IN DIESEM KAPITEL

- Was sind Algorithmen?
- Wie man Algorithmen klar formuliert
- Der Pseudocode
- Die Korrektheit von Algorithmen

Kapitel 1
Algorithmen

Die »bedrohliche Macht der Algorithmen« und die »finsteren Datenkraken«, die sie nutzen, um unser Leben in ihren Griff zu bekommen, sind in aller Munde. Gemeint sind natürlich Algorithmen zur Analyse der Daten, die jeder bei seinem Gang durchs Leben produziert. Man mag davon halten, was man will, ein Gutes hat die Sache: Jeder hat den Begriff *Algorithmus* schon einmal gehört. Doch was genau ist ein Algorithmus?

Das sind Algorithmen

Algorithmen sind Verfahren, mit denen ein Problem gelöst oder eine Aufgabe erfüllt wird. Ein Verfahren braucht jemanden, der es ausführt. Dieser Jemand kann vielleicht nur ein einziges Verfahren, dann ist es eine spezialisierte Maschine. Papas alte Quarzuhr aus den Achtzigern kann zum Beispiel nichts weiter als die Zeit anzeigen und lässt sich auch nicht umprogrammieren. Die geniale Idee der Informatik sind aber die allgemeinen Maschinen, die *Computer*, die Beschreibungen beliebiger Verfahren akzeptieren und ausführen können.

Die Verfahren für Computer werden als *Programme* verfasst. Warum reden wir also noch von Algorithmen und nicht gleich von Programmen? Nun, weil Programme und Algorithmen nicht das Gleiche sind. Algorithmen sind der wesentliche Kern, der Gedanke, der durch völlig verschiedene Programme ausgedrückt werden kann. Programme werden in Programmiersprachen aufgeschrieben, Algorithmen meist in *Pseudocode*. Pseudocode ist nicht noch eine Programmiersprache. Es ist eine Notation für Algorithmen, also eine spezielle Schreibweise, mit der sich Menschen besser über den Ablauf eines Verfahrens austauschen können. Pseudocode ist eine einfache und doch klare Methode, Algorithmen aufzuschreiben, die unabhängig von den Moden der Programmiersprachen auch noch in 20 Jahren aktuell und lesbar sein wird wie heute.

In diesem Abschnitt beschäftigen wir uns darum mit folgenden Themen etwas genauer:

- ✔ Was genau versteht man unter einem Algorithmus?
- ✔ In welchem Verhältnis stehen
  - Programme und Algorithmen sowie
  - Funktionen und Algorithmen?
- ✔ Was ist Pseudocode und warum wird er verwendet?
- ✔ Warum spielt die mathematische Modellierung eine wichtige Rolle bei der Formulierung von Algorithmen?

Algorithmen lösen Probleme

Backrezepte sind Algorithmen. Natürlich werden dabei keine Daten verarbeitet, sondern Backzutaten. Auch wenn man bei einem Algorithmus heute an Daten denkt, so ist doch der Begriff *Algorithmus* etwas allgemeiner definiert.

Ein *Algorithmus* ist ein

- ✔ wohldefiniertes,
- ✔ endlich beschreibbares,
- ✔ schrittweises
- ✔ Verfahren zum Lösen eines Problems oder der Erfüllung einer Aufgabe.

Bei einem Algorithmus geht es darum, dass nur Dinge formuliert werden, die wirklich machbar sind und keinerlei »magische Aktionen« erfordern. Beispielsweise ist folgendes Verfahren *kein* Algorithmus:

1. Hole einen Lottoschein.
2. Mache eine Zeitreise in die nächste Woche und beobachte, welche Zahlen gezogen werden.
3. Trage diese in den Schein ein und
4. gib ihn im Lottobüro ab.

Die notwendige Zeitreise ist nämlich etwas, das nur von ganz besonders befähigten Personen oder Maschinen ausgeführt werden kann. Wer einen Algorithmus liest und versteht, soll anschließend wissen, was genau er zu tun hat und wie er es tun soll. Auch das Pizzarezept in Abbildung 1.1 ist kein guter Algorithmus, denn auch wenn damit jeder zu einer Pizza kommt, lässt es doch zu viele Fragen offen. Und selbst wenn man in Einzelfällen vielleicht darüber streiten kann, was genau »wohldefiniert« ist und was nicht – das Prinzip ist sicher klar.

Abbildung 1.1: Dieses Backrezept erfüllt nicht die strengen Anforderungen an gute Algorithmen.

Bei den Algorithmen, die uns in diesem Buch am meisten interessieren, werden allerdings keine Backzutaten, sondern Daten verarbeitet. Freiheit von Magie bedeutet, dass alle Aktionen des Algorithmus auch von einem einfachen Computer ausgeführt werden können. Computer mit adressierbaren Speicherzellen, in denen Werte liegen, die durch Zuweisungen und Rechenoperationen modifiziert werden können, sind die Basis für alle solchen Algorithmen.

Jeder Algorithmus hat ein Ziel. Er soll eine Aufgabe erfüllen oder ein Problem lösen. Beginnen wir mal mit einer einfachen Aufgabe, die wir großspurig *Summationsproblem* nennen.

> **Problem:** Summation
>
> **Eingabe:** eine natürliche Zahl n
>
> **Berechne:** die Summe s aller natürlichen Zahlen, die kleiner oder gleich n sind:

$$s = \sum_{i=1}^{n} i = 1 + 2 + \cdots + n$$

Wollen wir die Summe bis zu einem bestimmten Wert berechnen, sagen wir bis $n = 100$, dann haben wir es mit einer *Instanz* des Problems zu tun. Die allgemeine Fassung der Aufgabenstellung ist also das Problem, eine ganz spezielle Aufgabenstellung mit festgelegten Eingabedaten ist eine Probleminstanz.

Probleme kann man auf verschiedene Arten lösen. Am einfachsten ist es, wenn man die Lösung in einem Buch nachschlagen oder im Internet suchen kann. Aber das zählt hier nicht. Unsere Methoden sollen das Problem nämlich selbst lösen.

Für das Summationsproblem gibt es zwei Verfahren, von denen zumindest das erste jeder kennt. Entweder können wir die Zahlen, eine nach der anderen, aufaddieren. Alternativ können wir auch die sogenannte gaußsche Summenformel anwenden (Abbildung 1.2). Ein Problem, zwei Algorithmen, die auf unterschiedlichen Ideen zur Lösung des Problems beruhen. Hier zunächst der naive Aufaddier-Algorithmus: »Nimm als Eingabe eine natürliche Zahl *n*. Nun initialisiere die Summe *s* mit 0. Zähle anschließend alle Zahlen von 1 bis *n* auf und addiere sie jeweils zu *s* hinzu. Gib am Ende *s* zurück.«

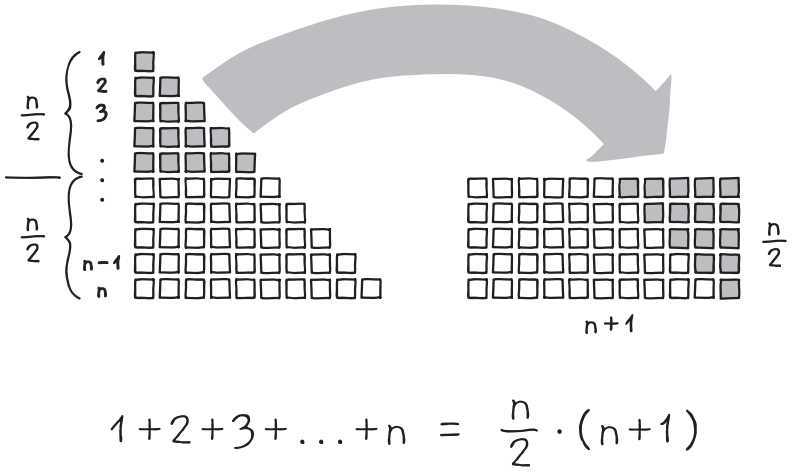

Abbildung 1.2: Die gaußsche Summenformel

Ein einfacher Algorithmus wie dieser lässt sich noch ganz gut auf Deutsch hinschreiben. Sobald die Algorithmen aber etwas komplizierter werden – und es werden in diesem Buch noch um einiges kompliziertere Algorithmen vorkommen –, wird es mit Pseudocode übersichtlicher. Hier also der gleiche Algorithmus in Pseudocode:

```
Summation1(n)                 n: eine natürliche Zahl
    s ← 0                     beginne mit 0 als bisher berechneter Summe
    s ← s + 1                 addiere 1 zur Summe s
    if n = 1                  wenn n = 1 ist,
        return s              fertig
    s ← s + 2                 addiere 2 zur Summe s
    if n = 2                  wenn n = 2 ist,
        return s              fertig
```

```
s ← s + 3                    addiere 3 zur Summe s
if n = 3                     wenn n = 3 ist,
    return s                 fertig
s ← s + 4                    addiere 4 zur Summe s
if n = 4                     wenn n = 4 ist,
    return s                 fertig
...                          und so weiter, und so weiter
```

