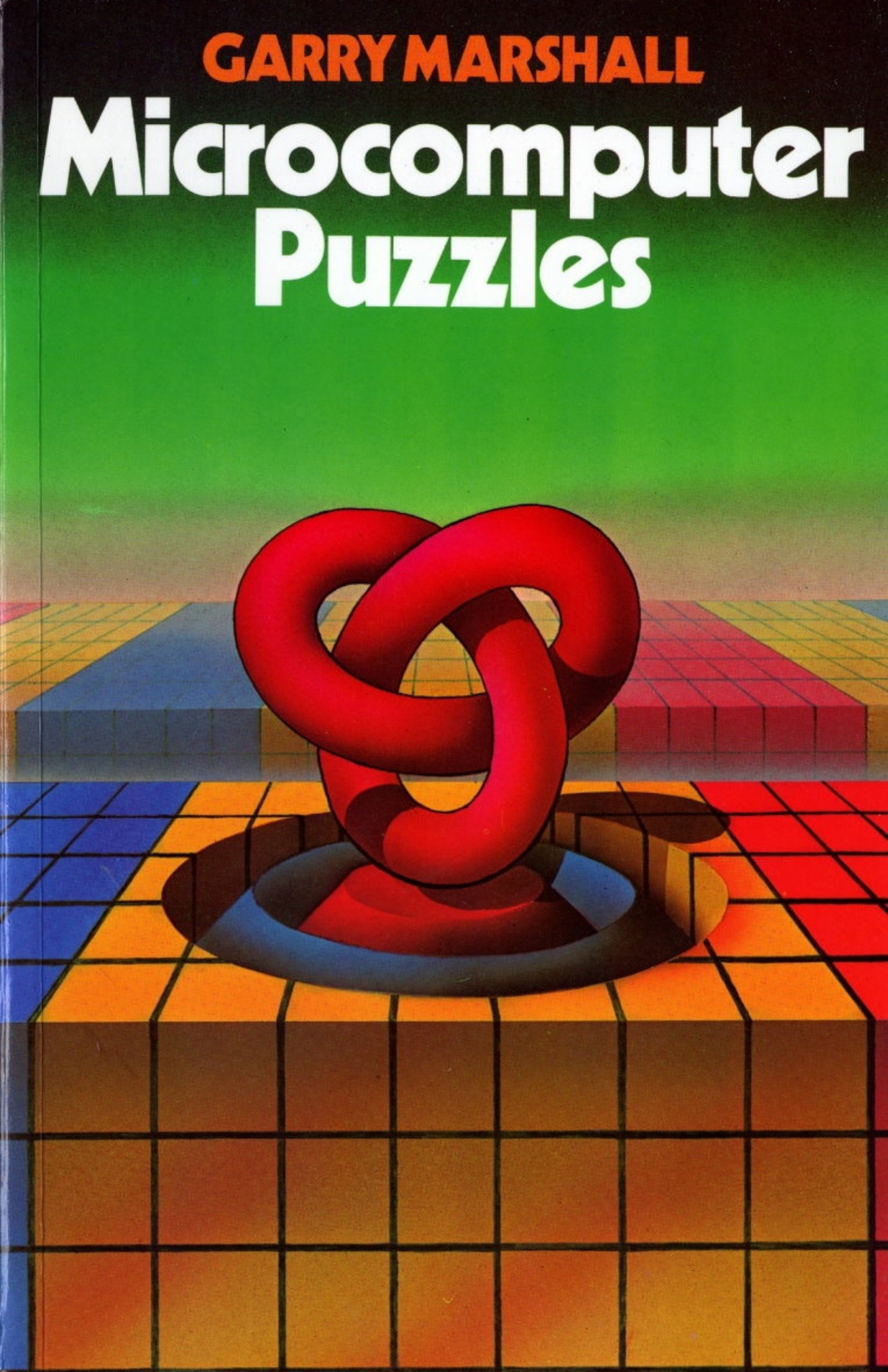
GARRY MARSHALL

Microcomputer Puzzles



Microcomputer Puzzles

Other books from Collins

Computer Languages and their Uses
Garry Marshall
0 246 12022 3

Inside Your Computer
Ian Sinclair
0 246 12235 8

Simple Interfacing Projects
Owen Bishop
0 246 12026 6

Programming with Graphics
Garry Marshall
0 246 12021 5

Get More from the Epson Printer
Susan Curran
0 00 383001 2

Microcomputer Puzzles

Garry Marshall

COLLINS
8 Grafton Street, London W1

Collins Professional and Technical Books
William Collins Sons & Co. Ltd
8 Grafton Street, London W1X 3LA

First published in Great Britain by
Collins Professional and Technical Books 1985

Distributed in the United States of America
by Sheridan House, Inc.

Copyright © Garry Marshall 1985

British Library Cataloguing in Publication Data
Marshall, Garry
Microcomputer puzzles.
1. Puzzles—Data processing 2. Microcomputers
I. Title
793.7'4 GV1493

ISBN 0-00-383125-6

Typeset by V & M Graphics Ltd, Aylesbury, Bucks
Printed and bound in Great Britain by
Mackays of Chatham, Kent

All rights reserved. No part of this publication may
be reproduced, stored in a retrieval system or transmitted,
in any form, or by any means, electronic, mechanical, photocopying,
recording or otherwise, without the prior permission of the
publishers.

Contents

Preface		vi
1	Word Puzzles	1
2	Number Puzzles	27
3	Graphics Puzzles	52
4	Puzzles Involving Programming Techniques	79
5	Puzzles Involving Computer Applications	102
Appendix: Further Reading		124

Preface

The puzzles that are collected in this book first appeared in *The Observer Magazine* in the first half of 1984. In fact, the first appeared on January 8 1984 and the others on the following 25 Sundays. They appeared under the title 'Computers' and shared the games page with the Azed crossword, a bridge column by Terence Reese and a chess column by Michael Stean. It is very flattering to find oneself in such exalted company, and I should like to record my appreciation and thanks to the games page's editor, Anthony Ellis, not only for asking me to do the column but also for treating me with consideration as one among equals at all times.

In appearing alongside 'the world's hardest crossword', and bridge and chess columns by experts of international renown, 'Computers' had to hold its own in terms of both content and standard. For this reason, the puzzles are by intention rather hard. But a solution for each was presented the following week, and these solutions are given here at the end of each chapter.

The puzzles are presented in this book essentially in the form in which they were submitted to *The Observer*. The pressures of combining at least four contributions each week to create a single games page often meant that the 'Computers' contributions had to be cut down from their submitted form and, very occasionally, they were cut rather more than I would have liked. Here, I have taken the opportunity not only to restore the puzzles and their solutions to their original forms but also to expand them.

Each puzzle was intended to illuminate some aspect of computers and computing and, to judge by the correspondence, this aim succeeded. On this basis, it seemed worthwhile to expand on the general area from which each puzzle was drawn, as well as on the puzzle itself. The extra material that I have added enlarges and sometimes clarifies the puzzles, fills in their backgrounds, and expands on the techniques involved in their solutions, the application

area from which they are drawn, and other matters of interest.

I have classified the puzzles into five rather arbitrary groups called Words, Numbers, Graphics, Applications and Techniques. This helps in presenting the background material as several of the puzzles share a common aspect of some kind.

Finally, I should like to emphasize that although I have presented a solution to each puzzle (and occasionally more than one), it is only one of many possible solutions. There is no such thing as *the* solution to a computer problem in the form of a unique computer program. So, in presenting my solutions, in no way do I imply that if your solution is different then it must be wrong. There are almost certainly other approaches to solving the puzzles. There are almost certainly better programs for solving them. I hope that you will write them.

Garry Marshall

Chapter One
Word Puzzles

Most people think of the prime task of computers as dealing with numbers. The term 'number crunching' has passed into the language. But one of the first insights that a familiarity with computers brings is that computers manipulate symbols. And although these symbols *can* represent numbers, they can represent other items quite as well. They can represent letters so that computers can handle text; they can represent graphics so that pictures can be produced.

One of the advantages of programming in BASIC is that it allows letters and, indeed, characters of any kind, to be manipulated quite as easily as numbers. For this reason, we begin with a section on word puzzles in which words and letters need to be handled in various ways. Answers to the puzzles are presented as BASIC programs.

The puzzles in this chapter naturally draw quite heavily on the capabilities of BASIC for handling character strings. Although BASIC is a much criticised language, the facilities it provides for this are superior to those of most other computer languages. The solutions to the puzzles make use of such things as arrays of string variables, and the functions for manipulating strings that extract a part of one and combine two or more into one.

It is worth making the point that there is a fundamental difference between computer puzzles and ordinary puzzles. With an ordinary puzzle, the objective is to find the answer. But with a computer puzzle the idea is to find a way of telling a computer to find the answer. In other words, when solving a computer puzzle we are not concentrating on the answer itself but on developing a program that will direct the computer to find the answer. One of my correspondents revealed a total failure to grasp this point in writing to say that he had written a correct program to solve a particular puzzle but that it did not give an answer, and could I let him know what the answer was. (It would also appear that his definition of a 'correct program' was fairly loose.)

1.1

WORDS

The first puzzle is to develop a program to write the instructions for creating one word from another. Its solution is a program that writes another program and, as such, is an example of a *program generator*.

Program generators will be familiar, even if not under this name. A BASIC interpreter provides an instance of one, for when it is given a BASIC program it writes a corresponding machine code program. In the same way, any high-level language translator can be seen as a program generator. But a program generator really comes into its own when it allows such things as accounting procedures and stock control methods to be specified in the kind of terminology that accountants and stock control experts use, and then produces a program in a computer language. This is a way of making computers much more readily useful, for it means that there is no need to learn a great deal about computers before being able to make use of one. The accountant, for example, can use his professional expertise to drive the computer, and need know nothing of computer programming or of how to express the procedures of accountancy in a computer language.

A program generator of this type is necessarily applicable in a restricted range of applications because it deals in the operations that are characteristic of a specific activity. By contrast, a general programming language is much more widely applicable, but requires a good deal more expertise from its users.

It is not necessary to write anything like a full-blown program generator to solve the puzzle below, but a solution does represent a first step towards producing one.

The puzzle

The sort of puzzle in which the solver must try to find all the words that can be formed from the letters of a given word is ideal for computer solution. If we are given the word PLEASED for a puzzle of this kind, a systematic approach is needed to guarantee that the

words PLEA, PEDAL, SLEEP, and so on are all found. If we have a program that can print out all the possible letter combinations of every length from one to that of the original word (seven in this case), we can select those that are proper words and reject the others. Better still, we could feed the letter strings to a dictionary program intended for checking the spelling in documents produced by a word processor and let *it* select the real words.

Now, when the word PLEASED is stored in the computer in a variable named WORD$, the word PLEAD can be formed from its letters by the command:

```
PRINT MID$(WORD$,1,4) + MID$(WORD$,7,1)
```

In this puzzle, however, we reverse the situation. We have a BASIC program that begins by storing one word in WORD$ and another, which can be formed by using the letters of the first word, in SUBWORD$. The problem is to complete this program in such a way that it automatically generates a command of the kind just given for producing the second word from the first.

To illustrate what is required, if the program is given the word BASIC to store under WORD$ and AS to store under SUBWORD$, so that it is to produce the command necessary to extract AS from BASIC, an acceptable output would be:

```
PRINT MID$(WORD$,2,2)
```

But this is not the only acceptable output. The following command would do just as well:

```
PRINT MID$(WORD$,2,1) + MID$(WORD$,3,1)
```

As a second illustration, if the program is given the word MATCHSTICK and then CHAT as the word to be extracted from it, then we might expect the output:

```
PRINT MID$(WORD$,4,2) + MID$(WORD$,2,2)
```

Again, there are other possibilities.

It may be worth mentioning that if you can arrange for your program to position the cursor at the end of the output it produces then, on most types of micro, it will only be necessary to press RETURN after the program has halted to make the computer obey the command produced by the program and so to determine if it is correct.

1.2

APPROXIMATIONS TO ENGLISH

The idea of generating approximations to English, and to other natural languages, originates in the study of the information content of language. Starting with the individual letters used in English, we can produce a rough approximation to English text by generating these letters in succession and at random. The result of such a process could be:

MXAFL EPSMDFDHRYFB BEVXT SBDHEYQ

This is clearly not a very good approximation to the English with which we are familiar; but by amending the generation process a series of approximations can be produced, each of which is an improvement on its predecessors.

The first improvement consists of producing the individual letters according to the probabilities with which they occur in a language rather than completely at random. The letter probabilities can be obtained by taking a book, counting the number of letters in it, and then calculating the proportion of them that are As, Bs, and so on. Interestingly, different languages sharing the same alphabet – such as English, French and Spanish – have significantly different sets of letter probabilities. Simply changing the letter probabilities will give text that is rather similar to that in the previous example except that the vowels, for instance, will occur a good deal more often than will Z, X and Q.

The model still does not give a particularly good approximation to English, and this is because letters do not occur independently in real languages. Consider the question of which letter comes next when we have a word starting with the letters 'st'. It can be r, y or a vowel, but not much else. So of the 26 candidates only seven are possible next letters. If a word starts with 'strin', then the next letter can be predicted with a high degree of certainty as a g. (My dictionary lists the Scottish word *strinkle*, but I don't think that would be in many people's vocabularies.) The most convincing demonstration is to consider which letter follows a 'q'.

An example of the text generated by this method is:

ON IE ANTSOUTINYS ARE T INCTOREST BE S DEAMY
ACHIN D ILONASIVE

There are (short) words here that actually are English words, as well as some interesting candidates that almost could be. (This example is taken from the reference listed in the appendix.)

After this further approximations can be generated by taking letters in pairs and using the probabilities with which the pairs occur in English, then by taking them in threes, and so on. Even better approximations can be obtained by using words rather than letters, and then word pairs, word triplets, and the rest. And after that the basic unit taken can be the sentence.

But this has brought us rather beyond the ambitions of our puzzle, which only deals with letters and letter pairs. Also, in the interests of making the solution programs short, the puzzle deals only with a group of six letters rather than the complete alphabet. It is possible to extend these programs so that they deal with the entire alphabet, and also to generalise them along the lines suggested by this discussion so that they will generate ever-improving approximations to English. The reference given in the appendix will enable you to locate the necessary letter and word probabilities.

The puzzle

The following program generates a very crude approximation to English by printing the letters of the alphabet, all with equal probability, or a space, with a higher probability. In fact, the letters each occur with probability 1/30 and the space with probability 4/30, so that the 'words' are approximately seven letters long on average. The program is:

```
10 X=INT(65+30*RND)
20 IF X>90 THEN X=32
30 PRINT CHR$(X);
40 GOTO 10
```

As you will see if you run the program, the output is a rather poor approximation to English. We shall consider two ways of improving it. The first is to make the letters appear with probabilities corresponding to the frequencies with which they occur in English usage rather than with equal probabilities. The second is to print the

current letter using probabilities that depend on the letter that has just been generated.

To keep the solution programs within limits, the puzzle is to produce approximations to English that are better than those of the program given above, using the six letters A, D, E, L, P, S and a space rather than the whole alphabet. First generate an approximation by printing repeatedly one of these characters with the probabilities given in Fig. 1.1(a). Then generate an improved approximation by repeatedly printing the six characters using probabilities that depend on the last character that was printed, as given by the table in Fig. 1.1(b). Make the initial state of the program that which it assumes when a space has just been printed.

Character	A	D	E	L	P	S	space
Probability	0.2	0.1	0.2	0.1	0.1	0.1	0.2

Last character	[Current character] A	D	E	L	P	S	space
A	0	0.3	0	0.2	0.2	0.3	0
D	0.2	0.1	0.2	0.1	0	0.2	0.2
E	0.1	0.2	0.1	0.2	0.1	0.1	0.2
L	0.1	0	0.2	0.2	0.1	0.2	0.2
P	0.2	0	0.2	0.2	0.1	0.1	0.2
S	0.2	0	0.2	0.1	0.1	0.1	0.3
space	0.2	0.1	0.2	0.1	0.1	0.1	0.2

Fig. 1.1. (a) Character probabilities. (b) Character probabilities given the previous character.

Having mastered the necessary techniques, it is possible to go on to create improved approximations to English in the ways outlined above. A useful addition to any of the programs that proceed letter by letter would be to determine the percentage of genuine English words generated. Ways of measuring the effectiveness of other methods can also be devised.

1.3

SUBSTITUTION

The process of substituting one word for another is of value in all kinds of text processing. A word processor of any sophistication, for example, will have a 'search and replace' facility with which a word or phrase can be located and, when found, replaced by another.

It was realised by early workers in artificial intelligence that this apparently mundane device could be used as the basis of a program to enable the computer to converse with humans in a natural way. The prime example of a program of this kind is ELIZA, which was developed by Professor Joseph Wiezenbaum at the Massachusetts Institute of Technology in 1966. ELIZA sustains a conversation by generating responses that are based on what is addressed to it. Each response is created by taking the input, searching it for particular key words which are replaced, and then rearranging it. This may not seem to be a very suitable way to create dialogue generally and, indeed, it is not. But the point about ELIZA is that it is intended to take part only in conversations of a very restricted kind.

In fact, ELIZA is intended to simulate the contributions of a nondirective therapist in conversation with a patient undergoing nondirective therapy. A therapist of this kind aims to draw emotions from the patient and then to convey them back to the patient in such a way that they can be recognised and acknowledged. To do this, the therapist must stimulate the patient, but in a non-committal way, to extract the emotions associated with the patient's remarks and then reflect them back again.

ELIZA's restricted field for conversation was chosen very deliberately, for it is one where almost no background knowledge is required to sustain conversations. In this respect it is unlike almost any other area of conversation. While ELIZA undoubtedly simulates the conversation of a nondirective therapist very successfully, no wider accomplishments can be claimed for it. In its way, it is typical of much work in artificial intelligence in that it is successful in a carefully restricted area of application.

But the way that ELIZA has been used has attracted a good deal of criticism, not least from its originator. For it has been used to engage

patients genuinely in need of psychotherapy in conversation. It has even been welcomed as a means of introducing automatic psychotherapy. The problem with this is that, although ELIZA can converse, it can reveal no understanding of what it is conversing about. Because it converses by applying mechanistic rules for the substitution and rearrangement of text, it cannot possibly offer any sympathetic recognition or understanding to the patient.

The puzzle

The topic of this puzzle is the apparently simple idea of searching a character string for a particular pattern of characters and, when it is found, replacing it by another pattern. As we have seen, this idea has applications of considerable sophistication. The way in which it is used by ELIZA to carry on an apparently meaningful conversation can be illustrated quite easily. By replacing 'you' with 'I' and 'me' with 'you', and adding a standard phrase, the program can respond to remarks such as 'You don't impress me' with 'Why do you think that I don't impress you?'.

With such matters in mind, this puzzle is to write a program to accept an input and store it under the name A$, to search it for all occurrences of a pattern stored under B$, and to replace each occurrence of this pattern with a second pattern which has also been previously stored under C$. The result of making the substitution should then be printed. As an example of what the solution program should do, if it contains the assignments:

```
B$="THATCHER"
C$="KINNOCK"
```

then the input 'MRS THATCHER MET TWO THATCHERS' should lead to the output 'MRS KINNOCK MET TWO KINNOCKS'.

1.4

THE QUOTES PUZZLE

Any character can be placed inside quotes as a member of a character string, except for a quotation mark itself. The reason for this is, of course, that a quotation mark that is intended as a member of a character string is indistinguishable from the quotation mark used to terminate the string.

The intended effect of a command such as:

```
PRINT "HE SAID "NO""
```

is probably clear to most people whether they are familiar with computers or not. Unfortunately, however, the response of a micro to the command is not to give this intended effect. The nearest it will come is to display HE SAID. This is because a PRINT command causes the characters between quotation marks that immediately follow it to be displayed. After the opening quotation mark, the next one to be found is automatically treated as the accompanying closing quotation mark. In this case, therefore, the character string immediately following the PRINT keyword is 'HE SAID'.

Actually, a number of micros will display HE SAID followed by a zero. This is because they treat the character string following PRINT as a list of three items: the character string 'HE SAID', the variable name NO and the null string. A few micros, including the Sinclair QL, will just treat the command as invalid.

The quotes problem is a particular case of a general problem that arises in any computer language when one of the standard characters in its character set is used artificially as a delimiter or a separator. Whenever a character is used in one of these ways, special steps must be taken to deal with any natural occurrences of the character between its artificial uses. A further example of this affecting many microcomputer dialects of BASIC is that a colon must not appear in a character string. This is because the colon is used artificially as a separator for multiple statements appearing on the same program line.

The puzzle

To make our problem specific, suppose we have a program that includes DATA statements, each containing a line of BASIC, so that the program can print out another program. Part of the program would be something like:

```
10 READ A$: PRINT A$
100 DATA "50 LET X=6"
```

But to place a program line such as 20 PRINT "BASIC" in a DATA statement, the program would need a line such as:

```
110 DATA "20 PRINT "BASIC""
```

As we have noted, this is not satisfactory.

