *Sei K ein Körper und sei $f \in K[X]$ irreduzibel. Dann lassen sich die Körperoperationen im Körper $L = K[X]/\langle f \rangle$ mit einem Aufwand von $\tilde{O}(\deg(f))$ Operationen in K durchführen.*

FOLG Rechnen in $\mathbb{F}_p$

∗ 8.10. **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“.

9. 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 $\mathbb{F}_q$.

Wiederholung: Endliche Körper.

Wir erinnern uns an einige wichtige Tatsachen über endliche Körper.

SATZ endliche Körper

9.1. **Satz.** *Ist K ein endlicher Körper, dann gibt es eine Primzahl p und eine positive ganze Zahl e, sodass $\#K = p^e$ ist. Der Körper K ist eine Körpererweiterung vom Grad e des „Primkörpers“ $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$.*

Beweis. Man betrachtet den kanonischen Ringhomomorphismus $\varphi\colon \mathbb{Z} \to K$. Sein 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. ❑

LEMMA Teilkörper von endl. Körpern

9.2. **Lemma.** *Sei K ein endlicher Körper mit $\#K = q$ und sei $K \subset L$ eine Körpererweiterung. Dann gilt $K = \{x \in L \mid x^q = x\}$.*

Beweis. 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:

FOLG Zerlegung von $X^q - X$

9.3. **Folgerung.** *Sei K ein endlicher Körper mit $\#K = q$. Dann gilt in $K[X]$:*

$$X^q - X = \prod_{\alpha \in K} (X - \alpha) .$$

LEMMA Frobenius-Homomorphismus

9.4. **Lemma.** *Sei $K \subset L$ eine Körpererweiterung, wobei K endlich sei mit q Elementen. Dann ist die Abbildung $\phi\colon L \to L$, $x \mapsto x^q$, ein K-linearer Ringhomomorphismus.*

Beweis. Dass $\phi(xy) = (xy)^q = x^q y^q = \phi(x)\phi(y)$ gilt, ist klar. Nach Lemma 9.2 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 $\phi(x+y) = (x+y)^q = x^q + y^q = \phi(x) + \phi(y)$. ❑

DEF Frobenius-Automorphismus

Ist die Körpererweiterung endlich, dann ist ϕ ein Automorphismus von $K \subset L$ (denn ϕ ist offensichtlich injektiv: $\ker(\phi) = \{0\}$, also wegen der endlichen Dimension des K-Vektorraums L auch surjektiv); ϕ heißt der *Frobenius-Automorphismus* von $K \subset L$.

SATZ Existenz und Eindeutigkeit endl. Körper

9.5. **Satz.** *Seien p prim, $e \geq 1$ und $q = p^e$. Dann gibt es bis auf Isomorphie genau einen Körper $\mathbb{F}_q$ mit q Elementen.*

Beweis. 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 Lemma 9.4 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 Lemma 9.2 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. ❑

Folgerung 9.3 hat eine Verallgemeinerung:

LEMMA Zerlegung von $X^{q^n} - X$

* 9.6. **Lemma.** *$X^{q^n} - X \in \mathbb{F}_q[X]$ ist das Produkt aller normierten irreduziblen Polynome in $\mathbb{F}_q[X]$, deren Grad ein Teiler von n ist.*

Beweis. Der Körper $\mathbb{F}_{q^n}$ ist eine Körpererweiterung vom Grad n von $\mathbb{F}_q$. Nach Folgerung 9.3 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

$$\prod_{M \in \{M_\alpha \mid \alpha \in \mathbb{F}_{q^n}\}} M \mid X^{q^n} - X\,.$$

Da die linke Seite aber alle $\alpha \in \mathbb{F}_{q^n}$ als Nullstellen hat (und beide Polynome normiert sind), gilt Gleichheit. Sei nun $f \in \mathbb{F}_q[X]$ irgendein 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 $f = M_\beta$. ❑

9.7. **Satz.** *Sei K ein endlicher Körper. Dann ist die multiplikative Gruppe $K^\times$ zyklisch.*

SATZ
$K^\times$ zyklisch

Beweis. 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. ❑

9.8. **Lemma.** *Sei K ein endlicher Körper und sei $k \in \mathbb{Z}_{>1}$ 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.

LEMMA
$k$te Potenzen

Beweis. Sei $\#K = q$. Nach Satz 9.7 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).

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 Folgerung 9.3, dass

$$\operatorname{ggT}(f, X^q - X) = \prod_{\alpha \in \mathbb{F}_q, f(\alpha)=0} (X - \alpha)$$

das Produkt der irreduziblen Faktoren von f vom Grad 1 ist. Die Berechnung von $\operatorname{ggT}(f, X^q - X)$ erfolgt dabei am besten als $\operatorname{ggT}(f, (X^q \operatorname{rem} f) - X)$, wobei wir $X^q \operatorname{rem} f$ durch sukzessives Quadrieren im Ring $\mathbb{F}_q[X]/\langle f \rangle$ effizient berechnen können.

Zur Isolation der Faktoren mit höherem Grad verwenden wir entsprechend Lemma 9.6. Das führt auf folgenden Algorithmus zur „Distinct Degree Factorization“.

ALGO
distinct degree factorization

---

function ddf(f)

input: $f \in \mathbb{F}_q[X]$ quadratfrei mit $\operatorname{lcf}(f) = 1$.

output: $(u_1, u_2, \ldots, u_d)$ mit $u_j \in \mathbb{F}_q[X]$ Produkt von normierten irreduziblen Polynomen vom Grad j, $u_d \neq 1$ und $f = u_1 \cdots u_d$.

$u \leftarrow ()$ // 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]/⟨f⟩
    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. In diesem Fall spart man sich so die Hälfte der Arbeit.

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 desselben 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 Lemma 9.8: 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:

ALGO
Nullstellen

---

function zeros(f)

input: $f \in \mathbb{F}_q[X]$ normiert mit $f \mid X^q - X$, q ungerade.

output: $Z \subset \mathbb{F}_q$, die Menge der Nullstellen von f.

  if $\deg(f) = 0$ **then return** $\emptyset$ **end if**
  if $\deg(f) = 1$ **then return** $\{-f(0)\}$ **end if**
  $a \leftarrow$ zufälliges Element von $\mathbb{F}_q$
  $h \leftarrow (X-a)^{(q-1)/2}$ rem f // sukzessives Quadrieren
  $f_1 \leftarrow \gcd(f, h-1)$
  $f_2 \leftarrow f/f_1$
  return zeros(f_1) $\cup$ zeros(f_2)
end function

Die Komplexität ist $\tilde{O}(n \log q)$ Operationen in $\mathbb{F}_q$ mal die maximale Rekursionstiefe r (mit $n = \deg(f)$ wie oben), denn der Aufwand für einen Durchlauf ist $\tilde{O}(n \log q)$ Operationen in $\mathbb{F}_q$ (analog zum vorigen Algorithmus), und die Grade aller Polynome, die in Aufrufen einer festen Rekursionstiefe bearbeitet werden, summieren sich höchstens zu n (ihr Produkt ist ein Teiler von f). Man kann erwarten, dass $r \ll \log n$ 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. (Die genaue Wahrscheinlichkeit ist $\frac{1}{2}(1-q^{-2})$.) 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 $k/\log n$ groß ist. Insgesamt ist die erwartete Komplexität

$$\tilde{O}(n \log n \log q) = \tilde{O}(n \log q) \quad \text{Operationen in } \mathbb{F}_q.$$

Wenn $q = 2^m$ ist, $\mathbb{F}_q$ also Charakteristik 2 hat, kann man statt dessen verwenden, dass die Funktion

$$\mathrm{Tr}_{\mathbb{F}_q/\mathbb{F}_2} \colon x \longmapsto x + x^2 + x^4 + \ldots + x^{q/2}$$

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 $\mathbb{F}_2 \subset \mathbb{F}_q$, eine $\mathbb{F}_2$-lineare Abbildung $\mathbb{F}_q \to \mathbb{F}_2$, die surjektiv ist, weil $\mathbb{F}_2 \subset \mathbb{F}_q$ separabel ist.) Man

kann zeigen, dass man jede Untergruppe vom Index 2 der additiven Gruppe $\mathbb{F}_q$ 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:

ALGO Nullstellen in Char. 2

function zeros2(f)
input: $f \in \mathbb{F}_q[X]$ normiert mit $f \mid X^q - X$, $q = 2^m$.
output: $Z \subset \mathbb{F}_q$, 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
```

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

$$\begin{aligned} \varphi\colon \mathbb{F}_q[X]/\langle f\rangle &\longrightarrow \prod_{j=1}^{k} \mathbb{F}_q[X]/\langle h_j\rangle \cong (\mathbb{F}_{q^d})^k \\ a &\longmapsto (a \bmod h_1, \ldots, a \bmod h_k) \end{aligned}$$

ein Isomorphismus von Ringen ist. Wählen wir a zufällig und gleichverteilt in $\mathbb{F}_q[X]/\langle f\rangle$, dann ist $\varphi(a)$ ein zufälliges Element in $(\mathbb{F}_{q^d})^k$. Wenn $a \perp f$ ist, dann ist $\varphi(a)$ ein zufälliges Element von $(\mathbb{F}_{q^d}^\times)^k$, 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

$$g := \mathrm{ggT}(f, a^{(q^d-1)/2} - 1) = \prod_{j\colon b_j = 1} h_j\,.$$

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 $g \ne f$, sodass $f = g \cdot (f/g)$ eine nichttriviale Faktorisierung ist. Wir wenden dann dieselbe Idee rekursiv auf g und auf f/g an und erhalten folgenden Algorithmus für die „Equal Degree Factorization“.

ALGO equal degree factorization

function edf(f, d)

input: $f \in \mathbb{F}_q[X]$, q ungerade, f normiert und Produkt von verschiedenen irreduziblen Polynomen vom Grad d.

output: $(h_1, \ldots, h_k)$ mit $h_j \in \mathbb{F}_q[X]$ irreduzibel und normiert, $f = h_1 \cdots h_k$.

if $\deg(f) = 0$ **then return** () **end if** // f konstant
if $\deg(f) = d$ **then return** (f) **end if** // f irreduzibel
// ab hier ist $k \geq 2$.
$a \leftarrow$ zufälliges Polynom in $\mathbb{F}_q[X]$ vom Grad $< \deg(f)$
$g \leftarrow \gcd(f, a)$ // Test ob $a \perp f$
if $g \neq 1$ **then return** $\mathrm{edf}(g, d)\,\mathrm{cat}\,\mathrm{edf}(f/g, d)$ **end if**
$g \leftarrow \gcd(f, a^{(q^d-1)/2} \text{ rem } f - 1)$
return $\mathrm{edf}(g, d)\,\mathrm{cat}\,\mathrm{edf}(f/g, d)$
end function

Hier ist

$$(a_1, \ldots, a_m)\,\mathrm{cat}(b_1, \ldots, b_n) = (a_1, \ldots, a_m, b_1, \ldots, b_n)\,.$$

Die Kosten für einen Durchlauf betragen ($n = \deg(f)$ wie oben)

- $O(n)$ 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 (analog wie bei der Bestimmung der Nullstellen) die Summe der Werte von n aller Aufrufe auf derselben Ebene des Baumes immer höchstens gleich dem ursprünglichen n (denn $\deg(g) + \deg(f/g) = \deg(f)$). Der Gesamtaufwand ist damit höchstens

$$\tilde{O}(rnd \log q) \quad \text{Operationen in } \mathbb{F}_q,$$

wobei r wieder die maximale Rekursionstiefe bezeichnet. Wir müssen also den Erwartungswert von r betrachten. Wie bei der Trennung der Nullstellen sieht man, dass dieser Erwartungswert in $O(\log k)$ ist.

Damit erhalten wir (unter Beachtung von $k \leq n$, also $\log k \leq \log n$) folgende Aussage:

Der Erwartungswert für den Aufwand des obigen Algorithmus ist

$$\in \tilde{O}(nd \log q \log k) = \tilde{O}(nd \log q) \quad \textit{Operationen in } \mathbb{F}_q.$$

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.

ALGO
Faktorisierung in $\mathbb{F}_q[X]$

function factor(f)
input: $f \in \mathbb{F}_q[X]$ normiert, q ungerade.
output: $((h_1, e_1), \ldots, (h_k, e_k))$ mit $h_j \in \mathbb{F}_q[X]$ normiert und irreduzibel, $e_j \geq 1$, $f = \prod_j h_j^{e_j}$.