Die drei Pünktchen »...« sollen andeuten, dass der Pseudocode immer weiter und weiter und weiter geht. Schließlich kann das *n* beliebig groß sein. Aber Moment mal, das ist ja blöd, denn ein Algorithmus soll schließlich *endlich beschreibbar* sein! Wenn man einen unendlich langen Text braucht, um den Algorithmus aufzuschreiben – dann ist das überhaupt kein Algorithmus. Zugegeben: Wir haben auch keinen unendlich langen Text hingeschrieben (sonst wäre das Buch ja viel zu dick geworden), sondern ihn mit »...« abgekürzt. Diese drei Punkte sind allerdings ein wenig ungenau, denn es wird ja nicht direkt gesagt, was wiederholt werden soll. Wenn man den Algorithmus verstehen will, muss man im Abschnitt vor den drei Punkten nach einem wiederkehrenden Muster suchen. In diesem Fall wäre das zum Beispiel: »Die Zahl, die zu *s* addiert wird, wird jedes mal um 1 größer, und anschließend vergleicht man *n* auch immer mit dieser um 1 größeren Zahl«. Ein solcher Interpretationsbedarf ist aber umständlich, und darum wollen wir hier lieber eine Schreibweise verwenden, die *explizit*, also klipp und klar sagt, was genau zu tun ist:

```
Summation1(n)                n: eine natürliche Zahl
    s ← 0                    beginne mit 0 als bisher berechneter Summe
    for i ← 1, ..., n        nimm jede Zahl i von 1 bis n
        s ← s + i            und addiere sie zur bisher berechneten Summe
    return s                 fertig
```

Das ist immer noch der gleiche Algorithmus Summation1 wie oben, nur deutlicher hingeschrieben. Was die ganzen Zeichen wie der Pfeil ← oder Wörter wie **for** und **return** bedeuten, werden wir etwas später noch genauer sagen. Nun erst einmal zur alternativen Methode mit gaußscher Summenformel:

```
Summation2(n)
    s ← n·(n+1)/2            berechne die Summe mit der gaußschen Formel
    return s                 fertig
```

Bei der Summation handelt es sich um ein *Berechnungsproblem*, bei dem aus Eingabedaten Ausgabedaten berechnet werden. Wenn man nur eine Ja-/Nein-Antwort haben will, dann handelt es sich um ein *Entscheidungsproblem*, und wenn man die beste unter vielen möglichen Lösungen benötigt, dann hat man es mit einem *Optimierungsproblem* zu tun.

Die *Lösung*, die ein Algorithmus zu einer Instanz des Problems liefert, muss natürlich korrekt sein. Beim einfachen Aufaddieren ist die Korrektheit offensichtlich. Die Formel liefert ebenfalls das korrekte Ergebnis. Das ist nicht ganz so offensichtlich, aber mit etwas Nachdenken oder der Hilfe von Wikipedia kann man sich davon überzeugen.

Fassen wir zusammen:

Ein *Algorithmus* ist ein Verfahren zur Lösung eines *Problems*. Es gibt verschiedene Arten von Problemen:

- ✔ Berechnungsprobleme,
- ✔ Entscheidungsprobleme und
- ✔ Optimierungsprobleme.

Eine *Probleminstanz* ist ein Problem für ganz bestimmte Eingabedaten.

Algorithmen basieren auf einem einfachen Maschinenmodell

Am Beginn der Informatik standen folgende Fragen aus dem Bereich der Grundlagenforschung in Mathematik und Informatik: Was darf man denn überhaupt hinschreiben, wenn man eine Berechnung definiert? Welche Rechnungen können tatsächlich ausgeführt werden? Was also ist überhaupt »rechenbar«? Daraus ist die Theorie der *Berechenbarkeit* entstanden. Am Schluss war man sich einig, dass etwas »rechenbar« ist, wenn es von einer wohldefinierten einfachen Maschine ausgeführt werden kann. Von allen Maschinenmodellen ist die *Turingmaschine* (Abbildung 1.3) wohl die bekannteste. Sie besteht aus einem einfachen Steuerautomaten, der sozusagen das Programm der Maschine beinhaltet, und hat darüber hinaus ein unbegrenzt langes Speicherband, in dem Buchstaben aus einem endlichen Alphabet gespeichert werden können. Mit einem Schreib-/Lesekopf kann die Turingmaschine in jedem Schritt eine Zelle des Speicherbands auslesen und verändern und anschließend darf sie den Schreib-/Lesekopf um ein Feld nach links oder rechts bewegen. Das ist also eine denkbar einfache Maschine, die jedoch erstaunlich vielseitig ist.

Manch andere Maschinen sind eingeschränkter und man kann weniger mit ihnen berechnen als mit einer Turingmaschine. Nimmt man der Turingmaschine beispielsweise ihr Speicherband weg, sodass sie nur noch ein endlicher Automat ist, verliert sie deutlich an Möglichkeiten. Andere Maschinenmodelle hingegen können genauso viel wie die Turingmaschine, sie sind *Turing-vollständig*. Es gibt die unterschiedlichsten Turing-vollständigen Maschinenmodelle, die nach sehr verschiedenen Prinzipien funktionieren, aber wenn die eine Maschine ans Ziel kommt, dann schaffen das die anderen auch immer irgendwie.

Für die Theorie der Algorithmen verwendet man meist die sogenannte *Random Access Machine* (RAM), was auf Deutsch mit »Maschine mit wahlfreiem Speicherzugriff« übersetzt werden könnte. Das ist ein einfacher hypothetischer Computer mit einer unbegrenzten Zahl an adressierbaren Speicherstellen, auf dem Programme mit Zuweisungen und den üblichen Kontrollanweisungen ablaufen können. Das Gute an der RAM: Sie ist den heute üblichen Computern hinreichend ähnlich, sodass jeder mit ein wenig Programmiererfahrung sofort ein Gefühl dafür hat, was für die RAM eine »elementare« Operation sein könnte und was nicht. Wie reale Computer ist sie außerdem Turing-vollständig und eignet sich damit ausgezeichnet als Grundlage für Algorithmen. In Pseudocode soll alles erlaubt sein, was in Aktionen einer RAM übersetzt werden kann.

