Texts in Computer Science

Series Editors

David Gries, Department of Computer Science, Cornell University, Ithaca, NY, USA

Orit Hazzan, Faculty of Education in Technology and Science, Technion—Israel Institute of Technology, Haifa, Israel

Titles in this series now included in the Thomson Reuters Book Citation Index!

'Texts in Computer Science' (TCS) delivers high-quality instructional content for undergraduates and graduates in all areas of computing and information science, with a strong emphasis on core foundational and theoretical material but inclusive of some prominent applications-related content. TCS books should be reasonably self-contained and aim to provide students with modern and clear accounts of topics ranging across the computing curriculum. As a result, the books are ideal for semester courses or for individual self-study in cases where people need to expand their knowledge. All texts are authored by established experts in their fields, reviewed internally and by the series editors, and provide numerous examples, problems, and other pedagogical tools; many contain fully worked solutions.

The TCS series is comprised of high-quality, self-contained books that have broad and comprehensive coverage and are generally in hardback format and sometimes contain color. For undergraduate textbooks that are likely to be more brief and modular in their approach, require only black and white, and are under 275 pages, Springer offers the flexibly designed Undergraduate Topics in Computer Science series, to which we refer potential authors.

Marco T. Morazán

Animated Program Design

Intermediate Program Design Using Video Game Development

Springer

Marco T. Morazán
Department of Computer Science
Seton Hall University
South Orange, NJ, USA

ISSN 1868-0941 ISSN 1868-095X (electronic)
Texts in Computer Science
ISBN 978-3-031-04316-1 ISBN 978-3-031-04317-8 (eBook)
https://doi.org/10.1007/978-3-031-04317-8

© The Editor(s) (if applicable) and The Author(s), under exclusive license to Springer Nature Switzerland AG 2022
This work is subject to copyright. All rights are solely and exclusively licensed 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, expressed 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.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

To my students for educating me on how to teach problem solving and program design.

Marco

Preface

Everybody engages in problem solving, and this book is about the science of problem solving. It aims to teach its readers a new way of thinking about designing solutions to problems that takes them beyond trial-and-error thinking. Trial and error is, indeed, a fundamental problem-solving technique but must be tightly coupled with design to be effectively used. Trials or experiments must be thought out and planned. What does this mean? It means that problem solvers must engage in careful and thorough consideration of the different options they have to solve a problem. Problem-solving techniques must be used appropriately, and different solutions to a problem must be evaluated to choose the best one. The evaluation of a solution is done both mathematically and empirically. That is, theory and practice play a pivotal role in problem solving. Rest assured that the problem-solving and programming techniques you learn may be used to solve problems using any programming language.

This textbook continues the journey started in *Animated Problem Solving* and completes a year-long (two semesters) curriculum for first-year students. Readers of this book are likely to be familiar with writing expressions, defining data, divide and conquer, iterative design, designing functions using structural recursion, abstraction and abstract functions, and even distributed programming. Indeed, you are likely to already be very powerful problem solvers and programmers. Now it is time to become even more powerful programmers. How is this achieved? This book aids this quest by exploring with you new types of recursion, by introducing you to the use of randomness, by taking the first steps into experimental Computer Science and algorithm analysis, by taking a peek into Artificial Intelligence, and by presenting a disciplined approach to the use of mutation—also known as assignment which is routinely abused and misused every day giving rise to the majority of programming bugs today.

At the heart of this exploration is the *design recipe*—the steps to go from a problem statement to a working and tested solution. The new design recipes studied in this textbook are less prescriptive than those used for solutions based on structural recursion. In this regard, they are akin to the design recipe for distributed programming found in *Animated Problem Solving*. One of the most attractive features of structural recursion is that it suggests how to divide and conquer a problem. For example, structural recursion suggests that solving a problem for a nonleaf binary tree is done by solving the same problem for the left and/or right subtrees. In contrast, heap sorting, an efficient sorting algorithm studied in this textbook, creates a new binary tree to solve the problem. In essence, there is no prescriptive design recipe for divide and conquer when structural recursion is not used. In such cases, a problem solver must rely on insights gained from problem analysis to perform divide and conquer. An interesting and powerful consequence is that a solution to a problem using structural recursion may be refined/improved based on insights gained to use other forms of recursion.

