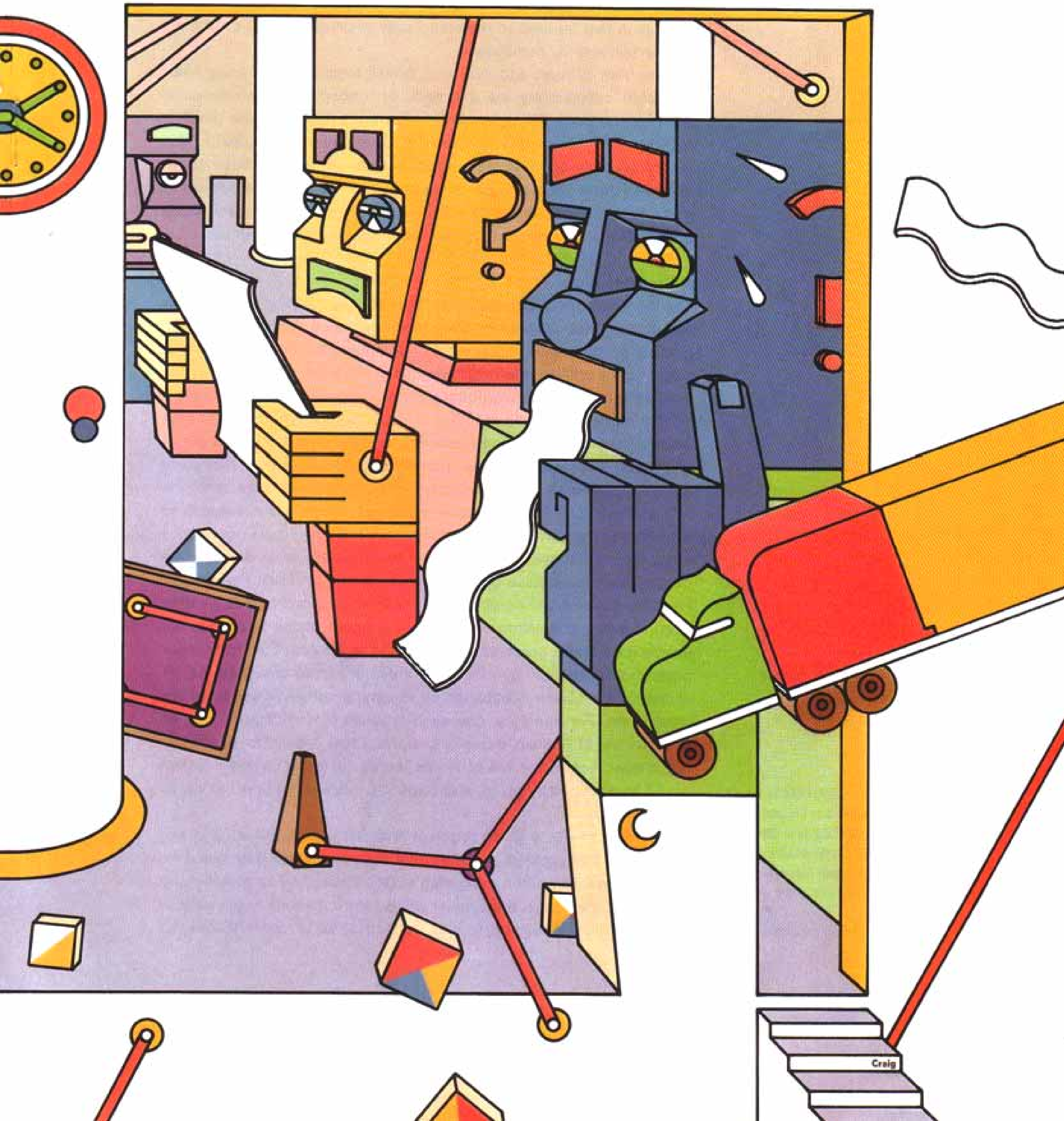
# The Limits to Computation

by Ronald L. Graham and Michael R. Garey

Even with the help of modern computers, mathematicians often fail to find adequate solutions to many complex problems of practical significance. Recently, several important breakthroughs have been achieved.



*Computation time is shown for various time complexity functions and for several values of size* n *for a problem instance. See text on p. 180.*

Writers of science fiction are fond of imagining colossal, anthropomorphic supercomputers, capable of solving almost instantly any problem posed to them. These fantastic machines are in many ways natural extrapolations of the digital computers of the 1970s, which already have become the most sophisticated tools ever devised by mankind. As one views the rapid progress in computer technology and the wide variety of tasks currently performed using computers, it seems reasonable that supercomputers will someday become reality and that there is in fact no limit to the complexity of problems that eventually will be solvable by computers.

More than 40 years ago, however, British logician Alan Turing demonstrated conclusively the existence of fundamental limitations on what can be done using computers. His discoveries laid the groundwork for what is known as the theory of computational complexity. This theory seeks to quantify the amount of time and other resources required to solve problems on computers and has been the subject of particularly intense research activity during the last ten years. Quite recently a number of startling new results have been obtained.

