Algorithmen – Eine Einführung

von

Prof. Dr. Thomas H. Cormen

Prof. Dr. Charles E. Leiserson

Prof. Dr. Ronald Rivest

Prof. Dr. Clifford Stein

Aus dem Englischen von

Prof. Dr. rer. nat. habil. Paul Molitor

Martin-Luther-Universität Halle-Wittenberg

4., durchgesehene und korrigierte Auflage

Oldenbourg Verlag München

Autorisierte Übersetzung der englischensprachigen Ausgabe, die bei *The MIT Press, Massachusetts Institute of Technology und McGraw-Hill Book Company* unter dem Titel *Introduction to Algorithms, Third Edition* erschienen ist.
Copyright © 2009 MIT Press, McGraw-Hill Book Company

Übersetzung:
Prof. Dr. rer. nat. habil. Paul Molitor, Institut für Informatik,
Martin-Luther-Universität Halle-Wittenberg

Lektorat: Johannes Breimeier
Herstellung: Tina Bonertz
Titelbild: www.thinkstockphotos.de
Einbandgestaltung: hauser lacour

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Library of Congress Cataloging-in-Publication Data
A CIP catalog record for this book has been applied for at the Library of Congress.

Dieses Werk ist urheberrechtlich geschützt. Die dadurch begründeten Rechte, insbesondere die der Übersetzung, des Nachdrucks, des Vortrags, der Entnahme von Abbildungen und Tabellen, der Funksendung, der Mikroverfilmung oder der Vervielfältigung auf anderen Wegen und der Speicherung in Datenverarbeitungsanlagen, bleiben, auch bei nur auszugsweiser Verwertung, vorbehalten. Eine Vervielfältigung dieses Werkes oder von Teilen dieses Werkes ist auch im Einzelfall nur in den Grenzen der gesetzlichen Bestimmungen des Urheberrechtsgesetzes in der jeweils geltenden Fassung zulässig. Sie ist grundsätzlich vergütungspflichtig. Zuwiderhandlungen unterliegen den Strafbestimmungen des Urheberrechts.

© 2013 Oldenbourg Wissenschaftsverlag GmbH
Rosenheimer Straße 143, 81671 München, Deutschland
www.degruyter.com/oldenbourg
Ein Unternehmen von De Gruyter

Gedruckt in Deutschland

Dieses Papier ist alterungsbeständig nach DIN/ISO 9706.

ISBN 978-3-486-74861-1

Inhaltsverzeichnis

Vorwort **XIII**

I Grundlagen **1**

1 Die Rolle von Algorithmen in der elektronischen Datenverarbeitung **5**

1.1 Algorithmen ... 5

1.2 Algorithmen als Technologie ... 11

2 Ein einführendes Beispiel **17**

2.1 Sortieren durch Einfügen ... 17

2.2 Analyse von Algorithmen ... 24

2.3 Entwurf von Algorithmen ... 30

3 Wachstum von Funktionen **45**

3.1 Asymptotische Notation ... 45

3.2 Standardnotationen und Standardfunktionen ... 55

4 Teile-und-Beherrsche **67**

4.1 Das Max-Teilfeld-Problem ... 70

4.2 Strassens Algorithmus zur Matrizenmultiplikation ... 77

4.3 Die Substitutionsmethode zum Lösen von Rekursionsgleichungen ... 85

4.4 Die Rekursionsbaum-Methode zum Lösen von Rekursionsgleichungen ... 89

4.5 Die Mastermethode zum Lösen von Rekursionsgleichungen ... 95

4.6 ∗Beweis des Mastertheorems ... 99

5 Probabilistische Analyse und randomisierte Algorithmen **115**

5.1 Das Bewerberproblem ... 115

5.2 Indikatorfunktionen ... 118

5.3 Randomisierte Algorithmen 123
5.4 * Probabilistische Analyse und mehr zur Verwendung der Indikatorfunktion 130

II Sortieren und Ranggrößen **147**

6 Heapsort **153**
6.1 Heaps 153
6.2 Die Heap-Eigenschaft aufrechterhalten 156
6.3 Einen Heap bauen 158
6.4 Der Heapsort-Algorithmus 161
6.5 Prioritätswarteschlangen 162

7 Quicksort **171**
7.1 Beschreibung von Quicksort 171
7.2 Die Performanz von Quicksort 175
7.3 Eine randomisierte Version von Quicksort 179
7.4 Analyse von Quicksort 181

8 Sortieren in linearer Zeit **191**
8.1 Untere Schranken für das Sortieren 191
8.2 Countingsort 194
8.3 Radixsort 197
8.4 Bucketsort 200

9 Mediane und Ranggrößen **213**
9.1 Minimum und Maximum 213
9.2 Auswahl in linearer erwarteter Zeit 215
9.3 Auswahl in linearer Zeit im schlechtesten Fall 219

III Datenstrukturen **227**

10 Elementare Datenstrukturen **233**
10.1 Stapel und Warteschlangen 233
10.2 Verkettete Listen 237

10.3 Implementierung von Zeigern und Objekten 242
10.4 Darstellung von gerichteten Bäumen 246

11 Hashtabellen **255**

11.1 Adresstabellen mit direktem Zugriff 256
11.2 Hashtabellen 258
11.3 Hashfunktionen 264
11.4 Offene Adressierung 272
11.5 * Perfektes Hashing 280

12 Binäre Suchbäume **289**

12.1 Was ist ein binärer Suchbaum? 289
12.2 Abfragen in einem binären Suchbaum 292
12.3 Einfügen und Löschen 296
12.4 * Zufällig erzeugte binäre Suchbäume 302

13 Rot-Schwarz-Bäume **311**

13.1 Eigenschaften von Rot-Schwarz-Bäumen 311
13.2 Rotationen 315
13.3 Einfügen eines Knotens 317
13.4 Löschen eines Knotens 325

14 Erweitern von Datenstrukturen **341**

14.1 Dynamische Ranggröße 341
14.2 Wie man eine Datenstruktur erweitert 347
14.3 Intervallbäume 350

IV Fortgeschrittene Entwurfs- und Analysetechniken **359**

15 Dynamische Programmierung **363**

15.1 Schneiden von Eisenstangen 364
15.2 Matrizen-Kettenmultiplikation 374
15.3 Elemente dynamischer Programmierung 381
15.4 Längste gemeinsame Teilsequenz 393
15.5 Optimale binäre Suchbäume 399

16 Greedy-Algorithmen **417**

16.1 Ein Aktivitäten-Auswahl-Problem 418
16.2 Elemente der Greedy-Strategie 425
16.3 Huffman-Codierungen 431
16.4 * Matroiden und Greedy-Methoden 440
16.5 * Ein Task-Scheduling-Problem als Matroid 447

17 Amortisierte Analyse **455**

17.1 Aggregat-Analyse 456
17.2 Account-Methode 460
17.3 Die Potentialmethode 462
17.4 Dynamische Tabellen 466

V Höhere Datenstrukturen **483**

18 B-Bäume **489**

18.1 Die Definition von B-Bäumen 493
18.2 Grundoperationen auf B-Bäumen 496
18.3 Löschen eines Schlüssels aus einem B-Baum 504

19 Fibonacci-Heaps **511**

19.1 Die Struktur von Fibonacci-Heaps 513
19.2 Operationen der fusionierbaren Heaps 516
19.3 Verringern eines Schlüssels und Entfernen eines Knotens 525
19.4 Beschränkung des maximalen Grades 529

20 van-Emde-Boas-Bäume **539**

20.1 Vorbereitende Ansätze 540
20.2 Eine rekursive Datenstruktur 544
20.3 Die van-Emde-Boas-Bäume 553

21 Datenstrukturen disjunkter Mengen **569**

21.1 Operationen auf disjunkten Mengen 569
21.2 Darstellung disjunkter Mengen mithilfe verketteter Listen 572

21.3 Wälder disjunkter Mengen ... 576
21.4 ∗ Analyse der Vereinigung nach dem Rang mit Pfadverkürzung ... 580

VI Graphenalgorithmen 595

22 Elementare Graphenalgorithmen 599

22.1 Darstellungen von Graphen ... 599
22.2 Breitensuche ... 603
22.3 Tiefensuche ... 613
22.4 Topologisches Sortieren ... 622
22.5 Starke Zusammenhangskomponenten ... 626

23 Minimale Spannbäume 635

23.1 Aufbau eines minimalen Spannbaums ... 636
23.2 Die Algorithmen von Kruskal und Prim ... 641

24 Kürzeste Pfade von einem Startknoten aus 655

24.1 Der Bellman-Ford-Algorithmus ... 663
24.2 Kürzeste Pfade von einem Startknoten aus in DAGs ... 667
24.3 Dijkstras Algorithmus ... 670
24.4 Differenzbedingungen und kürzeste Pfade ... 677
24.5 Beweise der Eigenschaften kürzester Pfade ... 683

25 Kürzeste Pfade für alle Knotenpaare 697

25.1 Kürzeste Pfade und Matrizenmultiplikation ... 699
25.2 Der Floyd-Warshall-Algorithmus ... 705
25.3 Johnsons Algorithmus für dünn besetzte Graphen ... 713

26 Maximaler Fluss 721

26.1 Flussnetzwerke ... 722
26.2 Die Ford-Fulkerson-Methode ... 727
26.3 Maximales bipartites Matching ... 745
26.4 ∗ Push/Relabel-Algorithmen ... 749
26.5 ∗ Der Relabel-to-Front-Algorithmus ... 762

VII Ausgewählte Themen **781**

27 Mehrfädige Algorithmen **785**

27.1 Grundlagen von dynamischem Multithreading 787
27.2 Mehrfädige Matrizenmultiplikation 806
27.3 Mehrfädiges Sortieren durch Mischen 811

28 Operationen auf Matrizen **827**

28.1 Lösen linearer Gleichungssysteme 827
28.2 Matrixinversion 841
28.3 Symmetrische positiv definite Matrizen, Summe der quadratischen Fehler 846

29 Lineare Programmierung **857**

29.1 Standard- und Schlupfform 864
29.2 Darstellung von Problemen als lineare Programme 872
29.3 Der Simplexalgorithmus 878
29.4 Dualität 893
29.5 Die initiale zulässige Basislösung 899

30 Polynome und die FFT **911**

30.1 Darstellung von Polynomen 913
30.2 Die DFT und FFT 919
30.3 Effiziente Implementierung der FFT 927

31 Zahlentheoretische Algorithmen **937**

31.1 Elementare zahlentheoretische Begriffe 938
31.2 Größter gemeinsamer Teiler 944
31.3 Modulare Arithmetik 950
31.4 Lösen modularer linearer Gleichungen 957
31.5 Der chinesische Restsatz 962
31.6 Potenzen eines Elements 965
31.7 Das RSA-Kryptosystem 970
31.8 ∗ Primzahltests 977
31.9 ∗ Primfaktorzerlegung 987

32 String-Matching **997**

32.1 Der naive String-Matching-Algorithmus ... 999
32.2 Der Rabin-Karp-Algorithmus ... 1002
32.3 String-Matching mit endlichen Automaten ... 1007
32.4 ∗ Der Knuth-Morris-Pratt-Algorithmus ... 1014

33 Algorithmische Geometrie **1025**

33.1 Eigenschaften von Strecken ... 1026
33.2 Bestimmung von Schnittpunkten in einer Menge von Strecken ... 1032
33.3 Bestimmen der konvexen Hülle ... 1039
33.4 Berechnung des dichtesten Punktepaares ... 1050

34 NP-Vollständigkeit **1059**

34.1 Polynomielle Zeit ... 1064
34.2 Verifikation in polynomieller Zeit ... 1072
34.3 NP-Vollständigkeit und Reduktion ... 1077
34.4 NP-Vollständigkeitsbeweise ... 1088
34.5 NP-vollständige Probleme ... 1096

35 Approximationsalgorithmen **1117**

35.1 Das Knotenüberdeckungsproblem ... 1119
35.2 Das Problem des Handelsreisenden ... 1122
35.3 Das Mengenüberdeckungsproblem ... 1128
35.4 Randomisierung und lineare Programmierung ... 1134
35.5 Das Teilsummenproblem ... 1139

VIII Anhang **1151**

A Summen **1155**

A.1 Summenformeln und Eigenschaften ... 1155
A.2 Abschätzungen für Summen ... 1159

B Mengen usw. **1169**

B.1 Mengen ... 1169
B.2 Relationen ... 1174

B.3 Funktionen ... 1176
B.4 Graphen ... 1178
B.5 Bäume ... 1183

C Kombinatorik und Wahrscheinlichkeitstheorie 1193

C.1 Kombinatorik ... 1193
C.2 Wahrscheinlichkeiten ... 1199
C.3 Diskrete Zufallsvariablen ... 1205
C.4 Die geometrische Verteilung und die Binomialverteilung ... 1211
C.5 ∗ Die Ränder der Binomialverteilung ... 1217

D Matrizen 1227

D.1 Matrizen und Matrizenoperationen ... 1227
D.2 Elementare Matrizeneigenschaften ... 1232

Literaturverzeichnis 1241

Index 1265

Vorwort

Bevor es Rechner gab, gab es Algorithmen. Aber jetzt, wo es Rechner gibt, gibt es sogar noch mehr Algorithmen. Die elektronische Datenverarbeitung beruht auf Algorithmen.

Dieses Buch bietet eine umfassende Einführung in das moderne Studium von Algorithmen. Es stellt viele Algorithmen vor, behandelt sie detailliert und macht zudem deren Entwurf und deren Analyse allen Leserschichten zugänglich. Wir haben uns bemüht, Erklärungen elementar zu halten, ohne auf Tiefe oder mathematische Exaktheit zu verzichten.

Jedes Kapitel stellt einen Algorithmus, eine Entwurfstechnik, ein Anwendungsgebiet oder ein verwandtes Thema vor. Algorithmen werden in deutscher Sprache beschrieben und in Pseudocode entworfen, der für jeden lesbar sein sollte, der schon selbst ein wenig programmiert hat. Das Buch enthält 244 Abbildungen – viele bestehen aus mehreren Teilen – die verdeutlichen, wie die Algorithmen arbeiten. Da wir *Effizienz* als Entwurfskriterium betonen, schließen wir eine sorgfältige Analyse der Laufzeiten all unserer Programme mit ein.

Der Text ist in erster Linie für den Gebrauch in Grundvorlesungen und graduierten Lehrveranstaltungen zu Algorithmen und Datenstrukturen gedacht. Da er sowohl technische Fragen als auch mathematische Aspekte des Algorithmenentwurfs diskutiert, ist er ebenso zum Selbststudium für Menschen mit technischen Berufen geeignet.

In dieser dritten Auflage haben wir abermals das gesamte Buch aktualisiert. Die Änderungen sind vielfältig und umfassen insbesondere neue Kapitel, überarbeiteter Pseudocode und einen lebhafteren Schreibstil.[1]

An die Lehrenden

Wir haben das Buch so gestaltet, dass es sowohl vielseitig als auch vollständig ist. Es sollte für Sie für eine Vielzahl von Lehrveranstaltungen von Nutzen sein, sowohl in einer Lehrveranstaltung zu Datenstrukturen für Studienanfänger als auch in einem Graduiertenkurs über Algorithmen. Da wir erheblich mehr Stoff bereitgestellt haben als eine typische, einsemestrige Veranstaltung umfasst, können Sie dieses Buch als eine Art „Buffet“ oder „bunte Mischung“ betrachten, aus der Sie denjenigen Stoff auswählen

[1] Die vorliegende deutsche Übersetzung enthält nicht nur die in der dritten Auflage des englischen Buches enthaltenen Änderungen. Wir haben auch die alte deutsche Auflage auf umständliche Übersetzungen durchforstet und sie entsprechend überarbeitet. Darüber hinaus sind in dem vorliegenden Druck Korrekturen aller Fehler aus dem Errata des englischen Originals (siehe http://www.cs.dartmouth.edu/ thc/clrs-bugs/bugs-3e.php) bis einschließlich 24.02.2013 eingepflegt.

und entnehmen können, der die Lehrveranstaltung, die Sie halten möchten, am besten unterstützt.

Es sollte Ihnen leicht fallen, Ihre Lehrveranstaltung einfach aus den Kapiteln aufzubauen, die sie benötigen. Wir haben die Kapitel relativ eigenständig gestaltet, sodass Sie sich keine Gedanken über unerwartete und unnötige Abhängigkeiten eines Kapitels gegenüber einem anderen machen müssen. Jedes Kapitel stellt zuerst den einfacheren Stoff bereit, später dann die komplizierteren Sachverhalte, wobei Abschnitte logische Zusammenhänge markieren. Innerhalb einer Lehrveranstaltung für Studienanfänger werden Sie möglicherweise nur die ersten Abschnitte eines Kapitels verwenden; innerhalb einer Graduiertenveranstaltung könnten Sie das gesamte Kapitel behandeln.

Wir haben insgesamt 957 Übungen und 158 Problemstellungen eingebunden. Jeder Abschnitt endet mit Übungen, Kapitel schließen mit Problemstellungen ab. Die Übungen sind im Allgemeinen kurze Fragen, die das Beherrschen des Lehrstoffs testen. Einige sind einfache, als Selbsttest gedachte Aufgaben, während andere wesentlich umfangreicher und als Hausarbeiten geeignet sind. Die Problemstellungen sind aufwendigere Fallstudien, die häufig neuen Stoff einführen; sie bestehen häufig aus mehreren Fragen, die die Studierenden durch die zum Erhalt der Lösung notwendigen Schritte führen.

Aus der Erfahrung mit den vorherigen Auflagen dieses Buches haben wir Lösungen zu einigen, aber beileibe nicht allen Problemstellungen und Übungen öffentlich zugänglich gemacht. Sie können über unsere Webseite http://mitpress.mit.edu/algorithms auf diese zugreifen. Sie sollten sich die Webseite anschauen, um sicher zu gehen, dass sie nicht eine Lösung einer Übungsaufgabe oder einer Problemstellung enthält, die Sie stellen wollen. Wir erwarten, dass sich die Menge der veröffentlichten Lösungen kontinuierlich über die Zeit vergrößern wird, sodass Sie sich die Webseite jedesmal anschauen sollten, wenn Sie die Lehrveranstaltung anbieten.

Wir haben diejenigen Abschnitte und Übungen mit einem Stern ($*$) versehen, die eher für fortgeschrittene Studierende als für Studienanfänger geeignet sind. Ein mit Stern versehener Abschnitt ist nicht notwendigerweise schwieriger als ein Abschnitt ohne Stern, er kann aber mehr Verständnis von höherer Mathematik erfordern. Ebenso kann eine mit Stern versehene Übung ein höheres Niveau oder mehr als nur durchschnittliche Kreativität voraussetzen.

An die Studierenden

Wir hoffen, dass Ihnen dieses Lehrbuch eine unterhaltsame Einführung in das Gebiet der Algorithmen liefert. Wir haben uns bemüht, jeden Algorithmus leicht zugänglich und interessant zu gestalten. Um Ihnen zu helfen, wenn Sie auf ungewohnte oder schwierige Algorithmen stoßen, beschreiben wir jeden Algorithmus Schritt für Schritt. Wir liefern außerdem sorgfältige Erklärungen zur Mathematik, die notwendig ist, um die Analyse der Algorithmen zu verstehen. Wenn Sie bereits über einige Vorkenntnisse auf einem Gebiet verfügen, dann werden Sie feststellen, dass die Kapitel so eingerichtet sind, dass Sie die einführenden Abschnitte überfliegen und schnell mit dem höheren Stoff fortfahren können.

Dies ist ein umfangreiches Buch und Ihre Vorlesung wird wahrscheinlich nur einen Teil des enthaltenen Stoffes behandeln. Wir haben uns bemüht, dieses Buch so zu gestalten, dass es für Sie jetzt als Lehrbuch und später während Ihrer Karriere auch als mathematisches Nachschlagewerk oder als technisches Handbuch nützlich sein wird.

Was sind die fachlichen Voraussetzungen für die Lektüre dieses Buches?

- Sie sollten etwas Programmiererfahrung haben. Insbesondere sollten Sie rekursive Prozeduren und einfache Datenstrukturen, wie zum Beispiel Felder und verkettete Listen, verstehen.

- Sie sollten Übung mit mathematischen Beweisen, im besonderen mit mathematischer Induktion haben. Einige Teile dieses Buches stützen sich auf Kenntnisse in elementarer Analysis. Darüber hinaus vermitteln Ihnen die Teile I und VIII dieses Buches alle mathematischen Methoden, die Sie benötigen werden.

Wir haben laut und deutlich Ihre Forderung gehört, Lösungen zu Problemstellungen und Übungen bereitzustellen. Unsere Webseite http://mitpress.mit.edu/algorithms verlinkt zu Lösungen für einige der Problemstellungen und Übungen. Überprüfen Sie Ihre Lösungen mit unseren. Wir bitten aber darum, uns Ihre Lösungen nicht zuzuschicken.

An die Fachleute

Die große Auswahl an Themen in diesem Buch macht dieses zu einem ausgezeichneten Handbuch zum Thema Algorithmen. Da jedes Kapitel in sich relativ geschlossen ist, können Sie sich auf die Themen konzentrieren, die Sie am meisten interessieren.

Die meisten der hier besprochenen Algorithmen haben großen praktischen Nutzen. Daher befassen wir uns auch mit den Belangen der Implementierung und anderen technischen Fragen. Wir stellen in der Regel praktische Alternativen zu den wenigen Algorithmen vor, die vorrangig von theoretischem Interesse sind.

Falls Sie einen der Algorithmen implementieren möchten, dann sollte es für Sie ziemlich einfach sein, den Pseudocode in Ihre bevorzugte Programmiersprache zu übersetzen. Wir haben den Pseudocode so entworfen, dass er jeden Algorithmus klar und knapp präsentiert. Folglich befassen wir uns nicht mit Fehlerbehandlung und anderen softwaretechnischen Fragen, die spezielle Annahmen über Ihre Programmierumgebung erfordern. Wir versuchen, jeden Algorithmus einfach und genau darzustellen, ohne es den Eigenheiten einer speziellen Programmiersprache zu ermöglichen, dessen Wesen zu verdecken.

Wir sehen ein, dass Sie Ihre Lösungen zu Problemen und Übungen nicht mit den von einem Dozenten zur Verfügung gestellten Lösungen vergleichen können, wenn Sie dieses Buch im Selbststudium benutzen. Unsere Webseite http://mitpress.mit.edu/algorithms verlinkt zu Lösungen für einige der Problemstellungen und Übungen, sodass Sie Ihre Lösungen überprüfen können. Senden Sie uns aber bitte nicht Ihre Lösungen zu.

An unsere Kollegen

Wir haben ein umfangreiches Quellenverzeichnis und Hinweise auf die aktuelle Literatur bereitgestellt. Jedes Kapitel endet mit einer Reihe von Kapitelbemerkungen, die historische Details und Hinweise geben. Die Kapitelbemerkungen bilden dennoch keine vollständige Referenz zum gesamten Gebiet der Algorithmen. Obwohl es von einem Buch dieses Umfangs schwer zu glauben sein mag, hinderte Platzmangel uns daran, viele interessante Algorithmen mit aufzunehmen.

Änderungen innerhalb der dritten Auflage

Was hat sich in der dritten Auflage in Bezug auf die zweite geändert? Die Anzahl der Änderungen entspricht in etwa der bei der zweiten Auflage. Wie wir bereits in Bezug auf diese Änderungen gesagt haben, hat sich das Buch also kaum oder ziemlich viel geändert.

Ein kurzer Blick in das Inhaltsverzeichnis zeigt, dass die meisten Kapitel und Abschnitte der zweiten Auflage auch in der dritten Auflage vorkommen. Wir haben zwei Kapitel und einen Abschnitt entfernt, haben dafür drei neue Kapitel und ansonsten zwei neue Abschnitte hinzugefügt.

Wir haben die gemischte Organisation der ersten beiden Auflagen beibehalten. Anstatt die Kapitel ausschließlich nach Problembereichen oder ausschließlich nach Techniken zu gliedern, findet man in diesem Buch beides vor. Es enthält technikbasierte Kapitel wie die Kapitel über die Teile-und-Beherrsche-Methode, dynamische Programmierung, Greedy-Algorithmen, amortisierte Analyse, NP-Vollständigkeit und Approximationsalgorithmen. Es enthält jedoch auch jeweils ganze Teile über Sortieren, Datenstrukturen für dynamische Mengen und Graphalgorithmen. Dies ist dadurch begründet, dass, wenngleich Sie wissen müssen, wie Techniken zum Entwurf oder zur Analyse von Algorithmen anzuwenden sind, aus den Problemstellungen nicht unmittelbar für Sie zu ersehen ist, welche Technik am ehesten hilft, das Problem zu lösen.

Wir geben eine Zusammenfassung der wichtigsten Änderungen in der dritten Auflage:

- Wir haben neue Kapitel zu van-Emde-Boas-Bäume und mehrfädigen (engl.: *multithreaded*) Algorithmen eingefügt. Die Grundlagen von Matrizen haben wir in den Anhang verschoben.

- Wir haben das Kapitel zu Rekursionsgleichungen überarbeitet, sodass es nunmehr die Teile-und-Beherrsche-Methode besser abdeckt. Die ersten zwei Abschnitte wenden die Teile-und-Beherrsche-Methode an, um zwei verschiedene Problemstellungen zu lösen. Der zweite Abschnitt dieses Kapitels stellt Strassens Algorithmus zur Matrixmultiplikation vor, den wir aus dem Kapitel über Matrixoperationen hierhin verschoben haben.

- Wir haben zwei Kapitel, die nur selten gelehrt werden, entfernt: binomiale Heaps und Sortiernetzwerke. Eine zentrale Idee bei Sortiernetzwerken, das 0-1-Prinzip,

findet sich in dieser Auflage als 0-1-Sortierlemma für Vergleiche-Vertausche Algorithmen in der Problemstellung 8-7 wieder. Die Behandlung von Fibonacci-Heaps beruht nicht mehr auf binomialen Heaps.

- Wir haben unsere Betrachtungen zu dynamischer Programmierung und Greedy-Algorithmen überarbeitet. Dynamische Programmierung leiten wir nun ab mit einem interessanteren Problem, dem Schneiden von Stahlstangen, als mit der Ablaufkoordinierung von Montagebändern aus der zweiten Auflage. Zudem heben wir ein bisschen mehr auf Memoisation ab, als wir dies in der zweiten Auflage gemacht haben, und führen den Begriff des Teilproblem-Graphen als eine Möglichkeit, die Laufzeit eines auf dynamischer Programmierung beruhender Algorithmus zu verstehen, ein. In unserem Anfangsbeispiel zu Greedy-Algorithmen, dem Aktivitäten-Auswahl-Problem, leiten wir Greedy-Algorithmen direkter ab, als wir dies in der zweiten Auflage getan haben.

- Die Methode, mit der wir einen Knoten aus einem binären Suchbaum (insbesondere aus Rot-Schwarz-Bäumen) löschen, gewährleistet nun, dass genau der Knoten, der gelöscht werden soll, auch tatsächlich gelöscht wird. In den ersten beiden Auflagen konnte in bestimmten Fällen ein anderer Knoten gelöscht werden, wobei zuvor dessen Inhalt in den Knoten kopiert wurde, der der Lösche-Prozedur an sich übergeben worden war. Mit dieser neuen Methode, Knoten zu löschen, kann es nicht mehr passieren, dass andere Programmteile, die Zeiger auf Knoten des Baumes verwalten, irrtümlicherweise alte Zeiger auf Knoten, die gelöscht wurden, haben.

- Die Ausführungen zu Flussnetzwerken basieren nun vollständig auf Flüssen auf Kanten. Dieser Ansatz ist intuitiver als der über den Nettofluss aus den ersten beiden Auflagen.

- Da wir die Ausführungen zu den Grundlagen von Matrizen und zu Strassens Algorithmus in andere Kapitel verschoben haben, ist das Kapitel zu Matrixoperationen kürzer als in der zweiten Auflage.

- Wir haben die Behandlung des Knuth-Morris-Patt String-Matching Algorithmus abgeändert.

- Wir haben mehrere Fehler korrigiert. Die meisten von ihnen, wenn auch nicht alle, wurden uns über das auf unserer Webseite verfügbare Fehlerverzeichnis zugeschickt.

- Aufgrund vieler Anfragen haben wir die (bisherige) Syntax unseres Pseudocodes geändert. Wir benutzen nun, wie dies auch in C, C++, Java und Python der Fall ist, "=" für eine Zuweisung und "==" für den Test auf Gleichheit. Desgleichen haben wir die Schlüsselwörter **do** und **then** entfernt und übernehmen "//" als unser Kommentarsymbol, wenn der Rest der entsprechende Zeile Kommentar sein soll. Zudem benutzen wir Punktnotation für die Angabe von Objektattributen. Unser Pseudocode bleibt prozedural. Anders formuliert, anstatt Methoden auf Objekten laufen zu lassen, rufen wir einfach Prozeduren auf, denen wir Objekte als Parameter mitgeben.

- Wir haben 100 neue Übungsaufgaben und 28 neue Problemstellungen eingefügt. Wir haben auch viele Bibliographieeinträge aktualisiert und mehrere neue hinzugefügt.
- Schlussendlich sind wir das ganze Buch durchgegangen und haben Sätze, Absätze und Abschnitte umgeschrieben, damit die Ausführungen klarer werden und der Schreibstil lebhafter wird.

Internetpräsenz

Sie können unsere Webseite http://mitpress.mit.edu/algorithms/ nutzen, um zusätzliche Informationen zu bekommen und um mit uns zu kommunizieren. Die Webseite verlinkt zu einer Liste bekannter Fehler, zu Lösungen ausgewählter Übungsaufgaben und Problemen sowie zu anderem Inhalt, den wir gegebenenfalls noch bereitstellen werden. Zudem erklärt sie (natürlich) die blöden Professorenwitze. Die Webseite sagt Ihnen auch, wie Sie uns Fehler oder Anregungen zukommen lassen können.

Wie wir das Buch hergestellt haben[2]

Wie die zweite Auflage haben wir auch die dritte Auflage mit LaTeX 2ε erstellt. Wir benutzten den Times Font mit dem auf den MathTime Pro 2 Fonts basierenden mathematischen Schriftsatz. Wir bedanken uns für die entsprechende technische Unterstützung bei Michael Spivak von Publish or Perish, Inc., Lance Carnes von Personal TeX, Inc., und Tim Tregubov vom Dartmouth College. Wie in den vorangehenden zwei Auflagen haben wir den Index mit Windex, einem C Programm, das wir geschrieben haben, übersetzt. Das Literaturverzeichnis wurde mit BibTeX erzeugt. Die PDF-Dateien für dieses Buch wurden auf einem MacBook unter OS 10.5 generiert.

Die Abbildungen der dritten Auflage haben wir mit MacDraw Pro gezeichnet, unter Zuhilfenahme des psfrag Pakets von LaTeX 2ε zur Darstellung mathematischer Ausdrücke in den Abbildungen. Leider ist MacDraw Pro eine alte Software, die seit über einem Jahrzehnt nicht mehr verkauft wird. Glücklicherweise haben wir noch eine Handvoll Macintoshes, die unter dem Betriebssystem OS 10.4 laufen und so MacDraw Pro ausführen können – jedenfalls meistens. Sogar unter dieser alten Umgebung finden wir MacDraw Pro um ein Vielfaches einfacher zu benutzen als irgendeine andere Zeichensoftware, wenn es um Zeichnungen geht, die Informatiktexte begleiten sollen. Zudem erzeugt dieses Programm schöne Ausgaben.[3] Wer weiß, wie lange unsere alten Macs noch laufen werden; also falls jemand von Apple zuhört: *Bitte bauen Sie eine MacDraw Pro Version, die unter OS X läuft!*

[2] Diese Ausführungen beziehen sich zum Teil nur auf die dritte Auflage der *englischen* Ausgabe.

[3] Wir haben uns mehrere Zeichenprogramme, die unter Mac OS X laufen, angeschaut, aber alle haben erhebliche Nachteile verglichen mit MacDraw Pro. Wir haben kurz versucht, die Abbildungen für dieses Buch mit einem anderen, gut bekannten Zeichenprogramm zu erstellen. Es stellte sich heraus, dass es wenigstens fünf Mal so lange dauert, eine Abbildung zu erzeugen, als mit MacDraw Pro, und das Ergebnis sah schlechter aus. Aus diesem Grund unsere Entscheidung, zu den alten Macintoshes und MacDraw Pro zurückzukehren.

Danksagungen zur dritten Auflage

Wir arbeiten nunmehr seit über zwei Jahrzehnte mit The MIT Press zusammen und was war das für eine wunderbare Zusammenarbeit! Wir danken Ellen Faran, Bob Prior, Ada Brunstein und Mary Reilly für ihre Hilfe und Unterstützung.

Während der Erstellung der dritten Auflage arbeiteten wir an geografisch unterschiedlichen Orten, am Dartmouth College Department of Computer Science, im MIT Computer Science and Artificial Intelligence Laboratory, und am Columbia University Department of Industrial Engineering and Operations Research. Wir danken unseren jeweiligen Universitäten und Kollegen, dass wir in dermaßen unterstützenden und anregenden Umgebungen arbeiten konnten.

Julie Sussman, P.P.A., stand uns erneut als Lektorin zur Verfügung. Wieder und wieder waren wir verblüfft über die Fehler, die wir übersehen hatten, die Julie aber fand. Sie half uns auch, die Darstellung an mehreren Stellen zu verbessern. Gäbe es eine Ruhmeshalle für Lektorinnen und Lektoren, Julie wäre ein Platz sicher. Sie ist einfach nur phänomenal. Danke, danke, danke, Julie! Priya Natarajan fand ebenfalls einige Fehler, die wir noch vor Drucklegung korrigieren konnten. Alle Fehler, die sich im Buch noch befinden (und es werden sicherlich welche geben) liegen in der Verantwortung der Autoren (und wurden vermutlich eingefügt, nachdem Julie den Text gelesen hat).

Die Behandlung der van-Emde-Boas-Bäume basiert auf Notizen von Erik Demaine, die wiederum durch Michael Bender beeinflusst sind. Wir haben auch Ideen von Javed Aslam, Bradley Kuszmaul und Hui Zha in diese Auflage eingearbeitet.

Das Kapitel über Multithreading basiert auf Notizen, die ursprünglich zusammen mit Harald Prokop geschrieben worden sind, und wurde beeinflusst durch mehrere andere Kollegen, die am Cilk Projekt des MIT beteiligt sind. Zu nennen sind insbesondere Bradley Kuszmaul und Matteo Frigo. Der Entwurf des mehrfädigen Pseudocodes wurde durch die MIT Cilk Erweiterung von C und die Cilk Arts Cilk++ Erweiterung von C++ inspiriert.

Wir bedanken uns auch bei den vielen Lesern der ersten und zweiten Auflage, die uns auf Fehler aufmerksam machten oder uns Anregungen zukommen ließen, wie dieses Buch verbessert werden könnte. Wir korrigierten alle echten Fehler, die uns gemeldet worden sind, und haben so viele Anregungen, wie wir konnten, eingebunden. Wir sind glücklich darüber, dass die Anzahl dieser Mitwirkenden so groß geworden ist, dass wir bedauern müssen, dass es nicht mehr möglich ist, alle aufzuzählen.

Zu guter Letzt danken wir unseren Frauen – Nicole Cormen, Wendy Leiserson, Gail Rivest, and Rebecca Ivry – und unseren Kindern – Ricky, Will, Debby und Katie Leiserson; Alex und Christopher Rivest; und Molly, Noah und Benjamin Stein – für ihre Liebe und Unterstützung, während wir das Buch geschrieben haben. Die Geduld und der Zuspruch unserer Familien machten dieses Projekt möglich. Wir widmen ihnen liebevoll dieses Buch.

Thomas H. Cormen — *Lebanon, New Hampshire*
Charles E. Leiserson — *Cambridge, Massachusetts*
Ronald L. Rivest — *Cambridge, Massachusetts*
Clifford Stein — *New York, New York*

Februar 2009

Teil I

Grundlagen

Einführung

Dieser Teil wird Sie an den Entwurf und die Analyse von Algorithmen heranführen. Er ist als behutsame Einführung gedacht – eine Einführung zu wie wir in diesem Buch Algorithmen spezifizieren, zu einigen Entwurfsstrategien, die wir innerhalb dieses Buches verwenden werden, und zu vielen fundamentalen Begriffen, die bei der Analyse von Algorithmen benutzt werden. Die nachfolgenden Teile des Buches werden auf dieser Grundlage aufbauen.

Kapitel 1 gibt einen Überblick über Algorithmen und ihre Rolle in modernen Rechensystemen. Dieses Kapitel definiert, was wir unter einem Algorithmus verstehen, und listet einige Beispiele auf. Es liefert Argumente dafür, dass wir Algorithmen als eine Technologie ansehen sollten, genauso wie schnelle Hardware, grafische Benutzeroberflächen, objektorientierte Systeme und Netzwerke Technologien sind.

In Kapitel 2 werden wir unsere ersten Algorithmen kennen lernen. Sie behandeln das Problem, n Zahlen zu sortieren. Die Algorithmen sind in Pseudocode beschrieben, der, obwohl er nicht unmittelbar in eine konventionelle Programmiersprache übersetzbar ist, die Struktur eines Algorithmus hinreichend klar vermittelt, sodass Sie ihn in einer Programmiersprache Ihrer Wahl implementieren können sollten. Die behandelten Sortieralgorithmen sind Sortieren durch Einfügen (engl.: *insertion sort*), das eine iterative Methode benutzt, und Sortieren durch Mischen (engl.: *merge sort*), das eine als „Teile-und-Beherrsche" (engl.: *divide and conquer*) bekannte rekursive Technik verwendet. Obwohl die von beiden benötigte Zeit mit dem Wert n anwächst, unterscheiden sich die Wachstumsraten beider Algorithmen voneinander. Wir bestimmen deren Laufzeiten im Kapitel 2 und entwickeln eine nützliche Notation, um diese zu beschreiben.

Kapitel 3 legt diese Notation genau fest, die wir als asymptotische Notation bezeichnen werden. Kapitel 3 beginnt mit der Definition einiger asymptotischer Notationen, die wir benutzen, um obere und/oder untere Schranken für Laufzeiten von Algorithmen anzugeben. Der Rest von Kapitel 3 ist in erster Linie eine Vorstellung mathematischer Notationen, mit dem vorrangigen Ziel, Ihnen unsere mathematischen Notationen nahe zu bringen, und nicht Ihnen neue Konzepte zu vermitteln.

Kapitel 4 befasst sich eingehender mit der Teile-und-Beherrsche-Methode, die bereits in Kapitel 2 eingeführt wurde. Es zeigt weitere Beispiele von Teile-und-Beherrsche-Algorithmen, insbesondere Strassens erstaunliche Methode zur Multiplikation zweier $n \times n$-Matrizen. Kapitel 4 enthält Methoden zum Auflösen von Rekursionsgleichungen, die für die Beschreibung von Laufzeiten rekursiver Algorithmen nützlich sind. Eine sehr leistungsfähige Methode ist die „Mastermethode", die wir oft beim Lösen von Rekursionsgleichungen, die bei Teile-und-Beherrsche-Algorithmen auftreten, anwenden. Wenngleich ein großer Teil von Kapitel 4 sich mit dem Korrektheitsbeweis der Mastermethode beschäftigt, können Sie den Beweis überspringen und doch die Mastermethode anwenden.

Kapitel 5 führt die probabilistische Analyse und randomisierte Algorithmen ein. Üblicherweise benutzen wir die probabilistische Analyse, um die Laufzeit eines Algorithmus zu bestimmen, wenn es sich um einen Algorithmus handelt, dessen Laufzeit bei Eingaben gleicher Größe aufgrund einer inhärenten Wahrscheinlichkeitsverteilung unterschiedlich sein kann. In einigen Fällen gehen wir davon aus, dass die Eingaben gemäß einer bekannten Wahrscheinlichkeitsverteilung verteilt sind, sodass wir die Laufzeit über alle möglichen Eingaben mitteln. In anderen Fällen ergibt sich die Wahrscheinlichkeitsverteilung nicht aus den Eingaben, sondern aus Zufallsentscheidungen, die während des Ablaufs des Algorithmus getroffen werden. Einen Algorithmus, dessen Verhalten nicht nur durch seine Eingabe bestimmt ist, sondern auch durch von einem Zufallszahlengenerator erzeugte Größen, nennt man randomisierter Algorithmus. Wir können randomisierte Algorithmen benutzen, um den Eingaben eine Wahrscheinlichkeitsverteilung aufzuerlegen und damit abzusichern, dass keine spezielle Eingabe eine schwache Performanz verursacht. Wir können die Idee der Randomisierung auch benutzen, um die Fehlerrate von Algorithmen zu begrenzen, bei denen es zulässig ist, dass sie eine beschränkte Zahl inkorrekter Resultate liefern.

Die Anhänge A–D enthalten zusätzlichen Stoff, der Ihnen hilfreich sein kann, wenn Sie dieses Buch lesen. Sie werden wahrscheinlich einen Großteil des Stoffes in den Kapiteln des Anhangs bereits kennen, auch wenn die verwendeten Definitionen und Schreibweisen an einigen Stellen möglicherweise von denen Ihnen bisher bekannten abweichen. Sie sollten deshalb die Anhänge als Referenzmaterial betrachten. Andererseits wird Ihnen wahrscheinlich ein großer Teil des Stoffes aus Teil I noch nicht vertraut sein. Alle Kapitel in Teil I und in den Anhängen haben einen einführenden Charakter.

1 Die Rolle von Algorithmen in der elektronischen Datenverarbeitung

Was sind Algorithmen? Warum ist das Studium von Algorithmen lohnenswert? Welche Rolle spielen Algorithmen im Vergleich zu anderen, in Rechnern verwendeten Technologien? Diese Fragen werden wir in diesem Kapitel beantworten.

1.1 Algorithmen

Ein ***Algorithmus*** ist grob gesprochen eine wohldefinierte Rechenvorschrift, die eine Größe oder eine Menge von Größen als ***Eingabe*** verwendet und eine Größe oder eine Menge von Größen als ***Ausgabe*** erzeugt. Somit ist ein Algorithmus eine Folge von Rechenschritten, die die Eingabe in die Ausgabe umwandeln.

Wir können einen Algorithmus auch als Hilfsmittel betrachten, um ein genau festgelegtes ***Rechenproblem*** zu lösen. Die Formulierung des Problems legt in allgemeiner Form die benötigte Eingabe-Ausgabe-Beziehung fest. Der Algorithmus beschreibt eine spezifische Rechenvorschrift, die diese Eingabe-Ausgabe-Beziehung erzeugt.

Es kann zum Beispiel sein, dass wir eine Folge von Zahlen in nichtfallender Reihenfolgen sortieren müssen. Dieses Problem kommt in der Praxis häufig vor und bietet eine ergiebige Grundlage für die Einführung vieler standardmäßiger Entwurfstechniken und Analysewerkzeuge. Das ***Sortierproblem*** definieren wir formal wie folgt:

Eingabe: Eine Folge von n Zahlen $\langle a_1, a_2, \ldots, a_n \rangle$.

Ausgabe: Eine Permutation (Umordnung) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ der Eingabefolge, sodass $a'_1 \leq a'_2 \leq \cdots \leq a'_n$ gilt.

Ist zum Beispiel eine Eingabefolge $\langle 31, 41, 59, 26, 41, 58 \rangle$ gegeben, so gibt ein Sortieralgorithmus die Folge $\langle 26, 31, 41, 41, 58, 59 \rangle$ als Ausgabe zurück. Solch eine Eingabefolge wird als ***Instanz*** des Sortierproblems bezeichnet. Im Allgemeinen besteht eine ***Instanz eines Problems*** aus einer Eingabe, die jegliche durch die Formulierung des Problems auferlegten Bedingungen erfüllt und alle notwendigen Daten umfasst, um die Lösung des Problems berechnen zu können.

Da viele Programme Daten in Zwischenschritten sortieren müssen, ist Sortieren eine Operation in der Informatik, die elementar ist. Infolgedessen stehen uns heute eine

Vielzahl guter Sortieralgorithmen zur Verfügung. Welcher Algorithmus der beste für eine gegebene Anwendung ist, hängt – neben anderen Faktoren – von der Zahl der zu sortierenden Daten, dem Umfang, in dem die Daten bereits vorsortiert sind, möglichen Einschränkungen, denen die Daten unterliegen, der Architektur des Rechners und der Art des eingesetzten Speichers (Hauptspeicher, Festplatten oder Magnetbänder) ab.

Ein Algorithmus wird als ***korrekt*** bezeichnet, wenn er für jede Eingabeinstanz mit der korrekten Ausgabe stoppt. Wir sagen, dass ein korrekter Algorithmus ein gegebenes Rechenproblem ***löst***. Ein inkorrekter Algorithmus stoppt bei einigen Eingabeinstanzen möglicherweise überhaupt nicht, oder er kann mit einer falschen Antwort anhalten. Im Gegensatz zu dem, was Sie erwarten würden, können inkorrekte Algorithmen manchmal nützlich sein, falls wir deren Fehlerrate kontrollieren können. Wir werden uns in Kapitel 31 ein Beispiel ansehen, wenn wir Algorithmen zum Auffinden von großen Primzahlen untersuchen. Gewöhnlich werden wir uns jedoch nur mit korrekten Algorithmen zu beschäftigen haben.

Ein Algorithmus kann mithilfe der natürlichen Sprache, als Computerprogramm oder sogar als Hardwareentwurf beschrieben werden. Die einzige Anforderung ist, dass die Spezifikation eine genaue Beschreibung der zu befolgenden Rechenvorschrift enthalten muss.

Welche Art von Problemen werden durch Algorithmen gelöst?

Sortieren ist keineswegs das einzige Rechenproblem, für das Algorithmen entwickelt wurden. (Was Sie bei dem Umfang des Buches wahrscheinlich bereits geahnt haben.) Praktische Anwendungen von Algorithmen sind ubiquitär und schließen die folgenden Beispiele ein:

- Das Human-Genome-Projekt hat große Fortschritte in Bezug auf das Ziel gemacht, alle 100 000 Gene der menschlichen DNA zu identifizieren, die Folge der 3 Milliarden chemischen Basenpaare, die eine menschliche DNA ausmachen, zu bestimmen, diese Informationen in einer Datenbank zu speichern und Werkzeuge zur Datenanalyse zu entwickeln. Jeder dieser Schritte erfordert anspruchsvolle Algorithmen. Obwohl die Lösungen der damit zusammenhängenden mannigfaltigen Probleme weit über den Rahmen dieses Buches hinausgehen, benutzen viele Methoden zur Lösung biologischer Probleme Ideen aus mehreren Kapiteln dieses Buches. Diese ermöglichen es den Wissenschaftlern, die Aufgaben unter effizienter Nutzung der Ressourcen zu bewältigen. Die Einsparungen liegen in der aufgewendeten Zeit, sowohl der menschlichen als auch der maschinellen, und dem Geld, da mehr Informationen aus Labortechniken gewonnen werden können.

- Das Internet macht Menschen auf der ganzen Welt eine große Menge an Informationen zugänglich und erlaubt ihnen, schnell darauf zuzugreifen. Mit Hilfe intelligenter Algorithmen können entsprechende Internetanbieter diese großen Datenmengen handhaben. Beispiele von Problemen, bei denen Algorithmen das A und O sind, sind insbesondere das Berechnen guter Routen für den Datentransport (Methoden zur Lösung solcher Probleme sind in Kapitel 24 zu finden) und die Verwendung von Suchmaschinen, um schnell diejenigen Seiten zu finden, die eine

bestimmte Information enthalten (entsprechende Methoden befinden sich in den Kapiteln 11 und 32).

- E-Commerce ermöglicht den elektronischen Handel von Waren und Dienstleistungen. Er hängt vom Schutz privater Informationen wie Kreditkartennummern, Passwörter und Bankauszügen ab. Die im elektronischen Handel benutzten Schlüsseltechnologien sind insbesondere Public-Key-Kryptographie und digitale Unterschriften, die in Kapitel 31 behandelt werden. Sie beruhen auf numerischen Algorithmen und Zahlentheorie.
- Verarbeitende Industrie und andere Unternehmen haben oft knappe Ressourcen bestmöglichst einzusetzen. Ein Ölkonzern wird wissen wollen, wo er seine Bohrlöcher platzieren soll, um den erwarteten Ertrag zu maximieren. Ein Politiker möchte entscheiden, an welcher Stelle er das Geld für Wahlwerbung ausgeben soll, um die Chancen für den Gewinn einer Wahl zu maximieren. Eine Fluggesellschaft möchte die Crews für Flüge so zusammenstellen, dass nur minimale Kosten entstehen. Es muss sichergestellt sein, dass jeder Flug abgedeckt ist und dass die staatlichen Vorschriften über die Einsatzzeiten für jede Crew eingehalten werden. Ein Internet-Service-Provider möchte bestimmen, wo zusätzliche Ressourcen eingesetzt werden sollen, um seine Kunden effizienter zu bedienen. All dies sind Beispiele für Probleme, die mit Hilfe linearer Programmierung gelöst werden können. Lineare Programmierung wird in Kapitel 29 untersucht.

