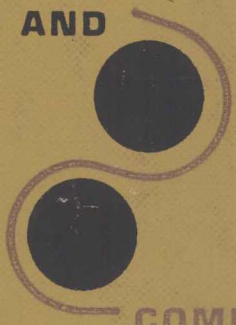
INTRODUCTION TO THE DESIGN AND ANALYSIS OF ALGORITHMS



COMPUTER SCIENCE SERIES

INTRODUCTION TO THE DESIGN AND ANALYSIS OF ALGORITHMS

S. E. Goodman
S. T. Hedetniemi
University of Virginia

McGraw-Hill Book Company
New York St. Louis San Francisco Auckland Bogotá Düsseldorf
Johannesburg London Madrid Mexico Montreal New Delhi
Panama Paris São Paulo Singapore Sydney Tokyo Toronto

To Dee and Judy

INTRODUCTION TO THE DESIGN AND ANALYSIS OF ALGORITHMS

234567890FGRFGR78321098

This book was set in Helvetica.
The editors were Peter D. Nalle and Laura D. Warner;
the designer was Laura D. Warner;
the production supervisor was Dennis J. Conroy.
The drawings were done by J & R Services, Inc.
Fairfield Graphics was printer and binder.

Library of Congress Cataloging in Publication Data

Goodman, Seymour E
Introduction to the design and analysis of algorithms.

(McGraw-Hill computer science series)
Includes bibliographical references and index.
1. Electronic digital computers- -Programming.
2. Algorithms. I. Hedetniemi, S. T., joint author. II. Title.
QA76.6.G66 511'.8 76-43363
ISBN 0-07-023753-0

INTRODUCTION TO THE DESIGN AND ANALYSIS OF ALGORITHMS

McGRAW-HILL
COMPUTER SCIENCE SERIES

RICHARD W. HAMMING
Bell Telephone Laboratories

EDWARD A. FEIGENBAUM
Stanford University

BELL and NEWELL Computer Structures: Readings and Examples
COLE Introduction to Computing
DONOVAN Systems Programming
GEAR Computer Organization and Programming
GIVONE Introduction to Switching Circuit Theory
GOODMAN and HEDETNIEMI Introduction to the Design and Analysis of Algorithms
HAMMING Computers and Society
HAMMING Introduction to Applied Numerical Analysis
HELLERMAN Digital Computer System Principles
HELLERMAN and CONROY Computer System Performance
KAIN Automata Theory: Machines and Languages
KATZAN Microprogramming Primer
KOHAVI Switching and Finite Automata Theory
LIU Elements of Discrete Mathematics
LIU Introduction to Combinatorial Mathematics
MADNICK and DONOVAN Operating Systems
MANNA Mathematical Theory of Computation
NEWMAN and SPROULL Principles of Interactive Computer Graphics
NILSSON Artificial Intelligence
RALSTON Introduction to Programming and Computer Science
ROSEN Programming Systems and Languages
SALTON Automatic Information Organization and Retrieval
STONE Introduction to Computer Organization and Data Structures
STONE and SIEWIOREK Introduction to Computer Organization and Data Structures: PDP-11 Edition
TONGE and FELDMAN Computing: An Introduction to Procedures and Procedure-Followers
TREMBLAY and MANOHAR Discrete Mathematical Structures with Applications to Computer Science
TREMBLAY and SORENSON An Introduction to Data Structures with Applications
TUCKER Programming Languages
WATSON Timesharing System Design Concepts
WEGNER Programming Languages, Information Structures, and Machine Organization
WIEDERHOLD Database Design
WINSTON The Psychology of Computer Vision

PREFACE

A representative sampling of undergraduate computer science courses would include the following (with varying titles): (1) introduction to computing, most often via Fortran, Basic, or PL/1; (2) assembly language programming; (3) data structures; (4) discrete structures; (5) machine organization; (6) numerical methods; (7) programming languages survey; (8) business data processing; (9) applications programming; and (10) systems programming. Through this book, we are proposing a course on the *design and analysis of algorithms*.

