## Contents

**About the Authors**

**Preface**

1. **Introduction: Some Representative Problems**
   - 1.1 A First Problem: Stable Matching  
   - 1.2 Five Representative Problems  
     - Solved Exercises  
     - Exercises  
     - Notes and Further Reading

2. **Basics of Algorithm Analysis**
   - 2.1 Computational Tractability  
   - 2.2 Asymptotic Order of Growth  
   - 2.3 Implementing the Stable Matching Algorithm Using Lists and Arrays  
   - 2.4 A Survey of Common Running Times  
   - 2.5 A More Complex Data Structure: Priority Queues  
     - Solved Exercises  
     - Exercises  
     - Notes and Further Reading

3. **Graphs**
   - 3.1 Basic Definitions and Applications  
   - 3.2 Graph Connectivity and Graph Traversal  
   - 3.3 Implementing Graph Traversal Using Queues and Stacks  
   - 3.4 Testing Bipartiteness: An Application of Breadth-First Search  
   - 3.5 Connectivity in Directed Graphs
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

Preface
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
Preface

We begin by reviewing the material covered in previous courses, with a focus on 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.

Finally, the book also naturally supports an introductory graduate course on algorithms. Our view of such a course is that it should introduce students 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.

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 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.

The book also naturally supports an introductory graduate course on algorithms. Our view of such a course is that it should introduce students 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.

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 might be able to use particular algorithm design techniques in the context of their own work. A number of graduate students and colleagues have used portions of the book in this way.

Acknowledgments

This book grew out of the sequence of algorithms courses that we have taught at Cornell. These courses have grown, as the field has grown, over a number of years, and they reflect the influence of the Cornell faculty who helped to shape them during this time, including Juris Hartmanis, Monika Henzinger, John Hopcroft, Dexter Kozen, Ronitt Rubinfeld, and Sam Toueg. More generally, we would like to thank all our colleagues at Cornell for countless discussions both on the material here and on broader issues about the nature of the field.

The course staffs we've had in teaching the subject have been tremendously helpful in the formulation of this material. We thank our undergraduate and graduate teaching assistants, Siddharth Alexander, Rie Ando, Elliot Anshelevich, Lars Backstrom, Steve Baker, Ralph Benzenz, John Bicket, Doug Burdick, Mike Connor, Vladimir Dizhoor, Shaddin Doghmi, Alexander Druyan, Bowei Du, Sasha Evfimievski, Ariful Gani, Vadim Grinshpun, Ara Hayrapetyan, Chris Jeuell, Igor Kats, Omar Khan, Mikhail Kobyakov, Alexei Kopylov, Brian Kulis, Amit Kumar, Yeongwoo Lee, Henry Lin, Ashwin Machanavajjhala, Ayan Mandal, Bill McCloskey, Leonid Meyerghuz, Evan Moran, Niranjan Nagarajan, Tina Nolte, Travis Ortogero, Martin Pál, Jon Peress, Matt Piotrowski, Joe Polastre, Mike Priscott, Xin Qi, Venu Ramasubramanian, Aditya Rao, David Richardson, Brian Sabino, Rachit Siamwalla, Sebastian Silgardo, Alex Slivkins, Chaithanya Swamy, Perry Tam, Nadya Travinin, Sergei Vassilvitski, Matthew Wachs, Tom Wexler, Shan-Leung Maverick Woo, Justin Yang, and Misha Zatsman. Many of them have provided valuable insights, suggestions, and comments on the text. We also thank all the students in these classes who have provided comments and feedback on early drafts of the book over the years.

For the past several years, the development of the book has benefited greatly from the feedback and advice of colleagues who have used prepublication drafts for teaching. Anna Karlin fearlessly adopted a draft as her course textbook at the University of Washington when it was still in an early stage of development; she was followed by a number of people who have used it either as a course textbook or as a resource for teaching: Paul Beame, Allan Borodin, Devdatt Dubhashi, David Kempe, Gene Kleinberg, Dexter Kozen, Amit Kumar, Mike Molloy, Yuval Rabani, Tim Roughgarden, Alexa Sharp, Shanghua Teng, Aravind Srinivasan, Dieter van Melkebeek, Kevin Wayne, Tom Wexler, and
Sue Whitesides. We deeply appreciate their input and advice, which has informed many of our revisions to the content. We would like to additionally thank Kevin Wayne for producing supplementary material associated with the book, which promises to greatly extend its utility to future instructors.

In a number of other cases, our approach to particular topics in the book reflects the influence of specific colleagues. Many of these contributions have undoubtedly escaped our notice, but we especially thank Yuri Boykov, Ron Elber, Dan Huttenlocher, Bobby Kleinberg, Evie Kleinberg, Lillian Lee, David McAllester, Mark Newman, Prabhakar Raghavan, Bart Selman, David Shmoys, Steve Strogatz, Olga Veksler, Duncan Watts, and Ramin Zabih.

It has been a pleasure working with Addison Wesley over the past year. First and foremost, we thank Matt Goldstein for all his advice and guidance in this process, and for helping us to synthesize a vast amount of review material into a concrete plan that improved the book. Our early conversations about the book with Susan Hartman were extremely valuable as well. We thank Matt and Susan, together with Michelle Brown, Marilyn Lloyd, Patty Mahtani, and Malte Suarez-Rivas at Addison Wesley, and Paul Anagnostopoulos and Jacqui Scarrott at Windfall Software, for all their work on the editing, production, and management of the project. We further thank Paul and Jacqui for their expert composition of the book. We thank Joyce Wells for the cover design, Nancy Murphy of Dartmouth Publishing for her work on the figures, Ted Laux for the indexing, and Carol Leyba and Jennifer McClain for the copyediting and proofreading.

We thank Anselm Blumer (Tufts University), Richard Chang (University of Maryland, Baltimore County), Kevin Compton (University of Michigan), Diane Cook (University of Texas, Arlington), Sariel Har-Peled (University of Illinois, Urbana-Champaign), Sanjeev Khanna (University of Pennsylvania), Philip Klein (Brown University), David Matthias (Ohio State University), Adam Meyerson (UCLA), Michael Mitzenmacher (Harvard University), Stephan Olariu (Old Dominion University), Mohan Paturi (UC San Diego), Edgar Ramos (University of Illinois, Urbana-Champaign), Sanjay Ranka (University of Florida, Gainesville), Leon Reznik (Rochester Institute of Technology), Subhash Suri (UC Santa Barbara), Dieter van Melkebeek (University of Wisconsin, Madison), and Bulent Yener (Rensselaer Polytechnic Institute) who generously contributed their time to provide detailed and thoughtful reviews of the manuscript; their comments led to numerous improvements, both large and small, in the final version of the text.

Finally, we thank our families—Lillian and Alice, and David, Rebecca, and Amy. We appreciate their support, patience, and many other contributions more than we can express in any acknowledgments here.

This book was begun amid the irrational exuberance of the late nineties, when the arc of computing technology seemed, to many of us, briefly to pass through a place traditionally occupied by celebrities and other inhabitants of the pop-cultural firmament. (It was probably just in our imaginations.) Now, several years after the hype and stock prices have come back to earth, one can appreciate that in some ways computer science was forever changed by this period, and in other ways it has remained the same: the driving excitement that has characterized the field since its early days is as strong and enticing as ever, the public's fascination with information technology is still vibrant, and the reach of computing continues to extend into new disciplines. And so to all students of the subject, drawn to it for so many different reasons, we hope you find this book an enjoyable and useful guide wherever your computational pursuits may take you.

Jon Kleinberg
Éva Tardos
Ithaca, 2005
1.1 A First Problem: Stable Matching

As an opening topic, we look at an algorithmic problem that nicely illustrates many of the themes we will be emphasizing. It is motivated by some very natural and practical concerns, and from these we formulate a clean and simple statement of a problem. The algorithm to solve the problem is very clean as well, and most of our work will be spent in proving that it is correct and giving an acceptable bound on the amount of time it takes to terminate with an answer. The problem itself—the Stable Matching Problem—has several origins.

The Problem

The Stable Matching Problem originated, in part, in 1962, when David Gale and Lloyd Shapley, two mathematical economists, asked the question: Could one design a college admissions process, or a job recruiting process, that was self-enforcing? What did they mean by this?

To set up the question, let’s first think informally about the kind of situation that might arise as a group of friends, all juniors in college majoring in computer science, begin applying to companies for summer internships. The crux of the application process is the interplay between two different types of parties: companies (the employers) and students (the applicants). Each applicant has a preference ordering on companies, and each company—once the applications come in—forms a preference ordering on its applicants. Based on these preferences, companies extend offers to some of their applicants, applicants choose which of their offers to accept, and people begin heading off to their summer internships.
Gale and Shapley considered the sorts of things that could start going wrong with this process, in the absence of any mechanism to enforce the status quo. Suppose, for example, that your friend Raj has just accepted a summer job at the large telecommunications company CluNet. A few days later, the small start-up company WebExodus, which had been dragging its feet on making a few final decisions, calls up Raj and offers him a summer job as well. Now, Raj actually prefers WebExodus to CluNet—won over perhaps by the laid-back, anything-can-happen atmosphere—and so this new development may well cause him to retract his acceptance of the CluNet offer and go to WebExodus instead. Suddenly down one summer intern, CluNet offers a job to one of its few final decision call-ups Ra…

Chapter 1 Introduction: Some Representative Problems

Situations like this can rapidly generate a lot of chaos, and many people—both applicants and employers—can end up unhappy with the process as well as the outcome. What has gone wrong? One basic problem is that the process is not self-enforcing—if people are allowed to act in their self-interest, then it risks breaking down.

We might well prefer the following, more stable situation, in which self-interest itself prevents offers from being retracted and redirected. Consider another student, who has arranged to spend the summer at CluNet but calls up WebExodus and reveals that he, too, would rather work for them. But in this case, based on the offers already accepted, they are able to reply, “No, it turns out that we prefer each of the students we’ve accepted to you, so we’re afraid there’s nothing we can do.” Or consider an employer, earnestly following up with its top applicants who went elsewhere, being told by each of them, “No, I’m happy where I am.” In such a case, all the outcomes are stable—there are no further outside deals that can be made.

So this is the question Gale and Shapley asked: Given a set of preferences among employers and applicants, can we assign applicants to employers so that for every employer E, and every applicant A who is not scheduled to work for E, at least one of the following two things is the case?

(i) E prefers every one of its accepted applicants to A; or
(ii) A prefers her current situation over working for employer E.

If this holds, the outcome is stable: individual self-interest will prevent any applicant/employer deal from being made behind the scenes.

Gale and Shapley proceeded to develop a striking algorithmic solution to this problem, which we will discuss presently. Before doing this, let’s note that this is not the only origin of the Stable Matching Problem. It turns out that for a decade before the work of Gale and Shapley, unbeknownst to them, the National Resident Matching Program had been using a very similar procedure, with the same underlying motivation, to match residents to hospitals. Indeed, this system, with relatively little change, is still in use today.

This is one testament to the problem’s fundamental appeal. And from the point of view of this book, it provides us with a nice first domain in which to reason about some basic combinatorial definitions and the algorithms that build on them.

Formulating the Problem To get at the essence of this concept, it helps to make the problem as clean as possible. The world of companies and applicants contains some distracting asymmetries. Each applicant is looking for a single company, but each company is looking for many applicants; moreover, there may be more (or, as is sometimes the case, fewer) applicants than there are available slots for summer jobs. Finally, each applicant does not typically apply to every company.

It is useful, at least initially, to eliminate these complications and arrive at a more “bare-bones” version of the problem: each of n applicants applies to each of n companies, and each company wants to accept a single applicant. We will see that doing this preserves the fundamental issues inherent in the problem; in particular, our solution to this simplified version will extend directly to the more general case as well.

Following Gale and Shapley, we observe that this special case can be viewed as the problem of devising a system by which each of n men and n women can end up getting married: our problem naturally has the analogue of two “genders”—the applicants and the companies—and in the case we are considering, everyone is seeking to be paired with exactly one individual of the opposite gender.1

1 Gale and Shapley considered the same-sex Stable Matching Problem as well, where there is only a single gender. This is motivated by related applications, but it turns out to be fairly different at a technical level. Given the applicant-employer application we’re considering here, we’ll be focusing on the version with two genders.
Chapter 1 Introduction: Some Representative Problems

So consider a set $M = \{m_1, \ldots, m_n\}$ of $n$ men, and a set $W = \{w_1, \ldots, w_n\}$ of $n$ women. Let $M \times W$ denote the set of all possible ordered pairs of the form $(m, w)$, where $m \in M$ and $w \in W$. A matching $S$ is a set of ordered pairs, each from $M \times W$, with the property that each member of $M$ and each member of $W$ appears in at most one pair in $S$. A perfect matching $S'$ is a matching with the property that each member of $M$ and each member of $W$ appears in exactly one pair in $S'$.

