Graded Problems in Computer Science

Andrew D. McGettrick

Peter D. Smith

Graded Problems in Computer Science

Andrew D. McGettrick

University of Strathclyde
Glasgow, Scotland

Peter D. Smith

California State University
Northridge, California, U.S.A.

ADDISON-WESLEY PUBLISHING COMPANY

London · Reading, Massachusetts · Menlo Park, California · Amsterdam
Don Mills, Ontario · Manila · Singapore · Sydney · Tokyo

To Peter James and
Peter, Bobby, Kate and Andrew

Set by the author in elite-12 using NROFF, the UNIX text-processing system, at the University of Strathclyde, Glasgow.

Cover design by Alan Rudge.
Printed in Finland by Werner Söderström Osakeyhtiö, Member of Finnprint.

British Library Cataloguing in Publication Data
Graded problems in computer science
McGettrick, Andrew D.
1. Electronic data processing – Problems, exercises, etc.
I. Title II. Smith, P.D.
001.6'4 QA76

ISBN 0-201-13787-9

Library of Congress Cataloging in Publication Data
McGettrick, Andrew D., 1944–
Graded problems in computer science.

Bibliography: p.
Includes index.
1. Electronic digital computers–Programming–Problems, exercises, etc. I. Smith, P.D. (Peter DeCost), 1941– . II. Title.
QA76.6.M3987 1983 001.64'076 82-11671
ISBN 0-201-13787-9

ABCDEF 89876543

Preface

The proper teaching of programming is a topic which is of concern to all who are involved with computing. Yet both introductory and advanced programming courses tend to suffer from certain rather severe deficiences.

It frequently happens that the teaching of a programming language and the teaching of programming itself are so intertwined that they become confused. The reason for this is, of course, perfectly understandable: to write programs that have to be executed it is necessary to use some programming language. Another criticism that can be levelled at many courses and indeed many texts is that they are designed in such a way that they incorporate clever tricks which show the abilities of the teacher to good effect but which can be confusing and offputting to the learner.

Our motivation for writing this book was to produce a set of carefully graded programming exercises which gradually increase in difficulty and complexity. Within each chapter the earlier exercises should be of such a standard that everyone should be able to solve them. It is hoped that the later exercises will tax even the better students. The problems and the treatment are suitable for all those involved in their first two years of computing. We would like to think that the text would naturally accompany any first and many second courses on programming. Only such knowledge as is commonly assumed in first year college or university courses is assumed in this book.

This book is basically about programming. We have tried to make the material independent of any particular programming language. Each chapter and each section contain explanations about how to write programs but only informal descriptions of programs are provided. Throughout we have tried to adhere to (what would normally be regarded as) the elements of good programming practice. Since the difficulty, the complexity and the size of programs increase as more programming language constructs emerge, it becomes desirable to introduce exercises which might be viewed

as student projects. Yet it is sensible to do this only when programmers have at their disposal the means of properly structuring their programs. Although they are not identified as such, exercises which could be regarded as projects start to appear after the introduction of subprograms such as procedures and functions.

In preparing the material for this book one important matter had to be given careful consideration. People may learn to program either in a batch environment or in an interactive environment; the latter is increasing in popularity with the introduction of relatively cheap microprocessors. Should interactive programs be included? Should the examples we provide be suitable for both environments? To cover as wide a spectrum of interest as possible we introduced a special chapter on interaction (which can be omitted if the environment so dictates); the examples in the remaining chapters can be run in both interactive and batch environments.

Unfortunately it has been necessary to limit the number of topics and the number of areas of programming that could be covered in the book. There is little or no mention of problems involving interrupts, parallel processing, graphics, and so on. Even the more interesting aspects of compiling techniques, operating systems and general systems software have had to be omitted. We hope that the few examples that do exist whet the appetite of the interested reader.

Finally we are aware of the duty to record our debt of gratitude to those who have helped in the preparation of the book. This is extremely difficult since similar questions and problems have occurred in several places; the origins of most problems are by no means clear. Therefore lest we unwittingly cause offence, we shall refrain from mentioning any names. However, special mention should be made of the help provided by Dr. Robin Hunter and Mr.Ray Welland and other colleagues at the University of Strathclyde in Glasgow and California State University at Northridge.

October 1982 A.D.McG. and P.D.S.