A first course in computing typically focuses on such topics as the workings of a computer, keypunching and data preparation, the syntax of a programming language, coding, input/output, the elementary aspects and uses of data structures, subroutine and function concepts, the art of debugging, the design of relatively simple programs, some machine-language concepts, and a few applications programs. There are a number of additional topics on programming which are neither covered in a first course in computing nor covered in much detail in any other undergraduate computer science course. These topics include:

1. The complete development, from start to finish, of a reasonably complicated problem for computer solution
2. Algorithm design techniques, such as subgoals, hill climbing, working backward, backtracking, branch and bound, recursion, and heuristics
3. Efficient and correct implementation of stated algorithms
4. Algorithm and program correctness (a matter that all too frequently encourages the question: Does the output look all right?)
5. Measures of algorithm efficiency, complexity, and overall effectiveness
6. Program testing, including tests for correctness, complexity, and general program behavior
7. More sophisticated mathematical thinking (involving probability, for example) required in designing and analyzing programs of reasonable complexity

This text is concerned with all the above-mentioned aspects of computing.

Although a course in the design and analysis of algorithms necessarily involves a significant amount of programming, it is not meant to serve simply as a second (or third) programming course. Consequently, a number of topics which might relate to the design and analysis of algorithms are not discussed in this text; these topics include matters of I/O, debugging techniques, optimizing compilers, and use of library routines. This text is intended to serve as a bridge between the more practical, programming-oriented courses and the more theoretical, mathematically oriented courses in computer science. The algorithms presented have a distinct mathematical flavor because of this orientation and the instructive nature of their design and analysis.

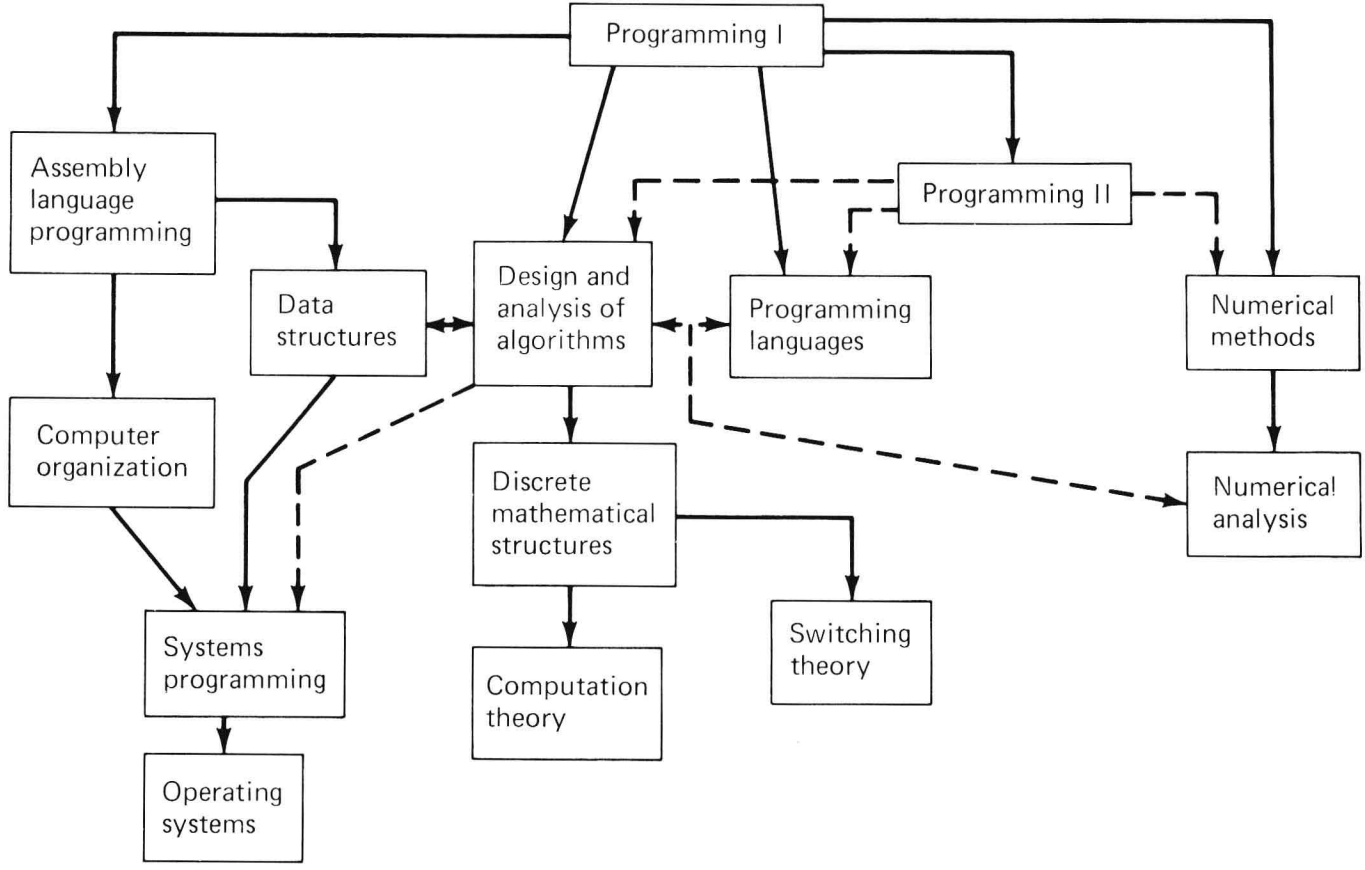
The figure on page ix shows how a course on the design and analysis of algorithms can fit into a typical undergraduate computer science curriculum. To some extent, we see this course as a substitute for the introduction to discrete structures, the course listed in the ACM Curriculum 68 (*Comm. ACM*, Mar. 1968). Although our subject matter has a healthy mathematical content, the level is lower, less theoretical, and less formal than that of an introduction to discrete structures; and the applications to computer science are more apparent.

