

Universitext

*Editorial Board
(North America):*

S. Axler
K.A. Ribet

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

Editorial Board:

Sheldon Axler; San Francisco State University, USA
Vincenzo Capasso; University of Milan, Italy
Carles Casacuberta; University of Barcelona, Spain
Angus J. Macintyre; Queen Mary, University of London. U.K.
K. A. Ribet; University of California, Berkeley, USA
Claude Sabbah; Ecole Polytechnique, France
Endre Süli; Oxford University Computing Laboratory, U.K.
Wojbor Andrzej Woyczynski; Case Western Reserve University, USA

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

© Springer Science+Business Media, LLC 2010

All rights reserved. This work may not be translated or copied in whole or in part without the written permission of the publisher (Springer Science+Business Media, LLC, 233 Spring Street, New York, NY 10013, USA), except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed is forbidden.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

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	xi
List of Acronyms	xv
I PROLEGOMENA	1
1 Introduction	3
1.1 The Three Pillars of Computation Theory	3
1.1.1 State	3
1.1.2 Encoding	4
1.1.3 Nondeterminism	7
1.2 The Nature of Computation Theory	10
2 Mathematical Preliminaries	13
2.1 Sets and Their Operations	13
2.2 Binary Relations	17
2.2.1 The Formal Notion of Binary Relation	17
2.2.2 Equivalence Relations	18
2.3 Functions	20
2.4 Formal Languages	22
2.4.1 The Notion of Language in Computation Theory	22
2.4.2 Languages as Metaphors for Computational Problems	24
2.5 Graphs and Trees	25
2.6 Useful Quantitative Notions	26
II STATE	31
3 Online Automata: Exemplars of “State”	33
3.1 Online Automata and Their “Languages”	33
3.2 A Myhill–Nerode-like Theorem for OAs	41
3.3 A Concrete OA: The Online Turing Machine	43

4	Finite Automata and Regular Languages	51
4.1	Introduction	51
4.2	Preliminaries	53
4.3	The Myhill–Nerode Theorem for FAs	56
4.3.1	The Theorem: States Are Equivalence Classes	57
4.3.2	What Do Equivalence Classes Look Like?	59
5	Applications of the Myhill–Nerode Theorem	63
5.1	Proving that Languages Are <i>Not</i> Regular	64
5.2	On Minimizing Finite Automata	66
5.3	Finite Automata with Probabilistic Transitions	70
5.3.1	PFAs and Their Languages	70
5.3.2	PFA Languages and Regular Languages	72
5.4	State as a Memory-Constraining Resource	79
5.4.1	$\mathcal{A}_M(n)$ for Two Specific Infinite OAs	80
5.4.2	A Bound on $\mathcal{A}_M(n)$ for Any OA M with Nonregular $L(M)$	81
5.5	State as a Time-Constraining Resource	83
5.5.1	Online TMs with Multiple Complex Tapes	84
5.5.2	An Information-Retrieval Problem as a Language	86
5.5.3	The Impact of Tape Structure on Memory Locality	87
5.5.4	Tape Dimensionality and the Time-Complexity of L_{DB}	88
6	Enrichment Topics	91
6.1	Pumping in Formal Languages	91
6.1.1	The Phenomenon of Pumping in Finite, Closed Systems	91
6.1.2	Pumping in Regular Languages	93
6.1.3	Pumping in Nonregular Languages	96
6.2	Closure Properties of the Regular Languages	101
6.3	Systems of Linear Equations with Languages as Coefficients	105
III	ENCODING	111
7	Countability and Uncountability: The Precursors of “Encoding”	113
7.1	Encoding Functions and Proofs of Countability	116
7.2	Diagonalization: Proofs of Uncountability	121
7.3	Where Has (Un)countability Led Us?	123
8	Enrichment Topic: “Efficient” Pairing Functions, with Applications	125
8.1	Background	126
8.2	The Prettiest Pairing Function(s)	127
8.2.1	The Diagonal PF $\mathcal{D}(x, y)$	127
8.2.2	Is $\mathcal{D}(x, y)$ the <i>Only</i> Polynomial PF?	129
8.3	Pairing Functions and the Storage of Extendible Arrays/Tables	130
8.3.1	Array-Storage Mappings via Pairing Functions	131
8.3.2	Pursuing <i>Compact</i> Pairing Functions	133
8.4	Pairing Functions and Volunteer Computing	139

