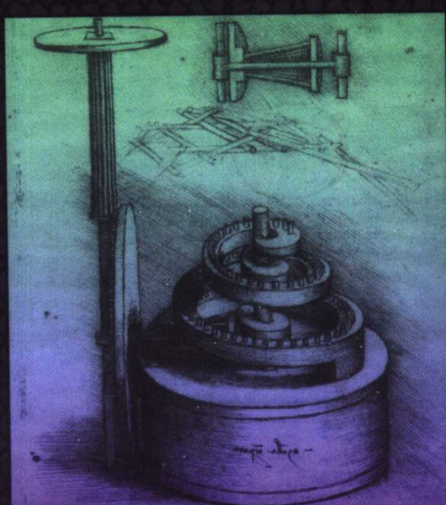
经典原版书库

计算理论导论

（英文版）

Introduction to the Theory of COMPUTATION



MICHAEL SIPSER

（美）Michael Sipser 著

机械工业出版社
China Machine Press

中信出版社
CITIC PUBLISHING HOUSE

THOMSON

经典原版书库

计算理论导论

(英文版)

Introduction to the Theory of Computation

(美) Michael Sipser 著

机械工业出版社
China Machine Press

中信出版社
CITIC PUBLISHING HOUSE

本书版权登记号：图字：01-2001-4914

图书在版编目（CIP）数据

计算理论导论/（美）塞普斯（Sipser, M.）著. - 北京：机械工业出版社，2002.8
（经典原版书库）
书名原文：Introduction to the Theory of Computation
ISBN 7-111-10840-X

Ⅰ. 计… Ⅱ. 塞… Ⅲ. 电子计算机 - 算法理论 - 英文 Ⅳ. TP301.6

中国版本图书馆CIP数据核字（2002）第063185号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）
责任编辑：华章
北京忠信诚印刷厂印刷 · 新华书店北京发行所发行
2002年8月第1版第1次印刷
787mm × 1092mm 1/16 · 25.75印张
印数：0 001-3 000册
定价：39.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

INTRODUCTION TO THE THEORY OF COMPUTATION

PREFACE

TO THE STUDENT

Welcome!

You are about to embark on the study of a fascinating and important subject: the theory of computation. It comprises the fundamental mathematical properties of computer hardware, software, and certain applications thereof. In studying this subject we seek to determine what can and cannot be computed, how quickly, with how much memory, and on which type of computational model. The subject has obvious connections with engineering practice, and, as in many sciences, it also has purely philosophical aspects.

I know that many of you are looking forward to studying this material but some may not be here out of choice. You may want to obtain a degree in computer science or engineering, and a course in theory is required—God knows why. After all, isn't theory arcane, boring, and worst of all, irrelevant?

To see that theory is neither arcane nor boring, but instead quite understandable and even interesting, read on. Theoretical computer science does have many fascinating big ideas, but it also has many small and sometimes dull details that can be tiresome. Learning any new subject is hard work, but it becomes easier and more enjoyable if the subject is properly presented. My primary objective in writing this book is to expose you to the genuinely exciting aspects of computer theory, without getting bogged down in the drudgery. Of course, the only way to determine whether theory interests you is to try learning it.

Theory is relevant to practice. It provides conceptual tools that practitioners use in computer engineering. Designing a new programming language for a specialized application? What you learned about *grammars* in this course comes in handy. Dealing with string searching and pattern matching? Remember *finite automata* and *regular expressions*. Confronted with a problem that seems to require more computer time than you can afford? Think back to what you learned about *NP-completeness*. Various application areas, such as modern cryptographic protocols, rely on theoretical principles that you will learn here.

Theory also is relevant to you because it shows you a new, simpler, and more elegant side of computers, which we normally consider to be complicated machines. The best computer designs and applications are conceived with elegance in mind. A theoretical course can heighten your aesthetic sense and help you build more beautiful systems.

Finally, theory is good for you because studying it expands your mind. Computer technology changes quickly. Specific technical knowledge, though useful today, becomes outdated in just a few years. Consider instead the abilities to think, to express yourself clearly and precisely, to solve problems, and to know when you haven't solved a problem. These abilities have lasting value. Studying theory trains you in these areas.

Practical considerations aside, nearly everyone working with computers is curious about these amazing creations, their capabilities, and their limitations. A whole new branch of mathematics has grown up in the past 30 years to answer certain basic questions. Here's a big one that remains unsolved: If I give you a large number, say, with 500 digits, can you find its factors (the numbers that divide it evenly), in a reasonable amount of time? Even using a supercomputer, no one presently knows how to do that in all cases *within the lifetime of the universe!* The factoring problem is connected to certain secret codes in modern cryptosystems. Find a fast way to factor and fame is yours!