Contents

1 | Straight line programs

This book is about programming. Its aim is to provide a set of programming problems which cover a wide spectrum of interest and are carefully graded in difficulty. Within each section the earlier problems should be within the grasp of everyone. The more difficult problems should challenge even the more gifted.

It is intended that this text should be used in conjunction with a book or course on some programming language. It is not intended that all the resulting programs should be entered into a computer and executed. Indeed such a practice should be discouraged. Certainly it is necessary to try to execute some of the programs but only a relatively small proportion, perhaps 5 to 10 per cent. The aim is to provide practice in thinking, not typing!

1.1 The nature of programming languages

To focus attention on programming as opposed to programming languages we have decided to make the contents of this book as language independent as possible. Yet it is important to comment on the sorts of programming language facilities we assume at each stage. Fortunately there is a set of commonly used languages which includes (in alphabetical order) Ada, Algol 60, Algol W, Algol 68, Basic, Cobol, Fortran 66, Fortran 77, Pascal and PL/I. These tend to have similar facilities for simple programming and so it is possible to make some progress.

Programming languages will typically have statements, instructions or facilities whereby it is possible to

*introduce identifiers which can be used to denote variables and possibly constants of a particular kind (and perhaps within some range)

*remember or store within variables certain values or the results of calculations

*perform calculations such as the evaluation of arithmetic or logical expressions

*read information from input files or a reader and send information to output files, a visual display unit or a printer

The more common programming languages all possess such facilities.

Typically simple variables and, if they are present, constants can only be of certain specified types such as integers, reals, characters, Booleans and possibly enumeration types. Where appropriate, declarations are usually used to introduce variables, constants, etc. and associate with them a type (and perhaps range).

The kind of arithmetic calculation commonly allowed includes the addition, subtraction, multiplication and division of integers and reals. There is usually also some means of finding the quotient and remainder when two integers are divided and a means of raising a number to a power. Apart from these there are usually standard functions or equivalent facilities for finding square roots (of non-negative numbers), absolute values and signs, and for applying trigonometric functions, logarithmic functions, exponential functions and so on.

The input of information is generally controlled by READ, INPUT or GET statements and the output by WRITE, OUTPUT or PUT statements or the equivalent.

1.2 Abstraction

When a programmer has to write a program for even a very simple task there are usually various decisions he will have to take. He will be given a problem phrased in terms of familiar concepts such as mortgage rates, account numbers, salaries, number of hours worked, names, addresses, dates of birth, examination marks, train times and so on. One of the important tasks a programmer has is to represent each such item by a

corresponding object in a program in the programming language. This object will typically be represented by the identifier of a variable of a particular type. How should the programmer select the identifier and type?

In choosing identifiers a programmer should have regard to the way in which the corresponding variable is to be used in the program. The identifier should be easy to remember and it should remind the programmer of the role it is to play and the item it represents in the eventual program. Long identifiers are more descriptive but are cumbersome and adversely affect the speed of thought and the speed at which the program can be written - a consideration which tends to encourage the most important property of correctness. On the other hand, short identifiers are easy to manipulate but are less descriptive. Usually some compromise is possible. At any point in a program the number of concepts under consideration should be severely limited in the interests of simplicity - the number of concepts is usually closely related to the number of variables. There is therefore usually little need for long identifiers. Note that the rules of the programming language itself, in the case of for example BASIC or FORTRAN, may force the programmer to adopt certain conventions.

In deciding on the precise representation of some item within a program a programmer must be able to focus attention on the relevant aspects of that item. He must be able to abstract the relevant qualities. In some cases this is straightforward. If for example the problem specification refers to measurements of some kind then it is natural to expect that these might be represented as real numbers held in real variables or perhaps as integers held in integer variables. Consider however a bank account number. Should this be treated as an integer or as a string of characters? The answer comes from observing the kinds of operation which the programming language will permit on integers and strings of characters. Typically

*integers can be added, subtracted, multiplied, divided and so on, as well as read, printed and compared

*strings of characters can be read and printed and

examined in limited kinds of ways, usually comparison of two strings is easy.

Other characteristics which might be important are the amount of space taken by a representation (integers are normally more compact), and any size limitations (integers are typically less than about ten decimal digits). It is not usually desirable or sensible to add or multiply account numbers. For this reason it is usually most appropriate to represent an account number as a string of characters. Note that the decision has been based both on the type, on properties such as range limitations and storage requirements and on the operations which can be performed on that type. Illegal operations (for example attempts to multiply account numbers held as character strings) will normally be detected and a message output to the programmer - no attempt will be made to perform them.