Obwohl einige Details dieser Beispiele über den Umfang dieses Buches hinausgehen, geben wir doch die diesen Problemen und Problemgebieten zugrunde liegenden Methoden an. Wir zeigen in diesem Buch auch, wie verschiedene konkrete Probleme gelöst werden können, darunter die folgenden:

- Gegeben ist eine Straßenkarte, auf der der Abstand zwischen jedem Paar benachbarter Kreuzungen eingezeichnet ist. Wir wollen den kürzesten Weg von einer Kreuzung zu einer anderen finden. Die Zahl der möglichen Wege kann riesig sein, selbst wenn wir Wege, die sich selbst kreuzen, verwerfen. Wie entscheiden wir tatsächlich, welcher der möglichen Wege der kürzeste ist? Wir modellieren in diesem Fall die Straßenkarte (die selbst nur ein Modell für die tatsächlichen Straßen ist) durch einen Graphen (siehe Teil VI und Anhang B) und wollen innerhalb des Graphen den kürzesten Pfad von einem Knoten zu einem anderen finden. In Kapitel 24 werden wir sehen, wie dieses Problem effizient gelöst werden kann.
- Gegeben seien zwei geordnete Folgen $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ von Symbolen und wir möchten eine längste gemeinsame Teilfolge von X und Y finden. Eine Teilfolge von X ist einfach nur X, aus der einige (möglicherweise alle oder auch keines) ihrer Elemente entfernt worden sind. Zum Beispiel ist eine Teilfolge von $\langle A, B, C, D, E, F, G \rangle$ die Folge $\langle B, C, E, G \rangle$. Die Länge einer längsten gemeinsamen Teilfolge von X und Y ist ein Maß, das angibt, wie ähnlich sich diese beiden Folgen sind. Sind die beiden Folgen Basenpaare in DNA-Strängen, so können wir diese Stränge als ähnlich ansehen, wenn sie eine lange gemeinsame Teilfolge besitzen. Besteht X aus m Symbolen und Y aus n Symbolen, dann besitzen X und Y genau 2^m beziehungsweise 2^n mögliche Teilfolgen. Alle Teilfolgen

von X mit allen Teilfolgen von Y zu vergleichen, benötigt untragbar viel Zeit, es sei denn m und n sind sehr klein. Wir werden in Kapitel 15 sehen, wie dieses Problem mit einer allgemeinen Methode, die als dynamische Programmierung bekannt ist, gelöst werden kann.

- Gegeben sei ein mechanischer Bauplan in Form einer Bibliothek von Teilen, wobei jedes Teil aus Instanzen anderer Teile aufgebaut sein kann. Wir benötigen eine Liste der Teile, in der jedes Teil vor jedem anderen Teil, bei dem es gebraucht wird, aufgezählt wird. Wenn der Bauplan aus n Teilen besteht, dann gibt es $n!$ mögliche Reihenfolgen der Objekte, wobei $n!$ die Fakultät von n bezeichnet. Da die Fakultätsfunktion sogar noch schneller wächst als die Exponentialfunktion, können wir nicht jede Reihenfolge betrachten und für diese überprüfen, ob sie die angegebene Eigenschaft erfüllt (es sei denn, der Bauplan besteht nur aus sehr wenigen Teilen). Diese Problemstellung ist eine Instanz des Problems des topologischen Sortierens. Wir werden in Kapitel 22 sehen, wie dieses Problem effizient gelöst werden kann.

- Gegeben seien n Punkte in einer Ebene, und wir möchten die konvexe Hülle dieser Punkte finden. Die konvexe Hülle ist das kleinste konvexe Polygon, das die Punkte enthält. Anschaulich können wir uns jeden Punkt als einen Nagel vorstellen, der aus einer Tafel herausragt. Die konvexe Hülle würde durch ein enges Gummiband, das alle Nägel umschließt, dargestellt. Jeder Nagel, um den sich das Gummiband legt, ist eine Ecke der konvexen Hülle. (Ein Beispiel finden Sie in Abbildung 33.6 auf Seite 1040.) Jede der 2^n Teilmengen der Punkte kann die Menge der Ecken der konvexen Hülle darstellen. Zudem reicht das Wissen, welche Punkte Ecken der konvexen Hülle sind, nicht ganz aus, da wir auch wissen müssen, in welcher Reihenfolge die Ecken auf der Hülle vorkommen. Folglich gibt es einen großen Suchraum, in dem wir die Lösung, sprich die Ecken der konvexen Hülle, finden müssen. Kapitel 33 stellt zwei effiziente Methoden zur Berechnung der konvexen Hülle vor.

Diese Auflistung ist weit davon entfernt, vollständig zu sein (was Sie wahrscheinlich aufgrund des Gewichtes des Buches bereits vermutet haben), zeigt aber zwei Charakteristiken, die vielen interessanten algorithmischen Problemen gemeinsam sind.

1. Sie besitzen viele Lösungskandidaten; die überwiegende Mehrheit von ihnen lösen das vorliegende Problem jedoch nicht. Einen Lösungskandidaten zu finden, der dies tut, oder der der „beste“ ist, kann eine ziemliche Herausforderung darstellen.

2. Sie haben praktische Anwendungen. Aus der Liste der obigen Problemstellungen liefert das Problem der kürzesten Wege uns die einfachsten Beispiele. Ein Transportunternehmen, wie zum Beispiel eine Speditionsfirma oder eine Eisenbahngesellschaft, hat ein finanzielles Interesse daran, die kürzesten Wege innerhalb eines Straßen- oder Schienennetzes zu finden, da das Benutzen kürzerer Wege zu niedrigerem Arbeitsaufwand und niedrigeren Treibstoffkosten führt. Oder ein Routing-Knoten muss den kürzesten Weg durch das Netzwerk finden, um eine Meldung schnell weiterleiten zu können. Oder eine Person, die von New York nach Boston

fahren will, möchte eine Fahrroute von einer entsprechenden Webseite erhalten oder sie benutzt ihr GPS-Navigationsgerät während der Fahrt.

Nicht jedes Problem, das durch einen Algorithmus gelöst werden kann, besitzt eine leicht zu identifizierende Menge von Lösungskandidaten. Wir wollen ein Beispiel angeben. Nehmen Sie an, wir hätten eine Menge von numerischen Werten gegeben, die Abtastpunkte eines Signals darstellen, und wir wollen die diskrete Fourier-Transformierte dieser Punkte berechnen. Die diskrete Fourier-Transformation erlaubt die Transformation des Zeitbereiches in den Frequenzbereich, indem numerische Koeffizienten berechnet werden, mit denen wir die unterschiedlichen Frequenzen des abgetasteten Signals bestimmen können. Neben der Tatsache, dass die diskrete Fourier-Transformation eine zentrale Rolle in der Signalverarbeitung spielt, hat sie Anwendungen im Bereich der Datenkompression und bei der Multiplikation großer Polynome und ganzer Zahlen. Kapitel 30 stellt einen effizienten Algorithmus, die so genannte schnelle Fourier-Transformation (oft einfach nur FFT genannt) für dieses Problem vor. Zudem skizziert das Kapitel einen Entwurf einer Schaltung, mit der die FFT berechnet werden kann.

Datenstrukturen

Dieses Buch stellt auch verschiedene Datenstrukturen vor. Eine ***Datenstruktur*** ist eine Methode, Daten abzuspeichern und zu organisieren sowie den Zugriff auf die Daten und die Modifikation der Daten zu erleichtern. Keine Datenstruktur arbeitet für alle Zwecke gleich gut. Deshalb ist es wichtig, die Vorteile und Einschränkungen jeder einzelnen zu kennen.

Methodik

Obwohl es möglich ist, dieses Buch wie ein „Kochbuch“ für Algorithmen zu verwenden, könnten Sie eines Tages auf ein Problem stoßen, für das Sie nicht einfach einen veröffentlichten Algorithmus finden werden – die Übungsaufgaben und Problemstellungen, die Sie jeweils im Anschluss an die Abschnitte und Kapitel finden, sind zum Teil Beispiele hierfür. Dieses Buch wird Ihnen Methoden des Algorithmenentwurfes und der Analyse vermitteln, sodass Sie selbstständig Algorithmen entwickeln können, zeigen können, dass diese die korrekte Antwort berechnen, und deren Effizienz verstehen können. Verschiedene Kapitel zeigen unterschiedliche Aspekte des algorithmischen Problemlösens. Einige Kapitel behandeln spezielle Probleme, wie zum Beispiel Kapitel 9 die Berechnung von Medianen und Ranggrößen, Kapitel 23 die Berechnung minimaler Spannbäume und Kapitel 26 die Bestimmung maximaler Flüsse in Netzwerken. Andere Kapitel gehen auf Techniken ein, wie zum Beispiel Kapitel 4 auf Teile-und-Beherrsche, Kapitel 15 auf dynamische Programmierung und Kapitel 17 auf amortisierte Analyse.

Harte Probleme

Der größte Teil des Buches behandelt effiziente Algorithmen. Unser übliches Maß für Effizienz ist Geschwindigkeit, d. h. die Zeit, die der Algorithmus benötigt, um sein Ergebnis zu produzieren. Dennoch gibt es einige Problemstellungen, für die keine effiziente

Lösungsmethode bekannt ist. Kapitel 34 untersucht eine interessante Teilmenge dieser Probleme, die als NP-vollständig bekannt sind.

Weshalb sind NP-vollständige Probleme interessant? Erstens hat noch niemand bewiesen, dass ein effizienter Algorithmus für diese nicht existieren kann, obwohl bisher kein effizienter Algorithmus für ein NP-vollständiges Problem gefunden worden ist. Mit anderen Worten, keiner weiß, ob ein effizienter Algorithmus für ein NP-vollständiges Problem existiert oder nicht. Zweitens haben NP-vollständige Probleme die bemerkenswerte Eigenschaft, dass, wenn ein effizienter Algorithmus für eines von ihnen existiert, effiziente Algorithmen für alle von ihnen existieren. Die Verknüpfung unter den NP-vollständigen Problemen macht das Fehlen effizienter Lösungen um so spannender. Drittens sind verschiedene NP-vollständige Probleme ähnlich zu Problemen, für die wir effiziente Algorithmen tatsächlich kennen. Sie sind aber nicht identisch mit unseren NP-vollständigen Problemen. Informatiker sind fasziniert davon, dass eine kleine Änderung an der Problemstellung einen großen Unterschied in Bezug auf die Effizienz des jeweils besten bekannten Algorithmus zur Folge haben kann.

Sie sollten etwas über NP-vollständige Probleme wissen, da einige von ihnen überraschend häufig in realen Anwendungen auftreten. Wenn Sie dazu aufgefordert werden, einen effizienten Algorithmus für ein NP-vollständiges Problem zu finden, dann könnten Sie ansonsten viel Zeit mit einer ergebnislosen Suche verbringen. Wenn Sie zeigen können, dass das Problem NP-vollständig ist, dann können Sie stattdessen Ihre Zeit damit verbringen, einen effizienten Algorithmus zu entwickeln, der eine gute, wenn in der Regel auch nicht die bestmögliche Lösung liefert.

Als ein konkretes Beispiel betrachten wir einen Dienstleister für Kurierdienstleistungen mit einem zentralen Lager. Jeden Tag belädt er die Lieferwagen am Lager und schickt sie jeweils zu den verschiedenen Adressen, wo die entsprechenden Güter abzugeben sind. Am Ende des Tages müssen die Lieferwagen wieder zurück ins Lager kommen, damit sie für den nächsten Tag wieder beladen werden können. Um die Kosten zu reduzieren, möchte das Unternehmen für jeden der Lieferwagen diejenige Reihenfolge von Lieferstopps finden, die zu der geringsten von ihm insgesamt zurückzulegenden Entfernung führt. Diese Problemstellung ist das wohlbekannte Problem des „Handelsreisenden“ und ist NP-vollständig. Es ist kein effizienter Algorithmus dafür bekannt. Sind jedoch bestimmte Annahmen erfüllt, so kennen wir effiziente Algorithmen, die jeweils Touren mit einer Gesamtdistanz berechnen, die nicht allzu weit über der Distanz der kürzesten Tour liegt. Kapitel 35 diskutiert solche „Approximationsalgorithmen“.

Parallelität

Während vielen Jahren konnten wir darauf zählen, dass die Taktfrequenz der Prozessoren stetig anwächst. Physikalische Beschränkungen stellen aber eine elementare Hürde für eine ewig steigende Taktfrequenz dar: Da die Leistungsdichte superlinear mit der Taktfrequenz wächst, riskieren die Chips zu „schmelzen“, sobald die Taktfrequenz zu hoch ist. Um dennoch die Anzahl der Operationen pro Sekunde weiter zu steigern, werden Chips deshalb heutzutage so entworfen, dass sie nicht nur einen sondern mehrere Berechnungskerne enthalten. Wir können diese Mehrkern-Rechner (engl.: *multicore computer*) ansehen als ob mehrere sequentielle Rechner auf einem Chip integriert wä-

ren, d. h. sie stellen eine Art „Parallelrechner“ dar. Um diesen Mehrkern-Rechnern die bestmögliche Performanz zu entlocken, müssen wir die vorhandene Parallelität im Auge behalten, wenn wir Algorithmen entwerfen. Kapitel 27 stellt ein Modell für „mehrfädige“ Algorithmen vor, die mehrere Kerne ausnutzen können. Dieses Modell ist aus theoretischer Sicht von Nutzen und stellt die Basis für einige erfolgreiche Computerprogramme, insbesondere einem preisgekrönten Schachprogramm.

Übungen

1.1-1 Geben Sie ein Beispiel aus der Praxis an, in dem sortiert werden muss oder in dem eine konvexe Hülle berechnet werden muss.

1.1-2 Welche anderen Maße könnte man außer der Geschwindigkeit in einem realistischen Szenario für die Effizienz verwenden?

1.1-3 Wählen Sie eine Datenstruktur aus, der Sie schon begegnet sind, und diskutieren Sie deren Stärken und Schwächen.

1.1-4 Worin ähneln sich die oben erwähnten Probleme der kürzesten Wege und des Handelsreisenden? Worin unterscheiden sie sich?

1.1-5 Lassen Sie sich ein realistisches Problem einfallen, für das nur die beste Lösung brauchbar ist. Benennen Sie anschließend ein Problem, bei dem die „annähernd“ beste Lösung als Ergebnis bereits ausreicht.

1.2 Algorithmen als Technologie

Angenommen, Rechner wären unendlich schnell und Speicher gäbe es umsonst. Gäbe es irgendeinen Grund, Algorithmen zu untersuchen? Die Antwort lautet ja. Wenn aus keinem anderen Grund, dann deshalb, weil Sie immer noch daran interessiert sind zu zeigen, dass Ihr Lösungsverfahren terminiert und es dies mit der korrekten Antwort tut.

Wenn Rechner unendlich schnell wären, wäre jedes korrekte Verfahren zur Lösung eines Problems okay. Sie würden zwar wahrscheinlich verlangen, dass Ihre Implementierung softwaretechnischen Kriterien genügt (beispielsweise sollte Ihre Implementierung gut entworfen und dokumentiert sein), Sie würden aber in der Regel das am einfachsten zu implementierende Verfahren einsetzen.

Natürlich mögen Rechner schnell sein, aber sie sind nicht unendlich schnell. Und Speicher mag billig sein, aber er ist nicht kostenlos. Rechenzeit ist deshalb eine beschränkte Ressource wie auch der Speicherplatz. Sie sollten diese Ressourcen mit Vernunft einsetzen. Algorithmen, die effizient im Sinne von Zeit oder Platzbedarf sind, werden Ihnen dabei helfen.

Effizienz

Verschiedene Algorithmen, die zur Lösung ein und desselben Problems entwickelt wurden, können sich häufig dramatisch in ihrer Effizienz unterscheiden. Diese Unterschiede können viel größer sein als die durch Hardware und Software bedingten Unterschiede.

Als Beispiel dafür werden wir in Kapitel 2 zwei Sortieralgorithmen betrachten. Der erste, der unter dem Namen ***Sortieren durch Einfügen*** (engl.: *insertion sort*) bekannt ist, benötigt ungefähr die Zeit $c_1 n^2$, um n Elemente zu sortieren, wobei c_1 eine von n unabhängige Konstante ist. Das heißt, er benötigt eine Zeit, die ungefähr proportional zu n^2 ist. Der zweite, ***Sortieren durch Mischen*** (engl.: *merge sort*), benötigt ungefähr die Zeit $c_2 n \lg n$, wobei $\lg n$ für $\log_2 n$ steht und c_2 ebenfalls eine von n unabhängige Konstante ist. Sortieren durch Einfügen hat normalerweise einen kleineren konstanten Faktor als Sortieren durch Mischen, sodass $c_1 < c_2$ gilt. Wir werden sehen, dass die konstanten Faktoren aber einen weit geringeren Einfluss auf die Laufzeit haben als die Abhängigkeit von der Eingabegröße n. Lassen Sie uns hierfür die Laufzeit von Sortieren durch Einfügen umschreiben zu $c_1 n \cdot n$ und die Laufzeit von Sortieren durch Mischen zu $c_2 n \cdot \lg n$. Dann sehen wir, dass die Laufzeit von Sortieren durch Einfügen einen Faktor von n enthält und die Laufzeit von Sortieren durch Mischen nur einen Faktor von $\lg n$, was viel kleiner ist. (Ist beispielsweise $n = 1000$, dann ist $\lg n$ ungefähr 10, und wenn n gleich einer Million ist, ist $\lg n$ ungefähr nur 20.) Obwohl Sortieren durch Einfügen für kleine Eingabegrößen gewöhnlich schneller ist als Sortieren durch Mischen, wird der auf dem Verhältnis zwischen $\lg n$ und n beruhende Vorteil von Sortieren durch Mischen den Unterschied im konstanten Faktor mehr als kompensieren, wenn die Eingabegröße n groß genug ist. Ganz gleich wieviel kleiner c_1 gegenüber c_2 ist, es wird immer eine Schwelle geben, ab dem Sortieren durch Mischen schneller ist.

Um ein konkretes Beispiel anzugeben, lassen wir einen schnellen Rechner (Rechner A), auf dem Sortieren durch Einfügen läuft, gegen einen langsamen Rechner (Rechner B), auf dem Sortieren durch Mischen läuft, antreten. Jeder der beiden hat ein Feld von 10 Millionen Zahlen zu sortieren. (Auch wenn sich 10 Millionen Zahlen nach viel anhören, das Feld belegt bei 64-Bit Zahlen ungefähr 80 Megabytes, was in den Hauptspeicher sogar eines einfachen Laptops passt.) Nehmen Sie an, dass Rechner A zehn Milliarden Anweisungen pro Sekunde (was weit mehr ist als ein einzelner sequentieller Rechner zum Zeitpunkt der Erstellung dieses Buches leisten konnte) und Rechner B nur zehn Millionen Anweisungen pro Sekunde ausführt, sodass Rechner A von der Leistungsfähigkeit her 1000-mal schneller als Rechner B ist. Um den Unterschied noch drastischer zu gestalten, nehmen wir an, dass auf Rechner A die weltweit pfiffigsten Programmierer Sortieren durch Einfügen in Maschinensprache kodiert haben und der entstandene Code $2\,n^2$ Anweisungen ausführen muss, um n Zahlen zu sortieren. Wir nehmen desweiteren an, dass für Rechner B Sortieren durch Mischen von einem durchschnittlichen Programmierer in einer höheren Programmiersprache mit Hilfe eines ineffizienten Compilers programmiert wurde, wobei der entstandene Code $50\,n \lg n$ Anweisungen benötigt. Um zehn Millionen Zahlen zu sortieren, benötigt Rechner A also

$$\frac{2 \cdot (10^7)^2 \text{Anweisungen}}{10^{10} \text{ Anweisungen/Sekunde}} = 20.000 \text{ Sekunden (mehr als } 5\tfrac{1}{2} \text{ Stunden)} ,$$

während Rechner B

$$\frac{50 \cdot 10^7 \lg 10^7 \text{ Anweisungen}}{10^7 \text{ Anweisungen/Sekunde}} \approx 1163 \text{ Sekunden (weniger als 20 Minuten)}$$

benötigt. Unter Verwendung eines Algorithmus, dessen Laufzeit langsamer anwächst, läuft Rechner B sogar mit einem schwachen Compiler mehr als 17-mal schneller als

Rechner A! Der Vorteil von Sortieren durch Mischen wird sogar noch deutlicher, wenn wir 100 Millionen Zahlen sortieren: Während Sortieren durch Einfügen ungefähr 23 Tage benötigt, braucht Sortieren durch Mischen weniger als vier Stunden. Im Allgemeinen gilt, dass in dem Maße, in dem der Umfang des Problems anwächst, der relative Vorteil von Sortieren durch Mischen größer wird.

Algorithmen und andere Technologien

Das oben genannte Beispiel zeigt, dass wir Algorithmen, ähnlich wie Computer-Hardware, als ***Technologie*** verstehen sollten. Die Gesamtleistung des Systems hängt in gleicher Weise von der Wahl eines effizienten Algorithmus wie von der Wahl schneller Hardware ab. Genauso schnell wie Fortschritte in anderen Computertechnologien gemacht werden, werden sie auch auf dem Gebiet der Algorithmen erzielt.

Sie werden sich fragen, ob Algorithmen wirklich so wesentlich für heutige Rechner sind, angesichts anderer fortgeschrittener Technologien wie

- moderne Rechnerarchitekturen und Fabrikationstechnologien,
- einfach zu bedienende, intuitive grafische Benutzeroberflächen (GUIs),
- objektorientierte Systeme
- integrierte Webtechnologien und
- schnelle Netzwerke, sowohl Kabel- als auch Funkbasierte.

Die Antwort lautet ja, wenngleich einige Anwendungen auf der Anwendungsebene keiner komplexen Algorithmen bedürfen (wie zum Beispiel einige einfache internetbasierte Anwendungen). Die meisten erfordern zu ihrer Realisierung jedoch ein gewisses Maß an algorithmischem Gehalt. Betrachten wir zum Beispiel einen internetbasierten Dienst, der bestimmt, wie man von einem Ort zu einem anderen gelangt. Dessen Implementierung würde auf schneller Hardware, einer grafischen Benutzeroberfläche, wide-area Vernetzung und möglicherweise Objektorientierung aufbauen. Für bestimmte Operationen müsste der Dienst auch Algorithmen nutzen. Dies könnten Algorithmen zum Finden von Routen (unter Verwendung kürzester-Pfad-Algorithmen), zur Wiedergabe von Karten oder zur Interpolation von Adressen sein.

Darüber hinaus ist sogar eine Anwendung, die auf der Anwendungsebene keine algorithmischen Inhalte erfordert, stark auf Algorithmen angewiesen. Ist die Anwendung von schneller Hardware abhängig? Bei der Hardwareentwicklung werden Algorithmen benutzt. Ist die Anwendung auf eine grafische Benutzeroberfläche angewiesen? Der Entwurf jeder grafischen Benutzeroberfläche stützt sich auf Algorithmen. Ist die Anwendung auf Vernetzung angewiesen? Das Routing in Netzwerken basiert stark auf Algorithmen. Wurde die Anwendung in einer anderen Programmiersprache als in Maschinencode geschrieben? Dann wurde sie durch einen Compiler, Interpreter oder Assembler bearbeitet, die alle umfangreichen Gebrauch von Algorithmen machen. Algorithmen stehen im Zentrum der meisten in modernen Rechnern verwendeten Technologien.

Des Weiteren benutzen wir die Rechner mit deren ständig anwachsender Kapazität, um immer größere Probleme zu lösen als jemals zuvor. Wie wir in dem vorangegangenen Vergleich zwischen Sortieren durch Einfügen und Sortieren durch Mischen gesehen haben, ist es bei größerem Problemumfang so, dass die Effizienzunterschiede zwischen Algorithmen besonders markant werden können.

Eine Eigenschaft, die die wirklich qualifizierten Programmierer von Anfängern unterscheidet, ist der Besitz einer soliden Basis von algorithmischem Fachwissen und Methoden. Mit moderner Rechentechnologie kann man einige Aufgaben ausführen, ohne viel über Algorithmen zu wissen; mit einem guten Hintergrund auf dem Gebiet der Algorithmen können Sie jedoch viel, viel mehr tun.

Übungen

1.2-1 Geben Sie ein Beispiel für eine Anwendung an, die algorithmischen Gehalt auf der Anwendungsebene erfordert, und diskutieren Sie die Funktion der eingesetzten Algorithmen.

1.2-2 Nehmen Sie an, dass wir die Implementierung von Sortieren durch Einfügen und Sortieren durch Mischen auf demselben Rechner vergleichen. Für Eingaben der Größe n benötigt Sortieren durch Einfügen $8\,n^2$ Schritte, während Sortieren durch Mischen $64\,n \lg n$ Schritte benötigt. Für welche Werte von n wird Sortieren durch Einfügen schneller als Sortieren durch Mischen sein?

1.2-3 Welcher ist der kleinste Wert von n, für den ein Algorithmus, dessen Laufzeit $100\,n^2$ ist, auf demselben Rechner schneller läuft als ein Algorithmus, dessen Laufzeit 2^n beträgt.

Problemstellungen

1-1 Vergleich von Laufzeiten

Bestimmen Sie für jede Funktion $f(n)$ und jede Zeit t den größten Wert von n für ein Problem, das innerhalb der Zeit t gelöst werden kann, wenn der Algorithmus $f(n)$ Mikrosekunden benötigt, um das Problem zu lösen.

	1 Sekunde	1 Minute	1 Stunde	1 Tag	1 Monat	1 Jahr	1 Jahrhundert
$\lg n$							
$\sqrt{n}$							
n							
$n \lg n$							
n^2							
n^3							
2^n							
$n!$							

Kapitelbemerkungen

Es gibt viele ausgezeichnete Lehrbücher über allgemeine Fragen zu Algorithmen, einschließlich denen von Aho, Hopcroft, and Ullman [5, 6], Baase and Van Gelder [28], Brassard and Bratley [54], Dasgupta, Papadimitriou, and Vazirani [82], Goodrich and Tamassia [148], Hofri [175], Horowitz, Sahni, and Rajasekaran [181], Johnsonbaugh and Schaefer [193], Kingston [205], Kleinberg and Tardos [208], Knuth [209, 210, 211], Kozen [220], Levitin [235], Manber [242], Mehlhorn [249, 250, 251], Purdom and Brown [287], Reingold, Nievergelt, and Deo [293], Sedgewick [306], Sedgewick and Flajolet [307], Skiena [318], und Wilf [356]. Einige der eher praktischen Aspekte des Algorithmenentwurfs werden von Bentley [42, 43] und Gonnet [145] diskutiert. Einen Abriss über das Gebiet der Algorithmen findet man auch im Handbook of Theoretical Computer Science, Band A [342] und im CRC Handbook on Algorithms and Theory of Computation [25]. Einen Überblick über Algorithmen, die in der Bioinformatik benutzt werden, findet man in den Lehrbüchern von Gusfield [156], Pevzner [275], Setubal und Meidanis [310] und Waterman [350].

2 Ein einführendes Beispiel

Dieses Kapitel wird Sie mit der Methodik vertraut machen, die wir in diesem Buch benutzen werden, um über den Entwurf und die Analyse von Algorithmen nachzudenken. Das Kapitel ist in sich geschlossen, enthält allerdings verschiedene Verweise auf den Stoff, der in Kapitel 3 und 4 eingeführt wird. (Es enthält auch einige Summenformeln, auf deren Berechnung im Anhang A eingegangen wird.)

Wir beginnen mit einer genaueren Betrachtung des in Kapitel 1 bereits eingeführten Algorithmus Sortieren durch Einfügen (engl.: *insertion sort*), der das Sortierproblem löst. Wir führen einen „Pseudocode" ein, der Ihnen vertraut sein sollte, wenn Sie bereits programmiert haben. Alle unsere Algorithmen werden wir über solchen Pseudocode beschreiben. Nachdem wir den Algorithmus zum Sortieren durch Einfügen beschrieben haben, diskutieren wir, warum er korrekt sortiert, und analysieren seine Laufzeit. Die Analyse führt eine Notation ein, die sich darauf bezieht, wie die Zeit mit der Anzahl der zu sortierenden Elemente anwächst. Nach der Diskussion von Sortieren durch Einfügen führen wir die Teile-und-Beherrsche-Methode (engl.: *divide and conquer*) für den Entwurf von Algorithmen ein und wenden diese Methode an, um einen Algorithmus zu entwickeln, den wir Sortieren durch Mischen (engl.: *merge sort*) nennen. Wir schließen mit der Analyse der Laufzeit von Sortieren durch Mischen.

2.1 Sortieren durch Einfügen

Unser erster Algorithmus, Sortieren durch Einfügen, löst das in Kapitel 1 eingeführte ***Sortierproblem***:

Eingabe: Eine Folge von n Zahlen $\langle a_1, a_2, \ldots, a_n \rangle$.

Ausgabe: Eine Permutation (Umordnung) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ der Eingabefolge, sodass $a'_1 \le a'_2 \le \cdots \le a'_n$ gilt.

Die Zahlen, die wir sortieren wollen, werden auch als ***Schlüssel*** bezeichnet. Auch wenn wir das Sortierproblem in Bezug auf eine zu sortierende Folge begreifen, die Eingabe erfolgt in Form eines Feldes, das aus n Elementen besteht.

In diesem Buch werden wir Algorithmen üblicherweise als Programme darstellen, die in einem ***Pseudocode*** geschrieben sind, der C, C++, Java, Python oder Pascal in vielerlei Hinsicht ähnelt. Wenn Sie mit einer dieser Sprachen vertraut sind, dann sollten Sie wenig Probleme beim Lesen unserer Algorithmen haben. Was den Pseudocode von „echtem" Code unterscheidet, ist, dass wir im Pseudocode diejenige Ausdrucksweise

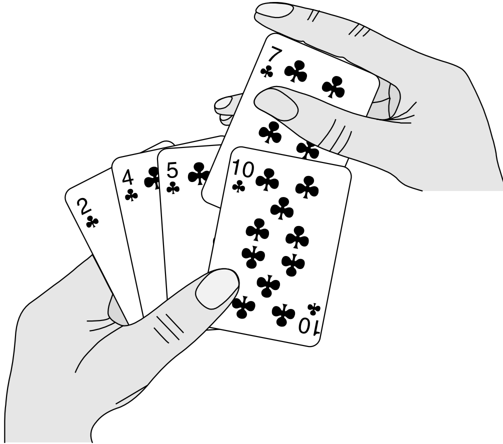


Abbildung 2.1: *Sortieren von Karten auf einer Hand unter Verwendung von Sortieren durch Einfügen.*

verwenden, die einen gegebenen Algorithmus am deutlichsten und prägnantesten spezifiziert. Manchmal ist die natürliche Sprache am anschaulichsten; seien Sie deshalb nicht überrascht, wenn Ihnen eine deutsche Wortgruppe oder ein Satz inmitten von „echtem“ Code begegnet. Ein weiterer Unterschied zwischen Pseudocode und echtem Code liegt darin, dass der Pseudocode sich üblicherweise nicht mit Aufgaben der Softwaretechnik befasst. Aufgaben der Datenabstraktion, der Modularität und der Fehlerbehandlung werden häufig ignoriert, um das Wesentliche des Algorithmus prägnanter zu vermitteln.

Wir beginnen mit ***Sortieren durch Einfügen***, einem Algorithmus, der für das Sortieren einer kleinen Anzahl von Elementen effizient ist. Sortieren durch Einfügen arbeitet auf die Art und Weise, wie viele Menschen Spielkarten in einer Hand sortieren. Wir beginnen mit einer leeren linken Hand und die Karten liegen umgedreht auf dem Tisch. Dann nehmen wir pro Zeiteinheit eine Karte vom Tisch und fügen sie an der richtigen Position in der linken Hand ein. Um die korrekte Position für eine Karte zu finden, vergleichen wir sie von rechts nach links mit jeder bereits auf der Hand befindlichen Karte, wie in Abbildung 2.1 dargestellt. Zu jeder Zeit sind die Karten auf der Hand sortiert, wobei diese Karten ursprünglich die obersten Karten des Haufens auf dem Tisch waren.

```
INSERTION-SORT(A)
1  for j = 2 to A.länge
2      schlüssel = A[j]
3      // Füge A[j] in die sortierte Sequenz A[1..j − 1] ein.
4      i = j − 1
5      while i > 0 und A[i] > schlüssel
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = schlüssel
```

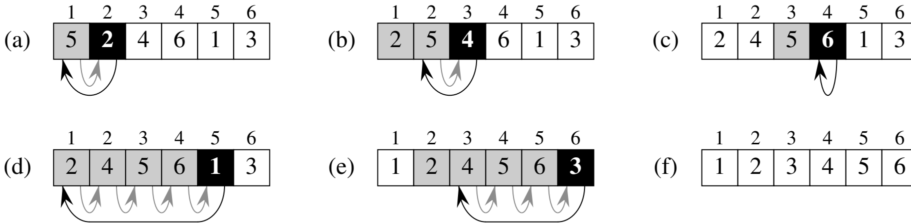


Abbildung 2.2: *Die Arbeitsweise von* Insertion-Sort *auf dem Feld* $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. *Die Feldindizes sind oberhalb der Rechtecke und die an den Positionen gespeicherten Werte innerhalb der Rechtecke vermerkt.* ***(a)–(e)*** *Die Iteration der* ***for****-Schleife in den Zeilen 1–8. Bei jeder Iteration enthält das schwarze Rechteck den aus* $A[j]$ *entnommenen Schlüssel, der im Test in Zeile 5 mit den Werten verglichen wird, die in den schattierten Rechtecken links davon stehen. Schattierte Pfeile zeigen Feldvariablen, die in Zeile 6 eine Stelle nach rechts verschoben werden und schwarze Pfeile kennzeichnen, wohin der Schlüssel in Zeile 8 verschoben wird.* ***(f)*** *Das endgültig sortierte Feld.*

Wir stellen den Pseudocode für Sortieren durch Einfügen als eine Prozedur mit dem Namen Insertion-Sort dar, die als Eingabeparameter ein Feld $A[1 \mathinner{..} n]$ erhält, das die Folge der n zu sortierenden Elementen enthält. (Im Code wird die Zahl n der Elemente in A mit $A.\mathit{länge}$ bezeichnet.) Der Algorithmus sortiert die eingegebenen Elemente ***in-place***, d. h. er ordnet die Elemente innerhalb des Feldes A um, wobei zu jedem Zeitpunkt höchstens eine konstante Anzahl von Elementen außerhalb des Feldes gespeichert werden. Das Eingabefeld A enthält die sortierte Ausgabefolge, wenn Insertion-Sort beendet ist.

Schleifeninvarianten und die Korrektheit von Sortieren durch Einfügen

Abbildung 2.2 zeigt, wie dieser Algorithmus angewendet auf $A = \langle 5, 2, 4, 6, 1, 3 \rangle$ arbeitet. Der Index j kennzeichnet die „aktuelle Karte“, die gerade in das Blatt einsortiert wird. Zu Beginn jeder Iteration der **for**-Schleife, die über die Laufvariable j gesteuert wird, bildet das aus den Elementen $A[1 \mathinner{..} j-1]$ bestehende Teilfeld das gegenwärtige sortierte Blatt, und die Elemente $A[j+1 \mathinner{..} n]$ entsprechen dem sich noch auf dem Tisch befindlichen Kartenhaufen. Tatsächlich sind die Elemente $A[1 \mathinner{..} j-1]$ auch diejenigen Elemente, die sich *ursprünglich* an den Positionen 1 bis $j-1$ befunden haben, sind aber nun in geordneter Reihenfolge. Wir bezeichnen diese Eigenschaften von $A[1 \mathinner{..} j-1]$ formal als ***Schleifeninvariante***:

> Zu Beginn jeder Iteration der **for**-Schleife in den Zeilen 1–8 besteht das Teilfeld $A[1 \mathinner{..} j-1]$ aus den ursprünglich in $A[1 \mathinner{..} j-1]$ enthaltenen Elementen, allerdings in geordneter Reihenfolge.

Wir benutzen Schleifeninvarianten als Hilfsmittel, um zu beweisen, dass ein Algorithmus korrekt ist. Wir müssen zeigen, dass eine Schleifeninvariante drei Dinge erfüllt:

Initialisierung: Sie ist vor der Iteration der Schleife wahr.

Fortsetzung: Wenn sie vor der Iteration einer Schleife wahr ist, dann bleibt sie auch bis zum Beginn der nächsten Iteration wahr.

Terminierung: Wenn die Schleife abbricht, dann liefert uns die Invariante eine zweckdienliche Eigenschaft, die uns hilft zu zeigen, dass der Algorithmus korrekt ist.

Wenn die ersten beiden Eigenschaften erfüllt sind, dann ist die Schleifeninvariante vor jeder Iteration der Schleife wahr. (Natürlich dürfen Sie auch andere begründete Sachverhalte als die Schleifeninvariante selbst benutzen, um zu beweisen, dass die Schleifeninvariante vor jeder Iteration erfüllt ist.) Beachten Sie die Ähnlichkeit mit der mathematischen Induktion, bei der Sie einen Induktionsanfang und einen Induktionsschritt beweisen, um zu zeigen, dass eine Eigenschaft erfüllt ist. Der Nachweis, dass die Invariante vor der ersten Iteration erfüllt ist, entspricht hier dem Induktionsanfang und der Nachweis, dass die Invariante von Iteration zu Iteration erhalten bleibt, dem Induktionsschritt.

Die dritte Eigenschaft ist die wahrscheinlich wichtigste, da wir die Schleifeninvariante benutzen, um die Korrektheit des untersuchten Algorithmus zu zeigen. Üblicherweise wenden wir die Schleifeninvariante in Verbindung mit der Bedingung an, unter der die Schleife beendet wird. Eine solche Terminierungseigenschaft findet man in der üblichen Verwendung der mathematischen Induktion, in der der Induktionsschritt unendlich oft angewendet wird, nicht vor; hier beenden wir die „Induktion“, wenn die Schleife abbricht.

Sehen wir uns an, ob die oben formulierte Schleifenvariante beim Sortieren durch Einfügen erfüllt ist.

Initialisierung Wir zeigen zunächst, dass die Schleifeninvariante vor der ersten Iteration, in der die Laufvariable $j = 2$ ist, gilt.[1] Das Teilfeld $A[1\,..\,j-1]$ besteht zu diesem Zeitpunkt nur aus dem einzigen Element $A[1]$, das tatsächlich das ursprüngliche Element in $A[1]$ ist. Zudem ist dieses Teilfeld sortiert (trivialerweise natürlich), was zeigt, dass die Schleifeninvariante vor der ersten Iteration der Schleife gilt.

Fortsetzung Als nächstes schauen wir uns die zweite Eigenschaft an. Es ist zu zeigen, dass die Schleifeninvariante bei jeder Iteration erhalten bleibt. Informal formuliert, arbeitet der Ausführungsblock der **for**-Schleife so, dass $A[j-1]$, $A[j-2]$, $A[j-3]$ und so weiter jeweils um eine Stelle nach rechts verschoben werden, bis die richtige Position von $A[j]$ gefunden ist (Zeilen 4–7) und der Wert von $A[j]$ dann an dieser Stelle eingefügt wird (Zeile 8). Das Teilfeld $A[1\,..\,j]$ besteht dann aus

[1] Wenn die Schleife eine **for**-Schleife ist, dann ist der Zeitpunkt, an dem wir die Schleifeninvariante vor der ersten Iteration testen, sofort nach der Anfangszuweisung an die Laufvariable und genau vor dem ersten Test im Schleifenkopf. Im Falle von INSERTION-SORT ist dies, nachdem j der Wert 2 zugewiesen wurde und vor dem ersten Test, ob $j \leq A.\mathit{länge}$ gilt.

Elementen, die ursprünglich in $A[1\,..\,j]$ waren, jetzt aber in geordneter Reihenfolge. Das Inkrementieren von j für die nächste Iteration der **for**-Schleife erhält die Schleifeninvariante.

Eine formalere Behandlung der zweiten Eigenschaft würde erfordern, eine Schleifeninvariante für die **while**-Schleife in Zeile 5–7 anzugeben und zu beweisen. An diesem Punkt ziehen wir es jedoch vor, uns nicht in solch einem Formalismus zu verlieren. Wir verlassen uns auf unsere informale Analyse, um zu zeigen, dass die zweite Eigenschaft für die äußere Schleife gilt.

Terminierung Zum Abschluss untersuchen wir, was passiert, wenn die Schleife abbricht. Die **for**-Schleife in unserem Algorithmus bricht ab, wenn $j > A.\mathit{länge} = n$ gilt. Da jede Schleifeniteration j um 1 erhöht, müssen wir zu diesem Zeitpunkt $j = n+1$ haben. Setzen wir $n+1$ für j innerhalb der Formulierung der Schleifeninvariante ein, dann erhalten wir, dass das Teilfeld $A[1\,..\,n]$ aus den ursprünglich in $A[1\,..\,n]$ enthaltenen Elementen besteht, allerdings in geordneter Reihenfolge. Das Teilfeld $A[1\,..\,n]$ ist aber bereits das gesamte Feld! Somit können wir schließen, dass das gesamte Feld sortiert ist. Der Algorithmus arbeitet also korrekt.

Wir werden diesen Ansatz der Schleifeninvarianten später im Kapitel und auch in anderen Kapiteln verwenden, um die Korrektheit von Algorithmen zu zeigen.

Pseudocode-Konventionen

Wir verwenden in unserem Pseudocode die folgenden Konventionen.

- Einrücken kennzeichnet die Blockstruktur. Zum Beispiel besteht der Schleifenrumpf der **for**-Schleife, der in Zeile 1 beginnt, aus den Zeilen 2–8, und der Schleifenrumpf der **while**-Schleife, der in Zeile 5 beginnt, enthält die Zeilen 6–7, aber nicht die Zeile 8. Diesen Stil wenden wir auch bei **if-else**-Anweisungen[2] an. Das Verwenden von Einrückungen anstelle von konventionellen Markierungen der Blockstruktur, wie **begin-** und **end**-Anweisungen, bewahrt die Übersichtlichkeit der Programme, wenn sie sie nicht sogar erhöht.[3]

- Die Schleifenkonstrukte **while**, **for** und **repeat-until** und das Konstrukt **if-else** zur bedingten Abfrage haben Interpretationen, die denen in C, C++, Java, Python und Pascal ähneln.[4] In diesem Buch behält die Laufvariable ihren Wert nach Verlassen der Schleife, im Unterschied zu C++, Java und Pascal. Deshalb ist der Wert der Laufvariablen unmittelbar nach der **for**-Schleife derjenige Wert,

[2] In einer **if-else**-Anweisung, rücken wir **else** an die gleiche Stelle ein wie das zugehörige **if**. Auch wenn wir das Schlüsselwort **then** weglassen, sprechen wir bisweilen dennoch vom ***then-Fall***, um auf das Codestück zu verweisen, das ausgeführt wird, wenn die Bedingung der **if**-Anweisung wahr ist. Bei einer Vielfachabfrage, verwenden wir **elseif**, um zu der nächsten Abfrage zu gelangen.

[3] Alle Pseudocode-Prozeduren in diesem Buch erscheinen jeweils auf einer Seite ohne Seitenumbruch, sodass Sie nicht die Einrücktiefen in Code, der über mehrere Seiten geht, zu bestimmen haben.

[4] Die meisten blockstrukturierten Sprachen besitzen äquivalente Konstrukte, auch wenn sich die genaue Syntax unterscheiden kann. In Python fehlen **repeat-until**-Schleifen und die **for**-Schleifen arbeiten ein bisschen anders als die **for**-Schleifen in diesem Buch.

der zuerst die Schranke der **for**-Schleife verletzt hat. Wir haben diese Eigenschaft im Korrektheitsbeweis von Sortieren durch Einfügen benutzt. Der Kopf der **for**-Schleife in Zeile 1 lautet **for** $j = 2$ **to** $A.\mathit{länge}$ und so gilt $j = A.\mathit{länge} + 1$ (d. h. $j = n + 1$, wegen $n = A.\mathit{länge}$), wenn die Schleife abbricht. Wir verwenden das Schlüsselwort **to**, falls die **for**-Schleife ihre Laufvariable in jeder Iteration inkrementiert, und das Schlüsselwort **downto**, falls die **for**-Schleife ihre Laufvariable in jeder Iteration dekrementiert. Falls die Laufvariable um einen Betrag größer als 1 verändert werden soll, wird die entsprechende Schrittweite nach dem Schlüsselwort **by** angegeben.

- Das Symbol „//" zeigt an, dass der Rest der Zeile ein Kommentar ist.

- Eine Mehrfachzuweisung der Form $i = j = e$ übergibt den beiden Variablen i und j den Wert des Ausdruckes e; sie sollte als äquivalent zu der Zuweisung $j = e$ gefolgt von der Zuweisung $i = j$ betrachtet werden.

- Variablen (wie zum Beispiel i, j und *schlüssel*) gelten als lokal innerhalb der gegebenen Prozedur. Wir werden ohne eine jeweilige explizite Erwähnung keine globalen Variablen verwenden.

- Wir greifen auf Elemente eines Feldes zu, indem der Name des Feldes, gefolgt vom Index, der in eckigen Klammern steht, angegeben wird. Zum Beispiel bezeichnet $A[i]$ das i-te Element des Feldes A. Die Bezeichnung „. ." wird verwendet, um einen Bereich von Werten innerhalb eines Feldes zu kennzeichnen. So bezeichnet $A[1\,..\,j]$ ein Teilfeld von A, das aus den j Elementen $A[1], A[2], \ldots, A[j]$ besteht.

- Normalerweise organisieren wir zusammenhängende Daten in ***Objekten***, die sich aus ***Attributen*** zusammensetzen. Wir greifen auf ein spezielles Attribut eines Objektes zu, indem wir die Syntax benutzen, die in vielen objektorientierten Programmiersprachen zu finden ist: Objektname gefolgt von einem Punkt und dem Attributnamen. So betrachten wir ein Feld als ein Objekt mit dem Attribut *länge*, das die Anzahl der Elemente angibt, die in dem Feld enthalten sind. Um die Anzahl der Elemente eines Feldes A anzugeben, schreiben wir $A.\mathit{länge}$.

  Wir behandeln eine Variable, die ein Feld oder ein Objekt darstellt, als Zeiger auf die sie repräsentierenden Daten. Für jedes Attribut f eines Objektes x bewirkt die Zuweisung $y = x$, dass anschließend $y.f$ gleich $x.f$ ist. Wenn wir dann $x.f = 3$ setzen, dann gilt anschließend nicht nur $x.f = 3$, sondern auch $y.f = 3$. Mit anderen Worten, x und y zeigen nach der Zuweisung $y = x$ auf das gleiche Objekt.

  Unsere Notation für Attribute können wir „hintereinander schalten". Nehmen Sie beispielsweise an, dass das Attribut f selbst ein Zeiger auf ein Objekt ist, das ein Attribut g besitzt. Dann wird die Notation $x.f.g$ implizit geklammert als $(x.f).g$. Anders formuliert, wenn $y = x.f$ gilt, dann ist $x.f.g$ das gleiche wie $y.g$.

  Zuweilen wird ein Zeiger auf überhaupt kein Objekt verweisen. In diesem Fall weisen wir ihm den speziellen Wert NIL zu.

- Parameter werden einer Prozedur als Werte übergeben („***call by value***"): Die aufgerufene Prozedur erhält ihre eigene Kopie der Parameter, und falls sie einem Parameter einen Wert zuweist, dann wird diese Veränderung von der aufrufenden

Prozedur *nicht* wahrgenommen. Falls Objekte übergeben werden, wird der Zeiger auf die das Objekt repräsentierenden Daten kopiert, die Attribute des Objektes allerdings nicht. Wenn zum Beispiel x ein Parameter einer aufgerufenen Prozedur ist, dann ist die Zuweisung $x = y$ innerhalb der aufgerufenen Prozedur für die aufrufende Prozedur selbst nicht sichtbar. Die Zuweisung $x.f = 3$ ist jedoch sichtbar. Gleichermaßen werden Felder als Zeiger übergeben, sodass nur ein Zeiger und nicht das gesamte Feld übergeben wird. Änderungen an einzelnen Elementen des Feldes sind für die aufrufende Prozedur sichtbar.

- Eine **return**-Anweisung übergibt unmittelbar die Kontrolle zurück an die aufrufende Prozedur, die an der Stelle des Prozeduraufrufs weiterarbeitet. Die meisten **return**-Anweisungen erhalten einen Wert, den sie an die aufrufende Prozedur zurückgeben. Unser Pseudocode unterscheidet sich von vielen Programmiersprachen dahingehend, dass wir erlauben, dass mehrere Werte durch eine einzige **return**-Anweisung zurückgegeben werden können.

