common algorithms in Pascal

with programs for reading

David V. Moffat

Common Algorithms in Pascal

with Programs for Reading

DAVID V. MOFFAT

PRENTICE-HALL, INC., Englewood Cliffs, New Jersey 07632

Library of Congress Cataloging in Publication Data

Moffat, David V.
Common algorithms in Pascal with programs for reading.

Bibliography: p. 225
Includes index.
1. PASCAL (Computer program language) 2. Algorithms.
I. Title.
QA76.73.P2M63 1984 001.64'25 82-18623
ISBN 0-13-152637-5

Production supervision by Linda Mihatov
Manufacturing buyer: Gordon Osbourne

TO SUSIE AND MEGHAN

© 1984 by PRENTICE-HALL, INC.,
Englewood Cliffs, New Jersey 07632

All rights reserved. No part of this book may be
reproduced, in any form or by any means,
without permission in writing from the publisher.

Printed in the United States of America

10 9 8 7 6 5 4 3

ISBN 0-13-152637-5

Prentice-Hall International, Inc., *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Prentice-Hall of Southeast Asia Pte. Ltd., *Singapore*
Whitehall Books Limited, *Wellington, New Zealand*

Common Algorithms in Pascal

with Programs for Reading

Prentice-Hall Software Series
Brian W. Kernighan, advisor

此为试读,需要完整PDF请访问: www.ertongbook.com

Preface

PURPOSE

This book is a collection of algorithms that are common to a wide variety of computer programming applications. It gives the reader a vocabulary or repertoire of useful algorithms, bringing together and identifying more algorithms of general utility than any programming text. It is intended to be used as a reference or as a supplement to a programming text. It smooths the transition from introductory programming to the formal study of algorithms.

The book also serves as a source of examples, exercises, and readings. Many exercises ask the reader to think about the algorithms or to write variations. Each part of the book includes one or more completely self-contained "programs for reading" that show practical applications of many of the algorithms. These are followed by more exercises that offer motivation for reading the programs.

WHO CAN USE THIS BOOK

Any person who is learning to program in Pascal will want eventually to know these common algorithms. In particular, students in first and second programming courses should find it a useful supplement, whether it is assigned reading or not; it contains most or all of the algorithms taught in those courses.

Instructors of these or of other courses will find here a good source of blackboard examples, reading exercises, written exercises, and test questions.

Programmers who use other languages, but who wish to learn Pascal, can see how the common algorithms are expressed in Pascal.

ORGANIZATION

The material is presented in an order common to many introductory texts so that it will complement rather than conflict with them. (The introduction to Part I lists the first occurrence of each language feature.) In particular, the book relegates procedures and functions to the latter half because many texts do so. Some very basic material is included for beginning programmers. In each section I have arranged the algorithms so that the easier ones appear first. In some sections it was also possible to show how complex algorithms may be constructed from simpler ones.

Part II contains the only major departure from the "typical" ordering of material: the second half presents simple state-transition techniques (without lookahead) for solving character-processing problems. This subject is too useful to be omitted, but it can also be left for later reading.

Part VIII presents some information-hiding techniques and other special topics.

The algorithms are indexed and cross-indexed.

PROGRAMMING STYLE

The algorithms are written with "header" comments identifying their purposes and with step-by-step comments explaining how they work. The reader is expected to understand the algorithms by reading them together with these comments, rather than by reading a separate prose paragraph. Drawings illustrate the more difficult ones.

The style of layout, indention, and commenting are entirely consistent throughout the book, so that the reader can become familiar with it. The stylistic details were carefully selected to reflect the program structure and to make a clear distinction between the descriptive material and the algorithms.

This is one of the very few books presenting programs that stand by themselves. Both the internal and the more often external documentation are included *within* each example program.

THE LANGUAGE USED

All the programs were compiled and executed by the OS version of the Waterloo Pascal interpreter running under MVS on an IBM 3081 at the Triangle Universities Computation Center, Research Triangle Park, North Carolina.

Every attempt was made to present only standard Pascal constructs as specified by the International Standards Organization Draft Proposal 7185, which supersedes the original *Pascal User Manual and Report* by Jensen and Wirth (see the Bibliography), although the two are quite similar. In particular, all the examples of input and output assume a batch environment. None of the algorithms rely upon the nonstandard aspects of the Waterloo implementation.

