17

Limits to Computation

This book describes many data structures that can be used in a wide variety of problems. There are also many examples of efficient algorithms. In general, our search algorithms strive to be at worst in $\mathrm{O}(\log n)$ to find a record, while our sorting algorithms strive to be in $\mathrm{O}(n \log n)$. A few algorithms have higher asymptotic complexity, such as Floyd's all-pairs shortest-paths algorithm, whose running time is $\Theta(n^3)$.

We can solve many problems efficiently because we have available (and choose to use) efficient algorithms. Given any problem for which you know *some* algorithm, it is always possible to write an inefficient algorithm to "solve" the problem. For example, consider a sorting algorithm that tests every possible permutation of its input until it finds the correct permutation that provides a sorted list. The running time for this algorithm would be unacceptably high, because it is proportional to the number of permutations which is $n!$ for n inputs. When solving the minimum-cost spanning tree problem, if we were to test every possible subset of edges to see which forms the shortest minimum spanning tree, the amount of work would be proportional to $2^{|\mathrm{E}|}$ for a graph with $|\mathrm{E}|$ edges. Fortunately, for both of these problems we have more clever algorithms that allow us to find answers (relatively) quickly without explicitly testing every possible solution.

Unfortunately, there are many computing problems for which the best possible algorithm takes a long time to run. A simple example is the Towers of Hanoi problem, which requires 2^n moves to "solve" a tower with n disks. It is not possible for any computer program that solves the Towers of Hanoi problem to run in less than $\Omega(2^n)$ time, because that many moves must be printed out.

Besides those problems whose solutions *must* take a long time to run, there are also many problems for which we simply do not know if there are efficient algorithms or not. The best algorithms that we know for such problems are very slow, but perhaps there are better ones waiting to be discovered. Of course, while having a problem with high running time is bad, it is even worse to have a problem that cannot be solved at all! Problems of the later type do exist, and some are presented in Section 17.3.

This chapter presents a brief introduction to the theory of expensive and impossible problems. Section 17.1 presents the concept of a reduction, which is the central tool used for analyzing the difficulty of a problem (as opposed to analyzing the cost of an algorithm). Reductions allow us to relate the difficulty of various problems, which is often much easier than doing the analysis for a problem from first principles. Section 17.2 discusses "hard" problems, by which we mean problems that require, or at least appear to require, time exponential on the input size. Finally, Section 17.3 considers various problems that, while often simple to define and comprehend, are in fact impossible to solve using a computer program. The classic example of such a problem is deciding whether an arbitrary computer program will go into an infinite loop when processing a specified input. This is known as the **halting problem**.

17.1 Reductions

We begin with an important concept for understanding the relationships between problems, called **reduction**. Reduction allows us to solve one problem in terms of another. Equally importantly, when we wish to understand the difficulty of a problem, reduction allows us to make relative statements about upper and lower bounds on the cost of a problem (as opposed to an algorithm or program).

Because the concept of a problem is discussed extensively in this chapter, we want notation to simplify problem descriptions. Throughout this chapter, a problem will be defined in terms of a mapping between inputs and outputs, and the name of the problem will be given in all capital letters. Thus, a complete definition of the sorting problem could appear as follows:

SORTING:
Input: A sequence of integers x_0, x_1, x_2, ..., x_{n-1}.
Output: A permutation y_0, y_1, y_2, ..., y_{n-1} of the sequence such that $y_i \leq y_j$ whenever $i < j$.

Once you have bought or written a program to solve one problem, such as sorting, you might be able to use it as a tool to solve a different problem. This is

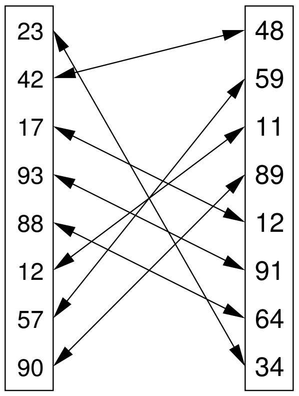


Figure 17.1 An illustration of PAIRING. The two lists of numbers are paired up so that the least values from each list make a pair, the next smallest values from each list make a pair, and so on.

known in software engineering as **software reuse**. To illustrate this, let us consider another problem.

PAIRING:

Input: Two sequences of integers $X = (x_0, x_1, ..., x_{n-1})$ and $Y = (y_0, y_1, ..., y_{n-1})$.

Output: A pairing of the elements in the two sequences such that the least value in X is paired with the least value in Y, the next least value in X is paired with the next least value in Y, and so on.

Figure 17.1 illustrates PAIRING. One way to solve PAIRING is to use an existing sorting program by sorting each of the two sequences, and then pairing-off items based on their position in sorted order. Technically, we say that PAIRING is **reduced** to SORTING, because SORTING is used to solve PAIRING.

Notice that reduction is a three-step process. The first step is to convert an instance of PAIRING into two instances of SORTING. The conversion step is not very interesting; it simply takes each sequence and assigns it to an array to be passed to SORTING. The second step is to sort the two arrays (i.e., apply SORTING to each array). The third step is to convert the output of SORTING to the output for PAIRING. This is done by pairing the first elements in the sorted arrays, the second elements, and so on.

The reduction of PAIRING to SORTING helps to establish an upper bound on the cost of PAIRING. In terms of asymptotic notation, assuming that we can find one method to convert the inputs to PAIRING into inputs to SORTING "fast enough," and a second method to convert the result of SORTING back to the correct result for PAIRING "fast enough," then the asymptotic cost of PAIRING cannot be more than the cost of SORTING. In this case, there is little work to be done to convert from PAIRING to SORTING, or to convert the answer from SORTING back to the answer for PAIRING, so the dominant cost of this solution is performing the sort operation. Thus, an upper bound for PAIRING is in $\mathrm{O}(n \log n)$.

It is important to note that the pairing problem does *not* require that elements of the two sequences be sorted. This is merely one possible way to solve the problem. PAIRING only requires that the elements of the sequences be paired correctly. Perhaps there is another way to do it? Certainly if we use sorting to solve PAIRING, the algorithms will require $\Omega(n \log n)$ time. But, another approach might conceivably be faster.

There is another use of reductions aside from applying an old algorithm to solve a new problem (and thereby establishing an upper bound for the new problem). That is to prove a lower bound on the cost of a new problem by showing that it could be used as a solution for an old problem with a known lower bound.

Assume we can go the other way and convert SORTING to PAIRING "fast enough." What does this say about the minimum cost of PAIRING? We know from Section 7.9 that the cost of SORTING in the worst and average cases is in $\Omega(n \log n)$. In other words, the best possible algorithm for sorting requires at least $n \log n$ time.

Assume that PAIRING could be done in $\mathrm{O}(n)$ time. Then, one way to create a sorting algorithm would be to convert SORTING into PAIRING, run the algorithm for PAIRING, and finally convert the answer back to the answer for SORTING. Provided that we can convert SORTING to/from PAIRING "fast enough," this process would yield an $\mathrm{O}(n)$ algorithm for sorting! Because this contradicts what we know about the lower bound for SORTING, and the only flaw in the reasoning is the initial assumption that PAIRING can be done in $\mathrm{O}(n)$ time, we can conclude that there is no $\mathrm{O}(n)$ time algorithm for PAIRING. This reduction process tells us that PAIRING must be at least as expensive as SORTING and so must itself have a lower bound in $\Omega(n \log n)$.

To complete this proof regarding the lower bound for PAIRING, we need now to find a way to reduce SORTING to PAIRING. This is easily done. Take an instance of SORTING (i.e., an array *A* of n elements). A second array *B* is generated that simply stores i in position i for $0 \leq i < n$. Pass the two arrays to PAIRING. Take the resulting set of pairs, and use the value from the *B* half of the pair to tell which position in the sorted array the *A* half should take; that is, we can now reorder the records in the *A* array using the corresponding value in the *B* array as the sort key and running a simple $\Theta(n)$ Binsort. The conversion of SORTING to PAIRING can be done in $O(n)$ time, and likewise the conversion of the output of PAIRING can be converted to the correct output for SORTING in $O(n)$ time. Thus, the cost of this "sorting algorithm" is dominated by the cost for PAIRING.

