

# Chapter 14

## An Introduction to Computational Complexity

**I**n studying algorithms and computations, we have so far paid little attention to what actually can be expected when we apply these ideas to real computers. We have been almost exclusively concerned with questions of the existence or nonexistence of algorithms for certain problems. This is an appropriate starting point for a theory but clearly of limited practical significance. For actual computations, we need not only to know that a problem can be solved in principle, but we also must be able to construct algorithms that can be carried out with reasonable efficiency. Problems that can be solved effectively are called **tractable**, a descriptive term that will be given a more precise meaning in this chapter.

In the practical world of software development, efficiency has many facets. Sometimes, we are concerned with the efficient use of the computer's resources, such as processor time and memory space. At other times, we may be more concerned with how quickly software can be created, how effectively it can be maintained, or how reliable it is. At still other times, we may emphasize the efficiency with which a user's problems can be solved. All this is much too complicated to be captured by any abstract theory. All we can do is to focus on some of the more tangible issues and create the

appropriate abstract framework for these. Most of the results that have been developed address the space and time efficiency of a computation, leading to the important topic of **complexity theory**. In the study of complexity, the primary concern is the efficiency of a computation as measured by its time and space requirements. We refer to this as the **time-complexity** and the **space-complexity** of algorithms.

Computational complexity theory is an extensive topic, most of which is well outside the scope of this text. There are some results, however, that are simply stated and easily appreciated, and that throw further light on the nature of languages and computations. In this section, we give a brief overview of some complexity results. For the most part, proofs are difficult and we will dispense with them by reference to appropriate sources. Our intent here is to present the flavor of the subject matter and show how it relates to what we know about languages and automata. For this reason we will allow ourselves a great deal of latitude, both in the selection of topics and in the formality of the discussion.

We will limit our discussion here to issues of time-complexity. There are similar results for space-complexity, but time-complexity is a little more accessible.

---

## 14.1 Efficiency of Computation

Let us start with a concrete example. Given a list of one thousand integers, we want to sort them in some way, say, in ascending order. Sorting is a simple problem but also one that is very fundamental in computer science. If we now ask the question “How long will it take to do this task?” we see immediately that much more information is needed before we can answer it. Clearly, the number of items in the list plays an important role in how much time will be taken, but there are other factors. There is the question of what computer we use and how we write the program. Also, there are a number of sorting methods so that selection of the algorithm is important. There are probably a few more things you can think of that need to be looked at before you can even make a rough guess of the time requirements. If we have any hope of producing some general picture of sorting, most of these issues have to be ignored, and we must concentrate on those that are most fundamental.

For our discussion of computational complexity, we will make the following simplifying assumptions.

1. The model for our study will be a Turing machine. The exact type of Turing machine to be used will be discussed below.
2. The size of the problem will be denoted by  $n$ . For our sorting problem,  $n$  is obviously the number of items in the list. Although the size of a

problem is not always so easily characterized, we can generally relate it in some way to a positive integer.

3. In analyzing an algorithm, we are less interested in its performance on a specific case than in its general behavior. We are particularly concerned with how the algorithm behaves when the problem size increases. Because of this, the primary question is with how fast the resource requirements grow as  $n$  becomes large.

Our immediate goal will then be to characterize the time requirement of a problem as a function of its size, using a Turing machine as the computer model.

First, we give some meaning to the concept of time for a Turing machine. We think of a Turing machine as making one move per time unit, so the time taken by a computation is the number of moves made. As stated, we want to study how the computational requirements grow with the size of the problem. Normally, in the set of all problems of a given size, there is some variation. Here we are interested only in the worst case that has the highest resource requirements. By saying that a computation has a time-complexity  $T(n)$ , we mean that the computation for any problem of size  $n$  can be completed in no more than  $T(n)$  moves on some Turing machine.

After settling on a specific type of Turing machine as a computational model, we could analyze algorithms by writing explicit programs and counting the number of steps involved in solving the problem. But, for a variety of reasons, this is not overly useful. First, the number of operations performed may vary with the small details of the program and so may depend strongly on the programmer. Second, from a practical standpoint, we are interested in how the algorithm performs in the real world, which may differ considerably from how it does on a Turing machine. The best we can hope for is that the Turing machine analysis is representative of the major aspects of the real-life performance, for example, the asymptotic growth rate of the time complexity. Our first attempt at understanding the resource requirements of an algorithm is therefore invariably an *order-of-magnitude* analysis in which we use the  $O$ ,  $\Theta$ , and  $\Omega$  notation introduced in Chapter 1. In spite of the apparent informality of this approach, we often get very useful information.

