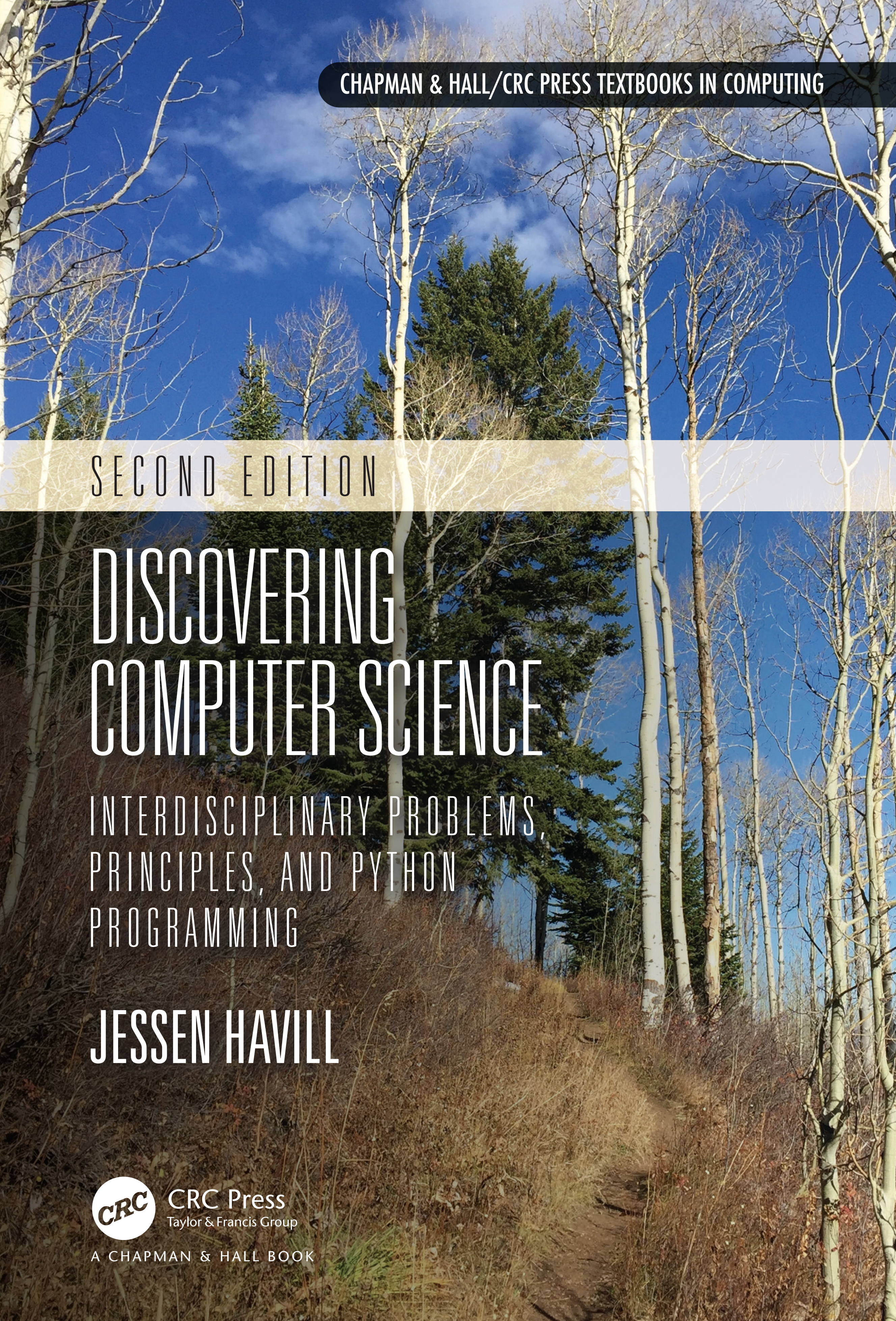

CHAPMAN & HALL/CRC PRESS TEXTBOOKS IN COMPUTING

SECOND EDITION

DISCOVERING COMPUTER SCIENCE

INTERDISCIPLINARY PROBLEMS, PRINCIPLES, AND PYTHON PROGRAMMING

JESSEN HAVILL

CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

Contents

*Sections with *** in lieu of a page number are available on the book website.

