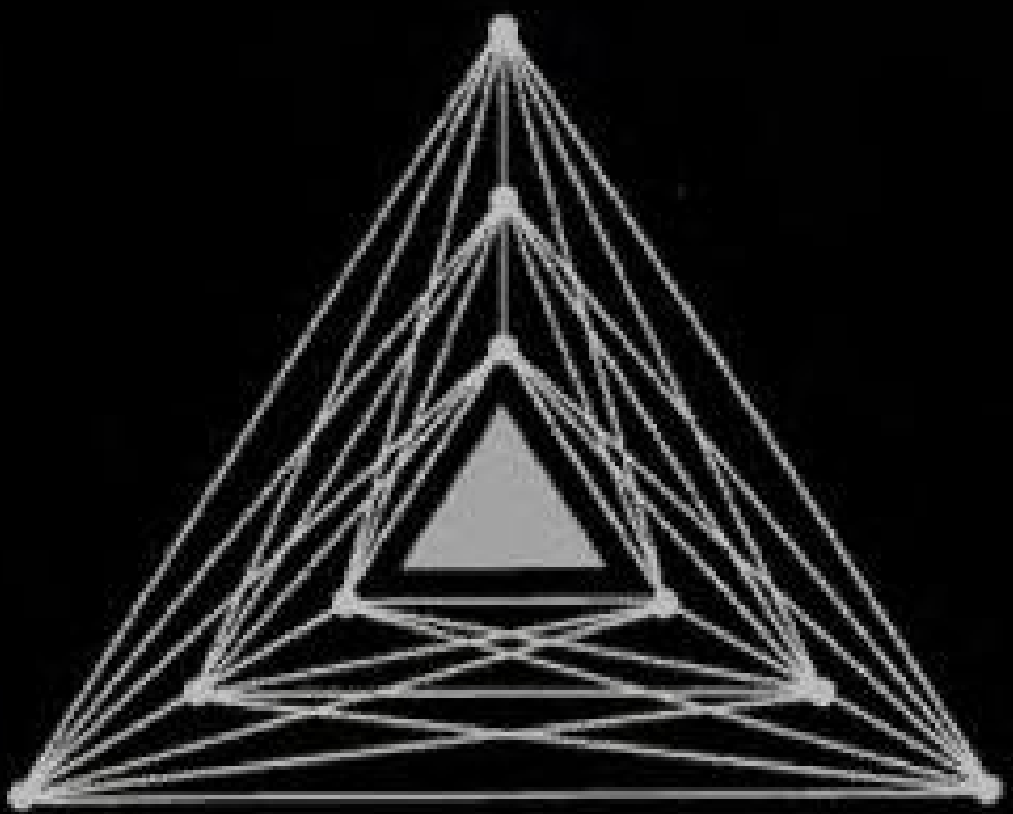
COMPUTERS AND INTRACTABILITY

A Guide to the Theory of NP-Completeness

Michael R. Garey / David S. Johnson



Library of Congress Cataloging in Publication Data

Garey, Michael R.
Computers and Intractability.

Bibliography: p.
Includes index.
1. Electronic digital computers--Programming.
2. Algorithms. 3. Computational complexity.
I. Johnson, David S., joint author. II. Title.
III. Title: NP-completeness.
QA76.6.G35 519.4 78-12361
ISBN 0-7167-1044-7
ISBN 0-7167-1045-5 pbk.

AMS Classification: Primary 68A20
Computer Science: Computational complexity and efficienc

Printed in the United States of America

6 7 8 9 MP 1 0 8 9 8 7 6 5

Contents

Preface ix

1 Computers, Complexity, and Intractability 1

1.1 Introduction 1
1.2 Problems, Algorithms, and Complexity 4
1.3 Polynomial Time Algorithms and Intractable Problems 6
1.4 Provably Intractable Problems 11
1.5 NP-Complete Problems 13
1.6 An Outline of the Book 14

2 The Theory of NP-Completeness 17

2.1 Decision Problems, Languages, and Encoding Schemes 18
2.2 Deterministic Turing Machines and the Class P 23
2.3 Nondeterministic Computation and the Class NP 27
2.4 The Relationship Between P and NP 32
2.5 Polynomial Transformations and NP-Completeness 34
2.6 Cook's Theorem 38

3 Proving NP-Completeness Results 45

3.1 Six Basic NP-Complete Problems 46
3.1.1 3-SATISFIABILITY 48
3.1.2 3-DIMENSIONAL MATCHING 50
3.1.3 VERTEX COVER and CLIQUE 53
3.1.4 HAMILTONIAN CIRCUIT 56
3.1.5 PARTITION 60
3.2 Some Techniques for Proving NP-Completeness 63
3.2.1 Restriction 63
3.2.2 Local Replacement 66
3.2.3 Component Design 72
3.3 Some Suggested Exercises 74

1

Computers, Complexity, and Intractability

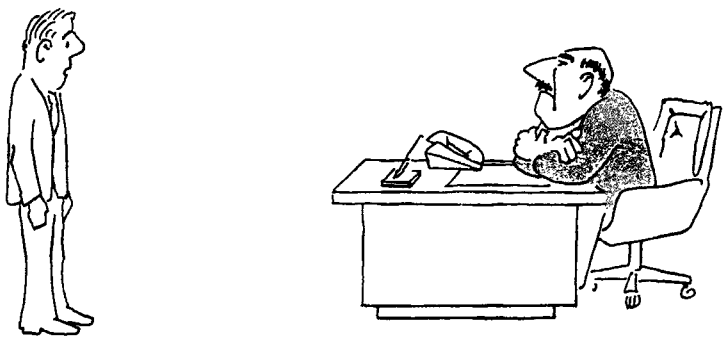
1.1 Introduction

The subject matter of this book is perhaps best introduced through the following, somewhat whimsical, example.

Suppose that you, like the authors, are employed in the halls of industry. One day your boss calls you into his office and confides that the company is about to enter the highly competitive "bandersnatch" market. For this reason, a good method is needed for determining whether or not any given set of specifications for a new bandersnatch component can be met and, if so, for constructing a design that meets them. Since you are the company's chief algorithm designer, your charge is to find an efficient algorithm for doing this.

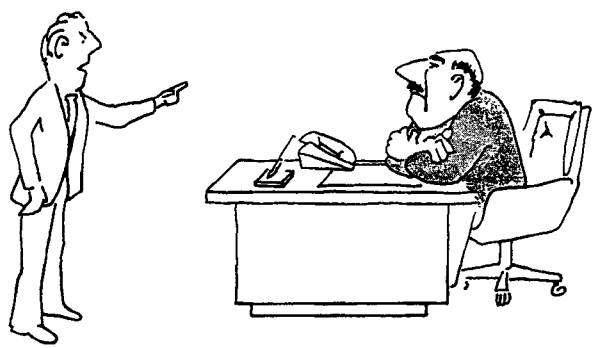
After consulting with the bandersnatch department to determine exactly what the problem is, you eagerly hurry back to your office, pull down your reference books, and plunge into the task with great enthusiasm. Some weeks later, your office filled with mountains of crumpled-up scratch paper, your enthusiasm has lessened considerably. So far you have not been able to come up with any algorithm substantially better than searching through all possible designs. This would not particularly endear you to your boss, since it would involve years of computation time for just one set of

specifications, and the bandersnatch department is already 13 components behind schedule. You certainly don't want to return to his office and report:



"I can't find an efficient algorithm, I guess I'm just too dumb."

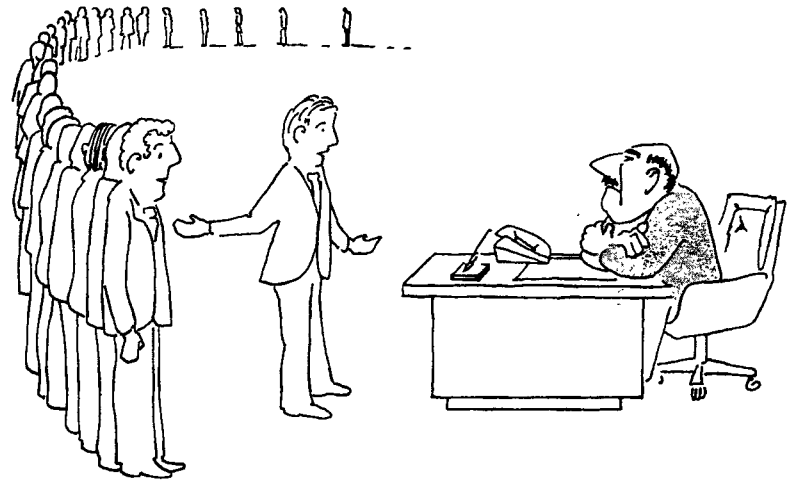
To avoid serious damage to your position within the company, it would be much better if you could prove that the bandersnatch problem is *inherently* intractable, that no algorithm could possibly solve it quickly. You then could stride confidently into the boss's office and proclaim:



"I can't find an efficient algorithm, because no such algorithm is possible!"

Unfortunately, proving inherent intractability can be just as hard as finding efficient algorithms. Even the best theoreticians have been stymied in their attempts to obtain such proofs for commonly encountered hard problems. However, having read this book, you have discovered something

almost as good. The theory of NP-completeness provides many straightforward techniques for proving that a given problem is "just as hard" as a large number of other problems that are widely recognized as being difficult and that have been confounding the experts for years. Armed with these techniques, you might be able to prove that the bandersnatch problem is NP-complete and, hence, that it is equivalent to all these other hard problems. Then you could march into your boss's office and announce:



"I can't find an efficient algorithm, but neither can all these famous people."

At the very least, this would inform your boss that it would do no good to fire you and hire another expert on algorithms.

Of course, our own bosses would frown upon our writing this book if its sole purpose was to protect the jobs of algorithm designers. Indeed, discovering that a problem is NP-complete is usually just the beginning of work on that problem. The needs of the bandersnatch department won't disappear overnight simply because their problem is known to be NP-complete. However, the knowledge that it is NP-complete does provide valuable information about what lines of approach have the potential of being most productive. Certainly the search for an efficient, exact algorithm should be accorded low priority. It is now more appropriate to concentrate on other, less ambitious, approaches. For example, you might look for efficient algorithms that solve various special cases of the general problem. You might look for algorithms that, though not guaranteed to run quickly, seem likely to do so most of the time. Or you might even relax the problem somewhat, looking for a fast algorithm that merely finds designs that

meet *most* of the component specifications. In short, the primary application of the theory of NP-completeness is to assist algorithm designers in directing their problem-solving efforts toward those approaches that have the greatest likelihood of leading to useful algorithms.

In the first chapter of this "guide" to NP-completeness, we introduce many of the underlying concepts, discuss their applicability (as well as give some cautions), and outline the remainder of the book.

1.2 Problems, Algorithms, and Complexity

In order to elaborate on what is meant by "inherently intractable" problems and problems having "equivalent" difficulty, it is important that we first agree on the meaning of several more basic terms.

Let us begin with the notion of a problem. For our purposes, a *problem* will be a general question to be answered, usually possessing several *parameters,* or free variables, whose values are left unspecified. A problem is described by giving: (1) a general description of all its parameters, and (2) a statement of what properties the answer, or *solution,* is required to satisfy. An *instance* of a problem is obtained by specifying particular values for all the problem parameters.

As an example, consider the classical "traveling salesman problem." The parameters of this problem consist of a finite set $C = \{c_1, c_2, \ldots, c_m\}$ of "cities" and, for each pair of cities c_i, c_j in C, the "distance" $d(c_i, c_j)$ between them. A solution is an ordering $\langle c_{\pi(1)}, c_{\pi(2)}, \ldots, c_{\pi(m)} \rangle$ of the given cities that minimizes

$$\left(\sum_{i=1}^{m-1} d(c_{\pi(i)}, c_{\pi(i+1)}) \right) + d(c_{\pi(m)}, c_{\pi(1)})$$

This expression gives the length of the "tour" that starts at $c_{\pi(1)}$, visits each city in sequence, and then returns directly to $c_{\pi(1)}$ from the last city $c_{\pi(m)}$.

One instance of the traveling salesman problem, illustrated in Figure 1.1, is given by $C = \{c_1, c_2, c_3, c_4\}$, $d(c_1, c_2) = 10$, $d(c_1, c_3) = 5$, $d(c_1, c_4) = 9$, $d(c_2, c_3) = 6$, $d(c_2, c_4) = 9$, and $d(c_3, c_4) = 3$. The ordering $\langle c_1, c_2, c_4, c_3 \rangle$ is a solution for this instance, as the corresponding tour has the minimum possible tour length of 27.

