Algorithmen in der Arithmetik

Skriptum zur Vorlesung

Institut für Mathematik und ihre Didaktik
Universität Flensburg

1. Begriffliche Grundlagen

1.1. Was ist ein Algorithmus?

Historische Herkunft des Begriffs

Das Wort "Algorithmus" geht auf die lateinische Fassung eines arabischen Namens zurück. Dieser gehörte dem Gelehrten Al-Chwarizmi, der seine mathematischen Werke im 9. Jahrhundert am Hofe des Kalifen Al-Ma'mun in Bagdad schrieb. Eines von ihnen hieß *Hisab al-gabr wa'l-muqabala* (Rechenverfahren durch Ergänzen und Ausgleichen). Das Wort "Algebra" stammt aus diesem Titel und ist offenbar auch historisch mit dem Lösen von Gleichungen verbunden. In Spanien tauchte der Name des Verfassers rund drei Jahrhunderte später in einer lateinischen Bearbeitung seiner Bücher auf. Diese beginnt mit den Worten:
Dixit Algoritmi ... (Es sprach Algoritmi ...).
Im Laufe der Zeit verband man die Verballhornung "Algorithmus" ganz allgemein mit mechanisch ausführbaren Rechenverfahren.

Effektive mechanische Verfahren als Ziel der Mathematik

Viele Probleme, mit denen Mathematiker sich beschäftigen, bestehen darin, sich der Existenz bestimmter Objekte zu versichern: einer Zahl, einer Figur, einer Struktur, einer Antwort auf eine Entscheidungsfrage, u.a.m. Die beste Lösung solcher Aufgaben ist die, ein effektives Verfahren zu finden, mit dem sich quasi mechanisch das betreffende Objekt finden (berechnen, konstruieren usw.) lässt. Ein solches Verfahren nennt man *Algorithmus*. Wenn man die Lösung eines Problems algorithmisch bestimmen kann, so liegt darin gewiss mehr Information als in dem Wissen, *dass* es überhaupt eine Lösung gibt. (Manche Mathematiker haben sich sogar gefragt, welchen Sinn eine Existenzaussage hat, wenn man nicht auch das Objekt, dessen Existenz behauptet bzw. bewiesen wird, tatsächlich aufweisen oder konstruieren kann.)
Für viele Aufgaben und Probleme gibt es Verfahren, die auch demjenigen die Bestimmung einer Lösung erlauben, der die Begründung, ja nicht einmal das Verfahren als solches versteht.
Einige einfache Beispiele aus der Schulmathematik:

- Ausführung der Grundrechenarten im Dezimalsystem
- Berechnung der Summe zweier Brüche (als Bruch)
- Bestimmung der Lösungen einer quadratischen Gleichung mit Hilfe von Wurzeln
- Differentiation von Polynomen

Vom Standpunkt der Didaktik aus wird gewöhnlich dafür plädiert, diese und andere Algorithmen nicht als mechanische Routinen einzuüben, sondern auch zu *verstehen*. Das ist sicherlich ein sinnvolles Ziel. Löst man sich jedoch aus dem Bildungskontext, so wird deutlich, dass diese und viele andere (weitaus schwierigere) Probleme an erstrangigem mathematischen Interesse verloren haben – man ist versucht, überspitzt zu sagen: aus der Mathematik verschwunden sind –, gerade weil sie im Prinzip von einer Maschine gelöst werden können. Ein schlagkräftiges Beispiel dafür sind die – nach der Schulmethode ausgeführten – Grundrechenarten, die wir schon aus Gründen der Alltagsökonomie im allgemeinen nur automatisch anwenden. Dies ist – jenseits von aller Didaktik – ein praktischer Nutzen, den wir der *Mathematik als Technologie* verdanken. (Natürlich kann es innerhalb dieser Technologie noch von beträchtlichem Interesse sein, vorhandene Algorithmen weiter zu verbessern.)
Die besten Algorithmen sind inzwischen längst in Computer-Programme umgesetzt. Damit werden sie zu abgeschlossenen Modulen ("black boxes"). Das ursprüngliche Problem, zu dessen Lösung sie geschaffen wurden, erscheint nur noch als eine Routine-Aufgabe. Man steckt sein Problem in den Schwarzen Kasten hinein, und heraus kommt die Lösung. Auf diese Weise entfällt die Notwendigkeit, in jedem einzelnen Fall darüber nachzudenken, wie eine Aufgabe einer algorithmisch lösbaren Problemklasse angegangen werden soll. Man kann dies kaum prägnanter ausdrücken als durch folgenden Satz in [Ronald L. Graham, Donald E. Knuth, Oren Patashnik: *Concrete Mathematics*. Addison-Wesley Publishing Co.: Reading, MA 1989, p. 56]:

> *The ultimate goal of mathematics is to eliminate all need for intelligent thought.*

(Sicherheitshalber sei angemerkt: Problemlösen, Begriffs- und Theoriebildung sowie Anwenden mathematischer Konzepte sind selbstredend *keine* automatisierbaren Routinen.)
Wir schalten das Fernsehgerät ein, fotografieren mit vollautomatischen Kameras oder setzen uns ins Flugzeug, ohne auch nur im Traum einen Gedanken an die enorme Menge mathematisch-algorithmischer (und natürlich auch naturwissenschaftlicher) Grundlagen zu verschwenden, die in diesen Systemen stecken.

Charakteristische Eigenschaften eines Algorithmus

Wenden wir uns nun der Frage zu, welche Eigenschaften ein Verfahren haben muss, um als *Algorithmus* im Sinne der Mathematik gelten zu können. Wir alle kennen aus dem Alltag Situationen, in denen man ein Verfahren bzw. eine Verfahrensbeschreibung benötigt: etwa um ein neues Gericht nach einem bestimmten Rezept zu kochen oder um einen Kleiderschrank nach einer beiliegenden Anleitung zusammenbauen. Solche Verfahren kommen der Idee eines Algorithmus immerhin schon nahe; allerdings lassen sie – wie wir gleich sehen werden – wesentliche Eigenschaften eines "echten" Algorithmus vermissen.
Um von den wesentlichen Eigenschaften eines Algorithmus eine Vorstellung zu bekommen, orientiert man sich am besten an historischem Material, d.h. man studiert vorhandene (möglichst einfache) Verfahren und sucht nach deren Gemeinsamkeiten. Aus einer solchen Untersuchung könnte eine Liste von folgenden Eigenschaften hervorgehen:

I. **Diskretheit.** Ein Algorithmus besteht aus einer Folge von Schritten.
II. **Determiniertheit.** Bei gleichen Startbedingungen erzeugt er stets dasselbe Endergebnis.
III. **Eindeutigkeit.** Nach jedem Schritt lässt er sich auf höchstens eine Art fortsetzen.
IV. **Endlichkeit.** Er endet nach endlich vielen Schritten.

Es ist lehrreich, sich bei der Aufstellung und Untersuchung von Algorithmen die hier genannten charakterischen Eigenschaften ins Bewusstsein zu rufen.

Bemerkung
Mit der Angabe (oder Forderung) der Eigenschaften I-IV wird *keineswegs* der Begriff des Algorithmus (in irgendeinem mathematischen Sinn) *definiert*. Für ein solches Unterfangen braucht man weitergehende Mittel. Erst um 1930 ist führenden Logikern (Gödel, Church, Kleene, Turing, u.a.) eine exakte Definition des Algorithmus-Begriffs gelungen. Für unsere Zwecke ist eine solche Definition nicht erforderlich. Man muss erst dann wissen, was genau unter einem Algorithmus zu verstehen ist, wenn man Beweise darüber führen will, dass sich ein bestimmtes Problem (oder eine bestimmte Klasse von Problemen) *nicht algorithmisch* lösen lässt.

1.2. Die Beschreibung von Algorithmen

Beispiele aus der Schulmathematik

Im Mathematikunterricht kommen immer wieder Algorithmen vor. Typische Beispiele sind Rechenverfahren im Dezimalsystem, Formeln zur Lösung von Gleichungen, Konstruktionsvorschriften zur Erzeugung einer bestimmten geometrischen Figur. Der algorithmische Charakter dieser Verfahren kommt dabei durchaus zum Tragen: sie werden als Handlungsvorschriften angeeignet, eingeübt und künftig als Mittel zur Lösung von Aufgaben benutzt.

Bemerkung
Die zentrale didaktische Frage, ob und in welchem Umfang algorithmische Verfahren verstanden oder begründet werden sollten, kann hier nicht tiefergehend verfolgt werden. Die Beschäftigung mit Algorithmen in einem Studium für das Lehramt schließt aber geradezu die Forderung ein, mit Algorithmen reflektiert umzugehen. Dies ist zumindest eine unerlässliche Grundlage für didaktische Entscheidungen.

Beispiel 1: Lösung einer quadratischen Gleichung

Herstellung einer Normalform, etwa: $x^2 + a\,x\, + b = 0$ (zur Identifikation der Eingabegrößen a und b für die Lösungsformel);

Einsetzen in die bekannte Lösungsformel: $x_{1,2} = -\frac{a}{2} \pm \sqrt{\left(\frac{a}{2}\right)^2 - b}$

Untersuchung des Radikanden

Beispiel 2: Umkreis eines Dreiecks

Teilkonstruktion: Mittelsenkrechte einer Strecke (nach dem Prinzip der Modularisierung als Black-Box zu nutzen)
Vorwissen: Definition des Kreises, Ortsdefinition der Mittelsenkrechten
Umkreiskonstruktion aus zwei Mittelsenkrechten
Welche Seiten werden ausgewählt? (Optimalitätsüberlegung)

Beispiel 3: Schriftliche Division

Das Verfahren der schriftlichen Division ist ziemlich kompliziert. Entschließt man sich zu seiner Behandlung, so benutzt man die Divisionsschreibweise und übt die Schritte an Beispielen wachsender Komplexität. Vgl. [F. Padberg: *Didaktik der Arithmetik*. 2., vollst. überarb. u. erw. Aufl. BI-Wissenschaftsverlag: Mannheim 1992, S. 230].
Bei Padberg findet sich ein Flussdiagramm, das den Ablauf des Rechenverfahrens grob verdeutlicht.

Arten der Beschreibung

Damit ein Algorithmus von einer Person (oder einer Maschine) ausgeführt werden kann, muss er zuvor hinlänglich genau beschrieben werden. Dies kann auf unterschiedliche Weise geschehen:

- □ durch Abarbeitung mit Beispieldaten,
- □ durch übliche mathematische Prosa (Umgangssprache),
- □ durch verbale (halbformale) Spezifikation des Ablaufs.

Jede dieser Beschreibungsformen hat ihre Vor- und Nachteile:

Die Anwendung eines Verfahrens auf eine Beispielaufgabe setzt ein intelligent nachvollziehendes Subjekt voraus, das darüberhinaus in der Lage ist, trotz der Besonderheiten eines Einzelfalls die dahinter stehende allgemeine Regel zu erkennen (vgl. Beispiel 3).
Umgangssprachliche Beschreibungen sind meist gut verständlich und knüpfen an die in Mathematik-Lehrbüchern übliche Darstellung an; sie verlangen aber ein gewisses Verständnis dessen, was im einzelnen zu tun ist (vgl. Beispiel 1).
Die Darlegung eines Ablaufplans durch Spezifikation einer festgelegten Folge von Arbeitsschritten setzt im allgemeinen wenig oder gar kein Verständnis voraus, ist dafür aber in geringerem Maße durchschaubar (und oft auch nicht ohne Subtilitäten) (vgl. Beispiele 2 und 3).

Exakte Beschreibungsformen

Bei einer mehr systematischen Beschäftigung mit Algorithmen kommt man nicht umhin, die hier geschilderten Beschreibungsformen durch exaktere und das heißt: formalere zu ersetzen. Wer einen Algorithmus "richtig verstehen" und beurteilen (im Sinne von kritisieren, prüfen) will, der kann dies letztlich nur durch einen höheren Grad an Genauigkeit erreichen. Damit ist dann auch die Voraussetzung dafür geschaffen, am Ende einen technologisch nutzbaren Baustein der Mathematik zu erhalten. (Mit exakten Beschreibungen werden informelle Darstellungen keineswegs überflüssig, oft sind sie für ein intuitives Grundverständnis – vor allem in Lernzusammenhängen – unentbehrlich.)
Welche *exakten* Beschreibungsformen für Algorithmen bieten sich an?
Ein Algorithmus ist ein dynamisches System mit inneren Zuständen, die ihre Werte von Schritt zu Schritt verändern können. Diesen diskret getakteten Ablauf kann man grundsätzlich

- visuell, etwa durch Flussdiagramme oder vergleichbare Diagrammarten, oder:
- mit Hilfe geeigneter formaler Sprachen (Programmiersprachen)

wiedergeben. Häufig werden Diagramme als Hilfsmittel herangezogen, um einen Algorithmus als ganzes zu überblicken und seine innere Dynamik zu veranschaulichen. Die formal-sprachliche Darstellung bietet zwar nicht diese Anschaulichkeit, ist dafür aber geschmeidiger und erlaubt eine vollständige Beschreibung, die leichter auch von einer Maschine umgesetzt werden kann.
Eine formal-sprachliche Beschreibung eines Algorithmus heißt *Programm*. Im folgenden sollen Algorithmen (außer durch informelle Darstellungen) möglichst immer auch durch ein Programm beschrieben werden. Damit stellt sich die Frage nach einer dafür geeigneten Programmiersprache.

Bemerkung über Programmiersprachen

Es gibt zwei grundsätzlich verschiedene Prinzipien, nach denen Programmiersprachen arbeiten:

- Compiler-Prinzip (Übersetzer)
- Prinzip der Interpretierung

Ein *Compiler* übersetzt ein Programm *vor* seiner Ausführung auf einem Computer in die Maschinensprache, die dieser Computer "versteht". Aus dem Programmtext (der Quelle) stellt der Compiler ein neues Objekt her (Compilat oder Binärcode genannt). Zu den klassischen Compiler-Sprachen gehören z.B. Fortran, Cobol, Lisp, C und Pascal.
Ein *Interpretierer*, der eher wie ein Simultandolmetscher arbeitet, übersetzt den Programmtext *während* der Laufzeit. Beispiele für Interpretierer sind Basic, Smalltalk, Prolog und Perl.
Beide Sprachtypen haben ihre Vor- und Nachteile. Naturgemäß wird ein vorab übersetztes Programm schneller ausgeführt als ein interpretiertes Programm. Dafür sind Compilersysteme häufig komplexer in der Handhabung.
Es gibt eine Vielzahl von Programmiersprachen, darunter auch solche, für die sowohl Interpretierer wie Compiler entwickelt wurden (z.B. Prolog). In einigen Fällen übersetzt ein Compiler nur in eine Zwischensprache, die anschließend noch interpretiert werden muss (z.B. bei Basic oder Java). Solche Systeme heissen Pre-Compiler.
Ein Sonderfall ist Assembler. Hierbei handelt es sich um eine maschinennahe Sprache, die auf den Prozessor eines Computers zugeschnitten ist und direkt dessen Register zu manipulieren erlaubt. Assembler heißt auch die Software, die den Assembler-Code (d.h. den in Assembler geschriebenen Programmtext) in die zugehörige Maschinensprache übersetzt. Mit assemblierten Programmen kann man optimale Ausführungsgeschwindigkeiten erzielen.
Im folgenden wird die Programmiersprache des symbolischen Rechenprogramms *Mathematica* benutzt. Folgende Merkmale dieser (nach dem Interpretierer-Prinzip arbeitenden) Sprache machen sie für unsere Zwecke – exakte Beschreibung von Algorithmen – geeignet:
Die Sprache von *Mathematica* ...