ACKNOWLEDGEMENTS

I would like to thank the many reviewers who improved the manuscript. I especially appreciate Lionel Deimel's comments and enthusiasm.

Don Martin, our department head, supported the project from its inception and provided the means for testing the book in our classrooms at North Carolina State University, Raleigh.

Many of the programs for reading were prepared by John Potok. He also provided answers to the exercises.

The text was phototypeset on a Compugraphic EditWriter 7700 at the Printing and Duplicating Department of the University of North Carolina, Chapel Hill. The copy was prepared with the SCRIPT text formatting program, a product of the University of Waterloo, Ontario. The SCRIPT to Compugraphic translation was arranged by the Computation Center of the University of North Carolina, Chapel Hill. The prose is set in Baskerville, the code in OCR-B.

David V.Moffat

Contents

Part I: General Algorithms

INTRODUCTION AND OVERVIEW

An algorithm is a finite sequence of well-defined actions whose result is to accomplish a given task. If they are to be put to any use, whether as subjects of study or as instructions to a computer, algorithms must be expressed in a spoken or written language or notation. Whenever an algorithm is written in a programming language, it bears the imprint of that language. A page-long algorithm in one language, for example, may be a single statement in another—a "sequence" of one action! Nonetheless, an algorithm has some kind of expression in any language.

A large set of algorithms are common to a broad range of computer applications and are expressed in a variety of programming languages. Taken together, they form a kind of language themselves, in which the solutions to many programming problems can be described. This "language" of common algorithms, not the knowledge of a particular programming language, is the key to success in programming. This book expresses that language of algorithms in Pascal.

Very few problems that can be solved with computers are solved merely by stringing together some algorithms; there is more to programming than that. The solution to a problem is described in terms of some central algorithms. These are usually supported by other algorithms, such as to input data and to display results. Many problem solutions also refer to an environment or situation that must be simulated within the program. Finally, the objects and terminology of the problem area and its solution must be defined as constants, types, and variables in the program.

Yet algorithms are still the basic building blocks of programs, and familiarity with the common algorithms is basic to the act and art of programming.

You can use this book in two ways: as a reference or as an introduction to algorithms in conjunction with an introductory text. In either case, it is best to read all of Part I to become familiar with the method of presentation and with some basic terms and algorithms that are echoed throughout the book.

If you use the book as a supplement to a text, then try to read it in the same order as the material in your text. However, if you chance upon some unfamiliar material here, you can be sure that you will eventually want to learn it anyway; there are no "cute" or "one-shot" illustrative examples, nor any esoteric applications. All the algorithms are used again and again in general programming. You can also read each part in its entirety to find out "all about" the algorithms that apply to the given data structure.

If you use the book as a reference, you will find that each algorithm is indexed, cross-indexed, and identified with comments. Each part is as self-contained as possible, given the fact that many algorithms have variants for several data structures. It is important to recognize that, although some algorithms can be copied unchanged into any program, many algorithms serve only as outlines or skeletons that must be filled in with details specific to each program.

The outline that follows shows where each of the various Pascal language features is first introduced in this book. It can be used to determine the background information you may want to know to read each part:

Part I: VAR, CONST, INTEGER, REAL, IF-THEN-ELSE, FOR, WHILE, BEGIN-END, READ, READLN, WRITE, WRITELN, EOF, EOLN, assignment, and simple expressions. The programs for reading also introduce PAGE, CHAR, and CASE in simple contexts.

Part II: Set constants using character subranges, IN, ORD, CHR, BOOLEAN flags, TRUE, FALSE, TYPE for defining enumerated types, nested CASE statements, and one procedure. The example programs use REPEAT-UNTIL.

Part III: General TYPE definitions, integer subranges, ARRAY, boolean expressions, FOR with DOWNTO, and one RECORD in the last section.

Part IV: Matrices, nested FOR loops, RECORD, ARRAY of RECORD, WITH, and one PROCEDURE. The program for reading contains several PROCEDUREs.

Part V: Strings, a PROCEDURE or a FUNCTION for each algorithm, and the INPUT buffer variable in a simple application.

Part VI: FILE, file buffer variables, GET, PUT, REWRITE, RESET, and the full syntax for READ and WRITE.

Part VII: Pointers, NEW and DISPOSE, and NIL.

