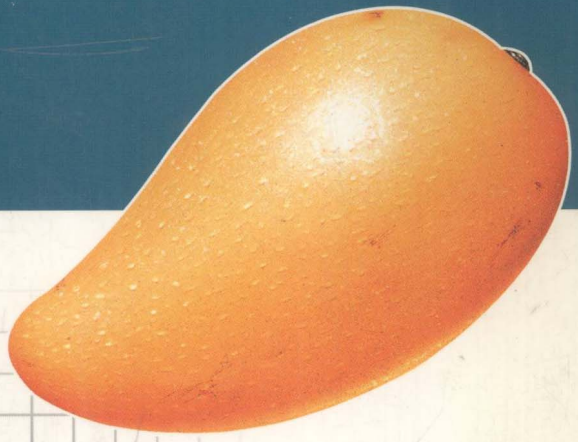
大学计算机教育国外著名教材系列（影印版）

PEARSON
Addison Wesley

DATA STRUCTURES AND ALGORITHMS

数据结构与算法



Alfred V. Aho
John E. Hopcroft 著
Jeffrey D. Ullman

清华大学出版社

大学计算机教育国外著名教材系列（影印版）

Data Structures and Algorithms

数据结构与算法

Alfred V. Aho

Bell Laboratories Murray Hill, New Jersey

John E. Hopcroft

Cornell University Ithaca, New York

Jeffrey D. Ullman

Stanford University Stanford, California

清华大学出版社

北京

北京市版权局著作权合同登记号　图字：01-2003-3539

本书封面贴有 Pearson Education(培生教育出版集团)激光防伪标签，无标签者不得销售。

图书在版编目(CIP)数据

数据结构与算法 = Data Structures and Algorithms / [美]阿霍（Aho, A. V.），[美]霍普克罗夫特（Hopcroft, J. E.），[美]厄尔曼（Ullman, J. D.）著. —影印本. —北京：清华大学出版社，2003
（大学计算机教育国外著名教材系列）
ISBN 7-302-07564-6

Ⅰ. 数… Ⅱ. ①阿… ②霍… ③厄… Ⅲ. ①数据结构－高等学校－教材－英文 ②算法分析－高等学校－教材－英文 Ⅳ. TP311.12

中国版本图书馆 CIP 数据核字（2003）第 100712 号

出 版 者：清华大学出版社　　**地　　址**：北京清华大学学研大厦
http://www.tup.com.cn　　**邮　　编**：100084
社总机：(010) 6277 0175　　**客户服务**：(010) 6277 6969
责任编辑：周维焜
印 刷 者：清华大学印刷厂
装 订 者：三河市新茂装订有限公司
发 行 者：新华书店总店北京发行所
开　　本：185×230　**印张**：27.5　**插页**：1
版　　次：2003 年 12 月第 1 版　2003 年 12 月第 1 次印刷
书　　号：ISBN 7-302-07564-6/TP・5563
印　　数：1~5000
定　　价：40.00 元

出 版 说 明

进入 21 世纪，世界各国的经济、科技以及综合国力的竞争将更加激烈。竞争的中心无疑是对人才的争夺。谁拥有大量高素质的人才，谁就能在竞争中取得优势。高等教育，作为培养高素质人才的事业，必然受到高度重视。目前我国高等教育的教材更新较慢，为了加快教材的更新频率，教育部正在大力促进我国高校采用国外原版教材。

清华大学出版社从 1996 年开始，与国外著名出版公司合作，影印出版了“大学计算机教育丛书（影印版）”等一系列引进图书，受到了国内读者的欢迎和支持。跨入 21 世纪，我们本着为我国高等教育教材建设服务的初衷，在已有的基础上，进一步扩大选题内容，改变图书开本尺寸，一如既往地请有关专家挑选适用于我国高校本科及研究生计算机教育的国外经典教材或著名教材，组成本套“大学计算机教育国外著名教材系列（影印版）”，以飨读者。深切期盼读者及时将使用本系列教材的效果和意见反馈给我们。更希望国内专家、教授积极向我们推荐国外计算机教育的优秀教材，以利我们把“大学计算机教育国外著名教材系列（影印版）”做得更好，更适合高校师生的需要。

