A "BIG-IDEAS" COMPUTATION THEORY COURSE FOR THE UNDERGRADUATE

Arnold L. Rosenberg
Department of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003, USA
rsnbrg@cs.umass.edu

Abstract

A "big-ideas" approach to an undergraduate Computation Theory course is described. The aim of this approach to the Theory is to focus the student on those of the Theory's concepts and tools that are more likely to be relevant to a student's non-theoretical endeavors. By explaining why these are the "big" concepts, the course also prepares the student to assimilate these concepts into his/her conceptual toolkit.

1 The Ideal Computation Theory Course

1.1 Course Goals

What should we expect from an undergraduate course that covers the abstract branch of theoretical computer science that one can call "Computation Theory" (in counterpoint to "Algorithms")? The course, which may be a student's unique exposure to this genre of material, should impart to the undergraduate student:

1. the need for theoretical/mathematical underpinnings for what is predominantly an engineering discipline.

   This demands careful selection of topics. One must assiduously include foundational material that is relevant to "practical"[1] computer science and avoid material that only the dedicated specialist is likely to appreciate. (The first of these goals may require tailoring the choice of theoretical material

[1] I always put the word "practical" in quotes because the material in academic courses is virtually never applied in an unmodified form in practice.

to complement the "practical" topics that appear elsewhere in the students' curriculum.)

2. the rudiments of the "theoretical method," as it applies to computer science.

   This demands including material that demonstrates the benefits of being able to think *rigorously* about the artifacts and processes of "practical" computing. It also, in my eyes, demands clear identification of the basic mathematical concepts and tools needed for the rigorous thinking.

3. a firm foundation in the most important concepts of Computation Theory, that is adequate for subsequent study of advanced topics.

   This demands giving students an *operational* command of the basic concepts and tools of Computation Theory, and of the underlying mathematics. What is *not* helpful is a cursory exposure to this material.

By including topics from Computation Theory that have a clear path to major topics in "practical" computing and by elucidating the theoretical/mathematical underpinnings of these topics, one enhances the chances of convincing the student of the relevance of the Theory to her professional development and, ultimately, her professional life.

My motivation in writing this (hopefully thought-provoking) article and pursuing the development of a specific "big-ideas" approach to Computation Theory is that I would argue, with regret, that most current Computation Theory curricula—as inferred from the contents of the standard texts (see the bibliography)—neither focus on nor satisfy the three enunciated objectives. Particularly regrettable (in my eyes) is that many of the texts facilitate courses that present an untextured grand tour of esoteric concepts, so that the students "see" topics X and Y and ... but are never told, "Topic X is central to the following 'practical' pursuit," or "Topic Y is included because of its inherent beauty, even though it does not relate to the specific problems you will encounter in your 'practical' pursuits." I view such an approach to the material as counterproductive toward the goal of making the theoretical method part of a student's professional life.

1.2 Course Organization

Computation Theory seldom occupies more than one semester in an undergraduate curriculum. (Regrettably, even that one semester is often not compulsory.) Time restrictions therefore force an instructor to make tough choices, not only about what material to cover, but also about how to organize that material in a way that suggests to the student what message(s) to take away from the course. In my view, there are basically three imperfect, incompatible "message-oriented"

organizations of a course in Computation Theory. (I ignore the message-less organization that just covers material from a text seriatim.) The material can be:

1. organized around underlying mathematical concepts and techniques.

   Unfortunately, this approach almost inevitably violates boundaries mandated by
   computation-theoretic themes.

2. organized around basic computation-theoretic themes.

   Unfortunately, this approach usually obscures commonalities in underlying concepts and mathematical underpinnings within apparently diverse topics. These commonalities are essential if the student is to apply these concepts and techniques in novel situations that are unlike the "textbook" situations..

3. organized in a way that emphasizes applications to real computational (hardware and software) artifacts.

   Unfortunately, this approach can obscure the underlying "pure" concepts, both mathematical and computational.

In order to garner much of the benefit of each of these organizations, without suffering too many of the shortcomings, one is probably best advised to organize the course via a *matrix organization*, rather than a hierarchical one. Of course, such an organization places a greater burden of contextualization on the instructor, but in the legendary words of A. Nonymous, "That's why they pay us the big bucks (or euros)."

The next two sections are dedicated, respectively, to a view (admittedly opinionated) of how most current Computation Theory courses are organized and a description of a proposed alternative organization, with a rationale for specific choices of topic and approach.

