Cambridge Learning Limited

Computer Programming in BASIC

A SELF INSTRUCTION COURSE in 4 Volumes

PART 3 — Applying BASIC

By:
Ian Williamson Rodney Dale Tim Eiloart

COMPUTER GAMES 1

Though computers are serious machines, and programming is a serious business, there is no reason why it cannot be fun as well. This program is quite complex, but once written, can be very entertaining.

Enough of the serious stuff; computers can be great fun especially for playing games.

PROBLEM

In the game of "pubsticks" (which some people call "Nim", though Nim is really much more complicated) there are two players. Each player, in turn, draws one, two, or three matches from a pile. There are 17 matches in the pile to start with. The last player to draw a match wins. The first player can always win. How? Try playing the game with a friend and see if you can work out the winning strategy. Look for winning positions when almost all the matches have been drawn.

If you cannot work out the strategy then try following this series of hints. At any time you feel you are able you should try to complete the thinking needed to find the winning strategy.

HINT 1

If it is the other player's turn, and there are four matches left, can I be sure to win? Remember, the last person to draw a match is the winner.

ANSWER

Yes. However many matches my opponent draws I can draw all the rest. If opponent takes one, I take three, etc.

HINT 2

So I am sure to win. If it is my turn, and I want to leave only
many must there be? I could win if there are 5. What other

ANSWER

I could win from 5, 6 or 7. I can draw one, t
my opponent four to draw from. Look back
am then certain to win.

COMPUTER PROGRAMMING in BASIC

PART 1 - BASIC BASICS

PART 2 - INTRODUCING BASIC

PART 3 - APPLYING BASIC

PART 4 - ADVANCED BASIC

SUMMARY of BASIC LANGUAGE

Lesson numbers printed in italics

STATEMENTS

		Lessons
DATA	DATA 4,3.12,-24	*4,25*
DEF FN	DEF FNA(N)=(1+3.14159*SIN(N))	*52*
DIM	DIM A(2Ø),B(1ØØ),C(Y+2*Z)	*42*
END	END	*4*
FILES	FILES NAMES	*6Ø*
FOR...TO...STEP	FOR N=1 TO 1Ø STEP 2	*36*
GOSUB	GOSUB 1ØØØ or GOSUB (X+Y)	*5Ø*
GO TO	GO TO 8Ø or GO TO (X+Y)	*3Ø*
IF...THEN...	IF X>=Y THEN 9Ø	*3Ø*
IF END	IF END # 1 GO TO 9Ø	*6Ø*
INPUT	INPUT A,B,C	*28*
INPUT (of a file)	INPUT # 1,L,N$(I),X$(I)	*6Ø*
LET	LET A=B+1Ø	*6,13*
MAT	MAT X=Y+Z	*57*
NEXT	NEXT N	*36*
ON...GO TO...	ON N GO TO 1ØØ,12Ø,13Ø	*57*
PRINT	PRINT, "LABEL";N,A	*4,14,29*
PRINT USING	PRINT USING 4Ø,X,Y	*57*
RANDOMISE	RANDOMISE	*46*
READ	READ A,B,C	*4,25*
REM	REM THIS IS A REMARK	*27*
RESTORE	RESTORE	*39*
RETURN	RETURN	*5Ø*
STOP	STOP	*31*

NUMBERS

123	-456	7.981	-Ø.ØØ4	1E1Ø	-1.4E-4
(4)	*(4)*	*(4)*	*(4)*	*(11)*	*(11)*

VARIABLE NAMES

A	Z1	B(1)	X(12Ø)	Z(I+4*J)	T(4,4Ø)	C$	D1$	Y$(15)
(12)	*(12)*	*(41)*	*(42)*	*(42)*	*(42)*	*(54)*	*(54)*	*(54)*

FUNCTIONS

ABS	ATN	COS	EXP	INT	LOG	RND	SGN	SIN	SQR	TAB	TAN
(24)	*(23)*	*(23)*	*(24)*	*(24)*	*(24)*	*(46)*	*(24)*	*(23)*	*(9)*	*(48)*	*(23)*

OPERATORS

( )	↑	*	/	+	-	:	=	>	<	>=	<=	<>
(8)	*(6)*	*(4)*	*(5)*	*(4)*	*(6)*	*(SL3)*	*(4,54)*	*3Ø & 54*				

COMMANDS

DELETE	KILL	LIST	NEW	OLD	REPLACE	RUN	SAVE	UNSAVE	\	←
(SL2)	*(5)*	*(SL2)*	*(5)*	*(5)*	*(SL2)*	*(5)*	*(5)*	*(SL2)*	*(SL1)*	*(SL1)*

Cambridge Learning Limited,
Rivermill Lodge,
St. Ives,
Huntingdon,
Cambs PE17 4EP.

ISBN 0 905946 05 7

Computer Programming in BASIC

A unique, teach-yourself course

Part 3: Applying BASIC

by

Ian Williamson, Rodney Dale and **Tim Eiloart**

First published 1979.
Reprinted 1980 (twice), 1981, 1982

ISBN 0 905946 05 7

published by
Cambridge Learning Limited
Rivermill Lodge
St. Ives
Huntingdon
Cambridgeshire
U.K.

printed in England by
Cambridge Learning Limited

CONTENTS with Summaries

LESSON 35

COMPILERS AND INTERPRETERS

When you are writing programs, it is a good idea to know how the computer works inside, so that you can use it to full advantage. This lesson tells you about some of the internal workings of computers.

In Part 2 your understanding of computer programming and the BASIC language was developed considerably. But BASIC is only one computer language, and this course is as much concerned with computer programming, as with the mechanics of coding in a specific language. In this book, the emphasis is on developing your programming style, and extending the range of problems you can tackle successfully.

In describing BASIC programs we tend to say that the computer adds A to B and multiplies the result by C, and so on. So it does, but the machine must first be told exactly how to add A to B and multiply the result by C in the machine-code which it can execute. The translation of our statement LET T=(A+B)*C is done by a program, just like our BASIC program, which is executed by the computer.

So it is programs, normally supplied by the computer manufacturer, which translate our high-level language programs (such as BASIC and FORTRAN) into the machine-code of the system.

High-level languages are translated either with a *compiler* or an *interpreter*, depending on the language, and its application.

A compiler translates the whole of a high-level language program into machine-code before any part of the program is executed. It translates the high-level language instructions into the appropriate sets of machine-code instructions specific to that computer system. Each compiler matches the individual characteristics of its computer, so that the high-level language is said to be *transportable*; that is to say it can be used on a number of different computers with little modification. Apart from generating instructions in machine-code, the compiler assigns areas in the memory to the program and data. The programmer need never know the details of the inner workings of the computer.

An *interpreter* differs from a compiler in that it translates the high-level language instructions into machine-code one by one, and causes the computer to execute them immediately. The high-level language is therefore executed line-by-line. For small calculations this is an effective technique, because in this mode the computer resembles a powerful programmable calculator. For larger computations, using an interpreter may result in the execution time being long, when compared with that needed for a program which has been compiled. Consider a program which must be executed three times, each time with fresh input data. A compiler would translate the program only once, and the resulting machine-code would be executed three times by the computer. An interpreter would translate the program from scratch during each execution - three translations and three executions.

Compilers and interpreters also differ in their use of computer memory. The interpreter program is resident in the computer memory along with the high-level language it is executing. However, because a compiler carries out the

complete translation prior to any execution, the compiler is not required in the computer memory during program execution, and may therefore be stored on tape or disc. This distinction is important when the computer memory is limited, as in minicomputer systems.

Compilers are generally used where programs are to be executed frequently, since compilation is a once-only operation.

Interpreters are useful for small programs, individual calculations, and programs which require modification. The BASIC language is often translated using an interpreter. Whether the translation of a high-level language is by a compiler or by an interpreter, the machine-code which is generated will not be either as compact or as efficient as that produced by a good assembly-language programmer. This is currently inevitable, since the only way to make full use of the computer facilities is to program the computer in machine-code or its assembly-language equivalent. In addition, compilers and interpreters introduce some unavoidable *redundancy* into a machine-code, thus reducing the efficiency of the translation.

The relative efficiencies of compilers and interpreters is a topic of consuming interest to computer scientists - and of little relevance to computer users except where computer memory is strictly limited. A major cost in using a computer is the development and maintenance of its programs, often a more significant element than the cost of computer time for program execution. Programs written in high-level languages generally cost less to develop and maintain than those written in assembly language. This is because a high-level language program is succinct and readable, whereas assembly-language programs are often lengthy, difficult to understand, and dependent upon the idiosyncrasies of the individual programmer.

QUESTIONS

1) What are compilers and interpreters, and what is the difference between them?

2) Why are high-level languages preferable to assembly languages for many computer applications?

ANSWERS

1) Compilers and interpreters are both programs, supplied by the computer manufacturer, which translate high-level languages into machine-code for execution by the computer. A compiler translates the whole of the high-level language program into machine-code: the program is then executed by the computer.
An interpreter translates the high-level language program a line at a time, and causes the computer to execute the resulting machine-code as it is fed to it.

2) Whilst high-level languages are less efficient than assembly languages in terms of computer execution time, high-level language programs cost less to develop and maintain, because they are succinct, readable, and easy to understand.

LESSON 36

LOOPS AND THE FOR ... NEXT STATEMENT

When you have many repetitive calculations to perform in a program, the FOR ... NEXT statement will enable you to do this easily and efficiently.

In the days before every shop and home could afford a pocket calculator, and before electronic cash registers became the norm, many shop assistants used ready reckoners to calculate the cost of large purchases of small items like balls of wool, or screws and fixings.

You may not even have seen a ready reckoner! Here is an example of a page of a ready reckoner for calculating the cost of up to 5Ø items with a unit cost of 2 pence.

Page 1					Unit Cost = 2 pence
Tens Units	Ø	1Ø	2Ø	3Ø	4Ø
Ø	Ø	2Ø	4Ø	6Ø	8Ø
1	2	22	42	62	82
2	4	24	44	64	84
3	6	26	46	66	86
4	8	28	48	68	88
5	1Ø	3Ø	5Ø	7Ø	9Ø
6	12	32	52	72	92
7	14	34	54	74	94
8	16	36	56	76	96
9	18	38	58	78	98

Now, with your current knowledge of BASIC, you could write a program to produce a ready reckoner page like this one - in fact your biggest difficulty would be the print-out since the ready reckoner above has six zones across the page - more than the five zones provided by the PRINT statement. To ease this problem our program calculates the value of Ø to 39 items, i.e. we will omit the right hand column. We would also have to rule the vertical lines on the table, of course.

QUESTION

On the next page we give you a partially written BASIC program; try and complete the middle section for yourself.

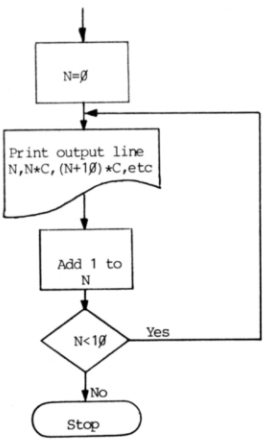
```
1Ø REM READY RECKONER PROGRAM FOR Ø TO 39 ITEMS
2Ø REM  I. WILLIAMSON      31-MAY-99
3Ø REM PROGRAM IS INCOMPLETE
4Ø READ C
5Ø PRINT "COST OF ITEM";C
6Ø PRINT
7Ø PRINT ," Ø", " 1Ø"," 2Ø"," 3Ø"
8Ø REM INSERT YOUR FOUR STATEMENTS HERE
9Ø REM          "
1ØØ REM         "
11Ø REM         "
12Ø DATA 2
13Ø END
```

HINT

N is the 'units column' and must take on all values from Ø to 9. You should think out how to write the program in only four lines.

ANSWER

```
8Ø LET N=Ø
9Ø PRINT N,N*C,(N+1Ø)*C,(N+2Ø)*C,(N+3Ø)*C
1ØØ LET N=N+1
11Ø IF N<1Ø THEN 9Ø
```



This section of code is called a *loop*, since the program loops round from 11Ø to 9Ø ten times in all, with N=Ø,1,2 .....9 as can be seen in the flowchart.

There is a more convenient way of programming loops in BASIC - using the FOR .... NEXT statement. Here is the same section of code programmed using the FOR .... NEXT statement:

```
8Ø FOR N=Ø TO 9
9Ø PRINT N,N*C,(N+1Ø)*C,(N+2Ø)*C,(N+3Ø)*C
1ØØ NEXT N
```

Like the READ and DATA statements, FOR and NEXT come together to set up a loop, where the PRINT statement is executed 1Ø times with N=Ø,1,2....9.

Here is another way in which the ready reckoner program could have been completed:

```
8Ø LET N=Ø
85 PRINT N,
9Ø LET T=Ø
95 PRINT (N+T)*C,
1ØØ LET T=T+1Ø
1Ø5 IF T<4Ø THEN 95
11Ø LET N=N+1
115 IF N<1Ø THEN 85
```

QUESTION

What is happening in this program?

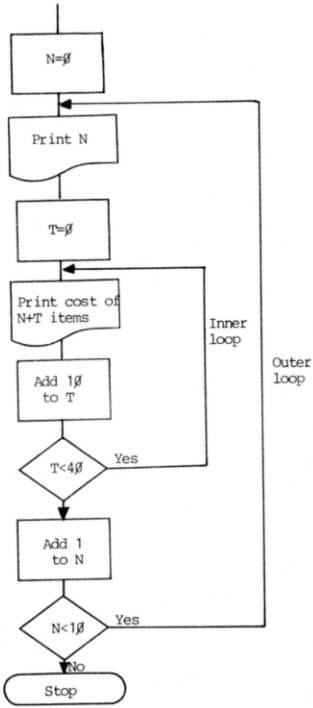
HINT

Try working through the program, going round the loops, and writing down the PRINT for yourself.

ANSWER

We now have two loops - our original loop made up of lines 8Ø,11Ø and 115, and a new inner loop made up of lines 9Ø, 1ØØ and 1Ø5. The second loop simplifies the PRINT statement in line 95, since only a single entry is printed each time the computer executes the inner loop.

