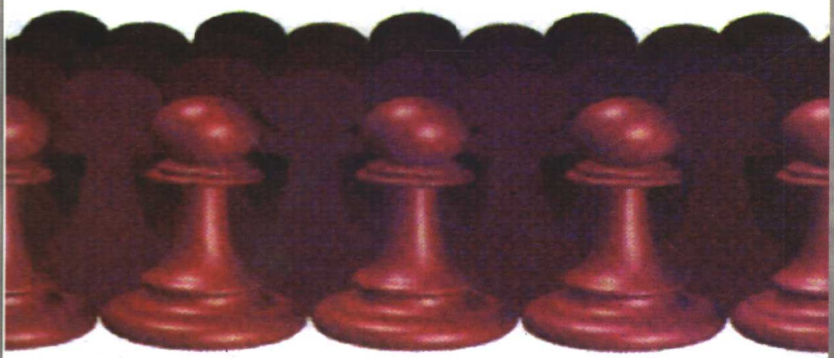
大学计算机教育丛书（影印版）

ELEMENTS OF THE THEORY OF COMPUTATION

Second Edition



计算理论基础

第2版

Harry R. Lewis / Christos H. Papadimitriou

清华大学出版社 • PRENTICE HALL
http://www.tup.tsinghua.edu.cn

ELEMENTS OF THE THEORY OF COMPUTATION

Second Edition

计算理论基础

第 2 版

Harry R. Lewis

Gordon McKay Professor of Computer Science

Harvard University

and Dean of Harvard College

Cambridge, Massachusetts

Christos H. Papadimitriou

C. Lester Hogan Professor of Electrical Engineering

and Computer Science

University of California

Berkeley, California

清华大学出版社

Prentice-Hall International, Inc.

(京) 新登字 158 号

北京市版权局著作权合同登记号: 01-1999-2523

图书在版编目(CIP)数据

计算理论基础:第 2 版 = Elements of The Theory of Compution 2E:英文/刘易斯(Lewis, H. R.),帕帕季米特里乌(Papadimiiow, C. H.)著. - 影印版. - 北京:清华大学出版社, 1999.7
(大学计算机教育丛书)
ISBN 7-302-03623-3

Ⅰ. 计… Ⅱ. ①刘… ②帕… Ⅲ. 计算技术-高等学校-教材-英文 Ⅳ. TP3

中国版本图书馆 CIP 数据核字(1999)第 30405 号

出版者: 清华大学出版社(北京清华大学学研大厦,邮编 100084)
http://www.tup.tsinghua.edu.cn
印刷者: 北京市振华印刷厂
发行者: 新华书店总店北京发行所
开 本: 850×1168 1/32 印张: 11.75
版 次: 1999 年 9 月第 1 版 2002 年 7 月第 4 次印刷
书 号: ISBN 7-302-03623-3/TP・2011
印 数: 9001～11000
定 价: 19.00 元

出版者的话

今天，我们的大学生、研究生和教学、科研工作者，面临的是一个国际化的信息时代。他们将需要随时查阅大量的外文资料；会有更多的机会参加国际性学术交流活动；接待外国学者；走上国际会议的讲坛。作为科技工作者，他们不仅应有与国外同行进行口头和书面交流的能力，更为重要的是，他们必须具备极强的查阅外文资料获取信息的能力。有鉴于此，在国家教委所颁布的“大学英语教学大纲”中有一条规定：专业阅读应作为必修课程开设。同时，在大纲中还规定了这门课程的学时和教学要求。有些高校除开设“专业阅读”课之外，还在某些专业课拟进行英语授课。但教、学双方都苦于没有一定数量的合适的英文原版教材作为教学参考书。为满足这方面的需要，我们陆续精选了一批国外计算机科学方面最新版本的著名教材，进行影印出版。我社获得国外著名出版公司和原著作者的授权将国际先进水平的教材引入我国高等学校，为师生们提供了教学用书，相信会对高校教材改革产生积极的影响。

我们欢迎高校师生将使用影印版教材的效果，意见反馈给我们，更欢迎国内专家、教授积极向我社推荐国外优秀计算机教育教材，以利我们将《大学计算机教育丛书（影印版）》做得更好，更适合高校师生的需要。

清华大学出版社

《大学计算机教育丛书（影印版）》项目组

1999.6

Preface to the First Edition

This book is an introduction, on the undergraduate level, to the classical and contemporary theory of computation. The topics covered are, in a few words, the theory of automata and formal languages, computability by Turing machines and recursive functions, uncomputability, computational complexity, and mathematical logic. The treatment is mathematical but the viewpoint is that of computer science; thus the chapter on context-free languages includes a discussion of parsing, and the chapters on logic establish the soundness and completeness of resolution theorem-proving.