清华大学出版社
2002 年 10 月

Preface

This book presents the data structures and algorithms that underpin much of today's computer programming. The basis of this book is the material contained in the first six chapters of our earlier work, *The Design and Analysis of Computer Algorithms*. We have expanded that coverage and have added material on algorithms for external storage and memory management. As a consequence, this book should be suitable as a text for a first course on data structures and algorithms. The only prerequisite we assume is familiarity with some high-level programming language such as Pascal.

We have attempted to cover data structures and algorithms in the broader context of solving problems using computers. We use abstract data types informally in the description and implementation of algorithms. Although abstract data types are only starting to appear in widely available programming languages, we feel they are a useful tool in designing programs, no matter what the language.

We also introduce the ideas of step counting and time complexity as an integral part of the problem solving process. This decision reflects our long-held belief that programmers are going to continue to tackle problems of progressively larger size as machines get faster, and that consequently the time complexity of algorithms will become of even greater importance, rather than of less importance, as new generations of hardware become available.

The Presentation of Algorithms

We have used the conventions of Pascal to describe our algorithms and data structures primarily because Pascal is so widely known. Initially we present several of our algorithms both abstractly and as Pascal programs, because we feel it is important to run the gamut of the problem solving process from problem formulation to a running program. The algorithms we present, however, can be readily implemented in any high-level programming language.

Use of the Book

Chapter 1 contains introductory remarks, including an explanation of our view of the problem-to-program process and the role of abstract data types in that process. Also appearing is an introduction to step counting and "big-oh" and "big-omega" notation.

Chapter 2 introduces the traditional list, stack and queue structures, and the mapping, which is an abstract data type based on the mathematical notion of a function. The third chapter introduces trees and the basic data structures

that can be used to support various operations on trees efficiently.

Chapters 4 and 5 introduce a number of important abstract data types that are based on the mathematical model of a set. Dictionaries and priority queues are covered in depth. Standard implementations for these concepts, including hash tables, binary search trees, partially ordered trees, tries, and 2-3 trees are covered, with the more advanced material clustered in Chapter 5.

Chapters 6 and 7 cover graphs, with directed graphs in Chapter 6 and undirected graphs in 7. These chapters begin a section of the book devoted more to issues of algorithms than data structures, although we do discuss the basics of data structures suitable for representing graphs. A number of important graph algorithms are presented, including depth-first search, finding minimal spanning trees, shortest paths, and maximal matchings.

Chapter 8 is devoted to the principal internal sorting algorithms: quicksort, heapsort, binsort, and the simpler, less efficient methods such as insertion sort. In this chapter we also cover the linear-time algorithms for finding medians and other order statistics.

Chapter 9 discusses the asymptotic analysis of recursive procedures, including, of course, recurrence relations and techniques for solving them.

Chapter 10 outlines the important techniques for designing algorithms, including divide-and-conquer, dynamic programming, local search algorithms, and various forms of organized tree searching.

The last two chapters are devoted to external storage organization and memory management. Chapter 11 covers external sorting and large-scale storage organization, including B-trees and index structures.

Chapter 12 contains material on memory management, divided into four subareas, depending on whether allocations involve fixed or varying sized blocks, and whether the freeing of blocks takes place by explicit program action or implicitly when garbage collection occurs.

Material from this book has been used by the authors in data structures and algorithms courses at Columbia, Cornell, and Stanford, at both undergraduate and graduate levels. For example, a preliminary version of this book was used at Stanford in a 10-week course on data structures, taught to a population consisting primarily of Juniors through first-year graduate students. The coverage was limited to Chapters 1-4, 9, 10, and 12, with parts of 5-7.

