Universitext

For other titles in this series, go to
http://www.springer.com/series/223

Arnold L. Rosenberg

The Pillars of Computation Theory

State, Encoding, Nondeterminism

Springer

Arnold L. Rosenberg
Research Professor
Colorado State University
and Distinguished University Professor Emeritus
University of Massachusetts, Amherst
rsnbrg@colostate.edu

ISBN 978-0-387-09638-4 e-ISBN 978-0-387-09639-1
DOI 10.1007/978-0-387-09639-1
Springer New York Dordrecht Heidelberg London

Library of Congress Control Number: 2009937878

Mathematics Subject Classification (2000): 03D05, 03D10, 03D15, 03D25, 03D45, 68Q01, 68Q05, 68Q10, 68Q15, 68Q17, 68Q45

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

To my wife, Susan, for her infinite patience during my extended episodes of "cerebral absence."

Contents

Preface

The abstract branch of theoretical computer science that we shall call "computation theory" typically appears in undergraduate academic curricula in a form that obscures both the mathematical concepts that are central to the various components of the theory and the relevance of the theory to the typical student. This regrettable situation is due largely to the thematic tension among three main competing principles for organizing the material in the course.

1. *One can organize material to emphasize underlying mathematical concepts.*
   The challenge with this approach is that it often violates boundaries mandated by computation-theoretic themes. A good example of this dilemma is seen wth our Section 5.5, which studies a computation that can be viewed as an abstraction of the following. Say that you have a database D, viewed abstractly as a sequence of binary strings. Say that you also have a (possibly very long) sequence of membership queries about the contents of D. What are the consequences, in terms of overall processing time, of demanding that a program respond to each of your queries as it arrives (the "online" scenario), in contrast to reading in all of your queries, preprocessing the sequence, and responding to all queries at once (the "offline" scenario)?
   The pedagogical dilemma here is that the computational model of Section 5.5 is a variant of the traditional Turing machine—material that traditionally comes quite a way into a course on computation theory—but the technical argumentation uses techniques that were developed originally for studying finite automata—material that traditionally comes at the very beginning of a course on computation theory.
2. *One can organize material to emphasize underlying computation-theoretic themes.*
   The challenge with this approach is almost the mirror image of the preceding one. Referring to Section 5.5, if one decides to cover this (quite illuminating!) material, but to place it in a chapter devoted to "powerful" computational models such as Turing machines, it will be quite challenging to expose the student to the fact that the mathematical underpinnings of this study actually hark back to material on finite automata that she has not seen since the earliest part of the course.

3. *One can organize material to emphasize the relevance of the Theory's concepts to real computational (hardware and software) artifacts.*
   Since theoretical computer science is, ostensibly, a branch of the more general field of computer science, arguing against this approach is almost like denying one's roots. That said, this approach would force one to cover material in a way that largely obscures the "pure" concepts that underlie the various artifacts, both the mathematical concepts and the computation-theoretic ones.

So, what is one to do? Almost all undergraduate texts on computation theory opt for the second of these alternatives; a very few opt for the third. We opt here for the first alternative! We are 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 the "big" ideas of the theory into her daily computational life.

Why do we need a new computation theory text? In order to answer this question, we must agree on what we want an upper-level undergraduate, lower-level graduate computation theory course to accomplish. In my opinion, the course should impart to the as yet uninitiated student of computer science:

1. the need for theoretical/mathematical underpinnings for what is predominantly an engineering discipline. This should include an appreciation of the need to think (and argue) rigorously about the artifacts and processes of "practical" computer science.
2. the rudiments of the "theoretical method," as it applies to computer science. This should include an *operational* command of the basic mathematical concepts and tools needed for the rigorous thinking of item 1.
3. a firm foundation in the most important concepts of theoretical computer science. This foundation should be adequate for subsequent navigation of (large portions of) advanced theoretical computer science.
4. topics from computation theory that have a clear path to major topics in general computer science. It is crucial that the student recognize the relevance of these topics to her professional development and, ultimately, her professional life.

(To me, a corollary of this presumed agenda is that an appropriately designed course in computation theory should be mandatory for all aspiring computer scientists.) With some regret, I would argue that most current curricula for computation theory courses—as inferred from the contents of the standard texts—neither focus on nor satisfy these objectives. Standard texts typically prescribe a two-module approach to the subject.

Module 1 comprises a smattering of topics that provide a formal-language-theory approach to the mathematical theories of automata and grammars. The main justification for much of the material in this module seems to be the long histories of these theories. Within the context of this module, I part ways with the major texts along two axes: (1) the inclusion of several topics of largely historical interest and the omission of several topics of central conceptual importance; (2) the way that they present certain topics. 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.

