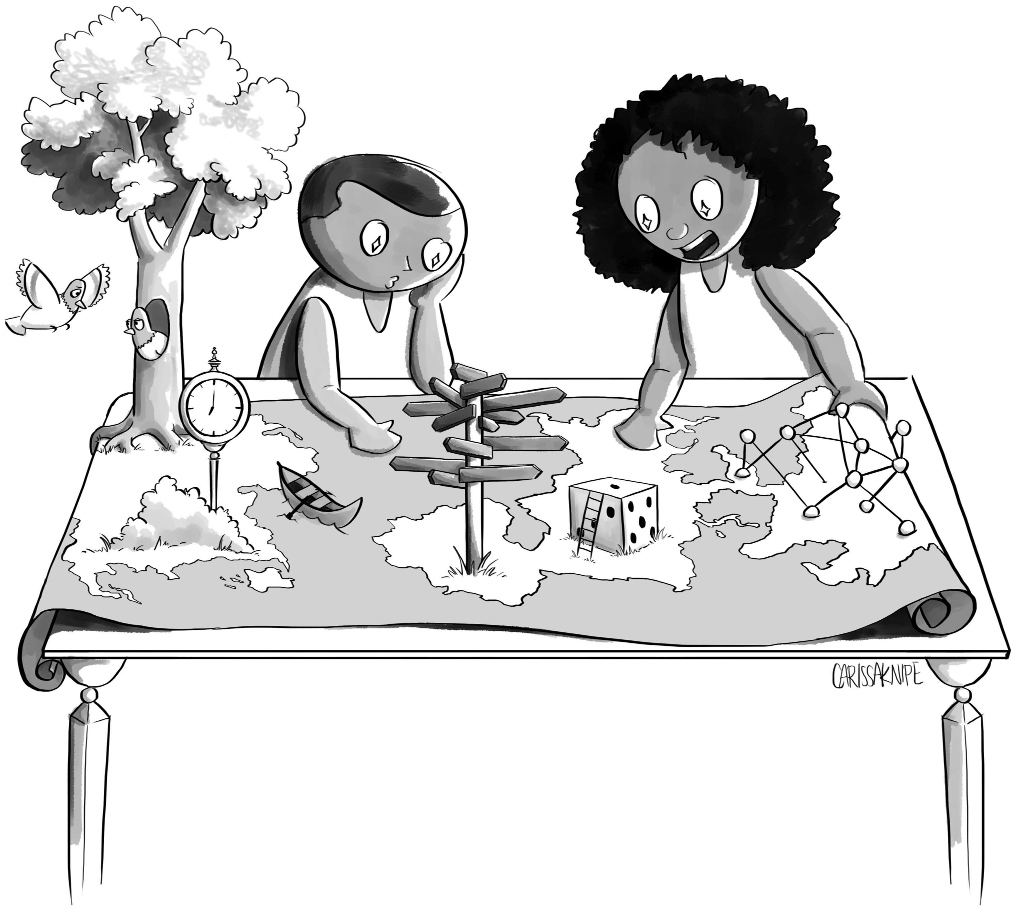
1
On the Point of this Book



In which our heroes decide, possibly encouraged by a requirement for graduation, to set out to explore the world.

Why You Might Care

Just because some of us can read and write and do a little math, that doesn't mean we deserve to conquer the Universe.

Kurt Vonnegut (1922–2007)
Hocus Pocus (1990)

This book is designed for an undergraduate student who has taken a computer science class or three—most likely, you are a sophomore or junior prospective or current computer science major taking your first non-programming-based CS class. If you are a student in this position, you may be wondering why you're taking this class (or why you *have* to take this class!). Computer science students taking a class like this one sometimes don't see why this material has anything to do with computer science—particularly if you enjoy CS because you enjoy programming.

I want to be clear: programming is awesome! I get lost in code all the time—let's not count the number of hours that I spent writing the code to draw the fractals in Figure 5.1 in LaTeX, for example. (LaTeX, the tool used to typeset this book, is the standard typesetting package for computer scientists, and it's actually also a full-fledged, if somewhat bizarre, programming language.)

But there's more to CS than programming. In fact, many seemingly unrelated problems rely on the same sorts of abstract thinking. It's not at all obvious that an optimizing compiler (a program that translates source code in a programming language like C into something directly executable by a computer) would have anything important in common with a program to play chess perfectly. But, in fact, they're both tasks that are best understood using *logic* (Chapter 3) as a central component of any solution. Similarly, filtering spam out of your inbox ("given a message m, should m be categorized as spam?") and doing speech recognition ("given an audio stream s of a person speaking in English, what is the best 'transcript' reflecting the words spoken in s?") are both best understood using *probability* (Chapter 10).

And these, of course, are just examples; there are many, many ways in which we can gain insight and efficiency by thinking more abstractly about the commonalities of interesting and important CS problems. That is the goal of this book: to introduce the kind of mathematical, formal thinking that will allow you to understand ideas that are shared among disparate applications of computer science—and to make it easier for you to make your own connections, and to extend CS in even more new directions.

How To Use This Book

Read much, but not many Books.

Benjamin Franklin (1706–1790)
Poor Richard's Almanack (1738)

The brief version of the advice for how to use this book is: *it's your book; use it however you'd like.* (Will Shortz, the puzzle editor of *The New York Times*, gives the analogous advice about crossword puzzles when he's asked whether Googling for an answer is cheating.) But my experience is that students do best when they read actively, with scrap paper close by; most people end up with a deeper understanding of a problem by trying to solve it themselves *first*, before they look at the solution.

