
Undergraduate Topics in Computer Science

Series editor

Ian Mackie

Advisory Board

Samson Abramsky, University of Oxford, Oxford, UK

Chris Hankin, Imperial College London, London, UK

Dexter C. Kozen, Cornell University, Ithaca, USA

Andrew Pitts, University of Cambridge, Cambridge, UK

Hanne Riis Nielson, Technical University of Denmark, Kongens Lyngby, Denmark

Steven S. Skiena, Stony Brook University, Stony Brook, USA

Iain Stewart, University of Durham, Durham, UK

Undergraduate Topics in Computer Science (UTiCS) delivers high-quality instructional content for undergraduates studying in all areas of computing and information science. From core foundational and theoretical material to final-year topics and applications, UTiCS books take a fresh, concise, and modern approach and are ideal for self-study or for a one- or two-semester course. The texts are all authored by established experts in their fields, reviewed by an international advisory board, and contain numerous examples and problems. Many include fully worked solutions.

More information about this series at <http://www.springer.com/series/7592>

Antti Laaksonen

Guide to Competitive Programming

Learning and Improving Algorithms
Through Contests

Antti Laaksonen
Department of Computer Science
University of Helsinki
Helsinki
Finland

ISSN 1863-7310 ISSN 2197-1781 (electronic)
Undergraduate Topics in Computer Science
ISBN 978-3-319-72546-8 ISBN 978-3-319-72547-5 (eBook)
<https://doi.org/10.1007/978-3-319-72547-5>

Library of Congress Control Number: 2017960923

© Springer International Publishing AG, part of Springer Nature 2017

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Printed on acid-free paper

This Springer imprint is published by the registered company is Springer International Publishing AG part of Springer Nature.

The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

The purpose of this book is to give you a comprehensive introduction to modern competitive programming. It is assumed that you already know the basics of programming, but previous background in algorithm design or programming contests is not necessary. Since the book covers a wide range of topics of various difficulty, it suits both for beginners and more experienced readers.

Programming contests already have a quite long history. The *International Collegiate Programming Contest* for university students was started during the 1970s, and the first *International Olympiad in Informatics* for secondary school students was organized in 1989. Both competitions are now established events with a large number of participants from all around the world.

Today, competitive programming is more popular than ever. The Internet has played a significant role in this progress. There is now an active online community of competitive programmers, and many contests are organized every week. At the same time, the difficulty of contests is increasing. Techniques that only the very best participants mastered some years ago are now standard tools known by a large number of people.

Competitive programming has its roots in the scientific study of algorithms. However, while a computer scientist writes a proof to show that their algorithm works, a competitive programmer *implements* their algorithm and submits it to a contest system. Then, the algorithm is tested using a set of test cases, and if it passes all of them, it is accepted. This is an essential element in competitive programming, because it provides a way to *automatically* get strong evidence that an algorithm works. In fact, competitive programming has proved to be an excellent way to learn algorithms, because it encourages to design algorithms that really work, instead of sketching ideas that may work or not.

Another benefit of competitive programming is that contest problems require *thinking*. In particular, there are no spoilers in problem statements. This is actually a severe problem in many algorithms courses. You are given a nice problem to solve, but then the last sentence says, for example: “*Hint*: modify Dijkstra’s algorithm to solve the problem.” After reading this, there is not much thinking needed, because you already know how to solve the problem. This never happens in competitive

programming. Instead, you have a full set of tools available, and you have to figure out *yourself* which of them to use.

Solving competitive programming problems also improves one's programming and debugging skills. Typically, a solution is awarded points only if it correctly solves all test cases, so a successful competitive programmer has to be able to implement programs that do not have bugs. This is a valuable skill in software engineering, and it is not a coincidence that IT companies are interested in people who have background in competitive programming.

It takes a long time to become a good competitive programmer, but it is also an opportunity to learn a lot. You can be sure that you will get a good general understanding of algorithms if you spend time reading the book, solving problems, and taking part in contests.

If you have any feedback, I would like to hear it! You can always send me a message to ahslaaks@cs.helsinki.fi.

I am very grateful to a large number of people who have sent me feedback on draft versions of this book. This feedback has greatly improved the quality of the book. I especially thank Mikko Ervasti, Janne Junnila, Janne Kokkala, Tuukka Korhonen, Patric Östergård, and Roope Salmi for giving detailed feedback on the manuscript. I also thank Simon Rees and Wayne Wheeler for excellent collaboration when publishing this book with Springer.

