HOW TO THINK ABOUT ALGORITHMS

There are many algorithm texts that provide lots of well-polished code and proofs of correctness. Instead, this one presents insights, notations, and analogies to help the novice describe and think about algorithms like an expert. It is a bit like a carpenter studying hammers instead of houses. Jeff Edmonds provides both the big picture and easy step-by-step methods for developing algorithms, while avoiding the common pitfalls. Paradigms such as loop invariants and recursion help to unify a huge range of algorithms into a few meta-algorithms. Part of the goal is to teach students to think abstractly. Without getting bogged down in formal proofs, the book fosters deeper understanding so that how and why each algorithm works is transparent. These insights are presented in a slow and clear manner accessible to second- or third-year students of computer science, preparing them to find on their own innovative ways to solve problems.

Abstraction is when you translate the equations, the rules, and the underlying essences of the problem not only into a language that can be communicated to your friend standing with you on a streetcar, but also into a form that can percolate down and dwell in your subconscious. Because, remember, it is your subconscious that makes the miraculous leaps of inspiration, not your plodding perspiration and not your cocky logic. And remember, unlike you, your subconscious does not understand Java code.
HOW TO THINK ABOUT ALGORITHMS

JEFF EDMONDS

York University
Dedicated to my father, Jack, and to my sons, Joshua and Micah.

May the love and the mathematics continue to flow between the generations.
Problem Solving
Out of the Box Leaping
Deep Thinking
Creative Abstracting
Logical Deducing
with Friends Working
Fun Having
Fumbling and Bumbling
Bravely Persevering
Joyfully Succeeding
CONTENTS

Preface ............................................................................................... xi

Introduction ...................................................................................... 1

PART ONE. ITERATIVE ALGORITHMS AND LOOP INVARIANTS

1 Iterative Algorithms: Measures of Progress and Loop Invariants ........ 5
   1.1 A Paradigm Shift: A Sequence of Actions vs. a Sequence of Assertions 5
   1.2 The Steps to Develop an Iterative Algorithm .................................. 8
   1.3 More about the Steps ................................................................ 12
   1.4 Different Types of Iterative Algorithms ..................................... 21
   1.5 Typical Errors ......................................................................... 26
   1.6 Exercises .............................................................................. 27

2 Examples Using More-of-the-Input Loop Invariants ......................... 29
   2.1 Coloring the Plane ................................................................ 29
   2.2 Deterministic Finite Automaton .............................................. 31
   2.3 More of the Input vs. More of the Output .................................. 39

3 Abstract Data Types ..................................................................... 43
   3.1 Specifications and Hints at Implementations ............................. 44
   3.2 Link List Implementation ...................................................... 51
   3.3 Merging with a Queue .......................................................... 56
   3.4 Parsing with a Stack ............................................................... 57

4 Narrowing the Search Space: Binary Search .................................... 60
   4.1 Binary Search Trees ............................................................... 60
   4.2 Magic Sevens ........................................................................ 62
   4.3 VLSI Chip Testing .................................................................. 65
   4.4 Exercises ............................................................................... 69

5 Iterative Sorting Algorithms .......................................................... 71
   5.1 Bucket Sort by Hand ............................................................... 71
Contents

5.2 Counting Sort (a Stable Sort) 72
5.3 Radix Sort 75
5.4 Radix Counting Sort 76

6 Euclid's GCD Algorithm ........................................ 79

7 The Loop Invariant for Lower Bounds .......................... 85

PART TWO. RECURSION

8 Abstractions, Techniques, and Theory .......................... 97
  8.1 Thinking about Recursion 97
  8.2 Looking Forward vs. Backward 99
  8.3 With a Little Help from Your Friends 100
  8.4 The Towers of Hanoi 102
  8.5 Checklist for Recursive Algorithms 104
  8.6 The Stack Frame 110
  8.7 Proving Correctness with Strong Induction 112

9 Some Simple Examples of Recursive Algorithms .................. 114
  9.1 Sorting and Selecting Algorithms 114
  9.2 Operations on Integers 122
  9.3 Ackermann's Function 127
  9.4 Exercises 128

10 Recursion on Trees ........................................... 130
  10.1 Tree Traversals 133
  10.2 Simple Examples 135
  10.3 Generalizing the Problem Solved 138
  10.4 Heap Sort and Priority Queues 141
  10.5 Representing Expressions with Trees 149