## Computer algorithms

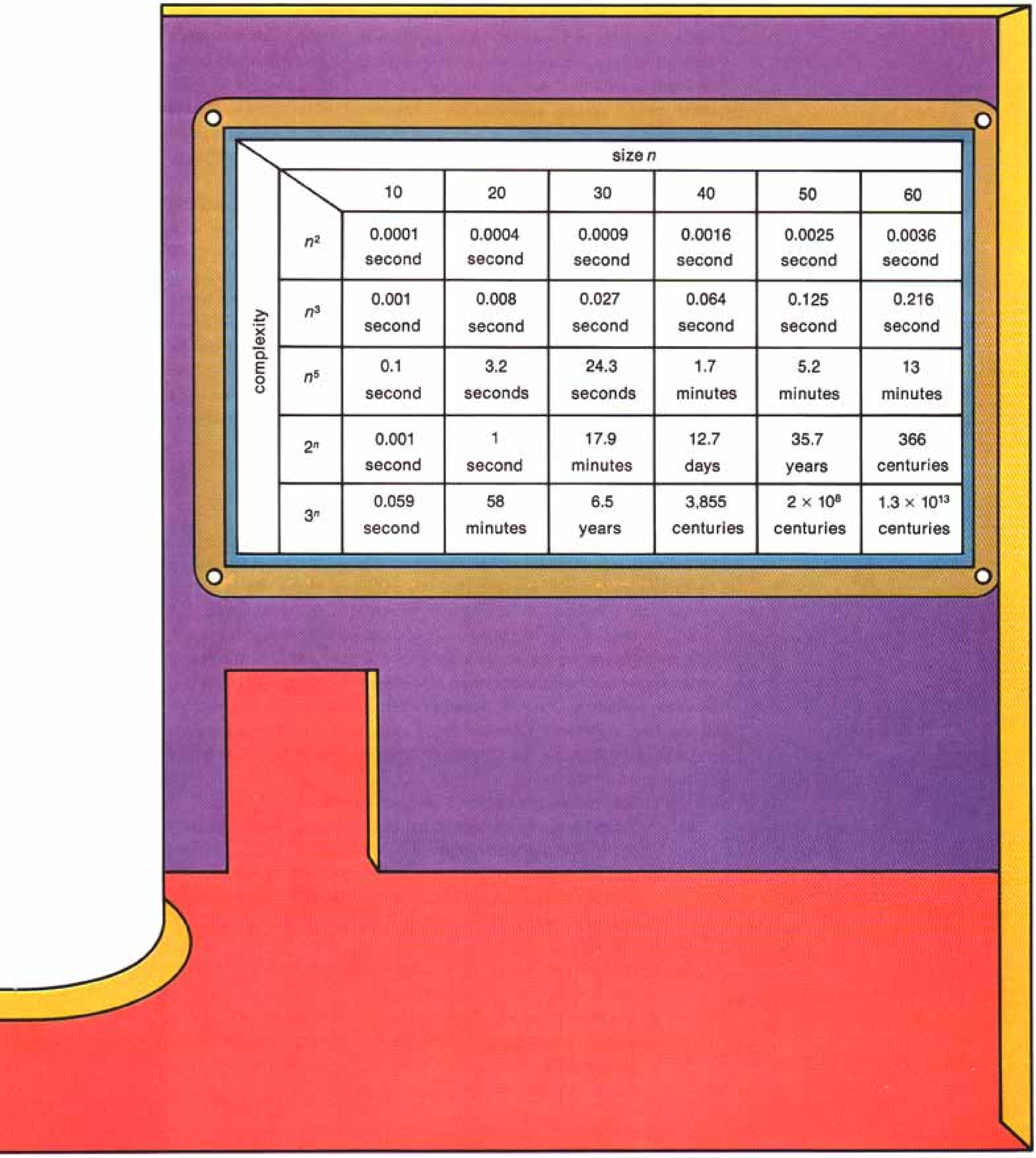
The theory of computational complexity studies the properties of general step-by-step problem-solving methods called algorithms. A great many algorithms are familiar tools of computation, although that term might not be used in referring to them. For instance, many people have learned (and probably forgotten) an algorithm for determining the square root of any given number. Also well known are algorithms for computing baseball batting averages, for determining the area of any given triangle, for finding the greatest common divisor of two given numbers, and for summing arbitrary lists of arbitrarily large numbers.

These examples indicate the very general capabilities possessed by algorithms. They are intended to solve not just one single occurrence of a problem but every occurrence of that problem, even though each occurrence may involve different values of the problem parameters. For this reason it will be convenient to think of a problem as a rather general entity, consisting of certain parameters whose values are left unspecified and a prescription of what is to be constructed or computed from those parameters once they have been specified. An instance of a problem is obtained by assigning actual values to all the parameters. For example, one such problem is that of arranging an arbitrary list of English words into alphabetical order. The parameter left unspecified is the list of words, so an instance of this problem would be obtained by giving a specific list, such as the first 100 words of this article.

An algorithm for a given problem must be capable of solving any instance of that problem. In order that others may use it or that it be executed on a computer, every step must be specified completely and in sufficient detail that there never will be any ambiguity about what to do next. Before going on, it may be instructive to consider how an

***RONALD L. GRAHAM*** *is Head, Discrete Mathematics Department, and* ***MICHAEL R. GAREY*** *is a Member of the Technical Staff, Mathematics Research Center, Bell Laboratories, Murray Hill, New Jersey.*

*Illustrations by John Craig*



| complexity | size $n$ = 10 | 20 | 30 | 40 | 50 | 60 |
|---|---|---|---|---|---|---|
| $n^2$ | 0.0001 second | 0.0004 second | 0.0009 second | 0.0016 second | 0.0025 second | 0.0036 second |
| $n^3$ | 0.001 second | 0.008 second | 0.027 second | 0.064 second | 0.125 second | 0.216 second |
| $n^5$ | 0.1 second | 3.2 seconds | 24.3 seconds | 1.7 minutes | 5.2 minutes | 13 minutes |
| $2^n$ | 0.001 second | 1 second | 17.9 minutes | 12.7 days | 35.7 years | 366 centuries |
| $3^n$ | 0.059 second | 58 minutes | 6.5 years | 3,855 centuries | $2 \times 10^8$ centuries | $1.3 \times 10^{13}$ centuries |

algorithm for alphabetizing a list of words could be so specified. Most natural languages have built-in ambiguities that make such specificity difficult, although people with common backgrounds often will agree on what is meant. The usual way of specifying an algorithm, in such detail that it can be executed on a computer, is to express it as a computer program written in a precise computer language. The input to such a computer program is used to describe the particular instance to be solved.