Matchings and perfect matchings are objects that will recur frequently throughout the book; they arise naturally in modeling a wide range of algorithmic problems. In the present situation, a perfect matching corresponds simply to a way of pairing off the men with the women, in such a way that everyone ends up married to somebody, and nobody is married to more than one person—there is neither singlehood nor polygamy.

Now we can add the notion of preferences to this setting. Each man $m \in M$ ranks all the women; we will say that $m$ prefers $w$ to $w'$ if $m$ ranks $w$ higher than $w'$. We will refer to the ordered ranking of $m$ as his preference list. We will not allow ties in the ranking. Each woman, analogously, ranks all the men.

Given a perfect matching $S$, what can go wrong? Guided by our initial motivation in terms of employers and applicants, we should be worried about the following situation: There are two pairs $(m, w)$ and $(m', w')$ in $S$ (as depicted in Figure 1.1) with the property that $m$ prefers $w'$ to $w$, and $m'$ prefers $m$ to $m'$. In this case, there's nothing to stop $m$ and $w'$ from abandoning their current partners and heading off together; the set of marriages is not self-enforcing. We'll say that such a pair $(m, w')$ is an instability with respect to $S$: $(m, w')$ does not belong to $S$, but each of $m$ and $w'$ prefers the other to their partner in $S$.

Our goal, then, is a set of marriages with no instabilities. We'll say that a matching $S$ is stable if (i) it is perfect, and (ii) there is no instability with respect to $S$. Two questions spring immediately to mind:

- Does there exist a stable matching for every set of preference lists?
- Given a set of preference lists, can we efficiently construct a stable matching if there is one?

**Some Examples** To illustrate these definitions, consider the following two very simple instances of the Stable Matching Problem.

First, suppose we have a set of two men, $\{m, m'\}$, and a set of two women, $\{w, w'\}$. The preference lists are as follows:

- $m$ prefers $w$ to $w'$.
- $m'$ prefers $w$ to $w'$.

If we think about this set of preference lists intuitively, it represents complete agreement: the men agree on the order of the women, and the women agree on the order of the men. There is a unique stable matching here, consisting of the pairs $(m, w)$ and $(m', w')$. The other perfect matching, consisting of the pairs $(m', w)$ and $(m, w')$, would not be a stable matching, because the pair $(m, w)$ would form an instability with respect to this matching. (Both $m$ and $w$ would want to leave their respective partners and pair up.)

Next, here's an example where things are a bit more intricate. Suppose the preferences are

- $m$ prefers $w$ to $w'$.
- $m'$ prefers $w'$ to $w$.
- $w$ prefers $m'$ to $m$.
- $w'$ prefers $m$ to $m'$.

What's going on in this case? The two men's preferences mesh perfectly with each other (they rank different women first), and the two women's preferences likewise mesh perfectly with each other. But the men's preferences clash completely with the women's preferences.

In this second example, there are two different stable matchings. The matching consisting of the pairs $(m, w)$ and $(m', w')$ is stable, because both men are as happy as possible, so neither would leave their matched partner. But the matching consisting of the pairs $(m', w)$ and $(m, w')$ is also stable, for the complementary reason that both women are as happy as possible. This is an important point to remember as we go forward—it's possible for an instance to have more than one stable matching.

**Designing the Algorithm**

We now show that there exists a stable matching for every set of preference lists among the men and women. Moreover, our means of showing this will also answer the second question that we asked above: we will give an efficient algorithm that takes the preference lists and constructs a stable matching.

Let us consider some of the basic ideas that motivate the algorithm.

- Initially, everyone is unmarried. Suppose an unmarried man $m$ chooses the woman $w$ who ranks highest on his preference list and proposes to her. Can we declare immediately that $(m, w)$ will be one of the pairs in our final stable matching? Not necessarily: at some point in the future, a man $m'$ whom $w$ prefers may propose to her. On the other hand, it would be
dangerous for \( w \) to reject \( m \) right away; she may never receive a proposal from someone she ranks as highly as \( m \). So a natural idea would be to have the pair \((m, w)\) enter an intermediate state—engagement.

- Suppose we are now at a state in which some men and women are free—not engaged—and some are engaged. The next step could look like this. An arbitrary free man \( m \) chooses the highest-ranked woman \( w \) to whom he has not yet proposed, and he proposes to her. If \( w \) is also free, then \( m \) and \( w \) become engaged. Otherwise, \( w \) is already engaged to some other man \( m' \). In this case, she determines which of \( m \) or \( m' \) ranks higher on her preference list; this man becomes engaged to \( w \) and the other becomes free.
- Finally, the algorithm will terminate when no one is free; at this moment, all engagements are declared final, and the resulting perfect matching is returned.

Here is a concrete description of the Gale-Shapley algorithm, with Figure 1.2 depicting a state of the algorithm.

Initially all \( m \in M \) and \( w \in W \) are free
While there is a man \( m \) who is free and hasn't proposed to every woman
    Choose such a man \( m \)
    Let \( w \) be the highest-ranked woman in \( m \)'s preference list
to whom \( m \) has not yet proposed
    If \( w \) is free then
        \((m, w)\) become engaged
    Else \( w \) is currently engaged to \( m' \)
        If \( w \) prefers \( m' \) to \( m \) then
            \( m \) remains free
        Else \( w \) prefers \( m \) to \( m' \)
            \((m, w)\) become engaged
            \( m' \) becomes free
    Endif
Endwhile
Return the set \( S \) of engaged pairs

An intriguing thing is that, although the G-S algorithm is quite simple to state, it is not immediately obvious that it returns a stable matching, or even a perfect matching. We proceed to prove this now, through a sequence of intermediate facts.

### Analyzing the Algorithm

First consider the view of a woman \( w \) during the execution of the algorithm. For a while, no one has proposed to her, and she is free. Then a man \( m \) may propose to her, and she becomes engaged. As time goes on, she may receive additional proposals, accepting those that increase the rank of her partner. So we discover the following.

1. \( w \) remains engaged from the point at which she receives her first proposal; and the sequence of partners to which she is engaged gets better and better (in terms of her preference list).

The view of a man \( m \) during the execution of the algorithm is rather different. He is free until he proposes to the highest-ranked woman on his list; at this point he may or may not become engaged. As time goes on, he may alternate between being free and being engaged; however, the following property does hold.

2. The sequence of women to whom \( m \) proposes gets worse and worse (in terms of his preference list).

Now we show that the algorithm terminates, and give a bound on the maximum number of iterations needed for termination.

3. The G-S algorithm terminates after at most \( n^2 \) iterations of the While loop.

**Proof.** A useful strategy for upper-bounding the running time of an algorithm, as we are trying to do here, is to find a measure of progress. Namely, we seek some precise way of saying that each step taken by the algorithm brings it closer to termination.

In the case of the present algorithm, each iteration consists of some man proposing (for the only time) to a woman he has never proposed to before. So if we let \( P(t) \) denote the set of pairs \((m, w)\) such that \( m \) has proposed to \( w \) by the end of iteration \( t \), we see that for all \( t \), the size of \( P(t + 1) \) is strictly greater than the size of \( P(t) \). But there are only \( n^2 \) possible pairs of men and women in total, so the value of \( P(\cdot) \) can increase at most \( n^2 \) times over the course of the algorithm. It follows that there can be at most \( n^2 \) iterations. 

Two points are worth noting about the previous fact and its proof. First, there are executions of the algorithm (with certain preference lists) that can involve close to \( n^2 \) iterations, so this analysis is not far from the best possible. Second, there are many quantities that would not have worked well as a progress measure for the algorithm, since they need not strictly increase in each
iteration. For example, the number of free individuals could remain constant from one iteration to the next, as could the number of engaged pairs. Thus, these quantities could not be used directly in giving an upper bound on the maximum possible number of iterations, in the style of the previous paragraph.

Let us now establish that the set $S$ returned at the termination of the algorithm is in fact a perfect matching. Why is this not immediately obvious? Essentially, we have to show that no man can “fall off” the end of his preference list; the only way for the while loop to exit is for there to be no free man. In this case, the set of engaged couples would indeed be a perfect matching.

So the main thing we need to show is the following.

(1.4) If $m$ is free at some point in the execution of the algorithm, then there is a woman to whom he has not yet proposed.

Proof. Suppose there comes a point when $m$ is free but has already proposed to every woman. Then by (1.1), each of the $n$ women is engaged at this point in time. Since the set of engaged pairs forms a matching, there must also be $n$ engaged men at this point in time. But there are only $n$ men total, and $m$ is not engaged, so this is a contradiction.

(1.5) The set $S$ returned at termination is a perfect matching.

Proof. The set of engaged pairs always forms a matching. Let us suppose that the algorithm terminates with a free man $m$. At termination, it must be the case that $m$ had already proposed to every woman, for otherwise the while loop would not have exited. But this contradicts (1.4), which says that there cannot be a free man who has proposed to every woman.

Finally, we prove the main property of the algorithm—namely, that it results in a stable matching.

(1.6) Consider an execution of the G-S algorithm that returns a set of pairs $S$. The set $S$ is a stable matching.

Proof. We have already seen, in (1.5), that $S$ is a perfect matching. Thus, to prove $S$ is a stable matching, we will assume that there is an instability with respect to $S$ and obtain a contradiction. As defined earlier, such an instability would involve two pairs, $(m, w)$ and $(m', w')$, in $S$ with the properties that

- $m$ prefers $w'$ to $w$,
- $w'$ prefers $m$ to $m'$.

In the execution of the algorithm that produced $S$, $m$’s last proposal was, by definition, to $w$. Now we ask: Did $m$ propose to $w'$ at some earlier point in this execution? If he didn’t, then $w$ must occur higher on $m$’s preference list than $w'$, contradicting our assumption that $m$ prefers $w'$ to $w$. If he did, then he was rejected by $w'$ in favor of some other man $m''$, whom $w'$ prefers to $m$. $m'$ is the final partner of $w'$, so either $m'' = m'$ or, by (1.1), $w'$ prefers her final partner $m'$ to $m''$; either way this contradicts our assumption that $w'$ prefers $m$ to $m'$.

It follows that $S$ is a stable matching.

Extensions

We began by defining the notion of a stable matching; we have just proven that the G-S algorithm actually constructs one. We now consider some further questions about the behavior of the G-S algorithm and its relation to the properties of different stable matchings.

To begin with, recall that we saw an example earlier in which there could be multiple stable matchings. To recap, the preference lists in this example were as follows:

- $m$ prefers $w$ to $w'$.
- $m'$ prefers $w'$ to $w$.
- $w$ prefers $m$ to $m'$.
- $w'$ prefers $m'$ to $m$.

Now, in any execution of the Gale-Shapley algorithm, $m$ will become engaged to $w$, $m'$ will become engaged to $w'$ (perhaps in the other order), and things will stop there. Thus, the other stable matching, consisting of the pairs $(m', w)$ and $(m, w')$, is not attainable from an execution of the G-S algorithm in which the men propose. On the other hand, it would be reached if we ran a version of the algorithm in which the women propose. And in larger examples, with more than two people on each side, we can have an even larger collection of possible stable matchings, many of them not achievable by any natural algorithm.

This example shows a certain “unfairness” in the G-S algorithm, favoring men. If the men’s preferences mesh perfectly (they all list different women as their first choice), then in all runs of the G-S algorithm all men end up matched with their first choice, independent of the preferences of the women. If the women’s preferences clash completely with the men’s preferences (as was the case in this example), then the resulting stable matching is as bad as possible for the women. So this simple set of preference lists compactly summarizes a world in which someone is destined to end up unhappy: women are unhappy if men propose, and men are unhappy if women propose.

Let’s now analyze the G-S algorithm in more detail and try to understand how general this “unfairness” phenomenon is.
To begin with, our example reinforces the point that the G-S algorithm is actually underspecified: as long as there is a free man, we are allowed to choose any free man to make the next proposal. Different choices specify different executions of the algorithm; this is why, to be careful, we stated (1.6) as “Consider an execution of the G-S algorithm that returns a set of pairs $S^*$,” instead of “Consider the set $S$ returned by the G-S algorithm.”