11 Recursive Images .............................................. 153
  11.1 Drawing a Recursive Image from a Fixed Recursive and a Base
        Case Image 153
  11.2 Randomly Generating a Maze 156

12 Parsing with Context-Free Grammars ........................... 159

PART THREE. OPTIMIZATION PROBLEMS

13 Definition of Optimization Problems .......................... 171

14 Graph Search Algorithms ...................................... 173
  14.1 A Generic Search Algorithm 174
  14.2 Breadth-First Search for Shortest Paths 179
  14.3 Dijkstra's Shortest-Weighted-Path Algorithm 183
  14.4 Depth-First Search 188
  14.5 Recursive Depth-First Search 192
  14.6 Linear Ordering of a Partial Order 194
  14.7 Exercise 196
## Contents

15 **Network Flows and Linear Programming** .......................... 198  
15.1 A Hill-Climbing Algorithm with a Small Local Maximum 200  
15.2 The Primal–Dual Hill-Climbing Method 206  
15.3 The Steepest-Ascent Hill-Climbing Algorithm 214  
15.4 Linear Programming 219  
15.5 Exercises 223  

16 **Greedy Algorithms** ........................................ 225  
16.1 Abstractions, Techniques, and Theory 225  
16.2 Examples of Greedy Algorithms 236  
16.2.1 Example: The Job/Event Scheduling Problem 236  
16.2.2 Example: The Interval Cover Problem 240  
16.2.3 Example: The Minimum-Spanning-Tree Problem 244  
16.3 Exercises 250  

17 **Recursive Backtracking** ...................................... 251  
17.1 Recursive Backtracking Algorithms 251  
17.2 The Steps in Developing a Recursive Backtracking 256  
17.3 Pruning Branches 260  
17.4 Satisfiability 261  
17.5 Exercises 265  

18 **Dynamic Programming Algorithms** .......................... 267  
18.1 Start by Developing a Recursive Backtracking 267  
18.2 The Steps in Developing a Dynamic Programming Algorithm 271  
18.3 Subtle Points 277  
18.3.1 The Question for the Little Bird 278  
18.3.2 Subinstances and Subsolutions 281  
18.3.3 The Set of Subinstances 284  
18.3.4 Decreasing Time and Space 288  
18.3.5 Counting the Number of Solutions 291  
18.3.6 The New Code 292  

19 **Examples of Dynamic Programs** .............................. 295  
19.1 The Longest-Common-Subsequence Problem 295  
19.2 Dynamic Programs as More-of-the-Input Iterative Loop  
Invariant Algorithms 300  
19.3 A Greedy Dynamic Program: The Weighted Job/Event  
Scheduling Problem 303  
19.4 The Solution Viewed as a Tree: Chains of Matrix Multiplications 306  
19.5 Generalizing the Problem Solved: Best AVL Tree 311  
19.6 All Pairs Using Matrix Multiplication 314  
19.7 Parsing with Context-Free Grammars 315  
19.8 Designing Dynamic Programming Algorithms via Reductions 318
Contents

20 Reductions and NP-Completeness .......................... 324
   20.1 Satisfiability Is at Least as Hard as Any Optimization Problem 326
   20.2 Steps to Prove NP-Completeness 330
   20.3 Example: 3-Coloring Is NP-Complete 338
   20.4 An Algorithm for Bipartite Matching Using the Network Flow Algorithm 342

21 Randomized Algorithms ................................. 346
   21.1 Using Randomness to Hide the Worst Cases 347
   21.2 Solutions of Optimization Problems with a Random Structure 350

PART FOUR. APPENDIX

22 Existential and Universal Quantifiers .................... 357

23 Time Complexity .................................. 366
   23.1 The Time (and Space) Complexity of an Algorithm 366
   23.2 The Time Complexity of a Computational Problem 371

24 Logarithms and Exponentials .......................... 374

25 Asymptotic Growth ................................ 377
   25.1 Steps to Classify a Function 379
   25.2 More about Asymptotic Notation 384

26 Adding-Made-Easy Approximations .................... 388
   26.1 The Technique 389
   26.2 Some Proofs for the Adding-Made-Easy Technique 393

27 Recurrence Relations ................................ 398
   27.1 The Technique 398
   27.2 Some Proofs 401

28 A Formal Proof of Correctness ....................... 408

PART FIVE. EXERCISE SOLUTIONS .......................... 411

Conclusion ........................................... 437

Index ............................................. 439
PREFACE

To the Educator and the Student

This book is designed to be used in a twelve-week, third-year algorithms course. The goal is to teach students to think abstractly about algorithms and about the key algorithmic techniques used to develop them.