8.4.1	A Methodology for Designing <i>Additive</i> Pairing Functions . . .	141
8.4.2	A Sampler of Explicit APFs	142
9	Computability Theory	147
9.1	Introduction and History	147
9.2	Preliminaries	151
9.2.1	Representing Computational Problems as Formal Languages .	151
9.2.2	Functions and Partial Functions	153
9.2.3	Self-Referential Programs: Interpreters and Compilers	155
9.3	The Halting Problem: The “Oldest” Unsolvable Problem	155
9.3.1	The Halting Problem Is Semisolvable but Not Solvable	156
9.3.2	Why We Care about the Halting Problem—An Example	158
9.4	Mapping Reducibility	160
9.4.1	Basic Properties of m -Reducibility	162
9.4.2	The s - m - n Theorem: Where Does One Find Encodings?	163
9.5	The Rice–Myhill–Shapiro Theorem	165
9.6	Complete, or “Hardest,” Semidecidable Problems	169
9.7	Some Important Limitations of Computability	172
9.8	(Online) Turing Machines and the Church–Turing Thesis	174
9.8.1	Simplifying an OTM without Diminishing Its Power	176
9.8.2	Augmented TMs That Are No More Powerful Than OTMs . .	195
IV	NONDETERMINISM	209
10	Nondeterministic Online Automata	211
10.1	Nondeterministic OAs	211
10.2	Nondeterminism as Unbounded Search, 1	213
10.3	An Overview of Nondeterminism in Computation Theory	215
11	Nondeterministic FAs	217
11.1	Nondeterministic FAs vs. Deterministic FAs	217
11.1.1	NFAs Are No More Powerful Than DFAs	217
11.1.2	Does the Subset Construction Waste DFA States?	219
11.2	An Application: The Kleene–Myhill Theorem	221
11.2.1	A Convenient Enhancement of NFAs	221
11.2.2	The Kleene–Myhill Theorem	223
12	Nondeterminism in Computability Theory	233
12.1	Introduction	233
12.2	Nondeterministic Turing Machines	234
12.2.1	The NTM Model	234
12.2.2	Deterministic Simulation of Nondeterminism: NTMs and OTMs	236
12.3	Nondeterminism as Unbounded Search, 2	241

13 Complexity Theory	245
13.1 Introduction	245
13.2 Time and Space Complexity	252
13.2.1 On Measuring Time Complexity	253
13.2.2 On Measuring Space Complexity	256
13.3 Reducibility, Hardness, and Completeness in Complexity Theory ...	263
13.3.1 A General Look at Resource-Bounded Computation	263
13.3.2 <i>Efficient</i> Mapping Reducibility	264
13.3.3 Hard Problems and Complete Problems	268
13.3.4 An NP -Complete Version of the Halting Problem	269
13.3.5 The Cook-Levin Theorem: The NP -Completeness of SAT ...	276
13.4 Nondeterminism and Space Complexity	285
13.4.1 Simulating Nondeterminism Space-Efficiently: Savitch's Theorem	287
13.4.2 Beyond Savitch's Theorem	297
V Sample Exercises	299
References	311
Index	315

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 with 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

$\iff$	“if and only if”
$\stackrel{\text{def}}{=}$	“equals, by definition”: used to define notions
$u \rightarrow v$	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
$\mathbb{O}$	the set of positive odd integers
$ S $	the cardinality of set S : its number of elements when S is finite
$\emptyset$	the empty set, which has no elements: $ \emptyset = 0$
$\mathcal{P}(S)$	the power set of set S : the set of all of S ’s subsets
κ_S	the characteristic function of the set S ; $\kappa_S(w) = \text{if } w \in S \text{ then } 1 \text{ else } 0$.
κ'_S	the semicharacteristic function of the set S ; $\kappa'_S(w) = \text{if } w \in S \text{ then } 1$ (undefined otherwise).
Σ	a finite set of “letters:” an <i>alphabet</i> ; usually the <i>input</i> 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
Σ^*	the set of all finite-length strings of letters from Σ
$\ell(w)$	the length of word $w \in \Sigma^*$; for all $w \in \Sigma^k$, $\ell(w) = k$
w^R	the reversal of word $w \in \Sigma^*$
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 k th power of language L : $L \cdots L$ (k occurrences of “ L ”)
L^*	the star-closure of language L
$\mathcal{R}$	a regular expression
$\mathcal{L}(\mathcal{R})$	the language denoted by regular expression $\mathcal{R}$
Γ	a finite set of “letters”: usually the <i>working alphabet</i> of a Turing machine (TM)

$\square$	the <i>blank</i> 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 $\Omega(f(n))$
$ x $	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
$\exp 2(n)$	an alternative notation for 2^n
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 Σ^* “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 $z \in \Sigma^*$, either both $\delta(p, z)$ and $\delta(q, z)$ are accepting states, or neither is.
$\equiv_L$	the (right-invariant) equivalence relation on Σ^* “defined” by a language $L \subseteq \Sigma^+$: $x \equiv_L y$ just when, for all $z \in \Sigma^*$, either both xz and yz belong to L , or neither does.
$\hat{\delta}$	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
$\mathcal{T}_M(x)$	the computation tree generated by the automaton M when processing input word x
$\widehat{\mathcal{T}}_M(x)$	the analogue of $\mathcal{T}_M(x)$ generated by a <i>space-bounded</i> 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
$\mathcal{A}_M(n)$	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
HP ^(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
$\leq_m$	“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