INTRODUCTION TO COMPUTERS,
STRUCTURED PROGRAMMING, AND APPLICATIONS

Module
A

Applications and Algorithms in Computer Science

C. WILLIAM GEAR

INTRODUCTION TO COMPUTERS, STRUCTURED PROGRAMMING, AND APPLICATIONS

Module

A

Applications and Algorithms in Computer Science

C. WILLIAM GEAR
University of Illinois
Urbana, Illinois

SRA
SCIENCE RESEARCH ASSOCIATES, INC.
Chicago, Palo Alto, Toronto, Henley-on-Thames, Sydney, Paris, Stuttgart
A Subsidiary of IBM

Compositor	Advanced Typesetting Services
Acquisition Editor	Robert L. Safran
Project Editor	Jay Schauer
Special Editorial Assistance	Stephen B. Chernicoff
Text Design	Judy Olson
Cover Design	Michael Rogondino

LIBRARY OF CONGRESS CATALOGING IN PUBLICATION DATA

Gear, Charles William.
Applications and algorithms in computer science.

(His Introduction to computers, structured programming, and applications)
Includes index.
1. Electronic data processing. 2. Electronic digital computers—Programming. 3. Algorithms. I. Title. II. Series: Gear, Charles William. Introduction to computers, structured programming, and applications.
QA76.G37 001.6′4 77-25546
ISBN 0-574-21189-6

