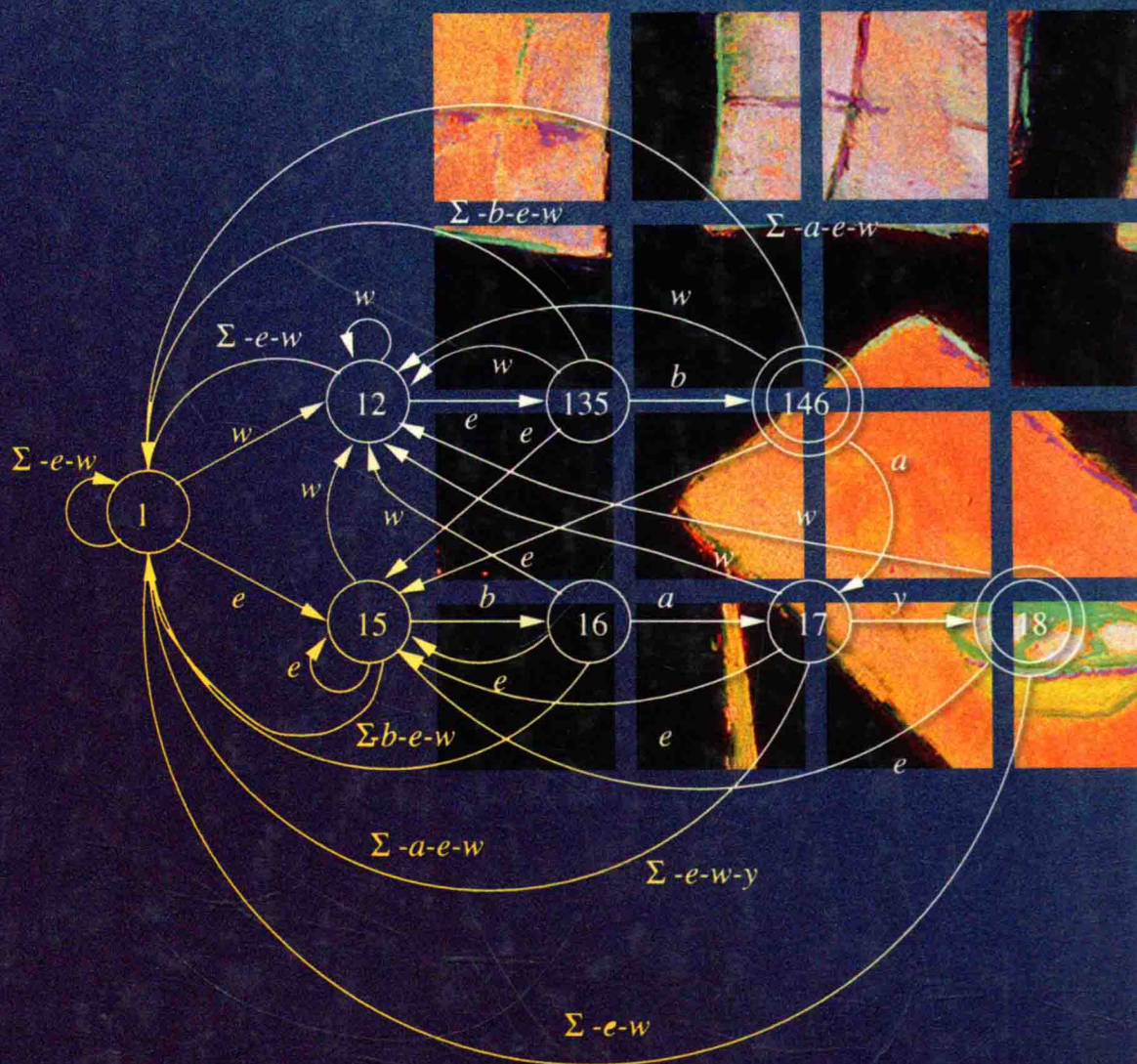


JOHN E. HOPCROFT

RAJEEV MOTWANI

JEFFREY D. ULLMAN

Introduction to Automata Theory, Languages, and Computation



SECOND EDITION

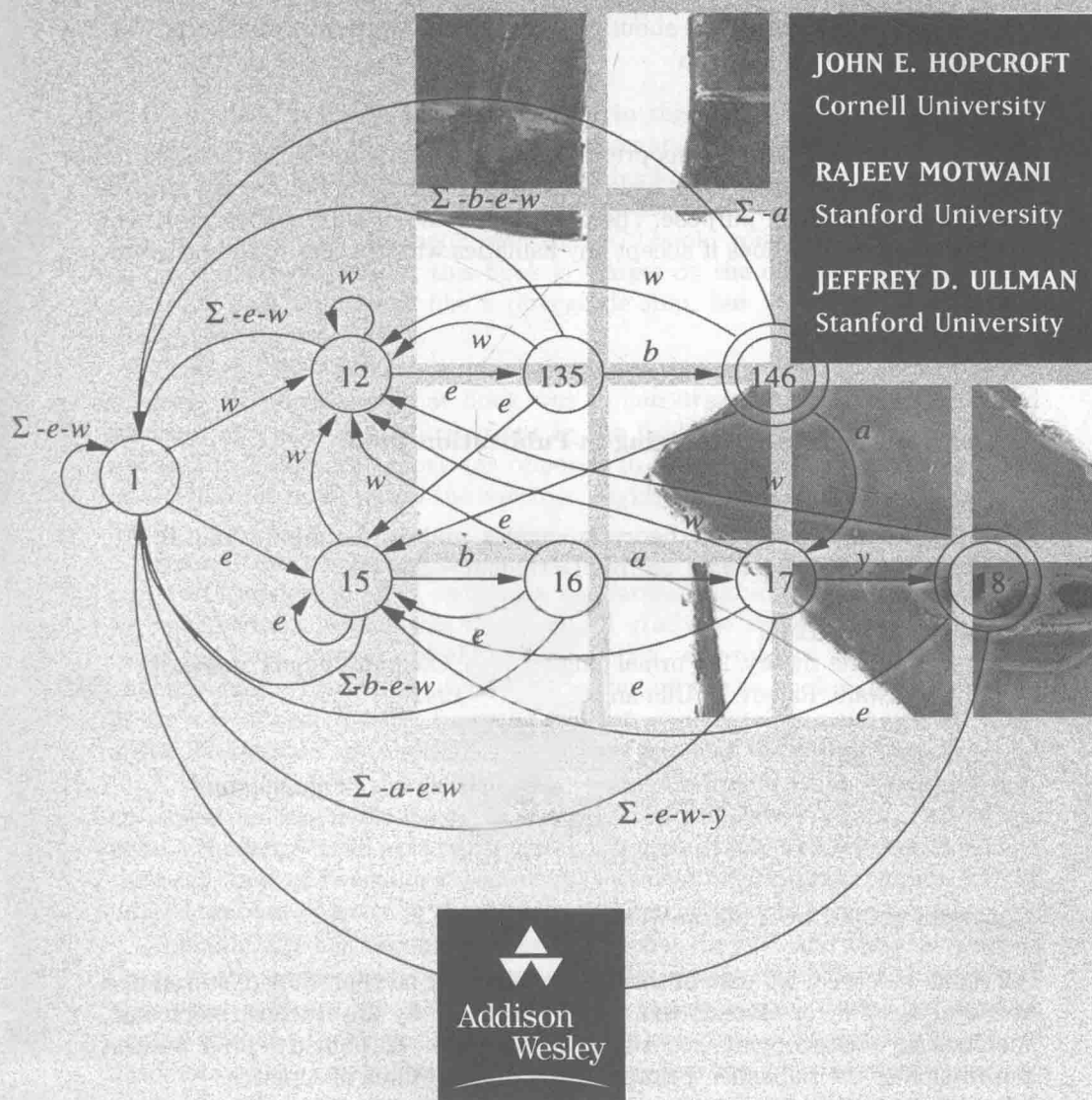
Introduction to Automata Theory, Languages, and Computation


SECOND
EDITION

JOHN E. HOPCROFT
Cornell University

RAJEEV MOTWANI
Stanford University

JEFFREY D. ULLMAN
Stanford University




Addison
Wesley

Boston San Francisco New York
London Toronto Sydney Tokyo Singapore Madrid
Mexico City Munich Paris Cape Town Hong Kong Montreal

Senior Acquisitions Editor	<i>Maite Suarez-Rivas</i>
Project Editor	<i>Katherine Harutunian</i>
Executive Marketing Manager	<i>Michael Hirsch</i>
Cover Design	<i>Leslie Haimes</i>
Art Direction	<i>Regina Hagen</i>
Prepress and Manufacturing	<i>Caroline Fell</i>

Access the latest information about Addison-Wesley titles from our World Wide Web site: <http://www.awl.com>

The programs and applications presented in this book have been included for their instructional value. They have been tested with care, but are not guaranteed for any particular purpose. The publisher does not offer any warranties or representations, nor does it accept any liabilities with respect to the programs or applications.

Library of Congress Cataloging-in-Publication Data

Hopcroft, John E., 1939-

Introduction to automata theory, languages, and computation / John E.

Hopcroft, Rajeev Motwani, Jeffrey D. Ullman.—2nd ed.

p. cm.