Note the crucial role played by types. Illegal operations can be detected and highlighted by a compiler so that the programmer can correct them. A somewhat similar role is played by ranges when they appear. But generally errors caused by range violation appear when a program is actually being executed. A programmer must strongly resist the temptation to alter ranges merely to accommodate particular values. Usually these errors are a symptom of some more fundamental error perhaps caused by a misunderstanding of some kind; this more fundamental error is what should be remedied.

1.3 Simple programs

For most simple programs of the kind we shall encounter in this chapter the structure of the program is straightforward:

*introduce identifiers, initialise, etc.
*read in information
*perform calculations
*print out results

Programs such as these are called straight line

programs; execution proceeds from the start and progresses step by step to the end of the program. No concept of branching, repetition or looping is present.

Example 1.3.1 Circumference and area of a circle

Given as data the radius of a circle, write a program which prints out the circumference and area of that circle. The program might take the following form

```
introduce the constant PI = 3.14159265
introduce identifiers R, CIRCUM, AREA
read value and store in R
evaluate  2 x PI x R and store the result in CIRCUM
evaluate PI x R^2 and store the result in AREA
print CIRCUM and AREA
```

The notation we have used here is informal but has the merit of being independent of any particular programming language. The task of translating the above into a particular programming language ought to be completely straightforward.

In the example above we have used capital letters for the identifiers of variables and will continue to do so but from now on we will omit references to the introduction of identifiers; the appearance of an identifier in an example should convey sufficient information.

Some advice should be given about the nature of input and output. Let us concentrate for the moment on output. Really the programmer ought to ask: for whom or for what purpose is the program being written? There are two situations worth mentioning.

If output is for human consumption and for reading then the output should be self explanatory and should usually contain, in some form, information about the input data it processed. This usually means that there should be text describing or explaining the significance of the various pieces of information in the output. This is the usual position with elementary programming and our examples will tend to reflect this view.

Sometimes programs provide output which has merely to be read and absorbed by another program. In this case explanatory text is often unnecessary and even wasteful.

All that is necessary is a set of results in stark form. However, we do stress that this is not the usual situation in real life or in this book.

1.4 Specification and testing

Associated with every problem there is usually an informal description of that problem. From this the programmer must extract a more formal program specification - an exact interpretation of what his program must accomplish. It is vital that such descriptions should be clear, unambiguous and accurate. Further it should be free of any mention of the particular variables, constants, etc. in the program. The format of the data should be clearly defined as should the nature of the resulting output. In particular the range of values with which the program deals should be clearly defined. In producing specifications of programs a programmer should pay particular attention to the use of words such as positive, negative, non-negative, and so on; he should be careful about including units of measurement where necessary; where numeric data should be supplied to a particular accuracy, this should be specified; and so on.

There is a difficulty associated with specifications and this arises from the nature of high-level programming languages. Programs written in Ada, Fortran, Pascal, etc. are intended to be portable in the sense that a program which runs on one machine should run with little or no extra effort on another machine of a different type. Now the range of numbers that can be handled may vary from machine to machine; consequently the specification details may also vary.

To overcome the need to give different specifications for different machines it is customary to write specifications in a way that is independent of particular machine limitations. A reader should then understand that if a program is run, and there are no storage violations or whatever, the result will be as predicted in the specification.

We illustrate a typical specification by describing

the form of the input and the nature of the output expected in Example 1.3.1.

input: one non-negative number which may be presented either as an integer or as a real number possibly preceded by a + sign; the number will be assumed to represent units of some kind.

output: two signed real numbers separated by a space. The input will be interpreted as the radius of a circle; the first number produced represents the circumference of the circle (in the assumed units) and the second represents its area (in square units). Both results are accurate to 8 decimal places.

Note that the statement of what output is produced assumes that the input is in the expected form. Should the input violate these rules, the specification says nothing whatsoever about the behaviour of the program. Basically the above can be understood by a client who can immediately determine how to use the program and can learn its effect; running the program should cause him no surprise.