#### Example 14.1

Given a set of  $n$  numbers  $x_1, x_2, \dots, x_n$  and a key number  $x$ , determine if the set contains  $x$ .

Unless the set is organized in some way, the simplest algorithm is just a *linear search* in which we compare  $x$  successively against  $x_1, x_2, \dots$ , until we either find a match or we get to the last element of the set. Since we may find a match on the first comparison or on the last, we cannot predict how much work is involved, but we know that, in the worst case, we have

to make  $n$  comparisons. We can then say that the time-complexity of this linear search is  $O(n)$ , or even better,  $\Theta(n)$ . In making this analysis, we made no specific assumptions about what machine this is run on or how the algorithm is implemented. ■

## EXERCISES

1. Suppose you are given a set of  $n$  numbers  $x_1, x_2, \dots, x_n$  and are asked to determine whether this set contains any duplicates.
  - (a) Suggest an algorithm and find an order-of-magnitude expression for its time-complexity.
  - (b) Examine if the implementation of the algorithm on a Turing machine affects your conclusions.
2. Repeat Exercise 2, this time determining if the set contains any triplicates. Is the algorithm as efficient as possible?
3. Review how the choice of algorithm affects the efficiency of sorting.

## 14.2 Turing Machines and Complexity

In Chapter 10 we argued that the various types of Turing machines were equivalent in their power to solve problems. This allowed us to take whatever type was most convenient for an argument and even use programs in higher-level computer languages to avoid some of the tedium involved in using the standard Turing machine model. But when we make complexity an issue, the equivalence between the various types of Turing machines no longer holds.

### Example 14.2

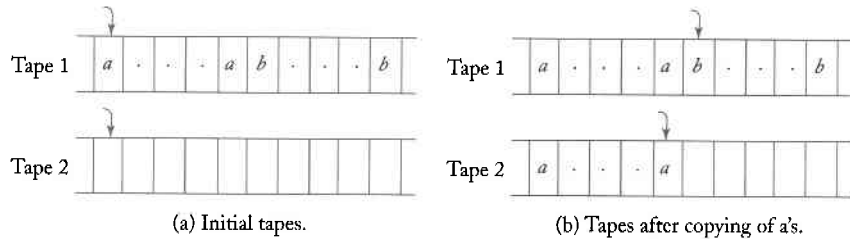
In Example 9.4 we constructed a single-tape Turing machine for the language

$$L = \{a^n b^n : n \geq 1\}.$$

A look at that algorithm will show that for  $w = a^n b^n$  it takes roughly  $2n$  steps to match each  $a$  with the corresponding  $b$ . Therefore the whole computation takes  $O(n^2)$  time.

But, as we later indicated in Example 10.1, with a two-tape machine we can use a different algorithm. We first copy all the  $a$ 's to the second

Figure 14.1



tape, then match them against the  $b$ 's on the first. The situation before and after the copying is shown in Figure 14.1. Both the copying and the matching can be done in  $O(n)$  time and we see that a two-tape machine has time-complexity  $O(n)$ . ■

**Example 14.3**

In Sections 5.2 and 6.3 we discussed the membership problem for context-free languages. If we take the length of the input string  $w$  as the problem size  $n$ , then the exhaustive search method has complexity  $O(n^M)$ , where  $M$  depends on the grammar. The more efficient CYK algorithm has complexity  $O(n^3)$ . Both these algorithms are deterministic.

A nondeterministic algorithm for this problem proceeds by simply guessing which sequence of productions is applied in the derivation of  $w$ . If we work with a grammar that has no unit or  $\lambda$ -productions, the length of the derivation is essentially  $|w|$ , so we have an  $O(n)$  algorithm. ■

**Example 14.4**

We now introduce the **satisfiability problem**, which plays an important role in complexity theory.