Exercises

A number of exercises of varying degrees of difficulty are found at the end of each chapter. Many of these are fairly straightforward tests of the mastery of the material of the chapter. Some exercises require more thought, and these have been singly starred. Doubly starred exercises are harder still, and are suitable for more advanced courses. The bibliographic notes at the end of each chapter provide references for additional reading.

Acknowledgments

We wish to acknowledge Bell Laboratories for the use of its excellent UNIX™-based text preparation and data communication facilities that significantly eased the preparation of a manuscript by geographically separated authors. Many of our colleagues have read various portions of the manuscript and have given us valuable comments and advice. In particular, we would like to thank Ed Beckham, Jon Bentley, Kenneth Chu, Janet Coursey, Hank Cox, Neil Immerman, Brian Kernighan, Steve Mahaney, Craig McMurray, Alberto Mendelzon, Alistair Moffat, Jeff Naughton, Kerry Nemovicher, Paul Niamkey, Yoshio Ohno, Rob Pike, Chris Rouen, Maurice Schlumberger, Stanley Selkow, Chengya Shih, Bob Tarjan, W. Van Snyder, Peter Weinberger, and Anthony Yeracaris for helpful suggestions. Finally, we would like to give our warmest thanks to Mrs. Claire Metzger for her expert assistance in helping prepare the manuscript for typesetting.

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

Contents

CHAPTER 1

Design and Analysis of Algorithms

There are many steps involved in writing a computer program to solve a given problem. The steps go from problem formulation and specification, to design of the solution, to implementation, testing and documentation, and finally to evaluation of the solution. This chapter outlines our approach to these steps. Subsequent chapters discuss the algorithms and data structures that are the building blocks of most computer programs.

1.1 From Problems to Programs

Half the battle is knowing what problem to solve. When initially approached, most problems have no simple, precise specification. In fact, certain problems, such as creating a "gourmet" recipe or preserving world peace, may be impossible to formulate in terms that admit of a computer solution. Even if we suspect our problem can be solved on a computer, there is usually considerable latitude in several problem parameters. Often it is only by experimentation that reasonable values for these parameters can be found.

If certain aspects of a problem can be expressed in terms of a formal model, it is usually beneficial to do so, for once a problem is formalized, we can look for solutions in terms of a precise model and determine whether a program already exists to solve that problem. Even if there is no existing program, at least we can discover what is known about this model and use the properties of the model to help construct a good solution.

Almost any branch of mathematics or science can be called into service to help model some problem domain. Problems essentially numerical in nature can be modeled by such common mathematical concepts as simultaneous linear equations (e.g., finding currents in electrical circuits, or finding stresses in frames made of connected beams) or differential equations (e.g., predicting population growth or the rate at which chemicals will react). Symbol and text processing problems can be modeled by character strings and formal grammars. Problems of this nature include compilation (the translation of programs written in a programming language into machine language) and information retrieval tasks such as recognizing particular words in lists of titles owned by a library.

Algorithms

Once we have a suitable mathematical model for our problem, we can attempt to find a solution in terms of that model. Our initial goal is to find a solution in the form of an *algorithm*, which is a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time. An integer assignment statement such as $x := y + z$ is an example of an instruction that can be executed in a finite amount of effort. In an algorithm instructions can be executed any number of times, provided the instructions themselves indicate the repetition. However, we require that, no matter what the input values may be, an algorithm terminate after executing a finite number of instructions. Thus, a program is an algorithm as long as it never enters an infinite loop on any input.

There is one aspect of this definition of an algorithm that needs some clarification. We said each instruction of an algorithm must have a "clear meaning" and must be executable with a "finite amount of effort." Now what is clear to one person may not be clear to another, and it is often difficult to prove rigorously that an instruction can be carried out in a finite amount of time. It is often difficult as well to prove that on any input, a sequence of instructions terminates, even if we understand clearly what each instruction means. By argument and counterargument, however, agreement can usually be reached as to whether a sequence of instructions constitutes an algorithm. The burden of proof lies with the person claiming to have an algorithm. In Section 1.5 we discuss how to estimate the running time of common programming language constructs that can be shown to require a finite amount of time for their execution.

