

1

Models of Computation and Complexity Classes

*O time! thou must untangle this, not I;
It is too hard a knot for me to untie.*
— William Shakespeare

The greatest friend of truth is time.
— Charles Caleb Colton

The notions of algorithms and complexity are meaningful only when they are defined in terms of formal computational models. In this chapter, we introduce our basic computational models: deterministic Turing machines and nondeterministic Turing machines. Based on these models, we define the notion of time and space complexity and the fundamental complexity classes including P and NP . In the last two sections, we study two best known proof techniques, diagonalization and simulation, that are used to separate and collapse complexity classes, respectively.

1.1 Strings, Coding, and Boolean Functions

Our basic data structure is a string. All other data structures are to be encoded and represented by strings. A *string* is a finite sequence of

Theory of Computational Complexity, Second Edition. Ding-Zhu Du and Ker-I Ko.
© 2014 John Wiley & Sons, Inc. Published 2014 by John Wiley & Sons, Inc.

symbols. For instance, the word *string* is a string over the symbols of English letters; the arithmetic expression “ $3 + 4 - 5$ ” is a string over symbols 3, 4, 5, +, and $-$. Thus, to describe a string, we must specify the set of symbols to occur in that string. We call a finite set of symbols to be used to define strings an *alphabet*. Note that not every finite set can be an alphabet. A finite set S can be an alphabet if and only if the following condition holds.

Property 1.1 *Two finite sequences of elements in S are identical if and only if the elements in the two sequences are identical respectively in ordering.*

For example, $\{0, 1\}$ and $\{00, 01\}$ are alphabets, but $\{1, 11\}$ is not an alphabet because 11 can be formed by either 11 or (1 and 1).

Assume that Σ is an alphabet. A set of strings over the alphabet Σ is called a *language*. A collection of languages is called a *language class*, or simply a *class*.

The length of a string x is the number of symbols in the string x , denoted by $|x|$. For example, $|string| = 6$ and $|3 + 4 - 5| = 5$. For convenience, we allow a string to contain no symbol. Such a string is called the *empty string*, which is denoted by λ . So, $|\lambda| = 0$. (The notation $|\cdot|$ is also used on sets. If S is a finite set, we write $|S|$ to denote its cardinality.)

There is a fundamental operation on strings. The *concatenation* of two strings x and y is the string xy . The concatenation follows associative law, that is, $x(yz) = (xy)z$. Moreover, $\lambda x = x\lambda = x$. Thus, all strings over an alphabet form a monoid under concatenation.¹ We denote $x^0 = \lambda$ and $x^n = xx^{n-1}$ for $n \geq 1$.

The concatenation operation on strings can be extended to languages. The concatenation of two languages A and B is the language $AB = \{ab : a \in A, b \in B\}$. We also denote $A^0 = \{\lambda\}$ and $A^n = AA^{n-1}$ for $n \geq 1$. In addition, we define $A^* = \bigcup_{i=0}^{\infty} A^i$. The language A^* is called the *Kleene closure* of A . The Kleene closure of an alphabet is the set of all strings over the alphabet.

For convenience, we will often work only on strings over the alphabet $\{0, 1\}$. To show that this does not impose a serious restriction on the theory, we note that there exists a simple way of encoding strings over any finite alphabet into the strings over $\{0, 1\}$. Let X be a finite set. A one–one mapping f from X to Σ^* is called a *coding* (of X in Σ^*). If both X and $\{f(x) : x \in X\}$ are alphabets, then, by Property 1.1, f induces a coding from X^* to Σ^* . Suppose that X is an alphabet of n elements. Choose $k = \lceil \log n \rceil$ and choose a one–one mapping f from X to $\{0, 1\}^k$.² Note

¹A set with an associative multiplication operation and an identity element is a *monoid*. A monoid is a *group* if every element in it has an inverse.

²Throughout this book, unless otherwise stated, \log denotes the logarithm function with base 2.



that any subset of $\{0, 1\}^k$ is an alphabet, and hence, f is a coding from X to $\{0, 1\}^*$ and f induces a coding from X^* to $\{0, 1\}^*$.

Given a linear ordering for an alphabet $\Sigma = \{a_1, \dots, a_n\}$, the *lexicographic ordering* $<$ on Σ^* is defined as follows: $x = a_{i_1}a_{i_2}\dots a_{i_m} < y = a_{j_1}a_{j_2}\dots a_{j_k}$ if and only if either $[m < k]$ or $[m = k$ and for some $\ell < m$, $i_1 = j_1, \dots, i_\ell = j_\ell$ and $i_{\ell+1} < j_{\ell+1}]$. The lexicographic ordering is a coding from natural numbers to all strings over an alphabet.

A coding from $\Sigma^* \times \Sigma^*$ to Σ^* is also called a *pairing function* on Σ^* . As an example, for $x, y \in \{0, 1\}^*$ define $\langle x, y \rangle = 0^{|x|}1xy$ and $x \# y = x0y1x^R$, where x^R is the reverse of x . Then $\langle \cdot, \cdot \rangle$ and “#” are pairing functions on $\{0, 1\}^*$. A pairing function induces a coding from $\underbrace{\Sigma^* \times \dots \times \Sigma^*}_n$ to Σ^* by

defining

$$\langle x_1, x_2, x_3, \dots, x_n \rangle = \langle \dots \langle \langle x_1, x_2 \rangle, x_3 \rangle, \dots, x_n \rangle.$$

Pairing functions can also be defined on natural numbers. For instance, let $\iota : \{0, 1\}^* \rightarrow \mathbb{N}$ be the lexicographic ordering function, that is, $\iota(x) = n$ if x is the n th string in $\{0, 1\}^*$ under the lexicographic ordering (starting with 0). Then, we can define a pairing function on natural numbers from a pairing function on binary strings: $\langle n, m \rangle = \iota(\langle \iota^{-1}(n), \iota^{-1}(m) \rangle)$.

In the above, we have seen some specific simple codings. In general, if A is a finite set of strings over some alphabet, when can A be an alphabet? Clearly, A cannot contain the empty string λ because $\lambda x = x\lambda$. The following theorem gives another necessary condition.

Theorem 1.2 (McMillan’s Theorem) *Let s_1, \dots, s_q be q nonempty strings over an alphabet of r symbols. If $\{s_1, \dots, s_q\}$ is an alphabet, then*

$$\sum_{i=1}^q r^{-|s_i|} \leq 1.$$

Proof. For any natural number n , consider

$$\left(\sum_{i=1}^q r^{-|s_i|} \right)^n = \sum_{k=n}^{n\ell} m_k r^{-k},$$

where $\ell = \max\{|s_1|, \dots, |s_q|\}$ and m_k is the number of elements in the following set:

$$A_k = \{(i_1, \dots, i_n) : 1 \leq i_1 \leq q, \dots, 1 \leq i_n \leq q, k = |s_{i_1}| + \dots + |s_{i_n}|\}.$$

As $\{s_1, \dots, s_q\}$ is an alphabet, different vectors (i_1, \dots, i_n) correspond to different strings $s_{i_1} \dots s_{i_n}$. The strings corresponding to vectors in A_k all have length k . Note that there are at most r^k strings of length k . Therefore, $m_k \leq r^k$. It implies



$$\left(\sum_{i=1}^q r^{-|s_i|} \right)^n \leq \sum_{k=n}^{n\ell} r^k r^{-k} = n\ell - (n-1) \leq n\ell. \quad (1.1)$$

Now, suppose $\sum_{i=1}^q r^{-|s_i|} > 1$. Then for sufficiently large n , $(\sum_{i=1}^q r^{-|s_i|})^n > n\ell$, contradicting (1.1). ■

A *Boolean function* is a function whose variable values and function value are all in {TRUE, FALSE}. We often denote TRUE by 1 and FALSE by 0. In the following table, we show two Boolean functions of two variables, *conjunction* \wedge and *disjunction* \vee , and a Boolean function of a variable, *negation* \neg .

x	y	$x \wedge y$	$x \vee y$	$\neg x$
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

All Boolean functions can be defined in terms of these three functions. For instance, the two-variable function *exclusive-or* \oplus can be defined by

$$x \oplus y = ((\neg x) \wedge y) \vee (x \wedge (\neg y)).$$

For simplicity, we also write xy for $x \wedge y$, $x + y$ for $x \vee y$, and \bar{x} for $\neg x$. A table like the above, in which the value of a Boolean function for each possible input is given explicitly, is called a *truth-table* for the Boolean function. For each Boolean function f over variables x_1, x_2, \dots, x_n , a function $\tau : \{x_1, x_2, \dots, x_n\} \rightarrow \{0, 1\}$ is called a *Boolean assignment* (or, simply, an *assignment*) for f . An assignment on n variables can be seen as a *binary* string of length n , that is, a string in $\{0, 1\}^n$. A function $\tau : Y \rightarrow \{0, 1\}$, where $Y = \{x_{i_1}, x_{i_2}, \dots, x_{i_k}\}$ is a subset of $X = \{x_1, x_2, \dots, x_n\}$, is called a *partial assignment* on X . A partial assignment τ on n variables can be seen as a string of length n over $\{0, 1, *\}$, with $*$ denoting “unchanged.” If $\tau : Y \rightarrow \{0, 1\}$ is a partial assignment for f , we write $f|_\tau$ or $f|_{x_{i_1}=\tau(x_{i_1}), \dots, x_{i_k}=\tau(x_{i_k})}$ to denote the function obtained by substituting $\tau(x_{i_j})$ for x_{i_j} , $1 \leq j \leq k$, into f . This function $f|_\tau$ is a Boolean function on $X - Y$ and is called the *restriction* of f by τ . We say a partial assignment τ *satisfies* f or τ is a *truth assignment* for f , if $f|_\tau = 1$.³

The functions conjunction, disjunction, and exclusive-or all follow the commutative and associative laws. The distributive law holds for conjunction to disjunction, disjunction to conjunction, and conjunction to exclusive-or, that is, $(x + y)z = xz + yz$, $xy + z = (x + z)(y + z)$, and

³In the literature, the term *truth assignment* sometimes simply means a Boolean assignment.



$(x \oplus y)z = xz \oplus yz$. An interesting and important law about negation is de Morgan's law, that is, $\overline{xy} = \overline{x} + \overline{y}$ and $\overline{x + y} = \overline{x} \overline{y}$. A *Boolean formula* is a formula over Boolean variables using operators \vee , \wedge , and \neg .

A *literal* is either a Boolean variable or the negation of a Boolean variable. An *elementary product* is a product of several literals. Consider an elementary product p and a Boolean function f . If $p = 1$ implies $f = 1$, then p is called an *implicant* of f . An implicant p is *prime* if no product of any proper subset of the literals defining p is an implicant of f . A prime implicant is also called a *minterm*. For example, function $f(x_1, x_2, x_3) = (x_1 + x_2)(\overline{x_2} + x_3)$ has minterms $x_1\overline{x_2}$, x_1x_3 , and x_2x_3 . $x_1\overline{x_2}x_3$ is an implicant of f but not a minterm. The *size* of an implicant is the number of variables in the implicant. We let $D_1(f)$ denote the maximum size of minterms of f . A DNF (*disjunctive normal form*) is a sum of elementary products. Every Boolean function is equal to the sum of all its minterms. So, every Boolean function can be represented by a DNF with terms of size at most $D_1(f)$. For a constant function $f \equiv 0$ or $f \equiv 1$, we define $D_1(f) = 0$. For a nonconstant function f , we always have $D_1(f) \geq 1$.

Similarly, an *elementary sum* is a sum of several literals. Consider an elementary sum c and a Boolean function f . If $c = 0$ implies $f = 0$, then c is called a *clause* of f . A minimal clause is also called a *prime clause*. The size of a clause is the number of literals in it. We let $D_0(f)$ denote the maximum size of prime clauses of f . A CNF (*conjunctive normal form*) is a product of elementary sums. Every Boolean function is equal to the product of all its prime clauses, which is a CNF with clauses of size at most $D_0(f)$. For a constant function $f \equiv 0$ or $f \equiv 1$, we define $D_0(f) = 0$. For a nonconstant function f , we always have $D_0(f) \geq 1$.

The following states a relation between implicants and clauses.

Proposition 1.3 *Any implicant and any clause of a Boolean function f have at least one variable in common.*

Proof. Let p and c be an implicant and a clause of f , respectively. Suppose that p and c have no variable in common. Then we can assign values to all variables in p to make $p = 1$ and to all variables in c to make $c = 0$ simultaneously. However, $p = 1$ implies $f = 1$ and $c = 0$ implies $f = 0$, which is a contradiction. ■

1.2 Deterministic Turing Machines

Turing machines (TMs) are simple and yet powerful enough computational models. Almost all reasonable general-purpose computational models have been known to be equivalent to TMs, in the sense that they define the same class of computable functions. There are many variations of TMs studied in literature. We are going to introduce, in this section,



the simplest model of TMs, namely, the *deterministic Turing machine* (DTM). Another model, the *nondeterministic Turing machine* (NTM), is to be defined in the next section. Other generalized TM models, such as deterministic and nondeterministic oracle TMs, will be defined in later chapters. In addition, we will introduce in Part II other *nonuniform* computational models which are not equivalent to TMs.

A deterministic (one-tape) TM (DTM) consists of two basic units: the *control unit* and the *memory unit*. The control unit contains a finite number of states. The memory unit is a tape that extends infinitely to both ends. The tape is divided into an infinite number of tape squares (or, tape cells). Each tape square stores one of a finite number of tape symbols. The communication between the control unit and the tape is through a *read/write tape head* that scans a tape square at a time (See Figure 1.1).

A normal *move* of a TM consists of the following actions:

- (1) Reading the tape symbol from the tape square currently scanned by the tape head;
- (2) Writing a new tape symbol on the tape square currently scanned by the tape head;
- (3) Moving the tape head to the right or left of the current square; and
- (4) Changing to a new control state.

The exact actions of (2)–(4) depend on the current control state and the tape symbol read in (1). This relation between the current state and the current tape symbol and actions (2)–(4) is predefined by a *program*.