To produce a portable program, we could replace the quotation marks in data items with, say, exclamation marks, which seldom appear in BASIC programs. The main program would then have to replace each occurrence of this symbol with a quotation mark after it had read the data but before printing it. The framework for such a program would be:

```
10 READ A$
110 DATA "20 PRINT !BASIC!"
```

The puzzle is to complete this program so that, by replacing the exclamation marks, it prints any statements represented artificially in its data in their intended form. Standard BASIC functions only should be used, so that the solution is a portable program.

1.5

CROSSWORD

A crossword is a natural arena for a word puzzle on a computer. Its solution will require the use of techniques for handling strings and arrays. The computer can be made to display the skeleton for a crossword, to accept entries for it, and even to verify and display these entries. By writing a program to do these things, an 'interactive crossword' can be created which can do a great deal more than a crossword that is merely printed in a newspaper. Besides verifying entries, it could even be made to fill in an entry if the person tackling the crossword were genuinely stuck.

The problems associated with a crossword include how the numbering plan can be computed from its skeleton and how the length of the solution to each clue can be found. They could also include verifying the solver's attempt at the answer to any clue. The solution of these problems leads to the creation of a truly interactive crossword that is much more user friendly than a conventional one. In creating this, it can be demonstrated that the computer is a much more advanced medium for supporting a crossword than the commonly used medium of paper.



Fig. 1.2. Crossword solution.

The solution of a small crossword is shown in Fig. 1.2. It consists only of black squares and white squares containing letters. The skeleton of the crossword consists of a similar array containing only black and white squares. The black squares are in the same positions as before, and plain white squares take the positions occupied by letters. In fact, it is preferable to represent the skeleton with a surround of dark squares, as shown in Fig. 1.3. If this is done, the squares at the edge of the crossword can be treated in the same way as all the others during computations on them. It also makes the display of the crossword on the screen more effective.

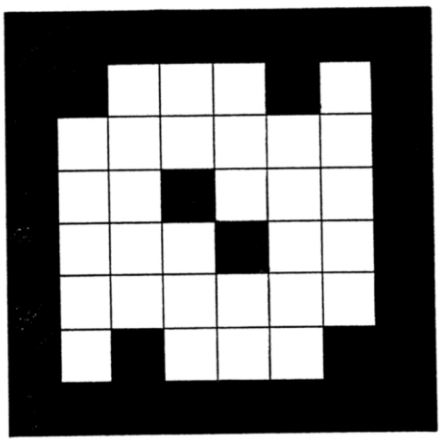


Fig. 1.3. Crossword skeleton.

The solution to a crossword can be represented in the computer by using a two-dimensional array of string variables. Each element of the array can 'cover' a single square of the crossword by containing the letter that should be placed in that square when the crossword is filled in or some other character to represent a black square.

The numbering plan for a crossword is the set of numbers that is added to its skeleton. They identify the positions in which the solutions of the clues with the corresponding numbers are to be entered on the skeleton. Each clue is identified by a number followed by the word 'across' or 'down'. The solution to 5 across, for example, must be entered starting in the square numbered 5 and going across

or horizontally from there. The solution to 8 down would start in the square numbered 8 and go down.

The puzzle

This puzzle has several parts. The first is to devise a way to represent the solution to a crossword puzzle, such as the one illustrated in Fig. 1.2, in a two-dimensional array. But this must be done in such a way as to facilitate the remaining parts of the puzzle. Once a way of representing the solution has been devised, a program to display its skeleton should be written. The program should then be extended to compute the numbering plan for the puzzle, and arrangements made to print the numbering plan on the skeleton. The numbering plan can be added subsequently to the display of the crossword, but it is probably simpler to write a single routine to display both. Then a means of finding the length of the solution to each clue by processing the skeleton should be devised. When this is done the program should be able to print a list of the numbers of all the clues to the crossword, with the length of the solution to each, merely by examining the skeleton. A typical item in the list would be '5 ACROSS (6)'. The final problem is to accept an attempt at the solution to a clue and to determine whether or not it is correct.

When all these problems have been solved, the elements of an interactive crossword have been created. The addition of routines for displaying the correct attempts at clues and for providing help would add the finishing touches.

SOLUTION 1.1: WORDS

The following solution program works on the assumption that the second word can be derived from the first. It takes each letter of the second word in turn, searches the first word for the letter and, when it finds it, generates a character string of the form MID$(WORD$,P,1) where P is the position of the letter in the first word. Preceding these output strings by the word PRINT gives a command that will generate the second word from the first. The program is:

```
10 INPUT WORD$
20 INPUT SUBWORD$
30 LW=LEN(WORD$)
40 LS=LEN(SUBWORD$)
50 FOR J=1 TO LS
60 FOR K=1 TO LW
70 IF MID$(SUBWORD$,J,1)=MID$(WORD$,K,1)
   THEN A(J)=K
80 NEXT K
90 NEXT J
100 PRINT "PRINT ";
110 FOR K=1 TO LS
120 PRINT "MID$(WORD$,"; A(K); ",1)";
130 IF K<>LS THEN PRINT "+";
140 NEXT K
```

When this program is given the inputs HORSE and SHE, it produces the output:

```
PRINT MID$(WORD$,4,1)+MID$(WORD$,1,1)+MID$
(WORD$,5,1)
```

This solution is fairly primitive, but it has the merits of being short and understandable. It also provides a skeleton that is sufficiently strong to support considerable modification, as is shown below.

The solution can be improved in at least two major ways. First, it ought to validate the input by checking that the second word given to the program actually can be derived from the first one. This is straightforward, and the donkey work for it has already been done by lines 50 to 90 of the program. After line 90, checking that LS elements

of the array A have non-zero values ensures that the conditions are satisfied. If the check fails then the input is not satisfactory.

The second improvement is that the program could write strings such as MID$(WORD$, K, 4) as suggested when the puzzle was posed. This can be done by modifying the second part of the program from line 110 onwards. The algorithm can be expressed using the standard structured repetition facilities as possessed by, for instance, Pascal as:

```
Initialise the position in the subword
REPEAT
  Initialise Count and Beginning for the next MID$
  WHILE not at the end of the subword
    WHILE A(K+1)=A(K)+1
      Increment Count and position
    ENDWHILE
  ENDWHILE
  Print MID$ command
  IF not at the end of the subword THEN Print +
  Increment position
UNTIL all letters are dealt with
```

This algorithm shows that FOR...NEXT is not the required repetition structure. In BASIC, the WHILE and REPEAT structures must be created from scratch using IF...THEN and GOTO instructions. The key instruction involves the condition A(K+1)=A(K)+1, which shows whether successive letters of the subword are also successive letters of the original word.

The resulting BASIC program is:

```
10 INPUT WORD$
20 INPUT SUBWORD$
30 LW=LEN(WORD$)
40 LS=LEN(SUBWORD$)
50 FOR J=1 TO LS
60 FOR K=1 TO LW
70 IF MID$(SUBWORD$,J,1)=MID$(WORD$,K,1)
   THEN A(J)=K
80 NEXT K
90 NEXT J
100 PRINT "PRINT ";
110 K=1
120 C=1: B=A(K)
```

```
13Ø IF K=LS THEN 15Ø
14Ø IF A(K+1)=A(K)+1 THEN 19Ø
15Ø PRINT "MID$(WORD$,";B;",";C;")";
16Ø IF K<>LS THEN PRINT "+";
17Ø K=K+1: IF K>LS THEN STOP
18Ø GOTO 12Ø
19Ø C=C+1: K=K+1: GOTO 13Ø
```

If this program is given the words HORSE and HOSE it will give the output:

```
PRINT MID$(WORD$,1,2)+MID$(WORD$,4,2)
```

This compares favourably with the previous solution program which would give four MID$ commands.

SOLUTION 1.2: APPROXIMATIONS TO ENGLISH

The first improvement can be generated by the following program which, after storing the letters to be used and their probabilities, uses the random number generation function RND to decide repeatedly which of the letters is to be displayed. That the values generated by RND cover the range from zero to one is a considerable convenience, for this is exactly the range of the total probability distribution. The program is:

```
10 DIM A$(7), P(7)
20 L$="ADELPS "
30 FOR K=1 TO LEN(L$)
40   A$(K) = MID$(L$,K,1): READ P(K)
50 NEXT K
60 DATA 0.2,0.1,0.2,0.1,0.1,0.1,0.2
70 X = RND: I=1: S=P(1)
80 IF X<S THEN PRINT A$(I);: GOTO 70
90 I = I + 1: S = S + P(I): GOTO 80
```

A typical output from this program begins ELPS DAP ESSEP PLASES A... The program continues to produce output until it is stopped, of course.

The second improvement can, after reading the probabilities into an array P(7,7), be produced by this program fragment:

```
200 J=7: PRINT A$(J)
210 X=RND: I=1:S=P(J,1)
220 IF X<S THEN PRINT A$(I);: J=I: GOTO 210
230 I=I+1: S=S + P(J,I): GOTO 220
```

Typical 'English' as produced by this program is SADE ELLAS PLADSEAL ADDED...

SOLUTION 1.3: SUBSTITUTION

The substitution program is:

```
1Ø B$="THATCHER": C$="KINNOCK"
2Ø B=LEN(B$): Z$="": P=B-1
3Ø INPUT A$: L=LEN(A$)
4Ø FOR K=1 TO L-B+1
5Ø IF MID$(A$,K,B) <> B$ THEN 9Ø
6Ø Z$=Z$ + C$: K=K+B-1
7Ø IF K > L-B+1 THEN P=L-K
8Ø GOTO 1ØØ
9Ø Z$=Z$ + MID$(A$,K,1)
1ØØ NEXT K
11Ø IF P > Ø THEN Z$=Z$+RIGHT$(A$,P)
12Ø PRINT Z$
```

This works by copying the input string to Z$, while searching it for the string stored under B$. If it finds the latter, then it copies the string stored under A$ to Z$ instead. When the part of the input string that remains to be searched is shorter than the string for which the program is searching, the program halts its search. This saves unnecessary work and, more importantly, it prevents illegal values for its arguments from reaching the function MID$. The variable P is used to hold the length of any remaining 'tail' to the input string that must finally be copied to Z$.

The following program, which is simpler than the solution presented, works on some types of micro:

```
1Ø B$="THATCHER": C$="KINNOCK"
2Ø B=LEN(B$): Z$=""
3Ø INPUT A$
5Ø FOR K=1 TO LEN(A$)
6Ø IF MID$(A$,K,B)=B$ THEN Z$=Z$+C$: K=K+B-1:
   GOTO 8Ø
7Ø Z$=Z$+MID$(A$,K,1)
8Ø NEXT K
9Ø PRINT Z$
```

Whether or not it works depends on the way that MID$ is

implemented, for if the evaluation of MID$(A$, K, B) is not permitted when K+B−1 > LEN(A$), which would be entirely reasonable and even desirable, then the program will fail. The 'jump' instructions in the previous solution are intended to prevent any attempt to evaluate MID$ in such dubious circumstances.

SOLUTION 1.4: THE QUOTES PUZZLE

This solution to the quotes puzzle makes use of the fact that the ASCII code for a quotation mark is 34. After reading a BASIC line from a DATA statement, it scans the line character by character searching for an exclamation mark. If it finds one then it replaces it by character 34, a quotation mark.

```
10 READ A$
20 L=LEN(A$)
30 FOR K=1 TO L
40 C$=MID$(A$,K,1)
50 IF C$ <> "!" THEN PRINT C$;: GOTO 70
60 PRINT CHR$(34);
70 NEXT K
100 DATA "10 PRINT !BASIC!"
```

SOLUTION 1.5: CROSSWORD

An array named CROSS$ will be used to store the crossword. As the crossword has 6 rows and 6 columns, we shall give CROSS$ dimensions of 7 by 7. Rows 0 and 7 and columns 0 and 7 can then be used to hold the border. The crossword itself can now be represented by:

```
1Ø DIM CROSS$(7,7),N(6,6)
2Ø FOR J=1 TO 6: FOR K=1 TO 6
3Ø READ CROSS$(J,K)
4Ø NEXT K: NEXT J
5Ø DATA " ","B","O","A"," ","C"
6Ø DATA "P","A","S","C","A","L"
7Ø DATA "A","S"," ","E","R","E"
8Ø DATA "R","I","O"," ","M","A"
9Ø DATA "S","C","R","E","E","N"
1ØØ DATA "E"," ","A","N","D"," "
```

The border can be added by:

```
11Ø FOR L=Ø TO 7
12Ø CROSS$(Ø,L)=" ": CROSS$(7,L)=" "
13Ø CROSS$(L,Ø)=" ": CROSS$(L,7)=" "
14Ø NEXT L
```

After this, the skeleton of the crossword can be displayed by a simple procedure which we shall package as a subroutine starting with line number 1000. It will display the crossword in the opposite style to that in which it appears on paper, i.e. the squares that are black on paper will appear light on the screen. After adding the calling instruction:

```
15Ø GOSUB 1ØØØ
```

the display subroutine is written as:

```
1ØØØ REM DISPLAYROUTINE
1Ø1Ø FOR J=Ø TO 7: FOR K=Ø TO 7
1Ø2Ø IF CROSS$(J,K)=" " THEN PRINT CHR$(166);:
     GOTO 1Ø4Ø
```

```
1Ø3Ø PRINT " ";
1Ø4Ø NEXT K: PRINT: NEXT J
1Ø5Ø RETURN
```

(This program is written for the Commodore 64, on which character 166 is a light character-sized block.)

The numbering plan can be determined automatically from the skeleton. This is done by examining each square in turn, row by row, starting at the top left and finishing at the bottom right. The numbers

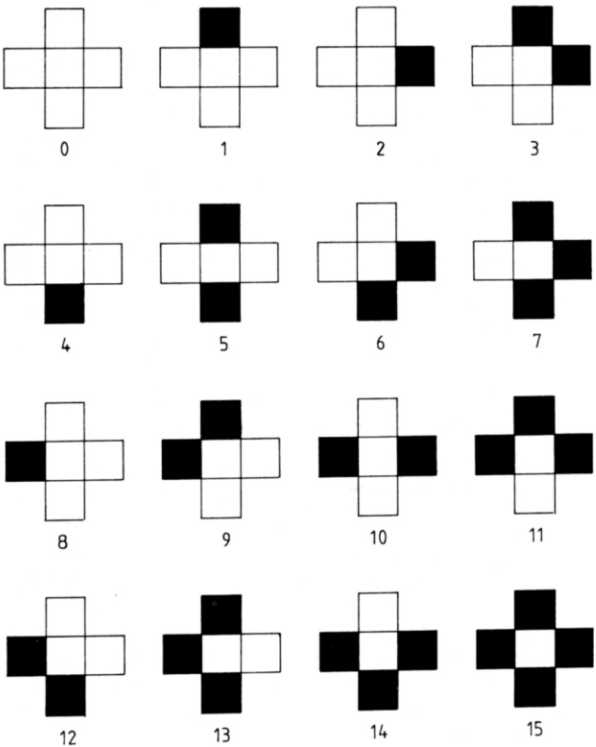


Fig. 1.4. Configurations and codes.

are associated only with blank squares, so the others may be ignored during this process. When each blank square is examined, a code is assigned to it in a way that reflects the four neighbouring squares to its right and left, and above and below. All the possible configurations for the squares surrounding a blank square are shown in Fig. 1.4. The code is determined by counting 1 for a black square above, 2 for one to the right, 4 for one below, and 8 for one to the left. The codes are also shown on Fig. 1.4. A little thought will show that solutions can only begin in a square having one of the codes 1, 3, 8, 9, 11, 12 and 13.

But the codes give more information than just where a solution begins. They also show whether the word goes across or down, and even if a square can have answers starting from it going both across and down. By referring to Fig. 1.4 again, we can see that words starting in a square with the code 8,12 or 13 go across, and those starting in a square coded as 1, 3 or 11 go down. If a square has the code 9 there will be words going across and down from it.

Using this information we can find the numbering plan and print it on the skeleton with the following program, which computes the numbers and stores them in the array named N. This section of program starts at line 150, overwriting the call to the display subroutine which is not needed again as this section of program contains its own display routine. After adding the dimensions for the array N to line 10 with:

```
10 DIM CROSS$(7,7), N(6,6)
```

the addition to the program is:

```
150 N=1
160 FOR ROW=1 TO 6: FOR COL=1 TO 6
170 IF CROSS$(ROW,COL)=" " THEN 240
180 T=0
190 IF CROSS$(ROW-1,COL)=" " THEN T=T+1
200 IF CROSS$(ROW,COL+1)=" " THEN T=T+2
210 IF CROSS$(ROW+1,COL)=" " THEN T=T+4
220 IF CROSS$(ROW,COL-1)=" " THEN T=T+8
230 IF T=1OR T=3OR T=8OR T=9OR T=11OR T=12OR
    T=13 THEN N(ROW,COL)=N: N=N+1
240 NEXT COL
250 NEXT ROW
260 FOR ROW=0 TO 7: FOR COL=0 TO 7
```

```
27Ø IF CROSS$(ROW,COL)=" " THEN PRINT
    CHR$(166);: GOTO 3ØØ
28Ø IF N(ROW,COL)<>Ø THEN PRINT RIGHT$
    (STR$(N(ROW,COL)),1);: GOTO 3ØØ
29Ø PRINT " ";
3ØØ NEXT COL:PRINT: NEXT ROW
```

Note that when the skeleton is printed complete with numbers, any number from 10 upwards is truncated so that only its least significant digit is displayed. This is because only one character can be displayed in any one character position. But despite the fact that only the last digit of the number is displayed, the number is computed in full. To show this we can print a list of the numbers and directions for all the clues in the crossword by adding the lines:

```
222 IF T=1 OR T=3 OR T=11 OR T=9 THEN PRINT N;
    " DOWN"
224 IF T=8 OR T=12 OR T=13 OR T=9 THEN PRINT N;
    " ACROSS"
```

The length of each solution can be computed and added to the list of clues by amending the two lines just given so that they each call a subroutine, and then adding the subroutines themselves. The amendments and additions are:

```
222 IF T=1OR T=3OR T=11OR T=9 THEN PRINT
    N;" DOWN";: GOSUB 9ØØ
224 IF T=8OR T=12OR T=13OR T=9 THEN PRINT
    N;" ACROSS";: GOSUB 8ØØ

8ØØ W=1: X=COL
81Ø X=X+1
82Ø IF CROSS$(ROW,X)<>" " THEN W=W+1: GOTO 81Ø
83Ø PRINT "("; W; ")",
84Ø RETURN
9ØØ W=1: Y=ROW
91Ø Y=Y+1
92Ø IF CROSS$(Y,COL)<>" " THEN W=W+1: GOTO 91Ø
93Ø PRINT "("; W; ")",
94Ø RETURN
```

Finally, a routine that accepts an attempt at a clue and determines whether it is correct is presented. It requires the user to enter the number of the clue, whether it goes across or down, and the attempt itself. It then finds where the answer begins in the array CROSS$,

ensures that there is such a clue and, if there is, checks the entered answer by comparing it letter by letter with the appropriate part of the array CROSS$. The routine is:

```
45Ø INPUT "NUMBER";M
46Ø INPUT "A FOR ACROSS, D FOR DOWN";D$
47Ø INPUT "SOLUTION"; S$
48Ø J=1: K=1
49Ø IF N(J,K)=M THEN 52Ø
5ØØ K=K+1: IF K=7 THEN K=1: J=J+1
51Ø GOTO 49Ø
52Ø T=Ø
53Ø IF CROSS$(J-1,K)=" " THEN T=T+1
54Ø IF CROSS$(J,K+1)=" " THEN T=T+2
55Ø IF CROSS$(J+1,K)=" " THEN T=T+4
56Ø IF CROSS$(J,K-1)=" " THEN T=T+8
57Ø IF (T=8OR T=9OR T=12OR T=13) AND D$="A"
   THEN 6ØØ
58Ø IF (T =1OR T=3OR T=9OR T=11) AND D$="D"
   THEN 65Ø
59Ø PRINT "NO CLUE ";M;D$: STOP
6ØØ W=1:X=K
61Ø IF MID$(S$,W,1)<>CROSS$(J,X) THEN PRINT
   "INCORRECT":STOP
62Ø X=X+1
63Ø IF CROSS$(J,K)<>" " THEN W=W+1:GOTO 61Ø
64Ø PRINT "CORRECT": STOP
65Ø W=1:Y=J
66Ø IF MID$(S$,W,1)<>CROSS$(Y,K) THEN PRINT
   "INCORRECT":STOP
67Ø Y=Y+1
68Ø IF CROSS$(Y,K)<>" " THEN W=W+1:GOTO 66Ø
69Ø PRINT "CORRECT": STOP
7ØØ END
```

Chapter Two
Number Puzzles

The puzzles in this section are all based on numbers. The first is a classical type of puzzle in which the computer is used to search for a solution. By checking all the members of a large class of possible solutions, it unerringly finds the actual solution. The second puzzle begins to explore the benefits that can result from rearranging numerical formulae before beginning to compute values from them. It shows that a little thought about the form in which a computation is presented to the computer can make the computation a good deal easier to carry out. The third involves binary numbers, a topic that is surely obligatory in any book dealing with computers. Although computers work in binary fashion, there is no reason why their users should know anything about the binary system. The use of a high-level language or an application program should shield them from such matters. But binary numbers provide the basis of many useful techniques, particularly for enumeration.