You may already have butterflies in your stomach anticipating a wealth of knowledge from the pages of this book. If that is the case, then you are on your way. Enthusiasm for knowledge and understanding is essential for a problem solver. Problem solving, however, can and ought to also be fun. To this end, this book promises to design and implement a video game using modern Artificial Intelligence techniques with you. To achieve this, however, there is a great deal about problem solving and programming you must learn. The game is developed using iterative refinement. That is, as your problem solving and programming knowledge grows, improved versions of the game are developed. Buckle up for fascinating and fun journey to expand your mind!

1 The Parts of the Book

The book is divided into four parts. Part I presents introductory material. It starts by reviewing the basic steps of a design recipe. It does so using a problem solved using structural recursion on a list. It then proceeds to review code refactoring—the restructuring of a function without changing its external behavior. Refactoring is a common technique used to refine programs when a better or more elegant way is found to solve a problem. For example, a problem involving a list may be solved using structural recursion and explicit use of `first` and `rest`. The solution may be refactored to eliminate low-level functions like `first` and `rest` by using a `match` expression. In turn, this solution may be refactored to eliminate recursive calls by using abstract functions like `map` and `filter`. Part I then moves to review abstract running time. In addition, this part introduces the N-Puzzle problem—the video game developed throughout the book—and introduces the use of randomness in problem solving.

Part II explores a new type of recursion called generative recursion. Instead of exploiting the structure of the data to make recursive calls, this type of recursion creates new instances of the data to make recursive calls. The study of generative recursion navigates the reader through examples involving fractal image generation, efficient sorting, and efficient searching techniques such as binary, depth-first, and breadth-first search. This part concludes presenting two refinements to the N-Puzzle game using generative recursion and the problems that they have including the loss of knowledge. Throughout, complexity analysis and empirical experimentation are used to evaluate solutions.

Part III explores a new type of recursion called accumulative (or accumulator) recursion. Accumulative recursion introduces one or more accumulators to a function designed using structural or generative recursion. Accumulators are used to solve the loss of knowledge problem or to make programs more efficient. Examples used include finding a path in a graph, improving insertion sorting, and list-folding operations. The study of list-folding operations leads to new abstract functions with an accumulator: `foldl` and `foldr`. The expertise developed using accumulative recursion is used to refine the N-Puzzle game to perform a heuristic search using the A^* algorithm—an algorithm used in Artificial Intelligence. Part III ends with a chapter introducing an important and powerful program transformation called continuation-passing style. Continuation-passing style allows programmers and compilers to optimize programs. Throughout, complexity analysis and empirical experimentation are used to evaluate solutions.

Part IV explores mutation. Mutation (or changing the value of a state variable) allows different parts of a program that do not call each other to share values. Interestingly enough, most textbooks on programming that use mutation fail to mention this. Abstracting over state variables leads to interfaces and object-oriented programming. The use of mutation, however, comes at a heavy price: the loss of referential transparency. That is, `(f x)` is not always equal to `(f x)`. This means programmers must be disciplined about the order in which mutations are done because knowing that a program works is suddenly much harder. To aid you in properly sequencing mutations, this part of the book teaches you about Hoare Logic and program correctness. In addition, it introduces vectors, vector processing, and in-place operations. Part IV ends by presenting a solution to the chicken or egg paradox in programming. Throughout, complexity analysis and empirical experimentation are used to evaluate solutions.

