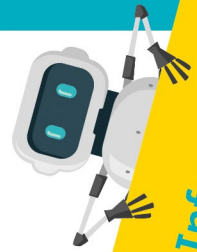


Stefan Neubert



**Informatik
verstehen**

GRUNKURS

Theoretische Informatik



- + Intuitive Zugänge, praktische Anwendungen, formale Methoden
- + Kreativ knobeln, Probleme lösen und Beweise finden
- + Ideal zum Selbststudium und als Vorlesungsbegleiter



Rheinwerk
Computing

Kapitel 1

Einführung

Was ist »theoretische Informatik« und wofür braucht man Theorie überhaupt, wenn man doch eigentlich praktisch arbeiten möchte?

Kaum jemand kann sich heutzutage noch der Allgegenwärtigkeit von digitalen Systemen entziehen – und immer weniger möchten das überhaupt, weil wir uns eine Welt ohne Smartphone, Computer und Internet kaum noch vorstellen können. Da Sie dieses Buch in den Händen halten, gehören Sie vermutlich zu den Menschen, die von der Informatik nicht nur als Nutzer in den Bann gezogen wurden: Vermutlich sind Sie genau wie ich selbst begeistert von den unendlichen Möglichkeiten und kleinen Erfindungen, die nur ein kurzes Computerprogramm entfernt liegen.

Zum Alltag von Informatikern gehört zwar auch, stundenlang im Programmcode nach winzigen Fehlern mit großen Auswirkungen zu suchen – aber die wunderbaren Erfolgsmomente, wenn eine eigene Idee zum ersten Mal auf dem Bildschirm funktionstüchtig zum Leben erwacht, machen das allemal wett. Leider können sich viele Studierende nicht im gleichen Maße für die theoretische Informatik begeistern, wie sie das für die praktische Seite tun:

»Ich möchte Software entwickeln, keine Beweise schreiben.«

Solche und ähnliche Aussagen höre ich oft, wenn die Sinnhaftigkeit der Ausbildung in den theoretischen Grundlagen der Informatik infrage gestellt wird. Rückblickend erkennen die meisten dann zwar, dass ihnen genau diese theoretischen Grundlagen eine große Hilfe bei der Entwicklung von Software sind. Die Lehrveranstaltung »Theoretische Informatik« ist dennoch für viele ein unbeliebtes bis zuweilen gefürchtetes Fach. Die Gründe hierfür sind vielfältig. Oft hängen sie aber damit zusammen, dass man zum Meistern der theoretischen Informatik viele Kompetenzen ausbilden und trainieren muss.

Theorie vs. Praxis

1.1 Kompetenzen für die theoretische Arbeit

Viele Lehrveranstaltungen konzentrieren sich in erster Linie auf die Sachinhalte des Fachs. In diesem Buch möchte ich neben diesen reinen Sachinhalten einen Schwerpunkt auf die Vermittlung und das Training der Kompetenzen legen, die zum theoretischen Arbeiten notwendig sind. Ganz unabhängig von den Erkenntnissen der theoretischen Informatik lohnt es sich ohnehin, die folgenden Fähigkeiten zu fördern:

1.1.1 Abstraktionsvermögen

Modelle analysieren

Auch wenn die Informatik am Ende das Ziel hat, einen Mehrwert für die echte Welt zu schaffen, so arbeiten wir doch immer nur mit *Modellen* der Wirklichkeit. Ein Modell ist ein abstraktes Abbild der Wirklichkeit, das relevante Eigenschaften abbildet und irrelevante weglässt. Die Entscheidung, was relevant ist und was nicht, ist alles andere als trivial und hängt zudem immer vom Kontext ab – wir können also für unterschiedliche Problemstellungen nicht zwingend dasselbe Modell verwenden. Hat man jedoch erst einmal den Dreh raus, hilft abstraktes Denken auch ganz unabhängig von der Informatik beim Lösen vieler kleiner und großer Alltagsprobleme.

1.1.2 Präzises Arbeiten

Aussagen beweisen

Zwei der wichtigsten Themen aus der mathematischen Grundausbildung, die für die theoretische Informatik benötigt werden, sind Logik und Beweistechnik, denn sie ermöglichen uns präzises Argumentieren. Ein einziger Fehler in einer Argumentationskette – und sei er auch noch so klein – kann die gesamte Beweisführung in sich zusammenfallen lassen. Da hilft es auch nichts, wenn in diesem speziellen Fall eine fehlerfreie Argumentation zum selben Ergebnis gekommen wäre.