Module 2 (which is usually the larger one) provides an intense study of one specific topic, complexity theory, preceded by some background on its (historical and intellectual) precursor, computability theory. This is indisputably important material, which does expose aspects of the intrinsic nature of computation by (digital) computers and does establish the theoretical underpinnings of important topics relating to the theory of algorithm design and analysis. That said, I feel 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 (as opposed to the aspiring theoretical computer scientist); moreover, these topics preclude (because of time demands) the inclusion of several topics that are more relevant to the development of embryonic computer scientists. Additionally, I am troubled by the typical presentation of much of the material via artificial, automata-theoretic models that arose during the heyday of automata theory in the 1970s.[1]

My proposed alternative to the preceding material is a "big-ideas" approach to computation theory that is based on the three computation-theoretic "pillars" that name this book. The mathematical correspondents of these concepts underlie much of the basic development of theoretical computer science; and the concepts themselves underlie many of the intellectual artifacts of practical computer science. Such an approach to the theory allows one to expose students to all of the major introductory-level ideas covered by present texts and courses, while augmenting these topics with others that are (in my opinion) at least as relevant to an aspiring computer scientist. I contend that, additionally, this approach gives one a chance to expose the student to important mathematical ideas that do not arise within the context of the topics covered in most current texts. I thus view the proposed "big-ideas" approach as strictly improving our progress toward all four educational goals enumerated previously. We thereby (again, in my opinion) enhance students' preparations for their futures, in terms of both the material covered and the intellectual tools for thinking about that material.

While my commitment to the proposed "big-ideas" approach has philosophical origins, it has been evolving over several decades, as I have taught versions of the material in this book to both graduate and undergraduate students at (in chronological order) Polytechnic Univ. (formerly, Brooklyn Poly.), NYU, Duke, and UMass Amherst. Each time I have offered the course, I have made further progress toward my goal of a "big-ideas" presentation of the material. My (obviously biased) perception is that my students (who have been statistically *very* unlikely to become computer theorists) have been leaving the course with better perspectives and improved technical abilities as the transition to this approach has progressed.

My dream is that this book, which has been developed around the just-stated philosophy, will make the goals and tools of computation theory as accessible to the "computer science student on the street" as David Harel's well-received book [35] has achieved with the algorithmic component of theoretical computer science.

[1] This position echoes that espoused in [32] and in the classical computability theory text [80].

I end this preface with expressions of gratitude to the many colleagues who have debated this educational approach with me and the even greater number of students who have suffered with me through the growing pains of the "big-ideas" approach. Both groups are too numerous to list, and I shall not attempt to do so, for fear of missing important names. I also wish to thank the UMass Center for Teaching for a grant that contributed to the costs of preparing this text.

Falmouth, MA, and Denver, CO *Arnold L. Rosenberg*
October 1, 2009

List of Acronyms

$\Longleftrightarrow$	"if and only if"
$\stackrel{\text{def}}{=}$	"equals, by definition": used to define notions
$u \longrightarrow v$	arc in a digraph; parenthood in a rooted tree
$u \Rightarrow v$	ancestry in a rooted tree
$u \stackrel{d}{\Rightarrow} v$	depth-d ancestry in a rooted tree
$\mathbb{N}$	the set of nonnegative integers
$\mathbb{N}^+$	the set of positive integers
$\mathbb{Z}$	the set of all integers
O	the set of positive odd integers
$\lvert S\rvert$	the cardinality of set S: its number of elements when S is finite
$\emptyset$	the empty set, which has no elements: $\lvert\emptyset\rvert = 0$
$\mathscr{P}(S)$	the power set of set S: the set of all of S's subsets
κ_S	the characteristic function of the set S; $\kappa_S(w) =$ **if** $w \in S$ **then** 1 **else** 0.
κ'_S	the semicharacteristic function of the set S; $\kappa'_S(w) =$ **if** $w \in S$ **then** 1 (undefined otherwise).
Σ	a finite set of "letters:" an *alphabet*; usually the *input* set of an automaton
Σ^k	the set of length-k strings of letters from Σ, where $k \in \mathbb{N}$, i.e., is a nonnegative integer
$\Sigma^\star$	the set of all finite-length strings of letters from Σ
$\ell(w)$	the length of word $w \in \Sigma^\star$; for all $w \in \Sigma^k$, $\ell(w) = k$
w^R	the reversal of word $w \in \Sigma^\star$
ε	the (unique) null string, of length 0; $\ell(\varepsilon) = 0$
$L_1 \cdot L_2$	the concatenation of languages L_1 and L_2
L^k	the kth power of language L: $L \cdots L$ (k occurrences of "L")
$L^\star$	the star-closure of language L
$\mathscr{R}$	a regular expression
$\mathscr{L}(\mathscr{R})$	the language denoted by regular expression $\mathscr{R}$
Γ	a finite set of "letters": usually the *working alphabet* of a Turing machine (TM)