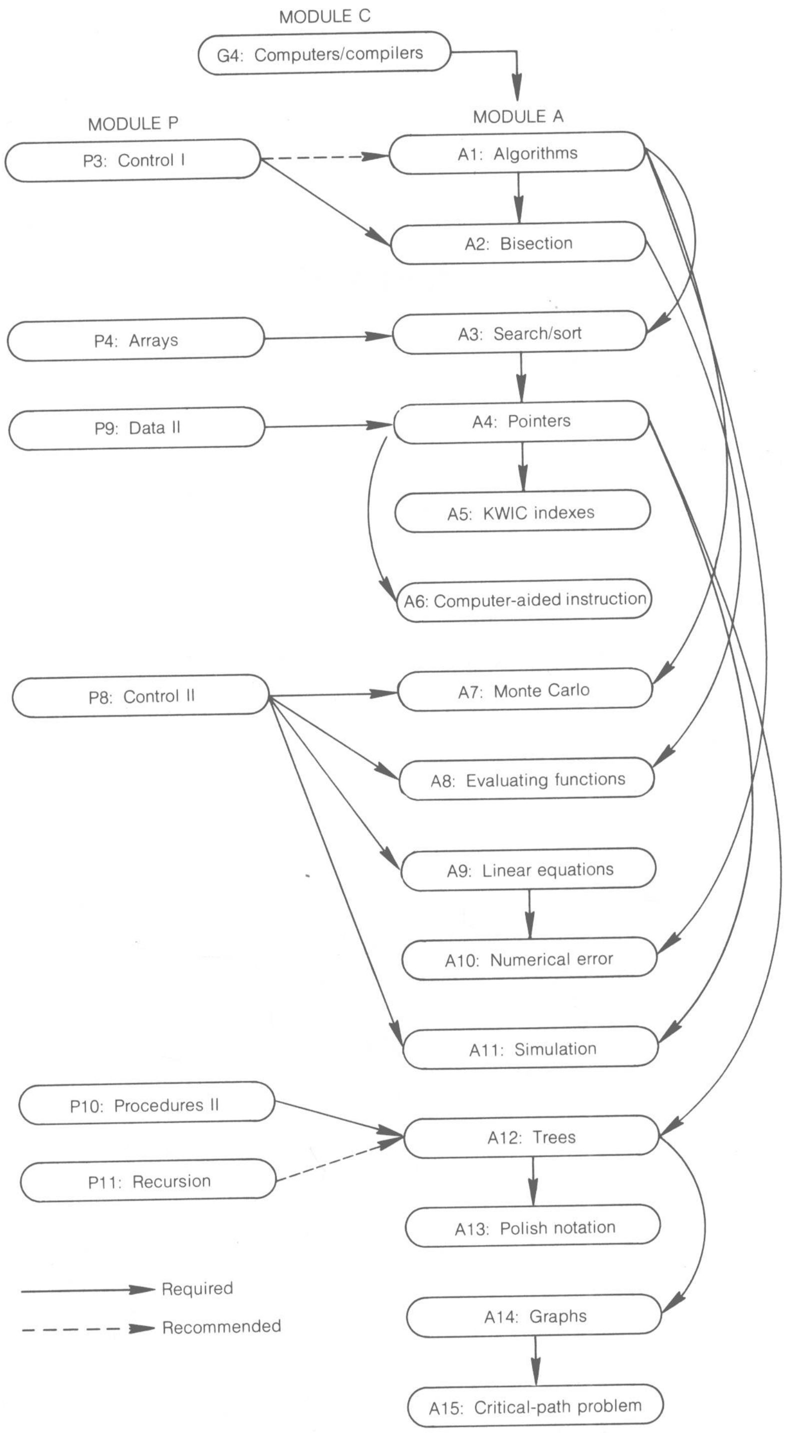
10 9 8 7 6 5 4 3

Preface

This version of Module A (Algorithms and Applications) is intended for use in a computer-science environment. Using the programming principles and informal language developed in Module P (Programming and Languages), a variety of techniques and methods of solution are presented that are useful in wide classes of fundamental numerical and nonnumerical problems. The material is organized so that later chapters depend minimally on earlier ones, to allow the instructor flexibility in the selection and ordering of topics. To help the instructor, a diagram appears at the beginning of this module, showing specific prerequisites for each chapter.

In a one-semester course, Module A can be used to amplify the material in Modules P and C by selecting example applications of interest to the students. For example, Chapters A1, A3, A4, A12, and A13 could be taught in order, skipping the intervening chapters. Alternatively, Modules A, C, and P together provide enough material for a two-semester computer-programming/computer-science sequence. The first semester can cover the first eight chapters of Module A along with most of Modules P and C and a language module; the remainder of Module A can be taught in the second semester, along with a second language if desired. The language modules are tied so closely to Module P that few additional ideas would have to be introduced: essentially it would only be necessary to provide exercises in the syntax of the second language. In a two-semester sequence, Chapters P10 and P11 could also be left for the second semester.

Module A can also be used in a separate course in computer applications for students with adequate background in a nontrivial programming language. The informal language used to describe algorithms is natural for anyone with moderate exposure to a structured language; for students with experience only in Fortran or Basic, a quick review of the General Introduction and a brief discussion of the basic structured language constructs would be needed.



Prerequisite structure for Module A

Contents

Module *A* *Algorithms and Applications in Computer Science*

Before we can write a program, we must design an algorithm to solve the problem at hand. The design of the algorithm depends strongly on the structure of the data. For example, the only way to search for an item in an unordered list is by some form of sequential search, in which each item is examined in turn. However, if the list is ordered—alphabetically, for example—we can apply more efficient procedures. Accordingly, we might consider sorting an unordered list before searching it. We must decide whether the time needed to sort the list is a reasonable trade for the time saved in searching it. If only a few items are to be looked up, it will take more time to sort the list than will be saved on the searches; but if many items are to be looked up, the saving can be considerable. The design of good algorithms must take into account such considerations as whether a change in the data structure may lead to a better method of solution.

Computer applications arise in many diverse areas of activity. In business, computers are often used for *data processing,* manipulating large amounts of data representing company or government files, updating those files, and generating summary reports and records of individual transactions (weekly pay slips, orders, invoices, airline tickets, and so forth); *information retrieval,* which allows managers to examine the status of company or government files stored in the computer and spot potential trouble areas quickly; and *simulation,* the manipulation of numerical models of the real world. Scientific and engineering applications include the approximate solution of numerical problems, correlation and comparison of large amounts of experimental data, and information retrieval applied to textual data.

Many application areas are concerned with common problems and methods of solution. Early chapters in this module will examine some simple problems common to many areas and develop some important methods of solution. Later chapters will then address more specialized problems from particular application areas.