Preface

In my view, an introductory computer science course should strive to accomplish three things. First, it should demonstrate to students how computing has become a powerful mode of inquiry, and a vehicle of discovery, in a wide variety of disciplines. This orientation is also inviting to students of the natural and social sciences, and the humanities, who increasingly benefit from an introduction to computational thinking, beyond the limited "black box" recipes often found in manuals and "Computing for X" books. Second, the course should engage students in computational problem solving, and lead them to discover the power of abstraction, efficiency, and data organization in the design of their solutions. Third, the course should teach students how to implement their solutions as computer programs. In learning how to program, students more deeply learn the core principles, and experience the thrill of seeing their solutions come to life.

Unlike most introductory computer science textbooks, which are organized around programming language constructs, I deliberately lead with interdisciplinary problems and techniques. This orientation is more interesting to a more diverse audience, and more accurately reflects the role of programming in problem solving and discovery. A computational discovery does not, of course, originate in a programming language feature in search of an application. Rather, it starts with a compelling problem which is modeled and solved algorithmically, by leveraging abstraction and prior experience with similar problems. Only then is the solution implemented as a program.

Like most introductory computer science textbooks, I introduce programming skills in an incremental fashion, and include many opportunities for students to practice them. The topics in this book are arranged to ease students into computational thinking, and encourage them to incrementally build on prior knowledge. Each chapter focuses on a general class of problems that is tackled by new algorithmic techniques and programming language features. My hope is that students will leave the course, not only with strong programming skills, but with a set of problem solving strategies and simulation techniques that they can apply in their future work, whether or not they take another computer science course.

I use Python to introduce computer programming for two reasons. First, Python's intuitive syntax allows students to focus on interesting problems and powerful principles, without unnecessary distractions. Learning how to think algorithmically is hard enough without also having to struggle with a non-intuitive syntax. Second, the expressiveness of Python (in particular, low-overhead lists and dictionaries) expands tremendously the range of accessible problems in the introductory course.

Teaching with Python over the last fifteen years has been a revelation; introductory computer science has become fun again.

Changes in the second edition

In this comprehensive, cover-to-cover update, some sections were entirely rewritten while others saw only minor revisions. Here are the highlights:

Problem solving The new first chapter, *How to Solve It*, sets the stage by focusing on Polya's elegant four-step problem solving process, adapted to a computational framework. I introduce informal pseudocode, functional decomposition, hand-execution with informal trace tables, and testing, practices that are now carried on throughout the book. The introduction to Python (formally Chapter 2) is integrated into this framework. Chapter 7, *Designing Programs*, from the first edition has been eliminated, with that material spread out more naturally among Chapters 1, 5, and 6 in the second edition.

Chapter 2, *Visualizing Abstraction* (based on the previous Chapter 3), elaborates on the themes in Chapter 1, and their implementations in Python, introducing turtle graphics, functions, and loops. The new Chapter 3, *Inside a Computer* (based on the previous Sections 1.4 and 2.5), takes students on a brief excursion into the simple principles underlying how computers work.

Online materials To reduce the size of the printed book, we have moved some sections and all of the projects online. These sections are marked in the table of contents with ***. Online materials are still indexed in the main book for convenience.

Exercises I've added exercises to most sections, bringing the total to about 750. Solutions to exercises marked with an asterisk are available online for both students and self-learners.

Digital humanities The interdisciplinary problems in the first edition were focused primarily in the natural and social sciences. In this edition, especially in Chapters 1, 6, and 7, we have added new material on text analysis techniques commonly used in the "digital humanities."