Thus, we encounter another very natural question: Do all executions of the G-S algorithm yield the same matching? This is a genre of question that arises in many settings in computer science: we have an algorithm that runs asynchronously, with different independent components performing actions that can be interleaved in complex ways, and we want to know how much variability this asynchrony causes in the final outcome. To consider a very different kind of example, the independent components may not be men and women but electronic components activating parts of an airplane wing; the effect of asynchrony in their behavior can be a big deal.

In the present context, we will see that the answer to our question is surprisingly clean: all executions of the G-S algorithm yield the same matching. We proceed to prove this now.

**All Executions Yield the Same Matching** There are a number of possible ways to prove a statement such as this, many of which would result in quite complicated arguments. It turns out that the easiest and most informative approach for us will be to uniquely characterize the matching that is obtained and then show that all executions result in the matching with this characterization.

What is the characterization? We’ll show that each man ends up with the “best possible partner” in a concrete sense. (Recall that this is true if all men prefer different women.) First, we will say that a woman $w$ is a valid partner of a man $m$ if there is a stable matching that contains the pair $(m, w)$. We will say that $w$ is the best valid partner of $m$ if $w$ is a valid partner of $m$, and no woman whom $m$ ranks higher than $w$ is a valid partner of his. We will use $\text{best}(m)$ to denote the best valid partner of $m$.

Now, let $S^*$ denote the set of pairs $\{(m, \text{best}(m)) : m \in M\}$. We will prove the following fact.

\begin{equation}
\text{(1.7)} \quad \text{Every execution of the G-S algorithm results in the set } S^*.
\end{equation}

This statement is surprising at a number of levels. First of all, as defined, there is no reason to believe that $S^*$ is a matching at all, let alone a stable matching. After all, why couldn’t it happen that two men have the same best valid partner? Second, the result shows that the G-S algorithm gives the best possible outcome for every man simultaneously; there is no stable matching in which any of the men could have hoped to do better. And finally, it answers our question above by showing that the order of proposals in the G-S algorithm has absolutely no effect on the final outcome.

Despite all this, the proof is not so difficult.

**Proof.** Let us suppose, by way of contradiction, that some execution $E$ of the G-S algorithm results in a matching $S$ in which some man is paired with a woman who is not his best valid partner. Since men propose in decreasing order of preference, this means that some man is rejected by a valid partner during the execution $E$ of the algorithm. So consider the first moment during the execution $E$ in which some man, say $m$, is rejected by a valid partner $w$. Again, since men propose in decreasing order of preference, and since this is the first time such a rejection has occurred, it must be that $w$ is $m$’s best valid partner $\text{best}(m)$.

The rejection of $m$ by $w$ may have happened either because $m$ proposed and was turned down in favor of $w$’s existing engagement, or because $w$ broke her engagement to $m$ in favor of a better proposal. But either way, at this moment $w$ forms or continues an engagement with a man $m’$ whom she prefers to $m$.

Since $w$ is a valid partner of $m$, there exists a stable matching $S’$ containing the pair $(m, w)$. Now we ask: Who is $m’$ paired with in this matching? Suppose it is a woman $w’ \neq w$.

Since the rejection of $m$ by $w$ was the first rejection of a man by a valid partner in the execution $E$, it must be that $m’$ had not been rejected by any valid partner at the point in $E$ when he became engaged to $w$. Since he proposed in decreasing order of preference, and since $w’$ is clearly a valid partner of $m’$, it must be that $m’$ prefers $w$ to $w’$. But we have already seen that $w$ prefers $m$ to $m’$, for in execution $E$ she rejected $m$ in favor of $m’$. Since $(m’, w) \notin S’$, it follows that $(m’, w)$ is an instability in $S’$.

This contradicts our claim that $S’$ is stable and hence contradicts our initial assumption. 

So for the men, the G-S algorithm is ideal. Unfortunately, the same cannot be said for the women. For a woman $w$, we say that $m$ is a valid partner if there is a stable matching that contains the pair $(m, w)$. We say that $m$ is the worst valid partner of $w$ if $m$ is a valid partner of $w$, and no man whom $w$ ranks lower than $m$ is a valid partner of hers.

\begin{equation}
\text{(1.8)} \quad \text{In the stable matching } S^*, \text{ each woman is paired with her worst valid partner.}
\end{equation}

**Proof.** Suppose there were a pair $(m, w)$ in $S^*$ such that $m$ is not the worst valid partner of $w$. Then there is a stable matching $S’$ in which $w$ is paired
with a man \( m' \) whom she likes less than \( m \). In \( S' \), \( m \) is paired with a woman \( w' \neq w \); since \( w \) is the best valid partner of \( m \), and \( w' \) is a valid partner of \( m \), we see that \( m \) prefers \( w \) to \( w' \).

But from this it follows that \((m, w)\) is an instability in \( S' \), contradicting the claim that \( S' \) is stable and hence contradicting our initial assumption.  

Thus, we find that our simple example above, in which the men's preferences clashed with the women's, hinted at a very general phenomenon: for any input, the side that does the proposing in the G-S algorithm ends up with the best possible stable matching (from their perspective), while the side that does not do the proposing correspondingly ends up with the worst possible stable matching.

1.2 Five Representative Problems

The Stable Matching Problem provides us with a rich example of the process of algorithm design. For many problems, this process involves a few significant, steps: formulating the problem with enough mathematical precision that we can ask a concrete question and start thinking about algorithms to solve it; designing an algorithm for the problem; and analyzing the algorithm by proving it is correct and giving a bound on the running time so as to establish the algorithm's efficiency.

This high-level strategy is carried out in practice with the help of a few fundamental design techniques, which are very useful in assessing the inherent complexity of a problem and in formulating an algorithm to solve it. As in any area, becoming familiar with these design techniques is a gradual process; but with experience one can start recognizing problems as belonging to identifiable genres and appreciating how subtle changes in the statement of a problem can have an enormous effect on its computational difficulty.

To get this discussion started, then, it helps to pick out a few representative milestones that we'll be encountering in our study of algorithms: cleanly formulated problems, all resembling one another at a general level, but differing greatly in their difficulty and in the kinds of approaches that one brings to bear on them. The first three will be solvable efficiently by a sequence of increasingly subtle algorithmic techniques; the fourth marks a major turning point in our discussion, serving as an example of a problem believed to be unsolvable by any efficient algorithm; and the fifth hints at a class of problems believed to be harder still.

The problems are self-contained and are all motivated by computing applications. To talk about some of them, though, it will help to use the terminology of graphs. While graphs are a common topic in earlier computer science courses, we'll be introducing them in a fair amount of depth in Chapter 3; due to their enormous expressive power, we'll also be using them extensively throughout the book. For the discussion here, it's enough to think of a graph \( G \) as simply a way of encoding pairwise relationships among a set of objects. Thus, \( G \) consists of a pair of sets \((V, E)\)—a collection \( V \) of nodes and a collection \( E \) of edges, each of which "joins" two of the nodes. We thus represent an edge \( e \in E \) as a two-element subset of \( V \): \( e = \{u, v\} \) for some \( u, v \in V \), where we call \( u \) and \( v \) the ends of \( e \). We typically draw graphs as in Figure 1.3, with each node as a small circle and each edge as a line segment joining its two ends.

Let's now turn to a discussion of the five representative problems.

Interval Scheduling

Consider the following very simple scheduling problem. You have a resource—it may be a lecture room, a supercomputer, or an electron microscope—and many people request to use the resource for periods of time. A request takes the form: Can I reserve the resource starting at time \( s \), until time \( f \)? We will assume that the resource can be used by at most one person at a time. A scheduler wants to accept a subset of these requests, rejecting all others, so that the accepted requests do not overlap in time. The goal is to maximize the number of requests accepted.

More formally, there will be \( n \) requests labeled \( 1, \ldots, n \), with each request \( i \) specifying a start time \( s_i \) and a finish time \( f_i \). Naturally, we have \( s_i < f_i \) for all \( i \). Two requests \( i \) and \( j \) are compatible if the requested intervals do not overlap: that is, either request \( i \) is for an earlier time interval than request \( j \) (\( f_i \leq s_j \)), or request \( i \) is for a later time than request \( j \) (\( f_i \leq s_j \)). We'll say more generally that a subset \( A \) of requests is compatible if all pairs of requests \( i, j \in A, i \neq j \) are compatible. The goal is to select a compatible subset of requests of maximum possible size.

We illustrate an instance of this Interval Scheduling Problem in Figure 1.4. Note that there is a single compatible set of size 4, and this is the largest compatible set.
We will see shortly that this problem can be solved by a very natural algorithm that orders the set of requests according to a certain heuristic and then "greedily" processes them in one pass, selecting as large a compatible subset as it can. This will be typical of a class of greedy algorithms that we will consider for various problems—myopic rules that process the input one piece at a time with no apparent look-ahead. When a greedy algorithm can be shown to find an optimal solution for all instances of a problem, it's often fairly surprising. We typically learn something about the structure of the underlying problem from the fact that such a simple approach can be optimal.

**Weighted Interval Scheduling**

In the Interval Scheduling Problem, we sought to maximize the number of requests that could be accommodated simultaneously. Now, suppose more generally that each request interval \( i \) has an associated value, or weight, \( v_i > 0 \); we could picture this as the amount of money we will make from the \( i^{th} \) individual if we schedule his or her request. Our goal will be to find a compatible subset of intervals of maximum total value.

The case in which \( v_i = 1 \) for each \( i \) is simply the basic Interval Scheduling Problem; but the appearance of arbitrary values changes the nature of the maximization problem quite a bit. Consider, for example, that if \( v_i \) exceeds the sum of all other \( v_i \), then the optimal solution must include interval 1 regardless of the configuration of the full set of intervals. So any algorithm for this problem must be very sensitive to the values, and yet degenerate to a method for solving (unweighted) interval scheduling when all the values are equal to 1.

There appears to be no simple greedy rule that walks through the intervals one at a time, making the correct decision in the presence of arbitrary values. Instead, we employ a technique, dynamic programming, that builds up the optimal value over all possible solutions in a compact, tabular way that leads to a very efficient algorithm.

**Bipartite Matching**

When we considered the Stable Matching Problem, we defined a matching to be a set of ordered pairs of men and women with the property that each man and each woman belong to at most one of the ordered pairs. We then defined a perfect matching to be a matching in which every man and every woman belong to some pair.

We can express these concepts more generally in terms of graphs, and in order to do this it is useful to define the notion of a bipartite graph. We say that a graph \( G = (V, E) \) is bipartite if its node set \( V \) can be partitioned into sets \( X \) and \( Y \) in such a way that every edge has one end in \( X \) and the other end in \( Y \). A bipartite graph is pictured in Figure 1.5; often, when we want to emphasize a graph's "bipartiteness," we will draw it this way, with the nodes in \( X \) and \( Y \) in two parallel columns. But notice, for example, that the two graphs in Figure 1.3 are also bipartite.

Now, in the problem of finding a stable matching, matchings were built from pairs of men and women. In the case of bipartite graphs, the edges are pairs of nodes, so we say that a matching in a graph \( G = (V, E) \) is a set of edges \( M \subseteq E \) with the property that each node appears in at most one edge of \( M \). \( M \) is a perfect matching if every node appears in exactly one edge of \( M \).

To see that this does capture the same notion we encountered in the Stable Matching Problem, consider a bipartite graph \( G' \) with a set \( X \) of \( n \) men, a set \( Y \) of \( n \) women, and an edge from every node in \( X \) to every node in \( Y \). Then the matchings and perfect matchings in \( G' \) are precisely the matchings and perfect matchings among the set of men and women.

In the Stable Matching Problem, we added preferences to this picture. Here, we do not consider preferences; but the nature of the problem in arbitrary bipartite graphs adds a different source of complexity: there is not necessarily an edge from every \( x \in X \) to every \( y \in Y \), so the set of possible matchings has quite a complicated structure. In other words, it is as though only certain pairs of men and women are willing to be paired off, and we want to figure out how to pair off many people in a way that is consistent with this. Consider, for example, the bipartite graph \( G \) in Figure 1.5: there are many matchings in \( G \), but there is only one perfect matching. (Do you see it?)