In addition to using Pascal programs as algorithms, we shall often present algorithms using a *pseudo-language* that is a combination of the constructs of a programming language together with informal English statements. We shall use Pascal as the programming language, but almost any common programming language could be used in place of Pascal for the algorithms we shall discuss. The following example illustrates many of the steps in our approach to writing a computer program.

Example 1.1. A mathematical model can be used to help design a traffic light for a complicated intersection of roads. To construct the pattern of lights, we shall create a program that takes as input a set of permitted turns at an intersection (continuing straight on a road is a "turn") and partitions this set into as few groups as possible such that all turns in a group are simultaneously permissible without collisions. We shall then associate a phase of the traffic light with each group in the partition. By finding a partition with the smallest number of groups, we can construct a traffic light with the smallest number of phases.

For example, the intersection shown in Fig. 1.1 occurs by a watering hole called JoJo's near Princeton University, and it has been known to cause some navigational difficulty, especially on the return trip. Roads C and E are one-way, the others two way. There are 13 turns one might make at this

intersection. Some pairs of turns, like *AB* (from *A* to *B*) and *EC*, can be carried out simultaneously, while others, like *AD* and *EB*, cause lines of traffic to cross and therefore cannot be carried out simultaneously. The light at the intersection must permit turns in such an order that *AD* and *EB* are never permitted at the same time, while the light might permit *AB* and *EC* to be made simultaneously.

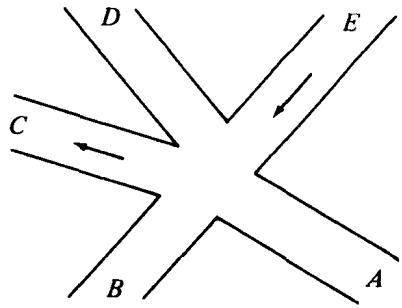


Fig. 1.1. An intersection.

We can model this problem with a mathematical structure known as a graph. A *graph* consists of a set of points called *vertices*, and lines connecting the points, called *edges*. For the traffic intersection problem we can draw a graph whose vertices represent turns and whose edges connect pairs of vertices whose turns cannot be performed simultaneously. For the intersection of Fig. 1.1, this graph is shown in Fig. 1.2, and in Fig. 1.3 we see another representation of this graph as a table with a 1 in row *i* and column *j* whenever there is an edge between vertices *i* and *j*.

The graph can aid us in solving the traffic light design problem. A *coloring* of a graph is an assignment of a color to each vertex of the graph so that no two vertices connected by an edge have the same color. It is not hard to see that our problem is one of coloring the graph of incompatible turns using as few colors as possible.

The problem of coloring graphs has been studied for many decades, and the theory of algorithms tells us a lot about this problem. Unfortunately, coloring an arbitrary graph with as few colors as possible is one of a large class of problems called "NP-complete problems," for which all known solutions are essentially of the type "try all possibilities." In the case of the coloring problem, "try all possibilities" means to try all assignments of colors to vertices using at first one color, then two colors, then three, and so on, until a legal coloring is found. With care, we can be a little speedier than this, but it is generally believed that no algorithm to solve this problem can be substantially more efficient than this most obvious approach.