Consider any two problems for which a suitable reduction from one to the other can be found. The first problem takes an arbitrary instance of its input, which we will call *I*, and transforms *I* to a solution, which we will call *SLN*. The second problem takes an arbitrary instance of its input, which we will call *I′*, and transforms *I′* to a solution, which we will call *SLN′*. We can define reduction more formally as a three-step process:

1. Transform an arbitrary instance of the first problem to an instance of the second problem. In other words, there must be a transformation from any instance *I* of the first problem to an instance *I′* of the second problem.
2. Apply an algorithm for the second problem to the instance *I′*, yielding a solution *SLN′*.
3. Transform *SLN′* to the solution of *I*, known as *SLN*. Note that *SLN* must in fact be the correct solution for *I* for the reduction to be acceptable.

It is important to note that the reduction process does not give us an algorithm for solving either problem by itself. It merely gives us a method for solving the first problem given that we already have a solution to the second. More importantly for the topics to be discussed in the remainder of this chapter, reduction gives us a way to understand the bounds of one problem in terms of another. Specifically, given efficient transformations, the upper bound of the first problem is at most the upper bound of the second. Conversely, the lower bound of the second problem is at least the lower bound of the first.

As a second example of reduction, consider the simple problem of multiplying two n-digit numbers. The standard long-hand method for multiplication is to multiply the last digit of the first number by the second number (taking $\Theta(n)$ time), multiply the second digit of the first number by the second number (again taking $\Theta(n)$ time), and so on for each of the n digits of the first number. Finally, the intermediate results are added together. Note that adding two numbers of length M and N can easily be done in $\Theta(M+N)$ time. Because each digit of the first number is multiplied against each digit of the second, this algorithm requires $\Theta(n^2)$ time. Asymptotically faster (but more complicated) algorithms are known, but none is so fast as to be in $O(n)$.

Next we ask the question: Is squaring an n-digit number as difficult as multiplying two n-digit numbers? We might hope that something about this special case will allow for a faster algorithm than is required by the more general multiplication problem. However, a simple reduction proof serves to show that squaring is "as hard" as multiplying.

The key to the reduction is the following formula:

$$X \times Y = \frac{(X+Y)^2 - (X-Y)^2}{4}.$$

The significance of this formula is that it allows us to convert an arbitrary instance of multiplication to a series of operations involving three addition/subtractions (each of which can be done in linear time), two squarings, and a division by 4. Note that the division by 4 can be done in linear time (simply convert to binary, shift right by two digits, and convert back).

This reduction shows that if a linear time algorithm for squaring can be found, it can be used to construct a linear time algorithm for multiplication.

Our next example of reduction concerns the multiplication of two $n \times n$ matrices. For this problem, we will assume that the values stored in the matrices are simple integers and that multiplying two simple integers takes constant time (because multiplication of two **`int`** variables takes a fixed number of machine instructions). The standard algorithm for multiplying two matrices is to multiply each element of the first matrix's first row by the corresponding element of the second matrix's first column, then adding the numbers. This takes $\Theta(n)$ time. Each of the n^2 elements of the solution are computed in similar fashion, requiring a total of $\Theta(n^3)$ time. Faster algorithms are known (see the discussion of Strassen's Algorithm in Section 16.4.3), but none are so fast as to be in $O(n^2)$.

Now, consider the case of multiplying two **symmetric** matrices. A symmetric matrix is one in which entry ij is equal to entry ji; that is, the upper-right triangle of the matrix is a mirror image of the lower-left triangle. Is there something about this restricted case that allows us to multiply two symmetric matrices faster than in the general case? The answer is no, as can be seen by the following reduction. Assume that we have been given two $n \times n$ matrices *A* and *B*. We can construct a $2n \times 2n$ symmetric matrix from an arbitrary matrix *A* as follows:

$$\left[\begin{array}{cc} 0 & A \\ A^{\mathrm{T}} & 0 \end{array}\right].$$

Here 0 stands for an $n \times n$ matrix composed of zero values, *A* is the original matrix, and A^{T} stands for the transpose of matrix *A*.[1] Note that the resulting matrix is now

[1]The transpose operation takes position ij of the original matrix and places it in position ji of the transpose matrix. This can easily be done in n^2 time for an $n \times n$ matrix.

symmetric. We can convert matrix B to a symmetric matrix in a similar manner. If symmetric matrices could be multiplied "quickly" (in particular, if they could be multiplied together in $\Theta(n^2)$ time), then we could find the result of multiplying two arbitrary $n \times n$ matrices in $\Theta(n^2)$ time by taking advantage of the following observation:

$$\begin{bmatrix} 0 & A \\ A^{\mathrm{T}} & 0 \end{bmatrix} \begin{bmatrix} 0 & B^{\mathrm{T}} \\ B & 0 \end{bmatrix} = \begin{bmatrix} AB & 0 \\ 0 & A^{\mathrm{T}}B^{\mathrm{T}} \end{bmatrix}.$$

In the above formula, AB is the result of multiplying matrices A and B together.

17.2 Hard Problems

There are several ways that a problem could be considered hard. For example, we might have trouble understanding the definition of the problem itself. At the beginning of a large data collection and analysis project, developers and their clients might have only a hazy notion of what their goals actually are, and need to work that out over time. For other types of problems, we might have trouble finding or understanding an algorithm to solve the problem. Understanding spoken Engish and translating it to written text is an example of a problem whose goals are easy to define, but whose solution is not easy to discover. But even though a natural language processing algorithm might be difficult to write, the program's running time might be fairly fast. There are many practical systems today that solve aspects of this problem in reasonable time.

None of these is what is commonly meant when a computer theoretician uses the word "hard." Throughout this section, "hard" means that the best-known algorithm for the problem is expensive in its running time. One example of a hard problem is Towers of Hanoi. It is easy to understand this problem and its solution. It is also easy to write a program to solve this problem. But, it takes an extremely long time to run for any "reasonably" large value of n. Try running a program to solve Towers of Hanoi for only 30 disks!

The Towers of Hanoi problem takes exponential time, that is, its running time is $\Theta(2^n)$. This is radically different from an algorithm that takes $\Theta(n \log n)$ time or $\Theta(n^2)$ time. It is even radically different from a problem that takes $\Theta(n^4)$ time. These are all examples of polynomial running time, because the exponents for all terms of these equations are constants. Recall from Chapter 3 that if we buy a new computer that runs twice as fast, the size of problem with complexity $\Theta(n^4)$ that we can solve in a certain amount of time is increased by the fourth root of two. In other words, there is a multiplicative factor increase, even if it is a rather small one. This is true for any algorithm whose running time can be represented by a polynomial.

Consider what happens if you buy a computer that is twice as fast and try to solve a bigger Towers of Hanoi problem in a given amount of time. Because its complexity is $\Theta(2^n)$, we can solve a problem only one disk bigger! There is no multiplicative factor, and this is true for any exponential algorithm: A constant factor increase in processing power results in only a fixed addition in problem-solving power.

There are a number of other fundamental differences between polynomial running times and exponential running times that argues for treating them as qualitatively different. Polynomials are closed under composition and addition. Thus, running polynomial-time programs in sequence, or having one program with polynomial running time call another a polynomial number of times yields polynomial time. Also, all computers known are polynomially related. That is, any program that runs in polynomial time on any computer today, when tranferred to any other computer, will still run in polynomial time.

There is a practical reason for recognizing a distinction. In practice, most polynomial time algorithms are "feasible" in that they can run reasonably large inputs in reasonable time. In contrast, most algorithms requiring exponential time are not practical to run even for fairly modest sizes of input. One could argue that a program with high polynomial degree (such as n^{100}) is not practical, while an exponential-time program with cost 1.001^n is practical. But the reality is that we know of almost no problems where the best polynomial-time algorithm has high degree (they nearly all have degree four or less), while almost no exponential-time algorithms (whose cost is $(O(c^n))$ have their constant c close to one. So there is not much gray area between polynomial and exponential time algorithms in practice.

For the rest of this chapter, we define a **hard algorithm** to be one that runs in exponential time, that is, in $\Omega(c^n)$ for some constant $c > 1$. A definition for a hard *problem* will be presented in the next section.

17.2.1 The Theory of $\mathcal{NP}$-Completeness

Imagine a magical computer that works by guessing the correct solution from among all of the possible solutions to a problem. Another way to look at this is to imagine a super parallel computer that could test all possible solutions simultaneously. Certainly this magical (or highly parallel) computer can do anything a normal computer can do. It might also solve some problems more quickly than a normal computer can. Consider some problem where, given a guess for a solution, checking the solution to see if it is correct can be done in polynomial time. Even if the number of possible solutions is exponential, any given guess can be checked in polynomial time (equivalently, all possible solutions are checked simultaneously in polynomial time), and thus the problem can be solved in polynomial time by our hypothetical magical computer. Another view of this concept is that if you cannot get the answer to a problem in polynomial time by guessing the right answer and then checking it, then you cannot do it in polynomial time in any other way.

