原 版 风 暴 系 列

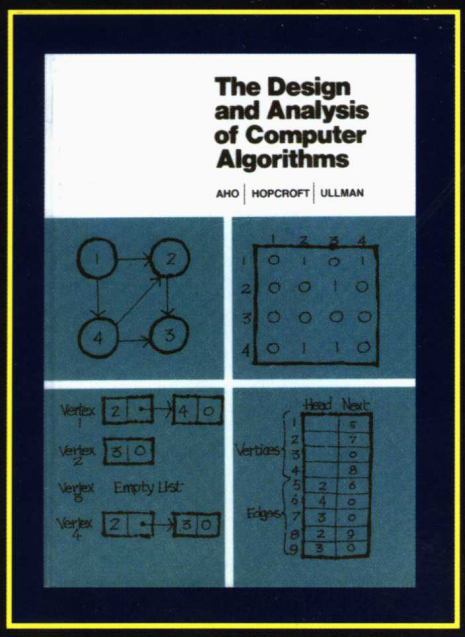
PEARSON
Addison Wesley

The Design and Analysis of Computer Algorithms

算法设计与分析

（影印版）

[美] Aho, Hopcroft, Ullman 著



- 3位美国国家工程院院士联手作品
- 算法设计与分析领域的绝对经典
- 无需数学和程序设计语言的专业背景

Pearson Education
中国电力出版社
www.infopower.com.cn

原 版 风 暴 系 列

The Design and Analysis of Computer Algorithms

算法设计与分析

（影印版）

[美] Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman 著

中国电力出版社

The Design and Analysis of Computer Algorithms (ISBN 0-201-00029-6)
Aho, Hopcroft, Ullman

北京市版权局著作合同登记号：图字：01-2003-1016

图书在版编目（CIP）数据

算法设计与分析 /（美）亚荷（Aho, A.V.）等编著. —影印本. —北京：中国电力出版社，2003
（原版风暴系列）
ISBN 7-5083-1804-8
Ⅰ.算… Ⅱ.亚… Ⅲ.①电子计算机－算法设计－英文
②电子计算机－算法分析－英文 Ⅳ.TP301.6
中国版本图书馆 CIP 数据核字（2003）第 086085 号

丛 书 名：原版风暴系列
书　　名：算法设计与分析（影印版）
编　　著：（美）Aho, Hopcroft, Ullman
责任编辑：夏平
出版发行：**中国电力出版社**
地址：北京市三里河路6号　　邮政编码：100044
电话：（010）88515918　　传　　真：（010）88518169
印　　刷：汇鑫印务有限公司
开　　本：787×1092 1/16　　印　　张：30.25
书　　号：ISBN 7-5083-1804-8
版　　次：2003年11月北京第一版　　2003年11月第一次印刷
定　　价：55.00 元

THE DESIGN AND ANALYSIS OF COMPUTER ALGORITHMS

Alfred V. Aho
Bell Laboratories

John E. Hopcroft
Cornell University

Jeffrey D. Ullman
Princeton University

PREFACE

The study of algorithms is at the very heart of computer science. In recent years a number of significant advances in the field of algorithms have been made. These advances have ranged from the development of faster algorithms, such as the fast Fourier transform, to the startling discovery of certain natural problems for which all algorithms are inefficient. These results have kindled considerable interest in the study of algorithms, and the area of algorithm design and analysis has blossomed into a field of intense interest. The intent of this book is to bring together the fundamental results in this area, so the unifying principles and underlying concepts of algorithm design may more easily be taught.

THE SCOPE OF THE BOOK

To analyze the performance of an algorithm some model of a computer is necessary. Our book begins by formulating several computer models which are simple enough to establish analytical results but which at the same time accurately reflect the salient features of real machines. These models include the random access register machine, the random access stored program machine, and some specialized variants of these. The Turing machine is introduced in order to prove the exponential lower bounds on efficiency in Chapters 10 and 11. Since the trend in program design is away from machine language, a high-level language called Pidgin ALGOL is introduced as the main vehicle for describing algorithms. The complexity of a Pidgin ALGOL program is related to the machine models.

The second chapter introduces basic data structures and programming techniques often used in efficient algorithms. It covers the use of lists, pushdown stores, queues, trees, and graphs. Detailed explanations of recursion, divide-and-conquer, and dynamic programming are given, along with examples of their use.