We are now confronted with the possibility that finding an optimal solution for the problem at hand is computationally very expensive. We can adopt

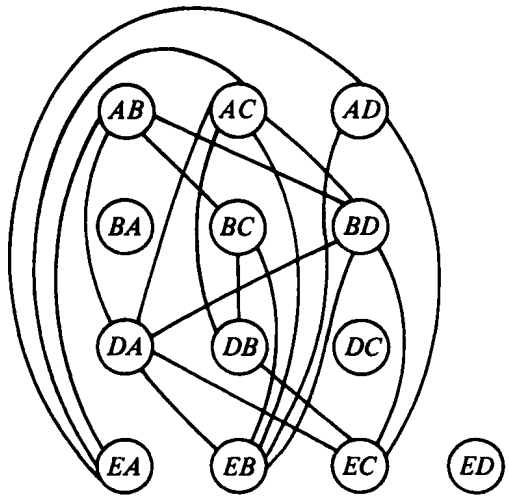


Fig. 1.2. Graph showing incompatible turns.

	AB	*AC*	*AD*	*BA*	*BC*	*BD*	*DA*	*DB*	*DC*	*EA*	*EB*	*EC*	*ED*
AB					1	1	1			1			
AC						1	1	1		1	1		
AD										1	1	1	
BA													
BC	1							1			1		
BD	1	1					1				1	1	
DA	1	1				1					1	1	
DB		1			1							1	
DC													
EA	1	1	1										
EB		1	1		1	1	1						
EC			1			1	1	1					
ED													

Fig. 1.3. Table of incompatible turns.

one of three approaches. If the graph is small, we might attempt to find an optimal solution exhaustively, trying all possibilities. This approach, however, becomes prohibitively expensive for large graphs, no matter how efficient we try to make the program. A second approach would be to look for additional information about the problem at hand. It may turn out that the graph has some special properties, which make it unnecessary to try all possibilities in finding an optimal solution. The third approach is to change the problem a little and look for a good but not necessarily optimal solution. We might be happy with a solution that gets close to the minimum number of colors on small graphs, and works quickly, since most intersections are not even as complex as Fig. 1.1. An algorithm that quickly produces good but not necessarily optimal solutions is called a *heuristic*.

One reasonable heuristic for graph coloring is the following "greedy" algorithm. Initially we try to color as many vertices as possible with the first color, then as many as possible of the uncolored vertices with the second color, and so on. To color vertices with a new color, we perform the following steps.

1. Select some uncolored vertex and color it with the new color.
2. Scan the list of uncolored vertices. For each uncolored vertex, determine whether it has an edge to any vertex already colored with the new color. If there is no such edge, color the present vertex with the new color.

This approach is called "greedy" because it colors a vertex whenever it can, without considering the potential drawbacks inherent in making such a move. There are situations where we could color more vertices with one color if we were less "greedy" and skipped some vertex we could legally color. For example, consider the graph of Fig. 1.4, where having colored vertex 1 red, we can color vertices 3 and 4 red also, provided we do not color 2 first. The greedy algorithm would tell us to color 1 and 2 red, assuming we considered vertices in numerical order.

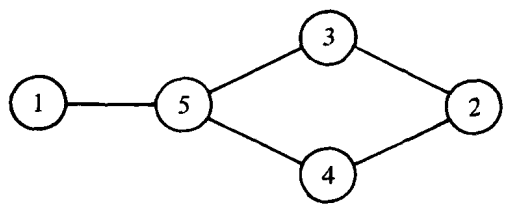


Fig. 1.4. A graph.

As an example of the greedy approach applied to Fig. 1.2, suppose we start by coloring *AB* blue. We can color *AC*, *AD*, and *BA* blue, because none of these four vertices has an edge in common. We cannot color *BC* blue because there is an edge between *AB* and *BC*. Similarly, we cannot color *BD*,

DA, or *DB* blue because each of these vertices is connected by an edge to one or more vertices already colored blue. However, we can color *DC* blue. Then *EA*, *EB*, and *EC* cannot be colored blue, but *ED* can.

Now we start a second color, say by coloring *BC* red. *BD* can be colored red, but *DA* cannot, because of the edge between *BD* and *DA*. Similarly, *DB* cannot be colored red, and *DC* is already blue, but *EA* can be colored red. Each other uncolored vertex has an edge to a red vertex, so no other vertex can be colored red.