The idea of "guessing" the right answer to a problem — or checking all possible solutions in parallel to determine which is correct — is called **non-determinism**. An algorithm that works in this manner is called a **non-deterministic algorithm**, and any problem with an algorithm that runs on a non-deterministic machine in polynomial time is given a special name: It is said to be a problem in $\mathcal{NP}$. Thus, problems in $\mathcal{NP}$ are those problems that can be solved in polynomial time on a non-deterministic machine.

Not all problems requiring exponential time on a regular computer are in $\mathcal{NP}$. For example, Towers of Hanoi is *not* in $\mathcal{NP}$, because it must print out O(2^n) moves for n disks. A non-deterministic machine cannot "guess" and print the correct answer in less time.

On the other hand, consider what is commonly known as the Traveling Salesman problem.

TRAVELING SALESMAN (1)
Input: A complete, directed graph **G** with positive distances assigned to each edge in the graph.
Output: The shortest simple cycle that includes every vertex.

Figure 17.2 illustrates this problem. Five vertices are shown, with edges and associated costs between each pair of edges. (For simplicity, we assume that the cost is the same in both directions, though this need not be the case.) If the salesman visits the cities in the order ABCDEA, he will travel a total distance of 13. A better route would be ABDCEA, with cost 11. The best route for this particular graph would be ABEDCA, with cost 9.

We cannot solve this problem in polynomial time with a guess-and-test non-deterministic computer. The problem is that, given a candidate cycle, while we can quickly check that the answer is indeed a cycle of the appropriate form, and while we can quickly calculate the length of the cycle, we have no easy way of knowing if it is in fact the *shortest* such cycle. However, we can solve a variant of this problem cast in the form of a **decision problem**. A decision problem is simply one whose answer is either YES or NO. The decision problem form of TRAVELING SALESMAN is as follows:

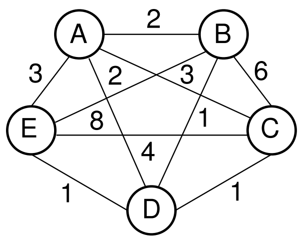


Figure 17.2 An illustration of the TRAVELING SALESMAN problem. Five vertices are shown, with edges between each pair of cities. The problem is to visit all of the cities exactly once, returning to the start city, with the least total cost.

TRAVELING SALESMAN (2)

Input: A complete, directed graph **G** with positive distances assigned to each edge in the graph, and an integer k.

Output: YES if there is a simple cycle with total distance $\leq k$ containing every vertex in **G**, and NO otherwise.

We can solve this version of the problem in polynomial time with a non-deterministic computer. The non-deterministic algorithm simply checks all of the possible subsets of edges in the graph, in parallel. If any subset of the edges is an appropriate cycle of total length less than or equal to k, the answer is YES; otherwise the answer is NO. Note that it is only necessary that *some* subset meet the requirement; it does not matter how many subsets fail. Checking a particular subset is done in polynomial time by adding the distances of the edges and verifying that the edges form a cycle that visits each vertex exactly once. Thus, the checking algorithm runs in polynomial time. Unfortunately, there are $2^{|\mathrm{E}|}$ subsets to check, so this algorithm cannot be converted to a polynomial time algorithm on a regular computer. Nor does anybody in the world know of any other polynomial time algorithm to solve TRAVELING SALESMAN on a regular computer, despite the fact that the problem has been studied extensively by many computer scientists for many years.

It turns out that there is a large collection of problems with this property: We know efficient non-deterministic algorithms, but we do not know if there are efficient deterministic algorithms. At the same time, we have not been able to prove that any of these problems do *not* have efficient deterministic algorithms. This class of problems is called $\mathcal{NP}$**-complete**. What is truly strange and fascinating about $\mathcal{NP}$-complete problems is that if anybody ever finds the solution to any one of them that runs in polynomial time on a regular computer, then by a series of reductions, every other problem that is in $\mathcal{NP}$ can also be solved in polynomial time on a regular computer!

Define a problem to be $\mathcal{NP}$**-hard** if *any* problem in $\mathcal{NP}$ can be reduced to X in polynomial time. Thus, X is *as hard as* any problem in $\mathcal{NP}$. A problem X is defined to be $\mathcal{NP}$-complete if

1. X is in $\mathcal{NP}$, and
2. X is $\mathcal{NP}$-hard.

The requirement that a problem be $\mathcal{NP}$-hard might seem to be impossible, but in fact there are hundreds of such problems, including TRAVELING SALESMAN. Another such problem is called CLIQUE.

CLIQUE
Input: An arbitrary undirected graph **G** and an integer k.
Output: YES if there is a complete subgraph of at least k vertices, and NO otherwise.

Nobody knows whether there is a polynomial time solution for CLIQUE, but if such an algorithm is found for CLIQUE *or* for TRAVELING SALESMAN, then that solution can be modified to solve the other, or any other problem in $\mathcal{NP}$, in polynomial time.

The primary theoretical advantage of knowing that a problem P1 is $\mathcal{NP}$-complete is that it can be used to show that another problem P2 is $\mathcal{NP}$-complete. This is done by finding a polynomial time reduction of P1 to P2. Because we already know that all problems in $\mathcal{NP}$ can be reduced to P1 in polynomial time (by the definition of $\mathcal{NP}$-complete), we now know that all problems can be reduced to P2 as well by the simple algorithm of reducing to P1 and then from there reducing to P2.

There is a practical advantage to knowing that a problem is $\mathcal{NP}$-complete. It relates to knowing that if a polynomial time solution can be found for *any* problem that is $\mathcal{NP}$-complete, then a polynomial solution can be found for *all* such problems. The implication is that,

1. Because no one has yet found such a solution, it must be difficult or impossible to do; and
2. Effort to find a polynomial time solution for one $\mathcal{NP}$-complete problem can be considered to have been expended for all $\mathcal{NP}$-complete problems.

How is $\mathcal{NP}$-completeness of practical significance for typical programmers? Well, if your boss demands that you provide a fast algorithm to solve a problem, she will not be happy if you come back saying that the best you could do was

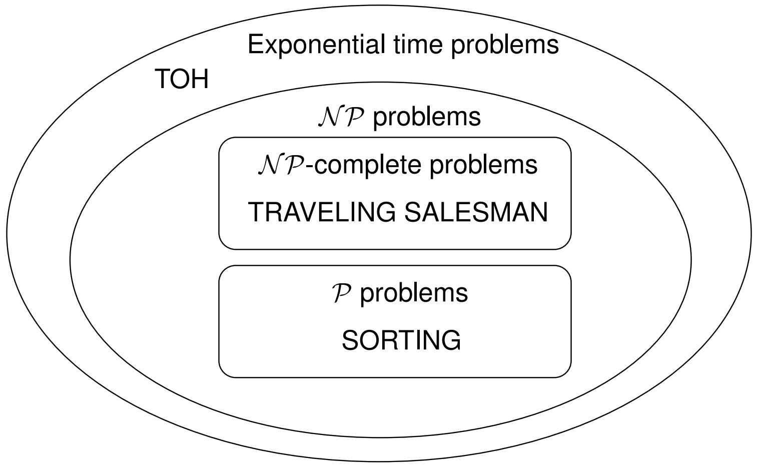


Figure 17.3 Our knowledge regarding the world of problems requiring exponential time or less. Some of these problems are solvable in polynomial time by a non-deterministic computer. Of these, some are known to be $\mathcal{NP}$-complete, and some are known to be solvable in polynomial time on a regular computer.

an exponential time algorithm. But, if you can prove that the problem is $\mathcal{NP}$-complete, while she still won't be happy, at least she should not be mad at you! By showing that her problem is $\mathcal{NP}$-complete, you are in effect saying that the most brilliant computer scientists for the last 50 years have been trying and failing to find a polynomial time algorithm for her problem.

Problems that are solvable in polynomial time on a regular computer are said to be in class $\mathcal{P}$. Clearly, all problems in $\mathcal{P}$ are solvable in polynomial time on a non-deterministic computer simply by neglecting to use the non-deterministic capability. Some problems in $\mathcal{NP}$ are $\mathcal{NP}$-complete. We can consider all problems solvable in exponential time or better as an even bigger class of problems because all problems solvable in polynomial time are solvable in exponential time. Thus, we can view the world of exponential-time-or-better problems in terms of Figure 17.3.

