Kapitel 3

Einführung in die Berechenbarkeitstheorie

Was können wir alles berechnen – und was nicht? Wir füllen den Begriff »berechenbar« mit Leben und zeigen, was verschiedene Modelle dafür an Konsequenzen mit sich bringen.

Zwei grundlegende Fragestellungen beschäftigen uns in der theoretischen Informatik. Die aus Sicht der Praxis offensichtlich interessante Frage ist: Wie schnell können wir ein Problem lösen? Jede Person, die schon einmal vor dem Computer auf ein Ergebnis gewartet hat, wird die Relevanz dieser Frage nachvollziehen können.

Davor kommt jedoch eine Frage, die sich Außenstehende häufig gar nicht stellen: Welche Probleme können wir überhaupt mit einem Computer lösen, welche sind also *berechenbar*? Die vielleicht etwas überraschende Antwort lautet: Im Vergleich zur Anzahl der Probleme, die wir nicht lösen können, ist es nur für einen verschwindend kleinen Teil von Problemen möglich, Lösungsalgorithmen zu entwickeln.

Das allein könnte aus Anwendungssicht herzlich egal sein, wenn die nicht lösbaren Probleme irrelevant wären. Leider ist dem nicht so, wie ich Ihnen in diesem Kapitel unter anderem an dem prominenten Beispiel des *Halteproblems* zeigen werde: Es ist nicht möglich, für beliebige Programme festzustellen, ob deren Berechnung in einer Endlosschleife festhängt und nie ein Ergebnis produzieren wird.

Berechnungsmodell

Damit wir solche Aussagen überhaupt tätigen können, müssen wir aber zunächst klären, was Berechenbarkeit eigentlich ist. Dafür entwickeln wir *Berechnungsmodelle*: einfache abstrakte Rechenmaschinen, die wir präzise analysieren können. Unsere simplen Modelle sind zum Teil aber genauso leistungsstark (bezogen auf die Problemlösungsfähigkeiten) wie die aus dem Alltag bekannten Maschinen. An den Modellen können wir deshalb

zeigen, was echte Computer leisten können, die weitaus komplizierter aufgebaut sind und sich deswegen nicht so einfach direkt analysieren lassen.

Nicht alle unsere Modelle werden so viel leisten können; diese haben dann aber andere Vorteile: Zum Beispiel ist es mit einigen schwächeren Modellen möglich, das oben beschriebene Halteproblem zu beantworten. Aus diesem Grund kommen auch diese Modelle in der Praxis zur Anwendung, wenn für eine Berechnung nicht die volle Komplexität der leistungsstarken Modelle benötigt wird – im Austausch bekommt man dann nämlich ein paar mehr Garantien über die Korrektheit eines Algorithmus in diesem Modell.

Wir beginnen damit, zunächst die »Grundregeln« für Berechnungen festzulegen und einige Begriffe einzuführen, die modellunabhängig verwendet werden.

3.1 Algorithmus

Algorithmus

Berechnungen werden in der Informatik immer von einem *Algorithmus* durchgeführt. Ein Algorithmus ist eine endliche Abfolge von unmissverständlichen Befehlen eines vorgegebenen Befehlssatzes und wandelt anhand dieser Anweisungen eine *Eingabe* in eine *Ausgabe* um.

Wie Algorithmen diese Umwandlung durchführen, beschäftigt uns später noch genauer – also zum Beispiel wie genau solche Befehle aussehen können. Für den Moment betrachten wir Algorithmen von außen als Blackbox, deren innere Abläufe wir nicht kennen. Wir schauen uns wie in Abbildung 3.1 nur an, *was* ein Algorithmus tut, also wie Ein- und Ausgabe zusammenhängen.

Abbildung 3.1 Grundschema eines Algorithmus

Diese Betrachtung passt gut zu unserer mathematischen Vorstellung von Funktionen: Die Funktion erhält ein Argument als Eingabe und produziert einen Funktionswert als Ausgabe. Ist einem Argument kein Funktionswert zugewiesen, ist die Funktion also an der Stelle undefiniert, so korrespondiert dies damit, dass ein Algorithmus in einer Endlosschleife festhängt und für diese Eingabe nie ein Ergebnis liefert.

Ganz ähnlich betrachten wir auch Funktionen als Blackbox. Beispielsweise ist für uns $f(x) = 2x$ und $f(x) = x + x$ dieselbe Funktion, die ihre Eingabe verdoppelt. Zwar unterscheidet sich die Berechnungsvorschrift (im Algorithmus wären dies die verwendeten Befehle), die Zuordnung von Ausgaben zu Eingaben ist aber identisch.

Berechnen

In der theoretischen Informatik betrachten wir Algorithmen als Umsetzung von Funktionen; wir sagen auch: Ein Algorithmus *berechnet* eine Funktion.

3.2 Zu viele Funktionen

Wir können nun die Menge aller möglichen Funktionen ansehen als die Menge der Dinge, die wir potenziell berechnen wollen. Diese Menge ist unvorstellbar groß. Bereits die Menge der totalen Funktionen ist überabzählbar unendlich! Einen Beweis hierfür finden Sie in Abschnitt 2.8.11 bei den Lösungen der Aufgaben.