The flowchart is now more complicated - but the inner and outer loops can be picked out.



The inner loop can also be coded using the more general form of the FOR ...NEXT statement:

```
8Ø FOR N=Ø TO 9
85 PRINT N,
9Ø FOR T=Ø TO 3Ø STEP 1Ø
95 PRINT (N+T)*C,
1ØØ NEXT T
1Ø5 NEXT N
```

The inner loop is executed four times with T=Ø,1Ø,2Ø and 3Ø - the increase being specified by the STEP.

Here are the rules for using FOR...NEXT statements:

Any step sizes may be specified in the more general form of the FOR...NEXT statement.

```
1Ø FOR X=15 TO 17.5 STEP Ø.5

6Ø NEXT X
```

This FOR...NEXT statement causes the computer to step through the loop five times (the first step followed by five times through the loop) with X=15, 15.5, 16, 16.5, 17, and 17.5.

Unless told otherwise by a STEP, the computer assumes that the STEP is 1, for example, if the STEP is omitted in the example above, the computer will execute the loop three times with X=15, X=16, and X=17.

Any BASIC expression can be used to specify the initial value, the final value and the STEP size in the FOR...NEXT statement. The following statements are valid:

```
7Ø FOR X=1+A*B TO C↑2 STEP A/B

9Ø FOR I=1 TO INT(A*B) STEP ABS(X)
```

The expressions are evaluated at the beginning of the loop, and the initial value, final value, and step size then remain constant whilst the loop is executed.

The step may be negative, in which case the initial value must be greater than the final value, or the loop will be ignored.

If you write FOR X=1Ø TO 1ØØ STEP 35 then the computer will step to 45 then 8Ø, then it will stop. This is just like a conditional jump with X=<100.

QUESTIONS

Which of the FOR...NEXT statements given below are invalid, and why?

(a) `2Ø FOR X=-1 TO 1Ø`

(b) `3Ø FOR I=9 TO 5 STEP 2`

(c) `6Ø FOR J=4 TO 15 STEP -Ø.2`

(d) `7Ø FOR N=4 TO 11 STEP Ø`

(e) `9Ø FOR Z=-5 TO -1Ø STEP -1`

(f) `1ØØ FOR Y=24 TO 3Ø STEP 1Ø`

ANSWERS

b) Final value must exceed initial value if the step is positive.

c) Final value must be less than initial value if the step is negative.

d) Zero step not permitted.

Note: f) is acceptable. The program will evaluate Y=24 then, after a single calculation, leave the loop.

When loops operate within loops, as in the ready-reckoner example, such loops are said to be nested.

It goes without saying (doesn't it?) that a programmer must make sure that his loops are truly nested and not crossed as shown in the two examples below. If you are in the habit of drawing a flowchart, this should never happen to you.

```
FOR L=1 TO 10
FOR W=1 TO 5
.
.
.
NEXT W
NEXT L
```

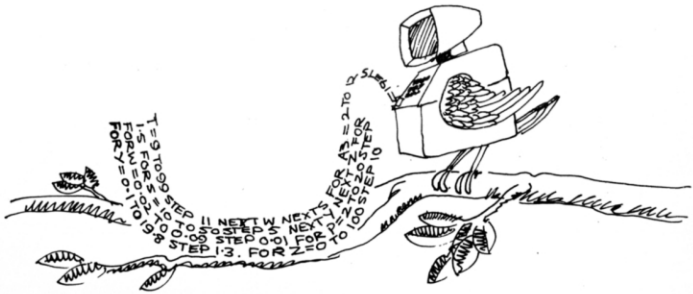
(a) right

✓

```
FOR L=1 TO 10
FOR W=1 TO 5
.
.
.
NEXT L
NEXT W
```

(b) wrong

✗



LESSON 37

TWO SIMPLE PROBLEMS WITH LOOPS

Ways of using FOR...NEXT statements in everyday problems.

Here are two simple problems for you to program. The solutions should contain a program loop or loops. If you find the first problem very easy - skip the second problem and go on to Lesson 38.

PROBLEM 1

Write a program to find the largest number in a set of any eight numbers. Draw a flowchart before attempting the program coding.

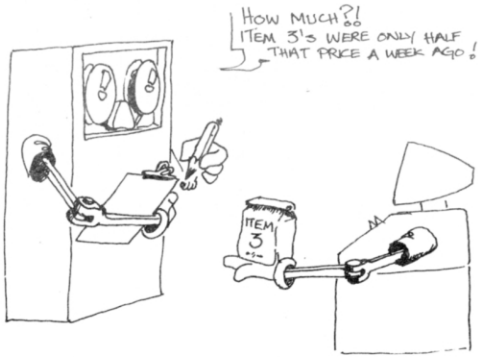
Use the following numbers in the DATA statement: 4.2, 3.1, 2.4, 7.6, 9.3, -4.6, -22.1, 5.6.

PROBLEM 2

Write a program to tabulate the cost of four items in quantities of 1Ø off, 2Ø off, and 3Ø off.

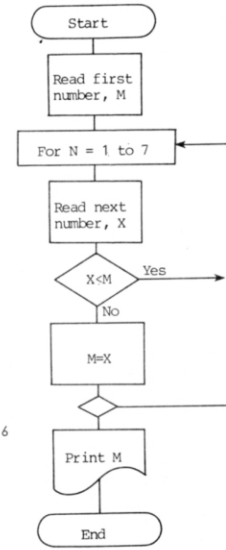
Item 1 costs 5.42 icu
Item 2 costs 3.61 icu
Item 3 costs 2.54 icu
Item 4 costs 1.21 icu

The four columns of your tabulation should be headed 'ITEM', 'UNIT COST', 'QUANTITY', 'TOTAL COST'.



SOLUTION - PROBLEM 1

FLOWCHART

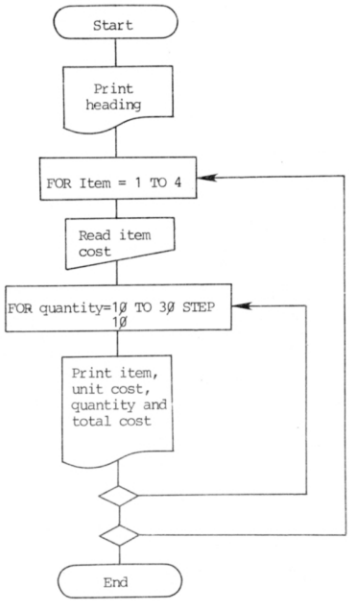


PROGRAM

```
10 READ M
20 FOR N=1 TO 7
30    READ X
40    IF X<M THEN 60
50    LET M=X
60 NEXT N
70 PRINT "MAXIMUM NUMBER IN SET IS ";M
80 DATA 4.2,3.1,2.4,7.6,9.3,-4.6,-22.1,5.6
90 END
READY

RUN

MAXIMUM NUMBER IN SET IS  9.3

END AT 90
READY
```

SOLUTION - PROBLEM 2

FLOWCHART



SOLUTION

```
10 PRINT
15 PRINT "ITEM","UNIT COST","QUANTITY","TOTAL COST"
20 FOR I=1 TO 4
30    READ C
40    FOR N=10 TO 30 STEP 10
50       PRINT I,C,N,N*C
60    NEXT N
65    PRINT
70 NEXT I
80 DATA 5.42,3.61,2.54,1.21
90 END
READY

RUN

ITEM            UNIT COST       QUANTITY        TOTAL COST
 1               5.42            10              54.2
 1               5.42            20              108.4
 1               5.42            30              162.6

 2               3.61            10              36.1
 2               3.61            20              72.2
 2               3.61            30              108.3

 3               2.54            10              25.4
 3               2.54            20              50.8
 3               2.54            30              76.2

 4               1.21            10              12.1
 4               1.21            20              24.2
 4               1.21            30              36.3

END AT 90
READY
```

LESSON 38

PROGRAM EXAMPLE - COMPOUND INTEREST AND AN ELABORATE FORMAT

A practical problem using nested loops - and the best way to display lots of data.

This program example is about compound interest - if you are puzzled by compound interest read Supplementary Lesson 5 before attempting this example.

In a regular savings account payments of between 1 and 2Ø units are made at monthly intervals. The account is opened with a deposit equal to the monthly payment. Interest is calculated on a monthly basis and added to the capital sum at each anniversary. Write a program which shows the growth of the capital every two years over a ten-year period, with interst rates between 8.5 and 1Ø per cent in increments of Ø.5 per cent and monthly payments of either 5 icu, 1Ø icu, 15 icu, or 2Ø icu.

Whilst the capital can be calculated using a complex formula, construct your program so that it calculates the payment each month, as indeed the computer at the savings bank must do.

QUESTION

Problem definition - what aspect of this problem should you investigate and define first?

ANSWER

The problem requires a considerable quantity of print-out, and, for this problem you should start by defining the format of the print-out in detail.

QUESTION

So what should the print-out look like?

HINT

Remember that we are told to present the accumulation of capital over a ten-year period, for four interest rates, and four monthly payments.

ANSWER

On the next page we show alternative output formats. We prefer B - can you see why?

A) Using the five zones of the BASIC print-out we are able to present the accumulation of capital with respect to the four interest rates as follows:

CAPITAL SUM ACCUMULATED OVER 1Ø YEAR PERIOD FOR 5 ICU MONTHLY PAYMENT

PERIOD	INTEREST RATES			
YEARS	8.5	9.Ø	9.5	1Ø.Ø
2				
4				
6				
8				
1Ø				

This print-out is repeated four times for each of the four monthly payments. i.e. for 5 icu per month, 1Ø icu per month, 15 icu per month and 2Ø icu per month.

B) Or, also using the five zones of the BASIC print-out, we can present the accumulation of capital with respect to the four monthly payments.

CAPITAL SUM ACCUMULATED OVER 1Ø YEAR PERIOD WITH 8.5% ANNUAL INTEREST

PERIOD	MONTHLY PAYMENTS			
YEARS	U5	U1Ø	U15	U2Ø
2				
4				
6				
8				
1Ø				

Again, this print-out is repeated four times for each of the four interest rates, i.e. 8.5%, 9%, 9.5%, and 1Ø%.

QUESTION

So why is B) a better output format? - assume that you want to use the format which requires least computation.

HINT

The accumulated capital is directly proportional to monthly payment at a fixed interest rate.

ANSWER

Look at the first line in print-out A. To calculate the capital sum after 2 years at 8.5% interest is a complicated and lengthy computation, but similarly lengthy computations are required at 9%, 9.5% and 1Ø.Ø%. These four calculations are for each line of print-out. But in print-out B, we compute the capital sum after 2 years at 8.5% interest with a 5-unit monthly payment - and need only multiply this value by 2 for the 1Ø-unit monthly payment, by 3 for 15-units, and by 4 for 2Ø-units. In each line of print-out in B we have only one complicated calculation, and three simple multiplications.

B is a lot less work for the computer than A, and if you are paying for your computer time, B will cost less to execute. There may be other reasons for preferring A to B, and you as a programmer should question the users to find out exactly which will be easier for them to use. But our hint asked for the least computation.

So we see that the output format can also influence the computer processing required, and the overall efficiency of the computer program.

QUESTION

The tables in the print-out will be easier to understand if we print the capital to the nearest penny or one hundredth of an icu. The INT function can be used to perform a rounding function - can you think how?

HINT

You have already learnt how to *truncate* using the INT function. Truncation gives the integer part of a number - rounding gives the *nearest* whole number, e.g.

truncating 9.7 gives 9
rounding 9.7 gives 1Ø.

HINT

Examine the following expressions and calculate the results:

1) INT(X); where X = 4.7

2) INT(X+Ø.5); where X = 4.4 and 4.6

3) INT(X*1ØØ+Ø.5)/1ØØ; where X = 4.4755 and 3.1245.

ANSWER

1) INT(4.7) = 4

4.7 is *truncated* to the integer 4.

2) INT(4.4 + Ø.5) = 4 but INT(4.6 + Ø.5) = INT(5.1) = 5

Thus, both 4.4 and 4.6 are rounded to the nearest integer value.

3) INT(4.4755*1ØØ+Ø.5)/1ØØ = INT(447.55+.5)/1ØØ = INT(448.Ø5)/1ØØ
= 448/1ØØ = 4.48

INT(3.1245*1ØØ+Ø.5)/1ØØ = INT(312.45+.5)/1ØØ = INT(312.95)/1ØØ
= 312/1ØØ = 3.12

4.4755 and 3.1245 are rounded to 2 decimal places i.e. they are correctly rounded to the nearest penny.

QUESTION

Now write a program to show the growth of capital in a regular savings account. In some savings accounts the interest is incorporated monthly into the capital. Assume that for the purpose of this exercise the interest is added to capital once a year.

Remember to construct a flowchart before coding the program. We give you one more HINT below. Then you are on your own!

HINT

Our solution has four nested loops.

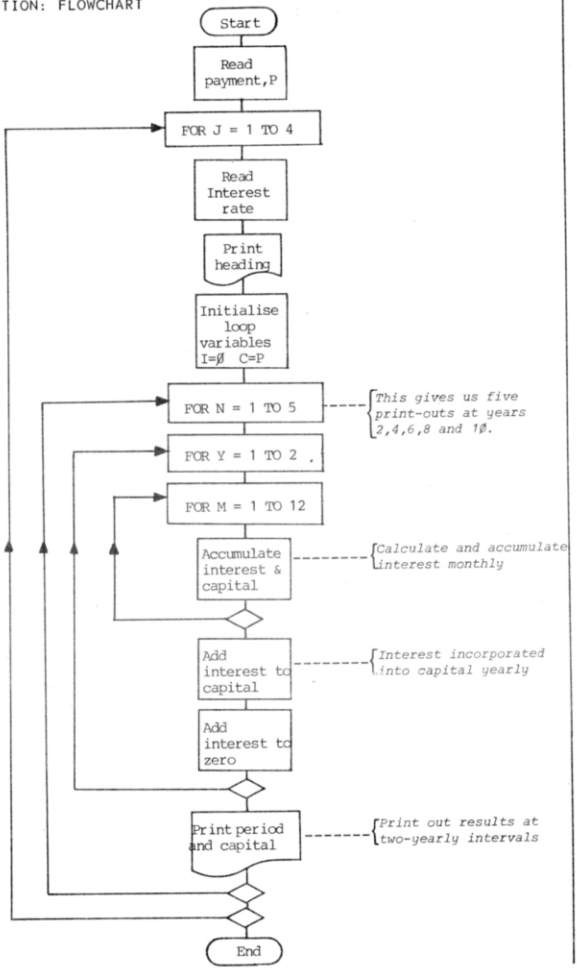
The innermost loop accumulates interest and capital monthly.