The mathematical content in the design and analysis of algorithms comprises introductory parts of the subjects of networks, combinatorics, probability, and statistics. These topics are covered not for their own sake but rather for use in algorithmic applications. To establish the correctness or properties of certain algorithms, proofs are presented throughout the book, many of which use mathematical induction. We do not attempt to teach the student how to prove the theorems, but we do expect that he or she will be able to follow their logic.

We also believe that the subject matter in this text plays a central role in computer science. In addition to serving as the first theoretical course in a computer science curriculum, the subject relates well to courses in data structures, programming languages, applications programming, and numerical analysis.

In principle, a student whose background includes one semester of calculus, a first computer programming course, and a high-school-level knowledge of permutations, combinations, and sets should be adequately prepared to read this book. Calculus is used infrequently, but it is regarded as a prerequisite to ensure some mathematical maturity. We have found that the material is slightly difficult for most sophomores and appears to be most appropriate for the junior or senior level.

Since implementation and testing of programs are an important part of the development of algorithms, computer code has to be exhibited. Although some computer scientists may not agree with our choice, we have decided to exhibit all programs in Fortran for the simple reason that it is the only programming language that is close to being universally known. There does not appear to be an alternative that neither forces additional prerequisites nor makes the book so long as to dilute the material of primary interest. For those readers who are familiar with more structured programming languages, we have included an appendix containing Algol or PL/1 equivalents of most of the programs in the text.



Denotes prerequisite

Denotes desired prerequisite or corequisite

Our format for the presentation of algorithms is a cross between the heavily annotated, step-by-step style popularized by Knuth in the first three volumes of *The Art of Computer Programming* and the Algol-like format that is currently popular in the literature. Constructs such as **do-while** and **if-then-else** are used regularly and implemented in obvious ways. A reasonable effort has been made to adhere to the principles of structured programming. Appendix A summarizes the conventions used to state algorithms in this text.

Chapter 1 is primarily a qualitative introduction to the notions of an algorithm and the basic steps that go into its complete development. Unfortunately, considerations of space do not permit us to develop most of the algorithms in this text in this much detail. In later chapters, some of the development steps have been left as exercises.

Chapter 2 develops several tools that are useful in the design and analysis of algorithms. Elementary concepts of structured programming, networks, data structures, probability, and statistics are considered. For supplementation, an appendix—on sets and elementary proof techniques—is provided at the end of the book. An effort has been made to retain an "algorithmic flavor" throughout this chapter and the appendices and, whenever possible, new algorithms are introduced to illustrate concepts.