Kommt Ihnen das bekannt vor? Richtig: Genauso unbarmherzig, wie die mathematische Logik mit Denk- oder Notationsfehlern umgeht, zeigt sich ein Compiler angesichts von Tippfehlern im Code. Eigentlich ist präzises Arbeiten also ein Heimspiel für Informatiker*innen, für viele stellt die Schönheit einer eleganten Beweisführung jedoch nicht denselben Ausgleich für die harte Arbeit dar, wie es ein funktionstüchtiges Programm vermag.

Vielleicht kann ich Sie in diesem Buch davon überzeugen, dass die theoretische Arbeit Sie auch für die praktische Softwareentwicklung voranbrin-

gen wird. Ich möchte Ihnen zeigen, dass die theoretischen Themen große Relevanz für die praktische Informatik haben. Zudem werde ich Ihnen dabei helfen, ähnlich wie beim Debuggen eines Programms, Fehler in Argumentationsketten zu erkennen und zu vermeiden.

1.1.3 Frustrationstoleranz und Kreativität

Theoretische Fragestellungen und deren Lösungen lassen sich oft in wenigen Zeilen vollständig formulieren. Die kompakten Notationen verstecken dabei, dass der Weg von der Fragestellung zur Lösung alles andere als geradlinig und kurz ist. Im Gegenteil: Bei der Lösung einer spannenden Aufgabe gehört es dazu, einige Umwege und Sackgassen zu erkunden, bevor man die tatsächliche Lösung entdecken kann. Die abschließende schriftliche Darstellung entspricht daher meist nicht dem ursprünglichen Lösungsweg.

Es ist wichtig, dabei den Wert von gescheiterten Versuchen zu erkennen und aus allen erforschten Umwegen neues Wissen herauszuziehen. Ganz genau zu verstehen, *warum* eine Lösungsidee nicht funktioniert hat, ist oftmals der entscheidende Schritt, um eine funktionstüchtige Lösung zu entwickeln. Zusätzlich zu mathematisch präzise Arbeiten ist hierfür nun kreatives Denken und Knobeln gefordert – und eine hohe Frustrationstoleranz.

Aus diesem Grund zeige ich Ihnen in diesem Buch für einige Fragestellungen nicht nur die Lösung, sondern auch den umständlichen Weg zu dieser Lösung. So erfahren Sie direkt an Beispielen, wie Sie sich Schritt für Schritt der Lösung nähern und von Sackgassen den korrekten Weg zeigen lassen können.

1.1.4 Kommunikationsfähigkeit

Ist die Lösung erst einmal gefunden, so ist die Arbeit aber noch lange nicht getan. Neben der eigenen Erkenntnis ist eine der wichtigsten Tätigkeiten jeder Wissenschaft, die Erkenntnis für andere aufzubereiten. Im Studienalltag dient der Lösungsaufschrieb oft leider nur der Leistungsbewertung. Tatsächlich geht es aber eigentlich darum, verständlich die eigenen Gedanken an eine andere Person zu übermitteln. Ob man sich dabei mathematischer Formeln, erklärender Texte, anschaulicher Grafiken oder eines anderen Mediums bedient, ist unerheblich – am Ende muss die Sprache von Beweisen genauso erlernt und geübt werden wie jede andere Sprache auch.

Lösungen finden

Lösungen kommunizieren

Auch diese Sprache hat eine Art Dialekte: Über die Verwendung vieler Symbole und Formulierungen herrscht einigermaßen Einigkeit. Manche Notationen bedeuten jedoch in verschiedenen Teildisziplinen der theoretischen Informatik etwas ganz Unterschiedliches und spätestens, wenn es um den Notationsstil geht, werden Sie zu jeder Empfehlung auch eine Person finden können, die Ihnen genau davon abrät.

Die Notation in diesem Buch folgt selbstverständlich meinen persönlichen Vorlieben, die ich mir wiederum von anderen Wissenschaftler*innen abgeschaut habe. Ich versuche es Ihnen so leicht wie möglich zu machen, die Inhalte auch auf andere Notationsstile zu übertragen.

1.2 Themen der theoretischen Informatik

Die Informatik als Gesamtes beschäftigt sich mit der automatisierbaren Verarbeitung von Informationen aller Art und gliedert sich in drei Bereiche. Während in der technischen und der praktischen Informatik die Realisierung von Maschinen beziehungsweise von Anwendungen für solche Maschinen im Fokus stehen, beschäftigt sich die theoretische Informatik mit den grundlegenden Rahmenbedingungen für diese Informationsverarbeitung. Ich gruppiere in diesem Buch die Grundlagenthemen der theoretischen Informatik grob in drei Gebiete: Berechenbarkeitstheorie, Algorithmik und Komplexitätstheorie.

