CHAPMAN & HALL/CRC
TEXTBOOKS IN COMPUTING

DISCOVERING COMPUTER SCIENCE

Interdisciplinary Problems, Principles, and Python Programming

Jessen Havill
Denison University
Granville, Ohio, USA

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

Contents

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, who increasingly benefit from an introduction to computational thinking, beyond the limited “black box” recipes often found in manuals. 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 ten years has been a revelation; introductory computer science has become fun again.

Web resources

The text, exercises, and projects often refer to files on the book's accompanying web site, which can be found at

`http://discoverCS.denison.edu .`

This web site also includes pointers for further exploration, links to additional documentation, and errata.

To students

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.

There are many opportunities to delve into topics more deeply. Boxes scattered throughout the text briefly introduce related, but more technical, 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 "Further discovery" section of each chapter, you can find additional pointers to explore chapter topics in more depth.

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. All later chapters conclude with a selection of more involved interdisciplinary projects that you may be asked by your instructor to tackle.

The book assumes no prior knowledge of computer science. However, it does assume a modest comfort with high school algebra and mathematical functions. Occasionally, trigonometry is mentioned, as is the idea of convergence to a limit, but these are not crucial to an understanding of the main topics in this book.

To instructors

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

As suggested above, I emphasize computer science principles and the role of abstraction, both functional and data, throughout the book. I motivate functions as implementations of functional abstractions, and point out that strings, lists, and dictionaries are all abstract data types that allow us to solve more interesting problems than would otherwise be possible. I introduce the idea of time complexity

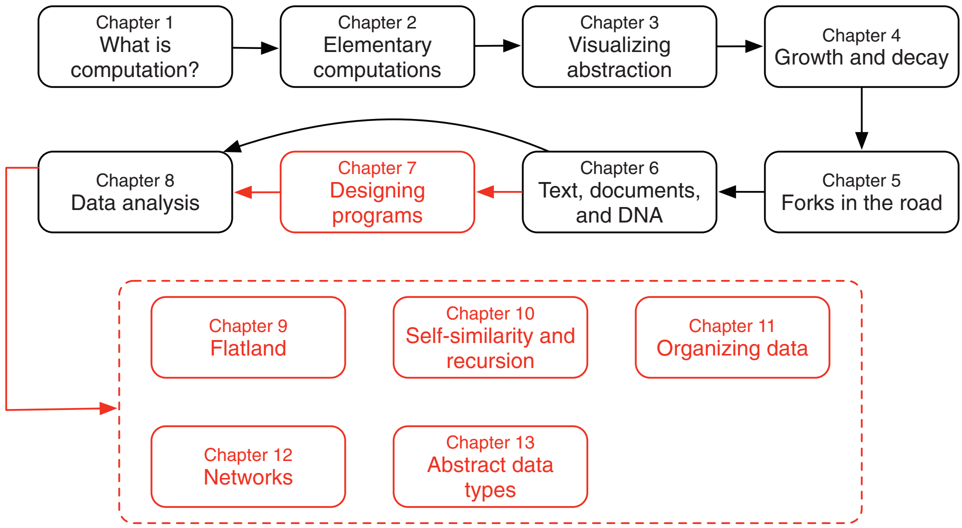

Figure 1 An overview of chapter dependencies.

intuitively, without formal definitions, in the first chapter and return to it several times as more sophisticated algorithms are developed. The book uses a spiral approach for many topics, returning to them repeatedly in increasingly complex contexts. Where appropriate, I also weave into the book topics that are traditionally left for later computer science courses. A few of these are presented in boxes that may be covered at your discretion. None of these topics is introduced rigorously, as they would be in a data structures course. Rather, I introduce them informally and intuitively to give students a sense of the problems and techniques used in computer science. I hope that the tables below will help you navigate the book, and see where particular topics are covered.

This book contains over 600 end-of-section exercises and over 300 in-text reflection questions that may be assigned as homework or discussed in class. At the end of most chapters is a selection of projects (about 30 in all) 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.

Because this book is intended for a student who may take additional courses in computer science and learn other programming languages, I intentionally omit some features of Python that are not commonly found elsewhere (e.g., simultaneous swap, chained comparisons, `enumerate` in `for` loops). You may, of course, supplement with these additional syntactical features.

There is more in this book than can be covered in a single semester, giving an instructor the opportunity to tailor the content to his or her particular situation and interests. Generally speaking, as illustrated in Figure 1, Chapters 1–6 and 8 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 7, 10, 11, and 13. Chapter 7 contains additional material on program design, including design by contract, assertions and unit testing that may be skipped without consequences for later chapters. Chapters 9–13 are, more or less, independent of each other. Sections marked with an asterisk are optional, in the sense that they are not assumed for future sections in that chapter. When 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 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. The first column lists a selection of representative *problems* that are used to motivate the material. The second column lists computer science *principles* that are introduced in that chapter. Finally, the third column lists Python *programming* topics that are either introduced or reinforced in that chapter to implement the principles and/or solve the problems.

