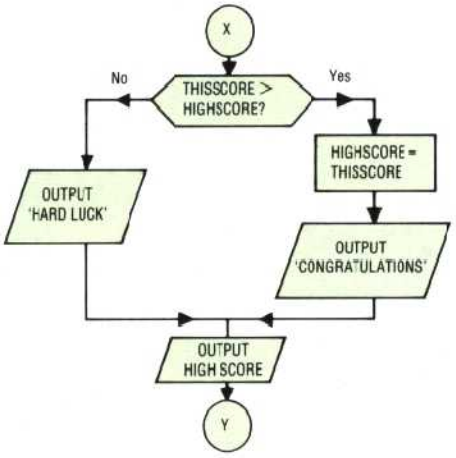
```
810 PRINT "AGAIN? (Y/N)";
820 INPUT RS
830 IF RS = "Y" THEN GOTO 100
840 END
```

We may wish to make a decision that will result in one of two distinctly different courses of action being followed. In the example shown below, we compare a player's game score to the highest previous score:




```
1200 IF THISSCORE > HIGHSCORE THEN GOTO
     1230
1210 PRINT "HARD LUCK. YOU HAVE TO BEAT";
1220 GOTO 1250
1230 LET HIGHSCORE = THISSCORE
1240 PRINT "CONGRATULATIONS! A NEW HIGH
     OF";
1250 PRINT HIGHSCORE
```

Note that the value of HIGHSCORE is printed in both events, and that the two possible flow paths rejoin in the process to become the single entry to this output operation.

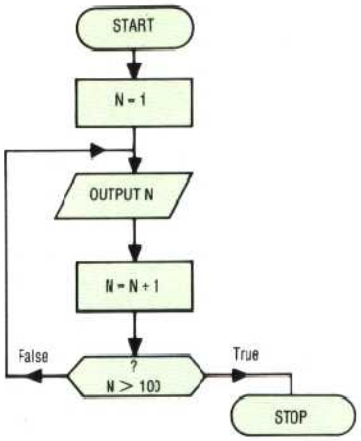
All decisions are taken as a result of tests similar to this, which deliver a positive or negative, a True or a False result. As you can see, this purely binary decision-making process denies the possibility of a 'maybe' answer. You can use whatever terms you wish, but don't forget to label the two exit paths accordingly!

All programming languages have an inherent decision statement which, if the True condition is satisfied, cause a conditional branch, but which drops control through to the next statement if the result is False. In the case of a dialect of BASIC that allows only a simple IF-THEN, we must mimic the conditional branch by means of a GOTO statement, as in line 1200 of the last example. The statement in line 1210 will only be executed if the result of the test in line 1200 is False.

But what about the second use of GOTO in line 1220? As you can see, the use of GOTO at the end of the test, to solve the problem of the destination of the conditional branch, has forced us to use this method to 'join up' the two possible control paths again, in this case at line 1250.

The use of flowcharts usually encourages the introduction of GOTOs as a means of following the point to point graphical representation of the program. In general, this use of unconditional jumps is rather dangerous. If the version of BASIC that is being used forces this solution, then a flow diagram is an excellent method of assessing the way in which control passes out of the program's normal succession.

Let's use one last example to examine how the use of a flowchart allows us to represent accurately the necessary steps to perform a simple task: printing out all the numbers between one and one hundred.




```
10 LET N = 1
20 PRINT N
30 LET N = N + 1
40 IF N > 100 THEN END
50 GOTO 20
```

The use of flow diagrams in this way tends to encourage a step by step approach to program writing which, especially in larger projects, often leads to a rather inelegant result. For those with even a passing knowledge of the BASIC language, the use of a FOR-NEXT loop is obviously indicated. For example:

```
10 FOR N = 1 TO 100
20 PRINT N
30 NEXT N
40 END
```

The flowchart is incapable of representing this piece of BASIC 'shorthand', and to follow it exactly would lead one to a less efficient way of solving the problem. It does, however, give us some information on the structure of the FOR-NEXT loop, and so is of value when we come to examine this and other BASIC functions, to determine how they are constructed.

Flow diagrams are particularly useful during the planning or conceptualising stage of programming, especially in the 'tricky' parts. Experienced programmers tend to use them less than beginners, and will often resort to a flow diagram to illustrate and document a piece of software written without their aid. But whether a flow diagram is drawn out on paper, or it simply exists inside the programmer's subconscious mind, the concept of charting the flow of information and control is central to the use of computers as a problem-solving tool.