```
u ← () // leere Liste
d ← 0 // Grad der aktuell betrachteten Faktoren
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]/⟨f⟩
  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
      // Exponent e = e_j von h_j bestimmen
      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.

Quadratfreie Faktorisierung.

Alternativ kann man zuerst f in quadratfreie Bestandteile zerlegen. Dabei bestimmt man paarweise teilerfremde quadratfreie Polynome $h_1, h_2, \ldots, h_m$, sodass $f = h_1 h_2^2 \cdots h_m^m$. Hat f diese Form, dann gilt in Charakteristik 0, dass

$$\operatorname{ggT}(f, f') = h_2 h_3^2 \cdots h_m^{m-1} \quad \text{und} \quad \frac{f}{\operatorname{ggT}(f, f')} = h_1 h_2 \cdots h_m \,,$$

denn

$$f' = h_2 h_3^2 \cdots h_m^{m-1} (h_1' h_2 \cdots h_m + 2 h_1 h_2' h_3 \cdots h_m + \ldots + m h_1 \cdots h_{m_1} h_m') \,,$$

und der zweite Faktor, er sei mit h bezeichnet, ist wegen

$$\operatorname{ggT}(h, h_j) = \operatorname{ggT}(j h_1 \cdots h_{j-1} h_j' h_{j+1} \cdots h_m, h_j) = \operatorname{ggT}(j h_j', h_j) = \operatorname{ggT}(j, h_j)$$

(beachte $h_j \perp h_i$ für $i \neq j$ und $h_j \perp h_j'$) teilerfremd zu f. Damit lassen sich die h_j dann iterativ bestimmen.

In Charakteristik p gilt wegen $p = 0$ abweichend, dass

$$\operatorname{ggT}(f, f') = h_2 h_3^2 \cdots h_m^{m-1} \cdot \prod_{p|k} h_k \,,$$

also

$$\frac{f}{\operatorname{ggT}(f, f')} = \prod_{p\nmid k} h_k \,.$$

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 weitermachen. Eine genaue Beschreibung findet sich in [GG, § 14.6].

Effizientere Berechnung der q-ten Potenzen.

Während der Faktorisierung von f müssen wir die iterierten q-ten Potenzen von X im Restklassenring $\mathbb{F}_q[X]/\langle f\rangle$ 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. (Ist K ein Körper, dann ist eine *K-Algebra* ein Ring R mit einem Ringhomomorphismus $\phi\colon K \to R$. Mittels $\lambda \cdot r := \phi(\lambda) r$ wird dann R zu einem K-Vektorraum.) Insbesondere ist

$$\phi\colon \mathbb{F}_q[X]/\langle f\rangle \longrightarrow \mathbb{F}_q[X]/\langle f\rangle \,, \qquad a \longmapsto a^q$$

eine $\mathbb{F}_q$-lineare Abbildung. Wenn wir eine $\mathbb{F}_q$-Basis von $\mathbb{F}_q[X]/\langle f\rangle$ 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 $a \leftarrow a^q \operatorname{rem} f$ 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 $X^{q^j} \bmod f$.

ALGO
Potenzen
$X^{q^j} \bmod f$

---

function itfrob(f, m)
input: $f \in \mathbb{F}_q[X]$, $m \geq 0$.
output: $(X^q \operatorname{rem} f, X^{q^2} \operatorname{rem} f, \ldots, X^{q^d} \operatorname{rem} f)$ mit $d \geq m$.

```
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 Satz 8.1; dabei sei $(u_1, \ldots, u_k) \bmod f = (u_1 \bmod f, \ldots, u_k \bmod f)$ ein Tupel von Elementen von $\mathbb{F}_q[X]/\langle f\rangle$. 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

$$u = \left(X^q \text{ rem } f, X^{q^2} \text{ rem } f, \ldots, X^{q^{2^k}} \text{ rem } f\right).$$

(Das gilt zu Beginn mit $k = 0$.) Dann wird $h = X^{q^{2^k}} \text{ rem } f$, und wir werten h aus an den Stellen

$$X^q \bmod f, X^{q^2} \bmod f, \ldots, X^{q^{2^k}} \bmod f \in \mathbb{F}_q[X]/\langle f\rangle.$$

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 $\mathbb{F}_q$. (Siehe Satz 8.1. Wenn $l > k$ ist, dann berechnet man zuerst $f \text{ rem } m$ (wobei m das Produkt der $X - \alpha$ ist für die Auswertungsstellen α; f ist hier das auszuwertende Polynom wie im Satz); das liefert den Beitrag $\tilde{O}(l)$. Ist l deutlich kleiner als k, dann teilt man die k Auswertungsstellen in $\approx k/l$ Gruppen zu je $\approx l$ Stellen auf. Das liefert einen Aufwand von $\tilde{O}((k/l) \cdot l) = \tilde{O}(k)$.) 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

$$\frac{q^d - 1}{2} = \left(1 + q + q^2 + \cdots + q^{d-1}\right) \frac{q - 1}{2}$$

verwendet. Wenn man die Werte von $X^{q^j} \text{ rem } f$ 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.

* 9.9. **Satz.** *Die vollständige Faktorisierung eines normierten Polynoms $f \in \mathbb{F}_q[X]$ vom Grad n kann mit einem erwarteten Aufwand von*

$$\tilde{O}\big(n^2 \log q + n(\log q)^2\big) \quad \textit{Wortoperationen}$$

bestimmt werden.

SATZ Faktorisierung in $\mathbb{F}_q[X]$

Man kann sich fragen, wie wahrscheinlich es ist, dass die „Distinct Degree Factorization“ ein gegebenes normiertes Polynom schon vollständig faktorisiert. Das ist also die Wahrscheinlichkeit $P(q,n)$ dafür, dass ein zufälliges normiertes Polynom vom Grad n in $\mathbb{F}_q[X]$ quadratfrei ist und in irreduzible Faktoren lauter verschiedener Grade zerfällt. Für großes n und q ist die Antwort „größer als 56%“. Genauer gilt

$$\lim_{n,q\to\infty} P(q,n) = e^{-\gamma} \approx 0.5614594836\,,$$

wobei

$$\gamma = \lim_{N\to\infty}\Big(\sum_{n=1}^{N} \frac{1}{n} - \log N\Big)$$

die Euler-Mascheronische Konstante ist.[1]

Die Idee für den Beweis ist ungefähr die folgende. Für q groß ist die Wahrscheinlichkeit dafür, dass ein normiertes Polynom vom Grad n irreduzibel ist, nahe bei $1/n$. Daraus folgt, dass $P(n) = \lim_{q\to\infty} P(q,n)$ der Koeffizient von z^n in der Potenzreihe

$$F(z) = \prod_{m=1}^{\infty}\Big(1 + \frac{z^m}{m}\Big)$$

ist. Unter Verwendung von $-\log(1-z) = \sum_{m=1}^{\infty} \frac{z^m}{m}$ kann man das schreiben als

$$F(z) = \frac{1}{1-z}\prod_{m=1}^{\infty}\Big(1 + \frac{z^m}{m}\Big)e^{-z^m/m} =: \frac{1}{1-z}G(z)\,,$$

wobei die Funktion G auf der abgeschlossenen Einheitskreisscheibe in $\mathbb{C}$ hinreichend schöne Eigenschaften hat (inbesondere ist sie stetig). Die Aussage folgt dann im Wesentlichen daraus, dass

$$G(1) = \prod_{m=1}^{\infty}\frac{m+1}{m}e^{-1/m} = \lim_{N\to\infty} N e^{-\sum_{m=1}^{N-1} 1/m} = \lim_{N\to\infty}\exp\Big(\log N - \sum_{m=1}^{N}\frac{1}{m}\Big) = e^{-\gamma}$$

ist, denn das bedeutet, dass $F(z)$ sich ähnlich wie $e^{-\gamma}/(1-z)$ verhält. Die Details sind allerdings etwas komplizierter.

[1] P. Flajolet, X. Gourdon, D. Panario: *The complete analysis of a polynomial factorization algorithm over finite fields,* J. Algorithms **40** (2001), no. 1, 37–81.

10. Primzahltests

In diesem Abschnitt wollen wir folgendes Problem (effizient) lösen:

Stelle fest, ob eine gegebene natürliche Zahl N eine Primzahl ist!

Dieses Problem ist durchaus von praktischer Relevanz, da viele moderne kryptographische Verfahren wie zum Beispiel RSA große Primzahlen (mit mehreren Hundert Dezimalstellen) benötigen.

Man wird sich vielleicht zuerst an die Definition einer Primzahl erinnern als eine Zahl, die (> 1 ist und) außer 1 und sich selbst keine Teiler hat. Das ergibt den folgenden Algorithmus:

ALGO
Probedivision

---

function isprime(N)

input: $N \in \mathbb{Z}_{\geq 2}$

output: **true**, wenn N Primzahl ist, sonst **false**.

```
  for d = 2 to ⌊√N⌋ do
    if N rem d = 0 then return false end if
  end for
  return true
end function
```

Die Schranke $\lfloor\sqrt{N}\rfloor$ kommt daher, dass für jeden Teiler d von N auch N/d ein Teiler ist; wenn es also einen nichttrivialen Teiler gibt, dann gibt es auch einen, der $\leq \sqrt{N}$ ist.

Der Aufwand dafür beträgt im Fall, dass N tatsächlich prim ist (dann wird die Schleife komplett durchlaufen) $\sqrt{N}$ Divisionen von Zahlen der Länge $\lambda(N)$, also grob $\tilde{O}(\sqrt{N})$ Wortoperationen. Das ist keine polynomiale Komplexität, denn die Größe der Eingabe ist $\lambda(N) = \Theta(\log N)$ und $\sqrt{N} = e^{(\log N)/2}$ ist exponentiell in $\log N$.

Wie kann man es schneller hinbekommen? Da es bisher keinen Algorithmus gibt, der Zahlen in Polynomzeit faktorisieren kann (siehe nächsten Abschnitt), kann der Ansatz, die Nicht-Existenz eines nichttrivialen Teilers nachzuweisen, nicht zum Ziel führen. Stattdessen hilft es, sich zu überlegen, wie man zeigen kann, dass eine Zahl *nicht* prim ist, ohne sie zu faktorisieren. Dafür kann man Aussagen verwenden, die für Primzahlen gelten, aber für zusammengesetzte Zahlen im Allgemeinen nicht. Eine solche Aussage ist der kleine Satz von Fermat:

10.1. **Satz.** *Seien p eine Primzahl und $a \in \mathbb{Z}$ mit $p \nmid a$. Dann ist $a^{p-1} \equiv 1 \bmod p$.*

SATZ
kleiner Satz von Fermat

Im Umkehrschluss gilt dann: Ist $N \in \mathbb{Z}_{\geq 2}$ und $a \in \mathbb{Z}$ mit $0 < a < N$ und $a^{N-1} \not\equiv 1 \bmod N$, dann kann N keine Primzahl sein.

Euler hat diesen Satz verallgemeinert:

10.2. **Satz.** *Sei $N \in \mathbb{Z}_{>0}$ und sei $a \in \mathbb{Z}$ mit $a \perp N$. Dann ist $a^{\varphi(N)} \equiv 1 \bmod N$.*

SATZ
Satz von Euler

Hierbei ist $\varphi(N) = \#(\mathbb{Z}/N\mathbb{Z})^\times = \#\{1 \leq m \leq N \mid m \perp N\}$ die *Eulersche φ-Funktion*.

DEF
Euler-φ

Beweis. Die Aussage ist äquivalent zu der Gleichung $\bar{a}^{\varphi(N)} = 1$ in der Einheitengruppe $(\mathbb{Z}/N\mathbb{Z})^\times$ von $\mathbb{Z}/N\mathbb{Z}$. Sie folgt aus dem Satz von Lagrange aus der Gruppentheorie, der impliziert, dass die Ordnung eines Elements einer endlichen Gruppe stets die Gruppenordnung teilt. ❑

Wir erinnern uns daran, dass wir Reste von Potenzen der Form a^e rem N effizient berechnen können, nämlich mit einem Aufwand von $O(\log e)$ Operationen in $\mathbb{Z}/N\mathbb{Z}$, also $\tilde{O}\big((\log e)(\log N)\big)$ Wortoperationen; siehe Lemma 4.1. Dort wurde die Funktion modpower(N, a, e) definiert, die a^e rem N mit der angegebenen Komplexität berechnet.

Wir können also die Relation $a^{N-1} \equiv 1 \bmod N$, die zu a^{N-1} rem $N = 1$ äquivalent ist, mit einem Aufwand von $\tilde{O}((\log N)^2)$ Wortoperationen testen.

Der kleine Satz von Fermat ergibt dann den „Fermat-Test“:

ALGO Fermat-Test

function fermat(N, m)
input: $N \in \mathbb{Z}_{\geq 3}$, $m \geq 1$: Anzahl der Tests.
output: „zusammengesetzt“ oder „möglicherweise prim“.