Algorithms are general, step-by-step procedures for solving problems. For concreteness, we can think of them simply as being computer programs, written in some precise computer language. An algorithm is said to *solve* a problem Π if that algorithm can be applied to any instance I of Π and is guaranteed always to produce a solution for that instance I. We emphasize that the term "solution" is intended here strictly in the sense introduced above, so that, in particular, an algorithm does not "solve" the traveling

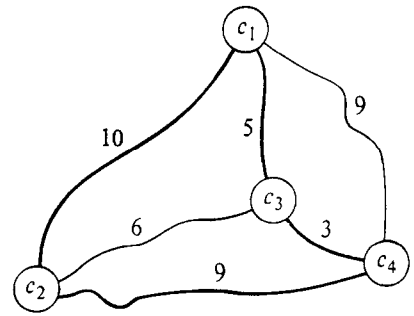


Figure 1.1 An instance of the traveling salesman problem and a tour of length 27, which is the minimum possible in this case.

salesman problem unless it always constructs an ordering that gives a minimum length tour.

In general, we are interested in finding the most "efficient" algorithm for solving a problem. In its broadest sense, the notion of efficiency involves all the various computing resources needed for executing an algorithm. However, by the "most efficient" algorithm one normally means the fastest. Since time requirements are often a dominant factor determining whether or not a particular algorithm is efficient enough to be useful in practice, we shall concentrate primarily on this single resource.

The time requirements of an algorithm are conveniently expressed in terms of a single variable, the "size" of a problem instance, which is intended to reflect the amount of input data needed to describe the instance. This is convenient because we would expect the relative difficulty of problem instances to vary roughly with their size. Often the size of a problem instance is measured in an informal way. For the traveling salesman problem, for example, the number of cities is commonly used for this purpose. However, an m-city problem instance includes, in addition to the labels of the m cities, a collection of $m(m-1)/2$ numbers defining the inter-city distances, and the sizes of these numbers also contribute to the amount of input data. If we are to deal with time requirements in a precise, mathematical manner, we must take care to define instance size in such a way that all these factors are taken into account.

To do this, observe that the description of a problem instance that we provide as input to the computer can be viewed as a single finite string of symbols chosen from a finite input alphabet. Although there are many different ways in which instances of a given problem might be described, let us assume that one particular way has been chosen in advance and that each problem has associated with it a fixed *encoding scheme*, which maps problem

instances into the strings describing them. The *input length* for an instance I of a problem Π is defined to be the number of symbols in the description of I obtained from the encoding scheme for Π. It is this number, the input length, that is used as the formal measure of instance size.

For example, instances of the traveling salesman problem might be described using the alphabet $\{c, [,], /, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, with our previous example of a problem instance being encoded by the string "$c[1]c[2]c[3]c[4]//10/5/9//6/9//3$." More complicated instances would be encoded in analogous fashion. If this were the encoding scheme associated with the traveling salesman problem, then the input length for our example would be 32.

The *time complexity function* for an algorithm expresses its time requirements by giving, for each possible input length, the largest amount of time needed by the algorithm to solve a problem instance of that size. Of course, this function is not well-defined until one fixes the encoding scheme to be used for determining input length and the computer or computer model to be used for determining execution time. However, as we shall see, the particular choices made for these will have little effect on the broad distinctions made in the theory of NP-completeness. Hence, in what follows, the reader is advised merely to fix in mind a particular encoding scheme for each problem and a particular computer or computer model, and to think in terms of time complexity as determined from the corresponding input lengths and execution times.

1.3 Polynomial Time Algorithms and Intractable Problems

Different algorithms possess a wide variety of different time complexity functions, and the characterization of which of these are "efficient enough" and which are "too inefficient" will always depend on the situation at hand. However, computer scientists recognize a simple distinction that offers considerable insight into these matters. This is the distinction between polynomial time algorithms and exponential time algorithms.

Let us say that a function $f(n)$ is $O(g(n))$ whenever there exists a constant c such that $|f(n)| \leqslant c \cdot |g(n)|$ for all values of $n \geqslant 0$. A *polynomial time algorithm* is defined to be one whose time complexity function is $O(p(n))$ for some polynomial function p, where n is used to denote the input length. Any algorithm whose time complexity function cannot be so bounded is called an *exponential time algorithm* (although it should be noted that this definition includes certain non-polynomial time complexity functions, like $n^{\log n}$, which are not normally regarded as exponential functions).

The distinction between these two types of algorithms has particular significance when considering the solution of large problem instances. Figure 1.2 illustrates the differences in growth rates among several typical complexity functions of each type, where the functions express execution time

in terms of microseconds. Notice the much more explosive growth rates for the two exponential complexity functions.

Time complexity function	Size n					
	10	20	30	40	50	60
n	.00001 second	.00002 second	.00003 second	.00004 second	.00005 second	.00006 second
n^2	.0001 second	.0004 second	.0009 second	.0016 second	.0025 second	.0036 second
n^3	.001 second	.008 second	.027 second	.064 second	.125 second	.216 second
n^5	.1 second	3.2 seconds	24.3 seconds	1.7 minutes	5.2 minutes	13.0 minutes
2^n	.001 second	1.0 second	17.9 minutes	12.7 days	35.7 years	366 centuries
3^n	.059 second	58 minutes	6.5 years	3855 centuries	2×10^8 centuries	1.3×10^{13} centuries

Figure 1.2 Comparison of several polynomial and exponential time complexity functions.

Even more revealing is an examination of the effects of improved computer technology on algorithms having these time complexity functions. Figure 1.3 shows how the largest problem instance solvable in one hour would change if we had a computer 100 or 1000 times faster than our present machine. Observe that with the 2^n algorithm a thousand-fold increase in computing speed only adds 10 to the size of the largest problem instance we can solve in an hour, whereas with the n^5 algorithm this size almost quadruples.

These tables indicate some of the reasons why polynomial time algorithms are generally regarded as being much more desirable than exponential time algorithms. This view, and the distinction between the two types of algorithms, is central to our notion of inherent intractability and to the theory of NP-completeness.

The fundamental nature of this distinction was first discussed in [Cobham, 1964] and [Edmonds, 1965a]. Edmonds, in particular, equated poly-

Size of Largest Problem Instance Solvable in 1 Hour

Time complexity function	With present computer	With computer 100 times faster	With computer 1000 times faster
n	N_1	$100\ N_1$	$1000\ N_1$
n^2	N_2	$10\ N_2$	$31.6\ N_2$
n^3	N_3	$4.64\ N_3$	$10\ N_3$
n^5	N_4	$2.5\ N_4$	$3.98\ N_4$
2^n	N_5	$N_5+6.64$	$N_5+9.97$
3^n	N_6	$N_6+4.19$	$N_6+6.29$

Figure 1.3 Effect of improved technology on several polynomial and exponential time algorithms.

nomial time algorithms with "good" algorithms and conjectured that certain integer programming problems might not be solvable by such "good" algorithms. This reflects the viewpoint that exponential time algorithms should not be considered "good" algorithms, and indeed this usually is the case. Most exponential time algorithms are merely variations on exhaustive search, whereas polynomial time algorithms generally are made possible only through the gain of some deeper insight into the structure of a problem. There is wide agreement that a problem has not been "well-solved" until a polynomial time algorithm is known for it. Hence, we shall refer to a problem as *intractable* if it is so hard that no polynomial time algorithm can possibly solve it.

Of course, this formal use of "intractable" should be viewed only as a rough approximation to its dictionary meaning. The distinction between "efficient" polynomial time algorithms and "inefficient" exponential time algorithms admits of many exceptions when the problem instances of interest have limited size. Even in Figure 1.2, the 2^n algorithm is faster than the n^5 algorithm for $n \leqslant 20$. More extreme examples can be constructed easily.

Furthermore, there are some exponential time algorithms that have been quite useful in practice. Time complexity as defined is a *worst-case* measure, and the fact that an algorithm has time complexity 2^n means only that at least one problem instance of size n requires that much time. Most problem instances might actually require far less time than that, a situation

that appears to hold for several well-known algorithms. The simplex algorithm for linear programming has been shown to have exponential time complexity [Klee and Minty, 1972], [Zadeh, 1973], but it has an impressive record of running quickly in practice. Likewise, branch-and-bound algorithms for the knapsack problem have been so successful that many consider it to be a "well-solved" problem, even though these algorithms, too, have exponential time complexity.

Unfortunately, examples like these are quite rare. Although exponential time algorithms are known for many problems, few of them are regarded as being very useful in practice. Even the successful exponential time algorithms mentioned above have not stopped researchers from continuing to search for polynomial time algorithms for solving those problems. In fact, the very success of these algorithms has led to the suspicion that they somehow capture a crucial property of the problems whose refinement could lead to still better methods. So far, little progress has been made toward explaining this success, and no methods are known for predicting in advance that a given exponential time algorithm will run quickly in practice.

On the other hand, the much more stringent bounds on execution time satisfied by polynomial time algorithms often permit such predictions to be made. Even though an algorithm having time complexity n^{100} or $10^{99}n^2$ might not be considered likely to run quickly in practice, the polynomially solvable problems that arise naturally tend to be solvable within polynomial time bounds that have degree 2 or 3 at worst and that do not involve extremely large coefficients. Algorithms satisfying such bounds *can* be considered to be "provably efficient," and it is this much-desired property that makes polynomial time algorithms the preferred way to solve problems.

Our definition of "intractable" also provides a theoretical framework of considerable generality and power. The intractability of a problem turns out to be essentially independent of the particular encoding scheme and computer model used for determining time complexity.

Let us first consider encoding schemes. Suppose for example that we are dealing with a problem in which each instance is a graph $G = (V,E)$, where V is the set of vertices and E is the set of edges, each edge being an unordered pair of vertices. Such an instance might be described (see Figure 1.4) by simply listing all the vertices and edges, or by listing the rows of the adjacency matrix for the graph, or by listing for each vertex all the other vertices sharing a common edge with it (a "neighbor" list). Each of these encodings can give a different input length for the same graph. However, it is easy to verify (see Figure 1.5) that the input lengths they determine differ at most polynomially from one another, so that any algorithm having polynomial time complexity under one of these encoding schemes also will have polynomial time complexity under all the others. In fact, the standard encoding schemes used in practice for any particular problem always seem to differ at most polynomially from one another. It would be difficult to imagine a "reasonable" encoding scheme for a problem that differs more

than polynomially from the standard ones. Although what we mean here by "reasonable" cannot be formalized, the following two conditions capture much of the notion:

(1) the encoding of an instance I should be concise and not "padded" with unnecessary information or symbols, and

(2) numbers occurring in I should be represented in binary (or decimal, or octal, or in any fixed base other than 1).

If we restrict ourselves to encoding schemes satisfying these conditions, then the particular encoding scheme used should not affect the determination of whether a given problem is intractable.

Encoding Scheme	String	Length
Vertex list, Edge list	V[1]V[2]V[3]V[4](V[1]V[2])(V[2]V[3])	36
Neighbor list	(V[2])(V[1]V[3])(V[2])( )	24
Adjacency matrix rows	0100/1010/0010/0000	19

Figure 1.4 Descriptions of the graph $G = (V,E)$ where $V = \{V_1, V_2, V_3, V_4\}$ and $E = \{\{V_1, V_2\}, \{V_2, V_3\}\}$, under three different encoding schemes.

Encoding Scheme	Lower Bound	Upper Bound
Vertex list, Edge list	$4v + 10e$	$4v + 10e + (v+2e)\cdot\lceil\log_{10} v\rceil$
Neighbor list	$2v + 8e$	$2v + 8e + 2e\cdot\lceil\log_{10} v\rceil$
Adjacency matrix	$v^2 + v - 1$	$v^2 + v - 1$

Figure 1.5 General bounds on input lengths for the three encoding schemes of Figure 1.4 for graphs $G = (V,E)$ with $|V| = v$, $|E| = e$. Since $e < v^2$, these show that the input lengths differ at most polynomially from each other. ($\lceil x\rceil$ denotes the least integer not less than x.)

Similar comments can be made concerning the choice of computer models. All the realistic models of computers studied so far, such as one-tape Turing machines, multi-tape Turing machines, and random-access machines (RAMs), are equivalent with respect to polynomial time complexity (for example, see Figure 1.6). One would expect any other "reasonable" model to share in this equivalence. The notion of "reasonable" intended here is essentially that there is a polynomial bound on the amount of work that can be done in a single unit of time. Thus, for example, a model having the capability of performing arbitrarily many operations in parallel would not be considered "reasonable," and indeed no existing (or planned) computer has this capability. At any rate, so long as we restrict ourselves to the standard models of realistic computers, the class of intractable problems will be unaffected by the particular model used, and we can make our choice on the basis of convenience without sacrificing the applicability of our results.