The most important unanswered question in theoretical computer science is whether $\mathcal{P} = \mathcal{NP}$. If they are equal, then there is a polynomial time algorithm for TRAVELING SALESMAN and all related problems. Because TRAVELING SALESMAN is known to be $\mathcal{NP}$-complete, if a polynomial time algorithm were to be found for this problem, then *all* problems in $\mathcal{NP}$ would also be solvable in polynomial time. Conversely, if we were able to prove that TRAVELING SALESMAN has an exponential time lower bound, then we would know that $\mathcal{P} \neq \mathcal{NP}$.

17.2.2 $\mathcal{NP}$-Completeness Proofs

To start the process of being able to prove problems are $\mathcal{NP}$-complete, we need to prove just one problem H is $\mathcal{NP}$-complete. After that, to show that any problem X is $\mathcal{NP}$-hard, we just need to reduce H to X. When doing $\mathcal{NP}$-completeness proofs, it is very important not to get this reduction backwards! If we reduce candidate problem X to known hard problem H, this means that we use H as a step to solving X. All that means is that we have found a (known) hard way to solve X. However, when we reduce known hard problem H to candidate problem X, that means we are using X as a step to solve H. And if we know that H is hard, that means X must also be hard.

So a crucial first step to getting this whole theory off the ground is finding one problem that is $\mathcal{NP}$-hard. The first proof that a problem is $\mathcal{NP}$-hard (and because it is in $\mathcal{NP}$, therefore $\mathcal{NP}$-complete) was done by Stephen Cook. For this feat, Cook won the first Turing award, which is the closest Computer Science equivalent to the Nobel Prize. The "grand-daddy" $\mathcal{NP}$-complete problem that Cook used is call SATISFIABILITY (or SAT for short).

A **Boolean expression** includes Boolean variables combined using the operators AND ($\cdot$), OR ($+$), and NOT (to negate Boolean variable x we write $\overline{x}$). A **literal** is a Boolean variable or its negation. A **clause** is one or more literals OR'ed together. Let E be a Boolean expression over variables $x_1, x_2, ..., x_n$. Then we define **Conjunctive Normal Form** (CNF) to be a boolean expression written as a series of clauses that are AND'ed together. For example,

$$E = (x_5 + x_7 + \overline{x_8} + x_{10}) \cdot (\overline{x_2} + x_3) \cdot (x_1 + \overline{x_3} + x_6)$$

is in CNF, and has three clauses. Now we can define the problem SAT.

SATISFIABILITY (SAT)

Input: A Boolean expression E over variables $x_1, x_2, ...$ in Conjunctive Normal Form.

Output: YES if there is an assignment to the variables that makes E true, NO otherwise.

Cook proved that SAT is $\mathcal{NP}$-hard. Explaining this proof is beyond the scope of this book. But we can briefly summarize it as follows. Any decision problem F can be recast as some language acceptance problem L:

$$F(I) = \text{YES} \Leftrightarrow L(I') = \text{ACCEPT}.$$

That is, if a decision problem F yields YES on input I, then there is a language L containing string I' where I' is some suitable transformation of input I. Conversely, if F would give answer NO for input I, then I's transformed version I' is not in the language L.

Turing machines are a simple model of computation for writing programs that are language acceptors. There is a "universal" Turing machine that can take as input a description for a Turing machine, and an input string, and return the execution of that machine on that string. This Turing machine in turn can be cast as a boolean expression such that the expression is satisfiable if and only if the Turing machine yields ACCEPT for that string. Cook used Turing machines in his proof because they are simple enough that he could develop this transformation of Turing machines to Boolean expressions, but rich enough to be able to compute any function that a regular computer can compute. The significance of this transformation is that *any* decision problem that is performable by the Turing machine is transformable to SAT. Thus, SAT is $\mathcal{NP}$-hard.

As explained above, to show that a decision problem X is $\mathcal{NP}$-complete, we prove that X is in $\mathcal{NP}$ (normally easy, and normally done by giving a suitable polynomial-time, nondeterministic algorithm) and then prove that X is $\mathcal{NP}$-hard. To prove that X is $\mathcal{NP}$-hard, we choose a known $\mathcal{NP}$-complete problem, say A. We describe a polynomial-time transformation that takes an *arbitrary* instance $\mathbf{I}$ of A to an instance $\mathbf{I}'$ of X. We then describe a polynomial-time transformation from $\mathbf{S}'$ to $\mathbf{S}$ such that $\mathbf{S}$ is the solution for $\mathbf{I}$. The following example provides a model for how an $\mathcal{NP}$-completeness proof is done.

3-SATISFIABILITY (3 SAT)

Input: A Boolean expression E in CNF such that each clause contains exactly 3 literals.

Output: YES if the expression can be satisfied, NO otherwise.

Example 17.1 3 SAT is a special case of SAT. Is 3 SAT easier than SAT? Not if we can prove it to be $\mathcal{NP}$-complete.

Theorem 17.1 *3 SAT is $\mathcal{NP}$-complete.*

Proof: Prove that 3 SAT is in $\mathcal{NP}$: Guess (nondeterministically) truth values for the variables. The correctness of the guess can be verified in polynomial time.

Prove that 3 SAT is $\mathcal{NP}$-hard: We need a polynomial-time reduction from SAT to 3 SAT. Let $\mathbf{E} = C_1 \cdot C_2 \cdot ... \cdot C_k$ be any instance of SAT. Our strategy is to replace any clause C_i that does not have exactly three literals with a set of clauses each having exactly three literals. (Recall that a literal can be a variable such as x, or the negation of a variable such as $\overline{x}$.) Let $C_i = x_1 + x_2 + ... + x_j$ where $x_1, ..., x_j$ are literals.

1. $j = 1$, so $C_i = x_1$. Replace C_i with C_i':

$$(x_1 + y + z) \cdot (x_1 + \overline{y} + z) \cdot (x_1 + y + \overline{z}) \cdot (x_1 + \overline{y} + \overline{z})$$

where y and z are variables not appearing in **E**. Clearly, C_i' is satisfiable if and only if (x_1) is satisfiable, meaning that x_1 is **`true`**.

2. $J = 2$, so $C_i = (x_1 + x_2)$. Replace C_i with

$$(x_1 + x_2 + z) \cdot (x_1 + x_2 + \overline{z})$$

where z is a new variable not appearing in **E**. This new pair of clauses is satisfiable if and only if $(x_1 + x_2)$ is satisfiable, that is, either x_1 or x_2 must be true.

3. $j > 3$. Replace $C_i = (x_1 + x_2 + \cdots + x_j)$ with

$$(x_1 + x_2 + z_1) \cdot (x_3 + \overline{z_1} + z_2) \cdot (x_4 + \overline{z_2} + z_3) \cdot ...$$

$$\cdot(x_{j-2} + \overline{z_{j-4}} + z_{j-3}) \cdot (x_{j-1} + x_j + \overline{z_{j-3}})$$

where $z_1, ..., z_{j-3}$ are new variables.

After appropriate replacements have been made for each C_i, a Boolean expression results that is an instance of 3 SAT. Each replacement is satisfiable if and only if the original clause is satisfiable. The reduction is clearly polynomial time.

For the first two cases it is fairly easy to see that the original clause is satisfiable if and only if the resulting clauses are satisfiable. For the case were we replaced a clause with more than three literals, consider the following.

1. If E is satisfiable, then E' is satisfiable: Assume x_m is assigned **`true`**. Then assign $z_t, t \leq m - 2$ as **`true`** and $z_k, t \geq m - 1$ as **`false`**. Then all clauses in Case (3) are satisfied.

2. If $x_1, x_2, ..., x_j$ are all **`false`**, then $z_1, z_2, ..., z_{j-3}$ are all **`true`**. But then $(x_{j-1} + x_{j-2} + \overline{z_{j-3}})$ is **`false`**.

□

Next we define the problem VERTEX COVER for use in further examples.

VERTEX COVER:
Input: A graph **G** and an integer k.
Output: YES if there is a subset **S** of the vertices in **G** of size k or less such that every edge of **G** has at least one of its endpoints in **S**, and NO otherwise.

Example 17.2 In this example, we make use of a simple conversion between two graph problems.

Theorem 17.2 *VERTEX COVER is $\mathcal{NP}$-complete.*

Proof: Prove that VERTEX COVER is in $\mathcal{NP}$: Simply guess a subset of the graph and determine in polynomial time whether that subset is in fact a vertex cover of size k or less.

Prove that VERTEX COVER is $\mathcal{NP}$-hard: We will assume that CLIQUE is already known to be $\mathcal{NP}$-complete. (We will see this proof in the next example. For now, just accept that it is true.)