- ist logisch konsequent aufgebaut;
- erlaubt unterschiedliche Programmierstile;
- eignet sich besonders für mathematische Zwecke;
- gestattet die Verabeitung symbolischer Ausdrücke;
- ermöglicht den Gebrauch abstrakter Datentypen;
- ist leicht zu erlernen.

Im nächsten Abschnitt werden wir uns anhand einfacher Beispiele mit einigen fundamentalen Elementen der *Mathematica*-Sprache vertraut machen.

2. Euklidischer Algorithmus

2.1. Division mit Rest

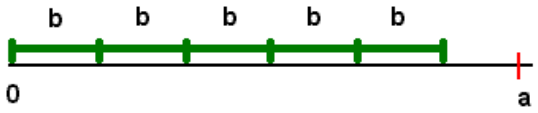
Vorbemerkung

Die *Division mit Rest* spielt eine grundlegende Rolle in der Arithmetik und Algebra. Sie bietet mit ihren arithmetischen Anwendungen (Euklidischer Algorithmus, Stellenwertsysteme, Kettenbruchentwicklung) einen guten Einstieg in die Algorithmik.
Der Satz von der Division mit Rest sollte gut begründet werden. Er lässt sich leicht veranschaulichen und unmittelbar verstehen; andererseits zeigt sich bei sorgfältiger Analyse, dass sein Beweis einen expliziten Rückgriff auf die *Wohlordnung* der natürlichen Zahlen verlangt.

Veranschaulichung an der Zahlengeraden

Die Division mit Rest ergibt sich auf natürliche Weise aus der Aufgabe, eine Größe a in b gleiche Teile zu teilen, wobei $a \geq b > 0$ (a, b ganz).

Man kann b solange von a subtrahieren, bis der Rest r (zum erstenmal!) kleiner als b wird; es ist dann $r \geq 0$. Der Sachverhalt lässt sich auch additiv deuten: Dazu trägt man a auf der Zahlengeraden als Strecke von 0 bis a ab und anschließend (ebenfalls bei 0 beginnend) die Strecke b. Solange der Endpunkt von b noch links von a liegt, wird die Strecke b addiert, d.h. an den rechten Endpunkt der zuletzt abgetragenen Strecke angehängt:



Dieser Veranschaulichung ist zu entnehmen, dass es einen Vervielfacher q (nicht-negative ganze Zahl) gibt, für den $0 \leq q\,b \leq a$ gilt, jedoch $(q+1)\,b > a$. Die Division $a : b$ hat demnach den Rest $r = a - q\,b$.

2.1.1. Satz

Zu $a, b \in \mathbb{Z}$, $b > 0$ gibt es eindeutig bestimmte $q, r \in \mathbb{Z}$, so dass gilt:
$a = q\,b + r$ und $0 \leq r < b$.

Beweis (Existenz) mit Hilfe des Minimumprinzips

Wir nehmen zunächst $a > 0$ an. Ist $b > a$, so ist die Behauptung mit $q = 0$ und $r = a$ erfüllt. Für das Folgende wird daher $b \leq a$ vorausgesetzt.
Wir definieren q als kleinste positive Zahl k, für die $(k+1)b > a$ gilt, was (nach dem Minimumprinzip) möglich ist, da es mindestens ein k mit der geforderten Eigenschaft gibt (etwa $k = a$).
Es bleibt $qb \leq a$ zu zeigen. Wäre $qb > a$, so hätten wir mit $q_1 = q - 1$ eine positive Zahl kleiner als q mit $(q_1 + 1)\,b > a$. Dies steht im Widerspruch zur Minimalität von q.
Setzen wir für den gesuchten Rest $r = a - q\,b$, so ist $r \geq 0$; es sind also alle Bedingungen erfüllt. ◆

Beweis (Existenz) durch vollständige Induktion

Für $a \geq 0$ zeigen wir die Existenz der behaupteten Zahlen q, r durch Induktion nach a.
Induktionsanfang: $a = 0$. Wähle $q = r = 0$.
Induktionsschritt: $a \to a + 1$. Nach Induktionsvoraussetzung hat man $a = q\,b + r$, $0 \leq r < b$. Hieraus ergibt sich $a + 1 = q\,b + r + 1$. Im Falle $r + 1 < b$ ist bereits alles gezeigt; bei $r + 1 = b$ erhalten wir $a + 1 = (q+1)\,b + 0$. ◆

Abschluss des Beweises

Um den Beweis zu vervollständigen, ist noch zu zeigen:
(1) die Gültigkeit der Existenzbehauptung für $a < 0$;
(2) die Eindeutigkeit von q und r.
Zu (1):
Wir erhalten (nach dem bereits Bewiesenen): $-a = q_1\,b + r_1$ und $0 \leq r_1 < b$. Mithin gilt $a = (-q_1)\,b + (-r_1)$. Im Fall $r_1 = 0$ ist nichts mehr zu zeigen. Andernfalls setzen wir $r = b - r_1$. Es ist dann $b > r > 0$ und $a = (-q_1 - 1)\,b + r$.
Zu (2):
Sei $a = q\,b + r = q_1\,b + r_1$ und $0 \leq r, r_1 < b$ angenommen. Es ergibt sich $(q - q_1)\,b = r_1 - r$. Wir nehmen ohne Einschränkung $r_1 > r$ an und erhalten: $0 \leq (q - q_1)\,b < b$. Letzteres kann nur für $q - q_1 = 0$ erfüllt sein, es folgt also $q = q_1$ und $r = r_1$. ◆

Ganzes und Rest, Teilbarkeit, Primzahl

Ganzes und Rest

Im Anschluss an den Satz 2.1.1 von der Division mit Rest werden folgende Bezeichnungen vereinbart:

Die eindeutig bestimmte ganze Zahl q heisst Quotient (oder Ganzes) der Division a:b. Die eindeutig bestimmte Zahl $r = a - q\,b$ heißt Rest.

Mathematica hält für die Berechnung jedes der beiden Werte eine eigene Funktion bereit. Das Ganze berechnet Quotient:

```
Quotient[17, 5]
```

```
3
```

```
Quotient[-17, 5]
```

```
-4
```

Den Rest, der bei der Division auftritt, berechnet Mod:

```
Mod[17, 5]
```

```
2
```

```
Mod[-17, 5]
```

```
3
```

Wir wollen eine eigene Funktion definieren, die für ganze *a*, *b* (*b* > 0) den Quotienten *q* und den Rest *r* gleichzeitig berechnet und beide Werte als geordnetes Paar {*q*, *r*} ausgibt. Das Programm wiederholt im Kern den Beweis des Satzes von der Division mit Rest:

```
divisionMitRest[a_, b_] := Module[ { a1 = Abs[a], b1 = b, q = 0, r},
  r = a1;
  While[r >= b1, q = q + 1; r = r - b1];
  If[a >= 0, Return[{q, r}], Return[{-q - 1, b - r}]]
 ]

divisionMitRest[-17, 5]
```

```
{-4, 3}
```

Behauptung:
Der Algorithmus endet nach endlich vielen Schritten und berechnet tatsächlich die gewünschte Zahlen.
Beweis:
Vor Eintritt in die While-Schleife gilt die Gleichung $a_1 = q * b_1 + r$ und die Ungleichung $r \geq 0$. Man überzeugt sich davon, dass ein Durchlauf durch die Schleife daran nichts ändert. Die beiden Bedingungen nennt man deshalb auch *Schleifen-Invarianten*. Das Verfahren terminiert, da durch die Anweisung $r = r - b_1$ die Kontrollbedingung $r \geq b_1$nach endlich vielen Schritten verletzt wird. Dann wird die Schleife verlassen, und es gilt: $0 \leq r < b_1$. ◆

Teilbarkeit, Primzahl

Eine ganze Zahl a ist teilbar durch eine ganze Zahl b ≠ 0, wenn der Rest der Division |a|:b gleich 0 ist. In diesem Fall heisst b auch Teiler von a.

Mathematica verfügt über die eingebaute Funktion Divisors, die zu einer ganzen Zahl sämtliche positiven Teiler in einer Liste ausgibt:

```
Divisors[12345]
```

```
{1, 3, 5, 15, 823, 2469, 4115, 12345}
```

Es folgt eine Definition des Begriffs Primzahl:

Eine ganze Zahl a heisst prim (oder Primzahl), wenn a ≥ 2 und für alle Teiler b von a gilt: |b| = 1 oder |b| = a.

Es ist nicht schwer, eine Funktion zu definieren, die diese Erklärung direkt wiedergibt (Übung!). Dabei sind die ganzen Zahlen *b* zwischen 1 und *a* zu durchmustern (bis zu welcher Schranke?) und abzubrechen, sobald *b* als Teiler erkannt ist. Das Verfahren ist zwar gedanklich einfach, aber überhaupt nicht effizient.

Die *Mathematica*-Funktion, die entscheidet, ob ihr Argument eine Primzahl ist, heißt PrimeQ:

```
PrimeQ[131171171]
```

```
True
```

2.2. Zwei Fassungen des Euklidischen Algorithmus

Iterative Fassung

Der Euklidische Algorithmus besteht im wesentlichen in dem Verfahren der Wechselwegnahme, wobei statt fortgesetzter Subtraktion Division mit Rest durchgeführt wird.

2.2.1. Satz

Seien a, b natürliche Zahlen mit a > b > 0. Dann wird durch $r_0 = a$, $r_1 = b$ und fortgesetzte Division mit Rest: $r_k = q_{k+1} r_{k+1} + r_{k+2}$ (k = 0, 1, 2, ...) eine streng-monoton abnehmende Folge $r_0, r_1, r_2,$ ... nicht-negativer ganzer Zahlen definiert, deren letztes nicht-verschwindendes Glied grösster gemeinsamer Teiler (ggT) von a und b ist.

Der Beweis wird hier nur in seinen wesentlichen Punkten rekapituliert:
1. Die Folge der Reste nimmt ab, da jeder neue Rest durch die Division mit dem unmittelbaren Vorgängerrest entsteht. Nach endlich vielen Schritten entsteht auf diese Weise der Rest 0.

2. Offenbar hat das Restepaar r_k, r_{k+1} dieselben Teiler wie das Restepaar r_{k+1}, r_{k+2}. Mithin sind die gemeinsamen Teiler von a, b dieselben wie die von r_n, 0 (wenn $r_n > 0$ und $r_{n+1} = 0$). Infolgedessen ist r_n der ggT von a und b. ◆

Bemerkung

Dem Beweis ist unmittelbar zu entnehmen, dass die Menge der gemeinsamen Teiler des jeweiligen Restepaars die Schleifeninvariante des Algorithmus ist (und damit der Algorithmus, wie behauptet, den ggT berechnet).

In seiner ursprünglichen Form ist der Euklidische Algorithmus ein iteratives Verfahren:

```
ggTit[a_, b_] := Module[{x = Abs[a], y = Abs[b]}, While[y > 0, {x, y} = {y, Mod[x, y]}];
  Return[x]
 ]
```

```
ggTit[18, 12]
```

```
6
```

Wird die While-Schleife verlassen, ist $y = 0$ und daher (Schleifeninvariante!): ggT(a, b) = ggT(x, y) = ggT(x, 0) = x.

Rekursive Fassung

Der Euklidische Algorithmus lässt sich auch rekursiv beschreiben. Dazu betrachte man eine beliebige Division-mit-Rest-Zeile: $a = qb + r$, $0 \le r < b$. Da a, b dieselben gemeinsamen Teiler haben wie b, r, gilt auch ggT(a, b) = ggT(b, r). Dieser Abarbeitungsschritt führt die Berechnung des ggTs von a und b auf eine um 1 Schritt kürzere ggT-Berechnung zurück. Er lässt sich daher unmittelbar als Rekursionsgleichung zur Definition der entsprechenden Funktion ggTrek verwenden:

```
ggTrek[a_, 0] := Abs[a];
ggTrek[a_, b_] := ggTrek[b, Mod[a, b]];
```

Es ist aufschlussreich, einmal das Rückwärtslaufen des Verfahrens an einem Beispiel zu verfolgen:

```
Trace[ggTrek[8, 13], ggTrek[_, _]] // TableForm
```

```
ggTrek[8, 13]
ggTrek[13, Mod[8, 13]]
ggTrek[13, 8]
ggTrek[8, Mod[13, 8]]
ggTrek[8, 5]
ggTrek[5, Mod[8, 5]]
ggTrek[5, 3]
ggTrek[3, Mod[5, 3]]
ggTrek[3, 2]
ggTrek[2, Mod[3, 2]]
ggTrek[2, 1]
ggTrek[1, Mod[2, 1]]
ggTrek[1, 0]
```

2.3. Der ggT als Vielfachensumme

Die wichtigste Folgerung aus dem Satz über den Euklidischen Algorithmus ist die Darstellbarkeit des ggTs zweier Zahlen als deren Vielfachensumme.

2.3.1. Lemma (Bachet)

Der grösste gemeinsame Teiler d zweier ganzer Zahlen a, b lässt sich als Vielfachensumme von a und b darstellen, d.h. es gibt x, y ϵ $\mathbb{Z}$ mit d = ax + by,

Beweis:

Wir zeigen die Behauptung statt für d gleich für alle beim Euklidischen Algorithmus auftretenden Reste r_k (k = 0, 1, 2, ...) durch vollständige Induktion (Abschnittinduktion).

Induktionsanfang (bei einer Abschnittinduktion nicht unbedingt erforderlich, hier aber aufschlussreich): Die ersten beiden Reste im Euklidischen Algorithmus entstehen noch nicht durch eine Division mit Rest: $r_0 = a = a \cdot 1 + b \cdot 0$, ferner $r_1 = b = a \cdot 0 + b \cdot 1$.