Chapter A1 Types of Algorithm

There is no universal set of rules for designing algorithms: each new problem may need a totally new approach. Indeed, it is this aspect of computer programming that can be the most pleasurable, providing a challenge akin to a crossword puzzle or chess problem and giving an outlet to the ingenuity and creativity of the programmer. There are, however, a number of basic *types* of algorithm that can frequently be used to solve a particular problem.

Five common types of algorithm are given below, followed by examples and a discussion of each:

- *Direct computation*—in which the exact answer is obtained by a sequence of elementary computations.
- *Enumeration*—in which all possible "answers" are tried in order to find one that solves the problem.
- *Divide and conquer*—in which the problem is divided into similar but smaller problems that can either be solved directly or be further subdivided by the same technique.
- *Iteration*—in which a series of increasingly precise approximate answers are computed until one is obtained that is "close enough." (An exact solution would require an infinite number of operations.)
- *Trial and error*—a type of iteration in which each successive approximation is based on the degree of error in the previous approximation.

Direct computation. The income-tax computation of Chapter G2 is an example of direct computation. This form of solution is applicable to simple problems in which the problem description itself specifies the computation needed to solve the problem.

Enumeration. A sequential search is an example of enumeration: each entry in a list is checked to see if it is the one sought. Enumeration is usually very slow, but sometimes it is the only method available. Often it is possible to start with an enumeration method and improve it by avoiding obviously impossible cases, as the following example shows.

Example A1.1 ***Prime Numbers***

Given a positive number N greater than 1, find the smallest integer $M > 1$ that divides N exactly.

If the smallest divisor is N, then N must be prime. An enumerative method for solving this problem is simply to test each integer less than N, starting with 2, to see if it divides N. If one is found, it is the smallest. To program this solution, we need to be able to test whether M divides N. This is a basic operation in some computers and programming languages, but not in others. However, it can be programmed in terms of more elementary operations by testing whether N is equal to $(N \div M) * M$ (see Chapter P2). Our first attempt at this program is shown in Program A1.1. If N is prime, the loop is executed $N - 2$ times (for the values $M = 2, 3, \ldots, N - 1$). A little thought reveals that if N is not prime, one of its divisors must be less than or equal to the square root of N, so there is no need to test any values above that. Program A1.1a gives a revised version. For the case $N = 127$, Program A1.1 executes its loop 125 times, whereas Program A1.1a executes its loop only 10 times. A further improvement is possible by checking only for $M = 2$ and the odd numbers between 3 and the square root of N.

Program A1.1 ***Find a divisor of*** N

```
SMALLEST__DIVISOR: program
    integer M,N
        M←2
        do while M*(N÷M)≠N
            M←M+1
            enddo
        output M
    endprogram SMALLEST__DIVISOR
```

Program A1.1a ***Improved divisor program***

```
SMALLEST_DIVISOR: program
    integer M,N
        M←2
        do while M↑2≤N and M*(N÷M)≠N
            M←M+1
            enddo
        if M↑2>N then M←N endif
        output M
    endprogram SMALLEST_DIVISOR
```

Enumeration methods are the basis of many programs for nonnumerical problems, but because they can be so slow, it is essential to conduct a careful analysis to look for improvements.

Divide and conquer. Breaking a problem into simpler subproblems is a very powerful technique, useful for both numerical and nonnumerical problems. We will illustrate it with a search in an ordered list, such as a telephone book. One way of doing such a search is to open the book in the middle and see whether the item sought is before or after the middle entry. This can be done by a single comparison with the middle entry, because it is known that all items before that entry are alphabetically less and all items after it are alphabetically greater. Thus in one step we have reduced the size of the list to be searched by half. The same technique can now be applied to the smaller list. Thus, if the original list had 16 items in it, the first comparison leaves us with a list of 8 items to consider, the second with a list of 4, the third with a list of 2, and the last with a list of 1. A list of one item can be searched very quickly indeed! This particular search method, called a *binary search*, is a very important technique and is the basis for many related algorithms. We will be discussing it in more detail in Chapter A3.