Formally, a TM M is defined by the following information:

- (1) A finite set Q of states;
- (2) An initial state $q_0 \in Q$;
- (3) A subset $F \subseteq Q$ of accepting states;
- (4) A finite set Σ of input symbols;
- (5) A finite set $\Gamma \supset \Sigma$ of tape symbols, including a special blank symbol $B \in \Gamma - \Sigma$; and

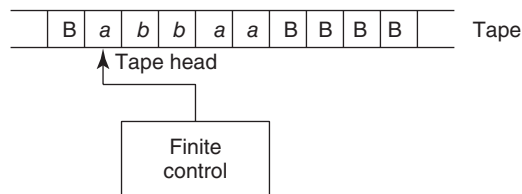


Figure 1.1 A Turing machine.



- (6) A *partial* transition function δ that maps $(Q - F) \times \Gamma$ to $Q \times \Gamma \times \{L, R\}$ (the *program*).

In the above, the transition function δ is a partial function, meaning that the function δ may be undefined on some pairs $(q, s) \in (Q - F) \times \Gamma$. The use of the initial state, accepting states, and the blank symbol is explained in the following text.

In order to discuss the notions of *accepting a language* and *computing a function* by a TM, we add some convention to the computation of a TM. First, we assume that initially an *input* string w is stored in the consecutive squares of the tape of M , and the other squares contain the blank symbol \mathbf{B} . The tape head of M is initially scanning the leftmost square of the input w , and the machine starts at the initial state q_0 . (Figure 1.1 shows the initial setting for a TM with input *abbaa*.) Starting from this initial configuration, the machine M operates move by move according to the transition function δ . The machine may either operate forever or halt when it enters a control state q and reads a tape symbol s for which $\delta(q, s)$ is undefined. If a TM M eventually halts at an accepting state $q \in F$ on input w , then we say M *accepts* w . If M halts at a nonaccepting state $q \notin F$ on input w , then we say M *rejects* w .

To formally define the notion of accepting an input string, we need to define the concept of *configurations*. A configuration α of a TM M is a record of all the information of the computation of M at a specific moment, which includes the current state, the current symbols in the tape, and the current position of the tape head. From this information, one can determine what the future computation is. Formally, a configuration of a TM M is an element (q, x, y) of $Q \times \Gamma^* \times \Gamma^*$ such that the leftmost symbol of x and the rightmost symbol of y are not \mathbf{B} . A configuration (q, x, y) denotes that the current state is q , the current nonblank symbols in the tape are the string xy , and the tape head is scanning the leftmost symbol of y (when y is empty, the tape head is scanning the blank that is immediately to the right of the rightmost symbol of x).⁴ Assuming $Q \cap \Gamma = \emptyset$, we also write xqy to stand for (q, x, y) .

We now generalize the transition function δ of a TM M to the *next configuration function* \vdash_M (or, simply \vdash if M is understood) defined on configurations of M . Intuitively, the function \vdash maps each configuration to the next configuration after one move of M . To handle the special nonblank requirement in the definition of configurations, we define two simple functions: $\ell(x)$ = the string x with the leading blanks removed and $r(x)$ = the string x with the trailing blanks removed. Assume that (q_1, x_1, y_1) is a configuration of M . If y_1 is not the empty string, then let $y_1 = s_1 y_2$ for some $s_1 \in \Gamma$ and $y_2 \in \Gamma^*$; if $y_1 = \lambda$, then let $s_1 = \mathbf{B}$ and

⁴The nonblank requirement for the leftmost symbol of x and for the rightmost symbol of y is added so that each configuration has a unique finite representation.



$y_2 = \lambda$. Then, we can formally define the function \vdash as follows (we write $\alpha \vdash \beta$ for $\vdash(\alpha) = \beta$):

Case 1. $\delta(q_1, s_1) = (q_2, s_2, L)$ for some $q_2 \in Q$ and $s_2 \in \Gamma$. If $x_1 = \lambda$, then let $s_3 = \mathbf{B}$ and $x_2 = \lambda$; otherwise, let $x_1 = x_2 s_3$ for some $x_2 \in \Gamma^*$ and $s_3 \in \Gamma$. Then, $(q_1, x_1, y_1) \vdash (q_2, \ell(x_2), r(s_3 s_2 y_2))$.

Case 2. $\delta(q_1, s_1) = (q_2, s_2, R)$ for some $q_2 \in Q$ and $s_2 \in \Gamma$. Then, $(q_1, x_1, y_1) \vdash (q_2, \ell(x_1 s_2), r(y_2))$.

Case 3. $\delta(q_1, s_1)$ is undefined. Then, \vdash is undefined on (q_1, x_1, y_1) .

Now we define the notion of the computation of a TM. A TM M *halts* on an input string w if there exists a finite sequence of configurations $\alpha_0, \alpha_1, \dots, \alpha_n$ such that

- (1) $\alpha_0 = (q_0, \lambda, w)$ (this is called the *initial configuration* for input w);
- (2) $\alpha_i \vdash \alpha_{i+1}$ for all $i = 0, 1, \dots, n-1$; and
- (3) $\vdash(\alpha_n)$ is undefined.

A TM M *accepts* an input string w if M halts on w and, in addition, the halting state is in F , that is, in (3) above, $\alpha_n = (q, x, y)$ for some $q \in F$ and $x, y \in \Gamma^*$. A TM M *outputs* $y \in \Sigma^*$ on input w if M halts on w and, in addition, the final configuration α_n is of the form $\alpha_n = (q, \lambda, y)$ for some $q \in F$.

Example 1.4 We describe a TM M that accepts the strings in $L = \{a^i b a^j : 0 \leq i \leq j\}$. The machine M has states $Q = \{q_0, q_1, \dots, q_5\}$, with the initial state q_0 and accepting state q_5 (i.e., $F = \{q_5\}$). It accepts input symbols from $\Sigma = \{a, b\}$ and uses tape symbols in $\Gamma = \{a, b, c, \mathbf{B}\}$. Figure 1.2 is the transition function δ of M .

It is not hard to check that M halts at state q_5 on all strings in L , that it halts at a state q_i , $0 \leq i \leq 4$, on strings having zero or more than one b , and that it does not halt on strings $a^i b a^j$ with $i > j \geq 0$. In the following, we show the computation paths of machine M on some inputs (we write $xq_i y$ to denote the configuration (q_i, x, y)):

δ	a	b	c	\mathbf{B}
q_0	q_1, c, R	q_4, \mathbf{B}, R	q_0, \mathbf{B}, R	
q_1	q_1, a, R	q_2, b, R		
q_2	q_3, c, L		q_2, c, R	q_2, \mathbf{B}, R
q_3	q_3, a, L	q_3, b, L	q_3, c, L	q_0, \mathbf{B}, R
q_4	q_4, a, R		q_4, \mathbf{B}, R	q_5, \mathbf{B}, R

Figure 1.2 The transition function of machine M .



On input $abaa$: $q_0abaa \vdash cq_1baa \vdash cbq_2aa \vdash cq_3bca \vdash q_3cbca \vdash q_3Bcbca \vdash q_0cbca \vdash q_0bca \vdash q_4ca \vdash q_4a \vdash aq_4 \vdash aBq_5$.

On input $aaba$: $q_0aaba \vdash cq_1aba \vdash caq_1ba \vdash cabq_2a \vdash caq_3bc \vdash cq_3abc \vdash q_3cabc \vdash q_3Bcabc \vdash q_0cabc \vdash q_0abc \vdash cq_1bc \vdash cbq_2c \vdash cbcq_2 \vdash cbcBq_2 \vdash cbcBBq_2 \vdash \dots$.

On input $abab$: $q_0abab \vdash cq_1bab \vdash cbq_2ab \vdash cq_3bcb \vdash q_3cbcb \vdash q_3Bcbcb \vdash q_0cbcb \vdash q_0bcb \vdash q_4cb \vdash q_4b$. \square

The notion of computable languages and computable functions can now be formally defined. In the following, we say f is a *partial function* defined on Σ^* if the domain of f is a subset of Σ^* , and f is a *total function* defined on Σ^* if the domain of f is Σ^* .

Definition 1.5 (a) A language A over a finite alphabet Σ is recursively enumerable (r.e.) if there exists a TM M that halts on all strings w in A and does not halt on any string w in $\Sigma^* - A$.

(b) A language A over a finite alphabet Σ is computable (or, recursive) if there exists a TM M that halts on all strings w in Σ^* , accepts all strings w in A and does not accept any string w in $\Sigma^* - A$.

(c) A partial function f defined from Σ^* to Σ^* is partial computable (or, partial recursive) if there exists a TM M that outputs $f(w)$ on all w in the domain of f and does not halt on any w not in the domain of f .

(d) A (total) function $f : \Sigma^* \rightarrow \Sigma^*$ is computable (or, recursive) if it is partial computable (i.e., the TM M that computes it halts on all $w \in \Sigma^*$).

For each TM M with the input alphabet Σ , we let $L(M)$ denote the set of all strings $w \in \Sigma^*$ that are accepted by M . Thus, a language A is recursively enumerable if and only if $A = L(M)$ for some TM M . Also, a language A is recursive if and only if $A = L(M)$ for some TM M that halts on all inputs w .

Recursive sets, recursively enumerable sets, partial recursive functions, and recursive functions are the main objects studied in *recursive function theory*, or, *recursion theory*. See, for instance, Rogers (1967) for a complete treatment.

The above classes of recursive sets and recursively enumerable sets are defined based on the model of deterministic, one-tape TMs. As TMs look very primitive, the question arises whether TMs are as powerful as other machine models. In other words, do the classes of recursive sets and recursively enumerable sets remain the same if they are defined based on different computational models? The answer is yes, according to the famous Church–Turing Thesis.

Church–Turing Thesis. A function computable in any reasonable computational model is computable by a TM.



What is a *reasonable computational model*? Intuitively, it is a model in which the following conditions hold:

- (1) The computation of a function is given by a set of finite instructions.
- (2) Each instruction can be carried out in this model in a finite number of steps or in a finite amount of time.
- (3) Each instruction can be carried out in this model in a deterministic manner.⁵

As the notion of *reasonable computational models* in the Church–Turing Thesis is not well defined mathematically, we cannot prove the Church–Turing Thesis as a mathematical statement but can only collect mathematical proofs as evidence to support it. So far, many different computational models have been proposed and compared with the TM model, and all *reasonable* ones are proven to be equivalent to TMs. The Church–Turing Thesis thus remains trustworthy.

In the following, we show that multi-tape TMs compute the same class of functions as one-tape TMs. A *multi-tape TM* is similar to a one-tape TM with the following exceptions. First, it has a finite number of tapes that extends infinitely to the both ends. Each tape is equipped with its own tape head. All tape heads are controlled by a common finite control. There are two special tapes: an *input tape* and an *output tape*. The input tape is used to hold the input strings only; it is a read-only tape that prohibits erasing and writing. The output tape is used to hold the output string when the computation of a function is concerned; it is a write-only tape. The other tapes are called the *storage tapes* or the *work tapes*. All work tapes are allowed to read, erase, and write (see Figure 1.3).

Next, we allow each tape head in a multi-tape TM, during a move, to stay at the same square without moving to the right or the left. Thus, each move of a k -tape TM is defined by a partial transition function δ that maps $(Q - F) \times \Gamma^k$ to $Q \times \Gamma^k \times \{L, R, S\}^k$ (where S stands for *stay*). The initial setting of the input tape of the multi-tape TM is the same as that of the one-tape TM, and other tapes of the multi-tape TM initially contain only blanks. The formal definition of the computation of a multi-tape TM on an input string x and the concepts of accepting a language and computing a function by a multi-tape TM can be defined similar to that of a one-tape TM. We leave it as an exercise.

⁵By condition (1), we exclude the nonuniform models; by condition (2), we exclude the models with infinite amount of resources; and by condition (3), we exclude the nondeterministic models and probabilistic models. Although they are considered unreasonable, meaning probably not realizable by reliable physical devices, these nonstandard models remain as interesting mathematical models and will be studied extensively in the rest of the book. In fact, we will see that the Church–Turing Thesis still holds even if we allow nondeterministic or probabilistic instructions in the computational model.

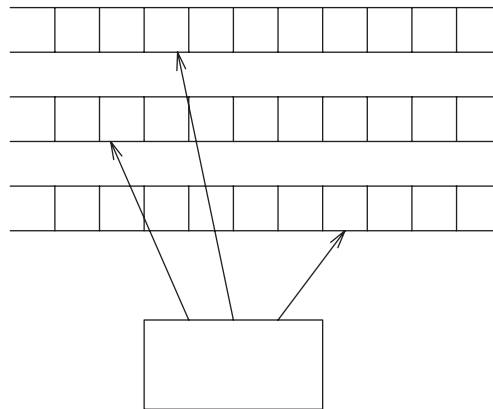


Figure 1.3 A multitape TM.

Theorem 1.6 For any multi-tape TM M , there exists a one-tape TM M_1 computing the same function as M .

Proof. Suppose that M has k tapes and the tape symbol set is Γ . Then we use the tape symbol set $\Gamma_1 = (\Gamma \times \{X, B\})^k$ for M_1 , where X is a symbol not in Γ . This means that we divide the one tape of M_1 into $2k$ sections which form k groups. Each group contains two sections: one uses the tape symbol set Γ and the other uses the tape symbol set $\{X, B\}$. Thus, the blank symbol in Γ_1 is (B, \dots, B) (with $2k$ B's). Each group records the information about a tape of M , with the symbol X in the second section indicating the position of the tape head, and the first section containing the corresponding symbol used by M . For instance, Figure 1.4 shows the machine M_1 that simulates a three-tape TM M . The nonblank symbols in the three tapes of M are 0110101, 0011110, and 1100110, and the tape heads of the three tapes are scanning the second, the fifth, and the last symbol of the nonblank symbols, respectively.

For each move of M , M_1 does the following to simulate it. First, we assume that after each simulation step, M_1 is scanning a square such that all symbols X appear to the right of that square. To begin the simulation, M_1 moves from left to right scanning all groups to look for the X symbols and the symbols in Γ that appear in the same groups as X 's. After it finds all X symbols, it has also collected all the tape symbols that are currently scanned by the tape heads of M (cf. Exercise 1.10). Next, M_1 moves back from right to left and looks for each X symbol again. This time, for each X symbol, M_1 properly simulates the action of M on that tape. Namely, it may write over the symbol of the first section of the square where the second section has an X symbol, or it may move the X symbol to the right or the left. The simulation finishes when actions on all k tapes are taken

	0	1	1	0	1	0	1				Tape 1
		X									Head 1
	0	0	1	1	1	1	0				Tape 2
					X						Head 2
	1	1	0	0	1	1	0				Tape 3
							X				Head 3

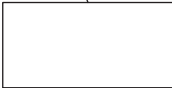


Figure 1.4 The TM M_1 .

care of. Note that, by then, all X symbols appear to the right of the tape head of M_1 .