2 Acknowledgments

This book is the product of over 10 years of work at Seton Hall University building on the shoulders of giants in Computer Science and on the shoulders of Seton Hall undergraduate CS1 and CS2 students. A heartfelt thank you is offered to all the students that throughout the years helped me understand

what works and does not work when teaching program design. Many of the giants in Computer Science that have informed my teaching efforts are members of the PLT research group, especially those responsible for developing the student programming languages used in this textbook and for penning *How to Design Programs*—an inspiration for *Animated Problem Solving* and for *Animated Programming*. It is impossible not to explicitly express my heart-felt appreciation for Matthias Felleisen from Northeastern University and Shriram Krishnamurthi from Brown University for having my back, for debating with me, and for encouraging the work done at Seton Hall University—all of which led to this textbook.

There are many other professional colleagues that deserve credit for inspiring the lessons found in this textbook. Chief among them is Doug Troeger, my Ph.D. advisor, from City College of New York (CCNY). Together, we taught CCNY undergraduates about program correctness. Some of the material in Part IV of this textbook is inspired by those efforts. A great deal of the material in this textbook is based on my Computer Science Education peer-reviewed publications. I was mostly blessed with thoughtful and conscientious reviewers that offered honest feedback on the good and the bad, but that always made an effort to provide thought-provoking comments and suggestions for future work. Collectively, they have also influenced the lessons in this book. I am deeply appreciative to the venues that have published my articles such as the Trends in Functional Programming in Education Workshop, the Journal of Functional Programming , and the Trends in Functional Programming Symposium.

Finally, there is a gifted group of individuals that have been or still are invaluable in making the courses taught using the material in this textbook successful: my undergraduate research/teaching assistants. They have been responsible for making sure I explain the material clearly and for helping answer student questions using their own perspective on the material. This group includes: Shamil Dzhatdoyev, Josie Des Rosiers, Nicholas Olson, Nicholas Nelson, Lindsey Reams, Craig Pelling, Barbara Mucha, Joshua Schappel, Sachin Mahashabde, Rositsa Abrasheva, Isabella Felix, Sena Karsavran, and Julia "Ohio" Wilkins. Their feedback and the feedback they collected from enrolled students have influenced every topic presented. In closing, my appreciation goes out to Seton Hall University and its Department of Computer Science for supporting the development of the work presented in this textbook.

South Orange, NJ, USA Marco T. Morazán

Contents

Preface vii
1 The Parts of the Book viii
2 Acknowledgments ix

Part I Basic Problem Solving and Program Design

1 The Science of Problem Solving 3
3 The Design Recipe 4
4 Computing the Area of a Triangle 4
4.1 Exercises 7
5 Doubling a List of Numbers 7
5.1 Step 1: Data Analysis and Design Idea 8
5.2 Step 2: Sample Expressions 9
5.3 Step 3: Differences Among Sample Expressions 10
5.4 Steps 4–5: Signature, Purpose Statement, and Function Header 10
5.5 Step 6: Tests 10
5.6 Step 7: Function Body 11
5.7 Step 8: Run Tests 11
5.8 Exercises 13
6 Code Refactoring 13
6.1 Exercises 15
7 Abstract Running Time 15
7.1 Exercises 18
8 What Have We Learned in This Chapter? 18

2 The N-Puzzle Problem 21
9 The `world` and the `run` Function 22
10 Useful Constants 26
11 The `draw-world` Handler 30
12 The `game-over?` Handler 32

13 The `process-key` Handler 36
13.1 The Design of `vk?` 39
13.2 The Design of `process-vk` 39
14 What Have We Learned in This Chapter? 45

3 Randomness 47
15 `ISL+`'s `random` Function 47
16 N-Puzzle Version 1 50
17 Generating Random Passwords 54
18 Distributed Fortune Teller Game 58
18.1 Design Recipe for Distributed Computing 58
18.2 The Components 59
18.3 Data Definitions 60
18.4 Communication Protocol 64
18.5 Marshaling and Unmarshaling Functions 67
18.6 The Player Component 67
18.7 The Server Component 72
19 What Have We Learned in This Chapter? 76

Part II Generative Recursion

4 Introduction to Generative Recursion 81
20 Generating a Nested Square Image 82
21 The Design Recipe for Generative Recursion 86
22 All Primes ≤ `n` 91
23 What Have We Learned in This Chapter? 101