Instead of thinking in terms of any single existing computer, mathematicians have found it useful to introduce an abstract model of a computer, called a Turing machine. This model is conceptually simple, yet broad enough to reflect accurately the behavior of any existing or planned computer. Thus the Turing machine provides a common framework for mathematicians' investigations into the properties of algorithms. For the purposes of this article, however, it will not be necessary to define a Turing machine in rigorous detail. It will suffice simply to think in terms of any standard computer, and the following discussions will not depend on any special capabilities possessed by one computer but not another.

**An alphabetizing algorithm**

One simple algorithm for alphabetizing an arbitrary list of words can be described as follows. Assume that the positions in the list are numbered from 1 through $m$ in order, in which $m$ is the total number of words on the list. The algorithm will scan repeatedly through the list, interchanging adjacent words whenever they are not in alphabetical order, until the whole list has been alphabetized. To do this, it uses two auxiliary variables. The first, called POINTER, will always be the number of the next position on the list to be examined. The second, called CHANGE, will always be the number of interchanges made so far in the current scan of the list.

*step 1:* Set POINTER to position 1 and set CHANGE to 0.

*step 2:* Compare the two words in list positions POINTER and POINTER + 1 to find the leftmost place in which they differ. If the word in position POINTER + 1 has the alphabetically earlier letter in this place (no letter or "blank" is considered earlier than "a"), then interchange the positions of the two words and add 1 to CHANGE.

*step 3:* Increase POINTER by 1. If POINTER is less than $m$, return to *step 2* to continue this scan of the list.

*step 4:* (Reaching this step implies that POINTER equals $m$, so the algorithm is at the end of the list.) If CHANGE is 0, the list is now in alphabetical order. If CHANGE is larger than 0, however, return to *step 1* to begin another scan of the list.

## Undecidable problems

In 1936 Turing proved the existence of a class of problems, called undecidable problems, which are so difficult that no algorithm for solving them can ever be devised. This astounding discovery, which ranks among the most profound intellectual achievements of the 20th century, is especially significant because computers seem to embody all the logical and computational capabilities imaginable. Thus, whatever instruments man may ever have at his disposal, the undecidable problems are destined to remain forever beyond his computational reach.

Turing showed specifically that one particular problem, called the Halting Problem, is undecidable. The Halting Problem is that of deciding, given an arbitrary computer program and an arbitrary input for that program, whether or not the program will eventually stop when given that input. An algorithm for this problem would have obvious uses, for example, in "debugging" computer programs and in avoiding embarrassing computer-budget overruns.

The proof that such an algorithm can never be given proceeds by assuming the existence of a program P* that actually can solve the Halting Problem; *i.e.,* one that can determine for any given ~~problem~~ program P and specified input I whether or not P will halt when applied to I. Suppose one's attention is now restricted to those programs P that themselves answer questions about programs, so that the inputs I to which they are applied are actually descriptions of other programs. In such situations P* can be used to decide whether or not a given program P of this type will halt when applied to a description of P itself. Suppose P* were so structured that, whenever it decides that P halts, it prints "P halts" indefinitely, without stopping; otherwise, it simply prints "P does not halt" and stops. Next consider what P* will do when asked to decide about *itself*. It either will continue printing "P* halts" indefinitely, without ever halting, or will print "P* does not halt" and then halt. Hence, whatever the program decides about itself, it obviously will contradict that decision by the manner in which it prints its answer. Because only one assumption was made—that a program for solving the Halting Problem existed—and because that assumption leads to a contradiction, it must be admitted that no such program can exist. The Halting Problem is undecidable.

In the years since Turing's initial discovery, many other problems from a variety of mathematical realms have been shown to be undecidable. One of the most picturesque is known as the Tiling Problem. In the Tiling Problem, one is given a finite variety of equal-sized square tiles. Each tile has each of its four edges colored with a specific color and is given a fixed orientation, with its edges running horizontally and vertically. It is assumed that one can obtain as many copies of each kind of tile as are needed. These can be put together, like dominoes, to form various configurations. In doing this, it is required that abutting edges of adjacent tiles be of the same color and that no tile be rotated from its initial orientation. The question to be decided is whether or not the given tiles can be used to form increasingly large, completely filled-in, *In this variation of the Tiling Problem, copies of 19 tiles are used to form arbitrarily large, completely filled-in, square configurations with the requirement that the purple, white, and blue tile be used at least once; other conditions described in text also hold. Depicted is a portion of a solid tiling that appears indefinitely extensible in all directions. Yet proving this property seems beyond the scope of present-day mathematics.*