To complete the description of the machine M_1 , we only need to add the initialization and the ending parts of the machine program for M_1 . That is, we first initialize the tape of M_1 so that it contains the input in the first group and all blanks in the other groups. At the end, when M reaches a halting configuration, M_1 erases all symbols in all groups except the output group. We omit the details. ■

It is obvious that a one-tape TM can be simulated by a multi-tape TM. So, an immediate corollary is that the set of functions computable by one-tape TMs is the same as that by multi-tape TMs. Similarly, we conclude that the set of recursive sets as well as the set of r.e. sets defined by one-tape TMs are the same as those defined by multi-tape TMs.

1.3 Nondeterministic Turing Machines

The TMs we defined in the last section are *deterministic*, because from each configuration of a machine there is at most one move to make, and hence, there is at most one next configuration. If we allow more than one moves for some configurations, and hence those configurations have more than one next configurations, then the machine is called a *nondeterministic Turing machine* (NTM).

Formally, an NTM M is defined by the following information: states Q ; initial state q_0 ; accepting states F ; input symbols Σ ; tape symbols Γ , including the blank symbol B ; and the transition relation Δ . All information except the transition relation Δ is defined in the same form as



a DTM. The transition relation Δ is a subset of $(Q - F) \times \Gamma \times Q \times \Gamma \times \{L, R\}$. Each quintuple (q_1, s_1, q_2, s_2, D) in Δ indicates that one of the possible moves of M , when it is in state q_1 and scanning symbol s_1 , is to change the current state to q_2 , to overwrite symbol s_1 by s_2 , and to move the tape head to the direction D .

The computation of an NTM can be defined similar to that of a DTM. First, we consider a way of restricting an NTM to a DTM. Let M be an NTM defined by $(Q, q_0, F, \Sigma, \Gamma, \Delta)$ as above. We say M_1 is a *restricted DTM* of M if M_1 has the same components $Q, q_0, F, \Sigma, \Gamma$ as M and it has a transition function δ_1 that is a subrelation of Δ satisfying the property that for each $q_1 \in Q$ and $s_1 \in \Gamma$, there is at most one triple (q_2, s_2, D) , $D \in \{L, R\}$, such that $(q_1, s_1, q_2, s_2, D) \in \delta_1$. Now we can define the notion of the next configurations of an NTM easily: For each configuration $\alpha = (q_1, x_1, y_1)$ of M , we let $\vdash_M(\alpha)$ be the set of all configurations β such that $\alpha \vdash_{M_1} \beta$ for some restricted DTM M_1 of M . We write $\alpha \vdash_M \beta$ if $\beta \in \vdash_M(\alpha)$. As each configuration of M may have more than one next configurations, the computation of an NTM on an input w is, in general, a *computation tree* rather than a single computation path (as it is in the case of DTMs). In the computation tree, each node is a configuration α and all its next configurations are its children. The root of the tree is the initial configuration.

We say an NTM M *halts* on an input string $w \in \Sigma^*$ if there exists a finite sequence of configurations $\alpha_0, \alpha_1, \dots, \alpha_n$ such that

- (1) $\alpha_0 = (q_0, \lambda, w)$;
- (2) $\alpha_i \vdash_M \alpha_{i+1}$ for all $i = 0, 1, \dots, n-1$; and
- (3) $\vdash_M(\alpha_n)$ is undefined (i.e., it is an empty set).

The notion of an NTM M halting on input w can be rephrased in terms of its computation tree as follows: M halts on w if the computation tree of $M(w)$ contains a finite path. This finite path (i.e., the sequence $\alpha_0, \alpha_1, \dots, \alpha_n$ of configurations satisfying the above conditions (1)–(3)) is called a *halting path* for $M(w)$.

A halting path is called an *accepting path* if the state of the last configuration is in F . An NTM M *accepts* an input string w if there exists an accepting path in the computation tree of M on w . (Note that this computation tree may contain some halting paths that are not accepting paths and may contain some nonhalting paths. As long as there exists at least one accepting path, we will say that M accepts w .) We say an NTM M *accepts a language* $A \subseteq \Sigma^*$ if M accepts all $w \in A$ and does not accept any $w \in \Sigma^* - A$. For each NTM M , we write $L(M)$ to denote the language accepted by M .

Example 1.7 Let $L = \{a^i b a^{i_2} b \cdots b a^{i_k} b b a^j : i_1, \dots, i_k, j > 0, \sum_{r \in A} i_r = j$ for some $A \subseteq \{1, 2, \dots, k\}\}$. We define an NTM $M = (Q = \{q_0, \dots, q_9\}$,



Δ	a	b	c	B
q_0	q_1, B, R q_2, c, R	q_7, B, R		
q_1	q_1, B, R	q_0, B, R		
q_2	q_2, a, R	q_3, b, R		
q_3	q_2, a, R	q_4, b, R		
q_4	q_5, c, L		q_4, c, R	q_4, B, R
q_5	q_5, a, L	q_5, b, L	q_5, c, L	q_6, B, R
q_6	q_2, c, R	q_0, B, R	q_6, B, R	
q_7			q_8, B, R	
q_8			q_8, B, R	q_9, B, R

Figure 1.5 The transition function of machine M .

q_0 , $F = \{q_9\}$, $\Sigma = \{a, b\}$, $\Gamma = \{a, b, c, B\}$, Δ) that accepts the set L . We show Δ in Figure 1.5.

The main idea of the machine M is that, for each block of a 's, except for the last one, it nondeterministically chooses (at state q_0) to either erase the whole block or, similar to Example 1.4, erase the block and the same number of a 's from the last block. Thus, if all blocks, including the last one, are erased, then accept. States q_2 to q_6 are devoted to the task of the second choice. We show the computation tree of M on input $aababbaa$ in Figure 1.6. \square

The notion of an NTM M computing a function is potentially ambiguous because for each input w , the computation tree of $M(w)$ may contain more than one halting path and each halting path may output a different value. We impose a strong requirement to eliminate the ambiguity.

Definition 1.8 (a) We say that an NTM M outputs y on input x if (i) there exists at least one computation path of $M(x)$ that halts in an accepting state with output y and (ii) whenever a computation path of $M(x)$ halts in an accepting state, its output is y .

(b) We say that an NTM M computes a partial function f from Σ^* to Σ^* if for each input $x \in \Sigma^*$ that is in the domain of f , M outputs $f(x)$, and for each input $x \in \Sigma^*$ that is not in the domain of f , M does not accept x .

The Church–Turing Thesis claims that DTMs are powerful enough to simulate other types of machine models as long as the machine model is a reasonable implementation of the intuitive notion of algorithms. NTMs are, however, not a very reasonable machine model, because the nondeterministic moves are obviously beyond the capability of currently existing and foreseeable physical computing devices. Nevertheless, DTMs are still

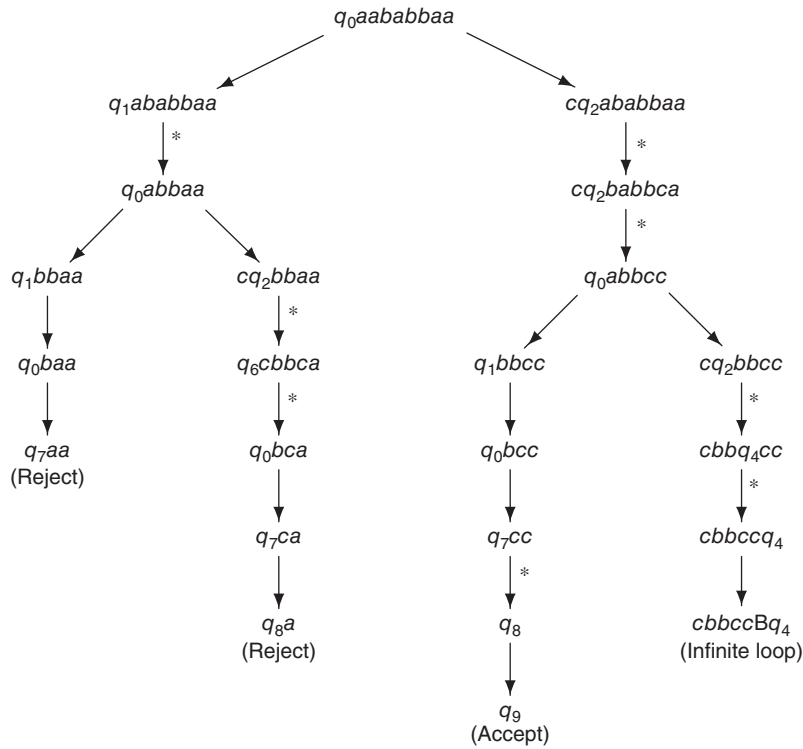


Figure 1.6 The computation tree of machine M on input $aababbaa$. An edge with $*$ means that the transition between configurations takes more than one deterministic moves.

powerful enough to be able to simulate NTMs (although this simulation may require much more resources for DTMs than the corresponding NTMs, as we will see in later sections).

Theorem 1.9

- (a) All languages accepted by NTMs are recursively enumerable.
- (b) All functions computed by NTMs are partial recursive.

Proof. Let M be a one-tape NTM defined by $(Q, q_0, F, \Sigma, \Gamma, \Delta)$. We may regard Δ as a multivalued function from $Q \times \Gamma$ to $Q \times \Gamma \times \{L, R\}$. Let k be the maximum number of values that Δ can assume on some $(q, s) \in Q \times \Gamma$.

We are going to design a DTM M_1 to simulate M . Our DTM M_1 is a three-tape DTM that uses k additional symbols η_1, \dots, η_k that are not in Γ . The DTM M_1 uses the third tape to simulate the tape of M and uses the second tape to store the current simulation information. More

precisely, M_1 operates on input x in stages. At stage $r > 0$, M_1 performs the following actions:

(1) M_1 erases anything in tape 3 that may have been left over from stage $r - 1$ and copies the input x from tape 1 to tape 3.

(2) M_1 generates the r th string $\eta_{i_1} \dots \eta_{i_m}$ in $\{\eta_1, \dots, \eta_k\}^*$, in the lexicographic ordering, on tape 2. (This string overwrites the $(r - 1)$ th string generated in stage $r - 1$.)

(3) M_1 simulates M on input x on tape 3 for at most m moves. At the j th move, $1 \leq j \leq m$, M_1 examines the j th symbol η_{i_j} on the second tape to determine which transition of the relation Δ is to be simulated. More precisely, if the current state is q and the symbol currently scanned on tape 3 is s , and $\Delta(q, s)$ contains at least i_j values, then M_1 follows the i_j th move of $\Delta(q, s)$; if $\Delta(q, s)$ has less than i_j values, then M_1 goes to stage $r + 1$.

(4) If the simulation halts within m moves in an accepting state, then M_1 enters a state in F and halts; otherwise, it goes to stage $r + 1$.

It is clear that if M does not accept x then M_1 never accepts x either. Conversely, if M accepts x , then there is a finite string $\eta_{i_1} \dots \eta_{i_m}$ with respect to which M_1 will simulate the accepting path of $M(x)$ and, hence, will accept x .

The above proves part (a). In addition, the above simulation of M_1 also yields the same output as M and so part (b) also follows. ■

Informally we often describe a nondeterministic algorithm as a *guess-and-verify* algorithm. That is, the nondeterministic moves of the algorithm are to guess a computation path and, for each computation path, a deterministic subroutine verifies that it is indeed an accepting path. The critical guess that determines an accepting path is called a *witness* to the input. For instance, the NTM of Example 1.7 guesses a subset $A \subseteq \{1, \dots, k\}$ and then verifies that $\sum_{r \in A} i_r = j$. In the above simulation of NTM M by a DTM M_1 , M_1 generates all strings $w = \eta_{i_1} \dots \eta_{i_m}$ in tape 2 one by one and then verifies that this string w is a witness to the input.

1.4 Complexity Classes

Computational complexity of a machine is the measure of the resources used by the machine in the computation. Computational complexity of a problem is the measure of the minimum resources required by any machine that solves the problem. For TMs, time and space are the two most important types of resources of concern. Let M be a one-tape DTM. For an input string x , the *running time* of M on x is the number of moves made by M from the beginning until it halts, denoted by $time_M(x)$. (We allow $time_M(x)$ to assume the value ∞ .) The *working space* of M



on input x is the number of squares which the tape head of M visited at least once during the computation, denoted by $space_M(x)$. (Again, $space_M(x)$ may be equal to ∞ . Note that $space_M(x)$ may be finite even if M does not halt on x .) For a multi-tape DTM M , the time complexity $time_M(x)$ is defined in a similar way, and the space complexity $space_M(x)$ is defined to count only the squares visited by the heads of working tapes, excluding the input and output tapes. This allows the possibility of having $space_M(x) < |x|$.

The functions $time_M$ and $space_M$ are defined on each input string. This makes it difficult to compare the complexity of two different machines. The common practice in complexity theory is to compare the complexity of two different machines based on their growth rates with respect to the input length. Thus, we define the time complexity of a DTM M to be the function $t_M : \mathbb{N} \rightarrow \mathbb{N}$ with $t_M(n) = \max\{time_M(x) : |x| = n\}$. The space complexity function s_M of M is similarly defined to be $s_M(n) = \max\{space_M(x) : |x| = n\}$.

We would like to define the (deterministic) time complexity of a function f to be the minimum t_M with respect to DTMs M which compute f . Unfortunately, for some recursive function f , there is no *best* DTM. In other words, for any machine M_1 that computes f , there is another machine M_2 also computing f such that $t_{M_2}(n) < t_{M_1}(n)$ for infinitely many n (*Blum's speed-up theorem*). So, formally, we can only talk about the upper bound and lower bound of the time complexity of a function. We say that the time (space) complexity of a recursive function f is bounded (above) by the function $\phi : \mathbb{N} \rightarrow \mathbb{N}$ if there exists a TM M that computes f such that for almost all $n \in \mathbb{N}$, $t_M(n) \leq \phi(n)$ ($s_M(n) \leq \phi(n)$, respectively). The time and space complexity of a recursive language L is defined to be, respectively, the time and space complexity of its characteristic function χ_L . (The *characteristic function* χ_L of a language L is defined by $\chi_L(x) = 1$ if $x \in L$ and $\chi_L(x) = 0$ if $x \notin L$.)

