Preface

"We make ourselves no promises, but we cherish the hope that the unobstructed pursuit of useless knowledge will prove to have consequences in the future as in the past" ... "An institution which sets free successive generations of human souls is amply justified whether or not this graduate or that makes a so-called useful contribution to human knowledge. A poem, a symphony, a painting, a mathematical truth, a new scientific fact, all bear in themselves all the justification that universities, colleges, and institutes of research need or require", Abraham Flexner, The Usefulness of Useless Knowledge, 1939.

"I suggest that you take the hardest courses that you can, because you learn the most when you challenge yourself... CS 121 I found pretty hard.", Mark Zuckerberg, 2005.

This is a textbook for an undergraduate introductory course on Theoretical Computer Science. The educational goals of this course are to convey the following:

- That computation but arises in a variety of natural and manmade systems, and not only in modern silicon-based computers.
- Similarly, beyond being an extremely important *tool*, computation also serves as a useful *lens* to describe natural, physical, mathematical and even social concepts.
- The notion of *universality* of many different computational models, and the related notion of the duality between *code* and *data*.
- The idea that one can precisely define a mathematical model of computation, and then use that to prove (or sometimes only conjecture) lower bounds and impossibility results.
- Some of the surprising results and discoveries in modern theoretical computer science, including the prevalence of NP completeness, the power of interaction, the power of randomness on one hand

and the possibility of derandomization on the other, the ability to use hardness "for good" in cryptography, and the fascinating possibility of quantum computing.

I hope that following this course, students would be able to recognize computation, with both its power and pitfalls, as it arises in various settings, including seemingly "static" content or "restricted" formalisms such as macros and scripts. They should be able to follow through the logic of *proofs* about computation, including the pervasive notion of a *reduction* and understanding the subtle but crucial "self referential" proofs (such as proofs involving programs that use their own code as input). Students should understand the concept that some problems are intractable, and have the ability to recognize the potential for intractability when they are faced with a new problem. While this course only touches on cryptography, students should understand the basic idea of how computational hardness can be utilized for cryptographic purposes. But more than any specific skill, this course aims to introduce students to a new way of thinking of computation as an object in its own right, and illustrate how this new way of thinking leads to far reaching insights and applications.

My aim in writing this text is to try to convey these concepts in the simplest possible way and try to make sure that the formal notation and model help elucidate, rather than obscure, the main ideas. I also tried to take advantage of modern students' familiarity (or at least interest!) in programming, and hence use (highly simplified) programming languages as the main model of computation, as opposed to automata or Turing machines. That said, this course does not really assume fluency with any particular programming language, but more a familiarity with the general *notion* of programming. We will use programming metaphors and idioms, occasionally mentioning concrete languages such as *Python*, *C*, or *Lisp*, but students should be able to follow these descriptions even if they are not familiar with these languages.

Proofs in this course, including the existence of a universal Turing Machine, the fact that every finite function can be computed by some circuit, the Cook-Levin theorem, and many others, are often constructive and algorithmic, in the sense that they ultimately involve transforming one program to another. While the code of these transformations (like any code) is not always easy to read, and the ideas behind the proofs can be grasped without seeing it, I do think that having access to the code, and the ability to play around with it and see how it acts on various programs, can make these theorems more concrete for the students. To that end, an accompanying website (which is still work in progress) allows executing programs in the various computational models we define, as well as see constructive proofs of some of the theorems.

0.1 TO THE STUDENT

This course can be fairly challenging, mainly because it brings together a variety of ideas and techniques in the study of computation. There are quite a few technical hurdles to master, whether it is following the diagonalization argument in proving the Halting Problem is undecidable, combinatorial gadgets in NP-completeness reductions, analyzing probabilistic algorithms, or arguing about the adversary to prove security of cryptographic primitives.

The best way to engage with the material is to read these notes **actively**, so make sure you have a pen ready. While reading, I encourage you to stop and think about the following:

- When I state a theorem, stop and try to think of how you would prove it yourself *before* reading the proof in the notes. You will be amazed by how much you can understand a proof better even after only 5 minutes of attempting it yourself.

- When reading a definition, make sure that you understand what the definition means, and what are natural examples of objects that satisfy it and objects that don't. Try to think of the motivation behind the definition, and whether there are other natural ways to formalize the same concept.

- Actively notice which questions arise in your mind as you read the text, and whether or not they are answered in the text.

As a general rule, it is more important you understand the **definitions** than the **theorems**, and it is more important you understand a **theorem statement** than its **proof**. After all, before you prove a theorem, you need to understand what it states, and to understand what a theorem is about, you need to know the definitions of the objects involved. Whenever a proof of a theorem is at least somewhat complex, I provide a "proof idea". Feel free to skip the actual proof in a first reading, focusing only on the proof idea.