Simulated machine B	Simulating machine A		
	1TM	kTM	RAM
1-Tape Turing Machine (1TM)	—	$O(T(n))$	$O(T(n)\log T(n))$
k-Tape Turing Machine (kTM)	$O(T^2(n))$	—	$O(T(n)\log T(n))$
Random Access Machine (RAM)	$O(T^3(n))$	$O(T^2(n))$	—

Figure 1.6 Time required by machine A to simulate the execution of an algorithm of time complexity $T(n)$ on Machine B (for example, see [Hopcroft and Ullman, 1969] and [Aho, Hopcroft, and Ullman, 1974]).

1.4 Provably Intractable Problems

Now that we have discussed the formal meaning of "intractable problem," it is appropriate that we briefly survey the current state of knowledge about the existence of intractable problems.

It is useful to begin by distinguishing between two different causes of intractability allowed by our definition. The first, which is the one we usually have in mind, is that the problem is so difficult that an exponential amount of time is needed to discover a solution. The second is that the solution *itself* is required to be so extensive that it cannot be described with an expression having length bounded by a polynomial function of the input length.

This second cause occurs, for example, in the variant of the traveling salesman problem that includes a number B as an additional parameter and that asks for *all* tours having total length B or less. It is easy to construct instances of this problem in which exponentially many tours are shorter than the given bound, so that no polynomial time algorithm could possibly list them all.

Intractability of this sort is by no means insignificant, and it is important to recognize it when it occurs. However, in most cases its existence is

apparent from the problem definition. In fact, this type of intractability can be regarded as a signal that the problem is not defined realistically, because we are asking for more information than we could ever hope to use. Thus, from now on we shall restrict our attention to the first type of intractability. Accordingly, only problems for which the solution length is bounded by a polynomial function of the input length will be considered.

The earliest intractability results for such problems are the classical undecidability results of Alan Turing. Over 40 years ago, Turing demonstrated that certain problems are so hard that they are "undecidable," in the sense that no algorithm at all can be given for solving them. He proved, for example, that it is impossible to specify *any* algorithm which, given an arbitrary computer program and an arbitrary input to that program, can decide whether or not the program will eventually halt when applied to that input [Turing, 1936]. A variety of other problems are now known to be undecidable, including the triviality problem for finitely presented groups [Rabin, 1958], Hilbert's tenth problem (solvability of polynomial equations in integers) [Matijasevic, 1970], and several problems of "tiling the plane" [Berger, 1966]. Since these undecidable problems cannot be solved by *any* algorithm, much less a polynomial time algorithm, they indeed are intractable in an especially strong sense.

The first examples of intractable "decidable" problems were obtained in the early 1960's, as part of work on complexity "hierarchies" by Hartmanis and Stearns [1965]. However, these results involved only "artificial" problems, specifically constructed to have the appropriate properties. It was not until the early 1970's that Meyer and Stockmeyer [1972], Fischer and Rabin [1974], and others finally succeeded in proving some "natural" decidable problems to be intractable. These include a variety of previously studied problems from automata theory, formal language theory, and mathematical logic. In fact, the proofs show that these problems cannot be solved in polynomial time using even a "nondeterministic" computer model, which has the ability to pursue an unbounded number of independent computational sequences in parallel. We shall see that this "unreasonable" computer model plays an important role in the theory of NP-completeness, and its capabilities will be specified more fully in Chapter 2.

All the provably intractable problems known to date fall into the two categories we have just mentioned. They are either undecidable or "nondeterministically" intractable. However, most of the apparently intractable problems encountered in practice *are* decidable and *can* be solved in polynomial time with the aid of a nondeterministic computer. Thus, none of the proof techniques developed so far is powerful enough to verify the apparent intractability of these problems.

1.5 NP-Complete Problems

As theoreticians continue to seek more powerful methods for proving problems intractable, parallel efforts focus on learning more about the ways in which various problems are interrelated with respect to their difficulty. As we suggested earlier, the discovery of such relationships between problems often can provide information useful to algorithm designers.

The principal technique used for demonstrating that two problems are related is that of "reducing" one to the other, by giving a constructive transformation that maps any instance of the first problem into an equivalent instance of the second. Such a transformation provides the means for converting any algorithm that solves the second problem into a corresponding algorithm for solving the first problem.

Many simple examples of such reductions have been known for some time. For example, Dantzig [1960] reduced a number of combinatorial optimization problems to the general zero-one integer linear programming problem. Edmonds [1962] reduced the graph theoretic problems of "covering all edges with a minimum number of vertices" and "finding a maximum independent set of vertices" to the general "set covering problem." Gimpel [1965] reduced the general set covering problem to the "prime implicant covering problem" of logic design. Dantzig, Blattner, and Rao [1966] described a "well-known" reduction from the traveling salesman problem to the "shortest path problem" with negative edge lengths allowed.

These early reductions, although rather isolated and limited in scope, foreshadow the kind of results proved in the theory of NP-completeness.

The foundations for the theory of NP-completeness were laid in a paper of Stephen Cook, presented in 1971, entitled "The Complexity of Theorem Proving Procedures" [Cook, 1971a]. In this brief but elegant paper Cook did several important things.

First, he emphasized the significance of "polynomial time reducibility," that is, reductions for which the required transformation can be executed by a polynomial time algorithm. If we have a polynomial time reduction from one problem to another, this ensures that any polynomial time algorithm for the second problem can be converted into a corresponding polynomial time algorithm for the first problem.

Second, he focused attention on the class NP of decision problems that can be solved in polynomial time by a nondeterministic computer. (A decision problem is one whose solution is either "yes" or "no".) Most of the apparently intractable problems encountered in practice, when phrased as decision problems, belong to this class.

Third, he proved that one particular problem in NP, called the "satisfiability" problem, has the property that every other problem in NP can be polynomially reduced to it. If the satisfiability problem can be solved with a polynomial time algorithm, then so can every problem in NP, and if any problem in NP is intractable, then the satisfiability problem also must

be intractable. Thus, in a sense, the satisfiability problem is the "hardest" problem in NP.

Finally, Cook suggested that other problems in NP might share with the satisfiability problem this property of being the "hardest" member of NP. He showed this to be the case for the problem "Does a given graph G contain a complete subgraph on a given number k of vertices?"

Subsequently, Richard Karp presented a collection of results [Karp, 1972] proving that indeed the decision problem versions of many well known combinatorial problems, including the traveling salesman problem, are just as "hard" as the satisfiability problem. Since then a wide variety of other problems have been proved equivalent in difficulty to these problems, and this equivalence class, consisting of the "hardest" problems in NP, has been given a name: the class of *NP-complete problems.*

Cook's original ideas have turned out to be remarkably powerful. They have provided the means for combining many individual complexity questions into the single question: Are the NP-complete problems intractable? The lists included in the Appendix of this book contain literally hundreds of different problems now known to be NP-complete. As more and more problems of independent interest are shown to belong to this equivalence class, its importance is continually reinforced.

The question of whether or not the NP-complete problems are intractable is now considered to be one of the foremost open questions of contemporary mathematics and computer science. Despite the willingness of most researchers to conjecture that the NP-complete problems are all intractable, little progress has yet been made toward establishing either a proof or a disproof of this far-reaching conjecture. However, even without a proof that NP-completeness implies intractability, the knowledge that a problem is NP-complete suggests, at the very least, that a major breakthrough will be needed to solve it with a polynomial time algorithm.

1.6 An Outline of the Book

Although this book is intended mainly as a primer on how to determine whether or not any particular problem is NP-complete (either by looking it up in the lists we present or by proving it yourself), we shall also discuss some of the options available for dealing with a problem that is known to be NP-complete. A brief outline of subsequent chapters follows.

In Chapter 2, we present the formal underpinnings of NP-completeness and prove Cook's theorem. The central definitions involve certain theoretical concepts, such as "languages" and "Turing machines," which we develop in a straightforward manner, relating them to the notions of problems and computer models already discussed. This chapter should give the reader a good understanding of the technical meaning of NP-completeness.

Chapter 3 is devoted to methods for proving a problem NP-complete. A number of examples are presented to illustrate the usual structure of such proofs, and to indicate how one goes about generating one. In essence, one proves a new problem to be NP-complete by polynomially reducing a known NP-complete problem to it. We survey the known NP-complete problems that have been most useful for this purpose and demonstrate their use.

In Chapter 4, we examine the ways in which the theory of NP-completeness can be used for conducting a detailed analysis of the complexity of a problem, seeking to determine the "boundary" between those cases of the problem that are polynomially solvable and those that are NP-complete.

In Chapter 5, we show how the techniques used for proving NP-completeness can be generalized so that problems other than just decision problems can be proved to be "as hard as" the NP-complete problems. As an aid to reading the published literature on the theory of NP-completeness, we also provide a brief historical survey of the development of the main ideas and the varying terminology that has been used for discussing them.

In Chapter 6, we discuss several approaches for dealing with intractable problems, especially that of finding near-optimal solutions using fast algorithms. Examples of the successes and failures of each approach are described, and we illustrate how the theory of NP-completeness can be applied even here.

Chapter 7 is intended to acquaint the reader with some of the theoretical issues and ideas that have arisen in parallel with the theory of NP-completeness. Among other topics we discuss the polynomial hierarchy, #P-completeness, polynomial space completeness, and the "relativization" of the question of the intractability of the NP-complete problems.

The last third of the book consists of the Appendix, an extensive and annotated list of problems known to be NP-complete or harder. The list is divided into sections, each devoted to problems from a particular subject area, such as graph theory, scheduling, algebra and number theory, covering and partitioning, mathematical programming, program optimization, automata and language theory, and, of course, miscellaneous topics. The list includes references to related problems known to be solvable in polynomial time and to problems whose status remains open in that neither polynomial time algorithms nor NP-completeness proofs are known for them.

2

The Theory of NP-Completeness

In this chapter we present the formal details of the theory of NP-completeness. So that the theory can be defined in a mathematically rigorous way, it will be necessary to introduce formal counterparts for many of our informal notions, such as "problems" and "algorithms." Indeed, one of the main goals of this chapter is to make explicit the connection between the formal terminology and the more intuitive, informal shorthand that is commonly used in its place. Once we have this connection well in hand, it will be possible for us to pursue our discussions primarily at the informal level in later chapters, reverting to the formal level only when necessary for clarity and rigor.

The chapter begins by discussing decision problems and their representation as "languages," equating "solving" a decision problem with "recognizing" the corresponding language. The one-tape Turing machine is introduced as our basic model for computation and is used to define the class P of all languages recognizable deterministically in polynomial time. This model is then augmented with a hypothetical "guessing" ability, and the augmented model is used to define the class NP of all languages recognizable "nondeterministically" in polynomial time. After discussing the relationship between P and NP, we define the notion of a polynomial transformation from one language to another and use it to define what will be our

most important class, the class of NP-complete problems. The chapter concludes with the statement and proof of Cook's fundamental theorem, which provides us with our first bona fide NP-complete problem.

2.1 Decision Problems, Languages, and Encoding Schemes

As a matter of convenience, the theory of NP-completeness is designed to be applied only to *decision problems*. Such problems, as mentioned in Chapter 1, have only two possible solutions, either the answer "yes" or the answer "no." Abstractly, a decision problem Π consists simply of a set D_Π of *instances* and a subset $Y_\Pi \subseteq D_\Pi$ of *yes-instances*. However, most decision problems of interest possess a considerable amount of additional structure, and we will describe them in a way that emphasizes this structure. The standard format we will use for specifying problems consists of two parts, the first part specifying a *generic instance* of the problem in terms of various components, which are sets, graphs, functions, numbers, etc., and the second part stating a yes-no *question* asked in terms of the generic instance. The way in which this specifies D_Π and Y_Π should be apparent. An instance belongs to D_Π if and only if it can be obtained from the generic instance by substituting particular objects of the specified types for all the generic components, and the instance belongs to Y_Π if and only if the answer for the stated question, when particularized to that instance, is "yes."

For example, the following describes a well-known decision problem from graph theory:

SUBGRAPH ISOMORPHISM
INSTANCE: Two graphs, $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$.
QUESTION: Does G_1 contain a subgraph isomorphic to G_2, that is, a subset $V' \subseteq V_1$ and a subset $E' \subseteq E_1$ such that $|V'| = |V_2|$, $|E'| = |E_2|$, and there exists a one-to-one function $f: V_2 \rightarrow V'$ satisfying $\{u, v\} \in E_2$ if and only if $\{f(u), f(v)\} \in E'$?