A statement should be made about the elementary aspects of probability and statistics covered in Chap. 2. Many interesting and important problems in the design and analysis of algorithms are probabilistic in nature; for example, a strong argument can be made that the most useful measure of the quality of an algorithm is its average or expected performance. On the other hand, we recognize that many students have difficulty with the study of probability and statistics. With this in mind, we have written this text so that an instructor may omit probabilistic material without much trouble.

A number of useful algorithm design techniques are studied in Chap. 3, in which each section includes at least one new problem and/or algorithm. An algorithm for finding a minimum-weight spanning tree is completely developed in Chap. 4, and it is used to illustrate some simple program-testing procedures.

Chapters 5 and 6 contain a variety of examples and applications, most of which reinforce ideas introduced in the first four chapters. Instructors can select topics from these chapters on the basis of class interest, available time, mathematical level, and so forth. Chapter 7 is designed as a reference tool for the reader.

Almost 300 exercises have been included in this text. They vary widely in difficulty; many are open-ended and experimental. The exercises are an important part of the book, and it is hoped that instructors will find time to discuss some of the more interesting and difficult problems in class. Note that the more asterisks (*) preceding an exercise, the greater its difficulty. An exercise with an "L" label will require considerable time. Some of the exercises are "titled" to indicate the coverage of certain subject matter or concepts. Instructors might consider using groups of two or three students for some of the longer and more difficult assignments, particularly those that require extensive program testing.

It is difficult to acknowledge everyone who has contributed time and effort to

this book. Many people have read it and offered their comments, constructive criticism, and encouragement; unfortunately, it is not possible to recognize them all individually. Above all, we thank Diane Goodman, our typist and chief debugger. Dana Richards contributed more than half of Sec. 6.1, and Bruce Chartres is responsible for the ecological model in Sec. 3.6. Others who have made major contributions to the manuscript include Linwood Ferguson, Richard Armentrout, and Clay Pendergrast. Thanks are also due to Art Fleck, Harold Stone, Ken Bowman, Wayne Madison, and an anonymous reader for their useful comments. Mouton Publishers kindly gave us permission to use about 12 figures and several pages of text in Sec. 2.2. Diane Spresser, Sandee Mitchell, and Jennifer Ward helped with the exercises and the proofreading of the manuscript. George Eade and Louis Rowley wrote the Algol and PL/1 programs in Appendix C. Last, but not least, we would like to thank the students in our course in the design and analysis of algorithms, who have been such willing guinea pigs for the last three years.

No book of this sort can ever be completely free of "bugs." In spite of the efforts of many people, there undoubtedly remain errors in the text. We would appreciate having these brought to our attention by our readers.

S. E. Goodman
S. T. Hedetniemi

CONTENTS

1 THE COMPLETE DEVELOPMENT OF AN ALGORITHM

1.1 INTRODUCTION

It is only fair to start by telling you what we hope to accomplish. Throughout your academic and professional careers people will present you with problems, and you will be expected to use your background in mathematics and computer science to solve them. This will often entail the development of an algorithm. Our purpose in this text is to help you learn how to (1) get started on a problem, (2) design an algorithm that works, (3) implement the algorithm as a computer program, and (4) judge the effectiveness of an algorithm.

This is easier said than done. The world of computing is littered with the remains of computer programs which were once considered to be finished products, but which later were found to be incorrect, inefficient, unintelligible, or worthless for some other reason. Perhaps the simplest explanation for this is that the computer is a relatively new and complex tool and it takes time to learn how to use it well. The skills necessary to use a computer as a powerful instrument for solving problems, particularly mathematical problems, are not easy to acquire.

The objective of this chapter is to make a "first pass" at the concept of the *complete development of an algorithm*, the basic steps of which are:

1. Statement of the problem
2. Development of a model
3. Design of the algorithm
4. Correctness of the algorithm
5. Implementation
6. Analysis and complexity of the algorithm
7. Program testing
8. Documentation

The remainder of the book is devoted to the detailed study and illustration of these fundamental steps. For the time being we shall limit ourselves to brief discussions of algorithms (Sec. 1.2) and the steps leading to their complete development (Sec. 1.3).