The next loop incorporates the interest in the capital annually.

The third loop prints out the accumulated savings at 2 yearly intervals.

The outer loop repeats the print-out B (shown previously) four times, with interest rates 8.5%, 9.Ø%, 9.5% and 1Ø.Ø%.

SOLUTION: FLOWCHART



To construct a flowchart of this complexity, start with a central core loop, in this case the monthly calculation of capital and interest, and then add the outer loops, taking care that loop variables are properly initialised and reset if necessary.

Translating the flowchart into a BASIC program is now quite straightforward.

```
10 REM PROGRAM TO CALCULATE CAPITAL APPRECIATION
20 REM IN REGULAR SAVINGS ACCOUNT OVER 10-YEAR PERIOD EVERY TWO YEAR
30 REM DATA IS MINIMUM MONTHLY PAYMENT AND FOUR INTEREST RATES
40 REM     J.BLOGGS              31-MAY-99
50 READ P
60 REM OUTER LOOP REPEATS CALCULATION WITH NEW INTEREST RATE
70 FOR J=1 TO 4
80    READ R
90    PRINT
100    PRINT
110    PRINT "CAPITAL SUM ACCUMULATED OVER 10 YEARS WITH";R;"% ANNUAL
115    PRINT "INTEREST RATE"
120    PRINT
130    PRINT "PERIOD","MONTHLY PAYMENT"
140    PRINT "YEARS","  5","  10","  15","  20"  - Note the way we leave
150    LET R=R/12/100                               spaces so that the
160    LET I=0                                      number of years is
170    LET C=P                                      central on the column.
175    REM LOOP TO PRINT OUT RESULTS AT TWO-YEARLY INTERVALS
180    FOR N=1 TO 5
190      REM LOOP TO ACCUMULATE CAPITAL AT YEARLY INTERVALS
200      FOR Y=1 TO 2
210        REM LOOP TO CALCULATE MONTHLY INTEREST
220        FOR M=1 TO 12
230          LET I=C*R+I
240          LET C=C+P
250        NEXT M
260        LET C=C+I                    Here are the variables on this page:
270        LET I=0                      C = capital accumulated
280      NEXT Y                         C1 = rounded interest (line 290)
290      LET C1=INT(C*100+.5)/100       I = interest amount (calculated monthl
300      PRINT Y*N,C1,2*C1,3*C1,4*C1    J = loop changing interest rate
310    NEXT N                           M = month
320 NEXT J                              N = number of two-year periods
330 DATA 5,8.5,9,9.5,10                 P = monthly payment made by investor
340 END                                 R = interest rate (line 150) changes
READY                                   annual percentage interest rate - (e.g
                                        10%) to monthly multiplier (e.g. 0.1/1
```

```
RUN

CAPITAL SUM ACCUMULATED OVER 1Ø YEARS WITH 8.5 % ANNUAL
INTEREST RATE

PERIOD          MONTHLY PAYMENT
YEARS
                  5             1Ø             15             2Ø
 2              135.86         271.72         4Ø7.58         543.44
 4              289.91         579.82         869.73         1159.64
 6              471.26         942.52         1413.78        1885.Ø4
 8              684.76         1369.52        2Ø54.28        2739.Ø4
 1Ø             936.Ø9         1872.18        28Ø8.27        3744.36


CAPITAL SUM ACCUMULATED OVER 1Ø YEARS WITH 9 % ANNUAL
INTEREST RATE

PERIOD          MONTHLY PAYMENT
YEARS             5             1Ø             15             2Ø
 2              136.51         273.Ø2         4Ø9.53         546.Ø4
 4              292.76         585.52         878.28         1171.Ø4
 6              478.41         956.82         1435.23        1913.64
 8              698.97         1397.94        2Ø96.91        2795.88
 1Ø             961.Ø2         1922.Ø4        2883.Ø6        3844.Ø8


CAPITAL SUM ACCUMULATED OVER 1Ø YEARS WITH 9.5 % ANNUAL
INTEREST RATE

PERIOD          MONTHLY PAYMENT
YEARS             5             1Ø             15             2Ø
 2              137.17         274.34         411.51         548.68
 4              295.64         591.28         886.92         1182.56
 6              485.65         971.3          1456.95        1942.6
 8              713.49         1426.98        214Ø.47        2853.96
 1Ø             986.66         1973.32        2959.98        3946.64


CAPITAL SUM ACCUMULATED OVER 1Ø YEARS WITH 1Ø % ANNUAL
INTEREST RATE

PERIOD          MONTHLY PAYMENT
YEARS             5             1Ø             15             2Ø
 2              137.82         275.64         413.46         551.28
 4              298.54         597.Ø8         895.62         1194.16
 6              493.Ø1         986.Ø2         1479.Ø3        1972.Ø4
 8              728.32         1456.64        2184.96        2913.28
 1Ø             1Ø13.Ø4        2Ø26.Ø8        3Ø39.12        4Ø52.16

END AT 34Ø
READY
```

For further practice you might like to think how you would rewrite the program; try using a single expression to calculate the new annual capital sum on the basis of the last annual capital sum, the interest rate, and the monthly payment - we do not provide an answer to this one.

LESSON 39

THE RESTORE STATEMENT

It is possible that you may want to perform many different operations with one set of data. The RESTORE statement allows you to do this.

You have already learnt that in a batch-processing system, all data input is by means of the READ and DATA statements.

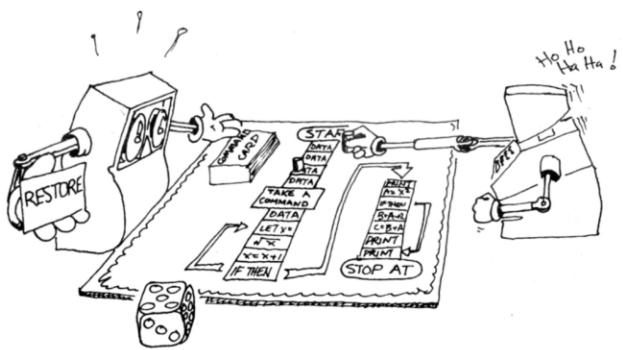
In an on-line system the INPUT statement is a convenient means of entering data, particularly where there are only a few variables, and where the variables and their values are frequently changed. However, the READ and DATA statements are often used, even in on-line systems, for constants where the value may be changed infrequently, or where a large quantity of data has to be processed.

BASIC provides an additional statement - RESTORE - which enhances the power of the READ and DATA statements.

The BASIC interpreter organises the contents of the DATA statements into a *data block*. During program execution a pointer is in effect moved down the data block, pointing to the next number not yet allocated to a variable. The RESTORE statement simply returns the pointer to the top of the data block. Hence the same set of data can be processed several times using the restore statement.

PROBLEM

Two dice are thrown 2Ø times and the value of each throw is recorded. Write a program which computes the average value of a throw, and then counts the number of throws which exceed 1½ times the average throw.



ANSWER

```
2 REM PROGRAM TO COMPUTE AVERAGE VALUE OF 2Ø THROWS OF TWO DICE
4 REM AND SUM THE NUMBER OF THROWS EXCEEDING 1.5 TIMES THE AVERAGE
6 REM     T. EILOART     31-MAY-99
8 REM FIRST LOOP CALCULATES AVERAGE THROW
1Ø LET S=Ø
2Ø FOR I=1 TO 2Ø
3Ø    READ N
4Ø    LET S=S+N
5Ø NEXT I
6Ø LET S=S/2Ø
7Ø LET A=S*1.5
8Ø RESTORE
82 REM DATA BLOCK NOW RESTORED, SECOND LOOP SCANS DATA AND
84 REM COUNTS NUMBER OF THROWS EXCEEDING 1.5 * AVERAGE
9Ø LET C=Ø
1ØØ FOR I=1 TO 2Ø
11Ø    READ N
12Ø    IF N<=A THEN 14Ø
13Ø    LET C =C+1
14Ø NEXT I
15Ø PRINT "AVERAGE VALUE OF THROW IS";S;"AND";C;"THROWS EXCEED";A
16Ø DATA 4,7,9,12,2,7,4,9,1Ø,5,11,6,6,2,9,12,3,8,1Ø,5
17Ø END
READY

RUN
AVERAGE VALUE OF THROW IS 7.Ø5 AND 3 THROWS EXCEED 1Ø.575

END AT 17Ø
READY
```

There is a more efficient way of solving this problem - but it requires a feature of BASIC - arrays - which we introduce in Lesson 42.

LESSON 40

DEBUGGING

Everyone makes mistakes - you probably will. This lesson tells you some of the ways in which you can get yourself out of trouble if your program breaks down.

It is almost inevitable that your first attempts at computer programming will contain errors, indeed it is rare for even experienced programmers to pass through the stages of problem definition, flowcharting, and program-coding without error. An important element in the programming process is therefore the *testing* and *debugging* of the coded program.

In Lesson 15 we described two types of programming errors: *syntax errors* and *run-time errors*. Syntax errors are typographical (typing errors) or grammatical (illegal variables, operations, or statements).

Fortunately the computer is very good at detecting syntax errors, and, during execution of the program it will print out a number of *diagnostic messages* which help you to locate and correct sources of errors. An example of an erroneous program and associated messages is given below - the program is intended to calculate T=(2Y+X)+Z.

```
1Ø PROGRAM CONTAINS FIVE ERRORS, ONE IS A RUN-TIME ERROR
2Ø READ X,Y
3Ø LET  T=2Y+X
4Ø READ Z
5Ø LET T=S+Z
6Ø PRINT "TOTAL = T
7Ø DATA 1Ø,2Ø
8Ø END
READY
```

(The errors in this program will be pointed out over the next six pages. See if you can spot them anyway!)

```
RUN
ILLEGAL INSTRUCTION IN 1Ø
ILLEGAL FORMAT IN 3Ø
ILLEGAL FORMAT IN 6Ø
READY
```

QUESTION

Can you spot the program errors referred to here in lines 1Ø, 3Ø, and 6Ø?

ANSWER

We correct the errors by typing in the new lines:

```
1Ø REM PROGRAM CONTAINS FIVE ERRORS, ONE IS A RUN-TIME ERROR
3Ø LET T=2*Y+X
6Ø PRINT "TOTAL =";T
```

We can now run the program again:

```
RUN

OUT OF DATA IN 4Ø          Insufficient data in the DATA statement which
READY                      we correct by typing in a new DATA statement

7Ø DATA 1Ø,2Ø,3Ø

RUN

TOTAL = 3Ø

END AT 8Ø                  Now the program is free of syntax errors and
READY                      is executed.
```

The diagnostic messages given above indicate fatal errors in the program. A *fatal error* is one which prevents execution. During the run the computer may generate other diagnostics which do not cause it to stop execution. An example of this type of diagnostic is the BAD DATA INTO LINE .... message introduced in Lesson 28. This class of diagnostic message indicates a non-fatal error. An example is shown below:

```
INPUT THREE NUMBERS
 ? 1,2,A
BAD DATA INTO LINE 2Ø... DATA A
 ?3
 1              2              3

END AT 4Ø
READY
```

This type of error occurs during program execution, that is at *run-time*, but is non-fatal since the computer continues to execute the program. A more serious *run-time error*, sometimes referred to as a semantic error, is an error in the structure and logic of the program. With such an error, the program will be accepted by the computer as a ' legal' BASIC program, will be executed, and a print-out produced. The programmer must now test the program to ensure that it is completely free from errors. This lesson is mainly concerned with the techniques which you can apply to test and debug your programs.

QUESTION

Turn back to the first program in this lesson and look at the result. Can you see anything wrong with the program?

HINT

What value has the variable named S in line 5Ø?

ANSWER

The program is intended to calculate:

T = (2Y + X) + Z
and line number 3Ø should be
3Ø LET S=2*Y+X

Even though the computer executes the program it is incorrect. There is an error in the *structure* of the program.

In fact, this program could be easily tested since we are able to supply data which will generate known results. This is a very effective means of program testing.

QUESTION

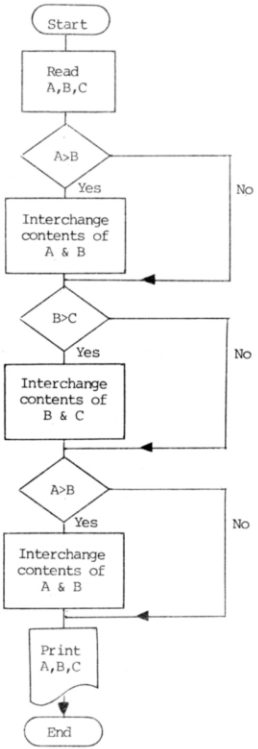
But before we go on to describe program testing techniques, are you sure that you understand the distinction between fatal and non-fatal errors?

ANSWER

A fatal error will prevent or abort execution of the program - whereas a non-fatal error will simply cause a diagnostic to be printed out.

In the simple examples which we have used in the text so far, the calculations can usually be performed manually, possibly with the assistance of a slide-rule or calculator. Selecting input data which generates known results does not present any significant problems. However, some care must be exercised in selecting suitable data for 'hand checking'.