A logic or boolean constant or variable is one that can take on exactly two values, true or false, which we will denote by 1 and 0, respectively. Boolean operators are then used to combine boolean constants and variables into boolean expressions. The simplest boolean operators are *or*, denoted by  $\vee$  and defined by

$$\begin{aligned} 0 \vee 1 &= 1 \vee 0 = 1 \vee 1 = 1, \\ 0 \vee 0 &= 0, \end{aligned}$$

and the *and* operation ( $\wedge$ ) defined by

$$\begin{aligned} 0 \wedge 0 &= 0 \wedge 1 = 1 \wedge 0 = 0, \\ 1 \wedge 1 &= 1. \end{aligned}$$

Also needed is *negation*, denoted by a bar, and defined by

$$\begin{aligned} \overline{0} &= 1, \\ \overline{1} &= 0. \end{aligned}$$

We consider now boolean expressions in **conjunction normal form**. In this form, we create expressions from variables  $x_1, x_2, \dots, x_n$ , starting with

$$e = t_i \wedge t_j \wedge \dots \wedge t_k. \quad (14.1)$$

The terms  $t_i, t_j, \dots, t_k$  are created by or-ing together variables and their negation, that is,

$$t_i = s_l \vee s_m \vee \dots \vee s_p, \quad (14.2)$$

where each  $s_l, s_m, \dots, s_p$  stands for some variable or the negation of a variable.

The satisfiability problem is then simply stated: given an expression  $e$  in conjunctive normal form, is there an assignment of values to the variables  $x_1, x_2, \dots, x_n$  that will make the value of  $e$  true. For a specific case, look at

$$e_1 = (\overline{x}_1 \vee x_2) \wedge (x_1 \vee x_3).$$

The assignment  $x_1 = 0, x_2 = 1, x_3 = 1$  makes  $e_1$  true so that this expression is satisfiable. On the other hand,

$$e_2 = (x_1 \vee x_2) \wedge \overline{x}_1 \wedge \overline{x}_2$$

is not satisfiable because every assignment for the variables  $x_1$  and  $x_2$  will make  $e_2$  false.

A deterministic algorithm for the satisfiability problem is easy to discover. We take all possible values for the variables  $x_1, x_2, \dots, x_n$  and evaluate the expression. Since there are  $2^n$  such choices, this exhaustive approach has exponential time complexity.

Again, the nondeterministic approach simplifies matters. If  $e$  is satisfiable, we guess the value of each  $x_i$  and then evaluate  $e$ . This is essentially an  $O(n)$  algorithm. As in Example 14.3, we have a deterministic exhaustive search algorithm whose complexity is exponential and a linear nondeterministic one. However, unlike the previous example, we do not know of any nonexponential deterministic algorithm. ■

These examples suggest that complexity questions are affected by the type of Turing machine we use and that the issue of determinism versus nondeterminism is a particularly crucial one. Example 14.1 suggests that algorithms for a multitape machine may be reasonably close to what we might use when we program in a computer language. For this reason, we will use a multitape Turing machine as our model for studying complexity issues.

## EXERCISES

For the exercises in this set, assume that the Turing machines involved are all deterministic.

1. Find a linear-time algorithm for membership in  $\{ww : w \in \{a, b\}^*\}$  using a two-tape Turing machine. What is the best you could expect on a one-tape machine?
2. Show that any computation that can be performed on a single-tape, off-line Turing machine in time  $O(T(n))$  also can be performed on a standard Turing machine in time  $O(T(n))$ .
3. Show that any computation that can be performed on a standard Turing machine in time  $O(T(n))$  also can be performed on a Turing machine with one semi-infinite tape in time  $O(T(n))$ .
4. Show that any computation that can be performed on a two-tape machine in time  $O(T(n))$  can be performed on a standard Turing machine in time  $O(T^2(n))$ .
5. Rewrite the boolean expression

$$(x_1 \wedge x_2) \vee x_3$$

in conjunctive normal form.

6. Determine whether or not the expression

$$(x_1 \vee \bar{x}_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$$

is satisfiable.

7. In Example 14.2 we claimed that the first algorithm had time complexity  $O(n^2)$  and the second  $O(n)$ . Can we be more precise and claim that  $T(n) = \Theta(n^2)$  for the first case, and  $T(n) = \Theta(n)$  for the second? How this strengthen the argument in Example 14.2?

### 14.3 Language Families and Complexity Classes

In the Chomsky hierarchy for language classification, we associate language families with classes of automata, where each class of automata is defined by the nature of its temporary storage. Another way of classifying languages is to use a Turing machine of a particular type but consider time complexity a distinguishing factor. To do so, we first define the time complexity of a language.