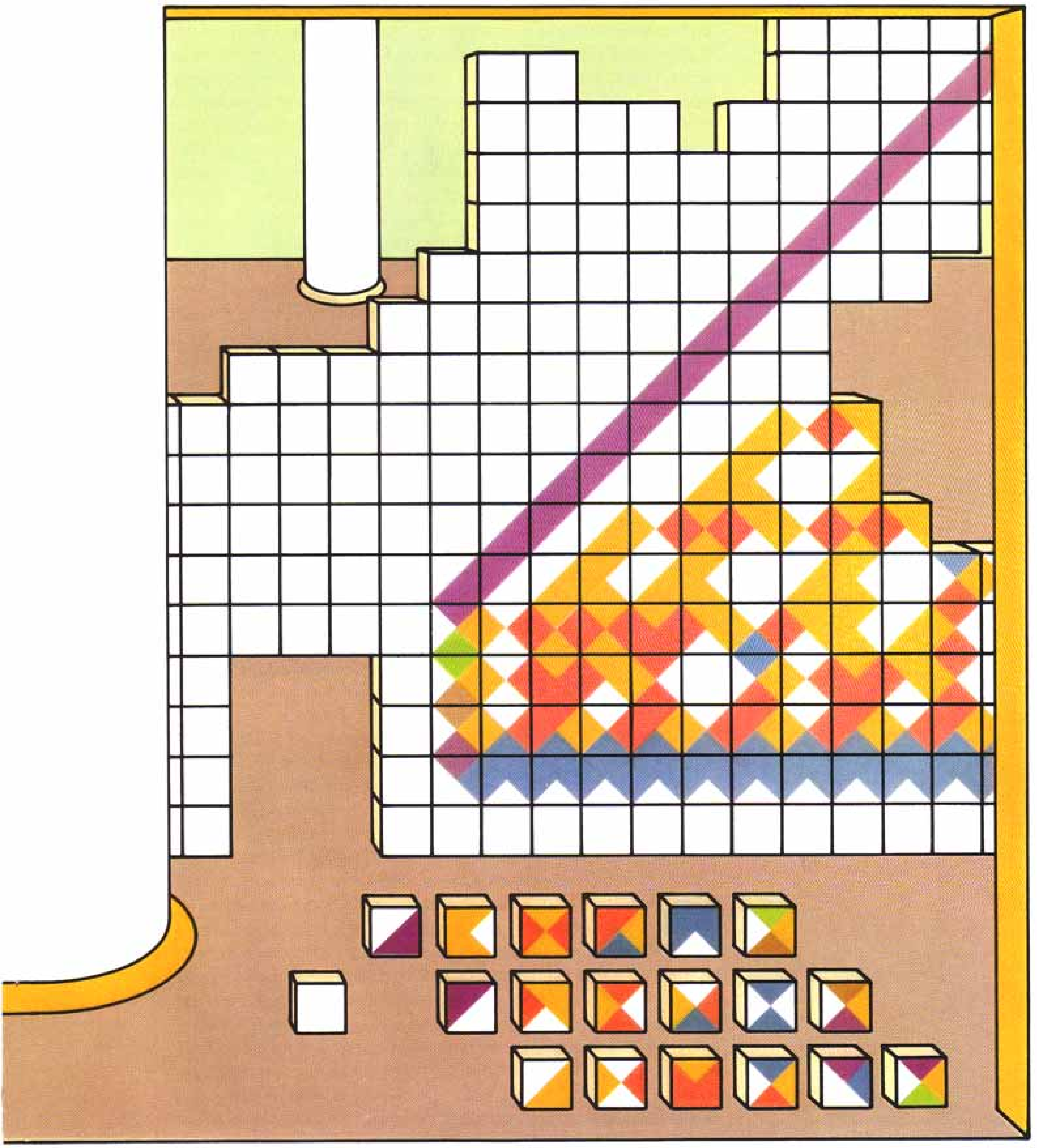
square configurations. If such tilings are possible, the given set of tile varieties is said to be solvable.

Notice that there is an effective, though possibly lengthy, method for demonstrating that a given set of tiles is *not* solvable. For if it is not solvable, then there must exist a square of some size, say 1,000 by 1,000 tiles, that cannot be formed. Thus, if one considers each size square in turn, in order of increasing size, and examines all possible ways of forming a square of that size with the given set of tiles, one will eventually discover some size that cannot possibly be formed. However, if the set of tiles *is* solvable, this procedure will never demonstrate this fact conclusively because it will never terminate.

How then could it ever be shown that a set is solvable? One approach is as follows. Suppose one were able to form a particular square configuration R, say of size *r* by *r*, in such a way that both horizontal boundaries had the same pattern of colors and that both vertical boundaries had the same pattern of colors. Then it is easy to see that any square could be formed with side length a multiple of *r* simply by placing together copies of R. Of course, every other size square occurs as a portion of one of these squares, so this method would show that the given set of tiles is solvable.

If every solvable set of tiles could be used to form some such configuration R, then one would have an algorithm for the Tiling Problem. First, all possible 2-by-2 squares could be formed from the tiles, then all the 3-by-3 squares, then all the 4-by-4 squares, and so on. Eventually one of two things would happen. If the set of tiles is solvable, one should eventually find a square, R, having the desired repeated color patterns. If the set is not solvable, one should find some size square for which no tiling is possible. In either case, the procedure will eventually terminate. However, it turns out that there exist solvable sets of tiles that cannot be used to form any square with such repeated color patterns. The first such set, constructed by Robert Berger of Harvard University in 1964, contained more than 20,000 types of tiles. It played a major role in his proof that the Tiling Problem is undecidable, completing a line of attack initiated some years earlier by the logician Hao Wang. Very recently Raphael Robinson of the University of California at Berkeley and Roger Penrose of the University of Oxford succeeded in reducing the number of needed tile varieties to only 24.

Of course, the existence of these sets of tiles alone is not enough to prove the undecidability of the Tiling Problem. They only show that the approach proposed above cannot give an algorithm. The main portion of Berger's proof shows that the computation of any computer program can be simulated exactly by an appropriately chosen set of tiles. In particular, to each computer program and each input to that program there corresponds a set of tiles that is solvable if and only if the program will eventually stop when given that input. This implies that any algorithm for solving the Tiling Problem could also be used to solve the Halting Problem. But because the Halting Problem has been proved undecidable, it follows that the Tiling Problem also is undecidable.



*An instance of the Traveling Salesman Problem is shown for 22 U.S. state capitals. A salesman begins at his home city, visits the other cities, and returns home, traveling the least total distance in the process. The route traced in red has a total length of 8,119 miles. The shortest route, however, has a total length of 8,117 miles. The reader is invited to find it.*