Matchings in bipartite graphs can model situations in which objects are being assigned to other objects. Thus, the nodes in \( X \) can represent jobs, the nodes in \( Y \) can represent machines, and an edge \((x_i, y_j)\) can indicate that machine \( y_j \) is capable of processing job \( x_i \). A perfect matching is then a way of assigning each job to a machine that can process it, with the property that each machine is assigned exactly one job. In the spring, computer science departments across the country are often seen pondering a bipartite graph in which \( X \) is the set of professors in the department, \( Y \) is the set of offered courses, and an edge \((x_i, y_j)\) indicates that professor \( x_i \) is capable of teaching course \( y_j \). A perfect matching in this graph consists of an assignment of each professor to a course that he or she can teach, in such a way that every course is covered.

Thus the Bipartite Matching Problem is the following: Given an arbitrary bipartite graph \( G \), find a matching of maximum size. If \(|X| = |Y| = n\), then there is a perfect matching if and only if the maximum matching has size \( n \). We will find that the algorithmic techniques discussed earlier do not seem adequate
for providing an efficient algorithm for this problem. There is, however, a very
elegant and efficient algorithm to find a maximum matching; it inductively
builds up larger and larger matchings, selectively backtracking along the way.
This process is called augmentation, and it forms the central component in
a large class of efficiently solvable problems called network flow problems.

Independent Set

Now let’s talk about an extremely general problem, which includes most of
these earlier problems as special cases. Given a graph $G = (V, E)$, we say
a set of nodes $S \subseteq V$ is independent if no two nodes in $S$ are joined by an
edge. The Independent Set Problem is, then, the following: Given $G$, find an
independent set that is as large as possible. For example, the maximum size of
an independent set in the graph in Figure 1.6 is four, achieved by the four-node
independent set $\{1, 4, 5, 6\}$. The Independent Set Problem encodes any situation in which you are
trying to choose from among a collection of objects and there are pairwise
conflicts among some of the objects. Say you have $n$ friends, and some pairs
of them don’t get along. How large a group of your friends can you invite to
dinner if you don’t want any interpersonal tensions? This is simply the largest
independent set in the graph whose nodes are your friends, with an edge
between each conflicting pair.

Interval Scheduling and Bipartite Matching can both be encoded as special
cases of the Independent Set Problem. For Interval Scheduling, define a graph
$G = (V, E)$ in which the nodes are the intervals and there is an edge between
each pair of them that overlap; the independent sets in $G$ are then just the
compatible subsets of intervals. Encoding Bipartite Matching as a special case
of Independent Set is a little trickier to see. Given a bipartite graph $G' = (V', E')$,
the objects being chosen are edges, and the conflicts arise between two edges
that share an end. (These, indeed, are the pairs of edges that cannot belong
to a common matching.) So we define a graph $G = (V, E)$ in which the node
set $V$ is equal to the edge set $E'$ of $G'$. We define an edge between each pair
of elements in $V$ that correspond to edges of $G'$ with a common end. We can
now check that the independent sets of $G$ are precisely the matchings of $G'$.
While it is not complicated to check this, it takes a little concentration to deal
with this type of “edges-to-nodes, nodes-to-edges” transformation.\(^2\)

Given the generality of the Independent Set Problem, an efficient algorithm
to solve it would be quite impressive. It would have to implicitly contain
algorithms for Interval Scheduling, Bipartite Matching, and a host of other
natural optimization problems.

The current status of Independent Set is this: no efficient algorithm is
known for the problem, and it is conjectured that no such algorithm exists.
The obvious brute-force algorithm would try all subsets of the nodes, checking
each to see if it is independent, and then recording the largest one encountered.
It is possible that this is close to the best we can do on this problem. We will
see later in the book that Independent Set is one of a large class of problems
that are termed NP-complete. No efficient algorithm is known for any of them;
but they are all equivalent in the sense that a solution to any one of them
would imply, in a precise sense, a solution to all of them.

Here’s a natural question: Is there anything good we can say about the
complexity of the Independent Set Problem? One positive thing is the following:
If we have a graph $G$ on 1,000 nodes, and we want to convince you that it
contains an independent set $S$ of size 100, then it’s quite easy. We simply
show you the graph $G$, circle the nodes of $S$ in red, and let you check that
no two of them are joined by an edge. So there really seems to be a great
difference in difficulty between checking that something is a large independent
set and actually finding a large independent set. This may look like a very basic
observation—and it is—but it turns out to be crucial in understanding this class
of problems. Furthermore, as we’ll see next, it’s possible for a problem to be
so hard that there isn’t even an easy way to “check” solutions in this sense.

Competitive Facility Location

Finally, we come to our fifth problem, which is based on the following two-
player game. Consider two large companies that operate café franchises across
the country—let’s call them JavaPlanet and Queequeg’s Coffee—and they are
currently competing for market share in a geographic area. First JavaPlanet
opens a franchise; then Queequeg’s Coffee opens a franchise; then JavaPlanet;
then Queequeg’s; and so on. Suppose they must deal with zoning regulations
that require no two franchises be located too close together, and each is trying
to make its locations as convenient as possible. Who will win?

Let’s make the rules of this “game” more concrete. The geographic region
in question is divided into $n$ zones, labeled $1, 2, \ldots, n$. Each zone $i$ has a

\(^2\) For those who are curious, we note that not every instance of the Independent Set Problem can arise
in this way from Interval Scheduling or from Bipartite Matching; the full Independent Set Problem
really is more general. The graph in Figure 1.3(a) cannot arise as the “conflict graph” in an instance of
Interval Scheduling, and the graph in Figure 1.3(b) cannot arise as the “conflict graph” in an instance of
Bipartite Matching.
value $b_i$, which is the revenue obtained by either of the companies if it opens a franchise there. Finally, certain pairs of zones $(i, j)$ are adjacent, and local zoning laws prevent two adjacent zones from each containing a franchise, regardless of which company owns them. (They also prevent two franchises of being opened in the same zone.) We model these conflicts via a graph $G = (V, E)$, where $V$ is the set of zones, and $(i, j)$ is an edge in $E$ if the zones $i$ and $j$ are adjacent. The zoning requirement then says that the full set of franchises opened must form an independent set in $G$.

Thus our game consists of two players, $P_1$ and $P_2$, alternately selecting nodes in $G$, with $P_1$ moving first. At all times, the set of all selected nodes must form an independent set in $G$. Suppose that player $P_2$ has a target bound $B$, and we want to know: is there a strategy for $P_2$ so that no matter how $P_1$ plays, $P_2$ will be able to select a set of nodes with a total value of at least $B$?
We will call this an instance of the Competitive Facility Location Problem.

Consider, for example, the instance pictured in Figure 1.7, and suppose that $P_2$'s target bound is $B = 20$. Then $P_2$ does have a winning strategy. On the other hand, if $B = 25$, then $P_2$ does not.

One can work this out by looking at the figure for a while; but it requires some amount of case-checking of the form, “If $P_1$ goes here, then $P_2$ will go there; but if $P_1$ goes over there, then $P_2$ will go here. . . . ” And this appears to be intrinsic to the problem: not only is it computationally difficult to determine whether $P_2$ has a winning strategy; on a reasonably sized graph, it would even be hard for us to convince you that $P_2$ has a winning strategy. There does not seem to be a short proof we could present; rather, we’d have to lead you on a lengthy case-by-case analysis of the set of possible moves.

This is in contrast to the Independent Set Problem, where we believe that finding a large solution is hard but checking a proposed large solution is easy. This contrast can be formalized in the class of PSPACE-complete problems, of which Competitive Facility Location is an example. PSPACE-complete problems are believed to be strictly harder than NP-complete problems, and this conjectured lack of short “proofs” for their solutions is one indication of this greater hardness. The notion of PSPACE-completeness turns out to capture a large collection of problems involving game-playing and planning; many of these are fundamental issues in the area of artificial intelligence.

---

**Solved Exercises**

**Solved Exercise 1**

Consider a town with $n$ men and $n$ women seeking to get married to one another. Each man has a preference list that ranks all the women, and each woman has a preference list that ranks all the men.

The set of all $2n$ people is divided into two categories: good people and bad people. Suppose that for some number $k$, $1 \leq k \leq n-1$, there are $k$ good men and $k$ good women; thus there are $n-k$ bad men and $n-k$ bad women.

Everyone would rather marry any good person than any bad person. Formally, each preference list has the property that it ranks each good person of the opposite gender higher than each bad person of the opposite gender: its first $k$ entries are the good people (of the opposite gender) in some order, and its next $n-k$ are the bad people (of the opposite gender) in some order.

Show that in every stable matching, every good man is married to a good woman.

**Solution** A natural way to get started thinking about this problem is to assume the claim is false and try to work toward obtaining a contradiction. What would it mean for the claim to be false? There would exist some stable matching $M$ in which a good man $m$ was married to a bad woman $w$.

Now, let’s consider what the other pairs in $M$ look like. There are $k$ good men and $k$ good women. Could it be the case that every good woman is married to a good man in this matching $M$? No: one of the good men (namely, $m$) is already married to a bad woman, and that leaves only $k-1$ other good men. So even if all of them were married to good women, that would still leave some good woman who is married to a bad man.

Let $w'$ be such a good woman, who is married to a bad man. It is now easy to identify an instability in $M$: consider the pair $(m, w')$. Each is good, but is married to a bad partner. Thus, each of $m$ and $w'$ prefers the other to their current partner, and hence $(m, w')$ is an instability. This contradicts our assumption that $M$ is stable, and hence concludes the proof.

**Solved Exercise 2**

We can think about a generalization of the Stable Matching Problem in which certain man-woman pairs are explicitly forbidden. In the case of employers and applicants, we could imagine that certain applicants simply lack the necessary qualifications or certifications, and so they cannot be employed at certain companies, however desirable they may seem. Using the analogy to marriage between men and women, we have a set $M$ of $n$ men, a set $W$ of $n$ women,
and a set $F \subseteq M \times W$ of pairs who are simply not allowed to get married. Each man $m$ ranks all the women $w$ for which $(m, w) \notin F$, and each woman $w'$ ranks all the men $m'$ for which $(m', w') \notin F$.

In this more general setting, we say that a matching $S$ is stable if it does not exhibit any of the following types of instability.

(i) There are two pairs $(m, w)$ and $(m', w')$ in $S$ with the property that $(m, w') \notin F$, $m$ prefers $w'$ to $w$, and $w'$ prefers $m$ to $m'$. (The usual kind of instability.)

(ii) There is a pair $(m, w) \in S$, and a man $m'$, so that $m'$ is not part of any pair in the matching, $(m', w) \notin F$, and $w$ prefers $m'$ to $m$. (A single man is more desirable and not forbidden.)

(iii) There is a pair $(m, w) \in S$, and a woman $w'$, so that $w'$ is not part of any pair in the matching, $(m, w') \notin F$, and $m$ prefers $w'$ to $w$. (A single woman is more desirable and not forbidden.)

(iv) There is a man $m$ and a woman $w$, neither of whom is part of any pair in the matching, so that $(m, w) \notin F$. (There are two single people with nothing preventing them from getting married to each other.)

Note that under these more general definitions, a stable matching need not be a perfect matching.

Now we can ask: For every set of preference lists and every set of forbidden pairs, is there always a stable matching? Resolve this question by doing one of the following two things: (a) give an algorithm that, for any set of preference lists and forbidden pairs, produces a stable matching; or (b) give an example of a set of preference lists and forbidden pairs for which there is no stable matching.

**Solution** The Gale-Shapley algorithm is remarkably robust to variations on the Stable Matching Problem. So, if you’re faced with a new variation of the problem and can’t find a counterexample to stability, it’s often a good idea to check whether a direct adaptation of the G-S algorithm will in fact produce stable matchings.

That turns out to be the case here. We will show that there is always a stable matching, even in this more general model with forbidden pairs, and we will do this by adapting the G-S algorithm. To do this, let’s consider why the original G-S algorithm can’t be used directly. The difficulty, of course, is that the G-S algorithm doesn’t know anything about forbidden pairs, and so the condition in the While loop,

While there is a man $m$ who is free and hasn’t proposed to every woman,

won’t work: we don’t want $m$ to propose to a woman $w$ for which the pair $(m, w)$ is forbidden.

Thus, let’s consider a variation of the G-S algorithm in which we make only one change: we modify the While loop to say,

While there is a man $m$ who is free and hasn’t proposed to every woman $w$ for which $(m, w) \notin F$.