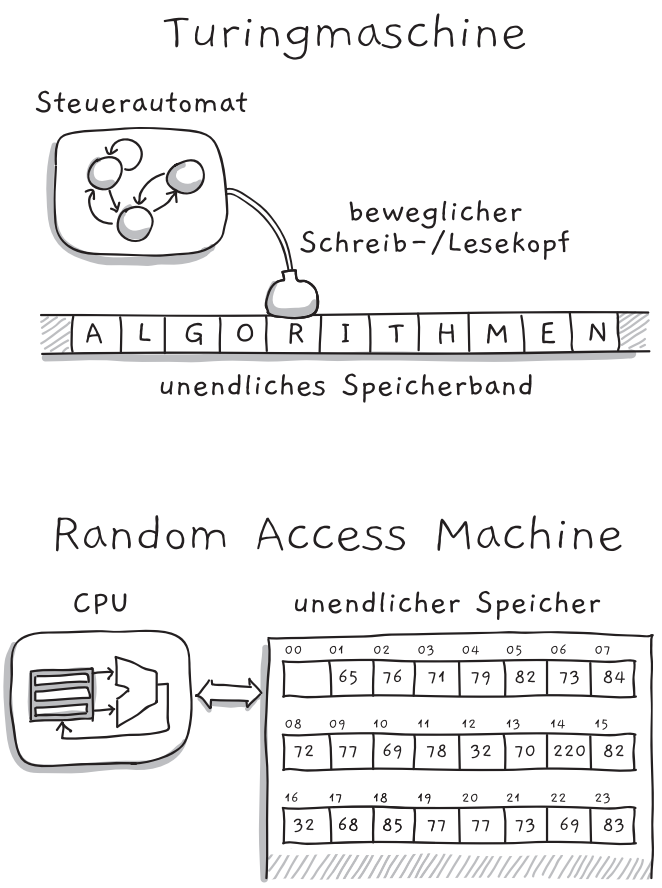

Abbildung 1.3: Turingmaschine und RAM sind zwei Maschinenmodelle.

Die RAM hat folgende Fähigkeiten:

- ✔ Sie hat Speicherstellen für beliebige endliche Daten.
- ✔ Diese können mit Zuweisungen und den üblichen arithmetischen Operationen manipuliert werden.

Daneben kann die RAM auch Kontrollanweisungen ausführen, nämlich

- ✔ bedingte Anweisungen,
- ✔ Schleifen sowie
- ✔ (rekursive) Funktions- und Prozedurdefinitionen.

Im Zweifelsfall müsste noch spezifiziert werden, welche Operationen im Detail zur Verfügung stehen. Reelle Zahlen sind beispielsweise nicht mit endlichem Speicher exakt darstellbar, sie gehören darum nicht zu den elementaren Daten der RAM. Wenn das

von Relevanz ist, dann muss zum Algorithmus angegeben werden, wie genau die Zahlen angenähert dargestellt werden und welche Operationen mit wie vielen Schritten diese Zahlen verarbeiten. Ist es irrelevant, dann wird das Thema ignoriert.

Algorithmen sind bewertbar

Wenn man nicht gerade ein Lehrer ist, der mal seine Ruhe haben will und seine Schüler dazu mit Rechenarbeiten beschäftigt hält, dann ist die zweite Methode der Summation, die mit der gaußschen Formel, natürlich die bessere. Statt wie Summation1 $n-1$ Additionen zu benötigen, wird von Summation2 die Summe von 1 bis n mit nur einer Addition, einer Multiplikation und einer Division berechnet.

Wir haben gerade die *Effizienz* der beiden Algorithmen bewertet. Genauer gesagt die *Laufzeit-Effizienz*. Dabei geht es darum, wie viele elementare Rechenschritte ein Algorithmus ausführen muss, um das Problem zu lösen. Die Zahl der Rechenschritte hängt in der Regel von der Eingabe ab. Für ein großes n werden bei der ersten Summationsmethode mehr Rechenschritte benötigt als bei einem kleinen n. Die Zahl der Rechenschritte bei der Anwendung der gaußschen Formel ist dagegen unabhängig von der Eingabe. Allerdings ist für sehr kleines n wie zum Beispiel $n = 2$ die Berechnung der Summe mit der gaußschen Formel

$$s = \frac{2 \cdot (2+1)}{2} = 3$$

sicher weniger effizient als das einfache Aufaddieren

$$s = 1 + 2 = 3$$

Um solche Ausreißer zu vermeiden, lässt man kleine Werte darum bei der Bewertung der Effizienz eines Algorithmus außer Acht.

Die Laufzeit-Effizienz eines Algorithmus sagt etwas darüber aus, wie die Zahl der Rechenschritte, die der Algorithmus für eine bestimmte Eingabe ausführt, von deren Größe abhängt. Mehr darüber erfahren Sie in Kapitel 2.

Fassen wir zusammen:

Ein Problem kann oft von mehreren unterschiedlichen Algorithmen gelöst werden, die sich in ihrer Effizienz unterscheiden. Die *Laufzeiteffizienz* ist das wichtigste Effizienzkriterium. Sie gibt an, wie viele Rechenschritte ein Algorithmus bei der Ausführung benötigt.

Algorithmen agieren in Modellwelten

Im Prinzip kann ein Algorithmus mit beliebigen Dingen umgehen. Ein Kochrezept ist ein Algorithmus, der mit Kochzutaten arbeitet. Wir betrachten aber nur Algorithmen, die Daten verarbeiten. Natürlich stammen die Daten meist aus der realen Welt und die vom Algorithmus berechneten Daten wirken auch wieder auf die reale Welt ein.

Ein Programm, das den kürzesten Weg zwischen zwei Orten sucht, läuft nicht in der Gegend herum, sondern arbeitet mit Daten, welche die Landschaft repräsentieren. Die digitale Version einer Landkarte enthält viele Informationen, von denen nur manche für das aktuelle Problem relevant sind. Bei dem Problem des kürzesten Weges sind das beispielsweise die Orte und Wege, die sie verbinden, und deren jeweilige Länge.

Ein Algorithmus zur Wegfindung wird so formuliert, dass er nur die auf das Wesentliche konzentrierten Informationen als Eingabe hat. Bei der Wegfindung ist das ein Graph: Objekte und ihre mit einer Bewertung versehen Verbindungen. In der Welt der Algorithmen und Datenstrukturen verwendet man gerne Konzepte aus der diskreten Mathematik: Mengen, Relationen, Graphen. Damit können die wesentlichen Informationen einer Problemstellung und deren Lösung leicht und übersichtlich zum Ausdruck gebracht werden.

Ein Programm, das mit einer bestimmte Form einer digitalen Landkarte arbeitet, muss natürlich mit den Daten in der konkreten Form arbeiten, in der sie zur Verfügung stehen. Das können Tabellen einer Datenbank sein, Klassen und Objekte einer objektorientierten Programmiersprache und vieles mehr. Ein Algorithmus löst das Problem in der mathematischen Modellwelt. Das ist sozusagen die »Zeichnung« der Lösung. Wird der Algorithmus in einer bestimmten Programmiersprache implementiert, dann transferiert man die Aktionen des Algorithmus auf »mathematischen Objekten« in Aktionen des Programms auf konkreten Daten (Abbildung 1.4).

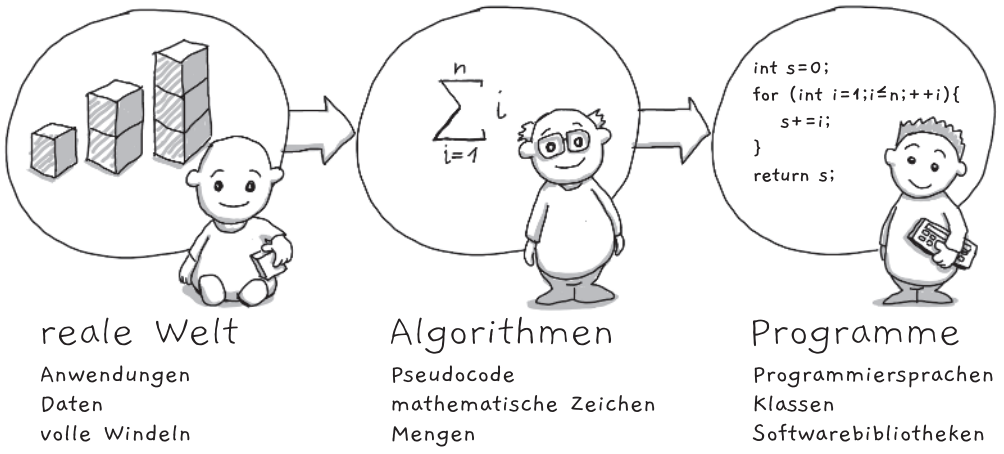

Abbildung 1.4: Algorithmen und Programme in Modellwelten

Algorithmen sind keine Programme

Ein Algorithmus ist ein Verfahren zur Lösung eines Problems, bei dem man sich auf das Wesentliche konzentriert: Welche relevanten Informationen werden mit welchen elementaren Schritten in eine Lösung transformiert? Das ist wie eine Melodie, die man singen oder auf diversen Musikinstrumenten spielen kann. Unterschiedlich, aber irgendwie doch immer das erkennbar Gleiche.