Algorithmencodierung

Auf der anderen Seite steht die Menge aller möglichen Algorithmen. Jeder einzelne Algorithmus kann vollständig als natürliche Zahl codiert werden: Stellen Sie sich den Algorithmus als Text notiert und in einer Datei auf Ihrem Computer gespeichert vor. Dann liegt diese Datei auf der Festplatte in Form von Nullen und Einsen vor. Dieselbe Folge von Nullen und Einsen können wir nun als (sehr große) natürliche Zahl interpretieren.

Nicht berechenbar

Es kann also maximal so viele Algorithmen geben, wie es natürliche Zahlen gibt, also abzählbar unendlich viele. Im Vergleich zu den überabzählbar vielen totalen Funktionen ist die Menge der Algorithmen demnach verschwindend klein. Da jeder Algorithmus genau eine Funktion berechnet, muss es also sehr viele Funktionen geben, die nicht durch Algorithmen realisiert werden können und somit *nicht berechenbar* sind.

3.3 Das Halteproblem

Tatsächlich gibt es diverse nicht-berechenbare Funktionen, die eigentlich einen praktischen Nutzen hätten. Die bekannteste nicht-berechenbare Funktion löst für beliebige Algorithmen das sogenannte *Halteproblem*.

> **Halteproblem**
> Gegeben als Eingabe sind ein Algorithmus und eine natürliche Zahl. Hält dieser Algorithmus an, wenn er mit der Zahl als Eingabe aufgerufen wird, oder gerät er in eine Endlosschleife?

Haben Sie sich schon einmal gefragt, ob sich Ihr Computer aufgehängt hat oder ob er vielleicht nur etwas länger als erwartet für eine Aktion benötigt und gleich wieder normal funktioniert? Es stellt sich heraus, dass kein Algorithmus diese Entscheidung immer korrekt treffen könnte. Wir beweisen diese Aussage nun mit einem Widerspruchsbeweis.

Angenommen, es gäbe einen Algorithmus, der das Halteproblem entscheiden kann, also eine berechenbare Funktion umsetzt, die das obige Problem löst. Dieser Algorithmus (nennen wir ihn A) bekommt also den Quellcode eines Algorithmus B (wie oben beschrieben als Zahl codiert) sowie eine Zahl x als Eingabe. Als Ausgabe soll er *wahr* ausgeben, wenn B, auf x ausgeführt, anhält, und *falsch*, wenn dies nicht der Fall ist. Abbildung 3.2 stellt das noch einmal als Blackbox dar.

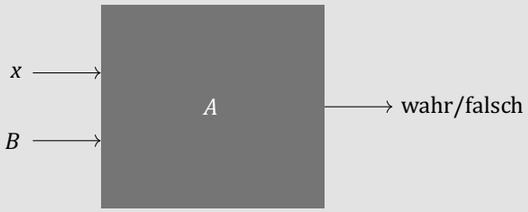

Abbildung 3.2 Der hypothetische Lösungsalgorithmus A für das Halteproblem

Aus diesem Algorithmus A konstruieren wir einen neuen Algorithmus A', der intern A verwendet. A' bekommt als Eingabe einen Algorithmus C, ruft den Algorithmus A mit $x = C$ und $B = C$ auf und behandelt dann die Ausgabe von A folgendermaßen: Falls A als Ausgabe *wahr* liefert, geht A' in eine

Endlosschleife. Sollte A dagegen *falsch* zurückgeben, so gibt A' die Zahl 0 aus. Der neue Algorithmus ist in Abbildung 3.3 dargestellt.

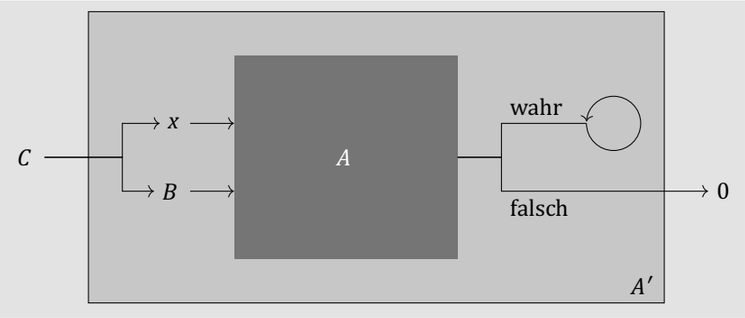

Abbildung 3.3 Der Algorithmus A' benutzt intern A.

Was passiert nun, wenn wir A' den eigenen Quellcode als Eingabe geben? Da es nur zwei Möglichkeiten gibt, können wir beide durchspielen:

- Angenommen, der Algorithmus A' gibt 0 aus, hält also auf der Eingabe A' an. Zu diesem Ergebnis muss auch die Blackbox A kommen, da sie ja angeblich das Halteproblem löst und somit *wahr* zurückgibt. In dem Fall geht A' jedoch in eine Endlosschleife und hält nicht an – ein Widerspruch!
- Gehen wir umgekehrt davon aus, dass A' nicht anhält. Das stellt nach Annahme auch die Blackbox fest und gibt *falsch* zurück. Nach Definition gibt A' also eine 0 aus und hält an – ebenfalls ein Widerspruch!

Nicht entscheidbar

Da die Konstruktion von A' so zulässig ist (den Beweis hierfür besprechen wir später), muss der Fehler in der Annahme liegen. Es kann also den hypothetischen Algorithmus A nicht geben, der das Halteproblem für alle möglichen Eingaben korrekt entscheiden würde. Wir sagen deshalb: Das Halteproblem ist *nicht entscheidbar*.