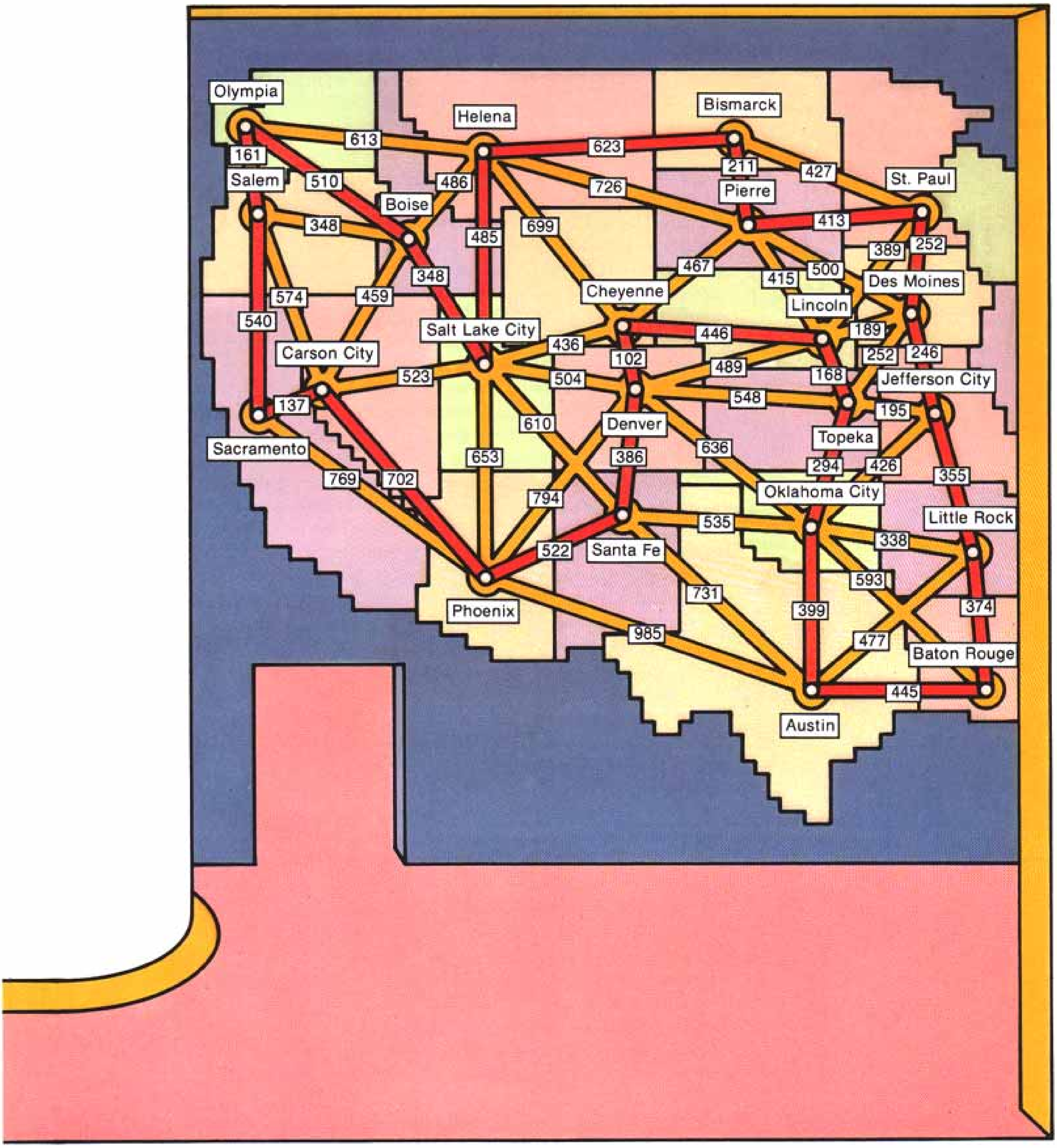
## Decidable problems that are hard

Undecidable problems like the Halting Problem and the Tiling Problem are beyond the capabilities of any computer because no algorithm for solving them can exist, even in principle. However, there is another sense in which a problem can be too hard. This second sense applies to those problems for which algorithms can be specified, the so-called decidable problems. A decidable problem can be hard because, even though algorithms exist for its solution, none of them can possibly operate in a reasonable amount of time. This is somewhat akin to the problem of automatic weather prediction; a computer program for predicting one day's weather from that of the preceding day would not be very useful if it required more than 24 hours to do it. Similarly, an algorithm for solving instances of a problem of current interest might not be useful if it required many years of computer time. Thus, it is not enough that an algorithm exists to solve a particular problem; it is important that a suitably fast algorithm exists.

Consideration of what is meant by "suitably fast" forms the main thrust of the theory of computational complexity; namely, analyzing the amount of time required by algorithms for solving particular problems. Since there often are many different ways to solve a given problem, such analyses are important for comparing different algorithms as well as for determining beforehand whether an algorithm is fast enough to be used in practice.

Of course there is not just a single length of time associated with an algorithm, but rather a whole collection of execution times, one for each instance of the problem. The time required by an algorithm is really a mathematical function, one that gives for each problem instance the amount of time needed by the algorithm to solve that instance. This function is usually quite complicated and not particularly amenable to mathematical analysis. For some purposes, it is more convenient to express computation time as a function of the "size" of an instance. The size of a problem instance is defined to be the number of symbols needed to describe that instance. For example, the size of a list of words to be alphabetized would be the sum of the letter counts of all the words on the list. The function that gives for each size the largest amount of time required by the algorithm to solve any problem instance of that size is called the time complexity function for the algorithm. It is usually possible to determine this function, or a close approximation to it, by a mathematical analysis of the algorithm.

An important distinction between algorithms can be based on the rate at which the values of their time complexity functions grow with problem instances of increasing size. This distinction essentially partitions algorithms into two classes, although all algorithms are not necessarily one type or the other. The first class includes those for which the time complexity function grows only at a moderate rate, whereas the second includes those for which this function grows explosively. In defining these classes precisely the variable *n* can be used to denote the size of a problem instance. The first class, called


Olympia
Helena
Bismarck
613
623
161
211
427
Salem
510
486
726
Pierre
St. Paul
Boise
348
485
699
413
389
252
348
467
415
500
Des Moines
574
459
Cheyenne
Lincoln
540
Salt Lake City
446
189
Carson City
436
102
252
246
523
504
489
168
Jefferson City
137
548
195
610
Denver
Topeka
Sacramento
653
386
636
294
426
769
702
355
794
Oklahoma City
535
Little Rock
522
Santa Fe
338
593
Phoenix
731
399
374
985
477
Baton Rouge
445
Austin

*In one example of the Bin Packing Problem, four trucks, each with a capacity of 10,000 pounds, can be used to deliver the loads depicted. In fact, with an additional 1,800 pounds, four trucks will still suffice. Again, the reader is invited to determine how such a task can be accomplished.*

polynomial algorithms, consists of those algorithms for which the time complexity function grows no faster than $n$ to some constant power, such as $n^2$, $n^5$, or $n^{10}$. Notice here that the size $n$ does not appear in the exponent. The second class, called exponential algorithms, consists of those for which the time complexity function grows as fast as some constant to the power $n$, such as $2^n$, $3^n$, or $7^n$. Here the size $n$ does appear in the exponent.

