ROUTINE CHECK UP

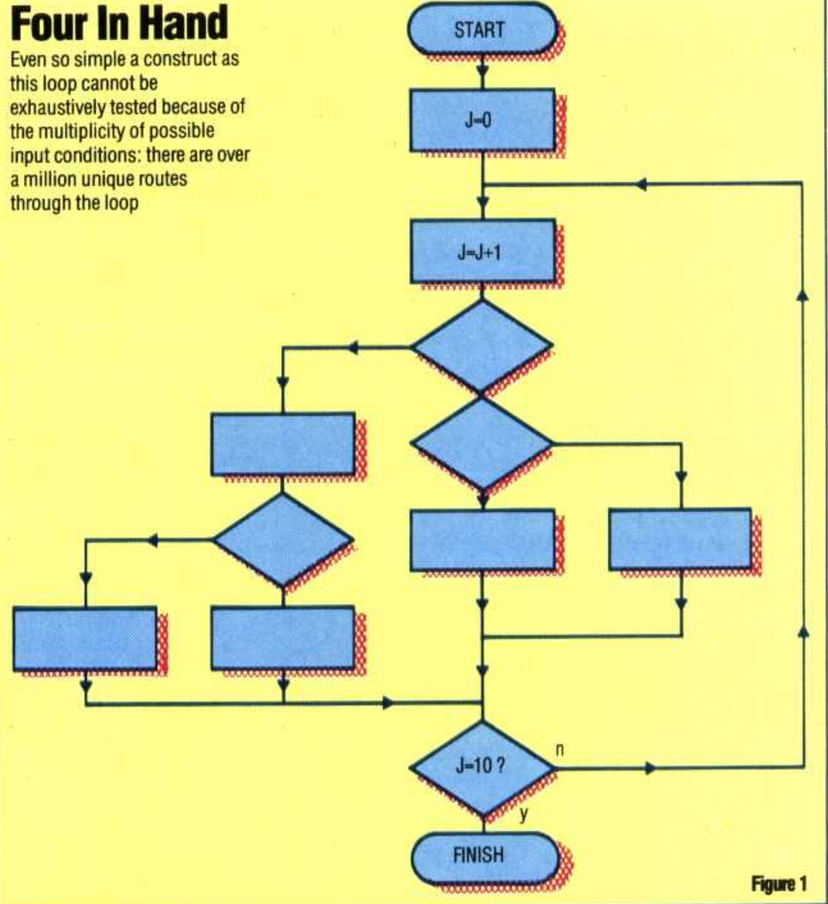
Our series of articles on programming techniques should have provided plenty of ideas for program design and development. In this final part, we discuss the methods that may be used to test a finished program.

One of the great advantages of programming in an interpreted language like BASIC is that code can be tested as it is being written. The programmer can, at any time, type RUN and see what happens. On most machines, it is a simple matter to 'break' into a running program, PRINT the values of key variables, change these values and then CONTinue. All this means that most of the more obvious mistakes will have been spotted and corrected. Yet this kind of *ad hoc* debugging is not a substitute for testing, which must be done when the program is in its complete and final form.

Validation testing aims to ensure that a program will do exactly what it is meant to do. For any legal set of input data it must produce the correct output, and for any illegal input it must take the appropriate actions. A simple way to test a program might seem to be to give it a sample of every legal input and then check that the results are as expected. For almost every program, this will be impossible, however. Even a program that takes two integers, adds them and prints the result would need to be tested for every possible integer value! Yet this is only part of the problem, as every *illegal* value would need to be tested, too.

Another possibility might be to look at every 'path' through the program. A particular path can be found by following one route through a control flow diagram (flowchart) from beginning to end. Each branch on the way allows for alternate paths and each loop adds more. Figure 1 shows a simple program that is a loop containing a number of IF... THEN statements. There are four paths within the body of the loop and the loop is executed 10 times. This means that the number of unique routes from 'start' to 'finish' is 1,398,100 — a staggering number for what would probably amount to a dozen lines of code. Clearly, testing this way would be out of the question.

So, if exhaustive data testing does not work and exhaustive logic testing does not work, what does? The surprising answer is that nothing does. There is no way to test completely a reasonably complex program in a realistic time. Partly for this reason, testing follows the law of diminishing returns — the number of errors found per unit of effort decreases with each extra unit. So, the time to stop is when the effort of doing it outweighs the cost of the program's (as yet undetected) faults.



Four In Hand
Even so simple a construct as this loop cannot be exhaustively tested because of the multiplicity of possible input conditions: there are over a million unique routes through the loop

Figure 1

However, despite these drawbacks, it is worth devising some method of testing. A reasonable assumption is that if a machine will operate correctly on one datum of a particular type it will operate correctly on all data of the same type. So, if a subroutine works for one positive integer within its range, it should work for all positive integers in that range. This leads us to a type of testing known as 'equivalence class testing'. The idea is to develop a set of test cases that are each representative of a class of cases that should all behave in the same way. Thus, if a piece of code checks that an input is in the range 1 to 100, we should test for inputs that are less than the lowest value expected, greater than the highest value, and within the expected range (value<1; value>100; and 1=<value=<100).

Examining every logic path can also be simplified to invoking each point of entry to all routines (although ideally there should only be one for each) and, inside each routine, covering each possible outcome of every decision branch. In figure 2 we have a routine for adjusting bonus points in a game. It takes the input parameters

Just Testing
A complete set of hand-calculated test data for the example illustrated in the flowcharts might look like this:

	INPUT		OUTPUT
LEVEL	HITS	BONUS	BONUS
6	10	200	1300
4	10	550	2300
7	10	550	3950
4	10	200	800
7	10	200	1400
1	20	2500	2600
1	20	550	550
6	5	200	300
6	50	200	300
4	5	2500	2600
7	50	2500	2600
4	50	550	550
7	5	550	550