$\boxed{\text{B}}$	the *blank* symbol, which resides in the working alphabet of every Turing machine (TM)
$o(1)$	any function $f(n)$ that tends to the limit 0 as n grows without bound
$O(f(n))$	the class of functions $g(n)$ whose graphs eventually (for large n) stay below that of $c_g \cdot f(n)$ for some constant $c_g > 0$
$\Omega(f(n))$	the class of functions $g(n)$ whose graphs eventually (for large n) stay above that of $c_g \cdot f(n)$ for some constant $c_g > 0$
$\Theta(f(n))$	the intersection of the classes $O(f(n))$ and $\Omega(f(n))$
$\lvert x \rvert$	the absolute value, or magnitude, of the number x
$\log_b x$	the base-b logarithm of positive number x
$\log x$	$\log_2 x$: the base-2 logarithm of positive number x
$\exp2(n)$	an alternative notation for 2^n
$\underline{R_k}$	the (perforce, integer) contents of register R_k of a Register Machine
CFG	context-free grammar
CFL	context-free language
FA	finite automaton
IRTM	input-recording Turing machine
NFA	nondeterministic finite automaton
NTM	nondeterministic Turing machine
OA	online automaton
OTM	online Turing machine
PFA	probabilistic finite automaton
TM	Turing machine
G	a context-free grammar (CFG)
Q	the set of states of an automaton
q_0	the initial state of an automaton
F	the set of final, or accepting, states of an automaton
$E(q)$	the ε-reachability set of an NFA's state q
A-POP	the register machine analogue of the POP operation on a stack
A-PUSH	the register machine analogue of the PUSH operation on a stack
POP	the operation that removes a symbol from a stack
PUSH	the operation that adds a symbol to a stack
δ	the state-transition function of an automaton. For an OA, δ maps $Q \times \Sigma$ into Q.
$\equiv_M$	the (right-invariant) equivalence relation on $\Sigma^\star$ "defined" by an OA: $x \equiv_M y$ just when $\delta(q_0,x) = \delta(q_0,y)$.
$\equiv_M^{(t)}$	a time-parameterized version of $\equiv_M$ that is studied in Section 5.5.
$\equiv_\delta$	the equivalence relation on Q, the set of states of an OA: $p \equiv_\delta q$ just when for all $z \in \Sigma^\star$, either both $\delta(p,z)$ and $\delta(q,z)$ are accepting states, or neither is.
$\equiv_L$	the (right-invariant) equivalence relation on $\Sigma^\star$ "defined" by a language $L \subseteq \Sigma^\rfloor star$: $x \equiv_L y$ just when, for all $z \in \Sigma^\star$, either both xz and yz belong to L, or neither does.
$\widehat{\delta}$	the function δ extended to strings rather than single letters.
$L(M)$	the language accepted by automaton M

$L(G)$	the language generated by CFG G
$\mathscr{T}_M(x)$	the computation tree generated by the automaton M when processing input word x
$\widehat{\mathscr{T}}_M(x)$	the analogue of $\mathscr{T}_M(x)$ generated by a *space-bounded* automaton M, truncated to eliminate nonhalting branches
$L(M,\theta)$	the language accepted by the PFA M with acceptance threshold θ
L_{DB}	the database language of Section 5.5
$\mathscr{A}_M(n)$	the number of states in the smallest FA that is an order-n approximation of the OA M.
DHP	the "diagonal" halting problem: the set of strings x such that program x halts on input x
EMPTY	the set of programs that do not halt on any input
HP	the halting problem: the set of program-input pairs $\langle x,y\rangle$ such that program x halts on input y
$\text{HP}^{(\text{poly})}$	the poly-time version of the halting problem
SAT	the satisfiability problem: the set of CNF formulas that can be made TRUE by some assignment of truth values to logical variables
TOT	the set of programs that halt on all inputs
NP	the family of languages accepted by nondeterministic Turing machines that operate in (nondeterministic) polynomial time
P	the family of languages accepted by deterministic Turing machines that operate in polynomial time
$\ominus$	positive subtraction
$\leq_{\text{m}}$	"is m-reducible to"; "is mapping reducible to"
$\leq_{\text{poly}}$	"is polynomial-time-reducible to"
$\leq_R$	"is mapping-reducible to" within resource bound R