5 Sorting 103
24 Insertion Sorting 104
25 Quick Sorting 106
25.1 Problem Analysis 107
25.2 Sample Expressions and Differences 107
25.3 Signature, Statements, and Function Header 109
25.4 Tests 109
25.5 Function Body 110
25.6 Termination Argument 111
25.7 Performance 111
25.8 Complexity Analysis 111
26 Merge Sorting 114
26.1 Problem Analysis 115
26.2 The `merge-sorting` Function 115
26.3 The `merge-sort-helper` Function 117
26.4 The `merge-neighs` Function 121
26.5 The `merge` Function 124

26.6 Performance 127
26.7 Complexity Analysis 128
27 What Have We Learned in This Chapter? 130

6 Searching 131
28 Linear Searching 131
28.1 Problem Analysis 132
28.2 Sample Expressions and Differences 132
28.3 Signature, Purpose, and Function Header 134
28.4 Tests 134
28.5 Function Body 135
28.6 Performance and Complexity 135
29 Binary Search 136
29.1 The `binary-search` Function 137
29.2 The `bin-search` Function 140
29.3 Termination Argument 144
29.4 Performance and Complexity 144
30 Trees 146
31 Depth-First Search 151
31.1 Problem Analysis 151
31.2 Sample Expressions and Differences 152
31.3 Signature, Purpose, and Function Header 152
31.4 Tests 152
31.5 The Function Body 153
31.6 The `node-dfs-contains?` Function 153
31.7 Signature, Purpose, and Function Definition 155
31.8 Performance and Complexity 156
32 Breadth-First Search 157
32.1 Problem Analysis for `ton-bfs-contains?` 157
32.2 Sample Expressions and Differences for `ton-bfs-contains?` 160
32.3 Tests for `ton-bfs-contains?` 161
32.4 Function Definition for `ton-bfs-contains?` 161
32.5 Problem Analysis for `bfs-helper` 162
32.6 Sample Expressions and Differences for `bfs-helper` 162
32.7 Tests for `bfs-helper` 163
32.8 Signature, Statements, and Function Definition for `bfs-helper` 164
32.9 Performance and Complexity 165
33 What Have We Learned in This Chapter? 167

7 N-Puzzle Version 2 169
34 Design and Implementation of `make-move` 170
34.1 Problem Analysis 170
34.2 Sample Expressions and Differences 171
34.3 Signature, Purpose, and Function Header 172
34.4 Tests 172
34.5 Function Body 173
35 Design and Implementation of `find-solution` 173
35.1 Problem Analysis 173
35.2 Sample Expressions and Differences 174
35.3 Signature, Statements, and Function Header 175
35.4 Tests 176
35.5 Function Body 177
35.6 Termination Argument 178
36 New Tests for `process-key` 178
37 A Bug: Infinite Recursion 179
38 What Have We Learned in This Chapter? 182

8 N-Puzzle Version 3 185
39 The Design of `make-move` 185
39.1 Problem Analysis 185
39.2 Sample Expressions and Differences 186
39.3 Signature, Purpose, and Function Header 186
39.4 Tests 187
39.5 Function Body 188
40 The Design of `find-solution-bfs` 189
40.1 Problem Analysis 189
40.2 Sample Expressions and Differences 191
40.3 Signature, Statements, and Function Header 193
40.4 Tests 193
40.5 Function Body 195
40.6 Termination Argument 195
41 Performance 196
42 What Have We Learned in This Chapter? 199

Part III Accumulative Recursion

9 Accumulators 203
43 Running Totals 204
43.1 Problem Analysis for `lon-running-totals` 204
43.2 Sample Expressions and Differences for `lon-running-totals` 205
43.3 Signature, Function Definition, and Tests for `lon-running-totals` 205