Pseudocode entspricht der Notenschrift, in der das Wesentliche aufgeschrieben wird: knapp, klar und unabhängig von bestimmten Instrumenten. Ein Programm ist etwas, das ein Prozessor ausführen kann, zum Beispiel eine bestimmte synaptische Struktur im Gehirn eines Sängers, eine Rille in einer Schallplatte, Nullen und Einsen in einer MP3-Datei und so weiter. Bei einer Ausführung beziehungsweise Aufführung wird das Programm vom Prozessor abgearbeitet (Abbildung 1.5). Die Musik ertönt entsprechend der Melodie, und wenn die Noten nicht verloren gehen, wird sie noch genauso zum Klingen gebracht werden können, wenn jede Erinnerung an Schallplatten und MP3-Dateien vergangen ist.

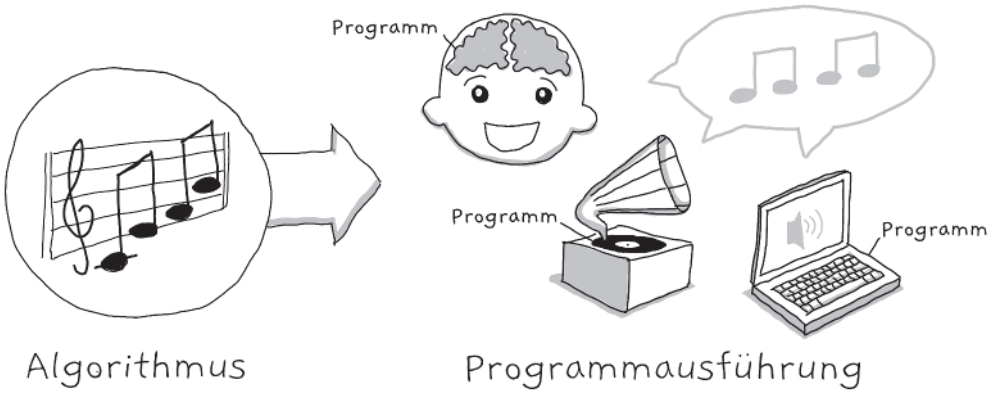

Abbildung 1.5: Ein Algorithmus und drei ausführbare Programme, die ihn implementieren

Programme und Algorithmen sind also unterschiedliche Dinge. Eine *Programmiersprache* hat auch viel mehr zu leisten, als nur eine Notation für die Ausführung von Algorithmen zu sein. Sie muss

- ✔ in effizienten Maschinencode übersetzbar oder von einem Interpreter ausführbar sein,
- ✔ von ihren Anwendern effektiv zu nutzen sein und
- ✔ bestimmten softwaretechnischen Anforderungen genügen.

Programmiersprachen haben immer ein exakt definiertes beschränktes und festes Repertoire an Ausdrucksmitteln. Dazu gehört vieles, das nicht direkt mit Algorithmen zu tun hat. Manche legen Wert darauf, dass die Sprache mit wenig Mühe schnell erlernbar ist, andere dass sie in einem bestimmten Anwendungsgebiet besonders effektives Arbeiten ermöglicht oder dass sie die Konstruktion von Anwendungen mit hunderttausenden von Zeilen Quellcode effektiv unterstützt etc.

Diese Ziele widersprechen sich teilweise. Um sie zu erreichen, gibt es diverse Ebenen, auf denen sich die Programmierer bewegen können, und eine mehr oder weniger umfangreiche Sammlung von Konzepten zur Modularisierung und Abstraktion: Klassen, Funktionen, Pakete, generische Klassen, Module, was auch immer. All das unterliegt nicht nur dem aktuellen Zeitgeist, der mal hierhin und mal dahin weht. Es unterscheidet sich auch drastisch von Sprache zu Sprache je nach deren Einsatzgebiet, vermuteter Kompetenz der Entwickler und so weiter.

All dies ist sehr wichtig, lenkt aber hier nur vom eigentlichen Kern der Dinge, dem Algorithmus, ab.

Algorithmen klar beschreiben

Algorithmen werden oft in Pseudocode beschrieben. Pseudocode ist für Menschen gemacht. Er darf darum deren Kreativität ausschöpfen. Alles Magische ist verboten, alles andere prinzipiell erlaubt. Magisch ist das, was sich nicht in Aktionen der RAM ausdrücken lässt. Im Sinne einer guten Lesbarkeit und Präzision versucht man in der Regel alles möglichst formal hinzuschreiben. Unter guten Freunden darf man aber auch manchmal etwas vage bleiben und einzelne Schritte zum Beispiel auf Deutsch beschreiben. Es muss aber immer klar sein, wie diese Notation in zählbare Aktionen auf der RAM umgesetzt werden kann.

Sprechen Sie Pseudocode?

Der Sprachumfang des Pseudocodes muss nicht bis ins Detail formal definiert werden, denn es handelt sich ja nicht um eine Programmiersprache, sondern um ein Kommunikationsmedium, mit dem sich Menschen über Algorithmen austauschen. Wir wollen trotzdem einmal die wichtigsten »Sprachelemente« durchgehen und uns darüber verständigen, was wir damit meinen.

In diesem Buch wird ein Pseudocode benutzt, der weithin übliche Sprachelemente benutzt und vielleicht ein wenig an die Programmiersprache Python erinnert. Anweisungsblöcke in einer Schleife oder solche, die in einer **if**-Anweisung ausgeführt werden, werden durch eine gemeinsame Einrückung nach rechts gekennzeichnet, enden also da, wo die Anweisungen nicht mehr rechts eingerückt sind.

Algorithmen

Algorithmen im Pseudocode entsprechen grob den Funktionen, Prozeduren oder Methoden in den Programmiersprachen. Jeder Algorithmus beginnt zunächst mit seinem Namen gefolgt von einer Liste an Eingabeparametern in runden Klammern:

```
Algo(n)          Algo ist ein Algorithmus, der n als Eingabe bekommt
```

Die gleiche Notation verwenden wir beim Aufruf eines Algorithmus.

Variablen und Zuweisungen

Variablen dienen zum Speichern von Daten aller Art. Sie ändern ihren Wert durch eine Zuweisung. Wir verwenden für Zuweisungen nicht wie oft üblich ein Gleichheitszeichen, sondern einen Pfeil:

```
i ← 17           die Variable i wird auf den Wert 17 gesetzt
j ← i + 1        j ist nun 18, i bleibt 17
```

Das Gleichheitszeichen »=« reservieren wir für Vergleiche.

Variablen sind in der Regel lokal, das heißt, sie gelten nur innerhalb des aktuellen Aufrufs des Algorithmus. Das ist insbesondere bei rekursiven Algorithmen nützlich.

Schleifen

Bei einer Schleife wird ein Anweisungsblock mehrfach ausgeführt. Wir verwenden zwei unterschiedliche Arten von Schleifen, die **for**- und die **while**-Schleife.