Now we can define complexity classes of languages as follows: Let $t : \mathbb{N} \rightarrow \mathbb{N}$ be a nondecreasing function from integers to integers, and C be a collection of such functions.

Definition 1.10 (a) We define $DTIME(t)$ to be the class of languages L that are accepted by DTMs M with $t_M(n) \leq t(n)$ for almost all $n \geq 0$. We let $DTIME(C) = \bigcup_{t \in C} DTIME(t)$.

(b) Similarly, we define $DSPACE(s)$ to be the class of languages L that are accepted by DTMs M with $s_M(n) \leq s(n)$ for almost all $n \geq 0$. We let $DSPACE(C) = \bigcup_{s \in C} DSPACE(s)$.

For NTMs, the notion of time and space complexity is a little more complex. Let M be an NTM. On each input x , $M(x)$ defines a computation tree. For each finite computation path of this tree, we define the time of the path to be the number of moves in the path and the space



of the path to be the total number of squares visited in the path by the tape head of M . We define the time complexity $time_M(x)$ of M on an input x to be the minimum time of the computation paths among all *accepting* paths of the computation tree of $M(x)$. The space complexity $space_M(x)$ is the minimum space of the computation paths among all *accepting* paths of the computation tree of $M(x)$. (When M is a multi-tape NTM, $space_M(x)$ only counts the space in the work tapes.) If M does not accept x (and, hence, there is no accepting path in the computation tree of $M(x)$), then $time_M(x)$ and $space_M(x)$ are undefined. (Note that the minimum time computation path and the minimum space computation path in the computation tree of $M(x)$ are not necessarily the same path.)

We now define the time complexity function $t_M : \mathbb{N} \rightarrow \mathbb{N}$ of an NTM M to be $t_M(n) = \max(\{n + 1\} \cup \{time_M(x) : |x| = n, M \text{ accepts } x\})$.⁶ Similarly, the space complexity function s_M of an NTM M is defined to be $s_M(n) = \max(\{1\} \cup \{space_M(x) : |x| = n, M \text{ accepts } x\})$.

Definition 1.11 (a) $NTIME(t) = \{L : L \text{ is accepted by an NTM } M \text{ with } t_M(n) \leq t(n) \text{ for almost all } n > 0\}$; $NTIME(C) = \bigcup_{t \in C} NTIME(t)$.

(b) $NSPACE(s) = \{L : L \text{ is accepted by an NTM } M \text{ with } s_M(n) \leq s(n) \text{ for almost all } n > 0\}$; $NSPACE(C) = \bigcup_{s \in C} NSPACE(s)$.

It is interesting to point out that for these complexity classes, an increase of a constant factor on the time or the space bounds does not change the complexity classes. The reason is that a TM with a larger alphabet set and a larger set of states can simulate, in one move, a constant number of moves of a TM with a smaller alphabet set and a smaller number of states. We only show these properties for deterministic complexity classes.

Proposition 1.12 (Tape Compression Theorem) *Assume that $c > 0$. Then, $DSPACE(s(n)) = DSPACE(c \cdot s(n))$.*

Proof. Let M be a TM that runs in space bound $s(n)$. We will construct a new TM M' that simulates M within space bound $c \cdot s(n)$, using the same number of tapes. Assume that M uses the alphabet Γ of size k , has r states $Q = \{q_1, \dots, q_r\}$, and uses t tapes. Let m be a positive integer such that $m > 1/c$. Divide the squares in each work tape of M into groups; each group contains exactly m squares. The machine M' uses an alphabet of size k^m so that each square of M' can simulate a group of squares of M . In addition, M' has rm^t states, each state represented by $\langle q_i, j_1, \dots, j_t \rangle$, for some $q_i \in Q$, $1 \leq j_1 \leq m, \dots, 1 \leq j_t \leq m$. Each state $\langle q_i, j_1, \dots, j_t \rangle$ encodes the information that the machine M is in state q_i and the local positions

⁶The extra value $n + 1$ is added so that when M does not accept any x of length n ; the time complexity is $n + 1$, the time to read the input x .



of its t tape heads within a group of squares. So, together with the position of its own tape heads, it is easy for M' to simulate each move of M using a smaller size of space. For instance, if a tape head of M moves left and its current local position within the group is j and $j > 1$, then the corresponding tape head of M' does not move but the state of M is modified so that its current local position within the group becomes $j - 1$; if $j = 1$, then the corresponding tape head of M' moves left and the local position is changed to m . By this simulation, $L(M) = L(M')$ and M' has a space bound $s(n)/m \leq c \cdot s(n)$. ■

Proposition 1.13 (Linear Speed-up Theorem) *Suppose $\lim_{n \rightarrow \infty} t(n)/n = \infty$. Then for any $c > 0$, $DTIME(t(n)) = DTIME(c \cdot t(n))$.*

Proof. Let M be a TM that runs in time bound $t(n)$. We will construct a new TM M' that simulates M within time $c \cdot t(n)$, using an additional tape. Let m be a large positive integer whose exact value is to be determined later. Divide the squares in each tape of M into groups; each group contains m squares. That is, if M uses alphabet Γ , then each group g is an element of Γ^m . For each group g , call its left neighboring group g_ℓ and the right neighboring group g_r . Let $H(g, g_\ell, g_r)$ be a history of the moves of M around g , that is, the collection of the moves of M starting from entering the group g until halting, entering a loop, or leaving the three groups g, g_ℓ , and g_r . (Note that the number of possible histories $H(g, g_\ell, g_r)$ is bounded by a function of m , independent of the input size n .)

The new machine M' encodes each group of M into a symbol. Initially, it encodes the input and copies it to tape 2. Then, it simulates M using the same number of work tapes (in addition to tape 2). It simulates each history $H(g, g_\ell, g_r)$ of M by a constant number of moves as follows: M' visits the groups g, g_ℓ , and g_r . Then, according to the contents of the three squares, it finds the history $H(g, g_\ell, g_r)$ of M . From the history $H(g, g_\ell, g_r)$, it overwrites new symbols on g, g_ℓ , and g_r and sets up the new configuration for the simulation of the next step. In the above simulation step, the contents of the three groups are stored in the states of M' and the history $H(g, g_\ell, g_r)$ is stored in the form of the transition function of M' . For instance, after M' gets the contents a, a_ℓ , and a_r of the three groups g, g_ℓ , and g_r , respectively, the state of M' is of the form $\langle q_i, j, a, a_\ell, a_r \rangle$, where q_i is a state of M and j is the starting position of the tape head of M within the group g ($j = 1, \dots, m$). (Here, we assume that M' uses only one tape.)

Note that each history $H(g, g_\ell, g_r)$ encodes at least m moves of M , and its simulation takes c_1 moves of M' where c_1 is an absolute constant, independent of m . As the initial setup of M' , including the encoding of the input string and returning the tape head to the leftmost square, takes $n + \lceil n/m \rceil$ moves, M' has the time bound

$$n + \lceil n/m \rceil + c_1 \lceil t(n)/m \rceil \leq n + n/m + c_1 t(n)/m + c_1 + 1.$$



Choose $m > c_1/c$. Because $\lim_{n \rightarrow \infty} t(n)/n = \infty$, we have that for sufficiently large n , $n + n/m + c_1 \cdot t(n)/m + c_1 + 1 \leq c \cdot t(n)$. Thus, $L(M) \in DTIME(c \cdot t(n))$. ■

The above two theorems allow us to write simply, for instance, $DSPACE(\log n)$ to mean $\bigcup_{c>0} DSPACE(c \cdot \log n)$ and $NTIME(n^2)$ to mean $\bigcup_{c>0} NTIME(c \cdot n^2)$.

Next, let us introduce the notion of polynomial-time computability. Let $poly$ be the collection of all integer polynomial functions with nonnegative coefficients. Define

$$\begin{aligned} P &= DTIME(poly), \\ NP &= NTIME(poly), \\ PSPACE &= DSPACE(poly), \\ NPSPACE &= NSPACE(poly). \end{aligned}$$

We say a language L is polynomial-time (polynomial-space) computable if $L \in P$ ($L \in PSPACE$, respectively).

In addition to polynomial-time/space-bounded complexity classes, some other important complexity classes are

$$\begin{aligned} LOGSPACE &= DSPACE(\log n), \\ NLOGSPACE &= NSPACE(\log n), \\ EXP &= \bigcup_{c>0} DTIME(2^{cn}), \\ NEXP &= \bigcup_{c>0} NTIME(2^{cn}), \\ EXPSPACE &= \bigcup_{c>0} DSPACE(2^{cn}). \end{aligned}$$

For the classes of functions, we only define the classes of (deterministically) polynomial-time and polynomial-space computable functions. Other complexity classes of functions will be defined later. We let FP to denote the class of all recursive functions f that are computable by a DTM M of time complexity $t_M(n) \leq p(n)$ for some $p \in poly$ and let $FPSPACE$ to denote the class of all recursive functions f that are computable by a DTM M of space complexity $s_M(n) \leq p(n)$ for some $p \in poly$.

The classes P and FP are often referred as the mathematical equivalence of the classes of *feasibly computable problems*. That is, a problem is considered to be feasibly solvable if it has a solution whose time complexity grows in a polynomial rate; on the other hand, if a solution has a superpolynomial growth rate on its running time, then the solution is not



feasible. Although this formulation is not totally accepted by practitioners, so far it remains the best one, because these complexity classes have many nice mathematical properties. For instance, if two functions f, g are feasibly computable, then we expect their composition $h(x) = f(g(x))$ also to be feasibly computable. The class FP is indeed closed under composition; as a matter of fact, it is the smallest class of functions that contains all quadratic-time computable functions and is closed under composition.

Another important property is that the classes P and FP are *independent of computational models*. We note that we have defined the classes P and FP based on the TM model. We have argued, based on the Church–Turing Thesis, that the TM model is as general as any other models, as far as the notion of computability is concerned. Here, when the notion of time and space complexity is concerned, the Church–Turing Thesis is no longer applicable. It is obvious that different machine models define different complexity classes of languages. However, as far as the notion of polynomial-time computability is concerned, it can be verified that most familiar computational models define the same classes P and FP . The idea that the classes P and FP are machine independent can be formulated as follows.

Extended Church–Turing Thesis. A function computable in polynomial time in any *reasonable* computational model using a *reasonable* time complexity measure is computable by a DTM in polynomial time.

The intuitive notion of *reasonable computational models* has been discussed in Section 1.2. They exclude any machine models that are not realizable by physical devices. In particular, we do not consider an NTM to be a reasonable model. In addition, we also exclude any time complexity measures that do not reflect the physical time requirements. For instance, in a model like the random access machine (see Exercises 1.14 and 1.15), the *uniform time complexity measure* for which an arithmetic operation on integers counts only one unit of time, no matter how large the integers are, is not considered as a reasonable time complexity measure.⁷

The Extended Church–Turing Thesis is a much stronger statement than the Church–Turing Thesis, because it implies that TMs can simulate each instruction of any other reasonable models within time polynomial in the length of the input. It is not as widely accepted as the Church–Turing Thesis. To disprove it, however, one needs to demonstrate in a formal manner a reasonable model and a problem A and to prove

⁷The reader should keep in mind that we are here developing a general theory of computational complexity for all computational problems. Depending on the nature of the problems, we may sometime want to apply more specific complexity measures to specific problems. For instance, when we compare different algorithms for the integer matrix multiplication problem, the uniform time complexity measure appears to be more convenient than the bit-operation (or, logarithmic) time complexity measure.



formally that A is computable in the new model in polynomial time and A is not solvable by DTMs in polynomial time. Although there are some candidates for the counterexample A , such a formal proof has not yet been found.⁸

To partially support the Extended Church–Turing Thesis, we verify that the simulation of a multi-tape TM by a one-tape TM described in Theorem 1.6 can be implemented in polynomial time.

Corollary 1.14 *For any multi-tape TM M , there exists a one-tape TM M_1 that computes the same function as M in time $t_{M_1}(n) = O((t_M(n))^2)$.*

Proof. In the simulation of Theorem 1.6, for each move of M , the machine M_1 makes one pass from left to right and then one pass from right to left. In the first pass, the machine M_1 looks for all X symbols and then stores the information of tape symbols currently scanned by M in its states, and it takes at most $time_M(x)$ moves to find all X symbols. (Note that at any point of computation, the tape heads of M can move at most $time_M(x)$ squares away from the starting square.) In the second pass, M_1 needs to adjust the positions of the X symbols and to erase and write new symbols in the first section of each group. For each group, these actions take two additional moves, and the total time for the second pass takes only $time_M(x) + 2k$, where k is the number of tapes of M . Thus, the total running time of M_1 on x is at most $(time_M(x))^2 + (2k + 2)time_M(x)$ (where the extra $2time_M(x)$ is for the initialization and ending stages). ■

The above quadratic simulation time can be reduced to quasi-linear time (i.e., $t_M(n) \cdot \log(t_M(n))$) if we allow M_1 to have two work tapes (in addition to the input tape).

Theorem 1.15 *For any multi-tape TM M , there exists a two-worktape TM M_1 that computes the same function as M in time $t_{M_1}(n) = O(t_M(n) \cdot \log(t_M(n)))$.*

Proof. See Exercise 1.19. ■

For the space complexity, we observe that the machine M_1 in Theorem 1.6 does not use any extra space than M_1 .

Corollary 1.16 *For any multi-tape TM M , there exists a one-worktape TM M_1 that computes the same function as M in space $s_{M_1}(n) = O(s_M(n))$.*

⁸One of the new computational models that challenges the validity of the Extended Church–Turing Thesis is the quantum TM model. It has been proved recently that some number theoretic problems which are believed to be not solvable in polynomial time by DTMs (or, even by probabilistic TMs) are solvable in polynomial time by quantum TMs. See, for instance, Shor (1997) and Bernstein and Vazirani (1997) for more discussions.