- Die Booleschen Operatoren "und" und "oder" sind ***träge*** Operatoren. Das heißt, wenn wir den Ausdruck "x und y" bestimmen, dann werten wir zuerst x aus. Wenn x den Wert FALSCH annimmt, dann kann der gesamte Ausdruck auch nicht mehr WAHR sein, weshalb wir y gar nicht erst auswerten. Wenn x andererseits den Wert WAHR annimmt, dann müssen wir y auswerten, um den Wert für den gesamten Ausdruck zu bestimmen. Analog dazu werten wir bei dem Ausdruck "x oder y" den Ausdruck y nur dann aus, wenn x den Wert FALSCH annimmt. Träge Operatoren erlauben es uns, Boolesche Ausdrücke wie „$x \neq$ NIL und $x.f = y$“ zu schreiben, ohne uns darüber Gedanken machen zu müssen, was passiert, wenn wir versuchen, $x.f$ auszuwerten, wenn x NIL ist.

- Das Schlüsselwort **error** gibt an, dass ein Fehler aufgetreten ist, weil Bedingungen einer Prozedur, die aufgerufen worden ist, verletzt worden sind. Die aufrufende Prozedur ist für die Fehlerbehandlung verantwortlich, sodass wir nicht anzugeben brauchen, welche Maßnahmen im Falle eines Fehlers getroffen werden müssen.

Übungen

2.1-1 Zeigen Sie analog zu Abbildung 2.2 die Arbeitsweise von INSERTION-SORT, wenn die Prozedur auf das Feld $A = \langle 31, 41, 59, 26, 41, 58 \rangle$ angewendet wird.

2.1-2 Schreiben Sie die Prozedur INSERTION-SORT so um, dass in nichtsteigender anstatt nichtfallender Ordnung sortiert wird.

2.1-3 Betrachten Sie das ***Suchproblem***:

Eingabe: Eine Sequenz von n Zahlen $A = \langle a_1, a_2, \ldots, a_n \rangle$ und ein Wert v.

Ausgabe: Ein Index i mit $v = A[i]$, falls v in A vorkommt, und den speziellen Wert NIL, wenn der Wert v nicht in A vorkommt.

Schreiben Sie ein Programm in Pseudocode für die ***lineare Suche***, die eine Sequenz nach v durchsucht. Beweisen Sie unter Verwendung einer Schleifeninvariante, dass Ihr Algorithmus korrekt ist. Achten Sie darauf, dass Ihre Schleifeninvariante die drei notwendigen Eigenschaften erfüllt.

2.1-4 Betrachten Sie das Problem der Addition von zwei n-Bit Binärzahlen, die in zwei Feldern A und B der Länge n gespeichert sind. Die Summe der beiden ganzen Zahlen soll in einem Feld C der Länge $n+1$ gespeichert werden. Beschreiben Sie das Problem formal und geben Sie ein Programm in Pseudocode für die Addition von zwei ganzen Zahlen an.

2.2 Analyse von Algorithmen

Die ***Analyse*** eines Algorithmus ist von Bedeutung, um die Ressourcen vorauszubestimmen, die der Algorithmus benötigt. Gelegentlich sind Ressourcen wie Speicher, Kommunikationsbandbreite oder Computerhardware von grundlegendem Interesse, aber am häufigsten ist es die Rechenzeit, die wir abschätzen möchten. Gewöhnlich können wir durch Analyse mehrerer Algorithmen für ein Problem den effizientesten von ihnen identifizieren. Eine solche Analyse kann auf mehr als einen brauchbaren Kandidaten hinweisen, die schlechteren Algorithmen können jedoch durch diesen Prozess verworfen werden.

Bevor wir einen Algorithmus analysieren können, müssen wir über ein Modell der Technologie, auf der wir den Algorithmus implementieren wollen, verfügen, einschließlich eines Modells der Ressourcen dieser Technologie und deren Kosten. Meistens werden wir in diesem Buch von dem allgemeinen Modell einer ***Maschine mit wahlfreiem Zugriff (RAM)*** (mit einem Prozessor) ausgehen und annehmen, dass die von uns angegebenen Algorithmen als Computerprogramme auf diesem Modell ausgeführt werden. Innerhalb des RAM-Modells werden Anweisungen nacheinander ausgeführt. Operationen können nicht parallel ablaufen.

Streng genommen sollte man die Befehle des RAM-Modells und deren Aufwand genau bestimmen. Dies zu tun wäre jedoch mühevoll und würde wenig Einblick in den Entwurf von Algorithmen und deren Analyse bieten. Jedoch müssen wir vorsichtig sein, das RAM-Modell nicht zu missbrauchen. Was wäre zum Beispiel, wenn ein RAM über einen Befehl zum Sortieren verfügte? Dann könnten wir mit nur einem Befehl sortieren. Solch ein RAM wäre unrealistisch, da reale Rechner nicht über solche Befehle verfügen. Wir wollen uns am Aufbau realer Rechner orientieren. Das RAM-Modell enthält Befehle, die man üblicherweise in realen Rechnern findet: arithmetische (wie zum Beispiel Addieren, Subtrahieren, Multiplizieren, Dividieren, Restbilden, Abrunden, Aufrunden), Daten bewegende (Laden, Speichern, Kopieren) und kontrollierende (bedingte und unbedingte Verzweigung, Aufruf einer Unterroutine und Rücksprung) Maschinenbefehle. Jeder dieser Befehle benötigt ein festes Maß an Zeit.

Die Datentypen innerhalb des RAM-Modells sind Integer-Zahlen und Gleitkommazahlen (zum Speichern reeller Zahlen). Obwohl wir uns in diesem Buch normalerweise nicht mit Rechengenauigkeit beschäftigen, ist bei einigen Anwendungen die Rechengenauigkeit ausschlaggebend. Wir gehen im Buch außerdem davon aus, dass die Breite eines Datenwortes begrenzt ist. Wenn wir zum Beispiel mit Werten bis n arbeiten, dann nehmen wir in der Regel an, dass Integer-Zahlen mit $c \lg n$ Bits repräsentiert werden, wobei $c \geq 1$ eine von n unabhängige Konstante ist. Wir fordern $c \geq 1$, sodass jedes Wort den Wert n speichern kann. Wir fordern, dass c eine Konstante ist, damit die Wortbreite

nicht beliebig anwachsen kann. (Wenn die Wortbreite beliebig wachsen könnte, könnten wir riesige Datenmengen in einem Wort speichern und Operationen auf ihnen jeweils in konstanter Zeit ausführen – eindeutig ein unrealistisches Szenario.)

Reale Rechner enthalten weitere Befehle, die wir oben nicht aufgelistet haben. Solche Befehle bilden eine Grauzone im RAM-Modell. Ist zum Beispiel die Potenzierung ein zeitkonstanter Befehl? Im allgemeinen Fall nein; es sind einige Befehle notwendig, um x^y zu berechnen, wenn x und y reelle Zahlen sind. Dennoch ist die Potenzierung in eingeschränkten Fällen eine zeitkonstante Operation. Viele Rechner verfügen über einen Maschinenbefehl für einen Linksshift einer Integer-Zahl um k Positionen. Ein solcher Maschinenbefehl wird in konstanter Zeit ausgeführt. Bei den meisten Rechnern ist das Verschieben der Bits einer ganzen Zahl um eine Position nach links äquivalent zur Multiplikation mit 2, sodass eine Verschiebung der Bits um k Positionen nach links äquivalent zur Multiplikation mit 2^k ist. Deshalb können solche Rechner den Wert 2^k in einem zeitkonstanten Befehl berechnen, indem sie die Integer-Zahl 1 um k Positionen nach links verschieben, solange k nicht größer als die Wortbreite ist. Wir werden uns bemühen, solche Grauzonen des RAM-Modells zu meiden. Wir werden aber die Berechnung von 2^k als zeitkonstante Operation ansehen, wenn k eine ausreichend kleine positive ganze Zahl ist.

Im RAM-Modell versuchen wir nicht, die Konzepte der Speicherhierarchie, wie sie in heutigen Rechnern realisiert sind, zu modellieren. Wir modellieren weder den oder die Caches noch virtuellen Speicher. Verschiedene Modelle versuchen, die zum Teil signifikanten Speicherhierarchie-Effekte bei realen Programmen auf realen Rechnern zu berücksichtigen. Eine Hand voll von Problemen, die Sie in diesem Buch finden, untersuchen Speicherhierarchie-Effekte. In den meisten Fällen betrachten die Analysen in diesem Buch diese Effekte aber nicht. Modelle, die die Speicherhierarchie einschließen, sind um einiges komplexer als das RAM-Modell, sodass es schwieriger sein kann, mit ihnen zu arbeiten. Darüber hinaus liefern die RAM-Modell-Analysen gewöhnlich ausgezeichnete Vorhersagen des Leistungsverhaltens auf derzeitigen Maschinen.

Schon einen einfachen Algorithmus im RAM-Modell zu untersuchen, kann eine große Herausforderung sein. Die erforderlichen mathematischen Hilfsmittel können Kombinatorik, Wahrscheinlichkeitstheorie, Algebra und die Fähigkeit, die maßgeblichen Terme in einer Formel zu identifizieren, einschließen. Da sich das Verhalten eines Algorithmus für jede mögliche Eingabe unterscheiden kann, benötigen wir ein Hilfsmittel, um dieses Verhalten in einfachen, leicht verständlichen Formeln zusammenzufassen.

Selbst wenn wir normalerweise nur ein Maschinenmodell auswählen, um einen gegebenen Algorithmus zu analysieren, müssen wir uns noch für einen Weg entscheiden, wie wir unsere Analysen formulieren. Hier existieren verschiedene Alternativen. Wir hätten gern einen Weg, der einfach aufzuschreiben und zu handhaben ist, die wichtigsten Charakteristiken eines Algorithmus für dessen Anforderungen an die Ressourcen aufzeigt und unwesentliche Details vernachlässigt.

Analyse des Sortierens durch Einfügen

Die beim Sortieren durch Einfügen benötigte Zeit hängt von der Eingabe ab: Das Sortieren von Tausenden von Zahlen dauert länger als das Sortieren von drei Zahlen. Au-

ßerdem kann das Sortieren durch Einfügen unterschiedlichen Zeitaufwand für Eingabefolgen der gleichen Länge erfordern, da dieser davon abhängt, wie gut die Folgen bereits vorsortiert sind. Im Allgemeinen wächst die Zeit, die ein Algorithmus benötigt, mit der Größe der Eingabe. Daher ist es üblich, die Laufzeit eines Programmes als Funktion der Eingabegröße zu beschreiben. Um dies zu tun, müssen wir die Begriffe „Laufzeit“ und „Eingabegröße“ sorgfältiger definieren.

Was die beste Definition für die ***Eingabegröße*** ist, hängt vom betrachteten Problem ab. Für viele Probleme, wie für das Sortieren oder Berechnen von diskreten Fourier-Transformierten, ist das natürliche Maß die *Anzahl der Datensätze der Eingabe* – beim Sortieren zum Beispiel die Länge n des zu sortierenden Feldes. Für viele andere Probleme, wie zum Beispiel die Multiplikation zweier Zahlen, ist das beste Maß für die Eingabegröße die *Anzahl der Bits*, die nötig sind, um die Eingabe in gewöhnlicher binärer Notation darzustellen. Manchmal ist es angemessener, die Eingabegröße durch zwei Zahlen anstatt durch eine einzige zu beschreiben. Wenn zum Beispiel die Eingabe eines Algorithmus ein Graph ist, kann die Eingabegröße durch die Anzahl der Knoten und die Anzahl der Kanten des Graphen beschrieben werden. Wir werden für jedes betrachtete Problem angeben, welches Maß für die Eingabegröße benutzt wird.

Die ***Laufzeit*** eines Algorithmus für eine spezielle Eingabe ist die Anzahl der ausgeführten Grundoperationen oder „Schritte“. Es ist günstig, den Begriff Schritt so zu definieren, dass er so maschinenunabhängig wie möglich wird. Zunächst wollen wir uns die folgende Sichtweise zueigen machen. Für das Ausführen jeder Zeile unseres Pseudocodes ist ein konstanter Zeitaufwand erforderlich. Verschiedene Zeilen können unterschiedlichen Zeitaufwand erfordern. Wir gehen aber davon aus, dass jede Zeile i nur einen konstanten Zeitaufwand c_i benötigt. Diese Sichtweise ist im Einklang mit dem RAM-Modell und sie spiegelt auch wider, wie der Pseudocode auf den meisten heutigen Rechnern zu implementieren wäre.[5]

In der folgenden Diskussion wird sich unser Ausdruck für die Laufzeit des Sortierens durch Einfügen von einer unhandlichen Formel, die die Kosten c_i für sämtliche Anweisungen berücksichtigt, zu einer wesentlich einfacheren Notation entwickeln. Dieser Ausdruck ist prägnanter und einfacher zu handhaben.

Wir beginnen, indem wir die Prozedur des Sortierens durch Einfügen nochmals hinschreiben und jede Anweisung mit den Kosten, also dem Zeitbedarf für die einmalige Ausführung, und der Anzahl der Durchläufe annotieren. Für jedes $j = 2, 3, \ldots, n$ mit $n = A.\mathit{länge}$ geben wir mit t_j an, wie viele Male die Abfrage der **while**-Schleife in Zeile 5 für den Wert j ausgeführt wird. Wenn eine **for**- oder **while**-Schleife auf normalem Weg verlassen wird (d. h. infolge der Abfrage im Schleifenkopf), dann wird die Abfrage genau einmal mehr ausgeführt als der Schleifenrumpf. Wir gehen davon aus, dass Kommentare keine ausführbaren Anweisungen sind und daher keine Zeit beanspruchen.

[5]Es gibt hier ein paar Feinheiten. Rechenschritte, die wir in natürlicher Sprache formulieren, sind häufig Varianten einer Prozedur, die mehr als nur einen konstanten Zeitaufwand erfordern. Zum Beispiel werden wir weiter hinten im Buch als ein Rechenschritt „sortiere die Punkte nach der x-Koordinate“ angeben, was, wie wir sehen werden, einen nichtkonstanten Zeitaufwand erfordert. Beachten Sie außerdem, dass eine Anweisung, die eine Unterroutine aufruft, eine konstante Zeit benötigt, während die Unterroutine selbst unterschiedlich lange dauern kann. Das heißt, dass wir den Prozess des ***Aufrufs*** einer Unterroutine – das Übergeben der Parameter, usw. – vom Prozess des ***Ausführens*** einer Unterroutine unterscheiden.

INSERTION-SORT(A)	*Kosten*	*Anzahl*
1 **for** $j = 2$ **to** $A.länge$	c_1	n
2 $\quad schlüssel = A[j]$	c_2	$n-1$
3 $\quad$ // Setze $A[j]$ in das sortierte		
$\quad$ // Teilfeld $A[1\,..\,j-1]$ ein.	0	$n-1$
4 $\quad i = j - 1$	c_4	$n-1$
5 $\quad$ **while** $i > 0$ und $A[i] > schlüssel$	c_5	$\sum_{j=2}^{n} t_j$
6 $\qquad A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7 $\qquad i = i - 1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8 $\quad A[i+1] = schlüssel$	c_8	$n-1$

Die Laufzeit des Algorithmus ist die Summe der Laufzeiten aller ausgeführten Anweisungen; eine Anweisung, die c_i Schritte benötigt, um ausgeführt zu werden und die n-mal ausgeführt wird, liefert den Beitrag $c_i\,n$ zur gesamten Laufzeit.[6] Um die Laufzeit $T(n)$ für das Sortieren durch Einfügen zu berechnen, summieren wir die Produkte aus *Kosten* und *Anzahl* und erhalten

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) \\ &\quad + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1)\ . \end{aligned}$$

Selbst wenn die Eingabe eine fest vorgegebene Größe hat, kann die Laufzeit eines Algorithmus davon abhängen, *welche* Eingabe dieser Größe betrachtet wird. Zum Beispiel tritt beim Sortieren durch Einfügen der günstigste Fall ein, wenn das zu sortierende Feld bereits sortiert ist. Für jedes $j = 2, 3, \ldots, n$ stellen wir fest, dass in Zeile 5 $A[i] \leq schlüssel$ gilt, wenn i seinen Startwert $j-1$ annimmt. Daher gilt in diesem Fall $t_j = 1$ für $j = 2, 3, \ldots, n$ und die bestmögliche Laufzeit ist

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)\ . \end{aligned}$$

Wir können diese Laufzeit in der Form $a\,n + b$ ausdrücken, wobei a und b *Konstanten* sind, die von den Kosten c_i der Anweisungen abhängen; sie ist also eine ***lineare Funktion*** in n.

Falls das Feld in umgekehrter Reihenfolge – d. h. in absteigender Reihenfolge – sortiert ist, liegt der schlechteste Fall vor. Wir müssen jedes Element $A[j]$ mit sämtlichen Elementen des bereits sortierten Teils $A[1\,..\,j-1]$ des Feldes vergleichen, und damit wird $t_j = j$ für alle $j = 2, 3, \ldots, n$. Wegen

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

[6] Dies gilt nicht zwingend auch für eine Ressource wie den Speicherplatz. Eine Anweisung, die m Speicherwörter belegt und n-mal ausgeführt wird, muss insgesamt nicht notwendigerweise Speicherplatz in der Größe von $m\,n$ Wörtern verbrauchen.

und

$$\sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}$$

(Anhang A gibt einen Überblick, wie diese Summenformeln berechnet werden) erhalten wir, dass die Laufzeit beim Sortieren durch Einfügen im schlechtesten Fall

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

ist. Wir können also die Laufzeit im schlechtesten Fall in der Form $a\,n^2 + b\,n + c$ angeben, wobei die Konstanten a, b und c wiederum nur von den Kosten c_i der Anweisungen abhängen. Dies ist eine ***quadratische Funktion*** in n.

Typischerweise ist die Laufzeit eines Algorithmus wie beim Sortieren durch Einfügen für eine bestimmte Eingabe konstant. Allerdings werden wir in späteren Kapiteln einige interessante „randomisierte" Algorithmen betrachten, deren Verhalten selbst bei fester Eingabe variieren kann.

Analyse des schlechtesten Falls und des mittleren Falls

Bei unserer Analyse des Sortierens durch Einfügen haben wir uns sowohl den günstigsten Fall, in dem das Eingabefeld bereits sortiert war, als auch den schlechtesten Fall, in dem das Eingabefeld in umgekehrter Reihenfolge sortiert war, angesehen. Für den Rest des Buches werden wir uns jedoch üblicherweise darauf konzentrieren, nur die ***Laufzeit im schlechtesten Fall***, d. h. die längste Laufzeit, die das entsprechende Verfahren angewendet auf *beliebige* Eingaben der Größe n (höchstens) benötigt, anzugeben. Wir geben drei Gründe für diese Ausrichtung an.

- Die Laufzeit eines Algorithmus im schlechtesten Fall gibt uns eine obere Schranke für die Laufzeit einer beliebigen Eingabe. Deren Kenntnis gibt uns eine Garantie dafür, dass der Algorithmus niemals länger brauchen wird.

- Bei einigen Algorithmen tritt der schlechteste Fall ziemlich häufig auf. Zum Beispiel tritt beim Durchsuchen einer Datenbank nach einer speziellen Information der schlechteste Fall für den Suchalgorithmus oft dann auf, wenn die gesuchte Information nicht in der Datenbank vorhanden ist, und bei einigen Anwendungen mag das Suchen nach nicht vorhandener Information häufig vorkommen.

- Der „mittlere Fall“ ist oft annähernd genauso schlecht wie der schlechteste Fall. Nehmen wir an, dass wir n Zahlen zufällig auswählen und Sortieren durch Einfügen darauf anwenden. Wie lange dauert es zu bestimmen, an welcher Stelle des Teilfeldes $A[1 \mathinner{.\,.} j-1]$ das Element $A[j]$ einzuordnen ist? Im Mittel ist die Hälfte der Elemente in $A[1 \mathinner{.\,.} j-1]$ kleiner als $A[j]$ und die Hälfte der Elemente ist größer. Deshalb prüfen wir im Mittel die Hälfte des Teilfeldes $A[1 \mathinner{.\,.} j-1]$ und so ist t_j ungefähr $j/2$. Es stellt sich heraus, dass die daraus resultierende mittlere Laufzeit eine quadratische Funktion in der Größe der Eingabe ist, genau wie die Laufzeit im schlechtesten Fall.

In einigen besonderen Fällen sind wir an der ***mittleren*** Laufzeit eines Algorithmus interessiert. Wir werden die Methode der ***probabilistischen Analyse*** angewendet auf unterschiedliche Algorithmen das ganze Buch hindurch begegnen. Der Anwendungsbereich der Analyse des mittleren Falls ist beschränkt, da es nicht unbedingt offensichtlich ist, wie die „durchschnittliche“ Eingabe eines speziellen Problems aussieht. Häufig werden wir annehmen, dass alle Eingaben einer gegebenen Größe gleichwahrscheinlich sind. In der Praxis mag diese Annahme verletzt werden, aber wir können in einigen Fällen einen ***randomisierten Algorithmus*** benutzen, der durch „Würfeln“ eine probabilistische Analyse ermöglicht und zu einer ***erwarteten Laufzeit*** führt. Wir untersuchen randomisierte Algorithmen in Kapitel 5 und in einzelnen weiteren, nachfolgenden Kapiteln.

Wachstumsgrad

Wir haben einige vereinfachende Abstraktionen benutzt, um unsere Analyse des Sortierens durch Einfügen zu erleichtern. Zuerst haben wir von den tatsächlichen Kosten jeder Anweisung abstrahiert, indem wir die Konstanten c_i für die Darstellung dieser Kosten benutzt haben. Dann haben wir beobachtet, dass uns sogar diese Konstanten mehr Details liefern als wir tatsächlich benötigen: Wir haben die Laufzeit im schlechtesten Fall mit $a\,n^2 + b\,n + c$ angegeben, wobei a, b und c Konstanten sind, die (nur) von den Kosten c_i der Anweisungen abhängen. So haben wir nicht nur von den tatsächlichen Kosten der Anweisung, sondern auch von den abstrakten Kosten c_i abstrahiert.

Wir werden nun eine weitere vereinfachende Abstraktion machen: Es ist die ***Wachstumsrate***, oder der ***Wachstumsgrad*** der Laufzeit, der uns eigentlich interessiert. Wir betrachten deshalb nur den führenden Term einer Formel (zum Beispiel $a\,n^2$), da die Terme niedrigerer Ordnung für große Werte von n relativ unwesentlich sind. Wir ignorieren auch den konstanten Koeffizienten des führenden Terms, da konstante Faktoren weniger signifikant als die Wachstumsrate sind, wenn man Recheneffizienz für große Eingaben bestimmt. Bei Sortieren durch Einfügen bleiben wir mit dem Faktor von n^2 des führenden Terms, wenn wir von den Termen niedrigerer Ordnung und von der Konstante des führenden Terms abstrahieren. Wir schreiben, dass Sortieren durch Einfügen eine Laufzeit im schlechtesten Fall von $\Theta(n^2)$ (gesprochen „Theta von n-Quadrat“) besitzt. Wir werden die Θ-Notation bereits in diesem Kapitel informal benutzen und sie im Kapitel 3 exakt definieren.

Gewöhnlich betrachten wir einen Algorithmus als effizienter als einen anderen, wenn seine Laufzeit im schlechtesten Fall eine geringere Wachstumsrate aufweist. Aufgrund

der konstanten Faktoren und Terme niedrigerer Ordnung kann ein Algorithmus, dessen Laufzeit eine höhere Wachstumsrate hat, weniger Zeit bei kleinen Eingabegrößen haben, als ein Algorithmus, dessen Laufzeit einer niedrigeren Wachstumsrate unterliegt. Für hinreichend große Eingaben wird jedoch zum Beispiel ein $\Theta(n^2)$-Algorithmus im schlechtesten Fall schneller laufen als ein $\Theta(n^3)$-Algorithmus.

Übungen

2.2-1 Drücken Sie die Formel $n^3/1000 - 100\,n^2 - 100\,n + 3$ im Sinne der Θ-Notation aus.

2.2-2 Betrachten Sie das Sortieren von n Zahlen, die in einem Feld A gespeichert sind. Finden Sie zuerst das kleinste Element in A und tauschen Sie es mit $A[1]$ aus. Finden Sie dann das zweitkleinste Element und tauschen Sie es mit $A[2]$ aus. Fahren Sie für die ersten $n-1$ Elemente von A in dieser Weise fort. Schreiben Sie ein Programm in Pseudocode für diesen als ***Sortieren durch Auswählen*** (engl.: *selection sort*) bekannten Algorithmus. Welche Schleifeninvariante gilt für diesen Algorithmus? Warum müssen nur die ersten $n-1$ Elemente anstatt n Elemente durchlaufen werden? Geben Sie die günstigste Laufzeit und die Laufzeit im schlechtesten Fall für Sortieren durch Auswählen in der Θ-Notation an.

2.2-3 Betrachten Sie noch einmal die lineare Suche (siehe Übung 2.1-3). Wie viele Elemente der Eingabefolge müssen im Mittel geprüft werden, wenn Sie davon ausgehen können, dass das gesuchte Element gleichwahrscheinlich irgendein Element von A ist? Wie verhält es sich im schlechtesten Fall? Wie ist die mittlere Laufzeit und wie die Laufzeit im schlechtesten Fall der linearen Suche, ausgedrückt in Θ-Notation? Begründen Sie ihre Antworten.

2.2-4 Wie können wir fast jeden Algorithmus modifizieren, um eine gute Laufzeit im günstigsten Fall zu erhalten?

2.3 Entwurf von Algorithmen

Wir können aus einer Vielfalt von Techniken zum Entwurf von Algorithmen auswählen. Bei Sortieren durch Einfügen wählten wir eine ***inkrementelle*** Herangehensweise: Nachdem wir das Teilfeld $A[1\,..\,j-1]$ sortiert haben, fügen wir das Element $A[j]$ an der passenden Stelle ein, woraus sich das sortierte Teilfeld $A[1\,..\,j]$ ergibt.

In diesem Abschnitt untersuchen wir eine alternative Entwurfstechnik, die als „Teile-und-Beherrsche“ (engl.: *divide and conquer*) bekannt ist und die wir in Kapitel 4 detailliert untersuchen werden. Wir werden die Teile-und-Beherrsche-Methode benutzen, um einen Sortieralgorithmus zu entwerfen, dessen Laufzeit im schlechtesten Fall viel geringer als die von Sortieren durch Einfügen ist. Ein Vorteil solcher „Teile-und-Beherrsche“-Algorithmen ist es, dass deren Laufzeiten häufig einfach durch die in Kapitel 4 eingeführten Techniken bestimmt werden können.

2.3.1 Die Teile-und-Beherrsche-Methode

Viele nützliche Algorithmen sind bezüglich ihrer Struktur ***rekursiv***: Um ein gegebenes Problem zu lösen, rufen sie sich selbst einmal oder mehrmals auf, um eng verwandte Teilprobleme zu behandeln. Diese Algorithmen folgen üblicherweise der Methode von ***Teile-und-Beherrsche***: Sie teilen das Problem in mehrere Teilprobleme auf, die dem Ausgangsproblem ähneln, jedoch von kleinerer Größe sind. Sie lösen die Teilprobleme rekursiv und kombinieren diese Lösungen dann, um die Lösung des eigentlichen Problems zu erstellen.

Das Paradigma von Teile-und-Beherrsche umfasst drei Schritte auf jeder Rekursionsebene:

Teile das Problem in mehrere Teilprobleme auf, die kleinere Instanzen des gleichen Problems darstellen.

Beherrsche die Teilprobleme rekursiv. Wenn die Teilprobleme klein genug sind, dann löse die Teilprobleme auf direktem Wege.

Vereinige die Lösungen der Teilprobleme zur Lösung des ursprünglichen Problems.

Der Algorithmus des ***Sortierens durch Mischen*** folgt genau dem Paradigma von Teile-und-Beherrsche. Intuitiv arbeitet er wie folgt.

Teile: Teile die Folge von n Elementen in zwei Teilfolgen von je $n/2$ Elementen auf.

Beherrsche: Sortiere die zwei Teilfolgen rekursiv mithilfe von Sortieren durch Mischen.

Vereinige: Mische die zwei sortierten Teilfolgen, um die sortierte Lösung zu erzeugen.

Die Rekursion bricht ab, wenn die zu sortierende Teilfolge die Länge 1 hat. In diesem Fall ist nichts mehr zu tun, da jede Folge der Länge 1 bereits sortiert ist.

Die zentrale Operation des Sortierens durch Mischen ist das Mischen der zwei sortierten Folgen im dritten Schritt. Wir mischen, indem wir eine Hilfsprozedur $\text{Merge}(A, p, q, r)$ aufrufen, wobei A ein Feld ist und p, q und r Indizes des Feldes sind, für die $p \leq q < r$ gilt. Die Prozedur geht davon aus, dass sich die Teilfelder $A[p \mathinner{..} q]$ und $A[q+1 \mathinner{..} r]$ in sortierter Reihenfolge befinden. Sie mischt diese, um so ein einziges sortiertes Teilfeld zu bekommen, das das aktuelle Teilfeld $A[p \mathinner{..} r]$ ersetzt.

Unsere Merge-Prozedur benötigt Zeit $\Theta(n)$, wobei $n = r - p + 1$ die Gesamtanzahl der zu mischenden Elemente ist. Sie arbeitet wie folgt: In der Sprache des Kartenspiels formuliert, haben wir zwei aufgedeckte Kartenhaufen auf dem Tisch liegen. Jeder Haufen ist sortiert, wobei die Karte mit dem kleinsten Wert oben liegt. Wir möchten diese zwei Haufen zu einem einzigen sortierten Ausgabehaufen vereinigen, der verdeckt auf dem Tisch liegt. Unser grundlegender Schritt besteht darin, die kleinere der beiden auf den zwei Haufen oben liegenden Karten auszuwählen, sie vom Haufen wegzunehmen (wobei eine neue oberste Karte aufgedeckt wird) und diese Karte mit dem Gesicht nach unten auf den Ausgabehaufen zu legen. Wir wiederholen diesen Schritt, bis einer der

beiden ursprünglichen Haufen aufgebraucht ist, wobei wir danach einfach den verbliebenen Haufen nehmen und ihn mit dem Gesicht nach unten auf den Ausgabehaufen legen. Von der Berechnung her benötigt jeder Grundschritt eine konstante Zeit, da wir lediglich die zwei obersten Karten zu vergleichen haben. Da wir höchstens n Grundschritte ausführen, benötigt das Mischen die Zeit $\Theta(n)$.

Der folgende Pseudocode setzt obige Ideen um, allerdings mit einem zusätzlichen Trick, der vermeidet, dass wir bei jedem Grundschritt überprüfen müssen, ob einer der beiden Haufen aufgebraucht ist. Wir legen an das Ende eines jeden Haufens eine ***Wächterkarte***, die einen speziellen Wert enthält. Wir benutzen sie, um unseren Code zu vereinfachen. Als Wächterwert verwenden wir hier ∞, sodass wenn eine Karte mit dem Wert ∞ aufgedeckt wird, dies nicht die kleinere Karte sein kann, außer wenn bei beiden Haufen die Wächterkarte aufgedeckt ist. Allerdings sind, wenn dies eintritt, bereits alle „normalen“ Karten auf den Ausgabehaufen gelegt worden. Da wir im Voraus wissen, dass genau $r-p+1$ Karten auf den Ausgabehaufen gelegt werden, können wir stoppen, wenn wir genau so viele Grundschritte ausgeführt haben.

```
MERGE(A, p, q, r)
 1  n1 = q - p + 1
 2  n2 = r - q
 3  seien L[1 .. n1 + 1] und R[1 .. n2 + 1] zwei neue Felder
 4  for i = 1 to n1
 5          L[i] = A[p + i - 1]
 6  for j = 1 to n2
 7          R[j] = A[q + j]
 8  L[n1 + 1] = ∞
 9  R[n2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13          if L[i] ≤ R[j]
14                  A[k] = L[i]
15                  i = i + 1
16          else A[k] = R[j]
17                  j = j + 1
```

Im Detail arbeitet die Prozedur MERGE wie folgt. Zeile 1 bestimmt die Länge n_1 des Teilfeldes $A[p \,..\, q]$ und Zeile 2 die Länge n_2 des Teilfeldes $A[q+1 \,..\, r]$. In Zeile 3 legen wir die Felder L und R („links“ und „rechts“) mit der Länge n_1+1 beziehungsweise n_2+1 an. Die **for**-Schleife in den Zeilen 4–5 kopiert das Teilfeld $A[p \,..\, q]$ in $L[1 \,..\, n_1]$ und die **for**-Schleife in den Zeilen 6–7 kopiert das Teilfeld $A[q+1 \,..\, r]$ in $R[1 \,..\, n_2]$. Die Zeilen 8–9 setzen die Wächter an das Ende von L und R. Die Zeilen 10–17, die in Abbildung 2.3 illustriert werden, führen die $r-p+1$ Grundschritte aus, wobei die folgende Schleifeninvariante aufrecht erhalten wird:

> Zu Beginn jeder Iteration der **for**-Schleife in den Zeilen 12–17 enthält das Teilfeld $A[p \,..\, k-1]$ die $k-p$ kleinsten Elemente aus $L[1 \,..\, n_1+1]$ und

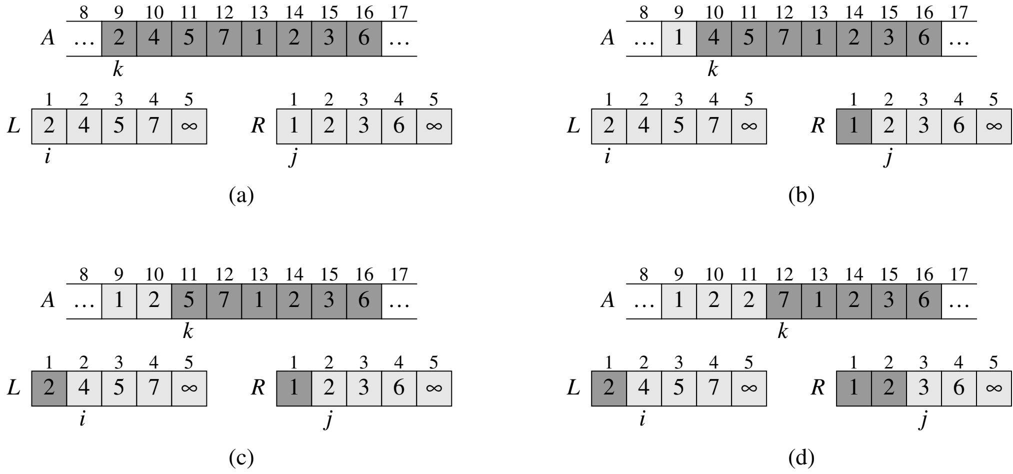


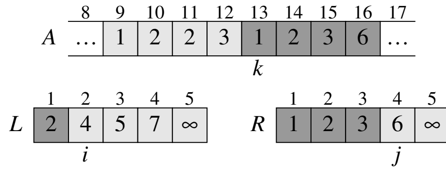
Abbildung 2.3: *Die Operation der Zeilen 10–17 beim Aufruf von* MERGE$(A, 9, 12, 16)$*, wenn das Teilfeld* $A[9..16]$ *die Folge* $\langle 2, 4, 5, 7, 1, 2, 3, 6\rangle$ *enthält. Nach dem Kopieren und dem Einfügen der Wächter enthält das Feld L die Folge* $\langle 2, 4, 5, 7, \infty\rangle$ *und das Feld R die Folge* $\langle 1, 2, 3, 6, \infty\rangle$*. Schwach schattierte Positionen in A enthalten bereits ihre endgültigen Werte, und schwach schattierte Positionen in L und R enthalten Werte, die noch nach A zurückkopiert werden müssen. Zusammengenommen beinhalten die schattierten Positionen immer die ursprünglich in* $A[9..16]$ *enthaltenen Werte zusammen mit den beiden Wächtern. Stark schattierte Positionen in A enthalten Werte, die noch überschrieben werden, und stark schattierte Positionen in L und R enthalten Werte, die bereits nach A zurückkopiert wurden.* ***(a)–(h)*** *Die Felder A, L und R und deren entsprechenden Indizes k, i und j vor jeder Iteration der Schleife in den Zeilen 12–17.*

$R[1..n_2 + 1]$ in sortierter Reihenfolge. Darüber hinaus sind $L[i]$ und $R[j]$ die kleinsten Elemente ihrer Felder, die noch nicht nach A zurückkopiert wurden.

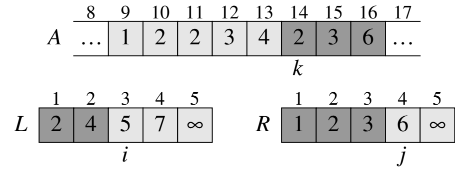
Wir müssen zeigen, dass die Schleifeninvariante vor der ersten Iteration der **for**-Schleife in den Zeilen 12–17 erfüllt wird, keine Iteration der Schleife die Invariante verletzt und die Invariante bei Abbruch der Schleife eine Eigenschaft liefert, die geeignet ist, um die Korrektheit des Algorithmus zu zeigen.

Initialisierung Vor der ersten Iteration der Schleife gilt $k = p$, sodass das Teilfeld $A[p..k-1]$ leer ist. Dieses Teilfeld enthält die $k-p = 0$ kleinsten Elemente von L und R, und wegen $i = j = 1$ sind sowohl $L[i]$ als auch $R[j]$ die kleinsten Elemente ihrer Felder, die noch nicht nach A zurückkopiert wurden.

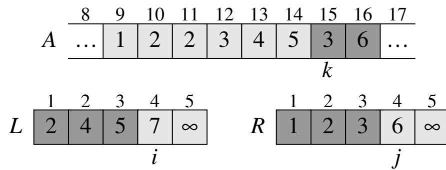
Fortsetzung Um zu zeigen, dass jede Iteration die Schleifeninvariante erhält, nehmen wir zunächst $L[i] \leq R[j]$ an. Dann ist $L[i]$ das kleinste Element, das noch nicht nach A zurückkopiert worden ist. Weil $A[p..k-1]$ die $k-p$ kleinsten Elemente enthält, wird das Teilfeld $A[p..k]$ die $k-p+1$ kleinsten Elemente enthalten,



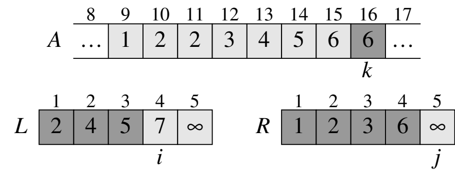
(e)



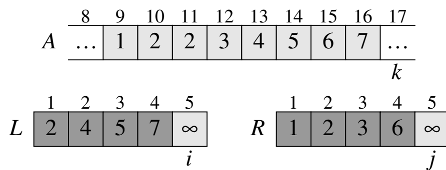
(f)



(g)



(h)



(i)

Abbildung 2.3, fortgesetzt: (i) *Die Felder und Indizes nach Terminierung. Zu diesem Zeitpunkt ist das Teilfeld $A[9\,..\,16]$ sortiert und die zwei Wächter in L und R sind die einzigen zwei Elemente in diesen Feldern, die nicht nach A kopiert worden sind.*

nachdem in Zeile 14 der Wert $L[i]$ nach $A[k]$ kopiert wurde. Inkrementieren von k (die Inkrementierung erfolgt durch die implizite Laufvariablen-Aktualisierung der **for**-Schleife) und i (in Zeile 15) stellt die Schleifeninvariante für die nächste Iteration wieder her. Wenn dagegen $L[i] > R[j]$ gilt, führen die Zeilen 16–17 die entsprechende Aktion aus, um die Schleifeninvariante zu erhalten.

Terminierung Beim Abbruch gilt $k = r + 1$. Wegen der Schleifeninvariante enthält das Teilfeld $A[p\,..\,k-1]$, d. h. $A[p\,..\,r]$, die $k - p = r - p + 1$ kleinsten Elemente von $L[1\,..\,n_1 + 1]$ und $R[1\,..\,n_2 + 1]$ in sortierter Reihenfolge. Die Felder L und R enthalten zusammen $n_1 + n_2 + 2 = r - p + 3$ Elemente. Alle Elemente außer den beiden größten sind zurück nach A kopiert worden, wobei diese größten Elemente die beiden Wächter sind.

Um zu zeigen, dass die Prozedur MERGE in Zeit $\Theta(n)$ mit $n = r - p + 1$ läuft, stellen wir fest, dass jede der Zeilen 1–3 und 8–11 konstante Zeit benötigt, die **for**-Schleifen in den Zeilen 4–7 benötigen die Zeit $\Theta(n_1 + n_2) = \Theta(n)$ [7] und es gibt n Iterationen der

[7] Wir werden in Kapitel 3 sehen, wie wir Gleichungen formal zu interpretieren haben, die die Θ-Notation enthalten.

for-Schleife in den Zeilen 12–17, von denen jede konstante Zeit benötigt.

Wir können nun die Prozedur MERGE als Unterroutine des Algorithmus Sortieren durch Mischen benutzen. Die Prozedur MERGE-SORT(A, p, r) sortiert die Elemente im Teilfeld $A[p\,..\,r]$. Falls $p \geq r$ gilt, hat das Teilfeld höchstens ein Element und ist deshalb bereits sortiert. Anderenfalls berechnet der Schritt des Teilens lediglich einen Index q, der $A[p\,..\,r]$ in zwei Teilfelder aufteilt: $A[p\,..\,q]$ enthält $\lceil n/2 \rceil$ Elemente und $A[q+1\,..\,r]$ enthält $\lfloor n/2 \rfloor$ Elemente.[8]

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

Um eine Folge A zu sortieren, rufen wir MERGE-SORT$(A, 1, A.\mathit{länge})$ auf, wobei wieder einmal $A.\mathit{länge} = n$ gilt. Abbildung 2.4 illustriert die Vorgehensweise der Prozedur für den Fall, dass n eine Potenz von 2 ist. Der Algorithmus besteht darin, Paare von einelementigen Folgen zu mischen und Folgen der Länge 2 zu bilden, Paare von Folgen der Länge 2 zu mischen und Folgen der Länge 4 zu bilden, usw., bis zwei Folgen der Länge $n/2$ gemischt werden und die endgültig sortierte Folge der Länge n gebildet wird.

2.3.2 Analyse von Teile-und-Beherrsche-Algorithmen

Wenn ein Algorithmus sich selbst rekursiv aufruft, dann können wir oftmals seine Laufzeit in Form einer ***Rekursionsgleichung*** angeben, die die Gesamtlaufzeit für ein Problem der Größe n in Abhängigkeit der Laufzeit für Probleme kleinerer Größe beschreibt. Wir können dann mittels mathematischer Hilfsmittel die Rekursionsgleichung lösen und Schranken für die Performanz des Algorithmus liefern.

Eine Rekursionsgleichung für die Laufzeit eines Teile-und-Beherrsche-Algorithmus basiert auf den drei Schritten des zugrunde liegenden Paradigmas. Wie vorhin sei $T(n)$ die Laufzeit eines Problems der Größe n. Wenn die Größe des Problems hinreichend klein ist, zum Beispiel $n \leq c$ für eine Konstante c, dann benötigt die direkte Lösung eine konstante Zeit, die wir mit $\Theta(1)$ beschreiben. Nehmen wir an, dass unsere Aufteilung des Problems zu a Teilproblemen führt, von denen jedes die Größe $1/b$ der Größe des ursprünglichen Problems hat. (Für Sortieren durch Mischen haben sowohl a als auch b den Wert 2; wir werden aber viele Teile-und-Beherrsche Algorithmen kennenlernen, bei denen $a \neq b$ gilt.) Das Lösen eines Teilproblems der Größe n/b dauert Zeit $T(n/b)$ und so benötigt das Lösen von a Problemen der Größe n/b Zeit $a\,T(n/b)$. Wenn wir die Zeit $D(n)$ benötigen, um das Problem in Teilprobleme aufzuteilen und die Zeit $C(n)$, um die

[8] Der Ausdruck $\lceil x \rceil$ bezeichnet die kleinste ganze Zahl, die größer oder gleich x ist und $\lfloor x \rfloor$ bezeichnet die größte ganze Zahl, die kleiner oder gleich x ist. Diese Bezeichnungen werden in Kapitel 3 definiert. Der einfachste Weg, um zu zeigen, dass wir zwei Teilfelder $A[p\,..\,q]$ und $A[q+1\,..\,r]$, die jeweils die Größe $\lceil n/2 \rceil$ bzw. $\lfloor n/2 \rfloor$ haben, erhalten, wenn wir q gleich $\lfloor (p+r)/2 \rfloor$ setzen, ist, die vier Fälle zu untersuchen, die abhängig davon, ob p und r ungerade oder gerade sind, auftreten.

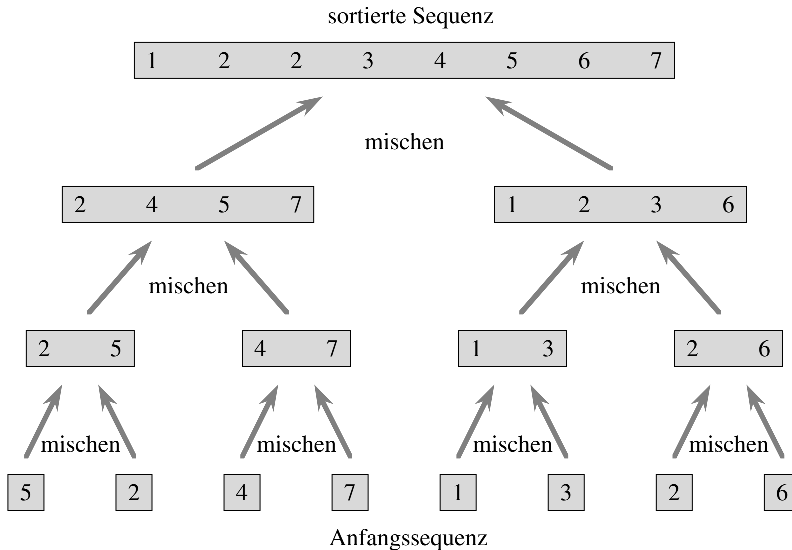


Abbildung 2.4: *Die Vorgehensweise von Sortieren durch Mischen auf dem Feld* $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$*. Die Länge der zu mischenden sortierten Folgen erhöht sich mit Fortschreiten des Algorithmus von unten nach oben.*

Lösung der Teilprobleme zur Lösung des ursprünglichen Problems zusammenzufügen, dann erhalten wir die Rekursionsgleichung

$$T(n) = \begin{cases} \Theta(1) & \text{falls } n \leq c \ , \\ a\,T(n/b) + D(n) + C(n) & \text{sonst} \ . \end{cases}$$

In Kapitel 4 werden wir sehen, wie wir häufig auftretende Rekursionsgleichungen dieser Art lösen können.

Analyse von Sortieren durch Mischen.

Auch wenn der Pseudocode von MERGE-SORT korrekt arbeitet, wenn die Anzahl der Elemente nicht gerade ist, vereinfacht sich unsere auf einer Rekursionsgleichung basierende Analyse, wenn wir annehmen, dass die Größe des ursprünglichen Problems eine Potenz von 2 ist. Das Teilen führt dann zu zwei Teilproblemen, die genau die Größe $n/2$ besitzen. In Kapitel 4 werden wir sehen, dass diese Annahme den Wachstumsgrad der Laufzeit für die Lösung der Rekursionsgleichung nicht beeinflusst.

Wir erörtern im Folgenden, wie wir zu der Rekursionsgleichung für die Laufzeit $T(n)$ von Sortieren durch Mischen von n Zahlen im schlechtesten Fall kommen können. Sortieren durch Mischen benötigt für das Bearbeiten eines Elementes eine konstante Zeit. Wenn wir $n > 1$ Elemente vorliegen haben, dann schlüsseln wir die Laufzeit wie folgt auf.

Teile Während der Teile-Phase wird einfach nur die Mitte des Feldes berechnet. Dies benötigt (bei einmaliger Ausführung) konstante Zeit. Somit gilt $D(n) = \Theta(1)$.

Beherrsche Wir lösen zwei Teilprobleme der Größe $n/2$ rekursiv, was $2\,T(n/2)$ zur Laufzeit beiträgt.

Vereinige Wir haben bereits festgestellt, dass die Prozedur MERGE auf einem Teilfeld der Länge n die Zeit $\Theta(n)$ benötigt, und so $C(n) = \Theta(n)$ gilt.