2 Today's Typical Curricula

Almost all undergraduate Computation Theory texts opt for the second organizational approach of Section 1.2; a very few opt for the third. Standard texts (see the bibliography) typically employ a two-module approach to the subject.

Module 1 comprises a smattering of topics that provide a language-theoretic approach to the theories of automata and grammars. The main justification for much of the included material seems to be the long histories of these theories. Within the context of this module, I part ways with the major texts along three axes:

1. the inclusion of many topics whose only interest is largely historical

   Example: arcane closure properties of language families

2. the omission of many topics of central conceptual importance to "practical" computation or computer science

   Example: the foundational theory underlying state-minimization algorithms

3. the presentation of topics via techniques that focus on establishing a specific result, rather than presenting a basic mathematical technique that recurs in somewhat different guises in many results

   Example: not isolating the many aspects of encoding and diagonalization that transcend individual applications.

Most of the material in this module and the approaches to that material seem to be passed from one generation of texts to the next, without a critical analysis of what is relevant to the general student of computer science (in contrast to the aspiring theorist).

Module 2 (usually the larger one) provides an intense study of Complexity Theory, perhaps preceded by some background on its (historical and intellectual) precursor, Computability Theory. This is indisputably important material: it exposes aspects of the intrinsic nature of (digital) computation; it establishes the theoretical underpinnings of important topics relating to "Algorithms;" it has quite important applications in areas as diverse as cryptology and program verification. That said, I would argue that much of what is typically included in this module goes beyond what is essential for, or even relevant to, the general computer science student (again, as opposed to the aspiring theorist). Moreover, because of restricted time, these topics preclude the inclusion of several topics that are more relevant to the development of embryonic computer scientists. Additionally, I fear that the typical presentation of much of the material via artificial, automata-theoretic models obscures the relevance of the material to "practical" computing.[2]

3 A "Big Ideas" Approach

I heartily endorse the first organizational alternative among the three that begin Section 2, motivated by the belief that a deep understanding of, *and operational control over* the few "big" mathematical ideas that underlie the Theory is the best way to enable the typical student to assimilate theoretical thinking into her computational life.

[2]This last position echoes that espoused in [6] and in the classical Computability Theory text [21].

3.1 The "Pillars" of the Proposed Course

In a famous Talmudic story, Rabbi Hillel is challenged to encapsulate all of the voluminous laws of Judaism while standing on one leg. (His response was, "What you find hateful, do not unto others.") What would a Computation Theorist respond when similarly challenged? It turns out that virtually every major result in elementary Computation Theory—the portion of the Theory that every computer scientist should have in his/her conceptual kitbag—refers in some fundamental way to one or more of three conceptual "pillars"—*State*, *Encoding*, and *Nondeterminism*—upon which I propose to build a "big-ideas" approach to Computation Theory. The mathematical correspondents of these "pillars" underlie most of the basic developments in the Theory; and the concepts themselves underlie many of the intellectual artifacts of "practical" computing.

A "big-ideas" approach to the Theory allows one to expose students to all of the major introductory-level ideas covered by present texts and courses, while also covering other topics that are (in my opinion) at least as relevant to an aspiring computer scientist (indeed, an aspiring computer professional). Additionally, this approach gives one a chance to expose the student to important, relevant mathematical ideas that are not covered in most current texts. A "big ideas" approach thus strictly improves our progress toward all four educational goals enumerated earlier, enhancing students' preparations for their futures in terms of both material and the intellectual tools for thinking about that material. I now briefly discuss my proposed three "pillars" of Computation Theory.

3.1.1 State

Myriad computational systems, both hardware and software, are organized as state-transition systems. Such a system evolves over time (or, computes) by continually changing state in response to one or more discrete stimuli (typically termed "inputs"). When in a "stable" situation, the system is in a well-defined one of its (finitely or infinitely many) states. At any such moment, in response to any valid stimulus, the system goes through some process, ending up in another "stable" situation, in some well-defined state. One of the conceptual gems of Finite-Automata Theory, the *Myhill-Nerode Theorem* [16, 17], offers a complete mathematical characterization of the concept of state within a state-transition system. Although the Theorem focuses solely on *finite* state-transition systems, one can fruitfully formulate a version of the Theorem that applies also to (discrete) infinite-state systems, not just finite ones. The Theorem's characterization of state allows one to analyze many diverse aspects of state-transition systems, with an eye toward improving their designs and/or exposing and quantifying their limitations. Indeed, one can find in the literature applications of the mathematical

characterization that involve several diverse aspects of systems, ranging from size to computational memory resources to computing time. Citing just one specific example: being in a department where Markov decision processes and their variants permeate the air, I have found that students react with widened eyes—and a renewed respect for Computation Theory—to Rabin's classical paper on probabilistic automata [19].