Object-oriented design Chapter 12 begins with a new section to introduce object-oriented design in a more concrete way through the development of an agent-based simulation of a viral epidemic. The following sections flesh out more details on how to implement polymorphic operators and collection classes.

Book website

Online materials for this book are available at

`https://www.discoveringCS.net`.

Here you will find

- additional "optional" sections, marked with an asterisk in the main text,
- over thirty interdisciplinary programming projects,
- solutions to selected exercises,
- programs and data files referenced in the text, exercises, and projects, and
- pointers for further exploration and links to additional documentation.

To students

Active learning Learning how to solve computational problems and implement them as computer programs requires daily practice. Like an athlete, you will get out of shape and fall behind quickly if you skip it. There are no shortcuts. Your instructor is there to help, but he or she cannot do the work for you.

With this in mind, it is important that you type in and try the examples throughout the text, and then go beyond them. Be curious! There are numbered "Reflection" questions throughout the book that ask you to stop and think about, or apply, something that you just read. Often, the question is answered in the book immediately thereafter, so that you can check your understanding, but peeking ahead will rob you of an important opportunity.

Further discovery There are many opportunities to delve into topics more deeply. "Tangent" boxes scattered throughout the text briefly introduce related, but more technical or applied, topics. For the most part, these are not strictly required to understand what comes next, but I encourage you to read them anyway. In the "Summary and Further Discovery" section of each chapter, you can find both a high-level summary of the chapter and additional pointers to explore chapter topics in more depth.

Exercises and projects At the end of most sections are several programming exercises that ask you to further apply concepts from that section. Often, the exercises assume that you have already worked through all of the examples in that section. Solutions to the starred exercises are available on the book website. There are also more involved projects available on the book website that challenge you to solve a variety of interdisciplinary problems.

No prerequisites The book assumes no prior knowledge of computer science. However, it does assume a modest comfort with high school algebra. In optional sections, trigonometry is occasionally mentioned, as is the idea of convergence to a limit, but these are not relevant to understanding the main topics in the book.

Have fun! Programming and problem solving should be a fun, creative activity. I hope that this book sparks your curiosity and love of learning, and that you enjoy the journey as much as I have enjoyed writing this book.

To instructors

This book is appropriate for a traditional CS1 course for majors, a CS0 course for non-majors (at a slower pace and omitting more material), or a targeted introductory computing course for students in the natural sciences, social sciences, or humanities.

The approach is gentle and holistic, introducing programming concepts in the context of interdisciplinary problems. We start with problem-solving, featuring pseudocode and hand-execution with trace tables, and carry these techniques forward, especially in the first half of the book.

Problem focus Most chapters begin with an interesting problem, and new concepts and programming techniques are introduced in the context of solving it. As new techniques are introduced, students are frequently challenged to re-solve old problems in different ways. They are also encouraged to reuse their previous functions as components in later programs.

Reflection questions, exercises, and projects "Reflection" questions are embedded in every section to encourage active reading. These may also be assigned as "reading questions" before class. The end-of-section exercises are appropriate for regular homework, and some more complex ones may form the basis of longer-term assignments. The book website also hosts a few dozen interdisciplinary projects that students may work on independently or in pairs over a longer time frame. I believe that projects like these are crucial for students to develop both problem solving skills and an appreciation for the many fascinating applications of computer science.

Additional instructor resources All of the reflection questions and exercises are available to instructors as Jupyter notebooks. Solutions to all exercises and projects are also available. Please visit the publisher's website to request access.

Python coverage This book is not intended to be a Python manual. Some features of the language were intentionally omitted because they would have muddled the core problem solving focus or are not commonly found in other languages that students may see in future CS courses (e.g., simultaneous swap, chained comparisons, `zip`, `enumerate` in `for` loops).

