computer systems, though there are occasions when the criteria by which data is accessed are so unpredictable that a pile is as good a data structure as any.

A more organised data structure, and one much easier for both people and computers to use, is achieved when the data is organised according to a recognised and simple system. A telephone directory is a good example of a set of information (names, addresses and telephone numbers) where the name field is ordered according to simple rules of alphabetic sequencing. The numbers themselves are, to all intents and purposes, randomly ordered, but the names — which are more 'meaningful' — are organised according to easy-to-follow rules.

Inasmuch as we have thought about the internal organisation of the data in our computerised address book, the data is organised as a pile, with one record being stored in name array element X, street array element X and so on, and the next record being stored in name array element X+1, street array element X+1 and so on. Finding a particular item of data — BILL SMITH, for example — would therefore involve looking at the first element in the name array and seeing if it was BILL SMITH, looking at the second element and seeing if it was BILL SMITH and so on until we had either located the field or discovered that there was no entry for BILL SMITH.

If the data we want to search for has already been ordered into a recognisable structure, we can see how it will simplify the search. Suppose you have a database on football teams, and one of the fields in the records is the score for a particular week. A powerful database might allow you to find which team or teams had scored 11 goals in that week. Here is the array holding team scores for the week in question:

```
1,6,2,2,1,9,0,0,2,1,4,11,4,2,12,5,2,1,0,1
```

It should be obvious that the scores are in team order and not in score order. Twenty teams are involved and only one team actually managed to score 11 goals that week. This was the 12th team entered in the array. With unstructured data like this, the only way to find the information you want is to look at the first element and see whether it was 11; if it was not, look at the next element to see whether it was 11, and so on until either an 11 was located or no element equal in value to 11 was found.

If we analyse this data, we will see that there was a total of 20 scores, ranging in value from 0 to 12. This example is relatively trivial, and even if we had to search through every item it would not take long to discover that 11 was in the 12th element of the array. But what if there were thousands of elements in a large array? Searching through numerous unstructured data items could slow down a program to an undesirable extent.

The solution is to order the data first, so that searches can take place far more quickly. Here is the array of scores again, arranged in numerical order:

```
0,0,0,1,1,1,1,1,2,2,2,2,2,2,4,4,5,6,9,11,12
```

If we know the number of teams is 20, then the quickest way to find the position of the score we want is to split the array into two parts and search only the part likely to contain the number we want. Remember that sifting through large quantities of data is likely to take far more time than simple arithmetic operations such as dividing a number by two. The algorithm for locating the score would now look like this:

```
Find the array containing the scores
Read the number we want to search for
Find the length of the array
Find the midpoint of the array
Loop until the number is located
   If the item at the midpoint is equal to the
   number we are searching for, then the
   number has been located
   If not, see if the number sought is larger or
   smaller than the number at the midpoint
   If the number sought is larger than the
   number at the midpoint, then find the
   midpoint of the upper part of the array
   If the required number is smaller than the
   number at the midpoint, then find the
   midpoint of the lower part of the array
   (Repeat this until the number is located)
```

This can be formalised to:

```
BEGIN
   Find the array of scores
   INPUT NUMBER (to be searched for)
   LOOP until the number is located
      IF NUMBER = (midpoint)
         THEN note position of midpoint
         ELSE
            IF NUMBER > (midpoint)
               THEN find midpoint of upper half
               ELSE find midpoint of lower half
            ENDIF
      ENDIF
   ENDLOOP
   IF NUMBER is located
      THEN PRINT position of midpoint
      ELSE PRINT "NUMBER NOT FOUND"
   ENDIF
END
```

If you think through this program in pseudo-language you will see that it cannot fail eventually to locate the number being searched for if it exists in the array. Let's develop this pseudo-language until we can arrive at a working program. This process of searching by repeated subdivision is called a 'binary search'.

A program in BASIC based on the pseudo-language above is presented for you to try. It creates an array and reads in the scores from a data statement. It then prompts for the score to be searched. If it finds the score, it prints the element of the array the number was found in.