Here is the algorithm in full.

Initially all $m \in M$ and $w \in W$ are free.

While there is a man $m$ who is free and hasn’t proposed to every woman $w$ for which $(m, w) \notin F$.

Choose such a man $m$.

Let $w$ be the highest-ranked woman in $m$’s preference list to which $m$ has not yet proposed.

If $w$ is free then

$(m, w)$ become engaged.

Else $w$ is currently engaged to $m'$.

If $w$ prefers $m'$ to $m$ then

$m$ remains free.

Else $w$ prefers $m$ to $m'$.

$(m, w)$ become engaged.

$m'$ becomes free.

Endif.

Endif.

Endwhile.

Return the set $S$ of engaged pairs.

We now prove that this yields a stable matching, under our new definition of stability.

To begin with, facts (1.1), (1.2), and (1.3) from the text remain true (in particular, the algorithm will terminate in at most $n^2$ iterations). Also, we don’t have to worry about establishing that the resulting matching $S$ is perfect (indeed, it may not be). We also notice an additional pairs of facts. If $m$ is a man who is not part of a pair in $S$, then $m$ must have proposed to every nonforbidden woman; and if $w$ is a woman who is not part of a pair in $S$, then it must be that no man ever proposed to $w$.

Finally, we need only show

(1.9) There is no instability with respect to the returned matching $S$. 

Proof. Our general definition of instability has four parts: This means that we have to make sure that none of the four bad things happens.

First, suppose there is an instability of type (i), consisting of pairs \((m, w)\) and \((m', w')\) in \(S\) with the property that \((m, w') \notin F\), \(m\) prefers \(w'\) to \(w\), and \(w'\) prefers \(m\) to \(m'\). It follows that \(m\) must have proposed to \(w'\); so \(w'\) rejected \(m\), and thus she prefers her final partner to \(m\)—a contradiction.

Next, suppose there is an instability of type (ii), consisting of a pair \((m, w) \in S\), and a man \(m'\), so that \(m'\) is not part of any pair in the matching, \((m', w) \notin F\), and \(w\) prefers \(m'\) to \(m\). Then \(m'\) must have proposed to \(w\) and been rejected; again, it follows that \(w\) prefers her final partner to \(m'\)—a contradiction.

Third, suppose there is an instability of type (iii), consisting of a pair \((m, w) \in S\), and a woman \(w'\), so that \(w'\) is not part of any pair in the matching, \((m, w') \notin F\), and \(m\) prefers \(w'\) to \(w\). Then no man proposed to \(w'\) at all; in particular, \(m\) never proposed to \(w'\), and so he must prefer \(w\) to \(w'\)—a contradiction.

Finally, suppose there is an instability of type (iv), consisting of a man \(m\) and a woman \(w\), neither of which is part of any pair in the matching, so that \((m, w) \notin F\). But for \(m\) to be single, he must have proposed to every nonforbidden woman; in particular, he must have proposed to \(w\), which means she would no longer be single—a contradiction.

**Exercises**

1. Decide whether you think the following statement is true or false. If it is true, give a short explanation. If it is false, give a counterexample.

   True or false? In every instance of the Stable Matching Problem, there is a stable matching containing a pair \((m, w)\) such that \(m\) is ranked first on the preference list of \(w\) and \(w\) is ranked first on the preference list of \(m\).

2. Decide whether you think the following statement is true or false. If it is true, give a short explanation. If it is false, give a counterexample.

   True or false? Consider an instance of the Stable Matching Problem in which there exists a man \(m\) and a woman \(w\) such that \(m\) is ranked first on the preference list of \(w\) and \(w\) is ranked first on the preference list of \(m\). Then in every stable matching \(S\) for this instance, the pair \((m, w)\) belongs to \(S\).

3. There are many other settings in which we can ask questions related to some type of “stability” principle. Here’s one, involving competition between two enterprises.

   Suppose we have two television networks, whom we’ll call \(A\) and \(B\). There are \(n\) prime-time programming slots, and each network has \(n\) TV shows. Each network wants to devise a schedule—an assignment of each show to a distinct slot—so as to attract as much market share as possible.

   Here is the way we determine how well the two networks perform relative to each other, given their schedules. Each show has a fixed rating, which is based on the number of people who watched it last year; we’ll assume that no two shows have exactly the same rating. A network wins a given time slot if the show that it schedules for the time slot has a larger rating than the show the other network schedules for that time slot. The goal of each network is to win as many time slots as possible.

   Suppose in the opening week of the fall season, Network \(A\) reveals a schedule \(S\) and Network \(B\) reveals a schedule \(T\). On the basis of this pair of schedules, each network wins certain time slots, according to the rule above. We’ll say that the pair of schedules \((S, T)\) is stable if neither network can unilaterally change its own schedule and win more time slots. That is, there is no schedule \(S'\) such that Network \(A\) wins more slots with the pair \((S', T)\) than it did with the pair \((S, T)\); and symmetrically, there is no schedule \(T'\) such that Network \(B\) wins more slots with the pair \((S, T')\) than it did with the pair \((S, T)\).

   The analogue of Gale and Shapley’s question for this kind of stability is the following: For every set of TV shows and ratings, is there always a stable pair of schedules? Resolve this question by doing one of the following two things:

   (a) give an algorithm that, for any set of TV shows and associated ratings, produces a stable pair of schedules; or

   (b) give an example of a set of TV shows and associated ratings for which there is no stable pair of schedules.

4. Gale and Shapley published their paper on the Stable Matching Problem in 1962; but a version of their algorithm had already been in use for ten years by the National Resident Matching Program, for the problem of assigning medical residents to hospitals.

   Basically, the situation was the following. There were \(m\) hospitals, each with a certain number of available positions for hiring residents. There were \(n\) medical students graduating in a given year, each interested in joining one of the hospitals. Each hospital had a ranking of the students in order of preference, and each student had a ranking of the hospitals in order of preference. We will assume that there were more students graduating than there were slots available in the \(m\) hospitals.
The interest, naturally, was in finding a way of assigning each student to at most one hospital, in such a way that all available positions in all hospitals were filled. (Since we are assuming a surplus of students, there would be some students who do not get assigned to any hospital.)

We say that an assignment of students to hospitals is **stable** if neither of the following situations arises.

- First type of instability: There are students \( s \) and \( s' \), and a hospital \( h \), so that
  - \( s \) is assigned to \( h \), and
  - \( s' \) is assigned to no hospital, and
  - \( h \) prefers \( s' \) to \( s \).

- Second type of instability: There are students \( s \) and \( s' \), and hospitals \( h \) and \( h' \), so that
  - \( s \) is assigned to \( h \), and
  - \( s' \) is assigned to \( h' \), and
  - \( h \) prefers \( s' \) to \( s \), and
  - \( s' \) prefers \( h \) to \( h' \).

So we basically have the Stable Matching Problem, except that (i) hospitals generally want more than one resident, and (ii) there is a surplus of medical students.

Show that there is always a stable assignment of students to hospitals, and give an algorithm to find one.

5. The Stable Matching Problem, as discussed in the text, assumes that all men and women have a fully ordered list of preferences. In this problem we will consider a version of the problem in which men and women can be **indifferent** between certain options. As before we have a set \( M \) of \( n \) men and a set \( W \) of \( n \) women. Assume each man and each woman ranks the members of the opposite gender, but now we allow ties in the ranking. For example (with \( n = 4 \)), a woman could say that \( m_1 \) is ranked in first place; second place is a tie between \( m_2 \) and \( m_3 \) (she has no preference between them); and \( m_4 \) is in last place. We will say that \( w \) prefers \( m \) to \( m' \) if \( m \) is ranked higher than \( m' \) on her preference list (they are not tied).

With indifferences in the rankings, there could be two natural notions for stability. And for each, we can ask about the existence of stable matchings, as follows.

(a) A **strong instability** in a perfect matching \( S \) consists of a man \( m \) and a woman \( w \), such that each of \( m \) and \( w \) prefers the other to their partner in \( S \). Does there always exist a perfect matching with no strong instability? Either give an example of a set of men and women with preference lists for which every perfect matching has a strong instability; or give an algorithm that is guaranteed to find a perfect matching with no strong instability.

(b) A **weak instability** in a perfect matching \( S \) consists of a man \( m \) and a woman \( w \), such that their partners in \( S \) are \( w' \) and \( m' \), respectively, and one of the following holds:
  - \( m \) prefers \( w \) to \( w' \), and \( w \) either prefers \( m \) to \( m' \) or is indifferent between these two choices; or
  - \( w \) prefers \( m \) to \( m' \), and \( m \) either prefers \( w \) to \( w' \) or is indifferent between these two choices.

In other words, the pairing between \( m \) and \( w \) is either preferred by both, or preferred by one while the other is indifferent. Does there always exist a perfect matching with no weak instability? Either give an example of a set of men and women with preference lists for which every perfect matching has a weak instability; or give an algorithm that is guaranteed to find a perfect matching with no weak instability.

6. Peripatetic Shipping Lines, Inc., is a shipping company that owns \( n \) ships and provides service to \( n \) ports. Each of its ships has a **schedule** that says, for each day of the month, which of the ports it's currently visiting, or whether it's out at sea. (You can assume the "month" here has \( m \) days, for some \( m > n \).) Each ship visits each port for exactly one day during the month. For safety reasons, PSL Inc. has the following strict requirement:

1. **No two ships can be in the same port on the same day.**

The company wants to perform maintenance on all the ships this month, via the following scheme. They want to **truncate** each ship's schedule: for each ship \( S_i \), there will be some day when it arrives in its scheduled port and simply remains there for the rest of the month (for maintenance). This means that \( S_i \) will not visit the remaining ports on its schedule (if any) that month, but this is okay. So the **truncation** of \( S_i \)'s schedule will simply consist of its original schedule up to a certain specified day on which it is in a port \( P \); the remainder of the truncated schedule simply has it remain in port \( P \).

Now the company's question to you is the following: Given the schedule for each ship, find a truncation of each so that condition (1) continues to hold: no two ships are ever in the same port on the same day.

Show that such a set of truncations can always be found, and give an algorithm to find them.
Example. Suppose we have two ships and two ports, and the “month” has four days. Suppose the first ship’s schedule is

\[ \text{port P}_1; \text{at sea}; \text{port P}_2; \text{at sea} \]

and the second ship’s schedule is

\[ \text{at sea}; \text{port P}_1; \text{at sea}; \text{port P}_2 \]

Then the (only) way to choose truncations would be to have the first ship remain in port \( P_2 \) starting on day 3, and have the second ship remain in port \( P_1 \) starting on day 2.

7. Some of your friends are working for CluNet, a builder of large communication networks, and they are looking at algorithms for switching in a particular type of input/output crossbar.

Here is the setup. There are \( n \) input wires and \( n \) output wires, each directed from a source to a terminus. Each input wire meets each output wire in exactly one distinct point, at a special piece of hardware called a junction box. Points on the wire are naturally ordered in the direction from source to terminus; for two distinct points \( x \) and \( y \) on the same wire, we say that \( x \) is upstream from \( y \) if \( x \) is closer to the source than \( y \), and otherwise we say \( x \) is downstream from \( y \). The order in which one input wire meets the output wires is not necessarily the same as the order in which another input wire meets the output wires. (And similarly for the orders in which output wires meet input wires.) Figure 1.8 gives an example of such a collection of input and output wires.

Now, here’s the switching component of this situation. Each input wire is carrying a distinct data stream, and this data stream must be switched onto one of the output wires. If the stream of Input \( i \) is switched onto Output \( j \), at junction box \( B \), then this stream passes through all junction boxes upstream from \( B \) on Input \( i \), then through \( B \), then through all junction boxes downstream from \( B \) on Output \( j \). It does not matter which input data stream gets switched onto which output wire, but each input data stream must be switched onto a different output wire. Furthermore—and this is the tricky constraint—no two data streams can pass through the same junction box following the switching operation.

Finally, here’s the problem. Show that for any specified pattern in which the input wires and output wires meet each other (each pair meeting exactly once), a valid switching of the data streams can always be found—one in which each input data stream is switched onto a different output, and no two of the resulting streams pass through the same junction box. Additionally, give an algorithm to find such a valid switching.