Topic coverage There is more in this book than can be covered in a single semester, giving instructors the opportunity to tailor the content to their particular situation

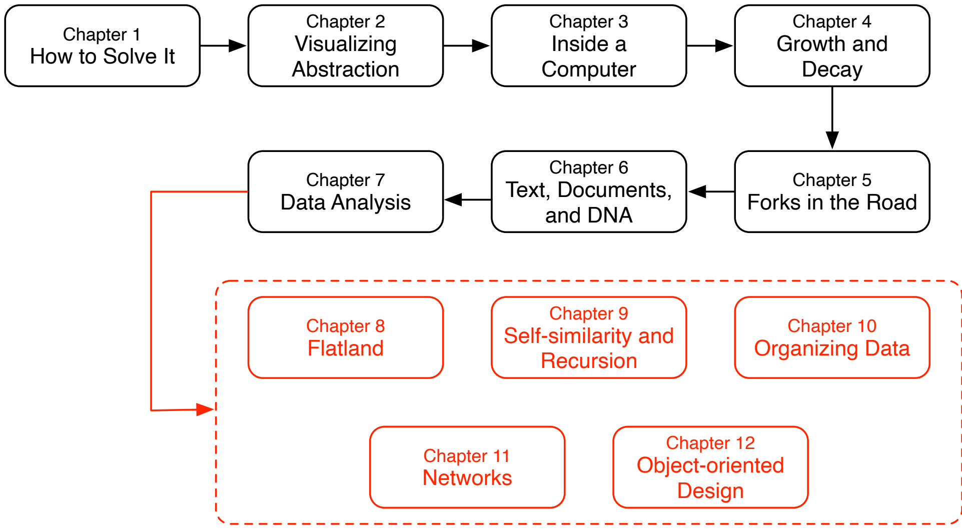

Figure 1 An overview of chapter dependencies.

and interests. As illustrated in Figure 1, Chapters 1–7 form the core of the book, and should be covered sequentially. The remaining chapters can be covered, partially or entirely, at your discretion, although I would expect that most instructors will cover at least parts of Chapters 8–10, and 12 if the course covers object-oriented design. Chapter 11 introduces social network graphs and small-world and scale-free networks as additional powerful applications of dictionaries, and may come any time after Chapter 7. Sections marked with an asterisk are optional, in the sense that they are not assumed for future sections in that chapter. When exercises and projects depend on optional sections, they are also marked with an asterisk, and the dependency is stated at the beginning of the project.

Chapter outlines The following tables provide brief overviews of what is available in each chapter. Each table's three columns, reflecting the three parts of the book's subtitle, provide three lenses through which to view the chapter.

1 How to Solve It

Sample problems	Principles	Programming
• reading level	• problems, input/output	• `int`, `float`, `str` types
• counting syllables, words	• functional abstraction	• arithmetic
• sphere volume	• functional decomposition	• assignment
• digital music	• top-down design	• variable names
• search engines	• bottom-up implementation	• calling built-in functions
• GPS devices	• algorithms and programs	• using strings
• phone trees	• pseudocode	• string operators
• wind chill	• names as references	• `print` and `input`
• compounding interest	• trace tables	
• Mad Libs	• constant- vs. linear-time	

2 Visualizing Abstraction

Sample problems	Principles	Programming
• visualizing earthquakes • drawing flowers • random walks • ideal gas • groundwater flow • demand functions • reading level	• using abstract data types • creating functional abstractions • functional decomposition • bottom-up implementation • turtle graphics • trace tables with loops	• using classes and objects • `turtle` module • `for` loops (`range` and lists) • using and writing functions • `return` vs. `print` • namespaces and scope • docstrings and comments • self-documenting code • program structure

3 Inside a Computer

Principles	Programming
• computer organization • machine language • binary representations • computer arithmetic • finite precision, error propagation • Boolean logic, truth tables, logic gates • Turing machines, finite state machines	• `int` and `float` types • arithmetic errors • true vs. floor division

