Chapter Algorithms

We have seen that before a computer can perform a task, it must be given an algorithm telling it precisely what to do; consequently, the study of algorithms is the cornerstone of computer science. In this chapter we introduce many of the fundamental concepts of this study, including the issues of algorithm discovery and representation as well as the major control concepts of iteration and recursion. In so doing we also present a few well-known algorithms for searching and sorting.
long as the value of neither $X$ nor $Y$ is zero, continue dividing the
smallest by the smaller and assigning $X$ and $Y$ the values of
divisor and remainder, respectively. (The final value of $X$ is the great-
common divisor.)

this algorithm in our pseudocode.
ribe a collection of primitives that are used in a subject other than
outer programming.

4.3 Algorithm Discovery

The development of a program consists of two activities—discovering the
underlying algorithm and representing that algorithm as a program. Up to this
point we have been concerned with the issues of algorithm representation without
considering the question of how algorithms are found in the first place. Yet
algorithm discovery is usually the more challenging step in the software develop-
ment process. After all, to discover an algorithm is to find a method of solv-
ing that problem whose solution the algorithm is to compute. Thus, to
understand how algorithms are discovered is to understand the problem-solving
process.

The Theory of Problem Solving

The techniques of problem solving and the need to learn more about them are
not unique to computer science, but rather they are topics pertinent to almost
any field. The close association between the process of algorithm discovery and
that of general problem solving has caused computer scientists to join with
those of other disciplines in the search for better problem-solving techniques.
Ultimately, one would like to reduce the process of problem solving to an algo-
ithm in itself, but this has been shown to be impossible. (This is a result of the
material in Chapter 11, where we show that there are problems that do not
have algorithmic solutions.) Thus the ability to solve problems remains more of
an artistic skill to be developed than a precise science to be learned.

As evidence of the illusive, artistic nature of problem solving, the following
loosely defined problem-solving phases presented by the mathematician G.
Polya in 1945 remain the basic principles on which attempts to teach problem-
solving skills are based today.

Phase 1. Understand the problem.
Phase 2. Devise a plan for solving the problem.
Phase 3. Carry out the plan.
Phase 4. Evaluate the solution for accuracy and for its potential as a tool
for solving other problems.
Translated into the context of program development, these phases become

**Phase 1.** Understand the problem.
**Phase 2.** Get an idea as to how an algorithmic procedure might solve the problem.
**Phase 3.** Formulate the algorithm and represent it as a program.
**Phase 4.** Evaluate the program for accuracy and for its potential as a tool for solving other problems.

Having presented Polya’s list, we should emphasize that these phases are not steps to be followed when trying to solve a problem but rather phases that will be completed sometime during the solution process. The key word here is *followed*. You do not solve problems by following. Rather, to solve a problem, you must take the initiative and lead. If you approach the task of solving a problem in the frame of mind depicted by “Now I’ve finished Phase 1, it’s time to move on to Phase 2,” you are not likely to be successful. However, if you become involved with the problem and ultimately solve it, you most likely can look back at what you did and realize that Polya’s four phases had been completed.

Another important observation is that Polya’s four phases are not necessarily completed in sequence. Contrary to the claim made by many authors, successful problem solvers often start formulating strategies for solving a problem (Phase 2) before the problem itself is entirely understood (Phase 1). Then, if these strategies fail (during Phases 3 or 4), the potential problem solver gains a deeper understanding of the intricacies of the problem and, based on this deeper understanding, can return to form other and hopefully more successful strategies.

Keep in mind that we are discussing how problems are solved—not how we would like them to be solved. Ideally, we would like to eliminate the waste inherent in the trial-and-error process just described. In the case of developing large software systems, discovering a misunderstanding as late as Phase 4 can represent a tremendous loss in resources. Avoiding such catastrophes is a major goal of software engineers (Chapter 6), who have traditionally insisted on a thorough understanding of a problem before proceeding with a solution. One could argue, however, that a true understanding of a problem is not obtained until a solution has been found, that the mere fact that a problem is unsolved implies a lack of understanding. To insist on a complete understanding of the problem before proposing any solutions is therefore somewhat idealistic.

As an example, consider the following problem:

Person A is charged with the task of determining the ages of person B’s three children. B tells A that the product of the children’s ages is 36.
After considering this clue, A replies that another clue is required, so B tells A the sum of the children’s ages. Again, A replies that another clue is needed, so B tells A that the oldest child plays the piano. After hearing this clue, A tells B the ages of the three children.
How old are the three children?
At first glance the last clue seems to be totally unrelated to the problem, yet it is this clue that allows A to finally determine the ages of the children. How can this be? Let us proceed by formulating a plan of attack and following this plan, even though we still have many questions about the problem. Our plan will be to trace the steps described by the problem statement while keeping track of the information available to person A as the story progresses.