8. For this problem, we will explore the issue of truthfulness in the Stable Matching Problem and specifically in the Gale-Shapley algorithm. The basic question is: Can a man or a woman end up better off by lying about his or her preferences? More concretely, we suppose each participant has a true preference order. Now consider a woman \( w \). Suppose \( w \) prefers man \( m \) to \( m’ \), but both \( m \) and \( m’ \) are low on her list of preferences. Can it be the case that by switching the order of \( m \) and \( m’ \) on her list of preferences (i.e., by falsely claiming that she prefers \( m’ \) to \( m \) and running the algorithm with this false preference list, \( w \) will end up with a man \( m’’ \) that she truly prefers to both \( m \) and \( m’ \)? (We can ask the same question for men, but will focus on the case of women for purposes of this question.)

Resolve this question by doing one of the following two things:

(a) Give a proof that, for any set of preference lists, switching the order of a pair on the list cannot improve a woman’s partner in the Gale-Shapley algorithm; or
(b) Give an example of a set of preference lists for which there is a switch that would improve the partner of a woman who switched preferences.

Notes and Further Reading

The Stable Matching Problem was first defined and analyzed by Gale and Shapley (1962); according to David Gale, their motivation for the problem came from a story they had recently read in the New Yorker about the intricacies of the college admissions process (Gale, 2001). Stable matching has grown into an area of study in its own right, covered in books by Gusfield and Irving (1989) and Knuth (1997c). Gusfield and Irving also provide a nice survey of the "parallel" history of the Stable Matching Problem as a technique invented for matching applicants with employers in medicine and other professions.

As discussed in the chapter, our five representative problems will be central to the book's discussions, respectively, of greedy algorithms, dynamic programming, network flow, NP-completeness, and PSPACE-completeness. We will discuss the problems in these contexts later in the book.

Chapter 2
Basics of Algorithm Analysis

Analyzing algorithms involves thinking about how their resource requirements—the amount of time and space they use—will scale with increasing input size. We begin this chapter by talking about how to put this notion on a concrete footing, as making it concrete opens the door to a rich understanding of computational tractability. Having done this, we develop the mathematical machinery needed to talk about the way in which different functions scale with increasing input size, making precise what it means for one function to grow faster than another.

We then develop running-time bounds for some basic algorithms, beginning with an implementation of the Gale-Shapley algorithm from Chapter 1 and continuing to a survey of many different running times and certain characteristic types of algorithms that achieve these running times. In some cases, obtaining a good running-time bound relies on the use of more sophisticated data structures, and we conclude this chapter with a very useful example of such a data structure: priority queues and their implementation using heaps.

2.1 Computational Tractability

A major focus of this book is to find efficient algorithms for computational problems. At this level of generality, our topic seems to encompass the whole of computer science; so what is specific to our approach here?

First, we will try to identify broad themes and design principles in the development of algorithms. We will look for paradigmatic problems and approaches that illustrate, with a minimum of irrelevant detail, the basic approaches to designing efficient algorithms. At the same time, it would be pointless to pursue these design principles in a vacuum, so the problems and
Chapter 2  Basics of Algorithm Analysis

approaches we consider are drawn from fundamental issues that arise throughout computer science, and a general study of algorithms turns out to serve as a nice survey of computational ideas that arise in many areas.

Another property shared by many of the problems we study is their fundamentally discrete nature. That is, like the Stable Matching Problem, they will involve an implicit search over a large set of combinatorial possibilities; and the goal will be to efficiently find a solution that satisfies certain clearly delineated conditions.

As we seek to understand the general notion of computational efficiency, we will focus primarily on efficiency in running time: we want algorithms that run quickly. But it is important that algorithms be efficient in their use of other resources as well. In particular, the amount of space (or memory) used by an algorithm is an issue that will also arise at a number of points in the book, and we will see techniques for reducing the amount of space needed to perform a computation.

Some Initial Attempts at Defining Efficiency

The first major question we need to answer is the following: How should we turn the fuzzy notion of an "efficient" algorithm into something more concrete?

A first attempt at a working definition of efficiency is the following.

Proposed Definition of Efficiency (1): An algorithm is efficient if, when implemented, it runs quickly on real input instances.

Let's spend a little time considering this definition. At a certain level, it's hard to argue with: one of the goals at the bedrock of our study of algorithms is solving real problems quickly. And indeed, there is a significant area of research devoted to the careful implementation and profiling of different algorithms for discrete computational problems.

But there are some crucial things missing from this definition, even if our main goal is to solve real problem instances quickly on real computers. The first is the omission of where, and how well, we implement an algorithm. Even bad algorithms can run quickly when applied to small test cases on extremely fast processors; even good algorithms can run slowly when they are coded sloppily. Also, what is a "real" input instance? We don't know the full range of input instances that will be encountered in practice, and some input instances can be much harder than others. Finally, this proposed definition above does not consider how well, or badly, an algorithm may scale as problem sizes grow to unexpected levels. A common situation is that two very different algorithms will perform comparably on inputs of size 100; multiply the input size tenfold, and one will still run quickly while the other consumes a huge amount of time.

So what we could ask for is a concrete definition of efficiency that is platform-independent, instance-independent, and of predictive value with respect to increasing input sizes. Before focusing on any specific consequences of this claim, we can at least explore its implicit, high-level suggestion: that we need to take a more mathematical view of the situation.

We can use the Stable Matching Problem as an example to guide us. The input has a natural "size" parameter $N$; we could take this to be the total size of the representation of all preference lists, since this is what any algorithm for the problem will receive as input. $N$ is closely related to the other natural parameter in this problem: $n$, the number of men and the number of women. Since there are $2n$ preference lists, each of length $n$, we can view $N = 2n^2$, suppressing more fine-grained details of how the data is represented. In considering the problem, we will seek to describe an algorithm at a high level, and then analyze its running time mathematically as a function of this input size $N$.

Worst-Case Running Times and Brute-Force Search

To begin with, we will focus on analyzing the worst-case running time: we will look for a bound on the largest possible running time the algorithm could have over all inputs of a given size $N$, and see how this scales with $N$. The focus on worst-case performance initially seems quite draconian: what if an algorithm performs well on most instances and just has a few pathological inputs on which it is very slow? This certainly is an issue in some cases, but in general the worst-case analysis of an algorithm has been found to do a reasonable job of capturing its efficiency in practice. Moreover, once we have decided to go the route of mathematical analysis, it is hard to find an effective alternative to worst-case analysis. Average-case analysis—the obvious appealing alternative, in which one studies the performance of an algorithm averaged over "random" instances—can sometimes provide considerable insight, but very often it can also become a quagmire. As we observed earlier, it's very hard to express the full range of input instances that arise in practice, and so attempts to study an algorithm's performance on "random" input instances can quickly devolve into debates over how a random input should be generated: the same algorithm can perform very well on one class of random inputs and very poorly on another. After all, real inputs to an algorithm are generally not being produced from a random distribution, and so average-case analysis risks telling us more about the means by which the random inputs were generated than about the algorithm itself.

So in general we will think about the worst-case analysis of an algorithm's running time. But what is a reasonable analytical benchmark that can tell us whether a running-time bound is impressive or weak? A first simple guide
is by comparison with brute-force search over the search space of possible solutions.

Let's return to the example of the Stable Matching Problem. Even when the size of a Stable Matching input instance is relatively small, the search space it defines is enormous (there are \( n! \) possible perfect matchings between \( n \) men and \( n \) women), and we need to find a matching that is stable. The natural "brute-force" algorithm for this problem would plow through all perfect matchings by enumeration, checking each to see if it is stable. The surprising punchline, in a sense, to our solution of the Stable Matching Problem is that we needed to spend time proportional only to \( N \) in finding a stable matching from among this stupendously large space of possibilities. This was a conclusion we reached at an analytical level. We did not implement the algorithm and try it out on sample preference lists; we reasoned about it mathematically. Yet, at the same time, our analysis indicated how the algorithm could be implemented in practice and gave fairly conclusive evidence that it would be a big improvement over exhaustive enumeration.

This will be a common theme in most of the problems we study: a compact representation, implicitly specifying a giant search space. For most of these problems, there will be an obvious brute-force solution: try all possibilities and see if any one of them works. Not only is this approach almost always too slow to be useful, it is an intellectual cop-out; it provides us with absolutely no insight into the structure of the problem we are studying. And so if there is a common thread in the algorithms we emphasize in this book, it would be the following alternative definition of efficiency.

Proposed Definition of Efficiency (2): An algorithm is efficient if it achieves qualitatively better worst-case performance, at an analytical level, than brute-force search.

This will turn out to be a very useful working definition for us. Algorithms that improve substantially on brute-force search nearly always contain a valuable heuristic idea that makes them work; and they tell us something about the intrinsic structure, and computational tractability, of the underlying problem itself.

But if there is a problem with our second working definition, it is vagueness. What do we mean by "qualitatively better performance?" This suggests that we consider the actual running time of algorithms more carefully, and try to quantify what a reasonable running time would be.

Polynomial Time as a Definition of Efficiency

When people first began analyzing discrete algorithms mathematically—a thread of research that began gathering momentum through the 1960s—a consensus began to emerge on how to quantify the notion of a "reasonable" running time. Search spaces for natural combinatorial problems tend to grow exponentially in the size \( N \) of the input; if the input size increases by one, the number of possibilities increases multiplicatively. We'd like a good algorithm for such a problem to have a better scaling property: when the input size increases by a constant factor—say, a factor of 2—the algorithm should only slow down by some constant factor \( C \).

Arithmetically, we can formulate this scaling behavior as follows. Suppose an algorithm has the following property: There are absolute constants \( c > 0 \) and \( d > 0 \) so that on every input instance of size \( N \), its running time is bounded by \( cn^d \) primitive computational steps. (In other words, its running time is at most proportional to \( N^d \).) For now, we will remain deliberately vague on what we mean by the notion of a "primitive computational step"—but it can be easily formalized in a model where each step corresponds to a single assembly-language instruction on a standard processor, or one line of a standard programming language such as C or Java. In any case, if this running-time bound holds, for some \( c \) and \( d \), then we say that the algorithm has a polynomial running time, or that it is a polynomial-time algorithm. Note that any polynomial-time bound has the scaling property we're looking for. If the input size increases from \( N \) to \( 2N \), the bound on the running time increases from \( cn^d \) to \( c(2N)^d = c \cdot 2^d N^d \), which is a slow-down by a factor of \( 2^d \). Since \( d \) is a constant, so is \( 2^d \); of course, as one might expect, lower-degree polynomials exhibit better scaling behavior than higher-degree polynomials.

From this notion, and the intuition expressed above, emerges our third attempt at a working definition of efficiency.

Proposed Definition of Efficiency (3): An algorithm is efficient if it has a polynomial running time.

Where our previous definition seemed overly vague, this one seems much too prescriptive. Wouldn't an algorithm with running time proportional to \( n^{100} \)—and hence polynomial—be hopelessly inefficient? Wouldn't we be relatively pleased with a nonpolynomial running time of \( n^{1 + 0.02 \log \log n} \)? The answers are, of course, "yes" and "yes." And indeed, however much one may try to abstractly motivate the definition of efficiency in terms of polynomial time, a primary justification for it is this: It really works. Problems for which polynomial-time algorithms exist almost invariably turn out to have algorithms with running times proportional to very moderately growing polynomials like \( n, n \log n, n^2, \) or \( n^3 \). Conversely, problems for which no polynomial-time algorithm is known tend to be very difficult in practice. There are certainly exceptions to this principle in both directions: there are cases, for example, in
Chapter 2  Basics of Algorithm Analysis

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{17} years, we simply record the algorithm as taking a very long time.

<table>
<thead>
<tr>
<th>n log_2 n</th>
<th>n^2</th>
<th>n^3</th>
<th>1.5^n</th>
<th>2^n</th>
<th>n!</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>4 sec</td>
</tr>
<tr>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>18 min</td>
<td>10^{25} years</td>
</tr>
<tr>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>11 min</td>
<td>36 years</td>
<td>very long</td>
</tr>
<tr>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>12,892 years</td>
<td>10^{17} years</td>
<td>very long</td>
</tr>
<tr>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>18 min</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>&lt; 1 sec</td>
<td>2 min</td>
<td>12 days</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>2 sec</td>
<td>3 hours</td>
<td>32 years</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>20 sec</td>
<td>12 days</td>
<td>31,710 years</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
</tbody>
</table>

which an algorithm with exponential worst-case behavior generally runs well
on the kinds of instances that arise in practice; and there are also cases where
the best polynomial-time algorithm for a problem is completely impractical
due to large constants or a high exponent on the polynomial bound. All this
serves to reinforce the point that our emphasis on worst-case, polynomial-time
bounds is only an abstraction of practical situations. But overwhelmingly, the
concrete mathematical definition of polynomial time has turned out to corre-
spond surprisingly well in practice to what we observe about the efficiency of
algorithms, and the tractability of problems, in real life.