Finally, try to do the exercises—or at least think about them—as you encounter them; they are designed and strategically placed to reinforce your understanding of the algorithms. Answers and hints for exercises that are marked with "†" are given in the back of the book.

此为试读,需要完整PDF请访问: www.ertongbook.com

SIMPLE COMPARISONS AND EXCHANGES

As you will see throughout this book, comparisons and exchanges of values are important classes of algorithms. In this section we start with the simplest ones.

Assume these declarations:

```
VAR
   A : INTEGER;          (* A, B, and C are any arbitrary  *)
   B : INTEGER;          (*    integers.                   *)
   C : INTEGER;
   SMALL : INTEGER;      (* SMALL and HOLD will be         *)
   HOLD : INTEGER;       (*    explained in the discussion.*)
```

and assume that A, B, and C have been assigned values.

The first algorithm selects one of two values:

```
(* Save the smaller of two values in SMALL:              *)
IF A < B THEN
   SMALL := A
ELSE
   SMALL := B
```

Often, given one small value, we will want to find a still smaller value:

```
(* Save the value of A if it is smaller than SMALL:      *)
IF A < SMALL THEN
   SMALL := A
```

So we could then have:

```
(* Assign to SMALL the smallest of three values:         *)
IF A < B THEN
   SMALL := A
ELSE
   SMALL := B;
IF C < SMALL THEN
   SMALL := C
```

Ex. 1: Tell why this does not make sense:

```
IF A = SMALL THEN
   SMALL := A
```

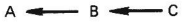
and why this might make sense:

```
IF A <= SMALL THEN
   SMALL := A
```

There is a simple-looking algorithm called a *shift* that has important applications. Assume that A and B have been the variables of interest and that C contains the "next" value of interest:

```
(* Shift the value of C into the A B pair, losing A:      *)
A := B;
B := C
```

The shift would be diagrammed this way:



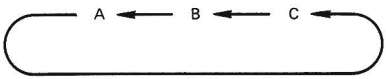
Its applications will be seen shortly.

All the preceding algorithms *substitute* a new value for the current value of a variable. The old value is lost. We often must instead *exchange* the values of two variables:

```
(* Exchange (or "swap") the values of A and B:            *)
HOLD := A;
A := B;
B := HOLD
```

where HOLD is any variable set aside for this purpose.

†*Ex. 2: How many extra variables like HOLD would you need to exchange the values of three variables? Try it this way:*



Exchanges are always done with some purpose in mind. For example:

```
(* Arrange the values of A and B so that A is smaller:    *)
IF A > B THEN
   BEGIN (* exchange *)
   HOLD := A;
   A := B;
   B := HOLD
   END (* exchange *)
```

Ex. 3: Does this algorithm guarantee that A is less than B?

This rearrangement of values is called *ordering* or *sorting*. We can order or sort any number of values:

```
(* Order the values of A, B, and C from smallest (in A)    *)
(* to largest (in C):                                       *)
IF B < A THEN
   BEGIN (* swap A & B *)
   HOLD := A;
   A := B;
   B := HOLD
   END; (* swap *)
(* Now A <= B.                                              *)
IF C < A THEN
   BEGIN (* swap A & C *)
   HOLD := A;
   A := C;
   C := HOLD
   END; (* swap *)
(* Now A <= B and A <= C.                                   *)
IF C < B THEN
   BEGIN (* swap B & C *)
   HOLD := B;
   B := C;
   C := HOLD
   END (* swap *)
(* Now A <= B and B <= C, as desired.                       *)
```

Ex. 4: In general, how many exchanges must you write to order the values of n variables?

As you can see, this technique for ordering values would be very cumbersome for large numbers of values. That problem will be solved later. For now, review these algorithms to see how to find the larger of two values and how to order several values from largest to smallest.

BASIC INPUT AND OUTPUT

Most interesting programs use loops to manipulate large amounts of data. This section shows three kinds of loops for getting (and displaying) a quantity of data.