---

#### Definition 14.1

We say that a Turing machine accepts a language  $L$  in time  $T(n)$  if every  $w$  in  $L$  with  $|w| \leq n$  is accepted in  $O(T(n))$  moves. If  $M$  is nondeterministic, this implies that for every  $w \in L$ , there is at least one sequence of moves of length  $O(T(|w|))$  that leads to acceptance.

---



---

#### Definition 14.2

A language  $L$  is said to be a member of the class  $DTIME(T(n))$  if there exists a deterministic multitape Turing machine that accepts  $L$  in time  $T(n)$ .

A language  $L$  is said to be a member of the class  $NTIME(T(n))$  if there exists a nondeterministic multitape Turing machine that accepts  $L$  in time  $T(n)$ .

---

Some relations between these complexity classes such as

$$DTIME(T(n)) \subseteq NTIME(T(n)),$$

and

$$T_1(n) = O(T_2(n))$$

implies

$$DTIME(T_1(n)) \subseteq DTIME(T_2(n)),$$

are obvious, but from here the situation gets obscure quickly. What we can say is that as the order of  $T(n)$  increases, we take in progressively more languages.



**Theorem 14.1**

For every integer  $k \geq 1$ ,

$$DTIME(n^k) \subset DTIME(n^{k+1}).$$

**Proof:** This follows from a result in Hopcroft and Ullman (1979, p. 299). ■

The conclusion we can draw from this is that there are some languages that can be accepted in time  $n^2$  for which there is no linear membership algorithm, that there are languages in  $DTIME(n^3)$  that are not in  $DTIME(n^2)$ , and so on. This gives us an infinite number of nested complexity classes. We get even more if we allow exponential time complexity. In fact, there is no limit to this; no matter how rapidly the complexity function  $T(n)$  grows, there is always something outside  $DTIME(T(n))$ .

**Theorem 14.2**

There is no total Turing computable function  $f(n)$  such that every recursive language is in  $DTIME(f(n))$ .

**Proof:** Consider the alphabet  $\Sigma = \{0, 1\}$ , with all strings in  $\Sigma^+$  arranged in proper order  $w_1, w_2, \dots$ . Also, assume that we have a proper ordering for the Turing machines in  $M_1, M_2, \dots$ .

Assume now that the function  $f(n)$  in the statement of the theorem exists. We can then define the language

$$L = \{w_i : M_i \text{ does not accept } w_i \text{ in } f(|w_i|) \text{ steps}\}. \quad (14.3)$$

We claim that  $L$  is recursive. To see this, consider any  $w \in L$  and compute first  $f(|w|)$ . By assuming that  $f$  is a total Turing computable function, this is possible. We next find the position  $i$  of  $w$  in the sequence  $w_1, w_2, \dots$ . This is also possible because the sequence is in proper order. When we have  $i$ , we find  $M_i$  and let it operate on  $w$  for  $f(|w|)$  steps. This will tell us whether or not  $w$  is in  $L$ , so  $L$  is recursive.

But we can now show that  $L$  is not in  $DTIME(f(n))$ . Suppose it were. Since  $L$  is recursive, there is some  $M_k$ , such that  $L = L(M_k)$ . Is  $w_k$  in  $L$ ? If we claim that  $w_k$  is in  $L$ , then  $M_k$  accepts  $w_k$  in  $f(|w_k|)$  steps. This is because  $L \in DTIME(f(n))$  and every  $w \in L$  is accepted by  $M_k$  in time  $f(|w|)$ . But this contradicts (14.3). Conversely, we get a contradiction if we assume that  $w_k \notin L$ . The inability to resolve this issue is a typical diagonalization result and leads us to conclude that the original assumption, namely the existence of a computable  $f(n)$ , must be false. ■

Theorem 14.1 and 14.2 allow us to make various claims; for example, that there is a language in  $DTIME(n^4)$  that is not in  $DTIME(n^3)$ . Although this may be of theoretical interest, it is not clear that such a result

has any practical significance. At this point, we have no clue what the characteristics of a language in  $DTIME(n^4)$  might be. We can get a little more insight into the matter if we relate the complexity classification to the languages in the Chomsky hierarchy. We will look at some simple examples that give some of the more obvious results.