One further reason why the mathematical formalism and the empirical
evidence seem to line up well in the case of polynomial-time solvability is that
the gulf between the growth rates of polynomial and exponential functions
is enormous. Suppose, for example, that we have a processor that executes
a million high-level instructions per second, and we have algorithms with
running-time bounds of \( n \), \( n \log_2 n \), \( n^2 \), \( n^3 \), \( 1.5^n \), \( 2^n \), and \( n! \). In Table 2.1,
we show the running times of these algorithms (in seconds, minutes, days,
or years) for inputs of size \( n = 10, 30, 50, 100, 1,000, 10,000, 100,000 \), and 1,000,000.

There is a final, fundamental benefit to making our definition of efficiency
so specific: it becomes negatable. It becomes possible to express the notion
that there is no efficient algorithm for a particular problem. In a sense, being
able to do this is a prerequisite for turning our study of algorithms into
good science, for it allows us to ask about the existence or nonexistence
of efficient algorithms as a well-defined question. In contrast, both of our
previous definitions were completely subjective, and hence limited the extent
to which we could discuss certain issues in concrete terms.

In particular, the first of our definitions, which was tied to the specific
implementation of an algorithm, turned efficiency into a moving target: as
processor speeds increase, more and more algorithms fall under this notion of
efficiency. Our definition in terms of polynomial time is much more an absolute
notion; it is closely connected with the idea that each problem has an intrinsic
level of computational tractability: some admit efficient solutions, and others
do not.

2.2 Asymptotic Order of Growth

Our discussion of computational tractability has turned out to be intrinsically
based on our ability to express the notion that an algorithm's worst-case
running time on inputs of size \( n \) grows at a rate that is at most proportional to
some function \( f(n) \). The function \( f(n) \) then becomes a bound on the running
time of the algorithm. We now discuss a framework for talking about this
concept.

We will mainly express algorithms in the pseudo-code style that we used
for the Gale-Shapley algorithm. At times we will need to become more formal,
but this style of specifying algorithms will be completely adequate for most
purposes. When we provide a bound on the running time of an algorithm,
we will generally be counting the number of such pseudo-code steps that
are executed; in this context, one step will consist of assigning a value to a
variable, looking up an entry in an array, following a pointer, or performing
an arithmetic operation on a fixed-size integer.

When we seek to say something about the running time of an algorithm on
inputs of size \( n \), one thing we could aim for would be a very concrete statement
such as, "On any input of size \( n \), the algorithm runs for at most 1.62n^2 +
3.5n + 8 steps." This may be an interesting statement in some contexts, but as
a general goal there are several things wrong with it. First, getting such a precise
bound may be an exhausting activity, and more detail than we wanted anyway.
Second, because our ultimate goal is to identify broad classes of algorithms that
have similar behavior, we'd actually like to classify running times at a coarser
level of granularity so that similarities among different algorithms, and among
different problems, show up more clearly. And finally, extremely detailed
statements about the number of steps an algorithm executes are often—in
a strong sense—meaningless. As just discussed, we will generally be counting
steps in a pseudo-code specification of an algorithm that resembles a high-
level programming language. Each one of these steps will typically unfold
into some fixed number of primitive steps when the program is compiled into
an intermediate representation, and then into some further number of steps depending on the particular architecture being used to do the computing. So the most we can safely say is that as we look at different levels of computational abstraction, the notion of a “step” may grow or shrink by a constant factor—for example, if it takes 25 low-level machine instructions to perform one operation in our high-level language, then our algorithm that took at most \(1.62n^2 + 3.5n + 8\) steps can also be viewed as taking \(40.5n^2 + 87.5n + 200\) steps when we analyze it at a level that is closer to the actual hardware.

\(O, \Omega, \text{ and } \Theta\)

For all these reasons, we want to express the growth rate of running times and other functions in a way that is insensitive to constant factors and low-order terms. In other words, we’d like to be able to take a running time like the one we discussed above, \(1.62n^2 + 3.5n + 8\), and say that it grows like \(n^2\), up to constant factors. We now discuss a precise way to do this.

**Asymptotic Upper Bounds** Let \(T(n)\) be a function—say, the worst-case running time of a certain algorithm on an input of size \(n\). (We will assume that all the functions we talk about here take nonnegative values.) Given another function \(f(n)\), we say that \(T(n) = O(f(n))\) (read as “\(T(n)\) is order \(f(n)\)”) if, for sufficiently large \(n\), the function \(T(n)\) is bounded above by a constant multiple of \(f(n)\). We will also sometimes write this as \(T(n) = O(f(n))\). More precisely, \(T(n) = O(f(n))\) if there exist constants \(c > 0\) and \(n_0 \geq 0\) so that for all \(n \geq n_0\), we have \(T(n) \leq c \cdot f(n)\). In this case, we will say that \(T\) is asymptotically upper-bounded by \(f\). It is important to note that this definition requires a constant \(c\) to exist that works for all \(n\); in particular, \(c\) cannot depend on \(n\).

As an example of how this definition lets us express upper bounds on running times, consider an algorithm whose running time (as in the earlier discussion) has the form \(T(n) = pn^2 + qn + r\) for positive constants \(p, q,\) and \(r\). We’d like to claim that any such function is \(O(n^2)\). To see why, we notice that for all \(n \geq 1\), we have \(qn \leq qn^2\), and \(r \leq rn^2\). So we can write

\[
T(n) = pn^2 + qn + r \leq pn^2 + qn^2 + m^2 = (p + q + r)n^2
\]

for all \(n \geq 1\). This inequality is exactly what the definition of \(O(\cdot)\) requires: \(T(n) \leq cn^2\), where \(c = p + q + r\).

Note that \(O(\cdot)\) expresses only an upper bound, not the exact growth rate of the function. For example, just as we claimed that the function \(T(n) = pn^2 + qn + r\) is \(O(n^2)\), it’s also correct to say that it’s \(O(n^3)\). Indeed, we just argued that \(T(n) \leq (p + q + r)n^2\), and since we also have \(n^2 \leq n^3\), we can conclude that \(T(n) \leq (p + q + r)n^3\) as the definition of \(O(n^3)\) requires. The fact that a function can have many upper bounds is not just a trick of the notation; it shows up in the analysis of running times as well. There are cases where an algorithm has been proved to have running time \(O(n^3)\); some years pass, people analyze the same algorithm more carefully, and they show that in fact its running time is \(O(n^2)\). There was nothing wrong with the first result; it was a correct upper bound. It’s simply that it wasn’t the “tightest” possible running time.

**Asymptotic Lower Bounds** There is a complementary notation for lower bounds. Often when we analyze an algorithm—say we have just proven that its worst-case running time \(T(n)\) is \(O(n^2)\)—we want to show that this upper bound is the best one possible. To do this, we want to express the notion that for arbitrarily large input sizes \(n\), the function \(T(n)\) is at least a constant multiple of some specific function \(f(n)\). (In this example, \(f(n)\) happens to be \(n^2\).) Thus, we say that \(T(n) = \Omega(f(n))\) (also written \(T(n) = \Omega(f(n))\)) if there exist constants \(\epsilon > 0\) and \(n_0 \geq 0\) so that for all \(n \geq n_0\), we have \(T(n) \geq \epsilon \cdot f(n)\). By analogy with \(O(\cdot)\) notation, we will refer to \(T\) in this case as being asymptotically lower-bounded by \(f\). Again, note that the constant \(\epsilon\) must be fixed, independent of \(n\).

This definition works just like \(O(\cdot)\), except that we are bounding the function \(T(n)\) from below, rather than from above. For example, returning to the function \(T(n) = pn^2 + qn + r\), where \(p, q,\) and \(r\) are positive constants, let’s claim that \(T(n) = \Omega(n^2)\). Whereas establishing the upper bound involved “inflating” the terms in \(T(n)\) until it looked like a constant times \(n^2\), now we need to do the opposite: we need to reduce the size of \(T(n)\) until it looks like a constant times \(n^2\). It is not hard to do this; for all \(n \geq 0\), we have

\[
T(n) = pn^2 + qn + r \geq pn^2
\]

which meets what is required by the definition of \(\Omega(\cdot)\) with \(\epsilon = p > 0\).

Just as we discussed the notion of “tighter” and “weaker” upper bounds, the same issue arises for lower bounds. For example, it is correct to say that our function \(T(n) = pn^2 + qn + r\) is \(\Omega(n)\), since \(T(n) \geq pn^2 \geq pn\).

**Asymptotically Tight Bounds** If we can show that a running time \(T(n)\) is both \(O(f(n))\) and \(\Omega(f(n))\), then in a natural sense we’ve found the “right” bound: \(T(n)\) grows exactly like \(f(n)\) to within a constant factor. This, for example, is the conclusion we can draw from the fact that \(T(n) = pn^2 + qn + r\) is both \(O(n^2)\) and \(\Omega(n^2)\).

There is a notation to express this: if a function \(T(n)\) is both \(O(f(n))\) and \(\Omega(f(n))\), we say that \(T(n) = \Theta(f(n))\). In this case, we say that \(f(n)\) is an asymptotically tight bound for \(T(n)\). So, for example, our analysis above shows that \(T(n) = pn^2 + qn + r\) is \(\Theta(n^2)\).

Asymptotically tight bounds on worst-case running times are nice things to find, since they characterize the worst-case performance of an algorithm.
Chapter 2 Basics of Algorithm Analysis

58

precisely up to constant factors. And as the definition of \( \Theta(.) \) shows, one can obtain such bounds by closing the gap between an upper bound and a lower bound. For example, sometimes you will read a (slightly informally phrased) sentence such as "An upper bound of \( f(n) \) has been shown on the worst-case running time of the algorithm, but there is no example known on which the algorithm runs for more than \( n^2 \) steps." This is implicitly an invitation to search for an asymptotically tight bound on the algorithm's worst-case running time.

Sometimes one can also obtain an asymptotically tight bound directly by computing a limit as \( n \) goes to infinity. Essentially, if the ratio of functions \( f(n) \) and \( g(n) \) converges to a positive constant as \( n \) goes to infinity, then \( f(n) = \Theta(g(n)) \).

(2.1) Let \( f \) and \( g \) be two functions that

\[
\lim_{n \to \infty} \frac{f(n)}{g(n)} = c > 0.
\]

exists and is equal to some number \( c > 0 \). Then \( f(n) = \Theta(g(n)) \).

Proof. We will use the fact that the limit exists and is positive to show that \( f(n) = O(g(n)) \) and \( f(n) = \Omega(g(n)) \), as required by the definition of \( \Theta(.) \).

Since

\[
\lim_{n \to \infty} \frac{f(n)}{g(n)} = c > 0,
\]

it follows from the definition of a limit that there is some \( n_0 \) beyond which the ratio is always between \( \frac{1}{2}c \) and \( 2c \). Thus, \( f(n) \leq 2cg(n) \) for all \( n \geq n_0 \), which implies that \( f(n) = O(g(n)) \); and \( f(n) \geq \frac{1}{2}cg(n) \) for all \( n \geq n_0 \), which implies that \( f(n) = \Omega(g(n)) \). \( \blacksquare \)

Properties of Asymptotic Growth Rates

Having seen the definitions of \( O \), \( \Omega \), and \( \Theta \), it is useful to explore some of their basic properties.

Transitivity A first property is transitivity: if a function \( f \) is asymptotically upper-bounded by a function \( g \), and if \( g \) in turn is asymptotically upper-bounded by a function \( h \), then \( f \) is asymptotically upper-bounded by \( h \). A similar property holds for lower bounds. We write this more precisely as follows.

(2.2)

(a) If \( f = O(g) \) and \( g = O(h) \), then \( f = O(h) \).

(b) If \( f = \Omega(g) \) and \( g = \Omega(h) \), then \( f = \Omega(h) \).

Proof. We'll prove part (a) of this claim; the proof of part (b) is very similar.