The next puzzle takes advantage of the fundamental technique of numerical integration, which involves little more than counting the squares covered by a region in order to find its area. As the squares are made smaller their number increases and it would become more tedious for us to count them – but not for a computer to do so. The last two puzzles are based on the use of an array of numbers to represent a network. To represent a picture, even a schematic picture, of a network in a computer at all is a considerable achievement. But doing it using an existing mathematical structure, a *matrix*, has the advantage that well known operations on matrices can provide exactly the tools needed to solve problems involving networks, and consequently those involving pictures.

2.1

SEARCHING FOR A SOLUTION

Many numerical problems can be solved by searching through sets of numbers for one that provides a solution to the problem. The basic procedure is to generate numbers in some suitable order, to treat each as a potential solution and then to test whether it really is a solution. This 'brute force' method often works, but there are some problems associated with it. The major one is that, if many numbers are to be searched through, it can take a long time to come to one that provides a solution to the problem. It may even take an impossibly long time.

To illustrate this, when the three-digit numbers are considered as potential solutions to a problem, there are a thousand of them to be examined. If four-digit numbers are considered, there are ten thousand of them. And every time the number of digits increases by one, the search area increases by a factor of ten. The time needed for a search to reveal a solution will increase correspondingly.

Many problems in artificial intelligence are solved by conducting a search of the potential solutions in just this way. Problems of any complexity naturally have huge numbers of them. Often what makes an artificial intelligence program successful is the discovery of a method of limiting the search pattern to a small proportion of the potential solutions without discarding any actual solutions.

The puzzle

There is a four-digit number, containing three different digits which we can represent by x, y and z, which takes the same value whether it is written as $x^{y}z^{x}$ or as xyzx. The puzzle is to find the values of x, y and z that make $x^{y}z^{x}$ take the value given by xyzx. Once a program to do this is written, it is a small matter to make it confirm that there is only one solution to the problem.

2.2

REARRANGEMENTS

Series such as the one in the following puzzle are known as *power series* because they give an expression for a function of x as a series of powers of x. The series are infinite in that they have an infinite number of terms. This may make it seem as if the series is of limited value in computing the values of a function, as not even a computer can add an infinite number of terms. But this is not so. The sum of a few terms gives the value of the function with a certain accuracy, and adding more terms from the series improves the accuracy with which the value of the function is computed. The accuracy of the result is controlled by the value of the first term to be omitted from the sum.

The puzzle gives the exponential series, and there are also power series for all the common functions that are available on a calculator or in a scientific programming language. The series for sin x, cos x and the natural logarithm of x, for example, are:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots$$

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \ldots$$

$$\log_e (1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \ldots$$

Values of these functions, and others, are often computed using their power series by both calculators and programming languages. Fortunately, as the solution to the puzzle shows, it is by no means necessary to store an infinite series of terms in the computer to do this. If the summation is to proceed starting from the lowest power of x, then all that is needed is the first term and a rule for generating successive terms from it thereafter. Alternatively, if the summation is to proceed in the opposite direction the end point and the rule for generating the terms from there to the beginning of the series are needed.

When dealing with infinite series, great care is needed in any

circumstances but particularly in computations. All the series we have mentioned so far converge to values of the functions indicated. But the series:

$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \ldots$$

does not converge at all. That is to say, its sum is infinity. This can be demonstrated by reasoning that the first two terms add to one and a half, the next two terms contribute more than one half, the next four more than another half, as do the following eight, and the 16 after that, and so on. Thus, the total sum is more than one and a half plus an infinite number of halves. But if we set the computer to add the terms of the series, at some stage the term $1/n$ will become indistinguishable from zero to the computer and from that point on the computer will be adding nothing to the sum, thereby giving a finite and wrong answer. We also have to be careful about rearranging the order of the terms in some infinite series, for this can also cause incorrect results to be obtained.

The puzzle

When computing values from a formula, it is often a considerable convenience to rearrange the formula prior to the computation. To illustrate this, when computing values of the expression $4x^3 + 3x^2 + 2x + 1$, say, on a calculator, it pays to rearrange the expression as $((4x + 3)x + 2)x + 1$. This is because values of the expression can then be calculated, starting with the x in the innermost bracket, by a sequence of alternating multiplications and additions and without the need to store any intermediate results.

Similarly, when calculating values of a function from its series representation the series may first be rearranged to advantage. Thus when calculating values of the exponential function from its series, the series can be written in its usual form, which is:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots$$

But the series can be rearranged as:

$$e = 1 + x(1 + \frac{x}{2}(1 + \frac{x}{3}(1 + \frac{x}{4}(1 + \ldots$$

When written in the second way, an approximation to the value of the exponential function can be calculated from the truncated series by using a simple repetitive procedure. Additionally, of course, the computation must proceed from right to left, rather than from left to right as is the natural direction with the series in its original form.

The puzzle is to write compact programs to compute the approximate value of the exponential function using the series in its usual form and then in its rearranged form, and to compare the lengths of time taken for the two computations.

2.3

BINARY DISPLAY

The internal operations of computers are, of course, entirely binary. One of the main functions of BASIC, and of all other high-level languages, is to conceal this and make the computer easy to use. But one of the breakthroughs of microcomputer dialects of BASIC is their possession of the PEEK and POKE instructions. These allow the BASIC programmer to see into the hardware of the computer to examine and to change its contents, thereby making the binary digits stored in the computer accessible. Access to the binary digits stored in the computer opens up a completely new dimension of computer use to the BASIC programmer. It brings, for example, the possibility of writing BASIC programs that amend existing programs, and the ability to control the detailed operation of the hardware, perhaps in an application where the computer is controlling or communicating with other equipment.

But these instructions must be used with care or, at least, the POKE instruction must. While mere examination of the binary digits stored in the computer can do no harm, the indiscriminate changing of binary digits with the POKE instruction can cause untold damage. It may result not only in a change to a program stored in the computer but even in changes to the BASIC translation system itself. This explains in some measure why the PEEK and POKE instructions are not available on computers that are shared by many users, for it is one thing for a programmer to spoil or destroy his own efforts and quite another to do this to others.

Once the BASIC programmer can access the binary digits stored in the computer, some means of manipulating them are needed. Most dialects of BASIC provide the AND operator described in the puzzle and an OR operator which also operates on two binary digits to give a result as shown in the following table:

X	Y	X OR Y
0	0	0
0	1	1
1	0	1
1	1	1

The operator NOT is usually also provided.

Unfortunately some dialects of BASIC, including that for the Sinclair Spectrum and ZX81, implement the AND, OR and NOT operators in a much more simplistic fashion which really does not deal in terms of the binary digits themselves at all. The handling of binary digits is not, therefore, a practical proposition in these dialects of BASIC.

The puzzle

This puzzle is to write a compact BASIC program to examine any location in the computer's memory and to display its contents in binary form. Thus the output of the program should appear as a sequence of binary digits representing in a natural way the contents of the location, with the most significant digit on the left and the least significant on the right. Given the address, a genuine three-line program can be written for this (that is, one with no program lines containing multiple statements).

The PEEK function can be used to examine the contents of a memory location, but it returns a decimal number. This makes the crux of the problem the conversion of a decimal number to binary form. The most common way of doing this is with repeated division by two, when the remainders of the successive divisions give the binary form. Unfortunately the binary digits are generated with the least significant first, so they must be stored if the most significant is to be written first.

An alternative way of doing the conversion uses the AND operator. AND takes two inputs and gives one output, as in this table:

X	Y	X AND Y
0	0	0
0	1	0
1	0	0
1	1	1

When used in BASIC, the AND operator can be applied to two decimal numbers. It then operates on their binary forms, as shown in the table, using their two least significant digits to set the least significant digit of the result – and so on, digit by digit, up to the most significant. Thus the value of (189 AND 102) is 36, as shown by:

```
      1 0 1 1 1 1 0 1
AND   0 1 1 0 0 1 1 0
      -----------------
      0 0 1 0 0 1 0 0
      -----------------
```

In similar fashion, the binary form of any number can be examined by using AND with a second number. The second number constitutes a 'mask' because the zeros in its binary form 'cover up' the corresponding digits in the original number. After the AND operation, the only parts of the original number that remain are those in the positions corresponding to those of the ones in the 'mask'.

2.4

THE TETHERED GOAT

The problem of the tethered goat is an old puzzle and one which, as far as I know, is not capable of exact mathematical solution. It provides a perfect illustration of how a numerical method can give an approximation to the solution of a problem that cannot be solved precisely. Further, the availability of a computer to carry out the numerical procedure for finding the approximate solution allows us to obtain approximations with errors so small as to be negligible. This means that the inability to solve a problem exactly need be of no consequence whatever. It even happens, on occasion, that the exact solution of a problem is such a horrendous mathematical expression that it is preferable to obtain numerical results using a numerical method that gives a good approximation rather than to try to evaluate the expression giving the exact solution.

The technique of finding an area by counting the squares that it

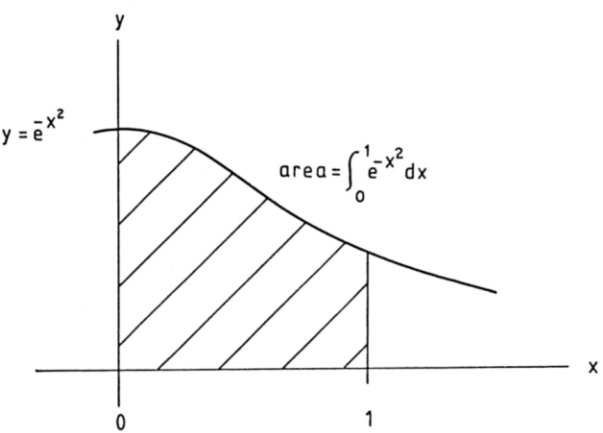


Fig. 2.1. An integral as an area.

covers and adding *their* areas is one that is habitually used to evaluate integrals. This is because the value of any definite integral of the form $\int_a^b f(x)dx$ can be interpreted as an area, as shown in Fig. 2.1 where the area in question is shaded. So even when it is not possible to integrate the function appearing in a definite integral, it is still possible to find a good estimate of the value of the integral in a fairly simple way.

It is this idea of counting the squares that we borrow for the solution of our problem.

The puzzle

The familiar idea from geometry and computational methods that the area surrounded by a curve can be estimated with some degree of accuracy by subdividing the area into squares is taken as the basis for solving this problem. After counting the number of squares that fall within the boundary defined by the curve, the area is found by multiplying the area of one of the squares by this number. The smaller the subdivisions, the more accurate the approximation to the enclosed area will be. The way that the squares only partly within the boundary are treated also affects the accuracy of the result. A strategy for dealing with them that gives, on average, good results is to count a square only if its centre falls within the boundary.

The diagram in Fig. 2.2 shows how the area of a circle might be found by this method. The following program, based on this diagram, computes an approximation to the area of a circle with radius 10 and centred at the origin by counting all the unit squares with their centres within its perimeter. Using the formula for the area of a circle, such a program could also compute an approximation to the value of pi.

```
10 N=10
20 R=N: AREA=0
30 FOR J= -N TO N-1
40   FOR K= -N TO N-1
50     D=SQR( (J+0.5)^2 + (K+0.5)^2 )
60     IF D<R THEN AREA = AREA + 1
70   NEXT K
80 NEXT J
90 PRINT "AREA IS "; AREA
```

The puzzle is to write a program, based on this method of estimating an area, to find an approximation to the following

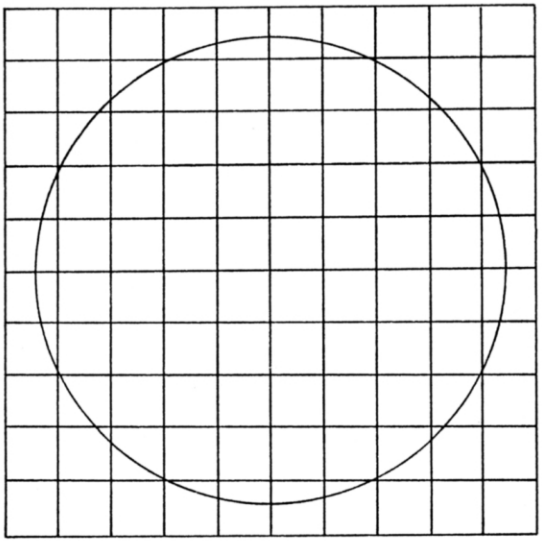


Fig. 2.2. Finding the area of a circle.

problem. A goat is tethered at the edge of a circular meadow. The meadow is of radius R, and the lead has length L, as shown in Fig. 2.3. What is the value of the ratio L/R when the area that the goat can graze is exactly half that of the meadow?

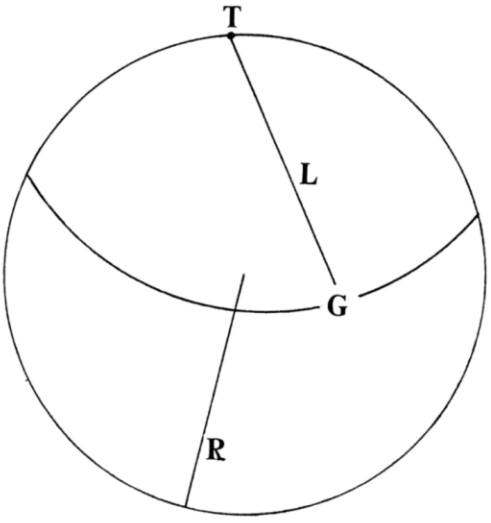


Fig. 2.3. A goat (G) is tethered at T by a lead of length L in a circular field of radius R.

2.5

NETWORKS

Networks of the type illustrated in Fig. 2.4, consisting of nodes and directed connections between them, are known as *directed graphs*. They can be used to model situations that involve flows between various points. As explained in the puzzle below, a directed graph can be represented by recording its connections with a two-dimensional array of numbers known as a *connection matrix*.

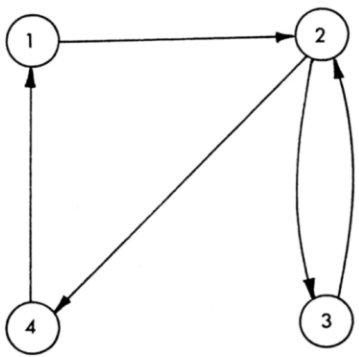


Fig. 2.4. Network.

This puzzle involves writing a routine for matrix multiplication. Now, matrix multiplication is different from number multiplication in one very significant, and perhaps surprising, way. With any two matrices A and B it is true, in general, that A×B is different from B×A, so the order in which two matrices are multiplied is important. This can be illustrated with a simple example, for when:

$$A = \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix} \text{ and } B = \begin{bmatrix} 0 & 1 \\ 1 & 2 \end{bmatrix}$$

we have

$$A \times B = \begin{bmatrix} 2 & 5 \\ 1 & 2 \end{bmatrix} \text{ and } B \times A = \begin{bmatrix} 0 & 1 \\ 1 & 4 \end{bmatrix}$$

There are other ways to represent networks with matrices, and we can illustrate two of them with reference to the non-directed graph in Fig. 2.5. This graph consists of a set of nodes and a set of edges, both

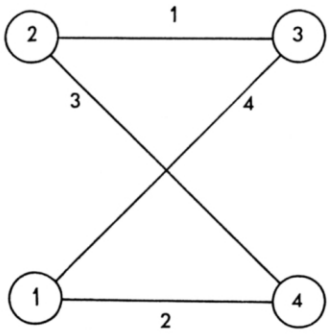


Fig. 2.5. A network of nodes and edges.

of which are numbered. We can represent this network with a matrix, N, that records which nodes are on which edges by defining the terms of the matrix as:

$$N(J,K) = \begin{cases} 1 \text{ if node J is on edge K} \\ 0 \text{ otherwise} \end{cases}$$

This gives for the illustrated network:

$$N = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

We can also represent it with a matrix, E, that records which edges come to which nodes by defining the matrix terms as:

$$E(J,K) = \begin{cases} 1 \text{ if edge J is on node K} \\ 0 \text{ otherwise} \end{cases}$$

This gives:

$$E = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

Having derived these two matrices, if we multiply them together to get N×E we get:

$$N{\times}E = \begin{bmatrix} 2 & 0 & 1 & 1 \\ 0 & 2 & 1 & 1 \\ 1 & 1 & 2 & 0 \\ 1 & 1 & 0 & 2 \end{bmatrix}$$

As it happens, this matrix is the connection matrix for the network in Fig. 2.5, but it is also something more for in its diagonal line of 2s it gives the number of edges impinging on each node.

The puzzle

A network such as that in Fig. 2.4 can be used when modelling a considerable number of activities. It consists of nodes, indicated by circles containing numbers, and one-way paths with their associated directions indicated by arrows. A system of one-way streets joining different locations in a town can be modelled with its use, as can a network of pipes carrying oil from place to place or an electrical circuit.

A matrix, or two-dimensional array, can be used to represent a network so that it can be stored in a computer. When stored, it can be manipulated in various ways and can be used in a simulation program. A two-dimensional array named, say, C can be used to hold the representation of a network by defining each of its terms, C(J,K), to be equal to the number of paths from node J to node K. In this way the network in the illustration is represented by the matrix:

$$C = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

The matrix contains five ones, each representing one of the five one-way paths in the network. It represents the network as a whole by recording the ways in which its nodes are connected to each other by paths. This matrix can be stored in the computer by:

```
10 DIM C(4,4)
20 FOR J=1 TO 4: FOR K=1 TO 4
30    C(J,K)=0
40 NEXT K: NEXT J
50 C(1,2)=1: C(2,3)=1: C(2,4)=1
60 C(3,2)=1: C(4,1)=1
```

For the first part of the problem, print out the values stored in the two-dimensional array so that they are presented in the same way as in the matrix given above – in four rows and four columns with the value of C(J,K) appearing in row J and column K.

The matrix C can be squared, that is, multiplied by itself, to give a second matrix called, say, C2. The elements of C2 are given by:

```
C2(J,K)=C(J,1)*C(1,K)+C(J,2)*C(2,K)+
C(J,3)*C(3,K)+C(J,4)*C(4,K)
```

As the second part of the problem, write a compact subroutine that can be called, after C has been initialised, to compute the square of C. What is the significance of the values of the elements of C2 in terms of the original network?

2.6

REVERSAL

A system pipes water through a network of pipes and pumping stations from one reservoir to another. The system is represented in Fig. 2.6(a) as a directed graph, with pumping stations shown by numbered circles and pipes by directed lines with their arrows indicating the direction in which water can be pumped. In the situation illustrated, water is pumped from the reservoir where pumping station 1 is sited towards pumping station 5 which is on the other reservoir.

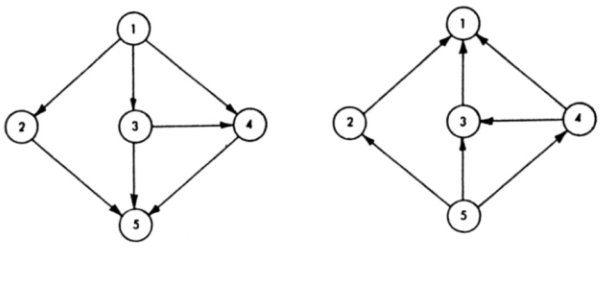


Fig. 2.6. (a) Normal flow in a network. (b) Reversed flow.

The network can be represented in a BASIC program by using a two-dimensional array named, say, P to hold the connection matrix of the network. (The idea of a connection matrix was explained in the previous puzzle, 'Networks'.) By doing this, the element of P in row J and column K of the matrix will record the number of pipes leading from station J to station K. In this fashion the matrix representing the network in the illustration can be initialised by:

```
1Ø DIM P(5,5)
2Ø FOR J=1 TO 5: FOR K=1 TO 5
3Ø    READ P(J,K)
```

```
40 NEXT K: NEXT J
50 DATA 0,1,1,1,0,0,0,0,0,1 ' 0 0
60 DATA 1,1,0,0,0,0,1,0,0,' 0 0
```

The puzzle

Occasionally the water must be pumped from station 5 to station 1, in the opposite direction to normal. This is done by reversing the direction of flow in every pipe, so that the network appears as in Fig. 2.6(b).