Wenn wir die Funktionen $D(n)$ und $C(n)$ zur Analyse des Sortierens durch Mischen addieren, dann addieren wir eine Funktion der Ordnung $\Theta(n)$ und eine Funktion der Ordnung $\Theta(1)$. Diese Summe ist eine lineare Funktion in n, d. h. der Ordnung $\Theta(n)$. Wenn wir sie zum Term $2\,T(n/2)$ aus dem Schritt „Beherrsche" addieren, dann ergibt sich als Rekursionsgleichung für die Laufzeit $T(n)$ von Sortieren durch Mischen im schlechtesten Fall

$$T(n) = \begin{cases} \Theta(1) & \text{falls } n = 1\ , \\ 2\,T(n/2) + \Theta(n) & \text{falls } n > 1\ . \end{cases} \tag{2.1}$$

In Kapitel 4 werden wir das Mastertheorem kennen lernen, mit dem wir zeigen können, dass $T(n)$ von der Ordnung $\Theta(n \lg n)$ ist, wobei $\lg n$ für $\log_2 n$ steht. Da die Logarithmusfunktion langsamer anwächst als die lineare Funktion, ist Sortieren durch Mischen mit seiner Laufzeit von $\Theta(n \lg n)$ bei hinreichend großen Zahlen besser als Sortieren durch Einfügen, dessen Laufzeit im schlechtesten Fall $\Theta(n^2)$ ist.

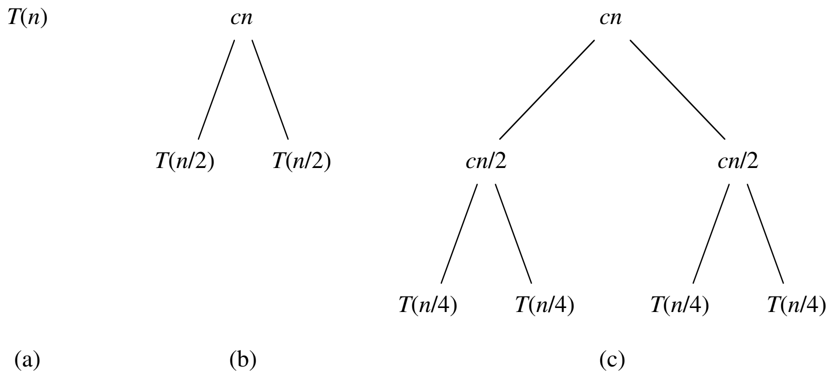
Um intuitiv zu verstehen, warum sich aus der Lösung der Rekursionsgleichung (2.1) $T(n) = \Theta(n \lg n)$ ergibt, benötigen wir das Mastertheorem nicht. Wir formen die Rekursionsgleichung (2.1) zu

$$T(n) = \begin{cases} c & \text{falls } n = 1\ , \\ 2\,T(n/2) + c\,n & \text{falls } n > 1\ , \end{cases} \tag{2.2}$$

um, wobei die Konstante c sowohl für die zum Lösen eines Problems der Größe 1 als auch für die pro Feldelement für die Schritte des Teilens und Kombinierens benötigte Zeit steht.[9]

Abbildung 2.5 zeigt, wie wir die Rekursionsgleichung (2.2) lösen können. Der Einfachheit halber nehmen wir an, dass n eine Potenz von 2 ist. Teil **(a)** der Abbildung zeigt $T(n)$, das in Teil **(b)** zu einem äquivalenten Baum erweitert wurde, der die Rekursionsgleichung veranschaulicht. Der Term $c\,n$ steht in der Wurzel und stellt den Aufwand der Rekursion in der ersten Ebene dar; die Teilbäume der Wurzel stehen für die beiden kleineren Rekurrenzen $T(n/2)$. Teil **(c)** zeigt diesen Prozess, nachdem die Rekursionsgleichung jeweils einmal auf $T(n/2)$ angewendet wurde. Die Kosten für jeden der beiden Knoten in der zweiten Rekursionsebene sind $c\,n/2$. Wir setzen die Expansion der Baumknoten fort, indem wir sie in ihre durch die Rekursion bestimmten Bestandteile zerle-

[9] Es ist unwahrscheinlich, dass genau dieselbe Konstante sowohl für die zum Lösen eines Problems der Größe 1 als auch für die pro Feldelement in den Schritten des Teilens und Zusammenfügens benötigte Zeit steht. Wir können dieses Problem umgehen, indem wir c den größeren dieser beiden Werte zuweisen und annehmen, dass unsere Rekursionsgleichung eine obere Schranke für die Laufzeit liefert, oder indem wir c den kleineren der beiden Werte zuweisen und annehmen, dass unsere Rekursionsgleichung eine untere Schranke für die Laufzeit liefert. Beide Schranken sind von der Ordnung $n \lg n$, und zusammengenommen ergibt dies eine Laufzeit von $\Theta(n \lg n)$.



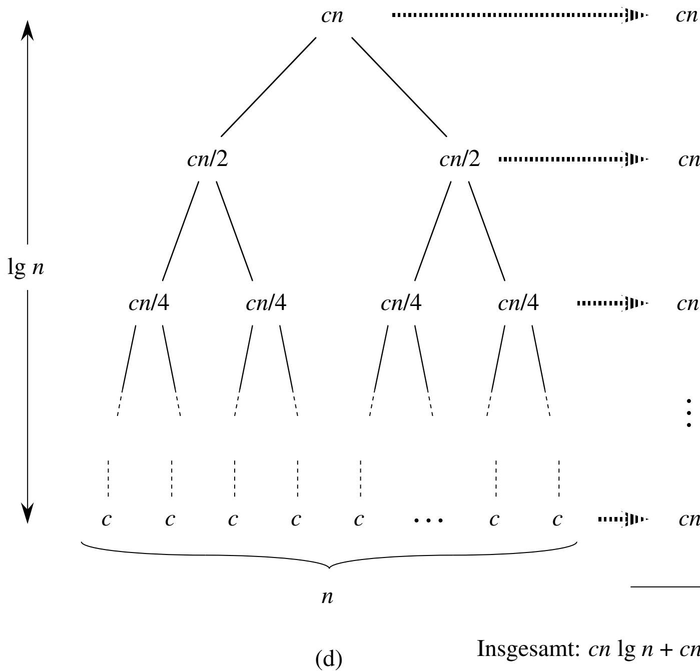


Abbildung 2.5: *Die Konstruktion eines Rekursionsbaumes für die Rekursionsgleichung* $T(n) = 2\,T(n/2) + c\,n$*.* ***(a)*** *zeigt* $T(n)$*, welches sich in* ***(b)–(d)*** *schrittweise entwickelt, was zum Rekursionsbaum führt. Der voll expandierte Baum, der in Teil* ***(d)*** *gezeigt wird, hat* $\lg n + 1$ *Ebenen (d. h. er besitzt wie angegeben die Höhe* $\lg n$*). Jede Ebene des voll expandierten Baumes trägt die Kosten* $c\,n$ *zu den Gesamtkosten bei. Deshalb ergibt sich für die Gesamtkosten* $c\,n \lg n + c\,n$*, was von der Ordnung* $\Theta(n \lg n)$ *ist.*

gen, bis Probleme der Größe 1 vorliegen, die jeweils mit konstanten Kosten c gelöst werden können. Teil **(d)** zeigt den resultierenden ***Rekursionsbaum***.

Als nächstes addieren wir die Kosten innerhalb jeder Ebene des Baumes. Für die oberste Ebene ergeben sich die Kosten $c\,n$, für die nächste Ebene ergeben sich Gesamtkosten von $c(n/2)+c(n/2) = c\,n$, die Ebene darunter hat die Kosten $c(n/4)+c(n/4)+c(n/4)+c(n/4) = c\,n$ usw. Im Allgemeinen besitzt die i-te Ebene unterhalb der Wurzel 2^i Knoten, jeder von ihnen trägt mit $c(n/2^i)$ zu den Kosten bei, sodass sich für die i-te Ebene die Gesamtkosten von $2^i\,c(n/2^i) = c\,n$ ergeben. Die unterste Ebene besteht aus n Knoten, die jeweils den Wert c zu den Kosten beitragen, was wiederum Gesamtkosten von $c\,n$ ergibt.

Die Gesamtzahl der Ebenen des Rekursionsbaumes in Abbildung 2.5 beträgt $\lg n+1$, wobei n die Anzahl der Blätter darstellt, welche zu der Größe der Eingabe korrespondiert. Ein (eher informaler) Induktionsbeweis beweist diese Aussage. Den Induktionsanfang bildet der Fall $n = 1$, in dem nur eine Ebene existiert. Wegen $\lg 1 = 0$ ergibt sich aus $\lg n + 1$ die korrekte Anzahl der Ebenen für diesen Fall. Nehmen Sie nun als Induktionshypothese an, dass die Anzahl der Ebenen eines Rekursionsbaumes mit 2^i Blättern gleich $\lg 2^i + 1 = i + 1$ ist (für jeden Wert von i gilt $\lg 2^i = i$). Da wir annehmen, dass die tatsächliche Eingabegröße eine Potenz von 2 ist, ist die nächste zu betrachtende Eingabegröße 2^{i+1}. Ein Baum mit 2^{i+1} Blättern hat eine Ebene mehr als ein Baum mit 2^i Blättern und so erhalten wir für die Gesamtanzahl der Ebenen $(i+1)+1 = \lg 2^{i+1} + 1$.

Um die Gesamtkosten, die durch die Rekursionsgleichung (2.2) beschrieben werden, zu berechnen, addieren wir einfach die Kosten aller Ebenen. Der Rekursionsbaum besteht aus $\lg n + 1$ Ebenen. Jede dieser Ebenen verursacht Kosten in Höhe von $c\,n$, was zu Gesamtkosten von $c\,n(\lg n+1) = c\,n \lg n + c\,n$ führt. Ignorieren wir den Term niedrigerer Ordnung und die Konstante c, so erhalten wir das gewünschte Resultat von $\Theta(n \lg n)$.

Übungen

2.3-1 Illustrieren Sie gemäß Abbildung 2.4 die Arbeitsweise von Sortieren durch Mischen angewendet auf das Feld $A = \langle 3, 41, 52, 26, 38, 57, 9, 49\rangle$

2.3-2 Schreiben Sie die Prozedur MERGE so um, dass sie keine Wächter benutzt, sondern dann anhält, wenn entweder aus L oder R alle Elemente nach A zurückkopiert wurden. Der Rest des anderen Feldes soll danach in A zurückkopiert werden.

2.3-3 Zeigen Sie durch mathematische Induktion, dass sich für die Lösung der Rekursionsgleichung

$$T(n) = \begin{cases} 2 & \text{falls } n = 2\ , \\ 2\,T(n/2) + n & \text{falls } n = 2^k,\ \text{für } k > 1 \end{cases}$$

der Wert $T(n) = n \lg n$ ergibt, wenn n eine Potenz von 2 ist.

2.3-4 Wir können Sortieren durch Einfügen wie folgt als rekursive Prozedur beschreiben. Um $A[1\,.\,.\,n]$ zu ordnen, sortieren wir $A[1\,.\,.\,n-1]$ rekursiv und setzen $A[n]$ in das sortierte Feld $A[1\,.\,.\,n-1]$ ein. Schreiben Sie eine Rekursionsgleichung für

die Laufzeit im schlechtesten Fall dieser rekursiven Version von Sortieren durch Einfügen.

2.3-5 Wir kommen noch einmal auf das Suchproblem (siehe Übung 2.1-3) zurück. Beachten Sie, dass, wenn die Folge A sortiert ist, wir v mit dem mittleren Element der Folge vergleichen können und nach diesem Vergleich die Hälfte der Folge nicht weiter betrachten brauchen. Die ***binäre Suche*** ist ein Algorithmus, der diese Prozedur wiederholt anwendet, wobei der verbleibende Teil der Folge jedes Mal halbiert wird. Schreiben Sie ein iteratives oder rekursives Programm in Pseudocode für die binäre Suche. Begründen Sie, warum die Laufzeit der binären Suche im schlechtesten Fall $\Theta(\lg n)$ ist.

2.3-6 Beachten Sie, dass die **while**-Schleife in den Zeilen 5–7 der Prozedur INSERTION-SORT in Abschnitt 2.1 die lineare Suche benutzt, um das sortierte Teilfeld $A[1 \mathinner{..} j-1]$ (rückwärts) zu durchsuchen. Können wir stattdessen die binäre Suche (siehe Übung 2.3-5) benutzen, um die Laufzeit $\Theta(n \lg n)$ von Sortieren durch Einfügen im schlechtesten Fall zu verbessern?

2.3-7* Beschreiben Sie einen Algorithmus der Ordnung $\Theta(n \lg n)$, der für eine gegebene Menge S von n ganzen Zahlen und eine weitere ganze Zahl x bestimmt, ob es zwei Elemente in S gibt, deren Summe genau x ist.

Problemstellungen

2-1 Sortieren durch Einfügen auf kleinen Feldern in Sortieren durch Mischen

Obwohl die Laufzeit von Sortieren durch Mischen im schlechtesten Fall von der Ordnung $\Theta(n \lg n)$ und die von Sortieren durch Einfügen von der Ordnung $\Theta(n^2)$ ist, ist Sortieren durch Einfügen bei kleinen n aufgrund der unterschiedlichen in den Θ-Notationen versteckten konstanten Faktoren auf vielen Maschinen schneller. Deshalb ist es sinnvoll, Sortieren durch Einfügen innerhalb des Sortierens durch Mischen zu verwenden, wenn die Teilprobleme hinreichend klein sind. Betrachten Sie eine Modifikation von Sortieren durch Mischen, bei der n/k Teillisten der Länge k durch Sortieren durch Einfügen sortiert und anschließend durch den Standardmechanismus gemischt werden, wobei k eine zu bestimmende Größe ist.

a. Zeigen Sie, dass Sortieren durch Einfügen n/k Teillisten jeweils der Länge k im schlechtesten Fall in Zeit $\Theta(n\,k)$ sortieren kann.

b. Zeigen Sie, wie man die Teillisten im schlechtesten Fall in Zeit $\Theta(n \lg(n/k))$ mischen kann.

c. Davon ausgehend, dass der modifizierte Algorithmus im schlechtesten Fall in Zeit $\Theta(n\,k + n \lg(n/k))$ läuft, welches ist der größte Wert von k als Funktion von n, für den der modifizierte Algorithmus die gleiche asymptotische Laufzeit besitzt wie das normale Sortieren durch Mischen? Geben Sie den Wert von k in Θ-Notation an!

d. Wie sollte k in der Praxis gewählt werden?

2-2 *Die Korrektheit von Sortieren durch Vertauschen*

Sortieren durch Vertauschen (engl.: *bubblesort*) ist ein populärer, wenn auch ineffizienter Sortieralgorithmus. Er arbeitet, indem er wiederholt benachbarte Elemente vertauscht, die sich nicht in der richtigen Reihenfolge befinden.

```
BUBBLESORT(A)
1  for i = 1 to A.länge − 1
2      for j = A.länge downto i + 1
3          if A[j] < A[j − 1]
4              vertausche A[j] mit A[j − 1]
```

a. Sei A' die Ausgabe von BUBBLESORT(A). Um die Korrektheit von BUBBLESORT zu beweisen, müssen wir zeigen, dass die Prozedur terminiert und dass

$$A'[1] \leq A'[2] \leq \cdots \leq A'[n] \tag{2.3}$$

gilt, mit $n = A.\mathit{länge}$. Was müssen wir außerdem beweisen, um zu zeigen, dass BUBBLESORT tatsächlich sortiert?

Die nächsten beiden Punkte beweisen die Ungleichung (2.3).

b. Geben Sie eine Schleifeninvariante für die **for**-Schleife in den Zeilen 2–4 an und beweisen Sie, dass die Schleifeninvariante erhalten bleibt. Ihr Beweis sollte wie die in diesem Kapitel gezeigten Beweise von Schleifeninvarianten aufgebaut sein.

c. Geben Sie unter Verwendung der Abbruchbedingung für die in (b) bewiesene Schleifeninvariante eine Schleifeninvariante für die **for**-Schleife in den Zeilen 1–4 an, die es Ihnen erlaubt, die Ungleichung (2.3) zu beweisen. Wiederum sollte Ihr Beweis wie die in diesem Kapitel gezeigten Beweise von Schleifeninvarianten aufgebaut sein.

d. Was ist die Laufzeit von Sortieren durch Vertauschen im schlechtesten Fall? Wie steht diese Laufzeit im Vergleich zu der von Sortieren durch Einfügen?

2-3 *Die Korrektheit des Horner-Schemas*

Das folgende Codestück führt das Horner-Schema ein, um ein Polynom

$$\begin{aligned} P(x) &= \sum_{k=0}^{n} a_k x^k \\ &= a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + x a_n) \cdots)) \end{aligned}$$

bei gegebenen Koeffizienten $a_0, a_1, \ldots, a_n$ und einem gegebenen Wert für x auszuwerten:

```
1  y = 0
2  for i = n downto 0
3      y = a_i + x · y
```

a. Welche Laufzeit (in Θ-Notation) hat dieses Codestück für das Horner-Schema?

b. Schreiben Sie ein Programm in Pseudocode, das den naiven Algorithmus zur Polynomauswertung, der jeden Term des Polynoms von Grund auf neu berechnet, implementiert. Wie groß ist die Laufzeit dieses Algorithmus? Wie ist sie im Vergleich zu der des Horner-Schemas?

c. Betrachten Sie die folgende Schleifeninvariante:

Zu Beginn jeder Iteration der **for**-Schleife in den Zeilen 2–3 gilt

$$y = \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k \ .$$

Interpretieren Sie eine Summenformel, die keine Terme enthält, als den Wert 0. Wenden Sie die Schleifenvariante an, um nach dem Vorbild der in diesem Kapitel bereits gezeigten Schleifeninvariantenbeweise zu beweisen, dass nach Abbruch der Schleife $y = \sum_{k=0}^{n} a_k \, x^k$ gilt.

d. Schließen Sie den Beweis ab, indem Sie erörtern, warum das gegebene Codestück ein Polynom, das durch die Koeffizienten $a_0, a_1, \ldots, a_n$ bestimmt ist, korrekt berechnet.

2-4 Inversionen

Sei $A[1\,..\,n]$ ein Feld mit n verschiedenen Zahlen. Wenn $i < j$ und $A[i] > A[j]$ gelten, dann wird das Paar (i, j) als ***Inversion*** von A bezeichnet.

a. Geben Sie die fünf Inversionen des Feldes $\langle 2, 3, 8, 6, 1 \rangle$ an.

b. Welches Feld mit Elementen aus der Menge $\{1, 2, \ldots, n\}$ besitzt die meisten Inversionen? Wie viele Inversionen sind in diesem Feld enthalten?

c. Wie ist die Beziehung zwischen der Laufzeit von Sortieren durch Einfügen und der Anzahl der Inversionen im Eingabefeld? Begründen Sie Ihre Antwort.

d. Geben Sie einen Algorithmus an, der die Anzahl der Inversionen einer Permutation von n Elementen bestimmt und dessen Laufzeit im schlechtesten Fall in $\Theta(n \lg n)$ liegt. (Hinweis: Modifizieren Sie Sortieren durch Mischen.)

Kapitelbemerkungen

Im Jahre 1968 veröffentlichte Knuth den ersten von drei Bänden mit dem allgemeinen Titel *The Art of Computer Programming* [209, 210, 211]. Der erste Band führte in das zeitgemäße Studium der Computeralgorithmen unter dem Gesichtspunkt der Analyse der Laufzeit ein. Die gesamte Ausgabe bleibt eine verbindliche und wertvolle Referenz für viele der hier vorgestellten Themen. Nach Knuth ist das Wort „Algorithmus“ vom Namen des Persischen Mathematikers „al-Khowârizmî“ aus dem neunten Jahrhundert abgeleitet.

Aho, Hopcroft und Ullman [5] verwenden die asymptotische Analyse von Algorithmen als Mittel zum Vergleich der relativen Performanz von Algorithmen – sie verwendeten

hierbei Notationen, die Kapitel 3 einführt, insbesondere Θ-Notation. Sie machten auch die Verwendung von Rekursionsgleichungen bekannt, um die Laufzeiten von rekursiven Algorithmen zu bestimmen.

Knuth [211] liefert eine enzyklopädische Betrachtung vieler Sortieralgorithmen. Sein Vergleich von Sortieralgorithmen (Seite 381) schließt eine exakte schrittzählende Analyse, ähnlich der hier für Sortieren durch Einfügen durchgeführten, ein. Knuths Diskussion des Sortierens durch Einfügen umfasst verschiedene Variationen des Algorithmus. Die wichtigste von ihnen ist das Sortieren nach Shell, eingeführt von D. L. Shell, der Sortieren durch Einfügen auf periodische Teilsequenzen der Eingabe anwendet, was zu einem schnelleren Sortieralgorithmus führt.

Sortieren durch Mischen wird von Knuth ebenfalls beschrieben. Er erwähnt, dass 1938 ein mechanischer Kartenmischer erfunden wurde, der in der Lage war, zwei Kartenstapel aus Lochkarten in einem Durchlauf zu mischen. J. von Neumann, einer der Pioniere der Informatik, entwickelte 1945 offenbar ein Programm für das Sortieren durch Mischen auf dem EDVAC-Computer.

Die frühe Geschichte der Korrektheitsbeweise von Programmen wird durch Gries [153] beschrieben, der P. Naur für den ersten Artikel auf diesem Gebiet würdigt. Gries schreibt R. W. Floyd die Einführung der Schleifeninvarianten zu. Das Lehrbuch von Mitchell [256] beschreibt neueste Fortschritte auf dem Gebiet der Korrektheitsbeweise von Programmen.

3 Wachstum von Funktionen

Die in Kapitel 2 definierte Wachstumsrate der Laufzeit eines Algorithmus stellt ein einfaches Kriterium für dessen Effizienz dar und erlaubt es uns, die relative Leistungsfähigkeit alternativer Algorithmen zu vergleichen. Ist die Eingabegröße n hinreichend groß, dann schlägt Sortieren durch Mischen mit seiner Laufzeit von $\Theta(n \lg n)$ im schlechtesten Fall das Sortieren durch Einfügen, dessen Laufzeit im schlechtesten Fall $\Theta(n^2)$ ist. Obwohl wir manchmal die exakte Laufzeit eines Algorithmus bestimmen können – siehe z. B. Kapitel 2, in dem wir dies für Sortieren durch Einfügen getan haben –,ist die zusätzliche Genauigkeit den Aufwand, sie zu berechnen, nicht wert. Für ausreichend große Eingaben werden die multiplikativen Konstanten und Terme niedrigerer Ordnung der exakten Laufzeit von den Effekten der Eingabegröße selbst dominiert.

Wenn wir Eingaben betrachten, die hinreichend groß sind, dass nur die Ordnung des Wachstums der Laufzeit relevant ist, dann untersuchen wir die ***asymptotische*** Effizienz eines Algorithmus. Das heißt, wir beschäftigen uns damit, wie die Laufzeit eines Algorithmus mit der Größe der Eingabe *im Limes* zunimmt, wenn die Eingabe ohne Beschränkung anwächst. Gewöhnlich wird ein asymptotisch effizienterer Algorithmus bei allen Eingaben, eventuell mit Ausnahme der kleinen Eingaben, die beste Wahl sein.

Dieses Kapitel gibt verschiedene Standardmethoden zur Vereinfachung der asymptotischen Analyse von Algorithmen an. Der nächste Abschnitt beginnt mit der Definition verschiedener Typen von „asymptotischer Notation“, von denen wir bereits ein Beispiel in Θ-Notation kennen gelernt haben. Wir stellen dann verschiedene Bezeichnungskonventionen vor, die durchgehend im Buch verwendet werden. Abschließend geben wir einen Überblick über das Verhalten von Funktionen, die bei der Analyse von Algorithmen häufig auftreten.

3.1 Asymptotische Notation

Die Notationen, die wir verwenden, um die asymptotischen Laufzeiten von Algorithmen zu beschreiben, werden mithilfe von Funktionen formuliert, deren Definitionsbereich die Menge der natürlichen Zahlen $\mathbf{N} = \{0, 1, 2, \ldots\}$ ist. Diese Bezeichnungen sind für die Beschreibung der Funktionen $T(n)$, die die Laufzeit im schlechtesten Fall beschreiben und gewöhnlich nur für ganzzahlige Eingabegrößen definiert sind, geeignet. Wir finden es manchmal zweckmäßig, die asymptotische Notation auf verschiedene Art und Weisen *zu missbrauchen.* Wir dehnen an einigen Stellen die Notation z. B. auf den Definitionsbereich der reellen Zahlen aus, oder schränken sie auf eine Teilmenge der natürlichen Zahlen ein. Wir sollten uns aber sicher sein, dass wir die exakte Bedeutung der Notation verstehen, damit wir sie nicht *falsch anwenden*, wenn wir sie missbräuchlich verwenden. Dieser Abschnitt definiert die grundlegenden asymptotischen Notationen und führt auch

häufig benutzte missbräuchliche Anwendungen von ihnen ein.

Asymptotische Notationen, Funktionen und Laufzeiten

Wir nutzen asymptotische Notationen hauptsächlich, um Laufzeiten von Algorithmen anzugeben, so wie wir schon in einem der vorherigen Kapiteln angegeben haben, dass die Laufzeit von Sortieren durch Einfügen im schlechtesten Fall gleich $\Theta(n^2)$ ist. Asymptotische Notationen werden aber eigentlich auf Funktionen angewendet. Erinnern Sie sich daran, dass wir die Laufzeit von Sortieren durch Einfügen im schlechtesten Fall durch $a\,n^2 + b\,n + c$ für geeignete Konstanten a, b und c charakterisiert haben. Die Laufzeit von Sortieren durch Einfügen als $\Theta(n^2)$ anzugeben, bedeutet, dass wir von einigen Details der Funktion abstrahiert haben. Da asymptotische Notationen auf Funktionen angewendet werden, haben wir mit $\Theta(n^2)$ die Funktion $a\,n^2 + b\,n + c$ charakterisiert, die in diesem Fall zufälligerweise für die Laufzeit von Sortieren durch Einfügen im schlechtesten Fall steht.

In diesem Buch werden die Funktionen, auf die wir asymptotische Notationen anwenden werden, üblicherweise Laufzeiten von Algorithmen charakterisieren. Asymptotische Notationen können jedoch auch auf Funktionen angewendet werden, die andere Aspekte eines Algorithmus (z. B. die Menge an Platz, die er benötigt) beschreiben, oder sogar überhaupt nichts mit Algorithmen zu tun haben.

Auch dann wenn wir asymptotische Notationen auf Laufzeiten von Algorithmen anwenden, müssen wir verstehen, *welche* Laufzeit wir meinen. In einigen Fällen sind wir an der Laufzeit im schlechtesten Fall interessiert. Oft wollen wir aber auch die Laufzeit für eine beliebige Eingabe, welche das auch immer sein mag, charakterisieren. In anderen Worten formuliert: wir wollen oft eine umfassende Aussage haben, die über jede Eingabe etwas aussagt und nicht nur über die schlechtesten. Wir werden sehen, dass asymptotische Notationen auch hierfür gut geeignet sind.

Θ-Notation

Im Kapitel 2 haben wir herausgefunden, dass die Laufzeit $T(n)$ von Sortieren durch Einfügen im schlechtesten Fall $\Theta(n^2)$ ist. Lassen Sie uns definieren, was diese Notation bedeutet. Für eine gegebene Funktion g bezeichnen wir mit $\Theta(g)$ die *Menge der Funktionen*

$$\Theta(g) = \{f : \text{es existieren positive Konstanten } c_1,\ c_2, \text{ und } n_0, \text{ sodass } 0 \leq c_1\, g(n) \leq f(n) \leq c_2\, g(n) \text{ für alle } n \geq n_0\}\ .$$ [1] [2]

Eine Funktion f gehört zur Menge $\Theta(g)$, wenn positive Konstanten c_1 und c_2 existieren, sodass $f(n)$ zwischen $c_1\, g(n)$ und $c_2\, g(n)$ für hinreichend große n eingeschlossen werden kann. Da $\Theta(g)$ eine Menge ist, könnten wir „$f \in \Theta(g)$“ schreiben, um deutlich zu machen, dass f ein Element von $\Theta(g)$ ist. Stattdessen werden wir gewöhnlich „$f = \Theta(g)$“

[1] Innerhalb der Mengendefinition sollte ein Doppelpunkt als „sodass“ gelesen werden.

[2] Bemerkung des Übersetzers: Der Formalismus wurde an dieser Stelle gegenüber dem englischsprachigen Buch geringfügig geändert.

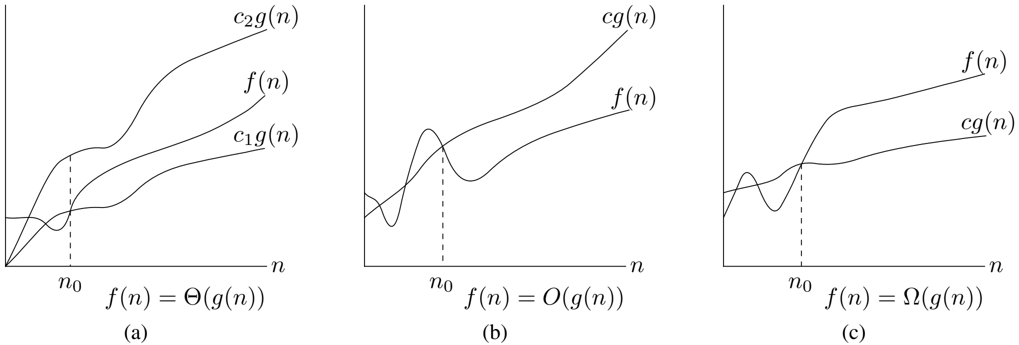


Abbildung 3.1: *Grafische Beispiele für die Θ-, O- und Ω-Notation. In jeder Teilabbildung ist illustriert, wie klein der Wert n_0 höchstens gewählt werden kann; jeder größere Wert wäre auch möglich.* ***(a)*** *Die Θ-Notation schränkt eine Funktion bis auf konstante Faktoren ein. Wir schreiben $f(n) = \Theta(g(n))$, wenn positive Konstanten n_0, c_1 und c_2 existieren, sodass ab der Stelle n_0 der Wert von $f(n)$ immer zwischen $c_1\,g(n)$ und $c_2\,g(n)$ liegt.* ***(b)*** *Die O-Notation liefert eine obere Schranke (bis auf einen konstanten Faktor) für eine Funktion. Wir schreiben $f(n) = O(g(n))$, wenn positive Konstanten n_0 und c existieren, sodass ab der Stelle n_0 der Wert von $f(n)$ immer unterhalb von $c\,g(n)$ liegt.* ***(c)*** *Die Ω-Notation liefert eine untere Schranke (bis auf einen konstanten Faktor) einer Funktion. Wir schreiben $f(n) = \Omega(g(n))$, wenn positive Konstanten n_0 und c existieren, sodass ab der Stelle n_0 die Werte von $f(n)$ immer oberhalb von $c\,g(n)$ liegen.*

schreiben, um den gleichen Sachverhalt zu beschreiben, oder sogar „$f(n) = \Theta(g(n))$“, um deutlich zu machen, dass die Funktionen über der Variable n definiert sind. Sie sind vielleicht verwirrt, dass wir das Gleichheitszeichen in dieser Art und Weise missbrauchen, aber wir werden in diesem Abschnitt noch sehen, dass diese Notation Vorteile hat.

Abbildung 3.1(a) gibt ein intuitives Bild der Funktionen $f(n)$ und $g(n)$ für den Fall $f(n) = \Theta(g(n))$. Für n gleich n_0 und alle rechts von n_0 liegenden Werte n liegen die Werte von $f(n)$ über $c_1\,g(n)$ (genauer: $f(n) \geq c_1\,g(n)$) und unter $c_2\,g(n)$ (genauer: $f(n) \leq c_2\,g(n)$). Mit anderen Worten: die Funktion $f(n)$ ist bis auf einen konstanten Faktor für alle $n \geq n_0$ gleich $g(n)$. Wir sagen, dass $g(n)$ eine ***asymptotisch scharfe Schranke*** von $f(n)$ ist.

Die Definition von $\Theta(g)$ fordert, dass jedes Element $f \in \Theta(g)$ ***asymptotisch nichtnegativ*** ist, d. h. dass $f(n)$ für hinreichend große n nichtnegativ ist. (Eine ***asymptotisch positive*** Funktion ist eine Funktion, die für hinreichend große n positiv ist.) Folglich muss die Funktion g selbst asymptotisch nichtnegativ sein, anderenfalls ist die Menge $\Theta(g)$ leer. Wir werden aus diesem Grund annehmen, dass jede Funktion innerhalb der Θ-Notation asymptotisch nichtnegativ ist. Diese Annahme trifft ebenso auf die anderen in diesem Kapitel eingeführten Notationen zu.

Im Kapitel 2 haben wir eine formlose Version der Θ-Notation eingeführt, die darin bestand, die Terme niedrigerer Ordnung zu vernachlässigen und den führenden Koeffizienten des Terms mit der höchsten Ordnung zu ignorieren. Lassen Sie uns kurz diese

heuristische Herangehensweise rechtfertigen, indem wir die formale Definition benutzen, um zu zeigen, dass $\frac{1}{2}n^2 - 3n = \Theta(n^2)$ erfüllt ist. Dafür müssen wir positive Konstanten c_1, c_2 und n_0 bestimmen, sodass

$$c_1\, n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2\, n^2$$

für alle $n \geq n_0$ gilt. Division durch n^2 ergibt

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2 \ .$$

Wir können die Ungleichung auf der rechten Seite für jeden Wert $n \geq 1$ erfüllen, indem wir für c_2 eine beliebige Konstante größer gleich 1/2 wählen. Entsprechend können wir die Ungleichung auf der linken Seite für jedes $n \geq 7$ erfüllen, indem wir für c_1 eine beliebige Konstante kleiner gleich 1/14 wählen. Indem wir $c_1 = 1/14$, $c_2 = 1/2$ und $n_0 = 7$ wählen, können wir überprüfen, dass $\frac{1}{2}n^2 - 3n = \Theta(n^2)$ gilt. Natürlich gibt es für die Konstanten auch andere Wahlmöglichkeiten, aber der wesentliche Punkt ist, dass *irgendeine* Wahlmöglichkeit existiert. Beachten Sie, dass diese Konstanten von der Funktion $\frac{1}{2}n^2 - 3n$ abhängen; eine andere zu $\Theta(n^2)$ gehörige Funktion würde normalerweise andere Konstanten erfordern.

Wir können auch die formale Definition benutzen, um nachzuprüfen, dass $6n^3 \neq \Theta(n^2)$ gilt. Zum indirekten Beweis nehmen wir an, dass c_2 und n_0 existieren, sodass $6n^3 \leq c_2\, n^2$ für alle $n \geq n_0$ erfüllt ist. Mit Division durch n^2 folgt dann aber $n \leq c_2/6$, was für hinreichend große n doch nicht gilt, da c_2 konstant ist.

Intuitiv können die Terme niedrigerer Ordnung einer asymptotisch positiven Funktion ignoriert werden, wenn asymptotisch scharfe Schranken bestimmt werden. Terme niedrigerer Ordnung sind für große n nicht signifikant. Für große n reicht ein winziger Bruchteil des Terms mit der höchsten Ordnung aus, um die Terme niedrigerer Ordnung zu dominieren. Wählen wir für c_1 einen Wert, der etwas kleiner ist als der Koeffizient des Terms höchster Ordnung und für c_2 einen etwas höheren Wert, dann sind damit die Ungleichungen in der Definition der Θ-Notation erfüllt. Der Koeffizient des Terms höchster Ordnung kann auch ignoriert werden, da er c_1 und c_2 seinem Wert entsprechend nur um einen konstanten Faktor ändert.

Als Beispiel betrachten wir eine quadratische Funktion $f(n) = a\,n^2 + b\,n + c$, wobei a, b und c Konstanten sind und $a > 0$ gilt. Vernachlässigen der Terme niedrigerer Ordnung und Ignorieren der Konstanten liefert $f(n) = \Theta(n^2)$. Um dies formal zu zeigen, nehmen wir die Konstanten $c_1 = a/4$, $c_2 = 7a/4$ und $n_0 = 2 \cdot \max((|b|/a), \sqrt{(|c|/a)})$. Sie können überprüfen, dass für alle $n \geq n_0$ die Ungleichung $0 \leq c_1\, n^2 \leq a\,n^2 + b\,n + c \leq c_2\, n^2$ erfüllt ist. Allgemeinen gilt $p(n) = \Theta(n^d)$ für jedes Polynom $p(n) = \sum_{i=0}^{d} a_i\, n^i$, wenn a_i Konstanten sind und $a_d > 0$ gilt (siehe Problemstellung 3-1).

Da jede Konstante ein Polynom vom Grad 0 ist, können wir jede Konstante als $\Theta(n^0)$ oder $\Theta(1)$ angeben. Letztere Bezeichnung stellt jedoch einen geringfügigen Missbrauch dar, da der Ausdruck nicht angibt, über welche Variable die Funktion definiert ist.[3] Wir werden häufig die Bezeichnung $\Theta(1)$ benutzen, womit wir entweder eine Konstante oder eine Funktion meinen, die bezüglich einer Variablen konstant ist.

[3] Das eigentliche Problem ist, dass unsere einfache Notation Funktionen nicht von Werten unter-

$\boldsymbol{O}$-Notation

Die Θ-Notation beschränkt eine Funktion asymptotisch von oben und unten. Wenn wir nur über eine ***obere asymptotische Schranke*** verfügen, benutzen wir die O-Notation. Bei einer gegebenen Funktion $g(n)$ bezeichnen wir mit $O(g(n))$ (ausgesprochen als „groß O von g von n“ oder manchmal einfach als „O von g von n“) die Menge der Funktionen

$$O(g(n)) = \{f(n)\colon \text{es existieren positive Konstanten } c \text{ und } n_0, \text{ sodass } 0 \leq f(n) \leq c\, g(n) \text{ für alle } n \geq n_0\}\,.$$ [4]

Wir benutzen die O-Notation, um bis auf einen konstanten Faktor eine obere Schranke einer Funktion anzugeben. Abbildung 3.1(b) zeigt die Idee hinter der O-Notation. Für n gleich n_0 und alle rechts von n_0 liegenden Werte n ist der Wert von $f(n)$ kleiner gleich $c\, g(n)$.

Wir schreiben $f(n) = O(g(n))$, um anzugeben, dass $f(n)$ ein Element der Menge $O(g(n))$ ist. Beachten Sie, dass aus $f(n) = \Theta(g(n))$ auch $f(n) = O(g(n))$ folgt, da die Θ-Notation stärker als die O-Notation ist. In der Sprache der Mengenlehre haben wir $\Theta(g(n)) \subseteq O(g(n))$. Somit zeigt unser Beweis zu $a\,n^2 + b\,n + c = \Theta(n^2)$, mit $a > 0$, dass diese quadratische Funktion auch in $O(n^2)$ liegt. Was überraschender sein mag, ist, dass jede *lineare* Funktion $a\,n + b$ mit $a > 0$ in $O(n^2)$ liegt. Dies können wir aber leicht nachprüfen, indem man $c = a + |b|$ und $n_0 = \max\{1, -\frac{b}{a}\}$ setzen.

Wenn Sie bereits mit der O-Notation vertraut sind, finden Sie es möglicherweise ungewöhnlich, dass wir beispielsweise $n = O(n^2)$ schreiben. In der Literatur finden wir zuweilen die O-Notation, um asymptotisch scharfe Schranken anzugeben; dies entspricht unserer Definition der Θ-Notation. In diesem Buch behaupten wir mit $f(n) = O(g(n))$ jedoch lediglich, dass ein konstantes Vielfaches von $g(n)$ eine asymptotisch obere Schranke von $f(n)$ ist, ohne zu zeigen, wie scharf diese obere Schranke ist. Das Unterscheiden asymptotisch oberer Schranken von asymptotisch scharfen Schranken ist Standard in der Literatur zu Algorithmen.

Mit der O-Notation können wir die Laufzeit eines Algorithmus oft angeben, indem wir lediglich die Gesamtstruktur des Algorithmus untersuchen. Zum Beispiel liefert die doppelt verschachtelte Schleifenstruktur des Algorithmus Sortieren durch Einfügen aus Kapitel 2 sofort eine obere Schranke von $O(n^2)$ für die Laufzeit im schlechtesten Fall: Die Kosten jeder Iteration der inneren Schleife sind von oben durch $O(1)$ (konstant) beschränkt, die Indizes i und j haben höchstens den Wert n, und die innere Schleife wird höchstens einmal für jedes der n^2 Paare der Werte von i und j ausgeführt.

Weil die O-Notation eine obere Schranke darstellt, wenn wir sie anwenden, um die Laufzeit im schlechtesten Fall zu beschränken, haben wir eine Schranke für die Laufzeit

---

scheidet. Innerhalb des λ-Kalküls werden Parameter einer Funktion klar spezifiziert: Die Funktion n^2 könnte als $\lambda n.n^2$ oder sogar als $\lambda r.r^2$ geschrieben werden. Die Annahme einer strengeren Notation würde jedoch die algebraischen Manipulationen komplizieren, weshalb wir den eigentlich unzulässigen Gebrauch tolerieren.

[4] Bemerkung des Übersetzers: Nachdem wir im Abschnitt zu der Θ-Notation eine mathematisch saubere Definition angegeben haben, wollen wir es im Folgenden bei der einfacheren Formulierung aus dem englischsprachigen Buch belassen, in der nicht zwischen einer Funktion und einem Funktionswert unterschieden wird. Aus dem Kontext sollte jedoch jeweils klar sein, was an welcher Stelle gemeint ist.

des Algorithmus für jede Eingabe – also eine umfassende Aussage, wie vorhin diskutiert. Folglich gilt die $O(n^2)$-Schranke für die Laufzeit von Sortieren durch Einfügen im schlechtesten Fall auch für dessen Laufzeit bei beliebigen Eingaben. Die $\Theta(n^2)$-Schranke für die Laufzeit für Sortieren durch Einfügen im schlechtesten Fall impliziert jedoch keine $\Theta(n^2)$-Schranke für die Laufzeit von Sortieren durch Einfügen für *jede* Eingabe. Wir haben in Kapitel 2 beispielsweise gesehen, dass die Laufzeit von Sortieren durch Einfügen in $\Theta(n)$ ist, wenn die Eingabe bereits sortiert ist.

Technisch gesehen ist es an sich missbräuchlich zu sagen, dass die Laufzeit von Sortieren durch Einfügen in $O(n^2)$ ist, da für ein gegebenes n die tatsächliche Laufzeit abhängig von der speziellen Eingabe der Größe n variiert. Wenn wir sagen „die Laufzeit ist in $O(n^2)$“, meinen wir, dass es eine Funktion $f(n)$ in $O(n^2)$ gibt, sodass für jeden Wert von n, ungeachtet der speziell gewählten Eingabe der Größe n, die Laufzeit für diese Eingabe von oben durch den Wert $f(n)$ beschränkt ist. Entsprechend meinen wir, dass die Laufzeit im schlechtesten Fall in $O(n^2)$ liegt.

$\boldsymbol{\Omega}$-Notation

Ebenso wie die O-Notation eine asymptotische *obere* Schranke einer Funktion liefert, liefert die Ω-Notation eine ***asymptotische untere Schranke***. Wenn eine Funktion $g(n)$ gegeben ist, dann bezeichnen wir mit $\Omega(g(n))$ (ausgesprochen als „groß Omega von g von n“ oder manchmal einfach „Omega von g von n“) die Menge der Funktionen

$$\{f(n)\colon \text{es existieren positive Konstanten } c \text{ und } n_0, \text{ sodass} \\ 0 \le c\, g(n) \le f(n) \text{ für alle } n \ge n_0\} \ .$$

Die Idee hinter der Ω-Notation wird in Abbildung 3.1(c) veranschaulicht. Für n gleich n_0 und alle rechts von n_0 liegenden Werte n ist der Wert von $f(n)$ größer gleich $c\, g(n)$.

Ausgehend von den Definitionen für die asymptotischen Notationen, die wir bisher kennen gelernt haben, ist es einfach, das folgende wichtige Theorem zu beweisen (siehe Übung 3.1-5).

Theorem 3.1

Für zwei beliebige Funktionen $f(n)$ und $g(n)$ gilt $f(n) = \Theta(g(n))$ genau dann, wenn die Gleichungen $f(n) = O(g(n))$ und $f(n) = \Omega(g(n))$ erfüllt sind.

Als Anwendungsbeispiel dieses Theorems impliziert unser Beweis zu $a\,n^2 + b\,n + c = \Theta(n^2)$ für alle Konstanten a, b und c mit $a > 0$, dass $a\,n^2 + b\,n + c = \Omega(n^2)$ und $a\,n^2 + b\,n + c = O(n^2)$ gilt. Wir haben in diesem Beispiel das Theorem benutzt, um aus asymptotisch scharfen Schranken asymptotisch untere und obere Schranken zu erhalten. In der Praxis benutzen wir Theorem 3.1 gewöhnlich eher dazu, asymptotisch scharfe Schranken ausgehend von asymptotisch unteren und oberen Schranken zu beweisen.

Wenn wir sagen, dass die *Laufzeit* (ohne den Laufzeitbegriff in welcher Form auch immer zu beschränken) eines Algorithmus in $\Omega(g(n))$ liegt, meinen wir, dass für jeden ausreichend großen Wert von n die Laufzeit des Algorithmus angewendet auf *egal welche*

spezielle Eingabe der Größe n bis auf einen konstanten Faktor größer gleich $g(n)$ ist. Dies ist gleichbedeutend dazu, dass wir eine untere Schranke für die Laufzeit eines Algorithmus im günstigsten Fall angeben. Beispielsweise ist die Laufzeit von Sortieren durch Einfügen im günstigsten Fall in $\Omega(n)$, woraus sich ergibt, dass die Laufzeit von Sortieren durch Einfügen in $\Omega(n)$ liegt.

Die Laufzeit von Sortieren durch Einfügen ist demnach sowohl in $\Omega(n)$ als auch in $O(n^2)$, da sie an allen Stellen zwischen einer in n linearen Funktion und einer in n quadratischen Funktion liegt. Darüber hinaus sind diese Schranken so scharf wie möglich: Beispielsweise ist die Laufzeit von Sortieren durch Einfügen nicht in $\Omega(n^2)$, da eine Eingabe existiert, für die Sortieren durch Einfügen in $\Theta(n)$ läuft (zum Beispiel wenn die Eingabe bereits sortiert ist). Es ist jedoch kein Widerspruch zu sagen, dass die Laufzeit von Sortieren durch Einfügen im schlechtesten Fall Zeit $\Omega(n^2)$ benötigt, denn es gibt eine Eingabe, die bewirkt, dass der Algorithmus eine Zeit der Ordnung $\Omega(n^2)$ benötigt.

Asymptotische Notation in Gleichungen und Ungleichungen

Wir haben bereits gesehen, wie die asymptotische Notation in mathematischen Formeln benutzt wird. Zum Beispiel haben wir bei der Einführung der O-Notation „$n = O(n^2)$" geschrieben. Wir könnten auch schreiben $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$. Wie interpretieren wir solche Formeln?

Wenn die asymptotische Notation auf der rechten Seite einer Gleichung (oder Ungleichung) allein (d. h. nicht als Teil eines größeren Ausdrucks) steht, wie zum Beispiel in $n = O(n^2)$, dann haben wir bereits definiert, dass das Gleichheitszeichen die Zugehörigkeit zu einer Menge bezeichnet, also in unserem Falle $n \in O(n^2)$. Tritt die asymptotische Notation jedoch in einer Formel auf, dann interpretieren wir sie als Platzhalter für eine anonyme Funktion, die wir nicht näher benennen wollen. Die Formel $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ zum Beispiel bedeutet, dass $2n^2 + 3n + 1 = 2n^2 + f(n)$ für eine bestimmte Funktion $f(n)$ aus der Menge $\Theta(n)$ gilt. In diesem Beispiel gilt $f(n) = 3n + 1$, die tatsächlich von der Ordnung $\Theta(n)$ ist.

Die Verwendung der asymptotischen Notation kann auf diese Weise dazu beitragen, unwesentliche Details und Wirrwarr zu eliminieren. Zum Beispiel haben wir in Kapitel 2 die Laufzeit von Sortieren durch Mischen im schlechtesten Fall durch die Rekursionsgleichung

$$T(n) = 2\,T(n/2) + \Theta(n)$$

ausgedrückt. Wenn wir nur am asymptotischen Verhalten von $T(n)$ interessiert sind, dann gibt es keinen Grund dafür, alle Terme niedrigerer Ordnung genau zu spezifizieren; wir betrachten sie alle als in der anonymen Funktion $\Theta(n)$ enthalten.