3.1.2 Encoding

Arguably the most fundamental results in Computability Theory and Complexity Theory depend on the ability to encode one computational problem *A* as another computational problem *B*, in a way that yields a solution to (an instance of) *A* from a solution to (the corresponding instance of) *B*. Within Computability Theory, one demands that this encoding (called a *reduction*) be supplied via a program that translates each instance of problem *A* to an instance of problem *B*; this guarantees the computability of the encoding. Within Complexity Theory, the translating program must be *efficient*, with the notion of efficiency depending on the notion of computational complexity being studied. An even more basic use of encodings is found in Turing's original study of inherent limitations of any "reasonable"[3] digital computing system [26]. Turing's work closely followed Gödel's seminal work [5], which demonstrates the inability of any "reasonable" logical system to capture through proof all true arithmetic facts. Both of these *tours de force* use encodings to demonstrate rigorously the stark distinctness of two notions that were intimately entwined in our imaginations (truth and theoremhood for Gödel, functions and programs for Turing). Importantly for the viewpoint espoused here, the encodings in both Gödel's and Turing's work are based on the relatively simple mathematics underlying the following results of Cantor [1]. (1) There exist one-to-one associations (based on computationally simple *pairing functions*) between the positive integers and the rationals. (2) There can be no one-to-one association between the rationals and the reals. While not central to Cantor's set-theoretic theme, pairing functions can be used to show that simple integer arithmetic (addition and multiplication) suffices to *encode* elaborate finite structures—e.g., finite graphs, arithmetic expressions, strings of integers—as single integers. Also relevant to the viewpoint espoused here, even the original, unembellished notion of pairing function has meat to chew on that retains juice to this day; cf. [22]! Such encodability was crucial to Gödel and Turing, for it showed that, quite remarkably, even primitive formal systems can encode *self-referential* sentences—consider the sentence, "This sentence is false." Thus, integers in a logical sentence could be encodings of sentences; integer inputs to a

[3] Reasonableness here and with Gödel's work essentially precludes systems that have answers "wired in."

program could be encodings of programs! Turing's encodings have evolved into the mapping-reductions of Computability Theory and their resource-bounded analogues in Complexity Theory. An amazing concomitant of reductions is that there sometimes exists a single problem within a class of problems that is a "hardest" one, in the sense that every problem in the class reduces to it: the Halting Problem for Turing Machines is complete for the class of "semi-decidable" problems [26]; the Satisfiability Problem is complete for the class **NP** of languages decidable in nondeterministic polynomial time [2].

3.1.3 Nondeterminism

Nondeterminism is a mathematical fiction that allows a state-transition system to "hedge its bets" by transitioning to several parallel, noncommunicating universes at each step. Real systems are typically, but not universally, deterministic (asynchrony can engender nondeterministic behavior), for this is the only known avenue to verifiable correctness and efficiency. Since nondeterminism seems, thus, to have only undesirable properties, it came as a surprise when nondeterminism was shown in the 1950s to lead to *dramatically* simplified algorithms for generating regular expressions from Finite Automata [18, 20]. This surprise became an intellectual supernova in the early 1970s with the discovery of **NP**-Completeness and its attendant **P**-vs.-**NP** problem [2, 13]. Nondeterminism was therein exposed as a *fundamental* computational notion that explains the apparent intractability of many important computational problems. Subsequent studies (cf. the early encyclopedic review of [4]) have exposed myriad problems, in areas ranging from constraint satisfaction to structure mapping to scheduling and beyond, that would admit simple, computationally efficient—indeed, often *linear time*—solutions on a truly nondeterministic computing platform but that, to this day, defy efficient—indeed, *subexponential time*—solution on any known deterministic platform. Interestingly, the benefits of nondeterminism can be explained—but not explained away!—easily and have been known for decades. Nondeterminism in an "algorithm" (of course, nondeterministic "algorithms" are not really algorithms, as they cannot be executed directly on any real computing platform) essentially abbreviate a possibly lengthy, arduous search that is part of an algorithm. The step-by-step search appears explicitly in the "algorithm," as a super-algorithmic efficient primitive of the form "Search for x," but it is usually woven intricately into an algorithm. The existence of *complete* problems that admit efficient "algorithms" means that a speedy algorithm for any of myriad important intractable problems will automatically provide speedy algorithms for all problems in the class.