43.4 Problem Analysis for `lon-running-totals-helper` ... 206
43.5 Sample Expressions and Differences for `lon-running-totals-helper` ... 206
43.6 Signature, Function Definition, and Tests for `lon-running-totals-helper` ... 207
43.7 The `lon-sum` Function ... 208
44 Running Totals Using an Accumulator ... 209
44.1 Problem Analysis ... 209
44.2 Sample Expressions and Differences for `lon-running-totals-v2` ... 210
44.3 Function Definition for `lon-running-totals-v2` ... 210
44.4 Tests for `lon-running-totals-v2` ... 211
44.5 Problem Analysis for `lon-running-totals-helper-v2` ... 211
44.6 Sample Expressions and Differences for `lon-running-totals-helper-v2` ... 212
44.7 Signature, Statements, and Function Header for `lon-running-totals-helper-v2` ... 213
44.8 Tests for `lon-running-totals-helper-v2` ... 214
44.9 Function Body for `lon-running-totals-helper-v2` ... 214
45 Performance and Complexity Analysis ... 215
46 Finding a Path in a Directed Graph ... 217
46.1 Data Analysis ... 218
46.2 Design and Implementation of `find-path` ... 221
46.3 Design and Implementation of `find-path-acc` ... 224
46.4 Design and Implementation of `find-path-from-neighbors` ... 228
46.5 Termination Argument ... 231
47 Revisiting Insertion Sorting ... 232
47.1 The Redesign of `insert` ... 233
47.2 The Redesign of `insertion-sorting` ... 237
47.3 Performance and Complexity Analysis ... 240
48 What Have We Learned in This Chapter? ... 242

10 N-Puzzle Versions 4 and 5 ... 245
49 N-Puzzle Version 4 ... 245
49.1 The Design and Implementation of `find-solution` ... 246
49.2 The `find-solution-from-any-succ` Design and Implementation ... 251
49.3 Termination Argument ... 258
50 N-Puzzle Version 5 ... 260
50.1 Problem Analysis ... 260

50.2 Sample Expressions and Differences 261
50.3 Signature, Statements, and Function Header 263
50.4 Tests 264
50.5 Function Body 266
50.6 Termination Argument 266
51 Complexity and Performance 267
52 What Have We Learned in This Chapter? 270

11 Iteration 271
53 List-Folding Operations from the Left 274
53.1 Summing a List of Numbers 274
53.2 Reversing a List 279
54 List-Folding Operations from the Right 285
54.1 Computing String Lengths from a List of Strings .. 287
54.2 Summing a List of Numbers Revisited 292
55 Functional Abstraction 293
56 Abstraction over Left to Right Accumulating Folding Functions 294
56.1 The Abstraction 294
56.2 Performance 297
57 Abstraction Over Right to Left Accumulating Folding Functions 298
58 What Have We Learned in This Chapter? 301

12 N-Puzzle Version 6 303
59 The Manhattan Distance Heuristic 304
60 Problem Analysis 307
61 Sample Expressions and Differences for `find-solution-a*` 309
62 Signature, Statements, and Function Header 314
63 Tests for `find-solution-a*` 315
64 Function Body for `find-solution-a*` 316
65 Termination Argument 317
66 Performance 317
67 What Have We Learned in This Chapter? 320

13 Continuation-Passing Style 321
68 Accumulating Control 322
69 The CPS Design Recipe 326
70 Computing Fibonacci Numbers 327
70.1 Transforming to `CPS` 328
70.2 Performance 330
70.3 Going Beyond the Design Recipe 331
71 Revisiting List Reversal 333
72 What Have We Learned in This Chapter? 338

Part IV Mutation

14 Sharing Values 341
73 `set!` and `begin` Expressions 343
74 Design Recipe for Mutators 348
75 A Bank Account 349
75.1 Problem and Data Analysis 350
75.2 State Variable Definitions 350
75.3 State Variable Initializers 350
75.4 The Mutator for Deposits 352
75.5 The Mutator for Withdrawals 353
75.6 The Observer for Getting the Balance 355
76 Abstraction Over State Variables 357
76.1 Bank Account State Variables and Interface 359
76.2 Bank Account Class Template 359
76.3 Bank Account Message-Passing Function Design 359
76.4 Bank Account Auxiliary Function Design 361
76.5 Bank Account Wrapper Functions and Tests 362
77 A Design Recipe for Interfaces 366
78 Mutation and Structures 369
79 The Concept of Equality 371
80 What Have We Learned in This Chapter? 374