Auswirkungen des Halteproblems

Aufbauend auf diesem Ergebnis werden wir in Kapitel 10 beweisen, dass im Prinzip alle Aussagen über das Verhalten eines beliebigen Algorithmus nicht entscheidbar sind. Intuitiv ergibt sich diese Schlussfolgerung so:

1. Es ist nicht für beliebige Algorithmen entscheidbar, ob diese auf einer bestimmten Eingabe anhalten.
2. Wenn nicht entscheidbar ist, ob ein Algorithmus auf einer Eingabe anhält, kann auch nicht entschieden werden, ob er mit dem richtigen Ergebnis anhält.

3. Wenn dies noch nicht einmal für eine einzelne Eingabe entscheidbar ist, dann erst recht nicht für alle möglichen Eingaben.
4. Damit kann nicht entschieden werden, ob ein Algorithmus für alle Eingaben das korrekte Ergebnis liefert.

Für die praktische Softwareentwicklung ist dies ein deprimierendes Ergebnis: Da nicht entscheidbar ist, ob ein Algorithmus korrekt funktioniert, müssen wir immer befürchten, dass unsere geschriebenen Programme fehlerhaft sind. Übertragen auf andere Eigenschaften heißt dies zum Beispiel auch, dass wir nie vollständig davon überzeugt sein können, dass ein Softwaresystem keine Sicherheitslücken enthält. Die theoretische Informatik lehrt uns also für die praktische Softwareentwicklung und den Einsatz digitaler Systeme, dass man niemals blind auf deren Ergebnisse vertrauen sollte.

3.4 Kontrollfragen

1. Was ist ein Algorithmus?
2. Können alle mathematischen Funktionen von Algorithmen berechnet werden?
3. Warum würden wir gerne das Halteproblem für beliebige Algorithmen entscheiden können?

3.5 Antworten

1. Algorithmen wandeln mit Befehlen Eingaben in Ausgaben um (genaue Definition siehe Abschnitt 3.1).
2. Da es wesentlich weniger mögliche Algorithmen als mögliche Funktionen gibt, können nicht alle mathematischen Funktionen von Algorithmen berechnet werden.
3. Könnten wir dieses Problem (effizient) entscheiden, so könnten wir viele Softwarefehler erkennen und vermeiden; insbesondere Programme, die in Endlosschleifen festhängen.

Kapitel 12
Obere Schranken für Laufzeiten

Wie misst man geräteunabhängig die Laufzeit eines Algorithmus?
Wir führen dafür ein Maschinenmodell und eine Laufzeitnotation ein.

Für die Analyse von Laufzeiten von Algorithmen werden wir bestimmen, wie viele Rechenoperationen abhängig von der Größe der Eingabe benötigt werden. Die Eingabegröße ist dabei vereinfacht gesagt der Speicherplatzbedarf der Eingabe. Anstatt die Anzahl an Rechenoperationen exakt zu ermitteln, bestimmen wir Funktionen, die eine obere Schranke für diese Schrittanzahl angeben.

Laufzeitwachstum

Dabei interessieren wir uns primär für die Größenordnung des Laufzeitwachstums und ignorieren deshalb konstante Faktoren und Summanden in den Laufzeitfunktionen. Grob gesprochen ist es für die Praxis zwar nicht irrelevant, ob ein Algorithmus 10 Minuten oder 20 Minuten läuft (konstanter Faktor 2), dieser Unterschied kann aber im Zweifel schon durch die Wahl des genutzten Computers ausgeglichen werden. Wichtiger ist es jedoch, zu unterscheiden, ob ein Programm sein Ergebnis nach wenigen Minuten oder erst nach vielen Jahren ausgibt – unabhängig von der genutzten Hardware. Mit der *Landau-Notation* zeige ich Ihnen das Standardwerkzeug, um die Größenordnung einer Laufzeit formal anzugeben.

Rechenschritt

Zuvor müssen wir jedoch klären, was genau mit einem Rechenschritt gemeint ist. Ist dies (wie in Abschnitt 9.4) ein Befehl eines While-Programms? Oder ein Schritt einer Turingmaschine? Die genaue Definition hat einen großen Einfluss auf die darauf aufbauenden Analysen, und je tiefer man in die Komplexitätstheorie einsteigt, desto schwieriger wird die Antwort. Wir begnügen uns in diesem Buch größtenteils mit dem sogenannten *uniformen Kostenmaß auf RAMs* (Random Access Machines), mit dem wir für die meisten Probleme recht gut das Laufzeitverhalten auf echten Computern modellieren können. Grob gesprochen, benötigt in diesem Modell jede Grundoperation auf einer Zahl einen Berechnungsschritt.

Hintergrund: Unpräzise Laufzeitanalysen?

Wir nutzen Schranken statt exakten Werten, ignorieren Faktoren und sind uns noch nicht einmal sicher, was ein einzelner Berechnungsschritt eigentlich genau ist. Bei all dieser Unschärfe in der Analyse mag es fraglich erscheinen, ob die Resultate der Laufzeituntersuchungen für die Praxis überhaupt noch gelten.