Here is a flowchart we have used before. It is for the problem of sorting three numbers into ascending order. (See Part 2, page 19)


Start
Read
A,B,C
A>B
Yes
No
Interchange
contents of
A & B
B>C
Yes
No
Interchange
contents of
B & C
A>B
Yes
No
Interchange
contents of
A & B
Print
A,B,C
End

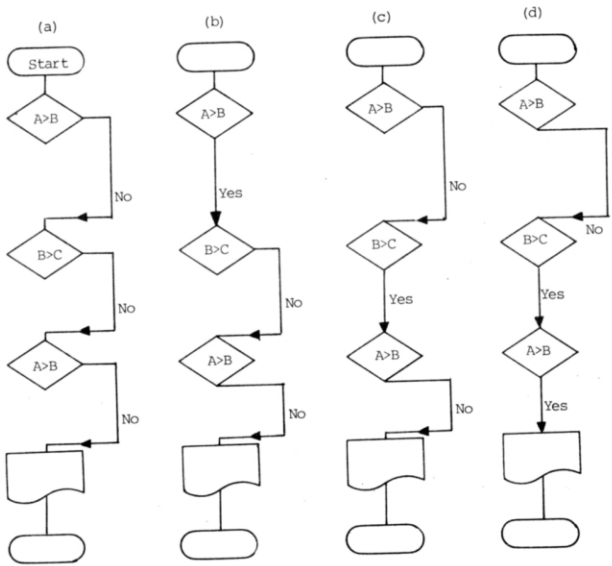
QUESTION

What data would you apply to this program to verify its operation?

ANSWER

(a) A = 1, B = 2, C = 3
(b) A = 2, B = 1, C = 3
(c) A = 1, B = 3, C = 2
(d) A = 2, B = 3, C = 1

The skeleton maps shown below indicate the paths that the computer takes through the program, using these sets of data:



Now these four sets of data not only test the program, but will also locate the process which is in error.

QUESTION

If the computer generates the following print-out in response to these four sets of data, which process is faulty, and can you deduce the error?

(a) 1, 2, 3,

(b) 1, 2, 3,

(c) 1, 2, 2,

(d) 1, 2, 1.

ANSWER

The second interchange of the value of variables named B and C is in error. The computer is setting the variable-name B equal to the value of variable-name C at this stage in the program. Here is the relevant section of the program:

INCORRECT	CORRECT
LET T = C	LET T = C
LET B = C	LET C = B
LET B = T	LET B = T

You may have found the location of this error difficult.

Sometimes programs are difficult to test using test data alone, or more often, a program is known to be in error, but the error proves difficult to locate simply by supplying test data. A technique called *tracing* is used in these circumstances.

To trace a program, the programmer inserts additional PRINT statements which supply the results of intermediate calculations, providing information on the actual flow of the program and the nature of the calculation. These intermediate results can then be checked by hand to enable the programmer to locate the source of an error.

It is often necessary to trace the logical flow of the program, and this is accomplished by inserting a PRINT statement before or after the IF...THEN... statements.

On the next page we show the complete number-sorting program used in the program testing example above; we have included the incorrect code also shown above.

```
1Ø REM PROGRAM TO SORT THREE NUMBERS INTO ASCENDING ORDER
2Ø REM          I.WILLIAMSON          31-MAY-99
3Ø READ A,B,C
4Ø IF A<B THEN 8Ø
5Ø LET T=A
6Ø LET A=B
7Ø LET B=T
8Ø IF B<C THEN 12Ø
9Ø LET T=C
1ØØ LET B=C
11Ø LET B=T
12Ø IF A<B THEN 16Ø
13Ø LET T=A
14Ø LET A=B
15Ø LET B=T
16Ø PRINT A,B,C
17Ø DATA 2,3,1
18Ø END
```

QUESTION

Where would you insert trace statements, and what would be printed out if the program was executed with data set (d); A=2, B=3 and C=1?

HINT

We suggest you require the program to print the values of the variables named A, B and C at certain points in the program.

ANSWER

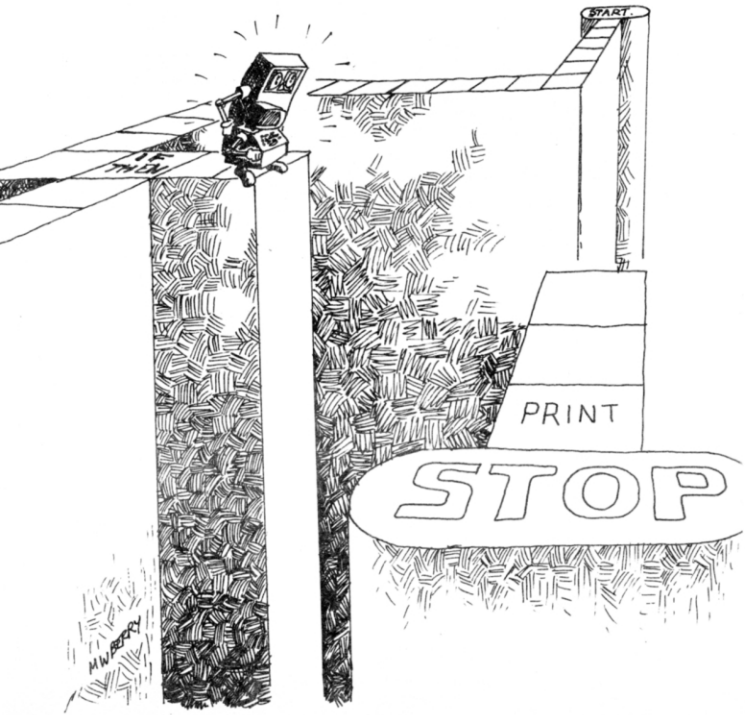
```
75 PRINT "TRACE 1",A;B;C
115 PRINT "TRACE 2",A;B;C
145 PRINT "TRACE 3",A;B;C
READY

RUN
TRACE 2          2   1   1
TRACE 3          1   1   1
 1               2               1

END AT 18Ø
```

The trace statements tell us just what route the computer has taken through the program, and where the error must have occurred.

Tracing can be a powerful tool for debugging programs. Indeed, it is worthwhile incorporating trace statements in your initial program, particularly if you are not confident of the logical flow of the program. But remember that tracing is no replacement for thorough program-testing with data that yields known results. Testing checks the total program; tracing helps you locate the errors.

Many complex programs must be tested and debugged with care and attention, since relatively simple errors can easily lead to infinite loops which are very costly in computer time. However, many novice programmers fall into the trap of tracing before thinking. Quite often one finds that an elusive error, which remains undiscovered after an intensive session at the computer terminal, stands out like a sore thumb when the program-listing is examined in the quiet of one's office or study.

If a program is in error, and the source of the error is not immediately apparent, *save* the program, take a clean listing, *log-off* the computer and retire to a quiet spot and carefully inspect the purpose of each statement.



LESSON 41

COMPUTER GAMES 1

Though computers are serious machines, and programming is a serious business, there is no reason why it cannot be fun as well. This program is quite complex, but once written, can be very entertaining.

Enough of the serious stuff; computers can be great fun especially for playing games.

PROBLEM

In the game of "pubsticks" (which some people call "Nim", though Nim is really much more complicated) there are two players. Each player, in turn, draws one, two, or three matches from a pile. There are 17 matches in the pile to start with. The last player to draw a match wins. The first player can always win. How? Try playing the game with a friend and see if you can work out the winning strategy. Look for winning positions when almost all the matches have been drawn.

If you cannot work out the strategy then try following this series of hints. At any time you feel you are able you should try to complete the thinking needed to find the winning strategy.

HINT 1

If it is the other player's turn, and there are four matches left, can I be sure to win? Remember the last person to draw a match is the winner.

ANSWER

Yes. However many matches my opponent draws I can draw all the rest. If opponent takes one, I take three, etc.

HINT 2

So I am sure to win. If it is my turn, and I want to leave only four matches, then how many must there be? I could win if there are 5. What other number could I win from?

ANSWER

I could win from 5, 6 or 7. I can draw one, two, or three respectively. Then I leave my opponent four to draw from. Look back to the *first hint* if you are not sure why I am then certain to win.

HINT 3

How can I leave the other player so that when it is my turn there are certain to be 5, 6 or 7 matches?
If he draws 1 from 8, I am left with 7 and I draw 3.
If he draws 2 from 8, I am left with 6 and I draw 2.
If he draws 3 from 8, I am left with 5 and I draw 1.
So, if I leave the other player 8 to draw from, I will win.
Now can you see the winning strategy?

HINT 4

Look at both the previous winning strategies
1) I win if the other player draws from 4
2) I win if the other player draws from 8
Can you *guess* the next number so that I will win if the other player draws from it?

HINT 5

You may have guessed that I will win if the other player draws from 4, 8, 12 or 16.
How can you be sure that I will win if I draw from any multiple of 4?

HINT 6

Imagine dividing the original heap of matches into a number of smaller heaps.

ANSWER

Imagine dividing the large heap into heaps of four. Whenever the opponent draws matches from one heap I draw the remaining matches from that heap, so I am bound to draw the last match from the last heap. If the opponent draws matches from more than one heap, then I rearrange the matches into as many heaps of four as I can, and draw *all* the matches from the heap with less than four.

QUESTION

Now let's imagine that the computer is to "play" this game.
First I draw up the statements that I would expect the computer to type out:
a) when it is my turn to draw
b) when I draw more than 3 matches or less than 1 (that is, I cheat)
c) when the computer draws matches
d) when the computer wins
e) when the computer is sure to lose
f) to ask if I want another game.

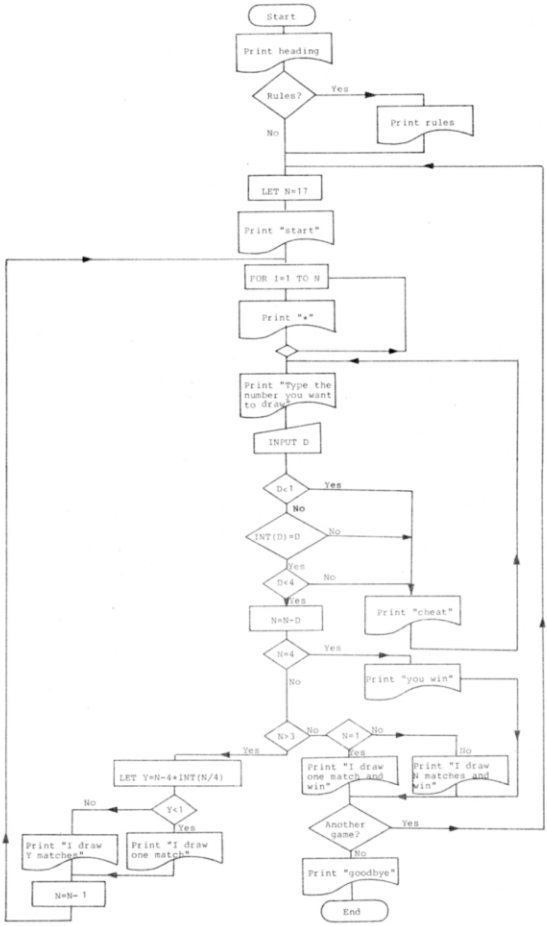
ANSWERS

(a) THERE ARE NOW .... MATCHES. YOUR TURN. TYPE THE NUMBER YOU WANT TO DRAW, EITHER 1, 2, OR 3.
(b) CHEAT! YOU MUST DRAW 1, 2 OR 3 MATCHES. TRY AGAIN.
(c) DRAW ... MATCHES. YOUR TURN (+ (a))
(d) DRAW ... MATCHES AND WIN. BAD LUCK! (+(f))
(e) YOU ARE BOUND TO WIN. (THERE ARE FOUR MATCHES LEFT) I CANNOT WIN, SO I RESIGN, WELL DONE (+(f)).
(f) IF YOU WANT ANOTHER GAME, TYPE 1 OTHERWISE TYPE Ø.

Now try to draw the flowchart and to write the program - remember that the computer never draws first. It could always win if it did.

N.B.

For our program we print an asterisk like this * for each match remaining when PRINT (a). Do the same in your program. This is more like the game in the pub, and it is less easy for a beginner to spot the winning strategy.