I've assumed throughout that you're comfortable with programming in at least one language, including familiarity with recursion. It doesn't much matter which particular programming language you know; we'll use features that are shared by almost all modern languages—things like conditionals, loops, functions, and recursion. You may or may not have had more than one programming-based CS course; many, but not all, institutions require Data Structures as a prerequisite for this material. There are times in the book when a data structures background may give you a deeper understanding (but the same is true in reverse if you study data structures after this material). There are similarly a handful of topics for which rudimentary calculus background is valuable. But knowing/remembering calculus will be specifically useful only a handful of times in this book; the mathematical prerequisite for this material is really algebra and "mathematical maturity," which basically means having some degree of comfort with the idea of a mathematical definition and with the manipulation of a mathematical expression. (The few places where calculus is helpful are explicitly marked.)

There are 10 chapters after this one in the book. Their dependencies are as shown at right. Aside from these dependencies, there are some occasional references to other chapters, but these references are light. If you've skipped Chapter 6—many instructors will choose not cover this material, as it is frequently included in a course on Algorithms instead of this one—then it will still be useful to have an informal sense of O, Ω, and Θ notation in the context of the worst-case running time of an algorithm. (You might skim Sections 6.1 and 6.6 before reading Chapters 7–11.)

$$\begin{tikzcd}
 & & 2 \text{ data types} \arrow[d] & & \\
 & & 3 \text{ logic} \arrow[d] & & \\
 & & 4 \text{ proofs} \arrow[d] & & \\
 & & 5 \text{ induction} \arrow[dll] \arrow[dl] \arrow[d] \arrow[dr] \arrow[drr] & & \\
 \underset{\text{analysis of algorithms}}{6} & \underset{\text{number theory}}{7} & \underset{\text{relations}}{8} & \underset{\text{counting}}{9} \arrow[d] & \underset{\text{graphs/trees}}{11} \\
 & & & \underset{\text{probability}}{10} &
\end{tikzcd}$$

I've tried to include some helpful tips for problem solving in the margins throughout the book, along with a few warnings about common confusions and some notes on terminology/notation that may be helpful in keeping the words and symbols straight. There are also two kinds of extensions to the main material. The "Taking it Further" blocks give more technical details about the material under discussion—an alternate way of thinking about a definition, or a way that a concept is used in CS or a related field. You should read the "Taking it Further" blocks if—but only if!—you find them engaging. Each section also ends with one or more boxed-off "Computer Science Connections" that show how the core material can be used to solve a wide variety of (interesting, I hope!) CS applications. No matter how interesting the core technical material may be, I think that it is what we can *do* with it that makes it worth studying.

What This Book Is About

All truths are easy to understand once they are discovered; the point is to discover them.

Galileo Galilei (1564–1642)

This book focuses on *discrete* mathematics, in which the entities of interest are distinct and separate. Discrete mathematics contrasts with *continuous* mathematics, as in calculus, which addresses infinitesimally small objects, which cannot be separated. We'll use summations rather than integrals, and we'll generally be thinking about things more like the integers ("$1, 2, 3, \ldots$") than like the real numbers ("all numbers between π and 42"). Because this book is mostly focused on non-programming-based parts of computer science, in general the "output" that you produce when solving a problem will be something different from a program. Most typically, you will be asked to answer some question (quantitatively or qualitatively) and to justify that answer—that is, to *prove* your answer. (A *proof* is an ironclad, airtight argument that convinces its reader of your claim.) Remember that your task in solving a problem is to persuade your reader that your purported solution genuinely solves the problem. Above all, that means that your main task in writing is communication and persuasion.

Be careful; there are two different words that are pronounced identically:

discrete, adj.: individually separate and distinct.

discreet, adj.: careful and judicious in speech, especially to maintain privacy or avoid embarrassment.

You wouldn't read a book about discreet mathematics; instead, someone who trusts you might quietly share it while making sure no one was eavesdropping.

There are three very reasonable ways of thinking about this book.

View #1 is that this book is about the mathematical foundations of computation. This book is designed to give you a firm foundation in mathematical concepts that are crucial to computer science: sets and sequences and functions, logic, proofs, probability, number theory, graphs, and so forth.

View #2 is that this book is about practice. Essentially no particular example that we consider matters; what's crucial is for you to get exposure to and experience with formal reasoning. Learning specific facts about specific topics is less important than developing your ability to reason rigorously about formally defined structures.

View #3 is that this book is about applications of computer science: it's about error-correcting codes (how to represent data redundantly so that the original information is recoverable even in the face of data corruption); cryptography (how to communicate securely so that your information is understood by its intended recipient but not by anyone else); natural language processing (how to interpret the "meaning" of an English sentence spoken by a human using an automated customer service system); and so forth. But, because solutions to these problems rely fundamentally on sets and counting and number theory and logic, we have to understand basic abstract structures in order to understand the solutions to these applied problems.

In the end, of course, all three views are right: I hope that this book will help to introduce some of the foundational technical concepts and techniques of theoretical computer science, and I hope that it will also help demonstrate that these theoretical approaches have relevance and value in work throughout computer science—in topics both theoretical and applied. And I hope that it will be at least a little bit of fun.

Bon voyage!