Tatsächlich sind all diese Ungenauigkeiten für gewöhnlich kleiner als die Leistungsunterschiede zwischen verschiedenen Computern. Dort bestimmt neben hochgradig unterschiedlichen Taktraten der Prozessoren nämlich auch der auf der CPU verfügbare Befehlssatz oder der Umfang und die Geschwindigkeit des verfügbaren Speichers massiv, wie schnell ein Programm am Ende wirklich ist. Eine grobe theoretische Abschätzung des Laufzeitverhaltens eines Algorithmus gibt Ihnen aber zumindest die Sicherheit, dass für größere Eingaben als Ihre Testdaten keine Überraschungen auftreten.

12.1 Das Maschinenmodell

Zuallererst müssen wir klären, was die Einheit unserer Zeitmessungen sein soll – was ist also ein Berechnungsschritt? In Abschnitt 9.4 haben wir dies anhand der Syntax von While-Programmen getan. Zwar hilft diese Definition, Fragen der Berechenbarkeitstheorie exakt zu beantworten, die Analysen würden sich jedoch nur sehr eingeschränkt mit Messungen in der Praxis decken. Zudem sind While-Programme zur Notation von komplexeren Algorithmen ohnehin nicht sonderlich gut geeignet.

Technisch perfekt wäre es, Algorithmen in einer echten Programmiersprache wie C zu notieren und dann die Anzahl der Prozessorzyklen zu berechnen, die ein Computer bei der Ausführung des kompilierten Programms benötigt. Angesichts von unzähligen Arten von Prozessoren und von hochkomplexen Optimierungen in der Prozessorentwicklung und im Compilerbau ist dies jedoch nicht realistisch umsetzbar.

Notation von Algorithmen

Stattdessen werden wir Algorithmen in Pseudocode oder Prosa verfassen; auch Ablaufdiagramme und Funktionsschreibweisen sind denkbar und generell alle Notationsmöglichkeiten, die drei Eigenschaften erfüllen:

1. Die Notation muss geeignet sein, den Algorithmus für Menschen verständlich darzustellen. Es geht darum, dass Ihre Leser*innen den Algorithmus verstehen, nicht um das Aufschreiben eines kompilierbaren Programms.
2. Die Darstellung muss so präzise sein, dass unmissverständlich klar ist, welche Operationen in welcher Reihenfolge ausgeführt werden sollen. Nur dann kann später die Korrektheit des Algorithmus bewiesen werden.
3. Die Operationen müssen simpel genug sein, dass sich der dahinter verborgene Rechenaufwand analysieren lässt.

Wie immer ist es für alle drei Punkte möglich und sinnvoll, ein Problem und dessen Lösung in Teile zu zerlegen und die Teile getrennt voneinander zu beschreiben und zu analysieren. Wir beginnen daher mit simplen Programmen und Datenstrukturen, die wir später für komplexere Algorithmen als Grundbausteine voraussetzen.

RAM-Modell

Die Laufzeit eines so notierten Algorithmus ergibt sich dann aus der Anzahl von *Berechnungsschritten* auf dem *RAM*-Maschinenmodell, das einen stark vereinfachten Computer abbildet. Der Speicher der Maschine setzt sich aus vielen Zellen zusammen, auf denen einfache Operationen ausgeführt werden können:

- Jeder Schreib- oder Lesezugriff auf eine Speicherzelle kostet einen Schritt.
- Jeder Vergleich zweier Speicherzellen (Schleifenbedingungen und Verzweigungen) kostet einen Schritt.
- Jede elementare Rechenoperation auf zwei Speicherzellen (Addition, Subtraktion, Multiplikation, Division, Modulo, logisches Und/Oder/Nicht, bitweise Operationen) kostet einen Schritt.

Adressierter Speicher

Im Gegensatz zu einer Turingmaschine, die ihren Lese-/Schreibkopf immer erst auf dem Band zur gewünschten Speicherzelle bewegen muss, erlauben wir den Zugriff auf beliebige Speicherzellen in einem einzelnen Schritt. Daher heißt diese Maschine auch *RAM* (engl. *Random Access Machine*). Die Speicherzellen sind durchnummeriert, damit jede Zelle über ihre eindeutige *Adresse* referenziert werden kann. Auf diese Weise können wir nicht nur mit Daten, sondern auch mit Speicheradressen rechnen. Dies wird insbesondere im Umgang mit Datenstrukturen wie Arrays und Listen relevant.

Wortbreite

Welche Daten passen nun in eine Speicherzelle? Wir gehen grundsätzlich von binär codierten Daten aus. Meist nimmt man an, dass die Maschine eine sogenannte *Wortbreite* hat: eine Anzahl an Bits, die in einer Zelle gespeichert sind. Ob die Bits dann als Zahl, als Wahrheitswert oder als beliebiger anderer codierter Datenwert verstanden werden, ist der Semantik des

Programms überlassen. In der Praxis ist das genauso, dort haben heutige Computer zumeist eine Wortbreite von 64 Bit. Eine Zelle kann also eine natürliche Zahl zwischen 0 und $2^{64} - 1$ speichern.

Word-RAM

Dieses *Word-RAM*-Modell ist in der Wissenschaft eines der verbreitetsten und realistischsten Modelle für Komplexitätsanalysen. Ein Nachteil an diesem Modell ist, dass man ununterbrochen die Größe der verarbeiteten Zahlen im Blick behalten muss: Sobald eine Zahl zu groß wird, um in einer einzigen Zelle Platz zu finden, muss man für alle oben genannten Operationen neu bestimmen, wie viele Schritte für diese in der Analyse gezählt werden müssen.