This book contains some code snippets, but this is by no means a programming course. You don't need to know how to program to follow this material. The reason we use code is that it is a *precise* way to describe computation. Particular implementation details are not as important to us, and so we will emphasize code readability at the expense of considerations such as error handling, encapsulation, etc.. that can be extremely important for real-world programming.

0.1.1 Is the effort worth it?

This is not an easy course, and you might reasonably wonder why should you spend the effort taking it. A traditional justification for such courses is that you might encounter these concepts in your career. Perhaps you will come across a hard problem and realize it is NP complete, or find a need to use what you learned about regular expressions. This might very well be true, but the main benefit of this course is not in teaching you any practical tool or technique, but rather in giving you a *different way of thinking*: an ability to recognize computational phenomena even when they occur in non-obvious settings, a way to model computational tasks and questions, and to reason about them.

Regardless of any use you will derive from it, I believe this course is important because it teaches concepts that are both beautiful and fundamental. The role that *energy* and *matter* played in the 20th century is played in the 21st by *computation* and *information*, not just as tools for our technology and economy, but also as the basic building blocks we use to understand the world. This course will give you a taste of some of the theory behind those, and hopefully spark your curiosity to study more.

0.2 TO POTENTIAL INSTRUCTORS

This book was initially written for my course at Harvard, but I hope that other lecturers will find it useful as well. To some extent, it is similar in content to “Theory of Computation” or “Great Ideas” courses such as those taught at CMU or MIT.

The most significant difference between our approach and more traditional ones (such as Hopcroft and Ullman’s [HU69; HU79] and Sipser’s [Sip97]) is that we do not start with *finite automata* as the basic computational model. Rather, our initial computational model is *Boolean Circuits*.[1] We believe that Boolean Circuits are more fundamental to the theory of computing (and even its practice!) than automata. In particular, Boolean Circuits are a prerequisite for many concepts that one would want to teach in a modern course on Theoretical Computer Science, including cryptography, quantum computing, derandomization, attempts at proving $\mathbf{NP} \neq \mathbf{NP}$, and more. Even in cases where Boolean Circuits are not strictly required, they can often offer significant simplifications (as in the case of the proof of the Cook-Levin Theorem).

[1] An earlier book that starts with circuits as the initial model is John Savage’s [Sav98].

Furthermore, I believe there are pedagogical reasons to start with Boolean circuits as opposed to finite automata. Boolean circuits are a more natural model of computation, and one that corresponds more closely to computing in Silicon, making the connection to practice more immediate to the students. Finite functions are arguably easier to grasp than infinite ones, as we can fully write down their truth table. The theorem that *every* finite function can be computed by some Boolean circuit is both simple enough and important enough to serve as an excellent starting point for this course. Moreover, many of the main conceptual points of the theory of computation, including the notions of the duality between *code* and *data*, and the idea of *universality*, can already be seen in this context.

After Boolean circuits, we move on to Turing machines, and prove results such as the existence of a universal Turing machine, the uncomputability of the halting problem, and Rice's Theorem. Automata are discussed after we see Turing machines and undecidability, as an example for a *restricted computational model* where problems such as determining halting can be effectively solved.

While this is not our motivation, the order we present circuits, Turing machines, and automata roughly corresponds to the chronological order of their discovery. Boolean algebra goes back to Boole's and DeMorgan's works in the 1840's [Boo47; De 47] (though the definition of Boolean circuits and the connection to physical computation was given 90 years later by Shannon [Sha38]). Alan Turing defined what we now call "Turing Machines" in the 1930's [Tur37], while finite automata were introduced in the 1943 work of McCulloch and Pitts [MP43] but only really understood in the seminal 1959 work of Rabin and Scott [RS59].

More importantly, while models such as finite-state machines, regular expressions, and context-free grammars are incredibly important for practice, the main applications for these models (whether it is for parsing, for analyzing properties such as *liveness* and *safety*, or even for software defined routing tables) arise precisely due to the fact that these are *tractable* models in which *semantic questions* can be effectively answered. This practical motivation can be better appreciated *after* students see the undecidability of semantic properties of general computing models.

The fact that we start with circuits makes proving the Cook Levin Theorem much easier. In fact, the heart of our proof of this theorem can (and is) done in a handful of lines of Python, and combining this with the standard reductions (which are also implemented in Python) allows students to appreciate visually how a question about computation can be mapped into a question about (for example) the existence of an independent set in a graph.

Some other differences between this book and prior texts are the following:

1. For measuring *time complexity*, we use the standard RAM machine model used (implicitly) in algorithms courses, rather than Tur-

ing machines. While these two models are of course polynomially equivalent, and hence make no difference for the definitions of the classes **P**, **NP**, and **EXP**, our choice makes the distinction between notions such as $O(n)$ or $O(n^2)$ time more meaningful, and ensures these finer-grained time complexity classes correspond to the informal definitions of linear and quadratic time that students encounter in their algorithms lectures (or their whiteboard coding interviews..).