1.2.1 Berechenbarkeitstheorie

Eine der wichtigsten Fragen, die die theoretische Informatik beantworten möchte, lautet: Für welche Problemstellungen lassen sich algorithmisch Lösungen berechnen – und für welche nicht? Um diese Frage anzugehen, muss zuvor geklärt werden, was *berechnen* überhaupt bedeutet. Wir werden sehen, dass es für die Mächtigkeit eines Berechnungsmodells unerheblich ist, ob dieses auf Zahlen, Texten oder ganz anderen Daten arbeiten kann, und dass es ebenso egal ist, ob eine komplizierte Ausgabe produziert wird oder das Modell nur zwischen »ja« und »nein« unterscheiden kann.

Berechnungs-
modelle

Es gibt unzählige solcher *Berechnungsmodelle*, von denen bei Weitem nicht alle in diesem Buch Platz finden können. Prominente Vertreter, die wir uns in Teil I des Buches anschauen werden, sind *Formale Sprachen*, *Automaten* und *Grammatiken*, *Turingmaschinen* und *Loop/While-Sprachen*.

Mit diesen Modellen werden wir für sehr einfache Probleme Berechnungsvorschriften entwickeln und deren Korrektheit beweisen. Umgekehrt werden wir aber auch die Grenzen des Machbaren erkunden und zeigen, dass einige für die praktische Arbeit in der Softwareentwicklung sehr wichtige Fragestellungen leider algorithmisch unlösbar sind. Als Kernaussage fungiert hier die *Unentscheidbarkeit des Halteproblems*; diese Aussage werden Sie zudem mithilfe von *Reduktionen* auf andere Probleme zu übertragen lernen.

Grenzen der
Berechenbarkeit

1.2.2 Algorithmik

Die Algorithmik beschäftigt sich damit, für Probleme Lösungsverfahren zu entwickeln und zu analysieren. Im Gegensatz zur Berechenbarkeitstheorie interessieren wir uns in Teil II des Buches nicht mehr für einen beliebigen Lösungsalgorithmus, sondern für einen möglichst »guten« – wobei wir im Allgemeinen davon ausgehen, dass ein Algorithmus dann besonders gut ist, wenn er zu einer Eingabe möglichst *schnell* eine korrekte Ausgabe liefert.

Effiziente
Algorithmen

Die Einführung von vier Paradigmen – *Brute Force*, *Greedy*, *Divide and Conquer* sowie *Dynamische Programmierung* – wird Ihnen dabei helfen, für eine Vielzahl von Problemen schnell passende und effiziente Algorithmen zu konstruieren.

Entwicklungs-
paradigmen

Zu jedem Algorithmus gehört eine Analyse der *Laufzeit* und *Korrektheit*, damit wir Garantien für die Qualität des entwickelten Algorithmus bekommen. Für jedes Paradigma lernen Sie, solche Analysen durchzuführen. Mit der *Landau-Notation* werden Sie für die Analyse ein Maß für die Effizienz eines Algorithmus kennenlernen. Als Beweistechniken für die Korrektheit der Lösungsverfahren werden *Invarianten* und (*strukturelle*) *Induktion* zum Einsatz kommen.

Analysetechniken

1.2.3 Komplexitätstheorie

Die Analyse eines Algorithmus ermöglicht zwar Rückschlüsse auf die Effizienz dieses speziellen Verfahrens, lässt aber die Frage offen, ob es effizientere Algorithmen geben könnte. Im Teil III des Buches beschäftigen wir uns deshalb mit der Analyse der zugrunde liegenden Problemstellung: Gibt es Probleme, die inhärent schwierig zu lösen sind, für die wir also niemals einen effizienten Algorithmus finden können?

Schwierige
Probleme

Mithilfe von *unteren Laufzeitschranken* können wir für manche Probleme beweisen, dass wir bereits optimale Algorithmen gefunden haben. Für

P vs. NP

viele Probleme ist dies bislang jedoch niemandem gelungen. In der Komplexitätstheorie untersucht man unter anderem zwei bedeutende Gruppen von Problemen, P und NP genannt. Eine der wichtigsten offenen Fragen der Informatik lautet, ob diese Gruppen identisch sind, ob also die Probleme in NP genauso effizient lösbar sind wie die vermeintlich leichteren Probleme in P . Naheliegenderweise werden wir die Frage in diesem Buch nicht beantworten können. In Form von *Polynomialzeitreduktionen* werden Sie jedoch ein mächtiges Werkzeug kennenlernen, um Probleme anhand ihrer Komplexität zu vergleichen.

Im Ausblick werde ich Ihnen abschließend noch einen Einblick darin geben, wie man auch für sehr schwierige Probleme, die scheinbar keine effiziente Lösung zulassen, für die Praxis brauchbare Algorithmen entwickeln kann. Als Technik stelle ich Ihnen dafür die *Parametrisierte Analyse* vor.

