Problem size n	Solution time (seconds)
25	2.8
26	5.6
27	11.3
28	22.5
29	45.1
30	89.8

Table 9.1: Time taken by function `subsetsum` (Figure 9.4) for various values of `n`, measured in CPU seconds on a 700 MHz Intel Pentium III. In each case `k` is chosen so that no subset exists and the value returned is zero.

One problem with generate and test-style algorithms is also brought out in Figure 9.4, and that is the cost of exhaustive enumeration. To assert that no subset with the right sum is possible, every combination of the `n` items must be calculated, and there are an exponentially large number of such combinations. Table 9.1 documents the cost of searching for a subset that does not exist. Each time `n` is increased by one, the time taken to decide the outcome doubles. Unfortunately, this growth rate means that large problems cannot be solved – even though the program exists, and is just a dozen or so lines long. For example, extrapolating from the times in the table, a problem with `n` equal to 40 will require approximately one day of computation; a problem with `n` equal to 50 will take more than three *years* of non-stop calculation; and a problem with `n` equal to 60 will requires more than three *thousand* years. As for a problem of size 100, the mind just boggles – it simply can't be done.

In Chapter 1 an analogy was drawn between physics, the science of energy, and computer science, which is the study of information. From physics, we know that perpetual motion is an impossibility; and from chemistry we know that transmutation of lead into gold is an unrealistic aim. So too in computing it appears that there are things that are extremely hard to achieve.

Unfortunately, there are a large number of other problems like the subset sum problem, for which the only known algorithms require exponentially growing time. Finding ways to obtain partial or approximate solutions to non-trivial instances of these problems is one of the key challenges in Computer Science.

There are many problems for which all known algorithms require exponentially growing time. It is also likely that no fast algorithms will ever be found, making the exact solutions to even mid-sized instances of these problems impossible to obtain.

The number of moves required by a towers of Hanoi problem is also exponential, this time in n, the number of disks. Now the high cost arises because there are that many moves necessary to accomplish the transformation, and not because different move sequences have to be explored to find a "right" one. A generate and test approach to the towers of Hanoi problem might never end, as there is no obvious ordering in which to generate possible move sequences, and then evaluate them.

The original setting of the towers of Hanoi problem is said to be in a remote mountainous temple, with generation after generation of dedicated monks manually working with a stack of 60 disks. The lore goes on to say that when the last disk is moved into its final position, it will mark the end of the world. Based upon the estimates made above, and the fact that computers are around ten million times faster than humans, we are probably safe for a few more years yet.

The situation is not so gloomy with respect to other problems such as sorting. Even the "agricultural" bubble sort is such that arrays containing thousands of items can be sorted in a few dozens of seconds, and the sorting algorithms introduced in Chapter 12 are faster still. Several of those techniques exploit the divide and conquer strategy to obtain their efficiency.

One particular type of divide and conquer algorithm is worth attention, and that is the *greedy* heuristic. The underlying assumption of the greedy heuristic is that a solution that is good overall can be found by making a sequence of choices at a low level, each of which maximizes some simple definition of progress. For example, selection sort (Exercise 7.6 on page 130) is a greedy process, since at each stage the biggest remaining item is located, and swapped into its correct position. By being greedy at each individual step, a solution is arrived at to the larger problem.

The greedy heuristic seeks a globally good solution to a problem through the use of locally maximal choices.

9.3 Simulation

Consider the following game of chance. To enter the game, contestants pay \$1. They then roll two dice. If the total on the dice is eight or more, they are paid their original stake, plus another \$1; except that if the total is twelve, they are paid back their original stake, plus an additional \$5. On the other hand, if the total is less than eight, they get nothing back.

Suppose a player enters the game with a \$5 initial float, and plays these \$1 games until either all their money is gone, or they have reached a total of \$20, at which time they retire happy. How many turns does it take on average before they leave the gaming table? And what fraction of the time do they leave happy?

It is relatively easy to determine that the casino running this game has a slight edge, and that in the long run, more players should lose than win. Indeed, a mathematician might be able to precisely calculate the average number of games played by each player before one of the two stopping conditions is met. Another way to answer such questions is to make use of a *simulation*. The top box in Figure 9.5 shows a program that tracks the outcome of one player, and their initial \$5.

A key component of Figure 9.5 is use of the functions `srand` and `rand`, described in `stdlib.h`. The first function initializes a pseudo-random number generator by passing in an integer seed; thereafter, each call to `rand` returns a positive integer that for most purposes can be assumed to be quite unrelated to the previous values returned. The actual mechanism involved is beyond the scope of this book. What is worth stressing is that the sequence is not really random at all, and is completely and unambiguously determined by the initial seed. Hence, if you are using such a