
Contents

Preface	xii
I The Basics	1
1 Introduction	2
1.1 Why Study Algorithms?	2
1.2 Integer Multiplication	3
1.3 Karatsuba Multiplication	6
1.4 MergeSort: The Algorithm	10
1.5 MergeSort: The Analysis	14
1.6 Guiding Principles for the Analysis of Algorithms	20
Problems	24
2 Asymptotic Notation	27
2.1 The Gist	27
2.2 Big-O Notation	33
2.3 Two Basic Examples	35
2.4 Big-Omega and Big-Theta Notation	37
2.5 Additional Examples	40
Problems	42
3 Divide-and-Conquer Algorithms	45
3.1 The Divide-and-Conquer Paradigm	45
3.2 Counting Inversions in $O(n \log n)$ Time	46
3.3 Strassen's Matrix Multiplication Algorithm	53
*3.4 An $O(n \log n)$ -Time Algorithm for the Closest Pair	58
Problems	68
4 The Master Method	71
4.1 Integer Multiplication Revisited	71
4.2 Formal Statement	73
4.3 Six Examples	75
*4.4 Proof of the Master Method	79
Problems	87

5	QuickSort	90
5.1	Overview	90
5.2	Partitioning Around a Pivot Element	93
5.3	The Importance of Good Pivots	98
5.4	Randomized QuickSort	101
*5.5	Analysis of Randomized QuickSort	104
*5.6	Sorting Requires $\Omega(n \log n)$ Comparisons	112
	Problems	116
6	Linear-Time Selection	120
6.1	The RSelect Algorithm	120
*6.2	Analysis of RSelect	125
*6.3	The DSelect Algorithm	129
*6.4	Analysis of DSelect	132
	Problems	138
II	Graph Algorithms and Data Structures	141
7	Graphs: The Basics	142
7.1	Some Vocabulary	142
7.2	A Few Applications	143
7.3	Measuring the Size of a Graph	144
7.4	Representing a Graph	146
	Problems	151
8	Graph Search and Its Applications	153
8.1	Overview	153
8.2	Breadth-First Search and Shortest Paths	159
8.3	Computing Connected Components	166
8.4	Depth-First Search	170
8.5	Topological Sort	174
*8.6	Computing Strongly Connected Components	181
8.7	The Structure of the Web	190
	Problems	193
9	Dijkstra's Shortest-Path Algorithm	198
9.1	The Single-Source Shortest Path Problem	198
9.2	Dijkstra's Algorithm	202
*9.3	Why Is Dijkstra's Algorithm Correct?	204
9.4	Implementation and Running Time	209
	Problems	210
10	The Heap Data Structure	214
10.1	Data Structures: An Overview	214
10.2	Supported Operations	216
10.3	Applications	218

10.4	Speeding Up Dijkstra's Algorithm	222
*10.5	Implementation Details	226
	Problems	235
11	Search Trees	237
11.1	Sorted Arrays	237
11.2	Search Trees: Supported Operations	239
*11.3	Implementation Details	241
*11.4	Balanced Search Trees	251
	Problems	254
12	Hash Tables and Bloom Filters	257
12.1	Supported Operations	257
12.2	Applications	259
*12.3	Implementation: High-Level Ideas	263
*12.4	Further Implementation Details	273
12.5	Bloom Filters: The Basics	277
*12.6	Bloom Filters: Heuristic Analysis	282
	Problems	286
III	Greedy Algorithms and Dynamic Programming	290
13	Introduction to Greedy Algorithms	291
13.1	The Greedy Algorithm Design Paradigm	291
13.2	A Scheduling Problem	293
13.3	Developing a Greedy Algorithm	295
13.4	Proof of Correctness	299
	Problems	303
14	Huffman Codes	306
14.1	Codes	306
14.2	Codes as Trees	309
14.3	Huffman's Greedy Algorithm	313
*14.4	Proof of Correctness	320
	Problems	326
15	Minimum Spanning Trees	328
15.1	Problem Definition	328
15.2	Prim's Algorithm	331
*15.3	Speeding Up Prim's Algorithm via Heaps	335
*15.4	Prim's Algorithm: Proof of Correctness	340
15.5	Kruskal's Algorithm	346
*15.6	Speeding Up Kruskal's Algorithm via Union-Find	349
*15.7	Kruskal's Algorithm: Proof of Correctness	357
15.8	Application: Single-Link Clustering	358
	Problems	362