Assume that we have a constant and variables like these:

```
CONST
   SENTINEL = ...;        (*      (To be explained.)          *)

VAR
   DATUM : INTEGER;       (* One input value.                 *)
   N : INTEGER;           (* Number of values to input.       *)
   I : INTEGER;           (* Loop index.                      *)
```

Assume that the input is a sequence of integers, one integer per line. If the lines of input could be counted beforehand, we could put that number as an *extra* first input number, using it to get the rest:

```
(* Find out how much input to get:                            *)
READ( N );

(* Get and echo the N data values:                            *)
FOR I:=1 TO N DO
   BEGIN (* each value *)
   READLN( DATUM );
   WRITELN( DATUM )
   END (* each value *)
```

†*Ex. 5: What is a major disadvantage of this method?*

Rather than count the data, we could put an extra, but unusual, value as the *last* data item (called a *sentinel* value), then input everything up to that value. Assume that SENTINEL was set to that special value:

```
(* Get and echo data up to the sentinel (SENTINEL) value: *)
READLN( DATUM );
WHILE DATUM <> SENTINEL DO
   BEGIN (* each value *)
   WRITELN( DATUM );
   READLN( DATUM )
   END (* each value *)
```

*Ex. 6: What if all integer values are possible input values (that is, none is unusual enough to be used as a sentinel); can you use "END" or "***" as a sentinel?*

†*Ex. 7: You can use an unusual sequence of two values as a sentinel even if any single value is valid data. Write the algorithm for this, remembering that the first value of the pair is part of the sentinel if and only if the proper second value follows it—otherwise it is data. (Hint: include a "shift" algorithm.)*

The third way to get an input sequence is the easiest and most commonly used:

```
(* Get and echo data values until end-of-file:                *)
WHILE NOT EOF(INPUT) DO
   BEGIN (* each value *)
   READLN( DATUM );
   WRITELN( DATUM )
   END (* each value *)
```

Ex. 8: In which of the three input loop control techniques can READ be substituted for READLN?

†*Ex. 9: Rewrite the EOF loop assuming that any number of values can appear on one line.*

There are many ways to vary and to combine these three input techniques. For example, the input sequence might be divided into smaller groups by one sentinel value (say, 0), with the whole sequence ended by another (maybe -1), like this:

```
 1  2  1  0
 4  0
 3  1  4  2  0
-1
```

Let GROUPEND be 0, and let SENTINEL be -1. (Actual data will be any positive integers.) The data might be processed this way:

```
(* Get and echo groups of data up to the final sentinel:      *)
READ( DATUM );
WHILE DATUM <> SENTINEL DO
   BEGIN (* each group *)

   (* Get and echo group values until the end of group:       *)
   WHILE DATUM <> GROUPEND DO
      BEGIN (* each value *)
      WRITE( DATUM );
      READ( DATUM )
      END; (* each value *)
   WRITELN;
   READ( DATUM )
   END (* each group *)
```

Ex. 10: Does this algorithm handle empty groups or a lack of groups properly?

†*Ex. 11: Rewrite the algorithm to use the EOF function instead of using the -1 sentinel value. (Take care about the ends of lines.)*

BASIC OPERATIONS ON DATA

In this section we explore some of the general kinds of operations that are commonly performed on sequences of input data.

Assume that we have these declarations:

```
VAR
   DATUM : INTEGER;      (* One input value.              *)
   N : INTEGER;          (* Number of input values.       *)
   SUM : INTEGER;        (* Sum of the input values.      *)
   SMALL : INTEGER;      (* Smallest input value.         *)
```

Assume also that the data appear one value per line. EOF loops will be used throughout this section to get the data.

The list of values would usually be echoed in a single column. Sometimes, however, the single input sequence is divided between two output columns, using some criterion to decide the column in which each value will appear. Here negative versus nonnegative will be used as an example criterion to select a column:

```
(* Echo input data to selected output columns:            *)

(* Print column headings:                                 *)
WRITELN( 'Negative':10, '0 or Positive':15 );

(* Get and echo data until end-of-file, putting the       *)
(* negative and nonnegative values into separate columns: *)
WHILE NOT EOF(INPUT) DO
   BEGIN (* each value *)
   READLN( DATUM );
   (* Select appropriate output column, then echo:       *)
   IF DATUM < 0 THEN
      WRITELN( DATUM:10 )
   ELSE
      WRITELN( ' ':10, DATUM:15 )
   END (* each value *)
```

†*Ex. 12: Write an algorithm to select one of three columns, given an appropriate selection criterion.*

此为试读,需要完整PDF请访问: www.ertongbook.com