In the undergraduate curriculum, exposure to this subject tends to come late, if at all, and collaterally with courses on the design and analysis of algorithms. It is our view that computer science students should be exposed to this material earlier —as sophomores or juniors— both because of the deeper insights it yields on specific topics in computer science, and because it serves to establish essential mathematical paradigms. But we have found teaching to a rigorous undergraduate course on the subject a difficult undertaking because of the mathematical maturity assumed by the more advanced textbooks. Our goal in writing this book has been to make the essentials of the subject accessible to a broad undergraduate audience in a way that is mathematically sound but presupposes no special mathematical experience.

The whole book represents about a year's worth of coursework. We have each taught a one-term course covering much of the material in Chapters 1 through 6, omitting on various occasions and in various combinations the sections of parsing, on recursive functions, and on particular unsolvable decision problems. Other selections are possible; for example, a course emphasizing computability and the foundations of mechanical logic might skip quickly over Chapters 1 through 3 and concentrate on Chapters 4, 6, 8, and 9. However, it is used, our fervent hope is that the book will contribute to the intellectual development

of the next generation of computer scientists by introducing them at an early stage of their education to crisp and methodical thinking about computational problems.

We take this opportunity to thank all from whom we have learned, both teachers and students. Specific thanks go to Larry Denenberg and Aaron Temin for their proofreading of early drafts, and to Michael Kahl and Oded Shmueli for their assistance and advice as teaching assistants. In the spring of 1980 Albert Meyer taught a course at M.I.T. from a draft of this book, and we thank him warmly for his criticisms and corrections. Of course, the blame for any remaining errors rests with us alone. Renate D'Arcangelo typed and illustrated the manuscript with her characteristic but extraordinary perfectionism and rapidity.

Preface to the Second Edition

Much has changed in the fifteen years since the *Elements of the Theory of Computation* first appeared —and much has remained the same. Computer science is now a much more mature and established discipline, playing a rôle of ever increasing importance in a world of ubiquitous computing, globalized information, and galloping complexity —more reasons to keep in touch with its foundations. The authors of the *Elements* are now themselves much more mature and busy —that is why this second edition has been so long in coming. We undertook it because we felt that a few things could be said better, a few made simpler —some even omitted altogether. More importantly, we wanted the book to reflect how the theory of computation, and its students, have evolved during these years. Although the theory of computation is now taught more widely in absolute terms, its relative position within the computer science curriculum, for example *vis à vis* the subject of algorithms, has not been strengthened. In fact, the field of the design and analysis of algorithms is now so mature, that its elementary principles are arguably a part of a basic course on the theory of computation. Besides, undergraduates today, with their extensive and early computational experience, are much more aware of the applications of automata in compilers, for example, and more suspicious when simple models such as the Turing machine are presented as general computers. Evidently, the treatment of these subjects needs some updating.

Concretely, these are the major differences from the first edition:

- Rudiments of the design and analysis of algorithms are introduced informally already in Chapter 1 (in connection with closures), and algorithmic questions are pursued throughout the book. There are sections on algorithmic problems in connection with finite automata and context-free grammars in Chapters 2 and 3 (including state minimization and context-free recognition), algorithms for easy variants of $\mathcal{NP}$-complete problems,

and a section that reviews algorithmic techniques for "coping with $\mathcal{NP}$-completeness" (special case algorithms, approximation algorithms, backtracking and branch-and-bound, local improvement, and simulated annealing algorithms).