```
  for i = 1 to m do
    a ← random(2, N − 1)
    if modpower(N, a, N − 1) ≠ 1 then
      return „zusammengesetzt“
    end if
  end for
  return „möglicherweise prim“
end function
```

Hierbei soll random(A, B) für ganze Zahlen $A \leq B$ eine Zufallszahl aus $[A, B] \cap \mathbb{Z}$ liefern. Der Parameter m gibt an, für wie viele Werte von a die Aussage des kleinen Fermatschen Satzes getestet wird.

Wenn wir annehmen, dass der Aufwand für die Bestimmung einer Zufallszahl von der Größenordnung $O(\log N)$ ist („Zufallszahlen“ werden im Computer durch Pseudo-Zufallszahlen-Generatoren erzeugt, die bei jedem Aufruf eine gewisse Folge von Operationen auf Zahlen einer festen Länge ausführen; damit bekommt man eine feste Zahl von zufälligen Bits in konstanter Zeit, also $\lambda(N)$ zufällige Worte in $O(\lambda(N))$ Wortoperationen), dann ist der Aufwand für den Fermat-Test $\tilde{O}(m(\log N)^2)$ Wortoperationen.

Jetzt stellt sich die Frage, ob dieser Test auch *zuverlässig* ist: Falls N zusammengesetzt ist, wird der Test das für geeignetes a auch nachweisen? Und ist der Anteil der geeigneten a unabhängig von N durch eine positive Konstante nach unten beschränkt? (Dann kann man m so wählen, dass der Test mit einer Wahrscheinlichkeit beliebig nahe bei 1 feststellt, dass N zusammengesetzt ist.)

Dazu geben wir den „schlechten“ Zahlen N erst einmal eine Bezeichnung.

* 10.3. **Definition.** Sei $a \in \mathbb{Z}$. Eine ganze Zahl $N \geq 2$ heißt *Pseudoprimzahl* zur Basis a, wenn N nicht prim ist, aber $a^{N-1} \equiv 1 \bmod N$ gilt.

DEF Pseudoprimzahl

N heißt *Carmichael-Zahl*, wenn N Pseudoprimzahl zur Basis a ist für alle zu N teilerfremden $a \in \mathbb{Z}$. ♢

Carmichael-Zahl

Jede zusammengesetzte Zahl ist Pseudoprimzahl zur Basis 1, und jede ungerade zusammengesetzte Zahl ist Pseudoprimzahl zur Basis -1. Es sind also nur Zahlen a mit $|a| \geq 2$ interessant.

BSP Pseudoprimzahlen

10.4. **Beispiele.** Es gibt Pseudoprimzahlen zu Basen $a \neq \pm 1$. Die kleinste Pseudoprimzahl zur Basis 2 ist zum Beispiel $N = 341 = 11 \cdot 31$. Die kleinste Pseudoprimzahl zur Basis 3 ist $N = 91 = 7 \cdot 13$.

Es gibt tatsächlich auch Carmichael-Zahlen. Die kleinste ist $N = 561 = 3{\cdot}11{\cdot}17$. Es gibt sogar unendlich viele davon; das wurde von Alford, Granville und Pomerance 1994 bewiesen.[2] ♣

Wir formulieren noch eine Charakterisierung von Carmichael-Zahlen, die wir später brauchen werden.

SATZ Charakterisierung von Carmichael-Zahlen

10.5. **Satz.** *Eine zusammengesetzte Zahl N ist genau dann eine Carmichael-Zahl, wenn N quadratfrei ist und für jeden Primteiler p von N die Relation $p-1 \mid N-1$ gilt.*

Eine Carmichael-Zahl ist stets ungerade und hat mindestens drei Primfaktoren.

Beweis. Übung. ❑

Die Existenz von Carmichael-Zahlen zeigt, dass der Fermat-Test noch nicht befriedigend ist, weil er Carmichael-Zahlen nicht (oder nur mit verschwindend kleiner Wahrscheinlichkeit) als zusammengesetzt erkennen kann. Man muss ihn also noch verfeinern. Dazu verwenden wir eine weitere Eigenschaft von Primzahlen:

LEMMA $x^2 \equiv 1 \bmod p$

10.6. **Lemma.** *Seien $p > 2$ prim und $e \geq 1$. Dann hat die Kongruenz*

$$x^2 \equiv 1 \bmod p^e$$

genau zwei Lösungen $0 \leq x < p^e$ in $\mathbb{Z}$.

Beweis. Es gibt stets die Lösungen $x = 1$ und $x = p^e - 1$; wegen $p^e > 2$ sind sie verschieden.

Umgekehrt folgt aus $p^e \mid x^2 - 1 = (x-1)(x+1)$ und $\operatorname{ggT}(x-1, x+1) \mid 2 \perp p$, dass p^e entweder $x-1$ oder $x+1$ teilen muss, woraus $x = 1$ oder $x = p^e - 1$ folgt. ❑

Wenn wir also eine Zahl a finden mit $a^2 \equiv 1 \bmod N$, aber $a \not\equiv \pm 1 \bmod N$, dann kann N nicht prim sein. Wir nutzen das wie folgt: Wir können annehmen, dass N ungerade ist (sonst ist entweder $N = 2$ oder N ist offensichtlich zusammengesetzt). Dann ist $N-1$ gerade, und wir schreiben $N - 1 = q2^e$ mit q ungerade und $e \geq 1$. Wir können a^{N-1} rem N berechnen, indem wir

$$b_0 = a^q \text{ rem } N\,, \quad b_1 = b_0^2 \text{ rem } N\,, \quad b_2 = b_1^2 \text{ rem } N\,, \quad \ldots\,, \quad b_e = b_{e-1}^2 \text{ rem } N$$

setzen; dann ist $b_e = a^{N-1}$ rem N. Das liefert uns einige Gelegenheiten, das Kriterium von Lemma 10.6 zu überprüfen: Wenn N den Fermat-Test besteht, dann ist $b_{e-1}^2 \equiv b_e = 1 \bmod N$. Falls also $b_{e-1} \not\equiv \pm 1 \bmod N$ ist, dann kann N nicht prim sein. Falls $b_{e-1} = 1$ ist (und $e \geq 2$), dann können wir b_{e-2} testen, und so weiter. Wir erhalten auf diese Weise den Miller-Rabin-Test:

[2] W.R. Alford, A. Granville, C. Pomerance: *There are infinitely many Carmichael numbers*, Ann. Math. **139** (1994), 703–722.

ALGO Miller-Rabin-Test

function MillerRabin(N, m)
input: $N \in \mathbb{Z}_{\geq 5}$ ungerade, $m \geq 1$: Anzahl der Tests.
output: „zusammengesetzt“ oder „wahrscheinlich prim“.