**Example 14.5**

Every regular language can be recognized by a deterministic finite automaton in time proportional to the length of the input. Therefore

$$L_{REG} \subseteq DTIME(n).$$

But  $DTIME(n)$  includes much more than  $L_{REG}$ . We have already established in Example 13.7 that the context-free language  $\{a^n b^n : n \geq 0\}$  can be recognized in time  $O(n)$ . The argument given there can be used for even more complicated languages. ■

**Example 14.6**

The non-context-free language  $L = \{ww : w \in \{a, b\}^*\}$  is in  $NTIME(n)$ . This is straightforward, as we can recognize strings in this language by the algorithm

1. Copy the input from the input file to tape 1. Nondeterministically guess the middle of this string.
2. Copy the second part to tape 2.
3. Compare the symbols on tape 1 and tape 2 one by one.

Clearly all of the steps can be done in  $O(|w|)$  time, so  $L \in NTIME(n)$ .

Actually, we can show that  $L \in DTIME(n)$  if we can devise an algorithm for finding the middle of a string in  $O(n)$  time. This can be done: we look at each symbol on tape 1, keeping a count on tape 2, but counting only every second symbol. We leave the details as an exercise. ■

**Example 14.7**

It follows from Example 14.2 that

$$L_{CF} \subseteq DTIME(n^3)$$

and

$$L_{CF} \subseteq NTIME(n).$$

Consider now the family of context-sensitive languages. Exhaustive search parsing is possible here also since at every step only a limited number of productions are applicable. Therefore, every string of length  $n$  can be parsed in time  $n^M$ , where  $M$  depends on the grammar. Note, however, that we cannot claim from this that

$$L_{CS} \subseteq DTIME(n^M)$$

because we cannot put an upper bound on  $M$ . ■

From these examples we note a trend: as  $T(n)$  increases, more and more of the families  $L_{REG}$ ,  $L_{CF}$ ,  $L_{CS}$  are covered. But the connection between the Chomsky hierarchy and the complexity classes is tenuous and not very clear.

## EXERCISES

1. Complete the argument in Example 14.5.
2. Show that  $L = \{ww^Rw : w \in \{a, b\}^+\}$  is in  $DTIME(n)$ .
3. Show that  $L = \{www : w \in \{a, b\}^+\}$  is in  $DTIME(n)$ .
4. Show that there are languages that are not in  $NTIME(2^n)$ .

## 14.4 The Complexity Classes P and NP

Since the attempt to produce meaningful hierarchies via time-complexities with different growth rates appears to be unproductive, let us ignore some factors that are less important, for example by removing some uninteresting distinctions, such as that between  $DTIME(n^k)$  and  $DTIME(n^{k+1})$ . We can argue that the difference between, say,  $DTIME(n)$  and  $DTIME(n^2)$  is not fundamental, since some of it depends on the specific model of Turing machine we have (e.g., how many tapes), and it is not a priori clear which model is most appropriate for a real computer. This leads us to consider the famous complexity class

$$\mathbf{P} = \bigcup_{i \geq 1} DTIME(n^i).$$

This class includes all languages that are accepted by some deterministic Turing machine in polynomial time, without any regard to the degree of the polynomial. As we have already seen,  $L_{REG}$  and  $L_{CF}$  are in  $\mathbf{P}$ .

Since the distinction between deterministic and nondeterministic complexity classes appears to be fundamental, we also introduce

$$\mathbf{NP} = \bigcup_{i \geq 1} \mathbf{NTIME}(n^i).$$

Obviously

$$\mathbf{P} \subseteq \mathbf{NP},$$

but what is not known is if this containment is proper. While it is generally believed that there are some languages in  $\mathbf{NP}$  that are not in  $\mathbf{P}$ , no one has yet found an example of this.

The interest in these complexity classes, particularly in the class  $\mathbf{P}$ , comes from an attempt to distinguish between realistic and unrealistic computations. Certain computations, although theoretically possible, have such high resource requirements that in practice they must be rejected as unrealistic on existing computers, as well as on supercomputers yet to be designed. Such problems are sometimes called **intractable** to indicate that, while in principle computable, there is no realistic hope of a practical algorithm. To understand this better, computer scientists have attempted to put the idea of intractability on a formal basis. One attempt to define the term intractable is made in what is generally called the **Cook-Karp thesis**. In the Cook-Karp thesis, a problem that is in  $\mathbf{P}$  is called tractable, and one that is not is said to be intractable.