For (a), we're given that for some constants \( c \) and \( n_0 \), we have \( f(n) \leq cg(n) \) for all \( n \geq n_0 \). Also, for some (potentially different) constants \( c' \) and \( n_0' \), we have \( g(n) \leq c'h(n) \) for all \( n \geq n_0' \). So consider any number \( n \) that is at least as large as both \( n_0 \) and \( n_0' \). We have \( f(n) \leq cg(n) \leq c'c'h(n) \), and so \( f(n) \leq c'c'h(n) \) for all \( n \geq \max(n_0, n_0') \). This latter inequality is exactly what is required for showing that \( f = O(h) \). \( \blacksquare \)

Combining parts (a) and (b) of (2.2), we can obtain a similar result for asymptotically tight bounds. Suppose we know that \( f = \Theta(g) \) and that \( g = \Theta(h) \). Then since \( f = O(g) \) and \( g = O(h) \), we know from part (a) that \( f = O(h) \); since \( f = \Omega(g) \) and \( g = \Omega(h) \), we know from part (b) that \( f = \Omega(h) \).

It follows that \( f = \Theta(h) \). Thus we have shown

(2.3) If \( f = \Theta(g) \) and \( g = \Theta(h) \), then \( f = \Theta(h) \).

Sums of Functions It is also useful to have results that quantify the effect of adding two functions. First, if we have an asymptotic upper bound that applies to each of two functions \( f \) and \( g \), then it applies to their sum.

(2.4) Suppose that \( f \) and \( g \) are two functions such that for some other function \( h \), we have \( f = O(h) \) and \( g = O(h) \). Then \( f + g = O(h) \).

Proof. We're given that for some constants \( c \) and \( n_0 \), we have \( f(n) \leq ch(n) \) for all \( n \geq n_0 \). Also, for some (potentially different) constants \( c' \) and \( n_0' \), we have \( g(n) \leq c'h(n) \) for all \( n \geq n_0' \). So consider any number \( n \) that is at least as large as both \( n_0 \) and \( n_0' \). We have \( f(n) + g(n) \leq ch(n) + c'h(n) \). Thus \( f(n) + g(n) \leq (c + c')h(n) \) for all \( n \geq \max(n_0, n_0') \), which is exactly what is required for showing that \( f + g = O(h) \). \( \blacksquare \)

There is a generalization of this to sums of a fixed constant number of functions \( k \), where \( k \) may be larger than two. The result can be stated precisely as follows; we omit the proof, since it is essentially the same as the proof of (2.4), adapted to sums consisting of \( k \) terms rather than just two.

(2.5) Let \( k \) be a fixed constant, and let \( f_1, f_2, \ldots, f_k \) and \( h \) be functions such that \( f_i = O(h) \) for all \( i \). Then \( f_1 + f_2 + \cdots + f_k = O(h) \).

There is also a consequence of (2.4) that covers the following kind of situation. It frequently happens that we're analyzing an algorithm with two high-level parts, and it is easy to show that one of the two parts is slower than the other. We'd like to be able to say that the running time of the whole algorithm is asymptotically comparable to the running time of the slow part. Since the overall running time is a sum of two functions (the running times of
Chapter 2 Basics of Algorithm Analysis

the two parts), results on asymptotic bounds for sums of functions are directly relevant.

(2.6) Suppose that f and g are two functions (taking nonnegative values) such that \( g = O(f) \). Then \( f + g = \Theta(f) \). In other words, f is an asymptotically tight bound for the combined function \( f + g \).

**Proof.** Clearly \( f + g = O(f) \), since for all \( n \geq 0 \), we have \( f(n) + g(n) \geq f(n) \). So to complete the proof, we need to show that \( f + g = O(f) \).

But this is a direct consequence of (2.4): we're given the fact that \( g = O(f) \), and also \( f = O(f) \) holds for any function, so by (2.4) we have \( f + g = O(f) \). \( \square \)

This result also extends to the sum of any fixed, constant number of functions: the most rapidly growing among the functions is an asymptotically tight bound for the sum.

**Asymptotic Bounds for Some Common Functions**

There are a number of functions that come up repeatedly in the analysis of algorithms, and it is useful to consider the asymptotic properties of some of the most basic of these: polynomials, logarithms, and exponentials.

**Polynomials** Recall that a polynomial is a function that can be written in the form \( f(n) = a_0 + a_1n + a_2n^2 + \cdots + a_dn^d \) for some integer constant \( d > 0 \), where the final coefficient \( a_d \) is nonzero. This value \( d \) is called the degree of the polynomial. For example, the functions of the form \( pn^2 + qn + r \) (with \( p \neq 0 \)) that we considered earlier are polynomials of degree 2.

A basic fact about polynomials is that their asymptotic rate of growth is determined by their "high-order term"—the one that determines the degree. We state this more formally in the following claim. Since we are concerned here only with functions that take nonnegative values, we will restrict our attention to polynomials for which the high-order term has a positive coefficient \( a_d > 0 \).

(2.7) Let \( f \) be a polynomial of degree \( d \), in which the coefficient \( a_d \) is positive. Then \( f = O(n^d) \).

**Proof.** We write \( f = a_0 + a_1n + a_2n^2 + \cdots + a_dn^d \), where \( a_d > 0 \). The upper bound is a direct application of (2.5). First, notice that coefficients \( a_j \) for \( j < d \) may be negative, but in any case we have \( a_jn^j \leq |a_j|n^d \) for all \( n \geq 1 \). Thus each term in the polynomial is \( O(n^d) \). Since \( f \) is a sum of a constant number of functions, each of which is \( O(n^d) \), it follows from (2.5) that \( f = O(n^d) \). \( \square \)

One can also show that under the conditions of (2.7), we have \( f = \Omega(n^d) \), and hence it follows that in fact \( f = \Theta(n^d) \).

This is a good point at which to discuss the relationship between these types of asymptotic bounds and the notion of polynomial time, which we arrived at in the previous section as a way to formalize the more elusive concept of efficiency. Using \( O(f) \) notation, it's easy to formally define polynomial time: a polynomial-time algorithm is one whose running time \( T(n) \) is \( O(n^d) \) for some constant \( d \), where \( d \) is independent of the input size.

So algorithms with running-time bounds like \( O(n^2) \) and \( O(n^3) \) are polynomial-time algorithms. But it's important to realize that an algorithm can be polynomial time even if its running time is not written as \( n \) raised to some integer power. To begin with, a number of algorithms have running times of the form \( O(n^d) \) for some number \( x \) that is not an integer. For example, in Chapter 5 we will see an algorithm whose running time is \( O(n^{1.5}) \); we will also see exponents less than 1, as in bounds like \( O(\sqrt{n}) = O(n^{1/2}) \).

To take another common kind of example, we will see many algorithms whose running times have the form \( O(n \log n) \). Such algorithms are also polynomial time: as we will see next, \( \log n \leq n \) for all \( n \geq 1 \), and hence \( n \log n \leq n^2 \) for all \( n \geq 1 \). In other words, if an algorithm has running time \( O(n \log n) \), then it also has running time \( O(n^2) \), and so it is a polynomial-time algorithm.

**Logarithms** Recall that \( \log_b n \) is the number \( x \) such that \( b^x = n \). One way to get an approximate sense of how fast \( \log_b n \) grows is to note that, if we round it down to the nearest integer, it is one less than the number of digits in the base-\( b \) representation of the number \( n \). (Thus, for example, \( 1 + \log_2 n \), rounded down, is the number of bits needed to represent \( n \).)

So logarithms are very slowly growing functions. In particular, for every base \( b \), the function \( \log_b n \) is asymptotically bounded by every function of the form \( n^x \), even for (noninteger) values of \( x \) arbitrary close to 0.

(2.8) For every \( b > 1 \) and every \( x > 0 \), we have \( \log_b n = O(n^x) \).

One can directly translate between logarithms of different bases using the following fundamental identity:

\[
\log_a n = \frac{\log_b n}{\log_b a}.
\]

This equation explains why you'll often notice people writing bounds like \( O(\log n) \) without indicating the base of the logarithm. This is not sloppy usage: the identity above says that \( \log_a n = \frac{1}{\log_b a} \cdot \log_b n \), so the point is that \( \log_a n = \Theta(\log_b n) \), and the base of the logarithm is not important when writing bounds using asymptotic notation.
Exponentials

Exponential functions are functions of the form $f(n) = r^n$ for some constant base $r$. Here we will be concerned with the case in which $r > 1$, which results in a very fast-growing function.

In particular, where polynomials raise $n$ to a fixed exponent, exponentials raise a fixed number to $n$ as a power; this leads to much faster rates of growth.

One way to summarize the relationship between polynomials and exponentials is as follows.

\[(2.9) \quad \text{For every } r > 1 \text{ and every } d > 0, \text{ we have } n^d = O(r^n).\]

In particular, every exponential grows faster than every polynomial. And as we saw in Table 2.1, when you plug in actual values of $n$, the differences in growth rates are really quite impressive.

Just as people write $O(\log n)$ without specifying the base, you'll also see people write “The running time of this algorithm is exponential,” without specifying which exponential function they have in mind. Unlike the liberal use of $\log n$, which is justified by ignoring constant factors, this generic use of the term “exponential” is somewhat sloppy. In particular, for different bases $r > s > 1$, it is never the case that $r^n = \Theta(s^n)$. Indeed, this would require that for some constant $c > 0$, we would have $r^n \leq cs^n$ for all sufficiently large $n$. But rearranging this inequality would give $(r/s)^n \leq c$ for all sufficiently large $n$. Since $r > s$, the expression $(r/s)^n$ is tending to infinity with $n$, and so it cannot possibly remain bounded by a fixed constant $c$.

So asymptotically speaking, exponential functions are all different. Still, it's usually clear what people intend when they inexacty write “The running time of this algorithm is exponential”—they typically mean that the running time grows at least as fast as some exponential function, and all exponentials grow so fast that we can effectively dismiss this algorithm without working out further details of the exact running time. This is not entirely fair. Occasionally there's more going on with an exponential algorithm than first appears, as we'll see, for example, in Chapter 10; but as we argued in the first section of this chapter, it's a reasonable rule of thumb.

Taken together, then, logarithms, polynomials, and exponentials serve as useful landmarks in the range of possible functions that you encounter when analyzing running times. Logarithms grow more slowly than polynomials, and polynomials grow more slowly than exponentials.

2.3 Implementing the Stable Matching Algorithm Using Lists and Arrays

We've now seen a general approach for expressing bounds on the running time of an algorithm. In order to asymptotically analyze the running time of an algorithm expressed in a high-level fashion—as we expressed the Gale-Shapley Stable Matching algorithm in Chapter 1, for example—one doesn't have to actually program, compile, and execute it, but one does have to think about how the data will be represented and manipulated in an implementation of the algorithm, so as to bound the number of computational steps it takes.

The implementation of basic algorithms using data structures is something that you probably have had some experience with. In this book, data structures will be covered in the context of implementing specific algorithms, and so we will encounter different data structures based on the needs of the algorithms we are developing. To get this process started, we consider an implementation of the Gale-Shapley Stable Matching algorithm; we showed earlier that the algorithm terminates in at most $n^2$ iterations, and our implementation here provides a corresponding worst-case running time of $O(n^2)$, counting actual computational steps rather than simply the total number of iterations. To get such a bound for the Stable Matching algorithm, we will only need to use two of the simplest data structures: lists and arrays. Thus, our implementation also provides a good chance to review the use of these basic data structures as well.

In the Stable Matching Problem, each man and each woman has a ranking of all members of the opposite gender. The very first question we need to discuss is how such a ranking will be represented. Further, the algorithm maintains a matching and will need to know at each step which men and women are free, and who is matched with whom. In order to implement the algorithm, we need to decide which data structures we will use for all these things.

An important issue to note here is that the choice of data structure is up to the algorithm designer; for each algorithm we will choose data structures that make it efficient and easy to implement. In some cases, this may involve preprocessing the input to convert it from its given input representation into a data structure that is more appropriate for the problem being solved.

Arrays and Lists

To start our discussion we will focus on a single list, such as the list of women in order of preference by a single man. Maybe the simplest way to keep a list of $n$ elements is to use an array $A$ of length $n$, and have $A[i]$ be the $i^{th}$ element of the list. Such an array is simple to implement in essentially all standard programming languages, and it has the following properties.

- We can answer a query of the form “What is the $i^{th}$ element on the list?” in $O(1)$ time, by a direct access to the value $A[i]$.
- If we want to determine whether a particular element $e$ belongs to the list (i.e., whether it is equal to $A[i]$ for some $i$), we need to check the