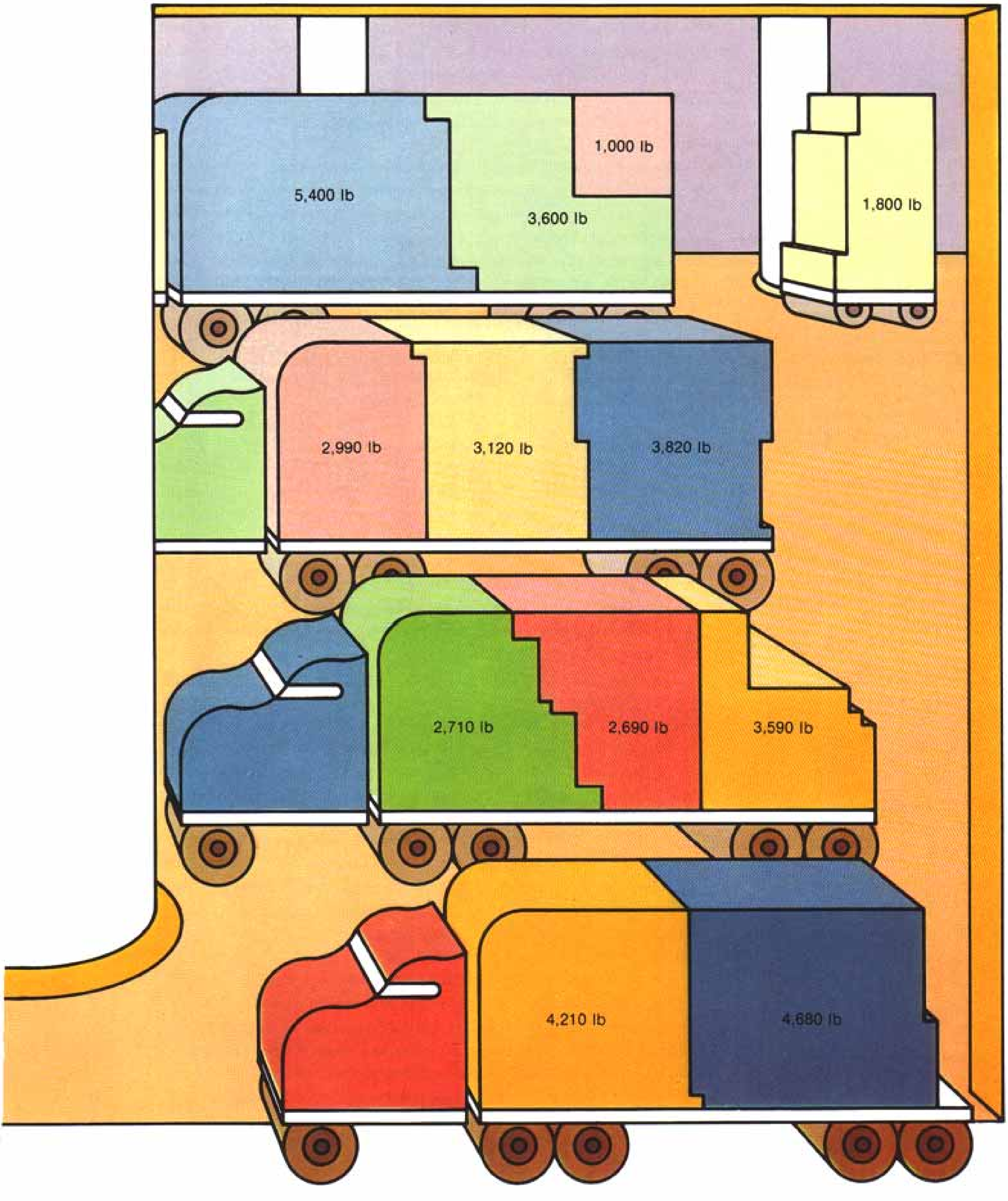
Perhaps the easiest way to see why this distinction is important is by examining some examples. The table on page 173 illustrates the computation time for various time complexity functions of each type and for various values of the size $n$ for a problem instance. The first three functions, $n^2$, $n^3$, and $n^5$, are time complexity functions for typical polynomial algorithms; the last two functions, $2^n$ and $3^n$, are time complexity functions for typical exponential algorithms. Each is to be interpreted as giving the time in microseconds required by that particular algorithm for solving an instance of size $n$.

The immense difference between the growth rates of computation time for the two types of algorithms is quite striking. All five algorithms require comparable amounts of time for instances of size 10, but as $n$ grows they diverge rapidly, and the polynomial algorithms become far superior. In general, the growth rate for exponential algorithms is so explosive that they are not reasonable to use for solving problem instances of even moderately large size. For these reasons it is important to find polynomial algorithms for solving problems. A decidable problem for which it is impossible to give a polynomial algorithm is said to be intractable, because even with future computers there is little hope for the solution of large-sized instances of such a problem.

What is known about the existence of intractable problems? It is only within the past few years that researchers in mathematics and computer science have succeeded in proving that any of the classical decidable problems are intractable. The first results of this type were obtained in 1972 by Albert Meyer and Larry Stockmeyer of the Massachusetts Institute of Technology. Subsequent work has lengthened the list of intractable problems.