The first clue given A is that the product of the children's ages is 36. This means that the triple representing the three ages is one of those listed in Figure 4.5(a). The next clue is the sum of the desired triple. We are not told what this sum is, but we are told that this information is not enough for A to isolate the correct triple; therefore the desired triple must be one whose sum appears at least twice in the table of Figure 4.5(b). But the only triples appearing in (b) with identical sums are (1,6,6) and (2,2,9), both of which produce the sum 13. This is the information available to A at the time the last clue is given. It is at this point that we finally understand the significance of the last clue. It has nothing to do with playing the piano; rather it is the fact that there is an oldest child. This rules out the triple (1,6,6) and thus allows us to conclude that the children's ages are 2, 2, and 9.

In this case, then, it is not until we attempt to implement our plan for solving the problem (Phase 3) that we gain a complete understanding of the problem (Phase 1). Had we insisted on completing phase 1 before proceeding, we would probably never have found the children's ages. Such irregularities in the problem-solving process are fundamental to the difficulties in developing systematic approaches to problem solving.

Another irregularity is the mysterious inspiration that may come to a potential problem solver who, having worked on a problem without apparent success, may at a later time suddenly see the solution while doing another task. This phenomenon was identified by H. von Helmholtz as early as 1896 and was discussed by the mathematician Henri Poincaré in a lecture before the Psychological Society in Paris. There, Poincaré described his experiences of real-
izing the solution to a problem he had worked on after he had set it aside and begun other projects. The phenomenon reflects a process in which a subconscious part of the mind appears to continue working and, if successful, forces the solution into the conscious mind. Today, the period between conscious work on a problem and the sudden inspiration is known as an incubation period, and its understanding remains a goal of current research.

**Getting a Foot in the Door**

We have been discussing problem solving from a somewhat philosophical point of view while avoiding a direct confrontation with the question of how we should go about trying to solve a problem. There are, of course, numerous problem-solving approaches, each of which can be successful in certain settings. We will identify some of them shortly. For now, we note that there seems to be a common thread running through these techniques, which simply stated is "get your foot in the door." As an example, let us consider the following simple problem:

Before A, B, C, and D ran a race they made the following predictions:

- A predicted that B would win.
- B predicted that D would be last.
- C predicted that A would be third.
- D predicted that A’s prediction would be correct.

Only one of these predictions was true, and this was the prediction made by the winner. In what order did A, B, C, and D finish the race?

After reading the problem and analyzing the data, it should not take long to realize that since the predictions of A and D were equivalent and only one prediction was true, the predictions of both A and D must be false. Thus neither A nor D were winners. At this point we have our foot in the door, and obtaining the complete solution to our problem is merely a matter of extending our knowledge from here. If A’s prediction was false, then B did not win either. The only remaining choice for the winner is C. Thus, C won the race, and C’s prediction was true. Consequently, we know that A came in third. That means that the finishing order was either CBAD or CDAB. But the former is ruled out because B’s prediction must be false. Therefore the finishing order was CDAB.

Of course, being told to get our foot in the door is not the same as being told how to do it. Obtaining this toehold, as well as realizing how to expand this initial thrust into a complete solution to the problem, requires creative input from the would-be problem solver. There are, however, several general approaches that have been proposed by Polya and others for how one might go about getting a foot in the door. One is to try working the problem backward. For instance, if the problem is to find a way of producing a particular output from a given input, one might start with that output and attempt to back up to
the given input. This approach is typical of someone trying to discover the bird-folding algorithm in the previous section. They tend to unfold a completed bird in an attempt to see how it is constructed.

Another general problem-solving approach is to look for a related problem that is either easier to solve or has been solved before and then try to apply its solution to the current problem. This technique is of particular value in the context of program development. Often the major difficulty in program development is not that of solving a particular instance of a problem but rather of finding a general algorithm that can be used to solve all instances of the problem. More precisely, if we were faced with the task of developing a program for alphabetizing lists of names, our task would not be to sort a particular list but to find a general algorithm that could be used to sort any list of names. Thus, although the instructions

Interchange the names David and Alice.
Move the name Carol to the position between Alice and David.
Move the name Bob to the position between Alice and Carol.

correctly sort the list David, Alice, Carol, and Bob, they do not constitute the general-purpose algorithm we desire. What we need is an algorithm that can sort this list as well as other lists we may encounter. This is not to say that our solution for sorting a particular list is totally worthless in our search for a general-purpose algorithm. We might, for instance, get our foot in the door by considering such special cases in an attempt to find general principles that can in turn be used to develop the desired general-purpose algorithm. In this case, then, our solution is obtained by the technique of solving a collection of related problems.

Still another approach to getting a foot in the door is to apply stepwise refinement, which is essentially the technique of not trying to conquer an entire task (in all its gory detail) at once. Rather, stepwise refinement proposes that one first view the problem at hand in terms of several subproblems. The idea is that by breaking the original problem into subproblems, one is able to approach the overall solution in terms of steps, each of which is easier to solve than the entire original problem. In turn, stepwise refinement proposes that these steps be decomposed into smaller steps and these smaller steps be broken into still smaller ones until the entire problem has been reduced to a collection of easily solved subproblems.