Induktionsschluss: Im k-ten Schritt entsteht r_k durch eine Division mit Rest: $r_{k-2} = q_{k-1} r_{k-1} + r_k$ (*). Induktionsannahme: r_{k-2} und r_{k-1} lassen sich als Vielfachensumme von a und b darstellen: $r_{k-2} = a x_{k-2} + b y_{k-2}, r_{k-1} = a x_{k-1} + b y_{k-1}$. Hieraus gewinnt man durch Auflösung der Gleichung (*) nach r_k eine Linearkombination für r_k:

$$r_k = a(x_{k-2} - q_{k-1} x_{k-1}) + b(y_{k-2} - q_{k-1} y_{k-1})$$

Unter diesen Resten ist der ggT von a und b, der sich mithin als Vielfachensumme von a und b darstellen lässt. ◆

Wir wollen aus dem Beweis dieser Folgerung einen Algorithmus gewinnen, der die gesuchten Vervielfacher x und y effektiv berechnet; nebenbei wird der ggT noch einmal berechnet. Das betreffende Verfahren nimmt die Zahlen a, b als Argumente auf und gibt die gewünschte Linearkombination in Form einer Liste {d, {x, y}} zurück. Der ggT d wird iterativ als letzter (nicht verschwindender) Rest im

Euklidischen Algorithmus ermittelt. Parallel dazu werden die gesuchten *x*, *y* bei jedem Verfahrensschritt aktualisiert. Der Beweis zu 2.3.1 zeigt, dass dabei jeweils zwei Paare zu verarbeiten sind: das Paar x_{k-2}, y_{k-2} (im untenstehenden Algorithmus bachet *u*, *v* genannt) und das Paar x_{k-1}, y_{k-1} (im Algorithmus *x*, *y* genannt). Aus ihnen werden im Folgeschritt wieder zwei Paare erzeugt: Das neue *u*, *v* ist das aktuelle *x*, *y*, und das neue *x*, *y* ist gerade das der Linearkombination für r_k zu entnehmende Wertepaar $u - q\,x$, $v - q\,y$. Selbstredend bedeuten *q* und *r* in jedem Schritt den Quotienten und den Rest der Division. Die Initialisierung der Variablen erklärt sich unmittelbar aus dem Induktionsanfang.

```
bachet[a_, b_] := Module[{u = 1, v = 0, x = 0, y = 1, q, r, a0 = a, b0 = b},
  q = Quotient[a0, b0]; r = Mod[a0, b0];
  While[r != 0,
   {a0, b0} = {b0, r};
   {u, x} = {x, u - q * x};
   {v, y} = {y, v - q * y}; {q, r} = {Quotient[a0, b0], Mod[a0, b0]}]; Return[{b0, {x, y}}]
 ]
```

```
bachet[17, 5]
```

```
{1, {-2, 7}}
```

Die eingebaute *Mathematica*-Funktion ExtendedGCD liefert das Ergebnis in derselben Form:

```
ExtendedGCD[17, 5]
```

```
{1, {-2, 7}}
```

2.4. Berechnung des kgV

Der folgende Algorithmus berechnet das kleinste gemeinsame Vielfache (kgV) von *a* und *b*:

```
kgV[a_, b_] := Module[{x = a, y = b}, While[x != y, If[x < y, x = x + a, y = y + b]];
  Return[x]
 ]
```

```
kgV[6, 8]
```

```
24
```

Der Beweis bleibt hier zur Übung.
Man überlege sich dazu die Fragen: Inwiefern ist die Existenz des kgV gesichert? Warum terminiert die While-Schleife? Was passiert, wenn der Wert *y* (anstelle von *x*) zurückgegeben wird?

Zwischen dem kgV und dem ggT besteht ein direkter Zusammenhang:

$$\text{ggT}(a, b) \times \text{kgV}(a, b) = a \times b \qquad (*)$$

Mit dieser Gleichung kann man die Berechnung des kgV auf die Berechnung des ggT zurückführen. Einen einfachen Beweis von (*) findet man z.B. in [A. Bartholomé; J. Rung; H. Kern: *Zahlentheorie für Einsteiger*. Vieweg: Braunschweig; Wiesbaden 1995, S. 51].

Ein geschicktes Verfahren, mit dem sich ggT und kgV beide in *einem* Rechengang ermitteln lassen, stammt von Stanley Gill. Die hier angegebene Funktion gibt ihr Ergebnis als Liste {ggT, kgV} zurück:

```
gill[a_, b_] := Module[{x = a, y = b, u = a, v = b},
  While[x != y,
   If[x < y, y = y - x; v = v + u, x = x - y; u = u + v]];
  Return[{x, Quotient[u + v, 2]}]
 ]
```

```
gill[8, 6]
```

```
{2, 24}
```

Für einen Beweis dieses Algorithmus vgl. [A. Engel: *Mathematisches Experimentieren mit dem PC*. Klett Schulbuchverlag: 1. Aufl. Stuttgart 1991, S. 35].

3. Darstellung von Zahlen

3.1. B-adische Stellenwertsysteme

Bezeichnungssysteme für Zahlen

Im Umgang mit Zahlen greifen wir auf Zahlennamen (Zahlwörter) wie *eins*, *zwei*, *elf*, *hundert* oder spezielle Zahlsymbole wie 1, 2, 3, usw. zurück. Diese Bezeichnungen sind historisch und in ihrer Ausprägung mehr oder weniger willkürlich. Für die Arithmetik benötigen wir darüberhinaus ein Bezeichnungssystem, das ...

(1) auf einer einfachen (leicht durchschaubaren) Systematik aufbaut;
(2) im Prinzip alle (unendlich vielen) natürlichen Zahlen abbildet und dabei ...
(3) nicht zu schnell zu langen Bezeichnungen führt;
(4) die Grundrechenarten vergleichsweise bequem durchzuführen gestattet.

Das einfache Hintereinanderschreiben von Einheiten

|, ||, |||, ||||, |||||, ||||||, |||||||, ...

erfüllt zwar die Forderungen (1) und (2), jedoch bei weitem nicht (3) und (4).

Die römische Zahlschrift

... notiert ebenfalls zunächst Einheiten: I, II, III, mischt in diese Methode dann aber Subtraktions- und Additionsvorschriften hinein (IV für 4, V für 5, VI für 6 usw.), wodurch die Darstellung größerer Zahlen und vor allem die grundlegenden Rechenarten rasch unübersichtlich werden.

Das Stellenwertsystem bzw. die ihm zugrundeliegende Idee des iterierten Bündelns

... liefert eine Lösung, mit der sich die Forderungen (1) bis (4) weitgehend erfüllen lassen. In Ansätzen war das Bündeln bereits bei alten Kulturvölkern ausgeprägt (z.B. bei den Maya zu Grössen von 20; bei den Babyloniern zu 12 und 60, die noch in unserer heutigen Zeit- und Winkelmessung vorkommen). Die *Stellenwertsystematik* wurde ca. 500 Jahre n. Chr. in Indien erfunden. Dabei wurde die Null als Zahl "anerkannt" und mit einem eigenen Symbol 0 für "die Leere" in das System eingeführt, um ausdrücken zu können, dass zu einer bestimmten Bündelgrösse *keine* Bündel vorliegen. Das Wort "Ziffer" stammt aus dem Arabischen und bedeutet Null. Die Araber haben das Stellenwertsystem (zur Basis bzw. Bündelgrösse zehn) nach Westen überliefert.

B-adische Darstellung ganzer Zahlen

Wir bezeichnen die Bündelgrösse (auch *Basis* genannt) mit B, wobei $B \geq 2$ ganz. Der Wert $B = 1$ ist uninteressant, weil er nicht zu echtem Bündeln führt. Im Dezimalsystem ist $B = 10$. Man benötigt B Zahlwörter – die sogenannten B-adischen Ziffern – zur Bezeichnung der möglichen Bündelanzahlen, angefangen von Null (0) bis zur größten Zahl unterhalb der Bündelgrösse ($B - 1$).
Das B-adische Stellenwertsystem beruht auf der Tatsache, dass sich eine natürliche Zahl stets als Wert eines Polynoms mit B-adischen Ziffern als Koeffizienten zum Argument B darstellen lässt.

3.1.1. Satz

Jede natürliche Zahl a ist eindeutig in der Form $a = \sum_{i=0}^{n} z_i B^i$ darstellbar, wobei $0 \leq z_i < B$ für $i = 1, 2, ..., n$.

$\sum_{i=0}^{n} z_i B^i$ wird in diesem Zusammenhang abgekürzt durch : $(z_n\, z_{n-1} \ldots z_1\, z_0)_B$

Der Bezug auf die Basis B kann auch fortfallen, wenn keine Missverständisse zu befürchten sind (etwa im Dezimalsystem).

Der (hier nicht im Detail zu wiederholende) Beweis verläuft induktiv, indem eine Division mit Rest ($a = qB + r$) durchgeführt und eine nach Induktionsannahme verfügbare B-adische Darstellung für das Ganze von $a : B = q\,(< a)$ in eine geeignete Darstellung für a "hochgerechnet" wird.
Dieser Idee folgend liefert die wiederholt ausgeführte Division mit Rest (bei stets gleichem Divisor B) auch die Ziffernfolge für die B-adische Darstellung von a.

Solange $a > 0$, führe aus:
1. Dividiere a durch B und schreibe den Rest auf.
2. Ersetze a durch das Ganze der Division $a : B$.

Wenn wir die Reste sukzessive an den Anfang einer Liste schreiben, erhalten wir schließlich die B-adische Darstellung von a in der Form: $\{z_n, z_{n-1}, ..., z_1, z_0\}$.
Die folgende *Mathematica*-Funktion berechnet diese Liste zu den Argumenten a und B:

```
ziffern[a_, B_] := Module[{q = Abs[a], z, ziff = {}}, While[q > 0,
    z = Mod[q, B];
    PrependTo[ziff, z];
    q = Quotient[q, B]];
  Return[ziff]
 ]
```

```
ziffern[243, 5]
```

```
{1, 4, 3, 3}
```

Dasselbe leistet die eingebaute Funktionen IntegerDigits[*n*, *b*]:

```
IntegerDigits[243, 5]
```

```
{1, 4, 3, 3}
```

Das Horner-Schema

Ist umgekehrt eine *B*-adische Darstellung $(z_n\, z_{n-1}\, ...\, z_1\, z_0)_B$ gegeben, so ist die von ihr repräsentierte ganze Zahl *a* nichts anderes als der Wert des Polynoms $p(X) = \sum_{i=0}^{n} z_i\, X^i$ an der Stelle $X = B$, also: $a = p(B)$.
Beispiel: $(1433)_5 = 1\times5^3 + 4\times5^2 + 3\times5^1 + 3\times5^0 = 243 = (243)_{10}$
Diese Berechnung erfordert 3 Additionen und 6 Multiplikationen. Durch Ausklammern lassen sich 3 Multiplikationen einsparen:

$$1\times5^3 + 4\times5^2 + 3\times5^1 + 3\times5^0 = (5^2 + 4\times5 + 3)\times5 + 3 = ((1\times5 + 4)\times5 + 3)\times5 + 3$$

Wertet man die Klammern schrittweise von innen beginnend aus, so erkennt man die Umkehrung des Bündelungsverfahrens (fortgesetzte Division durch die Basis *B*). In seiner allgemeinen Form heißt dieser Algorithmus *Horner-Schema*.

Hier zunächst eine umgangssprachliche Beschreibung des Horner-Schemas:
Gegeben: Polynom $p(X)$ mit den Koeffizienten $z_n, z_{n-1}, ..., z_0$; Argument *B*
Der Wert $h = p(B)$ wird wie folgt berechnet:

1. Ersetze *h* durch z_n.
2. Für $i = n - 1$ bis 0: Ersetze *h* durch $h\, B + z_i$.

Um die Zählschleife wiederzugeben, muss zuvor die Länge der Koeffizientenliste (der Grad des Polynoms) berechnet werden:

```
horner[koeffliste_, x_] := Module[
  {k = Length[koeffliste], h = koeffliste[[1]]}, Do[h = h x + koeffliste[[i]], {i, 2, k}];
  Return[h]
 ]
```

```
horner[{1, 4, 3, 3}, 5]
```

```
243
```

Spezielle Basen

Die Basis 10 des Dezimalsystems geht vermutlich auf die Anzahl der Finger zurück. Daneben spielen heute, vor allem in der Computertechnik, die Basen 2 und 16 eine wichtige Rolle.

1. *Dyadische* (oder *binäre*) Darstellungen, d.h. solche zur Basis 2, kommen mit den Ziffern 0 und 1 aus und lassen sich daher durch zweiwertige physikalische Zustände ("Strom fließt" – "Strom fließt nicht") modellieren. Natürliche Zahlen schreiben sich als *Bitvektoren* (0-1-Folgen):

```
ziffern[243, 2]
```

```
{1, 1, 1, 1, 0, 0, 1, 1}
```

Bitvektoren sind vor allem für technische Zwecke von Interesse; mit ihnen werden die Grundrechenoperationen überaus einfach, allerdings werden die Zahlwörter im Mittel mehr als dreimal so lang wie im Dezimalsystem.

2. Fasst man jeweils vier dyadische Ziffern zusammen und ersetzt sie durch ein einziges Symbol, so gelangt man zur Hexadezimal-Darstellung mit der Basis $B = 2^4 = 16$:

0000 = 0	0001 = 1	0010 = 2	0011 = 3
0100 = 4	0101 = 5	0110 = 6	0111 = 7
1000 = 8	1001 = 9	1010 = A	1011 = B
1100 = C	1110 = E	1111 = F	

Beispiel: $(1111\ 0011)_2 = (F3)_{16} = \$F3 = 243$
Die Schreibweise mit vorangestelltem $-Zeichen ist in der Computertechnik gebräuchlich.
Wir machen die Probe mit dem Horner-Schema:

```
F = 15;
horner[{F, 3}, 16]
```

```
243
```

B-adische Entwicklung von Brüchen

Im folgenden soll die B-adische Entwicklung rationaler Zahlen studiert werden. Ohne Einschränkung werden dazu nur positive gekürzte Brüche $\frac{u}{v} < 1$ betrachtet, d.h. es ist $0 < u < v$ mit teilerfremden ganzen Zahlen *u*, *v*.

Eine solche B-adische Entwicklung (wir sagen auch: Darstellung als Systembruch) kann

1. endlich
2. rein-periodisch
3. gemischt-periodisch

sein.

Zunächst wollen wir die nähere Untersuchung der dafür notwendigen und hinreichenden Kriterien außer Betracht lassen. Daher muss einem Algorithmus, der $\frac{u}{v}$ in einen Systembruch $0, z_1\, z_2\, z_3 \ldots$ entwickelt, die Anzahl s der gewünschten Nachkommastellen bekannt sein. Damit ist für eine Abbruchbedingung gesorgt.

Offenbar erfüllen die Ziffern z_k die Division-mit-Rest-Gleichungen:

$u_{k-1}\, B = z_k\, v + u_k$, wobei $0 \le u_k < v$

Für $k = 1$setzen wir $u_0 = u$. Damit ergibt sich unmittelbar ein iteratives Verfahren für eine Funktion, die die ersten s Ziffern hinter dem Komma (oder im Fall $B = 10$ Dezimalpunkt, allgemein: B-adisches Trennzeichen) in Form einer Liste berechnet und ausgibt:

```
systembruch1[u_, v_, B_, s_] := Module[{u0 = u, ziffer, m, ziff = {}}, Do[m = u0 * B;
    ziffer = Quotient[m, v];
    u0 = Mod[m, v];
    AppendTo[ziff, ziffer], {i, 1, s}];
  Return[ziff]
 ]
```

```
systembruch1[1, 7, 10, 11]
```

```
{1, 4, 2, 8, 5, 7, 1, 4, 2, 8, 5}
```

Offenbar hat der Bruch $\frac{1}{7}$eine rein-periodische Entwicklung mit dem wiederkehrenden Ziffernblock 142857 der Länge 6.

Natürlich werden auch endliche und gemischt-periodische Systembrüche (empirisch) identifiziert:

```
systembruch1[1, 5, 10, 4]
```

```
{2, 0, 0, 0}
```

```
systembruch1[1, 6, 10, 5]
```

```
{1, 6, 6, 6, 6}
```

Der folgende Entwicklungsalgorithmus benutzt die bereits früher definierte Funktion ziffern[.,.] und erscheint daher kompakter:

```
systembruch2[u_, v_, B_, s_] :=
  ziffern[Quotient[u*B^s, v], B]
```

```
systembruch2[1, 7, 10, 11]
```

```
{1, 4, 2, 8, 5, 7, 1, 4, 2, 8, 5}
```

Begründung:

Es ist $\frac{u}{v}\, B^s = z_1\, z_2 \ldots z_s,\, z_{s+1} \ldots$, wobei die z_k $(k = 1,\ \ldots,\ s)$ die ersten s B-adischen Nachkommaziffern von $\frac{u}{v}$bedeuten. Der Wert der links vom Komma stehenden Zifferngruppe ist somit das Ganze der Division von $u\, B^s$ durch v. ◆

Arithmetische Kennzeichnung der rationalen Systembrüche

Im folgenden soll untersucht werden, wie die B-adische Entwicklung eines (als gekürzt angenommenen) Bruchs $\frac{u}{v}$ $(0 < u < v)$ aussehen kann.