Iteration. Iteration techniques are usually applicable to numerical problems. An example is the computation of a function such as sin(X). It can be shown that the value of sin(X) is given by the expression

$$\sin(X) = X - X^3/3! + X^5/5! - X^7/7! + \ldots$$

(where 5! means $5 \times 4 \times 3 \times 2 \times 1$, or *factorial* 5). This does not lead to a direct algorithm, because it requires an infinite number of operations. However, for any desired degree of precision, it is sufficient to use only the first part of the infinite sequence. In particular, if we are content with a precision of $\pm 10^{-5}$ for all values of X between -1.0 and $+1.0$, it can be shown that we can use

$$\sin(X) \cong X - X^3/3! + X^5/5! - X^7/7!$$

This computation requires only a finite number of operations, and can now be coded directly. If more precision is needed, additional terms can be added. For example, the next term ($X^9/9!$) should be added if an accuracy of 10^{-7} is required for the same values of X. It can be shown for this example that the desired precision can be achieved by including all terms until a term is generated that is smaller than the error allowed, so a program can be written to *iterate* until the desired accuracy is obtained, as shown in Program A1.2.

Program A1.2 ***Compute sine by iteration***

```
SINE: program
    The sine of a number X is computed using a power series. Terms
    are added until the next term is less than ERROR.
    real SINE,X,ERROR,NEXT_TERM,I
        SINE←X
        I←4.0
        NEXT_TERM←−X↑3 / 6.0
        do while ABS(NEXT_TERM)≥ERROR
            SINE←SINE+NEXT_TERM
            NEXT_TERM←−NEXT_TERM*X↑2 / (I*(I+1.0))
            I←I+2.0
            enddo
        output 'SINE OF',X,'IS',SINE
    endprogram SINE
```

Trial and error. In the trial-and-error form of iteration, the amount by which the current approximation fails to satisfy the problem is used to determine the next approximation. The square-root example in Chapter G3 is of this type. The current approximation X to the square root of 2 is squared and compared to 2. If it is less, the answer is tentatively increased by a small amount to try and get closer.

A trial-and-error process can be likened to the way people perform many everyday actions—driving a car, putting an object down, or almost any action involving movement. A person steers a car in the desired direction by turning the wheel approximately the correct amount and observing whether more or less turn is needed; that is, she observes the error in a trial attempt and then corrects to reduce the error. No matter how well the driver knew the route, a car could not be driven blindfolded, even if there were no other cars on the road, because the measurement of the error is essential to the correction. Chapter A2 gives a trial-and-error method for solving numerical problems common to many scientific and business applications.

Chapter

A2 The Method of Bisection

Suppose you have decided to buy a car costing $3000, and the dealer says you can have it if you will put $500 down and pay $115 a month for 24 months. You will naturally ask yourself whether you can't find less expensive financing somewhere else. One approach is to call a number of banks and find out what interest rate they will charge on a loan for the balance of $2500, to be paid monthly over 24 months. Question: What is the equivalent interest rate of the financing offered by the dealer? (Legally the dealer must tell you, but since the finance charges probably also include a number of other items, such as insurance against the buyer defaulting, the declared interest rate may not be the same as the effective interest rate.) If you could calculate the effective interest rate, you could make a quick decision whether to choose alternate financing from a bank.

If the original cost is COST ($2500 in our example, because the amount to be borrowed is the balance after the down payment), the repayment rate is R per month ($115 in our example), and the interest rate is P percent per year, then the amount left to be repaid after N months is

$$\mathrm{COST} \times \left(1 + \frac{\mathrm{P}}{1200}\right)^{\mathrm{N}} - \mathrm{R} \times \frac{\left(1 + \frac{\mathrm{P}}{1200}\right)^{\mathrm{N}} - 1}{\frac{\mathrm{P}}{1200}}$$

We would like to know what value of P makes this value zero when COST = 2500, R = 115, and N = 24; that is, we want to solve the equation

$$2500 \times \left(1 + \frac{P}{1200}\right)^{24} - 115 \times \frac{\left(1 + \frac{P}{1200}\right)^{24} - 1}{\frac{P}{1200}} = 0$$

for P.