ISBN 0-201-44124-1

1. Machine theory. 2. Formal languages. 3. Computational complexity.

I. Motwani, Rajeev. II. Ullman, Jeffrey D., 1942-.

QA267 .H56 2001

511.3—dc21

00-064608

Copyright © 2001 by Addison-Wesley

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America.

2345678910-MA-04030201

Preface

In the preface from the 1979 predecessor to this book, Hopcroft and Ullman marveled at the fact that the subject of automata had exploded, compared with its state at the time they wrote their first book, in 1969. Truly, the 1979 book contained many topics not found in the earlier work and was about twice its size. If you compare this book with the 1979 book, you will find that, like the automobiles of the 1970's, this book is "larger on the outside, but smaller on the inside." That sounds like a retrograde step, but we are happy with the changes for several reasons.

First, in 1979, automata and language theory was still an area of active research. A purpose of that book was to encourage mathematically inclined students to make new contributions to the field. Today, there is little direct research in automata theory (as opposed to its applications), and thus little motivation for us to retain the succinct, highly mathematical tone of the 1979 book.

Second, the role of automata and language theory has changed over the past two decades. In 1979, automata was largely a graduate-level subject, and we imagined our reader was an advanced graduate student, especially those using the later chapters of the book. Today, the subject is a staple of the undergraduate curriculum. As such, the content of the book must assume less in the way of prerequisites from the student, and therefore must provide more of the background and details of arguments than did the earlier book.

A third change in the environment is that Computer Science has grown to an almost unimaginable degree in the past two decades. While in 1979 it was often a challenge to fill up a curriculum with material that we felt would survive the next wave of technology, today very many subdisciplines compete for the limited amount of space in the undergraduate curriculum.

Fourthly, CS has become a more vocational subject, and there is a severe pragmatism among many of its students. We continue to believe that aspects of automata theory are essential tools in a variety of new disciplines, and we believe that the theoretical, mind-expanding exercises embodied in the typical automata course retain their value, no matter how much the student prefers to learn only the most immediately monetizable technology. However, to assure a continued place for the subject on the menu of topics available to the computer science student, we believe it is necessary to emphasize the applications

along with the mathematics. Thus, we have replaced a number of the more abstruse topics in the earlier book with examples of how the ideas are used today. While applications of automata and language theory to compilers are now so well understood that they are normally covered in a compiler course, there are a variety of more recent uses, including model-checking algorithms to verify protocols and document-description languages that are patterned on context-free grammars.

A final explanation for the simultaneous growth and shrinkage of the book is that we were today able to take advantage of the \TeX and \LaTeX typesetting systems developed by Don Knuth and Les Lamport. The latter, especially, encourages the “open” style of typesetting that makes books larger, but easier to read. We appreciate the efforts of both men.

Use of the Book

This book is suitable for a quarter or semester course at the Junior level or above. At Stanford, we have used the notes in CS154, the course in automata and language theory. It is a one-quarter course, which both Rajeev and Jeff have taught. Because of the limited time available, Chapter 11 is not covered, and some of the later material, such as the more difficult polynomial-time reductions in Section 10.4 are omitted as well. The book’s Web site (see below) includes notes and syllabi for several offerings of CS154.

Some years ago, we found that many graduate students came to Stanford with a course in automata theory that did not include the theory of intractability. As the Stanford faculty believes that these ideas are essential for every computer scientist to know at more than the level of “NP-complete means it takes too long,” there is another course, CS154N, that students may take to cover only Chapters 8, 9, and 10. They actually participate in roughly the last third of CS154 to fulfill the CS154N requirement. Even today, we find several students each quarter availing themselves of this option. Since it requires little extra effort, we recommend the approach.

Prerequisites

To make best use of this book, students should have taken previously a course covering discrete mathematics, e.g., graphs, trees, logic, and proof techniques. We assume also that they have had several courses in programming, and are familiar with common data structures, recursion, and the role of major system components such as compilers. These prerequisites should be obtained in a typical freshman-sophomore CS program.

Exercises

The book contains extensive exercises, with some for almost every section. We indicate harder exercises or parts of exercises with an exclamation point. The hardest exercises have a double exclamation point.

Some of the exercises or parts are marked with a star. For these exercises, we shall endeavor to maintain solutions accessible through the book's Web page. These solutions are publicly available and should be used for self-testing. Note that in a few cases, one exercise B asks for modification or adaptation of your solution to another exercise A . If certain parts of A have solutions, then you should expect the corresponding parts of B to have solutions as well.