One of the most easily described examples of an intractable problem deals with what is called Presburger Arithmetic. Basically, Presburger Arithmetic involves a simple, logical system for writing down statements about the integers 0, 1, 2, 3,. . . . The statements are formed using only the logical connectives (and, or, not), quantifiers ([for all $x$], [there is a $y$ such that]), and plus (+) and equals (=) signs. For example, [for all $x$] [for all $y$] $(x + y = y + x)$ is such a statement. Another example is [for all $x$] [there is a $y$ such that] $(x + x = y + y + 1)$. Notice that the first statement is true; it merely expresses the familiar commutative law of addition. However, the second statement is not true, because $x + x$ is always an even number while $y + y + 1$ is always an odd number. The decision problem for Presburger Arithmetic is to decide, given any such statement, whether or not the statement is true.

In 1930 the Polish logician M. Presburger showed that the decision


1,000 lb
5,400 lb
3,600 lb
1,800 lb
2,990 lb
3,120 lb
3,820 lb
2,710 lb
2,690 lb
3,590 lb
4,210 lb
4,680 lb

*The Minimum Network Problem has as its objective the connection of a given set of points with a network having the shortest possible total length. Shown at the top of the facing page are three connecting networks for the corners of a square. The left one, using no junction points, has a total length of 3. The center one, with one junction point (a), has a total length of 2 $\sqrt{2}$, or 2.828. . . . The right network, with two junction points (a and b), has a total length of [redacted]$\sqrt{3}$) +([redacted], or* 2,732... *[redacted], and is the shortest possible connecting network for this situation. The network shown in the lower figure connects 11 oil-consuming and oil-producing locations in the U.S. It employs five additional junction points and is the shortest possible connecting network for these locations. Without junction points the network would be 5% longer.*

problem for Presburger Arithmetic is decidable. He gave an explicit algorithm that will always eventually determine whether or not any given statement of the above type is true. This was a major contribution, for it was by no means obvious beforehand that such an algorithm would exist. Unfortunately Presburger did not give a polynomial algorithm and subsequent efforts to improve upon his method met with little success.

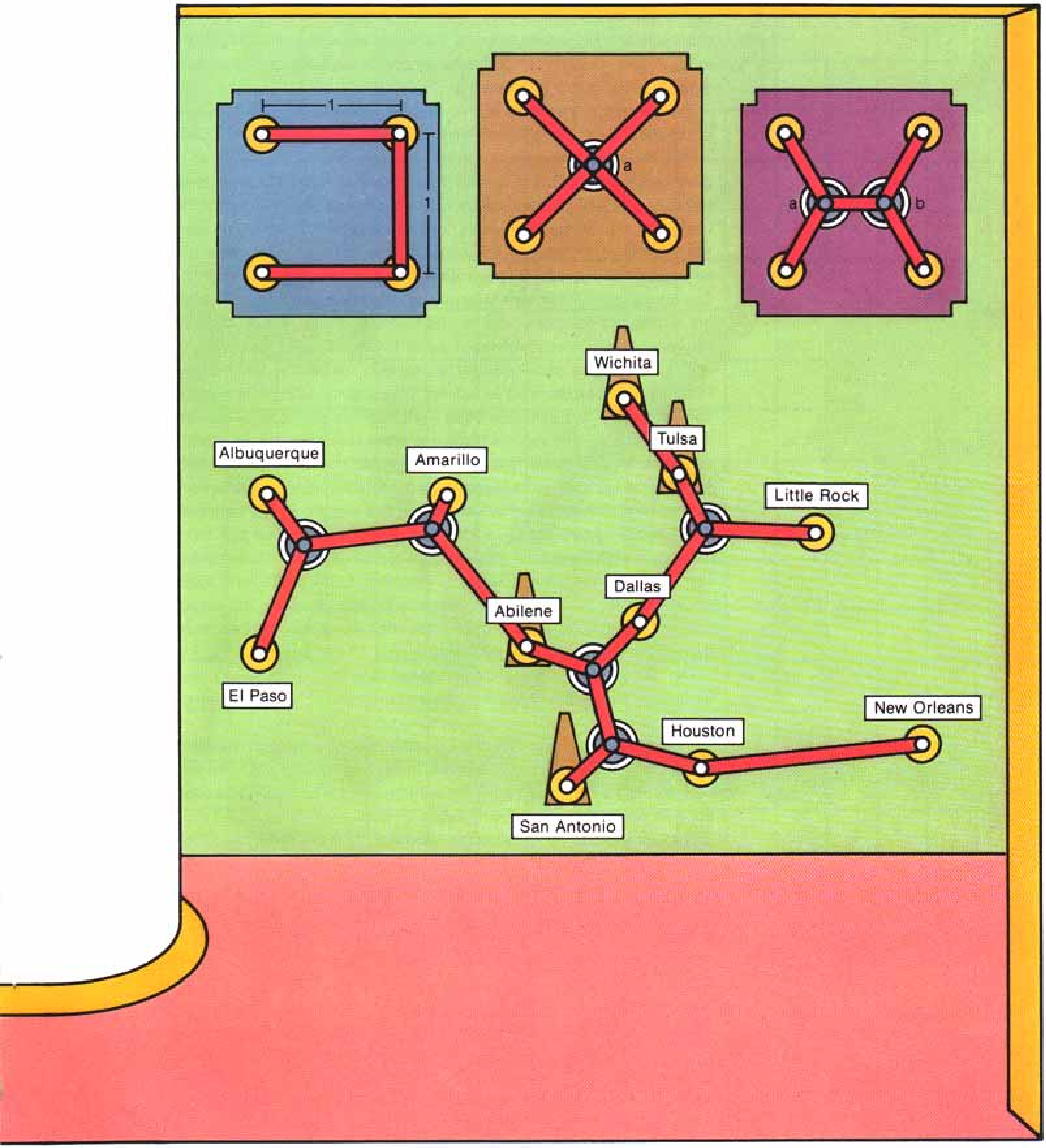
However, in 1974 Michael Fischer of MIT and Michael Rabin of the Hebrew University, Jerusalem, showed that the decision problem for Presburger Arithmetic is intractable. In fact, they showed something even stronger. They proved the existence of a number $c$ such that, for any algorithm that solves the decision problem for Presburger Arithmetic and for all sufficiently large numbers $n$, there are statements with no more than $n$ symbols that cause the algorithm to take more than $2^{2^{cn}}$ steps. This "superexponential" function grows much more rapidly than even the exponential functions discussed above. Such results certainly help to explain the lack of success in devising computer programs that would automatically prove theorems in mathematics.