Given that CLIQUE is $\mathcal{NP}$-complete, we need to find a polynomial-time transformation from the input to CLIQUE to the input to VERTEX COVER, and another polynomial-time transformation from the output for VERTEX COVER to the output for CLIQUE. This turns out to be a simple matter, given the following observation. Consider a graph **G** and a vertex cover **S** on **G**. Denote by $\mathbf{S}'$ the set of vertices in **G** but not in **S**. There can be no edge connecting any two vertices in $\mathbf{S}'$ because, if there were, then **S** would not be a vertex cover. Denote by $\mathbf{G}'$ the inverse graph for **G**, that is, the graph formed from the edges not in **G**. If **S** is of size k, then $\mathbf{S}'$ forms a clique of size $n - k$ in graph $\mathbf{G}'$. Thus, we can reduce CLIQUE to VERTEX COVER simply by converting graph **G** to $\mathbf{G}'$, and asking if $\mathbf{G}'$ has a VERTEX COVER of size $n - k$ or smaller. If YES, then there is a clique in **G** of size k; if NO then there is not. □

Example 17.3 So far, our $\mathcal{NP}$-completenss proofs have involved transformations between inputs of the same "type," such as from a Boolean expression to a Boolean expression or from a graph to a graph. Sometimes an $\mathcal{NP}$-completeness proof involves a transformation between types of inputs, as shown next.

Theorem 17.3 *CLIQUE is $\mathcal{NP}$-complete.*

Proof: CLIQUE is in $\mathcal{NP}$, because we can just guess a collection of *k* vertices and test in polynomial time if it is a clique. Now we show that CLIQUE is $\mathcal{NP}$-hard by using a reduction from SAT. An instance of SAT is a Boolean expression

$$B = C_1 \cdot C_2 \cdot ... \cdot C_m$$

whose clauses we will describe by the notation

$$C_i = y[i, 1] + y[i, 2] + ... + y[i, k_i]$$

where k_i is the number of literals in Clause c_i. We will transform this to an instance of CLIQUE as follows. We build a graph

$$G = \{v[i, j] | 1 \leq i \leq m, 1 \leq j \leq k_i\},$$

that is, there is a vertex in **G** corresponding to every literal in Boolean expression **B**. We will draw an edge between each pair of vertices $v[i_1, j_1]$ and $v[i_2, j_2]$ unless (1) they are two literals within the same clause ($i_1 = i_2$) or (2) they are opposite values for the same variable (i.e., one is negated and the other is not). Set $k = m$. Figure 17.4 shows an example of this transformation.

B is satisfiable if and only if **G** has a clique of size *k* or greater. **B** being satisfiable implies that there is a truth assignment such that at least one literal $y[i, j_i]$ is true for each i. If so, then these m literals must correspond to m vertices in a clique of size $k = m$. Conversely, if **G** has a clique of size *k* or greater, then the clique must have size exactly k (because no two vertices corresponding to literals in the same clause can be in the clique) and there is one vertex $v[i, j_i]$ in the clique for each i. There is a truth assignment making each $y[i, j_i]$ true. That truth assignment satisfies **B**.

We conclude that CLIQUE is $\mathcal{NP}$-hard, therefore $\mathcal{NP}$-complete. □

17.2.3 Coping with $\mathcal{NP}$-Complete Problems

Finding that your problem is $\mathcal{NP}$-complete might not mean that you can just forget about it. Traveling salesmen need to find reasonable sales routes regardless of the complexity of the problem. What do you do when faced with an $\mathcal{NP}$-complete problem that you must solve?

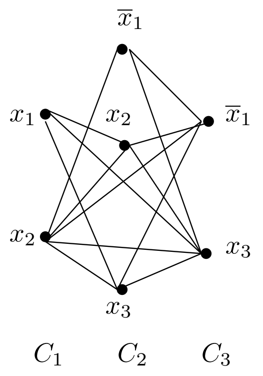


Figure 17.4 The graph generated from boolean expression $B = (x_1+x_2)\cdot(\overline{x_1}+x_2+x_3)\cdot(\overline{x_1}+x_3)$. Literals from the first clause are labeled C1, and literals from the second clause are labeled C2. There is an edge between every two pairs of vertices except when both vertices represent instances of literals from the same clause, or a negation of the same variable. Thus, the vertex labeled $C1 : y_1$ does not connect to the vertex labeled $C1 : y_2$ (because they are literals in the same clause) or the vertex labeled $C2 : \overline{y_1}$ (because they are opposite values for the same variable).

There are several techniques to try. One approach is to run only small instances of the problem. For some problems, this is not acceptable. For example, TRAVELING SALESMAN grows so quickly that it cannot be run on modern computers for problem sizes much over 20 cities, which is not an unreasonable problem size for real-life situations. However, some other problems in $\mathcal{NP}$, while requiring exponential time, still grow slowly enough that they allow solutions for problems of a useful size.

Consider the Knapsack problem from Section 16.2.1. We have a dynamic programming algorithm whose cost is $\Theta(nK)$ for n objects being fit into a knapsack of size K. But it turns out that Knapsack is $\mathcal{NP}$-complete. Isn't this a contradiction? Not when we consider the relationship between n and K. How big is K? Input size is typically $O(n \lg K)$ because the item sizes are smaller than K. Thus, $\Theta(nK)$ is exponential on input size.

This dynamic programming algorithm is tractable if the numbers are "reasonable." That is, we can successfully find solutions to the problem when nK is in the thousands. Such an algorithm is called a **pseudo-polynomial** time algorithm. This is different from TRAVELING SALESMAN which cannot possibly be solved when $n = 100$ given current algorithms.

A second approach to handling $\mathcal{NP}$-complete problems is to solve a special instance of the problem that is not so hard. For example, many problems on graphs are $\mathcal{NP}$-complete, but the same problem on certain restricted types of graphs is not as difficult. For example, while the VERTEX COVER and CLIQUE problems are $\mathcal{NP}$-complete in general, there are polynomial time solutions for bipartite graphs (i.e., graphs whose vertices can be separated into two subsets such that no pair of vertices within one of the subsets has an edge between them). 2-SATISFIABILITY (where every clause in a Boolean expression has at most two literals) has a polynomial time solution. Several geometric problems requre only polynomial time in two dimensions, but are $\mathcal{NP}$-complete in three dimensions or more. KNAPSACK is considered to run in polynomial time if the numbers (and K) are "small." Small here means that they are polynomial on n, the number of items.

In general, if we want to guarentee that we get the correct answer for an $\mathcal{NP}$-complete problem, we potentially need to examine all of the (exponential number of) possible solutions. However, with some organization, we might be able to either examine them quickly, or avoid examining a great many of the possible answers in some cases. For example, Dynamic Programming (Section 16.2) attempts to organize the processing of all the subproblems to a problem so that the work is done efficiently.

If we need to do a brute-force search of the entire solution space, we can use **backtracking** to visit all of the possible solutions organized in a solution tree. For example, SATISFIABILITY has 2^n possible ways to assign truth values to the n variables contained in the boolean expression being satisfied. We can view this as a tree of solutions by considering that we have a choice of making the first variable `true` or `false`. Thus, we can put all solutions where the first variable is `true` on one side of the tree, and the remaining solutions on the other. We then examine the solutions by moving down one branch of the tree, until we reach a point where we know the solution cannot be correct (such as if the current partial collection of assignments yields an unsatisfiable expression). At this point we backtrack and move back up a node in the tree, and then follow down the alternate branch. If this fails, we know to back up further in the tree as necessary and follow alternate branches, until finally we either find a solution that satisfies the expression or exhaust the tree. In some cases we avoid processing many potential solutions, or find a solution quickly. In others, we end up visiting a large portion of the 2^n possible solutions.

Banch-and-Bounds is an extension of backtracking that applies to **optimization problems** such as TRAVELING SALESMAN where we are trying to find the shortest tour through the cities. We traverse the solution tree as with backtracking. However, we remember the best value found so far. Proceeding down a given branch is equivalent to deciding which order to visit cities. So any node in the solution tree represents some collection of cities visited so far. If the sum of these distances exceeds the best tour found so far, then we know to stop pursuing this branch of the tree. At this point we can immediately back up and take another branch. If we have a quick method for finding a good (but not necessarily) best solution, we can use this as an initial bound value to effectively prune portions of the tree.