3.1.2. Satz

$\frac{u}{v}$ besitzt eine endliche B-adische Entwicklung genau dann, wenn v Teiler einer Potenz B^s, $s \ge 1$, ist.

Beweis:

1. Sei $\frac{u}{v} = (0,\ z_1\, z_2 \ldots z_s)_B = \sum_{k=1}^{s} z_k\, B^{-k}$. Es ist daher $\frac{u}{v}\, B^s$ ganz und v ein Teiler von B^s wegen der Teilerfremdheit von u, v.

2. Sei umgekehrt v Teiler von B^s, d.h. $v\, q = B^s$ für ein ganzes $q > 0$. Damit schreiben wir $\frac{u}{v} = \frac{u\, q}{B^s}$. Ist $u\, q < B$ (d.h. eine B-adische Ziffer), so sind wir fertig. Andernfalls machen wir Division durch B mit Rest: $u\, q = q_s\, B + r_s$, $0 \le r_s < B$, und erhalten: $\frac{u}{v} = \frac{q_s}{B^{s-1}} + \frac{r_s}{B^s}$. Solange der Zähler im ersten Summanden größer oder gleich B ist, setzt man das Verfahren sinngemäß fort. Nach endlich vielen Schritten sind sämtliche Zähler Werte B-adischer Ziffern. ◆

Bemerkung

Dem Beweis ist zu entnehmen: Das kleinste s, für das v Teiler von B^s ist, gibt die Anzahl der Ziffern hinter dem Komma (B-adischen

Trennzeichen) an.

Wir nehmen nun eine unendliche B-adische Entwicklung an:

$$\frac{u}{v} = (0,\ z_{1\ldots}\ z_m\ z_{m+1}\ \ldots\ z_n\ z_{n+1}\ \ldots)_B$$

Erinnern wir uns daran, dass sich die Ziffern aus sukzessiven Divisionen mit Rest ergeben: $u_{k-1}\,B = z_k\,v + u_k$, wobei $0 \le u_k < v$ und $k = 1,\ 2,\ \ldots,\ u_0 = u$. (Wichtige Anmerkung: Diese wesentliche Tatsache ist bei der B-adischen Entwicklung irrationaler Zahlen nicht gegeben!) Da die Reste u_k nur endlich viele Werte ($< v$) annehmen können, muss es Indizes m, n mit $m < n$ geben, so dass $u_m = u_n$. Ohne Einschränkung denken wir uns m minimal gewählt. Aus den beiden Gleichungen $u_m\,B = z_{m+1}\,v + u_{m+1}$ und $u_n\,B = z_{n+1}\,v + u_{n+1}$ ergibt sich sofort (Eindeutigkeit der Division mit Rest!): $z_{m+1} = z_{n+1}$ und $u_{m+1} = u_{n+1}$. Aus der zweiten Gleichung für die Reste folgt in derselben Weise: $z_{m+2} = z_{n+2}$, usw. Allgemein: $z_{m+i} = z_{n+i}$ für alle $i \ge 1$. Wir sehen also, dass der Ziffernblock $z_{m+1}\ \ldots\ z_n$ sich periodisch wiederholt.
Die Anzahl λ der Ziffern im Block $z_{m+1}\ \ldots\ z_n$ heißt Periodenlänge: $\lambda = n - m$. Da für die Reste sogar $u_k \in \{1,\ \ldots,\ v-1\}$ gilt – der Wert 0 führt zu einem endlichen Systembruch –, muss in einer Sequenz von v Resten (und d.h. auch: zugehörigen Ziffern) eine Wiederholung stattfinden. Das zeigt: $\lambda < v$.
Nun erhalten wir die Darstellung:
$\frac{u}{v} = (0,\ z_1\ \ldots\ z_m)_B + (0,\ 0\ \ldots \times 0\ \overline{z_{m+1}\ \ldots\ z_n})_B = (0,\ z_1\ \ldots\ z_m)_B + \frac{1}{B^m}\cdot(0,\ \overline{z_{m+1}\ \ldots\ z_n})_B$
Wie üblich wird der periodische Ziffernblock überstrichen notiert. Zwischen Komma und Ziffernblock stehen m Nullen.

An dieser Stelle ist es zweckmäßig, einen Hilfssatz einzuschalten, dem sich entnehmen lässt, wie ein rein-periodischer Systembruch auszuwerten ist:

3.1.3. Lemma

$$(0,\ \overline{z_1\ \ldots\ z_s})_B = (0,\ z_1\ \ldots\ z_s)_B \cdot \frac{B^s}{B^s - 1}$$

Beweisskizze:

Für jede Ziffer z_k ist folgende geometrische Reihe auszuwerten: $z_k\left(\frac{1}{B^k} + \frac{1}{B^{k+s}} + \frac{1}{B^{k+2s}} + \ldots\right) = \frac{z_k}{B^k}\sum_{j=0}^{\infty}\left(\frac{1}{B^s}\right)^j = \frac{z_k}{B^k}\cdot\frac{1}{1-\frac{1}{B^s}}$.

Aufsummieren für $k = 1,\ \ldots,\ s$ ergibt die gewünschte Behauptung. ◆

Bemerkung

Nach 3.1.3 erhalten wir $\left(0,\ \overline{B-1}\right)_B = (0,\ B-1)_B \cdot \frac{B}{B-1} = 1$. Aus diesem Grunde wird $\left(0,\ \overline{B-1}\right)_B$ nicht als *echter* periodischer Systembruch betrachtet.

Mit diesem Rüstzeug lässt sich die B-adische Entwicklung von $\frac{u}{v}$ nun wie folgt weiter verarbeiten:

$$\begin{aligned}\frac{u}{v} &= (0,\ z_1\ \ldots\ z_m)_B + \frac{1}{B^m}\cdot(0,\ \overline{z_{m+1}\ \ldots\ z_n})_B \\ &= (0,\ z_1\ \ldots\ z_m)_B + (0,\ z_{m+1}\ \ldots\ z_n)_B \cdot \frac{B^{n-m}}{B^{n-m}-1}\cdot\frac{1}{B^m}\end{aligned}$$

Nun multiplizieren wir diese Gleichung zunächst mit B^m. Man beachte, dass $B^m\cdot(0,\ z_1\ \ldots\ z_m)_B$ ganz ist. Anschließende Multiplikation mit $B^{n-m} - 1$ zeigt dann, dass $\frac{u}{v}\,B^m(B^{n-m} - 1)$ eine ganze Zahl ist. Ausgehend von diesem wichtigen Zwischenergebnis sollen nun die beiden noch verbleibenden Fälle einer rein-periodischen und einer gemischt-periodischen B-adischen Entwicklung näher untersucht werden.

Offenbar bedeutet $m = 0$, dass eine *rein-periodische* Entwicklung vorliegt. Da u und v teilerfremd sind, muss v Teiler von $B^n - 1$ sein, d.h. $v\,q = B^n - 1$ für ein geeignetes $q \in \mathbb{N}$. Es lässt sich daher 1 als Vielfachensumme von v und B darstellen: $1 = B\cdot B^{n-1} + (-q)\,v$, d.h. v, B sind teilerfremd.
Die Teilerfremdheit von v, B ist auch ein hinreichendes Kriterium dafür, dass eine rein-periodische Entwicklung vorliegt. Infolgedessen haben wir

3.1.4. Satz

$\frac{u}{v}$ besitzt eine rein-periodische B-adische Entwicklung genau dann, wenn v, B teilerfremd sind.

Beweis:
Die Richtung "$\frac{u}{v}$ rein-periodisch $\Longrightarrow$ v, B teilerfremd" wurde bereits gezeigt. Wir setzen nun umgekehrt die Teilerfremdheit von v und B voraus und zeigen, dass der B-adische Systembruch von $\frac{u}{v}$ rein-periodisch ist. Das Zwischenresultat der obigen Analyse lautet: $\frac{u}{v}\,B^m(B^{n-m} - 1)$ ist ganz. Nun ist v teilerfremd zu u, zu B und daher auch zu B^m. Damit erhalten wir $v\,q = B^{n-m} - 1$ für ein geeignetes ganzes q. Wir setzen $\lambda = n - m$ und erhalten: $\frac{u}{v} = u\,q\,\frac{1}{B^\lambda - 1} = \frac{u\,q}{B^\lambda}\cdot\frac{B^\lambda}{B^\lambda - 1}$. Der erste der beiden Faktoren lässt sich nach Satz 3.1.2 (vgl. die zugehörige Bemerkung) als endlicher Systembruch $(0,\ z_1\ \ldots\ z_\lambda)_B$ schreiben. Lemma 3.1.3 liefert damit: $\frac{u}{v} = (0,\ \overline{z_1\ \ldots\ z_\lambda})_B$. ◆

Ist schließlich v weder Teiler einer Potenz von B noch teilerfremd zu B, so kann m nicht gleich 0 sein, d.h. in diesem Fall liegt eine gemischt-periodische Entwicklung vor:

$$\frac{u}{v} = (0,\ z_{1\ldots}\ z_m\ z_{m+1}\ \ldots\ \overline{z_n})_B$$

Auch dieser Fall lässt sich durch eine einfache arithmetische Eigenschaft charakterisieren:

3.1.5. Satz

$\frac{u}{v}$ besitzt eine gemischt-periodische B-adische Entwicklung genau dann, wenn $v = v_1 v_2$ derart, dass $v_1$eine Potenz von B teilt und v_2, B teilerfremd sind.

Beweis:
1. Wir setzen $m > 0$ (gemischt-periodische Entwicklung) voraus und definieren $v_1 := \text{ggT}(v, B^m)$. Es ist $v_1 > 1$, da sonst v, B teilerfremd wären und Satz 3.1.4 zufolge $m = 0$ sein müsste (entgegen der Voraussetzung). Definitionsgemäß ist v_1 Teiler von B^m, d.h. $v_1 {v_1}^* = B^m$ für ein ${v_1}^*$. Schreiben wir $v = v_1 v_2$, so gilt: v_2, B sind teilerfremd. Denn einerseits hat $\frac{u}{v} B^m$ eine rein-periodische Entwicklung; andererseits gilt $\frac{u}{v} B^m = \frac{u {v_1}^*}{v_2}$, wobei der Bruch auf der rechten Seite dieser Gleichung teilerfremde Zähler und Nennen besitzt. Satz 3.1.4 liefert demnach die Teilerfremdheit von v_2 und B.
2. Sei nun umgekehrt eine Zerlegung $v = v_1 v_2$ mit den im Satz genannten Eigenschaften gegeben; etwa: $v_1 {v_1}^* = B^s$. Damit ergibt sich: $\frac{u}{v} = \frac{u}{v_1 v_2} = \frac{u {v_1}^*}{v_1 {v_1}^* v_2} = \frac{1}{B^s} \cdot \frac{u {v_1}^*}{v_2}$. Als Faktor von v ist v_2 zu u teilerfremd. Auch v_2 und ${v_1}^*$ haben keinen gemeinsamen Teiler. Anderenfalls wären nämlich v_2 und B^s und damit auch v_2 und B nicht teilerfremd (Begründung zur Übung!). Satz 3.1.4 liefert für $\frac{u {v_1}^*}{v_2}$demnach eine rein-periodische Entwicklung. Der vorgeschaltete Bruch $\frac{1}{B^s}$bewirkt, dass eine gewisse positive Anzahl von Ziffern, und zwar höchstens s, aus dem sich wiederholenden Ziffernblock herausgezogen werden. Der zugehörige Systembruch ist also insgesamt gemischt-periodisch. ◆

Mit den in 3.1.2, 3.1.4 und 3.1.5 gemachten Aussagen sind die B-adischen Entwicklungen rationaler Zahlen $\frac{u}{v}$ vollständig klassifiziert und durch arithmetische Eigenschaften charakterisiert. Es fällt auf, dass der Nenner u dabei keinen Einfluss auf den Typ des Systembruchs hat.
Die früheren Algorithmen systembruch1 und systembruch2 lassen sich verbessern, wenn man detailliertere Kenntnisse über die Periodenlänge λ heranzieht. Die in unserer früheren Überlegung nebenbei gewonnene Ungleichung $\lambda < v$ reicht dazu nicht aus. Eine etwas tieferliegende Erkenntnis enthält der folgende Satz:

3.1.6. Satz

1. λ ist die kleinste natürliche Zahl s, für die gilt: v ist Teiler von $B^s - 1$.
2. λ ist ein Teiler von $\varphi(v)$ (Eulersche Phi-Funktion).

Auf den Beweis und die Anwendung dieser Aussagen kann in diesem Rahmen nicht eingegangen werden. Vgl. [A. Engel: *Elementarmathematik vom algorithmischen Standpunkt*. Klett Verlag: Stuttgart 1977, S.55-56].

3.2. Kettenbrüche

(Hier ausgespart.)

4. Numerische Algorithmen

4.1. Nullstellen reeller Funktionen

Vorbemerkung

Ist f eine Funktion $\mathbb{R} \to \mathbb{R}$, so heißen alle $x \in \mathbb{R}$, für die $f(x) = 0$ gilt, *Nullstellen* von f. Das Lösen von Gleichungen lässt sich formal als Nullstellenbestimmung auffassen. Im folgenden sollen einige elementare Methoden zur numerischen Berechnung von Nullstellen erläutert werden.

Halbierungsmethode

Diese Methode eignet sich für Funktionen f, die in einem Intervall $I = [a, b]$ stetig sind und zwischen den Endpunkten a und b ihr Vorzeichen wechseln, d.h. $f(a)\, f(b) < 0$. Nach dem Zwischenwertsatz gibt es in diesem Fall ein $x_0 \in I$ derart, dass $f(x_0) = 0$. Ein solches x_0 kann beliebig genau angenähert werden, indem man durch sukzessive Halbierung das Intervall um die Stelle, an der das Vorzeichen wechselt, zusammenzieht. Mit einer (kleinen positiven) Fehlerschranke ε wird die Abbruchbedingung formuliert:

1. Ersetze x durch $\frac{a+b}{2}$;
2. Falls $f(x) = 0$, gehe nach 5;
3. Falls $f(a)\, f(x) > 0$, ersetze a durch x, sonst ersetze b durch x;
4. Falls $|a - b| > \varepsilon$, gehe nach 1;
5. Ausgabe: $\frac{a+b}{2}$.

Diese Befehlsfolge lässt sich noch etwas verbessern durch Benutzung einer While-Schleife anstelle von Sprungbefehlen, Verzicht auf den unwahrscheinlichen Fall $f(x) = 0$, multiplikationsfreie Umschreibung der Vorzeichengleichheit:

Ersetze x durch $\frac{a+b}{2}$;
Solange $|a - b| > \varepsilon$:

Falls $f(a)$, $f(x)$ vorzeichengleich sind, ersetze a durch x, sonst ersetze b durch x;

Ersetze x durch $\frac{a+b}{2}$;

Ausgabe: x.

