Algorithm Design

This page intentionally left blank

Algorithm Design

JON KLEINBERG • ÉVA TARDOS

Cornell University

Boston San Francisco New York
London Toronto Sydney Tokyo Singapore Madrid
Mexico City Munich Paris Cape Town Hong Kong Montreal

Acquisitions Editor: *Matt Goldstein*
Project Editor: *Maite Suarez-Rivas*
Production Supervisor: *Marilyn Lloyd*
Marketing Manager: *Michelle Brown*
Marketing Coordinator: *Jake Zavracky*
Project Management: *Windfall Software*
Composition: *Windfall Software, using ZzTEX*
Copyeditor: *Carol Leyba*
Technical Illustration: *Dartmouth Publishing*
Proofreader: *Jennifer McClain*
Indexer: *Ted Laux*
Cover Design: *Joyce Cosentino Wells*
Cover Photo: *© 2005 Tim Laman / National Geographic. A pair of weaverbirds work together on their nest in Africa.*
Prepress and Manufacturing: *Caroline Fell*
Printer: *Courier Westford*

Access the latest information about Addison-Wesley titles from our World Wide Web site: http://www.aw-bc.com/computing

Library of Congress Cataloging-in-Publication Data

Kleinberg, Jon.
Algorithm design / Jon Kleinberg, Éva Tardos.—1st ed.
p. cm.
Includes bibliographical references and index.
ISBN 0-321-29535-8 (alk. paper)
1. Computer algorithms. 2. Data structures (Computer science) I. Tardos, Éva. II. Title.

QA76.9.A43K54 2005
005.1—dc22 2005000401

ISBN 0-321-29535-8
1 2 3 4 5 6 7 8 9 10-CRW-08 07 06 05

About the Authors

Jon Kleinberg is a professor of Computer Science at Cornell University. He received his Ph.D. from M.I.T. in 1996. He is the recipient of an NSF Career Award, an ONR Young Investigator Award, an IBM Outstanding Innovation Award, the National Academy of Sciences Award for Initiatives in Research, research fellowships from the Packard and Sloan Foundations, and teaching awards from the Cornell Engineering College and Computer Science Department.

Kleinberg's research is centered around algorithms, particularly those concerned with the structure of networks and information, and with applications to information science, optimization, data mining, and computational biology. His work on network analysis using hubs and authorities helped form the foundation for the current generation of Internet search engines.

Éva Tardos is a professor of Computer Science at Cornell University. She received her Ph.D. from Eötvös University in Budapest, Hungary in 1984. She is a member of the American Academy of Arts and Sciences, and an ACM Fellow; she is the recipient of an NSF Presidential Young Investigator Award, the Fulkerson Prize, research fellowships from the Guggenheim, Packard, and Sloan Foundations, and teaching awards from the Cornell Engineering College and Computer Science Department.

Tardos's research interests are focused on the design and analysis of algorithms for problems on graphs or networks. She is most known for her work on network-flow algorithms and approximation algorithms for network problems. Her recent work focuses on algorithmic game theory, an emerging area concerned with designing systems and algorithms for selfish users.

This page intentionally left blank

Contents

* The star indicates an optional section. (See the Preface for more information about the relationships among the chapters and sections.)

This page intentionally left blank

Preface

Algorithmic ideas are pervasive, and their reach is apparent in examples both within computer science and beyond. Some of the major shifts in Internet routing standards can be viewed as debates over the deficiencies of one shortest-path algorithm and the relative advantages of another. The basic notions used by biologists to express similarities among genes and genomes have algorithmic definitions. The concerns voiced by economists over the feasibility of combinatorial auctions in practice are rooted partly in the fact that these auctions contain computationally intractable search problems as special cases. And algorithmic notions aren't just restricted to well-known and long-standing problems; one sees the reflections of these ideas on a regular basis, in novel issues arising across a wide range of areas. The scientist from Yahoo! who told us over lunch one day about their system for serving ads to users was describing a set of issues that, deep down, could be modeled as a network flow problem. So was the former student, now a management consultant working on staffing protocols for large hospitals, whom we happened to meet on a trip to New York City.