Die Anzahl der anonymen Funktionen in einem Ausdruck ist dadurch gegeben, wie oft die asymptotische Notation in dem Ausdruck auftritt. In dem Ausdruck

$$\sum_{i=1}^{n} O(i)$$

zum Beispiel gibt es nur eine anonyme Funktion (eine Funktion von i). Dieser Ausdruck ist deshalb *nicht* gleich $O(1) + O(2) + \cdots + O(n)$, wofür es tatsächlich keine klare Interpretation gibt.

In einigen Fällen tritt die asymptotische Notation auf der linken Seite einer Gleichung auf, wie im Ausdruck

$$2n^2 + \Theta(n) = \Theta(n^2) .$$

Wir interpretieren solche Gleichungen unter Verwendung der folgenden Regel: *Unabhängig davon, wie die anonymen Funktionen auf der linken Seite des Gleichheitszeichens gewählt werden, ist es möglich, die anonymen Funktionen rechts vom Gleichheitszeichen so zu wählen, dass Gleichheit gilt.* Somit bedeutet unser Beispiel, dass es für *jede* Funktion $f(n) \in \Theta(n)$ eine Funktion $g(n) \in \Theta(n^2)$ gibt, sodass $2n^2 + f(n) = g(n)$ für alle n gilt. Anders formuliert: die rechte Seite einer Gleichung bietet eine gröbere Detailsicht als die linke.

Wir können mehrere dieser Beziehungen hintereinander schreiben:

$$\begin{aligned} 2n^2 + 3n + 1 &= 2n^2 + \Theta(n) \\ &= \Theta(n^2) . \end{aligned}$$

Wir können jede Gleichung unter den oben genannten Regeln für sich interpretieren. Die erste Gleichung besagt, dass es *eine* Funktion $f(n) \in \Theta(n)$ gibt, sodass $2n^2 + 3n + 1 = 2n^2 + f(n)$ für alle n erfüllt ist. Die zweite Gleichung sagt aus, dass es für *jede* Funktion $g(n) \in \Theta(n)$ (wie die bereits benutzte Funktion $f(n)$) *eine* Funktion $h(n) \in \Theta(n^2)$ gibt, sodass $2n^2 + g(n) = h(n)$ für alle n gilt. Beachten Sie, dass aus der Hintereinanderausführung der Gleichungen die Gleichung $2n^2 + 3n + 1 = \Theta(n^2)$ folgt.

o-Notation

Die durch die O-Notation festgelegte asymptotisch obere Schranke kann asymptotisch scharf sein, muss aber nicht. Die Schranke $2n^2 = O(n^2)$ ist asymptotisch scharf, während es die Schranke $2n = O(n^2)$ nicht ist. Wir verwenden die o-Notation zur Bezeichnung einer oberen Schranke, die nicht asymptotisch scharf ist. Wir definieren $o(g(n))$ („klein-oh von g von n“) formal als die Menge

$$o(g(n)) = \{f(n) \colon \text{für jede positive Konstante } c > 0 \text{ existiert ein konstantes } n_0 > 0, \text{ sodass } 0 \leq f(n) < c\,g(n) \text{ für alle } n \geq n_0\} .$$

Zum Beispiel gilt $2n = o(n^2)$ und $2n^2 \neq o(n^2)$.

Die Definitionen der O-Notation und der o-Notation sind einander ähnlich. Der wesentliche Unterschied besteht darin, dass in $f(n) = O(g(n))$ die Schranke $0 \leq f(n) \leq c\,g(n)$ für *eine* Konstante $c > 0$ gilt, während in $f(n) = o(g(n))$ die Schranke $0 \leq f(n) < c\,g(n)$ für *alle* Konstanten $c > 0$ gilt. Intuitiv ist klar, dass in der o-Notation die Funktion $f(n)$ unbedeutend gegenüber $g(n)$ wird, wenn n gegen Unendlich geht; das heißt

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 . \tag{3.1}$$

Einige Autoren verwenden diesen Limes zur Definition der o-Notation; die Definition in diesem Buch fordert dagegen auch, dass die anonymen Funktionen asymptotisch nichtnegativ sind.

$\boldsymbol{\omega}$-Notation

Was die o-Notation im Vergleich zur O-Notation ist, ist die ω-Notation für die Ω-Notation. Wir benutzen die ω-Notation, um eine untere Schranke anzugeben, die nicht asymptotisch scharf ist. Ein Weg, diese Notation einzuführen, ist

$$f(n) \in \omega(g(n)) \text{ genau dann, wenn } g(n) \in o(f(n)) \ .$$

Formal haben wir $\omega(g(n))$ („klein-omega von g von n“) jedoch als die Menge

$$\omega(g(n)) = \{f(n)\colon \text{für jede positive Konstante } c > 0 \text{ existiert eine Konstante } n_0 > 0, \text{ sodass } 0 \leq c\, g(n) < f(n) \text{ für alle } n \geq n_0\}$$

zu definieren. Zum Beispiel gilt $\frac{n^2}{2} = \omega(n)$, aber $\frac{n^2}{2} \neq \omega(n^2)$. Die Relation $f(n) = \omega(g(n))$ bedeutet, dass

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

gilt, wenn der Limes existiert. Das heißt, dass $f(n)$ gegenüber $g(n)$ beliebig groß wird, wenn n gegen Unendlich geht.

Vergleich von Funktionen

Viele der Relationseigenschaften reeller Zahlen können auch auf asymptotische Vergleiche angewendet werden. Im Folgenden nehmen wir an, dass $f(n)$ und $g(n)$ asymptotisch positiv sind.

Transitivität:

$$\begin{aligned}
f(n) = \Theta(g(n)) &\text{ und } g(n) = \Theta(h(n)) \text{ impliziert } f(n) = \Theta(h(n)) \ , \\
f(n) = O(g(n)) &\text{ und } g(n) = O(h(n)) \text{ impliziert } f(n) = O(h(n)) \ , \\
f(n) = \Omega(g(n)) &\text{ und } g(n) = \Omega(h(n)) \text{ impliziert } f(n) = \Omega(h(n)) \ , \\
f(n) = o(g(n)) &\text{ und } g(n) = o(h(n)) \text{ impliziert } f(n) = o(h(n)) \ , \\
f(n) = \omega(g(n)) &\text{ und } g(n) = \omega(h(n)) \text{ impliziert } f(n) = \omega(h(n)) \ .
\end{aligned}$$

Reflexivität:

$$\begin{aligned}
f(n) &= \Theta(f(n)) \ , \\
f(n) &= O(f(n)) \ , \\
f(n) &= \Omega(f(n)) \ .
\end{aligned}$$

Symmetrie:

$$f(n) = \Theta(g(n)) \text{ genau dann, wenn } g(n) = \Theta(f(n)) \ .$$

Austausch-Symmetrie:

$$f(n) = O(g(n)) \text{ genau dann, wenn } g(n) = \Omega(f(n)) \ ,$$
$$f(n) = o(g(n)) \text{ genau dann, wenn } g(n) = \omega(f(n)) \ .$$

Da die asymptotischen Notationen diese Eigenschaften erfüllen, kann man eine Analogie zwischen dem asymptotischen Vergleich zweier Funktionen f und g und dem Vergleich von zwei reellen Zahlen a und b ziehen:

$$\begin{array}{lll} f(n) = O(g(n)) & \text{entspricht} & a \leq b \ , \\ f(n) = \Omega(g(n)) & \text{entspricht} & a \geq b \ , \\ f(n) = \Theta(g(n)) & \text{entspricht} & a = b \ , \\ f(n) = o(g(n)) & \text{entspricht} & a < b \ , \\ f(n) = \omega(g(n)) & \text{entspricht} & a > b \ . \end{array}$$

Wir sagen, dass $f(n)$ ***asymptotisch kleiner*** als $g(n)$ ist, wenn $f(n) = o(g(n))$ erfüllt ist und dass $f(n)$ ***asymptotisch größer*** als $g(n)$ ist, wenn $f(n) = \omega(g(n))$ gilt.

Eine Eigenschaft reeller Zahlen kann jedoch nicht auf die asymptotische Notation übertragen werden:

Trichotomie: Für zwei beliebige reelle Zahlen a und b muss exakt eine der folgenden Relationen erfüllt sein: $a < b$, $a = b$ oder $a > b$.

Obwohl jedes beliebige Paar reeller Zahlen verglichen werden kann, sind nicht alle Funktionen asymptotisch vergleichbar. Das heißt, für zwei Funktionen $f(n)$ und $g(n)$ kann der Fall eintreten, dass weder $f(n) = O(g(n))$ noch $f(n) = \Omega(g(n))$ gilt. Beispielsweise können wir die Funktionen n und $n^{1+\sin n}$ nicht mithilfe der asymptotischen Notation vergleichen, da der Wert des Exponenten in $n^{1+\sin n}$ zwischen 0 und 2 oszilliert und alle Zwischenwerte angenommen werden.

Übungen

3.1-1 Seien $f(n)$ und $g(n)$ asymptotisch nichtnegative Funktionen. Beweisen Sie

$$\max(f(n), g(n)) = \Theta(f(n) + g(n))$$

unter Verwendung der Definition der Θ-Notation.

3.1-2 Zeigen Sie, dass für beliebige reelle Konstanten a und b mit $b > 0$

$$(n + a)^b = \Theta(n^b) \tag{3.2}$$

gilt.

3.1-3 Erklären Sie, warum die Aussage „Die Laufzeit eines Algorithmus A beträgt mindestens $O(n^2)$“ keinen Sinn macht.

3.1-4 Gilt $2^{n+1} = O(2^n)$? Gilt $2^{2n} = O(2^n)$?

3.1-5 Beweisen Sie Theorem 3.1.

3.1-6 Beweisen Sie, dass die Laufzeit eines Algorithmus genau dann $\Theta(g(n))$ beträgt, wenn seine Laufzeit im schlechtesten Fall in $O(g(n))$ und seine Laufzeit im günstigsten Fall in $\Omega(g(n))$ liegt.

3.1-7 Beweisen Sie, dass $o(g(n)) \cap \omega(g(n))$ die leere Menge ist.

3.1-8 Wir können unsere Notation auf den Fall von zwei Parametern n und m, die unabhängig voneinander mit verschiedenen Geschwindigkeiten gegen Unendlich gehen, erweitern. Für eine Funktion $g(n, m)$ bezeichnen wir mit $O(g(n, m))$ die Menge der Funktionen

$$\begin{aligned} O(g(n,m)) = \{f(n,m) \colon & \text{ es existieren positive Konstanten } c, \\ & n_0 \text{ und } m_0, \text{ sodass } 0 \le f(n,m) \le c\,g(n,m) \\ & \text{für alle } n \ge n_0 \text{ oder } m \ge m_0\} \ . \end{aligned}$$

Geben Sie entsprechende Definitionen für $\Omega(g(n, m))$ und $\Theta(g(n, m))$ an.

3.2 Standardnotationen und Standardfunktionen

Dieser Abschnitt gibt einen Überblick über einige mathematische Standardfunktionen und Bezeichnungen und untersucht die Beziehungen zwischen ihnen. Er illustriert auch die Verwendung der asymptotischen Notation.

Monotonie

Eine Funktion $f(n)$ ist ***monoton steigend***, wenn $m \le n$ impliziert, dass $f(m) \le f(n)$ gilt. Entsprechend ist sie ***monoton fallend***, wenn aus $m \le n$ folgt, dass $f(m) \ge f(n)$ ist. Eine Funktion $f(n)$ wird als ***streng monoton steigend*** bezeichnet, wenn aus $m < n$ folgt, dass $f(m) < f(n)$ ist. Als ***streng monoton fallend*** wird sie bezeichnet, wenn $m < n$ die Beziehung $f(m) > f(n)$ impliziert.

Aufrunden und Abrunden

Gegeben sei eine beliebige reelle Zahl x. Wir bezeichnen die größte ganze Zahl, die kleiner oder gleich x ist, mit $\lfloor x \rfloor$ (gesprochen „*floor* von x“) und die kleinste ganze Zahl, die größer oder gleich x ist, mit $\lceil x \rceil$ (gesprochen „*ceil* von x“). Für alle reellen Zahlen x gilt

$$x - 1 \ < \ \lfloor x \rfloor \ \le \ x \ \le \ \lceil x \rceil \ < \ x + 1 \ . \tag{3.3}$$

Für beliebige ganze Zahlen n gilt

$$\left\lceil \frac{n}{2} \right\rceil + \left\lfloor \frac{n}{2} \right\rfloor = n \ .$$

Für beliebige reelle Zahlen $n \geq 0$ und ganze Zahlen $a, b > 0$ gelten die Beziehungen

$$\left\lceil \frac{\lceil x/a \rceil}{b} \right\rceil = \left\lceil \frac{x}{ab} \right\rceil \ , \tag{3.4}$$

$$\left\lfloor \frac{\lfloor x/a \rfloor}{b} \right\rfloor = \left\lfloor \frac{x}{ab} \right\rfloor \ , \tag{3.5}$$

$$\left\lceil \frac{a}{b} \right\rceil \leq \frac{a + (b-1)}{b} \ , \tag{3.6}$$

$$\left\lfloor \frac{a}{b} \right\rfloor \geq \frac{a - (b-1)}{b} \ . \tag{3.7}$$

Die *floor*-Funktion $f(x) = \lfloor x \rfloor$ ist monoton steigend, ebenso wie die *ceil*-Funktion $f(x) = \lceil x \rceil$.

Modulare Arithmetik

Für eine beliebige ganze Zahl a und eine beliebige positive ganze Zahl n ist der Wert $a \bmod n$ der ***Rest*** des Quotienten a/n:

$$a \bmod n = a - n \left\lfloor \frac{a}{n} \right\rfloor \ . \tag{3.8}$$

Es folgt

$$0 \leq a \bmod n < n \ . \tag{3.9}$$

Ist eine wohldefinierte Definition des Restes einer Division einer ganzen Zahl durch eine andere gegeben, so ist es zweckmäßig, einige spezielle Bezeichnungen einzuführen, um die Gleichheit von Resten auszudrücken. Wenn $(a \bmod n) = (b \bmod n)$ gilt, dann schreiben wir $a \equiv b \pmod{n}$ und sagen, dass a ***äquivalent*** zu b ***modulo*** n ist. Mit anderen Worten gilt $a \equiv b \pmod{n}$, wenn a und b bei der Division durch n den gleichen Rest haben. Entsprechend gilt $a \equiv b \pmod{n}$ genau dann, wenn n ein Teiler von $b - a$ ist. Wir schreiben $a \not\equiv b \pmod{n}$, wenn a nicht äquivalent zu b modulo n ist.

Polynome

Gegeben sei eine nichtnegative ganze Zahl d. Ein ***Polynom in n vom Grad d*** ist eine Funktion $p(n)$ der Form

$$p(n) = \sum_{i=0}^{d} a_i \, n^i \ ,$$

wobei die Konstanten $a_0, a_1, \ldots, a_d$ als ***Koeffizienten*** des Polynoms bezeichnet werden und $a_d \neq 0$ gilt. Ein Polynom ist demnach asymptotisch positiv, wenn $a_d > 0$ gilt. Für ein asymptotisch positives Polynom $p(n)$ vom Grad d gilt $p(n) = \Theta(n^d)$. Die Funktion n^a ist für beliebige reelle Konstanten $a \geq 0$ monoton steigend und für beliebige reelle Konstanten $a \leq 0$ monoton fallend. Wir sagen, dass eine Funktion $f(n)$ ***polynomiell beschränkt*** ist, wenn für eine Konstante k die Gleichung $f(n) = O(n^k)$ gilt.

Exponentialfunktionen

Für alle reellen Zahlen $a > 0$, m und n gelten folgende Identitäten:

$$\begin{aligned} a^0 &= 1 , \\ a^1 &= a , \\ a^{-1} &= 1/a , \\ (a^m)^n &= a^{mn} , \\ (a^m)^n &= (a^n)^m , \\ a^m a^n &= a^{m+n} . \end{aligned}$$

Für alle n und $a \geq 1$ ist die Funktion a^n monoton steigend in n. Wenn es zweckmäßig ist, nehmen wir $0^0 = 1$ an.

Wir können die Wachstumsraten von Polynomen und Exponentialfunktionen wie folgt zueinander in Beziehung setzen. Für alle reellen Konstanten a und b, mit $a > 1$ gilt

$$\lim_{n \to \infty} \frac{n^b}{a^n} = 0 , \tag{3.10}$$

woraus wir schließen können, dass

$$n^b = o(a^n)$$

gilt. Somit wächst jede Exponentialfunktion, deren Basis echt größer als 1 ist, schneller als jedes Polynom.

Unter Verwendung von $e = 2{,}71828\ldots$, der Basis des natürlichen Logarithmus, erhalten wir für alle reellen Zahlen x

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{i=0}^{\infty} \frac{x^i}{i!} , \tag{3.11}$$

wobei „ ! “ den später noch in diesem Abschnitt definierten Fakultätsoperator bezeichnet. Für alle reellen Zahlen x gilt die Ungleichung

$$e^x \geq 1 + x , \tag{3.12}$$

wobei Gleichheit nur für $x = 0$ gilt. Für $|x| \leq 1$ gilt die Approximation

$$1 + x \leq e^x \leq 1 + x + x^2 . \tag{3.13}$$

Für $x \to 0$ ist die Näherung $1 + x$ für e^x ziemlich gut:

$$e^x = 1 + x + \Theta(x^2) \ .$$

(In dieser Gleichung wird die asymptotische Notation dazu benutzt, das Verhalten im Limes $x \to 0$ zu beschreiben als im Limes $x \to \infty$.) Für alle x gilt

$$\lim_{n\to\infty} \left(1 + \frac{x}{n}\right)^n = e^x \ . \tag{3.14}$$

Logarithmen

Wir werden die folgenden Bezeichnungen verwenden:

$$\begin{aligned}
\lg n &= \log_2 n && \text{(binärer Logarithmus)} \ , \\
\ln n &= \log_e n && \text{(natürlicher Logarithmus)} \ , \\
\lg^k n &= (\lg n)^k && \text{(Potenzierung)} \ , \\
\lg\lg n &= \lg(\lg n) && \text{(Hintereinanderausführung)} \ .
\end{aligned}$$

Wir übernehmen die wichtige Konvention hinsichtlich der Bezeichnungen, dass sich *die logarithmischen Funktionen nur auf den unmittelbar nachfolgenden Term beziehen*; $\lg n + k$ bedeutet also $(\lg n) + k$ und nicht $\lg(n + k)$. Wenn wir $b > 1$ konstant halten, dann ist die Funktion $\log_b n$ für $n > 0$ streng monoton steigend.

Für alle reellen Zahlen $a > 0$, $b > 0$, $c > 0$ und n gilt

$$\begin{aligned}
a &= b^{\log_b a} \ , \\
\log_c(ab) &= \log_c a + \log_c b \ , \\
\log_b a^n &= n \log_b a \ , \\
\log_b a &= \frac{\log_c a}{\log_c b} \ , && (3.15) \\
\log_b \frac{1}{a} &= -\log_b a \ , \\
\log_b a &= \frac{1}{\log_a b} \ , \\
a^{\log_b c} &= c^{\log_b a} \ , && (3.16)
\end{aligned}$$

wobei die Basen der Logarithmen in jeder der obigen Gleichungen ungleich 1 sind.

Aus Gleichung (3.15) geht hervor, dass die Änderung der Basis des Logarithmus von einer Konstanten zu einer anderen den Wert des Logarithmus nur um einen konstanten Faktor verändert. Deshalb werden wir häufig die Bezeichnung „$\lg n$“ verwenden, wenn uns konstante Faktoren, wie zum Beispiel in der O-Notation, nicht interessieren. Informatiker empfinden die Zahl 2 sowieso als natürlichste Basis eines Logarithmus, da so viele Algorithmen und Datenstrukturen das Aufteilen eines Problems in zwei Teile beinhalten.

Wenn $|x| < 1$ gilt, gibt es für den Ausdruck $\ln(1+x)$ eine einfache Reihenentwicklung:

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \cdots \ .$$

Ist $x > -1$, so gilt

$$\frac{x}{1+x} \ \leq \ \ln(1+x) \ \leq \ x \ , \tag{3.17}$$

wobei Gleichheit nur für $x = 0$ gilt.

Wir sagen, dass eine Funktion $f(n)$ ***polylogarithmisch beschränkt*** ist, wenn für eine Konstante k die Beziehung $f(n) = O(\lg^k n)$ erfüllt ist. Wir können das Wachstum der Polynome und Polylogarithmen vergleichen, indem wir in Gleichung (3.10) n durch $\lg n$ und a durch 2^a ersetzen, was zu

$$\lim_{n\to\infty} \frac{\lg^b n}{(2^a)^{\lg n}} = \lim_{n\to\infty} \frac{\lg^b n}{n^a} = 0$$

führt. Aus diesem Limes können wir schlussfolgern, dass für jede Konstante $a > 0$ die Beziehung

$$\lg^b n = o(n^a)$$

gilt. Somit wächst jedes Polynom schneller als jede polylogarithmische Funktion.

Fakultät

Die Bezeichnung $n!$ (gesprochen „n Fakultät“) ist für ganze Zahlen $n \geq 0$ als

$$n! = \begin{cases} 1 & \text{falls } n = 0 \ , \\ n \cdot (n-1)! & \text{falls } n > 0 \end{cases}$$

definiert. Somit gilt $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$.

Eine grobe obere Schranke der Fakultät ist $n! \leq n^n$, da jeder der n Terme der Fakultät kleiner oder gleich n ist. Die ***Stirlingsche Näherung***

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right) \ , \tag{3.18}$$

wobei e die Basis des natürlichen Logarithmus ist, liefert uns eine schärfere obere Schranke und zusätzlich eine untere Schranke. Wie Übung 3.2-3 von Ihnen verlangt, zu beweisen, gilt

$$\begin{aligned} n! &= o(n^n) \ , \\ n! &= \omega(2^n) \ , \\ \lg(n!) &= \Theta(n \lg n) \ , \end{aligned} \tag{3.19}$$

wobei die Stirlingsche Näherung nützlich zum Beweisen der Gleichung (3.19) ist. Die Gleichung

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\alpha_n} \ , \tag{3.20}$$

ist auch für alle $n \geq 1$ erfüllt, wobei

$$\frac{1}{12n+1} < \alpha_n < \frac{1}{12n} \tag{3.21}$$

gilt.

Funktionale Iteration

Wir benutzen die Notation $f^{(i)}(n)$, um die Funktion $f(n)$, wenn i-mal iterativ auf einen Anfangswert n angewendet, zu beschreiben. Um den Begriff der funktionalen Iteration formal definieren zu können, nehmen wir an, dass $f(n)$ eine Funktion über den reellen Zahlen ist. Für nichtnegative ganze Zahlen i können wir dann die i-mal iterierte Funktion $f^{(i)}(n)$ wie folgt rekursiv definieren:

$$f^{(i)}(n) = \begin{cases} n & \text{falls } i = 0 \ , \\ f(f^{(i-1)}(n)) & \text{falls } i > 0 \ . \end{cases}$$

Wenn beispielsweise $f(n) = 2n$ gilt, dann ist $f^{(i)}(n) = 2^i n$.

Iterierte logarithmische Funktion

Wir benutzen die Notation $\lg^* n$ (gesprochen „log Stern von n“) als Bezeichnung für den iterierten Logarithmus, der wie folgt definiert ist. Sei $\lg^{(i)} n$ wie oben definiert, mit $f(n) = \lg n$. Da der Logarithmus einer nichtpositiven Zahl nicht definiert ist, ist $\lg^{(i)} n$ nur definiert, wenn $\lg^{(i-1)} n > 0$ gilt. Es sei daran erinnert, dass mit $\lg^{(i)} n$ der i-mal in Folge auf den Startwert n angewendete Logarithmus gemeint ist und nicht die Funktion $\log^i n$, die die i-te Potenz des Logarithmus von n darstellt. Dann definieren wir die iterierte logarithmische Funktion durch

$$\lg^* n = \min\left\{i \geq 0 : \lg^{(i)} n \leq 1\right\} \ .$$

Der iterierte Logarithmus ist eine *sehr* langsam steigende Funktion:

$$\begin{aligned} \lg^* 2 &= 1 \ , \\ \lg^* 4 &= 2 \ , \\ \lg^* 16 &= 3 \ , \\ \lg^* 65536 &= 4 \ , \\ \lg^*(2^{65536}) &= 5 \ . \end{aligned}$$

Da die Anzahl der Atome im beobachtbaren Universum auf ungefähr 10^{80} geschätzt wird, was viel weniger als 2^{65536} ist, werden wir selten auf eine Eingabegröße n stoßen, für die $\lg^* n > 5$ gilt.

Fibonacci-Zahlen

Wir definieren die ***Fibonacci-Zahlen*** durch die folgenden Rekursionsgleichungen:

$$\begin{aligned} F_0 &= 0 \ , \\ F_1 &= 1 \ , \\ F_i &= F_{i-1} + F_{i-2} \qquad \text{für } i \geq 2 \ . \end{aligned} \tag{3.22}$$

Somit ist jede Fibonacci-Zahl die Summe der beiden vorhergehenden, was zu der Folge

$$0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55,\ \ldots$$

führt. Fibonacci-Zahlen haben einen Bezug zum ***goldenen Schnitt*** ϕ und dessen konjugiert komplexen Wert $\widehat{\phi}$, die für die beiden Lösungen der Gleichung

$$x^2 = x + 1 \tag{3.23}$$

stehen; sie sind durch die Formeln

$$\begin{aligned} \phi &= \frac{1+\sqrt{5}}{2} \\ &= 1{,}61803\ldots \ , \\ \widehat{\phi} &= \frac{1-\sqrt{5}}{2} \\ &= -0{,}61803\ldots \ . \end{aligned} \tag{3.24}$$

gegeben. Es gilt speziell

$$F_i = \frac{\phi^i - \widehat{\phi}^i}{\sqrt{5}} \ ,$$

was wir mit Induktion beweisen können (Übung 3.2-7). Da $|\widehat{\phi}| < 1$ ist, gilt

$$\begin{aligned} \frac{\left|\widehat{\phi}^i\right|}{\sqrt{5}} &< \frac{1}{\sqrt{5}} \\ &< \frac{1}{2} \ , \end{aligned}$$

aus dem

$$F_i = \left\lfloor \frac{\phi^i}{\sqrt{5}} + \frac{1}{2} \right\rfloor \ , \tag{3.25}$$

folgt. Dies sagt aus, dass die i-te Fibonacci-Zahl F_i gleich dem auf eine ganze Zahl gerundeten Wert von $\phi^i/\sqrt{5}$ ist. Die Fibonacci-Zahlen wachsen also exponentiell.

Übungen

3.2-1 Zeigen Sie, dass für monoton steigende Funktionen $f(n)$ und $g(n)$ auch die Funktionen $f(n)+g(n)$ und $f(g(n))$ monoton steigend sind, und dass $f(n) \cdot g(n)$ monoton steigend ist, wenn $f(n)$ und $g(n)$ zudem nichtnegativ sind.

3.2-2 Beweisen Sie die Gleichung (3.16).

3.2-3 Beweisen Sie die Gleichung (3.19). Beweisen Sie auch die Beziehungen $n! = \omega(2^n)$ und $n! = o(n^n)$.

3.2-4* Ist die Funktion $\lceil \lg n \rceil!$ polynomial beschränkt? Ist die Funktion $\lceil \lg \lg n \rceil!$ polynomial beschränkt?

3.2-5* Was ist asymptotisch größer: $\lg(\lg^* n)$ oder $\lg^*(\lg n)$?

3.2-6 Zeigen Sie, dass der goldene Schnitt ϕ und dessen konjugiert komplexer Wert $\widehat{\phi}$ beide die Gleichung $x^2 = x + 1$ erfüllen.

3.2-7 Beweisen Sie durch Induktion, dass die i-te Fibonacci-Zahl die Gleichung

$$F_i = \frac{\phi^i - \widehat{\phi}^i}{\sqrt{5}}$$

erfüllt, wobei ϕ der goldene Schnitt und $\widehat{\phi}$ dessen konjugiert komplexer Wert ist.

3.2-8 Zeigen Sie, dass aus $k \ln k = \Theta(n)$ die Gleichung $k = \Theta(n/\ln n)$ folgt.

Problemstellungen

3-1 Asymptotisches Verhalten von Polynomen
Gegeben seien ein Polynom

$$p(n) = \sum_{i=0}^{d} a_i n^i$$

d-ten Grades in n mit $a_d > 0$ und eine Konstante k. Benutzen Sie die Definitionen der asymptotischen Notationen, um die folgenden Eigenschaften zu beweisen.

a. Ist $k \geq d$, dann gilt $p(n) = O(n^k)$.

b. Ist $k \leq d$, dann gilt $p(n) = \Omega(n^k)$.

c. Ist $k = d$, dann gilt $p(n) = \Theta(n^k)$.

d. Ist $k > d$, dann gilt $p(n) = o(n^k)$.

e. Ist $k < d$, dann gilt $p(n) = \omega(n^k)$.

3-2 *Relatives asymptotisches Wachstum*

Tragen Sie in der folgende Tabelle für jedes Paar (A, B) ein, ob A in der Ordnung O, o, Ω, ω oder Θ von B ist. Nehmen Sie an, dass $k \geq 1$, $\epsilon > 0$ und $c > 1$ Konstanten sind.

	A	B	O	o	Ω	ω	Θ
a.	$\lg^k n$	n^ϵ					
b.	n^k	c^n					
c.	$\sqrt{n}$	$n^{\sin n}$					
d.	2^n	$2^{n/2}$					
e.	$n^{\lg c}$	$c^{\lg n}$					
f.	$\lg(n!)$	$\lg(n^n)$					

3-3 *Ordnen nach asymptotischen Wachstumsraten*

a. Ordnen Sie die folgenden Funktionen nach ihrem Wachstumsgrad, d. h. finden Sie eine Reihenfolge $g_1, g_2, \ldots, g_{30}$ der unten angegebenen Funktionen, sodass $g_1 = \Omega(g_2)$, $g_2 = \Omega(g_3)$, ..., $g_{29} = \Omega(g_{30})$ gilt. Teilen Sie Ihre Liste in Äquivalenzklassen auf, sodass sich Funktionen $f(n)$ und $g(n)$ genau dann in derselben Äquivalenzklasse befinden, wenn $f(n) = \Theta(g(n))$ gilt.

$\lg(\lg^* n)$	$2^{\lg^* n}$	$(\sqrt{2})^{\lg n}$	n^2	$n!$	$(\lg n)!$
$(\frac{3}{2})^n$	n^3	$\lg^2 n$	$\lg(n!)$	2^{2^n}	$n^{1/\lg n}$
$\ln \ln n$	$\lg^* n$	$n \cdot 2^n$	$n^{\lg \lg n}$	$\ln n$	1
$2^{\lg n}$	$(\lg n)^{\lg n}$	e^n	$4^{\lg n}$	$(n+1)!$	$\sqrt{\lg n}$
$\lg^*(\lg n)$	$2^{\sqrt{2 \lg n}}$	n	2^n	$n \lg n$	$2^{2^{n+1}}$

b. Geben Sie ein Beispiel für eine einzelne nichtnegative Funktion $f(n)$ an, die für jede der Funktionen $g_i(n)$ aus Teil (a) weder in $O(g_i(n))$ noch in $\Omega(g_i(n))$ ist.

3-4 *Eigenschaften der asymptotischen Notation*

Seien $f(n)$ und $g(n)$ asymptotisch positive Funktionen. Beweisen oder widerlegen Sie jede der folgenden Vermutungen.

a. $f(n) = O(g(n))$ impliziert $g(n) = O(f(n))$.

b. $f(n) + g(n) = \Theta(\min(f(n), g(n)))$.

c. $f(n) = O(g(n))$ impliziert $\lg(f(n)) = O(\lg(g(n)))$, wenn $\lg(g(n)) \geq 1$ und $f(n) \geq 1$ für alle hinreichend großen n gilt.

d. $f(n) = O(g(n))$ impliziert $2^{f(n)} = O\left(2^{g(n)}\right)$.

e. $f(n) = O\left((f(n))^2\right)$.

f. $f(n) = O(g(n))$ impliziert $g(n) = \Omega(f(n))$.

g. $f(n) = \Theta(f(n/2))$.

h. $f(n) + o(f(n)) = \Theta(f(n))$.

3-5 Variationen zu O und Ω

Einige Autoren definieren Ω geringfügig anders als wir. Wir wollen die Notation $\overset{\infty}{\Omega}$ (gesprochen „omega unendlich“) für diese alternative Definition benutzen. Wir sagen, dass $f(n) = \overset{\infty}{\Omega}(g(n))$ gilt, wenn eine positive Konstante c existiert, sodass für unendlich viele ganze Zahlen n die Ungleichung $f(n) \geq c\,g(n) \geq 0$ erfüllt ist.

a. Zeigen Sie, dass für zwei beliebige asymptotisch nichtnegative Funktionen $f(n)$ und $g(n)$ entweder $f(n) = O(g(n))$ oder $f(n) = \overset{\infty}{\Omega}(g(n))$ oder beides gilt. Zeigen Sie, dass dies nicht zutrifft, wenn wir Ω anstelle von $\overset{\infty}{\Omega}$ einsetzen.

b. Beschreiben Sie die potentiellen Vorteile und Nachteile bei der Verwendung von $\overset{\infty}{\Omega}$ anstelle von Ω, um Laufzeiten von Programmen zu charakterisieren.

Einige Autoren definieren auch O geringfügig anders. Wir wollen die Bezeichnung O' für die alternative Definition verwenden. Wir sagen, dass $f(n) = O'(g(n))$ genau dann gilt, wenn $|f(n)| = O(g(n))$ erfüllt ist.

c. Gelten die beiden Implikationen der Aussage aus Theorem 3.1 noch, wenn wir O durch O' ersetzen, aber weiterhin Ω benutzen?

Einige Autoren definieren auch $\widetilde{O}$ (gesprochen „weiches-oh“), womit O unter Vernachlässigung logarithmischer Faktoren gemeint ist, genauer

$$\widetilde{O}(g(n)) = \{f(n)\colon \text{es existieren positive Konstanten } c, k \text{ und } n_0, \\ \text{sodass } 0 \leq f(n) \leq c\,g(n) \lg^k(n) \text{ für alle } n \geq n_0\} \;.$$

d. Definieren Sie in gleicher Weise $\widetilde{\Omega}$ und $\widetilde{\Theta}$. Beweisen Sie das Analogon zu Theorem 3.1.

3-6 Iterierte Funktionen

Wir können den in der Funktion $\lg^*$ verwendete Iterationsoperator * auf jede monoton steigende Funktion $f(n)$ über dem Körper der reellen Zahlen anwenden. Wir definieren die iterierte Funktion f_c^* für eine gegebene Konstante $c \in \mathbf{R}$ durch

$$f_c^*(n) = \min\left\{i \geq 0 : f^{(i)}(n) \leq c\right\} \;,$$

was nicht in allen Fällen wohldefiniert zu sein braucht. Die Größe $f_c^*(n)$ ist demnach die Anzahl der iterierten Anwendungen der Funktion f, die notwendig sind, um deren Argument kleiner oder gleich c zu machen.

Geben Sie für jede der folgenden Funktionen $f(n)$ und Konstanten c die schärfste Schranke für $f_c^*(n)$ an.

	$f(n)$	c	$f_c^*(n)$
a.	$n-1$	0	
b.	$\lg n$	1	
c.	$n/2$	1	
d.	$n/2$	2	
e.	$\sqrt{n}$	2	
f.	$\sqrt{n}$	1	
g.	$n^{1/3}$	2	
h.	$n/\lg n$	2	

Kapitelbemerkungen

Knuth [209] führt den Ursprung der O-Notation auf eine Arbeit über Zahlentheorie von P. Bachmann aus dem Jahre 1892 zurück. Die o-Notation wurde 1909 von E. Landau im Rahmen seiner Erörterung zur Verteilung der Primzahlen eingeführt. Die Ω- und Θ-Notationen setzte Knuth [213] durch, um die in der Literatur populäre, aber technisch gesehen unsaubere Praxis zu korrigieren, die O-Notation sowohl für obere als auch für untere Schranken anzuwenden. Häufig wird die O-Notation auch heute noch an Stellen benutzt, an denen die Θ-Notation technisch präziser ist. Weitere Bemerkungen zur Geschichte und zur Entwicklung der asymptotischen Notationen finden sich in den Arbeiten von Knuth [209, 213] sowie Brassard und Bratley [54].

Nicht alle Autoren definieren die asymptotischen Notationen in gleicher Weise, obwohl die verschiedenen Definitionen in den meisten Fällen übereinstimmen. Einige der alternativen Definitionen umfassen Funktionen, die nicht asymptotisch nichtnegativ sind, so lange deren Beträge hinreichend beschränkt sind.

Gleichung (3.20) geht auf Robbins [297] zurück. Andere Eigenschaften elementarer mathematischer Funktionen finden sich in jedem guten Nachschlagewerk, wie zum Beispiel in Abramowitz und Stegun [1] oder Zwillinger [362] oder in einem Buch über Analysis, wie zum Beispiel in Apostol [18] oder in Thomas et al. [334]. Die Werke von Knuth [209] und Graham, Knuth und Patashnik [152] enthalten eine Fülle von Informationen zur diskreten Mathematik, wie sie in der Informatik angewendet wird.

4 Teile-und-Beherrsche

In Abschnitt 2.3.1 haben wir gesehen, wie Sortieren mit Hilfe des Teile-und-Beherrsche-Paradigma gelöst werden kann. Erinnern Sie sich daran, dass wir in dem Teile-und-Beherrsche-Ansatz ein Problem rekursiv lösen, wobei auf einer Rekursionsebene drei Schritte ausgeführt werden:

Teilen Sie das Problem in mehrere Teilprobleme, die kleinere Instanzen des gleichen Problems darstellen, auf.

Beherrschen Sie die Teilprobleme, indem Sie sie rekursiv lösen. Wenn die Teilprobleme klein genug sind, dann lösen Sie die Teilprobleme auf direktem Wege.

Vereinigen Sie die Lösungen der Teilprobleme zur Lösung des ursprünglichen Problems.

Wenn die Teilprobleme ausreichend groß sind, um rekursiv gelöst werden zu können, dann sprechen wir vom ***rekursiven Fall***. Sind die Teilprobleme so klein, dass es keinen Sinn macht, weiter rekursiv abzusteigen, so hat die Rekursion „ihren Boden gefunden“ und der ***Basisfall*** (auch ***Rekursionsverankerung*** genannt) ist erreicht. Manchmal haben wir neben den Teilproblemen, die kleinere Instanzen des gleichen Problems sind, Teilprobleme zu lösen, die dem ursprünglichen Problem nicht ganz entsprechen. Das Lösen solcher Teilprobleme sehen wir als Bestandteil des Vereinigungsschritts an.

In diesem Kapitel werden wir weitere Algorithmen kennenlernen, die auf dem Teile-und-Beherrsche-Paradigma beruhen. Das erste Verfahren löst das Max-Teilfeld-Problem: das Verfahren erhält als Eingabe ein (eindimensionales) Feld von Zahlen und berechnet ein zusammenhängendes Teilfeld, sodass die Summe der in diesem Teilfeld gespeicherten Zahlen maximal ist. Dann werden wir zwei Teile-und-Beherrsche-Algorithmen zum Multiplizieren von $n \times n$ Matrizen kennenlernen. Der eine Algorithmus läuft in Zeit $\Theta(n^3)$, was nicht besser ist als die direkte Methode zum Multiplizieren quadratischer Matrizen. Der andere Algorithmus, Strassens Algorithmus, jedoch läuft in Zeit $O(n^{2,81})$, und schlägt somit die direkte Methode asymptotisch.

Rekursionsgleichungen

Rekursionsgleichungen gehen Hand in Hand mit dem Teile-und-Beherrsche-Paradigma, weil sie uns erlauben, in natürlicher Art und Weise die Laufzeit von Teile-und-Beherrsche-Algorithmen zu charakterisieren. Eine ***Rekursionsgleichung*** ist eine Gleichung oder eine Ungleichung, die eine Funktion durch ihre eigenen Funktionswerte für kleinere Eingaben beschreibt. In Abschnitt 2.3.2 haben wir zum Beispiel gesehen, dass die Laufzeit

$T(n)$ der Prozedur MERGE-SORT im schlechtesten Fall durch die Rekursionsgleichung

$$T(n) = \begin{cases} \Theta(1) & \text{falls } n = 1 \ , \\ 2\,T(n/2) + \Theta(n) & \text{falls } n > 1 \ , \end{cases} \tag{4.1}$$

beschrieben werden kann, deren Lösung, wie wir behauptet haben, $T(n) = \Theta(n \lg n)$ ist.

Rekursionsgleichungen können vielfältige Formen haben. Beispielsweise könnte ein rekursiver Algorithmus das ursprüngliche Problem in Teilprobleme unterschiedlicher Größe teilen, z. B. in einem 2/3-zu-1/3-Verhältnis. Falls die entsprechenden Teilungs- und Vereinigungsschritte jeweils lineare Zeit benötigen, würde ein solcher Algorithmus zu der Rekursionsgleichung $T(n) = T(2n/3) + T(n/3) + \Theta(n)$ führen.

Die Größe eines Teilproblems muss nicht ein konstanter Bruchteil der Größe des ursprünglichen Problems sein. Beispielsweise würde eine rekursive Version des linearen Suchens (siehe Übung 2.1-3) genau ein Teilproblem konstruieren, wobei dieses Teilproblem nur ein Element weniger als das ursprüngliche Problem hätte. Jeder rekursive Aufruf benötigt konstante Zeit plus die Zeit, die der von ihm aufgerufene rekursive Aufruf benötigt, was zur Rekursionsgleichung $T(n) = T(n-1) + \Theta(1)$ führt.

Dieses Kapitel bietet drei Methoden zur Lösung von Rekursionsgleichungen an – d. h. um asymptotische „Θ“- oder „O“-Schranken der Lösung zu erhalten:

- Bei der ***Substitutionsmethode*** erraten wir eine Schranke und benutzen dann mathematische Induktion, um die Korrektkeit unserer Vermutung zu beweisen.

- Die ***Rekursionsbaum-Methode*** wandelt die Rekursionsgleichung in einen Baum um, dessen Knoten die in den verschiedenen Ebenen der Rekursion anfallenden Kosten darstellen; wir benutzen Techniken zur Beschränkung von Summenformeln zum Lösen der Rekursionsgleichung.

- Die ***Mastermethode*** liefert Schranken für Rekursionsgleichungen der Form

  $$T(n) = a\,T(n/b) + f(n) \ , \tag{4.2}$$

  wobei $a \geq 1$, $b > 1$ gilt und $f(n)$ eine gegebene Funktion ist. Solche Rekursionsgleichungen treten häufig auf. Eine Rekursionsgleichung der Form (4.2) charakterisiert die Laufzeit eines Teile-und-Beherrsche-Algorithmus, der das ursprüngliche Problem in a Teilprobleme, die alle eine Größe von $1/b$ der Größe des ursprünglichen Problems haben, aufteilt und bei dem die Teilungs- und Vereinigungsschritte jeweils Zeit $f(n)$ benötigen.

  Um die Mastermethode anwenden zu können, müssen Sie sich drei Fälle einprägen. Wenn Sie dies getan haben, dann sind Sie fähig, asymptotische Schranken von vielen einfachen Rekursionsgleichungen ohne Umstände zu bestimmen. Wir werden die Mastermethode verwenden, um sowohl die Laufzeiten der Teile-und-Beherrsche-Algorithmen für das Max-Teilfeld-Problem und für die Matrizenmultiplikation zu bestimmen als auch die von vielen anderen Algorithmen aus diesem Buch, die auf Teile-und-Beherrsche basieren.

Bisweilen werden wir Rekursionen sehen, die keine Gleichungen sondern Ungleichungen sind, wie z. B. $T(n) \leq 2\,T(n/2) + \Theta(n)$. Da eine solche Rekursion eine obere Schranke von $T(n)$ beschreibt, werden wir ihre Lösung in O-Notation und nicht in Θ-Notation ausdrücken. Wäre die Ungleichung anders herum, also $T(n) \geq 2\,T(n/2) + \Theta(n)$, so würden wir die Ω-Notation verwenden, da die Rekursion eine untere Schranke von $T(n)$ beschreibt.

Technische Details in Rekursionsgleichungen

In der Praxis vernachlässigen wir bestimmte technische Details, wenn wir Rekursionsgleichungen aufstellen und lösen. Wenn wir beispielsweise MERGE-SORT angewendet auf eine ungerade Anzahl n von Elementen aufrufen, erhalten wir Teilprobleme der Größe $\lfloor n/2 \rfloor$ und $\lceil n/2 \rceil$. Keiner der beiden Teilprobleme hat Größe $n/2$, da $n/2$ keine ganze Zahl ist, wenn n ungerade ist. Aus dieser technischer Sicht heraus lautet die Rekursionsgleichung für die Laufzeit von MERGE-SORT im schlechtesten Fall tatsächlich

$$T(n) = \begin{cases} \Theta(1) & \text{falls } n = 1\ , \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{falls } n > 1\ . \end{cases} \tag{4.3}$$

Randbedingungen stellen eine andere Klasse von Details dar, die wir typischerweise ignorieren. Da die Laufzeit eines Algorithmus bei konstanter Eingabegröße eine Konstante ist, sind bei den Rekursionsgleichungen, die sich für die Laufzeiten von Algorithmen ergeben, die Funktionswerte $T(n)$ für hinreichend kleine n im Allgemeinen in $\Theta(1)$. Folglich werden wir in der Regel der Einfachheit halber die Angabe von Randbedingungen für Rekursionsgleichungen weglassen und annehmen, dass $T(n)$ für kleine n konstant ist. So geben wir zum Beispiel die Rekursionsgleichung (4.1) normalerweise als

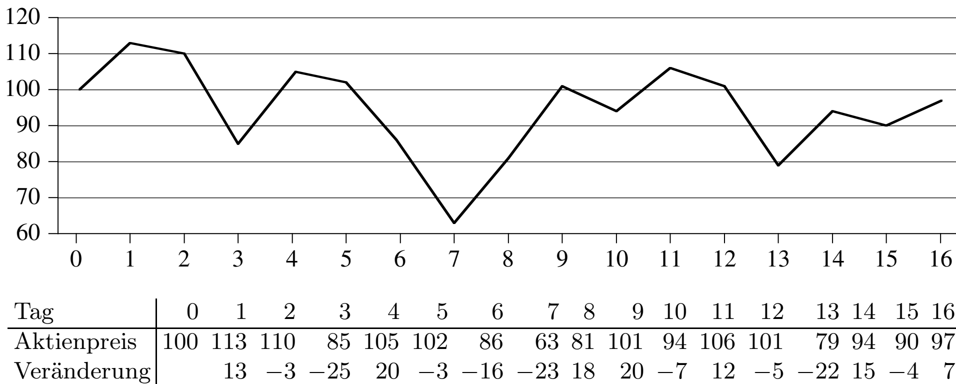
$$T(n) = 2\,T(n/2) + \Theta(n) \tag{4.4}$$

an, ohne explizit Werte für kleine n festzulegen. Gerechtfertigt wird dies dadurch, dass das Verändern des Wertes $T(1)$ die exakte Lösung der Rekursionsgleichung zwar beeinflusst, die Lösung aber üblicherweise nur um einen konstanten Faktor verändert wird und so der Wachstumsgrad unverändert bleibt.

Wenn wir Rekursionsgleichungen aufstellen und lösen, vernachlässigen wir oft das Runden auf ganze Zahlen und Randbedingungen. Wir kommen ohne diese Details besser voran und können später entscheiden, ob sie von Bedeutung sind oder nicht. Gewöhnlich sind sie es nicht. Sie sollten aber wissen, wann sie es sind. Die Erfahrung hilft, und ebenso einige Theoreme, die Aussagen darüber machen, dass diese Details die asymptotischen Schranken vieler Rekursionsgleichungen, die die Laufzeiten von Teile-und-Beherrsche-Algorithmen beschreiben, nicht beeinflussen (siehe Theorem 4.1). In diesem Kapitel werden wir jedoch auf einige dieser Details zu sprechen kommen und die Feinheiten der Lösungsmethoden für Rekursionsmethoden illustrieren.