The remaining uncolored vertices are *DA*, *DB*, *EB*, and *EC*. If we color *DA* green, then *DB* can be colored green, but *EB* and *EC* cannot. These two may be colored with a fourth color, say yellow. The colors are summarized in Fig. 1.5. The "extra" turns are determined by the greedy approach to be compatible with the turns already given that color, as well as with each other. When the traffic light allows turns of one color, it can also allow the extra turns safely.

color	turns	extras
blue	*AB*, *AC*, *AD*, *BA*, *DC*, *ED*	—
red	*BC*, *BD*, *EA*	*BA*, *DC*, *ED*
green	*DA*, *DB*	*AD*, *BA*, *DC*, *ED*
yellow	*EB*, *EC*	*BA*, *DC*, *EA*, *ED*

Fig. 1.5. A coloring of the graph of Fig. 1.2.

The greedy approach does not always use the minimum possible number of colors. We can use the theory of algorithms again to evaluate the goodness of the solution produced. In graph theory, a *k-clique* is a set of k vertices, every pair of which is connected by an edge. Obviously, k colors are needed to color a k-clique, since no two vertices in a clique may be given the same color.

In the graph of Fig. 1.2 the set of four vertices *AC*, *DA*, *BD*, *EB* is a 4-clique. Therefore, no coloring with three or fewer colors exists, and the solution of Fig. 1.5 is optimal in the sense that it uses the fewest colors possible. In terms of our original problem, no traffic light for the intersection of Fig. 1.1 can have fewer than four phases.

Therefore, consider a traffic light controller based on Fig. 1.5, where each phase of the controller corresponds to a color. At each phase the turns indicated by the row of the table corresponding to that color are permitted, and the other turns are forbidden. This pattern uses as few phases as possible. □

Pseudo-Language and Stepwise Refinement

Once we have an appropriate mathematical model for a problem, we can formulate an algorithm in terms of that model. The initial versions of the algorithm are often couched in general statements that will have to be refined subsequently into smaller, more definite instructions. For example, we described the greedy graph coloring algorithm in terms such as "select some uncolored vertex." These instructions are, we hope, sufficiently clear that the reader grasps our intent. To convert such an informal algorithm to a program, however, we must go through several stages of formalization (called *stepwise refinement*) until we arrive at a program the meaning of whose steps are formally defined by a language manual.

Example 1.2. Let us take the greedy algorithm for graph coloring part of the way towards a Pascal program. In what follows, we assume there is a graph G, some of whose vertices may be colored. The following program *greedy* determines a set of vertices called *newclr*, all of which can be colored with a new color. The program is called repeatedly, until all vertices are colored. At a coarse level, we might specify *greedy* in pseudo-language as in Fig. 1.6.

```
        procedure greedy ( var G: GRAPH; var newclr: SET );
            { greedy assigns to newclr a set of vertices of G that may be
                given the same color }
            begin
(1)             newclr := ∅; †
(2)             for each uncolored vertex v of G do
(3)                 if v is not adjacent to any vertex in newclr then begin
(4)                     mark v colored;
(5)                     add v to newclr
                    end
            end; { greedy }
```

Fig. 1.6. First refinement of *greedy* algorithm.

We notice from Fig. 1.6 certain salient features of our pseudo-language. First, we use boldface lower case keywords corresponding to Pascal reserved words, with the same meaning as in standard Pascal. Upper case types such as GRAPH and SET‡ are the names of "abstract data types." They will be defined by Pascal type definitions and the operations associated with these abstract data types will be defined by Pascal procedures when we create the final program. We shall discuss abstract data types in more detail in the next two sections.

The flow-of-control constructs of Pascal, like **if**, **for**, and **while**, are

† The symbol ∅ stands for the empty set.

‡ We distinguish the abstract data type SET from the built-in **set** type of Pascal.