The puzzle is to write a short program fragment in BASIC to process the matrix P, initialised as above, so that it represents the situation when the water flow is reversed.

SOLUTION 2.1: SEARCHING FOR A SOLUTION

In principle, the solution is:

```
10 FOR X=0 TO 9
20 FOR Y=0 TO 9
30 FOR Z=0 TO 9
40   IF (X^Y)*(Z^X) = 1000*X+100*Y+10*Z+X
     THEN PRINT X,Y,Z: STOP
50 NEXT Z
60 NEXT Y
70 NEXT X
```

However, on some micros this does not work because of rounding errors in the computations of the values compared in line 40. In this case, steps must be taken to ensure that the two values are converted to exact integers before they are compared. This can be done either by using names for the variables under which only integer values may be stored, such as X% and Y%, or by adding 0.5 to the values and taking their integer parts before comparing them.

The program systematically generates all the possible combinations of values of X, Y and Z and tests each to see if it is the one that gives the solution to the problem. The STOP command can be included in line 40 as there is only one solution to the problem, but if you want to be sure that there are no other solutions then it can be removed to allow all the combinations to be tested. The program provides a framework for finding the solution to any puzzle of the style 'find values of x, y and z such that...'. Only line 40 of the program will need to be changed to accommodate different puzzles. For example, the solutions of the problem of whether there are any values of x, y and z such that the three-digit number xyz is precisely one third of the number zyx will be found by changing line 40 to

```
40 IF 3*(100*X+10*Y+Z)=100*Z+10*Y+X
   THEN PRINT X, Y, Z
```

In this case it emerges that there are no solutions, and so the STOP should be omitted.

The problem assumes an altogether different complexion as soon as ways of reducing the search area, so as to cut down to a minimum

the amount of computation that needs to be performed, are examined. The number of times that the loops need to be repeated can be restricted in several ways. We can observe that neither x nor z can be zero. If x is zero then xyzx will not be a four-digit number; and, since x cannot be zero, neither can z, otherwise $x^y z^x$ would itself be zero. Neither can x be one, for then the value of $x^y z^x$ would be that of z, again not a four digit number. In addition, any loop can be aborted as soon as the value of $x^y z^x$ exceeds 9999.

The computation can be made more efficient by taking expressions such as X↑Y and 1ØØØ*X+1ØØ*Y+X, which do not involve Z, outside the loop controlled by Z so that they are not unnecessarily recomputed during each loop.

Such refinements, however, will destroy the program's essential simplicity and make it much harder to understand. Also, in the context of a computer puzzle, I rather doubt that the analysis of ways of improving the efficiency of the computation is altogether worthwhile, unless it happens to interest you particularly. The idea of using a computer is to make it do work for you, and it may prove hard to argue that an hour's analysis that results in saving one second of computer time is an hour well spent.

SOLUTION 2.2: REARRANGEMENTS

Both of the following programs require values to be assigned to X and N for the argument of the function and the number of terms to be summed respectively. A program to add the terms from left to right in the conventional manner, generating each new term from the previous one, is:

```
100 EX = 1: TERM = 1
110 FOR I=1 TO N
120   TERM=TERM*X/I
130   EX = EX + TERM
140 NEXT I
```

A program to add the terms from right to left is:

```
200 EX = 1
210 FOR I=N TO 1 STEP -1
220   EX = EX*X/I + 1
230 NEXT I
```

The second program always gives its result more quickly than the first, despite the fact that by generating each new term from the previous one the extra work that would have been required to calculate values of the factorial function has been removed from the comparison.

SOLUTION 2.3: BINARY DISPLAY

A three-line solution of the binary display problem is given by:

```
1Ø INPUT "ENTER ADDRESS", AD
2Ø FOR K=7 TO Ø STEP -1
3Ø PRINT ( PEEK(AD) AND (2^K) )/(2^K);
4Ø NEXT K
```

Variations on this solution can be worked by using the fact that a logical expression is either true or false, and taking advantage of the particular numerical values that an implementation of BASIC uses to represent true and false. Many implementations use −1 for true and 0 for false. In these circumstances, line 30 can be replaced by:

```
4Ø PRINT ((PEEK(AD) AND (2↑K))=Ø) + 1;
```

There is even a two-line solution which, although not particularly elegant, is:

```
1Ø DEF FNB(K)= (PEEK(AD) AND (2↑K))/(2↑K)
2Ø INPUT "ENTER ADDRESS";AD
3Ø PRINT FNB(7);FNB(6);FNB(5);FNB(4);FNB(3);
   FNB(2);FNB(1);FNB(Ø)
```

SOLUTION 2.4: THE TETHERED GOAT

The following solution program finds the area that the goat can graze when the length of the tether is L by counting the squares that are both inside the circular field and within a distance, L, of where the goat is tethered. The length of the tether is then increased repeatedly by a small amount until the area that the goat can graze exceeds half that of the meadow. The solution that is computed is, for the reasons pointed out in the puzzle, an approximation, but the accuracy of the approximation can be increased by making the squares that cover the area smaller. The program is:

```
10 N=10: R=N: L=N
20 AREA=0: GRAZE=0
30 FOR J= -N TO N-1
40   FOR K= -N TO N-1
50     D=SQR( (J+0.5)^2 + (K+0.5)^2 )
60     E=SQR( (R-J-0.5)^2 + (K+0.5)^2 )
70     IF D<R THEN AREA = AREA + 1
80     IF (D<R) AND (E<L) THEN
         GRAZE = GRAZE + 1
90   NEXT K
100 NEXT J
110 IF GRAZE/AREA > 0.5 THEN 140
120 L = L + 1
130 GOTO 20
140 PRINT L/R
```

SOLUTION 2.5: NETWORKS

The values of the elements of the matrix C can be represented in four rows and four columns by:

```
100 FOR J=1 TO 4
110   FOR K=1 TO 4
120     PRINT C(J,K);
130   NEXT K
140   PRINT
150 NEXT J
```

A subroutine to compute the matrix C2 as the square of C is:

```
1000 FOR J=1 TO 4: FOR K=1 TO 4
1010   C2(J,K)=0
1020   FOR L=1 TO 4
1030     C2(J,K)=C2(J,K)+C(J,L)*C(L,K)
1040   NEXT L
1050 NEXT K: NEXT J
1060 RETURN
```

This subroutine merely implements the mathematical operation of matrix multiplication, although it requires three nested loops. When it is given the connection matrix, C, listed in the puzzle, it will give its square as:

$$C2 = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

In this matrix, as in the square of any other connection matrix, the value of C2(J,K) gives the number of links that consist of two one-way paths from node J to node K. Thus because the value in row 1 and column 4 of C2 is 1, it shows that there is precisely one link consisting of two one-way paths from node 1 to node 4. This, in fact, can be seen from Fig. 2.4 to be the link via node 2. The zero in row 3 and column 2 of C2 shows that there are no links consisting of two one-way paths from node 3 to node 2. Again, this can be confirmed from Fig. 2.4. The matrix C2 gives the numbers of all such links in the network.

SOLUTION 2.6: REVERSAL

The puzzle is solved by the matrix operation of *transposition*, i.e. interchanging each pair of elements of the matrix that is located symmetrically with respect to the diagonal running from the top left to the bottom right of the array. When writing a program to do this, the swapping operation must be applied only to the elements above the diagonal (or only to those below it). If it is applied to every element, then each pair will be swapped over twice and each element will thus finish up exactly where it started!

The transposition can be carried out by:

```
100 FOR J =1 TO 4
110   FOR K=J+1 TO 5
120     TEMP=P(J,K): P(J,K)=P(K,J)
130     P(K,J)=TEMP
140   NEXT K
150 NEXT J
```

When this is applied to the connection matrix for the network in Fig. 2.6(a) with which the matrix P was initialised in the puzzle, the result is the matrix:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

It can be confirmed from Fig. 2.6(b) that this is the connection matrix for the reversed flow situation.

Chapter Three
Graphics Puzzles

Graphics is a satisfying theme for puzzles precisely because it is visual. The context of a puzzle can be displayed, as can the attempts of a program to solve it. The addition of graphics to other puzzles can enhance their presentation considerably as well as clarifying how a program goes about solving a puzzle, but that is not quite the same thing.

The puzzles in this chapter are mainly based on *block graphics* (or *tile graphics*, as they are also known). This is because the use of block graphics to create almost any non-trivial graphics effect requires a certain amount of ingenuity. Most effects can be created, but nothing like as easily as they can with high-resolution graphics.

The first puzzle is about designing block graphics characters, or 'tiles', that will give interesting patterns. There is endless scope for experiment here when using any micro that allows you to create your own graphics characters.

The next puzzle is about displaying a square grid, such as could be composed of matchsticks, so that the effect of removing any match from the grid can subsequently be displayed. This would be fairly easy using high-resolution graphics but is much more testing and interesting with block graphics.

The third puzzle is to create various effects associated with 'windows'. A window provides a valuable technique for use in many applications. The basic principle governing its use is that a program that computes different sets of results can maintain separate windows, each for the display of a different set of results. Each can operate independently in a different part of the screen.

The fourth puzzle concerns the creation of displays that appear to be of high resolution on a low-resolution block graphics system. Apart from testing the capabilities of block graphics systems, this shows that high-resolution displays can be created without the need to call on large amounts of memory.

The final puzzle in this chapter deals with the problem of determining whether the lines joining a set of points together cross over each other. This is an important question in several areas of application. It requires the knowledge of a little geometry for its solution, but this should not be surprising as geometry is clearly the branch of mathematics that provides the theory necessary to underpin any investigations in graphics.

3.1

PATTERNS

On a block graphics system with a screen having, say, 24 rows and 32 columns, the screen can be cleared by placing a space in every screen position. This can be done with a program such as:

```
10 FOR ROW=1 TO 24
20 FOR COLUMN=1 TO 32
30 CALL HCHAR(ROW, COLUMN, 32)
40 NEXT COLUMN
50 NEXT ROW
```

This program is actually specific to the Texas Instruments TI99/4A computer, for which CALL HCHAR places in the specified screen location the character with the code given by its third parameter (32 is the ASCII code for a space). But for every kind of micro there is an equivalent way of doing the same thing, although it may often not be so direct. When the screen is cleared by using CLS, CLEAR or whatever command is provided by a particular machine for this purpose, a machine code program performs the task much more quickly. Nevertheless, the machine code program is based on the same algorithm as our program.

Programs based on this framework, but placing a specially designed character rather than a space at every position on the screen, can produce intriguing patterns. As a simple example, a square character such as the one shown in Fig. 3.1(a) will produce the square grid of Fig. 3.1(c), although an L-shaped character like the one shown in Fig. 3.1(b) provides a minimal design for creating the same effect.

The puzzle

The puzzle is to design characters which, when they are placed in every screen position, give interesting patterns that cover the screen. Computers such as the Texas Instruments TI99/4A and the Sinclair Spectrum allow users to design their own characters, so they are ideal for this sort of exercise.

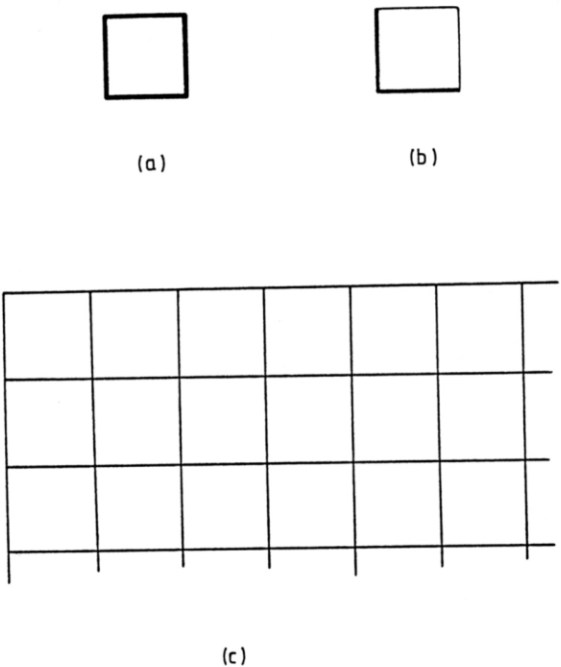


Fig. 3.1. (a) Square character. (b) L-shaped character. (c) Square grid.

Adding to the program given above the line:

```
60 GOTO 60
```

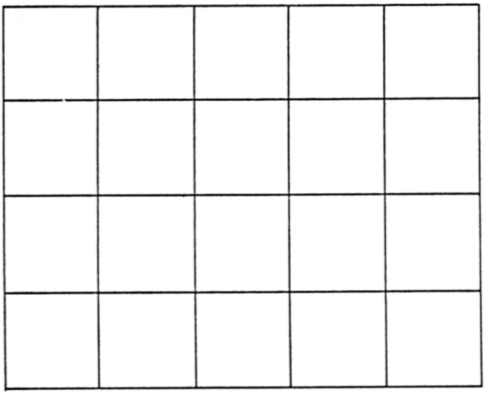
prevents the program from ever stopping, and thus from producing any messages or prompts to spoil the display.

More intricate screen patterns can be produced by using two or more specially designed characters and plotting them in some regular fashion.

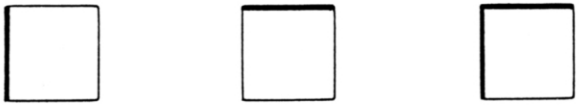
3.2

MATCHSTICKS

A rectangular grid composed of squares, such as the one in Fig. 3.2(a), provides the basis for a number of puzzles. If we regard the



(a)



(b)

Fig. 3.2. (a) Rectangular grid. (b) Graphics characters.

grid as being composed of matchsticks we can, for example, enquire how many of them are required to make up a rectangular grid with N squares horizontally and M squares vertically. We can also ask for the smallest number of matchsticks which, when removed, will break every square (that is, not just those with sides one stick long but also all those with sides two or more sticks long).

The puzzle

This puzzle is to write a program to accept values for N and M, and then to plot an array of M by N squares. The program should accept, repeatedly, the identification of a line segment making up the side of a square and corresponding to a matchstick, and then remove that line from the display. The program can then be used to investigate problems such as those described above.

Because this is comparatively straightforward using line-drawing commands on a high-resolution display, you might instead like to try it using just the graphics characters illustrated in Fig. 3.2(b) for a character-based display. Solutions are presented for both types of display. The solution for a character-based display is specifically for the Commodore 64 on which the codes for the vertical line, horizontal line, and inverted L characters are, respectively, 79, 99 and 101.

3.3

WINDOWS

'Windows' have become very popular recently as one of the ways of making computers easier to use. MicroSoft have produced software that makes good use of windows, and the Sinclair QL is also capable of maintaining them. A window is a part of the screen in which the output from, and the results of, a particular computation can appear quite independently of the rest of the screen. With the ability to maintain and manipulate more than one window at a time, a microcomputer can appear to be carrying on more than one application simultaneously. It could, for example, show the state of a word processing program in one window, the results of the sorting operations of a database program in another, and a chart produced by a graphics program in a third.

The computer will only *appear* to carry out several operations simultaneously, however. It will in fact be allocating a small ration of time to each active program in turn. In this way, at any time one process alone is active. But because the time slots are short (on a human time scale) and come round rapidly, the various operations seem to us to be taking place at the same time.

If information can readily be transferred from one window to another, a computer and its software can be made much simpler to use. For example, if the graph displayed by a graphics program can be transferred from its window to the window of a word processing program and, by so doing, can be automatically incorporated as an illustration in the document being produced there, then a complex operation can be achieved with ease.

The puzzle

This puzzle is to write a program to make your computer maintain a window on its screen in which text can be placed and automatically scrolled when necessary. A solution to this that is to be at all portable will almost certainly have to make use of the PEEK and POKE functions.

There are many other operations associated with windows that it is desirable to implement in order to make a window as flexible and usable as possible. These include scrolling it downwards and sideways, and moving it to another location.

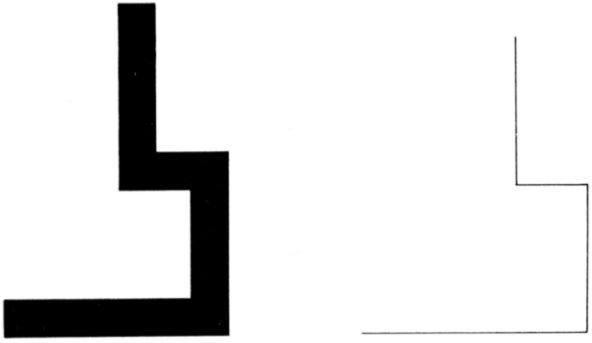
3.4

HIGH RESOLUTION FROM LOW-RESOLUTION SYSTEMS

On a low-resolution character-based system, the basic 'artist's drawing' program is similar to this:

```
1Ø ROW=13: COL=2Ø
2Ø PRINT AT (ROW, COL) CHR$(N)
3Ø GET C$: IF C$="" THEN 3Ø
4Ø IF C$="U" THEN ROW=ROW - 1: GOTO 2Ø
5Ø IF C$="D" THEN ROW=ROW + 1: GOTO 2Ø
6Ø IF C$="L" THEN COL=COL - 1: GOTO 2Ø
7Ø IF C$="R" THEN COL=COL+ 1: GOTO 2Ø
8Ø GOTO 3Ø
```

This program produces a trace made up from the character-sized block, the code for which must be stored under the name N. Pressing U extends the trace upwards, D extends it downwards, L to the left and R to the right. A typical trace from this program is shown in Fig. 3.3(a).



(a) (b)

Fig. 3.3. (a) Chunky trace. (b) High-resolution trace.

The puzzle

The puzzle is to adapt the program given here so that it produces a high-resolution, one-dot wide trace such as the one in Fig. 3.3(b) rather than the chunky one in Fig. 3.3(a). This can be done by using one of the seven characters shown in Fig. 3.4, as appropriate, to extend the trace. For instance, if the trace is proceeding horizontally and is to be further extended in the same direction, then the horizontal line character (top row, centre) will be the appropriate one to extend the trace. But if the trace is proceeding horizontally and to the left and is to be extended upwards, then the quartered L-shaped character (top row, far right) is needed to extend the trace.

The traces produced by this program will not appear as they should if they cross so you may care to refine the program further to take care of this. It will require the use of more graphics characters than are shown in Fig. 3.4 – in fact, sixteen characters in all will be needed.

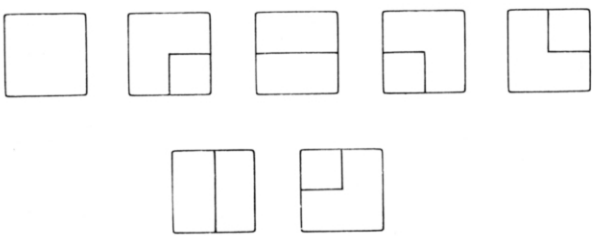


Fig. 3.4. Characters for extending the trace.

3.5

CROSSOVERS

We have already seen, in the 'Networks' and 'Reversal' puzzles of Chapter Two, that a network diagram is useful for modelling many situations. Two further applications in which it is valuable are the design of printed-circuit boards and the provision of utilities to a group of houses. In the design of printed-circuit boards, and in the design of large-scale integrated circuits, a network diagram can represent the position of the circuit elements on the board and the way in which they are connected to each other by conducting paths. In the provision of utilities such a diagram can represent the positions of the houses and the paths of the pipes carrying gas, water or electricity to them. These applications share a common problem, and that is the problem of 'crossovers'.

When designing printed-circuit boards with the aid of a computer, it is necessary to be able to record not just the positions of the circuit components on a board but also the ways in which they are connected to each other. When the connectivity of a design can be represented in the computer, a suitable program can obtain a satisfactory design for the circuit. To do this it must ensure that all the other constraints to which the design is subject are satisfied, while maintaining the connectivity of the circuit elements. It may be found necessary to change the location of a component on the board.