Chapters 3 to 9 provide a sampling of the diverse areas to which the fundamental techniques of Chapter 2 can be applied. Our emphasis in these chapters is on developing algorithms that are asymptotically the most efficient known. Because of this emphasis, some of the algorithms presented are suitable only for inputs whose size is much larger than what is currently encoun-

tered in practice. This is particularly true of some of the matrix multiplication algorithms in Chapter 6, the Schönhage-Strassen integer-multiplication algorithm of Chapter 7, and some of the polynomial and integer algorithms of Chapter 8.

On the other hand, most of the sorting algorithms of Chapter 3, the searching algorithms of Chapter 4, the graph algorithms of Chapter 5, the fast Fourier transform of Chapter 7, and the string-matching algorithms of Chapter 9 are widely used, since the sizes of inputs for which these algorithms are efficient are sufficiently small to be encountered in many practical situations.

Chapters 10 through 12 discuss lower bounds on computational complexity. The inherent computational difficulty of a problem is of universal interest, both to program design and to an understanding of the nature of computation. In Chapter 10 an important class of problems, the NP-complete problems, is studied. All problems in this class are equivalent in computational difficulty, in that if one problem in the class has an efficient (polynomial time-bounded) solution, then all problems in the class have efficient solutions. Since this class of problems contains many practically important and well-studied problems, such as the integer-programming problem and the traveling salesman problem, there is good reason to suspect that no problem in this class can be solved efficiently. Thus, if a program designer knows that the problem for which he is trying to find an efficient algorithm is in this class, then he may very well be content to try heuristic approaches to the problem. In spite of the overwhelming empirical evidence to the contrary, it is still an open question whether NP-complete problems admit of efficient solutions.

In Chapter 11 certain problems are defined for which we can actually prove that no efficient algorithms exist. The material in Chapters 10 and 11 draws heavily on the concept of Turing machines introduced in Sections 1.6 and 1.7.

In the final chapter of the book we relate concepts of computational difficulty to notions of linear independence in vector spaces. The material in this chapter provides techniques for proving lower bounds for much simpler problems than those considered in Chapters 10 and 11.

THE USE OF THE BOOK

This book is intended as a first course in the design and analysis of algorithms. The emphasis is on ideas and ease of understanding rather than implementation details or programming tricks. Informal, intuitive explanations are often used in place of long tedious proofs. The book is self-contained and assumes no specific background in mathematics or programming languages. However, a certain amount of maturity in being able to handle mathematical concepts is desirable, as is some exposure to a higher-level programming language such as FORTRAN or ALGOL. Some knowledge of linear algebra is needed for a full understanding of Chapters 6, 7, 8, and 12.

This book has been used in graduate and undergraduate courses in algorithm design. In a one-semester course most of Chapters 1–5 and 9–10 were covered, along with a smattering of topics from the remaining chapters. In introductory courses the emphasis was on material from Chapters 1–5, but Sections 1.6, 1.7, 4.13, 5.11, and Theorem 4.5 were generally not covered. The book can also be used in more advanced courses emphasizing the theory of algorithms. Chapters 6–12 could serve as the foundation for such a course.

Numerous exercises have been provided at the end of each chapter to provide an instructor with a wide range of homework problems. The exercises are graded according to difficulty. Exercises with no stars are suitable for introductory courses, singly starred exercises for more advanced courses, and doubly starred exercises for advanced graduate courses. The bibliographic notes at the end of every chapter attempt to point to a published source for each of the algorithms and results contained in the text and the exercises.

ACKNOWLEDGMENTS

The material in this book has been derived from class notes for courses taught by the authors at Cornell, Princeton, and Stevens Institute of Technology. The authors would like to thank the many people who have critically read various portions of the manuscript and offered many helpful suggestions. In particular we would like to thank Kellogg Booth, Stan Brown, Steve Chen, Allen Cypher,

Arch Davis, Mike Fischer, Hania Gajewska, Mike Garey, Udai Gupta, Mike Harrison, Matt Hecht, Harry Hunt, Dave Johnson, Marc Kaplan, Don Johnson, Steve Johnson, Brian Kernighan, Don Knuth, Richard Ladner, Anita LaSalle, Doug McIlroy, Albert Meyer, Christos Papadimitriou, Bill Plauger, John Savage, Howard Siegel, Ken Steiglitz, Larry Stockmeyer, Tom Szymanski, and Theodore Yen.

Special thanks go to Gemma Carnevale, Pauline Cameron, Hannah Kresse, Edith Purser, and Ruth Suzuki for their cheerful and careful typing of the manuscript.

The authors are also grateful to Bell Laboratories and Cornell, Princeton, and the University of California at Berkeley for providing facilities for the preparation of the manuscript.