Bei der **for**-Schleife wird eine Variable nacheinander auf eine Reihe von Werten gesetzt und dann jeweils ein Anweisungsblock ausgeführt:

```
for i ← 1, ..., 17          i wird nacheinander auf 1, 2, ..., 17 gesetzt ...
    Algo(i)                 ... und für jedes i der Algorithmus Algo aufgerufen
Algo(100)                   diese Zeile wird erst nach der Schleife ausgeführt
```

Bei der **while**-Schleife wird zunächst eine Bedingung geprüft und im Fall, dass diese erfüllt ist, ein Anweisungsblock ausgeführt. Dies wird so lange wiederholt, bis die Bedingung zum ersten Mal nicht erfüllt wurde:

```
i ← 1                       die Variable i wird auf den Wert 1 gesetzt
while i ≤ 17                solange i ≤ 17 ist ...
    Algo(i)                 ... rufe Algo mit i als Parameter auf ...
    i ← i + 1               ... und erhöhe i um 1
Algo(100)                   diese Zeile wird erst ausgeführt, wenn i > 17 ist.
```

Bedingte Anweisungen

Mit **if** wird eine Bedingung geprüft, und wenn sie erfüllt ist, wird ein Anweisungsblock ausgeführt:

```
if i ≥ 17                   wenn i ≥ 17 ist, ...
    Algo(i)                 ... rufe den Algorithmus Algo mit i als Parameter auf
```

Folgt nach einem **if** ein **else**, so wird der nachfolgende Anweisungsblock ausgeführt, wenn die Bedingung der vorangegangenen **if**-Anweisung nicht erfüllt war:

```
if i ≥ 17                   wenn i ≥ 17 ist, ...
    Algo(i)                 ... rufe den Algorithmus Algo mit i als Parameter auf
else                        ansonsten, also wenn i < 17 ist, ...
    Algo(100)               ... rufe den Algorithmus Algo mit Parameter 100 auf
```

Sonstiges

Mit break verlässt man die aktuelle Schleife.

Mit return beendet man den aktuellen Aufruf eines Algorithmus. Dabei kann man optional auch noch einen Rückgabewert für den aufrufenden Algorithmus zurückgeben.

Weitere Sprachelemente werden bei der Diskussion der Datenstrukturen in den Kapiteln 3 bis 6 eingeführt.

Mathematische Ausdrücke sind erlaubt

Im Pseudocode verwenden wir viele Elemente der üblichen mathematischen Notation. Wir nutzen zum Beispiel Mengendefinitionen, bei denen man nicht jedes einzelne Element aufzählt, sondern die Elemente der Menge über eine Eigenschaft bestimmt. Wenn M zum Beispiel eine Menge ist, können wir Folgendes schreiben:

```
a ← {x ∈ M | x ist gerade}        a ist die Menge aller geraden Zahlen in M
```

Entsprechendes kann man auch für geordnete Folgen L definieren, nur dann mit eckigen statt geschweiften Klammern:

```
b ← [x ∈ L | x ist gerade]        b ist eine Liste, die alle geraden Zahlen aus L enthält
```

Derartige Konstrukte erleichtern die Formulierung von Algorithmen enorm. Sie werden von vielen modernen Programmiersprachen unterstützt (leider nicht von Java) und dort zum Beispiel *For-Comprehension* genannt.

Wenn man mag, kann man es auch noch weitertreiben und zum Beispiel noch ein Summenzeichen davor schreiben:

```
s ← ∑{x ∈ M | x ist gerade}       s ist die Summe aller geraden Zahlen in M
```

Das ist deshalb kein Problem, weil jeder Leser des Pseudocodes sofort weiß, wie er das in »normalen« Pseudocode umsetzen kann, nämlich zum Beispiel so:

```
s ← 0                             s auf 0 initialisieren
for x ← M                         alle Elemente in M durchlaufen
    if x ist gerade               sobald eine gerade Zahl x gefunden wurde, ...
        s ← s + x                 ... wird sie zu s addiert
```

Die Regel lautet immer: Alles ist erlaubt, solange jeder versteht, was gemeint ist und wie man es in elementare Operationen des Maschinenmodells (also zum Beispiel der RAM) umsetzen kann.

Algorithmen sprechen sogar Deutsch

Wenn man möchte, darf man im Pseudocode sogar einfach die deutsche Sprache verwenden, wie wir das zum Beispiel gerade mit dem Ausdruck »x **ist gerade**« getan haben. Das ist vollkommen in Ordnung, solange jedem im Prinzip klar ist, wie man auf der RAM prüfen kann, ob eine Zahl gerade oder ungerade ist. Bei Bedarf kann man das natürlich weiter ausformulieren, zum Beispiel mit »x **modulo 2 = 0**«. Dabei gehen wir davon aus, dass für die RAM wie für alle gängigen Computer die Operation **modulo** zu den Basisoperationen gehört, so dass sie den Rest einer Division zweier ganzen Zahlen berechnen kann.

Wenn man es eilig hat und in einem Algorithmus die Summe aller geraden Zahlen in einer Menge M benötigt, so könnte man das im Pseudocode auch einfach so schreiben:

$s \leftarrow$ die Summe aller geraden Zahlen in M

Aber Vorsicht: Im Zweifelsfall müssen Sie immer in der Lage sein, das genauer zu erklären, es also bis auf elementare Operationen einer RAM herunterzubrechen. Nachdem wir im vorangegangenen Abschnitt ausführlich darüber geredet haben, sollte das nun ja auch kein Problem mehr sein.

Algorithmen sind Lösungen, keine Probleme

Wenn wir mathematische Definitionen oder sogar deutsche Sätze in unsere Algorithmen einbauen, müssen wir allerdings aufpassen, dass am Ende auch wirklich ein Algorithmus dabei herauskommt. Ein Algorithmus ist ein Verfahren zum Lösen eines Problems, nicht das Problem selbst. Mathematiker haben zum Beispiel keine Hemmungen, Mengen zu definieren, von denen niemand weiß, wie man sie tatsächlich berechnen könnte. Bei manchen mathematischen Mengen kann man sogar beweisen, dass man sie gar nicht berechnen kann. Im Abschnitt *Das Halteproblem ist unlösbar* werden Sie eine solche Menge kennenlernen. Ein Problem bloß zu definieren, ist also etwas ganz anderes, als es auch zu lösen.

Lassen Sie uns dazu eine Aufgabe betrachten, die schon die Landvermesser im alten Ägypten zu lösen hatten, nämlich der Suche nach dem größten gemeinsamen Teiler zweier natürlicher Zahlen, also zweier Zahlen aus der Menge $\mathbb{N} = \{1, 2, 3, \ldots\}$:

Problem: Größter Gemeinsamer Teiler (GGT)

Eingabe: zwei natürliche Zahlen n und m

Berechne: die größte natürliche Zahl t, die sowohl ein Teiler von n als auch von m ist

Man könnte versuchen, das Problem auf folgende Weise zu lösen:

```
GGT1(n, m)                                  n und m seien natürliche Zahlen
    t ← max{x ∈ ℕ | x teilt n und x teilt m}    berechne den GGT
    return t
```

Lassen Sie uns der Klarheit halber die Menge noch einmal in Form einer Schleife ausformulieren:

```
GGT1(n, m)                                  n und m seien natürliche Zahlen
    t ← 1                                   1 ist ein gemeinsamer Teiler von n und m
    for x ← ℕ                               durchlaufe alle Zahlen aus ℕ in aufsteigender Reihenfolge
        if x teilt n und x teilt m          wenn x gemeinsamer Teiler von n und m ist, ...
            t ← x                           ... merke dir x
    return t
```

Das ist der gleiche Algorithmus, nur anders hingeschrieben. Allerdings gibt es ein Problem: Es gibt unendlich viele natürliche Zahlen, also wird die **for**-Schleife niemals damit fertig, sie aufzuzählen. Der Algorithmus ist in einer unendlichen Schleife gefangen, umgangssprachlich ausgedrückt: *Er hängt sich auf*.