```
q ← N − 1; e ← 0
while q rem 2 = 0 do
  q ← q/2; e ← e + 1
end while    // jetzt ist N − 1 = 2^e q mit q ungerade
for i = 1 to m do
  a ← random(2, N − 2)
  b ← modpower(N, a, q)
  if b ≠ 1 then    // b = 1: Test OK, nächstes a
    j ← 1
    while b ≠ N − 1 do    // b ≡ −1: Test OK, nächstes a
      if j = e then return „zusammengesetzt“ end if
      b ← b^2 rem N
      if b = 1 then return „zusammengesetzt“ end if
      j ← j + 1
    end while
  end if
end for
return „wahrscheinlich prim“
end function
```

Wenn $a^q \equiv 1 \bmod N$ ist, dann sind alle $b_j = 1$, und es gibt keinen Widerspruch dazu, dass N prim sein könnte. Wenn $b_j = N-1$ ist für ein $0 \leq j < e$, dann gilt $b_{j+1} = \ldots = b_e = 1$; das liefert ebenfalls keinen Widerspruch. Ist $b_j = 1$ für $j > 0$ mit j minimal und ist $b_{j-1} \neq N-1$, dann sagt Lemma 10.6, dass N zusammengesetzt sein muss. Wenn wir b_e berechnen müssten und $b_{e-1} \neq 1, N-1$ ist, dann kann N ebenfalls keine Primzahl sein, denn entweder ist $b_e \neq 1$, womit N den Fermat-Test nicht besteht, oder $b_e = 1$ ist das Quadrat modulo N einer Zahl, die $\not\equiv \pm 1$ ist, womit das Kriterium aus Lemma 10.6 zieht.

Der Aufwand für den Miller-Rabin-Test entspricht dem für den Fermat-Test (oder ist sogar geringer, da eventuell ein Teil der Rechnung übersprungen wird), da im Wesentlichen nur die Potenz $a^{N-1} \bmod N$ berechnet wird.

Der große Vorteil des Miller-Rabin-Tests gegenüber dem Fermat-Test ist seine Zuverlässigkeit.

* 10.7. **Satz.** *Sei $N \in \mathbb{Z}_{\geq 15}$ ungerade und zusammengesetzt. Dann gibt es höchstens $(N-11)/4$ Zahlen $2 \leq a \leq N-2$, sodass N den Miller-Rabin-Test mit der Basis a besteht (also nicht als zusammengesetzt erkannt wird).*

SATZ M-R-Test ist zuverlässig

Beweis. Sei

$$S = \{\bar{a} \in \mathbb{Z}/N\mathbb{Z} \mid N \text{ besteht den Test mit Basis } a\}$$

die Menge der „schlechten“ Restklassen. Da eine Restklasse x, die nicht invertierbar ist, niemals die Gleichung $x^{N-1} = 1$ erfüllen kann, ist $S \subset (\mathbb{Z}/N\mathbb{Z})^\times$. Die Idee des Beweises ist es nun, eine Untergruppe M von $(\mathbb{Z}/N\mathbb{Z})^\times$ zu finden, sodass

$S \subset M$ ist und M Index mindestens 4 in $(\mathbb{Z}/N\mathbb{Z})^\times$ hat. Da die Restklassen $\bar{1}$ und $\overline{N-1}$ stets in S sind, folgt dann

$$\begin{aligned}\#\{2 \le a \le N-2 \mid \bar{a} \in S\} &= \#S - 2 \le \#M - 2 \\ &\le \frac{\varphi(N)}{4} - 2 \le \frac{N-3}{4} - 2 = \frac{N-11}{4}\,,\end{aligned}$$

wobei wir verwendet haben, dass $\varphi(N)$ für ungerades N stets gerade und für zusammengesetztes N stets $< N-1$ ist.

Sei $N - 1 = q2^e$ wie im Miller-Rabin-Test. Wir setzen, für $0 \le j \le e$,

$$M'_j = \{x \in (\mathbb{Z}/N\mathbb{Z})^\times \mid x^{q2^j} = 1\}$$

und für $1 \le j \le e$

$$M_j = \{x \in (\mathbb{Z}/N\mathbb{Z})^\times \mid x^{q2^{j-1}} = \pm 1\}\,.$$

Es gilt

$$M_1 \subset M'_1 \subset M_2 \subset M'_2 \subset \ldots \subset M_e \subset M'_e \subset (\mathbb{Z}/N\mathbb{Z})^\times\,.$$

Alle M_j und M'_j sind Untergruppen von $(\mathbb{Z}/N\mathbb{Z})^\times$, und es ist

$$S = M'_0 \cup (M_1 \setminus M'_0) \cup (M_2 \setminus M'_1) \cup \ldots \cup (M_e \setminus M'_{e-1})\,,$$

denn $\bar{a} \in S$ bedeutet $\bar{a}^q = 1$ oder $\bar{a}^{q2^j} = -1$ für ein $j < e$.

Sei $j_0 \ge 1$ maximal mit $M'_{j_0-1} \ne M_{j_0}$ (das gilt jedenfalls für $j_0 = 1$, denn $-1 \in M_1 \setminus M'_0$, also gibt es so ein maximales j_0). Wir setzen $M = M_{j_0}$; dann gilt offenbar $S \subset M$. Sei $N = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$ die Primfaktorzerlegung von N. Nach dem Chinesischen Restsatz ist der kanonische Homomorphismus

$$\psi\colon (\mathbb{Z}/N\mathbb{Z})^\times \longrightarrow \prod_{j=1}^{k} (\mathbb{Z}/p_j^{e_j}\mathbb{Z})^\times$$

ein Isomorphismus. Wir zeigen, dass $(M'_e : M) \ge (M'_{j_0} : M) = 2^{k-1}$ ist. Dazu sei

$$\psi_{j_0}\colon M'_{j_0} \longrightarrow \{\pm 1\}^k\,, \qquad x \longmapsto \psi(x^{q2^{j_0-1}})\,.$$

Dass das Bild von ψ_{j_0} in $\{\pm 1\}^k \subset \prod_{j=1}^k (\mathbb{Z}/p_j^{e_j}\mathbb{Z})^\times$ enthalten ist, folgt aus Lemma 10.6. Da $M \ne M'_{j_0-1}$ ist, gibt es $x \in M \subset M'_{j_0}$ mit $\psi_{j_0}(x) = (-1, \ldots, -1)$. Wir schreiben $\psi(x) = (x^{(1)}, \ldots, x^{(k)})$. Für $1 \le i \le k$ sei $x_i \in (\mathbb{Z}/N\mathbb{Z})^\times$ das Element mit $\psi(x_i) = (1, \ldots, 1, x^{(i)}, 1, \ldots, 1)$, wobei $x^{(i)}$ an der i-ten Stelle steht. Dann ist $x_i \in M'_{j_0}$ und $\psi_{j_0}(x_i) = (1, \ldots, 1, -1, 1, \ldots, 1)$. Es folgt, dass ψ_{j_0} surjektiv ist. $M = M_{j_0}$ ist genau das Urbild von $\{(1, \ldots, 1), (-1, \ldots, -1)\}$ unter ψ_{j_0}; da diese Untergruppe Index 2^{k-1} in $\{\pm 1\}^k$ hat, folgt die Behauptung.

Jetzt unterscheiden wir drei Fälle.

(1) $k = 1$. Dann ist $N = p^m$ eine Primzahlpotenz. Es gilt

$$S \subset M'_e = \{x \in (\mathbb{Z}/p^m\mathbb{Z})^\times \mid x^{p^m - 1} = 1\}\,.$$

Nach dem Satz 10.2 von Euler gilt aber für jedes $x \in (\mathbb{Z}/p^m\mathbb{Z})^\times$ auch, dass $x^{(p-1)p^{m-1}} = 1$ ist (denn $\varphi(p^m) = (p-1)p^{m-1}$). Es ist $\mathrm{ggT}(p^m - 1, (p-1)p^{m-1}) = p - 1$, also gilt für $x \in M'_e$ sogar $x^{p-1} = 1$. Aus Lemma 7.3 folgt aber, dass es in $\mathbb{Z}/p^m\mathbb{Z}$ nur $p - 1$ solche Elemente gibt. Damit ist $\#S \le \#M'_e \le p - 1$ und der Index von M'_e ist $\varphi(p^m)/(p-1) = p^{m-1} \ge 5$ (denn $p \ge 5$ und $m \ge 2$ oder $p = 3$ und $m \ge 3$).

(2) $k \geq 2$ und N ist keine Carmichael-Zahl. Letzteres bedeutet $M'_e \neq (\mathbb{Z}/N\mathbb{Z})^\times$, also $((\mathbb{Z}/N\mathbb{Z})^\times : M'_e) \geq 2$. Zusammen mit der obigen Behauptung erhalten wir

$$((\mathbb{Z}/N\mathbb{Z})^\times : M) = ((\mathbb{Z}/N\mathbb{Z})^\times : M'_e)(M'_e : M) \geq 2 \cdot 2^{k-1} = 2^k \geq 4\,.$$

(3) N ist eine Carmichael-Zahl. Nach Satz 10.5 folgt $k \geq 3$ und mit der obigen Behauptung dann $((\mathbb{Z}/N\mathbb{Z})^\times : M) = (M'_e : M) \geq 2^{k-1} \geq 4$.

Es gilt also in allen Fällen, dass der Index von M mindestens 4 ist. ❑

Für die kleinste ungerade zusammengesetzte Zahl $N = 9$ kann man direkt nachprüfen, dass keine Zahl $2 \leq a \leq 7$ den Miller-Rabin-Test besteht.

Man kann den Satz so interpretieren, dass bei zufälliger Wahl von a eine zusammengesetzte Zahl N mit Wahrscheinlichkeit $> 3/4$ auch als zusammengesetzt erkannt wird. (Für die allermeisten Zahlen ist der Index deutlich größer als 4 und damit die Wahrscheinlichkeit noch deutlich höher.) Wenn wir also den Miller-Rabin-Test mit m unabhängig voneinander zufällig gewählten Zahlen a durchführen, wird N mit einer Wahrscheinlichkeit von mindestens $1 - (1/4)^m$ als zusammengesetzt erkannt. Für $m = 20$, was ein üblicher Wert ist, ist $(1/4)^m = 2^{-40} < 10^{-12}$. Eine Zahl p, die diesen Test bestanden hat, ist also sehr wahrscheinlich prim. Wie wahrscheinlich? Die Dichte von Primzahlen in der Nähe von N ist nach dem Primzahlsatz etwa $1/\log N$. Da wir nur ungerade Zahlen betrachten (und die geraden alle zusammengesetzt sind), erhöht sich die Dichte auf $2/\log N$. Nach dem Satz von Bayes aus der elementaren Stochastik ergibt sich dann die bedingte Wahrscheinlichkeit dafür, dass eine Zahl N, die den Test überstanden hat, tatsächlich prim ist, zu $\geq 1 - \frac{1}{2}10^{-12}\log N$. Für $N \approx 10^{1000}$ ist $\log N \approx 1000 \log 10 \approx 2300$; das ergibt als grobe Schätzung eine Wahrscheinlichkeit von $\leq 10^{-9}$ dafür, dass eine tausendstellige „Miller-Rabin-Primzahl“ N nicht prim ist. Auf meinem Laptop dauert ein Miller-Rabin-Test mit $m = 20$ und $N \approx 10^{1000}$ (sehr wahrscheinlich) prim (sodass der Test komplett durchlaufen wird) etwa 0,32 Sekunden.

Als Mathematiker möchte man sich aber vielleicht nicht mit einer sehr hohen Wahrscheinlichkeit zufrieden geben, sondern mit Sicherheit wissen wollen, ob N prim ist oder nicht. Wir brauchen also noch eine effiziente Möglichkeit, von einer Zahl N, von der wir stark vermuten, dass sie prim ist (etwa weil sie den Miller-Rabin-Test bestanden hat), auch zu *beweisen*, dass sie prim ist. Eine Möglichkeit dafür basiert auf einer Art „Umkehrung“ des kleinen Satzes von Fermat. Dazu formulieren wir erst einmal eine Hilfsaussage.

LEMMA
Hilfsaussage für PL-Test

10.8. **Lemma.** *Seien $N > 0$ eine ganze Zahl und p ein Primteiler von $N - 1$. Sei weiter $a_p \in \mathbb{Z}$ mit*

$$a_p^{N-1} \equiv 1 \bmod N \qquad \textit{und} \qquad (a_p^{(N-1)/p} - 1) \perp N\,. \tag{10.1}$$

Sei außerdem p^{e_p} die höchste Potenz von p, die $N - 1$ teilt. Dann gilt für jeden (positiven) Teiler d von N, dass $d \equiv 1 \bmod p^{e_p}$ ist.

Beweis. Wir können uns auf Primteiler d beschränken. Da $a_p \perp N$, also auch $a_p \perp d$, folgt $a_p^{d-1} \equiv 1 \bmod d$. Andererseits ist $a_p^{(N-1)/p} \not\equiv 1 \bmod d$, da nach Voraussetzung $(a_p^{(N-1)/p} - 1) \perp N$ ist. Sei n die Ordnung von a_p mod d; dann folgt $n \mid d - 1$, $n \mid N - 1$ (denn $a_p^{N-1} \equiv 1 \bmod d$), aber $n \nmid (N-1)/p$. Aus den letzten beiden Eigenschaften folgt $p^{e_p} \mid n$, aus der ersten dann $p^{e_p} \mid d - 1$. ❑

Wenn wir über die Faktorisierung von $N - 1$ gut genug Bescheid wissen, können wir dieses Ergebnis nutzen, um zu beweisen, dass N prim ist.

FOLG Kriterium für Primzahl

10.9. **Folgerung.** *Sei $N > 0$ eine ganze Zahl, sei $N - 1 = F \cdot U$ mit $F \geq \sqrt{N}$, und alle Primteiler von F seien bekannt.*

N ist genau dann prim, wenn es für jeden Primteiler p von F eine Zahl $a_p \in \mathbb{Z}$ gibt, die (10.1) erfüllt.

Beweis. Sei zunächst N prim und sei $g \in \mathbb{Z}$ eine Primitivwurzel mod N, d.h., das Bild von g erzeugt die zyklische Gruppe $(\mathbb{Z}/N\mathbb{Z})^\times$. Dann hat $a_p = g$ die Eigenschaft (10.1).

Seien nun umgekehrt für alle $p \mid F$ Zahlen a_p mit (10.1) gegeben. Aus Lemma 10.8 folgt dann, dass jeder Teiler d von N die Kongruenz $d \equiv 1 \bmod F$ erfüllt. Insbesondere ist $d = 1$ oder $d > F \geq \sqrt{N}$. Wenn N zusammengesetzt wäre, hätte N einen nichttrivialen Teiler $\leq \sqrt{N}$, was wir gerade ausgeschlossen haben, also ist N prim. ❑

D.H. Lehmer
1905 – 1991

Aus diesem Ergebnis lässt sich direkt ein Primzahltest ableiten, der *Pocklington-Lehmer*[3]*-Test*. Er kann nachweisen, dass eine Zahl prim ist, aber nicht (mit vertretbarem Aufwand: Man könnte im Prinzip alle Möglichkeiten für die a_p in Folgerung 10.9 testen, aber das würde viel zu lange dauern), dass eine Zahl N zusammengesetzt ist. Man wird also erst den Miller-Rabin-Test verwenden, um ausreichend sicher zu sein, dass N prim ist, und dann den Pocklington-Lehmer-Test anwenden, um dies endgültig nachzuweisen.

Der Test basiert auf der Verwendung der zyklischen Gruppe $(\mathbb{Z}/N\mathbb{Z})^\times$ der Ordnung $N - 1$ (wenn N prim ist). Sein Nachteil ist, dass er eine gute Kenntnis der Faktorisierung von $N - 1$ erfordert, was in der Praxis ein großes Hindernis sein kann. Für Zahlen spezieller Form lässt der Test sich aber gut verwenden.

BSP Fermat-Zahlen

10.10. **Beispiel.** Eine Zahl der Form $F_n = 2^{2^n} + 1$ heißt *Fermat-Zahl*. Fermat hatte behauptet, dass alle Fermat-Zahlen Primzahlen sind. (Es ist leicht zu sehen, dass $2^m + 1$ nicht prim sein kann, wenn m keine Zweierpotenz ist.) Tatsächlich sind die Zahlen

$$F_0 = 3, \quad F_1 = 5, \quad F_2 = 17, \quad F_3 = 257, \quad F_4 = 65537$$

alle prim, aber Euler hat gezeigt, dass F_5 durch 641 teilbar und damit nicht prim ist. Tatsächlich ist bis heute keine weitere Fermatsche Primzahl bekannt, aber von $F_5, F_6, \ldots, F_{32}$ (und etlichen weiteren) ist bekannt, dass sie nicht prim sind.

Für Fermat-Zahlen lässt sich der Pocklington-Lehmer-Test noch etwas vereinfachen (diese Version heißt auch Pépin-Test):

Für $n \geq 1$ ist F_n genau dann prim, wenn $3^{2^{2^n-1}} \equiv -1 \bmod F_n$ ist.

Ist F_n prim, dann ist 3 ein quadratischer Nichtrest mod F_n; das folgt mit dem Quadratischen Reziprozitätsgesetz daraus, dass $F_n = 4^{2^{n-1}} + 1 \equiv 2 \bmod 3$ ist. Die Aussage ergibt sich dann aus dem Euler-Kriterium. Umgekehrt ist $a_2 = 3$ eine passende Zahl für den Pocklington-Lehmer-Test (und 2 ist der einzige Primteiler von $F_n - 1$).

[3]Bild: Oberwolfach Photo Collection, Copyright: George M. Bergman, Berkeley

Zum Beispiel erhalten wir für $n = 4$ (mit $2^n - 1 = 15$):

m	0	1	2	3	4	5	6	7
$3^{2^m} \bmod F_4$	3	9	81	6561	−11088	−3668	19139	15028
m	8	9	10	11	12	13	14	15
$3^{2^m} \bmod F_4$	282	13987	8224	−8	64	4096	−256	−1

♣

Man kann diesen Ansatz variieren, indem man statt $\mathbb{F}_N^\times$ die Untergruppe der Ordnung $N+1$ von $\mathbb{F}_{N^2}^\times$ benutzt. Dann braucht man Informationen über die Faktorisierung von $N+1$. Das führt zum Beispiel zum bekannten *Lucas-Lehmer-Test* für Mersennesche Primzahlen $2^p - 1$ (siehe unten). Eine weitere Alternative besteht darin, sogenannte „Elliptische Kurven“ zu verwenden. Sie stellen ebenfalls eine Gruppe der Größe etwa N zur Verfügung und können in ähnlicher Weise genutzt werden, haben dabei aber den großen Vorteil, dass viele verschiedene Kurven mit unterschiedlichen Gruppenordnungen zur Verfügung stehen, sodass man mit vernünftiger Wahrscheinlichkeit eine Kurve findet, deren Gruppenordnung gut faktorisierbar ist. Das führt auf den ECPP-Algorithmus („Elliptic curve primality proof“) von Goldwasser und Kilian, der in der Praxis in den meisten Fällen der zurzeit effizienteste Algorithmus für die Verifikation von Primzahlen ist.

Eine Diskussion von Primzahltests wäre nicht vollständig, ohne den deterministischen Polynomzeit-Algorithmus von Agrawal, Kayal und Saxena[4] zu erwähnen. Dieses Resultat löst ein altes Problem, denn bis dahin war kein Verfahren bekannt, das für eine beliebige natürliche Zahl *deterministisch* (d.h. ohne Zufallszahlen zu verwenden wie etwa der Miller-Rabin-Test oder auch ECPP) und in *polynomialer Laufzeit* feststellt, ob sie prim ist oder zusammengesetzt. Dieser Durchbruch hat sich aus einem Bachelorprojekt der beiden Studenten Kayal und Saxena entwickelt. Dieser Algorithmus ist allerdings bisher (auch nach einigen Verbesserungen) in der Praxis noch langsamer als Miller-Rabin plus ECPP.

Die zugrunde liegende Idee ist eine Verallgemeinerung des kleinen Satzes von Fermat auf Polynome, die zu einer Charakterisierung von Primzahlen führt: Für jede ganze Zahl $a \perp N$ gilt

$$N \text{ ist prim} \iff (X-a)^N \equiv X^N - a \bmod N$$

im Polynomring $\mathbb{Z}[X]$ (d.h., die Kongruenz mod N gilt koeffizientenweise). Die Verifikation der Kongruenz ist allerdings viel zu aufwendig. Deshalb betrachtet man stattdessen die Kongruenz

$$(X-a)^N \equiv X^N - a \bmod \langle N, X^r - 1\rangle$$

für geeignete $r \geq 1$. Die drei Autoren konnten zeigen, dass die Gültigkeit der Kongruenz für r und a wie im folgenden Algorithmus hinreichend dafür ist, dass N eine Primzahlpotenz ist.

ALGO
AKS-Primzahltest

function AKS(N)
input: $N > 1$.
output: **false**, wenn N zusammengesetzt ist; **true**, wenn N prim ist.