1.3 Anleitung fürs Buch

Voraussetzungen

Dieses Buch richtet sich primär an Leser*innen, die mit den Grundbegriffen der Informatik vertraut sind und sich nun mit der theoretischen Seite des Fachgebiets beschäftigen möchten. Sie sollten daher bereits wissen, was ein *Algorithmus* ist und einfachen (*Pseudo*-)Code schreiben und lesen können. Typische Datenstrukturen (*Arrays*, *Listen* und *Graphen*) sollten Ihnen ebenso geläufig sein wie die wichtigsten Standardalgorithmen der Informatik für das *Sortieren* und *Suchen*. Für eine Einführung oder Auffrischung dieser Themen empfehle ich Ihnen einen Blick in:

Boockmeyer, Fischbeck, Neubert: Fit fürs Studium – Informatik. Rheinwerk Computing 2017.

Grundlagen der diskreten Mathematik (*Zahlen*, *Mengen*, *Funktionen*, *Logik*, *einfache Beweistechnik*) werden in Kapitel 2 kurz besprochen. Dies kann jedoch keine mathematische Grundausbildung ersetzen, sondern soll lediglich Ihr Wissen auffrischen und die Begriffe und Notationen dieses Buches festlegen.

Ebenso wie diese Notation wird die Themenauswahl des Buches sich selten genau mit dem Inhalt einer Einführungsvorlesung in theoretischer Informatik decken. Mein Ziel ist es, Ihnen ein Verständnis zu vermitteln, das Sie dann auch auf andere Modelle und Themenschwerpunkte übertragen können.

Dabei können Sie selbst entscheiden, wie tief Sie in ein Thema einsteigen wollen: Genügt Ihnen die Kenntnis von den Grundaussagen, möchten Sie

diese konzeptuell verstehen, im Detail formal kennenlernen oder selbst mit den Modellen weiterarbeiten können?

Zu Beginn jedes Kapitels gebe ich Ihnen eine Einführung ins Thema und eine intuitive Erklärung der Aussagen: Worin liegt die praktische Relevanz des Themas? Was sind grundlegende Konzepte, Aussagen und deren Auswirkungen? Welche konkreten Sätze gelten, warum gelten diese und welche Idee steckt hinter deren Beweis?

An einigen Stellen vereinfacht diese Einführung. Im Anschluss gehe ich daher stets auf die formale Definition der Inhalte ein: Wie sind die zugrundeliegenden Modelle genau definiert, wie notieren wir die angesprochenen Aussagen präzise und formal korrekt?

Um mit den Modellen und Aussagen arbeiten zu können, zeige ich Ihnen dann die dazugehörigen Werkzeuge, jeweils in allgemeiner Erklärung und an einem Beispiel demonstriert: Welche Fragestellungen kann man jetzt mit dem Modell beantworten? Welche Werkzeuge gibt es dafür und wie benutzt man sie?

Selbstredend finden in diesem Buch nicht alle spannenden Konzepte der theoretischen Informatik Platz. Am Ende einiger Kapitel gebe ich Ihnen daher in einem Ausblick eine kurze Zusammenfassung von verwandten oder darüber hinausgehenden Inhalten: Welche Themen wurden hier nicht besprochen und wo finden Sie weiterführende Informationen dazu?

Zum Ende jedes Kapitels finden Sie Aufgaben und die dazugehörigen Lösungen. Zu einzelnen Aufgaben sind im Buch nur Lösungshinweise dargelegt, die ausführlichen Lösungen finden Sie als PDF-Download auf der Website des Rheinwerk-Verlags unter <https://www.rheinwerk-verlag.de/5092> auf der Registerkarte MATERIALIEN.

1.4 Danksagungen

Dieses Buch basiert auf den Erfahrungen, die ich in der Arbeit mit Studierenden in den Grundlagenvorlesungen zur Theoretischen Informatik sammeln konnte. Mein herzlicher Dank für viele anregende Gespräche, Konzepte und Experimente zu guter Lehre in theoretischen Veranstaltungen gilt Prof. Christoph Kreitz, Prof. Jürgen Dassow und insbesondere Prof. Tobias Friedrich und Dr. Timo Kötzing, deren Lehrveranstaltungen ich in den vergangenen Jahren mitgestalten durfte.