Helsinki, Finland
October 2017

Antti Laaksonen

Contents

1	Introduction	1
1.1	What is Competitive Programming?	1
1.1.1	Programming Contests	2
1.1.2	Tips for Practicing	3
1.2	About This Book	3
1.3	CSES Problem Set	5
1.4	Other Resources	7
2	Programming Techniques	9
2.1	Language Features	9
2.1.1	Input and Output	10
2.1.2	Working with Numbers	12
2.1.3	Shortening Code	14
2.2	Recursive Algorithms	15
2.2.1	Generating Subsets	15
2.2.2	Generating Permutations	16
2.2.3	Backtracking	18
2.3	Bit Manipulation	20
2.3.1	Bit Operations	21
2.3.2	Representing Sets	23
3	Efficiency	27
3.1	Time Complexity	27
3.1.1	Calculation Rules	27
3.1.2	Common Time Complexities	30
3.1.3	Estimating Efficiency	31
3.1.4	Formal Definitions	32
3.2	Examples	32
3.2.1	Maximum Subarray Sum	32
3.2.2	Two Queens Problem	35
4	Sorting and Searching	37
4.1	Sorting Algorithms	37
4.1.1	Bubble Sort	38

4.1.2	Merge Sort	39
4.1.3	Sorting Lower Bound	40
4.1.4	Counting Sort	41
4.1.5	Sorting in Practice	41
4.2	Solving Problems by Sorting	43
4.2.1	Sweep Line Algorithms	44
4.2.2	Scheduling Events	45
4.2.3	Tasks and Deadlines	45
4.3	Binary Search	46
4.3.1	Implementing the Search	47
4.3.2	Finding Optimal Solutions	48
5	Data Structures	51
5.1	Dynamic Arrays	51
5.1.1	Vectors	52
5.1.2	Iterators and Ranges	53
5.1.3	Other Structures	54
5.2	Set Structures	55
5.2.1	Sets and Multisets	55
5.2.2	Maps	57
5.2.3	Priority Queues	58
5.2.4	Policy-Based Sets	59
5.3	Experiments	60
5.3.1	Set Versus Sorting	60
5.3.2	Map Versus Array	61
5.3.3	Priority Queue Versus Multiset	62
6	Dynamic Programming	63
6.1	Basic Concepts	63
6.1.1	When Greedy Fails	63
6.1.2	Finding an Optimal Solution	64
6.1.3	Counting Solutions	67
6.2	Further Examples	68
6.2.1	Longest Increasing Subsequence	69
6.2.2	Paths in a Grid	70
6.2.3	Knapsack Problems	71
6.2.4	From Permutations to Subsets	72
6.2.5	Counting Tilings	74
7	Graph Algorithms	77
7.1	Basics of Graphs	78
7.1.1	Graph Terminology	78
7.1.2	Graph Representation	80
7.2	Graph Traversal	83
7.2.1	Depth-First Search	83

7.2.2	Breadth-First Search	85
7.2.3	Applications	86
7.3	Shortest Paths	87
7.3.1	Bellman–Ford Algorithm	88
7.3.2	Dijkstra’s Algorithm	89
7.3.3	Floyd–Warshall Algorithm	92
7.4	Directed Acyclic Graphs	94
7.4.1	Topological Sorting	94
7.4.2	Dynamic Programming	96
7.5	Successor Graphs	97
7.5.1	Finding Successors	98
7.5.2	Cycle Detection	99
7.6	Minimum Spanning Trees	100
7.6.1	Kruskal’s Algorithm	101
7.6.2	Union-Find Structure	103
7.6.3	Prim’s Algorithm	106
8	Algorithm Design Topics	107
8.1	Bit-Parallel Algorithms	107
8.1.1	Hamming Distances	107
8.1.2	Counting Subgrids	108
8.1.3	Reachability in Graphs	110
8.2	Amortized Analysis	111
8.2.1	Two Pointers Method	111
8.2.2	Nearest Smaller Elements	113
8.2.3	Sliding Window Minimum	114
8.3	Finding Minimum Values	115
8.3.1	Ternary Search	115
8.3.2	Convex Functions	116
8.3.3	Minimizing Sums	117
9	Range Queries	119
9.1	Queries on Static Arrays	119
9.1.1	Sum Queries	120
9.1.2	Minimum Queries	121
9.2	Tree Structures	122
9.2.1	Binary Indexed Trees	122
9.2.2	Segment Trees	125
9.2.3	Additional Techniques	128
10	Tree Algorithms	131
10.1	Basic Techniques	131
10.1.1	Tree Traversal	132
10.1.2	Calculating Diameters	134
10.1.3	All Longest Paths	135