## A perplexing class of problems

The subject of intractable problems remains one of the most active fields of research in contemporary mathematics and computer science. Many individual problems still have not been classified as to whether or not they are intractable; neither polynomial algorithms nor proofs of intractability are known for them. One particular collection of such problems has been singled out for special attention, both because of the practical importance of the problems in the collection and because of the intriguing way in which they are all related. These are known as NP-complete problems.

The class of NP-complete problems first arose during 1971 in the work of Stephen Cook of the University of Toronto, Ontario. While investigating the capabilities of hypothetical "nondeterministic" computers (the symbol NP in NP-complete stands for nondeterministic polynomial time), he succeeded in showing that several unclassified problems were equivalent in the sense that a polynomial algorithm for any one of them could be used to build a polynomial algorithm for each of the others. Thus either all of them are intractable or else none of them is. Subsequently, Richard Karp of the University of California at Berkeley and others showed that many additional problems, some of which had been studied for years in other contexts, were similarly equivalent to the problems of Cook. All such equivalent problems are called NP-complete problems.

This remarkable equivalence of so many different problems reduces a multitude of classification questions to one single question. Are all the NP-complete problems intractable or can they all be solved with polynomial algorithms? Little progress has been made in answering this perplexing question. However, based on many years of unsuccessful attempts to find polynomial algorithms for individual problems in


1
1
a
a
b
Wichita
Tulsa
Albuquerque
Amarillo
Little Rock
Dallas
Abilene
El Paso
New Orleans
Houston
San Antonio

this class, the intuition of most mathematicians is that all NP-complete problems are indeed intractable. Demonstration that a problem belongs to the class of NP-complete problems is widely accepted as a demonstration of its intractability.

Several typical examples can be used to illustrate the wide variety of problems currently known to be NP-complete. One example is known as the Traveling Salesman's Problem. In this problem a salesman is given a list of cities and a road map telling him the shortest route between each pair of cities. The salesman would like to begin at his home city, visit all the other cities, and return to his home city, traveling the least total distance in the process. The problem of finding this shortest route has recently been shown to be an NP-complete problem. The only algorithms known for solving it consider essentially all possible routes and compare them to find the best, a hopeless proposition for general instances involving even as few as 30 cities.

A different type of NP-complete problem, called the Bin Packing Problem, can be described as follows. One is given a list of items to be delivered—say, by truck—from one fixed location to another. Each item has a certain weight and the truck has some maximum total weight it can hold. The objective is to determine which items should be taken together on each trip so as to minimize the total number of trips needed for delivery of the items. For example, if the weights of the items are 4,500, 4,250, 3,500, 2,500, 2,250, and 2,000 pounds and the truck capacity is 10,000 pounds, then two trips would be sufficient. In the first trip the truck can carry the items weighing 4,500, 3,500, and 2,000 pounds and in the second trip it can carry the rest. However, if the truck can hold only 9,500 pounds then three trips would be required, even though the total weight of the items is only 19,000 pounds.

The Bin Packing Problem can appear in a variety of guises; *e.g.*, cutting up the minimum number of standard-length boards to produce pieces having prescribed lengths or scheduling a list of television commercials of various lengths into the smallest possible number of station breaks. Instances of this problem that involve many different items can be extremely difficult, and again no method short of considering essentially all possibilities is known always to work.

The last illustration is called the Minimum Network Problem. In this problem one is given a set of locations representing, for example, oil-producing sites in an oil field or branch locations of a large corporation. The object is to connect all the locations together with a network having the shortest possible total length (so that all the oil can reach a common refinery or shipping port, or so that all the corporate branch locations can communicate on a private-line telephone network). What makes this problem difficult is that junction points are allowed in the network if they can help decrease the total length. For example, if four oil wells are located at the four corners of a square one mile on a side, it is easy to envision a network of pipelines with a total length of three miles connecting them. This is not the shortest network, however. The shortest network requires two junction points and has a total length of

$(4/\sqrt{3}) + \frac{1}{2} = 2.808\ldots$ miles. The difficulty of determining in general precisely where to place these junctions for large sets of locations is what makes the Minimum Network Problem NP-complete.

Because of the considerable differences among the preceding problems, it is by no means obvious that they are actually equivalent; *i.e.*, that any polynomial algorithm to solve one of them (if it existed) could always be used to obtain polynomial algorithms for solving the others. In fact, the proofs that they are indeed equivalent are long and complicated. Techniques for proving such equivalences are constantly being developed and improved, and a substantial amount of theory is emerging that deals just with NP-complete problems. The examples presented are intended to indicate the great diversity of NP-complete problems and how remarkable it is that they can be shown to be computationally equivalent.

## Future directions

The field of computational complexity is still relatively young, and many questions remain to be answered. One of these has already been discussed, that of determining whether or not the NP-complete problems are all intractable. More generally there is a real need to assemble a collection of analytical tools for determining precisely the inherent complexity of problems. Efforts are proceeding in this direction, but progress is slow.

One direction in which substantial gains *are* being made is that of coping with the complexity of intractable problems that arise in practical applications. Many of these problems are optimization problems, which require one to find the best (cheapest, smallest, shortest, and so on) among all "feasible" solutions. The intractability of such problems often can be avoided by no longer requiring the best solution, but by merely asking for a good solution. For some intractable problems simple algorithms have been discovered that are guaranteed always to find feasible solutions coming within a fixed small percentage of the best solution. In these cases the tradeoffs between the time required to find a solution and the quality of that solution are important. Also under way is research into the possibility of devising simple algorithms capable of finding good solutions "on the average." It is anticipated that the next few years will see substantial progress on these fronts.