Wissen und Intuition

Formale Definition

Werkzeuge

Ausblick

Übungsaufgaben und Lösungen

Immer wieder sprechen mich Studierende darauf an, ob ich aus den gesammelten Erfahrungen nicht ein Buch als Vorlesungsbegleiter schreiben könnte. Dass dies nun tatsächlich geklappt hat, ist dem Team des Rheinwerk Verlags zu verdanken. Besonders danken möchte ich meiner Lektorin Almut Poll, die das Projekt betreut hat und trotz aller Verzögerungen und späten Anpassungen nie das Vertrauen in meine Arbeit verloren hat, sowie Norbert Englert, der die Umsetzung des von mir gewünschten Formelformats möglich gemacht hat.

Torsten T. Will ist es als Reviewer nicht nur gelungen, vielerlei Verbesserungen ins Buch einzubringen, sondern er hat darüber hinaus seine Anmerkungen und Gedanken auch noch so formuliert, dass ich mich über jeden Kommentar freuen konnte und durch seine Randnotizen sogar regelmäßig zum Schmunzeln gebracht wurde. Danke dafür!

Eine ganze Reihe von Personen hat Teile des Manuskripts testgelesen, mir dabei geholfen, Erklärungen noch verständlicher zu machen, und auch den einen oder anderen Fehler entdeckt. Vielen Dank an Arne Boockmeyer, Jonas Chromik, Philipp Fischbeck, Martin Krejca, Jannik Peters, Martin Schirneck, Sebastian Serth und Jennifer Stamm.

Trotz aller Sorgfalt und Reviews gibt es sicherlich immer noch einiges, was am Buch verbessert werden kann. Lassen Sie mir gerne über die Rheinwerk-Website Ihr Feedback zum Buch zukommen – egal ob Lob oder Kritik, mein Dank gilt Ihnen in jedem Fall!

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.

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

Berechnungsmodell

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.

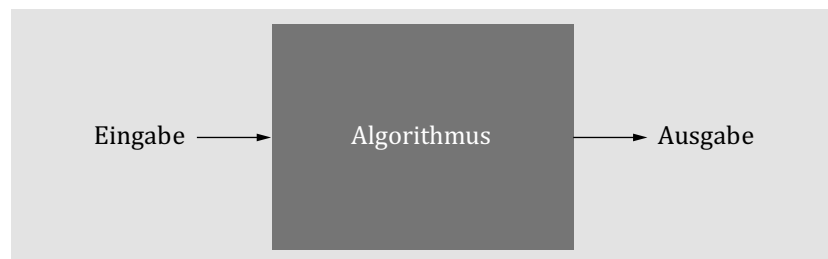


Abbildung 3.1 Grundschema eines Algorithmus

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.

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

Berechnen

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.

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.

Algorithmen-codierung

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.

Nicht berechenbar

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.

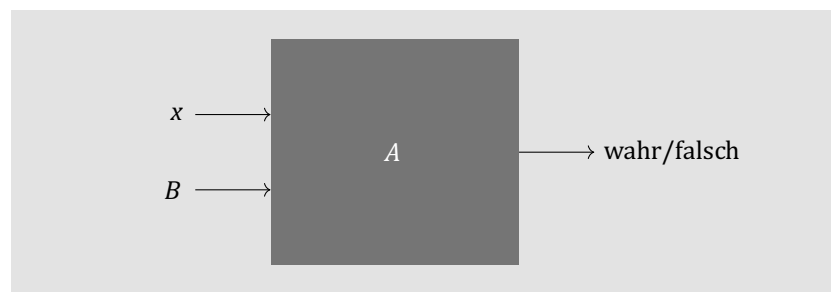


Abbildung 3.2 Der hypothetische Lösungsalgorithmus A für das Halteproblem

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.

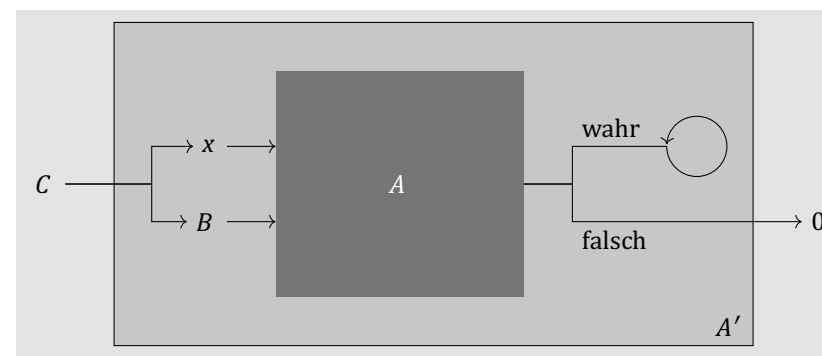


Abbildung 3.3 Der Algorithmus A' benutzt intern A .

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!

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*.

Nicht entscheidbar

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:

Auswirkungen des Halteproblems

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.

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.