Support on the World Wide Web

The book's home page is

<http://www-db.stanford.edu/~ullman/ialc.html>

Here are solutions to starred exercises, errata as we learn of them, and backup materials. We hope to make available the notes for each offering of CS154 as we teach it, including homeworks, solutions, and exams.

Acknowledgements

A handout on “how to do proofs” by Craig Silverstein influenced some of the material in Chapter 1. Comments and errata on drafts of this book were received from: Zoe Abrams, George Candea, Haowen Chen, Byong-Gun Chun, Jeffrey Shallit, Bret Taylor, Jason Townsend, and Erik Uzureau. They are gratefully acknowledged. Remaining errors are ours, of course.

J. E. H.
R. M.
J. D. U.
Ithaca NY and Stanford CA
September, 2000

Table of Contents

1 Automata: The Methods and the Madness	1
1.1 Why Study Automata Theory?	2
1.1.1 Introduction to Finite Automata	2
1.1.2 Structural Representations	4
1.1.3 Automata and Complexity	5
1.2 Introduction to Formal Proof	5
1.2.1 Deductive Proofs	6
1.2.2 Reduction to Definitions	8
1.2.3 Other Theorem Forms	10
1.2.4 Theorems That Appear Not to Be If-Then Statements . .	13
1.3 Additional Forms of Proof	13
1.3.1 Proving Equivalences About Sets	14
1.3.2 The Contrapositive	14
1.3.3 Proof by Contradiction	16
1.3.4 Counterexamples	17
1.4 Inductive Proofs	19
1.4.1 Inductions on Integers	19
1.4.2 More General Forms of Integer Inductions	22
1.4.3 Structural Inductions	23
1.4.4 Mutual Inductions	26
1.5 The Central Concepts of Automata Theory	28
1.5.1 Alphabets	28
1.5.2 Strings	29
1.5.3 Languages	30
1.5.4 Problems	31
1.6 Summary of Chapter 1	34
1.7 References for Chapter 1	35
2 Finite Automata	37
2.1 An Informal Picture of Finite Automata	38
2.1.1 The Ground Rules	38
2.1.2 The Protocol	39
2.1.3 Enabling the Automata to Ignore Actions	41

2.1.4	The Entire System as an Automaton	43
2.1.5	Using the Product Automaton to Validate the Protocol	45
2.2	Deterministic Finite Automata	45
2.2.1	Definition of a Deterministic Finite Automaton	46
2.2.2	How a DFA Processes Strings	46
2.2.3	Simpler Notations for DFA's	48
2.2.4	Extending the Transition Function to Strings	49
2.2.5	The Language of a DFA	52
2.2.6	Exercises for Section 2.2	53
2.3	Nondeterministic Finite Automata	55
2.3.1	An Informal View of Nondeterministic Finite Automata	56
2.3.2	Definition of Nondeterministic Finite Automata	57
2.3.3	The Extended Transition Function	58
2.3.4	The Language of an NFA	59
2.3.5	Equivalence of Deterministic and Nondeterministic Finite Automata	60
2.3.6	A Bad Case for the Subset Construction	65
2.3.7	Exercises for Section 2.3	66
2.4	An Application: Text Search	68
2.4.1	Finding Strings in Text	68
2.4.2	Nondeterministic Finite Automata for Text Search	69
2.4.3	A DFA to Recognize a Set of Keywords	70
2.4.4	Exercises for Section 2.4	72
2.5	Finite Automata With Epsilon-Transitions	72
2.5.1	Uses of ϵ -Transitions	72
2.5.2	The Formal Notation for an ϵ -NFA	74
2.5.3	Epsilon-Closures	75
2.5.4	Extended Transitions and Languages for ϵ -NFA's	76
2.5.5	Eliminating ϵ -Transitions	77
2.5.6	Exercises for Section 2.5	80
2.6	Summary of Chapter 2	80
2.7	References for Chapter 2	81
3	Regular Expressions and Languages	83
3.1	Regular Expressions	83
3.1.1	The Operators of Regular Expressions	84
3.1.2	Building Regular Expressions	85
3.1.3	Precedence of Regular-Expression Operators	88
3.1.4	Exercises for Section 3.1	89
3.2	Finite Automata and Regular Expressions	90
3.2.1	From DFA's to Regular Expressions	91
3.2.2	Converting DFA's to Regular Expressions by Eliminating States	96
3.2.3	Converting Regular Expressions to Automata	101
3.2.4	Exercises for Section 3.2	106