```
if N ist eine echte Potenz then return false end if
r ← 1; while ord_r(N) ≤ (log_2 N)^2 do r ← r + 1 end while
for a = 1 to r do
  if 1 < ggT(a, N) < N then return false end if
```

[4] Manindra Agrawal, Neeraj Kayal, Nitin Saxena. *PRIMES is in P,* Annals of Mathematics **160** (2004), no. 2, 781–793.

```
  end for
  if N ≤ r then return true end if
  for a = 1 to ⌊√φ(r) log₂ N⌋ do
    if (X − a)^N ≢ X^N − a mod (N, X^r − 1) then return false end if
  end for
  return true
end function
```

Hier bezeichnet $\mathrm{ord}_r(N)$ die Ordnung von N in der multiplikativen Gruppe $(\mathbb{Z}/r\mathbb{Z})^\times$.

Außerdem konnten sie zeigen, dass die Zahl r genügend klein ist, damit die Laufzeit durch ein Polynom in $\log N$ beschränkt werden kann (ursprünglich $\tilde{O}((\log N)^{12})$; diese Abschätzung wurde zwischenzeitlich aber verbessert).

F.É.A. Lucas
1842 – 1891

Zum Abschluss möchte ich noch erklären, wie man *Mersenne-Zahlen*, das sind Zahlen der Form $M_p = 2^p - 1$ mit p prim, auf Primalität testen kann. (Ist der Exponent n von 2 keine Primzahl, dann kann $2^n - 1$ nicht prim sein.) Dazu verwendet man, wie oben angedeutet, eine Variante des Pocklington-Lehmer-Tests, die mit einer Gruppe der Ordnung $N + 1$ arbeitet. (Sei N prim, dann kann man den Körper $\mathbb{F}_{N^2}$ betrachten. Seine multiplikative Gruppe $\mathbb{F}_{N^2}^\times$ ist zyklisch und hat die Ordnung $N^2-1 = (N-1)(N+1)$, hat also eine ebenfalls zyklische Untergruppe der Ordnung $N+1$.) Sein Spezialfall für Mersenne-Zahlen ist der Lucas-Lehmer-Test, der auf folgender Aussage beruht:

LEMMA Lucas-Lehmer-Test

10.11. **Lemma.** *Sei $p \geq 3$ prim. Wir definieren die Folge (S_n) durch $S_0 = 4$ und $S_{n+1} = S_n^2 - 2$. $M_p = 2^p - 1$ ist genau dann eine Primzahl, wenn S_{p-2} durch M_p teilbar ist.*

Beweis. Man zeigt zuerst durch Induktion, dass

$$(2+\sqrt{3})^{2^m} + (2-\sqrt{3})^{2^m} = S_m \qquad \text{und} \qquad (2+\sqrt{3})^{2^m}(2-\sqrt{3})^{2^m} = 1$$

gilt. Aus $S_{p-2} \equiv 0 \bmod M_p$ folgt $S_{p-1} \equiv -2 \bmod M_p$ und umgekehrt, und die Kongruenz bedeutet gerade, dass $[2+\sqrt{3}]^{2^{p-1}} = -1$ ist in $\mathbb{Z}/M_p\mathbb{Z}[\sqrt{3}]$, wobei $R[\sqrt{3}]$ für $R[x]/\langle x^2-3\rangle$ steht.

Ist M_p prim, dann ist 3 (wegen $M_p \equiv 7 \bmod 12$; hier benutzen wir $p \geq 3$) ein quadratischer Nichtrest mod M_p und daher $\mathbb{Z}/M_p\mathbb{Z}[\sqrt{3}] \cong \mathbb{F}_{M_p^2}$. Man kann zeigen, dass $[2+\sqrt{3}]$ zwar ein Quadrat in $\mathbb{F}_{M_p^2}$ ist, aber kein Quadrat eines Elements in der Untergruppe der Ordnung $M_p + 1$. Damit hat $[2+\sqrt{3}] \in \mathbb{F}_{M_p^2}^\times$ Ordnung $M_p + 1 = 2^p$; also muss $[2+\sqrt{3}]^{2^{p-1}} = -1$ sein, und das Kriterium ist erfüllt.

Sei umgekehrt das Kriterium erfüllt. M_p hat (wieder wegen $M_p \equiv 7 \bmod 12$) einen Primteiler $q \equiv 5$ oder $7 \bmod 12$, sodass 3 ein quadratischer Nichtrest mod q ist. Das Kriterium zeigt, dass $[2+\sqrt{3}] \in \mathbb{F}_{q^2}^\times$ Ordnung 2^p hat (denn $[2+\sqrt{3}]^{2^{p-1}} = -1$). Da $[2+\sqrt{3}]$ in einer Untergruppe der Ordnung $q+1$ liegt, folgt $q+1 \geq 2^p$, also $q \geq M_p$ und damit ist $M_p = q$ eine Primzahl. ❑

Wenn man den Test implementiert, dann berechnet man die Folge (S_n) modulo M_p, setzt also $S_{n+1} = (S_n^2 - 2) \operatorname{rem} M_p$, und testet am Ende, ob $S_{p-2} = 0$ ist.

Der Aufwand für den Test beträgt im Wesentlichen $p - 1$ Quadrierungen modulo M_p, das sind $\tilde{O}(p^2) = \tilde{O}(\lambda(M_p)^2)$ Wortoperationen. Damit ist dieser Test deutlich schneller als jeder allgemein anwendbare Primzahltest.

10.12. **Beispiele.** Ist $M_{11} = 2^{11} - 1 = 2047$ prim? Wir machen den Test: **BSP** LL-Test

m	0	1	2	3	4	5	6	7	8	9
$S_m \bmod M_{11}$	4	14	194	788	701	119	−170	240	282	−311

Da S_9 nicht durch M_{11} teilbar ist, ist M_{11} nicht prim (tatsächlich hat man die Faktorisierung $M_{11} = 23 \cdot 89$).

Ist $M_{13} = 2^{13} - 1 = 8191$ prim? Wir machen den Test:

m	0	1	2	3	4	5	6	7	8	9	10	11
$S_m \bmod M_{13}$	4	14	194	−3321	3953	−2221	1857	36	1294	3470	128	0

Da S_{11} durch M_{13} teilbar ist, ist M_{13} prim. ♣

Da der Lucas-Lehmer-Test so effizient ist, kann man mit ihm weit größere Zahlen testen als mit allgemeineren Tests. Das hat dazu geführt, dass die jeweils größte bekannte Primzahl fast immer (und immer seit 1992, siehe die Wikipedia-Seite zu Primzahlen) eine Mersenne-Primzahl war. Der aktuelle Rekord (Stand Januar 2018) ist $p = 2^{77\,232\,917} - 1$, eine Zahl mit über 23 Millionen Ziffern. Die nötigen Berechnungen werden vom Great Internet Mersenne Prime Search (GIMPS) verteilt im Internet organisiert. Wenn man so einen Test effizient durchführen will, sind schnelle Multiplikation und Division mit Rest essenziell!

11. Faktorisierung von ganzen Zahlen

Zum Abschluss dieser Vorlesung wollen wir noch folgendes Problem betrachten:

Gegeben sei eine zusammengesetzte natürliche Zahl N. Finde einen nichttrivialen Teiler d von N!

Da wir effizient feststellen können, ob eine gegebene natürliche Zahl prim ist oder nicht, ist die Lösung dieses Problems dazu äquivalent, die volle Primfaktorzerlegung von $N > 1$ zu finden: Entweder ist N prim, dann kennen wir die Primfaktorzerlegung, oder zusammengesetzt, dann finden wir einen nichttrivialen Teiler d von N und wenden das Verfahren rekursiv auf d und N/d an.

Wie wir bereits gesehen haben, benötigt der naive Algorithmus der *Probedivision*, bei dem man N nacheinander durch $2, 3, 4, \dots$ teilt, bis man einen Teiler gefunden hat, im schlechtesten Fall (der hier für $N = p^2$ mit p prim oder $N = pq$ mit $p \approx q$ und p, q prim eintritt) $\tilde{O}(\sqrt{N})$ Wortoperationen, was exponentiell in der Länge $\lambda(N) \asymp \log N$ ist. Selbst wenn man das Verfahren beschleunigt, indem man nur durch Primzahlen teilt, wird es nicht wesentlich schneller (es gibt nach dem Primzahlsatz etwa $2\sqrt{N}/\log N$ Primzahlen $\leq \sqrt{N}$; man gewinnt also nur einen logarithmischen Faktor). Wir werden zunächst Verfahren betrachten, deren Aufwand $\tilde{O}(\sqrt[4]{N})$ ist. Dieser Reduktion der Laufzeit auf im Wesentlichen die Quadratwurzel liegt das *Geburtstagsparadox* zugrunde: Schon bei einer zufälligen Ansammlung von 23 Leuten ist die Wahrscheinlichkeit, dass zwei davon denselben Geburtstag haben, größer als 50%. Das liegt daran, dass hier *Paare* von Personen relevant sind; davon gibt es $\binom{23}{2} = 253$, sodass die gesuchte Wahrscheinlichkeit etwa $1 - (1 - \frac{1}{365})^{253} > 0{,}5$ ist. (Tatsächlich ist sie sogar etwas größer, da die Paare nicht völlig unabhängig voneinander sind). Die allgemeine Strategie, die man daraus ableiten kann, besteht darin, die Menge von n Objekten, von denen man eines mit einer speziellen Eigenschaft finden will, in $\approx \sqrt{n}$ Teilmengen mit jeweils $\approx \sqrt{n}$ Elementen zu zerlegen und dann mit diesen Teilmengen zu arbeiten.

Im vorliegenden Fall wollen wir unter den Zahlen $2, 3, \dots, n = \lfloor \sqrt{N} \rfloor$ einen Teiler von N finden. Sei $c = \lceil \sqrt{n} \rceil$. Wir betrachten jeweils c aufeinanderfolgende Zahlen gemeinsam und testen für $k = 0, 1, \dots, c-1$, ob

$$\operatorname{ggT}\big(N, (kc+2)(kc+3)\cdots(kc+c+1)\big) > 1$$

ist. Haben wir so ein k gefunden, dann testen wir die Zahlen $kc+2, \dots, kc+c+1$ nacheinander einzeln, welche davon ein Teiler von N ist. Die Restklasse des Produkts $(kc+2)(kc+3)\cdots(kc+c+1)$ modulo N erhalten wir durch Auswerten des Polynoms

$$f(X) = (X + [2])(X + [3]) \dots (X + [c+1]) \in \mathbb{Z}/N\mathbb{Z}[X]$$

im Punkt $[kc]$. Das liefert folgenden Algorithmus von Pollard und Strassen:

ALGO Pollard-Strassen

function PollardStrassen(N)

input: $N \in \mathbb{Z}_{\geq 2}$ zusammengesetzt.

output: $1 < d \leq \sqrt{N}$, der kleinste Teiler > 1 von N.

$c \leftarrow \lceil \sqrt[4]{N} \rceil$
$f \leftarrow \prod_{j=2}^{c+1}(X + [j]) \in \mathbb{Z}/N\mathbb{Z}[X]$ // rekursive Berechnung wie in § 8
$([a_0], [a_1], \dots, [a_{c-1}]) \leftarrow (f([0]), f([c]), f([2c]), \dots, f([(c-1)c]))$
// parallele Auswertung nach Satz 8.1