3.2 A Specific "Big-Ideas" Course

I am currently working on an undergraduate Computation Theory text [23] that builds upon the three "pillars" of thepreceding section, via a (big) chapter devoted to each. This text will follow the approach of a course on the topic that I have been developing and teaching for decades, at several institutions: chronologically, Polytechnic Institute of Brooklyn, New York University, Duke University, and the University of Massachusetts. Table 1 depicts the organization of both the text and the course that are advocated here.

Topics/Pillars → ↓	**State**	**Encoding**	**Nondeterminism**
Finite Automata	*Myhill-Nerode Theorem* and Applications: Proofs of nonregularity State minimization Probabilistic FA [19]		*Kleene-Myhill Thm*
Computability		*Model independence* *Mapping-reductions* *Mapping completeness* *Rice-Myhill Theorem*	*Limited model independence* *Resource-bounded reductions* *Resource-bounded completeness*
Complexity	Memory-bounds for languages [10] Online Turing Machines [7]		*Cook-Levin Thm* Savitch's Thm [24]

Table 1: *The matrix organization of the proposed course, with selected topics and major foci highlighted.*

Within the organization depicted in the table, I develop the rudiments of the three interrelated, yet distinct, theories of *Finite Automata* (FA), *Computability*, and *Complexity*.

Finite Automata Theory. I develop FA Theory up to and including the Myhill-Nerode Theorem upon the State "pillar," following that classical result with many applications, ranging from the state-minimization algorithm for finite-state machines (which attracts the EE students), the regularity of many probabilistic automata languages [19] (which attracts the markov-decision types), time-restricted online Turing Machines [7] (which attracts the database-oriented students), and memory bounds for nonregular languages [10] (which I have found appeals to the theoretically-inclined students). The other "big" theorem of FA

Theory, the Kleene-Myhill Theorem ("Regular Expression" Theorem) [12], is developed upon the Nondeterminism "pillar," since I would argue that the topic would not likely have become algorithmically accessible without sources such as [18, 20], which exploit nondeterminism to derive the result.

Computability Theory. I cleave to the *model-independent manner* of [21] in building Computability Theory firmly on the Encoding pillar, from the underlying notions of (non)encodability [of one system as another] and reducibility [of one computational problem to another]; this development culminates in the notion of *completeness*. Perhaps the most exciting source of (mapping-)complete problems within Computability theory is the Rice-Myhill-Shapiro Theorem (cf. [21]) which, informally, demonstrates the impossibility of algorithmically deciding any property of the dynamic behavior of a program from the program's static description. Central to my presentation is the emphasis of the evolution of encoding related concepts from Cantor to Gödel to Turing, and beyond.

Complexity Theory. Within the preceding framework, it is natural to develop Complexity Theory as a (computational) resource-bounded extension of Computability theory. This view of Complexity Theory would not be supported by many of its practitioners, but I feel that it is a useful pedagogical ploy because of its allowing the students to see the notion of encoding evolve in a natural progression from Cantor's pairing functions, through mapping-reductions, to resource-bounded reductions, and to see a similar evolution in the notion of diagonalization. The central result of modern Complexity Theory, Cook's Theorem [2], and the attendant **P**-vs.-**NP** problem, is developed upon the Nondeterminism pillar, with lavish invocation of earlier-treated results about (efficient) mapping-reducibility and its attendant notion of completeness. The algorithmic consequences of the **P**-vs.-**NP** problem emerge in a very natural way, as one recognizes nondeterminism as a high-level search primitive.

4 Summation

Of course, I have no "hard" evidence of the success of my "big-ideas" approach to Computation Theory. (To be truthful, I have little faith in the kinds of evidence that are the stock in trade of schools of Education, at least in the U.S.) However, I do have considerable subjective, anecdotal evidence that the proposed approach does succeed with the "typical" computer science student, both undergraduate and graduate. Specifically, in none of the institutions where I have taught a version of the proposed course were my classes even moderately populated with aspiring theoretical computer scientists. As the course has evolved from a more-or-less standard treatment of the material (in the late 1960s) to the "big-ideas" approach described here, I have increasingly found the students talking about underlying

theoretical concepts when describing their own research. My pleasure at such moments was immense.