Hintergrund: Große Zahlen in Theorie und Praxis

Der komplizierte Umgang mit großen Zahlen ist eigentlich sehr realistisch: Auch beim Programmieren in der Praxis müssen Sie auf die Größe der verwendeten Zahlen achtgeben. Werden Zahlen größer als der verwendete Datentyp, so leidet in den meisten Programmiersprachen jedoch nicht die Laufzeit (wie bei unserem Modell in der Theorie), sondern die Korrektheit des Algorithmus, weil bei einem Überlauf der tatsächlich vorliegende Zahlenwert nicht mehr das gewünschte Ergebnis einer Berechnung ist.

Uniformes Kostenmaß

Wir werden stattdessen im Regelfall das sogenannte *uniforme Kostenmaß* betrachten und die Größe der Zahlen nicht weiter beachten. Umgekehrt formuliert, gehen wir also meist davon aus, dass unsere Zahlen jeweils in eine Speicherzelle passen.

Logarithmisches Kostenmaß

Sollte der Algorithmus sehr große Zahlen (im Verhältnis zur Größe der Eingabe) verarbeiten, muss man bei der Analyse genauer darauf achten, wie groß diese werden. Eine Möglichkeit dafür hält das *logarithmische Kostenmaß* bereit. Dieses betrachtet für jeden Wert genau die Anzahl an Bits, die für die Codierung des Werts benötigt werden; der Name kommt übrigens daher, dass eine Zahl $n \in \mathbb{N}$ binär in etwa $\log(n)$ Bits codiert wird. In diesem Maß wird jedes gelesene, geschriebene oder anderweitig verarbeitete Bit als einzelner Schritt berechnet.

Den Unterschied der beiden Maße betrachten wir an einem Algorithmus, der aus der Eingabe $n \in \mathbb{N}$ die Ausgabe 2^{2^n} berechnet, indem er zu Beginn eine Variable x auf den Wert 2 initialisiert, dann insgesamt n-mal $x = x \cdot x$ berechnet und abschließend x ausgibt. Wir analysieren hier nur die Anzahl der Rechenschritte für die Multiplikationen:

- Im uniformen Kostenmaß werden die n Multiplikationen mit je einem Berechnungsschritt bemessen. Der Algorithmus benötigt also n Schritte.
- Im logarithmischen Kostenmaß müssen wir den Speicherbedarf von x betrachten. Allein für die letzte Multiplikation müssen $(2^n + 1)$ Bits geschrieben werden, für alle zuvor durchgeführten Multiplikationen zusammen in etwa noch einmal so viele. Bei einer Eingabe n benötigt der Algorithmus also ungefähr 2^{n+1} Schritte.

Die Analyseergebnisse unterscheiden sich also immens: Die eine Schrittzahl wächst *linear*, die andere *exponentiell* im Wert der Eingabe n.

In diesem Buch werden wir vorwiegend Algorithmen analysieren, deren verarbeitete Zahlen nicht groß genug für solch dramatische Unterschiede sind. Wir machen uns deshalb das Leben mit dem uniformen Kostenmaß etwas einfacher.

12.2 Die Laufzeit eines Algorithmus

$\textsc{Time}_A$

Im Regelfall hängt die exakte Schrittanzahl eines Algorithmus von der genauen Eingabe ab. Für unsere binär codierten Eingaben können wir die Laufzeit eines Algorithmus A durch eine Funktion $\textsc{Time}_A : \{0,1\}^* \to \mathbb{N}^+$ beschreiben:

$$\forall w \in \{0,1\}^* : \textsc{Time}_A(w) := \text{Anz. Berechnungsschritte von } A \text{ auf Eingabe } w$$

Wir gehen davon aus, dass jeder Algorithmus zumindest einen Schritt für die Rückgabe des Ergebnisses benötigt, daher ist die Zielmenge von $\textsc{Time}_A$ die Menge der natürlichen Zahlen *ohne* die Null.

Hält ein Algorithmus auf einer Eingabe nicht an, so ist die Funktion für diese Eingabe undefiniert. In der Praxis wollen wir aber eigentlich nur mit Algorithmen arbeiten, die auf allen Eingaben nach endlicher Zeit eine Ausgabe liefern. Wir werden im Folgenden daher ausschließlich Laufzeitanalysen für Algorithmen durchführen, die totale Funktionen berechnen und auf jeder Eingabe anhalten.

Worst-Case-Laufzeit

Die Laufzeit zweier Algorithmen für dasselbe Problem lässt sich sehr schwer vergleichen, wenn man für jede Eingabe einen eigenen Wert gegenüberstellen müsste. Stattdessen betrachten wir in der Komplexitätstheorie meist abhängig von der Eingabegröße den sogenannten *Worst-Case*, also die längste Laufzeit, die der Algorithmus auf Eingaben einer bestimmten

Größe hat. Wir definieren dafür zusätzlich die Funktion $\textsc{WorstCaseTime}_A : \mathbb{N} \to \mathbb{N}^+$ als

$$\forall n \in \mathbb{N}: \textsc{WorstCaseTime}_A(n) := \max\{ \textsc{Time}_A(w) \mid w \in \{0,1\}^n \}.$$

Best- und Average-Case-Laufzeit

Neben dem Worst-Case kann man auch den *Best-Case*, also die kürzeste Laufzeit, und den *Average-Case*, also die durchschnittliche Laufzeit, auf Eingaben bestimmter Größe betrachten. Ohne genauere Angabe ist immer der Worst-Case gemeint: Standardmäßig sind wir daran interessiert, wie lange wir schlimmstenfalls auf das Berechnungsergebnis warten müssen.