When the positions for the circuit components on the board are determined, the direct conducting paths that join the components in the required ways may cross each other. This is clearly undesirable as, by touching each other, the conductors will create entirely unwanted electrical paths in the network, causing it not to behave as it is supposed to. To illustrate this, Fig. 3.5 shows a simple layout representing five components connected to each other. The figure shows that if the connections follow the direct paths from one component to another, there are three crossovers. A good computer-aided design program might be able to reroute connections as shown in Fig. 3.6, maintaining the connectivity while avoiding any crossovers. In some circumstances crossovers are unavoidable, and then the circuit board manufacturer must resort to the use of a

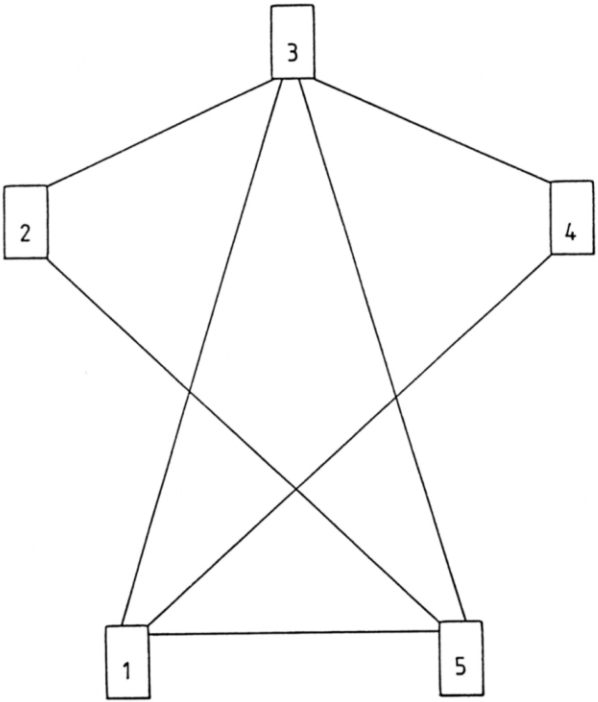


Fig. 3.5. Connections with crossovers.

double-sided board so that the connections that would cross over each other if placed on the same side of the board can be put on opposite sides of it. Naturally, the manufacture of double-sided boards is more expensive than that of single-sided ones, and is not to be undertaken if it can be avoided.

Figure 3.7 illustrates the other context in which the problem of crossovers assumes some importance. This is in providing utilities to houses. When providing gas, electricity and water, the ideal is for the various pipes and cables not to cross each other in order to reduce the risk of accidentally digging up one when dealing with another. But a brief examination of Fig. 3.7 should be enough to show that with

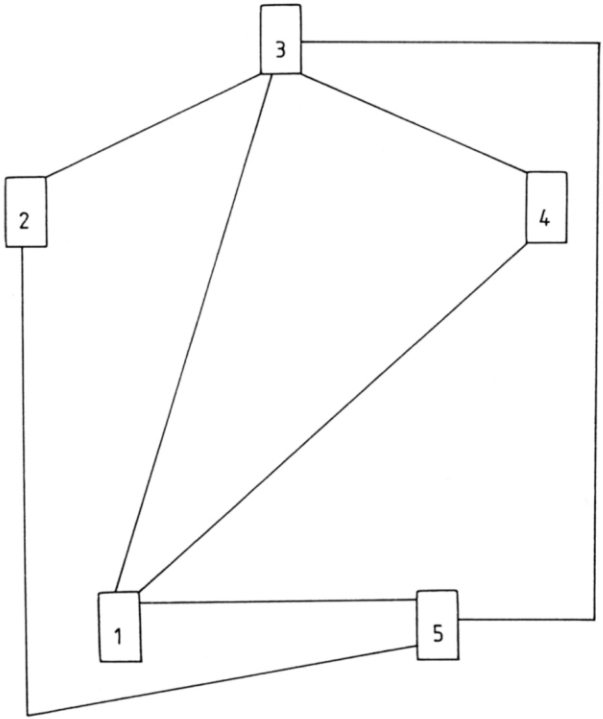


Fig. 3.6. Connections maintained, but crossovers removed.

three utilities this is not possible even if they are to be connected to only three houses!

The puzzle

When given the positions of a set of items and the connections between them (as for the five items in Fig. 3.5), the puzzle is to write a program to determine whether any of the connections cross over each other and, if so, how many. It should be assumed that the path of the

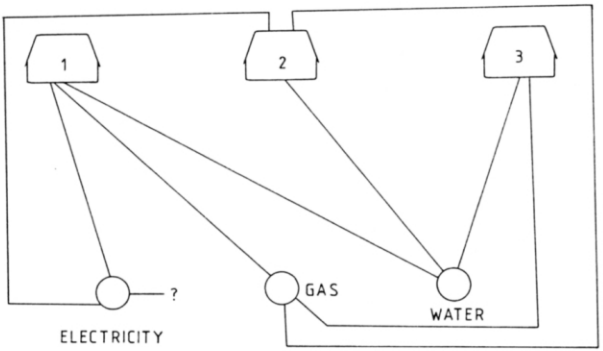


Fig. 3.7. Three utilities to three houses?

connection between any pair of points is the straight line joining them. For the situation in Fig. 3.5, for example, the program should report that there are three crossovers.

The positions of the items should be given explicitly. The connections between them can, of course, be recorded in a connection matrix.

SOLUTION 3.1: PATTERNS

It can, of course, be said that *any* pattern is interesting, and so there are very many potential solutions to this puzzle. How satisfactory any particular solution appears will depend on the aesthetic appeal it has for a particular individual. The first solutions presented here are restricted to single graphics characters which, when placed in every screen position, create a regular pattern though perhaps in an unexpected way. The cross character in Fig. 3.8(a) will fill the screen with diamonds, the line and triangle character of Fig. 3.8(b) with rows of triangles in such a way that every other triangle in every row is inverted, and the octagonal character in Fig. 3.8(c) with octagons and diamonds.



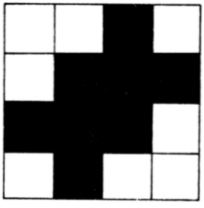
(a)



(b)



(c)

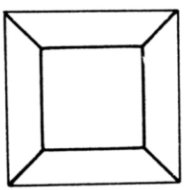


(d)

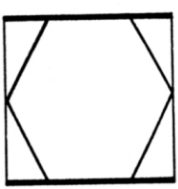
Fig. 3.8. Characters for interesting patterns.

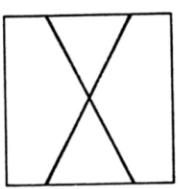
The character in Fig. 3.8(d) is particularly interesting in that it not only fills the screen with repetitions of the dark shape in the character itself, but also causes the (light) background of the display to consist of repetitions of the same shape. One way to approach finding more characters of this type is to take a repetitive pattern from any context (nature, Islamic art and wallpaper designs are all fruitful sources) and analyse it for its basic 'building block' or blocks.

Repeating two or more blocks obviously gives more scope for variety than a single one, particularly if the blocks are transformed in some way, e.g. by rotation or reflection, before they are repeated in the pattern. As two simple examples, plotting the character shown in Fig. 3.9(a) and a blank, or space, character alternately gives, rather unexpectedly, a familiar pattern, as does the alternate plotting of the two characters of Fig. 3.9(b).



(a)





(b)

Fig. 3.9. More characters.

SOLUTION 3.2: MATCHSTICKS

This character-based solution program for the 'Matchsticks' puzzle is specifically for the Commodore 64, but it may readily be adapted for other computers. The matchstick to be removed is identified by entering a three-character string starting with H or V (for horizontal or vertical) and followed by two numbers. The first gives the row containing the match and the second the position of the match in that row. Thus entering 'H23' identifies the third match in the second row of horizontally placed matches. Similarly, 'V34' identifies the fourth match in the third row of vertically placed matches. The program is:

```
  5 INPUT "COLUMNS, ROWS"; N,M
 10 DIM SQ(M+1,N+1), CODE(3)
 20 FOR R=1 TO M+1:FOR C=1 TO N+1
 30 SQ(R,C)=3
 40 IF R=M+1 THEN SQ(R,C)=SQ(R,C)-1
 50 IF C=N+1 THEN SQ(R,C)=SQ(R,C)-2
 60 NEXT C: NEXT R
 70 CODE(0)=32: CODE(1)=101
 80 CODE(2)=99: CODE(3)=79
 90 PRINT"[CLS]"
100 FOR R=1 TO M+1: FOR C=1 TO N+1
110 LOC=1230+40*R+C
115 POKE 54272+LOC,7
120 POKE LOC,CODE(SQ(R,C))
130 NEXT C: NEXT R
140 INPUT "REMOVE"; I$
150 X=VAL(MID$(I$,2,1)):Y=VAL(RIGHT$(I$,1))
160 IF LEFT$(I$,1)="V" THEN SQ(X,Y)=SQ(X,Y)-1
170 IF LEFT$(I$,1)="H" THEN SQ(X,Y)=SQ(X,Y)-2
180 GOTO 70
```

The solution is rather compact, but it does illustrate some typical techniques for manipulating graphics characters. By using the array named CODE, the codes for the characters involved in the program, i.e. for the space, vertical line, horizontal line and inverted L characters, can be mapped into the numbers 0 to 3. This also makes the program easy to adapt for any other computer, for it is only

necessary to replace the codes for the Commodore 64 with those of another machine. The manipulations can now be done with the converted codes 0 to 3, with addition as the natural operation for combining codes, because the inverted L character, which consists of a horizontal line plus a vertical line, now has a code that is the sum of those for the horizontal line character and the vertical line character.

The grid is set up in lines 20 to 60. It consists of a grid of M by N inverted Ls with vertical lines along the right of the grid and horizontal lines along the bottom. This shows that there are 2*M*N+M+N matches in the grid. The grid itself is plotted by lines 100 to 130, essentially by POKEing the codes for the characters into the screen memory. Line 115 is needed on the Commodore 64 because besides reserving one block of memory for the screen contents this computer reserves a second block for the screen's colours.

Line 140 accepts the identification of a match that is to be removed. As explained in the solution, the matches in the top row are identified from left to right by H11, H12, H13...; those in the second row by H21, H22, H23...; and so on. The matches in the left hand column are identified, from top to bottom, by V11, V21, V31...; those in the second column by V12, V22, V32, and so on. After a three-character identification has been accepted, lines 150 to 170 unpack the characters of the string and make the necessary adjustment to the converted code before going back to replot the grid with the match removed. A vertically placed match is removed by subtracting one from the current converted code, and a horizontally placed match by subtracting two.

There is no checking to ensure that only sensible instructions are given – that is to say, that only matches that are actually present in the grid are identified for removal. This checking can, however, be added by amending lines 160 and 170 as follows:

```
16Ø IF LEFT$(I$,1)="V" AND (SQ(X,Y)=1 OR SQ(X,Y)=3)
    THEN SQ(X,Y)=SQ(X,Y)-1

17Ø IF LEFT$(I$,1)="H" AND SQ(X,Y)>1 THEN
    SQ(X,Y)=SQ(X,Y)-2
```

Note that if the grid is to be larger than 9 by 9 squares, then the input string must have five characters in it rather than three so that, for example, the fifteenth match in the eleventh horizontal row can be identified by H1115. Matches such as the first one in the second row must then be identified by HØ2Ø1.

The following program supplies a solution for a high-resolution display. It is written for the Sinclair QL.

```
10 DIM i$(3)
20 INPUT "Columns, rows", m, n
30 PAPER 0: INK 7: CLS
40 FOR k=1 TO m+1
50 LINE 10*k,10 TO 10*k,10*(n+1)
60 END FOR k
70 FOR j=1 TO n+1
80 LINE 10,10*j TO 10*(m+1),10*j
90 END FOR j
100 INK 0
110 REPEAT loop
120 INPUT "Remove?", i$
130 IF i$(1)="h" THEN LINE 10*i$(3),10*i$(2) TO
    10*(i$(3)+1),10*i$(2)
140 IF i$(1)="v" THEN LINE 10*i$(3),10*i$(2) TO
    10*i$(3),10*(i$(2)+1)
150 END REPEAT loop
```

The QL can perform coercions, so that in lines 130 and 140 the numerical characters that are extracted from the input string are automatically converted to numbers before being multiplied by ten or passed to the line drawing command.

SOLUTION 3.3: WINDOWS

A program that places text entered at the keyboard in a window and scrolls the window upwards when necessary is listed below. The top left corner of the window will be in the row with its number stored under TR and the column with its number stored under TC. The window will be W columns wide and H rows high. SCRTOP holds the address of the memory location mapped to the position at the top left corner of the screen.

```
100 CR=TR: CC=TC-1
110 GET A$: IF A$="" THEN 110
120 CC=CC+1: IF CC>TC+W-1 THEN CC=TC:
    CR=CR+1
130 IF CR>TR+H-1 THEN CR=TR+H-1: GOSUB 1000
140 AD=SCRTOP+40*(CR-1)+CC-1
150 C=ASC(A$)-64: POKE AD, C
160 GOTO 110
1000 FOR R=TR+1 TO TR+H-1
1010 FOR C=TC TO TC+W-1
1020 AD=SCRTOP+40*(R-1)+C-1
1030 POKE AD-40, PEEK(AD)
1040 IF R=TR+H-1 THEN POKE AD,32
1050 NEXT C
1060 NEXT R
1070 RETURN
```

We can add extra capabilities to the window program, as suggested in the puzzle, but the question arises as to how to invoke them. The usual method is to agree to use keys that are rarely used for other purposes, and that is the approach we shall adopt here. But it must be remembered that these keys cannot be used for anything else while you are using the program. The following subroutines, starting with the line numbers 2000, 3000 and 4000 respectively, implement sideways scrolling, moving the window and reversing the foreground and background colours of the display in a window. The characters used to invoke these capabilities are, respectively, @, # and ←.

The subroutine for moving the window asks by how much the window is to be moved. Positive numbers indicate that it is to be

moved to the right and downwards. Negative numbers indicate movement to the left and upwards. This subroutine assumes that the new position for the window does not overlap the old one but the routine can be improved to accommodate this possibility by making it copy the entire window into a buffer before copying it to the new location.

The additions to the program that provide the extra capabilities are:

```
  112 IF A$="@" THEN GOSUB 2000: GOTO 110
  114 IF A$="#" THEN GOSUB 3000: GOTO 110
  116 IF A$="←" THEN GOSUB 4000: GOTO 110
 2000 C=TC
 2010 FOR R=TR TO TR+H-1
 2020 AD=SCRTOP+40*(R-1)+C-1
 2030 B(R-TR+1)=PEEK(AD)
 2040 NEXT R
 2050 FOR C=TC+1 TO TC+W-1
 2060 FOR R=TR TO TR+H-1
 2070 AD=SCRTOP+40*(R-1)+C-1
 2080 POKE AD-1,PEEK(AD)
 2090 NEXT R: NEXT C
 2100 C=TC+W-1
 2110 FOR R=TR TO TR+H-1
 2120 AD=SCRTOP+40*(R-1)+C-1
 2130 POKE AD, B(R-TR+1)
 2140 NEXT R
 2150 RETURN
 3000 INPUT "MOVE BY?(R AND C)"; NR,NC
 3010 M=40*NR+NC
 3020 FOR R=TR TO TR+H-1
 3030 FOR C=TC TO TC+W-1
 3040 AD=SCRTOP+40*(R-1)+C-1
 3050 POKE AD+M,PEEK(AD)
 3060 NEXT C: NEXT R
 3070 RETURN
 4000 FOR R=TR TO TR+H-1
 4010 FOR C=TC TO TC+W-1
 4020 AD=SCRTOP+40*(R-1)+C-1
 4030 IF PEEK(AD)<128 THEN POKE AD,PEEK(AD)+128:
      GOTO 4050
 4040 POKE AD, PEEK(AD)-128
```

```
4Ø5Ø NEXT C
4Ø6Ø NEXT R
4Ø7Ø RETURN
```

SOLUTION 3.4: HIGH RESOLUTION FROM LOW-RESOLUTION SYSTEMS

By giving the characters a logical code based on the numbers 1, 2, 4 and 8, as shown in Fig. 3.10, and using the array named C to convert from logical codes to the actual codes used by a particular computer,

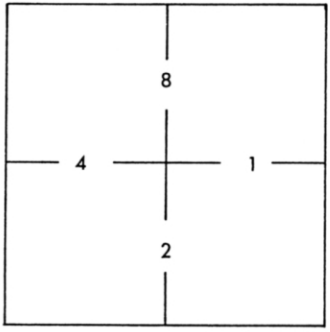


Fig. 3.10. Scheme for logical code.

the following program gives a high-resolution trace on a character-based screen. It requires a DATA statement giving in pairs each logical code, LC, with its corresponding actual code, AC. The program is:

```
 5 DIM C(15)
1Ø ROW=13: COL=2Ø
15 FOR K=1 TO 7
16 READ LC, AC: C(LC)=AC
17 NEXT K
18 N=5: DIR=1
2Ø PRINT AT (ROW, COL) CHR$(C(N))
3Ø GET C$: IF C$="" THEN 3Ø
4Ø IF DIR=1 THEN COL=COL+1: N=4
5Ø IF DIR=2 THEN ROW=ROW+1: N=8
```

```
60 IF DIR=4 THEN COL=COL-1: N=1
70 IF DIR=8 THEN ROW=ROW-1: N=2
80 IF C$="U" THEN DIR=8: GOTO 130
90 IF C$="D" THEN DIR=2: GOTO 130
100 IF C$="L" THEN DIR=4: GOTO 130
110 IF C$="R" THEN DIR=1: GOTO 130
120 GOTO 30
130 IF N=DIR THEN N=0: GOTO 20
140 N=N+DIR: GOTO 20
```

The required DATA statement for the Commodore 64 is:

```
DATA 0,32,3,112,5,64,6,110,9,109,10,93,12,125
```

(Note that the Commodore 64 does not have a PRINT AT statement, although it is possible to simulate such a statement.)

The DATA statement for any other computer is obtained by replacing the alternate numbers (starting from the second) in this DATA statement, with the codes for the graphics characters shown in Fig. 3.4.

To make the program produce a trace that will retain its proper appearance when it crosses itself, it is necessary to make the program 'look ahead' to the position at which it is to add a new character to extend the trace. If it finds a character there already, it must plot a character containing the lines in the existing character plus any extra ones that may be needed to superimpose on it the character that is to extend the trace. As PEEK will be needed to examine the next position, it may be more sensible to use POKE rather than PRINT AT in line 20 of the solution program, particularly as the address in screen memory must be calculated for PEEK anyway.

Figure 3.11 shows a few examples of what is required when different symbols are encountered at the position where a vertical line character is to be plotted to extend the trace. In all the illustrated cases except one, plotting the vertical line and not a composite character will destroy a part of the existing trace. The composite character that should be plotted to prevent this must contain the lines that are in either the existing character or the character that extends the trace. Because a binary code has been used for the line segments making up each character, the code for the composite character can be computed by performing the logical operation OR on the codes for the two characters to be superimposed. To illustrate why this is so, we can observe that 12 OR 10 gives 14 because, after conversion to binary, this becomes 1100 OR 1010 which gives 1110. Similarly, the

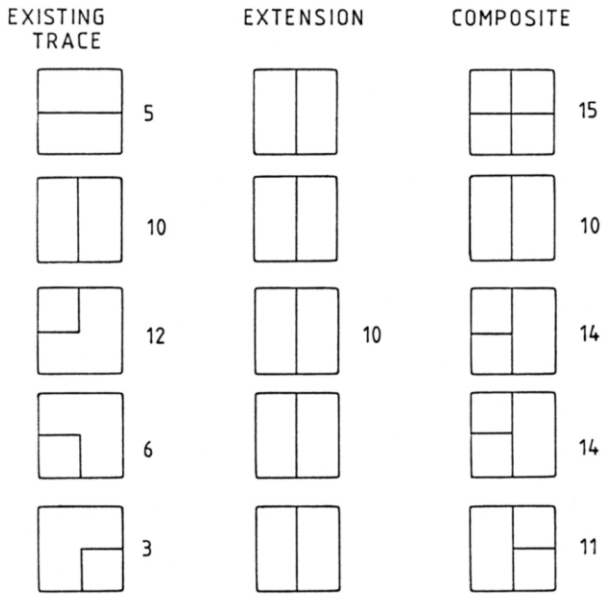


Fig. 3.11. Composite characters and codes for various combinations of existing characters and a vertical character.

character composed of the lines that are in the character with code 12 or the character with code 10 is the one with code 14.

A program for this is listed in full on pages 94 and 95 of the book *Computer Languages and Their Uses* (cited in the appendix).

SOLUTION 3.5: CROSSOVERS

A solution to the crossovers problem that draws on a little two-dimensional geometry is listed below. After reading the positions of the five elements and their connection matrix, it takes each distinct pair of elements and tests whether the line that joins them crosses the line joining any other pair of elements. If it does, a crossover is counted. But this exhaustive procedure causes every crossover to be counted twice, for if the program finds that line A crosses line B, then it will also find that line B crosses line A. The number of crossovers counted by the program is therefore halved in the program's last line so that the true number of crossovers can be printed.

The procedure for finding if the lines cross depends, in essence, on the idea that if the two points with coordinates (X1, Y1) and (X2, Y2) lie on a line, then so does the point with coordinates

((1−MU)*X1+MU*X2, (1−MU)*Y1+MU*Y2)

No matter what value is assigned to MU, the resulting coordinates will be those of a point on the line; but only values between zero and one will give the coordinates of a point that lies between the two original points.