16	Introduction to Dynamic Programming	366
16.1	The Weighted Independent Set Problem	366
16.2	A Linear-Time Algorithm for WIS in Paths	370
16.3	A Reconstruction Algorithm	376
16.4	The Principles of Dynamic Programming	377
16.5	The Knapsack Problem	381
	Problems	388
17	Advanced Dynamic Programming	392
17.1	Sequence Alignment	392
*17.2	Optimal Binary Search Trees	400
	Problems	411
18	Shortest Paths Revisited	415
18.1	Shortest Paths with Negative Edge Lengths	415
18.2	The Bellman-Ford Algorithm	418
18.3	The All-Pairs Shortest Path Problem	429
18.4	The Floyd-Warshall Algorithm	430
	Problems	438
IV	Algorithms for NP-Hard Problems	441
19	What Is NP-Hardness?	442
19.1	MST vs. TSP: An Algorithmic Mystery	442
19.2	Possible Levels of Expertise	446
19.3	Easy and Hard Problems	448
19.4	Algorithmic Strategies for NP-Hard Problems	453
19.5	Proving NP-Hardness: A Simple Recipe	457
19.6	Rookie Mistakes and Acceptable Inaccuracies	464
	Problems	467
20	Compromising on Correctness: Efficient Inexact Algorithms	471
20.1	Makespan Minimization	471
20.2	Maximum Coverage	481
*20.3	Influence Maximization	490
20.4	The 2-OPT Heuristic Algorithm for the TSP	497
20.5	Principles of Local Search	502
	Problems	511
21	Compromising on Speed: Exact Inefficient Algorithms	519
21.1	The Bellman-Held-Karp Algorithm for the TSP	519
*21.2	Finding Long Paths by Color Coding	525
21.3	Problem-Specific Algorithms vs. Magic Boxes	535
21.4	Mixed Integer Programming Solvers	537
21.5	Satisfiability Solvers	540
	Problems	545

22 Proving Problems NP-Hard	551
22.1 Reductions Revisited	551
22.2 3-SAT and the Cook-Levin Theorem	553
22.3 The Big Picture	554
22.4 A Template for Reductions	557
22.5 Independent Set Is NP-Hard	558
*22.6 Directed Hamiltonian Path Is NP-Hard	562
22.7 The TSP Is NP-Hard	567
22.8 Subset Sum Is NP-Hard	569
Problems	574
23 P, NP, and All That	577
*23.1 Amassing Evidence of Intractability	577
*23.2 Decision, Search, and Optimization	579
*23.3 \mathcal{NP} : Problems with Easily Recognized Solutions	580
*23.4 The $P \neq NP$ Conjecture	584
*23.5 The Exponential Time Hypothesis	587
*23.6 NP-Completeness	590
Problems	594
24 Case Study: The FCC Incentive Auction	596
24.1 Repurposing Wireless Spectrum	596
24.2 Greedy Heuristics for Buying Back Licenses	598
24.3 Feasibility Checking	604
24.4 Implementation as a Descending Clock Auction	609
24.5 The Final Outcome	613
Problems	615
A Quick Review of Proofs By Induction	617
A.1 A Template for Proofs by Induction	617
A.2 Example: A Closed-Form Formula	618
A.3 Example: The Size of a Complete Binary Tree	618
B Quick Review of Discrete Probability	620
B.1 Sample Spaces	620
B.2 Events	621
B.3 Random Variables	622
B.4 Expectation	623
B.5 Linearity of Expectation	624
B.6 Example: Load Balancing	627
Epilogue: A Field Guide to Algorithm Design	630
Hints and Solutions	632
Index	655

Preface

This book has only one goal: *to teach the basics of algorithms in the most accessible way possible*. Think of it as a transcript of what an expert algorithms tutor would say to you over a year of one-on-one lessons. This book is inspired by my online algorithms courses that have been running regularly since 2012, which in turn are based on an undergraduate course that I taught many times at Stanford University. People of all ages, backgrounds, and walks of life are well represented in these online courses, with large numbers of students (high-school, college, etc.), software engineers (both current and aspiring), scientists, and professionals hailing from all corners of the world.