In this light, stepwise refinement is a top-down methodology in that it progresses from the general to the specific. In contrast, bottom-up methodologies progress from the specific to the general. Although contrasting in theory, the two approaches actually complement each other in practice. For instance, the decomposition of a problem proposed by the top-down methodology of stepwise refinement is often guided by the problem solver’s intuition, which is working in a bottom-up mode.

Solutions produced by stepwise refinement possess a natural modular structure, and herein lies a major reason for the popularity of stepwise refinement in
algorithm design. If an algorithm has a natural modular structure, then it is easily adapted to a modular representation, which is conducive to the development of a manageable program. Furthermore, the modules produced by stepwise refinement are compatible with the concept of team programming, in which several people are assigned the task of developing a software product as a team. After all, once the task of the software has been broken into subproblems (or potential modules), the personnel on the team can work independently on these subtasks without getting in each other’s way.

These advantages of stepwise refinement in the context of software development have produced many followers of the technique. However, with all its good points, stepwise refinement is not the final word in algorithm discovery. Rather, it is essentially an organizational tool whose problem-solving attributes are consequences of this organization. Stepwise refinement is a natural methodology to use when organizing a nationwide political campaign, writing a term paper, or planning a sales convention. Similarly, most software development projects in the data processing community have a large organizational component. The task is not so much that of discovering a startling new algorithm as it is a problem of organizing the tasks to be performed into a coherent package. For these reasons, stepwise refinement has correctly become a major design methodology in data processing.

But stepwise refinement remains only one of many design methodologies of interest to computer scientists, and thus one should not be misled into believing that all algorithm discoveries can be achieved by means of stepwise refinement. In fact, bringing preconceived notions and preselected tools to the problem-solving task can sometimes mask a problem’s simplicity. Consider the following problem:

As you step from a pier into a boat, your hat falls into the water, unknownst to you. The river is flowing at 2.5 miles per hour so your hat begins to float downstream. In the meantime, you begin traveling upstream in the boat at a speed of 4.75 miles per hour relative to the water. After 10 minutes you realize that your hat is missing, turn the boat around, and begin to chase your hat downstream. How long will it take to catch up with your hat?

Most students of high school algebra as well as pocket calculator enthusiasts approach this problem by first determining how far upstream the boat will have traveled in 10 minutes as well as how far downstream the hat will have traveled during that same time. Then, they try to determine how long it will take for the boat to travel downstream to this position. But, when the boat reaches this position, the hat will have floated farther downstream! Thus, the would-be problem solver becomes trapped in a cycle of computing where the hat will be each time the boat goes to where the hat was.

The problem is much simpler than this, however. The trick is to resist the urge to begin writing formulas and making calculations. Instead, we need to put these skills aside and adjust our perspective. The entire problem takes place
in the river. The fact that the water is moving in relation to the shore is irrelevant. Think of the same problem posed on a large conveyor belt instead of a river. First, solve the problem with the conveyor belt stopped. If you place your hat at your feet while standing on the belt and then walk away from your hat for 10 minutes, it will take 10 minutes to return to your hat. Now turn on the conveyor belt. This means that the scenery will begin to move past the belt, but, because you are on the belt, this does not change your relationship to the belt or your hat. It will still take 10 minutes to return to your hat.

We conclude that algorithm discovery remains a challenging art that must be developed over a period of time rather than taught as a subject consisting of well-defined methodologies. Indeed, to train a potential problem solver to follow certain methodologies is to quash those creative skills that should instead be nurtured.

ESTIONS/EXERCISES

a. Find an algorithm for solving the following program: Given a positive integer $n$, find the list of positive integers whose product is the largest among all the lists of positive integers whose sum is $n$. For example, if $n$ is 4, the desired list is 2, 2 because $2 + 2$ is larger than $1 + 1 + 1 + 1, 2 + 1 + 1$, and $3 + 1$. If $n$ is 5, the desired list is 2, 3.

b. What is the desired list if $n = 2001$?

c. Explain how you got your foot in the door.

a. Suppose we are given a checkerboard consisting of $2^n$ rows and $2^n$ columns of squares, for some positive integer $n$, and a box of L-shaped tiles, each of which can cover exactly three squares on the board. If any single square is cut out of the board, can we cover the remaining board with tiles such that tiles do not overlap or hang off the edge of the board?

b. Explain how your solution to (a) can be used to show that $2^{2n} - 1$ is divisible by 3 for all positive integers $n$.

c. How are (a) and (b) related to Polya's phases of problem solving? Decode the following message, then explain how you got your foot in the door.

Pdeo eo pda yknnayp wjosan.

4.4 Iterative Structures

Our goal now is to study some of the repetitive structures used in describing algorithmic processes. In this section we discuss iterative structures in which a collection of instructions is repeated in a looping manner. In the next section