12.3 Die Größe einer Eingabe

Intuitive Eingabegrößen

Formal gesehen, betrachten wir als Eingabegröße immer die Länge (also die Anzahl der Bits) der Codierung der Eingabe. Oft ist das aber kein besonders praktisches oder intuitives Maß. Wir wollen Laufzeiten stattdessen lieber an einer greifbaren Größe festmachen. Geläufig sind für unterschiedliche Datenstrukturen insbesondere folgende Größen:

- der Wert einer Zahl
- die Länge n einer verketteten Liste bzw. eines Arrays
- die Anzahl der Knoten n und Kanten m eines Graphen
- die Anzahl der Zeilen m und Spalten n einer Matrix

Eingaben von Echtwelt-Problemen umfassen oftmals mehrere Datenstrukturen. Die Laufzeitfunktion eines Lösungsalgorithmus kann daher auch auf mehreren Variablen definiert sein, die die Größe der einzelnen Datenstrukturen angeben.

Achten Sie bei Algorithmen genau drauf, im Verhältnis zu was die Laufzeit angegeben ist! Der Wert einer Zahl ist exponentiell größer als die Länge der Codierung des Wertes. Verwenden zwei Laufzeitangaben unterschiedliche Basisgrößen, können Sie deren Laufzeit und Effizienz ansonsten nicht objektiv bewerten oder vergleichen.

12.4 Die Landau-Notation

Nehmen wir an, Sie analysieren zwei Algorithmen A und B auf ihre Laufzeit für verschiedene Eingabegrößen und erhalten Tabelle 12.1 als Resultat.

n	1	2	3	4	5	6	7	8	9
$\textsc{Time}_A(n)$	33	36	40	45	52	61	72	84	97
$\textsc{Time}_B(n)$	13	14	17	25	41	71	121	198	309

Tabelle 12.1 Fiktive Schrittanzahlen für zwei Algorithmen A, B und verschiedene Eingabegrößen n

Welchen Algorithmus sollten Sie wählen? Basierend auf den ersten paar Werten, scheint Algorithmus B schneller zu sein, ab der Eingabegröße 6 übersteigt seine Laufzeit jedoch die von Algorithmus A. Ohne zu wissen, wie sich die Zahlen weiterentwickeln, ist es unmöglich zu entscheiden. Es könnte sowohl sein, dass Algorithmus B bei größeren Eingaben wieder schneller ist als A, als auch, dass er massiv langsamer arbeitet.

Wachstum der Laufzeitfunktion

Generell sind wir in der theoretischen Analyse nicht so sehr an exakten Schrittzahlen für einzelne Größen interessiert, sondern wollen vielmehr wissen, wie sich die Laufzeit für immer größer werdende Eingaben entwickelt. Wächst der Rechenaufwand *linear*, *quadratisch* oder *kubisch* in der Eingabegröße? Oder wächst er sogar *exponentiell* oder noch schneller? Haben wir einen sehr effizienten Algorithmus vorliegen, dessen Laufzeit nur *logarithmisch* wächst oder sogar unabhängig von der Eingabegröße *konstant* ist?

Landau-Notation

Um diese Größenordnungen vergleichbar zu machen, verwenden wir bei Laufzeitanalysen die *Landau-Notation* und geben mit einfachen Funktionen eine möglichst enge obere Schranke für die Schrittanzahl eines Algorithmus an. Ich gebe Ihnen zunächst die fünf von uns verwendeten Definitionen an die Hand und erkläre dann, was es mit den einzelnen Bestandteilen auf sich hat.

Landau-Symbole

Seien $f, g: \mathbb{N} \to \mathbb{R}^+$ zwei Funktionen. Dann notieren wir:

- Die Funktion f wächst asymptotisch maximal so schnell wie g:
  $f \in O(g) \quad \Leftrightarrow \quad \exists c \in \mathbb{R}^+ \; \exists n_0 \in \mathbb{N} \; \forall n \geq n_0 : f(n) \leq c \cdot g(n)$
- Die Funktion f wächst asymptotisch mindestens so schnell wie g:
  $f \in \Omega(g) \quad \Leftrightarrow \quad g \in O(f)$
- Die Funktion f wächst asymptotisch genauso so schnell wie g:
  $f \in \Theta(g) \quad \Leftrightarrow \quad f \in O(g) \wedge g \in O(f)$

- Die Funktion f wächst asymptotisch echt langsamer als g:
  $f \in o(g) \quad \Leftrightarrow \quad \forall c \in \mathbb{R}^+ \, \exists n_0 \in \mathbb{N} \, \forall n \geq n_0 \colon f(n) < c \cdot g(n)$
- Die Funktion f wächst asymptotisch echt schneller als g:
  $f \in \omega(g) \quad \Leftrightarrow \quad g \in o(f)$

Asymptotisches Wachstum

Mit *asymptotischem Wachstum* beschreiben wir, dass uns nur das Verhalten der Funktionen für sehr große Werte (konkret: ab n_0) interessiert und dass wir durch den zusätzlichen Faktor c konstante Faktoren ignorieren können.