4.1 Das Max-Teilfeld-Problem

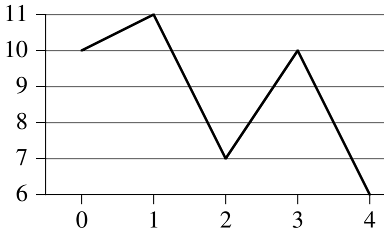
Nehmen Sie an, dass Sie das Angebot bekommen haben, in die Volatile Chemische Aktiengesellschaft zu investieren. Wie die Chemikalien, die das Unternehmen produziert, ist der Aktienkurs der Gesellschaft ziemlich volatil. Sie dürfen nur einmal Aktien kaufen, die sie dann später an einem der folgenden Tage wieder verkaufen dürfen. Kaufen und Verkaufen erfolgt jeweils abends nach Börsenschluss. Um diese Einschränkung zu kompensieren, sagt man Ihnen, wie der Kurs der Aktie sich in der Zukunft tagesgenau entwickeln wird. Ihr Ziel besteht darin, Ihren Profit zu maximieren. Abbildung 4.1 zeigt die Kurse der Aktie über einen Zeitraum von 17 Tagen. Sie dürfen die Aktien an einem beliebigen Tag kaufen, beginnend bei Tag 0, an dem der Aktienkurs bei 100 $ steht. Natürlich wollen Sie „billig kaufen und teuer verkaufen" – also zu dem niedrigsten möglichen Kurs kaufen und zu dem höchsten möglichen Kurs verkaufen –, um Ihren Profit zu maximieren. Leider wird Ihnen das aber in der Regel nicht möglich sein. In Abbildung 4.1 ist der Kurs der Aktie nach Börsenschluss des 7. Tages am niedrigsten und nach Börsenschluss des 1. Tages am höchsten.



Tag	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Aktienpreis	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Veränderung		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

Abbildung 4.1: *Kurse der Aktie der Volatilen Chemischen Aktiengesellschaft nach Börsenschluss für eine Periode von 17 Tagen. Die horizontale Achse des Diagramms gibt die Tage an, und die vertikale den Aktienpreis. Die untere Reihe der Tabelle enthält die Veränderungen des Aktienpreises jeweils verglichen mit dem Preis vom Vortag.*

Möglicherweise denken Sie, dass Sie den Profit immer maximieren können, wenn Sie entweder beim tiefsten Kurs kaufen oder beim höchsten Kurs verkaufen. In Abbildung 4.1, würden wir den Profit maximieren, wenn wir nach Börsenschluss des 7. Tages kaufen würden. Wenn diese Strategie funktionieren würde, dann wäre es einfach, den Profit zu maximieren: finde den höchsten und niedrigsten Kurs, suche dann links vom höchsten Kurs den vordran niedrigsten Kurs und rechts vom niedrigsten Kurs den höchsten Kurs, der sich nach diesem Tag noch einstellt, und nehme als Lösung dann das Zeitpunktenpaar mit der größten Differenz. Abbildung 4.2 zeigt ein einfaches Gegenbeispiel: sie zeigt, dass manchmal der maximale Profit erzielt werden kann, wenn man weder zum niedrigsten Kurs kauft noch zum höchsten Kurs verkauft.



Tag	0	1	2	3	4
Aktienkurs	10	11	7	10	6
Veränderung		1	−4	3	−4

Abbildung 4.2: *Das Beispiel zeigt, dass der maximale Profit nicht immer beim niedrigsten Kurs startet oder beim höchsten Kurs endet. Wie vorhin gibt die horizontale Achse die Tage an und die vertikale den Aktienkurs. In diesem Beispiel liegt der maximale Profit bei 3 $ pro Aktie, den man erzielen kann, wenn man nach Börsenschluss des 2. Tages kaufen und nach Börsenschluss des 3. Tages verkaufen würde. Der Aktienkurs von 7 $ nach Börsenschluss des 2. Tages ist nicht der insgesamt niedrigste Kurs und der von 10 $ nach Börsenschluss des 3. Tages ist nicht der insgesamt höchste Kurs.*

Ein Brute-Force-Ansatz

Wir können sehr leicht einen Brute-Force-Ansatz zum Lösen diese Problems angeben: wir überprüfen einfach jedes mögliche Paar von Kauf- und Verkaufszeitpunkten, in denen das Kaufdatum vor dem Verkaufsdatum liegt. Eine Periode über n Tage hat $\binom{n}{2}$ solcher Paare. Da $\binom{n}{2}$ in $\Theta(n^2)$ liegt, und eine konstante Laufzeit zur Auswertung eines jeden dieser Paare das Beste ist, was wir uns erhoffen können, würde dieser Ansatz $\Omega(n^2)$ Zeit benötigen. Können wir es besser?

Das Problem aus einem anderen Blickwinkel

Um einen Algorithmus mit einer Laufzeit von $o(n^2)$ zu entwerfen, wollen wir uns das Problem aus einem leicht anderen Blickwinkel anschauen. Wir wollen eine Folge von Tagen finden, über denen die aufgerechneten Tageskursänderungen vom ersten zum letzten Tag dieser Folge maximal ist. Anstatt auf den täglichen Aktienkurs zu schauen, betrachten wir die täglichen Kursänderungen, wobei die Kursänderung am i. Tag die Differenz des Aktienkurs zum Börsenschluss des i. Tages und des Aktienkurses zum Börsenschluss des $i-1$. Tages ist. Die Tabelle in Abbildung 4.1 zeigt diese täglichen Veränderungen in der unteren Zeile. Wenn wir diese Zeile, wie in Abbildung 4.3 gezeigt, als ein Feld A betrachten, besteht die Aufgaben darin, ein nichtleeres, zusammenhängendes Teilfeld von A zu finden, deren Werte aufaddiert zu der größten Summe führt. Wir nennen dieses Teilfeld das ***maximale Teilfeld***. Im Feld aus Abbildung 4.3 zum Beispiel ist das maximale Teilfeld des Feldes $A[1\,..\,16]$ das Teilfeld $A[8\,..\,11]$, mit einer Summe von 43. Sie sollten also die Aktie nach Börsenschluss des 7. Tages kaufen und nach Börsenschluss des 11. Tages verkaufen, um einen Profit von 43 $ pro Aktie einzufahren.

Auf den ersten Blick hilft diese Transformation des Problems nichts. Wir haben weiterhin $\binom{n-1}{2} = \Theta(n^2)$ Teilfelder bei einer Periode von n Tagen zu betrachten. Übung 4.1-2 verlangt von Ihnen zu zeigen, dass die Berechnung so organisiert werden kann, dass die Berechnung der Summe eines Teilfeldes in Zeit $O(1)$ erfolgen kann, wenn die im Vorfeld bereits berechneten Werte seiner Teilfelder gegeben sind, sodass der Brute-Force-Ansatz

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	–3	–25	20	–3	–16	–23	18	20	–7	12	–5	–22	15	–4	7

maximales Teilfeld

Abbildung 4.3: *Die Änderungen des Aktienkurses als Max-Teilfeld-Problem. In diesem Beispiel besitzt das Teilfeld* $A[8\,..\,11]$*, mit einer Summe von* 43*, die größte aufaddierte Summe aller zusammenhängenden Teilfelder des Feldes* A*.*

Zeit $\Theta(n^2)$ benötigt – wenngleich die Berechnung der Kosten eines Teilfeldes möglicherweise eine Laufzeit proportional zu der Länge des Teilfeldes hat.

Lassen Sie uns also nach einer effizienteren Lösung für das Max-Teilfeld-Problem suchen. In diesem Zusammenhang werden wir üblicherweise von „einem" maximalen Teilfeld und nicht von „dem" maximalen Teilfeld sprechen, da es mehr als ein Teilfeld geben könnte, das maximal ist.

Das Max-Teilfeld-Problem ist nur dann interessant, wenn das Feld einige negative Zahlen enthält. Wenn alle Feldeinträge nichtnegativ wären, würde das Max-Teilfeld-Problem keine Herausforderung darstellen, da das gesamte Feld die größte Summe ergeben würde.

Eine auf Teile-und-Beherrsche basierende Lösung

Lassen Sie uns darüber nachdenken, wie wir das Max-Teilfeld-Problem mittels der Teile-und-Beherrsche-Technik lösen können. Nehmen Sie an, wir wollten ein maximales Teilfeld des Teilfeldes $A[\mathit{links}\,..\,\mathit{rechts}]$ finden. Das Teile-und-Beherrsche-Paradigma regt an, das Teilfeld in zwei Teilfelder gleicher Größe zu teilen, soweit dies möglich ist. Wir haben also die Mitte des Teilfeldes, die wir mit *mitte* bezeichnen wollen, zu berechnen und dann die Teilfelder $A[\mathit{links}\,..\,\mathit{mitte}]$ und $A[\mathit{mitte}+1\,..\,\mathit{rechts}]$ zu betrachten. Wie Abbildung 4.4(a) zeigt, muss jedes zusammenhängende Teilfeld $A[i\,..\,j]$ von $A[\mathit{links}\,..\,\mathit{rechts}]$ in genau einer der folgenden Lagen liegen:

- komplett im Teilfeld $A[\mathit{links}\,..\,\mathit{mitte}]$, d. h. es gilt $\mathit{links} \leq i \leq j \leq \mathit{mitte}$,
- komplett im Teilfeld $A[\mathit{mitte}+1\,..\,\mathit{rechts}]$, d. h. es gilt $\mathit{mitte} < i \leq j \leq \mathit{rechts}$, oder
- mittig, d. h. es gilt $\mathit{links} \leq i \leq \mathit{mitte} < j \leq \mathit{rechts}$.

Somit muss auch das maximale Teilfeld von $A[\mathit{links}\,..\,\mathit{rechts}]$ in genau einer dieser Lagen liegen. In der Tat, das maximale Teilfeld von $A[\mathit{links}\,..\,\mathit{rechts}]$ muss die größte aufaddierte Summe von allen Teilfeldern haben, die komplett in $A[\mathit{links}\,..\,\mathit{mitte}]$ liegen, komplett in $A[\mathit{mitte}+1\,..\,\mathit{rechts}]$ liegen oder mittig angeordnet sind. Wir können maximale Teilfelder von $A[\mathit{links}\,..\,\mathit{mitte}]$ und $A[\mathit{mitte}+1\,..\,\mathit{rechts}]$ rekursiv berechnen, da diese zwei Teilprobleme kleinere Instanzen des Max-Teilfeld-Problems sind. Das einzige, das dann noch zu tun ist, ist ein maximales mittiges Teilfeld zu finden und dann das Teilfeld als

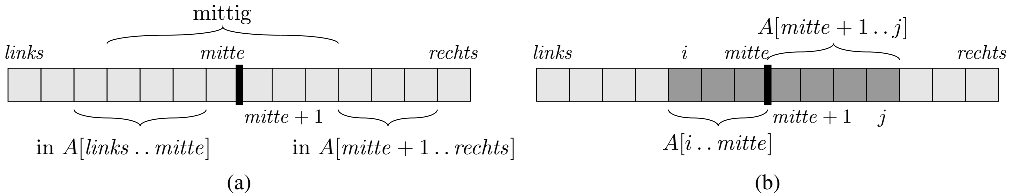


Abbildung 4.4: ***(a)*** *Mögliche Lagen der Teilfelder von $A[links\,..\,rechts]$: komplett in $A[links\,..\,mitte]$, komplett in $A[mitte+1\,..\,rechts]$, oder mittig.* ***(b)*** *Jedes Teilfeld von $A[links\,..\,rechts]$, das die Mitte enthält, besteht aus zwei Teilfeldern $A[i\,..\,mitte]$ und $A[mitte+1\,..\,j]$ mit $links \leq i \leq mitte$ und $mitte < j \leq rechts$.*

Lösung zu nehmen, das von diesen drei berechneten Teilfeldern die größte aufaddierte Summe besitzt.

Wir können ein maximales mittiges Teilfeld ohne Umstände in einer Zeit, die linear in der Größe des Teilfeldes $A[links\,..\,rechts]$ ist, berechnen. Dieses Problem ist *keine* kleinere Instanz des ursprünglichen Problems, da es die zusätzliche Einschränkung hat, dass das gesuchte Teilfeld die Mitte überschreiten muss. Wie Abbildung 4.4(b) zeigt, besteht jedes mittige Teilfeld selbst aus zwei Teilfelder $A[i\,..\,mitte]$ und $A[mitte+1\,..\,j]$ mit $links \leq i \leq mitte$ und $mitte < j \leq rechts$. Wir haben aus diesem Grund einfach nur maximale Teilfelder der Form $A[i\,..\,mitte]$ und $A[mitte+1\,..\,j]$ zu finden und diese zu vereinigen. Die Prozedur FIND-MAX-CROSSING-SUBARRAY bekommt als Eingabe ein Feld A und die Indizes *links*, *mitte* und *rechts* und gibt ein Tupel bestehend aus den Indizes, die ein maximales mittiges Teilfeld demarkieren, sowie die Summe der Werte in einem maximalen Teilfeld zurück.

```
FIND-MAX-CROSSING-SUBARRAY(A, links, mitte, rechts)
 1  linke-summe = −∞
 2  summe = 0
 3  for i = mitte downto links
 4      summe = summe + A[i]
 5      if summe > linke-summe
 6          linke-summe = summe
 7          max-links = i
 8  rechte-summe = −∞
 9  summe = 0
10  for j = mitte + 1 to rechts
11      summe = summe + A[j]
12      if summe > rechte-summe
13          rechte-summe = summe
14          max-rechts = j
15  return (max-links, max-rechts, linke-summe + rechte-summe)
```

Diese Prozedur arbeitet wie folgt. Die Zeilen 1–7 berechnen ein maximales Teilfeld in der linken Hälfte $A[links \, .\,. \, mitte]$ des Feldes. Da das Teilfeld, das in diesen Zeilen berechnet werden soll, $A[mitte]$ enthalten muss, startet die Laufvariable i der **for**-Schleife in den Zeilen 3–7 bei *mitte*; die Schleife arbeitet sich dann von der Mitte aus nach links zum linken Rand des Feldes vor, sodass jedes Teilfeld, das durch die Schleife betrachtet wird, von der Form $A[i \, .\,. \, mitte]$ ist. Die Zeilen 1–2 initialisieren die Variablen *linke-summe*, in der die bis dahin größte gefundene Summe abgespeichert wird, und *summe*, die die Summe der Werte aus $A[i \, .\,. \, mitte]$ enthält. Immer wenn wir in Zeile 5 ein Teilfeld $A[i \, .\,. \, mitte]$ finden, dessen aufaddierte Werte größer als *linke-summe* ist, aktualisieren wir in Zeile 6 *linke-summe* auf die Summe dieses Teilfeldes und in Zeile 7 die Variable *max-links*, in der wir uns den entsprechenden Index i merken. Die Zeilen 8–14 arbeiten analog für die rechte Hälfte $A[mitte + 1 \, .\,. \, rechts]$. Hier startet die Laufvariable j der **for**-Schleife in den Zeilen 10–14 bei $mitte + 1$ und die Schleife arbeitet sich von dieser Stelle aus nach rechts zum rechten Rand des Feldes vor, sodass sie jedes Teilfeld der Form $A[mitte + 1 \, .\,. \, j]$ betrachtet. In Zeile 15 werden die Indizes *max-links* and *max-rechts*, die ein maximales mittiges Teilfeld demarkieren, zusammen mit der Summe $linke\text{-}summe + rechte\text{-}summe$ der Werte des Teilfeldes $A[max\text{-}links \, .\,. \, max\text{-}rechts]$ schlussendlich zurückgegeben.

Wir behaupten, dass FIND-MAX-CROSSING-SUBARRAY$(A, links, mitte, rechts)$ Zeit $\Theta(n)$ benötigt, wenn das Teilfeld $A[links \, .\,. \, rechts]$ aus n Einträgen besteht, d. h. wenn $n = rechts - links + 1$ gilt. Da jede Iteration einer jeden der zwei **for**-Schleifen jeweils $\Theta(1)$ Zeit benötigt, haben wir nur zu zählen, wie viele Iterationen insgesamt ausgeführt werden. Die **for**-Schleife in den Zeilen 3–7 wird $mitte - links + 1$ mal ausgeführt und die **for**-Schleife in den Zeilen 10–14 $rechts - mitte$ mal. Somit beläuft sich die Gesamtzahl der Iterationen auf

$$\begin{aligned}(mitte - links + 1) + (rechts - mitte) &= rechts - links + 1 \\ &= n \,.\end{aligned}$$

Nachdem wir die in linearer Zeit ausführbare Prozedur FIND-MAX-CROSSING-SUBARRAY kennengelernt haben, können wir den Pseudocode für den Teile-und-Beherrsche-Algorithmus für das Max-Teilfeld-Problem aufschreiben:

```
FIND-MAXIMUM-SUBARRAY(A, links, rechts)
 1  if rechts == links
 2       return (links, rechts, A[links])          // Basisfall: nur ein Element
 3  else mitte = ⌊(links + rechts)/2⌋
 4       (links-links, links-rechts, linke-summe) =
              FIND-MAXIMUM-SUBARRAY(A, links, mitte)
 5       (rechts-links, rechts-rechts, rechte-summe) =
              FIND-MAXIMUM-SUBARRAY(A, mitte + 1, rechts)
 6       (mittig-links, mittig-rechts, mittige-summe) =
              FIND-MAX-CROSSING-SUBARRAY(A, links, mitte, rechts)
 7       if linke-summe ≥ rechte-summe und linke-summe ≥ mittige-summe
 8            return (links-links, links-rechts, linke-summe)
 9       elseif rechte-summe ≥ linke-summe und rechte-summe ≥ mittige-summe
10            return (rechts-links, rechts-rechts, rechte-summe)
11       else return (mittig-links, mittig-rechts, mittige-summe)
```

Der initiale Aufruf FIND-MAXIMUM-SUBARRAY$(A, 1, A.\mathit{länge})$ berechnet das maximale Teilfeld von $A[1\,..\,n]$.

Wie FIND-MAX-CROSSING-SUBARRAY gibt die rekursive Prozedur FIND-MAXIMUM-SUBARRAY ein Tupel zurück, das die Indizes enthält, die das maximale Teilfeld demarkieren, zusammen mit der Summe der Werte eines maximalen Teilfeldes. Zeile 1 testet auf den Basisfall, in dem das Teilfeld nur aus einem Element besteht. Ein Teilfeld, das nur aus einem Element besteht, hat nur ein Teilfeld – nämlich sich selbst – und so gibt Zeile 2 das Tupel bestehend aus dem Index dieses einzigen Elementes zusammen mit seinem Wert zurück. Die Zeilen 3–11 behandeln den rekursiven Fall. In Zeile 3 erfolgt das Aufteilen des Problems, indem der Index *mitte* der Mitte des Teilfeldes berechnet wird. Lassen Sie uns das Teilfeld $A[\mathit{links}\,..\,\mathit{mitte}]$ das ***linke Teilfeld*** und das Teilfeld $A[\mathit{mitte}+1\,..\,\mathit{rechts}]$ das ***rechte Teilfeld*** nennen. Da wir wissen, dass das Teilfeld $A[\mathit{links}\,..\,\mathit{rechts}]$ wenigstens zwei Elemente enthält, enthält das linke und das rechte Teilfeld jeweils wenigstens ein Element. Die Beherrsche-Phase erfolgt in den Zeilen 4 und 5, in denen das maximale Teilfeld des linken Teilfeldes und das des rechten Teilfeldes berechnet werden. Die Zeilen 6–11 realisieren die Vereinigung der rekursiv berechneten Lösungen der Teilprobleme. Zeile 6 berechnet das maximale mittige Teilfeld. (Erinnern Sie sich bitte daran, dass Zeile 6 ein Teilproblem löst, das nicht eine kleinere Instanz des ursprünglichen Problems ist, und wir somit die Zeile zum Vereinigungsschritt zählen.) Zeile 7 testet, ob das linke Teilfeld ein Teilfeld mit maximaler Summe enthält und Zeile 8 gibt in diesem Fall dieses maximale Teilfeld zurück. Ansonsten testet Zeile 9, ob das rechte Teilfeld ein Teilfeld mit maximaler Summe enthält und Zeile 10 gibt dieses maximale Teilfeld zurück. Wenn weder das linke noch das rechte Teilfeld ein Teilfeld mit maximaler Summe enthält, so muss ein maximales Teilfeld mittig angeordnet sein; Zeile 11 gibt dieses zurück.

Analyse des Teile-und-Beherrsche-Algorithmus

Als nächstes stellen wir eine Rekursionsgleichung auf, die die Laufzeit der rekursiven Prozedur FIND-MAXIMUM-SUBARRAY beschreibt. Genauso wie wir dies bei der Analyse von Sortieren durch Mischen in Abschnitt 2.3.2 getan haben, machen wir auch hier die vereinfachende Annahme, dass die Größe des ursprünglichen Problems eine Zweierpotenz ist, sodass die Größe aller Teilprobleme ebenfalls Zweierpotenzen sind und bei der Berechnung der Mitte nicht abgerundet werden muss. Wir bezeichnen die Laufzeit von FIND-MAXIMUM-SUBARRAY angewendet auf ein Teilfeld von n Elementen mit $T(n)$. Zunächst einmal benötigt Zeile 1 konstante Zeit. Der Basisfall, also wenn $n = 1$ gilt, ist einfach: Zeile 2 benötigt konstante Zeit und somit gilt:

$$T(1) = \Theta(1) \; . \tag{4.5}$$

Der rekursive Fall liegt vor, wenn $n > 1$ gilt. Die Zeilen 1 und 3 benötigen konstante Zeit. Jedes der Teilprobleme, die in den Zeilen 4 und 5 gelöst werden, sind von der Größe $n/2$ (unsere Annahme, dass die Größe des ursprünglichen Problems eine Zweierpotenz ist, gewährleistet, dass $n/2$ eine ganze Zahl ist) und somit verbrauchen wir $T(n/2)$ Zeit, um jedes von ihnen zu lösen. Da wir zwei Teilprobleme – für das linke Teilfeld und für das rechte Teilfeld – zu lösen haben, tragen die Zeilen 4 und 5 zusammen $2\,T(n/2)$

zur Gesamtlaufzeit bei. Wie wir bereits gesehen haben, benötigt die Ausführung von FIND-MAX-CROSSING-SUBARRAY in Zeile 6 Zeit $\Theta(n)$. Die Zeilen 7–11 benötigen nur Zeit $\Theta(1)$. Wir erhalten demnach für den rekursiven Fall

$$\begin{aligned} T(n) &= \Theta(1) + 2\,T(n/2) + \Theta(n) + \Theta(1) \\ &= 2\,T(n/2) + \Theta(n)\ . \end{aligned} \tag{4.6}$$

Die Gleichungen (4.5) und (4.6) stellen zusammen die Rekursionsgleichung für die Laufzeit $T(n)$ von FIND-MAXIMUM-SUBARRAY dar:

$$T(n) = \begin{cases} \Theta(1) & \text{falls } n = 1\ , \\ 2\,T(n/2) + \Theta(n) & \text{falls } n > 1\ . \end{cases} \tag{4.7}$$

Diese Rekursionsgleichung ist die gleiche wie die Rekursionsgleichung (4.1) für Sortieren durch Mischen. Wie wir bei der in Abschnitt 4.5 vorgestellten Mastermethode sehen werden, hat diese Rekursionsgleichung die Lösung $T(n) = \Theta(n \lg n)$. Sie können sich auch den Rekursionsbaum in Abbildung 2.5 nochmals anschauen, um zu verstehen, warum die Lösung $T(n) = \Theta(n \lg n)$ sein sollte.

Wir haben also gesehen, dass die Teile-und-Beherrsche-Methode zu einem Algorithmus führt, der asymptotisch schneller als die Brute-Force-Methode ist. Mit Sortieren durch Mischen und jetzt mit dem Max-Teilfeld-Problem bekommen wir eine Idee, wie mächtig die Teile-und-Beherrsche-Methode sein kann. In einigen Fällen liefert sie den asymptotisch besten Algorithmus für ein Problem, in anderen Fällen gibt es bessere Lösungen. Wie Übung 4.1-5 zeigt, gibt es tatsächlich ein Linearzeit-Algorithmus für das Max-Teilfeld-Problem und dieser benutzt Teile-und-Beherrsche nicht.

Übungen

4.1-1 Was berechnet FIND-MAXIMUM-SUBARRAY, wenn alle Einträge von A negativ sind?

4.1-2 Geben Sie den Pseudocode für die Brute-Force-Methode zum Lösen des Max-Teilfeld-Problems an. Ihre Prozedur sollte in $\Theta(n^2)$ Zeit laufen.

4.1-3 Implementieren Sie sowohl die Brute-Force-Methode als auch den rekursiven Algorithmus für das Max-Teilfeld-Problem auf Ihrem Rechner. Ab welcher Größe n_0 schlägt der rekursive Algorithmus den Brute-Force-Algorithmus? Ändern Sie dann den Basisfall des rekursiven Algorithmus, sodass die Brute-Force-Methode immer dann benutzt wird, wenn die Problemgröße kleiner als n_0 ist. Ab welcher Größe schlägt nun der rekursive Algorithmus die Brute-Force-Methode?

4.1-4 Nehmen Sie an, wir würden die Definition des Max-Teilfeld-Problems so ändern, dass das leere Teilfeld als Ergebnis zulässig ist, wobei die Summe der Werte eines leeren Teilfeldes gleich 0 ist. Wie müssten Sie die Algorithmen ändern?

4.1-5 Benutzen Sie die folgenden Ideen, um einen nichtrekursiven Algorithmus für das Max-Teilfeld-Problem, der lineare Zeit benötigt, zu entwickeln. Beginnen

Sie am linken Rand des Feldes und arbeiten Sie sich nach rechts vor, wobei Sie sich jeweils das bisher gefundene maximale Teilfeld merken. Nutzen Sie diese Information jeweils aus, um ein maximales Teilfeld, das an der Stelle $j+1$ endet, zu berechnen. Zentral ist hierbei die folgende Beobachtung: ein maximales Teilfeld von $A[1..j+1]$ ist entweder ein maximales Teilfeld von $A[1..j]$ oder ein Teilfeld $A[i..j+1]$ für ein $1 \leq i \leq j+1$. Berechnen Sie in konstanter Zeit ein maximales Teilfeld der Form $A[i..j+1]$, indem Sie ausnutzen, dass Sie bereits ein maximales Teilfeld, das an der Stelle j endet, kennen.

4.2 Strassens Algorithmus zur Matrizenmultiplikation

Wenn Sie Matrizen kennen, wissen Sie höchstwahrscheinlich auch wie man diese multipliziert. (Ansonsten sollten Sie den Abschnitt D.1 im Anhang D lesen.) Wenn $A = (a_{ij})$ und $B = (b_{ij})$ quadratische $n \times n$ Matrizen sind, dann sind die Einträge c_{ij}, für $i, j = 1, 2, \ldots, n$, im Produkt $C = A \cdot B$ durch

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \tag{4.8}$$

gegeben. Wir müssen also n^2 Matrizeneinträge berechnen und jeder dieser Einträge ist eine Summe von n Werten. Die folgende Prozedur erhält als Eingabe $n \times n$-Matrizen A und B und multipliziert sie. Wir nehmen an, dass jede Matrix ein Attribut *zeilen* besitzt, das die Anzahl der Zeilen in der Matrix angibt.

```
SQUARE-MATRIX-MULTIPLY(A, B)
1  n = A.zeilen
2  sei C eine neue n × n-Matrix
3  for i = 1 to n
4      for j = 1 to n
5          c_ij = 0
6          for k = 1 to n
7              c_ij = c_ij + a_ik · b_kj
8  return C
```

Die Prozedur SQUARE-MATRIX-MULTIPLY arbeitet wie folgt. Die **for**-Schleife in den Zeilen 3–7 berechnet die Einträge einer jeden Matrixzeile i und innerhalb einer gegebenen Matrixzeile i berechnet die **for**-Schleife in den Zeilen 4–7 die Einträge c_{ij} für jede Spalte j. Zeile 5 initialisiert c_{ij} mit 0, um dann mit der Berechnung der Summe aus Gleichung (4.8) zu beginnen. Jede Iteration der **for**-Schleife der Zeilen 6–7 addiert einen weiteren Term der Gleichung (4.8).

Da jede der dreifach geschachtelten **for**-Schleifen genau n Iterationen durchläuft und die Zeile 7 jeweils in konstanter Zeit ausgeführt werden kann, benötigt die Prozedur SQUARE-MATRIX-MULTIPLY $\Theta(n^3)$ Zeit.

Auf den ersten Blick könnten Sie vielleicht glauben, dass jeder Algorithmus zur Matrizenmultiplikation $\Omega(n^3)$ Zeit benötigen muss, da die normale Definition der Matrizenmultiplikation so viele Multiplikationen verlangt. Sie hätten unrecht: Wir kennen eine Möglichkeit, Matrizen in $o(n^3)$ Zeit zu multiplizieren. In diesem Abschnitt werden wir Strassens bemerkenswerten rekursiven Algorithmus zur Multiplikation von $n \times n$-Matrizen kennenlernen. Er benötigt $\Theta(n^{\lg 7})$ Zeit. Dies werden wir in Abschnitt 4.5 zeigen. Da $\lg 7$ zwischen $2,80$ und $2,81$ liegt, benötigt Strassens Algorithmus $O(n^{2,81})$ Zeit, was asymptotisch besser ist als die einfache SQUARE-MATRIX-MULTIPLY Prozedur.

Ein einfacher Teile-und-Beherrsche-Algorithmus

Der Einfachheit halber nehmen wir bei Teile-und-Beherrsche-Algorithmen zur Berechnung des Matrizenproduktes $C = A \cdot B$ von quadratischen $n \times n$-Matrizen A und B an, dass n eine Zweierpotenz ist. Wir machen diese Annahme, da in jedem Teilungsschritt $n \times n$-Matrizen jeweils in vier $n/2 \times n/2$-Matrizen aufgeteilt werden und durch unsere Annahme, dass n eine Zweierpotenz ist, wir garantieren können, dass die Größe $n/2$ eine ganze Zahl ist, solange $n \geq 2$ gilt.

Nehmen Sie an, dass wir jede der Matrizen A, B und C in jeweils vier $n/2 \times n/2$-Matrizen partitionieren

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} , \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} , \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} , \tag{4.9}$$

sodass wir die Gleichung $C = A \cdot B$ umschreiben können zu

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} . \tag{4.10}$$

Gleichung (4.10) entspricht den vier Gleichungen

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21} , \tag{4.11}$$
$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22} , \tag{4.12}$$
$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21} , \tag{4.13}$$
$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22} . \tag{4.14}$$

Jede dieser vier Gleichungen schreibt zwei Multiplikationen von $n/2 \times n/2$-Matrizen und die Addition ihrer $n/2 \times n/2$-Produkte vor. Wir können diese Gleichungen anwenden, um einen einfachen rekursiven Teile-und-Beherrsche-Algorithmus zu erhalten:

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
 1  n = A.zeilen
 2  sei C eine neue n × n-Matrix
 3  if n == 1
 4        c11 = a11 · b11
 5  else partitioniere A, B und C gemäß Gleichung (4.9)
 6        C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
                + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
 7        C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
                + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
 8        C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
                + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
 9        C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
                + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
10  return C
```

Dieser Pseudocode geht auf ein ausgetüfteltes aber wichtiges Detail der Implementierung nicht ein. Wie partitionieren wir die Matrizen in Zeile 5? Wenn wir 12 neue $n/2 \times n/2$-Matrizen generieren würden, müssten wir $\Theta(n^2)$ Zeit aufbringen, um die Einträge zu kopieren. Tatsächlich können wir die Matrizen partitionieren, ohne die Einträge zu kopieren. Der Trick besteht darin, dass wir mit Indexberechnungen arbeiten. Wir identifizieren eine Teilmatrix durch einen Bereich von Zeilenindizes und einen Bereich von Spaltenindizes der ursprünglichen Matrix. Dadurch müssen wir die Teilmatrizen ein bisschen anders darstellen als wir die ursprüngliche Matrix darstellen. Der Vorteil, dass wir Teilmatrizen mittels Indexberechnungen spezifizieren können, besteht darin, dass die Ausführung von Zeile 5 nur $\Theta(1)$ Zeit kostet (wenngleich wir sehen werden, dass es keinen Einfluss auf die asymptotische Gesamtlaufzeit hat, ob wir kopieren oder „in-place" partitionieren).

Wie leiten nun die Rekursionsgleichung her, die die Laufzeit von SQUARE-MATRIX-MULTIPLY-RECURSIVE beschreibt. Sei $T(n)$ die Laufzeit, die diese Prozedur zum Multiplizieren zweier $n \times n$-Matrizen benötigt. Im Basisfall, d. h. wenn $n = 1$ gilt, führen wir nur eine skalare Multiplikation in Zeile 4 aus und somit gilt

$$T(1) = \Theta(1) . \tag{4.15}$$

Der rekursive Fall liegt vor, wenn $n > 1$ gilt. Wie bereits diskutiert, benötigt das Partitionieren der Matrizen in Zeile 5 Zeit $\Theta(1)$, sofern wir mit Indexberechnungen arbeiten. In den Zeilen 6–9 rufen wir acht Mal SQUARE-MATRIX-MULTIPLY-RECURSIVE rekursiv auf. Da jeder rekursive Aufruf zwei $n/2 \times n/2$-Matrizen multipliziert und somit $T(n/2)$ Zeit zur Gesamtlaufzeit beiträgt, benötigen diese acht rekursiven Aufrufe zusammen $8\,T(n/2)$ Zeit. Wir müssen auch die vier Matrizenadditionen aus den Zeilen 6–9 berücksichtigen. Jeder dieser Matrizen enthält $n^2/4$ Einträge und so benötigt jede der vier Matrizenadditionen $\Theta(n^2)$ Zeit. Da die Anzahl auszuführender Matrizenadditionen konstant ist, ist die Gesamtzeit, die für die Matrizenadditionen aus den Zeilen 6–9 aufgebracht werden muss, gleich $\Theta(n^2)$. (Auch hier arbeiten wir mit Indexberechnungen,

um die Ergebnisse der Matrizenaddition auf der richtigen Stelle in der Matrix C abzuspeichern; die Berechnungszeit pro Eintrag ist hierfür in $\Theta(1)$.) Die Gesamtzeit für den rekursiven Fall ist deshalb gleich der Summe der Zeit zum Partitionieren der Matrizen, der Zeit für alle rekursiven Aufrufe und die Zeit zum Addieren der Matrizen, die durch die rekursiven Aufrufe erzeugt worden sind:

$$\begin{aligned} T(n) &= \Theta(1) + 8\,T(n/2) + \Theta(n^2) \\ &= 8\,T(n/2) + \Theta(n^2)\ . \end{aligned} \tag{4.16}$$

Hätten wir das Partitionieren der Matrizen durch Kopieren realisiert – was $\Theta(n^2)$ Zeit benötigt –, würde sich die Rekursionsgleichung nicht ändern und sich die Gesamtlaufzeit somit nur um einen konstanten Faktor erhöhen.

Die Gleichungen (4.15) und (4.16) stellen die vollständige Rekursionsgleichung für die Laufzeit von SQUARE-MATRIX-MULTIPLY-RECURSIVE dar:

$$T(n) = \begin{cases} \Theta(1) & \text{falls } n = 1\ , \\ 8\,T(n/2) + \Theta(n^2) & \text{falls } n > 1\ . \end{cases} \tag{4.17}$$

Wie wir später mit Hilfe der Mastermethode aus Abschnitt 4.5 sehen werden, hat die Rekursionsgleichung (4.17) die Lösung $T(n) = \Theta(n^3)$. Demnach ist dieser einfache Teile-und-Beherrsche-Ansatz nicht schneller als die SQUARE-MATRIX-MULTIPLY Prozedur.

Bevor wir weitermachen mit Strassens Algorithmus, lassen Sie uns nochmals zusammenfassen, wie sich die einzelnen Teilausdrücke der Gleichung (4.16) begründen. Das Partitionieren jeder $n \times n$-Matrix mit Indexberechnung kostet $\Theta(1)$ Zeit. Wir haben aber zwei Matrizen zu partitionieren. Wenngleich Sie argumentieren könnten, dass das Partitionieren von zwei Matrizen dann $\Theta(2)$ Zeit benötigt, wird die Konstante 2 durch die Θ-Notation subsumiert. Das Addieren von zwei Matrizen, jede sagen wir mit k Einträgen, benötigt $\Theta(k)$ Zeit. Da die Matrizen, die wir addieren, $n^2/4$ Einträge besitzen, können Sie behaupten, dass das Addieren eines Paares solcher Matrizen in $\Theta(n^2/4)$ Zeit erfolgt. Wiederum subsumiert jedoch die Θ-Notation den konstanten Faktor von $1/4$ und wir stellen fest, dass das Addieren von zwei $n/2 \times n/2$-Matrizen $\Theta(n^2)$ Zeit benötigt. Wir haben vier solcher Matrizenadditionen auszuführen und erneut können wir sagen, dass diese in einer Gesamtzeit von $\Theta(n^2)$ ausgeführt werden, anstatt zu sagen, dass sie Zeit $\Theta(4n^2)$ benötigen. (Möglicherweise haben Sie bemerkt, dass wir natürlich auch sagen können, dass die vier Matrizenadditionen zusammen $\Theta(4n^2/4)$ benötigen und dass $4n^2/4 = n^2$ gilt. Der Punkt an dieser Stelle ist aber, dass die Θ-Notation konstante Faktoren subsumiert, unabhängig welche Konstanten das sind.) Somit bleiben nur zwei Terme aus $\Theta(n^2)$ übrig, die wir in einem Term vereinigen können.

Wenn wir jedoch die Laufzeit der acht rekursiven Aufrufe betrachten, können wir nicht einfach den konstanten Faktor von 8 subsumieren. Anders formuliert, wir müssen angeben, dass diese rekursiven Aufrufe zusammen $8\,T(n/2)$ Zeit und nicht nur $T(n/2)$ Zeit benötigen. Warum das so ist, können wir uns klarmachen, wenn wir nochmals auf den Rekursionsbaum aus Abbildung 2.5 schauen – dieser Reduktionsbaum illustriert die Rekursionsgleichung (2.1), $T(n) = 2\,T(n/2) + \Theta(n)$, die identisch zu der Rekursionsgleichung (4.7) ist. Der Faktor von 2 gibt an, wie viele Kinder jeder Baumknoten hat,

was wiederum angibt, wie viele Terme zur Summe einer jeden Baumebene beitragen. Würden wir den Faktor von 8 in Gleichung (4.16) ignorieren, wäre der Rekursionsbaum nur linear anstatt „buschig“ und jede Ebene würde nur einen Term zur Summe beitragen.

Sie sollten sich aus diesem Grund merken, dass, wenngleich asymptotische Notationen konstante multiplikative Faktoren subsumieren, rekursive Notationen, wie z. B. $T(n/2)$, dies nicht tun.

Strassens Methode

Der entscheidende Punkt bei Strassens Methode ist, dass der Rekursionsbaum weniger buschig ist. Genauer, anstatt acht rekursive Multiplikationen von $n/2 \times n/2$-Matrizen auszuführen, führt Strassens Methode nur sieben aus. Den Preis, den wir für das Eliminieren einer Matrizenmultiplikation bezahlen müssen, besteht darin, dass wir zusätzliche Additionen von $n/2 \times n/2$-Matrizen auszuführen haben, aber weiterhin immer noch nur konstant viele. Wie vorhin wird die konstante Anzahl von Matrizenadditionen durch die Θ-Notation subsumiert, wenn wir die Rekursionsgleichung zur Beschreibung der Laufzeit aufstellen.

Strassens Methode ist keinesfalls naheliegend. (Das könnte die größte Untertreibung in diesem Buch sein.) Die Methode besteht aus vier Schritten:

1. Teilen Sie die Eingabematrizen A und B sowie die Ausgabematrix C wie in Gleichung (4.9) angegeben in $n/2 \times n/2$-Teilmatrizen. Dieser Schritt kann in $\Theta(1)$ Zeit ausgeführt werden, wenn wie in SQUARE-MATRIX-MULTIPLY-RECURSIVE mit Indexberechnung gearbeitet wird,

2. Konstruieren Sie 10 Matrizen $S_1, S_2, \ldots, S_{10}$, die jeweils die Größe $n/2 \times n/2$ haben und die Summe oder die Differenz von zwei Matrizen, die in Schritt 1 konstruiert wurden, darstellen. Wir können diese 10 Matrizen in $\Theta(n^2)$ Zeit konstruieren.

3. Berechnen Sie rekursiv sieben Matrizenprodukte $P_1, P_2, \ldots, P_7$ unter Zuhilfenahme der Teilmatrizen, die in Schritt 1 generiert worden sind, und der 10 Matrizen, die in Schritt 2 berechnet worden sind. Jede Matrix P_i ist eine $n/2 \times n/2$-Matrix.

4. Berechnen Sie die gewünschten Teilmatrizen $C_{11}, C_{12}, C_{21}, C_{22}$ der Ergebnismatrix C durch Addieren und Subtrahieren geeigneter Kombinationen der P_i Matrizen. Wir können alle vier Teilmatrizen in $\Theta(n^2)$ Zeit berechnen.

Wir werden uns die Details zu den Schritten 2–4 gleich anschauen, aber wir haben bereits jetzt ausreichend Informationen, um die Rekursionsgleichung für die Laufzeit von Strassens Methode aufzustellen. Wir nehmen an, dass wir, wie bereits in Zeile 4 von SQUARE-MATRIX-MULTIPLY-RECURSIVE, eine einfache skalare Multiplikation auszuführen haben, wenn die Matrizengröße n gleich 1 geworden ist. Im Fall $n > 1$ benötigen die Schritte 1, 2 und 4 zusammen $\Theta(n^2)$ Zeit. Schritt 3 verlangt von uns, sieben Multiplikationen von $n/2 \times n/2$-Matrizen auszuführen. Wir erhalten demnach die folgende

Rekursionsgleichung für die Laufzeit $T(n)$ von Strassens Algorithmus:

$$T(n) = \begin{cases} \Theta(1) & \text{falls } n = 1 \ , \\ 7\,T(n/2) + \Theta(n^2) & \text{falls } n > 1 \ . \end{cases} \tag{4.18}$$

Wir haben eine Matrizenmultiplikation gegen eine konstante Anzahl von Matrizenadditionen eingetauscht. Sobald wir Rekursionsgleichungen und ihre Auflösungen verstanden haben, werden wir sehen, dass dieser Handel tatsächlich zu einer kleineren asymptotischen Laufzeit führt. Mit Hilfe der Mastermethode aus Abschnitt 4.5 können wir folgern, dass die Rekursionsgleichung (4.18) die Lösung $T(n) = \Theta(n^{\lg 7})$ besitzt.

Wir kommen nun zu den Details der Methode. In Schritt 1 konstruieren wir die folgenden 10 Matrizen:

$$\begin{aligned}
S_1 &= B_{12} - B_{22} \ , \\
S_2 &= A_{11} + A_{12} \ , \\
S_3 &= A_{21} + A_{22} \ , \\
S_4 &= B_{21} - B_{11} \ , \\
S_5 &= A_{11} + A_{22} \ , \\
S_6 &= B_{11} + B_{22} \ , \\
S_7 &= A_{12} - A_{22} \ , \\
S_8 &= B_{21} + B_{22} \ , \\
S_9 &= A_{11} - A_{21} \ , \\
S_{10} &= B_{11} + B_{12} \ .
\end{aligned}$$

Da wir zehnmal jeweils zwei $n/2 \times n/2$-Matrizen addieren oder subtrahieren müssen, benötigt dieser Schritt in der Tat $\Theta(n^2)$ Zeit.

In Schritt 3 multiplizieren wir siebenmal rekursiv $n/2 \times n/2$-Matrizen, um die folgenden $n/2 \times n/2$-Matrizen zu berechnen. Jede dieser Matrizen lässt sich auch als Summe oder Differenz von Produkten von Teilmatrizen von A und B schreiben:

$$\begin{aligned}
P_1 &= A_{11} \cdot S_1 &&= A_{11} \cdot B_{12} - A_{11} \cdot B_{22} \ , \\
P_2 &= S_2 \cdot B_{22} &&= A_{11} \cdot B_{22} + A_{12} \cdot B_{22} \ , \\
P_3 &= S_3 \cdot B_{11} &&= A_{21} \cdot B_{11} + A_{22} \cdot B_{11} \ , \\
P_4 &= A_{22} \cdot S_4 &&= A_{22} \cdot B_{21} - A_{22} \cdot B_{11} \ , \\
P_5 &= S_5 \cdot S_6 &&= A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} \ , \\
P_6 &= S_7 \cdot S_8 &&= A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22} \ , \\
P_7 &= S_9 \cdot S_{10} &&= A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12} \ .
\end{aligned}$$

Nehmen Sie bitte zur Kenntnis, dass die einzigen Multiplikationen, die wir auszuführen haben, die sind, die in der mittleren Spalte der obigen Gleichungen stehen. Die rechte Spalte gibt lediglich an, wie diese Produkte von den ursprünglichen, in Schritt 1 konstruierten Teilmatrizen abhängen.

Schritt 4 addiert und subtrahiert die P_i Matrizen, die in Schritt 3 erzeugt worden sind, um die vier $n/2 \times n/2$ Teilmatrizen des Produktes C zu erhalten. Wir beginnen mit

$$C_{11} = P_5 + P_4 - P_2 + P_6 \ .$$

Expandiert man die rechte Seite dieser Gleichung – für eine bessere Übersicht ordnen wir jedem P_i eine eigene Zeile zu und schreiben die Terme, die sich gegenseitig herauskürzen, übereinander –, so sehen wir, dass sich C_{11} zu

$$\begin{array}{ccccccc}
A_{11}\cdot B_{11} & +A_{11}\cdot B_{22} & +A_{22}\cdot B_{11} & +A_{22}\cdot B_{22} & & & \\
 & & -A_{22}\cdot B_{11} & & +A_{22}\cdot B_{21} & & \\
 & -A_{11}\cdot B_{22} & & & & -A_{12}\cdot B_{22} & \\
 & & & -A_{22}\cdot B_{22} & -A_{22}\cdot B_{21} & +A_{12}\cdot B_{22} & +A_{12}\cdot B_{21} \\
\hline
A_{11}\cdot B_{11} & & & & & & +A_{12}\cdot B_{21}
\end{array}$$

ergibt, was der Gleichung (4.11) entspricht.

Gleichermaßen setzen wir

$$C_{12} = P_1 + P_2$$

und erhalten für C_{12}

$$\begin{array}{ccc}
A_{11}\cdot B_{12} & -A_{11}\cdot B_{22} & \\
 & +A_{11}\cdot B_{22} & +A_{12}\cdot B_{22} \\
\hline
A_{11}\cdot B_{12} & & +A_{12}\cdot B_{22}\ .
\end{array}$$

Dies entspricht Gleichung (4.12).

Die Zuweisung

$$C_{21} = P_3 + P_4$$

ergibt für C_{21}

$$\begin{array}{ccc}
A_{21}\cdot B_{11} & +A_{22}\cdot B_{11} & \\
 & -A_{22}\cdot B_{11} & +A_{22}\cdot B_{21} \\
\hline
A_{21}\cdot B_{11} & & +A_{22}\cdot B_{21}\ ,
\end{array}$$

was der Gleichung (4.13) entspricht.

Schlussendlich setzen wir

$$C_{22} = P_5 + P_1 - P_3 - P_7\ ,$$

sodass C_{22} gleich

$$\begin{array}{ccccccc}
A_{11}\cdot B_{11} & +A_{11}\cdot B_{22} & +A_{22}\cdot B_{11} & +A_{22}\cdot B_{22} & & & \\
 & -A_{11}\cdot B_{22} & & & +A_{11}\cdot B_{12} & & \\
 & & -A_{22}\cdot B_{11} & & & -A_{21}\cdot B_{11} & \\
-A_{11}\cdot B_{11} & & & & -A_{11}\cdot B_{12} & +A_{21}\cdot B_{11} & +A_{21}\cdot B_{12} \\
\hline
 & & & A_{22}\cdot B_{22} & & & +A_{21}\cdot B_{12}
\end{array}$$

ist, was wiederum Gleichung (4.14) entspricht. Insgesamt haben wir in Schritt 4 achtmal $n/2 \times n/2$-Matrizen addiert oder subtrahiert. Somit benötigt dieser Schritt $\Theta(n^2)$ Zeit.

Wir sehen also, dass Strassens Algorithmus, der aus den Schritten 1–4 besteht, das korrekte Matrizenprodukt berechnet und dass die Rekursionsgleichung (4.18) seine Laufzeit beschreibt. Da wir in Abschnitt 4.5 sehen werden, dass diese Rekursionsgleichung die Lösung $T(n) = \Theta(n^{\lg 7})$ besitzt, ist Strassens Methode asymptotisch schneller als die einfache SQUARE-MATRIX-MULTIPLY Prozedur. Die Bemerkungen am Ende dieses Kapitels greifen noch verschiedene praktische Aspekte von Strassens Algorithmus auf.

Übungen

Bemerkung: Auch wenn die Übungen 4.2-3, 4.2-4 und 4.2-5 Varianten von Strassens Algorithmus behandeln, sollten Sie Abschnitt 4.5 lesen, bevor Sie versuchen, diese Aufgaben zu lösen.

4.2-1 Berechnen Sie das Matrizenprodukt

$$\begin{pmatrix} 1 & 3 \\ 7 & 5 \end{pmatrix} \begin{pmatrix} 6 & 8 \\ 4 & 2 \end{pmatrix}$$

mit Hilfe von Strassens Algorihmus.

4.2-2 Geben Sie den Pseudocode für Strassens Algorithmus an.

4.2-3 Wie würden Sie Strassens Algorithmus modifizieren, um $n \times n$-Matrizen, bei denen n keine Potenz von 2 ist, miteinander zu multiplizieren? Zeigen Sie, dass der resultierende Algorithmus in Zeit $\Theta(n^{\lg 7})$ läuft.

4.2-4 Welcher ist der größte Wert k, für den Folgendes gilt: Wenn Sie 3×3-Matrizen unter Verwendung von k Multiplikationen (ohne die Kommutativität der Multiplikation anzunehmen) miteinander multiplizieren können, dann können Sie $n \times n$-Matrizen in Zeit $o(n^{\lg 7})$ miteinander multiplizieren. Wie groß wäre die Laufzeit dieses Algorithmus?

4.2-5 V. Pan hat eine Möglichkeit gefunden, 68×68-Matrizen unter Verwendung von 132.464 Multiplikationen miteinander zu multiplizieren, eine Möglichkeit 70×70-Matrizen unter Verwendung von 143.640 Multiplikationen miteinander zu multiplizieren und eine Möglichkeit 72×72-Matrizen unter Verwendung von 155.424 Multiplikationen miteinander zu multiplizieren. Welche Methode liefert die beste asymptotische Laufzeit, wenn diese innerhalb eines Teile-und-Beherrsche-Algorithmus zur Matrizenmultiplikation verwendet wird? Wie ist sie im Vergleich zur Laufzeit von Strassens Algorithmus?

4.2-6 Wie schnell können Sie eine $kn \times n$-Matrix mit einer $n \times kn$-Matrix multiplizieren, wenn Sie Strassens Algorithmus als Unterprogramm verwenden? Beantworten Sie die gleiche Frage, wenn die Reihenfolge der Eingabematrizen vertauscht ist.

4.2-7 Zeigen Sie, wie man komplexe Zahlen $a + b\,i$ and $c + d\,i$ mit nur drei Multiplikationen reeller Zahlen multiplizieren kann. Der Algorithmus sollte a, b, c und d als Eingabe bekommen und den Realteil $ac - bd$ sowie den Imaginärteil $ad + bc$ getrennt ausgeben.

4.3 Die Substitutionsmethode zum Lösen von Rekursionsgleichungen

Jetzt, wo wir gesehen haben, wie Rekursionsgleichungen die Laufzeit von Teile-und-Beherrsche-Algorithmen beschreiben können, werden wir lernen wie man Rekursionsgleichungen lösen kann. Wir beginnen in diesem Abschnitt mit der Substitutionsmethode.

Die Substitutionsmethode zur Lösung von Rekursionsgleichungen besteht aus zwei Schritten:

1. Erraten Sie die Form der Lösung.
2. Verwenden Sie mathematische Induktion, um die Konstanten zu finden und zu zeigen, dass die Lösung okay ist.

Wir ersetzen die Funktion durch die erratene Lösung, wenn wir die Induktionsannahme auf kleinere Werte anwenden; aus diesem Grunde der Name „Substitutionsmethode“. Diese Methode ist mächtig, wir müssen aber fähig sein, die Form der Lösung zu erraten, um sie anwenden zu können.

Wir können die Substitutionsmethode anwenden, um entweder obere oder untere Schranken für Rekursionsgleichungen aufzustellen. Als Beispiel wollen wir eine obere Schranke für die Rekursionsgleichung

$$T(n) = 2\,T(\lfloor n/2 \rfloor) + n \tag{4.19}$$

beweisen, die den Rekursionsgleichungen (4.3) und (4.4) ähnelt. Wir vermuten, dass die Lösung $T(n) = O(n \lg n)$ ist. Die Substitutionsmethode verlangt von uns, zu beweisen, dass für eine geeignete Wahl der Konstanten $c > 0$ die Gleichung $T(n) \leq c\,n \lg n$ gilt. Zuerst nehmen wir an, dass diese Schranke für alle positiven $m < n$, im Besonderen für $m = \lfloor n/2 \rfloor$ gilt. Mit dieser Annahme erhalten wir $T(\lfloor n/2 \rfloor) \leq c\,\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$. Das Einsetzen in die Rekursionsgleichung führt zu

$$\begin{aligned}
T(n) &\leq 2(c\,\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \\
&\leq c\,n \lg(n/2) + n \\
&= c\,n \lg n - c\,n \lg 2 + n \\
&= c\,n \lg n - c\,n + n \\
&\leq c\,n \lg n\ ,
\end{aligned}$$

wobei der letzte Schritt richtig ist, solange $c \geq 1$ gilt.

Mathematische Induktion verlangt von uns, zu zeigen, dass unsere Lösung die Randbedingungen erfüllt. Üblicherweise machen wir dies, indem wir zeigen, dass die Randbedingungen als Induktionsanfang für den induktiven Beweis geeignet sind. Im Falle der Rekursionsgleichung (4.19) müssen wir zeigen, dass wir die Konstante c hinreichend groß wählen können, sodass die Schranke $T(n) \leq c\,n \lg n$ auch für die Randbedingungen gilt. Diese Forderung kann manchmal zu Problemen führen. Nehmen wir für unsere Argumentation an, dass $T(1) = 1$ die einzige Randbedingung der Rekursionsgleichung ist. Dann führt $n = 1$ in der Schranke $T(n) \leq c\,n \lg n$ zu $T(1) \leq c\,1 \lg 1 = 0$, was im Widerspruch zu $T(1) = 1$ steht. Somit ist der Induktionsanfang unseres Induktionsbeweises nicht erfüllt.

Wir können diese Schwierigkeit beim Beweis der Induktionsvermutung für eine spezielle Randbedingung leicht in den Griff bekommen. In der Rekursionsgleichung (4.19) zum Beispiel können wir ausnutzen, dass asymptotische Notation von uns nur verlangt, die Ungleichung $T(n) \leq c\,n \lg n$ für $n \geq n_0$ zu beweisen, wobei n_0 eine Konstante ist, *die wir wählen dürfen*. Wir behalten die lästige Randbedingung $T(1) = 1$ bei, betrachten sie aber nicht in dem Induktionsbeweis. Wir gehen so vor, dass wir zuerst mal bemerken, dass die Rekursionsgleichung für $n > 3$ nicht direkt von $T(1)$ abhängt. Somit nehmen wir $T(2)$ und $T(3)$ statt $T(1)$ als die Basisfälle des Induktionsbeweises, und setzen $n_0 = 2$. Beachten Sie, dass wir einen Unterschied zwischen dem Basisfall der Rekursionsgleichung ($n = 1$) und dem Induktionsanfang im Induktionsbeweis ($n = 2$ und $n = 3$) machen. Mit $T(1) = 1$ erhalten wir aus der Rekursionsgleichung $T(2) = 4$ und $T(3) = 5$. Wir können nun den Induktionsbeweis, dass $T(n) \leq c\,n \lg n$ für eine Konstante $c \geq 1$ gilt, vervollständigen, indem wir c hinreichend groß wählen, sodass $T(2) \leq c\,2 \lg 2$ und $T(3) \leq c\,3 \lg 3$ gelten. Es stellt sich heraus, dass eine beliebige Konstante $c \geq 2$ gewählt werden kann, damit die Induktionsvermutung für die Basisfälle $n = 2$ und $n = 3$ erfüllt ist. Für die meisten Rekursionsgleichungen, die wir untersuchen werden, ist es einfach, die Randbedingungen so auszubauen, dass die Induktionsannahme für kleine n zutrifft. Wir werden nicht jedesmal die entsprechenden Details ausarbeiten.

Gut raten

Unglücklicherweise gibt es keinen allgemeingültigen Weg, um die korrekten Lösungen von Rekursionsgleichungen zu erraten. Dies setzt Erfahrung und gelegentlich Kreativität voraus. Glücklicherweise können Sie einige Heuristiken benutzen, die Ihnen helfen, gut zu raten. Sie können zum Generieren guter Vermutungen auch Rekursionsbäume verwenden, die wir in Abschnitt 4.4 vorstellen werden.

Wenn eine Rekursionsgleichung einer anderen ähnlich ist, die Sie bereits kennen gelernt haben, dann ist es naheliegend, eine ähnliche Lösung zu vermuten. Wir betrachten beispielsweise die Rekursionsgleichung

$$T(n) = 2\,T(\lfloor n/2 \rfloor + 17) + n \ ,$$

die wegen der auf der rechten Seite im Argument von T hinzugefügten „17“ schwierig erscheint. Die Intuition sagt uns jedoch, dass dieser zusätzliche Term keinen substanziellen Effekt auf die Lösung der Rekursionsgleichung ausüben kann. Wenn n groß ist,

dann ist der Unterschied zwischen $\lfloor n/2 \rfloor$ und $\lfloor n/2 \rfloor + 17$ nicht allzu groß; beide halbieren n in etwa. Folglich vermuten wir, dass $T(n) = O(n \lg n)$ gilt, was Sie mit Hilfe der Substitutionsmethode als korrekt verifizieren können (siehe Übung 4.3-6).

Eine andere Möglichkeit, gute Vermutungen aufzustellen, besteht darin, grobe obere und untere Schranken für die Rekursionsgleichung zu beweisen und damit den Bereich der Unsicherheit einzuschränken. Wir könnten zum Beispiel im Fall der Rekursionsgleichung (4.19) mit der unteren Schranke $T(n) = \Omega(n)$ beginnen, da der Term n in der Rekursionsgleichung auftaucht. Und wir können zeigen, dass $T(n) = O(n^2)$ eine erste obere Schranke ist. Dann senken wir allmählich die obere Schranke und erhöhen die untere Schranke, bis das Verfahren zur korrekten, asymptotischen Lösung von $T(n) = \Theta(n \lg n)$ konvergiert.

Feinheiten

Manchmal werden Sie möglicherweise zwar die asymptotische Schranke einer Rekursionsgleichung korrekt erraten haben, aber irgendwie scheitert die Mathematik daran, die Schranke im Rahmen einer Induktion zu beweisen. Gewöhnlich besteht das Problem darin, dass die Induktionsannahme nicht stark genug ist, um die genaue Schranke zu beweisen. Wenn Sie in einem solchen Fall Ihre Vermutung verändern, indem Sie einen Term niedrigerer Ordnung subtrahieren, dann führt dies oft zum Erfolg.

Betrachten Sie die Rekursionsgleichung

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1 \ .$$

Wir vermuten, dass die Lösung $T(n) = O(n)$ ist und versuchen zu zeigen, dass für eine geeignete Wahl der Konstanten c die Ungleichung $T(n) \leq cn$ erfüllt ist. Setzen wir unsere Vermutung in die Rekursionsgleichung ein, dann erhalten wir

$$\begin{aligned} T(n) &\leq c \lfloor n/2 \rfloor + c \lceil n/2 \rceil + 1 \\ &= cn + 1 \end{aligned}$$

anstelle der gewünschten Ungleichung $T(n) \leq cn$. Möglicherweise sind wir versucht, eine größere Schätzung auszuprobieren, sagen wir $T(n) = O(n^2)$. Wenngleich wir diese erratene größere Schranke für unsere Rekursionsgleichung beweisen können, ist unsere ursprüngliche Vermutung $T(n) = O(n)$ korrekt. Um zu beweisen, dass sie korrekt ist, müssen wir eine stärkere Induktionsannahme wählen.

Unsere Vermutung ist intuitiv fast richtig: Uns fehlt nur eine Konstante 1, ein Term niedrigerer Ordnung. Trotzdem arbeitet die mathematische Induktion nicht, es sei denn wir beweisen die exakte Form der Induktionsannahme. Wir bewältigen unsere Schwierigkeiten, indem wir einen Term niedrigerer Ordnung von unserer vorherigen Vermutung *subtrahieren*. Unsere neue Vermutung lautet $T(n) \leq cn - d$, wobei $d \geq 0$ konstant ist. Wir haben nun

$$\begin{aligned} T(n) &\leq (c \lfloor n/2 \rfloor - d) + (c \lceil n/2 \rceil - d) + 1 \\ &= cn - 2d + 1 \\ &\leq cn - d \ , \end{aligned}$$

sofern $d \geq 1$ gilt. Wie vorhin muss die Konstante c hinreichend groß gewählt werden, um die Randbedingungen in den Griff zu bekommen.

Möglicherweise finden Sie die Idee, einen Term niedriger Ordnung zu subtrahieren, nicht eingängig. Sollten wir nicht besser unsere Schätzung erhöhen, wenn die Mathematik unsere Vermutung nicht beweisen kann? Nicht unbedingt! Es kann tatsächlich bei einem Induktionsbeweis einer oberen Schranke sein, dass es schwieriger ist, eine schwächere Schranke zu beweisen, da um die schwächere Schranke zu beweisen, wir die gleiche schwächere Schranke induktiv im Beweis verwenden müssen. In unserem aktuellen Beispiel haben wir den Term niedriger Ordnung für jeden rekursiven Term jeweils einmal abzuziehen. Im obigen Beispiel subtrahieren wir die Konstante d zweimal, einmal für den $T(\lfloor n/2 \rfloor)$-Term und einmal für den $T(\lceil n/2 \rceil)$-Term. Somit kamen wir zu der Ungleichung $T(n) \leq cn - 2d + 1$ und es war dann einfach, Werte für d zu finden, sodass $cn - 2d + 1$ kleiner oder gleich $cn - d$ ist.

Fallen vermeiden

Es ist leicht, sich bei der Verwendung der asymptotischen Notation zu vertun. Im Fall der Rekursionsgleichung (4.19) können wir zum Beispiel fälschlicherweise die Aussage $T(n) = O(n)$ „beweisen", indem wir vermuten, dass $T(n) \leq cn$ gilt und anschließend wie folgt argumentieren:

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor) + n \\ &\leq cn + n \\ &= O(n) \ , \qquad \Longleftarrow \textit{falsch!!} \end{aligned}$$

da c eine Konstante ist. Der Fehler besteht darin, dass wir nicht die *exakte Form* $T(n) \leq cn$ der Induktionsannahme bewiesen haben. Wir werden aus diesem Grund explizit beweisen, dass $T(n) \leq cn$ gilt, wenn wir $T(n) = O(n)$ zeigen wollen.

Variablen transformieren

Manchmal können Sie eine Ihnen unbekannte Rekursionsgleichung durch eine kleine algebraische Umformung auf eine Ihnen bereits bekannte Gleichung zurückführen. Als Beispiel betrachten wir die Rekursionsgleichung

$$T(n) = 2\,T(\lfloor \sqrt{n} \rfloor) + \lg n \ ,$$

die schwierig aussieht. Wir können diese Rekursionsgleichung jedoch durch eine Variablentransformation vereinfachen. Der Bequemlichkeit halber werden wir uns jetzt nicht um die Rundung der Werte kümmern, zum Beispiel darum, ob $\sqrt{n}$ ganzzahlig ist. Definieren wir m als $\lg n$ und ersetzen wir n entsprechend, dann erhalten wir

$$T(2^m) = 2\,T(2^{m/2}) + m \ .$$

Nun können wir die Umbenennung $S(m) = T(2^m)$ vornehmen, um die neue Rekursionsgleichung

$$S(m) = 2S(m/2) + m$$

zu erhalten, die sehr ähnlich zur Rekursionsgleichung (4.19) ist. Diese Rekursionsgleichung hat in der Tat die gleiche Lösung: $S(m) = O(m \lg m)$. Transformieren wir $S(m)$ zu $T(n)$ zurück, erhalten wir $T(n) = T(2^m) = S(m) = O(m \lg m) = O(\lg n \lg \lg n)$.

Übungen

4.3-1 Zeigen Sie, dass die Lösung von $T(n) = T(n-1) + n$ in $O(n^2)$ liegt.

4.3-2 Zeigen Sie, dass die Lösung von $T(n) = T(\lceil n/2 \rceil) + 1$ in $O(\lg n)$ liegt.

4.3-3 Wir haben gesehen, dass die Lösung von $T(n) = 2\,T(\lfloor n/2 \rfloor) + n$ in $O(n \lg n)$ liegt. Zeigen Sie, dass die Lösung dieser Rekursionsgleichung auch in $\Omega(n \lg n)$ liegt. Schlussfolgern Sie, dass die Lösung dann auch in $\Theta(n \lg n)$ liegt.

4.3-4 Zeigen Sie, dass wir bei der Rekursionsgleichung (4.19) durch das Aufstellen einer anderen Induktionsannahme die Schwierigkeiten mit der Randbedingung $T(1) = 1$ umgehen können, ohne die Randbedingungen für den Induktionsbeweis anpassen zu müssen.

4.3-5 Zeigen Sie, dass $\Theta(n \lg n)$ die Lösung für die „exakte" Rekursionsgleichung (4.3) von Sortieren durch Mischen ist.

4.3-6 Zeigen Sie, dass die Lösung von $T(n) = 2\,T(\lfloor n/2 \rfloor + 17) + n$ in $O(n \lg n)$ liegt.

4.3-7 Mit Hilfe der Mastermethode aus Abschnitt 4.5 können Sie zeigen, dass die Lösung der Rekursionsgleichung $T(n) = 4\,T(n/3) + n$ gleich $T(n) = \Theta(n^{\log_3 4})$ ist. Zeigen Sie, dass ein Substitutionsbeweis mit der Vermutung $T(n) \leq c\,n^{\log_3 4}$ scheitert. Zeigen Sie dann, wie Sie durch Subtraktion eines Terms niedriger Ordnung einen Substitutionsbeweis hinbekommen.

4.3-8 Mit Hilfe der Mastermethode aus Abschnitt 4.5 können Sie zeigen, dass die Lösung der Rekursionsgleichung $T(n) = 4\,T(n/2) + n$ gleich $T(n) = \Theta(n^2)$ ist. Zeigen Sie, dass ein Substitutionsbeweis mit der Vermutung $T(n) \leq c\,n^2$ scheitert. Zeigen Sie dann, wie Sie durch Subtraktion eines Terms niedriger Ordnung einen Substitutionsbeweis hinbekommen.

4.3-9 Lösen Sie die Rekursionsgleichung $T(n) = 3\,T(\sqrt{n}) + \log n$, indem Sie die Variablen geeignet transformieren. Ihre Lösung sollte asymptotisch scharf sein. (*Hinweis:* Machen Sie sich keine Sorgen, wenn Ihre Werte Integralwerte sind.)

4.4 Die Rekursionsbaum-Methode zum Lösen von Rekursionsgleichungen

Wenngleich Sie die Substitutionsmethode benutzen können, um einen prägnanten Beweis zu bekommen, dass eine Lösung in Bezug auf eine Rekursionsgleichung korrekt ist, könnten Sie Probleme haben, eine gute Vermutung aufzustellen. Das Zeichnen eines Rekursionsbaumes, wie wir es bei unserer Analyse von Sortieren durch Mischen in

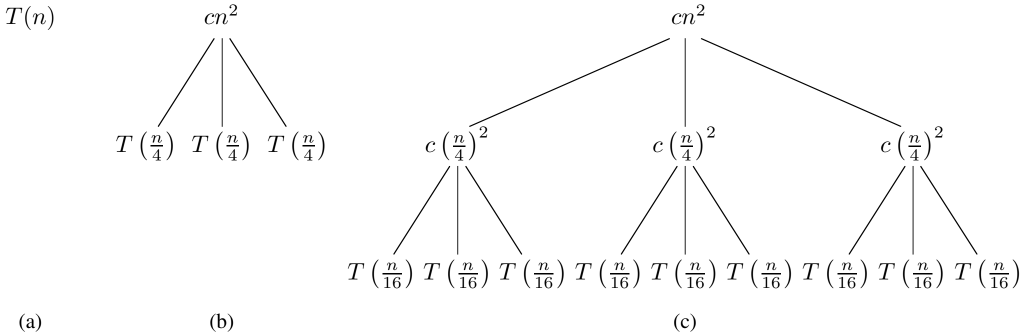
Abschnitt 2.3.2 getan haben, ist ein direkter Weg, eine gute Vermutung zu erhalten. In einem ***Rekursionsbaum*** stellt jeder Knoten die Kosten eines Teilproblems dar, das sich irgendwo in der Menge der rekursiven Funktionsaufrufe befindet. Wir addieren die Kosten innerhalb jeder Ebene des Baumes. Anschließend summieren wir über die Kosten der Ebenen und bestimmen so die Gesamtkosten der Rekursion.

Den meisten Nutzen eines Rekursionsbaumes haben Sie, wenn Sie ihn zum Aufstellen einer guten Vermutung benutzen, die dann mithilfe der Substitutionsmethode überprüft werden kann. Wenn Sie einen Rekursionsbaum zum Aufstellen einer guten Vermutung verwenden, dann können Sie sich häufig ein gewisses Maß an „Unschärfe“ erlauben, da Sie ja Ihre Vermutung später noch verifizieren werden. Wenn Sie beim Entwerfen des Rekursionsbaumes und der Summierung der Kosten sehr sorgsam vorgehen, dann können Sie einen Rekursionsbaum auch als einen direkten Beweis einer Lösung einer Rekursionsgleichung verwenden. In diesem Abschnitt werden wir einen Rekursionsbaum zum Aufstellen einer guten Vermutung verwenden. In Abschnitt 4.6 werden wir den Rekursionsbaum direkt zum Beweis des Theorems benutzen, das die Basis für die Mastermethode bildet.

Sehen wir uns an, wie uns ein Rekursionsbaum eine gute Vermutung zum Beispiel für die Rekursionsgleichung $T(n) = 3\,T(\lfloor n/4 \rfloor) + \Theta(n^2)$ liefert. Wir beginnen damit, uns auf das Auffinden einer oberen Schranke für die Lösung zu konzentrieren. Da wir wissen, dass das Aufrunden und das Abrunden beim Lösen von Rekursionsgleichungen in der Regel unerheblich sind (das ist ein Beispiel für die Unschärfe, die wir tolerieren können), entwerfen wir einen Rekursionsbaum für die Rekursionsgleichung $T(n) = 3\,T(n/4) + c\,n^2$, wobei wir den implizit enthaltenen Koeffizienten $c > 0$ ausgeschrieben haben.

Abbildung 4.5 zeigt, wie wir den Rekursionsbaum für $T(n) = 3\,T(n/4) + c\,n^2$ ableiten. Der Einfachheit halber nehmen wir an, dass n eine Potenz von 4 ist (ein anderes Beispiel für die tolerierbare Unschärfe), sodass $n/4$ für alle $n \geq 4$ eine ganze Zahl ist. Teil (a) der Abbildung zeigt $T(n)$, den wir in Teil (b) zu einem äquivalenten Baum, der die Rekursionsgleichung darstellt, expandieren. Der Term $c\,n^2$ in der Wurzel repräsentiert die „direkten Kosten“ von $T(n)$, d. h. die Kosten ohne die Kosten des rekursiven Abstiegs. Die drei von der Wurzel ausgehenden Teilbäume repräsentieren die aus den Teilproblemen der Größe $n/4$ hinzuzuziehenden Kosten. Teil (c) zeigt diesen Prozess expandiert um einen weiteren Schritt, indem jeder Knoten mit den Kosten $T(n/4)$ aus Teil (b) aufgefächert wird. Die Kosten jeder der drei Kinder der Wurzel sind $c(n/4)^2$. Wir setzen die Expansion der Knoten fort, indem wir jeden Knoten in die durch die Rekursionsgleichung vorgegebenen Bestandteile zerlegen.

Da sich die Größen der Teilprobleme jedesmal um einen Faktor von 4 verringern, wenn wir um eine weitere Ebene absteigen, müssen wir letztendlich an einer Randbedingung ankommen. In welcher Entfernung von der Wurzel erreichen wir diese? Die Größe des Teilproblems für einen Knoten der Tiefe i ist $n/4^i$. Somit erreichen wir Teilprobleme der Größe $n = 1$, wenn $n/4^i = 1$ ist, d. h. wenn $i = \log_4 n$ gilt. Also hat der Baum $\log_4 n + 1$ Ebenen (der Tiefen $0, 1, 2, \ldots, \log_4 n$).



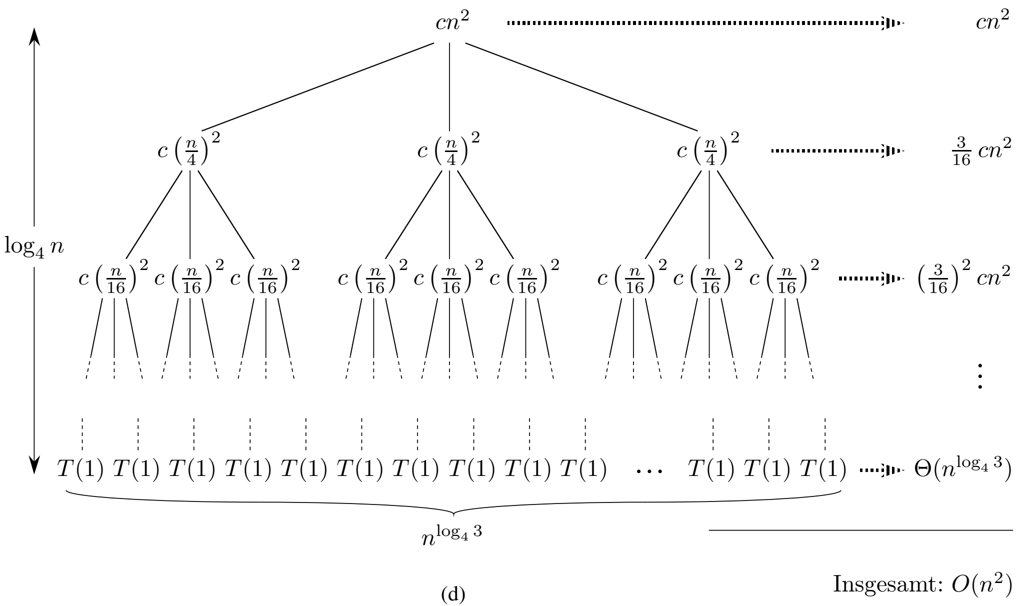


Abbildung 4.5: *Konstruieren eines Rekursionsbaumes für die Rekursionsgleichung* $T(n) = 3\,T(n/4) + c\,n^2$. *Teil **(a)** zeigt* $T(n)$, *der in **(b)-(d)** schrittweise zum Rekursionsbaum aufgefächert wird. Der voll expandierte Baum, der in Teil **(d)** zu sehen ist, besitzt die Höhe* $\log_4 n$ *(er verfügt über* $\log_4 n + 1$ *Ebenen).*

Als nächstes bestimmen wir die Kosten in jeder Ebene des Baumes. Jede Ebene hat dreimal so viele Knoten als die jeweils darüberliegende. Die Anzahl der Knoten in der Tiefe i beträgt somit 3^i. Da sich die Größe der Teilprobleme ausgehend von der Wurzel in jeder Ebene um den Faktor 4 reduziert, hat jeder Knoten der Tiefe i, für $i = 0, 1, 2, \ldots, \log_4 n - 1$ die Kosten $c\,(n/4^i)^2$. Multiplizieren wir dies aus, sehen wir, dass sich für die Gesamtkosten aller Knoten einer Ebene i mit $i = 0, 1, 2, \ldots, \log_4 n - 1$ der Wert $3^i c\,(n/4^i)^2 = (3/16)^i c\,n^2$ ergibt. Die untere Ebene, d. h. die Ebene der Tiefe $\log_4 n$, besitzt $3^{\log_4 n} = n^{\log_4 3}$ Knoten, von denen jeder die Kosten $T(1)$ zu den Gesamtkosten beiträgt. Daraus ergeben sich für die unterste Ebene Gesamtkosten von $n^{\log_4 3}T(1)$, was in $\Theta(n^{\log_4 3})$ liegt, da wir annehmen, dass $T(1)$ eine Konstante ist.

Nun addieren wir die Kosten aller Ebenen, um die Kosten des gesamten Baumes zu bestimmen:

$$\begin{aligned}
T(n) &= c\,n^2 + \frac{3}{16}\,c\,n^2 + \left(\frac{3}{16}\right)^2 c\,n^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n - 1} c\,n^2 + \Theta(n^{\log_4 3}) \\
&= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i c\,n^2 + \Theta(n^{\log_4 3}) \\
&= \frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1}\,c\,n^2 + \Theta(n^{\log_4 3}) \qquad \text{(wegen Gleichung (A.5))}\ .
\end{aligned}$$

Diese letzte Formel erscheint etwas kompliziert. Allerdings stellen wir fest, dass wir wiederum Nutzen aus einer unscharfen Betrachtungsweise ziehen können, indem wir als obere Schranke eine unendliche, fallende geometrische Reihe verwenden. Gehen wir eine Zeile zurück und wenden wir Gleichung (A.6) an, erhalten wir

$$\begin{aligned}
T(n) &= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i c\,n^2 + \Theta(n^{\log_4 3}) \\
&< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i c\,n^2 + \Theta(n^{\log_4 3}) \\
&= \frac{1}{1 - (3/16)}\,c\,n^2 + \Theta(n^{\log_4 3}) \\
&= \frac{16}{13}\,c\,n^2 + \Theta(n^{\log_4 3}) \\
&= O(n^2)\ .
\end{aligned}$$

Wir haben für unsere ursprüngliche Rekursionsgleichung $T(n) = 3\,T(\lfloor n/4 \rfloor) + \Theta(n^2)$ somit die Vermutung $T(n) = O(n^2)$ abgeleitet. In diesem Beispiel bilden die Koeffizienten von $c\,n^2$ eine fallende geometrische Reihe. Aus Gleichung (A.6) folgt, dass die Summe dieser Koeffizienten von oben durch die Konstante $16/13$ beschränkt ist. Da der Beitrag der Wurzel an den Gesamtkosten $c\,n^2$ beträgt, trägt die Wurzel einen konstanten Bruchteil zu den Gesamtkosten bei. Mit anderen Worten, die Kosten der Wurzel dominieren die Gesamtkosten.

Wenn $O(n^2)$ tatsächlich eine obere Schranke der Rekursionsgleichung ist (was wir gleich zeigen werden), dann muss sie eine scharfe Schranke sein. Warum? Der erste rekursive

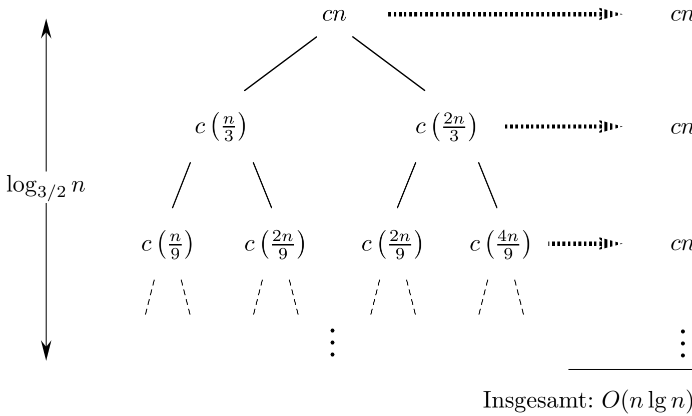


Abbildung 4.6: *Rekursionsbaum für die Rekursionsgleichung* $T(n) = T(n/3) + T(2n/3) + cn$.

Aufruf steuert die Kosten $\Theta(n^2)$ bei, und somit muss $\Omega(n^2)$ eine untere Schranke für die Rekursionsgleichung sein.

Nun können wir die Substitutionsmethode benutzen, um die Korrektheit unserer Vermutung zu verifizieren, nämlich dass $T(n) = O(n^2)$ eine obere Schranke der Rekursionsgleichung $T(n) = 3\,T(\lfloor n/4 \rfloor) + \Theta(n^2)$ ist. Wir wollen zeigen, dass für eine Konstante $d > 0$ die Beziehung $T(n) \leq d\,n^2$ gilt. Verwenden wir dieselbe Konstante $c > 0$ wie vorhin, dann erhalten wir

$$\begin{aligned}
T(n) &\leq 3\,T(\lfloor n/4 \rfloor) + c\,n^2 \\
&\leq 3\,d\,\lfloor n/4 \rfloor^2 + c\,n^2 \\
&\leq 3\,d(n/4)^2 + cn^2 \\
&= \frac{3}{16}\,d\,n^2 + c\,n^2 \\
&\leq d\,n^2 \ ,
\end{aligned}$$

wobei der letzte Schritt korrekt ist, falls $d \geq (16/13)c$ gilt.

Als weiteres, komplizierteres Beispiel zeigt Abbildung 4.6 einen Rekursionsbaum für

$$T(n) = T(n/3) + T(2n/3) + O(n) \ .$$

(Wiederum sparen wir Auf- und Abrunden von Werten der Einfachheit halber aus.) Wie zuvor bezeichnen wir den konstanten Faktor in dem $O(n)$-Term mit c. Addieren wir die Werte in den Ebenen des in der Abbildung gezeigten Rekursionsbaumes, dann erhalten wir für jede Ebene den Betrag $c\,n$. Der längste einfache Pfad von der Wurzel zu einem Blatt ist $n \to (2/3)\,n \to (2/3)^2\,n \to \cdots \to 1$. Da $(2/3)^k\,n = 1$ ist, wenn $k = \log_{3/2} n$ gilt, ist die Höhe des Baumes $\log_{3/2} n$.

Intuitiv erwarten wir, dass die Lösung der Rekursionsgleichung sich aus der Anzahl der Ebenen gewichtet mit den jeweiligen Kosten der Ebenen ergibt, also in $O(c\,n \log_{3/2} n)$,

d. h. in $O(n \lg n)$, liegt. Abbildung 4.6 zeigt nur die oberen Ebenen des Rekursionsbaums, nicht jede Ebene des Baumes trägt jedoch Kosten von cn bei. Betrachten Sie die Kosten der Blätter. Wäre dieser Rekursionsbaum ein vollständiger binärer Baum der Höhe $\log_{3/2} n$, dann gäbe es $2^{\log_{3/2} n} = n^{\log_{3/2} 2}$ Blätter. Da die Kosten jedes Blattes konstant sind, wären die Gesamtkosten der Blätter in $\Theta(n^{\log_{3/2} 2})$, was in $\omega(n \lg n)$ liegt, da $\log_{3/2} 2$ eine Konstante echt größer 0 ist. Dieser Rekursionsbaum ist aber kein vollständiger binärer Baum und hat weniger als $n^{\log_{3/2} 2}$ Blätter. Darüber hinaus fehlen immer mehr innere Knoten, je weiter wir uns von der Wurzel entfernen. Folglich liefern nicht alle Ebenen genau die Kosten von cn; die unteren Ebenen tragen weniger zu den Gesamtkosten bei. Wir könnten eine exakte Bilanz der Kosten aufstellen, aber wir sollten uns daran erinnern, dass wir nur versuchen, eine Vermutung aufzustellen, um sie in der Substitutionsmethode zu verwenden. Lassen Sie uns also diese unscharfe Sichtweise tolerieren und versuchen, zu zeigen, dass unsere Vermutung, dass $O(n \lg n)$ eine obere Schranke ist, korrekt ist.

Tatsächlich können wir die Substitutionsmethode anwenden, um zu verifizieren, dass $O(n \lg n)$ eine obere Schranke für die Lösung der Rekursionsgleichung ist. Wir zeigen, dass $T(n) \leq d\, n \lg n$ für eine geeignete positive Konstante d ist. Es gilt

$$
\begin{aligned}
T(n) &\leq T(n/3) + T(2n/3) + cn \\
&\leq d\,(n/3) \lg(n/3) + d\,(2n/3) \lg(2n/3) + cn \\
&= (d\,(n/3) \lg n - d\,(n/3) \lg 3) \\
&\qquad + (d\,(2n/3) \lg n - d\,(2n/3) \lg(3/2)) + cn \\
&= d\,n \lg n - d\,((n/3) \lg 3 + (2n/3) \lg(3/2)) + cn \\
&= d\,n \lg n - d\,((n/3) \lg 3 + (2n/3) \lg 3 - (2n/3) \lg 2) + cn \\
&= d\,n \lg n - d\,n(\lg 3 - 2/3) + cn \\
&\leq d\,n \lg n\ ,
\end{aligned}
$$

so lange wie $d \geq c/(\lg 3 - (2/3))$ ist. Es ist also nicht notwendig, dass wir eine präzisere Berechnung der Kosten im Rekursionsbaum vornehmen.

Übungen

4.4-1 Benutzen Sie einen Rekursionsbaum, um eine gute asymptotisch obere Schranke für die Rekursionsgleichung $T(n) = 3\,T(\lfloor n/2 \rfloor) + n$ zu bestimmen. Verwenden Sie die Substitutionsmethode, um Ihre Antwort zu verifizieren.

4.4-2 Benutzen Sie einen Rekursionsbaum, um eine gute asymptotisch obere Schranke für die Rekursionsgleichung $T(n) = T(n/2) + n^2$ zu bestimmen. Verwenden Sie die Substitutionsmethode, um Ihre Antwort zu verifizieren.

4.4-3 Benutzen Sie einen Rekursionsbaum, um eine gute asymptotisch obere Schranke für die Rekursionsgleichung $T(n) = 4\,T(n/2 + 2) + n$ zu bestimmen. Verwenden Sie die Substitutionsmethode, um Ihre Antwort zu verifizieren.

4.4-4 Benutzen Sie einen Rekursionsbaum, um eine gute asymptotisch obere Schranke für die Rekursionsgleichung $T(n) = 2\,T(n-1) + 1$ zu bestimmen. Verwenden Sie die Substitutionsmethode, um Ihre Antwort zu verifizieren.

4.4-5 Benutzen Sie einen Rekursionsbaum, um eine gute asymptotisch obere Schranke für die Rekursionsgleichung $T(n) = T(n-1) + T(n/2) + n$ zu bestimmen. Verwenden Sie die Substitutionsmethode, um Ihre Antwort zu verifizieren.

4.4-6 Zeigen Sie, dass die Lösung der Rekursionsgleichung $T(n) = T(n/3) + T(2n/3) + c\,n$ in $\Omega(n \lg n)$ ist, indem Sie auf das Konzept der Rekursionsbäume zurückgreifen. c steht für eine Konstante.

4.4-7 Entwerfen Sie einen Rekursionsbaum für $T(n) = 4\,T(\lfloor n/2 \rfloor) + c\,n$, wobei c eine Konstante ist und geben Sie eine asymptotisch scharfe Schranke für die Lösung der Rekursionsgleichung an. Verifizieren Sie ihre Schranke mithilfe der Substitutionsmethode.

4.4-8 Verwenden Sie einen Rekursionsbaum, um eine asymptotisch scharfe Lösung für die Rekursionsgleichung $T(n) = T(n-a) + T(a) + c\,n$ anzugeben, wobei $a \geq 1$ und $c > 0$ Konstanten sind.