What We'll Cover

Algorithms Illuminated will transform you from an algorithms newbie (say, a rising third-year undergraduate) to a seasoned veteran with expertise comparable to graduates of the world's best computer science master's degree programs. Specifically, this book will help you master the following topics.

Asymptotic analysis and big-O notation. Asymptotic notation provides the basic vocabulary for discussing the design and analysis of algorithms. The key concept here is “big-O” notation, which is a modeling choice about the granularity with which we measure the running time of an algorithm. We'll see that the sweet spot for clear high-level thinking about algorithm design is to ignore constant factors and lower-order terms, and to concentrate on how an algorithm's performance scales with the size of the input.

Divide-and-conquer algorithms and the master method. There's no silver bullet in algorithm design, no single problem-solving method that cracks all computational problems. However, there are a few general algorithm design techniques that find successful application across a range of different domains. In the “divide-and-conquer” technique, the key idea is to break a problem into smaller subproblems, solve the subproblems recursively, and then quickly combine their solutions into one for the original problem. We'll see fast divide-and-conquer algorithms for sorting, integer and matrix multiplication, and a basic problem in computational geometry. We'll also cover the master method, which is a powerful tool for analyzing the running time of divide-and-conquer algorithms.

Randomized algorithms. A randomized algorithm “flips coins” as it runs, and its behavior can depend on the outcomes of these coin flips. Surprisingly often, randomization leads to simple, elegant, and practical algorithms. Among other randomized algorithms, we'll describe and analyze in detail the canonical example of randomized QuickSort.

Sorting and selection. As a byproduct of studying the first three topics, we'll learn several famous algorithms for sorting and selection, including MergeSort, QuickSort, and linear-time selection (both randomized and deterministic). These computational primitives are so blazingly fast that they do not take much more time than that needed just to read the input. This book will help you cultivate a collection of such "for-free primitives," both to apply directly to data and to use as the building blocks for solutions to more difficult problems.

Graph search and applications. Graphs model many different types of networks, including road networks, communication networks, social networks, and networks of dependencies between tasks. Graphs can get complex, but there are several blazingly fast primitives for reasoning about graph structure. We begin with linear-time algorithms for searching a graph, with applications ranging from network analysis to task sequencing.

Shortest paths. In the shortest-path problem, the goal is to compute the best route in a network from point A to point B. The problem has obvious applications, like computing driving directions, and also shows up in disguise in many more general planning problems. We'll generalize one of our graph search algorithms and arrive at Dijkstra's famous shortest-path algorithm.

Data structures. This book will turn you into an educated client of several different data structures for maintaining an evolving set of objects with keys. The primary goal is to develop your intuition about which data structure is the right one for your application. The optional advanced sections provide guidance in how to implement these data structures from scratch.

We first discuss heaps, which can quickly identify the stored object with the smallest key and are useful for sorting, implementing a priority queue, and implementing Dijkstra's algorithm in near-linear time. Search trees maintain a total ordering over the keys of the stored objects and support an even richer array of operations. Hash tables are optimized for super-fast lookups and are ubiquitous in modern programs. We'll also cover the bloom filter, a close cousin of the hash table that uses less space at the expense of occasional errors, and the union-find (disjoint-set) data structure.

Greedy algorithms and applications. Greedy algorithms solve problems by making a sequence of myopic and irrevocable decisions. For many problems, they are easy to devise and often blazingly fast. Most greedy algorithms are not guaranteed to be correct, but we'll cover several killer applications that are exceptions to this rule. Examples include scheduling problems, optimal compression, and minimum spanning trees of graphs.

