PREFACE FOR INSTRUCTORS

This preface is intended for instructors and other experienced computer science practitioners who are trying to understand the motivation for this book. The preface focuses especially on the similarities and differences compared to other treatments of the same material. If you are a student or general reader, you can skip the preface and start reading at chapter 1.

THE INSPIRATION OF GEB

Ever since I read Douglas Hofstadter's classic *Gödel, Escher, Bach* (GEB) as a teenager, I've known that the field of computer science has, at its foundations, a collection of extraordinarily beautiful and profound ideas. And because of GEB's surprising accessibility, I've also known that these ideas can be presented to anyone interested in computer science. Decades later, when I consider the undergraduate computer science curriculum, I view it as an opportunity to energize and excite students with a similar collection of beautiful, profound ideas—taught in a way that is accessible to any computer science undergraduate.

This book—*What Can Be Computed?*, or WCBC for short—is an attempt to do just that. It would, of course, be outrageous hubris to claim some kind of equivalence with GEB; I have no intention of making such a claim. My only hope is to aspire towards the same type of excitement, around a somewhat similar collection of ideas, with a similar level of accessibility. If the book achieves its goals, students will gain insight into the power and limitations of computation: that some important and easily stated problems cannot be solved by computer; that computation is a universal and pervasive concept; that some problems can be solved efficiently in polynomial time, and others cannot; and that many important problems have not been proved intractable, but are nevertheless believed to have no efficient method of solution.

WHICH "THEORY" COURSE ARE WE TALKING ABOUT?

In addition to trying to capture some GEB-like excitement, a more prosaic goal for the book is to serve as a textbook for an undergraduate course on the theory of computation. It's not uncommon to speak of "the" theory course in the computer science curriculum, but the undergraduate theory course has several different guises, including the following possibilities:

- **Automata theory**. This approach covers deterministic and nondeterministic finite automata, pushdown automata, languages, grammars, Turing

machines, and computability. This provides an excellent prelude to a compilers course, and also gives good coverage of computability. Complexity theory, including NP-completeness, is probably covered, but it is not a central theme of the course. Peter Linz's *Introduction to Formal Languages and Automata* is a leading example of the automata theory approach.

- **Complexity theory and algorithms**. This approach emphasizes complexity theory right from the start, and ties it in with computability as a subtopic. Taking this approach allows extra time to cover real algorithms in detail, producing something of a hybrid between traditional complexity theory courses and algorithms courses. *The Nature of Computation*, by Cristopher Moore and Stephan Mertens, is perhaps too monumental to be considered as a strictly undergraduate textbook—but it is a wonderful example of this fusion between complexity theory and algorithms. It also includes a single magisterial chapter (chapter 7, "The Grand Unified Theory of Computation") linking complexity theory with the three key approaches to computability.
- **Computability and complexity theory**. This approach sacrifices some of the knowledge needed for leaping directly into an advanced compilers course. Automata theory is still covered, but in less detail. Instead, more time is spent studying complexity classes. Advanced topics such as interactive proofs and randomness can also be covered. Michael Sipser's *Introduction to the Theory of Computation* is an excellent (perhaps even canonical) instance of the computability-and-complexity-theory approach.

WCBC falls squarely in the third category above: it covers the basics of computability and complexity with roughly equal emphasis. In my view, this approach has the best chance of capturing the GEB-style excitement discussed above.

THE FEATURES THAT MIGHT MAKE THIS BOOK APPEALING

What is the difference between WCBC and highly regarded texts such as Sipser's? There are several features of WCBC that I hope instructors will find appealing:

1. **A practical approach**. It may seem oxymoronic for a "theory" course to take a "practical" approach, but there is no contradiction here. In fact, students can gain a visceral understanding of theoretical ideas by manipulating real computer programs that exploit those ideas. Therefore, this book uses Python programs wherever possible, both for presenting examples of ideas and for proving some important results. (The online materials also include Java versions of the programs.) Another aspect of practicality is that we mostly consider general computational problems (i.e., not just decision problems). Decision problems are an important special case, but computers are usually used to solve more general problems. By working primarily with general problems, students feel a more direct connection between the theory and practice of computation.
2. **Undergraduate focus**. Most CS-theory textbooks present a wealth of detail and technical material, making them suitable for graduate courses as well

as undergraduate courses. In contrast, WCBC focuses solely on presenting material for undergraduates. By eliminating advanced topics, we can spend more time discussing the details and nuances of the central ideas. Rigorous proofs are given for a strong majority of the results, but sometimes we veer towards providing the intuition behind important ideas rather than insisting on complete technical detail. Mathematical results are presented as "claims" rather than lemmas, theorems, propositions, and corollaries. This is a purely cosmetic feature but the intention is to maintain rigor while avoiding any unnecessary formality.