Start
Print heading
Rules?
Yes
Print rules
No
LET N=17
Print "start"
FOR I=1 TO N
Print "*"
Print "Type the number you want to draw"
INPUT D
D<1
Yes
No
INT(D)=D
No
Yes
D<4
No
Yes
Print "cheat"
N=N-D
N=4
Yes
Print "you win"
No
N>3
No
N=1
No
Yes
Yes
No
LET Y=N-4*INT(N/4)
Print "I draw one match and win"
Print "I draw N matches and win"
No
Y<1
Another game?
Yes
Yes
No
Print "I draw Y matches"
Print "I draw one match"
Print "goodbye"
N=N-1
End

```
10 REM  PUBSTICKS       T.EILOART & H.ASHTON        31-MAY-99
20 PRINT
30 PRINT "PUBSTICKS.  A GAME OF SKILL.  IF YOU WANT THE RULES"
40 PRINT "PLEASE TYPE 1, OTHERWISE 0";
50 INPUT Y
60 IF Y=0 THEN 120
70 PRINT "IN THE GAME OF PUBSTICKS, THERE ARE TWO PLAYERS.  YOU WILL"
80 PRINT "BE ONE, AND I, THE COMPUTER, WILL BE THE OTHER.  THERE ARE"
90 PRINT "17 MATCHES IN A PILE.  EACH OF US DRAWS IN TURN EITHER 1, 2,"
100 PRINT "OR 3 MATCHES.  THE LAST PERSON TO DRAW A MATCH WINS."
110 PRINT "YOU CAN ALWAYS WIN IF YOU PLAY WITH NO MISTAKES."
120 LET N=17
130 PRINT
140 PRINT "RIGHT - LET'S START!"
150 PRINT "THERE ARE NOW      ";
160 FOR I=1 TO N
170   PRINT "*";
180 NEXT I
190 PRINT "      MATCHES"
200 PRINT "YOUR TURN - TYPE THE NUMBER YOU WANT TO DRAW - ";
210 PRINT "EITHER 1,2, OR 3";
220 INPUT D
230 IF D<1 THEN 260
240 IF INT(D)<>D THEN 260
250 IF D<4 THEN 280
260 PRINT "CHEAT! YOU MUST DRAW 1,2,OR 3 MATCHES.  TRY AGAIN!"
270 GO TO 210
280 LET N=N-D
290 IF N=4 THEN 440
300 IF N<4 THEN 390
310 LET Y=N-4*INT(N/4)
320 IF Y<=1 THEN 360
330 PRINT "  I DRAW";Y;"MATCHES"
340 LET N=N-Y
350 GO TO 150
360 PRINT "  I DRAW ONE MATCH"
370 LET N=N-1
380 GO TO 150
390 IF N>1 THEN 420
400 PRINT "I DRAW ONE MATCH AND WIN.  BAD LUCK."
410 GO TO 460
420 PRINT "I DRAW";N;"MATCHES AND WIN.  BAD LUCK"
430 GO TO 460
440 PRINT "YOU ARE BOUND TO WIN.  THERE ARE FOUR MATCHES LEFT."
450 PRINT "I CANNOT WIN, SO I RESIGN.  WELL DONE!"
460 PRINT "IF YOU WANT ANOTHER GAME, TYPE 1, OTHERWISE 0";
470 INPUT G
480 IF G=1 THEN 120
490 PRINT
500 PRINT "GOODBYE!"
510 END
READY
```

```
RUN

PUBSTICKS.  A GAME OF SKILL.  IF YOU WANT THE RULES
PLEASE TYPE 1, OTHERWISE Ø ? 1
IN THE GAME OF PUBSTICKS, THERE ARE TWO PLAYERS.  YOU WILL
BE ONE, AND I, THE COMPUTER, WILL BE THE OTHER.  THERE ARE
17 MATCHES IN A PILE.  EACH OF US DRAWS IN TURN EITHER 1, 2,
OR 3 MATCHES.  THE LAST PERSON TO DRAW A MATCH WINS.
YOU CAN ALWAYS WIN IF YOU PLAY WITH NO MISTAKES.

RIGHT - LET'S START!
THERE ARE NOW      ****************      MATCHES
YOUR TURN - TYPE THE NUMBER YOU WANT TO DRAW - EITHER 1,2, OR 3 ? 2
  I DRAW 3 MATCHES
THERE ARE NOW      ************      MATCHES
YOUR TURN - TYPE THE NUMBER YOU WANT TO DRAW - EITHER 1,2, OR 3 ? 3
  I DRAW ONE MATCH
THERE ARE NOW      ********      MATCHES
YOUR TURN - TYPE THE NUMBER YOU WANT TO DRAW - EITHER 1,2, OR 3 ? 1
  I DRAW 3 MATCHES
THERE ARE NOW      ****      MATCHES
YOUR TURN - TYPE THE NUMBER YOU WANT TO DRAW - EITHER 1,2, OR 3 ? 2
I DRAW 2 MATCHES AND WIN.  BAD LUCK
IF YOU WANT ANOTHER GAME, TYPE 1, OTHERWISE Ø ? 1

RIGHT - LET'S START!
THERE ARE NOW      ****************      MATCHES
YOUR TURN - TYPE THE NUMBER YOU WANT TO DRAW - EITHER 1,2, OR 3 ? 1
  I DRAW ONE MATCH
THERE ARE NOW      ***************      MATCHES
YOUR TURN - TYPE THE NUMBER YOU WANT TO DRAW - EITHER 1,2, OR 3 ? 3
  I DRAW ONE MATCH
THERE ARE NOW      **********      MATCHES
YOUR TURN - TYPE THE NUMBER YOU WANT TO DRAW - EITHER 1,2, OR 3 ? 3
  I DRAW ONE MATCH
THERE ARE NOW      *******      MATCHES
YOUR TURN - TYPE THE NUMBER YOU WANT TO DRAW - EITHER 1,2, OR 3 ? 3
YOU ARE BOUND TO WIN.  THERE ARE FOUR MATCHES LEFT.
I CANNOT WIN, SO I RESIGN.  WELL DONE!
IF YOU WANT ANOTHER GAME, TYPE 1, OTHERWISE Ø ? Ø

GOODBYE!

END AT 51Ø
READY
```

LESSON 42

ARRAYS

There are six exams, each of which is attempted by all twelve pupils in a class. How many variables will you need to store all the examination results? Arrays help you round this problem, and solve others as well.

At the end of the school year, a class of 12 pupils sits school examinations in six subjects: Mathematics, English, French, Science, History, and Art. Each examination is nominally marked out of 1ØØ%, but wide variations in the standard of the questions produces wide variations in the marks scored in each subject. The class teacher has to take the examination results and translate them into grades 1, 2, 3 and 4 (or fail) for each subject, and produce an overall class position for each pupil.

The head teacher requires that these calculations are carried out to a standard procedure - as defined below:

1) For each subject, calculate the average examination mark of the 12 pupils.

2) Derive a new set of 'normalised' (i.e. adjusted) marks for each exam, thus:

   Multiply each exam mark, for a particular exam such as English, by the same number, N. (N may be a number such as 1.23, or 2, or Ø.895, or 1.Ø75, or 1.) N is chosen so that the top mark is now 1ØØ, or preferably (providing the top mark will not exceed 1ØØ) so that the average of the new set of marks will be 7Ø.

3) Grades are allocated according to the new mark; 4Ø% or below is a fail, above 4Ø% up to 6Ø% is a grade 3, above 6Ø% up to 8Ø% is a grade 2, and above 8Ø% is a grade 1.

4) To calculate the class positions, take the normalised marks as calculated above, and multiply the marks for Mathematics and English by 2, French and Science by 1½, and History and Art by 1. Add these marks together, and an additional mark out of 1ØØ for general class behaviour and attendance, giving a grand-total mark out of 1ØØØ. Class positions are determined from this total mark. Pupils with less than 4ØØ are to be expelled.

The school has just taken delivery of a new microcomputer with a BASIC interpreter. The maths teacher decides to write a program to take away the drudgery of all this arithmetic - how is he going to do it?
The key aspect of this problem, which is new to you, is the large quantity of data, 12 pupils and six subjects giving 72 marks to be manipulated into grades and overall positions. That is a lot of variables in BASIC.

For example, let's look at the marks for English:

Pupil number:	1	2	3	4	5	6	7	8	9	1Ø	11	12
Mark	2Ø	25	3Ø	35	4Ø	45	5Ø	Ø	4Ø	45	45	45

If we assigned a variable to each mark, the section of program to compute the average English mark might look like this:

```
1Ø READ A,B,C,D,E,F,G,H,I,J,K,L
2Ø LET Z=(A+B+C+D+E+F+G+H+I+J+K+L)/12
3Ø

4ØØ DATA 2Ø,25,3Ø,35,4Ø,45,5Ø,Ø,4Ø,45,45,45
```

We would soon run out of letters with six exam-subjects to process!

QUESTION

We could use A1, A2, A3 ..... instead of A,B,C ..... but can you see the objection to this?

HINT

There are only 1Ø digits.

ANSWER

With 12 pupils we have to use B1 and B2 as well. Our program now looks like this:

```
1Ø READ A1,A2,A3,A4,A5,A6,A7,A8,A9,AØ,B1,B2
2Ø LET Z=(A1+A2+A3+A4+A5+A6+A7+A8+A9+AØ+B1+B2)/12
3Ø

4ØØ DATA 2Ø,25,3Ø,35,4Ø,45,5Ø,Ø,4Ø,45,45,45
```

This program would work, but classes vary in size, and we would certainly have problems if there were 24 or 36 pupils.

The best solution to this problem would use *arrays* of variables to enter and process examination marks. We will examine the problem in a little more detail to see exactly what arrays are and how they are best applied.

The complete set of examination marks is shown in tabular form on the next page.

Pupil No	1	2	3	4	5	6	7	8	9	1Ø	11	12
English	2Ø	25	3Ø	35	4Ø	45	5Ø	Ø	4Ø	45	45	45
Maths	4	1Ø	1Ø	9	9	9	8	Ø	4	7	5	4
French	96	96	96	96	96	8Ø	8Ø	Ø	88	72	96	64
Science	4Ø	5Ø	6Ø	7Ø	8Ø	9Ø	1ØØ	Ø	8Ø	9Ø	9Ø	9Ø
History	8Ø	8Ø	Ø	Ø	2Ø	2Ø	2Ø	Ø	5Ø	5Ø	5Ø	5Ø
Art	2	2	2	2	2	2	2	Ø	2	2	2	1ØØ
Behaviour												

The rows of this table define the set of marks related to each subject, whereas the columns of the table define the set of marks for each pupil.

The number of pupils will vary, but the examination subjects will be constant, so we set out arrays around each subject. Separate arrays are formed for English, Maths, French, etc.; the number of variables in each array being varied according to the number of pupils in the class. This is an important characteristic of arrays in computer programming - arrays are used where the number of variables is uncertain, and varies according to the data being processed.

Let us now look at a program to read in the marks in English, and calculate the average mark.

```
1Ø DIM A(12)
15 LET Z=Ø
2Ø FOR I=1 TO 12
3Ø   READ A(I)
4Ø   LET Z=Z+A(I)
5Ø NEXT I
6Ø PRINT "AVERAGE IS",Z/12
7Ø DATA 2Ø,25,3Ø,35,4Ø,45,5Ø,Ø,4Ø,45,45,45
8Ø END
```

We will go through this program line by line.

```
1Ø DIM A(12)
```

The DIM statement reserves space in the computer memory for the array of variables. DIM stands for DIMension, and it specifies the maximum number of variables in the array. This statement says that there will be no more than 13 variables in the array, namely;

A(Ø), A(1), A(2), A(3), A(4), A(5), A(6), A(7), A(8), A(9), A(1Ø), A(11) and A(12).

In the program we use variables A(1) to A(12).

```
15 LET Z=Ø
2Ø FOR I=1 TO 12
```

We set up a FOR....NEXT loop with I incrementing from I=1 to I=12.

```
3Ø READ A(I)
```

Each time through the loop, an element of data is read from the DATA statement and assigned to the variable A(I). Thus A(1)=2Ø, A(2)=25, A(3)=3Ø, and so on.

```
4Ø LET Z=Z+A(I)
```

Z is the running total of the marks. Each time through the FOR....NEXT loop, another variable is added to the running total.

```
5Ø NEXT I
```

This completes the FOR....NEXT loop. The loop variable I is incremented until I=12.

```
6Ø PRINT "AVERAGE IS",Z/12
7Ø DATA 2Ø,25,3Ø,35,4Ø,45,5Ø,Ø,4Ø,45,45,45
8Ø END
```

The program is complete. The 12 marks for English are stored as A(1), A(2)... A(12), and the average mark has been calculated and printed out.

In similar manner we could enter and process the marks for Mathematics as B(1) to B(12), French as C(1) to C(12), and so on.

Now for the rules and regulations.

In BASIC a 'subscripted' variable is specified by a single letter followed by the 'subscript' in parentheses. Thus, up to 26 arrays of subscripted variables can be defined using the letters A to Z. The interpreter allocates computer storage to the arrays of variables, and must therefore be told of the *dimension* (size) of the array. In BASIC, dimension is specified using the DIM statement which must occur before the array is referenced.

For example:

```
5 DIM A(1ØØ)
```

will reserve 1ØØ locations for the array named A.

In fact the DIM statement is not required if the array contains fewer than 1Ø variables, since the interpreter automatically allocates storage for up to 1Ø variables when it encounters an array. However the DIM statement may be used to reduce the storage allocated, e.g.

```
5 DIM A(4),X(8),Y(6)
```

Here are some more facts about arrays.

a) An array is a set of variables given a single letter as a name and *subscripted* by a number, variable, or expression in parentheses. Thus

```
A(15Ø)
X(N)
Y(N+I)
Z(I+4*J)
```

are all permitted variables in BASIC programs.

b) A variable or an expression used as a subscript is truncated into an integer by the computer. For example,

P(5.21) P(5)

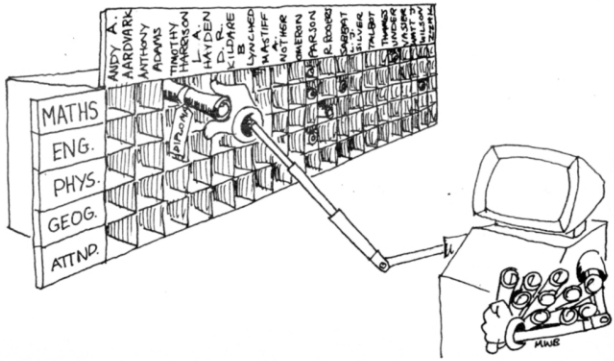
Q(16.95) Q(16)

c) Negative subscripts are not permitted.

d) If the letter A is used as an array name, it *should not* be used as an ordinary variable-name. This is because A(Ø) may be interpreted by the computer as the variable A, leading to error.

e) An array is assumed to be of dimension 1Ø unless it is redimensioned using a DIM statement. Note that any array of dimension 1Ø contains 11 variables if we include the variable of A(Ø). Specifying the size of an array in a DIM statement does not require the program to use the whole of the defined array.

f) The maximum permitted size of an array depends upon the computer system being used, and in particular the size of the computer memory. The smallest machines might typically support arrays of dimension 1ØØ to 2ØØ. Larger machines will support arrays ten times this size.



LESSON 43

MORE ON ARRAYS

Having learned the theory of arrays, we now proceed to a practical application.

We will now start to program the school teacher's problem. This is a long and complex program - there is no need to rush at it; we can write it in sections, testing and debugging each section in turn.

Starting just with the English marks, write a program which normalises the examination marks as follows:

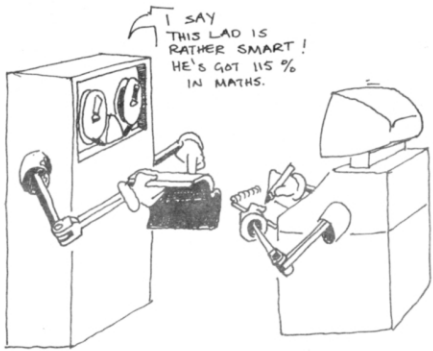
1) Calculate the average examination mark in each subject, for all 12 students. Add up all the 12 marks to make a total Z. The average is Z/12.

2) Divide the top mark T by the average mark Z/12. If the result is greater than 1ØØ then multiply all marks by 1ØØ/T. Here N is 1ØØ/T.