The point is not simply that algorithms have many applications. The deeper issue is that the subject of algorithms is a powerful lens through which to view the field of computer science in general. Algorithmic problems form the heart of computer science, but they rarely arrive as cleanly packaged, mathematically precise questions. Rather, they tend to come bundled together with lots of messy, application-specific detail, some of it essential, some of it extraneous. As a result, the algorithmic enterprise consists of two fundamental components: the task of getting to the mathematically clean core of a problem, and then the task of identifying the appropriate algorithm design techniques, based on the structure of the problem. These two components interact: the more comfortable one is with the full array of possible design techniques, the more one starts to recognize the clean formulations that lie within messy

problems out in the world. At their most effective, then, algorithmic ideas do not just provide solutions to well-posed problems; they form the language that lets you cleanly express the underlying questions.

The goal of our book is to convey this approach to algorithms, as a design process that begins with problems arising across the full range of computing applications, builds on an understanding of algorithm design techniques, and results in the development of efficient solutions to these problems. We seek to explore the role of algorithmic ideas in computer science generally, and relate these ideas to the range of precisely formulated problems for which we can design and analyze algorithms. In other words, what are the underlying issues that motivate these problems, and how did we choose these particular ways of formulating them? How did we recognize which design principles were appropriate in different situations?

In keeping with this, our goal is to offer advice on how to identify clean algorithmic problem formulations in complex issues from different areas of computing and, from this, how to design efficient algorithms for the resulting problems. Sophisticated algorithms are often best understood by reconstructing the sequence of ideas—including false starts and dead ends—that led from simpler initial approaches to the eventual solution. The result is a style of exposition that does not take the most direct route from problem statement to algorithm, but we feel it better reflects the way that we and our colleagues genuinely think about these questions.

Overview

The book is intended for students who have completed a programming-based two-semester introductory computer science sequence (the standard "CS1/CS2" courses) in which they have written programs that implement basic algorithms, manipulate discrete structures such as trees and graphs, and apply basic data structures such as arrays, lists, queues, and stacks. Since the interface between CS1/CS2 and a first algorithms course is not entirely standard, we begin the book with self-contained coverage of topics that at some institutions are familiar to students from CS1/CS2, but which at other institutions are included in the syllabi of the first algorithms course. This material can thus be treated either as a review or as new material; by including it, we hope the book can be used in a broader array of courses, and with more flexibility in the prerequisite knowledge that is assumed.

In keeping with the approach outlined above, we develop the basic algorithm design techniques by drawing on problems from across many areas of computer science and related fields. To mention a few representative examples here, we include fairly detailed discussions of applications from systems and networks (caching, switching, interdomain routing on the Internet), artificial

intelligence (planning, game playing, Hopfield networks), computer vision (image segmentation), data mining (change-point detection, clustering), operations research (airline scheduling), and computational biology (sequence alignment, RNA secondary structure).

The notion of computational intractability, and NP-completeness in particular, plays a large role in the book. This is consistent with how we think about the overall process of algorithm design. Some of the time, an interesting problem arising in an application area will be amenable to an efficient solution, and some of the time it will be provably NP-complete; in order to fully address a new algorithmic problem, one should be able to explore both of these options with equal familiarity. Since so many natural problems in computer science are NP-complete, the development of methods to deal with intractable problems has become a crucial issue in the study of algorithms, and our book heavily reflects this theme. The discovery that a problem is NP-complete should not be taken as the end of the story, but as an invitation to begin looking for approximation algorithms, heuristic local search techniques, or tractable special cases. We include extensive coverage of each of these three approaches.