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 Algo-

rithmus 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.

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.

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.

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

RAM-Modell

Adressierter Speicher

Wortbreite

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

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

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

Wir gehen davon aus, dass jeder Algorithmus zumindest einen Schritt für die Rückgabe des Ergebnisses benötigt, daher ist die Zielmenge von 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.

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

TIME_A

Worst-Case-Laufzeit

Größe hat. Wir definieren dafür zusätzlich die Funktion

$\text{WORSTCASETIME}_A: \mathbb{N} \rightarrow \mathbb{N}^+$ als

$$\forall n \in \mathbb{N}: \text{WORSTCASETIME}_A(n) := \max\{\text{TIME}_A(w) \mid w \in \{0, 1\}^n\}.$$

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
$\text{TIME}_A(n)$	33	36	40	45	52	61	72	84	97
$\text{TIME}_B(n)$	13	14	17	25	41	71	121	198	309

Tabelle 12.1 Fiktive Schrittzahlen 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.

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?

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

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

Wachstum der
Laufzeitfunktion

12

Landau-Notation

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

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 $f, g: \mathbb{N} \rightarrow \mathbb{R}^+$ der folgende Zusammenhang:

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

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.

Landau-Symbole für Laufzeiten

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 $f, g: \mathbb{N} \rightarrow \mathbb{R}^+$ 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

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.

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.

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

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

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 $\log(n!) \in \Theta(n \log(n))$.

Auf einen Blick

1	Einführung	15
2	Mathematische Notation	23

TEIL I Berechenbarkeit und formale Sprachen

3	Einführung in die Berechenbarkeitstheorie	67
4	Problemtypen	73
5	Einführung in formale Sprachen	85
6	Reguläre Sprachen	91
7	Kontextfreie Sprachen	161
8	Kontextsensitive Sprachen	193
9	Aufzählbare Sprachen	197
10	Nicht Berechenbares	241

TEIL II Algorithmik

11	Einführung in Algorithmik	261
12	Obere Schranken für Laufzeiten	263
13	Laufzeiten von Datenstrukturen	275
14	Brute-Force-Algorithmen	285
15	Greedy-Algorithmen	295
16	Divide and Conquer	313
17	Dynamische Programmierung	335
18	Amortisierte Analyse	351

TEIL III Komplexitätstheorie

19	Einführung in die Komplexitätstheorie	357
20	Beweistechniken für untere Schranken	361
21	P vs. NP: Bedingte untere Schranken	377
22	Ausblick: Parametrisierte Analyse	408

Inhalt

1	Einführung	15
1.1	Kompetenzen für die theoretische Arbeit	16
1.1.1	Abstraktionsvermögen	16
1.1.2	Präzises Arbeiten	16
1.1.3	Frustrationstoleranz und Kreativität	17
1.1.4	Kommunikationsfähigkeit	17
1.2	Themen der theoretischen Informatik	18
1.2.1	Berechenbarkeitstheorie	18
1.2.2	Algorithmik	19
1.2.3	Komplexitätstheorie	19
1.3	Anleitung fürs Buch	20
1.4	Danksagungen	21
2	Mathematische Notation	23
2.1	Logische Aussagen	24
2.1.1	Verknüpfung von logischen Aussagen	24
2.1.2	Vorrangregeln	25
2.1.3	Variablen	26
2.1.4	Umformungsregeln	26
2.2	Mengen	27
2.2.1	Mengenoperationen	28
2.2.2	Quantoren und Prädikate	29
2.2.3	Mengenbeziehungen	30
2.2.4	Tupel	31
2.3	Relationen und Funktionen	32
2.3.1	Binäre Relationen	32
2.3.2	Funktionen	33
2.3.3	Binäre Operationen über Mengen	36

2.4	Graphen	37
2.4.1	Grundbegriffe	37
2.4.2	Pfade, Kreise und Bäume	38
2.4.3	Teilgraphen und Spannbäume	39
2.4.4	Gerichtete Kanten	39
2.4.5	Gewichtete Graphen	39
2.5	Unendlichkeiten und Abzählbarkeit	40
2.5.1	Abzählbar unendliche Mengen	40
2.5.2	Überabzählbar unendliche Mengen	41
2.6	Beweistechniken	42
2.6.1	Grundprinzipien des Beweisens	42
2.6.2	Direkter Beweis	44
2.6.3	Fallunterscheidungen	46
2.6.4	Beweis per Kontraposition	48
2.6.5	Indirekter Beweis / Beweis per Widerspruch	49
2.6.6	Vollständige Induktion	50
2.6.7	Konstruktive und nichtkonstruktive Beweise	54
2.6.8	Typische Schwierigkeiten und Fehler beim Beweisen ...	54
2.7	Aufgaben	57
2.8	Lösungen	58