3.3	Applications of Regular Expressions	108
3.3.1	Regular Expressions in UNIX	108
3.3.2	Lexical Analysis	109
3.3.3	Finding Patterns in Text	111
3.3.4	Exercises for Section 3.3	113
3.4	Algebraic Laws for Regular Expressions	114
3.4.1	Associativity and Commutativity	114
3.4.2	Identities and Annihilators	115
3.4.3	Distributive Laws	115
3.4.4	The Idempotent Law	116
3.4.5	Laws Involving Closures	117
3.4.6	Discovering Laws for Regular Expressions	117
3.4.7	The Test for a Regular-Expression Algebraic Law	119
3.4.8	Exercises for Section 3.4	120
3.5	Summary of Chapter 3	122
3.6	References for Chapter 3	122
4	Properties of Regular Languages	125
4.1	Proving Languages not to be Regular	126
4.1.1	The Pumping Lemma for Regular Languages	126
4.1.2	Applications of the Pumping Lemma	127
4.1.3	Exercises for Section 4.1	129
4.2	Closure Properties of Regular Languages	131
4.2.1	Closure of Regular Languages Under Boolean Operations	131
4.2.2	Reversal	137
4.2.3	Homomorphisms	139
4.2.4	Inverse Homomorphisms	140
4.2.5	Exercises for Section 4.2	145
4.3	Decision Properties of Regular Languages	149
4.3.1	Converting Among Representations	149
4.3.2	Testing Emptiness of Regular Languages	151
4.3.3	Testing Membership in a Regular Language	153
4.3.4	Exercises for Section 4.3	153
4.4	Equivalence and Minimization of Automata	154
4.4.1	Testing Equivalence of States	154
4.4.2	Testing Equivalence of Regular Languages	157
4.4.3	Minimization of DFA's	159
4.4.4	Why the Minimized DFA Can't Be Beaten	162
4.4.5	Exercises for Section 4.4	164
4.5	Summary of Chapter 4	165
4.6	References for Chapter 4	166

5	Context-Free Grammars and Languages	169
5.1	Context-Free Grammars	169
5.1.1	An Informal Example	170
5.1.2	Definition of Context-Free Grammars	171
5.1.3	Derivations Using a Grammar	173
5.1.4	Leftmost and Rightmost Derivations	175
5.1.5	The Language of a Grammar	177
5.1.6	Sentential Forms	178
5.1.7	Exercises for Section 5.1	179
5.2	Parse Trees	181
5.2.1	Constructing Parse Trees	181
5.2.2	The Yield of a Parse Tree	183
5.2.3	Inference, Derivations, and Parse Trees	184
5.2.4	From Inferences to Trees	185
5.2.5	From Trees to Derivations	187
5.2.6	From Derivations to Recursive Inferences	190
5.2.7	Exercises for Section 5.2	191
5.3	Applications of Context-Free Grammars	191
5.3.1	Parsers	192
5.3.2	The YACC Parser-Generator	194
5.3.3	Markup Languages	196
5.3.4	XML and Document-Type Definitions	198
5.3.5	Exercises for Section 5.3	204
5.4	Ambiguity in Grammars and Languages	205
5.4.1	Ambiguous Grammars	205
5.4.2	Removing Ambiguity From Grammars	207
5.4.3	Leftmost Derivations as a Way to Express Ambiguity	211
5.4.4	Inherent Ambiguity	212
5.4.5	Exercises for Section 5.4	214
5.5	Summary of Chapter 5	215
5.6	References for Chapter 5	216
6	Pushdown Automata	219
6.1	Definition of the Pushdown Automaton	219
6.1.1	Informal Introduction	219
6.1.2	The Formal Definition of Pushdown Automata	221
6.1.3	A Graphical Notation for PDA's	223
6.1.4	Instantaneous Descriptions of a PDA	224
6.1.5	Exercises for Section 6.1	228
6.2	The Languages of a PDA	229
6.2.1	Acceptance by Final State	229
6.2.2	Acceptance by Empty Stack	230
6.2.3	From Empty Stack to Final State	231
6.2.4	From Final State to Empty Stack	234
6.2.5	Exercises for Section 6.2	236