Dass sich Algorithmen »aufhängen« können, wird uns gleich noch ausführlich beschäftigen. Zum Glück kann man den Algorithmus leicht retten: Alle Zahlen größer als n sind keine Teiler von n, und alle Zahlen größer als m keine Teiler von m. Der GGT muss also sowohl $\leq n$ also auch $\leq m$ bleiben, das heißt, es reicht aus, die Schleife bis zum Minimum von m und n laufen zu lassen:

```
GGT2(n, m)                          n und m seien natürliche Zahlen
    t ← 1                           1 ist ein gemeinsamer Teiler von n und m
    for x ← 2, ..., min(n, m)       durchlaufe alle Zahlen von 2 bis min(n, m)
        if x teilt n und x teilt m  wenn x gemeinsamer Teiler von n und m ist, ...
            t ← x                   ... merke dir x
    return t
```

Das können wir dann auch wieder verkürzt als mathematische Menge formulieren und wie folgt hinschreiben:

```
GGT2(m, n)                                  n und m seien natürliche Zahlen
    g ← min(n, m)                           g sei das Minimum von n und m
    t ← max{x ∈ {1, ..., g} | x teilt n und x teilt m}   berechne den GGT
  return t
```

Das ist zwar nicht der schnellste Weg zur Berechnung des GGT, aber immerhin eine Möglichkeit.

Algorithmen haben zählbare Schritte

Wenn wir später Algorithmen miteinander vergleichen wollen, so sollte exakt bestimmbar sein, wie viele elementare Schritte ein Algorithmus benötigt, wenn man ihn auf einer Eingabe einer gegebenen Größe ausführt. Die Laufzeit des Algorithmus GGT2 zur Berechnung des größten gemeinsamen Teilers wird zum Beispiel entscheidend vom Minimum der beiden Eingabewerte n und m beeinflusst: Bis zu diesem Wert läuft die Schleife.

In jedem Schleifendurchlauf des Algorithmus wird eine Zahl x daraufhin geprüft, ob sie ein Teiler von n und m ist. Diese Prüfung kann natürlich auch auf unterschiedliche Arten erfolgen. Man könnte beispielsweise x so lange von m beziehungsweise n subtrahieren, bis das Ergebnis 0 oder eine Zahl kleiner 0 ist. Endet man bei 0, dann ist x ein Teiler, ansonsten nicht. Damit hat unser Algorithmus dann die folgende Definition:

```
GGT3(n, m)                                  n und m seien natürliche Zahlen
    t ← 1                                   1 ist ein gemeinsamer Teiler von n und m
    for x ← 2, ..., min(n, m)               durchlaufe alle Zahlen von 2 bis min(n, m)
        if isDivisor(x, m) und isDivisor(x, n)   wenn x Teiler von n und m ist, ...
            t ← x                           ... merke dir x
    return t
```

```
isDivisor(x, z)                 prüft, ob x ein Teiler von z ist
    while z > 0                 solange noch z größer als 0 ist, ...
        z ← z − x               ... ziehe x von z ab
    return (z = 0)              wenn am Ende z = 0 ist, war z am Anfang
                                durch x teilbar, sonst nicht
```

Natürlich braucht ein Algorithmus, der die Teilbarkeit in einem Schritt bestimmen kann, weniger Schritte als einer, der für jeden Test eine Schleife laufen lassen muss. Die meisten modernen Prozessoren unterstützen die Modulo-Operation »n **modulo** x«, die den Rest einer ganzzahligen Division von n geteilt durch x berechnet. In vielen Programmiersprachen wird diese Operation mit einem %-Zeichen geschrieben, andere schreiben dafür mod. Ist dieser Rest gleich 0, so ist n offenbar durch x teilbar; ist der Rest hingegen ungleich 0, so ist n nicht durch x teilbar. Dieser Test geht schnell, in einem einzigen Schritt. Es ist darum legitim anzunehmen, dass der Test auf Teilbarkeit unabhängig von der Größe der involvierten Werte ist. Wir unterstellen unserer RAM einfach die entsprechende Fähigkeit.

Wenn der Algorithmus andererseits auf einer Maschine läuft, bei der dies nicht gegeben ist, dann sieht die Sache völlig anders aus. Die Berechnung kann dann unter Umständen nicht mehr in einem Schritt erfolgen, sondern benötigt zum Beispiel eine Extraschleife, deren Laufzeit von x und n abhängt. In diesem Fall sollte man das bei der Beschreibung des Algorithmus auch dringend so hinschreiben, damit niemand auf die irrige Idee kommt, dass die Teilbarkeit eine elementare Operation des hinter dem Algorithmus stehenden Maschinenmodells ist.

Mehr über die Abschätzung und den Vergleich der Laufzeiten von Algorithmen finden Sie in Kapitel 2.

Algorithmen sollten korrekt sein

Wenn Sie sich einen Algorithmus ausdenken und hinschreiben, dann möchten Sie natürlich auch, dass er korrekt ist. Er soll genau das tun, was Sie von ihm wollen, und zwar bei jeder zulässigen Eingabe. Wie aber kann man dafür sorgen, dass ein Algorithmus tatsächlich so funktioniert, wie man sich das wünscht?

Die kurze Antwort auf diese Frage: Dafür gibt es kein Patentrezept. Algorithmen können im Allgemeinen nämlich ziemlich schwer zu durchschauen sein.

Betrachten Sie zum Beispiel einmal den folgenden unbekannten Algorithmus, und überlegen Sie, was er berechnet, wenn man ihm zwei natürliche Zahlen x und y übergibt:

```
Riddle(n, m)                    n und m seien zwei natürliche Zahlen
    x ← n
    y ← m
    p ← 0
```

```
while x ≥ 1
    if x ist gerade
        x ← x/2
    else
        p ← p + y
        x ← (x−1)/2
    y ← 2 · y
return p
```

Am besten setzen Sie einfach ein paar Zahlen für n und m ein, finden heraus, was Riddle bei diesen Eingaben berechnet, und stellen anschließend eine Hypothese darüber auf, was der Algorithmus wohl berechnen könnte. Die Auflösung des Rätsels finden Sie im Abschnitt *Die Lösung des Rätsels*.

Algorithmen können sich aufhängen

Algorithmen sollten sich möglichst nicht »aufhängen«, also keine unendliche Laufzeit haben, weil sie sich zum Beispiel in einer unendlichen Schleife verfangen haben. Wenn sich ein Algorithmus einfach aufhängt, brauchen wir über seine Korrektheit gar nicht mehr nachzudenken. Ein Ausdruck wie

$$t \leftarrow \max\{x \in \mathbb{N} \mid x \text{ teilt } n \text{ und } x \text{ teilt } m\}$$

ist darum nicht erlaubt. Deswegen hängt sich der Algorithmus GGT1 im Abschnitt *Algorithmen sind Lösungen, keine Probleme* ja auch auf.

Im Fall einer Mengennotation, die ja letztlich nur eine verkürzte Schreibweise für eine **for**-Schleife ist, kann man das auch sehr gut erkennen und vermeiden, denn bei einer **for**-Schleife sieht man ja von Anfang an, wie oft die Schleife durchlaufen wird. Schwieriger wird das bei **while**-Schleifen oder auch bei rekursiven Algorithmen, also Algorithmen, die sich wiederholt selbst aufrufen. Da ist es oft gar nicht so leicht zu erkennen, was sie tun und ob sie überhaupt irgendwann einmal fertig werden. Betrachten Sie zum Beispiel folgenden einfachen Algorithmus:

```
Collatz(n)                          n: eine natürliche Zahl
    while n > 1                     solange n noch größer als 1 ist, tue Folgendes:
        if n ist gerade
            n ← n/2                 wenn n gerade ist, teile n durch 2
        else
            n ← 3 · n + 1           ansonsten multipliziere n mit 3 und addiere 1 dazu
```

Kommt dieser Algorithmus immer irgendwann zu einem Ende, oder gibt es Eingaben n, bei der er immer weiterläuft und niemals endet? Der Mathematiker Lothar Collatz hat diese Frage erstmals vor über 80 Jahren gestellt, und sie ist bis heute unbeantwortet geblieben. Zwar wurde noch keine Zahl n gefunden, bei der sich der Algorithmus aufhängen würde, umgekehrt hat aber auch noch niemand bewiesen, dass es keine solche Zahl gibt. Schaut man sich an, wie sich n im Laufe des Algorithmus verändert, so erkennt man, dass n immer

wieder wild rauf- und runterspringt. In Abbildung 1.6 sehen Sie das am Beispiel $n = 25$: Für diese Eingabe braucht der Algorithmus 23 Durchläufe der **while**-Schleife, und n erreicht dabei eine maximale Höhe von 88. Wären wir stattdessen bei $n = 27$ gestartet, hätten wir 111 Schleifendurchläufe benötigt, und n wäre dabei bis in eine Höhe von 9323 aufgestiegen. Größere Startwerte bedeuten aber nicht zwangsläufig längere Laufzeiten; bei $n = 32$ geht es zum Beispiel immer bergab und die »Talstation« ist nach nur fünf Schritten erreicht.