Ein paar Beispiele:

- Es gilt $n \in O(n^2)$, denn lineares Wachstum ist langsamer als quadratisches.
- Es gilt $n \in \Omega(\sqrt{n})$, da die Wurzelfunktion langsamer als linear wächst.
- Es gilt $3n^3 \in \Theta(n^3)$, da der konstante Faktor für das asymptotische Wachstum irrelevant ist.
- Es gilt $n^3 + 10 \in o(2^n)$. Auch, wenn der konstante Summand die kubische Funktion auf kleinen Eingaben größer werden lässt als die exponentielle, wächst 2^n dennoch deutlich schneller. Gut erkennbar ist das ab $n_0 = 10$.
- Es gilt $n^{4.1} \in \omega(n^4 + n^3 + n^2 + n + 100)$. Das Polynom mit dem echt größeren Grad wächst asymptotisch echt schneller.

Limes-Definition für Landau-Symbole

Generell ist beim Vergleich des asymptotischen Wachstums zweier Funktionen jeweils nur der Summand relevant, der am schnellsten wächst. Falls Sie mit der Grenzwertanalyse von Funktionen vertraut sind, kommt Ihnen das möglicherweise bekannt vor. Tatsächlich gilt für zwei Funktionen $f, g \colon \mathbb{N} \to \mathbb{R}^+$ der folgende Zusammenhang:

- Ist der Grenzwert $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ kleiner als unendlich, so gilt $f \in O(g)$. Ist er zusätzlich echt größer 0, so gilt $f \in \Theta(g)$.
- Gilt dagegen $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$, so folgt daraus $f \in \omega(g)$. Umgekehrt folgt aus $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, dass $f \in o(g)$ ist.

Hintergrund: Schlampige Konventionen in der Landau-Notation

Formal gesehen, beschreiben die Landau-Symbole Mengen von Funktionen. Daher schreiben wir auch $f \in O(g)$, wenn die Funktion f in der Menge der Funktionen ist, die maximal so schnell wie die Funktion g wachsen. Damit können wir auch Mengen vergleichen und zum Beispiel $O(f) \subseteq O(g)$ schreiben.

Leider ist es in der Literatur weit verbreitet, stattdessen die Notation $f = O(g)$ zu verwenden. Dabei darf das Gleichheitszeichen nicht missverstanden werden: Es beschreibt hier weder eine symmetrische Relation noch wird impliziert, dass f dasselbe asymptotische Wachstum wie g hätte. Bleiben Sie also am besten bei der korrekten Mengennotation, die weniger anfällig für Lesefehler ist.

Tatsächlich ist auch unsere Notation nicht vollständig präzise: Genau genommen müssten wir zur Aussage $n^2 \in O(n^3)$ den Hinweis ergänzen, dass wir das Wachstum der Funktionen in der Variablen n, also für $n \to \infty$ betrachten. Aus dem Kontext ist es aber fast immer offensichtlich, in welcher Variable das Wachstum untersucht wird; deshalb lassen wir die zusätzliche Angabe weg.

Natürlich müssen Sie stets beweisen, warum sich das Wachstum von Funktionen zueinander so verhält, wie Sie es behaupten. Dafür können Sie sowohl die Quantor-Definitionen verwenden als auch mit Grenzwerten argumentieren. Zumeist werden Ihnen in der Laufzeitanalyse von Algorithmen aber ohnehin nur eine Handvoll verschiedene Funktionen begegnen, deren Einordnung Ihnen dann bereits bekannt ist.

Landau-Symbole für Laufzeiten

In der Analyse von Laufzeiten von Algorithmen gibt man die obere Schranke an die Worst-Case-Laufzeit in der Regel mit dem großen O an. Möchte man betonen, dass die durchgeführte Analyse exakt ist und die angegebene Schranke zugleich auch eine untere Schranke an die Laufzeitkomplexität darstellt, kommt die stärkere Aussage mit Θ zum Einsatz.

12.5 Aufgaben

12.5.1 Kontrollfragen

1. Worauf müssen Sie bei der Notation eines Algorithmus achten?
2. Wofür berechnen wir im uniformen Kostenmaß einen Rechenschritt?
3. Im Verhältnis wozu geben wir Laufzeiten an?
4. Was ist der Unterschied zwischen Worst-Case und Best-Case?
5. Wofür stehen die Landau-Symbole O, Ω, Θ, o, ω?

12.5.2 Typische Laufzeiten

Ordnen Sie die folgenden Funktionen nach ihrem asymptotischen Wachstum und begründen Sie Ihre Ordnung knapp.

- n^2
- n^n
- $n \log(n)$
- $n!$
- n^3
- $\sqrt{n}$
- n
- $\log(n)$
- 2^n

12.5.3 Landau-Notation

Beweisen oder widerlegen Sie folgende Aussagen:

1. Für zwei Funktionen $f, g\colon \mathbb{N} \to \mathbb{R}^+$ gilt stets $f \in O(g)$ oder $f \in \Omega(g)$.
2. Es gilt, für alle $a, b > 1$, der Zusammenhang $\log_a(n) \in \Theta(\log_b(n))$.
3. Es gilt $\log(n!) \in \Theta(n \log(n))$. (Tipp: Schätzen Sie die Faktoren der Fakultät großzügig nach oben bzw. unten ab.)