The program is:

```
10 DIM X(5),Y(5),C(5,5)
20 FOR J=1 TO 5
25 READ X(J),Y(J)
30 FOR K=1 TO 5: READ C(J,K): NEXT K
40 NEXT J
50 DATA 1,0,0,0,1,1,1
60 DATA 0,1,0,0,1,0,1
70 DATA 2,5,1,1,0,1,1
80 DATA 5,3,1,0,1,0,0
90 DATA 3,0,1,1,1,0,0
100 CRO=0
110 FOR J=1 TO 4
120 FOR K=J+1 TO 5
130 IF C(J,K)=0 THEN 250
140 X1=X(J):X2=X(K):Y1=Y(J):Y2=Y(K)
```

```
150 FOR L=1 TO 4
160 FOR M=L+1 TO 5
170 IF J=L AND K=M THEN 230
180 IF C(L,M)=0 THEN 230
190 X3=X(L):X4=X(M):Y3=Y(L):Y4=Y(M)
200 DEL=(X1-X2)*(Y4-Y3)-(Y1-Y2)*(X4-X3)
210 MU=((Y4-Y3)*(X4-X2)-(X4-X3)*(Y4-Y2))/DEL
220 IF MU>0 AND MU<1 THEN CRO=CRO+1
230 NEXT M
240 NEXT L
250 NEXT K
260 NEXT J
270 CRO=CRO/2: PRINT CRO;" CROSSOVERS"
```

Chapter Four

Puzzles Involving Programming Techniques

The puzzles in this chapter have the common theme of programming techniques. Some either involve or require for their solution techniques that are readily available in languages other than BASIC, but which must be implemented from scratch by the BASIC programmer. To illustrate this, the first problem involves the creation of a linked list of items in which each item carries with it a pointer to the next item in the list. The list is supported naturally in Lisp and Logo, while Pascal provides pointer variables with which lists can readily be created. The creation of linked lists in BASIC requires some fairly tortuous manipulations with arrays. As a further example, another of the puzzles involves *recursion*. Some languages, including Pascal and Lisp, support recursion while BASIC, in general, does not. But recursion can be programmed in BASIC as long as the programmer is prepared to do all the necessary work behind the scenes.

Once the idea of a pointer is mastered it is possible to create tree structures in which each item can carry pointers to two or more successors, thereby creating a branching pattern, and association lists, in which items carry pointers to various properties that are associated with them. There are puzzles here involving associations and trees.

The final puzzle in this chapter is about coding, and the procedures involved can be expressed in BASIC as naturally as in any other language, particularly in the ways that they involve the logical manipulation of ASCII codes and the conversion between codes and characters with the functions ASC and CHR$.

One common aspect of the first three puzzles in this chapter is that they illustrate how the use of pointers makes possible the construction of data structures much better suited to the handling of the structures in the puzzles than the storage structure provided by BASIC. Lists, trees and association networks are all common and

naturally occurring data structures. If data possesses some natural structure, then it is obvious that the structure should be preserved when the data is stored in the computer. We have had lists occurring in other puzzles – as lists of instructions and, in slightly disguised form, as an ordered set. Any branching structure is a tree, and networks have cropped up with some regularity. Therefore the ability to handle pointers, even in BASIC, provides a powerful tool for use in many situations.

Recursion can also be a valuable tool, although if used indiscriminately it has drawbacks. It allows repetition by a form of self-reference, as is illustrated by the usual example of the factorial function, which can be defined recursively as:

factorial(N)=IF N=∅ THEN 1 ELSE N×factorial(N−1)

This is recursive because it defines a factorial in terms of a factorial; that is, in terms of itself. This can seem circular but there is a stopping condition, in the form of a value for factorial(0), which prevents the definition from consulting itself for ever. This gives the clue to one drawback of recursion, for in calculating recursively the value of a factorial, or of anything else, the computation must spiral in until it reaches the stopping condition, storing all the intermediate requirements along the way. Then it spirals out again, unfolding all the intermediate calculations that lead to the final result. In this way recursion can demand large amounts of storage to hold all the intermediate requirements, and can prove an expensive means of supporting an elegant description of a computation.

A final point worth making here is that a programmer who needs to call on the use of pointers, recursion and, indeed, any other feature that is not implemented in BASIC, would be well advised to investigate the possibility of using some other language in which the feature in question *is* provided.

4.1

CHAINING

Chaining is a technique used in many computer programs that deal with lists of data items. A word processing program deals in effect

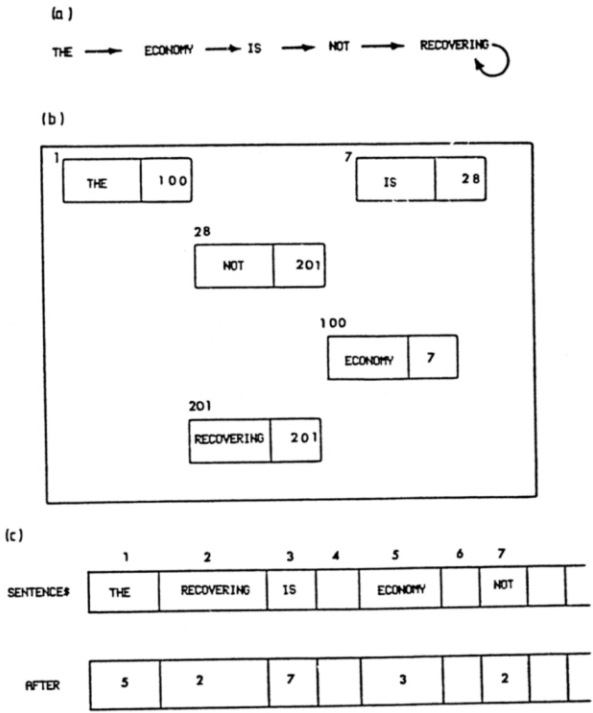


Fig. 4.1. A sentence as a chain of words.

with lists of words, and a program to help a robot to explore its environment would maintain a list of the obstacles that the robot might encounter so that it could check if a proposed movement would cause it to bump into one. Either of these programs could make use of chaining.

Figure 4.1(a) shows a sentence represented as a chain of words. This chain can be stored in the computer's memory by placing a word in any memory location and storing with it a pointer to its successor in the chain. The last word can point to itself as a conventional way of indicating that it is at the end of the chain.

The point of storing a list in this way is that items can be added to the list and deleted from it with a minimum of difficulty. Thus to delete the word 'NOT' from the sentence it is only necessary to change the address with the word 'IS' from 28 to 201 (see Fig. 4.1(b)). To add 'SLOWLY' to the end of the resulting sentence, store this word anywhere, with a pointer to itself as it is at the end of the sentence, and change the pointer with 'RECOVERING' so that it points to 'SLOWLY'.

The puzzle

When the sentence is stored in a BASIC program in two arrays, as in Fig. 4.1(c), the puzzle is to implement a system in BASIC for printing the sentence and for adding words to and deleting them from the chain. When deleting a word it is, of course, only necessary to give the word, but when adding a word the position that it is to occupy must be specified as well as the word itself.

4.2

CLASSIFICATION

A tree such as the one in Fig. 4.2 can be used for classification. The system by which library books acquire their classification numbers is essentially of this form. From the classification tree in the figure, Mathematics would acquire the classification number 100 and

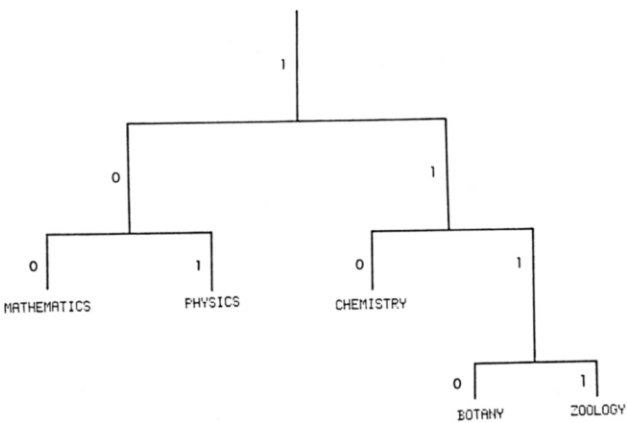


Fig. 4.2. Classification tree.

Zoology the number 1111. In each case the number is read by starting at the 'root' of the tree and moving through the tree to the 'leaf' labelled with the item to be classified, collecting the digits of the number along the way.

The puzzle

For this puzzle, write a BASIC program to store information organised with a tree structure such as that shown in the illustration.

It will be necessary to start at the root of the tree and give it two pointers. Each pointer must point either to an item that is to be classified, or to another branching point in the tree. All the other branching points of the tree must be treated in the same way. A pointer can be implemented in BASIC as described in the last puzzle, 'Chaining'. When the tree has been represented, the program should be extended so that it can accept one of the items labelling a 'leaf' and automatically read its classification number from the tree.

4.3

ASSOCIATIONS

To record information about the situation in Fig. 4.3, we could begin by storing the names of all the objects appearing there in an array, thus:

```
A$(1)="SPHERE"
A$(2)="LARGE CUBE"
A$(3)="CYLINDER"
```

and so on. When the names of all the objects are stored in the array we can represent relations between them quite simply. For example,

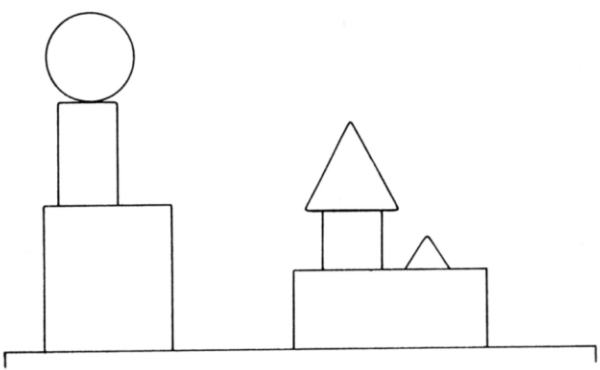


Fig. 4.3. An arrangement of objects.

we could represent the fact that the sphere is on the cylinder by using an array named ISON and writing

```
ISON(1)=3
```

The meaning that we have assigned to this statement is: 'the item stored in element 1 of the array A$ is on the item stored in element 3 of A$.' Similarly:

ISON(3)=2

represents the relation that the cylinder is on the large cube. We could record which item is to the left of another in similar fashion by using an array named, say, ATLEFT.

The puzzle

For this puzzle, write a program to record which objects in Fig. 4.3 are on which others, in the way outlined above. Then develop the program further so that it can respond correctly to enquiries about the illustrated situation, such as: 'Is the small pyramid on the large cube?' Remember that if object A is on object B, and object B is on object C, then we should expect the program to be able to tell us that object A is on object C. This means that if we enter the enquiry 'Is the sphere on the large cube?' then the program should be able to respond in the affirmative.

4.4

RECURSION

Recursion is a method of producing forms of repetition in which the result of any repetition depends on the outcome of the previous one. Computer languages such as Lisp and Logo support recursion. In general, BASIC does not although some dialects, such as that of the BBC Micro, do.

We can give a simple example of recursion by writing a recursive function to count the items in a list. The procedure to be carried out by the function can be expressed in ordinary English as: 'The number of items in a list is one (for the first item in the list) plus the number of items in the list that is obtained by removing the first item from the list.' This procedure is recursive because it gives the number of items in a list in terms of the number of items in a list – that is, in terms of itself. It is also not quite complete because it does not work for an empty list (a list where the number of items is zero). To cover this, and to provide a stopping condition for the recursion, we add to the procedure the statement: 'If a list is empty, then it has zero items.'

Before writing this in a computer language, it is useful to introduce a little notation so as to be able to write it fairly compactly. The notation is more or less that of Logo. A list is written by enclosing its items in square brackets, thus:

```
[ ALPHA BETA GAMMA DELTA ]
```

A list can be given a name, so let us call this one L.

Applying the function FIRST to a list gives its first item, so that FIRST(L) is ALPHA. Applying the function BUTFIRST to a list gives the list that results when the first item of the original list is removed. This means that BUTFIRST(L) gives the list [BETA GAMMA DELTA]. In similar manner the function LAST gives the last item of a list, making LAST(L) DELTA, and the function BUTLAST gives a list with its last item removed, making BUTLAST(L) the list [ALPHA BETA GAMMA].

With this notation and using these functions, a recursive function to count the items of a list is:

```
COUNT(L)
  IF EMPTY(L) THEN ∅
  ELSE ADD 1 TO COUNT(BUTFIRST(L))
```

A similar recursive function which expects a list of numbers, N, as its input is given below for you to determine what it does:

```
WHAT(N)
  IF EMPTY(N) THEN ∅
  ELSE ADD FIRST(N) TO WHAT(BUTFIRST(N))
```

The puzzle

The puzzle is to write a recursive function which, when given a list, produces a second list containing the items of the input list but in reverse order. Any of the functions introduced above may be used. In addition you may use a further function called, say, LIST which can be given an item and a list and which produces a list by adding the item it is given to the beginning of the list it is given. To illustrate the use of this new function, LIST(FIRST(L), BUTFIRST(L)) gives the list L.

4.5

CODING

It is possible to code messages, for purposes of secrecy and security, with BASIC programs. The complexity of the program will obviously depend upon the coding scheme. Since most microcomputer dialects of BASIC possess a function that gives the ASCII code for a character and many microcomputers use this code for representing characters internally, it is natural to incorporate the ASCII code into any coding method.

The following BASIC program accepts a message and converts it, character by character, to its coded form before printing it:

```
10 PRINT "ENTER MESSAGE"
20 INPUT M$
30 CM$=""
40 FOR K=1 TO LEN(M$)
50   X=ASC( MID$(M$,K,1) )
60   FOR I=1 TO 3
70     X=2*X
80     IF X>127 THEN X = (X AND 127) + 1
90   NEXT I
100  CM$ = CM$ + CHR$(X)
110 NEXT K
120 PRINT CM$
```

When given the message 'MIX WITH MILK', for example, the program codes it as ,LE=L%D,L$\.

The puzzle

The puzzle is to write a BASIC program to decode the coded message, and recover the original clear text. That is, when presented with the character string assigned to CM$ by the coding program it should reconstruct the character string originally presented to the coding program.

SOLUTION 4.1: CHAINING

The chaining is achieved by using an array named AFTER which records the location of the word following each word in the sentence. The assignment AFTER(1)=5 records that after the word in SENTENCE$(1) comes the word stored in SENTENCE$(5). The array can be seen as an array of pointers. AFTER(1) carries the pointer for SENTENCE$(1) just as each element of AFTER holds the pointer for the element of SENTENCE$ with the same index. The contents of AFTER(1) are 5 and so it points to SENTENCE$(5).

The chain of words can be printed by:

```
1Ø P=1
2Ø PRINT SENTENCE$(P);
3Ø IF P=AFTER(P) THEN PRINT ".": STOP
4Ø PRINT " ";
5Ø P=AFTER(P)
6Ø GOTO 2Ø
```

A word can be deleted with the following simple program, unless it is the first or the last word in the chain. There are particular problems associated with these cases because of the conventions that have been adopted for indicating the beginning and end of a chain.

```
1ØØ PRINT "ENTER WORD TO BE DELETED"
11Ø INPUT WORD$
12Ø P=1
13Ø IF SENTENCE$(AFTER(P))=WORD$ THEN 16Ø
14Ø P=AFTER(P)
15Ø GOTO 13Ø
16Ø AFTER(P)=AFTER(AFTER(P))
17Ø GOTO 1Ø: REM TO PRINT NEW SENTENCE
```

The next program provides a much more sophisticated implementation of the deletion operation for a chain of words. It allows any word in a sentence to be deleted, including the first and the last. It also allows a word to be inserted. The program is:

```
1Ø DIM SENTENCE$(10), AFTER(1Ø)
2Ø FOR K=1 TO 7
```

```
 3Ø READ SENTENCE$(K), AFTER(K)
 4Ø NEXT K
 5Ø DATA "THE",5,"RECOVERING",2,"IS",7
 6Ø DATA "",Ø,"ECONOMY",3,"",Ø,"NOT",2
1ØØ START=1
11Ø P=START
12Ø PRINT SENTENCE$(P);
13Ø IF P=AFTER(P) THEN PRINT".": GOTO 2ØØ
14Ø PRINT " ";
15Ø P=AFTER(P)
16Ø GOTO 12Ø
2ØØ PRINT "ENTER I TO INSERT, D TO DELETE"
21Ø GET A$: IF A$="" THEN 21Ø
22Ø IF A$="I" THEN 3ØØ
23Ø IF A$="D" THEN 4ØØ
24Ø GOTO 21Ø
3ØØ PRINT "ENTER NEW WORD": INPUT WORD$
31Ø PRINT "ENTER WORD IT IS TO FOLLOW":INPUT P$
32Ø K=1
33Ø IF SENTENCE$(K)="" THEN 35Ø
34Ø K=K+1: GOTO 33Ø
35Ø SENTENCE$(K)=WORD$
36Ø P=START
37Ø IF SENTENCE$(P)=P$ THEN 39Ø
38Ø P=AFTER(P): GOTO 37Ø
39Ø AFTER(K)=AFTER(P): AFTER(P)=K: GOTO 11Ø
4ØØ PRINT "ENTER WORD TO BE DELETED"
41Ø INPUT WORD$
42Ø P=START
425 IF SENTENCE$(P)=WORD$ THEN
    START=AFTER(START): GOTO 11Ø
43Ø IF SENTENCE$(AFTER(P))=WORD$ THEN 455
44Ø P=AFTER(P)
45Ø GOTO 43Ø
455 IF AFTER(P)=AFTER(AFTER(P)) THEN
    AFTER(P)=P: GOTO 11Ø
46Ø AFTER(P)=AFTER(AFTER(P))
47Ø GOTO 11Ø
```

SOLUTION 4.2: CLASSIFICATION

A tree is, naturally enough, just a branching structure. The tree in Fig. 4.2 develops exactly two branches at each of its branching points, providing an example of what is known as a *binary tree*. Thus we can represent the tree by using two arrays of pointers of exactly the same kind as the array AFTER in the chaining puzzle. It is convenient to refer to the branches in each pair as a left branch and a right branch. (We shall use 'left' and 'right' from the point of view of a person facing the tree drawn in Fig. 4.2.) This gives us the names LTREE and RTREE for the two arrays holding the pointers.

At the first branching point from the root of the tree, the branches are represented by

LTREE(1)=2 and RTREE(1)=3

This indicates that LTREE(2) and RTREE(2) point to the branches from the next branching point on the left, and that LTREE(3) and RTREE(3) point to the branches from the next branching point on the right. All the branches are pointed to in this way. We must not forget, however, that although a branching point is usually followed by another branching point it may also be followed by a 'leaf'; that is, by the end of a branch. In the latter case, LTREE and RTREE hold pointers to the appropriate element of the array SUBJECT$, but with a hundred added to the position of the element in the array in order to distinguish a pointer to a branch from a pointer to a leaf. With this convention, RTREE(2)=102 means that moving to the right from branching point number 2 leads to element 2 of SUBJECT$, that is, to PHYSICS.

In the solution program, lines 10 to 80 store a representation of the tree, line 90 accepts a subject to be classified from the labelling on the tree, and lines 100 to 120 identify the position of this subject in the array SUBJECT$. Then line 130 computes the value for the pointer to this subject, as used by LTREE and RTREE, and stores it under K. It also initialises the variables CLASS, P and J used in the procedure of lines 140 to 180 for finding the classification. The variable named CLASS is used to accumulate the classification itself, P holds a number to determine the weighting associated with the

labelling on the tree, and J holds the position of the pointer to the current leaf or branch.

The solution program is:

```
10 DIM SUBJECT$(100), LTREE(100), RTREE(100)
20 LTREE(1)=2: RTREE(1)=3
30 LTREE(2)=101: RTREE(2)=102
40 LTREE(3)=103: RTREE(3)=4
50 LTREE(4)=104: RTREE(4)=105
60 SUBJECT$(1)="MATHS": SUBJECT$(2)="PHYSICS"
70 SUBJECT$(3)="CHEMISTRY"
80 SUBJECT$(4)="BOTANY":
   SUBJECT$(5)="ZOOLOGY"
90 INPUT "SUBJECT"; T$
100 K=1
110 IF SUBJECT$(K)=T$ THEN 130
120 K=K+1: GOTO 110
130 K=100+K: CLASS=0: P=0: J=1
140 IF LTREE(J)=K THEN P=P+1: K=J: J=0: GOTO 160
150 IF RTREE(J)=K THEN
   CLASS=CLASS+10↑P:P=P+1: K=J: J=0
160 IF K=1 THEN CLASS=CLASS+10↑P: GOTO 180
170 J=J+1: GOTO 140
180 PRINT "CLASSIFICATION IS", CLASS
```

The classification procedure is far from transparent when written in BASIC, but it operates by tracing the path through the tree from the leaf representing the subject to be classified to the root, accumulating the classification from the numbers labelling the path as it encounters them. The first number back from the leaf is treated as a number of units, the next as a number of tens, the next as a number of hundreds, and so on, thereby taking advantage of the positional nature of the number system. Since the labelling always associates a one with a right branch and a zero with a left branch, it is only necessary to add a contribution to the classification when there is a right branch in the path. But the variable P that is used to hold the power of ten to be used for the weighting must be incremented in either case.