4.4-9 Benutzen Sie einen Rekursionsbaum, um eine asymptotisch scharfe Lösung für die Rekursionsgleichung $T(n) = T(\alpha\,n) + T((1-\alpha)\,n) + c\,n$ zu bestimmen, wobei α eine Konstante aus dem Bereich $0 < \alpha < 1$ ist und $c > 0$ ebenfalls eine Konstante ist.

4.5 Die Mastermethode zum Lösen von Rekursionsgleichungen

Die Mastermethode stellt ein Art „Kochbuchrezept“ für das Lösen von Rekursionsgleichungen der Form

$$T(n) = a\,T(n/b) + f(n) \tag{4.20}$$

dar, wobei $a \geq 1$ und $b > 1$ Konstanten sind und $f(n)$ eine asymptotisch positive Funktion. Um die Mastermethode anwenden zu können, müssen Sie sich drei Fälle einprägen; dann aber werden Sie in der Lage sein, viele Rekursionsgleichungen mehr oder weniger spielend zu lösen, oft sogar ohne Papier und Bleistift.

Die Rekursionsgleichung (4.20) beschreibt die Laufzeit eines Algorithmus, der ein Problem der Größe n in a Teilprobleme zerlegt, jedes mit der Größe n/b, wobei a und b positive Konstanten sind. Die a Teilprobleme werden rekursiv gelöst, jedes in der Zeit $T(n/b)$. Die Funktion $f(n)$ umfasst die Kosten, um das Problem in die Teilprobleme aufzuteilen und um die Lösungen der Teilprobleme zu vereinigen. Bei der Rekursionsgleichung von Strassens Algorithmus beispielsweise ist $a = 7$, $b = 2$ und $f(n) = \Theta(n^2)$.

Im Hinblick auf technische Korrektheit ist die Rekursionsgleichung nicht einmal wohldefiniert, da n/b möglicherweise keine ganze Zahl ist. Das Ersetzen der a Terme $T(n/b)$

durch entweder $T(\lfloor n/b \rfloor)$ oder $T(\lceil n/b \rceil)$ beeinflusst das asymptotische Verhalten der Rekursionsgleichung nicht. (Wir werden diese Aussage im nächsten Abschnitt beweisen.) Aus diesem Grunde lassen wir üblicherweise die Funktionen *floor* und *ceil* aus Bequemlichkeit einfach weg, wenn wir Rekursionsgleichungen für Teile-und-Beherrsche-Algorithmen aufschreiben.

Das Mastertheorem

Die Mastermethode beruht auf folgendem Theorem.

Theorem 4.1: ***(Mastertheorem)***

Seien $a \geq 1$ und $b > 1$ Konstanten. Sei $f(n)$ eine Funktion und sei $T(n)$ über den nichtnegativen ganzen Zahlen durch die Rekursionsgleichung

$$T(n) = a\,T(n/b) + f(n)$$

definiert, wobei wir n/b so interpretieren, dass damit entweder $\lfloor n/b \rfloor$ oder $\lceil n/b \rceil$ gemeint ist. Dann besitzt $T(n)$ die folgenden asymptotischen Schranken:

1. Gilt $f(n) = O(n^{\log_b a - \epsilon})$ für eine Konstante $\epsilon > 0$, dann gilt $T(n) = \Theta(n^{\log_b a})$.
2. Gilt $f(n) = \Theta(n^{\log_b a})$, dann gilt $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. Gilt $f(n) = \Omega(n^{\log_b a + \epsilon})$ für eine Konstante $\epsilon > 0$ und $a\,f(n/b) \leq c\,f(n)$ für eine Konstante $c < 1$ und hinreichend großen n, dann ist $T(n) = \Theta(f(n))$.

Bevor wir das Mastertheorem auf einige Beispiele anwenden, verwenden wir einen Moment darauf, die Aussage zu verstehen. In jedem der drei Fälle vergleichen wir die Funktion $f(n)$ mit der Funktion $n^{\log_b a}$. Offenbar bestimmt die größere der beiden Funktionen die Lösung der Rekursionsgleichung. Wenn, wie im Fall 1, die Funktion $n^{\log_b a}$ die größere Funktion ist, dann ist die Lösung durch $T(n) = \Theta(n^{\log_b a})$ gegeben. Wenn, wie im Fall 3, die Funktion $f(n)$ die größere Funktion ist, dann ist die Lösung durch $T(n) = \Theta(f(n))$ gegeben. Wenn, wie im Fall 2, die beiden Funktionen gleich schnell wachsen, dann ist die Lösung durch $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(f(n) \lg n)$ gegeben.

Jenseits der Intuition müssen Sie sich einiger Formalien bewusst sein. Im ersten Fall muss $f(n)$ nicht nur kleiner als $n^{\log_b a}$ sein, sondern sogar *polynomial* kleiner, d. h. $f(n)$ muss für eine Konstante $\epsilon > 0$ um den Faktor n^ϵ asymptotisch kleiner als $n^{\log_b a}$ sein. Im dritten Fall muss $f(n)$ nicht nur größer als $n^{\log_b a}$ sein, sondern polynomial größer und zusätzlich die „Regularitätsbedingung“ $a\,f(n/b) \leq c\,f(n)$ erfüllen. Diese Bedingung wird von den meisten polynomial beschränkten Funktionen, denen wir begegnen werden, erfüllt.

Bemerken Sie bitte, dass die drei Fälle nicht alle Möglichkeiten für $f(n)$ abdecken. Es gibt eine Lücke zwischen Fall 1 und Fall 2, wenn $f(n)$ kleiner als $n^{\log_b a}$ ist, aber nicht polynomial kleiner. Analog dazu gibt es eine Lücke zwischen den Fällen 2 und 3, wenn $f(n)$ größer als $n^{\log_b a}$ ist, aber nicht polynomial größer. Wenn die Funktion $f(n)$ in eine

dieser Lücken fällt oder die Regularitätsbedingung im Fall 3 nicht erfüllt ist, dann können Sie die Mastermethode nicht anwenden, um die entsprechende Rekursionsgleichung zu lösen.

Die Mastermethode anwenden

Um die Mastermethode anzuwenden, bestimmen wir einfach, welcher Fall des Mastertheorems, wenn überhaupt einer, vorliegt und schreiben die Antwort auf.

Als erstes Beispiel betrachten Sie

$$T(n) = 9\,T(n/3) + n\ .$$

Für diese Rekursionsgleichung gilt $a = 9$, $b = 3$, $f(n) = n$ und somit $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Da $f(n) = O(n^{\log_3 9-\epsilon})$ mit $\epsilon = 1$ gilt, können wir Fall 1 des Mastertheorems anwenden und schlussfolgern, dass $T(n) = \Theta(n^2)$ gilt.

Betrachten Sie nun die Rekursionsgleichung

$$T(n) = T(2n/3) + 1,$$

in der $a = 1$, $b = 3/2$, $f(n) = 1$ und $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$ gelten. Es kommt Fall 2 zur Anwendung, da $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$ ist. Somit ist die Lösung der Rekursionsgleichung $T(n) = \Theta(\lg n)$.

Für die Rekursionsgleichung

$$T(n) = 3\,T(n/4) + n \lg n$$

gilt $a = 3$, $b = 4$, $f(n) = n \lg n$ und $n^{\log_b a} = n^{\log_4 3} = O(n^{0{,}793})$. Da $f(n) = \Omega(n^{\log_4 3+\epsilon})$ mit $\epsilon \approx 0{,}2$ gilt, kommt Fall 3 zur Anwendung, wenn wir zeigen können, dass die Regularitätsbedingung erfüllt ist. Für hinreichend große n ist $a\,f(n/b) = 3\,(n/4)\lg(n/4) \leq (3/4)\,n \lg n = c\,f(n)$ für $c = 3/4$. Folglich ist nach Fall 3 die Lösung der obigen Rekursionsgleichung $T(n) = \Theta(n \lg n)$.

Die Mastermethode ist auf die Rekursionsgleichung

$$T(n) = 2\,T(n/2) + n \lg n$$

nicht anwendbar, obwohl es so aussieht, als ob sie die korrekte Form hätte: $a = 2$, $b = 2$, $f(n) = n \lg n$ und $n^{\log_b a} = n$. Möglicherweise denken Sie irrtümlicherweise, dass Fall 3 angewendet werden könnte, da $f(n) = n \lg n$ asymptotisch größer als $n^{\log_b a} = n$ ist. Das Problem besteht darin, das $f(n)$ nicht *polynomial* größer ist. Das Verhältnis $f(n)/n^{\log_b a} = (n \lg n)/n = \lg n$ ist asymptotisch kleiner als n^ϵ für jede positive Konstante ϵ. Folglich fällt die Rekursionsgleichung in die Lücke zwischen Fall 2 und Fall 3. (Siehe Übung 4.6-2 für eine Lösung.)

Lassen Sie uns die Mastermethode anwenden, um die Rekursionsgleichungen zu lösen, die wir in den Abschnitten 4.1 und 4.2 begegnet haben. Rekursionsgleichung (4.7)

$$T(n) = 2\,T(n/2) + \Theta(n)\ ,$$

beschreibt die Laufzeiten der Teile-und-Beherrsche-Algorithmen für das Max-Teilfeld-Problem und für Sortieren durch Mischen. (Gemäß unserer Vereinbarung, geben wir die Basisfälle der Rekursionsgleichungen nicht an.) Bei dieser Rekursionsgleichung gilt $a = 2$, $b = 2$, $f(n) = \Theta(n)$, und somit gilt $n^{\log_b a} = n^{\log_2 2} = n$. Fall 2 ist anwendbar, da $f(n) = \Theta(n)$, and wir erhalten als Lösung $T(n) = \Theta(n \lg n)$.

Rekursionsgleichung (4.17)

$$T(n) = 8\,T(n/2) + \Theta(n^2) \ ,$$

beschreibt die Laufzeit des ersten Teile-und-Beherrsche-Algorithmus, den wir uns für Matrizenmultiplikation überlegt haben. Hier gilt $a = 8$, $b = 2$ und $f(n) = \Theta(n^2)$ und somit $n^{\log_b a} = n^{\log_2 8} = n^3$. Da n^3 polynomial größer als $f(n)$ ist (d. h. $f(n) = O(n^{3-\epsilon})$ für $\epsilon = 1$), kommt Fall 1 zum Tragen und es gilt $T(n) = \Theta(n^3)$.

Betrachten Sie schlussendlich Rekursionsgleichung (4.18)

$$T(n) = 7\,T(n/2) + \Theta(n^2) \ ,$$

die die Laufzeit von Strassens Algorithmus angibt. Hier haben wir $a = 7$, $b = 2$, $f(n) = \Theta(n^2)$, und somit $n^{\log_b a} = n^{\log_2 7}$. Schreiben wir $\log_2 7$ als $\lg 7$ und erinnern wir uns, dass $2{,}80 < \lg 7 < 2{,}81$ gilt, so sehen wir, dass $f(n) = O(n^{\lg 7-\epsilon})$ für $\epsilon = 0{,}8$ gilt. Wieder kommt Fall 1 zum Tragen und wir erhalten die Lösung $T(n) = \Theta(n^{\lg 7})$.

Übungen

4.5-1 Wenden Sie die Mastermethode an, um asymptotisch scharfe Schranken für die folgenden Rekursionsgleichungen zu bestimmen:

(a) $T(n) = 2\,T(n/4) + 1$.

(b) $T(n) = 2\,T(n/4) + \sqrt{n}$.

(c) $T(n) = 2\,T(n/4) + n$.

(d) $T(n) = 2\,T(n/4) + n^2$.

4.5-2 Professor Caesar will einen Algorithmus für Matrizenmultiplikation entwickeln, der asymptotisch schneller als Strassens Algorithmus ist. Sein Algorithmus verwendet die Teile-und-Beherrsche-Methode und teilt jede Matrix in Teilmatrizen der Größe $n/4 \times n/4$, wobei der Teilungsschritt und der Vereinigungsschritt zusammen $\Theta(n^2)$ Zeit benötigen. Professor Caesar muss bestimmen, wie viele Teilprobleme sein Algorithmus höchstens generieren darf, um Strassens Algorithmus schlagen zu können. Wenn sein Algorithmus a Teilprobleme generiert, ist die Rekursionsgleichung für die Laufzeit $T(n)$ seines Algorithmus durch $T(n) = a\,T(n/4) + \Theta(n^2)$ gegeben. Wie groß darf die ganze Zahl a höchstens sein, damit der Algorithmus von Professor Caesar asymptotisch schneller als Strassens Algorithmus ist?

4.5-3 Zeigen Sie mit der Mastermethode, dass die Lösung der Rekursionsgleichung $T(n) = T(n/2) + \Theta(1)$ für die Laufzeit der binären Suche in $\Theta(\lg n)$ liegt. (Siehe Übung 2.3-5 für eine Beschreibung der binären Suche.)

4.5-4 Kann die Mastermethode auf die Rekursionsgleichung $T(n) = 4\,T(n/2) + n^2 \lg n$ angewendet werden? Warum oder warum nicht? Geben Sie eine asymptotisch obere Schranke für diese Rekursionsgleichung an.

4.5-5* Betrachten Sie die Regularitätsbedingung $a\,f(n/b) \leq c\,f(n)$ für eine Konstante $c < 1$, die eine Voraussetzung zu Fall 3 des Mastertheorems ist. Geben Sie ein Beispiel für die Konstanten $a \geq 1$ und $b > 1$ und die Funktion $f(n)$ an, die bis auf die Regularitätsbedingung alle Voraussetzungen zu Fall 3 des Mastertheorems erfüllen.

∗ 4.6 Beweis des Mastertheorems

Dieser Abschnitt enthält den Beweis des Mastertheorems (Theorem 4.1). Sie müssen den Beweis nicht verstehen, um das Theorem anwenden zu können.

Der Beweis besteht aus zwei Teilen. Der erste Teil analysiert die Masterrekursionsgleichung (4.20) unter der vereinfachenden Annahme, dass $T(n)$ nur für Potenzen von $b > 1$ definiert ist, d. h. für $n = 1, b, b^2, \ldots$. Dieser Teil gibt die Intuition, die notwendig ist, um zu verstehen, warum das Mastertheorem korrekt ist. Der zweite Teil zeigt, wie die Analyse auf alle positiven ganzen Zahlen n ausgedehnt werden kann; er wendet mathematische Techniken zum Auf- und Abrunden reeller Zahlen auf das Problem an.

In diesem Abschnitt werden wir unsere asymptotische Notation manchmal geringfügig missbräuchlich verwenden, indem wir sie verwenden, um das Verhalten von Funktionen zu beschreiben, die nur über Potenzen von b definiert sind. Erinnern Sie sich daran, dass die Definitionen der asymptotischen Notationen fordern, dass die Schranken für alle hinreichend großen Zahlen bewiesen werden, nicht nur für diejenigen, die Potenzen von b sind. Da wir neue asymptotische Notationen einführen könnten, die nur auf die Menge $\{b^i : i = 0, 1, \ldots\}$ anstatt auf die nichtnegativen Zahlen anwendbar sind, ist dieser Missbrauch geringfügig.

Trotzdem müssen wir immer vorsichtig sein, wenn wir die asymptotische Notation über einem eingeschränkten Bereich verwenden, damit wir keine falschen Schlüsse ziehen. Der Beweis, dass $T(n) = O(n)$ gilt, wenn n eine Potenz von 2 ist, garantiert zum Beispiel nicht, dass $T(n) = O(n)$ für alle n gilt. Die Funktion $T(n)$ könnten wir als

$$T(n) = \begin{cases} n & \text{falls } n = 1, 2, 4, 8, \ldots, \\ n^2 & \text{sonst} \end{cases}$$

definieren. In diesem Falle ist die beste zu beweisende obere Schranke, die für alle Werte von n gilt, $T(n) = O(n^2)$. Aufgrund dieser dramatischen Konsequenz sollten wir niemals die asymptotische Notation über einem eingeschränkten Bereich benutzen, ohne im Kontext hervorzuheben, dass wir dies tun.

4.6.1 Der Beweis für exakte Potenzen

Der erste Teil des Beweises des Mastertheorems analysiert die Rekursionsgleichung (4.20)

$$T(n) = a\,T(n/b) + f(n)$$

unter der Annahme, dass n eine Potenz von $b > 1$ ist, wobei b nicht ganzzahlig sein muss. Wir unterteilen die Analyse in drei Lemmata. Das erste Lemma reduziert das Problem, eine Lösung der Masterrekursionsgleichung zu finden, auf das Problem, einen Ausdruck auszuwerten, der eine Summenformel enthält. Das zweite Lemma bestimmt Schranken für die Summenformel. Das dritte Lemma führt die ersten beiden Ergebnisse zusammen, um eine Version des Mastertheorems zu beweisen, in der n eine exakte Potenz von b ist.

Lemma 4.2

Seien $a \geq 1$ und $b > 1$ Konstanten und sei $f(n)$ eine nichtnegative Funktion, die über den Potenzen von b definiert ist. Definieren wir $T(n)$ über die Potenzen von b durch die Rekursionsgleichung

$$T(n) = \begin{cases} \Theta(1) & \text{falls } n = 1 \ , \\ a\,T(n/b) + f(n) & \text{falls } n = b^i \ , \end{cases}$$

wobei i eine positive ganze Zahl ist, dann gilt

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) \ . \tag{4.21}$$

Beweis: Wir benutzen den Rekursionsbaum aus Abbildung 4.7. Die Wurzel des Baumes hat die Kosten $f(n)$, besitzt a Kinder, von denen jedes die Kosten $f(n/b)$ verursacht. (Es ist zweckmäßig, wenn auch von der Mathematik her nicht notwendig, a als eine ganze Zahl anzunehmen, wenn man den Rekursionsbaum grafisch darstellen will.) Jedes dieser Kinder hat selbst wieder a Kinder, und somit gibt es insgesamt a^2 Knoten der Tiefe 2, wobei jedes dieser Kinder Kosten $f(n/b^2)$ hat. Im Allgemeinen gibt es a^j Knoten der Tiefe j und jeder dieser Knoten hat Kosten $f(n/b^j)$. Die Kosten jedes Blattes betragen $T(1) = \Theta(1)$, wobei jedes Blatt wegen $n/b^{\log_b n} = 1$ die Tiefe $\log_b n$ besitzt. Der Baum verfügt somit über $a^{\log_b n} = n^{\log_b a}$ Blätter.

Wir können Gleichung (4.21) erhalten, indem wir die Kosten der Knoten einer jeden Ebene im Baum aufsummieren, wie dies in der Abbildung gezeigt wird. Die Kosten aller Knoten der Tiefe j betragen $a^j f(n/b^j)$, sodass sich die Gesamtkosten über alle Knoten durch

$$\sum_{j=0}^{\log_b n - 1} a^j f(n/b^j)$$

ergeben. Im zugrunde liegenden Teile-und-Beherrsche-Algorithmus repräsentiert diese Summe die Kosten dafür, das Problem in Teilprobleme zu teilen und am Ende die Lösungen der Teilprobleme zu vereinigen. Die Kosten aller Blätter, also die Kosten dafür, alle $n^{\log_b a}$ Teilprobleme der Größe 1 zu lösen, sind in $\Theta(n^{\log_b a})$. ■

Bezogen auf den Rekursionsbaum entsprechen die drei, im Mastertheorem unterschiedenen Fälle, den folgenden Fällen: Im Fall (1) werden die Gesamtkosten des Baumes

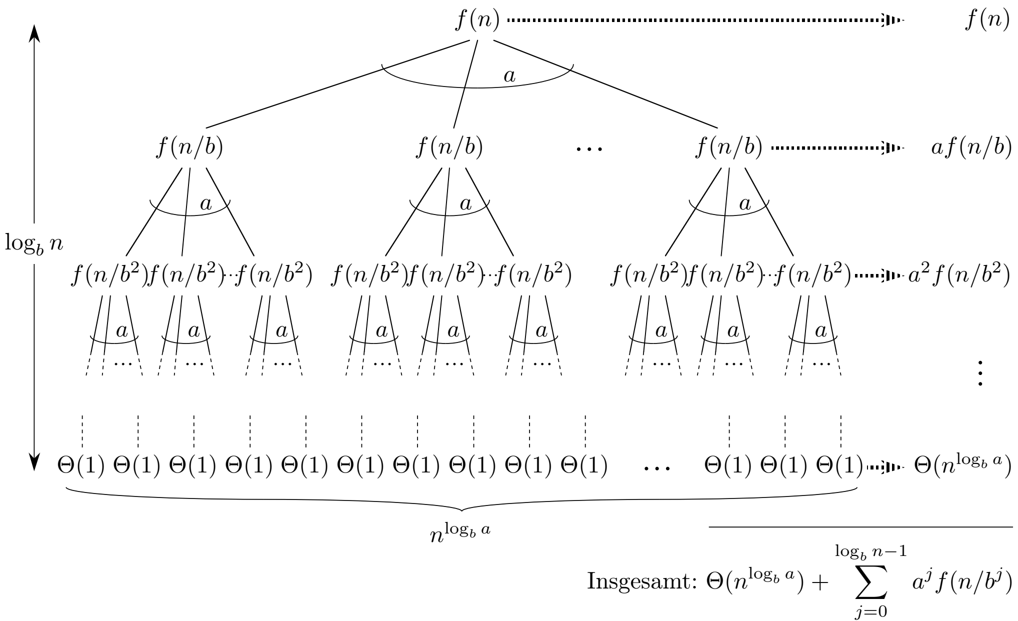


Abbildung 4.7: *Der durch $T(n) = a\,T(n/b) + f(n)$ generierte Rekursionsbaum. Der Baum ist ein vollständiger a-närer Baum mit $n^{\log_b a}$ Blättern und Höhe $\log_b n$. Die Kosten der Knoten auf jeder Ebene stehen rechts und deren Summe ist durch Gleichung (4.21) gegeben.*

von den Kosten der Blätter dominiert, im Fall (2) sind sie gleichmäßig über den Ebenen des Baumes verteilt und im Fall (3) werden Sie von den Kosten der Wurzel dominiert.

Die Summenformel in Gleichung (4.21) beschreibt die Kosten des Teilens und Zusammensetzens im zugrunde liegenden Teile-und-Beherrsche-Algorithmus. Das nächste Lemma liefert asymptotische Schranken für das Wachstum der Summenformel.

Lemma 4.3

Seien $a \geq 1$ und $b > 1$ Konstanten und sei $f(n)$ eine nichtnegative Funktion, die über den Potenzen von b definiert ist. Eine über den Potenzen von b durch

$$g(n) = \sum_{j=0}^{\log_b n-1} a^j f(n/b^j) \tag{4.22}$$

definierte Funktion $g(n)$ besitzt die folgenden asymptotischen Schranken für Potenzen von b:

1. Gilt $f(n) = O(n^{\log_b a-\epsilon})$ für eine Konstante $\epsilon > 0$, dann gilt $g(n) = O(n^{\log_b a})$.
2. Gilt $f(n) = \Theta(n^{\log_b a})$, dann gilt $g(n) = \Theta(n^{\log_b a} \lg n)$.

3. Gilt $a\,f(n/b) \leq c\,f(n)$ für eine Konstante $c < 1$ und für alle hinreichend großen n, dann gilt $g(n) = \Theta(f(n))$.

Beweis: Im Fall 1 gilt $f(n) = O(n^{\log_b a - \epsilon})$, was $f(n/b^j) = O((n/b^j)^{\log_b a - \epsilon})$ impliziert. Das Einsetzen in Gleichung (4.22) führt zu

$$g(n) = O\left(\sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j}\right)^{\log_b a - \epsilon}\right) . \tag{4.23}$$

Wir beschränken die Summenformel innerhalb der O-Notation, indem wir Terme ausklammern und vereinfachen, sodass wir zu einer steigenden geometrischen Reihe kommen:

$$\begin{aligned}
\sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j}\right)^{\log_b a - \epsilon} &= n^{\log_b a - \epsilon} \sum_{j=0}^{\log_b n - 1} \left(\frac{ab^\epsilon}{b^{\log_b a}}\right)^j \\
&= n^{\log_b a - \epsilon} \sum_{j=0}^{\log_b n - 1} (b^\epsilon)^j \\
&= n^{\log_b a - \epsilon} \left(\frac{b^{\epsilon \log_b n} - 1}{b^\epsilon - 1}\right) \\
&= n^{\log_b a - \epsilon} \left(\frac{n^\epsilon - 1}{b^\epsilon - 1}\right) .
\end{aligned}$$

Da b und ϵ Konstanten sind, können wir den letzten Term zu $n^{\log_b a - \epsilon} O(n^\epsilon) = O(n^{\log_b a})$ umformen. Setzen wir diesen Ausdruck an Stelle der Summenformel in Gleichung (4.23) ein, führt dies zu

$$g(n) = O(n^{\log_b a}) .$$

Damit ist Fall 1 bewiesen.

Da Fall 2 annimmt, dass $f(n) = \Theta(n^{\log_b a})$ gilt, erhalten wir in diesem Fall $f(n/b^j) = \Theta((n/b^j)^{\log_b a})$. Das Einsetzen in Gleichung (4.22) führt zu

$$g(n) = \Theta\left(\sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j}\right)^{\log_b a}\right) . \tag{4.24}$$

Wir beschränken die Summenformel innerhalb der Θ-Notation wie im Fall 1. Wir erhalten aber diesmal keine geometrische Reihe. Stattdessen stellen wir fest, dass alle Terme

der Summenformel gleich sind:

$$\begin{aligned}\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a} &= n^{\log_b a} \sum_{j=0}^{\log_b n-1} \left(\frac{a}{b^{\log_b a}}\right)^j \\ &= n^{\log_b a} \sum_{j=0}^{\log_b n-1} 1 \\ &= n^{\log_b a} \log_b n \ .\end{aligned}$$

Setzen wir diesen Ausdruck für die Summenformel in Gleichung (4.24) ein, führt dies zu

$$\begin{aligned}g(n) &= \Theta(n^{\log_b a} \log_b n) \\ &= \Theta(n^{\log_b a} \lg n) \ ,\end{aligned}$$

womit Fall 2 bewiesen ist.

Wir beweisen Fall 3 ganz ähnlich. Da $f(n)$ in der Definition (4.22) von $g(n)$ vorkommt und alle Terme von $g(n)$ nichtnegativ sind, können wir schlussfolgern, dass $g(n) = \Omega(f(n))$ für die Potenzen von b gilt. Wir nehmen in der Aussage des Lemmas an, dass die Ungleichung $a\,f(n/b) \leq c\,f(n)$ für eine Konstante $c < 1$ und hinreichend große n gilt. Wir schreiben die Ungleichung um zu $f(n/b) \leq (c/a)f(n)$. Iterieren wir j-mal, erhalten wir $f(n/b^j) \leq (c/a)^j f(n)$ oder äquivalent dazu $a^j f(n/b^j) \leq c^j f(n)$. Hierbei nehmen wir an, dass die Werte, die wir iterieren, hinreichend groß sind. Diese Ungleichung wird von höchstens konstant vielen Termen verletzt, nämlich möglicherweise von den Termen, bei denen n/b^{j-1} nicht hinreichend groß ist. Für diese gilt jedoch $a^j f(n/b^j) = O(1)$.

Das Einsetzen in Gleichung (4.22) und Vereinfachen führt zu einer geometrischen Reihe; aber im Gegensatz zum Fall 1 hat diese fallende Terme. Wir benutzen einen $O(1)$-Term, um all die Terme zu erfassen, die nicht durch unsere Prämisse „n hinreichend groß“ abgedeckt werden. Es gilt

$$\begin{aligned}g(n) &= \sum_{j=0}^{\log_b n-1} a^j f(n/b^j) \\ &\leq \sum_{j=0}^{\log_b n-1} c^j f(n) + O(1) \\ &\leq f(n) \sum_{j=0}^{\infty} c^j + O(1) \\ &= f(n) \left(\frac{1}{1-c}\right) + O(1) \\ &= O(f(n)) \ ,\end{aligned}$$

da c eine Konstante ist. Somit folgt, dass $g(n) = \Theta(f(n))$ für die Potenzen von b gilt. Damit ist Fall 3 bewiesen und der Beweis des Lemmas abgeschlossen. ■

Wir können nun eine Version des Mastertheorems für den Fall beweisen, in dem n eine Potenz von b ist.

Lemma 4.4

Seien $a \geq 1$ und $b > 1$ Konstanten und sei $f(n)$ eine über den Potenzen von b definierte nichtnegative Funktion. Sei $T(n)$ über den Potenzen von b durch die Rekursionsgleichung

$$T(n) = \begin{cases} \Theta(1) & \text{falls } n = 1 \ , \\ a\,T(n/b) + f(n) & \text{falls } n = b^i \ , \end{cases}$$

definiert, wobei i eine positive ganze Zahl ist. Dann besitzt $T(n)$ die folgenden asymptotischen Schranken für Potenzen von b:

1. Gilt $f(n) = O(n^{\log_b a - \epsilon})$ für eine Konstante $\epsilon > 0$, dann gilt $T(n) = \Theta(n^{\log_b a})$.
2. Gilt $f(n) = \Theta(n^{\log_b a})$, dann gilt $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. Gilt $f(n) = \Omega(n^{\log_b a + \epsilon})$ für eine Konstante $\epsilon > 0$ und $a\,f(n/b) \leq c\,f(n)$ für eine Konstante $c < 1$ und alle hinreichend großen n, dann gilt $T(n) = \Theta(f(n))$.

Beweis: Wir wenden die Schranken aus Lemma 4.3 an, um die Summenformel (4.21) in Lemma 4.2 auszuwerten. Im Fall 1 gilt

$$\begin{aligned} T(n) &= \Theta(n^{\log_b a}) + O(n^{\log_b a}) \\ &= \Theta(n^{\log_b a}) \end{aligned}$$

und im Fall 2

$$\begin{aligned} T(n) &= \Theta(n^{\log_b a}) + \Theta(n^{\log_b a} \lg n) \\ &= \Theta(n^{\log_b a} \lg n) \ . \end{aligned}$$

Im Fall 3 ergibt sich

$$\begin{aligned} T(n) &= \Theta(n^{\log_b a}) + \Theta(f(n)) \\ &= \Theta(f(n)) \ , \end{aligned}$$

da $f(n) = \Omega(n^{\log_b a + \epsilon})$ gilt. ■

4.6.2 Aufrunden und Abrunden

Um den Beweis des Mastertheorems zu vervollständigen, müssen wir unsere Analyse auf den Fall ausdehnen, in dem Aufrunden und Abrunden in der Masterrekursionsgleichung

verwendet werden. Damit ist die Rekursionsgleichung für alle ganzen Zahlen definiert, nicht nur für die Potenzen von b. Eine untere Schranke für

$$T(n) = a\,T(\lceil n/b \rceil) + f(n) \tag{4.25}$$

und eine obere Schranke für

$$T(n) = a\,T(\lfloor n/b \rfloor) + f(n) \tag{4.26}$$

zu erhalten, ist eine einfache Übung, da wir im ersten Fall die Ungleichung $\lceil n/b \rceil \geq n/b$ anwenden können, um das gewünschte Resultat zu erhalten, und im zweiten Fall die Ungleichung $\lfloor n/b \rfloor \leq n/b$. Wir wenden im Wesentlichen die gleiche Technik an, um die Rekursionsgleichung (4.26) nach unten zu beschränken, wie die, um die Rekursionsgleichung (4.25) nach oben zu beschränken, sodass wir nur die letztere der beiden Schranken hier beweisen werden.

Wir modifizieren den Rekursionsbaum in Abbildung 4.7, um den Rekursionsbaum in Abbildung 4.8 zu erhalten. Wenn wir im Rekursionsbaum nach unten absteigen, dann erhalten wir eine Folge von zu den rekursiven Aufrufen gehörigen Argumenten

$$\begin{array}{r} n\ , \\ \lceil n/b \rceil\ , \\ \lceil \lceil n/b \rceil /b \rceil\ , \\ \lceil \lceil \lceil n/b \rceil /b \rceil /b \rceil\ , \\ \vdots \end{array}$$

Lassen Sie uns das j-te Argument der Folge mit n_j bezeichnen, wobei

$$n_j = \begin{cases} n & \text{falls } j = 0\ , \\ \lceil n_{j-1}/b \rceil & \text{falls } j > 0 \end{cases} \tag{4.27}$$

gilt.

Unser erstes Ziel ist es, die Tiefe k zu bestimmen, sodass n_k eine Konstante ist. Unter Verwendung der Ungleichung $\lceil x \rceil \leq x + 1$ erhalten wir

$$\begin{aligned} n_0 &\leq n\ , \\ n_1 &\leq \frac{n}{b} + 1\ , \\ n_2 &\leq \frac{n}{b^2} + \frac{1}{b} + 1\ , \\ n_3 &\leq \frac{n}{b^3} + \frac{1}{b^2} + \frac{1}{b} + 1\ , \\ &\vdots \end{aligned}$$

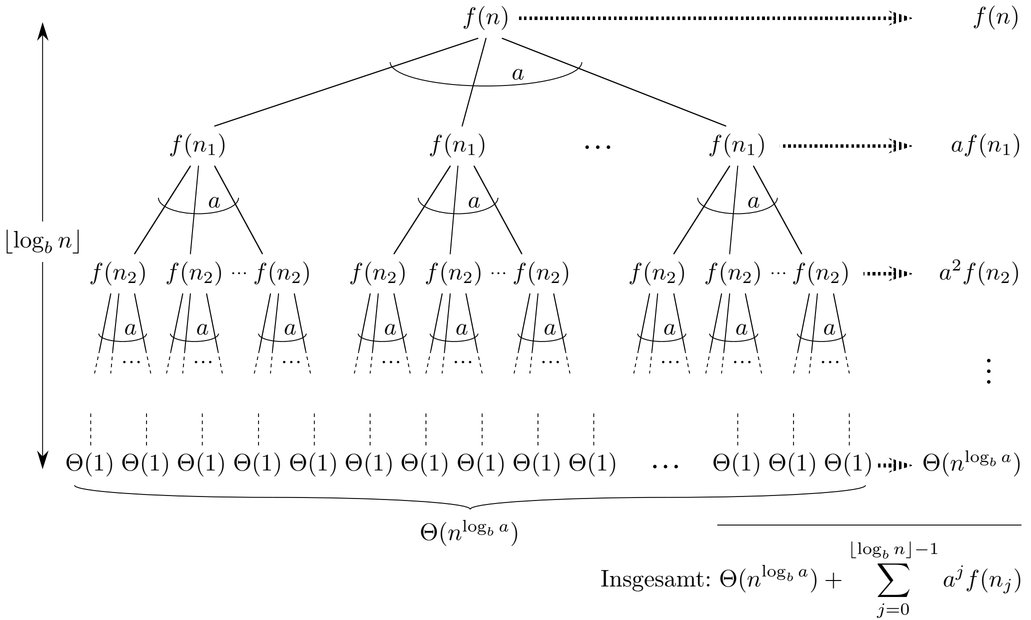


Abbildung 4.8: *Der durch die Rekursionsgleichung $T(n) = a\,T(\lceil n/b \rceil) + f(n)$ generierte Rekursionsbaum. Das rekursive Argument n_j ist durch Gleichung (4.27) gegeben.*

Im Allgemeinen gilt

$$\begin{aligned} n_j &\le \frac{n}{b^j} + \sum_{i=0}^{j-1} \frac{1}{b^i} \\ &< \frac{n}{b^j} + \sum_{i=0}^{\infty} \frac{1}{b^i} \\ &= \frac{n}{b^j} + \frac{b}{b-1} \; . \end{aligned}$$

Für $j = \lfloor \log_b n \rfloor$ erhalten wir

$$\begin{aligned} n_{\lfloor \log_b n \rfloor} &< \frac{n}{b^{\lfloor \log_b n \rfloor}} + \frac{b}{b-1} \\ &< \frac{n}{b^{\log_b n - 1}} + \frac{b}{b-1} \\ &= \frac{n}{n/b} + \frac{b}{b-1} \\ &= b + \frac{b}{b-1} \\ &= O(1) \; , \end{aligned}$$

und sehen somit, dass in der Tiefe $\lfloor \log_b n \rfloor$ die Problemgröße höchstens eine Konstante ist.

Aus Abbildung 4.8 entnehmen wir

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\lfloor \log_b n \rfloor - 1} a^j f(n_j) \ , \tag{4.28}$$

was im Wesentlichen der Gleichung (4.21) entspricht, außer dass n eine beliebige ganze Zahl sein kann und nicht auf eine Potenz von b beschränkt ist.

Wir können nun die Summenformel

$$g(n) = \sum_{j=0}^{\lfloor \log_b n \rfloor - 1} a^j f(n_j) \tag{4.29}$$

aus Gleichung (4.28) in einer zum Beweis von Lemma 4.3 analogen Weise auswerten. Wir beginnen mit Fall 3. Wenn $a\, f(\lceil n/b \rceil) \leq c\, f(n)$ für $n > b + b/(b-1)$ und eine Konstante $c < 1$ erfüllt ist, dann folgt daraus $a^j f(n_j) \leq c^j f(n)$. Deshalb können wir die Summe in Gleichung (4.29) wie in Lemma 4.3 berechnen. Im Fall 2 gilt $f(n) = \Theta(n^{\log_b a})$. Wenn wir zeigen können, dass $f(n_j) = O(n^{\log_b a}/a^j) = O((n/b^j)^{\log_b a})$ gilt, dann könnten wir den Beweis für Fall 2 aus Lemma 4.3 übernehmen. Beachten Sie, dass $j \leq \lfloor \log_b n \rfloor$ die Ungleichung $b^j/n \leq 1$ impliziert. Die Schranke $f(n) = O(n^{\log_b a})$ bedeutet, dass eine Konstante $c > 0$ existiert, sodass für alle hinreichend großen n_j gilt

$$\begin{aligned}
f(n_j) &\leq c \left(\frac{n}{b^j} + \frac{b}{b-1} \right)^{\log_b a} \\
&= c \left(\frac{n}{b^j} \left(1 + \frac{b^j}{n} \cdot \frac{b}{b-1} \right) \right)^{\log_b a} \\
&= c \left(\frac{n^{\log_b a}}{a^j} \right) \left(1 + \left(\frac{b^j}{n} \cdot \frac{b}{b-1} \right) \right)^{\log_b a} \\
&\leq c \left(\frac{n^{\log_b a}}{a^j} \right) \left(1 + \frac{b}{b-1} \right)^{\log_b a} \\
&= O\left(\frac{n^{\log_b a}}{a^j} \right) \ ,
\end{aligned}$$

da $c\,(1 + b/(b-1))^{\log_b a}$ eine Konstante ist. Damit haben wir Fall 2 bewiesen. Der Beweis von Fall 1 ist fast identisch. Die Idee besteht darin zu beweisen, dass die Schranke $f(n_j) = O((n/b^j)^{\log_b a - \epsilon})$ gilt. Dies geht analog zu dem entsprechenden Beweis von Fall 2, wenngleich die Rechnung aufwendiger ist.

Wir haben nun für alle ganzzahligen Werte n die oberen Schranken aus dem Mastertheorem bewiesen. Der Beweis der unteren Schranken läuft ähnlich ab.

Übungen

4.6-1* Geben Sie einen einfachen und exakten Ausdruck für n_j in Gleichung (4.27) für den Fall an, dass b statt einer beliebigen reellen Zahl eine positive ganze Zahl ist.

4.6-2* Zeigen Sie, dass die Masterrekursionsgleichung $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$ als Lösung hat, wenn $f(n) = \Theta(n^{\log_b a} \lg^k n)$ mit $k \geq 0$ gilt. Beschränken Sie Ihre Analyse der Einfachheit halber auf Potenzen von b.

4.6-3* Zeigen Sie, dass Fall 3 des Mastertheorems in dem Sinne zu streng formuliert ist, dass die Regularitätsbedingung $a\,f(n/b) \leq c\,f(n)$ für eine Konstante $c < 1$ impliziert, dass eine Konstante $\epsilon > 0$ existiert, sodass $f(n) = \Omega(n^{\log_b a + \epsilon})$ gilt.

Problemstellungen

4-1 Beispiele für Rekursionsgleichungen

Geben Sie für jede der folgenden Rekursionsgleichungen eine asymptotisch obere und untere Schranke für $T(n)$ an. Nehmen Sie an, dass $T(n)$ für $n \leq 2$ konstant ist. Geben Sie möglichst scharfe Schranken an und begründen Sie Ihre Antworten.

a. $T(n) = 2\,T(n/2) + n^4$.

b. $T(n) = T(7n/10) + n$.

c. $T(n) = 16\,T(n/4) + n^2$.

d. $T(n) = 7\,T(n/3) + n^2$.

e. $T(n) = 7\,T(n/2) + n^2$.

f. $T(n) = 2\,T(n/4) + \sqrt{n}$.

g. $T(n) = T(n-2) + n^2$.

4-2 Kosten der Parameterübergabe

Wir nehmen im Buch immer an, dass die Parameterübergabe beim Aufruf einer Prozedur konstante Zeit benötigt, auch wenn ein N-elementiges Feld übergeben wird. Diese Annahme ist in den meisten Systemen zutreffend, da ein Zeiger auf das Feld übergeben wird, nicht das Feld selbst. Die vorliegende Problemstellung untersucht die Folgen von drei Strategien zur Parameterübergabe:

1. Ein Feld wird durch einen Zeiger übergeben. Die hiefür benötigte Zeit liegt in $\Theta(1)$.
2. Ein Feld wird durch Kopieren übergeben. Dies erfolgt in Zeit $\Theta(N)$, wobei N die Größe des Feldes ist.
3. Ein Feld wird übergeben, indem nur der Bereich kopiert wird, auf den die aufgerufene Prozedur (möglicherweise) zugreift. Dies erfolgt in Zeit $\Theta(q - p + 1)$, wenn das Teilfeld $A[p\,..\,q]$ übergeben wird.

a. Betrachten Sie den Algorithmus der rekursiven binären Suche zum Finden einer Zahl in einem sortierten Feld (siehe Übung 2.3-5). Geben Sie für jeden der drei Methoden zur Parameterübergabe Rekursionsgleichungen für die Laufzeit der binären Suche im schlechtesten Fall an und geben Sie gute obere Schranken für die Lösungen der Rekursionsgleichungen an. N sei die Größe des ursprünglichen Problems und n die Größe des Teilproblems.

b. Lösen Sie Teil (a) auch für MERGE-SORT aus Abschnitt 2.3.1.

4-3 Weitere Beispiele für Rekursionsgleichungen

Geben Sie asymptotisch obere und untere Schranken für $T(n)$ in jeder der folgenden Rekursionsgleichungen an. Gehen Sie davon aus, dass $T(n)$ für hinreichend kleine n konstant ist. Geben Sie möglichst scharfe Schranken an und begründen Sie Ihre Antworten.

a. $T(n) = 4\,T(n/3) + n \lg n$.

b. $T(n) = 3\,T(n/3) + n/\lg n$.

c. $T(n) = 4\,T(n/2) + n^2\sqrt{n}$.

d. $T(n) = 3\,T(n/3 - 2) + n/2$.

e. $T(n) = 2\,T(n/2) + n/\lg n$.

f. $T(n) = T(n/2) + T(n/4) + T(n/8) + n$.

g. $T(n) = T(n-1) + 1/n$.

h. $T(n) = T(n-1) + \lg n$.

i. $T(n) = T(n-2) + 1/\lg n$.

j. $T(n) = \sqrt{n}\,T(\sqrt{n}) + n$.

4-4 Fibonacci-Zahlen

Diese Problemstellung erarbeitet Eigenschaften der Fibonacci-Zahlen, die durch die Rekursionsgleichung (3.22) definiert werden. Wir werden formale Potenzreihen benutzen, um die Fibonaccische Rekursionsgleichung zu lösen. Wir definieren die ***formale Potenzreihe*** $\mathcal{F}$ durch

$$\begin{aligned}\mathcal{F}(z) &= \sum_{i=0}^{\infty} F_i\, z^i \\ &= 0 + z + z^2 + 2z^3 + 3z^4 + 5z^5 + 8z^6 + 13z^7 + 21z^8 + \cdots\ ,\end{aligned}$$

wobei F_i die i-te Fibonacci-Zahl ist.

a. Zeigen Sie, dass $\mathcal{F}(z) = z + z\,\mathcal{F}(z) + z^2\mathcal{F}(z)$ gilt.

b. Zeigen Sie, dass

$$\begin{aligned}\mathcal{F}(z) &= \frac{z}{1 - z - z^2} \\ &= \frac{z}{(1 - \phi\, z)(1 - \widehat{\phi}\, z)} \\ &= \frac{1}{\sqrt{5}}\left(\frac{1}{1 - \phi\, z} - \frac{1}{1 - \widehat{\phi}\, z}\right)\end{aligned}$$

gilt, wobei

$$\phi = \frac{1+\sqrt{5}}{2} = 1{,}61803\ldots$$

und

$$\widehat{\phi} = \frac{1-\sqrt{5}}{2} = -0{,}61803\ldots$$

ist.

c. Zeigen Sie, dass

$$\mathcal{F}(z) = \sum_{i=0}^{\infty} \frac{1}{\sqrt{5}} (\phi^i - \widehat{\phi}^i)\, z^i$$

gilt.

d. Beweisen Sie mit Hilfe von Teil (c) die Gleichung $F_i = \phi^i/\sqrt{5}$, gerundet auf die nächste ganze Zahl, für $i > 0$. (*Hinweis:* Es gilt $\left|\widehat{\phi}\right| < 1$).

4-5 Chips testen

Professor Diogenes besitzt n angeblich identische integrierte Schaltungen als Chips, die sich an sich gegenseitig testen können. Die Testvorrichtung des Professors nimmt jeweils zwei Chips auf. Ist die Testvorrichtung mit zwei Chips belegt, dann testet jeder der beiden Chips den anderen und meldet, ob dieser fehlerfrei oder fehlerbehaftet ist. Ein fehlerfreier Chip gibt immer die korrekte Antwort, ob der andere Chip fehlerfrei oder fehlerbehaftet ist; aber der Professor kann der Antwort eines fehlerbehafteten Chips nicht trauen. Somit sind die folgenden vier Ausgänge eines Testes möglich:

Chip A sagt	Chip B sagt	Schlussfolgerung
B fehlerfrei	A fehlerfrei	beide fehlerfrei oder beide fehlerbehaftet
B fehlerfrei	A fehlerbehaftet	mindestens einer ist fehlerbehaftet
B fehlerbehaftet	A fehlerfrei	mindestens einer ist fehlerbehaftet
B fehlerbehaftet	A fehlerbehaftet	mindestens einer ist fehlerbehaftet

a. Zeigen Sie, dass der Professor mithilfe einer auf diesen paarweisen Tests aufbauenden Strategie nicht notwendigerweise bestimmen kann, welche Chips fehlerfrei sind, wenn mindestens $n/2$ Chips fehlerbehaftet sind. Nehmen Sie an, dass sich die fehlerbehafteten Chips verschwören können, um den Professor zu täuschen.

b. Betrachten Sie das Problem, einen einzigen fehlerfreien Chip unter n Chips zu finden, unter der Annahme, dass mehr als $n/2$ der Chips fehlerfrei sind. Zeigen Sie, dass $\lfloor n/2 \rfloor$ Tests ausreichend sind, um die Problemgröße nahezu zu halbieren.

c. Zeigen Sie, dass die fehlerfreien Chips mit $\Theta(n)$ Tests identifiziert werden können, unter der Annahme, dass mehr als $n/2$ der Chips fehlerfrei sind. Geben Sie die Rekursionsgleichung an, die die Anzahl der Tests beschreibt, und lösen Sie diese.

4-6 Monge-Felder

Ein $m \times n$ Feld A reeller Zahlen ist ein ***Monge-Feld***, wenn für alle i, j, k und l mit $1 \le i < k \le m$ und $1 \le j < l \le n$

$$A[i,j] + A[k,l] \le A[i,l] + A[k,j]$$

gilt. Mit anderen Worten, wann immer wir zwei Zeilen und zwei Spalten eines Monge-Feldes auswählen und die vier Elemente an den Überschneidungen von Zeilen und Spalten betrachten, ist die Summe aus den Elementen links oben und rechts unten kleiner oder gleich der Summe aus den Elementen links unten und rechts oben. Zum Beispiel ist das folgende Feld ein Monge-Feld:

$$\begin{array}{rrrrr}
10 & 17 & 13 & 28 & 23 \\
17 & 22 & 16 & 29 & 23 \\
24 & 28 & 22 & 34 & 24 \\
11 & 13 & 6 & 17 & 7 \\
45 & 44 & 32 & 37 & 23 \\
36 & 33 & 19 & 21 & 6 \\
75 & 66 & 51 & 53 & 34
\end{array}$$

a. Beweisen Sie, dass ein Feld genau dann ein Monge-Feld ist, wenn für alle $i = 1, 2, ..., m-1$ und $j = 1, 2, ..., n-1$

$$A[i,j] + A[i+1,j+1] \le A[i,j+1] + A[i+1,j]$$

gilt. (*Hinweis:* Benutzen Sie für den „wenn“-Teil Induktion getrennt nach Zeilen und Spalten.)

b. Das folgende Feld ist kein Monge-Feld. Verändern Sie ein Element, um es zu einem Monge-Feld zu machen. (*Hinweis:* Wenden Sie Teil (a) an.)

$$\begin{array}{rrrr}
37 & 23 & 22 & 32 \\
21 & 6 & 7 & 10 \\
53 & 34 & 30 & 31 \\
32 & 13 & 9 & 6 \\
43 & 21 & 15 & 8
\end{array}$$

c. Sei $f(i)$ der Index der Spalte, die das am weitesten links stehende minimale Element der Zeile i enthält. Beweisen Sie, dass für jedes $m \times n$ Monge-Feld die Ungleichung $f(1) \le f(2) \le \cdots \le f(m)$ gilt.

d. Hier ist die Beschreibung eines Algorithmus nach dem Teile-und-Beherrsche-Prinzip, der das am weitesten links stehende minimale Element in jeder Zeile eines $m \times n$ Monge-Feldes berechnet:

> Konstruieren Sie eine Teilmatrix A' von A, die aus den geradzahligen Zeilen von A besteht. Bestimmen Sie rekursiv das am weitesten links stehende minimale Element jeder Zeile von A'. Bestimmen Sie dann die am weitesten links stehenden minimalen Elemente der ungeradzahligen Zeilen von A.

Erklären Sie, wie Sie die am weitesten links stehenden minimalen Elemente der ungeradzahligen Zeilen von A in Zeit $O(m+n)$ berechnen können. Setzen Sie hierbei voraus, dass Sie die am weitesten links stehenden minimalen Elemente der geradzahligen Zeilen bereits kennen.

e. Geben Sie die Rekursionsgleichung für die Laufzeit des in Teil (d) beschriebenen Algorithmus an. Zeigen Sie, dass die Lösung in $O(m + n \log m)$ liegt.

Kapitelbemerkungen

Teile-und-Beherrsche ist eine Technik zum Entwurf von Algorithmen, die zumindest bereits im Jahr 1962 bekannt war und in einem Artikel von Karatsuba und Ofman [194] zu finden ist. Es kann jedoch gut sein, dass die Technik bereits schon früher angewendet worden ist; laut Heideman, Johnson und Burrus [163], hat C. F. Gauss sich in 1805 den ersten Algorithmus für Schnelle Fourier-Transformation überlegt, wobei die Beschreibung von Gauss das Problem in kleinere Probleme zerlegt, deren Lösungen miteinander kombiniert werden.

Das Max-Feld-Problem aus Abschnitt 4.1 ist eine leichte Variation eines Problems, welches durch Bentley [43, Kapitel 7] untersucht wurde.

Strassens Algorithmus [325] erregte große Aufmerksamkeit, als er in 1969 veröffentlicht wurde. Davor konnten sich nur wenige vorstellen, dass es einen Algorithmus zur Matrizenmultiplikation geben könnte, der asymptotisch schneller als die elementare SQUARE-MATRIX-MULTIPLY Prozedur ist. Die asymptotisch obere Schranke für Matrizenmultiplikation wurde seitdem verbessert. Der asymptotisch effizienteste Algorithmus, um $n \times n$-Matrizen zu multipizieren, geht auf Coppersmith und Winograd [78] zurück und hat eine Laufzeit von $O(n^{2{,}376})$. Die beste untere Schranke, die bekannt ist, ist die offensichtliche $\Omega(n^2)$ Schranke (offensichtlich, da wir n^2 Elemente in die Produktmatrix eintragen müssen).

Aus praktischer Sicht ist Strassens Algorithmus oft nicht die Methode, die wir für Matrizenmultiplikation wählen sollten. Dies hat vier Gründe:

1. Die konstanten Faktoren, die in der $\Theta(n^{\lg 7})$-Laufzeit versteckt sind, sind größer als die konstanten Faktoren, die in der $\Theta(n^3)$-Laufzeit der Prozedur SQUARE-MATRIX-MULTIPLY versteckt sind.

2. Wenn die Matrizen dünn besetzt sind, sind Methoden, die auf diesen speziellen Fall zugeschnitten sind, schneller.

3. Strassens Algorithmus ist numerisch nicht ganz so stabil als dies SQUARE-MATRIX-MULTIPLY ist. Anders formuliert, aufgrund der eingeschränkten Rechengenauigkeit der Computerarithmetik auf nichtganzen Zahlen ist der akkumulierte Fehler in Strassens Algorithmus größer als in SQUARE-MATRIX-MULTIPLY.

4. Die Teilmatrizen, die auf den einzelnen Rekursionsebenen generiert werden müssen, belegen Speicherplatz.

Die beiden letzten Gründe haben sich um 1990 entschärft. Higham [167] zeigt, dass der Unterschied in Bezug auf die numerische Stabilität überbetont wurde; wenngleich Strassens Algorithmus für einige Anwendungen numerisch zu instabil ist, arbeitet er bei anderen Anwendungen durchaus in einem tolerablen Bereich. Bailey, Lee und Simon [32] diskutieren Techniken, mit denen der Speicherplatzbedarf von Strassens Algorithmus reduziert werden kann.

In der Praxis setzen Implementierungen für die schnelle Multiplikation dichter Matrizen Strassens Algorithmus ab einer bestimmten Matrizengröße ein und wechseln zu einfacheren Methoden, wenn die Größe der Teilmatrizen unter diesen Schwellenwert fällt. Der exakte Wert dieses Schwellenwertes hängt stark von dem darunterliegenden System ab. Analysen, die die Operationen zählen, aber Effekte, die von Caches oder der Befehlspipeline herrühren, ignorieren, kommen auf Werte von $n = 8$ (siehe Higham [167]) oder $n = 12$ (siehe Huss-Lederman et al. [186]) für diesen Schwellenwert. D'Alberto und Nicolau [81] entwickelten ein adaptives Schema, das den Schwellenwert anhand von während der Installation ablaufenden Testläufen bestimmt. Sie fanden Schwellenwerte für unterschiedliche Systeme, die von $n = 400$ bis $n = 2150$ reichten.

Rekursionsgleichungen sind früh, 1202 von L. Fibonacci, nach dem die Fibonacci-Zahlen benannt sind, untersucht worden. A. de Moivre führte die Methode der formalen Potenzreihen (siehe Problemstellung 4-4) zur Lösung von Rekursionsgleichungen ein. Die Mastermethode wurde von Bentley, Haken und Saxe in [44] eingeführt, die die erweiterte Methode zur Verfügung stellt, die in Übung 4.6-2 begründet wird. Knuth [209] und Liu [237] zeigen, wie lineare Rekursionsgleichungen unter Verwendung der Methode der formalen Potenzreihen gelöst werden. Die Arbeiten von Purdom und Brown [287] und Graham, Knuth und Patashnik [152] enthalten umfangreiche Abhandlungen zur Lösung von Rekursionsgleichungen.

Viele Wissenschaftler, darunter Akra und Bazzi [13], Roura [299], Verma [346] und Yap [360], haben Methoden zur Lösung allgemeinerer Rekursionsgleichungen angegeben als die, die durch einen Teile-und-Beherrsche-Algorithmus begründet sind und mithilfe der Mastermethode gelöst werden. Wir beschreiben hier den Ansatz von Akra und Bazzi, wie von Leighton [228] überarbeitet, der für Rekursionsgleichungen der Form

$$T(x) = \begin{cases} \Theta(1) & \text{falls } 1 \le x \le x_0 \ , \\ \sum_{i=1}^{k} a_i T(b_i x) + f(x) & \text{falls } x > x_0 \ , \end{cases} \tag{4.30}$$

angewendet werden kann, mit

- $x \ge 1$ ist eine reelle Zahl,
- x_0 ist eine Konstante mit $x_0 \ge 1/b_i$ und $x_0 \ge 1/(1 - b_i)$ für $i = 1, 2, \ldots, k$,
- a_i ist eine positive Konstante für $i = 1, 2, \ldots, k$,
- b_i ist eine Konstante aus dem Bereich $0 < b_i < 1$ für $i = 1, 2, \ldots, k$,
- $k \ge 1$ ist eine ganzzahlige Konstante, und

- $f(x)$ ist eine nichtnegative Funktion, die der ***polynomialen Wachstumsbedingung*** genügt: es gibt positive Konstanten c_1 und c_2, sodass für alle $x \geq 1$, für $i = 1, 2, \ldots, k$ und für alle u mit $b_i\, x \leq u \leq x$, die Ungleichung $c_1\, f(x) \leq f(u) \leq c_2\, f(x)$ gilt. (Ist $|f'(x)|$ nach oben durch ein Polynom in x beschränkt, dann erfüllt $f(x)$ die polynomiale Wachstumsbedingung. Die Funktion $f(x) = x^\alpha \lg^\beta x$ beispielsweise erfüllt diese Bedingung für jede reelle Konstante α und β.)

Wenngleich die Mastermethode auf eine Rekursionsgleichung wie zum Beispiel $T(n) = T(\lfloor n/3 \rfloor) + T(\lfloor 2n/3 \rfloor) + O(n)$ nicht anwendbar ist, die Akra-Bazzi-Methode ist es. Um die Rekursionsgleichung (4.30) zu lösen, müssen wir zuerst die einzige reelle Zahl p finden, für die $\sum_{i=1}^{k} a_i\, b_i^p = 1$ gilt. (Eine solche Zahl p existiert immer.) Die Lösung der Rekursionsgleichung ist dann durch

$$T(n) = \Theta\left(x^p \left(1 + \int_1^x \frac{f(u)}{u^{p+1}}\, du\right)\right)$$

gegeben. Die Anwendung der Akra-Bazzi-Methode kann in gewissem Sinne schwierig sein, sie hilft aber Rekursionsgleichungen zu lösen, die die Laufzeit für den Fall beschreiben, in dem das Problem in Teilprobleme aufgeteilt werden, die von ihrer Größe her stark unterschiedlich sind. Die Mastermethode ist einfacherer, aber nur anwendbar, wenn die Größe der Teilprobleme gleich ist.