6.3	Equivalence of PDA's and CFG's	237
6.3.1	From Grammars to Pushdown Automata	237
6.3.2	From PDA's to Grammars	241
6.3.3	Exercises for Section 6.3	245
6.4	Deterministic Pushdown Automata	246
6.4.1	Definition of a Deterministic PDA	247
6.4.2	Regular Languages and Deterministic PDA's	247
6.4.3	DPDA's and Context-Free Languages	249
6.4.4	DPDA's and Ambiguous Grammars	249
6.4.5	Exercises for Section 6.4	251
6.5	Summary of Chapter 6	252
6.6	References for Chapter 6	253
7	Properties of Context-Free Languages	255
7.1	Normal Forms for Context-Free Grammars	255
7.1.1	Eliminating Useless Symbols	256
7.1.2	Computing the Generating and Reachable Symbols	258
7.1.3	Eliminating ϵ -Productions	259
7.1.4	Eliminating Unit Productions	262
7.1.5	Chomsky Normal Form	266
7.1.6	Exercises for Section 7.1	269
7.2	The Pumping Lemma for Context-Free Languages	274
7.2.1	The Size of Parse Trees	274
7.2.2	Statement of the Pumping Lemma	275
7.2.3	Applications of the Pumping Lemma for CFL's	276
7.2.4	Exercises for Section 7.2	280
7.3	Closure Properties of Context-Free Languages	281
7.3.1	Substitutions	282
7.3.2	Applications of the Substitution Theorem	284
7.3.3	Reversal	285
7.3.4	Intersection With a Regular Language	285
7.3.5	Inverse Homomorphism	289
7.3.6	Exercises for Section 7.3	291
7.4	Decision Properties of CFL's	293
7.4.1	Complexity of Converting Among CFG's and PDA's	294
7.4.2	Running Time of Conversion to Chomsky Normal Form	295
7.4.3	Testing Emptiness of CFL's	296
7.4.4	Testing Membership in a CFL	298
7.4.5	Preview of Undecidable CFL Problems	302
7.4.6	Exercises for Section 7.4	302
7.5	Summary of Chapter 7	303
7.6	References for Chapter 7	304

8	Introduction to Turing Machines	307
8.1	Problems That Computers Cannot Solve	307
8.1.1	Programs that Print “Hello, World”	308
8.1.2	The Hypothetical “Hello, World” Tester	310
8.1.3	Reducing One Problem to Another	313
8.1.4	Exercises for Section 8.1	316
8.2	The Turing Machine	316
8.2.1	The Quest to Decide All Mathematical Questions	317
8.2.2	Notation for the Turing Machine	318
8.2.3	Instantaneous Descriptions for Turing Machines	320
8.2.4	Transition Diagrams for Turing Machines	323
8.2.5	The Language of a Turing Machine	326
8.2.6	Turing Machines and Halting	327
8.2.7	Exercises for Section 8.2	328
8.3	Programming Techniques for Turing Machines	329
8.3.1	Storage in the State	330
8.3.2	Multiple Tracks	331
8.3.3	Subroutines	333
8.3.4	Exercises for Section 8.3	334
8.4	Extensions to the Basic Turing Machine	336
8.4.1	Multitape Turing Machines	336
8.4.2	Equivalence of One-Tape and Multitape TM’s	337
8.4.3	Running Time and the Many-Tapes-to-One Construction	339
8.4.4	Nondeterministic Turing Machines	340
8.4.5	Exercises for Section 8.4	342
8.5	Restricted Turing Machines	345
8.5.1	Turing Machines With Semi-infinite Tapes	345
8.5.2	Multistack Machines	348
8.5.3	Counter Machines	351
8.5.4	The Power of Counter Machines	352
8.5.5	Exercises for Section 8.5	354
8.6	Turing Machines and Computers	355
8.6.1	Simulating a Turing Machine by Computer	355
8.6.2	Simulating a Computer by a Turing Machine	356
8.6.3	Comparing the Running Times of Computers and Turing Machines	361
8.7	Summary of Chapter 8	363
8.8	References for Chapter 8	365
9	Undecidability	367
9.1	A Language That Is Not Recursively Enumerable	368
9.1.1	Enumerating the Binary Strings	369
9.1.2	Codes for Turing Machines	369
9.1.3	The Diagonalization Language	370
9.1.4	Proof that L_d is not Recursively Enumerable	372