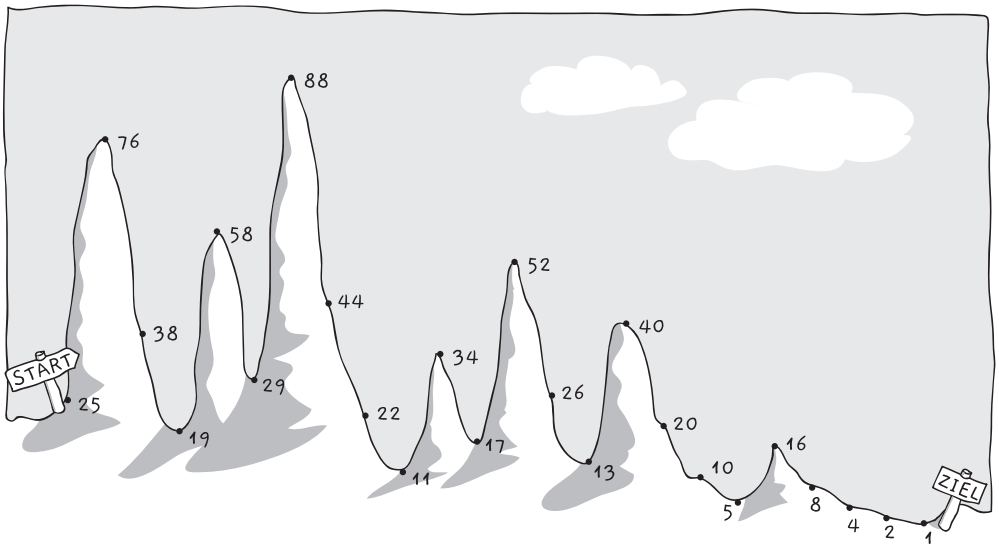

Abbildung 1.6: Der Startpunkt $n = 25$ für den Pfad durch das Collatz-Gebirge

Das Halteproblem ist unlösbar

Bei der Wanderung durch das »Collatz-Gebirge« sind Abstürze also nicht ausgeschlossen. Aber warum ist es bloß so schwer herauszufinden, ob ein Algorithmus irgendwann endet oder nicht? Könnte man nicht eine generelle Methode finden, um genau das zu entscheiden? Lassen Sie es uns als algorithmisches Problem formulieren:

> **Problem:** das Halteproblem
>
> **Eingabe:** ein Algorithmus A und eine Eingabe x
>
> **Berechne:** Entscheide, ob A gestartet auf x irgendwann anhält oder ob sich der Algorithmus aufhängt

Leider gibt es keinen Algorithmus, der das Halteproblem für beliebige Eingaben A und x löst. Das kann man sogar beweisen. Das Halteproblem ist ein klassisches Beispiel für ein nicht berechenbares Problem. Zwar könnte man einfach A mit der Eingabe x starten und sehen, was passiert, wenn aber A endlos weiterläuft, dann findet man das auf diese Weise auch erst nach unendlich langer Zeit – also niemals – heraus. Der Ausdruck

$$\{(A, x) | A \text{ ist ein Algorithmus, } x \text{ eine Eingabe, und } A(x) \text{hängt sich nicht auf}\}$$

ist also ein Beispiel für eine Menge, die man zwar durchaus mathematisch korrekt definieren kann, für die es aber keine Berechnungsmethode gibt.

Zwar kann man bei vielen Algorithmen tatsächlich beweisen, dass sie sich nicht aufhängen, es gibt nur kein Verfahren, mit dem sich das für beliebige Algorithmen zeigen ließe.

Wenn wir es also schwer haben, herauszufinden, ob sich ein Algorithmus aufhängt oder nicht, warum verändern wir dann nicht einfach unser Maschinenmodell auf eine Weise, dass sich Algorithmen grundsätzlich nicht mehr aufhängen können? Lassen Sie uns doch die ganzen bösen **while**-Schleifen und Rekursionen verbieten, die uns so große Probleme bereiten. Stattdessen könnten wir uns zum Beispiel auf die sicheren **for**-Schleifen beschränken. Ja, das würde gehen, hätte allerdings einen Nachteil: Unser Maschinenmodell wäre anschließend nicht mehr Turing-vollständig. Gäbe es nämlich ein Turing-vollständiges Maschinenmodell, das sich nicht aufhängen kann, so könnte man damit das Halteproblem lösen, und das ist wie gesagt gar nicht möglich. So gesehen ist es also eigentlich ganz gut, dass Algorithmen grundsätzlich die Möglichkeit haben, sich aufzuhängen, denn nur dadurch können Sie ihr volles Potenzial ausschöpfen.

Algorithmen können sich »aufhängen«, das heißt, sie laufen immer weiter und enden nie. Ob sich ein Algorithmus aufhängt oder nicht, lässt sich leider oft nur schwer abschätzen, und es gibt es keine formale Methode, mit der sich das immer sicher verhindern lässt.

Algorithmen richtig verstehen

Wenn es schon schwer ist, festzustellen, ob ein Algorithmus irgendwann anhält oder nicht, dann kann es sicherlich auch nicht leichter sein, herauszufinden, was der Algorithmus überhaupt tut, und ob er auch das tut, was man sich von ihm erwünscht.

Die Lösung des Rätsels

Um das zu veranschaulichen, lassen Sie uns über den geheimnisvollen Algorithmus Riddle weiter vorne in diesem Kapitel reden.

Man kann ziemlich leicht erkennen, dass sich der Algorithmus immerhin nicht aufhängt. Bei jedem Durchlauf der while-Schleife verringert sich nämlich der Wert von x, und das tut er so lange, bis $x < 1$ ist und die Schleife abbricht.

Was genau berechnet der Algorithmus denn nun? Haben Sie es herausgefunden?

Die Antwort ist einfach: Riddle berechnet das Produkt aus den beiden Eingaben n und m, also

$$\textbf{Riddle}(n, m) = n \cdot m$$

Übrigens nennt man den Algorithmus auch »ägyptische Multiplikation«. Die Methode scheint auch tatsächlich zu funktionieren. Wenn man eine Stichprobe macht und irgendwelche Werte für x und y einsetzt, kommt jedenfalls das Richtige dabei heraus.

Aber können wir uns sicher sein, dass Riddle tatsächlich *immer* das Produkt von n und m berechnet? Nur weil es bei ein paar Beispielen für n und m gestimmt hat, muss es ja nicht für alle Eingaben funktionieren. Wenn wir ganz sicher sein wollen, müssten wir das Ergebnis eigentlich für alle möglichen natürlichen Zahlen n und m überprüfen. Aber das geht nicht, denn dann hätten wir ja unendlich viel zu tun.

Nun hätte man das Gleiche auch für praktisch alle anderen Algorithmen sagen können, die wir bisher in diesem Buch besprochen haben. Wie können wir uns beispielsweise sicher sein, dass GGT2 im Abschnitt *Algorithmen sind Lösungen, keine Probleme* auch tatsächlich den größten gemeinsamen Teiler ausrechnet, und zwar für *alle* möglichen Eingaben? Auch hierfür gibt es ja wieder unendlich viele Möglichkeiten, und die können wir unmöglich alle ausprobiert haben. Der Unterschied zwischen den beiden Algorithmen ist, dass GGT2 sehr einfach zu verstehen ist, während man bei Riddle nicht sofort einsieht, warum der Algorithmus überhaupt funktioniert.