The relationship between the time- and space-bounded complexity classes and between the deterministic and nondeterministic complexity classes is one of the main topics in complexity theory. In particular, the relationship between the polynomial bounded classes, such as whether $P = NP$ and whether $P = PSPACE$, presents the ultimate challenge to the researchers. We discuss these questions in the following sections.

1.5 Universal Turing Machine

One of the most important properties of a computation system like TMs is that there exists a universal machine that can simulate each machine from its code.

Let us first consider one-tape DTMs with the input alphabet $\{0, 1\}$, the working alphabet $\{0, 1, \mathbf{B}\}$, the initial state q_0 , and the final state set $\{q_1\}$, that is, DTMs defined by $(Q, q_0, \{q_1\}, \{0, 1\}, \{0, 1, \mathbf{B}\}, \delta)$. Such a TM can be determined by the definition of the transition function δ only, for Q is assumed to be the set of states appearing in the definition of δ . Let us use the notation q_i , $0 \leq i \leq |Q| - 1$, for a state in Q , X_j , $j = 0, 1, 2$, for a tape symbol where $X_0 = 0$, $X_1 = 1$ and $X_2 = \mathbf{B}$, and D_k , $k = 0, 1$, for a moving direction for the tape head where $D_0 = \text{L}$ and $D_1 = \text{R}$. For each equation $\delta(q_i, X_j) = (q_k, X_\ell, D_h)$, we encode it by the following string in $\{0, 1\}^*$:

$$0^{i+1} 10^{j+1} 10^{k+1} 10^{\ell+1} 10^{h+1}. \quad (1.2)$$

Assume that there are m equations in the definition of δ . Let $code_i$ be the code of the i th equation. Then we combine the codes for equations together to get the following code for the TM:

$$code_1 11 code_2 11 \cdots 11 code_m. \quad (1.3)$$

Note that because different orderings of the equations give different codes, there are $m!$ equivalent codes for a TM of m equations.

The above coding system is a one-to-many mapping ϕ from TMs to $\{0, 1\}^*$. Each string x in $\{0, 1\}^*$ encodes at most one TM $\phi^{-1}(x)$. Let us extend ϕ^{-1} into a function mapping each string $x \in \{0, 1\}^*$ to a TM M by mapping each x not encoding a TM to a fixed empty TM M_0 whose code is λ and that rejects all strings. Call this mapping ι . Observe that ι is a mapping from $\{0, 1\}^*$ to TMs with the following properties:

- (i) For every $x \in \Sigma^*$, $\iota(x)$ represents a TM;
- (ii) Every TM is represented by at least one $\iota(x)$; and
- (iii) The transition function δ of the TM $\iota(x)$ can be easily decoded from the string x .

We say a coding system ι is an *enumeration* of one-tape DTMs if ι satisfies properties (i) and (ii). In addition, property (iii) means that this



enumeration admits a *universal Turing machine*. In the following, we write M_x to mean the TM $\iota(x)$. We assume that $\langle \cdot, \cdot \rangle$ is a pairing function on $\{0, 1\}^*$ such that both the function and its inverse are computable in linear time, for instance, $\langle x, y \rangle = 0^{|x|}1xy$.

Proposition 1.17 *There exists a TM M_u , which, on input $\langle x, y \rangle$, simulates the machine M_x on input y so that $L(M_u) = \{\langle x, y \rangle : y \in L(M_x)\}$. Furthermore, for each x , there is a constant c such that $\text{time}_{M_u}(\langle x, y \rangle) \leq c \cdot (\text{time}_{M_x}(y))^2$.*

Proof. We first construct a four-tape machine M_u . The machine M_u first decodes $\langle x, y \rangle$ and copies the string x to the second tape and copies string y to the third tape. After this, the machine M_u checks that the first string x is a legal code for a TM, that is, it verifies that it is of the form (1.3) such that each code_i is of the form (1.2) with $0 \leq j \leq 2$, $0 \leq \ell \leq 2$ and $0 \leq h \leq 1$. Furthermore, it verifies that it encodes a *deterministic* TM by verifying that no two code_p and code_q begin with the same initial segment $0^{i+1}10^{i+1}1$. If x does not legally encode a DTM, then M_u halts and rejects the input. Otherwise, M_u begins to simulate M_x on input y . During the simulation, the machine M_u stores the current state q_i (in the form 0^{i+1}) of M_x in tape 4 and, for each move, it looks on tape 2 for the equation of the transition function of M_x that applies to the current configuration on tape 3 and then simulates the move of M_x on tapes 3 and 4.

To analyze the running time of M_u , we note that decoding of $\langle x, y \rangle$ into x and y takes only time $c_1(|x| + |y|)$ for some constant c_1 , and the verification of x being a legal code for a DTM takes only $O(|x|^2)$ moves. In addition, the simulation of each move takes only time linearly proportional to the length of x . Thus, the total simulation time of M_u on $\langle x, y \rangle$ is only $c_2 \cdot \text{time}_{M_x}(y)$ for some constant c_2 that depends on x only. The proposition now follows from Theorem 1.14. ■

Although the above coding system and the universal TM are defined for a special class of one-tape TMs, it is not hard to see how to extend it to the class of multi-tape TMs. For instance, we note that a universal TM exists for the enumeration of one-worktape TMs and its simulation can be done in linear space.

Proposition 1.18 *Assume that $\{M_x\}$ is an enumeration of one-worktape TMs over the alphabet $\{0, 1, \mathbf{B}\}$. Then, there exists a one-worktape TM M_u such that $L(M_u) = \{\langle x, y \rangle : y \in L(M_x)\}$. Furthermore, for each x , there is a constant c such that $\text{space}_{M_u}(\langle x, y \rangle) \leq c \cdot \text{space}_{M_x}(y)$.*

Proof. Follows from Corollary 1.16. ■



The above results can be extended to other types of TMs. For instance, for each $k > 1$ and any fixed alphabet Σ , we may easily extend our coding system to get a one-to-one mapping from $\{0, 1\}^*$ to k -tape TMs over Σ and obtain an enumeration $\{M_x\}$ for such TMs. Based on Theorem 1.15 and Corollary 1.16, it can be easily seen that for this enumeration system, a universal TM M_u exists that simulates each machine with $time_{M_u}(\langle x, y \rangle) \leq c \cdot time_{M_x}(y) \cdot \log(time_{M_x}(y))$ and $space_{M_u}(\langle x, y \rangle) \leq c \cdot space_{M_x}(y)$ for some constant c (depending on k , Σ , and x). In addition, this enumeration can be extended to NTMs.

Proposition 1.19 *There exist a mapping from each string $x \in \{0, 1\}^*$ to a multi-tape NTM M_x and a universal NTM M_u , such that*

- (i) $L(M_u) = \{\langle x, y \rangle : y \in L(M_x)\}$; and
- (ii) For each x , there is a polynomial function p such that $time_{M_u}(\langle x, y \rangle) \leq p(time_{M_x}(y))$ for all y .

Proof. The proof is essentially the same as Proposition 1.17, together with the polynomial-time simulation of multi-tape NTMs by two-tape NTMs. We omit the details. ■

When we consider the complexity classes like P and NP , we are concerned only with a subclass of DTMs or NTMs. It is convenient to enumerate only these machines. We present an enumeration of polynomial-time “clocked” machines. First, we need some notations. A function $t : \mathbb{N} \rightarrow \mathbb{N}$ is called a *fully time-constructible* function if there is a multi-tape DTM that on any input x of length n halts in exactly $t(n)$ moves. Most common integer-valued functions such as cn , $n \cdot \lceil \log n \rceil$, n^c , $n^{\lceil \log n \rceil}$, 2^n , and $n!$ are known to be fully time constructible (see Exercise 1.20). A DTM that halts in exactly $t(n)$ moves on all inputs of length n is called a $t(n)$ -clock machine. For any k_1 -tape DTM M_1 and any fully time-constructible function t for which there exists a k_2 -tape clock machine M_2 , we can construct a $t(n)$ -clocked $(k_1 + k_2)$ -tape DTM M_3 that simulates M_1 for exactly $t(n)$ moves on inputs of length n . The machine M_3 first copies the input to the $(k_1 + 1)$ th tape and then simultaneously simulates on the first k_1 tapes machine M_1 and on the last k_2 tapes the $t(n)$ -clock machine M_2 . M_3 halts when either M_1 or M_2 halts, and it accepts only if M_1 accepts. The two moves of M_1 and M_2 can be combined into one move by using a state set that is the product of the state sets of M_1 and M_2 . For instance, assume that M_1 is a one-tape DTM defined by $(Q_1, q_0^1, F_1, \Sigma_1, \Gamma_1, \delta_1)$ and M_2 is a one-tape machine defined by $(Q_2, q_0^2, F_2, \Sigma_2, \Gamma_2, \delta_2)$. Then, M_3 is defined to be $(Q_1 \times Q_2, \langle q_0^1, q_0^2 \rangle, F_1 \times F_2, \Sigma_1 \times \Sigma_2, \Gamma_1 \times \Gamma_2, \delta_3)$, where

$$\delta_3(\langle q_i^1, q_j^2 \rangle, \langle s_i^1, s_j^2 \rangle) = (\langle q_k^1, q_{k'}^2 \rangle, \langle s_{\ell}^1, s_{\ell'}^2 \rangle, \langle D_h, D_{h'} \rangle)$$



if $\delta_1(q_i^1, s_j^1) = (q_k^1, s_{\ell'}^1, D_h)$ and $\delta_2(q_i^2, s_j^2) = (q_{k'}^2, s_{\ell'}^2, D_{h'})$. It is clear that machine M_3 always halts in time $t(n) + 2n$. Furthermore, if $t_{M_1}(n) \leq t(n)$ for all $n \geq 0$, then $L(M_2) = L(M_1)$.

Now, consider the functions $q_i(n) = n^i + i$, for $i \geq 1$. It is easy to see that for any polynomial function p , there exists an integer $i \geq 1$ such that $p(n) \leq q_i(n)$ for all $n \geq 0$. Assume that $\{M_x\}$ is an enumeration of DTMs as described above. For each string $x \in \{0, 1\}^*$ and each integer i , let $M_{\langle x, i \rangle}$ denote the machine M_x attached with the $q_i(n)$ -clock machine. Then, $\{M_{\langle x, i \rangle}\}$ is an enumeration of all polynomial-time clocked machines. This enumeration has the following properties and is called an effective enumeration of languages in P :

- (i) Every machine $M_{\langle x, i \rangle}$ in the enumeration accepts a set in P .
- (ii) Every set A in P is accepted by at least one machine in the enumeration (in fact, an infinite number of machines).
- (iii) There exists a universal machine M_u such that for given input $\langle x, i, y \rangle$, M_u simulates $M_{\langle x, i \rangle}$ on y in time $p(q_i(|y|))$, where p is a polynomial function depending on x only.

Note that the above statements claim that the class P of polynomial-time computable languages are enumerable through the enumeration of polynomial-time clocked machines. They do not claim that we can enumerate *all* polynomial-time-bounded machines, because the question of determining whether a given machine always halts in polynomial time is in fact not decidable (Exercise 1.23).

By a similar setting, we obtain an enumeration of clocked NTMs that accept exactly the languages in NP .

We can also enumerate all languages in $PSPACE$ through the enumeration of polynomial-space-marking machines. A function $s : \mathbb{N} \rightarrow \mathbb{N}$ is called a *fully space-constructible* function if there is a two-tape DTM that on any input x of length n halts visiting exactly $s(n)$ squares of the work tape. Again, most common functions, including $\lceil \log n \rceil$, $\lceil \log^2 n \rceil$, cn , n^c , are fully space constructible. For any fully space-constructible function s , we can construct a two-tape $s(n)$ -space marking machine M that, on any input of length n , places the symbol $\#$ on exactly $s(n)$ squares on the work tape. For any k -tape DTM M_1 , we can attach this $s(n)$ -space marking machine to it to form a new machine M_2 that does the following:

- (1) It first reads the input and simulates the marking machine M_1 to mark each tape with $s(n)$ $\#$ symbols.
- (2) It then simulates M_1 using these marked tapes, treating each $\#$ symbol as the blank symbol, such that whenever the machine M_1 sees a real blank symbol \mathbf{B} , the machine M_2 halts and rejects. (We assume that M_2 begins step (2) with each tape head scanning the square of the leftmost $\#$ mark. It can be proved that for any machine M_1 that



operates within a space bound $s(n)$, there is a machine M'_1 that computes the same function and operates in space bound $s(n)$ starting with this initial setting (Exercise 1.24.)

We can then enumerate all polynomial-space-marked machines as we did for polynomial-clocked machines. This enumeration effectively enumerates all languages in *PSPACE*.

1.6 Diagonalization

Diagonalization is an important proof technique widely used in recursive function theory and complexity theory. One of the earliest applications of diagonalization is Cantor's proof for the fact that the set of real numbers is not countable. We give a similar proof for the set of functions on $\{0, 1\}^*$. A set S is *countable* (or, *enumerable*) if there exists a one-one onto mapping from the set of natural numbers to S .

Proposition 1.20 *The set of functions from $\{0, 1\}^*$ to $\{0, 1\}$ is not countable.*

Proof. Suppose, by way of contradiction, that such a set is countable, that is, it can be represented as $\{f_0, f_1, f_2, \dots\}$. Let a_i denote the i th string in $\{0, 1\}^*$ under the lexicographic ordering. Then we can define a function f as follows: For each $i \geq 0$, $f(a_i) = 1$ if $f_i(a_i) = 0$ and $f(a_i) = 0$ if $f_i(a_i) = 1$. Clearly, f is a function from $\{0, 1\}^*$ to $\{0, 1\}$. However, it is not in the list f_0, f_1, f_2, \dots , because it differs from each f_i on at least one input string a_i . This establishes a contradiction. ■

An immediate consequence of Proposition 1.20 is that there exists a noncomputable function from $\{0, 1\}^*$ to $\{0, 1\}$, because we have just shown that the set of all TMs, and hence the set of all computable functions, is countable. In the following, we use diagonalization to construct directly an undecidable (i.e., nonrecursive) problem: the *halting problem*. The halting problem is the set $K = \{x \in \{0, 1\}^* : M_x \text{ halts on } x\}$, where $\{M_x\}$ is an enumeration of TMs.