15 Mutation Sequencing 377
81 Hoare Logic 380
81.1 Using Hoare Triples 383
81.2 Imperative Code Debugging 388
82 New Syntax: `while` Loops 392
82.1 Syntax and Semantics 392
82.2 Transformation from an Imperative Recursive Function to a `while` Loop 393
83 A Design Recipe for `while loops` 397
84 Determining if an Interval Contains a Prime 400
84.1 Problem Analysis 401
84.2 Signature, Statements, and Function Header 401
84.3 Tests 401
84.4 Loop Invariant 402
84.5 Function Body 403
84.6 The `begin` Expression 404
84.7 Post-Loop Code 406
84.8 Auxiliary Functions 406
84.9 Termination Argument 407
84.10 Run Tests 407

85 Finding the Maximum in a List of Numbers 407
85.1 Problem Analysis 407
85.2 Signature, Statements, and Function Header 409
85.3 Tests 409
85.4 Loop Invariant 409
85.5 Function Body 411
85.6 The `begin` Expression 411
85.7 Post-Loop Code 414
85.8 Termination Argument 414
85.9 Run Tests 414
86 What Have We Learned in This Chapter? 416

16 Vectors 419
87 Vector Basics 420
88 Vector Processing Using Structural Recursion 422
88.1 The Dot Product of Two Vectors of Numbers 423
88.2 Merging Two Sorted Vectors 427
89 Vector Processing Using Generative Recursion: Revisiting the Sieve of Eratosthenes 442
89.1 Problem Analysis 443
89.2 Signature, Statements, and Function Header 443
89.3 Tests 444
89.4 Loop Invariant 444
89.5 Function's Local Definitions 446
89.6 The `local`-expression's Body 448
89.7 Post-Loop Code 449
89.8 Auxiliary Functions 450
89.9 Termination Argument and Running Tests 452
89.10 Complexity and Performance 452
90 What Have We Learned in This Chapter? 453

17 In-Place Operations 455
91 Reversing a Vector 455
91.1 Problem Analysis 456
91.2 The `reverse-vector-in-place!` Mutator 457
91.3 Signature and Statements 457
91.4 The `rev-vector!` Mutator 458
91.5 Function Body 460
91.6 The `swap!` Mutator and Running Tests 460
91.7 Performance 461
92 In-Place Quick Sorting 462
92.1 The `qs-in-place!` Mutator 463
92.2 The `qs-aux!` Mutator 465
92.3 The `partition!` Mutator 467
92.4 The Design of `first<=` 471

92.5 The Design of `first>` ... 472
92.6 Completing the Design ... 473
93 In-Place Heap Sorting ... 473
93.1 Heaps ... 474
93.2 Sorting ... 474
93.3 Mapping a Heap to a Vector ... 477
93.4 The `heap-sort-in-place!` Mutator ... 478
93.5 The `sorter!` Mutator ... 481
93.6 The `trickle-down!` Mutator ... 483
93.7 The `heapify!` Mutator ... 485
94 Empirical Project ... 491
94.1 Radix Sorting ... 493
94.2 The Project ... 495
95 What Have We Learned in This Chapter? ... 496

18 The Chicken and the Egg Paradox ... 499
96 The Paradox in Programming ... 500
97 Solving the Paradox ... 501
97.1 Problem Analysis ... 501
97.2 Sample Expressions and Differences ... 502
97.3 Signature, Statements, and Function Header ... 502
97.4 Tests ... 503
97.5 Function Body ... 504
98 Adding Clients to a Bank ... 504
98.1 Problem and Data Analysis ... 504
98.2 State Variable Definition ... 505
98.3 Bank Initializer ... 505
98.4 The `add-account!` Mutator ... 505
99 What Have We Learned in This Chapter? ... 509

Part V Epilogue

19 Where to Go from Here ... 513