Is the Cook-Karp thesis a good way of separating problems we can work with realistically from those we cannot? The answer is not clear-cut. Obviously, any computation that is not in  $\mathbf{P}$  has time complexity that grows faster than any polynomial, and its requirements will increase very quickly with the problem size. Even for a function like  $2^{0.1n}$ , this will be excessive for large  $n$ , say  $n \geq 1000$ , and we might feel justified in calling a problem with this complexity intractable. But what about problems that are in  $\mathbf{DTIME}(n^{100})$ ? While the Cook-Karp thesis calls such a problem tractable, one surely cannot do much with it even for small  $n$ . The justification for the Cook-Karp thesis seems to lie in the empirical observation that most practical problems in  $\mathbf{P}$  are in  $\mathbf{DTIME}(n)$ ,  $\mathbf{DTIME}(n^2)$ , or  $\mathbf{DTIME}(n^3)$ , while those outside this class tend to have exponential complexities. Among practical problems, a clear distinction exists between problems in  $\mathbf{P}$  and those not in  $\mathbf{P}$ .

The study of the relation between the complexity classes  $\mathbf{P}$  and  $\mathbf{NP}$  has generated particular interest among computer scientists. At the root of this is the question whether or not

$$\mathbf{P} = \mathbf{NP}.$$

This is one of the fundamental unsolved problems in the theory of computation. To explore it, computer scientists have introduced a variety of

related concepts and questions. One of them is the idea of an **NP**-complete problem. Loosely speaking, an **NP**-complete problem is one that is as hard as any **NP** problem and in some sense is equivalent to all of them. What this means has to be explained.

---

**Definition 14.3**

A language  $L_1$  is said to be **polynomial-time reducible** to some language  $L_2$  if there exists a deterministic Turing machine by which any  $w_1$  in the alphabet of  $L_1$  can be transformed in polynomial time to a  $w_2$  in the alphabet of  $L_2$  in such a way that  $w_1 \in L_1$  if and only if  $w_2 \in L_2$ .

---

From this we see that if  $L_1$  is polynomial-time reducible to  $L_2$ , and if  $L_2 \in \mathbf{P}$ , then  $L_1 \in \mathbf{P}$ . Similarly, if  $L_2 \in \mathbf{NP}$ , then  $L_1 \in \mathbf{NP}$ .

---

**Definition 14.4**

A language  $L$  is said to be **NP**-complete if  $L \in \mathbf{NP}$  and if every  $L' \in \mathbf{NP}$  is polynomial-time reducible to  $L$ .

---


It follows easily from these definitions that if some  $L_1$  is **NP**-complete and polynomial-time reducible to  $L_2$ , then  $L_2$  is also **NP**-complete. The implication of this definition is that if we can find a deterministic polynomial-time algorithm for any **NP**-complete language, then every language in **NP** is also in **P**, that is

$$\mathbf{P} = \mathbf{NP}.$$

This puts **NP**-completeness in a central role for the study of this question.

---

**Example 14.8**

The satisfiability problem can be viewed as a language problem. We encode specific instances as a string that is accepted if and only if the expression is satisfiable. This problem is **NP**-complete. The statement that the satisfiability problem is **NP**-complete is known as **Cook's theorem**, a discussion of which can be found in Hopcroft and Ullman (1979). 

---

In addition to the satisfiability problem, a large number of other **NP**-complete problems have been found. For all of them we can find exponential

algorithms, but for none of them has anyone discovered a polynomial-time algorithm. These failures lead us to believe that probably

$$\mathbf{P} \neq \mathbf{NP},$$

but until someone produces an actual language in  $\mathbf{NP}$  that is not on  $\mathbf{P}$  or, alternatively, until someone proves that no such language exists, the question remains open.

## **EXERCISES**

1. Prove the statement that if a language  $L_1$  is  $\mathbf{NP}$ -complete and polynomial-time reducible to  $L_2$ , then  $L_2$  is also  $\mathbf{NP}$ -complete.
- ★★ 2. Consult books on complexity theory, and compile a list of problems that are  $\mathbf{NP}$ -complete.
3. Is it possible that the question  $\mathbf{P} = \mathbf{NP}$  is undecidable?