Theorem 1.21 *K is r.e. but not recursive.*

Proof. The fact that K is r.e. follows immediately from the existence of the universal TM M_u (Proposition 1.17). To see that K is not recursive, we note that the complement of a recursive set is also recursive and, hence, r.e. Thus, if K were recursive, then \bar{K} would be r.e. and there would be an integer y such that M_y halts on all $x \in \bar{K}$ and does not halt on any $x \in K$. Then, a contradiction could be found when we consider whether or not y itself is in K : if $y \in K$, then M_y must not halt on y and it follows from



the definition of K that $y \notin K$ and if $y \notin K$, then M_y must halt on y and it follows from the definition of K that $y \in K$. ■

Now we apply the diagonalization technique to separate complexity classes. We first consider deterministic space-bounded classes. From the tape compression theorem, we know that $DSPACE(s_1(n)) = DSPACE(s_2(n))$ if $c_1 \cdot s_1(n) \leq s_2(n) \leq c_2 \cdot s_1(n)$ for some constants c_1 and c_2 . The following theorem shows that if $s_2(n)$ asymptotically diverges faster than $c \cdot s_1(n)$ for all constants c , then $DSPACE(s_1(n)) \neq DSPACE(s_2(n))$.

Theorem 1.22 (Space Hierarchy Theorem) *Let $s_2(n)$ be a fully space-constructible function. Suppose that $\liminf_{n \rightarrow \infty} s_1(n)/s_2(n) = 0$ and $s_2(n) \geq s_1(n) \geq \log_2 n$. Then, $DSPACE(s_1(n)) \subsetneq DSPACE(s_2(n))$.*

Proof. We are going to construct a TM M^* that works within space $s_2(n)$ to diagonalize against all TMs M_x that use space $s_1(n)$. To set up for the diagonalization process, we modify the coding of the TMs a little. Each set A in $DSPACE(s_1(n))$ is computed by a multi-tape TM within space bound $s_1(n)$. By Corollary 1.16, together with a simple coding of any alphabet Γ by the fixed alphabet $\{0, 1\}$, there is a one-worktape TM over the alphabet $\{0, 1, \mathbf{B}\}$ that accepts A in space $c_1 \cdot s_1(n)$ for some constant $c_1 > 0$. So, it is sufficient to consider the enumeration $\{M_x\}$ of one-worktape TMs over the alphabet $\{0, 1, \mathbf{B}\}$ as defined in Section 1.5. Recall that each TM was encoded by a string of the form (1.3). We now let all strings of the form $1^k x$, $k \geq 0$, encode M_x if x is of the form (1.3); we still let all other illegal strings y encode the fixed empty TM M_0 . Thus, for each TM M , there now are infinitely many strings that encode it.

Our machine M^* is a two-worktape TM that uses the tape alphabet $\{0, 1, \mathbf{B}, \#\}$. On each input x , M^* does the following:

- (1) M^* simulates a $s_2(n)$ -space marking TM to place $s_2(n)$ $\#$ symbols on each work tape.
- (2) Let $t_2(n) = 2^{s_2(n)}$. M^* writes the integer $t_2(|x|)$, in the binary form, on tape 2. That is, M^* writes 1 to the left of the leftmost $\#$ symbol and writes 0 over all $\#$'s.
- (3) M^* simulates M_x on input x on tape 1, one move at a time. For each move of M_x , if M_x attempts to move off the $\#$ symbols, then M^* halts and rejects; otherwise, M^* subtracts one from the number on tape 2. If tape 2 contains the number 0, then M^* halts and accepts.
- (4) If M_x halts on x within space $s_2(|x|)$ and time $t_2(|x|)$, then M^* halts, and M^* accepts x if and only if M_x rejects x .

It is clear that M^* works within space $2s_2(n) + 1$, and so, by the tape compression theorem, $L(M^*) \in DSPACE(s_2(n))$. Now we claim



that $L(M^*) \notin DSPACE(s_1(n))$. To see this, assume by way of contradiction that $L(M^*)$ is accepted by a one-worktape TM M that works in space $c_1 s_1(n)$ and uses the tape alphabet $\{0, 1, \mathbf{B}\}$ for some constant $c_1 > 0$. Then, there is a sufficiently large x such that $L(M_x) = L(M^*)$, $c_1 s_1(|x|) \leq s_2(|x|)$, and $t_1(|x|) \leq t_2(|x|)$, where $t_1(n) = c_1 \cdot n^2 \cdot s_1(n) \cdot 3^{c_1 s_1(n)}$. (Note that $s_1(n) \geq \log n$ and $\liminf_{n \rightarrow \infty} s_1(n)/s_2(n) = 0$, and so such an x exists.) Consider the simulation of M_x on x by M^* . As $c_1 \cdot s_1(|x|) \leq s_2(|x|)$, M^* on x always works within the $\#$ marks.

Case 1. $M_x(x)$ halts in $t_2(|x|)$ moves. Then, M^* accepts x if and only if M_x rejects x . Thus, $x \in L(M^*)$ if and only if $x \notin L(M_x)$. This is a contradiction.

Case 2. $M_x(x)$ does not halt in $t_2(|x|)$ moves. As $M_x(x)$ uses only space $c_1 s_1(|x|)$, it may have at most $r \cdot c_1 \cdot s_1(|x|) \cdot |x| \cdot 3^{c_1 s_1(|x|)}$ different configurations, where r is the number of states of M_x . (The value $s_1(|x|)$ is the number of possible tape head positions for the worktape, $|x|$ is the number of possible tape head position for the input tape, and $3^{c_1 s_1(|x|)}$ is the number of possible tape configurations.) As $r \leq |x|$, we know that this value is bounded by $t_2(|x|)$. So, if M_x does not halt in $t_2(|x|)$ moves, then it must have reached a configuration twice already. For a DTM, this implies that it loops forever, and so $x \notin L(M_x)$. But M^* accepts x in this case, and again this is a contradiction. ■



For the time-bounded classes, we show a weaker result.

Theorem 1.23 (Time Hierarchy Theorem) *If t_2 is a fully time-constructible function, $t_2(n) \geq t_1(n) \geq n$ and*

$$\liminf_{n \rightarrow \infty} \frac{t_1(n) \log(t_1(n))}{t_2(n)} = 0,$$

then $DTIME(t_1(n)) \subsetneq DTIME(t_2(n))$.

Proof. The proof is similar to the space hierarchy theorem. The only thing that needs extra attention is that the machines M_x may have an arbitrarily large number of tapes over an arbitrarily large alphabet, while the new machine M^* has only a fixed number of tapes and a fixed-size alphabet. Thus, the simulation of M_x on x by M^* may require extra time. This problem is resolved by considering the enumeration $\{M_x\}$ of two-worktape TMs over a fixed alphabet $\{0, 1, \mathbf{B}\}$ and relaxing the time bound to $O(t_1(n) \log(t_1(n)))$ (Theorem 1.15). ■

A nondecreasing function $f(n)$ is called *superpolynomial* if

$$\liminf_{n \rightarrow \infty} \frac{n^i}{f(n)} = 0$$



for all $i \geq 1$. For instance, the exponential function 2^n is superpolynomial, and the functions $2^{\log^k n}$, for $k > 1$, are also superpolynomial.

Corollary 1.24 $P \subsetneq \text{DTIME}(f(n))$ for all superpolynomial functions f .

Proof. If $f(n)$ is superpolynomial, then so is $g(n) = (f(n))^{1/2}$. By the time hierarchy theorem, $P \subseteq \text{DTIME}(g(n)) \subsetneq \text{DTIME}(f(n))$. ■

Corollary 1.25 $PSPACE \subsetneq DSPACE(f(n))$ for all superpolynomial functions f .

It follows that $P \subsetneq EXP$ and $LOGSPACE \subsetneq PSPACE \subsetneq EXPSPACE$.

The above time and space hierarchy theorems can also be extended to nondeterministic time- and space-bounded complexity classes. The proofs, however, are more involved, because acceptance and rejection in NTMs are not symmetric. We only list the simpler results on nondeterministic space-bounded complexity classes, which can be proved by a straightforward diagonalization. For the nondeterministic time hierarchy, see Exercise 1.28.

Theorem 1.26 (a) $NLOGSPACE \subsetneq NPSPACE$.
(b) For $k \geq 1$, $NSPACE(n^k) \subsetneq NSPACE(n^{k+1})$.

Proof. See Exercise 1.27. ■

In addition to the diagonalization technique, some other proof techniques for separating complexity classes are known. For instance, the following result separating the classes EXP and $PSPACE$ is based on a closure property of $PSPACE$ that does not hold for EXP . Unfortunately, this type of indirect proof techniques is not able to resolve the question of whether $PSPACE \subseteq EXP$ or $EXP \subseteq PSPACE$.

Theorem 1.27 $EXP \neq PSPACE$.

Proof. From the time hierarchy theorem, we get $EXP \subseteq \text{DTIME}(2^{n^{3/2}}) \subsetneq \text{DTIME}(2^{n^2})$. Thus, it suffices to show that if $PSPACE = EXP$, then $\text{DTIME}(2^{n^2}) \subseteq PSPACE$.

Assume that $L \in \text{DTIME}(2^{n^2})$. Let $\$$ be a symbol not used in L , and let $L' = \{x \$^t : x \in L, |x| + t = |x|^2\}$. Clearly, $L' \in \text{DTIME}(2^n)$. So, by the assumption that $PSPACE = EXP$, we have $L' \in PSPACE$, that is, there exists an integer $k > 0$ such that $L' \in DSPACE(n^k)$. Let M be a DTM accepting L' with the space bound n^k . We can construct a new DTM M' that operates as follows.



On input x , M' copies x into a tape and then adds $|x|^2 - |x|$ '\$'s.
Then, M' simulates M on x $\$^{|x|^2 - |x|}$.

Clearly, $L(M') = L$. Note that M' uses space n^{2k} , and so $L \in PSPACE$.
Therefore, $DTIME(2^{n^2}) \subseteq PSPACE$, and the theorem is proven. ■

1.7 Simulation

We study, in this section, the relationship between deterministic and nondeterministic complexity classes, as well as the relationship between time- and space-bounded complexity classes. We show several different simulations of nondeterministic machines by deterministic ones.

Theorem 1.28 (a) For any fully space-constructible function $f(n) \geq n$,

$$DTIME(f(n)) \subseteq NTIME(f(n)) \subseteq DSPACE(f(n)).$$

(b) For any fully space-constructible function $f(n) \geq \log n$,

$$DSPACE(f(n)) \subseteq NSPACE(f(n)) \subseteq \bigcup_{c>0} DTIME(2^{cf(n)}).$$

Proof. (a): The relation $DTIME(f(n)) \subseteq NTIME(f(n))$ follows immediately from the fact that DTMs are just a subclass of NTMs. For the relation $NTIME(f(n)) \subseteq DSPACE(f(n))$, we recall the simulation of an NTM M by a DTM M_1 as described in Theorem 1.9. Suppose that M has time complexity bounded by $f(n)$; then M_1 needs to simulate M for at most $f(n)$ moves. That is, we restrict M_1 to only execute the first $\sum_{i=1}^{f(n)} k^i$ stages such that the strings written in tape 2 are at most $f(n)$ symbols long. As $f(n)$ is fully space constructible, this restriction can be done by first marking off $f(n)$ squares on tape 2. It is clear that such a restricted simulation works within space $f(n)$.

(b): Again, $DSPACE(f(n)) \subseteq NSPACE(f(n))$ is obvious. To show that $NSPACE(f(n)) \subseteq \bigcup_{c>0} DTIME(2^{cf(n)})$, assume that M is an NTM with the space bound $f(n)$. We are going to construct a DTM M_1 to simulate M in time $2^{cf(n)}$ for some $c > 0$. As M uses only space $f(n)$, there is a constant $c_1 > 0$ such that the shortest accepting computation for each $x \in L(M)$ is of length $\leq 2^{c_1 f(|x|)}$. Thus, the machine M_1 needs only to simulate $M(x)$ for, at most, $2^{c_1 f(n)}$ moves. However, M is a nondeterministic machine and so its computation tree of depth $2^{c_1 f(n)}$ could have $2^{O(f(n))}$ leaves, and the naive simulation as (a) above takes too much time.

To reduce the deterministic simulation time, we notice that this computation tree, although of size $2^{2^{O(f(n))}}$, has at most $2^{O(f(n))}$ different configurations: Each configuration is determined by at most $f(n)$ tape symbols on the work tape, one of $f(n)$ positions for the work tape head,



one of n positions for the input tape head, and one of r positions for states, where r is a constant. Thus, the total number of possible configurations of $M(x)$ is $2^{O(f(n))} \cdot f(n) \cdot n \cdot r = 2^{O(f(n))}$. (Note that $f(n) \geq \log n$ implies $n \leq 2^{f(n)}$.)

Let T be the computation tree of $M(x)$ with depth $2^{c_f f(n)}$, with each node labeled by its configuration. We define a breadth-first ordering $<$ on the nodes of T and prune the subtree of T rooted at a node v as long as there is a node $u < v$ that has the same configuration as v . Then, the resulting pruned tree T' has at most $2^{O(f(n))}$ internal nodes and hence is of size $2^{O(f(n))}$. In addition, the tree T' contains an accepting configuration if and only if $x \in L(M)$, because all deleted subtrees occur somewhere else in T' . In other words, our DTM M_1 works as follows: it simulates $M(x)$ by making a breadth-first traversal over the tree T , keeping a record of the configurations encountered so far. When it visits a new node v of the tree, it checks the history record to see if the configuration has occurred before and prunes the subtree rooted at v if this is the case. In this way, M_1 only visits the nodes in tree T' and works within time $2^{c_f f(n)}$ for some constant $c > 0$. ■

Corollary 1.29 $LOGSPACE \subseteq NLOGSPACE \subseteq P \subseteq NP \subseteq PSPACE$.

Next we consider the space complexity of the deterministic simulation of nondeterministic space-bounded machines.