3) Otherwise multiply all marks by 7Ø/(Z/12) to produce a new set of marks where the average will be 7Ø. Here N is 7Ø/(Z/12).

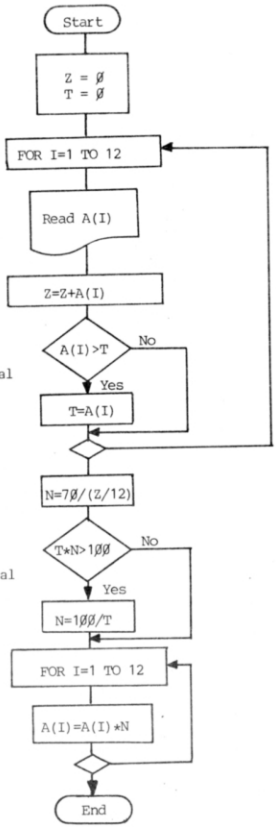
QUESTION
How many loops do you need?
ANSWER
Two, one loop to enter the data, calculate the average and top marks, and one loop to normalise the data.

PROBLEM
Now draw the flowchart.



QUESTIONS

ANSWERS



What is A(I)?

The Ith mark in the array of all 12 marks

What is Z?

Z is the cumulative total of the marks

What is T?

T is our top mark

What does this conditional jump do?

The conditional jump ensures that T always has a value equal to the highest mark yet seen.

What is Z/12?
What is N?

Z/12 is the average mark. N is 7Ø divided by the average mark.

What does this conditional jump do?

It ensures that no mark can be greater than 1ØØ

What does this loop do?

This loop either adjusts the average mark to 7Ø or scales up all the marks so the top mark is 1ØØ

Now write the program.

ANSWER

```
10 DIM A(12)
20 LET Z=0
30 LET T=0
40 FOR I=1 TO 12
50    READ A(I)
60    LET Z=Z+A(I)
70    IF A(I)<T THEN 90
80    LET T=A(I)
90 NEXT I
100 LET N=70/(Z/12)
110 IF T*N<=100 THEN 130
120 LET N=100/T
130 FOR I=1 TO 12
140    LET A(I)=A(I)*N
150 NEXT I
160 DATA 20,25,30,35,40,45,50,0,40,45,45,45
```

QUESTION

How would you test this program?

HINT

You need some print-out.

ANSWER

A further loop is required to verify that the new array has been correctly modified. We could simply add the following statements to print-out the array.

```
152 FOR I=1 TO 12
154    PRINT A(I);
156 NEXT I
READY

RUN
40  50  60  70  80  90  100  0  80  90  90  90
```

We have now completed the first section of our program. The English marks have been entered and processed according to the headmaster's rules. How would you process the marks for Mathematics, French, History, Science, and Art as well?

With your current knowledge of BASIC, you would have to repeat the section of code above for each subject to be processed. Some economy in coding can be achieved by interleaving the processing using common statements like lines 10, 40, 90, 130, 150, and 170; but the bulk of the code must be repeated for arrays A(I), B(I), C(I), D(I), E(I), and F(I) representing the subjects English, Mathematics, French, History, Science, and Art.

A much more efficient solution can be achieved using a *table* of variables. Here is a table of variables:

C_{11}	C_{12}	C_{13}	C_{14}	*1 and 4 are two subscripts*
C_{21}	C_{22}	C_{23}	C_{24}	
C_{31}	C_{32}	C_{33}	C_{34}	

A table is a two-dimensional array, and each number in the table is defined in BASIC by using the two subscripts in a subscripted variable.

The rules which apply to one-dimensional arrays apply equally to two-dimensional arrays or tables. For example, a table, T, is assumed to have dimension T(1Ø,1Ø) unless otherwise described in DIM statements.

An array T(1) and a table T(I,J) having the same array name are not permitted in the same program.

Using a table A(I,J) where I is the pupil number, and J is the subject number (J=1 for English, 2 for Mathematics, and so on), you can modify the section of program to process the examination marks quite simply.

QUESTION

Modify the program given above, using a table, to process all of the input data.

ANSWER

```
1Ø DIM A(12,6)
15 FOR J=1 TO 6        - this is the outer loop which takes us from
2Ø   LET Z=Ø             one exam subject to the next
3Ø   LET T=Ø
4Ø   FOR I=1 TO 12     - this is the inner loop which adjusts the
5Ø     READ A(I,J)       marks for each subject
6Ø     LET Z=Z+A(I,J)
7Ø     IF A(I,J)<T THEN 9Ø
8Ø     LET T=A(I,J)
9Ø   NEXT I
1ØØ   LET N=7Ø/(Z/12)
11Ø   IF T*N<=1ØØ THEN 13Ø
12Ø   LET N=1ØØ/T
13Ø   FOR I=1 TO 12
14Ø   LET A(I,J)=A(I,J)*N
15Ø   NEXT I
152   FOR I=1 TO 12
154     PRINT A(I,J);
156   NEXT I
158   PRINT
16Ø NEXT J
162 DATA 2Ø,25,3Ø,35,4Ø,45,5Ø,Ø,4Ø,45,45,45
164 DATA 4,1Ø,1Ø,9,9,9,8,Ø,4,7,5,4
166 DATA 96,96,96,96,96,8Ø,8Ø,Ø,88,72,96,64
168 DATA 4Ø,5Ø,6Ø,7Ø,8Ø,9Ø,1ØØ,Ø,8Ø,9Ø,9Ø,9Ø
17Ø DATA 8Ø,8Ø,Ø,Ø,2Ø,2Ø,2Ø,Ø,5Ø,5Ø,5Ø,5Ø
172 DATA 2,2,2,2,2,2,2,Ø,2,2,2,1ØØ
18Ø END
```

LESSON 44

THE TEACHER'S PROBLEM

A problem which could easily arise, and a way of solving it, using a computer.

You can now complete the teacher's problem.

QUESTION

What print-out is required?

ANSWER

For each pupil we want grades in each subject, the mark out of 1ØØ for behaviour and attendance, and the grand total weighted as described in Lesson 42.

QUESTION

What format will you adopt?

HINT

You have only five columns in the print-out.

ANSWER

There is no right or wrong answer. We suggest that you print out the results in two pages as shown below.

Page 1

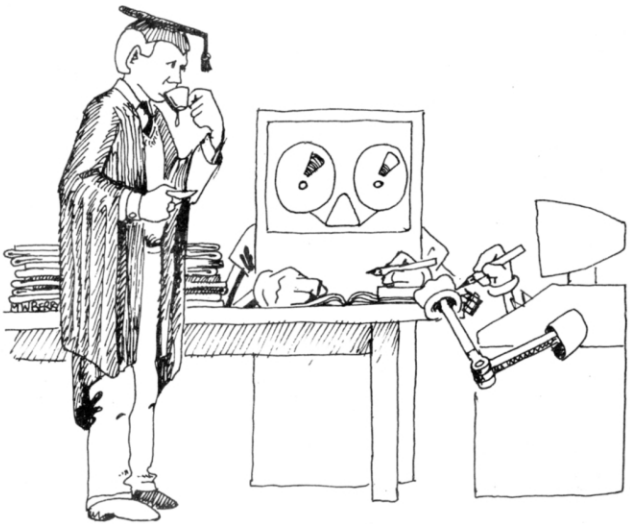
PUPIL NUMBER	ENGLISH	MATHS	FRENCH	SCIENCE
1				
2				
etc				

Page 2

PUPIL NUMBER	HISTORY	ART	BEHAVIOUR	TOTALS
1				
2				
etc				

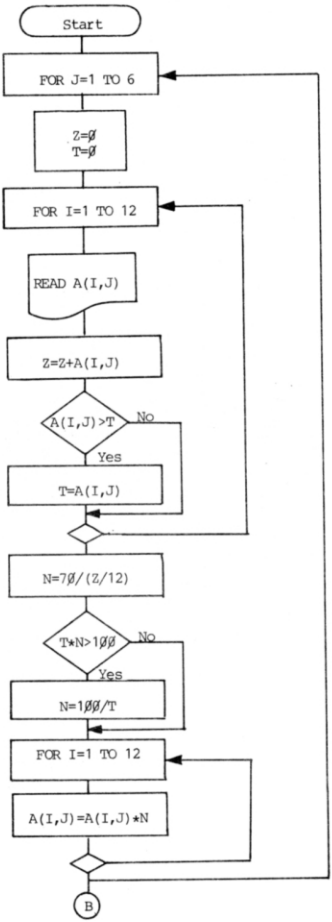
Write the program in four sections. The first section enters the marks in each subject and normalises them as discussed in Lesson 43. The second section enters the behaviour marks and computes the grand total. The third section replaces the marks in each subject for grades. The final section prints out the results in two pages.

Draw separate flowcharts for each section of program, and examine our solution to each section before going on to the next.



SOLUTION

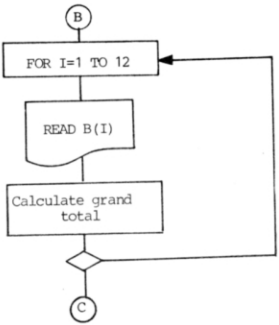
SECTION 1 : FLOWCHART



SECTION 1: PROGRAM

```
1Ø REM EXAMINATION RESULTS          I.WILLIAMSON     31-MAY-99
2Ø REM SECTION 1 PROCESSES DATA BY SUBJECT
3Ø DIM A(12,6),B(12),C(12)
4Ø FOR J=1 TO 6
5Ø   LET Z=Ø
6Ø   LET Z=Ø
7Ø   FOR I=1 TO 12
8Ø     READ A(I,J)
9Ø     LET Z=Z+A(I,J)
1ØØ      IF A(I,J)<T THEN 12Ø
11Ø      LET T=A(I,J)
12Ø   NEXT I
13Ø   LET N=7Ø/(Z/12)
14Ø   IF T*N<=1ØØ THEN 16Ø
15Ø   LET N=1ØØ/T
16Ø   FOR I=1 TO 12
17Ø     LET A(I,J)=A(I,J)*N
18Ø   NEXT I
19Ø NEXT J
2ØØ REM DATA BLOCK FOR RESULTS BY SUBJECT
21Ø DATA 2Ø,25,3Ø,35,4Ø,45,5Ø,Ø,4Ø,45,45,45
22Ø DATA 4,1Ø,1Ø,9,9,9,8,Ø,4,7,5,4
23Ø DATA 96,96,96,96,96,8Ø,8Ø,Ø,88,72,96,64
24Ø DATA 4Ø,5Ø,6Ø,7Ø,8Ø,9Ø,1ØØ,Ø,8Ø,9Ø,9Ø,9Ø
25Ø DATA 8Ø,8Ø,Ø,Ø,2Ø,2Ø,2Ø,Ø,5Ø,5Ø,5Ø,5Ø
26Ø DATA 2,2,2,2,2,2,2,Ø,2,2,2,1ØØ
```

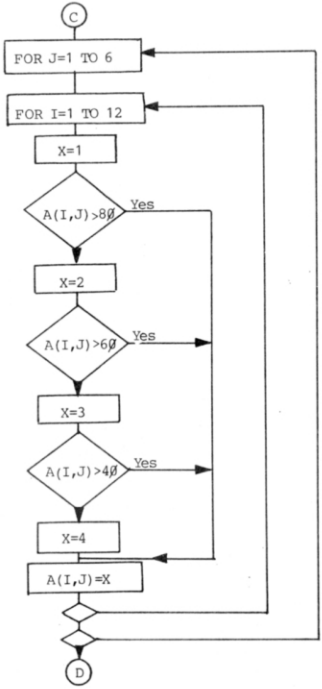
SECTION 2: FLOWCHART



SECTION 2: PROGRAM

```
27Ø REM SECTION 2 - CALCULATES GRAND TOTAL
28Ø FOR I=1 TO 12
29Ø    READ B(I)
3ØØ    LET C(I)=B(I)+2*(A(I,1)+A(I,2))+1.5*(A(I,3)+A(I,4))+A(I,5)+A(I,6)
31Ø NEXT I
32Ø REM BEHAVIOUR MARKS OUT OF 1ØØ
33Ø DATA 1Ø,2Ø,3Ø,4Ø,5Ø,6Ø,7Ø,Ø,Ø,8Ø,9Ø,1ØØ
```

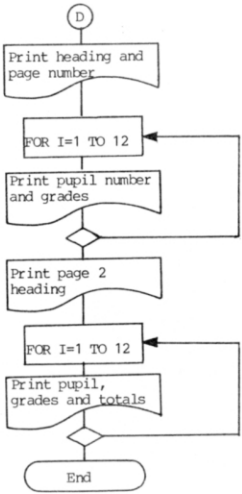
SECTION 3: FLOWCHART



SECTION 3: PROGRAM

```
34Ø REM SECTION 3 - REPLACES MARKS WITH GRADES
35Ø FOR J=1 TO 6
36Ø   FOR I=1 TO 12
37Ø     LET X=1
38Ø     IF A(I,J)>8Ø THEN 44Ø
39Ø     LET X=2
4ØØ     IF A(I,J)>6Ø THEN 44Ø
41Ø     LET X=3
42Ø     IF A(I,J)>4Ø THEN 44Ø
43Ø     LET X=4
44Ø     LET A(I,J)=X
45Ø   NEXT I
46Ø NEXT J
```

SECTION 4: FLOWCHART



SECTION 4: PROGRAM

```
47Ø REM SECTION 4 - PRINT OUT RESULTS
48Ø PRINT
49Ø PRINT "PAGE 1"
5ØØ PRINT
51Ø PRINT "PUPIL","ENGLISH","MATHS","FRENCH","SCIENCE"
52Ø FOR I=1 TO 12
53Ø   PRINT I,A(I,1),A(I,2),A(I,3),A(I,4)
54Ø NEXT I
55Ø PRINT
56Ø PRINT "PAGE 2"
57Ø PRINT
58Ø PRINT "PUPIL","HISTORY","ART","BEHAVIOUR","TOTAL"
59Ø FOR I=1 TO 12
6ØØ   PRINT I,A(I,5),A(I,6),B(I),C(I)
61Ø NEXT I
62Ø PRINT
63Ø END
RUN
```

```
RUN

PAGE 1

PUPIL        ENGLISH      MATHS        FRENCH       SCIENCE
 1            4            4            1            4
 2            3            1            1            3
 3            3            1            1            3
 4            2            1            1            2
 5            2            1            1            2
 6            1            1            2            1
 7            1            2            2            1
 8            4            4            4            4
 9            2            4            2            2
 1Ø           1            2            2            1
 11           1            3            1            1
 12           1            4            3            1

PAGE 2

PUPIL        HISTORY      ART          BEHAVIOUR    TOTAL
 1            1            4            1Ø           458
 2            1            4            2Ø           623
 3            4            4            3Ø           568
 4            4            4            4Ø           593
 5            4            4            5Ø           663
 6            4            4            6Ø           687
 7            4            4            7Ø           712
 8            4            4            Ø            Ø
 9            2            4            Ø            54Ø
 1Ø           2            4            8Ø           694
 11           2            4            9Ø           695.5
 12           2            1            1ØØ          741.5

END AT 63Ø
```

This is the longest program you have encountered in this course. If you are a little confused, don't worry, but do go back and examine the problem, and our solution, and satisfy yourself that you understand the major principles involved in the problem definition and solution. Remember that the coding of the program in BASIC is the least of your problems - it is good problem definition and solution to the stage of the flowchart which is the hallmark of a good programmer.