Dynamic programming and applications. Few benefits of a serious study of algorithms rival the empowerment that comes from mastering dynamic programming. This design paradigm takes a lot of practice to perfect, but it has countless applications to problems that appear unsolvable using any simpler method. Our dynamic programming boot camp will double as a tour of some of the paradigm's killer applications, including the knapsack problem, the Needleman-Wunsch genome sequence alignment algorithm, Knuth's algorithm for optimal binary search trees, and the Bellman-Ford and Floyd-Warshall shortest-path algorithms.

Algorithmic tools for tackling NP-hard problems. Many real-world problems are “NP-hard” and appear unsolvable by always-correct and always-fast algorithms. When an NP-hard problem shows up in your own work, you must compromise on either correctness or speed. We’ll see techniques old (like greedy algorithms) and new (like local search) for devising fast heuristic algorithms that are “approximately correct,” with applications to scheduling, influence maximization in social networks, and the traveling salesman problem. We’ll also cover techniques old (like dynamic programming) and new (like MIP and SAT solvers) for developing correct algorithms that improve dramatically over exhaustive search; applications here include the traveling salesman problem (again), finding signaling pathways in biological networks, and television station repacking in a recent and high-stakes spectrum auction in the United States.

Recognizing NP-hard problems. This book will also train you to quickly recognize an NP-hard problem so that you don’t inadvertently waste time trying to design a too-good-to-be-true algorithm for it. You’ll acquire familiarity with many famous and basic NP-hard problems, ranging from satisfiability to graph coloring to the Hamiltonian path problem. Through practice, you’ll learn the tricks of the trade in proving problems NP-hard via reductions.

For a more detailed look into the book’s contents, check out the “Upshot” sections that conclude each chapter and highlight the most important points. The “Field Guide to Algorithm Design” on page 630 provides a bird’s-eye view of how to apply the algorithmic toolbox you’ll acquire from this book to problems that you encounter in your own work.

The starred sections of the book are the most advanced ones. The time-constrained reader can skip these sections on a first reading without any loss of continuity.

Skills You’ll Learn

Mastering algorithms takes time and effort. Why bother?

Become a better programmer. You’ll learn several blazingly fast subroutines for processing data as well as several useful data structures for organizing data that you can deploy directly in your own programs. Implementing and using these algorithms will stretch and improve your programming skills. You’ll also learn general algorithm design paradigms that are relevant to many different problems across different domains, as well as tools for predicting the performance of such algorithms. These “algorithmic design patterns” can help you come up with new algorithms for problems that arise in your own work.

Sharpen your analytical skills. You’ll get lots of practice describing and reasoning about algorithms. Through mathematical analysis, you’ll gain a deep understanding of the specific algorithms and data structures that this book covers. You’ll acquire facility with several mathematical techniques that are broadly useful for analyzing algorithms.

Think algorithmically. After you learn about algorithms, you’ll start seeing them everywhere, whether you’re riding an elevator, watching a flock of birds, managing your investment portfolio, or even watching an infant learn. Algorithmic thinking is increasingly useful and prevalent in disciplines outside of computer science, including biology, statistics, and economics.

Literacy with computer science’s greatest hits. Studying algorithms can feel like watching a highlight reel of many of the greatest hits from the last sixty years of computer science. No longer will you feel excluded at that computer science cocktail party when someone cracks a joke about Dijkstra’s algorithm. After reading this book, you’ll know exactly what they mean.

Ace your technical interviews. Over the years, countless students have regaled me with stories about how mastering the concepts in this book enabled them to ace every technical interview question they were ever asked.

Using this Book in a Course

All of this book’s material has been battle-tested in a university setting—by yours truly at Stanford, and by many instructors at other schools. Parts I–III are tailor-made for serving as the primary text in an introductory undergraduate course on algorithms and data structures, focusing on the basics (Part I), graph algorithms and data structures (Part II), and greedy algorithms and dynamic programming (Part III). For example, I cover 75-80% of this content over nineteen 75-minute lectures; a semester-long course could accommodate the rest of it, or selected topics from Chapter 19 about NP-hard problems. Parts III and IV of the book form an ideal basis for a traditional master’s level course on basic and advanced algorithms, emphasizing greedy algorithms and their applications, dynamic programming and its applications, the recognition of NP-hard problems, and algorithmic tools for tackling such problems.