- The treatment of Turing machines in Chapter 4 is more informal, and the simulation arguments are simpler and more quantitative. A random access Turing machine is introduced, helping bridge the gap between the clumsiness of Turing machines and the power of computers and programming languages.
- We included in Chapter 5 on undecidability some recursive function theory (up to Rice's Theorem). Grammars and recursive numerical functions are introduced and proved equivalent to Turing machines earlier, and the proofs are simpler. The undecidability of problems related to context-free grammars is proved by a simple and direct argument, without recourse to the Post correspondence problem. We kept the tiling problem, which we revisit in the $\mathcal{NP}$-completeness chapter.
- Complexity is done in a rather novel way: In Chapter 6, we define no other time bounds besides the polynomial ones —thus $\mathcal{P}$ is the first complexity class and concept encountered. Diagonalization then shows that there are exponential problems not in $\mathcal{P}$. Real-life problems are introduced side-by-side with their language representations (a distinction that is deliberately blurred), and their algorithmic questions are examined extensively.
- There is a separate $\mathcal{NP}$-completeness chapter with a new, extensive, and, we think, pedagogically helpful suite of $\mathcal{NP}$-completeness reductions, culminating with the equivalence problem for regular expressions —closing a full circle to the first subject of the book. As mentioned above, the book ends with a section on algorithmic techniques for "coping with $\mathcal{NP}$-completeness."
- There are no logic chapters in the new edition. This was a difficult decision, made for two reasons: According to all evidence, these were the least read and taught chapters of the book; and there are now books that treat this subject better. However, there is extensive treatment of Boolean logic and its satisfiability problems in Chapter 6.
- Overall, proofs and exposition have been simplified and made more informal at some key points. In several occasions, as in the proof of the equivalence of context-free languages and pushdown automata, long technical proofs of inductive statements have become exercises. There are problems following each section.

As a result of these changes, there is now at least one more way of teaching out of the material of this book (besides the ones outlined in the first edition, and the ones that emerged from its use): A semester-length course aiming at the

coverage of the basics of *both* the theory of computation and algorithms may be based on a selection of material from Chapters 2 through 7.

We want to express our sincere thanks to all of our students and colleagues who provided feedback, ideas, errors, and corrections during these fifteen years—it is impossible to come up with a complete list. Special thanks to Martha Sideri for her help with the revision of Chapter 3. Also, many thanks to our editor, Alan Apt, and the people at Prentice-Hall -- Barbara Kraemer, Sondra Chavez, and Bayani de Leon— who have been so patient and helpful.

Finally, we would appreciate receiving error reports or other comments, preferably by electronic mail to the address elements@cs.berkeley.edu. Confirmed errors, corrections, and other information about the book can also be obtained by writing to this address.

Contents

Introduction

Look around you. Computation happens everywhere, all the time, initiated by everybody, and affecting us all. Computation can happen because computer scientists over the past decades have discovered sophisticated methods for managing computer resources, enabling communication, translating programs, designing chips and databases, creating computers and programs that are faster, cheaper, easier to use, more secure.

As it is usually the case with all major disciplines, the practical successes of computer science build on its *elegant and solid foundations.* At the basis of physical sciences lie fundamental questions such as *what is the nature of matter?* and *what is the basis and origin of organic life?* Computer science has its own set of fundamental questions: *What is an algorithm? What can and what cannot be computed? When should an algorithm be considered practically feasible?* For more than sixty years (starting even before the advent of the eiectronic computer) computer scientists have been pondering these questions, *and coming up with ingenious answers that have deeply influenced computer science.*

The purpose of this book is to introduce you to these fundamental ideas, models, and results that permeate computer science, the *basic paradigms* of our field. They are worth studying, for many reasons. First, much of modern computer science is based more or less explicitly on them —and much of the rest should... Also, these ideas and models are powerful and beautiful, excellent examples of mathematical modeling that is elegant, productive, and of lasting value. Besides, they are so much a part of the history and the "collective subconscious" of our field, that it is hard to understand computer science without first being exposed to them.

It probably comes as no surprise that these ideas and models are *mathematical* in nature. Although a computer is undeniably a physical object, it is also

true that very little that is useful can be said of its physical aspects, such as its molecules and its shape; the most useful abstractions of a computer are clearly mathematical, and so the techniques needed to argue about them are necessarily likewise. Besides, practical computational tasks require the ironclad guarantees that only mathematics provides (we want our compilers to translate correctly, our application programs to eventually terminate, and so on). However, the mathematics employed in the theory of computation is rather different from the mathematics used in other applied disciplines. It is generally *discrete*, in that the emphasis is not on real numbers and continuous variables, but on finite sets and sequences. It is based on very few and elementary concepts, and draws its power and depth from the careful, patient, extensive, layer-by-layer manipulation of these concepts —just like the computer. In the first chapter you will be reminded of these elementary concepts and techniques (sets, relations, and induction, among others), and you will be introduced to the style in which they are used in the theory of computation.