LESSON 45

BUBBLE-SORTING

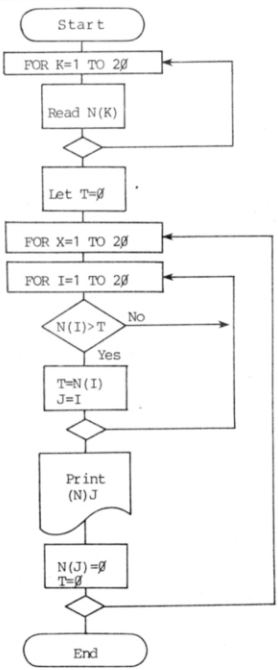
A picturesque name to describe a very useful technique. A sample problem is given for you to try.

Below we set a problem which will test your ability to use loops and arrays. This time there is no guidance, no hints - you are on your own.

PROBLEM

Write a program which takes in 2Ø non-zero, positive numbers and print them out in descending order. A program like this could be used to sort the examination results calculated in Lesson 44 into descending order.

SOLUTION



```
RUN

5 REM PROGRAM TO SORT 2Ø NUMBERS INTO DESCENDING ORDER
7 REM            I.WILLAMSON          31-MAY-99
1Ø DIM N(2Ø)
2Ø FOR I=1 TO 2Ø
3Ø   READ N(I)
4Ø NEXT I
5Ø LET T=Ø
6Ø FOR X=1 TO 2Ø
7Ø   FOR I=1 TO 2Ø
8Ø     IF N(I)<=T THEN 11Ø
9Ø     LET T=N(I)
1ØØ     LET J=I
11Ø     NEXT I
12Ø   PRINT N(J);
13Ø   LET N(J)=Ø
14Ø   LET T=Ø
15Ø NEXT X
16Ø DATA 2,5,8,13,2,6,3,5,42,31,23,24,7,12,14,1,41,24,8,15
17Ø END
READY

RUN
42  41  31  24  24  23  15  14  13  12  8  8  7  6  5  5  3  2  2
END AT 17Ø
READY
```

Is your solution the same as this one? Probably not, but the aspect of the solution which is important is the number of 'comparisons' (line number8Ø) required. In our solution line number 9Ø is executed 4ØØ times to sort 2Ø numbers into descending order. If you have more than 4ØØ comparisons in your program, then you should attempt to improve your solution.

However, a much better solution is available.

The problem of sorting numbers into ascending or descending order has been a recurrent example in this text from its first introduction in the discussion of flowcharting.

The method of *bubble-sorting* is a common solution to this problem. Suppose we have a list of numbers which we wish to re-arrange in ascending order. We compare the first pair of numbers, and interchange them if necessary. We then compare the second and third numbers, and so on down the list. In this way the largest number will propagate down the list and will end up at the bottom (hence the name bubble-sorting). Repeating the process will take the next largest down the list, although the last comparison is unnecessary since we know that the largest number is at the bottom. Thus, for a list of five numbers, we have four comparisons on the first pass, three on the second, two on the third and one on the fourth and last pass through the list. A bubble-sort is illustrated on the next page.

Original list	after 1st pass	after 2nd pass	after 3rd pass	after final pass
5	5	5	5	2
12	11	6	2	5
11	6	2	6	6
6	2	11	11	11
2	12	12	12	12

Bubble-sorting requires (4+3+2+1) comparisons to sort a list of 5 numbers into ascending or descending order.

For our problem with 2Ø numbers we have

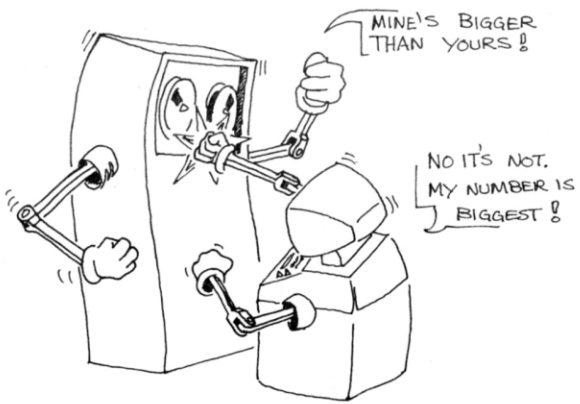
19 + 18 ..... +1 = 19Ø comparisons

instead of 2Ø*2Ø = 4ØØ comparisons in our first solution.

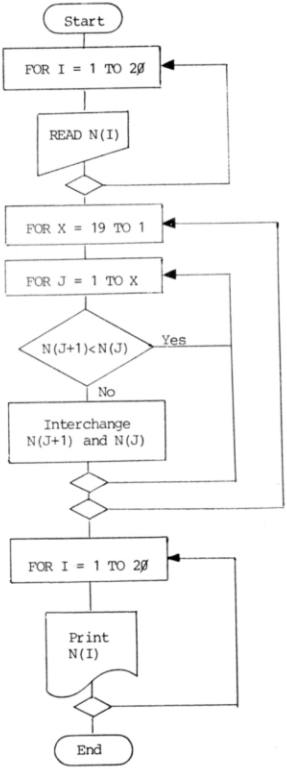
PROBLEM

Now write a new program to sort 2Ø numbers into ascending order using a bubble sort with 19Ø comparisons.

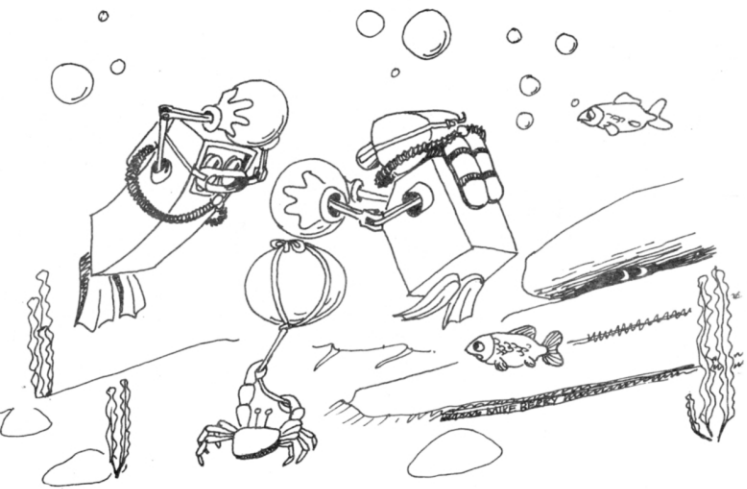
Draw a flowchart before coding your program.



SOLUTION: FLOWCHART



```
SOLUTION: PROGRAM

10 REM PROGRAM TO SORT 20 NUMBERS INTO DESCENDING ORDER
20 REM USING BUBBLE-SORT        I.WILLIAMSON   31-MAY-99
30 DIM N[20]
40 FOR I=1 TO 20
50   READ N(I)
60 NEXT I
70 FOR X=19 TO 1 STEP -1
80   FOR J=1 TO X
90     IF N(J+1)<N(J) THEN 130
100      LET T=N(J+1)
110      LET N(J+1)=N(J)
120      LET N(J)=T
130   NEXT J
140 NEXT X
150 FOR I=1 TO 20
160   PRINT N(I);
170 NEXT I
180 DATA 2,5,8,13,2,6,3,5,42,31,23,24,7,12,14,1,41,24,7,12
190 END
READY

RUN
 42  41  31  24  24  23  14  13  12  12  8  7  7  6  5  5  3  2  2  1

END AT 190
READY
```



LESSON 46

THE RND FUNCTION

The generation of random numbers may seem a pointless exercise, but in fact has many uses. Games can be played, using the computer as a die or dice, or even as a pack of cards.

In Part II we introduced 1Ø standard mathematical functions which are available in BASIC. These are repeated below.

FUNCTION	DESCRIPTION
SIN(X)	sine x, where x is measured in radians
COS(X)	cosine x, where x is measured in radians
TAN(X)	tangent x, where x is measured in radians
ATN(X)	arctangent x, where the value of ATN(X) is an angle, say θ, expressed in radians. Where $-\pi/2<=\theta<=\pi/2$
EXP(X)	exponential function , e^x
LOG(X)	natural logarithm of x; the sign of x is ignored
ABS(X)	absolute value of x (the value of x ignoring the sign)
SQR(X)	square root of x; the sign of x is ignored
INT(X)	integer part of x; the value of INT(X) is the largest integer not greater than x. For example, INT(5.9) = 5
SGN(X)	sign of x; +1 for x positive, Ø for x = Ø, -1 for x negative

Additional functions are available in some (but not all) versions of BASIC. One of the most useful is the RND function which generates a random number between Ø and 1 (but not including Ø and 1). A random number generator selects all numbers between Ø and 1 at random but with equal probability. It is like rolling a die with an almost infinite number of sides numbered between Ø and 1

The program below demonstrates the random number generator.

```
1Ø REM PROGRAM TO DEMONSTRATE RANDOM NUMBER GENERATOR
15 REM     I.WILLIAMSON        31-MAY-99
2Ø FOR I=1 TO 2Ø
3Ø   LET X=RND
4Ø   PRINT X,
5Ø NEXT I
6Ø END
READY

RUN
 .21132        .266713        .541468        .9Ø815         .745267
 .925689       .821691        .3142          .776455        7.86918E-Ø2
 .6Ø33Ø9       .219332        .3Ø7Ø16        .736366        .884865
 .62811        .34Ø135        .91Ø9Ø7        .763Ø97        2.32996E-Ø2

END AT 6Ø
READY
```

Note that, unlike the standard BASIC functions shown above, the RND function does not have an argument (a variable or expression in parentheses). However, in some versions of BASIC, RND has an argument (RND(X)) which has no meaning but is required to retain uniformity with other BASIC functions.

PROBLEM

Write a program which estimates the rolling of a die 1ØØ times and which produces a table giving the number of times each number was rolled.

HINT

The die value is generated by D=INT(6*RND+1), can you see why?

ANSWER

D=INT(6*.ØØØØØØ1 + 1) = 1

and D=INT(6*.9999999 + 1) = 1

hence the die values lie between 1 and 6

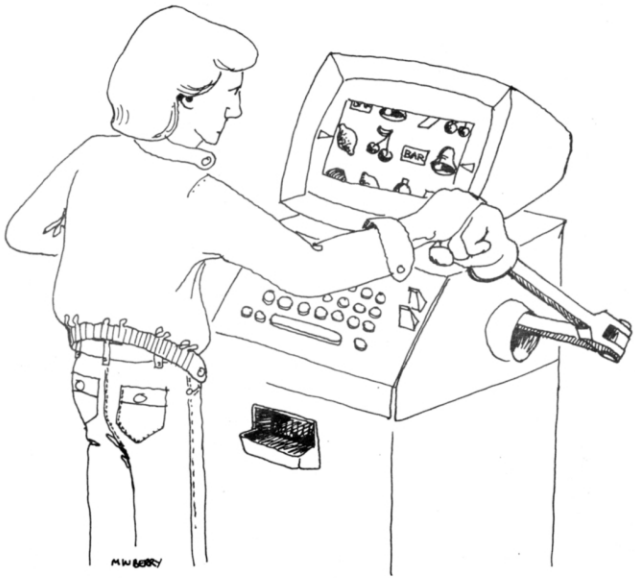
```
1Ø REM PROGRAM TO SIMULATE ROLLING OF A DIE 1ØØ TIMES
2Ø FOR I=1 TO 6
3Ø   LET N(I)=Ø
4Ø NEXT I
5Ø FOR J=1 TO 1ØØ
6Ø   LET D=INT(6*RND+1)
7Ø   LET N(D)=N(D)+1
8Ø NEXT J
9Ø PRINT "DIE VALUE","NO.OF THROWS"
1ØØ FOR I=1 TO 6
11Ø   PRINT I,N(I)
12Ø NEXT I
13Ø END
READY

RUN
DIE VALUE      NO.OF THROWS
 1              1Ø
 2              2Ø
 3              13
 4              2Ø
 5              16
 6              21

END AT 13Ø
READY
```

In generating numbers by digital means, as in a computer, an identical set of random numbers can be generated each time the program is executed. This possibility is avoided in BASIC using the RANDOMIZE statement which instructs the interpreter to select a random starting point for the random number generator. The RANDOMIZE statement must appear in the program before the random number generator RND is used. Thus the program example above requires an additional statement:

```
15 RANDOMIZE
```

to ensure that the die is thrown at random.

You might ask, why not make the random number generator RND always produce random numbers? The answer is that it helps in testing programs if the random number generator produces the same set of numbers each time the program is executed. Thus RANDOMIZE is omitted until the program has been fully tested.



LESSON 47

COMPUTER GAMES II

Using the RND function, we introduce another computer game; simple, but entertaining.