```
  for k = 0 to c − 1 do
    if ggT(N, a_k) > 1 then
      for j = kc + 2 to kc + c + 1 do
        if N rem j = 0 then return j end if
      end for
    end if
  end for    // das Ende der Schleife wird nicht erreicht
end function
```

Der Aufwand für die Berechnung und für die parallele Auswertung von f ist $\tilde{O}(c)$ Operationen in $\mathbb{Z}/N\mathbb{Z}$ und damit auch $\tilde{O}(c)$ Wortoperationen. Die Berechnung eines größten gemeinsamen Teilers $\operatorname{ggT}(N, a_k)$ benötigt $\tilde{O}(\log N)$ Wortoperationen; das ist logarithmisch im Vergleich zu c. Damit ist der Gesamtaufwand für die ggTs ebenfalls $\tilde{O}(c)$ Wortoperationen, und für die Divisionen in der innersten Schleife gilt das Gleiche. Insgesamt ist der Aufwand also $\tilde{O}(c) = \tilde{O}(\sqrt[4]{N})$ Wortoperationen.

Der nötige Speicherplatz ist $\asymp \sqrt[4]{N}\lambda(N)$ Datenworte, denn es müssen die $c+1$ Koeffizienten von f und die c Ergebnisse der Auswertung (beides sind Elemente von $\mathbb{Z}/N\mathbb{Z}$) gespeichert werden. Das ist ziemlich viel und schränkt die praktische Verwendbarkeit des Verfahrens deutlich ein.

Eine andere Möglichkeit, das Geburtstagsparadox auszunutzen, beruht auf folgender Formulierung:

LEMMA Geburtstagsparadox

11.1. **Lemma.** *Wenn man aus einer Menge von n Objekten nacheinander und mit Zurücklegen zufällig und gleichverteilt Objekte auswählt, dann ist der Erwartungswert für die Anzahl der Auswahlen, bis man ein Objekt zweimal ausgewählt hat, von der Größenordnung $\asymp \sqrt{n}$.*

Beweis. Sei K die Zufallsvariable, die die Anzahl der Auswahlen bis zur ersten „Kollision“ angibt. Dann gilt

$$\begin{aligned}\mathbb{P}(K \geq m) &= \prod_{j=0}^{m-2}\Big(1-\frac{j}{n}\Big) = \exp\Big(\sum_{j=0}^{m-2}\log\Big(1-\frac{j}{n}\Big)\Big)\\ &\leq \exp\Big(-\sum_{j=0}^{m-2}\frac{j}{n}\Big) = \exp\Big(-\frac{(m-2)(m-1)}{2n}\Big) \leq e^{-(m-2)^2/2n}\end{aligned}$$

und damit

$$\mathbb{E}(K) = \sum_{m=1}^{\infty}\mathbb{P}(K\geq m) \leq 1 + \sum_{m=0}^{\infty} e^{-m^2/2n} \leq 2 + \int_0^{\infty} e^{-x^2/2n}\,dx = 2 + \sqrt{\frac{\pi n}{2}}\,.$$

Umgekehrt ist

$$\mathbb{P}(K \geq \lceil\sqrt{n}\rceil) \geq \Big(1-\frac{1}{\sqrt{n}}\Big)^{\sqrt{n}} = e^{-1}\big(1+o_n(1)\big)$$

und damit

$$\mathbb{E}(K) \geq \lceil\sqrt{n}\rceil\mathbb{P}(K\geq\lceil\sqrt{n}\rceil) \geq \frac{\sqrt{n}}{e}\big(1+o_n(1)\big)\,. \qquad \square$$

Sei p ein Primteiler von N. Wählt man zufällig $x_0, x_1, x_2, \ldots \in \{0, 1, \ldots, N-1\}$, dann kann man damit rechnen, dass es nach $O(\sqrt{p})$ solchen Wahlen mit hoher Wahrscheinlichkeit ein Paar von Indizes $0 \leq i < j$ gibt mit $x_i \equiv x_j \bmod p$ (die zugrunde liegende Menge ist die Menge der Restklassen mod p); es ist sehr

wahrscheinlich (die Wahrscheinlichkeit ist $1 - p/N$), dass $x_i \neq x_j$ ist. Dann ist $\mathrm{ggT}(N, x_j - x_i)$ ein nichttrivialer Teiler von N.

Das Problem bei diesem Ansatz ist, dass die Berechnung der größten gemeinsamen Teiler für die $O(\sqrt{p}^2) = O(p)$ Paare (x_i, x_j) zu aufwendig ist. Dem begegnet man, indem man die zufällige Folge (x_j) durch eine deterministische Folge ersetzt, die durch $x_{j+1} = f(x_j)$ mit einer geeigneten Abbildung f gegeben ist. Dann gibt es folgenden Trick für die Kollisionsfeststellung, der den Aufwand auf linear (im größeren Index j) viele Berechnungen reduziert:

LEMMA Kollisionserkennung

11.2. **Lemma.** *Seien M eine Menge, $x_0 \in M$ und $f\colon M \to M$ eine Abbildung. Wir definieren die Folge (x_j) durch $x_{j+1} = f(x_j)$. Gibt es $0 \leq i < j$ mit $x_i = x_j$, dann gibt es auch $0 < k \leq j$ mit $x_k = x_{2k}$.*

Beweis. Aus $x_i = x_j$ folgt (durch Induktion mit $x_{l+1} = f(x_l)$) $x_{i+m} = x_{j+m}$ für alle $m \geq 0$. Daraus folgt wiederum durch Induktion auch $x_{i+m} = x_{i+l(j-i)+m}$ für alle $l, m \geq 0$. Sei $m \geq 0$ minimal, sodass $i+m$ echt positiv und durch $j-i$ teilbar ist. Dann ist $m \leq j - i$, also $k := m + i \leq j$, und es gilt mit $i + m = l(j - i)$

$$x_{2k} = x_{2i+2m} = x_{i+l(j-i)+m} = x_{i+m} = x_k\,.$$

❑

Eine lineare Funktion $f(x) = (ax + b)$ rem N ergibt Folgen, die sich sehr wenig wie zufällige Folgen verhalten; daher wählt man eine quadratische Funktion wie $f(x) = (x^2 + 1)$ rem N. Unter der Annahme, dass sich die entstehenden Folgen im Wesentlichen wie zufällige Folgen verhalten, hat der folgende Algorithmus eine Laufzeit von $O(\sqrt{p})$ Operationen auf Zahlen der Länge $\lambda(N)$, wobei $p \leq \sqrt{N}$ der kleinste Primteiler von N ist.

ALGO Pollard ρ

function PollardRho(N)
input: $N \in \mathbb{Z}_{\geq 2}$ zusammengesetzt.
output: $1 < d < N$ ein Teiler von N, oder „Misserfolg“.