A third approach is to find an approximate solution to the problem. There are many approaches to finding approximate solutions. One way is to use a heuristic to solve the problem, that is, an algorithm based on a "rule of thumb" that does not always give the best answer. For example, the TRAVELING SALESMAN problem can be solved approximately by using the heuristic that we start at an arbitrary city and then always proceed to the next unvisited city that is closest. This rarely gives the shortest path, but the solution might be good enough. There are many other heuristics for TRAVELING SALESMAN that do a better job.

Some approximation algorithms have guaranteed performance, such that the answer will be within a certain percentage of the best possible answer. For example, consider this simple heuristic for the VERTEX COVER problem: Let M be a maximal (not necessarily maximum) **matching** in G. A matching pairs vertices (with connecting edges) so that no vertex is paired with more than one partner. Maximal means to pick as many pairs as possible, selecting them in some order until there are no more available pairs to select. Maximum means the matching that gives the most pairs possible for a given graph. If OPT is the size of a minimum vertex cover, then $|M| \leq 2 \cdot \text{OPT}$ because at least one endpoint of every matched edge must be in *any* vertex cover.

A better example of a guarenteed bound on a solution comes from simple heuristics to solve the BIN PACKING problem.

BIN PACKING:

Input: Numbers $x_1, x_2, ..., x_n$ between 0 and 1, and an unlimited supply of bins of size 1 (no bin can hold numbers whose sum exceeds 1).

Output: An assignment of numbers to bins that requires the fewest possible bins.

BIN PACKING (in its decision tree form) is known to be $\mathcal{NP}$-complete. One simple heuristic for solving this problem is to use a "first fit" approach. We put the first number in the first bin. We then put the second number in the first bin if it fits, otherwise we put it in the second bin. For each subsequent number, we simply go through the bins in the order we generated them and place the number in the first bin that fits. The number of bins used is no more than twice the sum of the numbers, because every bin (except perhaps one) must be at least half full. However, this "first fit" heuristic can give us a result that is much worse than optimal. Consider the following collection of numbers: 6 of $1/7 + \epsilon$, 6 of $1/3 + \epsilon$, and 6 of $1/2 + \epsilon$, where ϵ is a small, positive number. Properly organized, this requires 6 bins. But if done wrongly, we might end up putting the numbers into 10 bins.

A better heuristic is to use decreasing first fit. This is the same as first fit, except that we keep the bins sorted from most full to least full. Then when deciding where to put the next item, we place it in the fullest bin that can hold it. This is similar to the "best fit" heuristic for memory management discussed in Section 12.3. The significant thing about this heuristic is not just that it tends to give better performance than simple first fit. This decreasing first fit heurstic can be proven to require no more than 11/9 the optimal number of bins. Thus, we have a guarentee on how much inefficiency can result when using the heuristic.

The theory of $\mathcal{NP}$-completeness gives a technique for separating tractable from (probably) untractable problems. Recalling the algorithm for generating algorithms in Section 15.1, we can refine it for problems that we suspect are $\mathcal{NP}$-complete. When faced with a new problem, we might alternate between checking if it is tractable (that is, we try to find a polynomial-time solution) and checking if it is intractable (we try to prove the problem is $\mathcal{NP}$-complete). While proving that some problem is $\mathcal{NP}$-complete does not actually make our upper bound for our algorithm match the lower bound for the problem with certainty, it is nearly as good. Once we realize that a problem is $\mathcal{NP}$-complete, then we know that our next step must either be to redefine the problem to make it easier, or else use one of the "coping" strategies discussed in this section.

17.3 Impossible Problems

Even the best programmer sometimes writes a program that goes into an infinite loop. Of course, when you run a program that has not stopped, you do not know for sure if it is just a slow program or a program in an infinite loop. After "enough time," you shut it down. Wouldn't it be great if your compiler could look at your program and tell you before you run it that it might get into an infinite loop? Alternatively, given a program and a particular input, it would be useful to know if executing the program on that input will result in an infinite loop without actually running the program.

Unfortunately, the **Halting Problem**, as this is called, cannot be solved. There will never be a computer program that can positively determine, for an arbitrary program **P**, if **P** will halt for all input. Nor will there even be a computer program that can positively determine if arbitrary program **P** will halt for a specified input I.

How can this be? Programmers look at programs regularly to determine if they will halt. Surely this can be automated. As a warning to those who believe any program can be analyzed in this way, carefully examine the following code fragment before reading on.

```
while (n > 1)
  if (ODD(n))
    n = 3 * n + 1;
   else
     n = n / 2;
```

This is a famous piece of code. The sequence of values that is assigned to n by this code is sometimes called the **Collatz sequence** for input value n. Does this code fragment halt for all values of n? Nobody knows the answer. Every input that has been tried halts. But does it always halt? Note that for this code fragment, because we do not know if it halts, we also do not know an upper bound for its running time. As for the lower bound, we can easily show $\Omega(\log n)$(see Exercise 3.14).

Personally, I have faith that someday some smart person will completely analyze the Collitz function and prove once and for all that the code fragment halts for all values of n. Doing so may well give us techniques that advance our ability to do algorithm analysis in general. Unfortunately, proofs from **computability** — the branch of computer science that studies what is impossible to do with a computer — compel us to believe that there will always be another bit of program code that we cannot analyze. This comes as a result of the fact that the Halting Problem is unsolvable.

17.3.1 Uncountability

Before proving that the Halting Problem is unsolvable, we first prove that not all functions can be implemented as a computer program. This is so because the number of programs is much smaller than the number of possible functions.

A set is said to be **countable** (or **countably infinite** if it is a set with infinite members) if every member of the set can be uniquely assigned to a positive integer. A set is said to be **uncountable** (or **uncountably infinite**) if it is not possible to assign every member of the set to a positive integer.

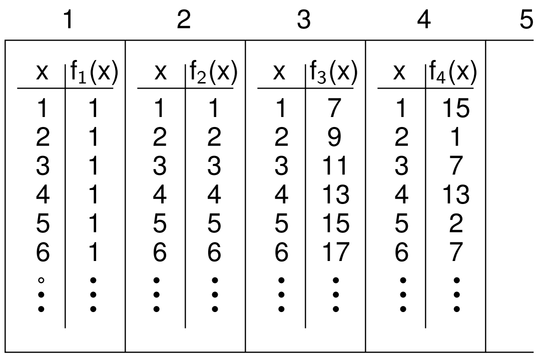
To understand what is meant when we say "assigned to a positive integer," imagine that there is an infinite row of bins, labeled 1, 2, 3, and so on. Take a set and start placing members of the set into bins, with at most one member per bin. If we can find a way to assign all of the members to bins, then the set is countable. For example, consider the set of positive even integers 2, 4, and so on. We can assign an integer i to bin $i/2$ (or, if we don't mind skipping some bins, then we can assign even number i to bin i). Thus, the set of even integers is countable. This should be no surprise, because intuitively there are "fewer" positive even integers than there are positive integers, even though both are infinite sets. But there are not really any more positive integers than there are positive even integers, because we can uniquely assign every positive integer to some positive even integer by simply assigning positive integer i to positive even integer $2i$.

On the other hand, the set of all integers is also countable, even though this set appears to be "bigger" than the set of positive integers. This is true because we can assign 0 to positive integer 1, 1 to positive integer 2, -1 to positive integer 3, 2 to positive integer 4, -2 to positive integer 5, and so on. In general, assign positive integer value i to positive integer value $2i$, and assign negative integer value $-i$ to positive integer value $2i + 1$. We will never run out of positive integers to assign, and we know exactly which positive integer every integer is assigned to. Because every integer gets an assignment, the set of integers is countably infinite.

Are the number of programs countable or uncountable? A program can be viewed as simply a string of characters (including special punctuation, spaces, and line breaks). Let us assume that the number of different characters that can appear in a program is P. (Using the ASCII character set, P must be less than 128, but the actual number does not matter). If the number of strings is countable, then surely the number of programs is also countable. We can assign strings to the bins as follows. Assign the null string to the first bin. Now, take all strings of one character, and assign them to the next P bins in "alphabetic" or ASCII code order. Next, take all strings of two characters, and assign them to the next P^2 bins, again in ASCII code order working from left to right. Strings of three characters are likewise assigned to bins, then strings of length four, and so on. In this way, a string of any given length can be assigned to some bin.

By this process, any string of finite length is assigned to some bin. So any program, which is merely a string of finite length, is assigned to some bin. Because all programs are assigned to some bin, the set of all programs is countable. Naturally most of the strings in the bins are not legal programs, but this is irrelevant. All that matters is that the strings that *do* correspond to programs are also in the bins.