Chaper 1. What is computation?

Sample problems	Principles	Programming
• digital music • search engines • GPS devices • smoothing data • phone trees	• problems, input/output • abstraction • algorithms and programs • computer architecture • binary representations • time complexity • Turing machine	—

Chapter 2. Elementary computations

Sample problems	Principles	Programming
• wind chill • geometry • compounding interest • Mad Libs	• finite precision • names as references • using functional abstractions • binary addition	• `int` and `float` numeric types • arithmetic and the `math` module • variable names and assignment • calling built-in functions • using strings, `+` and `*` operators • `print` and `input`

Chapter 3. Visualizing abstraction

Sample problems	Principles	Programming
• visualizing an archaeological dig • random walks • ideal gas • groundwater flow • demand functions	• using abstract data types • creating functional abstractions • basic functional decomposition	• using classes and objects • `turtle` module • basic `for` loops • writing functions • namespaces • docstrings and comments

Chapter 4. Growth and decay

Sample problems	Principles	Programming
• network value • demand and profit • loans and investing • bacterial growth • radiocarbon dating • diffusion models – SIR, SIS, Bass • competition models – Nicholson-Bailey – Lotka-Volterra – indirect	• accumulators • list accumulators • difference equations • approximating continuous models • accuracy vs. time • error propagation • numerical approximation • classes of growth	• `for` loops • format strings • `range` • `matplotlib` • appending to lists • `while` loops

Chapter 5. Forks in the road

Sample problems	Principles	Programming
• random walks • guessing games • polling and sampling • particle escape	• Monte Carlo simulation • pseudorandom number generators • simulating probabilities • flag variables • using uniform and normal distributions • DeMorgan's laws	• `random` module • `if/elif/else` • comparison operators • Boolean operators • `matplotlib` histograms • `while` loops

Chapter 6. Text, documents, and DNA

Sample problems	Principles	Programming
• word count • textual analysis • parsing XML • checksums • concordances • detecting plagiarism • congressional votes • genomics	• ASCII, Unicode • linear-time algorithms • asymptotic time complexity • linear search • dot plots • string accumulators	• `str` class and methods • iterating over strings • indexing and slices • iterating over indices • reading and writing text files • nested loops

Chapter 7. Designing programs

Sample problems	Principles	Programming
• word frequency analysis	• problem solving • top-down design • pre and postconditions • assertions • unit testing	• `assert` statement • conditional execution of `main` • writing modules

Chapter 8. Data analysis

Sample problems	Principles	Programming
• 100-year floods	• histograms	• `list` class
• traveling salesman	• hash tables	• iterating over lists
• Mohs scale	• tabular data files	• indexing and slicing
• meteorite sites	• efficient algorithms	• list operators and methods
• zebra migration	• linear regression	• lists in memory; mutability
• tumor diagnosis	• k-means clustering	• list parameters
• education levels	• heuristics	• tuples
• supply and demand		• list comprehensions
• voting methods		• dictionaries

Chapter 9. Flatland

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

Chapter 10. Self-similarity and recursion

Sample problems	Principles	Programming
• fractals	• self-similarity	• writing recursive functions
• cracking passwords	• recursion	
• Tower of Hanoi	• linear search	
• maximizing profit	• recurrence relations	
• path through a maze	• divide and conquer	
• Lindenmayer system	• depth-first search	
• electoral districting	• grammars	
• percolation		

Chapter 11. Organizing data

Sample problems	Principles	Programming
• spell check	• binary search	• nested loops
• querying data sets	• recurrence relations	• writing recursive functions
	• basic sorting algorithms	
	• quadratic-time algorithms	
	• parallel lists	
	• merge sort	
	• intractability	
	• P=NP (intuition)	
	• Moore's law	
	• binary search trees	

Chapter 12. Networks

Sample problems	Principles	Programming
• Facebook, Twitter, web graphs • diffusion of ideas • epidemics • Oracle of Bacon	• graphs • adjacency list • adjacency matrix • breadth-first search • distance and shortest paths • depth-first search • small-world networks • scale-free networks • clustering coefficient • uniform random graphs	• dictionaries

Chapter 13. Abstract data types

Sample problems	Principles	Programming
• data sets • genomic sequences • rational numbers • flocking behavior • slime mold aggregation	• abstract data types • data structures • stacks • hash tables • agent-based simulation • swarm intelligence	• 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 3.4 (or later) on your computer, and have access to IDLE or another programming environment. The book also assumes that you have installed the `matplotlib` and `numpy` modules. Please refer to Appendix A for more information.

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 web site. If you find an error in the text or have another suggestion, please let me know at `havill@denison.edu`.