Theorem 1.30 (Savitch's Theorem) *For any fully space-constructible function $s(n) \geq \log n$, $NSPACE(s(n)) \subseteq DSPACE((s(n))^2)$.*

Proof. Let M be a one-worktape $s(n)$ -space-bounded NTM. We are going to construct a DTM M_1 to simulate M using space $(s(n))^2$. Without loss of generality, we may assume that M has a unique accepting configuration for all inputs x , that is, we require that M cleans up the worktape and enters a fixed accepting state when it accepts an input x . As pointed out in the proof of Theorem 1.28, the shortest accepting computation of M on an $x \in L(M)$ is at most $2^{O(s(n))}$, and each configuration is of length $s(n)$. (We say a configuration is of length ℓ if its work tape has at most ℓ nonblank symbols. Note that there are only $2^{O(\ell)}$ configurations on input x that has length ℓ .) Thus, for each input x , the goal of the DTM M_1 is to search for a computation path of length at most $2^{O(s(n))}$ from the initial configuration α_0 to the accepting configuration α_f .

Define a predicate $reach(\beta, \gamma, k)$ to mean that both β and γ are configurations of M such that γ is *reachable* from β in at most k moves, that is, there exists a sequence $\beta = \beta_0, \beta_1, \dots, \beta_k = \gamma$ such that $\beta_i \vdash_M \beta_{i+1}$ or $\beta_i = \beta_{i+1}$, for each $i = 0, \dots, k-1$. Using this definition, we see that M accepts x if and only if $reach(\alpha_0, \alpha_f, 2^{cs(n)})$, where α_0 is the unique initial configuration of M on input x , α_f is the unique accepting configuration of



M on x , and $2^{cs(n)}$ is the number of possible configurations of M of length $s(n)$. The following is a recursive algorithm computing the predicate *reach*. The main observation here is that

$$\text{reach}(\alpha_1, \alpha_2, j+k) \iff (\exists \alpha_3) [\text{reach}(\alpha_1, \alpha_3, j) \text{ and } \text{reach}(\alpha_3, \alpha_2, k)].$$

Algorithm for reach(α_1, α_2, i):

First, if $i \leq 1$, then return TRUE if and only if $\alpha_1 = \alpha_2$ or

$\alpha_1 \vdash_M \alpha_2$.

If $i \geq 2$, then for all possible configurations α_3 of M of length $s(n)$, recursively compute whether it is true that *reach*($\alpha_1, \alpha_3, \lceil i/2 \rceil$) and *reach*($\alpha_3, \alpha_2, \lfloor i/2 \rfloor$); return TRUE if and only if there exists such an α_3 .

It is clear that the algorithm is correct. This recursive algorithm can be implemented by a standard nonrecursive simulation using a stack of depth $O(s(n))$, with each level of the stack using space $O(s(n))$ to keep track of the current configuration α_3 . Thus the total space used is $O((s(n))^2)$. (See Exercise 1.34 for the detail.) ■

Corollary 1.31 $PSPACE = NPSPACE$.



For any complexity class C , let *co C* (or, *co-C*) denote the class of complements of sets in C , that is, $\text{co } C = \{S : \bar{S} \in C\}$, where $\bar{S} = \Sigma^* - S$, and Σ is the smallest alphabet such that $S \subseteq \Sigma^*$. One of the differences between deterministic and nondeterministic complexity classes is that deterministic classes are closed under complementation, but this is not known to be true for nondeterministic classes. For instance, it is not known whether $NP = \text{co}NP$. The following theorem shows that for most interesting nondeterministic space-bounded classes C , $\text{co } C = C$.



Theorem 1.32 For any fully space-constructible function $s(n) \geq \log n$, $NPSPACE(s(n)) = \text{co}NPSPACE(s(n))$.

Proof. Let M be a one-worktape NTM with the space bound $s(n)$. Recall the predicate *reach*(α, β, k) defined in the proof of Theorem 1.30. In this proof, we further explore the concept of reachable configurations from a given configuration of length $s(n)$. Consider a fixed input x of length n . Let C_x be the class of all configurations of M on x that is of length $s(n)$. Let $<$ be a fixed ordering of configurations in C_x such that an $O(s(n))$ space-bounded DTM can generate these configurations in the increasing order. First, we observe that the predicate *reach*(α, β, k) is acceptable by an NTM M_1 in space $O(s(n) + \log k)$ if α and β are from C_x . The NTM M_1 operates as follows:



Machine M_1 . Let $\alpha_0 = \alpha$. For each $i = 0, \dots, k-2$, M_1 guesses a configuration $\alpha_{i+1} \in C_x$ and verifies that $\alpha_i = \alpha_{i+1}$ or $\alpha_i \vdash_M \alpha_{i+1}$. (If neither $\alpha_i \neq \alpha_{i+1}$ nor $\alpha_i \vdash_M \alpha_{i+1}$ holds, then M_1 rejects on this computation path.) Finally, M_1 verifies that $\alpha_{k-1} = \beta$ or $\alpha_{k-1} \vdash_M \beta$ and accepts if this holds.

Apparently, this NTM M_1 uses space $O(s(n) + \log k)$ and accepts (α, β, k) if and only if $reach(\alpha, \beta, k)$.

Next, we apply this machine M_1 to construct another NTM M_2 that, on any given configuration β , computes the exact number N of configurations in C_x that are reachable from β , using space $O(s(n))$. (This is an NTM computing a function as defined in Definition 1.8.) Let m be the maximum length of an accepting path on input x . Then, $m = 2^{O(s(n))}$. The machine M_2 uses the following algorithm to compute iteratively the number N_k of configurations in C_x that are reachable from β in at most k moves, for $k = 0, \dots, m+1$. Then, when for some k it is found that $N_k = N_{k+1}$, it halts and outputs $N = N_k$.

Algorithm for computing N_k :

For $k = 0$, just let $N_0 = 1$ (β is the only reachable configuration). For each $k > 0$, assume that N_k has been found. To compute N_{k+1} , machine M_2 maintains a counter N_{k+1} , which is initialized to 0, and then for each configuration $\alpha \in C_x$, M_2 does the following:

- (1) First, let r_α be FALSE. For each $i = 1, \dots, N_k$, M_2 guesses a configuration γ_i in C_x , and verifies that (i) $\gamma_{i-1} < \gamma_i$ if $i > 1$ and (ii) $reach(\beta, \gamma_i, k)$ (this can be done by machine M_1). It rejects this computation path if (i) or (ii) does not hold. Next, it deterministically checks whether $reach(\gamma_i, \alpha, 1)$. If it is true that $reach(\gamma_i, \alpha, 1)$, then it sets the flag r_α to TRUE. In either case, it continues to the next i .
- (2) When the above is done for all $i = 1, \dots, N_k$ and if the computation does not reject, then M_2 adds one to N_{k+1} if and only if $r_\alpha = \text{TRUE}$ and goes to the next configuration.

Note that M_2 uses only space $O(s(n))$, because at each step corresponding to configuration α and integer i , it only needs to keep the following information: i, k, N_k , the current N_{k+1} , r_α , α , γ_{i-1} and γ_i . Furthermore, it can be checked that this algorithm indeed computes the function that maps β to the number N of reachable configurations: At stage $k+1$, assume that N_k has been correctly computed. Then, for each α , there exists one nonrejecting path in stage $(k+1)$ —the path that guesses the N_k configurations γ_i that are reachable from β in k moves, in the increasing



order. All other paths are rejecting paths. For each α , this unique path must determine whether $\text{reach}(\beta, \alpha, k + 1)$ correctly. (See Exercise 1.35 for more discussions.)

Next, we construct a third NTM M_3 for $\overline{L(M)}$ as follows:

Machine M_3 . First, M_3 simulates M_2 to compute the number N of reachable configurations from the initial configuration α_0 . Then, it guesses N configurations $\gamma_1, \dots, \gamma_N$, one by one and in the increasing order as in M_2 above, and checks that each is reachable from α_0 (by machine M_1) and none of them is an accepting configuration. It accepts if the above are checked; otherwise, it rejects this computation path.

We claim that this machine M_3 accepts $\overline{L(M)}$. First, it is easy to see that if $x \notin L(M)$, then all reachable configurations from α_0 are nonaccepting configurations. So, the computation path of M_3 that guesses correctly all N reachable configurations of α_0 will accept x . Conversely, if $x \in L(M)$, then one of the reachable configuration from α_0 must be an accepting configuration. So, a computation path of M_3 must guess either all reachable configurations that include one accepting configuration or guess at least one nonreachable configuration. In either case, this computation path must reject. Thus, M_3 accepts exactly those $x \notin L(M)$.

Finally, the same argument for M_2 verifies that M_3 uses space $O(s(n))$. The theorem then follows from the tape compression theorem for NTMs. ■

Recall that in the formal language theory, a language L is called *context-sensitive* if there exists a grammar G generating the language L with the right-hand side of each grammar rule in G being at least as long as its left-hand side. A well-known characterization for the context-sensitive languages is that the context-sensitive languages are exactly the class $NSPACE(n)$.

Corollary 1.33 (a) $NLOGSPACE = coNLOGSPACE$.

(b) *The class of context-sensitive languages is closed under complementation.*

The above results, together with the ones proved in the last section, are the best we know about the relationship among time/space-bounded deterministic/nondeterministic complexity classes. Many other important relations are not known. We summarize in Figure 1.7 the known relations among the most familiar complexity classes. More complexity classes between P and $PSPACE$ will be introduced in Chapters 3, 8, 9, and 10. Complexity classes between $LOGSPACE$ and P will be introduced in Chapter 6.



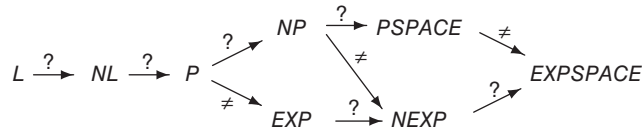


Figure 1.7 Inclusive relations among the complexity classes. We write L to denote $LOGSPACE$ and NL to denote $NLOGSPACE$. The label \neq means the proper inclusion. The label $?$ means the inclusion is not known to be proper.

Exercises

1.1 Let ℓ_1, \dots, ℓ_q be q natural numbers, and Σ be an alphabet of r symbols. Show that there exist q strings s_1, \dots, s_q over Σ , of lengths ℓ_1, \dots, ℓ_q , respectively, which form an alphabet if and only if $\sum_{i=1}^q r^{-\ell_i} \leq 1$.

1.2 (a) Let $c \geq 0$ be a constant. Prove that no pairing function $\langle \cdot, \cdot \rangle$ exists such that for all $x, y \in \Sigma^*$, $|\langle x, y \rangle| \leq |x| + |y| + c$.

(b) Prove that there exists a pairing function $\pi : \Sigma^* \times \Sigma^* \rightarrow \Sigma^*$ such that (i) for any $x, y \in \Sigma^*$, $|\pi(x, y)| \leq |x| + |y| + O(\log |x|)$, (ii) it is polynomial-time computable, and (iii) its inverse function is polynomial-time computable (if $z \notin \text{range}(\pi)$, then $\pi^{-1}(z) = (\lambda, \lambda)$).

(c) Let $f : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ be defined by $f(n, m) = (n + m)(n + m + 1)/2 + n$. Prove that f is one-one, onto, polynomial-time computable (with respect to the binary representations of natural numbers) and that f^{-1} is polynomial-time computable. (Therefore, f is an efficient pairing function on natural numbers.)

1.3 There are two cities T and F . The residents of city T always tell the truth and the residents of city F always lie. The two cities are very close. Their residents visit each other very often. When you walk in city T you may meet a person who came from city F , and vice versa. Now, suppose that you are in one of the two cities and you want to find out which city you are in. Also suppose that you are allowed to ask a person on the street for only one YES/NO question. What question should you ask? [Hint: You may first design a Boolean function f of two variables, the resident and the city, for your need, and then find a question corresponding to f .]

1.4 How many Boolean functions of n variables are there?

1.5 Assume that f is a Boolean function of more than two variables. Prove that for any minterm p of $f|_{x_1=0, x_2=1}$, there exists a minterm of f containing p as a subterm.

1.6 Design a multi-tape DTM M to accept the language $L = \{a^i b^j b^k : i, j, k \geq 0, i + j = k\}$. Show the computation paths of M on inputs $a^2 b a^3 b a^5$ and $a^2 b a b a^4$.

1.7 Design a multi-tape NTM M to accept the language $L = \{a^{i_1} b a^{i_2} b \cdots b a^{i_k} : i_1, i_2, \dots, i_k \geq 0, i_r = i_s \text{ for some } 1 \leq r < s \leq k\}$. Show

the computation tree of M on input $a^3ba^2ba^2ba^4ba^2$. What is the time complexity of M ?

1.8 Show that the problem of Example 1.7 can be solved by a DTM in polynomial time.

1.9 Give formal definitions of the configurations and the computation of a k -tape DTM.

1.10 Assume that M is a three-tape DTM using tape alphabet $\Sigma = \{a, b, B\}$. Consider the one-tape DTM M_1 that simulates M as described in Theorem 1.6. Describe explicitly the part of the transition function of M_1 that moves the tape head from left to right to collect the information of the current tape symbols scanned by M .

1.11 Let $C = \{\langle G, u, v \rangle : G = (V, E) \text{ is an undirected graph, } u, v \in V \text{ are connected}\}$. Design an NTM M that accepts set C in space $O(\log |V|)$. Can you find a DTM accepting C in space $O(\log |V|)$? (Assume that $V = \{v_1, \dots, v_n\}$. Then, the input to the machine M is $\langle 0^n 1 e_{11} e_{12} \dots e_{1n} e_{21} \dots e_{nn}, o^i, o^j \rangle$, where $e_{ij} = 1$ if $\{v_i, v_j\} \in E$ and $e_{ij} = 0$ otherwise.)

1.12 Prove that any finite set of strings belongs to $DTIME(n)$.

1.13 Estimate an upper bound for the number of possible computation histories $H(g, g_\ell, g_r)$ in the proof of Proposition 1.13.

1.14 A *random access machine* (RAM) is a machine model for computing integer functions. The memories of a RAM M consists of a one-way read-only input tape, a one-way write-only output tape, and an infinite number of registers named R_0, R_1, R_2, \dots . Each square of the input and output tapes and each register can store an integer of an arbitrary size. Let $c(R_i)$ denote the content of the register R_i . An instruction of a RAM may access a register R_i to get its content $c(R_i)$ or it may access the register $R_{c(R_i)}$ by the indirect addressing scheme. Each instruction has an integer label, beginning from 1. A RAM begins the computation on instruction with label 1 and halts when it reaches an empty instruction. The following table lists the instruction types of a RAM.