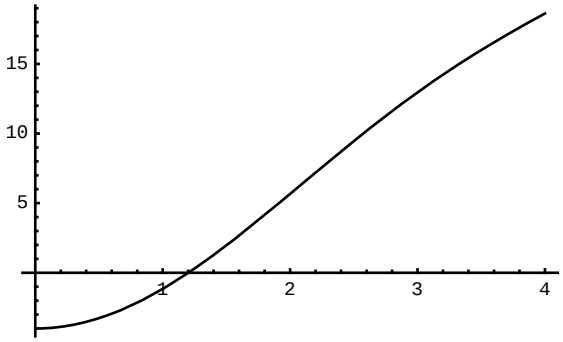
In dieser Form können wir den Algorithmus direkt in die Sprache von *Mathematica* übertragen. (Das boolesche Argument verbose entscheidet darüber, ob die in der Schleife berechneten Näherungswerte während des Programmlaufs ausgedruckt werden sollen.)

```
halbierungsmethode[f_, a_, b_, epsilon_, verbose_] := Module[{a1 = a, b1 = b, x},
  While[Abs[a1 - b1] > epsilon,
   x = (a1 + b1) / 2;
   If[verbose, Print[N[x]]];
   If[Sign[f[a1]] == Sign[f[x]], a1 = x, b1 = x]];
  Return[N[x]]
 ]
```

Beispiel: $f_1(x) = x^2 - 4\cos x$

```
f1[x_] := x^2 - 4 * Cos[x]
```

```
Plot[f1[x], {x, 0, 4}];
```




```
halbierungsmethode[f1, 1, 3, 0.0001, True]
```

```
2.
1.5
1.25
1.125
1.1875
1.21875
1.20313
1.19531
1.19922
1.20117
1.20215
1.20166
1.20142
1.20154
1.2016
1.2016
```

Dazu der Vergleich mit *Mathematica*'s eingebauter Funktion FindRoot:

```
FindRoot[f1[x] == 0, {x, 1}]
```

```
{x → 1.20154}
```

Bemerkung

Der Algorithmus berechnet die Nullstelle nur näherungsweise! Nach n Schritten ist der Fehler $\leq \frac{b-a}{2^{n+1}}$(einfacher Beweis bleibt als Übungsaufgabe).

Sekantenmethode

Das Verfahren ist der Halbierungsmethode sehr ähnlich. Es wird ebenfalls ein Intervall $I = [a, b]$ betrachtet. Die Nullstellen von f können aber auch außerhalb von I liegen, und die Funktion f muss somit als überall stetig vorausgesetzt werden. Im Unterschied zur Halbierungsmethode wird im Iterationsschritt x nicht als Intervallmittelpunkt $\frac{a+b}{2}$gewählt, sondern als Schnittpunkt der durch die Punkte $(a,\ f(a))$ und $(b,\ f(b))$ bestimmten Sekante mit der x-Achse.

Sekante, x-Achse und die Ordinatenstrecken über a und b bilden eine Strahlensatz-Figur, der man ohne weiteres die folgende Proportion entnimmt: $\frac{x-a}{-f(a)} = \frac{b-x}{f(b)}$, und hieraus:

(*) $\qquad x = \frac{a\,f(b)-b\,f(a)}{f(b)-f(a)}$.

Es folgt zunächst eine halbformale Beschreibung des Verfahrens:

Solange $|\,a - b\,| > \varepsilon$:
  Ersetze x durch den Ausdruck in Zeile (*);
  Ersetze a durch b;
  Ersetze b durch x;
Ausgabe: x

Bemerkungen

1) Häufig ist die Sekantenmethode schneller als die Halbierungsmethode, da in die Berechnung des "Zwischenpunkts" der Funktionsverlauf stärker eingeht. Bestimmte Verläufe machen das Verfahren aber langsamer, etwa wenn die Nullstelle am Rand des Intervalls liegt und der Funktionsgraph dort steil ansteigt.

2) Anders als bei der Halbierungsmethode kann durch die Schleife die Relation $a < b$ umgekehrt werden.

Der Algorithmus lässt sich nun als *Mathematica*-Funktion notieren:

```
sekantenmethode[f_, a_, b_, epsilon_, verbose_] := Module[{a1 = a, b1 = b, c, d, x},
  While[Abs[a1 - b1] > epsilon,
   c = f[a1]; d = f[b1];
   x = (a1 * d - b1 * c) / (d - c);
   If[verbose, Print[N[x]]];
   a1 = b1; b1 = x];
  Return[N[x]]
 ]
```

Wir führen die Sekantenmethode an unserem früheren Beispiel aus:

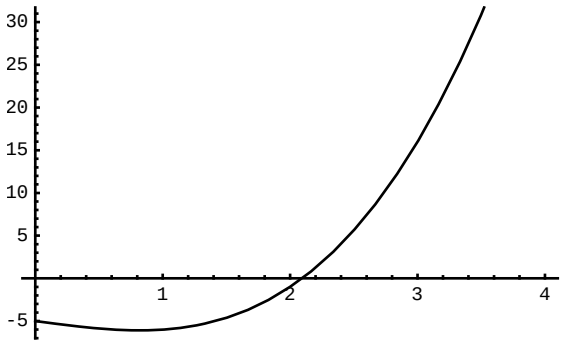
```
sekantenmethode[f1, 1, 3, 0.0001, True]
```

```
1.16446

1.19579

1.2016

1.20154

1.20154
```

In diesem Fall wird die Nullstelle in 4 Schritten genauer berechnet als mit der Halbierungsmethode, die 15 Schritt benötigte.
Wir experimentieren mit einem anderen Beispiel: $f_2(x) = x^3 - 2\,x - 5$

```
f2[x_] := x^3 - 2 * x - 5
```

```
Plot[f2[x], {x, 0, 4}];
```




```
halbierungsmethode[f2, 1, 3, 0.0001, True]
```

```
2.
2.5
2.25
2.125
2.0625
2.09375
2.10938
2.10156
2.09766
2.0957
2.09473
2.09424
2.09448
2.0946
2.09454
2.09454
```

```
sekantenmethode[f2, 1, 3, 0.0001, True]
```

```
1.54545
1.85916
2.20035
2.0798
2.0937
2.09456
2.09455
2.09455
```

Der Vergleich mit *Mathematica*'s eingebauter Funktion zeigt auch hier, dass die Sekantenmethode in weniger Schritten ein genaueres Ergebnis liefert. Allerdings ist das Verfahren auch unsicherer, da x nicht im betrachteten Intervall $[a, b]$ liegen muss.

```
FindRoot[f2[x] == 0, {x, 1}]
```

$$\left\{\frac{34\,317}{16\,384} \to 2.09455\right\}$$

Newton-Verfahren

Das sog. Newton-Raphson-Verfahren könnte man auch *Tangentenmethode* nennen. Die Funktion *f*, deren Nullstelle berechnet werden soll, muss differenzierbar sein. Der Punkt *a* befinde sich "in der Nähe" der Nullstelle. Wir legen in $(a,\ f(a))$ die Tangente an den Graphen von *f* und wählen ihren Schnittpunkt *x* mit der x-Achse als neue Näherung.

Die Steigung der Tangente ist der Quotient aus Ordinatenabschnitt und Abstand der Punkte *a* und *x* auf der x-Achse: $f'(a) = \frac{f(a)}{a-x}$. Löst man diese Gleichung nach *x* auf, so ergibt sich:

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

Die Konvergenz dieses Verfahrens übertrifft die Sekantenmethode im allgemeinen deutlich. Allerdings kann das Newton-Verfahren auch scheitern, z.B. wenn die Nullstelle Wendepunkt von *f* ist oder wenn *a* in der Nähe eines lokalen Extremums liegt. (Auf die Betrachtung der Konvergenzbedingungen und der Reichweite des Verfahrens wird zugunsten einer "experimentellen" Handhabung verzichtet.)

Bei der Formulierung des Algorithmus in der Sprache von *Mathematica* macht man vorteilhaften Gebrauch von dem bereits eingebauten symbolischen Differentialkalkül, der es uns erlaubt, die Ableitung von *f* nach *x* einfach "hinzuschreiben": *D*[*f*[*x*], *x*].

```
newton[f_, a_, epsilon_, verbose_] := Module[{x = a, u},
  While[Abs[f[x]] > epsilon,
   x = x - f[x] / D[f[u], u] /. u → x;
   If[verbose, Print[N[x]]]];
  Return[N[x]]
 ]
```

Die Hilfsvariable *u* wird dazu benutzt, mittels des Operators D die erste Ableitung $f'(u)$ zu bilden. Für *u* ist dann der aktuelle numerische Wert *x* einzusetzen, was durch den angehängten Operator /. *u* –> *x* geschieht.
Das Newton-Verfahren kommt im Fall der Funktion f_2 mit weniger Schritten aus als die Sekantenmethode:

```
newton[f2, 3, 0.0001, True]
```

```
2.36

2.1272

2.09514

2.09455

2.09455
```

4.2. Quadratwurzel nach dem Heron-Verfahren

Durch Heron von Alexandria (1. Jahrhundert n. Chr.) ist ein – schon den Babyloniern zugeschriebenes – Verfahren überliefert, mit dem man die Quadratwurzel *x* einer vorgegebenen nicht-negativen reellen Zahl *a* berechnen kann.
Die Idee dieses Algorithmus lässt sich gut in geometrischer Sprache nachvollziehen: Die gesuchte Größe *x* wird als Seitenlänge eines Quadrats mit Flächeninhalt *a* aufgefasst. Das Quadrat wird dann schrittweise durch ein flächeninhaltsgleiches Rechteck mit den Seiten *x* und *y* so angenähert, dass bei jedem Rechenschritt $|x - y|$ kleiner wird. Am Anfang setzt man einfach

$$x = a,\ y = 1$$

Für das nächste Rechteck wird die eine Seite *x'* als arithmetisches Mittel von *x* und *y* bestimmt, d.h.

$$x' = \frac{x+y}{2},$$

und die andere Seite *y'* so, dass wieder *x' y'* = *a*, d.h.

$$y' = \frac{a}{x'}.$$

Dies wird nun solange wiederholt, bis der Unterschied $|x - y|$ klein genug ist, z.B. kleiner als 0.000001. Das Verfahren ist (quadratisch) konvergent, und im allgemeinen genügen wenige Schritte, um diese Genauigkeit zu erzielen.

Bemerkung
Das Produkt zugehöriger *x*- und *y*-Werte ist stets = *a* (Flächeninhalt als Schleifeninvariante!). Daraus wird sofort ersichtlich, dass nur wenig voneinander abweichende *x*, *y* die gesuchte Quadratwurzel $\sqrt{a}$ approximieren.

Demonstration des Heron-Verfahrens zu $\sqrt{5}$

$$\left\{\{5,\ 1\},\ \left\{3,\ \frac{5}{3}\right\},\ \left\{\frac{7}{3},\ \frac{15}{7}\right\},\ \left\{\frac{47}{21},\ \frac{105}{47}\right\},\ \left\{\frac{2207}{987},\ \frac{4935}{2207}\right\},\ \left\{\frac{4\,870\,847}{2\,178\,309},\ \frac{10\,891\,545}{4\,870\,847}\right\}\right\}$$

Das Heron-Verfahren soll nun in der Sprache von *Mathematica* dargestellt werden:

```
heron[a_, epsilon_, verbose_] := Module[{x = a, y = 1, x1, y1},
  While[Abs[x - y] > epsilon,
   x1 = (x + y) / 2;
   y1 = a / x1;
   {x, y} = {x1, y1};
   If[verbose, Print[{x, y}]]];
  Return[x]
 ]
```

Die schnelle Konvergenz dieser Methode illustriert der folgende Aufruf:

```
heron[5, 0.00000000000000000001, True]
```

$$\left\{3, \frac{5}{3}\right\}$$

$$\left\{\frac{7}{3}, \frac{15}{7}\right\}$$

$$\left\{\frac{47}{21}, \frac{105}{47}\right\}$$

$$\left\{\frac{2207}{987}, \frac{4935}{2207}\right\}$$

$$\left\{\frac{4\,870\,847}{2\,178\,309}, \frac{10\,891\,545}{4\,870\,847}\right\}$$

$$\left\{\frac{23\,725\,150\,497\,407}{10\,610\,209\,857\,723}, \frac{53\,051\,049\,288\,615}{23\,725\,150\,497\,407}\right\}$$

$$\frac{23\,725\,150\,497\,407}{10\,610\,209\,857\,723}$$

```
N[%, 30]
```

```
2.23606797749978969640917367668
```

Zum Vergleich der zugehörige *y*-Wert:

```
N[53 051 049 288 615 / 23 725 150 497 407, 30]
```

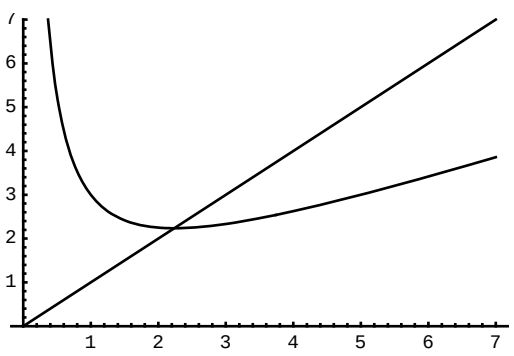
```
2.23606797749978969640917366079
```

Bemerkungen

1) Man überzeugt sich unschwer davon, dass der Heron-Algorithmus ein Sonderfall des Newton-Verfahrens ist. Dazu wählen wir $f(x) = a - x^2$, denn die Quadratwurzel aus a ist Nullstelle von f. Der Iterationsschritt im Newton-Verfahren ersetzt x durch $x - \frac{f(x)}{f'(x)} = x - \frac{a-x^2}{-2x} = \frac{1}{2}\left(x + \frac{a}{x}\right)$. Dasselbe Ergebnis erhalten wir, wenn in $\frac{x+y}{2}$ für y der aktualisierte Wert $y = \frac{a}{x}$ eingesetzt wird.

2) Offenbar ist $\sqrt{a}$ Fixpunkt der Abbildung $x \mapsto \frac{1}{2}\left(x + \frac{a}{x}\right)$:

```
Plot[{(x + 5 / x) / 2, x}, {x, 0, 7}, PlotRange → {0, 7}];
```



4.3. Berechnung von Potenzen

Rekursive und iterative Definition

Wir betrachten zunächst Potenzen mit nichtnegativen ganzen Exponenten.
Üblicherweise wird die Bildung einer Potenz a^n so gedeutet, dass der Ausgangswert $p = 1$ (als nullte Potenz von a) noch n-mal mit a multipliziert wird. Man kann dies auch in ein rekursives Gewand kleiden:

$a^0 = 1$,
$a^n = a^{n-1}\, a$ für $n \geqslant 1$.