The next two chapters, Chapters 2 and 3, describe certain restricted models of computation capable of performing very specialized string manipulation tasks, such as telling whether a given string, say the word *punk*, appears in a given text, such as the collective works of Shakespeare; or for testing whether a given string of parentheses is properly balanced —like () and (())(), but not)(). These restricted computational devices (called *finite-state automata* and *pushdown automata*, respectively) actually come up in practice as very useful and highly optimized components of more general systems such as circuits and compilers. Here they provide fine warm-up exercises in our quest for a formal, general definition of an algorithm. Furthermore, it is instructive to see how the power of these devices waxes and wanes (or, more often, is preserved) with the addition or removal of various features, most notably of *nondeterminism*, an intriguing aspect of computation which is as central as it is (quite paradoxically) unrealistic.

In Chapter 4 we study general models of algorithms, of which the most basic is the *Turing machine*,† a rather simple extension of the string-manipulating devices of Chapters 2 and 3 which turns out to be, surprisingly, a general frame-

† Named after Alan M. Turing (1912–1954), the brilliant English mathematician and philosopher whose seminal paper in 1936 marked the beginning of the theory of computation (and whose image, very appropriately, adorns the cover of this book). Turing also pioneered the fields of artificial intelligence and chess-playing by computer, as well as that of morphogenesis in biology, and was instrumental in breaking *Enigma*, the German naval code during World War II. For more on his fascinating life and times (and on his tragic end in the hands of official cruelty and bigotry) see the book *Alan Turing: The Enigma*, by Andrew Hodges, New York: Simon Schuster, 1983.

work for describing arbitrary algorithms. In order to argue this point, known as the *Church-Turing thesis*, we introduce more and more elaborate models of computation (more powerful variants of the Turing machine, even a *random access Turing machine* and *recursive definitions of numerical functions*), and show that they are all precisely equivalent in power to the basic Turing machine model.

The following chapter deals with *undecidability*, the surprising property of certain natural and well-defined computational tasks to lie *provably* beyond the reach of algorithmic solution. For example, suppose that you are asked whether we can use tiles from a given finite list of basic shapes to tile the whole plane. If the set of shapes contains a square, or even any triangle, then the answer is obviously "yes." But what if it consists of a few bizarre shapes, or if some of the shapes are mandatory, that is, they *must* be used at least once for the tiling to qualify? This is surely the kind of complicated question that you would like to have answered by a machine. In Chapter 5 we use the formalism of Turing machines to prove that this and many other problems *cannot be solved by computers at all.*

Even when a computational task is amenable to solution by *some* algorithm, it may be the case that there is no *reasonably fast, practically feasible* algorithm that solves it. In the last two chapters of this book we show how real-life computational problems can be categorized in terms of their *complexity:* Certain problems can be solved within reasonable, *polynomial* time bounds, whereas others seem to require amounts of time that grow astronomically, *exponentially.* In Chapter 7 we identify a class of common, practical, and notoriously difficult problems that are called $\mathcal{NP}$*-complete* (the traveling salesman problem is only one of them). We establish that all these problems are *equivalent* in that, if one of them has an efficient algorithm, then all of them do. It is widely believed that all $\mathcal{NP}$-complete problems are of inherently exponential complexity; whether this conjecture is actually true is the famous $\mathcal{P} \neq \mathcal{NP}$ problem, one of the most important and deep problems facing mathematicians and computer scientists today.

This book is very much about algorithms and their formal foundations. However, as you are perhaps aware, the subject of algorithms, their analysis and their design, is considered in today's computer science curriculum quite separate from that of the theory of computation. In the present edition of this book we have tried to restore some of the unity of the subject. As a result, this book also provides a decent, if somewhat specialized and unconventional, introduction to the subject of algorithms. Algorithms and their analysis are introduced informally in Chapter 1, and are picked up again and again in the context of the restricted models of computation studied in Chapters 2 and 3, and of the natural computational problems that they spawn. This way, when general models of algorithms are sought later, the reader is in a better position to appreciate the scope of the quest, and to judge its success. Algorithms play a

major role in our exposition of complexity as well, because there is no better way to appreciate a complex problem than to contrast it with another, amenable to an efficient algorithm. The last chapter culminates in a section on *coping with $\mathcal{NP}$-completeness*, where we present an array of algorithmic techniques that have been successfully used in attacking $\mathcal{NP}$-complete problems (approximation algorithms, exhaustive algorithms, local search heuristics, and so on).