Meta-Algorithms: Students must learn so many algorithms that they are sometimes overwhelmed. In order to facilitate their understanding, most textbooks cover the standard themes of iterative algorithms, recursion, greedy algorithms, and dynamic programming. Generally, however, when it comes to presenting the algorithms themselves and their proofs of correctness, the concepts are hidden within optimized code and slick proofs. One goal of this book is to present a uniform and clean way of thinking about algorithms. We do this by focusing on the structure and proof of correctness of iterative and recursive meta-algorithms, and within these the greedy and dynamic programming meta-algorithms. By learning these and their proofs of correctness, most actual algorithms can be easily understood. The challenge is that thinking about meta-algorithms requires a great deal of abstract thinking.

Abstract Thinking: Students are very good at learning how to apply a concrete code to a concrete input instance. They tend, however, to find it difficult to think abstractly about the algorithms. I maintain that the more abstractions a person has from which to view the problem, the deeper his understanding of it will be, the more tools he will have at his disposal, and the better prepared he will be to design his own innovative ways to solve new problems. Hence, I present a number of different notations, analogies, and paradigms within which to develop and to think about algorithms.
Preface

Way of Thinking: People who develop algorithms have various ways of thinking and intuition that tend not to get taught. The assumption, I suppose, is that these cannot be taught but must be figured out on one's own. This text attempts to teach students to think like a designer of algorithms.

Not a Reference Book: My intention is not to teach a specific selection of algorithms for specific purposes. Hence, the book is not organized according to the application of the algorithms, but according to the techniques and abstractions used to develop them.

Developing Algorithms: The goal is not to present completed algorithms in a nice clean package, but to go slowly through every step of the development. Many false starts have been added. The hope is that this will help students learn to develop algorithms on their own. The difference is a bit like the difference between studying carpentry by looking at houses and by looking at hammers.

Proof of Correctness: Our philosophy is not to follow an algorithm with a formal proof that it is correct. Instead, this text is about learning how to think about, develop, and describe algorithms in such way that their correctness is transparent.

Big Picture vs. Small Steps: For each topic, I attempt both to give the big picture and to break it down into easily understood steps.

Level of Presentation: This material is difficult. There is no getting around that. I have tried to figure out where confusion may arise and to cover these points in more detail. I try to balance the succinct clarity that comes with mathematical formalism against the personified analogies and metaphors that help to provide both intuition and humor.

Point Form: The text is organized into blocks, each containing a title and a single thought. Hopefully, this will make the text easier to lecture and study from.

Prerequisites: The text assumes that the students have completed a first-year programming course and have a general mathematical maturity. The Appendix (Part Four) covers much of the mathematics that will be needed.

Homework Questions: A few homework questions are included. I am hoping to develop many more, along with their solutions. Contributions are welcome.

Read Ahead: The student is expected to read the material before the lecture. This will facilitate productive discussion during class.

Explaining: To be able to prove yourself on a test or on the job, you need to be able to explain the material well. In addition, explaining it to someone else is the best way to learn it yourself. Hence, I highly recommend spending a lot of time explaining
Preface

the material over and over again out loud to yourself, to each other, and to your stuffed bear.

**Dreaming:** I would like to emphasis the importance of thinking, even daydreaming, about the material. This can be done while going through your day – while swimming, showering, cooking, or lying in bed. Ask questions. Why is it done this way and not that way? Invent other algorithms for solving a problem. Then look for input instances for which your algorithm gives the wrong answer. Mathematics is not all linear thinking.

If the essence of the material, what the questions are really asking, is allowed to seep down into your subconscious then with time little thoughts will begin to percolate up. Pursue these ideas. Sometimes even flashes of inspiration appear.

Acknowledgments

I would like to thank Andy Mirzaian, Franck van Breugel, James Elder, Suprakash Datta, Eric Ruppert, Russell Impagliazzo, Toniann Pitassi, and Kirk Pruhs, with whom I co-taught and co-researched algorithms for many years. I would like to thank Jennifer Wolfe and Lauren Cowles for their fantastic editing jobs. All of these people were a tremendous support for this work.