A decision problem related to the traveling salesman problem can be described as follows:

TRAVELING SALESMAN
INSTANCE: A finite set $C = \{c_1, c_2, \ldots, c_m\}$ of "cities," a "distance" $d(c_i, c_j) \in Z^+$ for each pair of cities $c_i, c_j \in C$, and a bound $B \in Z^+$ (where Z^+ denotes the positive integers).
QUESTION: Is there a "tour" of all the cities in C having total length no more than B, that is, an ordering $\langle c_{\pi(1)}, c_{\pi(2)}, \ldots, c_{\pi(m)} \rangle$ of C such that

$$\left(\sum_{i=1}^{m-1} d(c_{\pi(i)}, c_{\pi(i+1)}) \right) + d(c_{\pi(m)}, c_{\pi(1)}) \leq B \quad ?$$

The reader will find many more examples of the use of this format throughout the book, but these two should suffice for now to convey the basic idea. The second example also serves to illustrate an important point about how a decision problem can be derived from an optimization problem. If the optimization problem asks for a structure of a certain type that has minimum "cost" among all such structures (for example, a tour that has minimum length among all tours), we can associate with that problem the decision problem that includes a numerical bound B as an additional parameter and that asks whether there exists a structure of the required type having cost *no more than* B (for example, a tour of length no more than B). Decision problems can be derived from maximization problems in an analogous way, simply by replacing "no more than" by "at least."

The key point to observe about this correspondence is that, so long as the cost function is relatively easy to evaluate, the decision problem can be no harder than the corresponding optimization problem. Clearly, if we could find a minimum length tour for the traveling salesman problem in polynomial time, then we could also solve the associated decision problem in polynomial time. All we need do is find the minimum length tour, compute its length, and compare that length to the given bound B. Thus, if we could demonstrate that TRAVELING SALESMAN is NP-complete (as indeed it is), we would know that the traveling salesman optimization problem is at least as hard. In this way, even though the theory of NP-completeness restricts attention to only decision problems, we can extend the implications of the theory to optimization problems as well. (We shall see in Chapter 5 that decision problems and optimization problems are often even more closely tied: Many decision problems, including TRAVELING SALESMAN, can also be shown to be "no easier" than their corresponding optimization problems.)

The reason for the restriction to decision problems is that they have a very natural, formal counterpart, which is a suitable object to study in a mathematically precise theory of computation. This counterpart is called a "language" and is defined in the following way.

For any finite set Σ of symbols, we denote by Σ^* the set of all finite strings of symbols from Σ. For example, if $\Sigma = \{0,1\}$, then Σ^* consists of the empty string "ϵ," the strings 0,1,00,01,10,11,000,001, and all other finite strings of 0's and 1's. If L is a subset of Σ^*, we say that L is a *language* over the alphabet Σ. Thus {01,001,111,1101010} is a language over {0,1}, as is the set of all binary representations of integers that are perfect squares, as is the set $\{0,1\}^*$ itself.

The correspondence between decision problems and languages is brought about by the encoding schemes we use for specifying problem instances whenever we intend to compute with them. Recall that an encoding scheme e for a problem Π provides a way of describing each instance of Π by an appropriate string of symbols over some fixed alphabet Σ. Thus the problem Π and the encoding scheme e for Π partition Σ^* into three classes

of strings: those that are not encodings of instances of Π, those that encode instances of Π for which the answer is "no," and those that encode instances of Π for which the answer is "yes." This third class of strings is the language we associate with Π and e, setting

$$L[\Pi,e] = \left\{ x \in \Sigma^* : \begin{array}{l} \Sigma \textit{ is the alphabet used by } e, \textit{ and } x \textit{ is the} \\ \textit{encoding under } e \textit{ of an instance } I \in Y_\Pi \end{array} \right\}$$

Our formal theory is applied to decision problems by saying that, if a result holds for the language $L[\Pi,e]$, then it holds for the problem Π under the encoding scheme e.

In fact, we shall usually follow standard practice and be a bit more informal than this. Each time we introduce a new concept in terms of languages, we will observe that the property is essentially encoding independent, so long as we restrict ourselves to "reasonable" encoding schemes. That is, if e and e' are any two reasonable encoding schemes for Π, then the property holds either for both $L[\Pi,e]$ and $L[\Pi,e']$ or for neither. This will allow us to say, informally, that the property holds (or does not hold) for the problem Π, without actually specifying any encoding scheme. However, whenever we do so, the implicit assertion will be that we could, if requested, specify a particular reasonable encoding scheme e such that the property holds for $L[\Pi,e]$.

Notice that when we operate in this encoding-independent manner, we lose contact with any precise notion of "input length." Since we need some parameter in terms of which time complexity can be expressed, it is convenient to assume that every decision problem Π has an associated, encoding-independent function Length: $D_\Pi \to Z^+$, which is "polynomially related" to the input lengths we would obtain from a reasonable encoding scheme. By *polynomially related* we mean that, for any reasonable encoding scheme e for Π, there exist two polynomials p and p' such that if $I \in D_\Pi$ and x is a string encoding the instance I under e, then $\text{Length}[I] \leqslant p(|x|)$ and $|x| \leqslant p'(\text{Length}[I])$, where $|x|$ denotes the length of the string x. In the SUBGRAPH ISOMORPHISM problem, for example, we might take

$$\text{Length}[I] = |V_1| + |V_2|$$

where $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are the graphs making up an instance. In the TRAVELING SALESMAN decision problem we might take

$$\text{Length}[I] = m + \lceil \log_2 B \rceil + \max\{\lceil \log_2 d(c_i, c_j) \rceil : c_i, c_j \in C\}$$

Since any two reasonable encoding schemes for a problem Π will yield polynomially related input lengths, a wide variety of Length functions are possible for Π, and all our results will carry through for any such function that meets the above conditions.

The usefulness of this informal, encoding-independent approach depends, of course, on there being some agreement as to what constitutes a "reasonable" encoding scheme. The generally accepted meaning of "reasonable" includes both the notion of "conciseness," as captured by the two conditions mentioned in Chapter 1, and the notion of "decodability." The intent of "conciseness" is that instances of a problem should be described with the natural brevity we would use in actually specifying those instances for a computer, without any unnatural "padding" of the input. Such padding could be used, for example, to expand the input length so much that we artificially convert an exponential time algorithm into a polynomial time algorithm. The intent of "decodability" is that, given any particular component of a generic instance, one should be able to specify a polynomial time algorithm that is capable of extracting a description of that component from any given encoded instance.

Of course, these elaborations do not provide a formal definition of "reasonable encoding scheme," and we know of no satisfactory way of making such a definition. Even though most people would agree on whether or not a particular encoding scheme for a given problem is reasonable, the absence of a formal definition can be somewhat discomforting. One way of resolving this difficulty would be to require that generic problem instances always be formed from a fixed collection of basic types of set-theoretic objects. We will not impose such a constraint here, but, as an indication of our intent when we refer to "reasonable encoding schemes," we now give a brief description (which first time readers may wish to skip) of how such a standard encoding scheme could be defined.

Our standard encoding scheme will map instances into "structured strings" over the alphabet $\Psi = \{0, 1, -, [,], (,), ,\}$. We define *structured strings* recursively, as follows:

(1) The binary representation of an integer k as a string of 0's and 1's (preceded by a minus sign "$-$" if k is negative) is a structured string representing the integer k.

(2) If x is a structured string representing the integer k, then $[x]$ is a structured string that can be used as a "name" (for example, for a vertex in a graph, a set element, or a city in a traveling salesman instance).

(3) If $x_1, x_2, \ldots, x_m$ are structured strings representing the objects $X_1, X_2, \ldots, X_m$, then $(x_1, x_2, \ldots, x_m)$ is a structured string representing the sequence $\langle X_1, X_2, \ldots, X_m \rangle$.

To derive an encoding scheme for a particular decision problem specified in our standard format, we first note that, once we have built up a representation for each object in an instance as a structured string, the representation of the entire instance is determined using rule (3) above. Thus we need only specify how the representation for each type of object is constructed. For this we shall restrict ourselves to integers, "unstructured

elements" (vertices, elements, cities, etc.), sequences, sets, graphs, finite functions, and rational numbers.

Rules (1) and (3) already tell us how to represent integers and sequences. To represent each of the unstructured elements in an instance, we merely assign it a distinct "name," as constructed by rule (2), in such a way that if the total number of unstructured elements in an instance is N, then no name with magnitude exceeding N is used. The representations for the four other object types are as follows:

A *set* of objects is represented by ordering its elements as a sequence $\langle X_1, X_2, \ldots, X_m \rangle$ and taking the structured string corresponding to that sequence.

A *graph* with vertex set V and edge set E is represented by a structured string (x,y), where x is a structured string representing the set V, and y is a structured string representing the set E (the elements of E being the two-element subsets of V that are edges).

A *finite function* $f: \{U_1, U_2, \ldots, U_m\} \rightarrow W$ is represented by a structured string $((x_1,y_1),(x_2,y_2), \ldots, (x_m,y_m))$ where x_i is a structured string representing the object U_i and y_i is a structured string representing the object $f(U_i) \in W, 1 \leqslant i \leqslant m$.

A *rational number* q is represented by a structured string (x,y) where x is a structured string representing an integer a, y is a structured string representing an integer b, $a/b = q$, and the greatest common divisor of a and b is 1.

Although it might be convenient to have a wider collection of object types at our disposal, the ones above will suffice for most purposes and are enough to illustrate our notion of a reasonable encoding scheme. Furthermore, there would be no loss of generality in restricting ourselves to just these types for specifying generic instances, since other types of objects can always be expressed in terms of the ones above.

Note that our prescriptions are not sufficient to generate a *unique* string for encoding each instance but merely for ensuring that each string that does encode an instance obeys certain structural restrictions. A different choice of names for the basic elements or a different choice of order for the description of a set could lead to different strings that encode the same instance. In fact, it makes no difference how many strings encode an instance so long as we can decode each to obtain the essential components of the instance. Moreover, our definitions take this into account; for example, in $L[\Pi,e]$, the set of all strings that encode yes-instances of Π under e, each instance may be represented many times.

Before going on, we remind the reader that our standard encoding scheme is intended solely to illustrate how one might define such a standard scheme, although it also provides a reference point for what we mean by a "reasonable" encoding scheme. There is no reason why some other general scheme could not be used, or why we could not merely devise an individual encoding scheme for each problem of interest. If the chosen scheme is "equivalent" to ours, in the sense that there exist polynomial time algorithms for converting an encoding of an instance under either scheme to an encoding of that instance under the other scheme, then it, too, will be called "reasonable." If the chosen scheme is *not* equivalent to ours in this sense, then one can still prove results with respect to that scheme, but the encoding-independent terminology should not be used for describing them. Throughout this book we will restrict our attention to reasonable encoding schemes for problems.

2.2 Deterministic Turing Machines and the Class P

In order to formalize the notion of an algorithm, we will need to fix a particular model for computation. The model we choose is the *deterministic one-tape Turing machine* (abbreviated DTM), which is pictured schematically in Figure 2.1. It consists of a *finite state control*, a *read-write head*, and a *tape* made up of a two-way infinite sequence of *tape squares*, labeled $\ldots,-2,-1,0,1,2,3,\ldots$

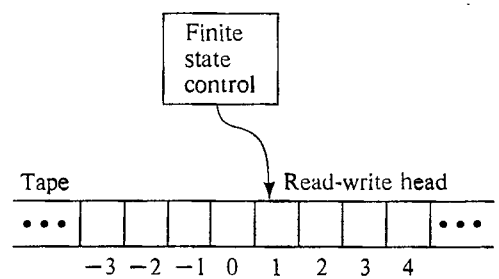


Figure 2.1 Schematic representation of a deterministic one-tape Turing machine (DTM).

A *program* for a DTM specifies the following information:

(1) A finite set Γ of tape *symbols*, including a subset $\Sigma \subset \Gamma$ of *input symbols* and a distinguished *blank symbol* $b \in \Gamma - \Sigma$;

(2) a finite set Q of *states*, including a distinguished *start-state* q_0 and two distinguished *halt-states* q_Y and q_N;

(3) a *transition function* $\delta: (Q - \{q_Y, q_N\}) \times \Gamma \rightarrow Q \times \Gamma \times \{-1,+1\}$.

The operation of such a program is straightforward. The *input* to the DTM is a string $x \in \Sigma^*$. The string x is placed in tape squares 1 through $|x|$, one symbol per square. All other squares initially contain the blank

symbol. The program starts its operation in state q_0, with the read-write head scanning tape square 1. The computation then proceeds in a step-by-step manner. If the current state q is either q_Y or q_N, then the computation has ended, with the answer being "yes" if $q = q_Y$ and "no" if $q = q_N$. Otherwise the current state q belongs to $Q - \{q_Y, q_N\}$, some symbol $s \in \Gamma$ is in the tape square being scanned, and the value of $\delta(q,s)$ is defined. Suppose $\delta(q,s) = (q',s',\Delta)$. The read-write head then erases s, writes s' in its place, and moves one square to the left if $\Delta = -1$, or one square to the right if $\Delta = +1$. At the same time, the finite state control changes its state from q to q'. This completes one "step" of the computation, and we are ready to proceed to the next step, if there is one.