4 Growth and Decay

Sample problems	Principles	Programming
• population models • network value • demand and profit • loans and investing • bacterial growth • radiocarbon dating • epidemics (SIR, SIS) • diffusion models	• accumulators • list accumulators • data visualization • conditional iteration • classes of growth • continuous models • accuracy vs. time • numerical approximation	• `for` loops, `range` • format strings • `matplotlib.pyplot` • appending to lists • `while` loops

5 Forks in the Road

Sample problems	Principles	Programming
• random walks • Monte Carlo simulation • guessing games • polling and sampling • particle escape	• random number generators • simulating probabilities • flag variables • using distributions • DeMorgan's laws • defensive programming • pre- and post-conditions • unit testing	• `random` module • `if/elif/else` • comparison operators • Boolean operators • short circuit evaluation • predicate functions • `assert`, `isinstance` • catching exceptions • histograms • `while` loops

6 Text, Documents, and DNA

Sample problems	Principles	Programming
• text analysis • word frequency trends • checksums • concordances • dot plots, plagiarism • congressional votes • genomics	• functional decomposition • unit testing • ASCII, Unicode • linear-time algorithms • time complexity • linear search • string accumulators	• `str` class and methods • iterating over strings, lists • indexing and slicing • iterating over indices • creating a module • text files and the web • `break` • nested loops

7 Data Analysis

Sample problems	Principles	Programming
• word, bigram frequencies • smoothing data • 100-year floods • traveling salesman • meteorite sites • zebra migration • tumor diagnosis • supply and demand • voting methods	• histograms • hash tables • tabular data files • efficient algorithm design • linear regression • k-means clustering • heuristics	• `list` class • indexing and slicing • list operators and methods • reading CSV files • modifying lists in place • list parameters • tuples • list comprehensions • dictionaries

8 Flatland

Sample problems	Principles	Programming
• earthquake data • Game of Life • image filters • racial segregation • ferromagnetism • dendrites • epidemics • tumor growth	• 2-D data • cellular automata • digital images • color models	• lists of lists • nested loops • 2-D data in a dictionary

9 Self-similarity and Recursion

Sample problems	Principles	Programming
• fractals • cracking passwords • Tower of Hanoi • maximizing profit • navigating a maze • Lindenmayer systems • gerrymandering • percolation	• self-similarity • recursion • linear search • recurrence relations • divide and conquer • depth-first search • grammars	• writing recursive functions • divide and conquer • backtracking

10 Organizing Data

Sample problems	Principles	Programming
• spell check • querying data sets	• binary search • quadratic-time sorting • parallel lists • merge sort • recurrence relations • intractability, P=NP?	• nested loops • writing recursive functions

11 Networks

Sample problems	Principles	Programming
• social media, web graphs • diffusion of ideas • epidemics • Oracle of Bacon	• graphs • adjacency list, matrix • breadth-first search • queues • shortest paths • depth-first search • small-world networks • scale-free networks • uniform random graphs	• dictionaries

12 Object-oriented Design

Sample problems	Principles	Programming
• epidemic simulation • data sets • genomic sequences • rational numbers • flocking behavior • slime mold aggregation	• abstract data types • encapsulation • polymorphism • data structures • stacks • hash tables • agent-based simulation • swarm intelligence	• object-oriented design • writing classes • special methods • overriding operators • modules

Software assumptions

To follow along in this book and complete the exercises, you will need to have installed Python 3.6 or later on your computer, and have access to IDLE or another programming environment. The book also assumes that you have installed the `matplotlib.pyplot` and `numpy` modules. The easiest way to get this software is to install the free open source Anaconda distribution from `http://www.anaconda.com`.

Errata

While I (and my students) have ferreted out many errors, readers will inevitably find more. You can find an up-to-date list of errata on the book website. If you find an error in the text or have another suggestion, please let me know at `havill@denison.edu`.