Wenn wir uns sicher sein wollen, dass ein Algorithmus korrekt ist, müssen wir ihn also *wirklich* verstehen. Ihn ganz und gar durchdringen. Dafür reicht es nicht aus, dass man den Algorithmus für Beispieleingaben Schritt für Schritt nachvollziehen kann. Jede einzelne Zeile in Riddle ist klar und verständlich, aber in ihrem Zusammenspiel ergeben sich Konsequenzen, die man nicht direkt überblickt.

Am besten wäre es, wenn uns jemand den Algorithmus erklären könnte; wenn wir einen Text hätten, der mit klaren, nachvollziehbaren Argumenten schlüssig begründet, warum der Algorithmus das Produkt seiner Eingaben n und m berechnet. Was wir also brauchen, ist ein *Beweis*.

Wenn bei einem Algorithmus nicht offensichtlich ist, dass er das tut, was er soll, dann benötigt man einen Beweis, um ihn richtig zu verstehen und seine Korrektheit einzusehen.

Korrektheit beweisen

Wenn ein Algorithmus nur eine feste Anzahl von Anweisungen nacheinander ausführt, dann ist es noch vergleichsweise einfach, seine Korrektheit zu überprüfen. Schwieriger wird es, wenn die Anzahl der Anweisungen, die während eines Algorithmus ausgeführten werden, nicht von Anfang an feststeht, sondern von der Eingabe abhängt. In Riddle gibt es eine **while**-Schleife, die so lange durchlaufen wird, bis der Wert $x < 1$ wird. Am Anfang wird x auf den Eingabewert n gesetzt und anschließend bei jedem Schleifendurchlauf halbiert und dabei abgerundet, bis x schließlich den Wert 0 erreicht hat und die Schleife abbricht. Je größer n ist, desto häufiger wird also die Schleife durchlaufen. Da n beliebig groß sein kann, kann die Schleife beliebig oft durchlaufen werden. Unser Beweis soll aber nicht aus unendlich viel Text bestehen, also brauchen wir eine Methode, die Korrektheit des Algorithmus zu begründen, ohne für jeden einzelnen Schleifendurchlauf extra zu argumentieren.

Der Trick, den man in diesem Fall gewöhnlich verwendet, heißt *Induktionsbeweis*. Die Idee dabei ist, dass man wie folgt argumentiert:

> »Wenn bis zum i-ten Schleifendurchlauf alles korrekt funktioniert hat, dann funktioniert auch im nachfolgenden $i + 1$-ten Schleifendurchlauf alles korrekt.«

Dabei muss man sich natürlich erst einmal im Klaren darüber sein, was es genau heißen soll, dass bei einem Schleifendurchlauf »alles korrekt funktioniert«. Hierzu verwendet man gewöhnlich eine sogenannte *Schleifeninvariante*. Das ist zum Beispiel eine Größe, die sowohl vor als auch nach jedem Schleifendurchlauf gleich bleibt, oder allgemein eine Aussage, die sowohl vor als auch nach jedem Schleifendurchlauf gültig ist. Im Fall der ägyptischen Multiplikation (also des Algorithmus `Riddle` weiter vorne in diesem Kapitel) könnten wir zum Beispiel folgende Gleichung als Schleifeninvariante (SI) verwenden:

$$n \cdot m = x \cdot y + p$$

Der Korrektheitsbeweis läuft dann so:

1. Ganz am Anfang, also bevor die **while**-Schleife das erste Mal durchlaufen wird, gilt die SI, denn es wird $x \leftarrow n$, $y \leftarrow m$ und $p \leftarrow 0$ gesetzt, und das bedeutet, dass $n \cdot m = x \cdot y + p$ ist (*Induktionsanfang*)

2. Angenommen, wir stehen am Anfang eines Schleifendurchlaufs, und bisher gilt die SI. Wir müssen zeigen, dass die SI auch am Ende des Schleifendurchlaufs gilt (*Induktionsschritt*). Zwei Fälle sind zu unterscheiden (siehe Abbildung 1.7):

   a. Fall 1: x ist am Anfang des Schleifendurchlaufs *gerade*. Dann wird x halbiert und y verdoppelt, das Produkt $x \cdot y$ bleibt also gleich. Da sich p ebenfalls nicht ändert, verändert sich der Ausdruck $x \cdot y + p$ nicht, und die SI behält auch nach dem Schleifendurchlauf ihre Gültigkeit.

   b. Fall 2: x ist am Anfang des Schleifendurchlaufs *ungerade*. Nun wird beim Halbieren von x abgerundet, und wie man in Abbildung 1.7 sehen kann, wird dadurch das Produkt $x \cdot y$ um die schwarzen Kästchen kleiner. Diese gehen zum Glück nicht verloren, sondern werden zu p hinzugezählt. Am Ende der Schleife hat sich $x \cdot y + p$ also nicht geändert, die SI gilt somit immer noch.

3. Die **while**-Schleife endet, wenn x nicht mehr ≥ 1 ist. Das passiert, sobald $x = 0$ geworden ist; und wenn $x = 0$ ist, dann ist auch $x \cdot y = 0$. Da die SI über alle Schleifendurchläufe hinweg bestanden hat, gilt sie auch jetzt noch, und somit ist $n \cdot m = x \cdot y + p = p$. Der Algorithmus liefert p als Ergebnis zurück und berechnet somit das Produkt aus n und m.

Korrektheitsbeweise von Algorithmen sind häufig Induktionsbeweise, die über die Durchläufe einer Schleife hinweg die Gültigkeit einer Schleifeninvariante belegen. Dabei ist eine Schleifeninvariante etwas, das über die Schleifendurchläufe hinweg »invariant« ist, also gleich bleibt.

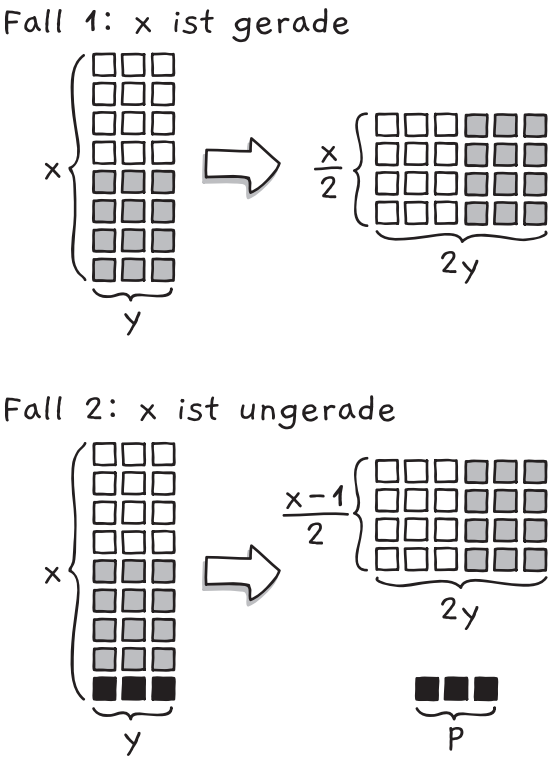

Abbildung 1.7: Die zwei Fälle der ägyptischen Multiplikation

Auf der Suche nach einem Korrektheitsbeweis können Schleifeninvarianten also nützlich sein. Dabei reicht es allerdings nicht aus, *irgendeine* Schleifeninvariante zu finden, denn letztlich soll sie auch dazu geeignet sein, die Korrektheit des Algorithmus zu beweisen. Aber wie findet man denn eine geeignete Schleifeninvariante? Sie ahnen es sicherlich: Auch dafür gibt es kein Patentrezept. Beim Führen von Beweisen sind Sie leider auf ihre eigene Kreativität angewiesen.