$\Gamma = \{0,1,b\}$, $\Sigma = \{0,1\}$

$Q = \{q_0, q_1, q_2, q_3, q_Y, q_N\}$

q	0	1	b
q_0	$(q_0,0,+1)$	$(q_0,1,+1)$	$(q_1,b,-1)$
q_1	$(q_2,b,-1)$	$(q_3,b,-1)$	$(q_N,b,-1)$
q_2	$(q_Y,b,-1)$	$(q_N,b,-1)$	$(q_N,b,-1)$
q_3	$(q_N,b,-1)$	$(q_N,b,-1)$	$(q_N,b,-1)$

$\delta(q,s)$

Figure 2.2 An example of a DTM program $M = (\Gamma, Q, \delta)$.

An example of a simple DTM program M is shown in Figure 2.2. The transition function δ for M is described in a tabular format, where the entry in row q and column s is the value of $\delta(q,s)$. Figure 2.3 illustrates the computation of M on the input $x = 10100$, giving the state, head position, and contents of the non-blank portion of the tape before and after each step.

Note that this computation halts after eight steps, in state q_Y, so the answer for 10100 is "yes." In general, we say that a DTM program M with input alphabet Σ *accepts* $x \in \Sigma^*$ if and only if M halts in state q_Y when applied to input x. The language L_M *recognized* by the program M is given by

$$L_M = \{x \in \Sigma^* : M \text{ accepts } x\}$$

It is not hard to see that the DTM program of Figure 2.2 recognizes the language

$$\{x \in \{0,1\}^* : \textit{the rightmost two symbols of } x \textit{ are both } 0\}$$

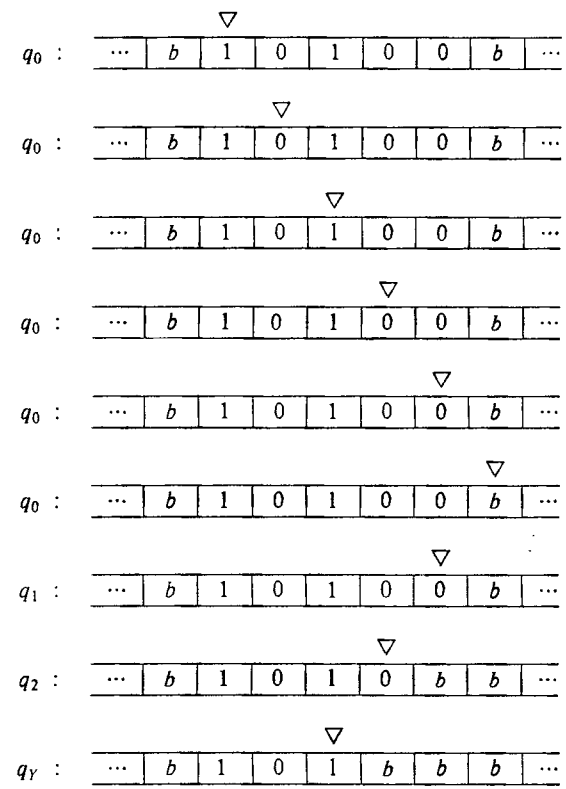


Figure 2.3 The computation of the program M from Figure 2.2 on input 10100.

Observe that this definition of language recognition does not require that M halt for *all* input strings in Σ^*, only for those in L_M. If x belongs to $\Sigma^* - L_M$, then the computation of M on x might halt in state q_N, or it might continue forever without halting. However, for a DTM program to correspond to our notion of an *algorithm*, it must halt on all possible strings over its input alphabet. In this sense, the DTM program of Figure 2.2 is algorithmic, since it will halt for any input string from $\{0,1\}^*$.

The correspondence between "recognizing" languages and "solving" decision problems is straightforward. We say that a DTM program M *solves* the decision problem Π under encoding scheme e if M halts for all input

strings over its input alphabet and $L_M = L[\Pi,e]$. The DTM program of Figure 2.2 once more provides an illustration. Consider the following number-theoretic decision problem:

INTEGER DIVISIBILITY BY FOUR
INSTANCE: A positive integer N.
QUESTION: Is there a positive integer m such that $N=4m$?

Under our standard encoding scheme, the integer N is represented by the string of 0's and 1's that is its binary representation. Since a positive integer is divisible by four if and only if the last two digits of its binary representation are 0, this DTM program "solves" the INTEGER DIVISIBILITY BY FOUR problem under our standard encoding scheme.

For future reference, we also point out that a DTM program can be used to compute functions. Suppose M is a DTM program with input alphabet Σ and tape alphabet Γ that halts for all input strings from Σ^*. Then M computes the *function* $f_M: \Sigma^* \rightarrow \Gamma^*$ where, for each $x \in \Sigma^*$, $f_M(x)$ is defined to be the string obtained by running M on input x until it halts and then forming a string from the symbols in tape squares 1, 2, 3, etc., in sequence, up to and including the rightmost non-blank tape square. The program M of Figure 2.2 computes the function $f_M: \{0,1\}^* \rightarrow \{0,1,b\}^*$ that maps each string $x \in \{0,1\}^*$ to the string $f_M(x)$ obtained by deleting the last two symbols of x (with $f_M(x)$ equal to the empty string if $|x|<2$).

It is well known that DTM programs are capable of performing much more complicated tasks than those illustrated by our simple example. Even though a DTM has only a single sequential tape and can perform only a very limited amount of work in a single step, a DTM program can be designed to perform any computation that can be performed on an ordinary computer, albeit more slowly. For the reader interested in how this is done, there are a number of excellent references, for example [Minsky, 1967] or [Hopcroft and Ullman, 1969]. For the reader who is *not* interested in how this is done, there is the welcome assurance that no expertise at programming DTMs will be required in this book. The reason for our introduction of the DTM model is to provide us with a formal counterpart of an algorithm upon which to base our definitions.

A formal definition of "time complexity" is now possible. The *time* used in the computation of a DTM program M on an input x is the number of steps occurring in that computation up until a halt state is entered. For a DTM program M that halts for all inputs $x \in \Sigma^*$, its *time complexity function* $T_M: Z^+ \rightarrow Z^+$ is given by

$$T_M(n) = \max\left\{ m: \begin{array}{l} \textit{there is an } x \in \Sigma^*, \textit{ with } |x|=n, \textit{ such that the} \\ \textit{computation of } M \textit{ on input } x \textit{ takes time } m \end{array} \right\}$$

Such a program M is called a *polynomial time DTM program* if there exists a polynomial p such that, for all $n \in Z^+$, $T_M(n) \leqslant p(n)$.

We are now ready to give the formal definition of the first important class of languages that we will be considering, the class P. It is defined as follows:

$$\mathrm{P} = \{ L: \textit{ there is a polynomial time DTM program } M \textit{ for which } L = L_M \}$$

We will say that a decision problem Π belongs to P under the encoding scheme e if $L[\Pi,e] \in \mathrm{P}$, that is, if there is a polynomial time DTM program that "solves" Π under encoding scheme e. In light of the previously mentioned equivalence between reasonable encoding schemes, we will usually omit the specification of a particular reasonable encoding scheme, simply saying that the decision problem Π belongs to P.

We also will be informal in our use of the term "polynomial time algorithm." Our formal counterpart for a polynomial time algorithm is the polynomial time DTM program. However, because of the equivalence between "realistic" computer models with respect to polynomial time pointed out in Chapter 1, the formal definition of P could have been rephrased in terms of programs for any such model and the same class of languages would have resulted. Thus we need not tie ourselves to the details of the DTM model when informally demonstrating that certain tasks can be performed by polynomial time algorithms. In fact, we will follow standard practice and discuss algorithms in an almost model-independent manner, speaking of them as operating directly on the components of an instance (the sets, graphs, numbers, etc.) rather than on their encoded descriptions. Here our implicit assertion is that one could, if one desired and had the patience, design a polynomial time DTM program corresponding to each polynomial time algorithm we discuss. Our informal demonstrations should be taken as indicating how this would be done and should be convincing to any reader familiar with the kinds of basic tasks that can be performed in polynomial time on an ordinary computer.

2.3 Nondeterministic Computation and the Class NP

In this section we introduce our second important class of languages/decision problems, the class NP. Before we proceed to the formal definitions in terms of languages and Turing machines, however, it will be useful to provide an intuitive idea of the informal notion this class is intended to capture.

Consider the TRAVELING SALESMAN problem described at the beginning of this chapter: Given a set of cities, the distances between them,

and a bound B, does there exist a tour of all the cities having total length B or less? There is no known polynomial time algorithm for solving this problem. However, suppose someone claimed, for a particular instance of this problem, that the answer for that instance is "yes." If we were skeptical, we could demand that they "prove" their claim by providing us with a tour having the required properties. It would then be a simple matter for us to verify the truth or falsity of their claim merely by checking that what they provided us with is actually a tour and, if so, computing its length and comparing that quantity to the given bound B. Furthermore, we could specify our "verification procedure" as a general algorithm that has time complexity polynomial in Length $[I]$.

Another example of a problem with this property is the SUBGRAPH ISOMORPHISM problem of Section 2.1. Given an arbitrary instance I of this problem, consisting of two graphs $G_1=(V_1,E_1)$ and $G_2=(V_2,E_2)$, if the answer for I is "yes," then this fact can be "proved" by giving the required subsets $V'\subseteq V_1$ and $E'\subseteq E_1$ and the required one-to-one function $f: V_2\rightarrow V'$. Again the validity of the claim can be verified easily in time polynomial in Length $[I]$, merely by checking that V', E', and f satisfy all the stated requirements.

It is this notion of polynomial time "verifiability" that the class NP is intended to isolate. Notice that polynomial time verifiability does not imply polynomial time solvability. In saying that one can verify a "yes" answer for a TRAVELING SALESMAN instance in polynomial time, we are not counting the time one might have to spend in searching among the exponentially many possible tours for one of the desired form. We merely assert that, given any tour for an instance I, we can verify in polynomial time whether or not that tour "proves" that the answer for I is "yes."

Informally we can define NP in terms of what we shall call a *nondeterministic algorithm*. We view such an algorithm as being composed of two separate stages, the first being a *guessing stage* and the second a *checking stage*. Given a problem instance I, the first stage merely "guesses" some structure S. We then provide both I and S as inputs to the checking stage, which proceeds to compute in a normal deterministic manner, either eventually halting with answer "yes," eventually halting with answer "no," or computing forever without halting (as we shall see, the latter two cases need not be distinguished). A nondeterministic algorithm "solves" a decision problem Π if the following two properties hold for all instances $I\in D_\Pi$:

1. If $I\in Y_\Pi$, then there exists some structure S that, when guessed for input I, will lead the checking stage to respond "yes" for I and S.
2. If $I\notin Y_\Pi$, then there exists *no* structure S that, when guessed for input I, will lead the checking stage to respond "yes" for I and S.

For example, a nondeterministic algorithm for TRAVELING SALESMAN could be constructed using a guessing stage that simply guesses an arbitrary sequence of the given cities and a checking stage that is identical to the aforementioned polynomial time "proof verifier" for TRAVELING SALESMAN. Clearly, for any instance I, there will exist a guess S that leads the checking stage to respond "yes" for I and S if and only if there is a tour of the desired length for I.

A nondeterministic algorithm that solves a decision problem Π is said to operate in "polynomial time" if there exists a polynomial p such that, for every instance $I\in Y_\Pi$, there is some guess S that leads the deterministic checking stage to respond "yes" for I and S within time $p(\text{Length}[I])$. Notice that this has the effect of imposing a polynomial bound on the "size" of the guessed structure S, since only a polynomially bounded amount of time can be spent examining that guess.

The class NP is defined informally to be the class of all decision problems Π that, under reasonable encoding schemes, can be solved by polynomial time nondeterministic algorithms. Our example above indicates that TRAVELING SALESMAN is one member of NP. The reader should have no difficulty in providing a similar demonstration for SUBGRAPH ISOMORPHISM.

The use of the term "solve" in these informal definitions should, of course, be taken with a grain of salt. It should be evident that a "polynomial time nondeterministic algorithm" is basically a definitional device for capturing the notion of polynomial time verifiability, rather than a realistic method for solving decision problems. Instead of having just one possible computation on a given input, it has many different ones, one for each possible guess.

There is another important way in which the "solution" of decision problems by nondeterministic algorithms differs from that for deterministic algorithms: the lack of symmetry between "yes" and "no." If the problem "Given I, is X true for I?" can be solved by a polynomial time (deterministic) algorithm, then so can the complementary problem "Given I, is X false for I?" This is because a deterministic algorithm halts for all inputs, so all we need do is interchange the "yes" and "no" responses (interchange states q_Y and q_N in a DTM program). It is not at all obvious that the same holds true for all problems solvable by polynomial time nondeterministic algorithms. Consider, for example, the complement of the TRAVELING SALESMAN problem: Given a set of cities, the intercity distances, and a bound B, is it true that *no* tour of all the cities has length B or less? There is no known way to verify a "yes" answer to this problem short of examining all possible tours (or a large proportion of them). In other words, no polynomial time nondeterministic algorithm for this complemen-

tary problem is known. The same is true of many other problems in NP. Thus, although membership in P for a problem Π implies membership in P for its complement, the analogous implication is not known to hold for NP.