9.1.5	Exercises for Section 9.1	372
9.2	An Undecidable Problem That is RE	373
9.2.1	Recursive Languages	373
9.2.2	Complements of Recursive and RE languages	374
9.2.3	The Universal Language	377
9.2.4	Undecidability of the Universal Language	379
9.2.5	Exercises for Section 9.2	381
9.3	Undecidable Problems About Turing Machines	383
9.3.1	Reductions	383
9.3.2	Turing Machines That Accept the Empty Language	384
9.3.3	Rice's Theorem and Properties of the RE Languages	387
9.3.4	Problems about Turing-Machine Specifications	390
9.3.5	Exercises for Section 9.3	390
9.4	Post's Correspondence Problem	392
9.4.1	Definition of Post's Correspondence Problem	392
9.4.2	The "Modified" PCP	394
9.4.3	Completion of the Proof of PCP Undecidability	397
9.4.4	Exercises for Section 9.4	403
9.5	Other Undecidable Problems	403
9.5.1	Problems About Programs	403
9.5.2	Undecidability of Ambiguity for CFG's	404
9.5.3	The Complement of a List Language	406
9.5.4	Exercises for Section 9.5	409
9.6	Summary of Chapter 9	410
9.7	References for Chapter 9	411
10	Intractable Problems	413
10.1	The Classes \mathcal{P} and \mathcal{NP}	414
10.1.1	Problems Solvable in Polynomial Time	414
10.1.2	An Example: Kruskal's Algorithm	414
10.1.3	Nondeterministic Polynomial Time	419
10.1.4	An \mathcal{NP} Example: The Traveling Salesman Problem	419
10.1.5	Polynomial-Time Reductions	421
10.1.6	NP-Complete Problems	422
10.1.7	Exercises for Section 10.1	423
10.2	An NP-Complete Problem	426
10.2.1	The Satisfiability Problem	426
10.2.2	Representing SAT Instances	427
10.2.3	NP-Completeness of the SAT Problem	428
10.2.4	Exercises for Section 10.2	434
10.3	A Restricted Satisfiability Problem	435
10.3.1	Normal Forms for Boolean Expressions	436
10.3.2	Converting Expressions to CNF	437
10.3.3	NP-Completeness of CSAT	440
10.3.4	NP-Completeness of 3SAT	445
10.3.5	Exercises for Section 10.3	446

10.4	Additional NP-Complete Problems	447
10.4.1	Describing NP-complete Problems	447
10.4.2	The Problem of Independent Sets	448
10.4.3	The Node-Cover Problem	452
10.4.4	The Directed Hamilton-Circuit Problem	453
10.4.5	Undirected Hamilton Circuits and the TSP	460
10.4.6	Summary of NP-Complete Problems	461
10.4.7	Exercises for Section 10.4	462
10.5	Summary of Chapter 10	466
10.6	References for Chapter 10	467
11	Additional Classes of Problems	469
11.1	Complements of Languages in \mathcal{NP}	470
11.1.1	The Class of Languages $\text{Co-}\mathcal{NP}$	470
11.1.2	NP-Complete Problems and $\text{Co-}\mathcal{NP}$	471
11.1.3	Exercises for Section 11.1	472
11.2	Problems Solvable in Polynomial Space	473
11.2.1	Polynomial-Space Turing Machines	473
11.2.2	Relationship of \mathcal{PS} and \mathcal{NPS} to Previously Defined Classes	474
11.2.3	Deterministic and Nondeterministic Polynomial Space . .	476
11.3	A Problem That Is Complete for \mathcal{PS}	478
11.3.1	PS-Completeness	478
11.3.2	Quantified Boolean Formulas	479
11.3.3	Evaluating Quantified Boolean Formulas	480
11.3.4	PS-Completeness of the QBF Problem	482
11.3.5	Exercises for Section 11.3	487
11.4	Language Classes Based on Randomization	487
11.4.1	Quicksort: an Example of a Randomized Algorithm . . .	488
11.4.2	A Turing-Machine Model Using Randomization	489
11.4.3	The Language of a Randomized Turing Machine	490
11.4.4	The Class \mathcal{RP}	492
11.4.5	Recognizing Languages in \mathcal{RP}	494
11.4.6	The Class \mathcal{ZPP}	495
11.4.7	Relationship Between \mathcal{RP} and \mathcal{ZPP}	496
11.4.8	Relationships to the Classes \mathcal{P} and \mathcal{NP}	497
11.5	The Complexity of Primality Testing	498
11.5.1	The Importance of Testing Primality	499
11.5.2	Introduction to Modular Arithmetic	501
11.5.3	The Complexity of Modular-Arithmetic Computations . .	503
11.5.4	Random-Polynomial Primality Testing	504
11.5.5	Nondeterministic Primality Tests	505
11.5.6	Exercises for Section 11.5	508
11.6	Summary of Chapter 11	508
11.7	References for Chapter 11	510

Chapter 1

Automata: The Methods and the Madness

Automata theory is the study of abstract computing devices, or “machines.” Before there were computers, in the 1930’s, A. Turing studied an abstract machine that had all the capabilities of today’s computers, at least as far as in what they could compute. Turing’s goal was to describe precisely the boundary between what a computing machine could do and what it could not do; his conclusions apply not only to his abstract *Turing machines*, but to today’s real machines.

In the 1940’s and 1950’s, simpler kinds of machines, which we today call “finite automata,” were studied by a number of researchers. These automata, originally proposed to model brain function, turned out to be extremely useful for a variety of other purposes, which we shall mention in Section 1.1. Also in the late 1950’s, the linguist N. Chomsky began the study of formal “grammars.” While not strictly machines, these grammars have close relationships to abstract automata and serve today as the basis of some important software components, including parts of compilers.