The *method of bisection* is a technique for solving problems of this type. It is, in fact, one of the simplest and most reliable methods for finding the solution to an equation, though not one of the fastest. The solution of equations arises in almost all applications of computers. For example, an engineer who wants to find values of variables to achieve certain objectives—such as selecting the thrust of a rocket to place a spacecraft in the correct orbit—must usually solve an equation.

We can write the equation to be solved for X as

$$F(X) = 0$$

(In the financing problem, X is replaced by P, the interest rate.) Let us suppose that we can determine easily that the function F(X) changes sign as X changes from one value to another. For example, a quick calculation of the remaining balance after 24 months in the example above reveals that the balance is negative if the interest rate is 1% and positive if the interest rate is 50%. Therefore, we know that an interest rate somewhere between 1% and 50% will make the balance exactly zero.

In the method of bisection, we start with a function F(X) and two values of X for which the values of F(X) have opposite signs. This tells us that there is a value of X, between the two values given, for which F(X) is zero. (Mathematically, we are assuming that F(X) is *continuous*, as it is for most reasonable problems.) Suppose, for example, that the function F(X) is given by $X^3 + 3X - 5$. Figure A2.1 shows the graph of this function. As can be seen, the function is below the X-axis at $X = 0$ and above it at $X = 2$. Since the function is continuous, it must cross the

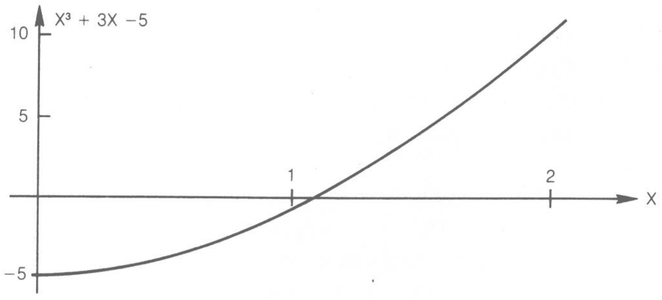


Figure A2.1 $X^2 + 3X - 5$

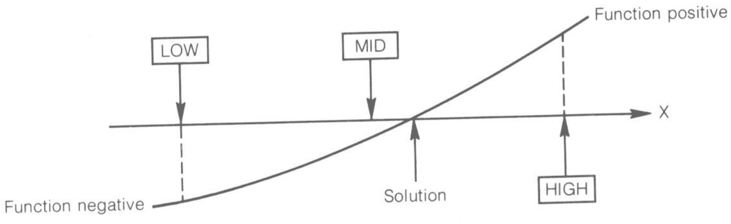


Figure A2.2 Relation of computed points to solution

X-axis somewhere between X = 0 and X = 2. Hence, for some value of X between 0 and 2, $X^3 + 3X - 5 = 0$.

If we set LOW to 0 and HIGH to 2, then there is a solution of the problem between LOW and HIGH, as shown in Figure A2.2. The important characteristic of the two points LOW and HIGH is that the value of the function is negative at LOW and positive at HIGH. Now let us find the point MID midway between LOW and HIGH, as shown in Figure A2.2. If we look at the sign of the function at MID we have two cases:

Case 1.

The function is negative at MID (as shown). Since the function is positive at HIGH, a solution lies between MID and HIGH. Consequently, we can move LOW over to the point MID to get Figure A2.3. (Note that the solution still lies between LOW and HIGH.)

Case 2.

The function is positive at MID. In this case a solution lies between LOW and MID. Consequently we can move HIGH over to the point MID, as shown in Figure A2.4.

In either case, we finish up with the points LOW and HIGH such that the function is still negative at LOW and positive at HIGH. Furthermore, the distance between LOW and HIGH is now half what it was initially.

This process can be repeated by setting MID to the midpoint between the new LOW and HIGH and repeating the analysis, as shown in Figure A2.5.

It can be seen that the points LOW and HIGH are getting closer to each other at each step. Since the solution of the original problem lies between LOW and HIGH after each step, we are getting a better and better approximation to the position of the solution X.

In the computer, we can represent only a finite number of values exactly. In general, the solution X will not be one of these values, so we