We conclude this section by formalizing our definition in terms of languages and Turing machines. The formal counterpart of a nondeterministic algorithm is a program for a *nondeterministic one-tape Turing machine* (NDTM). For simplicity, we will be using a slightly non-standard NDTM model. (More standard versions are described in [Hopcroft and Ullman, 1969] and [Aho, Hopcroft, and Ullman, 1974]. The reader may find it an interesting exercise to verify the equivalence of our model to these with respect to polynomial time.)

The NDTM model we will be using has exactly the same structure as a DTM, except that it is augmented with a *guessing module* having its own *write-only head*, as illustrated schematically in Figure 2.4. The guessing module provides the means for writing down the "guess" and will be used solely for this purpose.

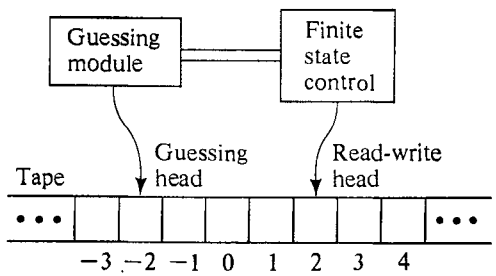


Figure 2.4 Schematic representation of a nondeterministic one-tape Turing machine (NDTM).

An *NDTM program* is specified in exactly the same way as a DTM program, including the tape alphabet Γ, input alphabet Σ, blank symbol b, state set Q, initial state q_0, halt states q_Y and q_N, and transition function δ: $(Q-\{q_Y,q_N\})\times\Gamma \rightarrow Q\times\Gamma\times\{-1,+1\}$. The computation of an NDTM program on an input string $x\in\Sigma^*$ differs from that of a DTM in that it takes place in two distinct stages.

The first stage is the "guessing" stage. Initially, the input string x is written in tape squares 1 through $|x|$ (while all other squares are blank), the read-write head is scanning square 1, the write-only head is scanning square -1, and the finite state control is "inactive." The guessing module then directs the write-only head, one step at a time, either to write some symbol from Γ in the tape square being scanned and move one square to the left, or to stop, at which point the guessing module becomes inactive

and the finite state control is activated in state q_0. The choice of whether to remain active, and, if so, which symbol from Γ to write, is made by the guessing module in a totally arbitrary manner. Thus the guessing module can write any string from Γ^* before it halts and, indeed, need never halt.

The "checking" stage begins when the finite state control is activated in state q_0. From this point on, the computation proceeds solely under the direction of the NDTM program according to exactly the same rules as for a DTM. The guessing module and its write-only head are no longer involved, having fulfilled their role by writing the guessed string on the tape. Of course, the guessed string can (and usually will) be examined during the checking stage. The computation ceases when and if the finite state control enters one of the two halt states (either q_Y or q_N) and is said to be an *accepting computation* if it halts in state q_Y. All other computations, halting or not, are classed together simply as *non-accepting computations*.

Notice that any NDTM program M will have an infinite number of possible computations for a given input string x, one for each possible guessed string from Γ^*. We say that the NDTM program M *accepts* x if at least one of these is an accepting computation. The language *recognized* by M is

$$L_M = \{x\in\Sigma^*: M \text{ accepts } x\}$$

The *time* required by an NDTM program M to accept the string $x\in L_M$ is defined to be the minimum, over all accepting computations of M for x, of the number of steps occurring in the guessing and checking stages up until the halt state q_Y is entered. The *time complexity function* T_M: $Z^+\rightarrow Z^+$ for M is

$$T_M(n) = \max\left\{\{1\}\cup\left\{m: \begin{array}{l}\text{there is an } x\in L_M \text{ with } |x|=n \text{ such}\\ \text{that the time to accept } x \text{ by } M \text{ is } m\end{array}\right\}\right\}$$

Note that the time complexity function for M depends only on the number of steps occurring in *accepting* computations, and that, by convention, $T_M(n)$ is set equal to 1 whenever no inputs of length n are accepted by M.

The NDTM program M is a *polynomial time NDTM program* if there exists a polynomial p such that $T_M(n)\leqslant p(n)$ for all $n\geqslant 1$. Finally, the class NP is formally defined as follows:

$$\text{NP} = \{L: \text{there is a polynomial time NDTM program } M \text{ for which } L_M = L\}$$

It is not hard to see how these formal definitions correspond to the informal definitions that preceded them. The only point deserving special mention is that, whereas we usually envision a nondeterministic algorithm as guessing a structure S that in some way depends on the given instance I, the guessing module of an NDTM entirely disregards the given input. However, since *every* string from Γ^* is a possible guess, we can always

design our NDTM program so that the checking stage begins by checking whether or not the guessed string corresponds (under the implicit interpretation our program places on strings) to an appropriate guess for the given input. If not, the program can immediately enter the halt state q_N.

A decision problem Π will be said to belong to NP under encoding scheme e if the language $L[\Pi, e] \in \text{NP}$. As with P, we shall feel free to say that Π is in NP without giving a specific encoding scheme, so long as it is clear that *some* reasonable encoding scheme for Π will yield a language that is in NP.

Furthermore, since any realistic computer model can be augmented with an analogue of our "guessing module with write-only head," we could have rephrased our formal definitions in terms of any of the other standard models of computation. Since all these models are equivalent with respect to deterministic polynomial time, the resulting versions of NP would all be identical. Thus we will be on firm ground when, as already proposed, we identify our formally defined class NP with the class of all decision problems "solvable" by polynomial time nondeterministic algorithms.

In the next section we discuss the relationship between the two classes P and NP as a preliminary to introducing our third and, for this book, most important class, the class of NP-complete problems.

2.4 The Relationship Between P and NP

The relationship between the classes P and NP is fundamental for the theory of NP-completeness. Our first observation, which is implicit in our earlier discussions but which has not been stated explicitly until now, is that $\text{P} \subseteq \text{NP}$. Every decision problem solvable by a polynomial time deterministic algorithm is also solvable by a polynomial time nondeterministic algorithm. To see this, one simply needs to observe that any deterministic algorithm can be used as the checking stage of a nondeterministic algorithm. If $\Pi \in \text{P}$, and A is any polynomial time deterministic algorithm for Π, we can obtain a polynomial time nondeterministic algorithm for Π merely by using A as the checking stage and ignoring the guess. Thus $\Pi \in \text{P}$ implies $\Pi \in \text{NP}$.

As we also hinted in our discussions, there are many reasons to believe that this inclusion is proper, that is, that P does not equal NP. Polynomial time nondeterministic algorithms certainly appear to be more powerful than polynomial time deterministic ones, and we know of no general methods for converting the former into the latter. In fact, the best general result we can state at present is given by the following:

Theorem 2.1 If $\Pi \in \text{NP}$, then there exists a polynomial p such that Π can be solved by a deterministic algorithm having time complexity $O(2^{p(n)})$.

Proof: Suppose A is a polynomial time nondeterministic algorithm for solving Π, and let $q(n)$ be a polynomial bound on the time complexity of A. (Without loss of generality, we can assume that q can be evaluated in polynomial time, for example, by taking $q(n) = c_1 n^{c_2}$ for suitably large integer constants c_1 and c_2.) Then we know that, for every accepted input of length n, there must exist some guessed string (over the tape alphabet Γ) of length at most $q(n)$ that leads the checking stage of A to respond "yes" for that input in no more than $q(n)$ steps. Thus the number of possible guesses that need be considered is at most $k^{q(n)}$, where $k = |\Gamma|$, since guesses shorter than $q(n)$ can be regarded as guesses of length exactly $q(n)$ by filling them out with blanks. We can deterministically discover whether A has an accepting computation for a given input of length n by applying the deterministic checking stage of A, until it halts or makes $q(n)$ steps, on each of the $k^{q(n)}$ possible guesses. The simulation responds "yes" if it encounters a guessed string that leads to an accepting computation within the time bound; otherwise it responds "no." This clearly yields a deterministic algorithm for solving Π. Furthermore, its time complexity is essentially $q(n) \cdot k^{q(n)}$, which, although exponential, is $O(2^{p(n)})$ for an appropriately chosen polynomial p. ■

Of course the simulation in the proof of Theorem 2.1 could be speeded up somewhat by using branch-and-bound techniques or backtrack search and by carefully enumerating the guesses so that obviously irrelevant strings are avoided. Nevertheless, despite the considerable savings that might be achieved, there is no known way to perform this simulation in less than exponential time.

Thus the ability of a nondeterministic algorithm to check an exponential number of possibilities in polynomial time might lead one to suspect that polynomial time nondeterministic algorithms are strictly more powerful than polynomial time deterministic algorithms. Indeed, for many individual problems in NP, such as TRAVELING SALESMAN, SUBGRAPH ISOMORPHISM, and a wide variety of others, no polynomial time solution algorithms have been found despite the efforts of many knowledgeable and persistent researchers.

For these reasons, it is not surprising that there is a widespread belief that $\text{P} \neq \text{NP}$, even though no proof of this conjecture appears on the horizon. Of course, a skeptic might say that our failure to find a proof that $\text{P} \neq \text{NP}$ is just as strong an argument in favor of $\text{P} = \text{NP}$ as our failure to find polynomial time algorithms is an argument for the opposite view. Problems always appear to be intractable until we discover efficient algorithms for solving them. Even a skeptic would be likely to agree, however, that, given our current state of knowledge, it seems more reasonable to operate under the assumption that $\text{P} \neq \text{NP}$ than to devote one's efforts to proving the contrary. In any case, we shall adopt a tentative picture of the world of NP as shown in Figure 2.5, with the expectation (but not the certainty) that the shaded region denoting $\text{NP} - \text{P}$ is not totally uninhabited.

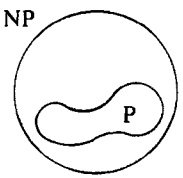


Figure 2.5 A tentative view of the world of NP.

2.5 Polynomial Transformations and NP-Completeness

If P differs from NP, then the distinction between P and NP − P is meaningful and important. All problems in P can be solved with polynomial time algorithms, whereas all problems in NP − P are intractable. Thus, given a decision problem $\Pi \in$ NP, if P $\neq$ NP, we would like to know which of these two possibilities holds for Π.

Of course, until we can prove that P $\neq$ NP, there is no hope of showing that any particular problem belongs to NP − P. For this reason, the theory of NP-completeness focuses on proving results of the weaker form "if P $\neq$ NP, then $\Pi \in$ NP−P." We shall see that, although these conditional results might appear to be almost as difficult to prove as the corresponding unconditional results, there are techniques available that often enable us to prove them in a straightforward way. The extent to which such results should be regarded as evidence for intractability depends on how strongly one believes that P differs from NP.

The key idea used in this conditional approach is that of a polynomial transformation. A *polynomial transformation* from a language $L_1 \subseteq \Sigma_1^*$ to a language $L_2 \subseteq \Sigma_2^*$ is a function $f: \Sigma_1^* \rightarrow \Sigma_2^*$ that satisfies the following two conditions:

1. There is a polynomial time DTM program that computes f.
2. For all $x \in \Sigma_1^*$, $x \in L_1$ if and only if $f(x) \in L_2$.

If there is a polynomial transformation from L_1 to L_2, we write $L_1 \propto L_2$, read "L_1 transforms to L_2" (dropping the modifier "polynomial," which is to be understood).

The significance of polynomial transformations comes from the following lemma:

Lemma 2.1 If $L_1 \propto L_2$, then $L_2 \in$ P implies $L_1 \in$ P (and, equivalently, $L_1 \notin$ P implies $L_2 \notin$ P).