3. **Minimal prerequisites**. This book aims to make the "theory" course accessible to any student moving through the computer science major. The prerequisites are essentially high-school math and an introductory programming course. More specifically, the book requires elementary knowledge of exponential, logarithmic, and polynomial functions, but does not use calculus. It makes heavy use of proof by contradiction and big-O notation, but both are explained from first principles. Proof by induction is not used at all.
4. **Historical perspective**. The book is in three parts. Part I covers computability theory and part II covers complexity theory. Part III of the book provides some historical perspective on the theory of computer science, by leading students through some of the original materials that have inspired the discipline. As discussed below, these can be interleaved with the technical material as the semester progresses, or taught as a special topic at the end of the semester.

WHAT'S IN AND WHAT'S OUT

We've seen that the book focuses on undergraduates, in part by jettisoning advanced topics. So, what has been left in, and what has been left out? Obviously, the table of contents provides all the details, but an informal summary is given in figure 1. Note that pushdown automata and context-free languages are covered in an online appendix.

Some readers may be surprised that WCBC covers Turing machines (chapter 5) before finite automata (chapter 9). There is no need for alarm: finite automata are a special case of Turing machines, and students appear to have no trouble with the inversion of the conventional ordering. Because WCBC's practical approach to the theory of computation emphasizes computer programs, it makes sense for us first to study abstract models of real computer programs (i.e., Turing machines), and later to focus on the special case of finite automata.

POSSIBLE COURSES BASED ON THIS BOOK

The primary application I have in mind for the book is a one-semester undergraduate course on the theory of computation. The content has been carefully selected so that the entire book can be covered in a single semester, even with an undergraduate class that has minimal prerequisites (programming experience and high-school math). Naturally, the chapters are arranged in my preferred

	Computability	Complexity
Included	Turing machines; undecidable problems including classics such as the halting problem; Rice's theorem; equivalence of Turing machines and real computers with infinite memory; universal computers; reductions for proving undecidability; nondeterminism; dfas, nfas, regular expressions, and regular languages. Pushdown automata and context-free languages are included in an online appendix.	complexity classes P, Exp, NP; problems in Exp but not P; NP-completeness, including an informal proof of the Cook–Levin theorem; P = NP; polynomial-time mapping reductions for proving NP-hardness.
Excluded	Unrestricted grammars; recursive function theory; lambda calculus; many details about recursively enumerable, recursive, and nonrecursive languages.	space complexity; hierarchy theorems; circuit complexity; randomness; interactive proofs.

Figure 1: Traditional theory topics that are included and excluded from this book.

teaching order. One of my goals is that the most advanced major topic, NP-completeness, is not eclipsed by being covered in the last one or two weeks of the semester. Hence, I try to finish NP-completeness with at least two weeks to spare and then cover the "historical perspective" topics of part III in a low-pressure way at the end of the course. This gives students time to complete homework assignments on NP-completeness and absorb the mysteries of P versus NP. I've also sometimes scheduled a creative final project in the last week or two. Examples include building an interesting Turing machine using JFLAP, implementing a challenging reduction in Python, and implementing simulation of nondeterministic automata in Python. Another approach is to interleave the historical and background topics of part III with the technical topics of parts I and II. Chapters 15 and 16 can be covered any time after chapter 6.

Chapter 9, on finite automata, could be omitted without causing much difficulty. Any or all of the chapters in part III can be skipped. Other sections that could be skipped or lightly skimmed include 4.5, 6.5, 7.6–7.7, 8.7–8.9, 9.4–9.6, 11.6–11.8, 12.6, 12.8, and 14.5.

COMPUTER SCIENCE AS A LIBERAL ART

Steve Jobs, the visionary former CEO of Apple, was well known for his insistence that success in the high-tech industry requires appreciation of both technology and the liberal arts. When studying the theory of computation, we have an opportunity to blend these two apparently competing areas. The discipline of computer science, as a whole, has some aspects that are engineering focused and other aspects that are more reminiscent of the liberal arts. Indeed, the

computer scientist Donald Knuth has observed that of the seven ancient liberal arts (grammar, rhetoric, logic, arithmetic, geometry, music, astronomy), at least three play important roles in computer science. I believe that studying the theory of computation allows students to apply engineering skill while also contemplating connections between philosophy, mathematics, and the use of computers in our society.