Problems and Solved Exercises

An important feature of the book is the collection of problems. Across all chapters, the book includes over 200 problems, almost all of them developed and class-tested in homework or exams as part of our teaching of the course at Cornell. We view the problems as a crucial component of the book, and they are structured in keeping with our overall approach to the material. Most of them consist of extended verbal descriptions of a problem arising in an application area in computer science or elsewhere out in the world, and part of the problem is to practice what we discuss in the text: setting up the necessary notation and formalization, designing an algorithm, and then analyzing it and proving it correct. (We view a complete answer to one of these problems as consisting of all these components: a fully explained algorithm, an analysis of the running time, and a proof of correctness.) The ideas for these problems come in large part from discussions we have had over the years with people working in different areas, and in some cases they serve the dual purpose of recording an interesting (though manageable) application of algorithms that we haven't seen written down anywhere else.

To help with the process of working on these problems, we include in each chapter a section entitled "Solved Exercises," where we take one or more problems and describe how to go about formulating a solution. The discussion devoted to each solved exercise is therefore significantly longer than what would be needed simply to write a complete, correct solution (in other words,

significantly longer than what it would take to receive full credit if these were being assigned as homework problems). Rather, as with the rest of the text, the discussions in these sections should be viewed as trying to give a sense of the larger process by which one might think about problems of this type, culminating in the specification of a precise solution.

It is worth mentioning two points concerning the use of these problems as homework in a course. First, the problems are sequenced roughly in order of increasing difficulty, but this is only an approximate guide and we advise against placing too much weight on it: since the bulk of the problems were designed as homework for our undergraduate class, large subsets of the problems in each chapter are really closely comparable in terms of difficulty. Second, aside from the lowest-numbered ones, the problems are designed to involve some investment of time, both to relate the problem description to the algorithmic techniques in the chapter, and then to actually design the necessary algorithm. In our undergraduate class, we have tended to assign roughly three of these problems per week.

Pedagogical Features and Supplements

In addition to the problems and solved exercises, the book has a number of further pedagogical features, as well as additional supplements to facilitate its use for teaching.

As noted earlier, a large number of the sections in the book are devoted to the formulation of an algorithmic problem—including its background and underlying motivation—and the design and analysis of an algorithm for this problem. To reflect this style, these sections are consistently structured around a sequence of subsections: "The Problem," where the problem is described and a precise formulation is worked out; "Designing the Algorithm," where the appropriate design technique is employed to develop an algorithm; and "Analyzing the Algorithm," which proves properties of the algorithm and analyzes its efficiency. These subsections are highlighted in the text with an icon depicting a feather. In cases where extensions to the problem or further analysis of the algorithm is pursued, there are additional subsections devoted to these issues. The goal of this structure is to offer a relatively uniform style of presentation that moves from the initial discussion of a problem arising in a computing application through to the detailed analysis of a method to solve it.

A number of supplements are available in support of the book itself. An instructor's manual works through all the problems, providing full solutions to each. A set of lecture slides, developed by Kevin Wayne of Princeton University, is also available; these slides follow the order of the book's sections and can thus be used as the foundation for lectures in a course based on the book. These files are available at *www.aw.com*. For instructions on obtaining a professor

login and password, search the site for either "Kleinberg" or "Tardos" or contact your local Addison-Wesley representative.

Finally, we would appreciate receiving feedback on the book. In particular, as in any book of this length, there are undoubtedly errors that have remained in the final version. Comments and reports of errors can be sent to us by e-mail, at the address algbook@cs.cornell.edu; please include the word "feedback" in the subject line of the message.

Chapter-by-Chapter Synopsis

Chapter 1 starts by introducing some representative algorithmic problems. We begin immediately with the Stable Matching Problem, since we feel it sets up the basic issues in algorithm design more concretely and more elegantly than any abstract discussion could: stable matching is motivated by a natural though complex real-world issue, from which one can abstract an interesting problem statement and a surprisingly effective algorithm to solve this problem. The remainder of Chapter 1 discusses a list of five "representative problems" that foreshadow topics from the remainder of the course. These five problems are interrelated in the sense that they are all variations and/or special cases of the Independent Set Problem; but one is solvable by a greedy algorithm, one by dynamic programming, one by network flow, one (the Independent Set Problem itself) is NP-complete, and one is PSPACE-complete. The fact that closely related problems can vary greatly in complexity is an important theme of the book, and these five problems serve as milestones that reappear as the book progresses.