So lässt sich programmiersprachlich die rekursive Fassung wiedergeben:

```
potrek[a_, n_] := If[n == 0, 1, potrek[a, n - 1] * a]

potrek[3, 5]

243
```

Die Funktion *potrek* ruft sich solange selbst auf, bis sie nur noch *potrek*[3,0] auszuwerten hat:

```
potrek[3,5]
3×potrek[3,4]
3×3×potrek[3,3]
3×3×3×potrek[3,2]
3×3×3×3×potrek[3,1]
3×3×3×3×3×potrek[3,0]
3×3×3×3×3×1
3×3×3×3×3
3×3×3×9
3×3×27
3×81
243
```

Für die iterative Variante legen wir einen kleinen Modul an:

```
potit[a_, n_] := Module[{i = n, p = 1},
 While[i > 0, p = p * a; i = i - 1];
  Return[p]
 ]

potit[3, 11]

177147
```

Natürlich hätte man denselben Effekt auch über Product[a, {i, 1, n}] erzielt.
Alle diese Definitionen haben gemeinsam, dass sie $n - 1$ Multiplikationen für die Berechnung von a^n benötigen.

Schnelles Potenzieren (nach Legendre)*

Schon einfache Beispiele lassen deutlich werden, dass bei dieser naiven Rechenweise zu viele Multiplikationen ausgeführt werden. So kommt man zur Berechnung von 3^9 schon mit 4 Multiplikationen aus:

$$3^9 = 3 \times 3^8 = 3 \times (3^4)^2 = 3 \times \left((3^2)^2\right)^2$$

Die systematische Anwendung dieser Idee führt zu einer stark verbesserten Methode der Potenzierung, die auf den französischen Mathematiker Legendre (Théorie des nombres, 1798) zurückgeht. Sie arbeitet nach dem Prinzip *Teile-und-Herrsche*, indem sie das Problem bei jedem Schritt angenähert halbiert.

Realisierung des schnellen (binären) Potenzierens durch die Funktion binpot:

```
binpot[a_, n_] := Module[{x = a, i = n, p = 1},
  While[i > 1,
   If[OddQ[i], p = p * x];
   i = Quotient[i, 2];
   x = x * x];
  Return[If[n != 0, p * x, 1]]
 ]
```

Die Schleife lässt den Ausdruck $p\, x^i$ ($= a^n$) invariant, so dass nach Austritt $i = 1$ und damit $p\, x^1 = a^n$ wird. Der Abbruch bei $i = 1$ vermeidet ein überflüssiges Quadrieren von x; dadurch muss der Exponent 0 bei der Wertrückgabe gesondert behandelt werden.

```
binpot[3, 7]
```

```
2187
```

Abschätzung des Rechenaufwands

(Hier ausgespart.)

5. Algorithmen aus der Zahlentheorie

5.1. Multiplikation ganzer Zahlen

Vorbemerkung

Die klassische Schulmethode der Multiplikation lässt sich wie die Addition in einem beliebigen Stellenwertsystem abbilden. Dabei müssen im allgemeinen mehrere Additionen ausgeführt werden, wobei die Summanden durch "elementare" Multiplikationen mit *B*-adischen Ziffernwerten entstehen ("Kleines 1×1"). Ein entsprechendes Programm ist also etwas komplizierter; allerdings gewinnt man in der Frage der formalen Beschreibung keine grundsätzlich neuen Erkenntnisse.
Ein interessanter Sonderfall ist die binäre Multiplikation, die das Produkt zweier ganzer Zahlen allein durch Additionen, Verdopplungen und Halbierungen berechnet.
Für die Multiplikation grosser Zahlen kennt man effizientere Verfahren als die Schulmethode, z.B. das Verfahren von Karatsuba und Ofman.

Die Schulmethode und die sog. ägyptische Multiplikationsregel

Das im Alltag gebräuchliche Multiplikationsverfahren erfordert folgende Basiskenntnisse:
a) das "kleine 1×1", d.h. die Werte sämtlicher Produkte $z_1 \times z_2$ ($0 \le z_1,\ z_2 < B$);
b) die Stellenaddition mit Übertrag.

Beispiel

```
   13 × 21
   26
+   13
   273
```

Dasselbe Beispiel lautet in binärer Darstellung:

```
   1101 × 10101
   1101
    0000
     1101
      0000
       1101
  100010001
```

Analyse

Man braucht offenbar nur zu wissen, dass $1 + 1 = 2 = (10)_2$ und 1×1 = 1. Bei der Addition entsteht naturgemäß häufig ein Übertrag (beteiligte Einsen sind im obigen Beispiel markiert).

Bei genauerem Hinschauen erkennt man, dass die Multiplikation in binärer Darstellung im wesentlichen durch sukzessives Verdoppeln und Halbieren erfolgt: Trifft man in 10101 auf eine 1 (ungerader Fall), so wird 1101 um eine Spalte versetzt (d.h. verdoppelt) und hingeschrieben, was insgesamt 3-mal geschieht. Trifft man auf eine 0 (gerader Fall), wird lediglich verdoppelt. Im ganzen erhält man das Ergebnis dann als Summe von geeignet verdoppelten Werten:

13 + 52 + 208 = 273.

Im einzelnen sieht die Rechnung so aus:

$$\begin{aligned} 13 \cdot 21 &= 13 \cdot (20 + 1) \\ &= 13 \cdot 20 + \underline{13} \\ &= (26 \cdot 2) \cdot 5 + \underline{13} \\ &= 52 \cdot (2 \cdot 2 + 1) + \underline{13} \\ &= 104 \cdot 2 + \underline{52} + \underline{13} \\ &= \underline{208} + \underline{52} + \underline{13} \end{aligned}$$

Man kann sich das Verfahren auch so zurechtlegen, dass beginnend mit *x* (= 13) und *y* (= 21) bei jedem Schritt *x* verdoppelt und *y* halbiert wird. Taucht ein ungerades *y* auf, nimmt man das Ganze von *y* : 2 und gleicht den "Verlust" des Restes durch Addition von *x* wieder aus.

Ägyptische (russische) Multiplikationsregel

In dieser Form ist das Verfahren als "russische Bauernregel" bekannt; es soll aber schon im alten Ägypten als Multiplikationsregel in Gebrauch gewesen sein:

x	×	y	z	*Bemerkung*
13		21	0	(Anfangswert)
26		10	13	(21 ist ungerade, also $z = z + 13$)
52		5	13	
104		2	65	(5 ist ungerade, also $z = z + 52$)
208		1	65	
		0	**273**	(1 ist ungerade, also $z = z + 208$)

Es bleibt $p = xy + z$ bei jedem Schritt konstant. Offenbar ist p das gesuchte Produkt xy (wähle $z = 0$), und am Ende ($y = 0$) hat man: $p = z$. Damit ist die Korrektheit des folgenden Algorithmus bewiesen:

```
binmult[a_, b_] := Module[{x = a, y = b, z = 0},
  While[y > 0,
   If[OddQ[y], z = z + x];
   {x, y} = {2 * x, Quotient[y, 2]}];
  Return[z]
 ]

binmult[13, 21]

273
```

Das Verfahren von Karatsuba und Ofman

(Hier ausgespart.)

5.2. Ganzzahlige Quadratwurzel

Vorbemerkung

Bei der Suche nach den Teilern t einer ganzen Zahl a können wir uns auf positive $t \le \sqrt{a}$ beschränken (vgl. dazu die Bemerkung im Anschluss an Satz 5.3.1). Tritt dies in einer Schleifenbedingung auf, z.B. in der Form $t^2 \le a$, so muss vor jedem Durchgang eine Multiplikation ausgewertet werden. Es ist ökonomischer, stattdessen einmalig (*vor* Eintritt in die Schleife) den ganzzahligen Teil g von $\sqrt{a}$ zu berechnen und dann innerhalb der Schleife nur noch die Bedingung $t \le g$ abzufragen.

Eine ganzzahlige Variante des Heron-Algorithmus

Einen brauchbaren Algorithmus zur Berechnung von g liefert das Heron-Verfahren. Obwohl es in seiner ursprünglichen Fassung rationale Näherungswerte von $\sqrt{a}$ berechnet, lässt es sich leicht in eine Form bringen, bei der sämtliche Rechnungen ganzzahlig bleiben.
Als Ausgangspunkt dient uns das in Abschnitt 4.2 beschriebene Vorgehen, bei dem (x, y) in ein neues Zahlenpaar (x', y') umgerechnet wird. Wir wollen uns auf den Übergang von x nach x' konzentrieren. Die zweite Variable y wird bei jedem Wiederholungsschritt $= \frac{a}{x}$ gesetzt und lässt sich daher leicht eliminieren:

$$x' = \frac{x+y}{2} = \frac{1}{2}\left(x + \frac{a}{x}\right)$$

Die Folge der Näherungswerte für $\sqrt{a}$ entsteht demnach durch Iteration (fortgesetzte Anwendung) der Funktion

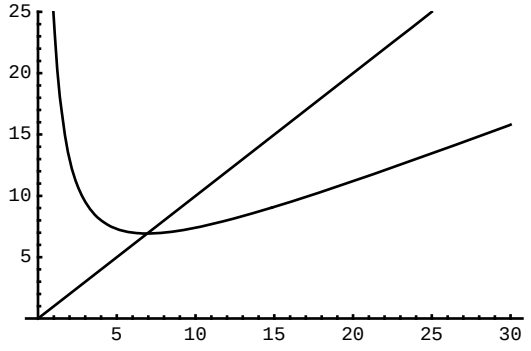
$$x \mapsto x' := \left\lfloor \frac{1}{2}\left(x + \frac{a}{x}\right)\right\rfloor = \left\lfloor \frac{x^2+a}{2x} \right\rfloor$$

Wir verschaffen uns zunächst einmal ein Bild dieser Zuordnung als reelle Funktion:

```
fheron[x_, a_] := (x + a/x)/2
```

Der zugehörige Funktionsgraph für $x > 0$ und $a > 0$:

```
Plot[{fheron[x, 48], x}, {x, 0, 30}, PlotRange -> {0, 25}];
```



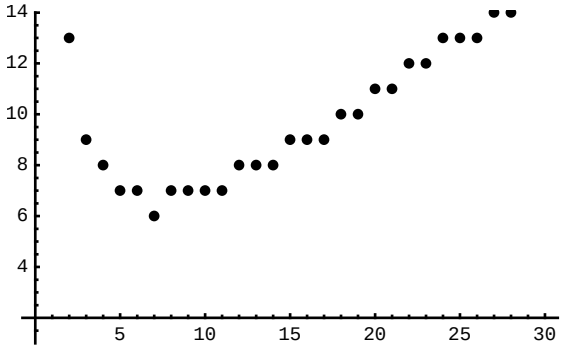
Mit einfachen Mitteln der Schulanalysis ("Kurvendiskussion") zeigt man:

Beh. 1.

Die Funktion fheron : $R^+ \to R^+$ besitzt genau ein relatives Minimum bei $x_{min} = \sqrt{a}$. Im Intervall $(0, \sqrt{a})$ ist fheron streng monoton fallend, im Intervall $(\sqrt{a}, \infty)$ streng monoton wachsend. Das Minimum ist zugleich absolutes Minimum und Fixpunkt: fheron($\sqrt{a}$,a)=$\sqrt{a}$.

Sei nun $a \geq 1$ ganz. Beschränkt man sich auf die ganzzahligen Teile der Funktionswerte, so erhält man (zu ganzen Argumenten) eine ähnlich aussehende Folge positiver ganzer Zahlen:

```
fhTabelle = Table[Floor[fheron[x, 48]], {x, 1, 30}];
ListPlot[fhTabelle, PlotJoined -> False,
  PlotStyle -> PointSize[0.02], PlotRange -> {1, 14}];
```



Der Ausreißer beim Funktionswert 6 wird uns später noch einmal beschäftigen.

Exkurs: Ganzteil einer Zahl

Gewöhnlich wird mit $[x]$ die grösste ganze Zahl $\leq x$ bezeichnet. In *Mathematica* heißt die entsprechende Funktion Floor. Für rationale Werte $x = \frac{a}{b}$stimmt $[x]$ offenbar mit dem Ganzen der Division $a : b$, also mit Quotient[a, b], überein.

Nützlich ist im folgenden die

Beh. 2.

Für alle reellen Zahlen x > 0 gilt:

1. $[x] + 1 > x$

2. $\left[\frac{x+\left[\frac{a}{x}\right]}{2}\right] = \left[\frac{x^2+a}{2x}\right]$

Beweis:

1. offensichtlich.

2. Man überlegt sich anhand von Teil 1, dass die Differenz von $\frac{x^2+a}{2x} = \frac{x+\frac{a}{x}}{2}$ und $\frac{x+\left[\frac{a}{x}\right]}{2}$ kleiner als $\frac{1}{2}$ ausfällt. ◆

Nun soll ein ganzzahliges Analogon zu *fheron* definiert werden:

```
gheron[k_, a_] := Quotient[k * k + a, 2 * k]
```

Mit Hilfe dieser Funktion definieren wir nun eine Folge (x_k), die den Startwert a hat und ihre nachfolgenden Glieder durch wiederholte Anwendung von *gheron* erzeugt:

```
ghfolge[n_, a_] := Module[{x = a}, Do[x = gheron[x, a], {n}]; x]
```

Dieses Modul entspricht der folgenden Definition:

$x_0 = a$,
$x_{n+1} = [\ fheron[x_n, a]\ ]$ *für* $n \geq 0$

Betrachten wir die ersten zehn Werte von (x_k) zum Radikanden $a = 20$:

```
Table[ghfolge[n, 20], {n, 1, 10}]
```

```
{10, 6, 4, 4, 4, 4, 4, 4, 4, 4}
```

Welches Bild erhalten wir bei anderen Radikanden?

```
Table[ghfolge[n, 21], {n, 1, 10}]
Table[ghfolge[n, 12], {n, 1, 10}]
Table[ghfolge[n, 35], {n, 1, 10}]
```

```
{11, 6, 4, 4, 4, 4, 4, 4, 4, 4}
```

```
{6, 4, 3, 3, 3, 3, 3, 3, 3, 3}
```

```
{18, 9, 6, 5, 6, 5, 6, 5, 6, 5}
```

Erste Beobachtung: Es gibt Folgen, die irgendwann konstant werden, und solche, die schließlich zwischen zwei (benachbarten) Werten oszillieren.

Das ursprüngliche Heron-Verfahren arbeitet mit zwei Werten x und y. Auch dies soll im Bereich der ganzen Zahlen nachgebildet werden:

$z_0 = a,\ y_0 = 1$

$z_{n+1} = \left[\frac{z_n + y_n}{2}\right],\ y_{n+1} = \left[\frac{a}{z_{n+1}}\right]$

Die Rolle des x übernimmt hier z, da nicht von vornherein klar ist, dass man auf diese Weise dieselben Folgen erzeugt. Man kann aber die Übereinstimmung von x und z nachweisen:

Beh. 3.

Für alle $n \geq 0$ gilt: $z_n = x_n$

Beweis:
Vollständige Induktion nach n unter Verwendung von Beh. 2 (als Übung!). ◆

Die betreffende Folge wird von nun ab nur noch mit x bzw. x_k bezeichnet. Sie soll im weiteren detailliert untersucht und in ihrem Verlauf lückenlos erklärt werden.

Die Tatsache, daß $\sqrt{a}$ das absolute Minimum von *fheron* ist, spiegelt sich im ganzzahligen Bereich wie folgt wider:

Beh. 4.