Acknowledgments I have benefited from discussions with and criticisms from colleagues and students too numerous to list. Special thanks, though, are due Oded Goldreich and Allan Borodin, two brilliant and perceptive computation theorists. While the comments they have generously offered have influenced me greatly, I have stubbornly opposed several of their suggestions, thereby earning whatever criticism the readers are willing to share with me.

References

[1] G. Cantor (1874): Über eine Eigenschaft des Inbegriffes aller reellen algebraischen Zahlen. *J. Reine und Angew. Math. 77*, 258–262.

[2] S.A. Cook (1971): The complexity of theorem-proving procedures. *ACM Symp. on Theory of Computing (STOC71)*, 151–158.

[3] M. Davis (1958): *Computability and Unsolvability.* McGraw-Hill, New York.

[4] M.R. Garey and D.S. Johnson (1979): *Computers and Intractability*. W.H. Freeman and Co., San Francisco.

[5] K. Gödel (1931): Über Formal Unentscheidbare Sätze der Principia Mathematica und Verwandter Systeme, I. *Monatshefte für Mathematik u. Physik 38*, 173–198.

[6] O. Goldreich (2006): On teaching the basics of complexity theory. In *Theoretical Computer Science: Essays in Memory of Shimon Even. Springer Festschrift series, Lecture Notes in Computer Science 3895*, Springer-Verlag, Heidelberg.

[7] F.C. Hennie (1966): On-line Turing machine computations. *IEEE Trans. Electronic Computers, EC-15*, 35–44.

[8] J.E. Hopcroft, R. Motwani, J.D. Ullman (2001): *Introduction to Automata Theory, Languages, and Computation* (2nd ed.). Addison-Wesley, Reading, MA.

[9] J.E. Hopcroft and J.D. Ullman (1979): *Introduction to Automata Theory, Languages, and Computation* (1st ed.) Addison-Wesley, Reading, MA.

[10] R.M. Karp (1967): Some bounds on the storage requirements of sequential machines and Turing machines. *J. ACM 14*, 478–489.

[11] R.M. Karp (1972): Reducibility among combinatorial problems. In *Complexity of Computer Computations* (R.E. Miller and J.W. Thatcher, eds.) Plenum Press, NY, pp. 85–103.

[12] S.C. Kleene (1956): Realization of events in nerve nets and finite automata. In *Automata Studies* (C.E. Shannon and J. McCarthy, Eds.) *[Ann. Math. Studies 34]*, Princeton Univ. Press, Princeton, NJ, pp. 3–42.

[13] L. Levin (1973): Universal search problems. *Problemy Peredachi Informatsii 9*, 265–266. Translated in, B.A. Trakhtenbrot (1984): A survey of Russian approaches to perebor (brute-force search) algorithms. *Annals of the History of Computing 6*, 384–400.

[14] H.R. Lewis and C.H. Papadimitriou (1981): *Elements of the Theory of Computation*. Prentice-Hall, Englewood Cliffs, NJ.

[15] B.M. Moret (1997): *The Theory of Computation*. Addison-Wesley, Reading, MA.

[16] J. Myhill (1957): Finite automata and the representation of events. WADD TR-57-624, Wright Patterson AFB, Ohio, pp. 112–137.

[17] A. Nerode (1958): Linear automaton transformations. *Proc. AMS 9*, 541–544.

[18] G.H. Ott, N.H. Feinstein (1961): Design of sequential machines from their regular expressions. *J. ACM 8*, 585–600.

[19] M.O. Rabin (1963): Probabilistic automata. *Inform. Control 6*, 230–245.

[20] M.O. Rabin and D. Scott (1959): Finite automata and their decision problems. *IBM J. Res. Develop. 3*, 114–125.

[21] H. Rogers, Jr. (1967): *Theory of Recursive Functions and Effective Computability*. McGraw Hill, New York. Reprinted in 1987 by MIT Press, Cambridge, MA.

[22] A.L. Rosenberg (2003): Efficient pairing functions—and why you should care. *Intl. J. Foundations of Computer Science 14*, 3–17.

[23] A.L. Rosenberg (2007): *The Pillars of Computation Theory: State, Nondeterminism, Encoding*. In preparation.

[24] W. Savitch (1969): Deterministic simulation of non-deterministic Turing machines. *1st ACM Symp. on Theory of Computing*, 247–248.

[25] M. Sipser (1997): *Introduction to the Theory of Computation*. PWS Publishing Co., Boston, MA.

[26] A.M. Turing (1936): On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.* (ser. 2, vol. 42) 230–265; Correction *ibid.* (vol. 43) 544–546.