TEIL I Berechenbarkeit und formale Sprachen

3	Einführung in die Berechenbarkeitstheorie	67
3.1	Algorithmus	68
3.2	Zu viele Funktionen	69
3.3	Das Halteproblem	70
3.4	Kontrollfragen	72
3.5	Antworten	72

4	Problemtypen	73
4.1	Formalisierung von Problemen	73
4.2	Funktionen berechnen	75
4.3	Datencodierung	75
4.4	Sprachen entscheiden	78
4.5	Problemklassen der Berechenbarkeitstheorie	79
4.6	Aufgaben	82
4.7	Lösungen	83
5	Einführung in formale Sprachen	85
5.1	Definition	85
5.2	Die Chomsky-Hierarchie	88
5.3	Aufgaben	89
5.4	Lösungen	90
6	Reguläre Sprachen	91
6.1	Deterministische endliche Automaten	92
6.1.1	Formale Definition	93
6.1.2	Sprachbeweise	95
6.1.3	Simultane strukturelle Induktion	98
6.1.4	Automatenentwurf	100
6.2	Nichtdeterministische endliche Automaten	103
6.2.1	Formale Definition	105
6.2.2	Erweiterung: Epsilonübergänge	106
6.2.3	Äquivalenz zu deterministischen Automaten	108
6.3	Grammatiken	111
6.3.1	Formale Definition	113
6.3.2	Reguläre Grammatiken	114

6.3.3	Varianten regulärer Grammatiken	115
6.3.4	Äquivalenz zu endlichen Automaten	116
6.4	Reguläre Ausdrücke	120
6.4.1	Formale Definition	120
6.4.2	Äquivalenz zu endlichen Automaten	121
6.5	Abschlusseigenschaften	127
6.5.1	Abgeschlossenheit unter Vereinigung, Konkatenation und kleenescher Hülle	127
6.5.2	Abgeschlossenheit unter Schnitt und Differenz	128
6.5.3	Abgeschlossenheit unter Komplement	129
6.5.4	Abgeschlossenheit unter Spiegelung	130
6.5.5	Abgeschlossenheit unter Homomorphismen	130
6.5.6	Beweise mit Abschlusseigenschaften	131
6.6	Entscheidungsprobleme auf regulären Sprachen	132
6.7	Äquivalenzklassenzerlegung	134
6.7.1	Die Nerode-Relation	135
6.7.2	Minimale Automaten	136
6.7.3	Der Table-Filling-Algorithmus	137
6.8	Nichtreguläre Sprachen	139
6.8.1	Der Satz von Myhill-Nerode	139
6.8.2	Das Pumping-Lemma für reguläre Sprachen	140
6.8.3	Beweise mit Abschlusseigenschaften	143
6.9	Ausblick	144
6.10	Aufgaben	144
6.11	Lösungen	149
7	Kontextfreie Sprachen	161
7.1	Kontextfreie Grammatiken	162
7.2	Eindeutige Ableitungsbäume	164
7.3	Chomsky-Normalform	166
7.4	Exkurs: Kellerautomaten	170
7.4.1	Formale Definition	172
7.4.2	Varianten	174

7.5	Abschlusseigenschaften	175
7.6	Entscheidungsprobleme auf kontextfreien Sprachen	176
7.6.1	Wortproblem: Der Cocke-Younger-Kasami- Algorithmus (CYK)	176
7.6.2	Leerheitsproblem	180
7.7	Nicht-kontextfreie Sprachen	181
7.7.1	Das Pumping-Lemma für kontextfreie Sprachen	181
7.7.2	Beweise mit Abschlusseigenschaften	183
7.8	Ausblick	183
7.9	Aufgaben	184
7.10	Lösungen	186
8	Kontextsensitive Sprachen	193
8.1	Kontextsensitive und monotone Grammatiken	194
8.2	Das Wortproblem auf kontextsensitiven Sprachen	195
9	Aufzählbare Sprachen	197
9.1	Turingmaschinen	199
9.1.1	Berechnungen mit Turingmaschinen	200
9.1.2	Varianten und Ausblick	201
9.2	While-Programme	202
9.2.1	Syntax und Semantik	203
9.2.2	Identität und konstante Funktionen	205
9.2.3	Einfache Arithmetik: Addition	206
9.2.4	Aufrufen von Unterprogrammen	209
9.2.5	Fallunterscheidungen, Logik und Vergleiche	213
9.2.6	For-Schleifen	215
9.2.7	Cantorsche Paarungsfunktion	215
9.2.8	Datenstrukturen	217
9.3	Gödelnummern	218
9.4	Das universelle While-Programm	220