June 1974

A. V. A.
J. E. H.
J. D. U.

CONTENTS

MODELS OF COMPUTATION

CHAPTER 1

Given a problem, how do we find an efficient algorithm for its solution? Once we have found an algorithm, how can we compare this algorithm with other algorithms that solve the same problem? How should we judge the goodness of an algorithm? Questions of this nature are of interest both to programmers and to theoretically oriented computer scientists. In this book we shall examine various lines of research that attempt to answer questions such as these.

In this chapter we consider several models of a computer—the random access machine, the random access stored program machine, and the Turing machine. We compare these models on the basis of their ability to reflect the complexity of an algorithm, and derive from them several more specialized models of computation, namely, straight-line arithmetic sequences, bitwise computations, bit vector computations, and decision trees. Finally, in the last section of this chapter we introduce a language called "Pidgin ALGOL" for describing algorithms.

1.1 ALGORITHMS AND THEIR COMPLEXITY

Algorithms can be evaluated by a variety of criteria. Most often we shall be interested in the rate of growth of the time or space required to solve larger and larger instances of a problem. We would like to associate with a problem an integer, called the *size* of the problem, which is a measure of the quantity of input data. For example, the size of a matrix multiplication problem might be the largest dimension of the matrices to be multiplied. The size of a graph problem might be the number of edges.

The time needed by an algorithm expressed as a function of the size of a problem is called the *time complexity* of the algorithm. The limiting behavior of the complexity as size increases is called the *asymptotic time complexity*. Analogous definitions can be made for *space complexity* and *asymptotic space complexity*.

It is the asymptotic complexity of an algorithm which ultimately determines the size of problems that can be solved by the algorithm. If an algorithm processes inputs of size n in time cn^2 for some constant c, then we say that the time complexity of that algorithm is $O(n^2)$, read "order n^2." More precisely, a function $g(n)$ is said to be $O(f(n))$ if there exists a constant c such that $g(n) \leq cf(n)$ for all but some finite (possibly empty) set of nonnegative values for n.

One might suspect that the tremendous increase in the speed of calculations brought about by the advent of the present generation of digital computers would decrease the importance of efficient algorithms. However, just the opposite is true. As computers become faster and we can handle larger problems, it is the complexity of an algorithm that determines the increase in problem size that can be achieved with an increase in computer speed.

Suppose we have five algorithms A_1–A_5 with the following time complexities.

Algorithm	Time complexity
A_1	n
A_2	$n \log n$†
A_3	n^2
A_4	n^3
A_5	2^n

The time complexity here is the number of time units required to process an input of size n. Assuming that one unit of time equals one millisecond, algorithm A_1 can process in one second an input of size 1000, whereas algorithm A_5 can process in one second an input of size at most 9. Figure 1.1 gives the sizes of problems that can be solved in one second, one minute, and one hour by each of these five algorithms.

Algorithm	Time complexity	Maximum problem size		
		1 sec	1 min	1 hour
A_1	n	1000	6×10^4	3.6×10^6
A_2	$n \log n$	140	4893	2.0×10^5
A_3	n^2	31	244	1897
A_4	n^3	10	39	153
A_5	2^n	9	15	21

Fig. 1.1. Limits on problem size as determined by growth rate.

Algorithm	Time complexity	Maximum problem size before speed-up	Maximum problem size after speed-up
A_1	n	s_1	$10s_1$
A_2	$n \log n$	s_2	Approximately $10s_2$ for large s_2
A_3	n^2	s_3	$3.16s_3$
A_4	n^3	s_4	$2.15s_4$
A_5	2^n	s_5	$s_5 + 3.3$

Fig. 1.2. Effect of tenfold speed-up.

† Unless otherwise stated, all logarithms in this book are to the base 2.

Suppose that the next generation of computers is ten times faster than the current generation. Figure 1.2 shows the increase in the size of the problem we can solve due to this increase in speed. Note that with algorithm A_5, a tenfold increase in speed only increases by three the size of problem that can be solved, whereas with algorithm A_3 the size more than triples.

Instead of an increase in speed, consider the effect of using a more efficient algorithm. Refer again to Fig. 1.1. Using one minute as a basis for comparison, by replacing algorithm A_4 with A_3 we can solve a problem six times larger; by replacing A_4 with A_2 we can solve a problem 125 times larger. These results are far more impressive than the twofold improvement obtained by a tenfold increase in speed. If an hour is used as the basis of comparison, the differences are even more significant. We conclude that the asymptotic complexity of an algorithm is an important measure of the goodness of an algorithm, one that promises to become even more important with future increases in computing speed.