Für alle $n \geq 0$ gilt: $a < (x_n + 1)^2$

Beweis:

Nach Beh. 1 gilt *fheron*$[x, a] \geq \sqrt{a}$ für alle $x > 0$, also: $\left[\frac{x^2+a}{2x}\right] \geq \left[\sqrt{a}\right]$. Es ergibt sich mit Beh. 2.1: $x_n + 1 = \left[\frac{x_{n-1}^2+a}{2\,x_{n-1}}\right] + 1 \geq \left[\sqrt{a}\right] + 1 > \sqrt{a}$, woraus durch Quadrieren die behauptete Ungleichung folgt. ◆

Der Beh. 4 ist unmittelbar zu entnehmen, dass alle $x_k \geq 1$ sind (wegen $a \geq 1$); die Folge besitzt also auch ein Minimum grösser als 1. Genauer lässt sich zeigen:

Beh. 5.

Ist x_m das erste Glied der Folge (x_k), welches ihr Minimum annimmt, so gilt:

1. $x_{m-1} > x_m \leq x_{m+1}$

2. $x_m^2 \leq a$

Beweis:
Zu 1. Da x_m minimal ist, hat man $x_m \leq x_{m-1}$ und $x_m \leq x_{m+1}$. Es kann ferner nicht $x_m = x_{m-1}$ sein, da sonst m nicht der kleinste Index k wäre, für den x_k minimal ist.

Zu 2. Es gilt $x_{m+1} = \left[\frac{x_m^2+a}{2\,x_m}\right]$, d.h. $x_m^2 + a = 2\,x_m\,x_{m+1} + r$ mit $0 \leq r < 2\,x_m$. Hieraus ergibt sich mit Teil 1: $a \geq 2\,x_m\,x_{m+1} - x_m^2 \geq 2\,x_m^2 - x_m^2 = x_m^2$. ◆

Nehmen wir die Behauptungen 4 und 5 zusammen, so erhalten wir $x_m^2 \leq a < (x_m + 1)^2$, was bedeutet: x_m ist die größte ganze Zahl z mit $z^2 \leq a$, d.h. $x_m = \left[\sqrt{a}\right]$.

Aus Beh. 5.1 lässt sich bereits ablesen, wie das Minimum zu berechnen ist. Die "begleitende" Folge (y_k) leistet dabei nützliche Dienste:

Beh. 6.

Für alle $n \geq 0$ gilt:

1. $x_n\,y_n \leq a$

2. $x_n > y_n \iff x_n^2 > a$

Beweis:
Zu 1. Aufgrund der Definition von y_n als Ganzes der Division $a : x_n$ hat man: $a = y_n\,x_n + r$, wobei $0 \leq r < x_n$. Daraus ergibt sich direkt die

behauptete Ungleichung.
Zu 2. Aus $x_n > y_n$ folgt $x_n \geq y_n + 1$, mithin $x_n^2 \geq x_n y_n + x_n > a$. Umgekehrt ergibt sich mit Teil 1: $x_n^2 > a \geq x_n y_n$ und nach Kürzen von x_n die Behauptung. ◆

Anhand von Beh. 5.2 und 6.2 wird klar, dass für das Minimum x_m der erste Index m zu suchen ist, für den $x_m \leq y_m$ wird. Das heißt, ausgehend von $x = a$ und $y = 1$ wiederholt man $x = \left[\frac{x+y}{2}\right]$ und $y = \left[\frac{a}{x}\right]$ solange wie $x > y$ ist. Ist nach Austritt aus dieser Schleife $x \leq y$, so gilt $x = \left[\sqrt{a}\right]$.
Damit haben wir den Satz über den ganzzahligen Heron-Algorithmus:

5.2.1. Satz

Die folgende Funktion introot berechnet $\left[\sqrt{a}\right]$.

```
introot[a_] := Module[{x = a, y = 1},
  While[x > y, x = Quotient[x + y, 2]; y = Quotient[a, x]]; Return[x]
 ]
```

Genauere Untersuchung der Folge
Wir hatten zu Beginn beobachtet, dass die Folge (x_k), verkörpert durch die Funktion *gheron*, nach erstmaliger Annahme ihres Minimums stationär oder oszillierend verlaufen kann; im zweiten Fall wird das Minimum dann unendlich oft wieder angenommen. Dieses Phänomen soll im folgenden erklärt werden. Zusätzlich ist der Folgenverlauf links vom Mininum zu untersuchen.

Beh. 7.

1. Ist $a + 1$ ein Quadrat, so gilt $x_{m+1} = x_m + 1$ und $x_{m+2} = x_m$ (oszillierender Verlauf); andernfalls gilt $x_k = x_m$ für alle $k > m$ (stationärer Verlauf).

2. Links vom Minimum ist die Folge streng fallend: $x_0 > x_1 > \ldots > x_{m-1} > x_m$.

Beweis: als Übung.

Ein Verfahren, das nur Additionen benutzt

(Hier ausgespart.)

Die Methode von Klostermaier und Lüneburg

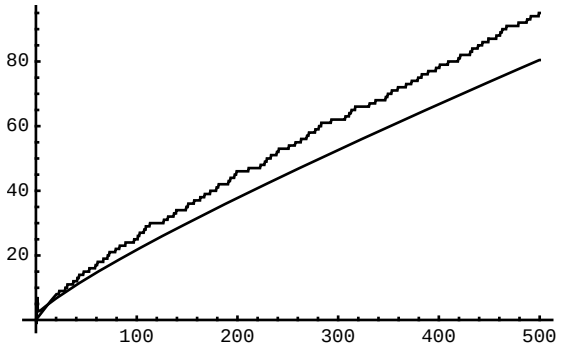
(Hier ausgespart.)

5.3. Euklids Primzahlenmaschine

Primzahlen

Primzahlen sind faszinierende Sonderlinge: Sie scheinen – zumindest am Anfang der Zahlenreihe – häufiger vorzukommen, doch nimmt ihre Dichte nach oben hin ab, obwohl es unendlich viele von ihnen gibt. Der berühmte *Primzahlsatz* sagt über die Anzahl $\pi(x)$ der Primzahlen $\leq x$, dass das Verhältnis von $\pi(x)$ zu $\frac{x}{\log x}$ gegen 1 strebt, wenn x größer wird.

```
Plot[{PrimePi[x], x / Log[x]}, {x, 0, 500}];
```



Dennoch vermittelt uns dieses Gesetz keine Erkenntnisse darüber, wie viele Primzahlen sich in einem bestimmten Abschnitt befinden, und es drängt sich der Eindruck einer eher unregelmäßigen Verteilung auf.
Ihre fundamentale Bedeutung erlangen Primzahlen dadurch, dass alle ganzen Zahlen in eindeutiger Weise als Produkt von Primfaktoren darstellbar sind (Hauptsatz der elementaren Arithmetik).
Natürlich hält *Mathematica* eine Reihe von Funktionen bereit, die mit Primzahlen zu tun haben. Die wichtigsten von ihnen sind Prime[n] (berechnet die n-te Primzahl), PrimeQ[n] (entscheidet, ob n prim ist) und FactorInteger[n] (berechnet die Zerlegung von n in seine Primfaktoren).
Hier sind eine Beispiel-Berechnungen:

```
Prime[10 000]
```

```
104 729
```

```
PrimeQ[104 733]
```

```
False
```

```
FactorInteger[104 733]
```

```
{{3, 5}, {431, 1}}
```

Der letzte Output ist als $3^5 \times 431$ (= 104 733) zu verstehen.

Die folgenden Betrachtungen beschränken sich auf die grundlegenden Aspekte der Primzahlen beim Aufbau der Arithmetik. Für weiteres Material vgl. [Bartholomé, u.a. 1995; Forster 1996; Niederdrenk-Felgner, u.a. 1988].

Der kleinste Teiler einer Zahl

Unser Ausgangspunkt ist der Satz vom Primteiler:

5.3.1. Satz

Ist n eine ganze Zahl > 1, so ist der kleinste Teiler t ≥ 2 von n stets prim.

Beweis:
Ist – indirekt angenommen – t zusammengesetzt, etwa: $t = r \cdot s$ mit r, $s \geq 2$, so hat man für ein geeignetes ganzes q die Gleichung $n = t \cdot q = r \cdot s \cdot q$. Wegen $r < t$ ist dann aber t nicht der kleinste Teiler (≥ 2) von n (Widerspruch). ◆

Bemerkung

Bei zusammengesetztem n gilt für den kleinsten Primteiler p die Abschätzung: $p \leq \sqrt{n}$. Aus der Minimalität von p ergibt sich nämlich unmittelbar: $n = p \cdot k \geq p^2$ (hierbei ist k der sogenannte Komplementärteiler von p).

Eine einfache und naive Methode, den kleinsten Primteiler von n zu bestimmen, besteht darin, alle in Frage kommenden Zahlen t auf ihre Teilereigenschaft hin zu testen. Man kann sich dabei auf den Bereich $2 \leq t \leq [\sqrt{n}\,]$ beschränken.
Die folgende Funktion berechnet den kleinsten Primteiler von n:

```
minteiler[1] = 1;
minteiler[n_] := Module[{mt = n, s = introot[n], t = 2},
  While[t <= s, If[Mod[n, t] == 0, (mt = t; Break[]), t++]]; Return[mt]
 ]
```

Die Rückgabe von 1 erfolgt in dem Fall, dass n keinen Teiler ≥ 2 besitzt. Die lokale Variable s beinhaltet die ganzzahlige Wurzel aus n als Schranke für den Suchbereich. Wird ein Teiler t gefunden, soll die Schleife sofort verlassen werden; dies bewirkt der Break-Befehl. Wird kein Teiler gefunden, gibt die Funktion das Argument n zurück, d.h. n ist in diesem Fall eine Primzahl.
Einige Beispiele:

```
minteiler[104 733]
```

```
3
```

```
minteiler[23 * 101]
```

```
23
```

Bei grösseren Zahlen und nahe an $\sqrt{n}$ gelegenen Primteilern nimmt die Suche beträchtliche Zeit in Anspruch:

```
minteiler[Prime[9999] * Prime[10 000]] // Timing
```

```
{4.01 Second, 104 723}
```

Der Algorithmus zu *minteiler* ist ziemlich schlecht. Zum einen prüft er zuviel, z.B. die Teilbarkeit durch 4, nachdem 2 schon als Teiler ausgeschieden ist. Das lässt sich vermeiden, indem man nur *Primzahlen* zwischen 2 und $\sqrt{n}$ berücksichtigt. Allerdings behebt dies nicht die grundlegende Schwäche des ganzen Verfahrens. Bei einer 100-stelligen Zahl hätte man im ungünstigsten Fall immer noch an die $\pi\left(\sqrt{10^{100}}\right) \approx 8 \cdot 10^{47}$ Primzahlen zu testen, was auch bei äußerst leistungsfähigen Computern die bisher verstrichene Zeit (Alter des Universums) um ein Vielfaches überschreiten würde. Die deutlich schnelleren Methoden, über die man inzwischen verfügt, beruhen auf einigermassen komplizierten theoretischen Grundlagen, die nicht mehr in unserem Rahmen liegen; vgl. etwa [H. Riesel: *Prime Numers and Computer Methods for Factorization*. Birkhäuser-Verlag: Boston; Basel; Stuttgart 1985].

Eine bescheidene Verbesserung des obigen *minteiler*-Algorithmus können wir durch eine geeignete Ausdünnung des Intervalls 2, ..., $\sqrt{n}$ erzielen (wodurch sich leider das ungünstige Verhalten bei großen Zahlen im Kern nicht beeinflussen lässt).
Das modifizierte Verfahren beruht auf folgendem

5.3.2. Hilfssatz

Sei $2 \le m < n$ und p ein Primteiler von n. Dann ist entweder p ein Teiler von m oder p gehört zu einer primen Restklasse mod m, d.h. $p = k \cdot m + r$ mit $0 \le r < m$ und $ggT(m, r) = 1$.

Beweis:
Ist p kein Teiler von m, so gilt entweder $p < m$ (in diesem Fall ist p wegen seiner Teilerfremdheit zu m ein primer Rest mod m), oder man hat $p > m$ und nach Division durch m mit Rest: $p = k\,m + r,\ 0 \le r < m$. Da p prim ist, besitzen m und r keinen gemeinsamen Teiler, es ist also r ein primer Rest mod m. ◆

Der Hilfssatz soll hier nur für den sehr einfachen Fall $m = 2 \cdot 3 = 6$ angewandt werden. Zunächst ist danach zu prüfen, ob 2 bzw. 3 die Zahl n teilen. Ist dies der Fall, ist man fertig. Andernfalls stellen wir einen potentiellen Primteiler p von n in der Form $p = 6k + r$ dar, wobei r primer Rest mod 6 ist, d.h. $r = 1$ oder $r = 5$. Anstatt nun bei 5 beginnend alle Zahlen bis $\sqrt{n}$ auf ihre Teilereigenschaft zu prüfen (wie dies in der Funktion *minteiler* geschieht), erhöht man den Startwert 5 abwechselnd um 2 – nämlich $2 = (1 - 5) \bmod 6$ – und um 4 ($= 5 - 1$). Auf diese Weise wählt aus jeder der beiden Restklassen zu 1 und zu 5 die jeweils nächstgrößere Zahl.
Die entsprechend modifizierte Funktion heiße primdiv:

```
primdiv[n_] := Module[{s = introot[n], diff = 2, t = 5},
  If[Mod[n, 2] == 0, Return[2]; Break[]]; If[Mod[n, 3] == 0, Return[3]; Break[]];
  While[t <= s,
   If[Mod[n, t] == 0, Return[t]; Break[]];
   t = t + diff; diff = 6 - diff];
  Return[n]
 ]
```

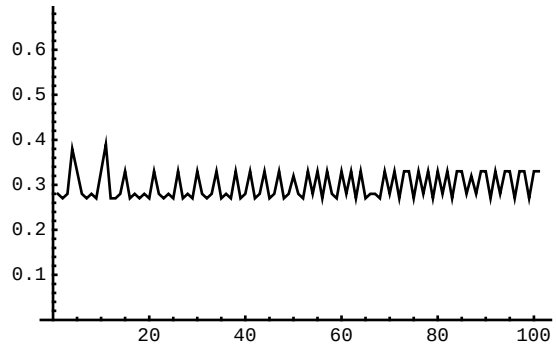
In folgendem Fall arbeitet primdiv etwa doppelt so schnell wie minteiler:

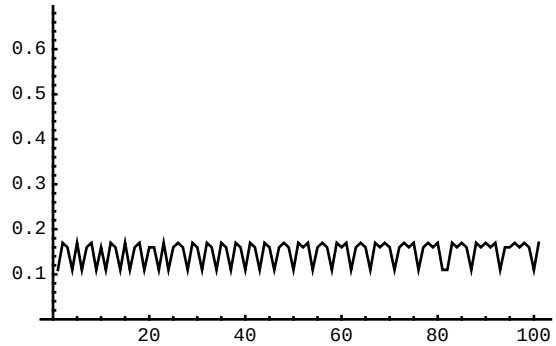
```
primdiv[Prime[9999] * Prime[10 000]] // Timing
```

```
{2.03 Second, 104 723}
```

Wir wollen die beiden Konkurrenten minteiler und primdiv einmal empirisch vergleichen. Dazu werden die Rechenzeiten für eine Reihe unbequemer Zahlen ermittelt:

```
zeit1 =
  Table[First[Timing[minteiler[Prime[k] * Prime[k + 1]]]] /. Second -> 1, {k, 1000, 1100}];
zeit2 =
  Table[First[Timing[primdiv[Prime[k] * Prime[k + 1]]]] /. Second -> 1, {k, 1000, 1100}];
ListPlot[zeit1, PlotJoined -> True, PlotRange -> {0, 0.7}];
ListPlot[zeit2, PlotJoined -> True, PlotRange -> {0, 0.7}];
```





Die Rechenzeit wird aufgrund der Modifikationen in dem gewählten Zahlenbereich ungefähr halbiert. Man erzielt so also lediglich einen linearen Effekt, der das Wachstumsverhalten nicht berührt.