The general idea of the procedure may be expressed as follows:

```
Find value of pointer to subject (K)
Set power of 10 for weighting to zero (P)
While not at the root of the tree
```

```
        Find J such that either LTREE(J)=K or RTREE(J)=K
        Add 10↑P to classification only if RTREE
        Increment P
        Set K to J
    End while
    Print classification
```

The routine is actually finding a path through the tree in a direction opposite to that indicated by the pointers. In essence, it is exactly opposite to the process of repeating P=AFTER(P) which follows a chain in the direction of the pointers.

On the general topic of trees, tracing paths through them and allocating binary codes to items, there is another method of doing this that is used in practice. This is the Huffman algorithm, and it has the advantage over the method just described of giving an optimum code for the items being coded. (Despite its use of powers of ten, our program is finding a binary code from the tree. The powers of ten are used so that the number that is printed has the appearance of a binary number despite the fact that it is decimal. By using powers of two instead of ten the binary number can be found, although BASIC will print it in its decimal form!) The Huffman algorithm is optimum in the sense that it gives a coding to a set of items that uses the least binary digits in total, so that the minimum amount of memory will be needed to store the coding. The classification codes would be as short as possible if the Huffman algorithm were used to determine them.

We can illustrate the algorithm by using our five subjects as the items to be coded. We could use codes of the same length for each item in this way:

```
MATHEMATICS - 011
PHYSICS     - 001
CHEMISTRY   - 101
BOTANY      - 100
ZOOLOGY     - 110
```

While such a code may have its advantages, the Huffman algorithm gives a shorter one. As Fig. 4.4 shows, by combining items in pairs repeatedly until all of them have been merged and then labelling the tree that results at each branching point we can read a code for each subject. Each code is read from right to left, starting from the root of the tree and ending at the item in question. This gives the codes as:

```
MATHEMATICS - 00
PHYSICS     - 01
```

CHEMISTRY – 10
BOTANY – 110
ZOOLOGY – 111

This requires three binary digits less than the previous assignment and, at three out of fifteen, gives a 20% advantage in memory usage.

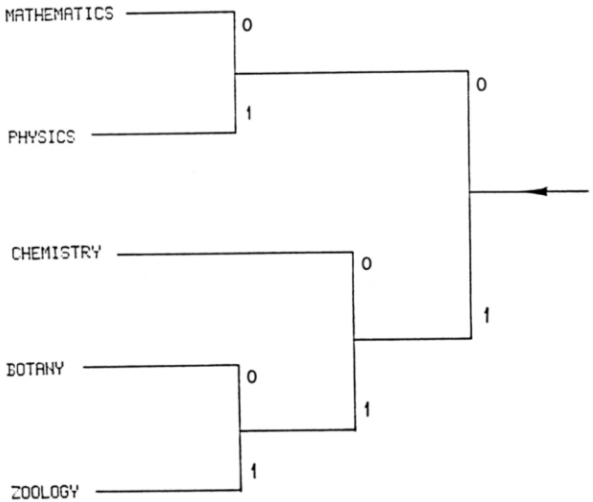


Fig. 4.4. Huffman coding.

SOLUTION 4.3: ASSOCIATIONS

In this puzzle the illustrated situation is represented by using the array ISON to hold pointers that associate each item with the item that it is on. As long as each item stands directly on not more than one other we can use a one-dimensional array to hold the pointers, and the resulting association network takes the form of a tree. (In terms of networks, a tree is a network with no loops in it.) However, if an item stands on more than one other – as would happen with an arch, for example, where the lintel rests on each of a pair of uprights – a more complex form of representation would be needed. Without it we would need to have ISON(LINTEL)=UPRIGHT1 and ISON (LINTEL)=UPRIGHT2 holding true at the same time, which would clearly be impossible. The increased complexity of the situation is reflected by the fact that the association network is no longer a tree.

The understanding of queries by the program is quite limited but is based on the generally applicable idea that a question put to the program should have the form 'Is the X on the Y?' Given this grammatical template, the program tries to match an X and a Y from its repertoire of objects. In this case it is arguable that the program 'understands' questions of this particular form. It can certainly answer them correctly, which may be as stringent a test of understanding as one can imagine! Its understanding is, at the least, an advance on the 'replace and rearrange' technique as used by ELIZA. Given a suitable set of grammatical templates, the program can display the same level of understanding over a wider range of natural language statements.

The solution program is:

```
 10 DIM A$(8), ISON(7)
 20 A$(1)="LARGE CUBE": A$(2)="CYLINDER"
 30 A$(3)="SPHERE": A$(4)="BOX"
 40 A$(5)="SMALL CUBE": A$(6)="SMALL PYRAMID"
 50 A$(7)="LARGE PYRAMID": A$(8)="TABLE"
 60 ISON(1)=8: ISON(2)=1: ISON(3)=2
 70 ISON(4)=8: ISON(5)=4: ISON(6)=4: ISON(7)=5
100 INPUT "QUERY"; Q$
110 X=0: Y=0
```

```
12Ø FOR J=1 TO 8: FOR K=1 TO 8
13Ø IF Q$="IS THE "+A$(J)+" ON THE "+A$(K)+"?"
    THEN X=J: Y=K
14Ø NEXT K: NEXT J
15Ø IF X=Ø OR Y=Ø THEN PRINT"NO": STOP
16Ø IF X=8 THEN PRINT"NO": STOP
17Ø IF ISON(X)=Y THEN PRINT "YES": STOP
18Ø X=ISON(X)
19Ø GOTO 16Ø
```

The way in which the program actually decides if one object is on another, whether directly on it or not, is recursive and can be expressed as:

> X is supported by Y if ISON(X)=Y or ISON(X)=Z and Z is supported by Y

Again, the definition of 'is supported by' is recursive because it includes a reference to itself.

This puzzle was inspired by Terry Winograd's creation of and investigations into his 'blocks world' as a means for exploring the automatic understanding of natural language. The blocks world provides another example of the success of artificial intelligence investigations in a restricted world – this time a specially created one – rather than in the world in general. To represent all the knowledge of a restricted world that is needed for experiments to be carried out is possible, whereas to do the same thing for the real world would be a task on an altogether different scale.

Interesting ways of extending the program presented here include giving colours to the objects so that an object may be referred to in different ways, either as the 'small block' or as the 'red block', for example. Further queries to which the program could be made to respond include those of the form 'Where is the X?' It is also interesting to make the program respond to commands to move objects, such as: 'Place the small block on the cylinder.' This is far from easy, though, for it will often involve removing objects from on top of the subject of the move (unless you wish those to be moved as well) and clearing others from the top of the destination.

SOLUTION 4.4: RECURSION

A list can be reversed with the recursive function:

```
REVERSE(L)
  IF EMPTY(L) THEN L
  ELSE LIST( LAST(L), REVERSE(BUTLAST(L) )
```

The use of pointers to create lists leads naturally to recursion as a means of handling the items of a list. The obvious way to process a list becomes to deal with the first item of the list and then process the rest of the list. The procedures for counting the items of a list and for adding the items of a numeric list work in just this way. As a further example, consider writing a program for printing the elements of a list based on the recursive scheme expressed by 'as long as the list is not empty, print a list by printing the first item and then applying this procedure for printing a list to the rest of the list'. A procedure based on this and written in the style of those in the puzzle is:

```
PRINTLIST(LIST)
  IF EMPTY(LIST) THEN STOP
  ELSE PRINT FIRST(LIST);
  PRINTLIST(BUTFIRST(LIST))
```

This could also be written as:

```
PRINTLIST(LIST)
  IF EMPTY(LIST) THEN STOP
  ELSE PRINT FIRST(LIST); MAKE LIST
  BUTFIRST(LIST);
  PRINTLIST(LIST)
```

When the procedure is written in the second form it becomes straightforward to show the relation of this form of recursion to the usual style of repetition. Changing the recursive call at the end of the procedure to a GOTO creates a loop. It is of the form that is repeatedly executed while a condition remains true. The non-recursive version of the procedure is:

```
PRINTLIST(LIST)
  LOOP: IF EMPTY(LIST) THEN STOP
```

```
ELSE PRINT FIRST(LIST);MAKE LIST
BUTFIRST(LIST);
GOTO LOOP
```

This demonstrates that the form of recursion known as 'tail recursion' is directly equivalent to a much more familiar form of repetition. The equivalence for other forms of recursion can be a good deal less obvious, however.

SOLUTION 4.5: CODING

The coding method takes the ASCII code, treats it as a seven-bit number and rotates all the bits three places to the left in circular fashion. If the decoder shifts the bits four more places to the left they will be in their original positions. The decoding program is:

```
2Ø Ø C$=""
21Ø FOR K=1 TO LEN(CM$)
22Ø   X=ASC( MID$(CM$,K,1) )
23Ø   FOR I=1 TO 4
24Ø     X=2*X
25Ø     IF X>127 THEN X = (X AND 127) + 1
26Ø   NEXT I
27Ø   C$=C$ + CHR$(X)
28Ø NEXT K
29Ø PRINT C$
```

When this puzzle was first published, I had a fairly aggressive letter from a reader claiming that the instruction X=(X AND 127) +1 made no sort of sense to him. The next week he wrote me a second letter! He used a Sinclair Spectrum, which helps to explain his difficulty (see the 'Binary display' puzzle in Chapter Two), but then, with unconscious perversity, proceeded to suggest that I write my puzzle solutions in Sinclair BASIC!

In fact, the seven-bit rotation in the puzzle and its solution is achieved by first doubling X, in line 70 of the program, to shift each bit to the left. If this makes bit 7 a 1, then the resulting number is at least 128. Performing the AND operation with 127 masks off bit 7, while adding one to the result of the AND operation completes the seven-bit rotation. Of course, if the doubling places a zero in bit 7 there is no need to do anything as the doubling, by producing an even result, has already placed a zero in bit 0.

As an example, the code for the letter I, obtained by ASC("I"), is 73, which is 1001001 in binary. Doubling it gives 146 or 10010010. As this is greater than 127, ANDing it with 127 gives 0010010, or 18, and adding one produces 19 or 0010011, which is a rotation of the original seven bits. Rotating to the left twice more gives 1001100, which is 76 and, as CHR$(76) shows, is the code for the letter L.

The ASCII code itself assigns the numbers from 0 to 127 (in decimal) to various characters. Those with codes from 0 to 31 are non-printing characters. In addition, most micros extend the ASCII code by adding 128 to the code for a printable character to give the code for the corresponding reversed character. (Advantage is taken of this in the 'Windows' puzzle in Chapter Three.) That some characters are not printable explains why the thirteen characters in the original message give only eleven in the printed version of the coded string. But there are thirteen characters in the stored version of the coded string. In fact, it is the spaces that give rise to the missing characters. Their code is 32 or 0100000, which when rotated three times becomes 0000010 or 2.

The decoding may be effected by completing seven shifts to the left or, alternatively, by shifting the bits three places to the right to put them back in their original positions. This shifting can be done by dividing by two and remembering that a 1 must be wrapped around if the original number was odd (an even number is exactly divisible by two and the least significant digit of its binary representation is a 0). This form of decoding is implemented by replacing lines 230 to 260 of the solution with:

```
230 FOR I=1 TO 3
240 IF (INT(X/2)*2)<>X THEN X=INT(X/2) OR 64:
    GOTO 260
250 X=X/2
260 NEXT I
```

Chapter Five

Puzzles Involving Computer Applications

The roles in which computers may be applied are tremendously varied. The puzzles in this chapter are concerned with a few that are typical and are encountered quite frequently. Two deal with the use of the computer other than in 'straight' computing, namely in communication and control.

All computers have sockets through which they can receive signals from the outside world and send signals to it, so they have the basic prerequisites for communications built into them. This gives any computer the capability to act as a communications terminal. With the aid of a modem it can communicate over the telephone network. If used as a word processor it can also communicate documents directly to other word processors for storage and for display on their screens, making paper no more necessary for the communication of documents than it is in their preparation. But communicating computers must agree on a common way to exchange signals, and that is the basis of the puzzle on communications.

Since a computer can exchange signals with equipment attached to it, it can also be programmed to control that equipment, be it a printer, a machine tool or a robot. As far as robot developments are concerned, the computer has provided the 'brains' for the robots and consequently has been the means of providing their 'intelligence'. The ability of a computer to run a maze has been used as a test-bed for developments in micro-based robotics as well as a means of stimulating them. How to guide a robot through a maze is the subject of another of the puzzles in this chapter.

Simulation is an important area of application for computers, and simulating the behaviour of, for example, buildings, cars and aeroplanes with a computer during their development has led to considerable savings in development costs as well as speeding up the development process. It also makes development much safer than when the traditional method of building and testing prototypes is used.

A further aspect of simulation is that computers can be programmed to simulate the behaviour of man-made systems so successfully that they can be used in the training of the human operators of those systems. The prime example of this is the flight simulator in which pilots can receive all their training so that when they take to the air for the first time it is as fully trained and certificated pilots.

Simulation is the basis of one of the puzzles where the behaviour of a swaying tower is simulated to determine how long it will take for it to fall down. It is also, rather loosely, the subject of a second puzzle concerning the generation of hands of cards. There is a random element to this that is not present in the simulation of the tower.

The puzzle that completes this chapter deals with a typical software tool that is useful in the development of programs for any application, and which is typical of those found in any of the so-called 'programmer's tool kits'.

5.1

MAZES

A robot can find its way through a maze when it is controlled by a computer running an appropriate program. If the robot is to be able to explore *any* maze that it might encounter, it might be expected that a fairly complex program would be necessary.

However, mazes ranging from the simple one illustrated in Fig. 5.1 to the Hampton Court maze, which have in common that they contain no parts that are completely disconnected from the rest, can all be traversed if you place your right hand on the wall of the maze when you enter it and walk on while keeping it there. The objective is always reached if this method is followed, although rarely by the shortest path.

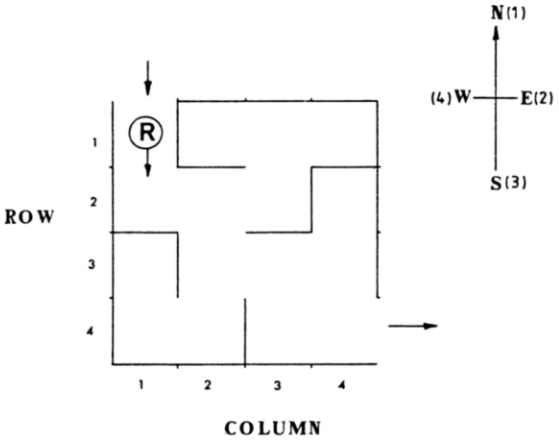


Fig. 5.1. A maze.

An idealised maze of the kind in Fig. 5.1 provides a useful simplification. A robot can occupy any position within the maze, and

need only face north, east, south or west to be able to move straight ahead to an adjacent position. Its position can be described by giving the row, R, and the column, C, that it occupies. The direction in which it faces can be represented by assigning a value to a variable, named DIR, say, using the code given in the figure. Thus on entering the maze, the state of the robot can be represented by:

```
R=1: C=1: DIR=3
```

The program can be halted when the robot reaches the exit by:

```
IF (R=4) AND (C=4) THEN STOP
```

To represent the maze itself, we propose the use of a two-dimensional array of string variables that can be declared by:

```
DIM M$(4,4)
```

The element M$(1,3), for example, is associated with the position in row 1 and column 3. It indicates that there are three paths from this position to adjacent ones to the east, south and west by the value it takes after the assignment:

```
M$(1,3)="234"
```

Similarly, with two positions next to the position in row 4 and column 2, we would have:

```
M$(4,2)="14"
```

The puzzle

Given a maze represented in the way described above and the strategy for traversing it, the puzzle is to write a BASIC program to guide the conceptual robot through the maze.

5.2

RENUMBER

BASIC is not a good language for writing programs to manipulate other programs. Not many people would consider writing a compiler or an interpreter in BASIC, for example. However, it is not too difficult to write a BASIC program that writes out another program. Consider the following:

```
10 READ I$
20 PRINT I$
30 IF RIGHT$(I$,3) = "END" THEN STOP
40 GOTO 10
100 DATA "1 FOR K=1 TO 10"
110 DATA "5 X=X+K"
120 DATA "17 NEXT K"
130 DATA "31 END"
```

The puzzle

This puzzle is to modify the program listed above so that it renumbers the lines of the program in the DATA statements before printing it out on the screen. The line number of the first line should be 100, and numbers for succeeding lines should increase in steps of 10. Just as the program given above displays any program that is placed in its DATA statements, so the solution should renumber any program.

The solution given does not deal with BASIC statements that include quotation marks since this has already been dealt with in 'The quotes puzzle' of Chapter One. Nor does it deal with the line numbers that are given in GOTO and GOSUB statements. These are both further aspects of the puzzle and the solution program may be generalised to take account of them.

5.3

A HAND OF CARDS

The puzzle

This puzzle is to write a program to generate a hand of cards. The cards should be generated in a random fashion so that, in general, the program creates a different hand each time that it is run. The program must also, of course, ensure that the same card does not appear twice in a hand.

When the hand is displayed, each card should be displayed as a character string such as 'TEN OF HEARTS'. This does not necessarily mean that all the cards in the pack must be represented in this way in the program. To make the problem precise, let us make it to generate and display a hand of five cards as in Poker. A program that does this successfully should require only trivial modification to generate a Bridge hand or a hand containing any given number of cards.

5.4

THE SWAYING TOWER

A cylindrical tower is ten metres high and one metre in diameter. Its position is fixed with respect to a rectangular coordinate system as shown in illustration Fig. 5.2(a). At a certain time, which we refer to as t=0, the tower begins to sway as a result of movements in the ground below it. As it sways, its motion is always in the plane of the x and y axes. The tower distorts as it moves, but it always retains a cylindrical shape. In fact, the movement of the tower is described by the following equations:

```
XT = X + Y*Ø.Ø1*(EXP(T/1ØØ)-1)*SIN(2*PI*T/1ØØ)
YT = Y
```

where (X,Y) is the position of any point on the tower at t=0 and (XT,YT) gives the position assumed by that point at any subsequent time t=T. The unit for time is the year.

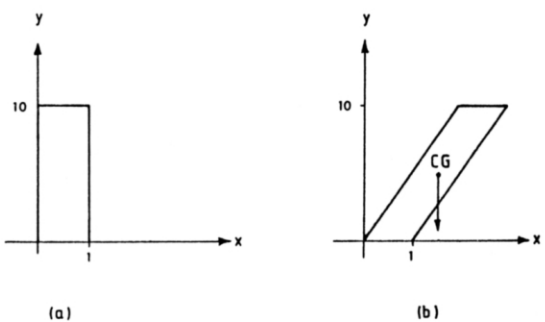


Fig. 5.2. The tower: (a) initial position (b) unstable position.

The puzzle

Of course, if the tower continues to tilt further and further it will eventually fall. The object of this puzzle is to determine when it will fall by writing a BASIC program to simulate its movement. It can be assumed that the tower falls as soon as it reaches an unstable position. Its position becomes unstable when its centre of gravity falls outside its base, as shown in Fig. 5.2(b).

5.5

COMMUNICATIONS

Some types of microcomputer possess a so-called 'user port' via which they can communicate with devices that may be connected to them. The user port is generally an eight-pin socket at which a ribbon cable can be connected. A byte can be communicated in this way by sending its eight bits in parallel along the ribbon cable. The user port of a micro is usually memory mapped so that, as far as the computer is concerned, it looks just like any eight-bit memory location. To communicate via the user port it is necessary to write programs for both the transmitting and receiving equipment since there is no built-in communication procedure as there is with a standard interface such as a Centronics or RS232C port.

Suppose that we want to exchange messages between two identical computers, and that these computers each possess a user port for sending, mapped to memory location 2000, and a second user port for receiving, mapped to location 3000. Also, the user port of the sending computer is attached to the receiving port of the second computer by a ribbon cable as shown in Fig. 5.3.

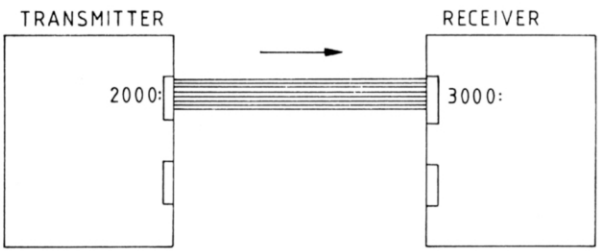


Fig. 5.3. Two computers in communication.

