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.

These are lecture notes 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 these notes 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**. 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 you can think of natural examples of objects that satisfy it and objects that don't. Try to think of the motivation behind the definition, and there other natural ways to formalize the same concept.
- At any point in the text, try to think what are the natural questions that arise, and see whether or not they are answered in the following.

0.1.1 Is this effort worth it?

A traditional justification for such a course 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 computation even when it might not be obvious that it occurs, a way to model computational tasks and questions, and to reason about them. But, 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

These lecture notes are written for my course at Harvard, but I hope that other lecturers will find them useful as well. To some extent, these notes are similar in content to "Theory of Computation" or "Great Ideas" courses such as those at CMU or MIT. There are however some differences, with the most significant being:

- I do not start with finite automata as the basic computational model, but rather with *straight-line programs* in an extremely simple programming language (or, equivalently, Boolean circuits). Automata are discussed later in the course in the context of space-bounded computation.
- Instead of Turing machines, I use an equivalent model obtained by extending the programming language above to include loops. I also introduce another extension of the programming language that allows pointers, and hence is essentially equivalent to the standard RAM machine model used (implicitly) in algorithms courses.

A much more minor notational difference is that rather than talking about *languages* (i.e., subsets $L \subseteq \{0,1\}^*$), I talk about Boolean functions (i.e., functions $f : \{0,1\}^* \rightarrow \{0,1\}$). These are of course equivalent, but the function notation extends more naturally to more general computational tasks.

Reducing the time dedicated to automata (and eliminating context free languages) allows 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 interaction of algorithms with society (with issues such as incentives, privacy, fairness), the basics of information theory, cryptography, and quantum computing.

My intention was to write these notes in a level of detail that will enable their use for self-study, and in particular for students to be able to read the notes *before* each lecture. This can help students keep

up with what is a fairly ambitious and fast-paced schedule.

0.3 Acknowledgements

These lecture notes are constantly evolving, and I am getting input from several people, for which I am deeply grateful. Thanks to Michele Amoretti, Sam Benkelman, Jarosław Błasiok, Christy Cheng, Daniel Chiu, Chi-Ning Chou, Michael Colavita, Robert Darley Waddilove, Juan Esteller, Leor Fishma, Mark Goldstein, Chan Kang, Estefania Lahera, Allison Lee, Ondřej Lengál, Emma Ling, Alex Lombardi, Lisa Lu, Aditya Mahadevan, Jacob Meyerson, Hamish Nicholson, Thomas Orton, Juan Perdomo, Aaron Sachs, Brian Sapozhnikov, Josh Seides, Alaisha Sharma, Hikari Sorensen, Alec Sun, Everett Sussman, Garrett Tanzer, Sarah Turnill, Salil Vadhan, Ryan Williams, Wanqian Yang, 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.

Thanks to Anil Ada, Venkat Guruswami, and Ryan O'Donnell for helpful tips from their experience in teaching CMU 15-251.

Thanks to Juan Esteller for his work on implementing the NAND* languages.

Thanks to David Steurer for writing the scripts (oringally produce for our joint notes on the sum of squares algorithm) that I am using to produce these notes, (themselves based on several other packages, including pandoc, LateX, and the Tufte LaTeX and CSS packages).