TO THE EDUCATOR

This book is intended as an upper-level undergraduate or introductory graduate text in computer science theory. It contains a mathematical treatment of the subject, designed around theorems and proofs. I have made some effort to accommodate students with little prior experience in proving theorems, though more experienced students will have an easier time.

My primary goal in presenting the material has been to make it clear and interesting. In so doing, I have emphasized intuition and "the big picture" in the subject over some lower level details.

For example, even though I present the method of proof by induction in Chapter 0 along with other mathematical preliminaries, it doesn't play an important role subsequently. Generally I do not present the usual induction proofs of the correctness of various constructions concerning automata. If presented clearly, these constructions convince and do not need further argument. An induction may confuse rather than enlighten because induction itself is a rather sophisticated technique that many find mysterious. Belaboring the obvious with an induction risks teaching students that mathematical proof is a formal manipulation instead of teaching them what is and what is not a cogent argument.

A second example occurs in Parts II and III, where I describe algorithms in prose instead of pseudocode. I don't spend much time programming Turing machines (or any other formal model). Students today come with a programming background and find the Church–Turing thesis to be self-evident. Hence I don't present lengthly simulations of one model by another to establish their equivalence.

Besides giving extra intuition and suppressing some details, I give what might be called a classical presentation of the subject material. Most theorists will find the choice of material, terminology, and order of presentation consistent with that of other widely used textbooks. I have introduced original terminology in only a few places, when I found the standard terminology particularly obscure or confusing. For example I introduce the term *mapping reducibility* instead of *many–one reducibility*.

Practice through solving problems is essential to learning any mathematical subject. In this book, the problems are organized into two main categories called *Exercises* and *Problems*. The Exercises review definitions and concepts. The Problems require some ingenuity. Problems marked with a star are more difficult. I have tried to make both the Exercises and Problems interesting challenges.

THE CURRENT EDITION

Introduction to the Theory of Computation first appeared as a Preliminary Edition in paperback. The current edition differs from the Preliminary Edition in several substantial ways. The final three chapters are new: Chapter 8 on space complexity; Chapter 9 on provable intractability; and Chapter 10 on advanced topics in complexity theory. Chapter 6 was expanded to include several advanced topics in computability theory. Other chapters were improved through the inclusion of additional examples and exercises.

Comments from instructors and students who used the Preliminary Edition were helpful in polishing Chapters 0–7. Of course, the errors they reported have been corrected in this edition.

Chapters 6 and 10 give a survey of several more advanced topics in computability and complexity theories. They are not intended to comprise a cohesive unit in the way that the remaining chapters are. These chapters are included to allow the instructor to select optional topics that may be of interest to the serious student. The topics themselves range widely. Some, such as *Turing reducibility* and *alternation*, are direct extensions of other concepts in the book. Others, such as *decidable logical theories* and *cryptography*, are brief introductions to large fields.

FEEDBACK TO THE AUTHOR

The internet provides new opportunities for interaction between authors and readers. I have received much e-mail offering suggestions, praise, and criticism, and reporting errors for the Preliminary Edition. Please continue to correspond!

I try to respond to each message personally, as time permits. The e-mail address for correspondence related to this book is

`sipserbook@math.mit.edu`.

A web site that contains a list of errata is maintained. Other material may be added to that site to assist instructors and students. Let me know what you would like to see there. The location for that site is

`http://www-math.mit.edu/~sipser/book.html`.

ACKNOWLEDGMENTS

I could not have written this book without the help of many friends, colleagues, and my family.

I wish to thank the teachers who helped shape my scientific viewpoint and educational style. Five of them stand out. My thesis advisor, Manuel Blum, is due a special note for his unique way of inspiring students through clarity of thought, enthusiasm, and caring. He is a model for me and for many others. I am grateful to Richard Karp for introducing me to complexity theory, to John Addison for teaching me logic and assigning those wonderful homework sets, to Juris Hartmanis for introducing me to the theory of computation, and to my father for introducing me to mathematics, computers, and the art of teaching.

This book grew out of notes from a course that I have taught at MIT for the past 15 years. Students in my classes took these notes from my lectures. I hope they will forgive me for not listing them all. My teaching assistants over the years, Avrim Blum, Thang Bui, Andrew Chou, Benny Chor, Stavros Cosmadakis, Aditi Dhagat, Wayne Goddard, Parry Husbands, Dina Kravets, Jakov Kučan, Brian O'Neill, Ioana Popescu, and Alex Russell, helped me to edit and expand these notes and provided some of the homework problems.