Many chapters of this book are logically independent of each other. For example, the design paradigms of divide-and-conquer algorithms (Chapters 3–6), greedy algorithms (Chapters 13–15), and dynamic programming (Chapter 16–18) can be covered in any order. Similarly, data structures (Chapters 10–12) can be taught before or after basic graph algorithms (Chapters 7–9), with the exception of the heap-based implementation of Dijkstra’s algorithm in Section 10.4.

As for prerequisites, students in an algorithms course generally have at least a little background in programming (including, for example, the use of arrays and recursion) and in mathematical reasoning (such as proofs by induction and by contradiction). Readers can level up their programming and mathematical skills with any number of free online resources (see www.algorithmsilluminated.org for specific recommendations). Appendices A and B also offer quick reviews of induction and discrete probability, respectively. Alternatively, readers without any programming experience can learn the basics of algorithm design and analysis from this book at the level of pseudocode descriptions (if not concrete implementations), and those with little mathematical background can focus squarely on algorithm design techniques (if not detailed algorithm analysis).

I’ve recorded over 200 YouTube videos that cover in detail every section of the book, as well as additional advanced content (on Karger’s random contraction algorithm, path compression in disjoint set data structures, Johnson’s all-pairs shortest-path algorithm, and more). These videos, which are catalogued at www.algorithmsilluminated.org, can serve an instructor in three different ways: (i) as part of a flipped classroom, with students watching lecture videos in advance of class time, which can then be used for discussion and problem-solving exercises; (ii) as a way to help students fill in missing prerequisites without

taking up class time; and (iii) as supplementary material above and beyond what is covered during class time (perhaps for an honors section or extra credit).

Additional Features and Resources

This book is based on online courses that are currently running on the Coursera and EdX platforms. I've made several resources available to help you replicate as much of the online course experience as you like.

Videos. If you're more in the mood to watch and listen than to read, check out the YouTube video playlists available at www.algorithmsilluminated.org. These videos feature yours truly teaching all the topics in this book, as well as additional advanced topics. I hope they exude a contagious enthusiasm for algorithms that, alas, is impossible to replicate fully on the printed page.

Quizzes. How can you know if you're truly absorbing the concepts in this book? Over 100 quizzes with solutions and explanations are scattered throughout the text; when you encounter one, I encourage you to pause and think about the answer before reading on.

End-of-chapter problems. At the end of each chapter, you'll find several relatively straightforward questions that test your understanding, followed by harder and more open-ended challenge problems. Hints or solutions to all of these problems (as indicated by an “(H)” or “(S),” respectively) are included at the end of the book. Readers can interact with me and each other about the end-of-chapter problems through the book's discussion forum (see below).

Programming problems. Most of the chapters conclude with suggested programming projects whose goal is to help you develop a detailed understanding of an algorithm by creating your own working implementation of it. Data sets, along with test cases and their solutions, can be found at www.algorithmsilluminated.org.

Discussion forums. A big reason for the success of online courses is the opportunities they provide for participants to help each other understand the course material and debug programs through discussion forums. Readers of this book have the same opportunity via the forums available at www.algorithmsilluminated.org.

Acknowledgments

This book would not exist without the passion and hunger supplied by the hundreds of thousands of participants in my algorithms courses over the years. I am particularly grateful to those who supplied detailed feedback on earlier drafts: Tonya Blust, Yuan Cao, Lauren Cowles, Leslie Damon, Tyler Dae Devlin, Roman Gafiteanu, Carlos Guia, Blanca Huergo, Jim Humelsine, Tim Kearns, Vladimir Kokshenev, Bayram Kuliyeu, Patrick Monkelban, Kyle Schiller, Nissanka Wickremasinghe, Clayton Wong, Lexin Ye, Daniel Zingaro, several anonymous reviewers, and many pseudonymous contributors to the book's discussion forums. Thanks also to several experts who provided technical advice: Amir Abboud, Vincent Conitzer, Christian Kroer, Aviad Rubinstein, and Ilya Segal.

I always appreciate suggestions and corrections from readers. These are best communicated through the discussion forums mentioned above.

Tim Roughgarden
New York, NY
April 2022