Der Satz von Euklid

Euklid hat in Buch IX seiner *Elemente* bewiesen, dass es unendlich viele Primzahlen gibt. Diesen Sachverhalt formuliert und beweist er in konstruktiver Form. Es folgt eine modernisierte Fassung seines Gedankengangs:

5.3.3. Satz

Ist P irgendeine Menge von Primzahlen, so kann man eine Primzahl q angeben, die nicht zu der Menge P gehört.

Beweis:
Wir geben einen Algorithmus an, der das gesuchte $q \notin P$ effektiv berechnet. Es bezeichne n das Produkt aller Zahlen aus P. Dann wird $q =$ kleinster Primteiler von $(n + 1)$ gesetzt. Damit gilt $n + 1 = m \cdot q$ für ein geeignetes ganzes m, mithin auch $m \cdot q - n = 1$. Wäre nun $q \in P$, so könnte q auf der linken Seite dieser Gleichung ausgeklammert werden (q ist nach indirekter Annahme ein Faktor in n) und müsste daher gleich 1 sein (Widerspruch!). ◆
Der Beweis funktioniert sogar für den trivialen Fall $P = \emptyset$. Denn das Produkt über der leeren Menge ist $n = 1$, und damit wird $q = 2$ (die erste Primzahl).
Die Berechnung von q aus der Zahlenmenge P leistet die folgendermassen definierte Funktion euklid:

```
euklid[zmenge_] := primdiv[1 + Times @@ zmenge]
```

Zunächst schauen wir uns einige Ergebnisse an:

```
euklid[{}]
```

```
2
```

```
euklid[{2}]
```

```
3
```

```
euklid[{2, 3}]
```

```
7
```

```
euklid[{2, 3, 5, 7}]
```

```
211
```

Nicht immer führt die Berechnung zu einer größeren Primzahl:

```
euklid[{5, 17, 19}]
```

```
2
```

Nicht alle Terme der Form $p_1\, p_2 \ldots p_m + 1$ (mit aufeinanderfolgenden Primzahlen p_i) sind ihrerseits wieder prim. Hier ist das erste Gegenbeispiel:

```
PrimeQ[2 * 3 * 5 * 7 * 11 * 13 + 1]
```

```
False
```

Natürlich liefert die Funktion *euklid* eine Primzahl:

```
euklid[{2, 3, 5, 7, 11, 13}]
```

```
59
```

Bemerkung
Bis heute ist es ein offenes Problem, ob es unendlich viele solcher "euklidischen" Primzahlen der Form $p_1\, p_2 \ldots p_m + 1$ gibt.

Wir verschaffen uns eine Tabelle der ersten n Primzahlen:

```
ptab[n_] := Table[Prime[k], {k, 1, n}];
```

```
ptab[10]
```

```
{2, 3, 5, 7, 11, 13, 17, 19, 23, 29}
```

Damit wird nun eine Funktion definiert, die große Primzahlen hervorbringt:

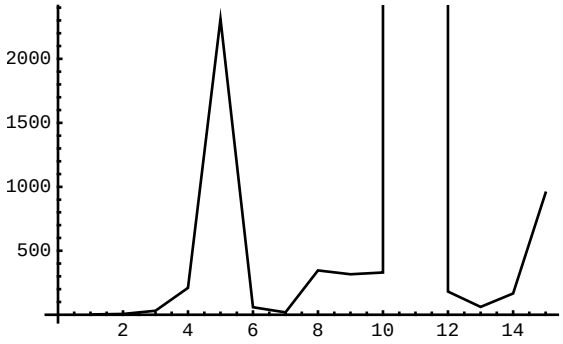
```
euprim[n_] := euklid[ptab[n]]
```

Allerdings wird die Rechenzeit schnell in kritische Zonen getrieben (im folgenden Fall wird der kleinste Primteiler von 267064515689275851355624017992791 mit Hilfe der nicht gerade effizienten Funktion primdiv berechnet):

```
euprim[23] // Timing
```

```
{5.21 Second, 265 739}
```

```
euklidTabelle = Table[euprim[k], {k, 1, 15}];
ListPlot[euklidTabelle, PlotJoined -> True];
```



Der Grund für die Verzögerung liegt offenbar bei $n = 11$:

```
euprim[11] // Timing
```

```
{8.62 Second, 200 560 490 131}
```

5.4. Das Sieb des Eratosthenes

Eratosthenes' Sieb im Handbetrieb

Von dem griechischen Universalgelehrten Eratosthenes, der im 3. Jahrhundert v. Chr. der berühmten Bibliothek von Alexandria vorstand, stammt eine interessante Methode, mit der aus einer vorgegebenen Menge ganzer Zahlen die Primzahlen herausgesiebt werden. Es ist überaus empfehlenswert, diesen Algorithmus an einem Beispiel zu beschreiben und dabei gleich "von Hand" durchzurechnen.

Wir wollen uns hier mit einer kleinen Zahlenmenge, etwa den natürlichen Zahlen von 1 bis 20, begnügen.

(1) Notiere alle Zahlen von 1 bis 30:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

(2) Streiche die Zahl 1 (nicht prim):
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

(3) Markiere die Zahl 2:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

(4) Streiche sämtliche echten Vielfachen von 2:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

(5) Markiere die erste Zahl, die noch nicht markiert ist (hier: 3):
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

(6) Streiche sämtliche echten Vielfachen von 3:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

(7) Markiere die erste Zahl, die noch nicht markiert ist (hier: 5):
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

(8) Streiche sämtliche echten Vielfachen von 5 (ohne Wirkung, da 10, 15, 20 schon gestrichen sind):
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

(9) Fahre sinngemäß solange fort, bis jede der Zahlen markiert oder gestrichen ist.
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

(10) Die markierten Zahlen sind alle prim.

Eine umgangssprachliche Fassung könnte etwa so aussehen:

1. Setze $p = 2$
2. Streiche alle Vielfachen von p
3. Setze p = erste nicht gestrichene Zahl > p
4. Gehe nach 2.

Umsetzung in ein Programm

Um diesen Algorithmus in ein Programm umzusetzen, sind einige Vorüberlegungen von Nutzen:
a) Das Verfahren benötigt eine Tabelle (Liste) {1, ..., n}, in der alle zusammengesetzten Zahlen gestrichen werden. "Streichen" soll im folgenden bedeuten: "durch 0 ersetzen". Als Ergebnis wird die Liste der Primzahlen aus {1, ..., n} zurückgegeben.
b) Der erste Siebdurchgang erfolgt mit der Primzahl 2; die erste zu streichende Zahl ist demnach 4. Als nächstes wird mit 3 gesiebt, und 3^2 (= 9) ist die nächste zu streichende Zahl. Wurde mit allen Primzahlen kleiner als p schon gesiebt, so sind alle $k \cdot p$ (mit $k < p$) bereits

gestrichen (k enthält einen Primfaktor < p); also ist p^2 die nächste zu streichende Zahl.
c) In der Tabelle {1, ..., n} braucht nur mit den Primzahlen $\leq \left\lfloor \sqrt{n} \right\rfloor$ gesiebt zu werden.

Der eigentliche Algorithmus lässt sich übersichtlicher aufschreiben, wenn man das Sieben mit einer festen Primzahl als eigenen Modul ausgliedert. Das Argument *ztab* steht hier für eine Tabelle aus Zahlen 1, ..., n:

```
sieb[p_, ztab_] := Module[{n = Length[ztab], zt = ztab, i = p * p},
  While[i <= n, zt[[i]] = 0; If[p > 2, i = i + p + p, i = i + p]];
  Return[zt]
 ]
```

Um Mehrfachstreichungen zu vermeiden, sollte für $p > 2$ der Index i jedesmal gleich um $2p$ erhöht werden, denn die Zahlen $p^2 + p$, $p^2 + 2\,p$, $p^2 + 3\,p$, ... sind sämtlich gerade (und wurden daher beim Sieben mit $p = 2$ schon vorher gestrichen!).
Damit können wir nun schon sieben:

```
ztab = Table[k, {k, 1, 30}];
ztab = sieb[2, ztab]
```

```
{1, 2, 3, 0, 5, 0, 7, 0, 9, 0, 11, 0, 13, 0,
 15, 0, 17, 0, 19, 0, 21, 0, 23, 0, 25, 0, 27, 0, 29, 0}
```

Die erste nicht gestrichene Zahl > 2 ist 3:

```
ztab = sieb[3, ztab]
```

```
{1, 2, 3, 0, 5, 0, 7, 0, 0, 0, 11, 0, 13, 0, 0, 0, 17, 0, 19, 0, 0, 0, 23, 0, 25, 0, 0, 0, 29, 0}
```

und so weiter.

Das Sieb des Eratosthenes wiederholt diesen Vorgang für alle primen $p \leq \left\lfloor \sqrt{n} \right\rfloor$ und lässt sich daher durch die nachstehende Funktion definieren:

```
eratosthenes[n_] := Module[{p = 2, i = 2, ztab = Table[k, {k, 1, n}], m},
  ztab[[1]] = 0;  (* Zahl 1 wird gestrichen *)
  m = introot[n]; (* Schranke für Siebzahlen *)
  While[p <= m,
   ztab = sieb[p, ztab]; i++; (* mit p sieben und weiterrücken *)
   While[ztab[[i]] == 0, i++]; (* gestrichene Zahlen überspringen *)
   p = ztab[[i]]]; (* die nächste nicht gestrichene Zahl holen *)
  Return[ztab]
 ]
```

```
eratosthenes[30]
```

```
{0, 2, 3, 0, 5, 0, 7, 0, 0, 0, 11, 0, 13, 0, 0, 0, 17, 0, 19, 0, 0, 0, 23, 0, 0, 0, 0, 0, 29, 0}
```

Mit dem Select-Befehl können nun noch alle Elemente, die ungleich 0 sind, herausgefischt werden:

```
Select[%, # != 0 &]
```

```
{2, 3, 5, 7, 11, 13, 17, 19, 23, 29}
```

Erläuterung: # steht für das erste Argument einer reinen, d.h. unbenannten Funktion; das Zeichen & schliesst den Rumpf dieser Funktion ab.

Man kann das Sieb-Verfahren noch etwas verbessern, indem von vornherein nur ungerade Zahlen $2\,i - 1$ betrachtet werden. In diesem Fall benötigen wir ein etwas abgeändertes sieb-Modul. Der Index (die Platznummer in der Liste) i von p^2 ist dabei nicht mehr p^2, sondern $\frac{p^2+1}{2}$.

```
sieb2[p_, ztab_] := Module[{n = Length[ztab], zt = ztab, i}, i = Quotient[p * p + 1, 2];
  While[i <= n, zt[[i]] = 0; i = i + p];
  Return[zt]
 ]
```

Die Funktion sieb2 übernimmt das Sieben in folgendem neuen Hauptmodul (worin das Aussondern der Nullen aus der Ergebnisliste gleich mit aufgenommen ist):

```
eratsieb[n_] := Module[{p = 3, i = 2, m = introot[n], ztab},
  ztab = Table[2 * k - 1, {k, 1, Quotient[n + 1, 2]}]; ztab[[1]] = 2;
  While[p <= m,
   ztab = sieb2[p, ztab]; i++;
   While[ztab[[i]] == 0, i++];
   p = ztab[[i]]];
  Return[Select[ztab, # != 0 &]]
 ]
```

Hier ein Probelauf bis zur Zahl 1000 mit Angabe der Rechenzeit:

```
eratsieb[1000] // Timing
```

```
{0.05 Second, {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73,
  79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163,
  167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257,
  263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359,
  367, 373, 379, 383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457,
  461, 463, 467, 479, 487, 491, 499, 503, 509, 521, 523, 541, 547, 557, 563, 569,
  571, 577, 587, 593, 599, 601, 607, 613, 617, 619, 631, 641, 643, 647, 653, 659,
  661, 673, 677, 683, 691, 701, 709, 719, 727, 733, 739, 743, 751, 757, 761, 769,
  773, 787, 797, 809, 811, 821, 823, 827, 829, 839, 853, 857, 859, 863, 877, 881,
  883, 887, 907, 911, 919, 929, 937, 941, 947, 953, 967, 971, 977, 983, 991, 997}}
```

5.5. Primfaktorzerlegung

Zerlegung in Primfaktoren

Der Hauptsatz der elementaren Zahlentheorie lautet:

5.5.1. Satz

Ist $n \geq 2$ ganz, so lässt sich n als Produkt von Primzahlpotenzen darstellen; die Darstellung ist eindeutig bis auf die Reihenfolge der Faktoren.

Die Existenz einer solchen Primfaktorzerlegung ergibt sich unmittelbar aus der Tatsache, dass *n* mindestens einen Primteiler *p* besitzt. Ist *n* nicht selbst eine Primzahl, so ist $\frac{n}{p} \geq 2$ und besitzt demnach wiederum einen Primteiler $p' \geq 2$ usw. Am Ende des Herausdividierens aller Primteiler *p*, *p*', ... erhält man 1, und das Verfahren endet.
Für den Beweis der Eindeutigkeit vgl. man etwa [Bartholomé, u.a. 1995].
Es läßt sich leicht eine Funktion definieren, die alle Primteiler in einer Liste sammelt:

```
primliste[1] = {};

primliste[n_] := Module[{pliste = {}, nn = n, p}, While[nn >= 2,
    p = primdiv[nn];
    pliste = Join[pliste, {p}]; nn = Quotient[nn, p]];
  Return[pliste]
 ]
```

```
primliste[2 * 2 * 3 * 5 * 7 * 7 * 7 * 17]
```

```
{2, 2, 3, 5, 7, 7, 7, 17}
```

Das Ergebnis von primliste kann nun noch so umgewandelt werden, dass die Primfaktorzerlegung der Rückgabe der *Mathematica*-Funktion FactorInteger entspricht (als Übung!).
Zum Vergleich:

```
FactorInteger[2 * 2 * 3 * 5 * 7 * 7 * 7 * 17]
```

```
{{2, 2}, {3, 1}, {5, 1}, {7, 3}, {17, 1}}
```

Test der Primzahleigenschaft

Die simpelste (aber auch am wenigsten effiziente) Überprüfung einer Zahl *n* auf Primheit geschieht durch eine Funktion wie diese:

```
prim[n_] := (n >= 2 && n == primdiv[n])
```

```
prim[200 560 490 131] // Timing
```

```
{8.62 Second, True}
```

Zum Vergleich:

```
PrimeQ[200 560 490 131] // Timing
Divisors[200 560 490 131] // Timing
```

```
{0. Second, True}
```

```
{0.05 Second, {1, 200 560 490 131}}
```

Eine elementare Hinführung zu den effizienteren Primzahltests (von denen auch *Mathematica* Gebrauch macht) findet man z.B. bei [Bartholomé, u.a. 1995] und [Niederdrenk-Felgner, u.a. 1988].