Chapters 2 and 3 cover the interface to the CS1/CS2 course sequence mentioned earlier. Chapter 2 introduces the key mathematical definitions and notations used for analyzing algorithms, as well as the motivating principles behind them. It begins with an informal overview of what it means for a problem to be computationally tractable, together with the concept of polynomial time as a formal notion of efficiency. It then discusses growth rates of functions and asymptotic analysis more formally, and offers a guide to commonly occurring functions in algorithm analysis, together with standard applications in which they arise. Chapter 3 covers the basic definitions and algorithmic primitives needed for working with graphs, which are central to so many of the problems in the book. A number of basic graph algorithms are often implemented by students late in the CS1/CS2 course sequence, but it is valuable to present the material here in a broader algorithm design context. In particular, we discuss basic graph definitions, graph traversal techniques such as breadth-first search and depth-first search, and directed graph concepts including strong connectivity and topological ordering.

Chapters 2 and 3 also present many of the basic data structures that will be used for implementing algorithms throughout the book; more advanced data structures are presented in subsequent chapters. Our approach to data structures is to introduce them as they are needed for the implementation of the algorithms being developed in the book. Thus, although many of the data structures covered here will be familiar to students from the CS1/CS2 sequence, our focus is on these data structures in the broader context of algorithm design and analysis.

Chapters 4 through 7 cover four major algorithm design techniques: greedy algorithms, divide and conquer, dynamic programming, and network flow. With greedy algorithms, the challenge is to recognize when they work and when they don't; our coverage of this topic is centered around a way of classifying the kinds of arguments used to prove greedy algorithms correct. This chapter concludes with some of the main applications of greedy algorithms, for shortest paths, undirected and directed spanning trees, clustering, and compression. For divide and conquer, we begin with a discussion of strategies for solving recurrence relations as bounds on running times; we then show how familiarity with these recurrences can guide the design of algorithms that improve over straightforward approaches to a number of basic problems, including the comparison of rankings, the computation of closest pairs of points in the plane, and the Fast Fourier Transform. Next we develop dynamic programming by starting with the recursive intuition behind it, and subsequently building up more and more expressive recurrence formulations through applications in which they naturally arise. This chapter concludes with extended discussions of the dynamic programming approach to two fundamental problems: sequence alignment, with applications in computational biology; and shortest paths in graphs, with connections to Internet routing protocols. Finally, we cover algorithms for network flow problems, devoting much of our focus in this chapter to discussing a large array of different flow applications. To the extent that network flow is covered in algorithms courses, students are often left without an appreciation for the wide range of problems to which it can be applied; we try to do justice to its versatility by presenting applications to load balancing, scheduling, image segmentation, and a number of other problems.

Chapters 8 and 9 cover computational intractability. We devote most of our attention to NP-completeness, organizing the basic NP-complete problems thematically to help students recognize candidates for reductions when they encounter new problems. We build up to some fairly complex proofs of NP-completeness, with guidance on how one goes about constructing a difficult reduction. We also consider types of computational hardness beyond NP-completeness, particularly through the topic of PSPACE-completeness. We

find this is a valuable way to emphasize that intractability doesn't end at NP-completeness, and PSPACE-completeness also forms the underpinning for some central notions from artificial intelligence—planning and game playing—that would otherwise not find a place in the algorithmic landscape we are surveying.