Instruction	Meaning
READ(R_i)	Read the next input integer into R_i
WRITE(R_i)	Write $c(R_i)$ on the output tape
COPY(R_i, R_j)	Write $c(R_i)$ to R_j
ADD(R_i, R_j, R_k)	Write $c(R_i) + c(R_j)$ to R_k
SUB(R_i, R_j, R_k)	Write $c(R_i) - c(R_j)$ to R_k
MULT(R_i, R_j, R_k)	Write $c(R_i) \cdot c(R_j)$ to R_k
DIV(R_i, R_j, R_k)	Write $\lfloor c(R_i)/c(R_j) \rfloor$ to R_k (write 0 if $c(R_j) = 0$)
GOTO(i)	Go to the instruction with label i
IF THEN(R_i, j)	If $c(R_i) \geq 0$, then go to the instruction with label j

In the above table, we only listed the arguments in the direct addressing scheme. It can be changed to indirect addressing scheme by changing the

argument R_i to R_i^* . For instance, $\text{ADD}(R_i, R_j^*, R_k)$ means to write $c(R_i) + c(R_{c(R_j)})$ to R_k . Each instruction can also use a constant argument i instead of R_j . For instance, $\text{COPY}(i, R_j^*)$ means to write integer i to the register $R_{c(R_j)}$.

(a) Design a RAM that reads inputs i_1, i_2, \dots, i_k, j and outputs 1 if $\sum_{r \in A} i_r = j$ for some $A \subseteq \{1, 2, \dots, k\}$ and outputs 0 otherwise (cf. Example 1.7).

(b) Show that for any TM M , there is a RAM M' that computes the same function as M . (You need to first specify how to encode tape symbols of the TM M by integers.)

(c) Show that for any RAM M , there is a TM M' that computes the same function as M (with the integer n encoded by the string a^n).

1.15 In this exercise, we consider the complexity of RAMs. There are two possible ways of defining the computational time of a RAM M on input x . The *uniform* time measure counts each instruction as taking one unit of time and so the total runtime of M on x is the number of times M executes an instruction. The *logarithmic* time measure counts, for each instruction, the number of bits of the arguments involved. For instance, the total time to execute the instruction $\text{MULT}(i, R_j^*, R_k)$ is $\lceil \log i \rceil + \lceil \log j \rceil + \lceil \log c(R_j) \rceil + \lceil \log c(R_{c(R_j)}) \rceil + \lceil \log k \rceil$. The total runtime of M on x , in the logarithmic time measure, is the sum of the instruction time over all instructions executed by M on input x .

Use both the uniform and the logarithmic time measures to analyze the simulations of parts (b) and (c) of Exercise 1.14. In particular, show that the notion of polynomial-time computability is equivalent between TMs and RAMs with respect to the logarithmic time measure.

1.16 Suppose that a TM is allowed to have infinitely many tapes. Does this increase its computation power as far as the class of computable sets is concerned?

1.17 Prove Blum's speed-up theorem. That is, find a function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ such that for any DTM M_1 computing f there exists another DTM M_2 computing f with $\text{time}_{M_2}(x) < \text{time}_{M_1}(x)$ for infinitely many $x \in \{0, 1\}^*$.

1.18 Prove that if $c > 1$, then for any $\epsilon > 0$,

$$\text{DTIME}(cn) = \text{DTIME}((1 + \epsilon)n).$$

1.19 Prove Theorem 1.15.

1.20 Prove that if $f(n)$ is an integer function such that (i) $f(n) \geq 2n$ and (ii) f , regarded as a function mapping a^n to $a^{f(n)}$, is computable in deterministic time $O(f(n))$, then f is fully time constructible. [Hint: Use the linear speed-up of Proposition 1.13 to compute the function f in less than $f(n)$ moves, with the output $a^{f(n)}$ compressed.]

1.21 Prove that if f is fully space constructible and f is not a constant function, then $f(n) = \Omega(\log \log n)$. (We say $g(n) = \Omega(h(n))$ if there exists a constant $c > 0$ such that $g(n) \geq c \cdot h(n)$ for almost all $n \geq 0$.)

1.22 Prove that $f(n) = \lceil \log n \rceil$ and $g(n) = \lceil \sqrt{n} \rceil$ are fully space constructible.

1.23 Prove that the question of determining whether a given TM halts in time $p(n)$ for some polynomial function p is undecidable.

1.24 Prove that for any one-worktape DTM M that uses space $s(n)$, there is a one-worktape DTM M' with $L(M') = L(M)$ such that it uses space $s(n)$ and its tape head never moves to the left of its starting square.

1.25 Recall that $\log^* n = \min\{k : k \geq 1, \log \log \dots \log n \text{ (} k \text{ iterations of log on } n) \leq 1\}$. Let $k > 0$ and t_2 be a fully time-constructible function. Assume that

$$\liminf_{n \rightarrow \infty} \frac{t_1(n) \log^*(t_1(n))}{t_2(n)} = 0.$$

Prove that there exists a language L that is accepted by a k -tape DTM in time $t_2(n)$ but not by any k -tape DTM in time $t_1(n)$.

1.26 Prove that $DTIME(n^2) \subsetneq DTIME(n^2 \log n)$.

1.27 (a) Prove Theorem 1.26 (cf. Theorems 1.30 and 1.32).

(b) Prove that for any integer $k \geq 1$ and any rational $\epsilon > 0$, $NSPACE(n^k) \subsetneq NSPACE(n^{k+\epsilon})$.

1.28 (a) Prove that $NP \subsetneq NEXP$.

(b) Prove that $NTIME(n^k) \subsetneq NTIME(n^{k+1})$ for any $k \geq 1$.

1.29 Prove that $P \neq DSPACE(n)$ and $NP \neq EXP$.

1.30 A set A is called a *tally set* if $A \subseteq \{a\}^*$ for some symbol a . A set A is called a *sparse set* if there exists a polynomial function p such that $|\{x \in A : |x| \leq n\}| \leq p(n)$ for all $n \geq 1$. Prove that the following are equivalent:

- (a) $EXP = NEXP$.
- (b) All tally sets in NP are actually in P .
- (c) All sparse sets in NP are actually in P .

(Note: The above implies that if $EXP \neq NEXP$ then $P \neq NP$.)

1.31 (BUSY BEAVER) In this exercise, we show the existence of an r.e., nonrecursive set without using the diagonalization technique. For each TM M , we let $|M|$ denote the length of the codes for M . We consider only one-tape TMs with a fixed alphabet $\Gamma = \{0, 1, \mathbf{B}\}$. Let $f(x) = \min\{|M| : M(\lambda) = x\}$ and $g(m) = \min\{x \in \{0, 1\}^* : |M| \leq m \Rightarrow M(\lambda) \neq x\}$. That is, $f(x)$ is the size of the minimum TM that outputs x on the input λ , and

$g(m)$ is the least x that is not printable by any TM M of size $|M| \leq m$ on the input λ . It is clear that both f and g are total functions.

(a) Prove that neither f nor g is a recursive function.

(b) Define $A = \{\langle x, m \rangle : (\exists M, |M| \leq m) M(\lambda) = x\}$. Apply part (a) above to show that A is r.e. but is not recursive.

1.32 (BUSY BEAVER, time-bounded version) In this exercise, we apply the busy beaver technique to show the existence of a set computable in exponential time but not in polynomial time.

(a) Let $g(0^m, 0^k)$ be the least $x \in \{0, 1\}^*$ that is not printable by any TM M of size $|M| \leq m$ on input λ in time $(m+k)^{\log k}$. It is easy to see that $|g(0^m, 0^k)| \leq m+1$, and hence, g is polynomial length bounded. Prove that $g(0^m, 0^k)$ is computable in time $2^{O(m+k)}$ but is not computable in time polynomial in $m+k$.

(b) Can you modify part (a) above to prove the existence of a function that is computable in subexponential time (e.g., $n^{O(\log n)}$) but is not polynomial-time computable?

1.33 Assume that an NTM computes the characteristic function χ_A of a set A in polynomial time, in the sense of Definition 1.8. What can you infer about the complexity of set A ?

1.34 In the proof of Theorem 1.30, the predicate *reach* was solved by a deterministic recursive algorithm. Convert it to a nonrecursive algorithm for the predicate *reach* that only uses space $O((s(n))^2)$.

1.35 What is wrong if we use the following simpler algorithm to compute N_k in the proof of Theorem 1.32?

For each k , to compute N_k , we generate each configuration $\alpha \in C_x$ one by one and, for each one, nondeterministically verify whether *reach*(β, α, k) (by machine M_1), and increments the counter for N_k by one if *reach*(β, α, k) holds.

1.36 In this exercise, we study the notion of computability of real numbers. First, for each $n \in \mathbb{N}$, we let *bin*(n) denote its binary expansion, and for each $x \in \mathbb{R}$, let *sgn*(x) = λ if $x \geq 0$, and *sgn*(x) = $-$ if $x < 0$. An integer $m \in \mathbb{Z}$ is represented as *sgn*(m)*bin*($|m|$). A rational number $r = \pm a/b$ with $a, b \in \mathbb{N}$, $b \neq 0$, and $\gcd(a, b) = 1$ has a unique representation over alphabet $\{-, 0, 1, /\}$: *sgn*(r)*bin*(a)/*bin*(b). We write \mathbb{N} (\mathbb{Z} and \mathbb{Q}) to denote both the set of natural numbers (integers and rational numbers, respectively) and the set of their representations.

We say a real number x is *Cauchy computable* if there exist two computable functions $f : \mathbb{N} \rightarrow \mathbb{Q}$ and $m : \mathbb{N} \rightarrow \mathbb{N}$ satisfying the property $C_{f,m}$: $n \geq m(k) \Rightarrow |f(n) - x| \leq 2^{-k}$. A real number x is *Dedekind computable* if the set $L_x = \{r \in \mathbb{Q} : r < x\}$ is computable. A real number x is *binary*



computable if there is a computable function $b_x : \mathbb{N} \rightarrow \mathbb{N}$, with the property $b_x(n) \in \{0, 1\}$ for all $n > 0$, such that

$$x = \text{sgn}_x \cdot \sum_{n \geq 0} b_x(n) 2^{-n},$$

where $\text{sgn}_x = 1$ if $x \geq 0$ and $\text{sgn}_x = -1$ if $x < 0$. (The function b_x is not unique for some x , but notice that for such x , both the functions b_x are computable.) A real number x is *Turing computable* if there exists a DTM M that on the empty input prints a binary expansion of x (i.e., it prints the string $b_x(0)$ followed by the binary point and then followed by the infinite string $b_x(1)b_x(2)\cdots$).

Prove that the above four notions of computability of real numbers are equivalent. That is, prove that a real number x is Cauchy computable if and only if it is Dedekind computable if and only if it is binary computable if and only if it is Turing computable.

1.37 In this exercise, we study the computational complexity of a real number. We define a real number x to be *polynomial-time Cauchy computable* if there exist a polynomial-time computable function $f : \mathbb{N} \rightarrow \mathbb{Q}$ and a polynomial function $m : \mathbb{N} \rightarrow \mathbb{N}$, satisfying $C_{f,m}$, where f being polynomial-time computable means that $f(n)$ is computable by a DTM in time $p(n)$ for some polynomial p (i.e., the input n is written in the unary form). We say x is *polynomial-time Dedekind computable* if L_x is in P . We say x is *polynomial-time binary computable* if b_x , restricted to inputs $n > 0$, is polynomial-time computable, assuming that the inputs n are written in the unary form.

Prove that the above three notions of polynomial-time computability of real numbers are not equivalent. That is, let P_C (P_D , and P_b) denote the class of polynomial-time Cauchy (Dedekind and binary, respectively) computable real numbers. Prove that $P_D \subsetneq P_b \subsetneq P_C$.

Historical Notes

McMillan's theorem is well known in coding theory; see, for example, Roman (1992). TMs were first defined by Turing (1936, 1937). The equivalent variations of TMs, the notion of computability, and the Church–Turing Thesis are the main topics of recursive function theory; see, for example, Rogers (1967) and Soare (1987). Kleene (1979) contains an interesting personal account of the history. The complexity theory based on TMs was developed by Hartmanis and Stearns (1965), Stearns, Hartmanis, and Lewis (1965) and Lewis, Stearns, and Hartmanis (1965). The tape compression theorem, the linear speed-up theorem, the tape-reduction simulations (Corollaries 1.14–1.16) and the time and space hierarchy theorems are from these works. A machine-independent complexity theory has been established by Blum (1967). Blum's speed-up



theorem is from there. Cook (1973a) first established the nondeterministic time hierarchy. Seiferas (1977a, 1977b) contain further studies on the hierarchy theorems for nondeterministic machines. The indirect separation result, Theorem 1.27, is from Book (1974a). The identification of P as the class of feasible problems was first suggested by Cobham (1964) and Edmonds (1965). Theorem 1.30 is from Savitch (1970). Theorem 1.32 was independently proved by Immerman (1988) and Szelepcsényi (1988). Hopcroft et al. (1977) and Paul et al. (1983) contain separation results between $DSPACE(n)$, $NTIME(O(n))$, and $DTIME(O(n))$. Context-sensitive languages and grammars are major topics in formal language theory; see, for example, Hopcroft and Ullman (1979).

RAMs were first studied in Shepherdson and Sturgis (1963) and Elgot and Robinson (1964). The improvements over the time hierarchy theorem, including Exercises 1.25 and 1.26, can be found in Paul (1979) and Fürer (1984). Exercise 1.30 is an application of the Translation Lemma of Book (1974b); see Hartmanis et al. (1983). The busy beaver problems (Exercises 1.31 and 1.32) are related to the notion of the Kolmogorov complexity of finite strings. The arguments based on the Kolmogorov complexity avoids the direct use of diagonalization. See Daley (1980) for discussions on the busy beaver proof technique, and Li and Vitányi (1997) for a complete treatment of the Kolmogorov complexity. Computable real numbers were first studied by Turing (1936) and Rice (1954). Polynomial-time computable real numbers were studied in Ko and Friedman (1982) and Ko (1991a).