9.5	Das schrittbeschränkte universelle While-Programm	223
9.6	Diagonalisierung und min-Suche	224
9.7	Prädikate für semi-entscheidbare Sprachen	226
9.8	Semi-Entscheidbarkeit vs. Aufzählbarkeit	227
9.9	Das S-m-n-Theorem	228
9.10	Das kleenesche Rekursionstheorem	230
9.11	Weitere Modelle und Charakterisierungen	233
9.12	Aufgaben	233
9.13	Lösungen	235

10 Nicht Berechenbares 241

10.1	Beweise mit KRT	243
10.2	Der Satz von Rice	244
10.3	Reduktionen	246
10.4	RE-Vollständigkeit	250
10.5	Ausblick: Die arithmetische Hierarchie	251
10.6	Aufgaben	252
10.7	Lösungen	254

TEIL II Algorithmik

11 Einführung in Algorithmik 261

12 Obere Schranken für Laufzeiten 263

12.1	Das Maschinenmodell	264
12.2	Die Laufzeit eines Algorithmus	267

12.3	Die Größe einer Eingabe	268
12.4	Die Landau-Notation	268
12.5	Aufgaben	271
12.6	Lösungen	272

13 Laufzeiten von Datenstrukturen 275

13.1	Arrays	275
13.2	Listen	277
13.3	Verschachtelte Datenstrukturen und Graphen	279
13.4	Aufgaben	281
13.5	Lösungen	282

14 Brute-Force-Algorithmen 285

14.1	Lineare Suche	286
14.2	Backtracking/Tiefensuche	288
14.3	Aufgaben	292
14.4	Lösungen	293

15 Greedy-Algorithmen 295

15.1	Beweis mit Austauschargument	296
15.1.1	Farbeimer kaufen	296
15.1.2	Minimale Spannäume	299
15.2	Greedy stays ahead	302
15.3	Aufgaben	304
15.4	Lösungen	306

16	Divide and Conquer	313
16.1	Mergesort	314
16.1.1	Algorithmus	314
16.1.2	Korrektheit	316
16.1.3	Laufzeit	317
16.2	Binäre Suche	319
16.2.1	Algorithmus	320
16.2.2	Korrektheit	320
16.2.3	Laufzeit	321
16.3	Multiplikation großer Zahlen	321
16.3.1	Algorithmus	322
16.3.2	Laufzeitanalyse	323
16.3.3	Der Algorithmus von Karazuba	324
16.4	Das Mastertheorem	325
16.5	Ausblick	326
16.6	Aufgaben	327
16.7	Lösungen	329
17	Dynamische Programmierung	335
17.1	Fibonacci-Zahlen	336
17.2	Rückgeld geben	337
17.3	Der Algorithmus von Dijkstra	341
17.4	Aufgaben	344
17.5	Lösungen	346
18	Amortisierte Analyse	351
18.1	Dynamische Arrays	351
18.2	Guthabenmethode	353
18.3	Ausblick	353

TEIL III Komplexitätstheorie		
19	Einführung in die Komplexitätstheorie	357
19.1	Die Komplexität eines Problems	358
19.2	Bedingte Schranken	358
19.3	Auswege für schwierige Probleme	359
20	Beweistechniken für untere Schranken	361
20.1	Die Ausgabegröße	362
20.2	Das informationstheoretische Argument	363
20.2.1	Analyse der Ablaufpfade	364
20.2.2	Analyse des Informationsgewinns pro Anfrage	366
20.3	Das Adversary-Argument	367
20.4	Reduktionen	370
20.5	Aufgaben	372
20.6	Lösungen	374
21	P vs. NP: Bedingte untere Schranken	377
21.1	Die Komplexitätsklasse P	378
21.2	Die Komplexitätsklasse NP	380
21.2.1	Ein Praxisbeispiel: Independent Set	381
21.2.2	Die Definition von (Nicht-)Determinismus	383
21.2.3	Nichtdeterministisch berechnen und deterministisch verifizieren	386
21.3	Polynomialzeitreduktionen	388
21.3.1	Definition	389
21.3.2	Beispiel: Independent Set und Vertex Cover	390
21.4	NP-schwere und NP-vollständige Probleme	392
21.4.1	Der Satz von Cook	395

21.4.2	Von CNF-SAT zu 3-SAT	396
21.4.3	Von 3-SAT zu IS	398
21.4.4	Von 3-SAT zu Subset-Sum	401
21.5	Ausblick: Mehr NP-vollständige Probleme	404
21.6	Aufgaben	405
21.7	Lösungen	406
22	<u>Ausblick: Parametrisierte Analyse</u>	408
Index		410