In 1969, S. Cook extended Turing’s study of what could and what could not be computed. Cook was able to separate those problems that can be solved efficiently by computer from those problems that can in principle be solved, but in practice take so much time that computers are useless for all but very small instances of the problem. The latter class of problems is called “intractable,” or “NP-hard.” It is highly unlikely that even the exponential improvement in computing speed that computer hardware has been following (“Moore’s Law”) will have significant impact on our ability to solve large instances of intractable problems.

All of these theoretical developments bear directly on what computer scientists do today. Some of the concepts, like finite automata and certain kinds of formal grammars, are used in the design and construction of important kinds of software. Other concepts, like the Turing machine, help us understand what

we can expect from our software. Especially, the theory of intractable problems lets us deduce whether we are likely to be able to meet a problem “head-on” and write a program to solve it (because it is not in the intractable class), or whether we have to find some way to work around the intractable problem: find an approximation, use a heuristic, or use some other method to limit the amount of time the program will spend solving the problem.

In this introductory chapter, we begin with a very high-level view of what automata theory is about, and what its uses are. Much of the chapter is devoted to a survey of proof techniques and tricks for discovering proofs. We cover deductive proofs, reformulating statements, proofs by contradiction, proofs by induction, and other important concepts. A final section introduces the concepts that pervade automata theory: alphabets, strings, and languages.

1.1 Why Study Automata Theory?

There are several reasons why the study of automata and complexity is an important part of the core of Computer Science. This section serves to introduce the reader to the principal motivation and also outlines the major topics covered in this book.

1.1.1 Introduction to Finite Automata

Finite automata are a useful model for many important kinds of hardware and software. We shall see, starting in Chapter 2, examples of how the concepts are used. For the moment, let us just list some of the most important kinds:

1. Software for designing and checking the behavior of digital circuits.
2. The “lexical analyzer” of a typical compiler, that is, the compiler component that breaks the input text into logical units, such as identifiers, keywords, and punctuation.
3. Software for scanning large bodies of text, such as collections of Web pages, to find occurrences of words, phrases, or other patterns.
4. Software for verifying systems of all types that have a finite number of distinct states, such as communications protocols or protocols for secure exchange of information.

While we shall soon meet a precise definition of automata of various types, let us begin our informal introduction with a sketch of what a finite automaton is and does. There are many systems or components, such as those enumerated above, that may be viewed as being at all times in one of a finite number of “states.” The purpose of a state is to remember the relevant portion of the system’s history. Since there are only a finite number of states, the entire history generally cannot be remembered, so the system must be designed carefully, to

remember what is important and forget what is not. The advantage of having only a finite number of states is that we can implement the system with a fixed set of resources. For example, we could implement it in hardware as a circuit, or as a simple form of program that can make decisions looking only at a limited amount of data or using the position in the code itself to make the decision.

Example 1.1: Perhaps the simplest nontrivial finite automaton is an on/off switch. The device remembers whether it is in the “on” state or the “off” state, and it allows the user to press a button whose effect is different, depending on the state of the switch. That is, if the switch is in the off state, then pressing the button changes it to the on state, and if the switch is in the on state, then pressing the same button turns it to the off state.

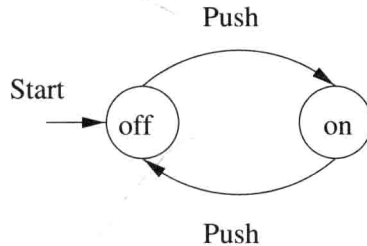


Figure 1.1: A finite automaton modeling an on/off switch

The finite-automaton model for the switch is shown in Fig. 1.1. As for all finite automata, the states are represented by circles; in this example, we have named the states *on* and *off*. Arcs between states are labeled by “inputs,” which represent external influences on the system. Here, both arcs are labeled by the input *Push*, which represents a user pushing the button. The intent of the two arcs is that whichever state the system is in, when the *Push* input is received it goes to the other state.

One of the states is designated the “start state,” the state in which the system is placed initially. In our example, the start state is *off*, and we conventionally indicate the start state by the word *Start* and an arrow leading to that state.

It is often necessary to indicate one or more states as “final” or “accepting” states. Entering one of these states after a sequence of inputs indicates that the input sequence is good in some way. For instance, we could have regarded the state *on* in Fig. 1.1 as accepting, because in that state, the device being controlled by the switch will operate. It is conventional to designate accepting states by a double circle, although we have not made any such designation in Fig. 1.1. □

Example 1.2: Sometimes, what is remembered by a state can be much more complex than an on/off choice. Figure 1.2 shows another finite automaton that could be part of a lexical analyzer. The job of this automaton is to recognize