

	1	2	3	4
1	t	h	i	s
2	w	a	t	s
3	o	a	h	g
4	f	g	d	t

Figure 1.1 Sample word puzzle

Chapter 1, page 2

A second problem is to solve a popular word puzzle. The input consists of a two-dimensional array of letters and a list of words. The object is to find the words in the puzzle. These words may be horizontal, vertical, or diagonal in any direction. As an example, the puzzle shown in Figure 1.1 contains the words *this*, *two*, *fat*, and *that*. The word *this* begins at row 1, column 1, or (1,1), and extends to (1,4); *two* goes from (1,1) to (3,1); *fat* goes from (4,1) to (2,3); and *that* goes from (4,4) to (1,1).

Again, there are at least two straightforward algorithms that solve the problem. For each word in the word list, we check each ordered triple (*row*, *column*, *orientation*) for the presence of the word. This amounts to lots of nested `for` loops but is basically straightforward.

Alternatively, for each ordered quadruple (*row*, *column*, *orientation*, *number of characters*) that doesn't run off an end of the puzzle, we can test whether the word indicated is in the word list. Again, this amounts to lots of nested `for` loops. It is possible to save some time if the maximum number of characters in any word is known.

It is relatively easy to code up either method of solution and solve many of the real-life puzzles commonly published in magazines. These typically have 16 rows, 16 columns, and 40 or so words. Suppose, however, we consider the variation where only the puzzle board is given and the word list is essentially an English dictionary. Both of the solutions proposed require considerable time to solve this problem and therefore are not acceptable. However, it is possible, even with a large word list, to solve the problem in a matter of seconds.

An important concept is that, in many problems, writing a working program is not good enough. If the program is to be run on a large data set, then the running time becomes an issue. Throughout this book we will see how to estimate the running time of a program for large inputs and, more important, how to compare the running times of two programs without actually coding them. We will see techniques for drastically improving the speed of a program and for determining program bottlenecks. These techniques will enable us to find the section of the code on which to concentrate our optimization efforts.

We close this chapter by returning to the word puzzle problem of Chapter 1. If the second algorithm described in Chapter 1 is used, and we assume that the maximum word size is some small constant, then the time to read in the dictionary containing W words and put it in a hash table is $O(W)$. This time is likely to be dominated by the disk I/O and not the hashing routines. The rest of the algorithm would test for the presence of a word for each ordered quadruple (*row, column, orientation, number of characters*). As each lookup would be $O(1)$, and there are only a constant number of orientations (8) and characters per word, the running time of this phase would be $O(R \cdot C)$. The total running time would be $O(R \cdot C + W)$, which is a distinct improvement over the original $O(R \cdot C \cdot W)$. We could make further optimizations, which would decrease the running time in practice; these are described in the exercises.