1.2 ALGORITHMS

Everyone who has solved problems with the aid of a digital computer has some intuitive idea of the meaning of the word "algorithm." One might even go so far as to argue that it is one of the most central concepts in computer science. The "official" definition from the most recent (1971) edition of the *Oxford English Dictionary* states that the word "algorithm" is an "erroneous refashioning of algorism"; in turn, algorism is considered to be more or less synonymous with algebra and arithmetic, and this definition dates back to the ninth century. Apparently the modern usage of the word is still strictly limited to the computer science community.

Let us first try to verbalize our intuitive ideas. We can loosely define an *algorithm* as an unambiguous procedure for solving a problem. A *procedure* is a finite sequence of well-defined steps or operations, each of which requires only a finite amount of memory or working storage and takes a finite amount of time to complete. We append the requirement that an algorithm must terminate in finite time for any input.

One difficulty with this definition is that the term "unambiguous" is very ambiguous. "Unambiguous" to whom? Or to what? Since nothing is universally clear or vague, the executor of the algorithm must be, at least implicitly, specified. An algorithm for computing the derivative of a cubic polynomial might be perfectly clear to someone who is familiar with calculus, but it may be totally incomprehensible to someone who is not. Thus, the executor's computational capabilities must also be specified.

There are other definitional problems. An algorithm may clearly exist for some task, but it may be difficult or impossible to describe in some given format. The human race has clearly developed efficient algorithms for tying shoelaces. Many children can tie their own shoes by the age of five. But it is very difficult (try it) to give a purely verbal—no pictures or demonstrations allowed—statement of such an algorithm.

This definition obviously has some defects. It is possible to avoid most of these defects by defining "mathematical machines" with very carefully specified capabilities. We then say that an algorithm is any procedure which can be executed by such a machine. Such attempts at defining an algorithm are very deep and difficult mathematically; they are also much too rigid for our purposes.

We would like to retain some of the flexibility and intuitive appeal of the first definition, and yet at least partially remove some of its ambiguities. This is easily done by specifying a "typical" modern digital computer and a language of communication with such a computer, and then licensing a procedure as an algorithm if it can be implemented on this machine using the given language.

This computer will have an unlimited random-access memory in which real numbers, integers, and logical constants can be stored. In one word of this memory we can store a number of arbitrary, but finite, size and can access any word in a fixed, constant amount of time (this may be a bit unrealistic, but it is almost true in practice and is a convenient assumption). This computer is

capable of executing a stored program that consists of a reasonable collection of instructions of a basic type, including all standard arithmetic operations, comparisons, branchings, etc. We will usually assign one unit of execution time to each such instruction. Some of our memory can be organized into one-, two-, and three-dimensional arrays (matrices) by simple declarations. When we want to be specific, we will use the Fortran language as a model for the capabilities of our machine.[1]

We have effectively said that we only have an algorithm for solving a problem when we can write a computer program that solves it. This is a debatable issue. Programs on the kind of computer described in the previous paragraph cannot tie shoes. The well-defined steps do not include those necessary to successfully tie shoes. A good argument can be made that we are really dealing with a restricted concept of an algorithm. The human machine is capable of executing a large variety of subtle steps that are beyond the range of our typical computer. However, the restricted definition is just what we want for this book.

The following example of an algorithm illustrates a level of detail that is consistent with our definition. Appendix A contains a detailed discussion of the conventions used to state algorithms in this text.

Consider the simple problem of finding the maximum number in a list of N real numbers R(1), R(2), . . . , R(N). The basic idea of the algorithm is to go through the entire list, one number at a time, and remember the largest number that we have seen so far. By the time the entire list is inspected, the largest number will have been retained. You might try to draw a flowchart for this algorithm before reading further.

The notation $A \leftarrow B$ denotes an assignment statement; that is, set variable A equal to the current value of B.

Algorithm MAX Given N real numbers in a one-dimensional array R(1), R(2), . . . , R(N), find M and J such that

$$M = R(J) = \max_{1 \le K \le N} R(K)$$

In the case where two or more elements of R have the largest value, the value of J retained will be the smallest possible.

Step 0. [Initialize] **Set** $M \leftarrow R(1)$; **and** $J \leftarrow 1$.