Nearly three years ago, Tom Leighton persuaded me to write a textbook on the theory of computation. I had been thinking of doing so for some time, but it took Tom's persuasion to turn theory into practice. I appreciate his generous advice on book writing and on many other things.

I wish to thank Eric Bach, Peter Beebee, Cris Calude, Marek Chrobak, Anna Chefter, Guang-Ien Cheng, Elias Dahlhaus, Michael Fischer, Steve Fisk, Lance Fortnow, Henry J. Friedman, Jack Fu, Seymour Ginsburg, Oded Goldreich, Brian Grossman, David Harel, Micha Hofri, Dung T. Huynh, Neil Jones, H. Chad Lane, Kevin Lin, Michael Loui, Silvio Micali, Tadao Murata, Christos Papadimitriou, Vaughan Pratt, Daniel Rosenband, Brian Scassellati, Ashish Sharma, Nir Shavit, Alexander Shen, Ilya Shlyakhter, Matt Stallman, Perry Susskind, Y. C. Tay, Joseph Traub, Osamu Watanabe, Peter Widmayer, David Williamson, Derick Wood, and Charles Yang for comments, suggestions, and assistance as the writing progressed.

The following people provided additional comments that have improved this book: Isam M. Abdelhameed, Eric Allender, Michelle Atherton, Rolfe Blodgett, Al Briggs, Brian E. Brooks, Jonathan Buss, Jin Yi Cai, Steve Chapel, David Chow,

Michael Ehrlich, Yaakov Eisenberg, Farzan Fallah, Shaun Flisakowski, Hjalmtyr Hafsteinsson, C. R. Hale, Maurice Herlihy, Vegard Holmedahl, Sandy Irani, Kevin Jiang, Rhys Price Jones, James M. Jowdy, David M. Martin Jr., Manrique Mata-Montero, Ryota Matsuura, Thomas Minka, Farooq Mohammed, Tadao Murata, Jason Murray, Hideo Nagahashi, Kazuo Ohta, Constantine Papageorgiou, Joseph Raj, Rick Regan, Rhonda A. Reumann, Michael Rintzler, Arnold L. Rosenberg, Larry Roske, Max Rozenoer, Walter L. Ruzzo, Sanatan Sahgal, Leonard Schulman, Steve Seiden, Joel Seiferas, Ambuj Singh, David J. Stucki, Jayram S. Thathachar, H. Venkateswaran, Tom Whaley, Christopher Van Wyk, Kyle Young, and Kyoung Hwan Yun.

Robert Sloan used an early version of the manuscript for this book in a class that he taught and provided me with invaluable commentary and ideas from his experience with it. Mark Herschberg, Kazuo Ohta, and Latanya Sweeney read over parts of the manuscript and suggested extensive improvements. Shafi Goldwasser helped me with material in Chapter 10.

I received expert technical support from William Baxter at Superscript, who wrote the LaTeX macro package implementing the interior design, and from Larry Nolan at the MIT mathematics department, who keeps everything running.

It has been a pleasure to work with the folks at PWS Publishing in creating the final product. I mention Michael Sugarman, David Dietz, Elise Kaiser, Monique Calello, Susan Garland and Tanja Brull because I have had the most contact with them, but I know that many others have been involved, too. Thanks to Jerry Moore for the copy editing, to Diane Levy for the cover design, and to Catherine Hawkes for the interior design.

I am grateful to the National Science Foundation for support provided under grant CCR-9503322.

My father, Kenneth Sipser, and sister, Laura Sipser, converted the book diagrams into electronic form. My other sister, Karen Fisch, saved us in various computer emergencies, and my mother, Justine Sipser, helped out with motherly advice. I thank them for contributing under difficult circumstances, including insane deadlines and recalcitrant software.

Finally, my love goes to my wife, Ina, and my daughter, Rachel. Thanks for putting up with all of this.

Cambridge, Massachusetts | *Michael Sipser*
October, 1996

CONTENTS

0

INTRODUCTION

Let's begin with an overview of thöse areas in the theory of computation that we present in this course. Then, you'll have a chance to learn and/or review some mathematical concepts that you will need later.

0.1

AUTOMATA, COMPUTABILITY, AND COMPLEXITY

This book focuses on three traditionally central areas of the theory of computation: automata, computability, and complexity. They are linked by the question:

What are the fundamental capabilities and limitations of computers?