The puzzle

This puzzle is to write short BASIC programs for the transmitting and receiving computers so that a message displayed on the screen of the first computer can be communicated to the second and reproduced on its screen. The message should be displayed starting at the top left of the screen of the sending computer, and should be terminated by a hash character (#). It can be assumed that the memory location to which the top left corner of the screen is mapped is stored under the name SCRTOP.

The transmitting computer must alert the receiving computer when it is about to send a message, and must send the message in such a form that the receiving computer can synchronise itself to it. This is to allow the receiving computer to accept the transmitted message character by character and to place each character on its screen once and once only. But while being able to do this, it must also be able to handle the situation when two successive transmitted characters are the same.

SOLUTION 5.1: MAZES

After declaring the necessary arrays with:

```
10 DIM M$(4,4), INC(4)
```

and initialising the array M$ that is used to represent the maze with a sequence of statements such as:

```
20 M$(1,1)="3": M$(1,2)="2"
```

a program to guide a robot through the maze represented by M$ is:

```
 80 INC(1)=1: INC(2)=0
 90 INC(3)=-1: INC(4)=2
100 R=1: C=1: DIR=3
110 IF (R=4) AND (C=4) THEN STOP
120 TRY=1
130 D=DIR+INC(TRY): IF D>4 THEN D=D-4:
    IF D<1 THEN D=D+4
140 FOUND=0
150 FOR L=1 TO LEN(M$(R,C))
160 IF VAL(MID$(M$(R, C), L, 1))=D THEN FOUND=1
170 NEXT L
180 IF FOUND=0 THEN TRY=TRY+1: GOTO 130
190 DIR=D
200 IF DIR=1 THEN R=R-1
210 IF DIR=2 THEN C=C+1
220 IF DIR=3 THEN R=R+1
230 IF DIR=4 THEN C=C-1
240 PRINT R,C
250 GOTO 110
```

In this program the loop from line 130 to line 180 makes the conceptual robot test if it can move from its starting position to an adjacent position while 'keeping its right hand on the wall'. This is done by testing in sequence whether the robot can move to the position it faces after turning to the right, to the square directly ahead of it, to the square it faces after turning to the left, or if it has no alternative but to retreat to the square behind it. These possibilities correspond to the robot moving through a gap to its right, moving

straight on with a wall at its right, turning left because there is a wall to the right and across the front, and retreating from a cul-de-sac. As soon as a position to which the robot can move is found, the variable FOUND is set to 1 making it possible to leave the loop. The position of the robot is adjusted by one of lines 200 to 230, as appropriate. The program then returns to line 110 to test if the exit from the maze has been reached. If it has the program terminates, but if it has not the process just described is repeated for the robot's next step.

On the subject of mazes, the 'Micromouse' competition involved the development of a small robot (the 'mouse' of the competition's name) that could find its way through a maze by itself. The idea of the competition was for the robot to explore the maze in order to obtain a map of it, and then to make its way through the maze as quickly as possible. The competition was devised to stimulate activities in microcomputer-based robotics, and within a few years the competition had clearly succeeded in its aims for a considerable number of entries in the competition had solved all the problems involved and could find their way through a maze about as quickly as could be expected. Consequently, Micromouse has been succeeded by a robot ping pong competition to stimulate further developments, particularly in new areas, such as robot vision, that played no part in the earlier competition.

Making a robot that can find its own way through a maze requires expertise from at least three separate disciplines: software, electronic hardware and mechanical engineering. The algorithms and programming needed for the robot to have the ability to explore and find its way through a maze quickly achieved near perfection. The necessary electronic components, including microprocessors, integrated circuits and so on, were all readily available and more than equal to the tasks required of them. The major problems were all rather mundane and arose from the mechanical engineering aspects of the project, such as moving the robot precisely both when advancing and when turning so that its position could be known exactly at all times.

The program given here for traversing a maze is by no means the quickest for the purpose. It can be speeded up – though care is needed – by preventing the robot from unnecessarily moving to a position that takes it further away from its destination. Also, the method may not work if the maze has any disconnected parts because should the robot ever find itself in one of them it might never be able to leave it.

SOLUTION 5.2: RENUMBER

The following solution program reads the statements of the program to be renumbered one at a time from its DATA statements until it has read an END statement. It applies the function VAL to each statement, and since each begins with a number (the line number) this number is returned by the function. The number is then tested to find the smallest power of ten that exceeds it, for this power gives the number of digits in the line number. Removing the number of characters given by this power from the left of the statement strips off the old line number, and then the function STR$ can be used with the new line number (stored under N) to place the new line number at the beginning of each statement. In this way the renumbering is achieved for each statement in the program.

The renumbering program is:

```
10 N=100
20 READ I$: L=LEN(I$)
30 V=VAL(I$)
40 P=1
50 IF V < 10^P THEN 70
60 P=P+1: GOTO 50
70 PRINT STR$(N) + RIGHT$(I$, L-P)
80 N=N+10
90 IF RIGHT$(I$,3)="END" THEN STOP
100 GOTO 20
200 DATA ...
```

A BASIC tool kit is usually supplied as a ROM to be inserted in one of the special sockets inside the computer. It makes available a number of routines that are helpful during the writing, developing and correcting stages of program production. Typical tool kit routines include RENUMBER which can be used to tidy up the line numbers in a program, AUTO which automatically generates the line numbers at the beginning of BASIC program lines, FIND which will find a specified pattern of characters wherever it appears in a program, and others. The three routines cited are clearly of value, respectively, in polishing the appearance of a program when it has

been finished, when it is being typed into the computer, and when it is being corrected.

In a ROM the routines will be expressed in machine code. They can, as this puzzle illustrates, be written in BASIC, but BASIC was not intended for applications of this kind. If a high-level language is to be used for writing this sort of program, one such as 'C' is much more suitable.

The solution program depends on the use of the VAL function. When given a string of numeric characters, this function returns their numerical value. As an example, VAL("12") gives the number twelve. Fortunately, if given a string of characters that starts with numeric characters but which also includes non-numeric characters, VAL will give the numerical value of the numeric characters at the beginning of the string. By taking advantage of this, the number at the beginning of each program line in the data statements can be stored under V. Then P is used to find the number of digits in this number by testing if it is less than 10 to the power of P. This makes it possible to strip the line number from the beginning of each line by removing the first P characters. The new line number that is needed for the renumbering is added to the front of the string in place of the old one by using STR$. This function performs exactly the opposite function to VAL.

The renumbering procedure stops by searching for the END instruction in the program that it is renumbering. This provides a graphic illustration of the difference between the STOP and END instructions, which seems to be the source of a good deal of confusion. The END instruction is intended to mark the end of the listing of a program, whereas STOP indicates the point at which the execution of the program will stop when it is run.

SOLUTION 5.3: A HAND OF CARDS

The first solution to be presented begins by initialising arrays with appropriate names with the face values of the cards in the pack and the names of the suits. A card can then be selected at random by generating a random number from 1 to 4 for the suit and a second random number from 1 to 13 for the face value. The description of the card that has been generated is then stored in an element of the array CARD$ at line 100. When more than one card has been generated, lines 120 to 140 check for duplication to prevent the same card from appearing twice in a hand. If the most recently generated card is already in the hand it is rejected and a new one is generated. The program is:

```
 10 DIM SUIT$(4), VALUE$(13), CARD$(5)
 20 FOR J=1 TO 4: READ SUIT$(J): NEXT J
 30 DATA "CLUBS","DIAMONDS","HEARTS","SPADES"
 40 FOR K=1 TO 13: READ VALUE$(K): NEXT K
 50 DATA "ACE","TWO","THREE","FOUR","FIVE"
 60 DATA "SIX","SEVEN","EIGHT","NINE","TEN"
 70 DATA "JACK","QUEEN","KING"
 80 L=1
 90 S=1+INT(4*RND): V=1+INT(13*RND)
100 CARD$(L)=VALUE$(V)+" OF "+SUIT$(S)
110 IF L=1 THEN 160
120 DUP=0
130 FOR M=1 TO L-1: IF CARD$(L)=CARD$(M)
    THEN DUP=1: NEXT M
140 IF DUP=1 THEN 90
150 IF L=5 THEN STOP
160 L=L+1: GOTO 90
```

A totally different approach to this problem starts by representing the cards of the pack by the numbers from 0 to 51 (rather than 1 to 52). This allows us to convert the numbers to card descriptions simply by dividing by thirteen, for the result of the division gives a suit number from 0 to 3, while the remainder from the division gives a face value number from 0 to 12. A card can then be 'dealt' by selecting it at random from the pack by generating a random number from 0 to 51.

Removing the card from the pack as it is dealt obviates the problem of checking for duplicates among the cards. The next card is dealt by selecting a random number from zero to the size of the reduced pack. A good test of one's understanding of how the program works is to try to predict what cards it will deal if the T is omitted from line 110. The program is:

```
10 DIM SUIT$(4), VALUE$(13), PACK(51)
20 FOR J=1 TO 4: READ SUIT$(J): NEXT J
30 DATA "CLUBS", "DIAMONDS", "HEARTS",
   "SPADES"
40 FOR K=1 TO 13: READ VALUE$(K): NEXT K
50 DATA "ACE","TWO","THREE","FOUR","FIVE"
60 DATA "SIX","SEVEN","EIGHT","NINE","TEN"
70 DATA "JACK","QUEEN","KING"
80 FOR I=0 TO 51: PACK(I)=I: NEXT I
90 T=51
100 FOR C=1 TO 5
110 X=INT(T*RND(0))
120 K=INT(PACK(X)/13): J=PACK(X)-13*K
130 PRINT VALUE$(J+1);" OF ";SUIT$(K+1)
140 PACK(X)=PACK(T): T=T-1
150 NEXT C
```

SOLUTION 5.4: THE SWAYING TOWER

A program to simulate the motion of the tower is listed below. The tower in fact oscillates when it moves in the way described by the given equations, which is why two conditions are required in line 60 of the program. The tower may become unstable by leaning too far to the right or to the left.

```
10 T=0: INTERVAL=1
20 PI=3.14159273
30 X=0.5: Y=5
40 T=T+INTERVAL
50 XT=X+Y*0.01*(EXP(T/100)-1)*SIN(2*PI*T/100)
60 IF (XT<1) AND (XT>0) THEN 40
70 PRINT "TOWER WILL FALL AFTER", T, "YEARS"
```

To determine when the tower will fall, the program simulates its movements at successive equally spaced intervals of time until the tower reaches an unstable position.

There are many other ways in which a description of the tower's movements can be provided. The one used in the puzzle gives a time-dependent deformation description of the kind that is used in continuum mechanics. It is not entirely realistic because, as Fig. 5.2 illustrates, the tower deforms as it oscillates in such a way that the height of its top above the ground remains the same. To do this, the tower must become longer and thinner the further it leans over.

A more realistic description of the movement would be to incorporate a time-dependent rotation of the tower, for then the tower would remain rigid as it tilted. If, with reference to Fig. 5.2, the rotation were clockwise about the origin and in the plane of the paper, then the bottom corner at the origin would remain in a fixed position but the other bottom corner (the one initially at (1,0)) would vanish below ground level, thereby simulating the effect of the ground giving way below the tower.

This rotation can be simulated by replacing the movement description given in the puzzle with:

```
XT= X*COS(T/100) + Y*SIN(T/100)
YT=-X*SIN(T/100) + Y*COS(T/100)
```

The solution program needs only slight modification to accommodate this change. Line 50 must be replaced by:

```
5Ø XT=X*COS(T/1ØØ)+Y*SIN(T/1ØØ)
```

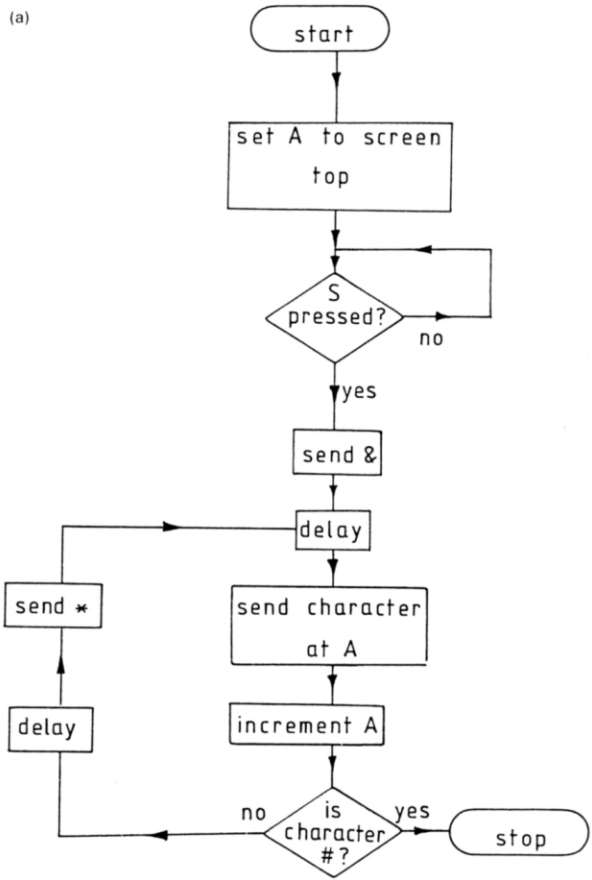
Line 60 will still work, but with the tower rotating rather than oscillating it need only be:

```
6Ø IF XT<1 THEN 4Ø
```

The addition of graphics to the programs to display the current state of the simulation would be a considerable improvement.

SOLUTION 5.5: COMMUNICATIONS

(a)



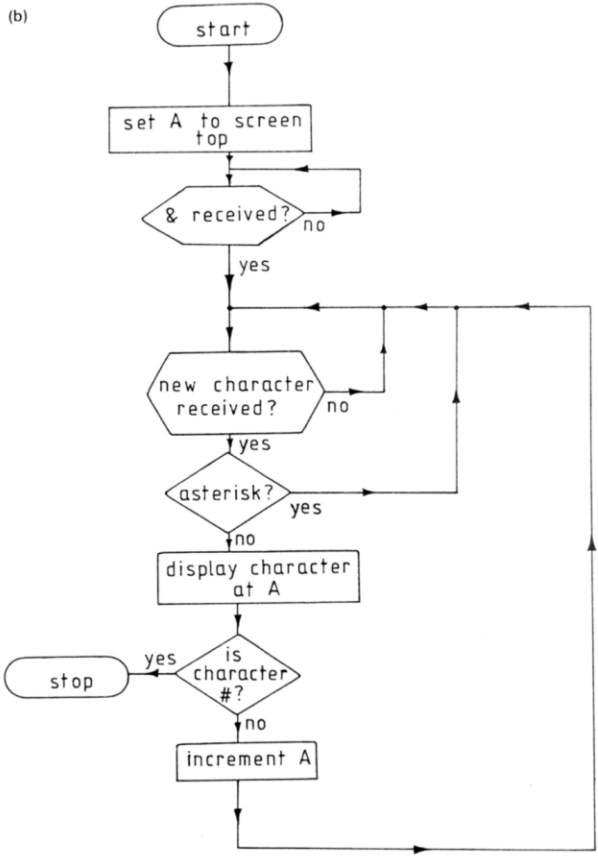


Fig. 5.4. Flowcharts: (a) transmitter (b) receiver.

A procedure for transmitting from one computer to another is shown in the flowchart of Fig. 5.4(a). A transmission program based on this is:

```
100 REM PRESS S TO SEND
110 A=SCRTOP
120 GET A$: IF A$<>"S" THEN 120
130 POKE 3000, ASC("&")
140 FOR K=1 TO 100: NEXT K
150 X=PEEK(A): POKE 3000,X: A=A+1
160 IF X=ASC("#") THEN STOP
170 FOR K=1 TO 100: NEXT K
180 POKE 3000, ASC("*")
190 GOTO 140
```

Figure 5.4(b) shows the flowchart of a receiving procedure. A program for reception based on this is:

```
100 A=SCRTOP
110 X=PEEK(2000): IF X<>ASC("&") THEN 110
120 IF PEEK(2000)=X THEN 120
130 X=PEEK(2000): IF X=ASC("*") THEN 120
140 POKE A,X: A=A+1
150 IF X=ASC("#") THEN STOP
160 GOTO 120
```

While some micros have two user ports, most only have one. When there is only one it must be able both to transmit and to receive signals, and for this reason it is accompanied by what is sometimes known as a 'data direction register'. This is a memory location the contents of which determine the direction in which signals can pass through the user port. Each bit of the data direction register determines whether the corresponding line at the user port carries signals to the computer or from it. A 1 in the data direction register can indicate output and a 0 input on the corresponding line from the port. By POKEing 255 to the data direction register the user port can be configured as an eight-line output port, while POKEing 0 to the register configures it as an eight-line input port. Any number between 0 and 255 causes some lines to be inputs and others outputs.

When computers with a single user port communicate with each other only one ribbon cable is necessary, but one of the computers must be configured to transmit and the other to receive. Computers with separate ports will need two cables connecting them.

The program for transmitting that is presented above stores the address to which the position at the top left of the screen is mapped under the name A and then waits until the S key is pressed. As soon as this key is pressed an & character is sent to alert the receiver to an

incoming message, and a delay is introduced to give the receiver long enough to 'see' this character before the characters of the message are sent one by one. After each character has been sent it is checked to see if it is the # character that will terminate the message. Each character is also left on the line during a delay for the same reason as the initial character was. The transmission of every character in the message is followed by the transmission of an asterisk so that if two successive characters of the message are the same the receiver will be able to detect both of them. Without the asterisk between them there would be no transition between the transmissions of the identical symbols, and therefore no way for the receiver to detect a second character.

The receiving computer scans the input port until an & character is received. Then it waits for a new character to appear, and displays it on the screen provided that it is not an asterisk. As soon as a # character is received and displayed the program halts.

Any difficulty in understanding these programs should be alleviated by referring to the flowcharts (Figs. 5.4(a) and (b)).

Appendix
Further Reading

This appendix gives some sources of further reading for most of the puzzles. From them it may be possible to trace the source of inspiration for a puzzle, further background information or more aspects of solution methods for a puzzle. Some of the puzzles, however, are sufficiently self-contained as to make further references unnecessary.

Words

Choosing and Using Program Generators, Chris Naylor, Sigma Technical Press, 1984.

Approximations to English

The Mathematical Theory of Communication, C. E. Shannon and W. Weaver, Illini Books, 1964.

Substitution *and* Associations

Artificial Intelligence and Natural Man, Margaret Boden, Harvester Press, 1977.

Rearrangements *and* The tethered goat

Numerical Analysis, F. Scheid, Schaum, 1968.

Binary display *and* Coding

A First Course in Computer Technology, M. G. Hartley and M. Healey, McGraw-Hill, 1978.

Networks *and* Reversal

Fundamental Algorithms, D. E. Knuth, Addison Wesley, 1975.

Patterns

Get More from the TI99/4A, Garry Marshall, Granada, 1983.

Matchsticks

Programming with Graphics, Garry Marshall, Granada, 1983.

Windows

PET Graphics, Nick Hampshire, Nick Hampshire/Commodore, 1981.

High resolution from low-resolution systems

Computer Languages and their Uses, Garry Marshall, Granada, 1983.

Crossovers

A Practical Introduction to Computer Graphics, I. O. Angell, Macmillan, 1981.

Chaining

A Practical Introduction to Pascal, I. R. Wilson and A. M. Addyman, Macmillan, 1978.

Classification

An Introduction to Algol68 Through Problems, A. Learner and A. J. Powell, Macmillan, 1976.

Recursion

The Little LISPer, D. P. Friedman, SRA, 1974.

Mazes

DIY Sensors and Robotics, John Billingsley, Sunshine, 1984.

Communications

Micros and Modems: Telecommunicating with Personal Computers, Jack Nilles, Reston, 1983.

A collection of stimulating brainteasers which first appeared week by week in The Observer Magazine. Fascinating background information is provided for each puzzle, and the various techniques involved in solving the problems are set out with many useful hints. You are also shown the ways in which each puzzle can be extended to provide hours of extra interest and pleasure.

The Author

Garry Marshall is a regular contributor to computing magazines, including *Computing Today* and *Computer and Video Games*, and was contributing consultant to *The Home Computer Course*, Britain's first partwork on computing.

Other computing books

PROGRAMMING WITH GRAPHICS
Garry Marshall
0 246 12021 5

COMPUTER LANGUAGES AND THEIR USES
Garry Marshall
0 246 12022 3

SIMPLE INTERFACING PROJECTS
Owen Bishop
0 246 12026 6

CHOOSING A MICROCOMPUTER
Francis Samish
0 246 12029 0

GET MORE FROM THE EPSON PRINTER
Susan Curran
0 00 383001 2

INSIDE YOUR COMPUTER
Ian Sinclair
0 246 12235 8

Front cover illustration by Godfrey Dowson

COLLINS
Printed in Great Britain
0 00 383125 6

£6.95 net