Now we consider the number of possible functions. To keep things simple, assume that all functions take a single positive integer as input and yield a single positive integer as output. We will call such functions **integer functions**. A function is simply a mapping from input values to output values. Of course, not all computer programs literally take integers as input and yield integers as output. However, everything that computers read and write is essentially a series of num-



x	f_1(x)	x	f_2(x)	x	f_3(x)	x	f_4(x)
1	1	1	1	1	7	1	15
2	1	2	2	2	9	2	1
3	1	3	3	3	11	3	7
4	1	4	4	4	13	4	13
5	1	5	5	5	15	5	2
6	1	6	6	6	17	6	7
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Figure 17.5 An illustration of assigning functions to bins.

bers, which may be interpreted as letters or something else. Any useful computer program's input and output can be coded as integer values, so our simple model of computer input and output is sufficiently general to cover all possible computer programs.

We now wish to see if it is possible to assign all of the integer functions to the infinite set of bins. If so, then the number of functions is countable, and it might then be possible to assign every integer function to a program. If the set of integer functions cannot be assigned to bins, then there will be integer functions that must have no corresponding program.

Imagine each integer function as a table with two columns and an infinite number of rows. The first column lists the positive integers starting at 1. The second column lists the output of the function when given the value in the first column as input. Thus, the table explicitly describes the mapping from input to output for each function. Call this a **function table**.

Next we will try to assign function tables to bins. To do so we must order the functions, but it does not matter what order we choose. For example, Bin 1 could store the function that always returns 1 regardless of the input value. Bin 2 could store the function that returns its input. Bin 3 could store the function that doubles its input and adds 5. Bin 4 could store a function for which we can see no simple relationship between input and output.[2] These four functions as assigned to the first four bins are shown in Figure 17.5.

Can we assign every function to a bin? The answer is no, because there is always a way to create a new function that is not in any of the bins. Suppose that

[2]There is no requirement for a function to have any discernible relationship between input and output. A function is simply a mapping of inputs to outputs, with no constraint on how the mapping is determined.

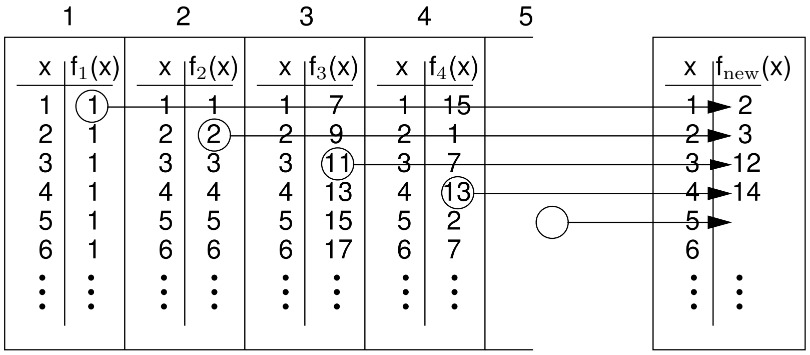


Figure 17.6 Illustration for the argument that the number of integer functions is uncountable.

somebody presents a way of assigning functions to bins that they claim includes all of the functions. We can build a new function that has not been assigned to any bin, as follows. Take the output value for input 1 from the function in the first bin. Call this value $F_1(1)$. Add 1 to it, and assign the result as the output of a new function for input value 1. Regardless of the remaining values assigned to our new function, it must be different from the first function in the table, because the two give different outputs for input 1. Now take the output value for 2 from the second function in the table (known as $F_2(2)$). Add 1 to this value and assign it as the output for 2 in our new function. Thus, our new function must be different from the function of Bin 2, because they will differ at least at the second value. Continue in this manner, assigning $F_{new}(i) = F_i(i) + 1$ for all values i. Thus, the new function must be different from any function F_i at least at position i. This procedure for constructing a new function not already in the table is called **diagonalization**. Because the new function is different from every other function, it must not be in the table. This is true no matter how we try to assign functions to bins, and so the number of integer functions is uncountable. The significance of this is that not all functions can possibly be assigned to programs, so there *must* be functions with no corresponding program. Figure 17.6 illustrates this argument.

17.3.2 The Halting Problem Is Unsolvable

While there might be intellectual appeal to knowing that there exists *some* function that cannot be computed by a computer program, does this mean that there is any such *useful* function? After all, does it really matter if no program can compute a “nonsense” function such as shown in Bin 4 of Figure 17.5? Now we will prove that the Halting Problem cannot be computed by any computer program. The proof is by contradiction.

We begin by assuming that there is a function named **halt** that can solve the Halting Problem. Obviously, it is not possible to write out something that does not exist, but here is a plausible sketch of what a function to solve the Halting Problem might look like if it did exist. Function **halt** takes two inputs: a string representing the source code for a program or function, and another string representing the input that we wish to determine if the input program or function halts on. Function **halt** does some work to make a decision (which is encasulated into some fictitious function named **PROGRAM_HALTS**). Function **halt** then returns **true** if the input program or function does halt on the given input, and **false** otherwise.

```
bool halt(String prog, String input) {
  if (PROGRAM_HALTS(prog, input))
    return true;
  else
    return false;
}
```

We now will examine two simple functions that clearly can exist because the complete source code for them is presented here:

```
// Return true if "prog" halts when given itself as input
bool selfhalt(String prog) {
  if (halt(prog, prog))
    return true;
  else
    return false;
}

// Return the reverse of what selfhalt returns on "prog"
void contrary(String prog) {
  if (selfhalt(prog))
    while (true); // Go into an infinite loop
}
```

What happens if we make a program whose sole purpose is to execute the function **contrary** and run that program with itself as input? One possibility is that the call to **selfhalt** returns **true**; that is, **selfhalt** claims that **contrary** will halt when run on itself. In that case, **contrary** goes into an infinite loop (and thus does not halt). On the other hand, if **selfhalt** returns **false**, then **halt** is proclaiming that **contrary** does not halt on itself, and **contrary** then returns, that is, it halts. Thus, **contrary** does the contrary of what **halt** says that it will do.

The action of **`contrary`** is logically inconsistent with the assumption that **`halt`** solves the Halting Problem correctly. There are no other assumptions we made that might cause this inconsistency. Thus, by contradiction, we have proved that **`halt`** cannot solve the Halting Problem correctly, and thus there is no program that can solve the Halting Problem.

Now that we have proved that the Halting Problem is unsolvable, we can use reduction arguments to prove that other problems are also unsolvable. The strategy is to assume the existence of a computer program that solves the problem in question and use that program to solve another problem that is already known to be unsolvable.

Example 17.4 Consider the following variation on the Halting Problem. Given a computer program, will it halt when its input is the empty string (i.e., will it halt when it is given no input)? To prove that this problem is unsolvable, we will employ a standard technique for computability proofs: Use a computer program to modify another computer program.

Proof: Assume that there is a function **`Ehalt`** that determines whether a given program halts when given no input. Recall that our proof for the Halting Problem involved functions that took as parameters a string representing a program and another string representing an input. Consider another function **`combine`** that takes a program P and an input string I as parameters. Function **`combine`** modifies P to store I as a static variable S and further modifies all calls to input functions within P to instead get their input from S. Call the resulting program P'. It should take no stretch of the imagination to believe that any decent compiler could be modified to take computer programs and input strings and produce a new computer program that has been modified in this way. Now, take P' and feed it to **`Ehalt`**. If **`Ehalt`** says that P' will halt, then we know that P would halt on input I. In other words, we now have a procedure for solving the original Halting Problem. The only assumption that we made was the existence of **`Ehalt`**. Thus, the problem of determining if a program will halt on no input must be unsolvable. □

Example 17.5 For arbitrary program P, does there exist *any* input for which P halts?

Proof: This problem is also uncomputable. Assume that we had a function **`Ahalt`** that, when given program P as input would determine if there is some input for which P halts. We could modify our compiler (or write a function as part of a program) to take P and some input string w, and modify it so that w is hardcoded inside P, with P reading no input. Call this modified program P'. Now, P' always behaves the same way regardless of its input, because it ignores all input. However, because w is now hardwired inside of P', the behavior we get is that of P when given w as input. So, P' will halt on any arbitrary input if and only if P would halt on input w. We now feed P' to function **`Ahalt`**. If **`Ahalt`** could determine that P' halts on some input, then that is the same as determining that P halts on input w. But we know that that is impossible. Therefore, **`Ahalt`** cannot exist. □

There are many things that we would like to have a computer do that are unsolvable. Many of these have to do with program behavior. For example, proving that an arbitrary program is "correct," that is, proving that a program computes a particular function, is a proof regarding program behavior. As such, what can be accomplished is severely limited. Some other unsolvable problems include:

- Does a program halt on every input?
- Does a program compute a particular function?
- Do two programs compute the same function?
- Does a particular line in a program get executed?