2. We use the terminology of *functions* rather than *languages*. That is, rather than saying that a Turing Machine M *decides a language* $L \subseteq \{0,1\}^*$, we say that is *computes a function* $F : \{0,1\}^* \rightarrow \{0,1\}$. The terminology of "languages" arises from Chomsky's work [Cho56], but it is often more confusing than illuminating. The language terminology also makes it cumbersome to discuss concepts such as algorithms that compute functions with more than one bit of output (including basic tasks such as addition, multiplication, etc..). The fact that we use functions rather than languages means we have to be extra vigilant about students distinguishing between the *specification* of a computational task (e.g., the *function*) and its *implementation* (e.g., the *program*). On the other hand, this point is so important that it is worth repeatedly emphasizing and drilling into the students, regardless of the notation used. The book does mention the language terminology and reminds of it occasionally, to make it easier for students to consult outside resources.

Reducing the time dedicated to finite automata and context free languages allows instructors to spend more time on topics that I believe that a modern course in the theory of computing needs to touch upon, including randomness and computation, the interactions between *proofs* and *programs* (including Gödel's incompleteness theorem, interactive proof systems, and even a bit on the λ-calculus and the Curry-Howard correspondence), cryptography, and quantum computing.

My intention was to write this text in a level of detail that will enable its use for self-study. Toward that end, every chapter starts with a list of learning objectives, ends with a recap, and is peppered with "pause boxes" which encourage students to stop and work out an argument or make sure they understand a definition before continuing further.

Section 0.4 contains a "roadmap" for this book, with descriptions of the different chapters, as well as the dependency structure between them. This can help in planning a course based on this book.

0.3 ACKNOWLEDGEMENTS

This text is constantly evolving, and I am getting input from many people, for which I am deeply grateful. Thanks to Scott Aaronson, Michele Amoretti, Marguerite Basta, Sam Benkelman, Jarosław Błasiok, Emily Chan, Christy Cheng, Michelle Chiang, Daniel Chiu, Chi-Ning Chou, Michael Colavita, Robert Darley Waddilove, Juan Esteller, Leor Fishman, William Fu, Piotr Galuszka, Mark Goldstein, Alexander Golovnev, Chan Kang, Nina Katz-Christy, Estefania Lahera, Allison Lee, Ondřej Lengál, Raymond Lin, Emma Ling, Alex Lombardi, Lisa Lu, Aditya Mahadevan, Jacob Meyerson, George Moe, Hamish Nicholson, Sandip Nirmel, Sebastian Oberhoff, Thomas Orton, Pablo Parrilo, Juan Perdomo, Aaron Sachs, Abdelrhman Saleh, Brian Sapozhnikov, Peter Schäfer, Josh Seides, Alaisha Sharma, Noah Singer, Matthew Smedberg, Hikari Sorensen, Alec Sun, Everett Sussman, Marika Swanberg, Garrett Tanzer, Sarah Turnill, Salil Vadhan, Patrick Watts, Ryan Williams, Licheng Xu, Wanqian Yang, Elizabeth Yeoh-Wang, Josh Zelinsky, and Jessica Zhu for helpful feedback.

I will keep adding names here as I get more comments. If you have any comments or suggestions, please do post them on the GitHub repository https://github.com/boazbk/tcs.

Salil Vadhan co-taught with me the first iteration of this course, and gave me a tremendous amount of useful feedback and insights during this process. Michele Amoretti and Marika Swanberg read carefully several chapters of this text and gave extremely helpful detailed comments.

Thanks to Anil Ada, Venkat Guruswami, and Ryan O'Donnell for helpful tips from their experience in teaching CMU 15-251. Juan Esteller and Gabe Montague originally implemented the NAND* programming languages in OCaml and Javascript .

I am using many open source software packages in the production of these notes for which I am grateful. In particular I am thankful to Donald Knuth and Leslie Lamport for LaTeX and to John MacFarlane for Pandoc. David Steurer wrote the original scripts to produce this text. The current version uses Sergio Correia's panflute. The templates for the LaTeX and HTML versions are derived from Tufte LaTeX, Gitbook and Bookdown. I used the [Atom editor] to write this book, and the Jupyter project to write the supplemental code snippets.

Finally, I'd like to thank my family: my wife Ravit, and my children Alma and Goren. Working on this book (and the corresponding course) took so much of my time that Alma wrote an essay for her fifth-grade class saying that "universities should not pressure professors to work too much". I'm afraid all I have to show for this effort is 600 pages of ultra boring mathematical text.