Chapters 10 through 12 cover three major techniques for dealing with computationally intractable problems: identification of structured special cases, approximation algorithms, and local search heuristics. Our chapter on tractable special cases emphasizes that instances of NP-complete problems arising in practice may not be nearly as hard as worst-case instances, because they often contain some structure that can be exploited in the design of an efficient algorithm. We illustrate how NP-complete problems are often efficiently solvable when restricted to tree-structured inputs, and we conclude with an extended discussion of tree decompositions of graphs. While this topic is more suitable for a graduate course than for an undergraduate one, it is a technique with considerable practical utility for which it is hard to find an existing accessible reference for students. Our chapter on approximation algorithms discusses both the process of designing effective algorithms and the task of understanding the optimal solution well enough to obtain good bounds on it. As design techniques for approximation algorithms, we focus on greedy algorithms, linear programming, and a third method we refer to as "pricing," which incorporates ideas from each of the first two. Finally, we discuss local search heuristics, including the Metropolis algorithm and simulated annealing. This topic is often missing from undergraduate algorithms courses, because very little is known in the way of provable guarantees for these algorithms; however, given their widespread use in practice, we feel it is valuable for students to know something about them, and we also include some cases in which guarantees can be proved.

Chapter 13 covers the use of randomization in the design of algorithms. This is a topic on which several nice graduate-level books have been written. Our goal here is to provide a more compact introduction to some of the ways in which students can apply randomized techniques using the kind of background in probability one typically gains from an undergraduate discrete math course.

Use of the Book

The book is primarily designed for use in a first undergraduate course on algorithms, but it can also be used as the basis for an introductory graduate course.

When we use the book at the undergraduate level, we spend roughly one lecture per numbered section; in cases where there is more than one

lecture's worth of material in a section (for example, when a section provides further applications as additional examples), we treat this extra material as a supplement that students can read about outside of lecture. We skip the starred sections; while these sections contain important topics, they are less central to the development of the subject, and in some cases they are harder as well. We also tend to skip one or two other sections per chapter in the first half of the book (for example, we tend to skip Sections 4.3, 4.7–4.8, 5.5–5.6, 6.5, 7.6, and 7.11). We cover roughly half of each of Chapters 11–13.

This last point is worth emphasizing: rather than viewing the later chapters as "advanced," and hence off-limits to undergraduate algorithms courses, we have designed them with the goal that the first few sections of each should be accessible to an undergraduate audience. Our own undergraduate course involves material from all these chapters, as we feel that all of these topics have an important place at the undergraduate level.

Finally, we treat Chapters 2 and 3 primarily as a review of material from earlier courses; but, as discussed above, the use of these two chapters depends heavily on the relationship of each specific course to its prerequisites.

The resulting syllabus looks roughly as follows: Chapter 1; Chapters 4–8 (excluding 4.3, 4.7–4.9, 5.5–5.6, 6.5, 6.10, 7.4, 7.6, 7.11, and 7.13); Chapter 9 (briefly); Chapter 10, Sections.10.1 and 10.2; Chapter 11, Sections 11.1, 11.2, 11.6, and 11.8; Chapter 12, Sections 12.1–12.3; and Chapter 13, Sections 13.1–13.5.

The book also naturally supports an introductory graduate course on algorithms. Our view of such a course is that it should introduce students destined for research in all different areas to the important current themes in algorithm design. Here we find the emphasis on formulating problems to be useful as well, since students will soon be trying to define their own research problems in many different subfields. For this type of course, we cover the later topics in Chapters 4 and 6 (Sections 4.5–4.9 and 6.5–6.10), cover all of Chapter 7 (moving more rapidly through the early sections), quickly cover NP-completeness in Chapter 8 (since many beginning graduate students will have seen this topic as undergraduates), and then spend the remainder of the time on Chapters 10–13. Although our focus in an introductory graduate course is on the more advanced sections, we find it useful for the students to have the full book to consult for reviewing or filling in background knowledge, given the range of different undergraduate backgrounds among the students in such a course.

Finally, the book can be used to support self-study by graduate students, researchers, or computer professionals who want to get a sense for how they