Proof: Let Σ_1 and Σ_2 be the alphabets of L_1 and L_2 respectively, let $f: \Sigma_1^* \rightarrow \Sigma_2^*$ be a polynomial transformation from L_1 to L_2, let M_f denote a polynomial time DTM program that computes f, and let M_2 be a polynomial time DTM program that recognizes L_2. A polynomial time DTM program for recognizing L_1 can be constructed by composing M_f with M_2. For an input $x \in \Sigma_1^*$, we first apply the portion corresponding to program M_f to construct $f(x) \in \Sigma_2^*$. We then apply the portion corresponding to program M_2 to determine if $f(x) \in L_2$. Since $x \in L_1$ if and only if $f(x) \in L_2$, this yields a DTM program that recognizes L_1. That this program operates in polynomial time follows immediately from the fact that M_f and M_2 are polynomial time algorithms. To be specific, if p_f and p_2 are polynomial functions bounding the running times of M_f and M_2, then $|f(x)| \leqslant p_f(|x|)$, and the running time of the constructed program is easily seen to be $O(p_f(|x|) + p_2(p_f(|x|)))$, which is bounded by a polynomial in $|x|$. ■

If Π_1 and Π_2 are decision problems, with associated encoding schemes e_1 and e_2, we shall write $\Pi_1 \propto \Pi_2$ (with respect to the given encoding schemes) whenever there exists a polynomial transformation from $L[\Pi_1, e_1]$ to $L[\Pi_2, e_2]$. As usual, we will omit the reference to specific encoding schemes when we are operating under our standard assumption that only reasonable encoding schemes are used. Thus, at the problem level, we can regard a polynomial transformation from the decision problem Π_1 to the decision problem Π_2 as a function $f: D_{\Pi_1} \rightarrow D_{\Pi_2}$ that satisfies the two conditions:

1. f is computable by a polynomial time algorithm; and
2. for all $I \in D_{\Pi_1}$, $I \in Y_{\Pi_1}$ if and only if $f(I) \in Y_{\Pi_2}$.

Let us obtain a more concrete idea of what this definition means by considering an example. For a graph $G = (V, E)$ with vertex set V and edge set E, a *simple circuit* in G is a sequence $< v_1, v_2, \ldots, v_k >$ of distinct vertices from V such that $\{v_i, v_{i+1}\} \in E$ for $1 \leqslant i < k$ and such that $\{v_k, v_1\} \in E$. A *Hamiltonian circuit* in G is a simple circuit that includes all the vertices of G. The HAMILTONIAN CIRCUIT problem is defined as follows:

HAMILTONIAN CIRCUIT
INSTANCE: A graph $G = (V, E)$.
QUESTION: Does G contain a Hamiltonian circuit?

The reader will no doubt recognize a certain similarity between this problem and the TRAVELING SALESMAN decision problem. We shall show that HAMILTONIAN CIRCUIT (HC) transforms to TRAVELING SALESMAN (TS). This requires that we specify a function f that maps

each instance of HC to a corresponding instance of TS and that we prove that this function satisfies the two properties required of a polynomial transformation.

The function f is defined quite simply. Suppose $G=(V,E)$, with $|V|=m$, is a given instance of HC. The corresponding instance of TS has a set C of cities that is identical to V. For any two cities $v_i, v_j \in C$, the intercity distance $d(v_i,v_j)$ is defined to be 1 if $\{v_i,v_j\} \in E$ and 2 otherwise. The bound B on the desired tour length is set equal to m.

It is easy to see (informally) that this transformation f can be computed by a polynomial time algorithm. For each of the $m(m-1)/2$ distances $d(v_i,v_j)$ that must be specified, it is necessary only to examine G to see whether or not $\{v_i,v_j\}$ is an edge in E. Thus the first required property is satisfied. To verify that the second requirement is met, we must show that G contains a Hamiltonian circuit if and only if there is a tour of all the cities in $f(G)$ that has total length no more than B. First, suppose that $< v_1, v_2, \ldots, v_m >$ is a Hamiltonian circuit for G. Then $< v_1, v_2, \ldots, v_m >$ is also a tour in $f(G)$, and this tour has total length $m=B$ because each intercity distance traveled in the tour corresponds to an edge of G and hence has length 1. Conversely, suppose that $< v_1, v_2, \ldots, v_m >$ is a tour in $f(G)$ with total length no more than B. Since any two cities are either distance 1 or distance 2 apart, and since exactly m such distances are summed in computing the tour length, the fact that $B=m$ implies that each pair of successively visited cities must be exactly distance 1 apart. By the definition of $f(G)$, it follows that $\{v_i, v_{i+1}\}$, $1 \leqslant i < m$, and $\{v_m, v_1\}$ are all edges of G, and hence $< v_1, v_2, \ldots, v_m >$ is a Hamiltonian circuit for G.

Thus we have shown that HC $\propto$ TS. Although this proof is much simpler than many we will be describing, it contains all the essential elements of a proof of polynomial transformability and can serve as a model for how such proofs are constructed at the informal level.

The significance of Lemma 2.1 for decision problems now can be illustrated in terms of what it says about HC and TS. In essence, we conclude that if TRAVELING SALESMAN can be solved by a polynomial time algorithm, then so can HAMILTONIAN CIRCUIT, and if HC is intractable, then so is TS. Thus Lemma 2.1 allows us to interpret $\Pi_1 \propto \Pi_2$ as meaning that Π_2 is "at least as hard" as Π_1.

The "polynomial transformability" relation is especially useful because it is transitive, a fact captured by our next lemma.

Lemma 2.2 If $L_1 \propto L_2$ and $L_2 \propto L_3$, then $L_1 \propto L_3$.

Proof: Let Σ_1, Σ_2, and Σ_3 be the alphabets of languages L_1, L_2, and L_3, respectively, let $f_1: \Sigma_1^* \rightarrow \Sigma_2^*$ be a polynomial transformation from L_1 to L_2, and let $f_2: \Sigma_2^* \rightarrow \Sigma_3^*$ be a polynomial transformation from L_2 to L_3. Then the function $f: \Sigma_1^* \rightarrow \Sigma_3^*$ defined by $f(x)=f_2(f_1(x))$ for all $x \in \Sigma_1^*$ is the desired transformation from L_1 to L_3. Clearly, $f(x) \in L_3$ if and only if $x \in L_1$, and the fact that f can be computed by a polynomial time DTM program follows from an argument analogous to that used in the proof of Lemma 2.1. ■

We can define two languages L_1 and L_2 (two decision problems Π_1 and Π_2) to be *polynomially equivalent* whenever both $L_1 \propto L_2$ and $L_2 \propto L_1$ (both $\Pi_1 \propto \Pi_2$ and $\Pi_2 \propto \Pi_1$). Lemma 2.2 tells us that this is a legitimate equivalence relation and, furthermore, that the relation "$\propto$" imposes a partial order on the resulting equivalence classes of languages (decision problems). In fact, the class P forms the "least" equivalence class under this partial order and hence can be viewed as consisting of the computationally "easiest" languages (decision problems). The class of NP-complete languages (problems) will form another such equivalence class, distinguished by the property that it contains the "hardest" languages (decision problems) in NP.

Formally, a language L is defined to be *NP-complete* if $L \in$ NP and, for all other languages $L' \in$ NP, $L' \propto L$. Informally, a decision problem Π is NP-complete if $\Pi \in$ NP and, for all other decision problems $\Pi' \in$ NP, $\Pi' \propto \Pi$. Lemma 2.1 then leads us to our identification of the NP-complete problems as "the hardest problems in NP." If any single NP-complete problem can be solved in polynomial time, then *all* problems in NP can be so solved. If any problem in NP is intractable, then so are all NP-complete problems. An NP-complete problem Π, therefore, has the property mentioned at the beginning of this section: If P $\neq$ NP, then $\Pi \in$ NP $-$ P. More precisely, $\Pi \in$ P if and only if P = NP.

Assuming that P $\neq$ NP, we now can give a more detailed picture of "the world of NP," as shown in Figure 2.6. Notice that NP is not simply partitioned into "the land of P" and "the land of NP-complete." As we shall see in Chapter 7, if P differs from NP, then there must exist problems in NP that are neither solvable in polynomial time nor NP-complete.

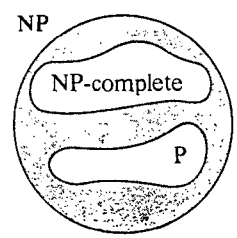


Figure 2.6 The world of NP, revisited.

Our main interest, however, is in the NP-complete problems themselves. Although we suggested at the outset of this section that there are straightforward techniques for proving that a problem is NP-complete, the

requirements we have just described would appear to be rather demanding. One must show that *every* problem in NP transforms to our prospective NP-complete problem Π. It is not at all obvious how one might go about doing this. *A priori*, it is not even apparent that any NP-complete problems need exist.

The following lemma, which is an immediate consequence of our definitions and the transitivity of $\propto$, shows that matters would be simplified considerably if we possessed just one problem that we knew to be NP-complete.

Lemma 2.3 If L_1 and L_2 belong to NP, L_1 is NP-complete, and $L_1 \propto L_2$, then L_2 is NP-complete.
Proof: Since $L_2 \in$ NP, all we need to do is show that, for every $L' \in$ NP, $L' \propto L_2$. Consider any $L' \in$ NP. Since L_1 is NP-complete, it must be the case that $L' \propto L_1$. The transitivity of $\propto$ and the fact that $L_1 \propto L_2$ then imply that $L' \propto L_2$. ■

Translated to the decision problem level, this lemma gives us a straightforward approach for proving new problems NP-complete, once we have at least one known NP-complete problem available. To prove that Π is NP-complete, we merely show that

1. $\Pi \in$ NP, and
2. some known NP-complete problem Π' transforms to Π.

Before we can use this approach, however, we still need some first NP-complete problem. Such a problem is provided by Cook's fundamental theorem, which we state and prove in the next section.

2.6 Cook's Theorem

The honor of being the "first" NP-complete problem goes to a decision problem from Boolean logic, which is usually referred to as the SATISFIABILITY problem (SAT, for short). The terms we shall use in describing it are defined as follows:

Let $U = \{u_1, u_2, \ldots, u_m\}$ be a set of Boolean *variables*. A *truth assignment* for U is a function $t: U \rightarrow \{T,F\}$. If $t(u) = T$ we say that u is "true" under t; if $t(u) = F$ we say that u is "false." If u is a variable in U, then u and $\bar{u}$ are *literals* over U. The literal u is true under t if and only if the variable u is true under t; the literal $\bar{u}$ is true if and only if the variable u is false.

A *clause* over U is a set of literals over U, such as $\{u_1, \bar{u}_3, u_8\}$. It represents the disjunction of those literals and is *satisfied* by a truth assignment if and only if at least one of its members is true under that assignment. The clause above will be satisfied by t *unless* $t(u_1) = F$, $t(u_3) = T$, and $t(u_8) = F$. A collection C of clauses over U is *satisfiable* if and only if there exists some truth assignment for U that simultaneously satisfies all the clauses in C. Such a truth assignment is called a *satisfying truth assignment* for C. The SATISFIABILITY problem is specified as follows:

SATISFIABILITY
INSTANCE: A set U of variables and a collection C of clauses over U.
QUESTION: Is there a satisfying truth assignment for C?

For example, $U = \{u_1, u_2\}$ and $C = \{\{u_1, \bar{u}_2\}, \{\bar{u}_1, u_2\}\}$ provide an instance of SAT for which the answer is "yes." A satisfying truth assignment is given by $t(u_1) = t(u_2) = T$. On the other hand, replacing C by $C' = \{\{u_1, u_2\}, \{u_1, \bar{u}_2\}, \{\bar{u}_1\}\}$ yields an instance for which the answer is "no"; C' is not satisfiable.

The seminal theorem of Cook [1971] can now be stated:

Theorem 2.1 (Cook's Theorem) SATISFIABILITY is NP-complete.
Proof: SAT is easily seen to be in NP. A nondeterministic algorithm for it need only guess a truth assignment for the given variables and check to see whether that assignment satisfies all the clauses in the given collection C. This is easy to do in (nondeterministic) polynomial time. Thus the first of the two requirements for NP-completeness is met.

For the second requirement, let us revert to the language level, where SAT is represented by a language $L_{SAT} = L[SAT,e]$ for some reasonable encoding scheme e. We must show that, for all languages $L \in$ NP, $L \propto L_{SAT}$. The languages in NP are a rather diverse lot, and there are infinitely many of them, so we cannot hope to present a separate transformation for each one of them. However, each of the languages in NP can be described in a standard way, simply by giving a polynomial time NDTM program that recognizes it. This allows us to work with a generic polynomial time NDTM program and to derive a generic transformation from the language it recognizes to L_{SAT}. This generic transformation, when specialized to a particular NDTM program M recognizing the language L_M, will give the desired polynomial transformation from L_M to L_{SAT}. Thus, in essence, we will present a simultaneous proof for all $L \in$ NP that $L \propto L_{SAT}$.

To begin, let M denote an arbitrary polynomial time NDTM program, specified by Γ, Σ, b, Q, q_0, q_Y, q_N, and δ, which recognizes the language $L = L_M$. In addition, let $p(n)$ be a polynomial over the integers that bounds the time complexity function $T_M(n)$. (Without loss of generality, we can assume that $p(n) \geqslant n$ for all $n \in Z^+$.) The generic transformation f_L will be derived in terms of M, Γ, Σ, b, Q, q_0, q_Y, q_N, δ, and p.

