CHAPTER 1

PROBLEMS AND ALGORITHMS

An algorithm is a detailed step-by-step method for solving a problem. But what is a problem? We introduce in this chapter three important examples.

1.1 GRAPH REACHABILITY

A graph $G = (V, E)$ is a finite set V of nodes and a set E of edges, which are pairs of nodes (see Figure 1.1 for an example; all our graphs will be finite and directed). Many computational problems are concerned with graphs. The most basic problem on graphs is this: Given a graph G and two nodes $1, n \in V$, is there a path from 1 to n? We call this problem REACHABILITY†. For example, in Figure 1 there is indeed a path from node 1 to $n = 5$, namely $(1, 4, 3, 5)$. If instead we reverse the direction of edge $(4, 3)$, then no such path exists.

Like most interesting problems, REACHABILITY has an infinite set of possible *instances*. Each instance is a mathematical object (in our case, a graph and two of its nodes), of which we ask a question and expect an answer. The specific kind of question asked characterizes the problem. Notice that REACHABILITY asks a question that requires either a "yes" or a "no" answer. Such problems are called *decision problems*. In complexity theory we usually find it conveniently unifying and simplifying to consider only decision problems, instead of problems requiring all sorts of different answers. So, decision problems will play an important role in this book.

† In complexity theory, computational problems are not just things to solve, but also mathematical objects that are interesting in their own right. When problems are treated as mathematical objects, their names are shown in capital letters.

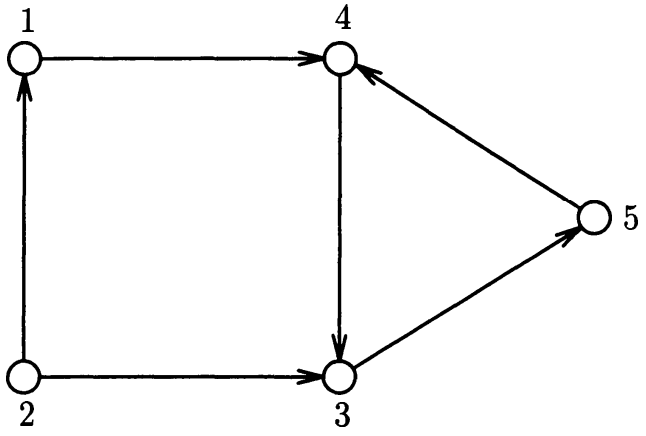

Figure 1-1. A graph.

We are interested in algorithms that solve our problems. In the next chapter we shall introduce the *Turing machine*, a formal model for expressing arbitrary algorithms. For the time being, let us describe our algorithms informally. For example, REACHABILITY can be solved by the so-called *search algorithm*. This algorithm works as follows: Throughout the algorithm we maintain a set of nodes, denoted S. Initially, $S = \{1\}$. Each node can be either *marked* or *unmarked*. That node i is marked means that i has been in S at some point in the past (or, it is presently in S). Initially, only 1 is marked. At each iteration of the algorithm, we choose a node $i \in S$ and remove it from S. We then process one by one all edges (i, j) out of i. If node j is unmarked, then we mark it, and add it to S. This process continues until S becomes empty. At this point, we answer "yes" if node n is marked, and "no" if it is not marked.

It should be clear that this familiar algorithm solves REACHABILITY. The proof would establish that a node is marked if and only if there is a path from 1 to it; both directions are easy inductions (see Problem 1.4.2). It is also clear, however, that there are important details that have been left out of our description of the algorithm. For example, how is the graph represented as an input to the algorithm? Since appropriate representations depend on the specific model of algorithms we use, this will have to wait until we have a specific model. The main point of that discussion (see Section 2.2) is that *the precise representation does not matter much*. You can assume in the meantime that the graph is given through its adjacency matrix (Figure 1.2), all entries of which can be accessed by the algorithm in a random access fashion‡.

There are also unclear spots in the algorithm itself: How is the element

‡ As a matter of fact, in Chapter 7 we shall see important applications of REACHABILITY to complexity theory in which the graph is given *implicitly*; that is, each entry of its adjacency matrix can be computed from the input data.

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Figure 1.2. Adjacency matrix.

$i \in S$ chosen among all elements of S? The choice here may affect significantly the style of the search. For example, if we always choose the node that has stayed in S the longest (in other words, if we implement S as a queue) then the resulting search is *breadth-first*, and a shortest path is found. If S is maintained as a stack (we choose the node added last), we have a kind of *depth-first* search. Other ways of maintaining S would result in completely different kinds of search. But the algorithm works correctly for all such choices.

Moreover, it works efficiently. To see that it does, notice that each entry of the adjacency matrix is visited only once, when the vertex corresponding to its row is chosen. Hence, we only spend about n^2 operations processing edges out of the chosen nodes (after all, there can be at most n^2 edges in a graph). Assuming that the other simple operations required (choosing an element of the set S, marking a vertex, and telling whether a vertex is marked) can each somehow be done in constant time, we conclude that the search algorithm determines whether two nodes in a graph with n nodes are connected in time at most proportional to n^2, or $\mathcal{O}(n^2)$.

The $\mathcal{O}$ notation we just used, and its relatives, are very useful in complexity theory, so we open here a brief parenthesis to define them formally.

Definition 1.1: We denote by $\mathbf{N}$ the set of all nonnegative integers. In complexity theory we concern ourselves with functions from $\mathbf{N}$ to $\mathbf{N}$, such as n^2, 2^n, and $n^3 - 2n + 5$. We shall use the letter n as the standard argument of such functions. Even when we denote our functions in a way that admits non-integer or negative values—such as $\sqrt{n}$, $\log n$, and $\sqrt{n} - 4\log^2 n$—we shall always think of the values as nonnegative integers. That is, any function denoted as $f(n)$, as in these examples, for us really means $\max\{\lceil f(n) \rceil, 0\}$.

So, let f and g be functions from $\mathbf{N}$ to $\mathbf{N}$. We write $f(n) = \mathcal{O}(g(n))$ (pronounced "$f(n)$ is big oh of $g(n)$," or "f is of the order of g") if there are positive integers c and n_0 such that, for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$. $f(n) = \mathcal{O}(g(n))$ means informally that f grows as g or slower. We write $f(n) = \Omega(g(n))$ if the opposite happens, that is, if $g(n) = \mathcal{O}(f(n))$. Finally, $f(n) = \Theta(g(n))$ means that $f = \mathcal{O}(g(n))$ and $f(n) = \Omega(g(n))$. The latter means that f and g have precisely the same *rate of growth*.

For example, it is easy to see that, if $p(n)$ is a polynomial of degree d, then