This does *not* mean that a computer program cannot be written that works on special cases, possibly even on most programs that we would be interested in checking. For example, some **C** compilers will check if the control expression for a **`while`** loop is a constant expression that evaluates to **`false`**. If it is, the compiler will issue a warning that the **`while`** loop code will never be executed. However, it is not possible to write a computer program that can check for *all* input programs whether a specified line of code will be executed when the program is given some specified input.

Another unsolvable problem is whether a program contains a computer virus. The property "contains a computer virus" is a matter of behavior. Thus, it is not possible to determine positively whether an arbitrary program contains a computer virus. Fortunately, there are many good heuristics for determining if a program is likely to contain a virus, and it is usually possible to determine if a program contains a particular virus, at least for the ones that are now known. Real virus checkers do a pretty good job, but, it will always be possible for malicious people to invent new viruses that no existing virus checker can recognize.

17.4 Further Reading

The classic text on the theory of $\mathcal{NP}$-completeness is *Computers and Intractability: A Guide to the Theory of $\mathcal{NP}$-completeness* by Garey and Johnston [GJ79]. *The Traveling Salesman Problem*, edited by Lawler et al. [LLKS85], discusses many approaches to finding an acceptable solution to this particular $\mathcal{NP}$-complete problem in a reasonable amount of time.

For more information about the Collatz function see "On the Ups and Downs of Hailstone Numbers" by B. Hayes [Hay84], and "The $3x + 1$ Problem and its Generalizations" by J.C. Lagarias [Lag85].

For an introduction to the field of computability and impossible problems, see *Discrete Structures, Logic, and Computability* by James L. Hein [Hei03].

17.5 Exercises

17.1 Consider this algorithm for finding the maximum element in an array: First sort the array and then select the last (maximum) element. What (if anything) does this reduction tell us about the upper and lower bounds to the problem of finding the maximum element in a sequence? Why can we not reduce SORTING to finding the maximum element?

17.2 Use a reduction to prove that squaring an $n \times n$ matrix is just as expensive (asymptotically) as multiplying two $n \times n$ matrices.

17.3 Use a reduction to prove that multiplying two upper triangular $n \times n$ matrices is just as expensive (asymptotically) as multiplying two arbitrary $n \times n$ matrices.

17.4 **(a)** Explain why computing the factorial of n by multiplying all values from 1 to n together is an exponential time algorithm.

(b) Explain why computing an approximation to the factorial of n by making use of Stirling's formula (see Section 2.2) is a polynomial time algorithm.

17.5 Consider this algorithm for solving the CLIQUE problem. First, generate all subsets of the vertices containing exactly k vertices. There are $O(n^k)$ such subsets altogether. Then, check whether any subgraphcs induced by these subsets is complete. If this algorithm ran in polynomial time, what would be its significance? Why is this not a polynomial-time algorithm for the CLIQUE problem?

17.6 Write the 3 SAT expression obtained from the reduction of SAT to 3 SAT described in Section 17.2.1 for the expression

$$(a + b + \overline{c} + d) \cdot (\overline{d}) \cdot (\overline{b} + \overline{c}) \cdot (\overline{a} + b) \cdot (a + c) \cdot (b).$$

Is this expression satisfiable?

17.7 Draw the graph obtained by the reduction of SAT to the CLIQUE problem given in Section 17.2.1 for the expression

$$(a + \overline{b} + c) \cdot (\overline{a} + b + \overline{c}) \cdot (\overline{a} + b + c) \cdot (a + \overline{b} + \overline{c}).$$

Is this expression satisfiable?

17.8 A **Hamiltonian cycle** in graph **G** is a cycle that visits every vertex in the graph exactly once before returning to the start vertex. The problem HAMILTONIAN CYCLE asks whether graph **G** does in fact contain a Hamiltonian cycle. Assuming that HAMILTONIAN CYCLE is $\mathcal{NP}$-complete, prove that the decision-problem form of TRAVELING SALESMAN is $\mathcal{NP}$-complete.

17.9 Assuming that VERTEX COVER is $\mathcal{NP}$-complete, prove that CLIQUE is also $\mathcal{NP}$-complete by finding a polynomial time reduction from VERTEX COVER to CLIQUE.

17.10 We define the problem INDEPENDENT SET as follows.

INDEPENDENT SET
Input: A graph **G** and an integer k.
Output: YES if there is a subset **S** of the vertices in **G** of size k or greater such that no edge connects any two vertices in **S**, and NO otherwise.

Assuming that CLIQUE is $\mathcal{NP}$-complete, prove that INDEPENDENT SET is $\mathcal{NP}$-complete.

17.11 Define the problem PARTITION as follows:

PARTITION
Input: A collection of integers.
Output: YES if the collection can be split into two such that the sum of the integers in each partition sums to the same amount. NO otherwise.

(a) Assuming that PARTITION is $\mathcal{NP}$-complete, prove that BIN PACKING is $\mathcal{NP}$-complete.

(b) Assuming that PARTITION is $\mathcal{NP}$-complete, prove that KNAPSACK is $\mathcal{NP}$-complete.

17.12 Imagine that you have a problem **P** that you know is $\mathcal{NP}$-complete. For this problem you have two algorithms to solve it. For each algorithm, some problem instances of **P** run in polynomial time and others run in exponential time (there are lots of heuristic-based algorithms for real $\mathcal{NP}$-complete problems with this behavior). You can't tell beforehand for any given problem instance whether it will run in polynomial or exponential time on either algorithm. However, you do know that for every problem instance, at least one of the two algorithms will solve it in polynomial time.

- **(a)** What should you do?
- **(b)** What is the running time of your solution?
- **(c)** What does it say about the question of $\mathcal{P} = \mathcal{NP}$ if the conditions described in this problem existed?

17.13 The last paragraph of Section 17.2.3 discusses a strategy for developing a solution to a new problem by alternating between finding a polynomial time solution and proving the problem $\mathcal{NP}$-complete. Refine the "algorithm for designing algorithms" from Section 15.1 to incorporate identifying and dealing with $\mathcal{NP}$-complete problems.

17.14 Prove that the set of real numbers is uncountable. Use a proof similar to the proof in Section 17.3.1 that the set of integer functions is uncountable.

17.15 Prove, using a reduction argument such as given in Section 17.3.2, that the problem of determining if an arbitrary program will print any output is unsolvable.

17.16 Prove, using a reduction argument such as given in Section 17.3.2, that the problem of determining if an arbitrary program executes a particular statement within that program is unsolvable.

17.17 Prove, using a reduction argument such as given in Section 17.3.2, that the problem of determining if two arbitrary programs halt on exactly the same inputs is unsolvable.

17.18 Prove, using a reduction argument such as given in Section 17.3.2, that the problem of determining whether there is some input on which two arbitrary programs will both halt is unsolvable.

17.19 Prove, using a reduction argument such as given in Section 17.3.2, that the problem of determining whether an arbitrary program halts on all inputs is unsolvable.

17.20 Prove, using a reduction argument such as given in Section 17.3.2, that the problem of determining whether an arbitrary program computes a specified function is unsolvable.

17.21 Consider a program named COMP that takes two strings as input. It returns **`TRUE`** if the strings are the same. It returns **`FALSE`** if the strings are different. Why doesn't the argument that we used to prove that a program to solve the halting problem does not exist work to prove that COMP does not exist?

17.6 Projects

17.1 Implement VERTEX COVER; that is, given graph **G** and integer k, answer the question of whether or not there is a vertex cover of size k or less. Begin by using a brute-force algorithm that checks all possible sets of vertices of size k to find an acceptable vertex cover, and measure the running time on a number of input graphs. Then try to reduce the running time through the use of any heuristics you can think of. Next, try to find approximate solutions to the problem in the sense of finding the smallest set of vertices that forms a vertex cover.

17.2 Implement KNAPSACK (see Section 16.2). Measure its running time on a number of inputs. What is the largest practical input size for this problem?

17.3 Implement an approximation of TRAVELING SALESMAN; that is, given a graph **G** with costs for all edges, find the cheapest cycle that visits all vertices in **G**. Try various heuristics to find the best approximations for a wide variety of input graphs.

17.4 Write a program that, given a positive integer n as input, prints out the Collatz sequence for that number. What can you say about the types of integers that have long Collatz sequences? What can you say about the length of the Collatz sequence for various types of integers?