It will be convenient to describe f_L as if it were a mapping from strings over Σ to instances of SAT, rather than to strings over the alphabet of our encoding scheme for SAT, since the details of the encoding scheme could

be filled in easily. Thus f_L will have the property that for all $x \in \Sigma^*$, $x \in L$ if and only if $f_L(x)$ has a satisfying truth assignment. The key to the construction of f_L is to show how a set of clauses can be used to check whether an input x is accepted by the NDTM program M, that is, whether $x \in L$.

If the input $x \in \Sigma^*$ is accepted by M, then we know that there is an accepting computation for M on x such that both the number of steps in the checking stage and the number of symbols in the guessed string are bounded by $p(n)$, where $n = |x|$. Such a computation cannot involve any tape squares except for those numbered $-p(n)$ through $p(n)+1$, since the read-write head begins at square 1 and moves at most one square in any single step. The status of the checking computation at any one time can be specified completely by giving the contents of these squares, the current state, and the position of the read-write head. Furthermore, since there are no more than $p(n)$ steps in the checking computation, there are at most $p(n)+1$ distinct times that must be considered. This will enable us to describe such a computation completely using only a limited number of Boolean variables and a truth assignment to them.

The variable set U that f_L constructs is intended for just this purpose. Label the elements of Q as $q_0, q_1=q_Y, q_2=q_N, q_3, \ldots, q_r$, where $r=|Q|-1$, and label the elements of Γ as $s_0=b, s_1, s_2, \ldots, s_v$, where $v=|\Gamma|-1$. There will be three types of variables, each of which has an intended meaning as specified in Figure 2.7. By the phrase "at time i" we mean "upon completion of the i^{th} step of the checking computation."

Variable	Range	Intended meaning
$Q[i,k]$	$0 \leqslant i \leqslant p(n)$ $0 \leqslant k \leqslant r$	At time i, M is in state q_k.
$H[i,j]$	$0 \leqslant i \leqslant p(n)$ $-p(n) \leqslant j \leqslant p(n)+1$	At time i, the read-write head is scanning tape square j.
$S[i,j,k]$	$0 \leqslant i \leqslant p(n)$ $-p(n) \leqslant j \leqslant p(n)+1$ $0 \leqslant k \leqslant v$	At time i, the contents of tape square j is symbol s_k.

Figure 2.7 Variables in $f_L(x)$ and their intended meanings.

A computation of M induces a truth assignment on these variables in the obvious way, under the convention that, if the program halts before time $p(n)$, the configuration remains static at all later times, maintaining the same halt-state, head position, and tape contents. The tape contents at

time 0 consists of the input x, written in squares 1 through n, and the guess w, written in squares -1 through $-|w|$, with all other squares blank.

On the other hand, an arbitrary truth assignment for these variables need not correspond at all to a computation, much less to an accepting computation. According to an arbitrary truth assignment, a given tape square might contain many symbols at one time, the machine might be simultaneously in several different states, and the read-write head could be in any subset of the positions $-p(n)$ through $p(n)+1$. The transformation f_L works by constructing a collection of clauses involving these variables such that a truth assignment is a *satisfying* truth assignment if and only if it is the truth assignment induced by an accepting computation for x whose checking stage takes $p(n)$ or fewer steps and whose guessed string has length at most $p(n)$. We thus will have

$x \in L$ $\Leftrightarrow$ there is an accepting computation of M on x

$\Leftrightarrow$ there is an accepting computation of M on x with $p(n)$ or fewer steps in its checking stage and with a guessed string w of length exactly $p(n)$

$\Leftrightarrow$ there is a satisfying truth assignment for the collection of clauses in $f_L(x)$.

This will mean that f_L satisfies one of the two conditions required of a polynomial transformation. The other condition, that f_L can be computed in polynomial time, will be verified easily once we have completed our description of f_L.

The clauses in $f_L(x)$ can be divided into six groups, each imposing a separate type of restriction on any satisfying truth assignment as given in Figure 2.8.

It is straightforward to observe that if all six clause groups perform their intended missions, then a satisfying truth assignment will have to correspond to the desired accepting computation for x. Thus all we need to show is how clause groups performing these missions can be constructed.

Group G_1 consists of the following clauses:

$$\{Q[i,0], Q[i,1], \ldots, Q[i,r]\}, \quad 0 \leqslant i \leqslant p(n)$$

$$\{\overline{Q[i,j]}, \overline{Q[i,j']}\}, \quad 0 \leqslant i \leqslant p(n), \ 0 \leqslant j < j' \leqslant r$$

The first $p(n)+1$ of these clauses can be simultaneously satisfied if and only if, for each time i, M is in *at least* one state. The remaining $(p(n)+1)(r+1)(r/2)$ clauses can be simultaneously satisfied if and only if at no time i is M in *more than one* state. Thus G_1 performs its mission.

Groups G_2 and G_3 are constructed similarly, and groups G_4 and G_5 are both quite simple, each consisting only of one-literal clauses. Figure 2.9 gives a complete specification of the first five groups. Note that the number

Clause group	Restriction imposed
G_1	At each time i, M is in exactly one state.
G_2	At each time i, the read-write head is scanning exactly one tape square.
G_3	At each time i, each tape square contains exactly one symbol from Γ.
G_4	At time 0, the computation is in the initial configuration of its checking stage for input x.
G_5	By time $p(n)$, M has entered state q_Y and hence has accepted x.
G_6	For each time i, $0 \leqslant i < p(n)$, the configuration of M at time $i+1$ follows by a single application of the transition function δ from the configuration at time i.

Figure 2.8 Clause groups in $f_L(x)$ and the restrictions they impose on satisfying truth assignments.

of clauses in these groups, and the maximum number of literals occurring in each clause, are both bounded by a polynomial function of n (since r and v are *constants* determined by M and hence by L).

The final clause group G_6, which ensures that each successive configuration in the computation follows from the previous one by a single step of program M, is a bit more complicated. It consists of two subgroups of clauses.

The first subgroup guarantees that if the read-write head is *not* scanning tape square j at time i, then the symbol in square j does not change between times i and $i+1$. The clauses in this subgroup are as follows:

$$\{\overline{S[i,j,l]}, H[i,j], S[i+1,j,l]\},\ 0 \leqslant i < p(n), -p(n) \leqslant j \leqslant p(n)+1, 0 \leqslant l \leqslant v$$

For any time i, tape square j, and symbol s_l, if the read-write head is not scanning square j at time i, and square j contains s_l at time i but not at time $i+1$, then the above clause based on i, j, and l will fail to be satisfied (otherwise it *will* be satisfied). Thus the $2(p(n)+1)^2(v+1)$ clauses in this subgroup perform their mission.

Clause group	Clauses in group
G_1	$\{Q[i,0], Q[i,1], \ldots, Q[i,r]\},\ 0 \leqslant i \leqslant p(n)$ $\{\overline{Q[i,j]}, \overline{Q[i,j']}\},\ 0 \leqslant i \leqslant p(n), 0 \leqslant j < j' \leqslant r$
G_2	$\{H[i,-p(n)], H[i,-p(n)+1], \ldots, H[i,p(n)+1]\},\ 0 \leqslant i \leqslant p(n)$ $\{\overline{H[i,j]}, \overline{H[i,j']}\},\ 0 \leqslant i \leqslant p(n), -p(n) \leqslant j < j' \leqslant p(n)+1$
G_3	$\{S[i,j,0], S[i,j,1], \ldots, S[i,j,v]\},\ 0 \leqslant i \leqslant p(n), -p(n) \leqslant j \leqslant p(n)+1$ $\{\overline{S[i,j,k]}, \overline{S[i,j,k']}\}, 0 \leqslant i \leqslant p(n), -p(n) \leqslant j \leqslant p(n)+1, 0 \leqslant k < k' \leqslant v$
G_4	$\{Q[0,0]\}, \{H[0,1]\}, \{S[0,0,0]\},$ $\{S[0,1,k_1]\}, \{S[0,2,k_2]\}, \cdots, \{S[0,n,k_n]\},$ $\{S[0,n+1,0]\}, \{S[0,n+2,0]\}, \ldots, \{S[0,p(n)+1,0]\},$ where $x = s_{k_1} s_{k_2} \cdots s_{k_n}$
G_5	$\{Q[p(n),1]\}$

Figure 2.9 The first five clause groups in $f_L(x)$.

The remaining subgroup of G_6 guarantees that the *changes* from one configuration to the next are in accord with the transition function δ for M. For each quadruple (i,j,k,l), $0 \leqslant i < p(n)$, $-p(n) \leqslant j \leqslant p(n)+1$, $0 \leqslant k \leqslant r$, and $0 \leqslant l \leqslant v$, this subgroup contains the following three clauses:

$$\{\overline{H[i,j]}, \overline{Q[i,k]}, \overline{S[i,j,l]}, H[i+1,j+\Delta]\}$$
$$\{\overline{H[i,j]}, \overline{Q[i,k]}, \overline{S[i,j,l]}, Q[i+1,k']\}$$
$$\{\overline{H[i,j]}, \overline{Q[i,k]}, \overline{S[i,j,l]}, S[i+1,j,l']\}$$

where if $q_k \in Q - \{q_Y, q_N\}$, then the values of Δ, k', and l' are such that $\delta(q_k, s_l) = (q_{k'}, s_{l'}, \Delta)$, and if $q_k \in \{q_Y, q_N\}$, then $\Delta = 0$, $k' = k$, and $l' = l$.

Although it may require a few minutes of thought, it is not difficult to see that these $6(p(n))(p(n)+1)(r+1)(v+1)$ clauses impose the desired restriction on satisfying truth assignments.

Thus we have shown how to construct clause groups G_1 through G_6 performing the previously stated missions. If $x \in L$, then there is an accepting computation of M on x of length $p(n)$ or less, and this computation, given the interpretation of the variables, imposes a truth assignment that satisfies all the clauses in $C = G_1 \cup G_2 \cup G_3 \cup G_4 \cup G_5 \cup G_6$.

Conversely, the construction of C is such that any satisfying truth assignment for C must correspond to an accepting computation of M on x. It follows that $f_L(x)$ has a satisfying truth assignment if and only if $x \in L$.

All that remains to be shown is that, for any fixed language L, $f_L(x)$ can be constructed from x in time bounded by a polynomial function of $n = |x|$. Given L, we choose a particular NDTM M that recognizes L in time bounded by a polynomial p (we need not find this NDTM itself in polynomial time, since we are only proving that the desired transformation f_L *exists*). Once we have a specific NDTM M and a specific polynomial p, the construction of the set U of variables and collection C of clauses amounts to little more than filling in the blanks in a standard (though complicated) formula. The polynomial boundedness of this computation will follow immediately once we show that Length $[f_L(x)]$ is bounded above by a polynomial function of n, where Length $[I]$ reflects the length of a string encoding the instance I under a reasonable encoding scheme, as discussed in Section 2.1. Such a "reasonable" Length function for SAT is given, for example, by $|U| \cdot |C|$. No clause can contain more than $2 \cdot |U|$ literals (that's all the literals there are), and the number of symbols required to describe an individual literal need only add an additional $\log |U|$ factor, which can be ignored when all that is at issue is polynomial boundedness. Since r and v are fixed in advance and can contribute only constant factors to $|U|$ and $|C|$, we have $|U| = O(p(n)^2)$ and $|C| = O(p(n)^2)$. Hence Length $[f_L(x)] = |U| \cdot |C| = O(p(n)^4)$, and is bounded by a polynomial function of n as desired.

Thus the transformation f_L can be computed by a polynomial time algorithm (although the particular polynomial bound it obeys will depend on L and on our choices for M and p), and we conclude that, for every $L \in$ NP, f_L is a polynomial transformation from L to SAT (technically, of course, from L to L_{SAT}). It follows, as claimed, that SAT is NP-complete. ■

3

Proving NP-Completeness Results

If every NP-completeness proof had to be as complicated as that for SATISFIABILITY, it is doubtful that the class of known NP-complete problems would have grown as fast as it has. However, as discussed in Section 2.4, once we have proved a single problem NP-complete, the procedure for proving additional problems NP-complete is greatly simplified. Given a problem $\Pi \in$ NP, all we need do is show that some already known NP-complete problem Π' can be transformed to Π. Thus, from now on, the process of devising an NP-completeness proof for a decision problem Π will consist of the following four steps:

(1) showing that Π is in NP,

(2) selecting a known NP-complete problem Π',

(3) constructing a transformation f from Π' to Π, and

(4) proving that f is a (polynomial) transformation.

In this chapter, we intend not only to acquaint readers with the end results of this process (the finished NP-completeness proofs) but also to prepare them for the task of constructing such proofs on their own. In Section 3.1 we present six problems that are commonly used as the "known NP-complete problem" in proofs of NP-completeness, and we prove that