```
x ← random(0, N − 1); y ← x
repeat
    x ← (x² + 1) rem N
    y ← (y² + 1) rem N; y ← (y² + 1) rem N
    g ← ggT(N, y − x)
until g > 1
if g < N then
    return g
else
    return „Misserfolg“
end if
end function
```

Die Wahrscheinlichkeit für einen „Misserfolg“ ist sehr gering. Man kann es dann mit einer anderen Wahl des Startwerts noch einmal probieren. (Für einige sehr kleine N wie $N = 4$ oder 8 gibt es keinen geeigneten Startwert. In der Praxis ist das jedoch kein Problem, da man zuerst „kleine“ Primfaktoren durch Probedivision entfernt; für die Zahlen, auf die man das Verfahren anwendet, stimmt dann die obige Aussage.)

Der Name „ρ“ kommt angeblich von der Form der Folge (x_j rem p), die nach einigen Schritten in einem Zykel endet, was sich in der Form eines ρ darstellen lässt.

Der Vorteil dieses Algorithmus' gegenüber dem vorigen ist, dass er nur wenig Platz benötigt (für die beiden Zahlen x und y der Länge $\lambda(N)$, plus Zwischenspeicher der gleichen Größenordnung für die Rechnungen) bei vermutlich vergleichbarer erwarteter Laufzeit. Allerdings gibt es bisher keinen Beweis dafür, dass für die deterministische Folge, wie sie im Algorithmus verwendet wird, der Erwartungswert der Schritte bis zur ersten Kollision ebenfalls $O(\sqrt{p})$ ist. In der Praxis hat sich diese Annahme aber als berechtigt herausgestellt. Mit dieser Methode wurde zum Beispiel ein 16-stelliger Primteiler der Fermat-Zahl F_8 gefunden.

11.3. **Beispiel.** Als einfaches Beispiel faktorisieren wir $N = 2047$. Wir nehmen $x_0 = 1$ und berechnen: **BSP** Pollard ρ

j	0	1	2	3	4	5	6
x_j	1	2	5	26	677	1849	312
x_{2j}	1	5	677	312	887	1347	289
$\mathrm{ggT}(N, x_{2j} - x_j)$		1	1	1	1	1	23

In diesem Fall finden wir also nach sechs Schritten den Teiler 23.

Die Folge (x_j) ist

j	0	1	2	3	4	5	6	7	8	9	10
x_j	1	2	5	26	677	1849	312	1136	887	722	1347

j	11	12	13	14	15	16	17	18	19
x_j	768	289	1642	266	1159	450	1895	588	1849

und wiederholt sich ab x_5 mit Periode 14. Modulo 23 hat die Periode Länge 2 und beginnt ebenfalls mit x_5, während modulo 89 die Periode Länge 14 hat, aber bereits mit $x_1 = 2$ beginnt. ♣

Als nächstes wollen wir eine Klasse von Algorithmen besprechen, die eine deutlich bessere Komplexität erreichen, die zwischen polynomialer und exponentieller Komplexität liegt. Sie beruhen auf der folgenden einfachen Beobachtung, die in ähnlicher Form bereits Fermat bekannt war.

11.4. **Lemma.** *Sei $N \in \mathbb{Z}_{\geq 2}$. Sind $x, y \in \mathbb{Z}$, sodass $x^2 \equiv y^2 \bmod N$, aber $x \not\equiv \pm y \bmod N$, dann ist $\mathrm{ggT}(x + y, N)$ ein nichttrivialer Teiler von N.* **LEMMA** $x^2 \equiv y^2$

Beweis. Nach Voraussetzung ist N ein Teiler von $x^2 - y^2 = (x + y)(x - y)$. Da N kein Teiler von $x + y$ ist, folgt jedenfalls $\mathrm{ggT}(x + y, N) < N$. Wären $x + y$ und N teilerfremd, dann würde $N \mid x - y$ folgen, was aber ausgeschlossen ist. Also ist $1 < \mathrm{ggT}(x + y, N) < N$ wie gewünscht. ❑

Sei $N = p_1^{e_1} \cdots p_k^{e_k}$ ungerade und zusammengesetzt mit $k \geq 2$ verschiedenen Primteilern. Nach Lemma 10.6 hat 1 in $\mathbb{Z}/p_j^{e_j}\mathbb{Z}$ genau die zwei Quadratwurzeln 1 und -1. Daraus folgt, dass es für jedes $x \perp N$ genau die zwei Quadratwurzeln $[x]$ und $[-x]$ von $[x^2]$ in $\mathbb{Z}/p_j^{e_j}\mathbb{Z}$ gibt. Nach dem Chinesischen Restsatz ist $\mathbb{Z}/N\mathbb{Z} \cong \mathbb{Z}/p_1^{e_1}\mathbb{Z} \times \cdots \times \mathbb{Z}/p_k^{e_k}\mathbb{Z}$. Die Quadratwurzeln von $[x^2]$ in jeder Komponente rechts lassen sich also beliebig zu einer Quadratwurzel von $[x^2]$ links kombinieren;

dafür gibt es 2^k Möglichkeiten. Genau zwei dieser 2^k Quadratwurzeln $[y]$ haben die Eigenschaft $x \equiv \pm y \bmod N$. Ist also y zufällig mit $x^2 \equiv y^2 \bmod N$, dann ist die Wahrscheinlichkeit dafür, dass wir einen nichttrivialen Teiler von N finden, $1 - 2^{1-k} \geq \frac{1}{2}$.

Die Frage ist nun, wie man solche x und y findet. Die Grundidee ist dabei folgende: Man fixiert $p_0 = -1$ und verschiedene Primzahlen p_1, p_2, …, p_m. Das Tupel $(p_0, p_1, \ldots, p_m)$ bildet die *Faktorenbasis*. Dann erzeugt man geeignete Zahlen x, sodass der absolut kleinste Rest von x^2 modulo N sich bis aufs Vorzeichen als Produkt von Potenzen der p_j schreiben lässt. Man erhält also Kongruenzen der Form

$$x_i^2 \equiv (-1)^{e_{i,0}} p_1^{e_{i,1}} p_2^{e_{i,2}} \cdots p_m^{e_{i,m}} \bmod N$$

für $i = 1, 2, 3, \ldots$. Hat man $n \geq m + 2$ solcher Kongruenzen gesammelt, dann ist garantiert, dass es einen 0-1-Vektor $(v_1, v_2, \ldots, v_n)$ gibt, sodass

$$v_1 e_{1,j} + v_2 e_{2,j} + \ldots + v_n e_{n,j} = 2w_j$$

gerade ist für alle j. (Denn die Matrix über $\mathbb{F}_2$, deren Einträge die modulo 2 reduzierten Exponenten $e_{i,j}$ sind, hat mehr Zeilen als Spalten und damit einen nichttrivialen linken Kern.) Es folgt

$$\left(x_1^{v_1} \cdots x_n^{v_n}\right)^2 \equiv \left(p_1^{w_1} \cdots p_m^{w_m}\right)^2 \bmod N\,,$$

womit wir passende $x = x_1^{v_1} \cdots x_n^{v_n}$ und $y = p_1^{w_1} \cdots p_m^{w_m}$ gefunden haben.

Das Grundgerüst des Faktorisierungsalgorithmus sieht damit so aus:

ALGO $x^2 \equiv y^2$: Gerüst

---

function factor(N)

input: $N \in \mathbb{Z}_{\geq 2}$ ungerade und zusammengesetzt.

output: $1 < d < N$ ein Teiler von N oder „Misserfolg“.

Wähle geeignete Faktorenbasis $(-1, p_1, p_2, \ldots, p_m)$
Erzeuge $n \geq m + 2$ Relationen der Form $x_i^2 \equiv (-1)^{e_{i,0}} p_1^{e_{i,1}} \cdots p_m^{e_{i,m}} \bmod N$
$A \leftarrow (e_{i,j} \bmod 2)_{1 \leq i \leq n, 0 \leq j \leq m} \in \mathrm{Mat}(n, m+1, \mathbb{F}_2)$
Berechne $v \in \mathbb{F}_2^n \setminus \{\mathbf{0}\}$ mit $v^\top A = \mathbf{0}$
$v \leftarrow \mathrm{lift}(v) \in \mathbb{Z}^n$
$x \leftarrow x_1^{v_1} \cdots x_n^{v_n} \operatorname{rem} N$
for $j = 1$ **to** m **do** $w_j \leftarrow (v_1 e_{1,j} + \ldots + v_n e_{n,j})/2$ **end for**
$y \leftarrow p_1^{w_1} \cdots p_m^{w_m} \operatorname{rem} N$
$d \leftarrow \gcd(x + y, N)$
if $1 < d < N$ **then return** d **else return** „Misserfolg“ **end if**
end function

„lift“ soll dabei komponentenweise den Repräsentanten in $\{0, 1\} \subset \mathbb{Z}$ einer Restklasse in $\mathbb{F}_2 = \mathbb{Z}/2\mathbb{Z}$ liefern.

Im Fall des „Misserfolgs“ wird man in der Praxis einige weitere Relationen berechnen und es dann mit den zusätzlichen Vektoren im Kern der Matrix noch einmal versuchen.

Die einfachste Variante ist dann durch folgende Spezifizierungen gegeben:

- Die Primzahlen in der Faktorenbasis sind alle Primzahlen $\leq B$ mit einem geeignet zu wählenden Parameter B.

- Wir wählen x zufällig und gleichverteilt in $\{\lfloor\sqrt{N}\rfloor+1,\dots,\lfloor N/2\rfloor\}$ und testen per Probedivision, ob der absolut kleinste Rest von x^2 mod N sich als Potenzprodukt über der Faktorenbasis schreiben lässt.

Für die Analyse der Laufzeit ist es wichtig zu wissen, wie groß die Wahrscheinlichkeit dafür ist, dass eine ganze Zahl z mit $|z| \le N/2$ nur Primteiler $\le B$ hat. Wir definieren erst einmal folgende Funktion:

* 11.5. **Definition.** Für reelles $x > e$ und $0 \le r \le 1$ sei

DEF $L_r(x)$

$$L_r(x) = e^{(\log x)^r (\log\log x)^{1-r}}.$$

Wir setzen noch

$$L(x) = L_{1/2}(x) = e^{\sqrt{(\log x)(\log\log x)}}.$$ ♢

Es ist $L_1(x)^c = x^c$ und $L_0(x)^c = (\log x)^c$; die Funktionen L_r interpolieren also zwischen polynomialem und exponentiellem Wachstum in $\log x$.

Nun hat man folgendes wichtiges Resultat von de Bruijn und Canfield, Erdős und Pomerance:

11.6. **Satz.** *Sei $\alpha > 0$. Für $x \to \infty$ ist der Anteil der positiven ganzen Zahlen $y < x$, sodass alle Primteiler von y durch $L(x)^\alpha$ beschränkt sind, gegeben durch $L(x)^{-1/2\alpha+o(1)}$.*

SATZ Satz von de Bruijn; Canfield, Erdős und Pomerance

„$o(1)$“ steht dabei für einen Term, der für $x \to \infty$ gegen null geht.

De Bruijn hat Resultate über die Asymptotik von

$$\Psi(x,y) = \#\{z \in \mathbb{Z} \mid 1 \le z \le x, \text{alle Primfaktoren von } z \text{ sind} \le y\}$$

bewiesen[5], die von Canfield, Erdős und Pomerance erweitert wurden[6]. Der obige Satz ergibt sich aus dem Corollary in Abschnitt 3 ihres Artikels, wenn man dort

$$u = \alpha^{-1}\sqrt{\log x/\log\log x}$$

setzt.

Wenn wir $B = L(N)^\alpha$ wählen, dann ist also die Wahrscheinlichkeit, dass x^2 rem N (oder der absolut kleinste Rest) als Potenzprodukt über der Faktorenbasis geschrieben werden kann, etwa $L(N)^{-1/2\alpha}$. (Man kann zeigen, dass das so auch für die Teilmenge der Quadrate in $\mathbb{Z}/N\mathbb{Z}$ gilt.) Der Erwartungswert für die Anzahl der x, die getestet werden müssen, ist damit etwa $L(N)^{1/2\alpha}m$, und nach dem Primzahlsatz ist

$$m = \pi(B) \approx B/\log B = \frac{L(N)^\alpha}{\alpha\sqrt{(\log N)(\log\log N)}},$$

was wir nach oben durch $L(N)^\alpha$ abschätzen. ($\pi(B)$ bezeichnet die Anzahl der Primzahlen $\le B$.) Der Aufwand für eine Probedivision durch die Primzahlen bis B ist $\tilde{O}(L(N)^\alpha)$, sodass der Aufwand für das Sammeln der Relationen insgesamt etwa $L(N)^{2\alpha+1/(2\alpha)}$ ist. Die Matrix A hat Größe etwa $L(N)^\alpha$. Standard-Algorithmen (Gauß-Elimination) zur Berechnung des Kerns haben eine kubische Komplexität, also hier $L(N)^{3\alpha}$. Die Matrix ist allerdings dünn besetzt (es können nicht mehr als $\log_2 N$ Primfaktoren auftreten), sodass spezielle Verfahren für dünn besetzte

[5] N.G. de Bruijn: On the number of positive integers $\le x$ and free of prime factors $> y$. II. Indag. Math. **28**, 239–247 (1966).

[6] E.R. Canfield, P. Erdős, C. Pomerance: On a problem of Oppenheim concerning “factorisatio numerorum”. J. Number Theory **17**, 1–28 (1983).

Matrizen verwendet werden können, deren Komplexität im Wesentlichen quadratisch ist. Der optimale Wert von α ist dann $\alpha = 1/2$ (das gilt auch für kubische Algorithmen für die lineare Algebra), und die Komplexität ist damit (grob)

$$L(N)^2 = e^{2\sqrt{(\log N)(\log\log N)}}.$$

Das liefert:

* 11.7. **Satz.** *Es gibt einen probabilistischen Algorithmus, der für eine zusammengesetzte Zahl N einen nichttrivialen Teiler d bestimmt und einen erwarteten Aufwand von*

$$\ll L(N)^c$$

Wortoperationen hat, mit einem $c > 0$.

SATZ subexponentielle Faktorisierung

$L(N)^c$ wächst zwar stärker als jedes Polynom in $\log N$, aber schwächer als jede Potenzfunktion $N^a = e^{a \log N}$; deswegen spricht man von *subexponentieller* Komplexität.

Der obige Algorithmus lässt sich zwar recht gut analysieren, ist aber für die Praxis nicht gut geeignet, da die Wahrscheinlichkeit, eine „glatte“, also über der Faktorenbasis faktorisierbare, Zahl zu finden, recht klein ist. Man versucht stattdessen, die Zahlen x so zu wählen, dass der absolut kleinste Rest von x^2 mod N nur von der Größenordnung $\sqrt{N}$ (oder evtl. $N^{1/2+\varepsilon}$) ist. Eine Möglichkeit dafür ist, die Kettenbruchentwicklung von $\sqrt{kN}$ für geeignetes kleines k zu verwenden. Sie liefert Relationen der Form $x^2 - kNz^2 = u$ mit u der Größenordnung $\sqrt{N}$.

Eine andere Möglichkeit ist das *Quadratische Sieb*. In der Grundvariante betrachtet man das Polynom

$$f(X) = \big(X + \lfloor\sqrt{N}\rfloor\big)^2 - N = X^2 + 2\lfloor\sqrt{N}\rfloor X + \lfloor\sqrt{N}\rfloor^2 - N\,.$$

Der konstante Term ist $\ll \sqrt{N}$. Für $a \ll N^\varepsilon$ mit ε klein ist dann

$$f(a) \ll N^{1/2+\varepsilon} \qquad \text{und} \qquad f(a) \equiv \big(a + \lfloor\sqrt{N}\rfloor\big)^2 \bmod N\,.$$

Das erlaubt es einem, $L(N)$ durch $L(\sqrt{N}) \approx L(N)^{1/\sqrt{2}}$ zu ersetzen. Wie unten erklärt, kann man hier die Kosten für die Probedivisionen soweit drücken, dass sie vernachlässigbar werden. Mit $B = L(\sqrt{N})^\alpha$ ist die Komplexität dann $L(\sqrt{N})^{\alpha+1/2\alpha}$; für $\alpha = 1/\sqrt{2}$ bekommt man einen Algorithmus der Komplexität etwa $L(\sqrt{N})^{\sqrt{2}} \approx L(N)$, was also den Exponenten c von 2 auf $c = 1$ verbessert.

Hier gilt stets $f(a) + N = \square$, was bedeutet, dass nur Primzahlen p in die Faktorenbasis aufgenommen werden müssen, für die N ein quadratischer Rest mod p ist. Das reduziert die Größe der Faktorenbasis auf etwa die Hälfte. Außerdem lässt sich die Probedivision in dieser Version stark beschleunigen: Ob $f(a)$ durch p_j^e teilbar ist, hängt nur von a mod p_j^e ab. Die (für p_j ungerade stets zwei) relevanten Restklassen kann man vorberechnen. Man füllt einen Array mit groben Näherungen von $\log_2 f(a)$ für a im Siebintervall $[-M, M]$. Für jede mögliche Potenz p_j^e zieht man dann von den Einträgen $\approx \log_2 p_j$ ab, für die a in einer der relevanten Restklassen mod p_j^e liegt. Die interessanten Werte von a sind dann die, für die der verbleibende Rest klein ist. Auf diese Weise werden die Divisionen durch Subtraktionen von kleinen Zahlen ersetzt, was in der Praxis einen deutlichen Geschwindigkeitsgewinn bringt. Bei aktuellen Implementationen von Varianten dieser Methode ist üblicherweise die Berechnung eines Vektors im Kern der Matrix der zeitaufwendigste Schritt.

BSP Quadratisches Sieb

11.8. **Beispiel.** Wir wollen $N = 4\,534\,511$ faktorisieren. Wir berechnen $f(a)$ für $-100 \le a \le 100$ und finden folgende Werte, die nur Primfaktoren < 50 haben:

$$\begin{aligned} f(-60) &= -2 \cdot 5^4 \cdot 7 \cdot 29 \\ f(-23) &= -5^2 \cdot 11 \cdot 19^2 \\ f(-12) &= -2 \cdot 7^4 \cdot 11 \\ f(0) &= -2 \cdot 5 \cdot 11 \cdot 17 \\ f(2) &= 2 \cdot 5^2 \cdot 7 \cdot 19 \\ f(17) &= 5 \cdot 7^2 \cdot 17^2 \\ f(65) &= 5^3 \cdot 7 \cdot 11 \cdot 29 \end{aligned}$$

Die Matrix (mit $(p_0, \dots, p_m) = (-1, 2, 5, 7, 11, 17, 19, 29)$) ist

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

und hat eindimensionalen linken Kern erzeugt von $(1, 0, 1, 0, 0, 1, 1)$. (Der Kern ist nichttrivial, obwohl die Matrix mehr Spalten als Zeilen hat.) Das liefert mit $s = \lfloor \sqrt{N} \rfloor = 2129$

$$x = (-60 + s)(-12 + s)(17 + s)(65 + s) \text{ rem } N = 973\,026$$

und

$$y = 2^1 \cdot 5^4 \cdot 7^4 \cdot 11^1 \cdot 17^1 \cdot 19^0 \cdot 29^1 \text{ rem } N = 1\,418\,771$$

und damit

$$\mathrm{ggT}(x + y, N) = 3001\,. \qquad \clubsuit$$

Eine in der Praxis eingesetzte Variante benutzt mehr als ein Polynom f, um mehr hinreichend kleine Werte zu produzieren: MPQS, *Multiple Polynomial Quadratic Sieve*. Zum Beispiel verwendet Magma MPQS, um verbleibende zusammengesetzte Zahlen mit 25 oder mehr Dezimalstellen zu faktorisieren.

Eine Weiterentwicklung des Quadratischen Siebs ist das *Zahlkörpersieb* NFS (*Number Field Sieve*). Dabei wird in geeigneten algebraischen Zahlkörpern gerechnet, um Relationen wie oben zu produzieren. Das resultiert in einem Algorithmus, der (unter unbewiesenen, aber vernünftigen Annahmen) eine Komplexität von nur noch etwa

$$L_{1/3}(N)^c = e^{c \sqrt[3]{(\log N)(\log\log N)^2}} \quad \text{Wortoperationen}$$

hat, wobei $c = \sqrt[3]{64/9}$ genommen werden kann. Dies ist asymptotisch noch einmal deutlich besser als die $L(N)^c$-Varianten. Auf der anderen Seite ist das Verfahren ziemlich komplex; die bessere Asymptotik macht sich erst für recht große Zahlen N bemerkbar. Mit dieser Methode sind im Wesentlichen alle Faktorisierungsrekorde der letzten Jahrzehnte erzielt worden.

Zum Abschluss gehen wir noch kurz auf eine weitere Klasse von Algorithmen ein, die mit Gruppen arbeiten. Der Prototyp ist der „$p-1$-Algorithmus“ von Pollard, der die multiplikative Gruppe $\mathbb{F}_p^\times$ verwendet. Er kann Primteiler p von N finden,

sodass $p-1$ „glatt“ ist in dem Sinn, dass $p-1$ nur durch Primzahlpotenzen unterhalb einer Schranke B teilbar ist. Sei

$$\ell(B) = \mathrm{kgV}(1,2,3,\ldots,B) = \prod_{p\le B} p^{\lfloor \log_p B\rfloor}.$$

Dann gilt nach Voraussetzung $p-1 \mid \ell(B)$, woraus mit dem kleinen Satz von Fermat folgt, dass $x^{\ell(B)} = 1$ ist für alle $x \in \mathbb{F}_p^\times$. Wenn N noch einen weiteren Primteiler q hat, für den $q-1$ nicht „B-potenzglatt“ ist im obigen Sinne, dann ist es sehr wahrscheinlich, dass $a^{\ell(B)} \not\equiv 1 \bmod q$ ist für ein zufälliges $a \in \{2,3,\ldots,N-2\}$. Dann ist $\mathrm{ggT}(N, a^{\ell(B)}-1)$ ein nichttrivialer Teiler von N. Ein passender Algorithmus sieht so aus:

ALGO
Pollard
$p-1$

function Pollardpminus1(N, B)
input: $N \in \mathbb{Z}_{\ge 2}$ ungerade und zusammengesetzt, B Glattheits-Schranke.
output: $1 < d < N$ ein Teiler von N oder „Misserfolg“.