This question goes back to the 1930s when mathematical logicians first began to explore the meaning of computation. Technological advances since that time have greatly increased our ability to compute and have brought this question out of the realm of theory into the world of practical concern.

In each of the three areas—automata, computability, and complexity—this question is interpreted differently, and the answers vary according to the interpretation. Following this introductory chapter, we'll explore each area in a separate part of this book. Here, we introduce these parts in reverse order because starting from the end you can better understand the reason for the beginning.

COMPLEXITY THEORY

Computer problems come in different varieties; some are easy and some hard. For example, the sorting problem is an easy one. Say that you need to arrange a list of numbers in ascending order. Even a small computer can sort a million numbers rather quickly. Compare that to a scheduling problem. Say that you must find a schedule of classes for the entire university to satisfy some reasonable constraints, such as that no two classes take place in the same room at the same time. The scheduling problem seems to be much harder than the sorting problem. If you have just a thousand classes, finding the best schedule may require centuries, even with a supercomputer.

What makes some problems computationally hard and others easy?

This is the central question of complexity theory. Remarkably, we don't know the answer to it, though it has been intensively researched for the past 25 years. Later, we explore this fascinating question and some of its ramifications.

In one of the important achievements of complexity theory thus far, researchers have discovered an elegant scheme for classifying problems according to their computational difficulty. It is analogous to the periodic table for classifying elements according to their chemical properties. Using this scheme, we can demonstrate a method for giving evidence that certain problems are computationally hard, even if we are unable to prove that they are so.

You have several options when you confront a problem that appears to be computationally hard. First, by understanding which aspect of the problem is at the root of the difficulty, you may be able to alter it so that the problem is more easily solvable. Second, you may be able to settle for less than a perfect solution to the problem. In certain cases finding solutions that only approximate the perfect one is relatively easy. Third, some problems are hard only in the worst case situation, but easy most of the time. Depending on the application, you may be satisfied with a procedure that occasionally is slow but usually runs quickly. Finally, you may consider alternative types of computation, such as randomized computation, that can speed up certain tasks.

One applied area that has been affected directly by complexity theory is the ancient field of cryptography. In most fields, an easy computational problem is preferable to a hard one because easy ones are cheaper to solve. Cryptography is unusual because it specifically requires computational problems that are hard, rather than easy, because secret codes should be hard to break without the secret key or password. Complexity theory has pointed cryptographers in the direction of computationally hard problems around which they have designed revolutionary new codes.

COMPUTABILITY THEORY

During the first half of the twentieth century, mathematicians such as Kurt Gödel, Alan Turing, and Alonzo Church discovered that certain basic problems cannot be solved by computers. One example of this phenomenon is the problem of determining whether a mathematical statement is true or false. This task is the bread and butter of mathematicians. It seems like a natural for solution by computer because it lies strictly within the realm of mathematics. But no computer algorithm can perform this task.

Among the consequences of this profound result was the development of ideas concerning theoretical models of computers that eventually would help lead to the construction of actual computers.

The theories of computability and complexity are closely related. In complexity theory, the objective is to classify problems as easy ones and hard ones, whereas in computability theory the classification of problems is by those that are solvable and those that are not. Computability theory introduces several of the concepts used in complexity theory.

AUTOMATA THEORY

Automata theory deals with the definitions and properties of mathematical models of computation. These models play a role in several applied areas of computer science. One model, called the *finite automaton*, is used in text processing, compilers, and hardware design. Another model, called the *context-free grammar*, is used in programming languages and artificial intelligence.

Automata theory is an excellent place to begin the study of the theory of computation. The theories of computability and complexity require a precise definition of a *computer*. Automata theory allows practice with formal definitions of computation as it introduces concepts relevant to other nontheoretical areas of computer science.

0.2 MATHEMATICAL NOTIONS AND TERMINOLOGY

As in any mathematical subject, we begin with a discussion of the basic mathematical objects, tools, and notation that we expect to use.

SETS

A ***set*** is a group of objects represented as a unit. Sets may contain any type of object, including numbers, symbols, and even other sets. The objects in a set are called its ***elements*** or ***members***. Sets may be described formally in several ways. One way is by listing its elements inside braces. Thus the set

$$\{7, 21, 57\}$$

contains the elements 7, 21, and 57. The symbols $\in$ and $\notin$ denote set membership and nonmembership, respectively. We write $7 \in \{7, 21, 57\}$ and $8 \notin \{7, 21, 57\}$. For two sets A and B, we say that A is a ***subset*** of B, written $A \subseteq B$, if every