Step 1. [N = 1?] **If** N = 1 **then** STOP **fi**.†

Step 2. [Inspect each number] **For** $K \leftarrow 2$ **to** N **do** step 3 **od**; **and** STOP.

Step 3. [Compare] **If** $M < R(K)$ **then set** $M \leftarrow R(K)$; **and** $J \leftarrow K$ **fi**.† (M is now the largest number we have inspected, and it is in the K*th* position of the array.)

[1]Many of the algorithms in this book have also been coded in Algol or PL/1. See Appendix C.

†The **fi** and **od** in this algorithm are used to denote the end of the **if** and **do** constructs, respectively. This will be discussed in more detail in Sec. 2.1.

Algorithm MAX is not in coded form. It is in a form that is generally easier to follow than computer code, but it is expressed in terms of steps which are available in every common computer language. The conversion to coded form is easy. However, this is not always the case. Some algorithms are too complicated for us to make the transition from the preceding verbal form to computer code in one step. At least one intermediate stage of development may have to be introduced.

1.3 THE BASIC STEPS IN THE COMPLETE DEVELOPMENT OF AN ALGORITHM

We will now briefly consider each of the basic steps listed near the end of Sec. 1.1. Our primary interest is to establish the function of each step and to gain some perspective on how these steps combine to form a coherent whole.

Statement of the Problem

Before we can understand a problem, we must be able to give it a precise statement. This condition is not, in itself, sufficient for understanding a problem, but it is absolutely necessary.

Developing a precise problem statement is usually a matter of asking the right questions. Some good questions to ask upon encountering a crudely formulated problem are:

Do I understand the vocabulary used in the raw formulation?

What information has been given?

What do I want to find out?

How would I recognize a solution?

What information is missing, and will any of this information be of use?

Is any of the given information worthless?

What assumptions have been made?

Other questions are possible, depending on the particular problem. Often questions such as these need to be asked again, after some of them have been given answers or partial answers.

Example Jack is a computer marketing representative (salesman) whose territory covers 20 cities scattered throughout Texas. He works for large commissions, but his company will reimburse him for only 50 percent of the actual cost of automobile travel for his business trips. Jack has taken the trouble to figure out how much it would cost him to travel by car between every

pair of cities in his territory. He would clearly like to keep his travel costs down.

What is given? The primary information is a list of the cities in Jack's territory and the associated cost matrix, that is, a square array with entry c_{ij} equal to the cost of going from city i to city j. In this case the cost matrix has 20 rows and 20 columns.

What do we want to find out? We want to help Jack keep his travel costs down. That is a bit vague. It really looks inadequate when we ask: How would we recognize a solution? After giving the matter some thought, we should conclude that we cannot do any better without additional information from Jack. Does Jack have more customers in some cities than in others? If he does, or if he has some special customers, Jack might want to visit certain cities more often. There may be other cities that Jack would not bother to visit unless he happened to find himself in a nearby city. In other words, we must know more about Jack's priorities and schedule preferences.

Therefore, we go back to Jack and ask him for additional information. He tells us that he would like an itinerary that would start at his base city, take him to each of the other cities in his territory exactly once, and return him to his base. Consequently, we would like a list of cities which contains each city exactly once, except for the base city which is listed first and last. The order of the cities on this list represents the order in which Jack should make the tour of his territory. The sum of the costs between every consecutive pair of cities on the list is the total cost of the tour represented by the list. We could solve Jack's problem if we could give him the list with the smallest possible total cost.

This is a good basic statement of the problem. We know what we have and what we want to find.

Development of a Model

Once a problem has been clearly stated, it is time to formulate it as a mathematical model. This is a very important step in the overall solution process and it should be given considerable thought. The choice of a model has substantial influence on the remainder of the solution process.

As you might imagine, it is impossible to provide a set of rules which automates the modeling stage. Most problems must be given individual attention. However, there are some useful guidelines. This topic is more of an art than a science and is likely to remain that way. The best way to become proficient is by acquiring the experience that comes from the study of successful models.

There are at least two basic questions to be asked in setting up a model:

1 Which mathematical structures seem best-suited for the problem?
2 Are there other problems that have been solved which resemble this one?