```
p ← 2
a ← random(2, N − 2)
while p ≤ B do
    a ← a^(p^⌊log_p B⌋) rem N
    p ← NextPrime(p)
end while
d ← ggT(N, a − 1)
if 1 < d < N then return d else return „Misserfolg“ end if
end function
```

Die Laufzeit ist im Wesentlichen die für die modulare Potenz $a^{\ell(B)}$ rem N. Aus dem Primzahlsatz folgt, dass $\ell(B) \approx e^B$ ist; damit ist der Aufwand im Wesentlichen B Operationen in $\mathbb{Z}/N\mathbb{Z}$, also $\tilde{O}(B(\log N))$ Wortoperationen. Der Speicherplatzbedarf ist gering ($O(\log N)$). Der große Nachteil dieses Verfahrens ist, dass es (bei realistischer Wahl von B) nur Primteiler einer speziellen Form findet.

Dieser Nachteil lässt sich umgehen, wenn man statt der multiplikativen Gruppe $\mathbb{F}_p^\times$ die Gruppe $E(\mathbb{F}_p)$ der Punkte auf einer elliptischen Kurve E über $\mathbb{F}_p$ verwendet. Davon gibt es eine große Auswahl, mit Gruppenordnungen, die einigermaßen gleichmäßig über das Intervall $[(\sqrt{p}-1)^2, (\sqrt{p}+1)^2]$ verteilt sind. Man wählt zufällig eine Kurve E über $\mathbb{Z}/N\mathbb{Z}$ und einen Punkt P darauf und berechnet $\ell(B)\cdot P$ (die Gruppenoperation wird additiv geschrieben). Ist die Gruppenordnung der entsprechenden Kurve über $\mathbb{F}_p$ glatt, aber nicht die der Kurve über $\mathbb{F}_q$, wobei p und q wieder Primteiler von N sind, dann führt das (sehr wahrscheinlich) dazu, dass im Lauf der Rechnung eine Zahl a modulo N invertiert werden müsste, die nicht durch N teilbar, aber auch nicht zu N teilerfremd ist. Wie üblich ist dann $\mathrm{ggT}(N,a)$ ein nichttrivialer Teiler. Das daraus abgeleitete Verfahren heißt ECM (*Elliptic Curve Method*). Seine großen Vorteile sind, dass es geringen Platzbedarf hat (ähnlich wie Pollards $p-1$-Algorithmus) und dass seine Laufzeit (unter plausiblen, aber unbewiesenen Annahmen) subexponentiell ist in der Größe des gesuchten Primfaktors p, nämlich $L(p)^{\sqrt{2}}$. Es wird daher üblicherweise für die Suche nach „mittelgroßen“ Primteilern eingesetzt mit bis zu etwa 20–25 Dezimalstellen. (Im „worst case“ $p \asymp \sqrt{N}$ hat man dieselbe asymptotische Komplexität

$L(\sqrt{N})^{\sqrt{2}} \approx L(N)$ wie MPQS. Da die elementaren Operationen bei MPQS aber deutlich schneller sind, ist für große N MPQS hier überlegen.)

Die Faktorisierungsfunktion von Magma verwendet zum Beispiel folgende Verfahren in der angegebenen Reihenfolge:

(1) Miller-Rabin-Test (plus ECPP, falls prim)

(2) Probedivision für $p < 10\,000$

(3) Pollard-ρ mit 8191 Iterationen

(4) ECM

(5) MPQS

Genauere Informationen über die erwähnten Algorithmen und Hinweise zur Optimierung findet man in den Büchern von Kaplan [Ka] (Abschnitt 5) und Cohen [Co] (Chapter 8 und 10).

Literatur

[Co] Henri Cohen: *A course in computational algebraic number theory,* Springer GTM **138**, Springer-Verlag Berlin Heidelberg, 1993.

[GG] Joachim von zur Gathen und Jürgen Gerhard: *Modern computer algebra,* 2nd edition, Cambridge University Press, 2003. (Es gibt auch eine neuere Ausgabe von 2013.)

[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