A simple computer game, which is quite entertaining, is called HI-LO. The computer generates a random number between 1 and 1ØØ; all you have to do is guess the number. Guess low and the computer prints LOW; guess high and the computer prints HIGH.

Here are a few reminders to help you write the program.

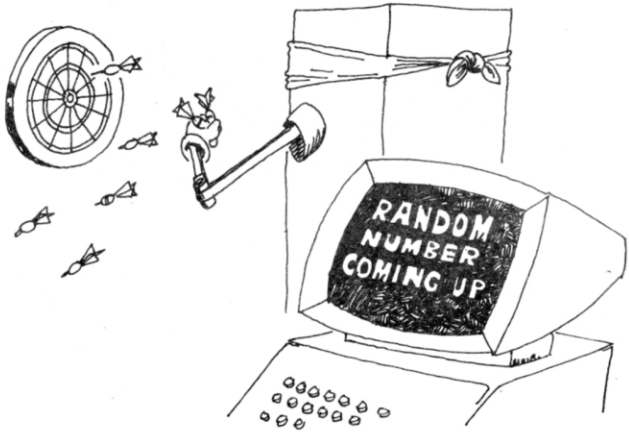
(i) Print out the rules for the new user.

(ii) Sort out how you want the computer to talk to you.

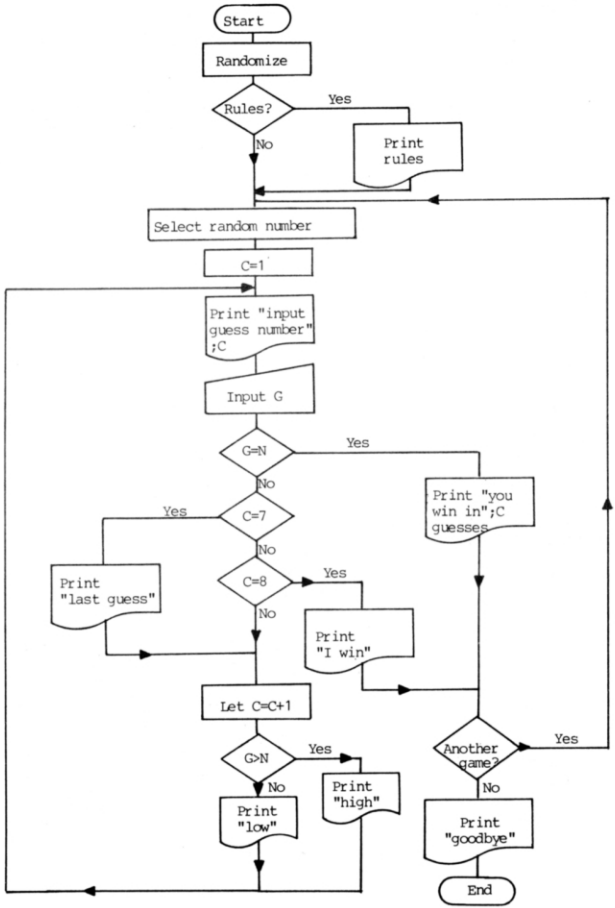
(iii) How many guesses are you allowed before the computer wins?

(iv) How will you generate the random number between 1 and 1ØØ?

No more hints - you're on your own!



Here is our program flowchart - does it match yours?



```
10 REM HI-LO PROGRAM    H.ASHTON      31-MAY-99
20 RANDOMIZE
30 PRINT "HI-LO - TYPE 1 FOR RULES, OTHERWISE 0";
40 INPUT T
50 IF T=0 THEN 90
60 PRINT "I SET A NUMBER BETWEEN 1 AND 1000.  YOU GUESS, AND"
70 PRINT "I TELL YOU IF YOU ARE HIGH OR LOW.  YOU HAVE 8"
80 PRINT "GUESSES TO GET THE RIGHT NUMBER."
90 LET N=INT(1000*RND(0)+1)
100 PRINT
110 LET C=1
120 PRINT "INPUT GUESS NUMBER";C;
130 INPUT G
140 IF G<>N THEN 170
150 PRINT "YOU WIN IN";C;"GUESSES.  WELL DONE!"
160 GO TO 280
170 IF C=7 THEN 250
180 IF C=8 THEN 270
190 LET C=C+1
200 IF G<N THEN 230
210 PRINT "HIGH, TRY AGAIN"
220 GO TO 120
230 PRINT "LOW, TRY AGAIN"
240 GO TO 120
250 PRINT "THIS IS YOUR LAST GUESS.  YOUR PREVIOUS GUESS WAS "; G
260 GO TO 190
270 PRINT "YOU HAVE HAD 8 GUESSES.  I WIN.  THE NUMBER WAS";N
280 PRINT "IF YOU WANT ANOTHER GAME, INPUT 1, OTHERWISE 0";
290 INPUT T
300 IF T=1 THEN 90
310 PRINT "GOODBYE"
320 END
```

*Note line 90, the random number is set by the statement INT(1000*RND(0)+1), which operates in the same way as the die simulator discussed in Lesson 46.*

```
RUN

HI-LO - TYPE 1 FOR RULES, OTHERWISE Ø ? 1
I SET A NUMBER BETWEEN 1 AND 1ØØØ.  YOU GUESS, AND
I TELL YOU IF YOU ARE HIGH OR LOW.  YOU HAVE 8
GUESSES TO GET THE NUMBER RIGHT.

INPUT GUESS NUMBER 1  ? 5ØØ
LOW, TRY AGAIN
INPUT GUESS NUMBER 2  ? 75Ø
LOW, TRY AGAIN
INPUT GUESS NUMBER 3  ? 825
HIGH, TRY AGAIN
INPUT GUESS NUMBER 4  ? 775
LOW, TRY AGAIN
INPUT GUESS NUMBER 5  ? 8ØØ
LOW, TRY AGAIN
INPUT GUESS NUMBER 6  ? 812
LOW, TRY AGAIN
INPUT GUESS NUMBER 7  ? 818
THIS IS YOUR LAST GUESS.  YOUR PREVIOUS GUESS WAS LOW,  TRY AGAIN
INPUT GUESS NUMBER 8  ? 823
YOU HAVE HAD 8 GUESSES.  I WIN.  THE NUMBER WAS 821
IF YOU WANT ANOTHER GAME, INPUT 1, OTHERWISE Ø ? 1

INPUT GUESS NUMBER 1  ? 5ØØ
HIGH, TRY AGAIN
INPUT GUESS NUMBER 2  ? 25Ø
HIGH, TRY AGAIN
INPUT GUESS NUMBER 3  ? 125
LOW, TRY AGAIN
INPUT GUESS NUMBER 4  ? 19Ø
HIGH, TRY AGAIN
INPUT GUESS NUMBER 5  ? 15Ø
LOW, TRY AGAIN
INPUT GUESS NUMBER 6  ? 17Ø
LOW, TRY AGAIN
INPUT GUESS NUMBER 7  ? 185
THIS IS YOUR LAST GUESS.  YOUR PREVIOUS GUESS WAS HIGH,  TRY AGAIN
INPUT GUESS NUMBER 8  ? 179
YOU WIN IN 8 GUESSES.  WELL DONE!
IF YOU WANT ANOTHER GAME, INPUT 1, OTHERWISE Ø ? Ø
GOODBYE

END AT 32Ø
READY
```

LESSON 48

THE TAB FUNCTION

This function, available in most versions of BASIC, allows you to print anywhere on the line. We use it to draw graphs of sine and cosine curves.

Another useful function which is available in some versions of BASIC is the TAB function. The TAB(N) function is used with the PRINT statement, and causes the teletypewriter carriage or VDU cursor to move to the Nth column of the print-out. This simple function allows the printer or display to be used to plot graphs and histograms.

Here is an example

```
1Ø REM PROGRAM TO PLOT SIN(X) BETWEEN X=Ø AND X=PI(3.14159)
2Ø REM INCREMENTS OF Ø.1 RADIANS
3Ø PRINT  TAB(15);
4Ø FOR I=1 TO 5Ø
5Ø   PRINT "-";
6Ø NEXT I
7Ø PRINT "-"
8Ø FOR X=Ø TO 3.14 STEP Ø.1
9Ø   PRINT X; TAB(INT(25*SIN(X)+Ø.5)+4Ø);"*"
1ØØ NEXT X
11Ø END
READY
```

```
RUN

Ø
.1
.2
.3
.4
.5
.6
.7
.8
.9
1
1.1
1.2
1.3
1.4
1.5
1.6
1.7
1.8
1.9
2
2.1
2.2
2.3
2.4
2.5
2.6
2.7
2.8
2.9
3
3.1

END AT 11Ø
READY
```

The TAB(N) function causes the printer to move across the page to the column defined by the integer part of the argument. The teletypewriter carriage cannot step backwards, and therefore the TAB function must always call up a column which is further along the line than the current position of the carriage. A print line contains approximately 75 characters.

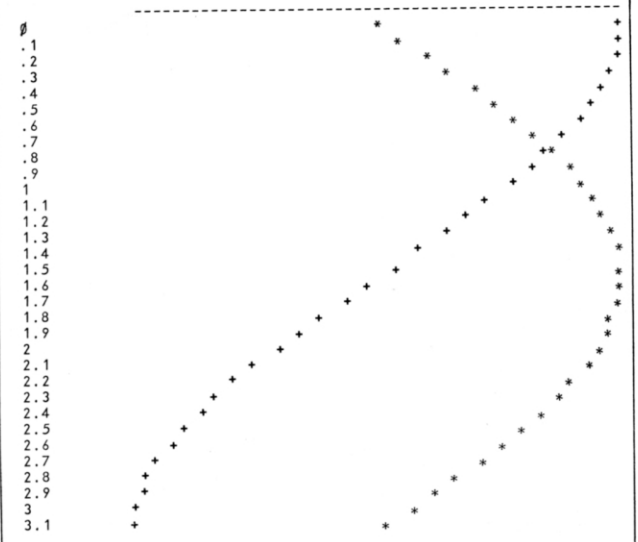
PROBLEM

Modify the plotter program given above to plot out sin x and cos x to the same scale. Use different symbols to represent each function.

SOLUTION

```
10 REM PROGRAM TO PLOT SIN(X) AND COS(X) BETWEEN X=0 AND X=PI
20 REM IN INCREMENTS OF 0.1 RADIANS, MARKING SIN(X) WITH A "*"
30 REM AND COS(X) WITH A "+".
40 REM          I.WILLIAMSON     31-MAY-99
50 PRINT  TAB(15);
60 FOR I=1 TO 50
70   PRINT "-";
80 NEXT I
90 PRINT "-"
100 FOR X=0 TO 3.14 STEP 0.1
110   LET S=INT(25*SIN(X)+0.5)+40
120   LET C=INT(25*COS(X)+0.5)+40
130   IF S<C THEN 160
140   PRINT X, TAB(C);"+"; TAB(S);"*"
150   GO TO 200
160   IF S=C THEN 190
170   PRINT X, TAB(S);"*"; TAB(C);"+"
180   GO TO 200
190   PRINT X, TAB(C);"X"
200 NEXT X
210 END
READY
```

```
RUN
```



```
Ø
.1
.2
.3
.4
.5
.6
.7
.8
.9
1
1.1
1.2
1.3
1.4
1.5
1.6
1.7
1.8
1.9
2
2.1
2.2
2.3
2.4
2.5
2.6
2.7
2.8
2.9
3
3.1

END AT 21Ø
READY
```

SUPPLEMENTARY LESSON 5

SIMPLE AND COMPOUND INTEREST

Simple and compound interest and the accumulation of capital.

If you borrow money from a bank, you will pay interest on the loan. If you save money in a bank, then the bank may pay you interest on the capital saved.

There are two ways of calculating the interest due - simple interest and compound interest.

In simple interest, the percentage annual interest rate is multiplied by the capital (the original amount), for the number of years the capital is saved or borrowed.

For example, if a capital sum of 1ØØ icu were lent for 4 years at an annual interest rate of 7%, the interest would be:

$$\text{Interest} = \frac{CR}{1ØØ} * T \quad \text{where } C = \text{capital, } R = \text{interest \% and } T = \text{period in years}$$

$$= \frac{1ØØ \times 7}{1ØØ} \times 4$$

$$= 28 \text{ icu.}$$

In compound interest, the interest due is added at regular intervals to the capital, and the next instalment of interest is calculated on the basis of the accumulated capital and interest.

Suppose in the example above, interest is accumulated into the capital annually. At the end of the first year, interest of 7 icu will be due on the capital of 1ØØ icu. For the second year this interest is accumulated into the capital - making a capital, or principal, of 1Ø7 icu. At the end of the second year, the interest due is 7.49 icu - the capital at the beginning of the third year is therefore 114.49 icu. At the end of four years the accumulated capital and interest is 131.Ø8 icu.

So to calculate compound interest you need to know both the interest rate and the compounding interval - the period after which interest is accumulated to the capital.

QUESTION

Using FOR...NEXT loops, write a section of a BASIC program which will calculate compound interest where R is the annual percentage interest rate and interest I and capital C are accumulated annually. Calculate the final sum after four years.

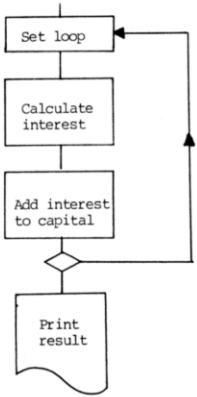
HINT

Use the FOR...NEXT loops for the years.

ANSWER

```
1ØØ FOR T=1 TO 4
11Ø REM WE ARE CALCULATING INTEREST FOR FOUR YEARS
12Ø LET I=(R*C/1ØØ)
13Ø REM WE HAVE PREVIOUSLY SET R AND C TO THEIR PROPER VALUES
14Ø LET C=C+I
15Ø REM THIS NOW ADDS THE INTEREST TO THE ORIGINAL CAPITAL
16Ø NEXT T
17Ø PRINT "FINAL AMOUNT IS";C;"ICU"
```

The flowchart for this is as follows:



Now return to Lesson 38.