Despite our concentration on order-of-magnitude performance, we should realize that an algorithm with a rapid growth rate might have a smaller constant of proportionality than one with a lower growth rate. In that case, the rapidly growing algorithm might be superior for small problems, possibly even for all problems of a size that would interest us. For example, suppose the time complexities of algorithms A_1, A_2, A_3, A_4, and A_5 were really $1000n$, $100n \log n$, $10n^2$, n^3, and 2^n. Then A_5 would be best for problems of size $2 \leq n \leq 9$, A_3 would be best for $10 \leq n \leq 58$, A_2 would be best for $59 \leq n \leq 1024$, and A_1 best for problems of size greater than 1024.

Before going further with our discussion of algorithms and their complexity, we must specify a model of a computing device for executing algorithms and define what is meant by a basic step in a computation. Unfortunately, there is no one computational model which is suitable for all situations. One of the main difficulties arises from the size of computer words. For example, if one assumes that a computer word can hold an integer of arbitrary size, then an entire problem could be encoded into a single integer in one computer word. On the other hand, if a computer word is assumed to be finite, one must consider the difficulty of simply storing arbitrarily large integers, as well as other problems which one often avoids when given problems of modest size. For each problem we must select an appropriate model which will accurately reflect the actual computation time on a real computer.

In the following sections we discuss several fundamental models of computing devices, the more important models being the random access machine, the random access stored program machine, and the Turing machine. These three models are equivalent in computational power but not in speed.

Perhaps the most important motivation for formal models of computation is the desire to discover the inherent computational difficulty of various problems. We would like to prove lower bounds on computation time. In order to show that there is no algorithm to perform a given task in less than a certain

amount of time, we need a precise and often highly stylized definition of what constitutes an algorithm. Turing machines (Section 1.6) are an example of such a definition.

In describing and communicating algorithms we would like a notation more natural and easy to understand than a program for a random access machine, random access stored program machine, or Turing machine. For this reason we shall also introduce a high-level language called Pidgin ALGOL. This is the language we shall use throughout the book to describe algorithms. However, to understand the computational complexity of an algorithm described in Pidgin ALGOL we must relate Pidgin ALGOL to the more formal models. This we do in the last section of this chapter.

1.2 RANDOM ACCESS MACHINES

A random access machine (RAM) models a one-accumulator computer in which instructions are not permitted to modify themselves.

A RAM consists of a read-only input tape, a write-only output tape, a program, and a memory (Fig. 1.3). The input tape is a sequence of squares, each of which holds an integer (possibly negative). Whenever a symbol is read from the input tape, the tape head moves one square to the right. The output is a write-only tape ruled into squares which are initially all blank. When a write instruction is executed, an integer is printed in the square of the

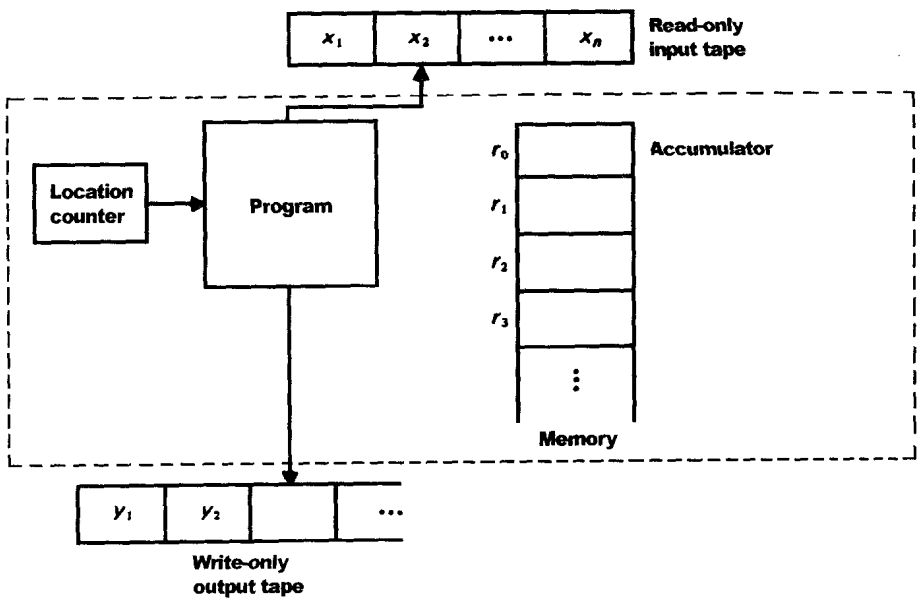


Fig. 1.3 A random access machine.