10.2	Tree Queries	137
10.2.1	Finding Ancestors	137
10.2.2	Subtrees and Paths	138
10.2.3	Lowest Common Ancestors	140
10.2.4	Merging Data Structures	142
10.3	Advanced Techniques	144
10.3.1	Centroid Decomposition	144
10.3.2	Heavy-Light Decomposition	145
11	Mathematics	147
11.1	Number Theory	147
11.1.1	Primes and Factors	148
11.1.2	Sieve of Eratosthenes	150
11.1.3	Euclid's Algorithm	151
11.1.4	Modular Exponentiation	153
11.1.5	Euler's Theorem	153
11.1.6	Solving Equations	155
11.2	Combinatorics	156
11.2.1	Binomial Coefficients	157
11.2.2	Catalan Numbers	159
11.2.3	Inclusion-Exclusion	161
11.2.4	Burnside's Lemma	163
11.2.5	Cayley's Formula	164
11.3	Matrices	164
11.3.1	Matrix Operations	165
11.3.2	Linear Recurrences	167
11.3.3	Graphs and Matrices	169
11.3.4	Gaussian Elimination	170
11.4	Probability	173
11.4.1	Working with Events	174
11.4.2	Random Variables	175
11.4.3	Markov Chains	178
11.4.4	Randomized Algorithms	179
11.5	Game Theory	181
11.5.1	Game States	181
11.5.2	Nim Game	182
11.5.3	Sprague-Grundy Theorem	184
12	Advanced Graph Algorithms	189
12.1	Strong Connectivity	189
12.1.1	Kosaraju's Algorithm	190
12.1.2	2SAT Problem	192
12.2	Complete Paths	193
12.2.1	Eulerian Paths	194

12.2.2	Hamiltonian Paths	195
12.2.3	Applications	196
12.3	Maximum Flows.	198
12.3.1	Ford–Fulkerson Algorithm.	199
12.3.2	Disjoint Paths	202
12.3.3	Maximum Matchings.	203
12.3.4	Path Covers	205
12.4	Depth-First Search Trees	207
12.4.1	Biconnectivity	207
12.4.2	Eulerian Subgraphs	209
13	Geometry	211
13.1	Geometric Techniques	211
13.1.1	Complex Numbers.	211
13.1.2	Points and Lines	213
13.1.3	Polygon Area	216
13.1.4	Distance Functions	218
13.2	Sweep Line Algorithms	220
13.2.1	Intersection Points	220
13.2.2	Closest Pair Problem	221
13.2.3	Convex Hull Problem	224
14	String Algorithms	225
14.1	Basic Topics.	225
14.1.1	Trie Structure	226
14.1.2	Dynamic Programming	227
14.2	String Hashing	228
14.2.1	Polynomial Hashing	228
14.2.2	Applications	229
14.2.3	Collisions and Parameters	230
14.3	Z-Algorithm	231
14.3.1	Constructing the Z-Array.	232
14.3.2	Applications	233
14.4	Suffix Arrays	234
14.4.1	Prefix Doubling Method	235
14.4.2	Finding Patterns.	236
14.4.3	LCP Arrays	236
15	Additional Topics	239
15.1	Square Root Techniques.	239
15.1.1	Data Structures	240
15.1.2	Subalgorithms	241
15.1.3	Integer Partitions	243
15.1.4	Mo’s Algorithm.	244

15.2	Segment Trees Revisited.	245
15.2.1	Lazy Propagation.	246
15.2.2	Dynamic Trees	249
15.2.3	Data Structures in Nodes.	251
15.2.4	Two-Dimensional Trees.	253
15.3	Treaps.	253
15.3.1	Splitting and Merging	253
15.3.2	Implementation	255
15.3.3	Additional Techniques.	257
15.4	Dynamic Programming Optimization	258
15.4.1	Convex Hull Trick.	258
15.4.2	Divide and Conquer Optimization	260
15.4.3	Knuth's Optimization	261
15.5	Miscellaneous	262
15.5.1	Meet in the Middle	263
15.5.2	Counting Subsets.	263
15.5.3	Parallel Binary Search.	265
15.5.4	Dynamic Connectivity.	266
	Appendix A: Mathematical Background	269
	References	277
	Index	279