We have seen that the program specification is extracted from the initial problem specification. From this it should also be clear that the final program should work for certain sets of input. Indeed sets of data to test the eventual program can be derived at the same time as the formal program specification is derived - before the program is even written. The testing stage involves running the program with certain kinds of data in an attempt to discover whether the program does what was intended. The choosing of test data is not always straightforward. For the kinds of program discussed in this chapter, tests should include typical sets of data but also peculiar or unlikely situations. In particular boundary conditions should always be tested, e.g. if an integer represents an age (at most 100) then tests should include typical ages that include both the ages 0 and 100.

It is important to be aware of the fact that there are severe limitations to the process of program testing. Tests are not a guarantee that a program is accurate;

they indicate the presence of errors, not their absence. Another activity, program proving or program verification, will demonstrate the latter. A study of this topic is beyond the scope of this book although many of its lessons are incorporated in a great deal of the material that is provided.

1.5 Miscellaneous exercises

In the exercises below the programmer should initially supply detailed program specifications for all the resulting programs. Appropriate testing should also be undertaken.

1 Code the solution of example 1.3.1 in a programming language of your choice.

2 Design and run a program to convert a measurement in metres and centimetres into centimetres.

3 Write a program to read three numbers and output them in reverse order.

4 Write a program which inputs a temperature in degrees Centigrade and outputs the corresponding temperature on the Fahrenheit scale.

5 Write a program which calculates the value of sin(x) where x is expressed in degrees. The data is just the value of x.

6 A rational number is usually written in the form P/Q where P and Q are integers. Write a program which reads a rational number in the form of the pair of integers corresponding to P and Q and outputs the equivalent real number.

7 At the beginning of a journey the reading on a car's odometer is S kilometres and the fuel tank is full. After the journey the reading is F kilometres and it

takes L litres to fill the tank.

Write a program which reads values of S,F, and L and outputs the rate of fuel consumption rounded to the nearest integer followed by the actual rate correct to four decimal places.

8 Write a program which reads a positive integer N and outputs the sum of the first N integers.

9 Write a program which reads a positive integer N and outputs the sum of the first N squares, that is 1+4+.....+NxN.

10 Show how to convert automatically from centimetres into metres and centimetres by writing a suitable program.

11 In the "old days" in Britain measurements used to be expressed in yards, feet and inches (12 inches = 1 foot, 3 feet = 1 yard). Show how to convert from inches into yards, feet and inches.

12 A local council levies rates on a house as follows. The total liability is made up of a water tax and a dwelling house tax. These are computed by multiplying the surveyed rateable value of the property (R) by the water rate (W) and the house rate (H) respectively. Householders are given a choice of two methods of payment:

(i) 10 monthly payments
(ii) 2 six-monthly payments

In the latter case, a discount of 5% is given.

Write a program which reads R, W and H and outputs an annotated Rates Notice.

13 If an amount of money A earns R% interest over a period of P years then at the end of that time the sum will be

$$T = A \times ((100 + R) / 100)^{P}$$

Write a program which inputs A,R and P and outputs T.

14 An object falls to the ground from a height h in time t given by

$$t = (2h/g)^{0.5}$$

where g is the gravitational constant (= 9.81 metres/sec^2).

(a) Write a program which computes the time it would take an apple to hit Newton on the head assuming he was 1.37 metres high (when sitting) and the apple was on a branch 6.7 metres high.

(b) At the time of writing, a typical computer takes a microsecond to obey an instruction. Write a program which outputs the number of instructions that it can obey during the time it takes for an egg to drop to the floor from a table 1 metre high.

15 In order to pay off in N years a mortgage of $P on which interest is charged at an annual rate of R% and computed annually, $A must be repaid every year where

$$A = \frac{P \times (1 + r)^N \times r}{(1 + r)^N - 1}$$

and r = R/100. Write a program which reads P,N and R and outputs A.

16 Write a program which inputs 3 quantities T,N,R and outputs the monthly repayments on a loan of $T over a period of N years at a fixed interest rate of R% when the interest is computed only once (at the beginning of the loan period).

17 A room is B metres wide, L metres long and H metres high. It has a door (B1 metres wide and H1 metres high) in one wall and a window (B2 metres wide and H2 metres high) in another. Wallpaper is available in rolls M metres long and F metres wide. Write a program which reads values for B, L, H, B1, H1, B2, H2, M and F and calculates how many rolls of wallpaper would be needed to paper the walls assuming no waste.