not be ambiguous in any way. Ambiguity is easy to introduce at an early stage when the algorithm is being written down in English. Words like 'and' and 'or' in English are very different from the AND and OR of Boolean logic. For example, if the algorithm was meant to select all the names in a list that begin with an 'A' *and* all those beginning with 'B', you could easily write code like:

```
IF FIRSTLETTER="A" AND FIRSTLETTER="B" THEN
```

which is wrong because a logical OR is needed!

The criterion of effectiveness is a demand that the program should not contain impossible instructions. An instruction is said to be effective if it can be done with a pencil and paper in a finite time. This means that instructions like 'let X equal the highest prime number' are not effective (there isn't a highest prime number).

GENERAL CONSIDERATIONS

There are also criteria to judge the algorithm as a whole. An algorithm must *terminate*. The algorithm that follows does not terminate (even though its instructions are definite and effective) and if this was coded into a program it would endlessly loop:

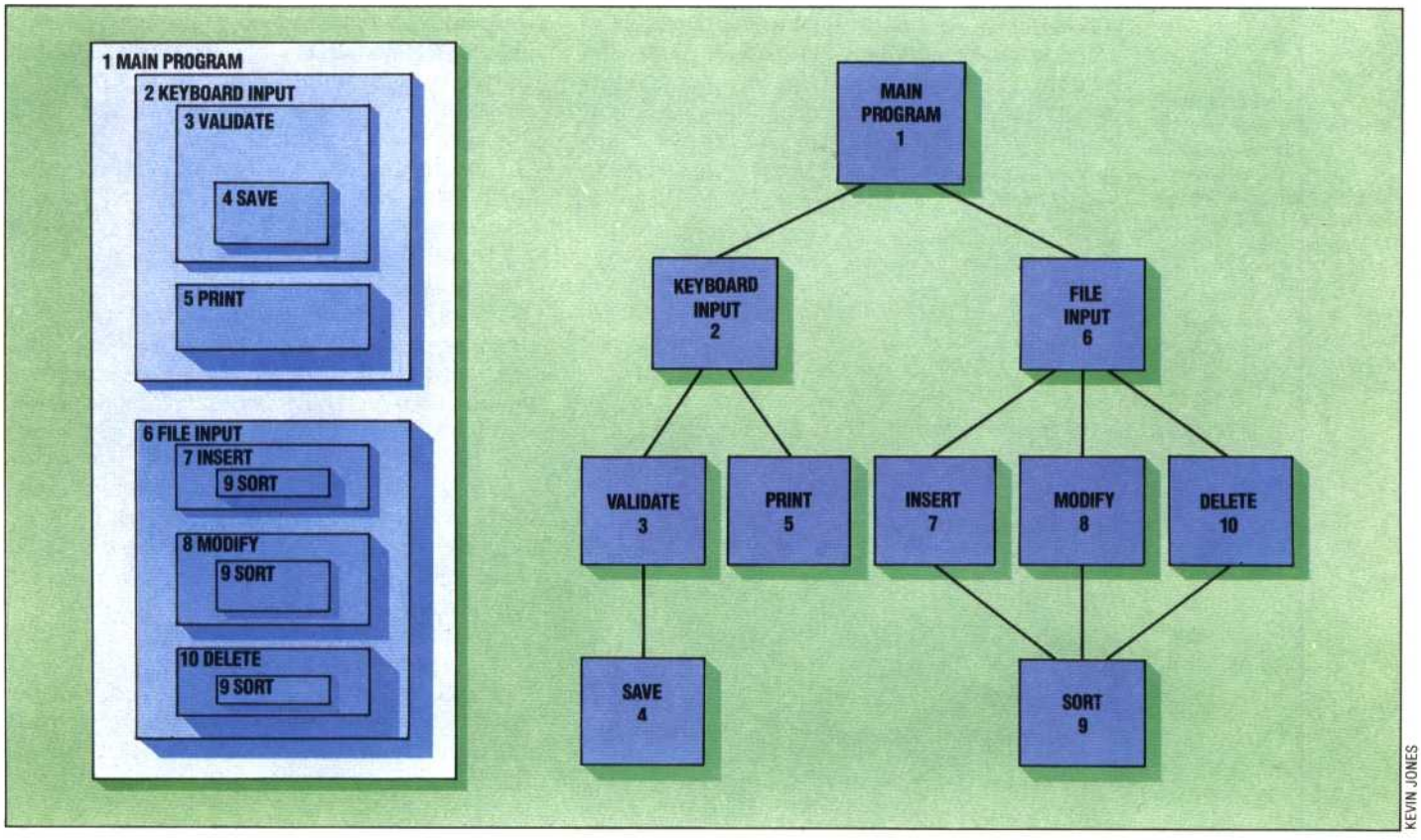
```
step 1 let I equal 1
step 2 if I > 3 then exit
step 3 goto step 1
```

Telling whether an algorithm will terminate is not always easy, but, in general, algorithms that involve loops test for a particular condition before they terminate (e.g. if I > 3 in our example), and it is necessary to check that it is possible to meet that condition.

Efficiency, generality and *elegance* are ways of judging between different algorithms. Efficiency is usually judged in terms of time and memory use. The two are usually quite compatible — fast code may need relatively little space, but bear in mind that this need not be so. Having found an algorithm, it can be 'tuned' for efficiency by changing its details. A calculation will be noticeably faster and will use less memory if, for example, integer rather than floating point arithmetic is used. Alternatively, a completely different algorithm for doing the same thing could be found.

Generality is the ability of an algorithm to cope with many different situations apart from the one for which it was designed. It is worth while, in the long run, to attempt to make all algorithms as general as possible. If a program called for a yes/no response several times, it would be worth writing a routine that prompts the user with 'please type y or n', accepts the input, checks whether it is 'y' or 'n', reprompts if it is neither and otherwise returns the appropriate response. However, the routine could be made more general if it could be fed with different prompts and potential replies, so it could be used in many different situations. Elegance means finding algorithms that are both simple and ingenious. In all cases it is more sensible to find efficient, general algorithms rather than elegant ones.

Another important aspect of algorithms is the flow of control and of data within them and how this can be represented with flow charts. This is the subject of the next instalment in this series.



KEVIN JONES

Pyramids And Primitives
The block-structure diagram on the left clearly shows the nesting of a program's algorithms, while the procedure flow diagram on the right emphasises the articulations and process levels of the same program. The most 'primitive' algorithms are the most deeply nested, and the lowest in the hierarchy