12.6 Lösungen

12.6.1 Kontrollfragen

1. Ihr Algorithmus muss verständlich, präzise und analysierbar notiert sein.
2. Für jeden Speicherzugriff auf einen Wert, jeden durchgeführten Vergleich zweier Werte und jede einfache Rechenoperation wird ein Berechnungsschritt veranschlagt.
3. Die Laufzeit wird im Verhältnis zur Eingabegröße angegeben. Formal ist dies die Codierungslänge der Eingabe; für gewöhnlich verwenden wir aber intuitivere Größenangaben wie die Anzahl der Zahlen in der Eingabe.
4. Eine Worst-Case-Analyse betrachtet für jede Eingabelänge die längste Laufzeit, die der Algorithmus für eine Eingabe dieser Länge haben kann. Beim Analysieren des Best-Case wird stattdessen für jede Eingabelänge die kürzeste auftretende Laufzeit bestimmt.
5. Das große O in $f \in O(g)$ gibt an, dass f asymptotisch maximal so schnell wächst wie g. Beim großen Omega in $f \in \Omega(g)$ wächst f asymptotisch mindestens so schnell wie g. Das große Theta (Θ) gibt an, dass die Funktionen im Unendlichen gleich schnell wachsen. Das kleine o in $f \in o(g)$ zeigt an, dass f asymptotisch echt langsamer wächst als g, und beim kleinen Omega (ω) ist es umgekehrt.

12.6.2 Typische Laufzeiten

Subpolynomiell

Der Logarithmus $\log(n)$ wächst *subpolynomiell*, also langsamer als jedes Polynom. Wir beweisen dies, indem wir den Logarithmus umschreiben:

$$\log(n) = n^{\log(\log(n))/\log(n)}$$

Da der Exponent monoton fällt, gilt für alle Exponenten $a \in \mathbb{R}^+$, dass $\log(n) \in O(n^a)$ ist.

Polynomiell

Anschließend kommen in aufsteigender Reihenfolge $\sqrt{n} = n^{0,5}$, n, $n \log(n)$, n^2 und n^3: Alle wachsen *polynomiell*, wobei die Ordnung der reinen Polynome untereinander klar sein sollte. Die Einsortierung von $n \log(n)$ ergibt sich erneut daraus, dass $\log(n)$ subpolynomiell wächst.

Superpolynomiell

Das größte Wachstum haben, erneut in aufsteigender Reihenfolge, die *superpolynomiellen* Funktionen 2^n, $n!$ und n^n. Deren Reihenfolge untereinander ergibt sich daraus, dass sie jeweils dieselbe Anzahl Faktoren multiplizieren, die Faktoren selbst aber unterschiedlich groß sind: im ersten Fall sind alle Faktoren konstant 2, bei der Fakultät werden alle Zahlen von 1 bis n multipliziert und im letzten Fall sind alle Faktoren n.

12.6.3 Landau-Notation

1. Die Aussage gilt nicht. Wir widerlegen sie mit einem Gegenbeispiel:

   Seien $f, g\colon \mathbb{N} \to \mathbb{R}^+$ folgendermaßen für alle $n \in \mathbb{N}$ definiert:

   $$f(n) = \begin{cases} n, & \text{falls } n \text{ gerade;} \\ 1, & \text{sonst.} \end{cases} \qquad g(n) = \begin{cases} 1, & \text{falls } n \text{ gerade;} \\ n, & \text{sonst.} \end{cases}$$

   Angenommen, es gilt $f \in O(g)$. Seien also $c \in \mathbb{R}^+$ und $n_0 \in \mathbb{N}$ so, dass für alle $n \geq n_0$ gilt: $f(n) \leq c \cdot g(n)$. Für $n' = 2(n_0 + \lceil c \rceil)$ gilt jedoch $f(n') = n' > c = g(n')$. Das ist ein Widerspruch. Also ist $f \notin O(g)$. Analog gilt $g \notin O(f)$ mit $n' = 2(n_0 + \lceil c \rceil) + 1$ und $f(n') < g(n')$. Das asymptotische Wachstum der Funktionen ist also unvergleichbar.
2. Das ist korrekt, da nach den Logarithmengesetzen $\log_a(n) = \frac{\log_b(n)}{\log_b(a)}$ gilt und für konstante a und b auch $\frac{1}{\log_b(a)}$ eine Konstante ist. Aus diesem Grund lässt man konstante Basen des Logarithmus in Laufzeitanalysen für gewöhnlich weg, da sie keinen Unterschied machen.
3. Die Aussage ist korrekt. Wir schätzen zunächst nach oben ab; es gilt für alle $n \in \mathbb{N}^+$:

   $$\log(n!) \leq \log(n^n) = n \log(n)$$

Es folgt direkt $\log(n!) \in O(n \log(n))$. Im nächsten Schritt schätzen wir großzügig nach unten ab. Für alle natürlichen Zahlen $n \geq 4$ gilt:

$$\log(n!) \geq \log((n/2)^{n/2}) = \frac{n}{2} \log(n/2) = \frac{n}{2} \log(n) - \frac{n}{2} \log(2) \geq \frac{1}{4} n \log(n)$$

Es folgt $\log(n!) \in \Omega(n \log(n))$ und somit insgesamt $\log(n!) \in \Theta(n \log(n))$.