Computation is essential, powerful, beautiful, challenging, ever-expanding —and so is its theory. This book only tells the beginning of an exciting story. It is a modest introduction to a few basic and carefully selected topics from the treasure chest of the theory of computation. We hope that it will motivate its readers to seek out more; the references at the end of each chapter point to good places to start.

1 Sets, Relations, and Languages

1.1 SETS

They say that mathematics is the language of science —it is certainly the language of the theory of computation, the scientific discipline we shall be studying in this book. And the language of mathematics deals with *sets*, and the complex ways in which they overlap, intersect, and in fact take part themselves in forming new sets.

A **set** is a collection of objects. For example, the collection of the four letters a, b, c, and d is a set, which we may name L; we write $L = \{a, b, c, d\}$. The objects comprising a set are called its **elements** or **members**. For example, b is an element of the set L; in symbols, $b \in L$. Sometimes we simply say that b is in L, or that L contains b. On the other hand, z is not an element of L, and we write $z \notin L$.

In a set we do not distinguish repetitions of the elements. Thus the set $\{\text{red}, \text{blue}, \text{red}\}$ is the same set as $\{\text{red}, \text{blue}\}$. Similarly, the order of the elements is immaterial; for example, $\{3, 1, 9\}$, $\{9, 3, 1\}$, and $\{1, 3, 9\}$ are the same set. To summarize: Two sets are equal (that is, the same) if and only if they have the same elements.

The elements of a set need not be related in any way (other than happening to be all members of the same set); for example, $\{3, \text{red}, \{d, \text{blue}\}\}$ is a set with three elements, one of which is itself a set. A set may have only one element; it is then called a **singleton**. For example, $\{1\}$ is the set with 1 as its only element; thus $\{1\}$ and 1 are quite different. There is also a set with no element at all. Naturally, there can be only one such set: it is called the **empty** set, and is denoted by $\emptyset$. Any set other than the empty set is said to be nonempty.

So far we have specified sets by simply listing all their elements, separated by commas and included in braces. Some sets cannot be written in this way,

because they are infinite. For example, the set $\mathbf{N}$ of natural numbers is infinite; we may suggest its elements by writing $\mathbf{N} = \{0, 1, 2, \ldots\}$, using the three dots and your intuition in place of an infinitely long list. A set that is not infinite is finite.

Another way to specify a set is by referring to other sets and to properties that elements may or may not have. Thus if $I = \{1, 3, 9\}$ and $G = \{3, 9\}$, G may be described as the set of elements of I that are greater than 2. We write this fact as follows.

$$G = \{x : x \in I \text{ and } x \text{ is greater than } 2\}.$$

In general, if a set A has been defined and P is a property that elements of A may or may not have, then we can define a new set

$$B = \{x : x \in A \text{ and } x \text{ has property } P\}.$$

As another example, the set of odd natural numbers is

$$O = \{x : x \in \mathbf{N} \text{ and } x \text{ is not divisible by } 2\}.$$

A set A is a **subset** of a set B — in symbols, $A \subseteq B$— if each element of A is also an element of B. Thus $O \subseteq \mathbf{N}$, since each odd natural number is a natural number. Note that any set is a subset of itself. If A is a subset of B but A is not the same as B, we say that A is a **proper subset** of B and write $A \subset B$. Also note that the empty set is a subset of every set. For if B is any set, then $\emptyset \subseteq B$, since each element of $\emptyset$ (of which there are none) is also an element of B.

To prove that two sets A and B are equal, we may prove that $A \subseteq B$ and $B \subseteq A$. Every element of A must then be an element of B and vice versa, so that A and B have the same elements and $A = B$.

Two sets can be combined to form a third by various *set operations*, just as numbers are combined by arithmetic operations such as addition. One set operation is **union**: the union of two sets is that set having as elements the objects that are elements of at least one of the two given sets, and possibly of both. We use the symbol $\cup$ to denote union, so that

$$A \cup B = \{x : x \in A \text{ or } x \in B\}.$$

For example,

$$\{1, 3, 9\} \cup \{3, 5, 7\} = \{1, 3, 5, 7, 9\}.$$

The **intersection** of two sets is the collection of all elements the two sets have in common; that is,

$$A \cap B = \{x : x \in A \text{ and } x \in B\}.$$