FUTURE VISION BIE

One Stop for All Study Materials & Lab Programs

Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page

Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU, Currently for CSE – Computer Science Engineering...

Join Telegram to get Instant Updates: https://bit.ly/2GKiHnJ

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

Preface

The enlarged third edition of *Theory of Computer Science* is the result of the enthusiastic reception given to earlier editions of this book and the feedback received from the students and teachers who used the second edition for several years.

The new edition deals with all aspects of theoretical computer science, namely **automata, formal languages, computability** and **complexity**. Very few books combine all these theories and give adequate examples. This book provides numerous examples that illustrate the basic concepts. It is profusely illustrated with diagrams. While dealing with theorems and algorithms, the emphasis is on constructions. Each construction is immediately followed by an example and only then the formal proof is given so that the student can master the technique involved in the construction before taking up the formal proof.

The key feature of the book that sets it apart from other books is the provision of detailed solutions (at the end of the book) to chapter-end exercises.

The chapter on Propositions and Predicates (Chapter 10 of the second edition) is now the first chapter in the new edition. The changes in other chapters have been made without affecting the structure of the second edition. The chapter on Turing machines (Chapter 7 of the second edition) has undergone major changes.

A novel feature of the third edition is the addition of objective type questions in each chapter under the heading Self-Test. This provides an opportunity to the student to test whether he has fully grasped the fundamental concepts. Besides, a total number of 83 additional solved examples have been added as Supplementary Examples which enhance the variety of problems dealt with in the book.

of COPY. The TM M_1 performs the subroutine COPY. The following moves take place for M_1: $q_1 0^n 1 \vdash 2q_2 0^{n-1} 1 \overset{*}{\vdash} 20^{n-1} 1 q_3 b \vdash 20^{n-1} q_3 10 \overset{*}{\vdash} 2q_1 0^{n-1} 10$. After exhausting 0's, q_1 encounters 1. M_1 moves to state q_4. All 2's are converted back to 0's and M_1 halts in q_5. The TM M picks up the computation by starting from q_5. The q_0 and q_6 are the states of M. Additional states are created to check whether each 0 in 0^m gives rise to 0^m at the end of the rightmost 1 in the input string. Once this is over, M erases 10^n1 and finds 0^{nm} in the input tape.

M can be defined by

$$M = (\{q_0, q_1, \ldots, q_{12}\}, \{0, 1\}, \{0, 1, 2, b\}, \delta, q_0, b, \{q_{12}\})$$

where δ is defined by Table 9.8.

TABLE 9.8 Transition Table for Example 9.10

	0	1	2	b
q_0	q_6bR	—	—	—
q_6	q_60R	q_11R	—	—
q_5	q_70L	—	—	—
q_7	—	q_81L	—	—
q_8	q_90L	—	—	$q_{10}bR$
q_9	q_90L	—	—	q_0bR
q_{10}	—	$q_{11}bR$	—	—
q_{11}	$q_{11}bR$	$q_{12}bR$	—	—

Thus M performs multiplication of two numbers in unary representation.

9.7 VARIANTS OF TURING MACHINES

The Turing machine we have introduced has a single tape. $\delta(q, a)$ is either a single triple (p, y, D), where $D = R$ or L, or is not defined. We introduce two new models of TM:

(i) a TM with more than one tape

(ii) a TM where $\delta(q, a) = \{(p_1, y_1, D_1), (p_2, y_2, D_2), \ldots, (p_r, y_r, D_r)\}$. The first model is called a multitape TM and the second a nondeterministic TM.

9.7.1 MULTITAPE TURING MACHINES

A multitape TM has a finite set Q of states, an initial state q_0, a subset F of Q called the set of final states, a set P of tape symbols, a new symbol b, not in P called the blank symbol. (We assume that $\Sigma \subseteq \Gamma$ and $b \notin \Sigma$.)

There are k tapes, each divided into cells. The first tape holds the input string w. Initially, all the other tapes hold the blank symbol.

Initially the head of the first tape (input tape) is at the left end of the input w. All the other heads can be placed at any cell initially.

δ is a partial function from $Q \times \Gamma^k$ into $Q \times \Gamma^k \times \{L, R, S\}^k$. We use implementation description to define δ. Figure 9.8 represents a multitape TM. A move depends on the current state and k tape symbols under k tape heads.

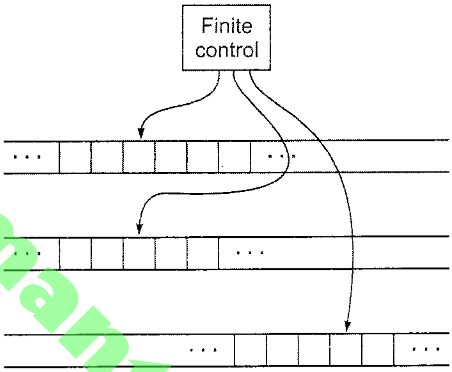

Fig. 9.8 Multitape Turing machine.

In a typical move:

(i) M enters a new state.
(ii) On each tape, a new symbol is written in the cell under the head.
(iii) Each tape head moves to the left or right or remains stationary. The heads move independently: some move to the left, some to the right and the remaining heads do not move.

The initial ID has the initial state q_0, the input string w in the first tape (input tape), empty strings of b's in the remaining $k - 1$ tapes. An accepting ID has a final state, some strings in each of the k tapes.

Theorem 9.1 Every language accepted by a multitape TM is acceptable by some single-tape TM (that is, the standard TM).

Proof Suppose a language L is accepted by a k-tape TM M. We simulate M with a single-tape TM with $2k$ tracks. The second, fourth, . . ., $(2k)$th tracks hold the contents of the k-tapes. The first, third, . . ., $(2k - 1)$th tracks hold a head marker (a symbol say X) to indicate the position of the respective tape head. We give an 'implementation description' of the simulation of M with a single-tape TM M_1. We give it for the case $k = 2$. The construction can be extended to the general case.

Figure 9.9 can be used to visualize the simulation. The symbols A_2 and B_5 are the current symbols to be scanned and so the headmarker X is above the two symbols.

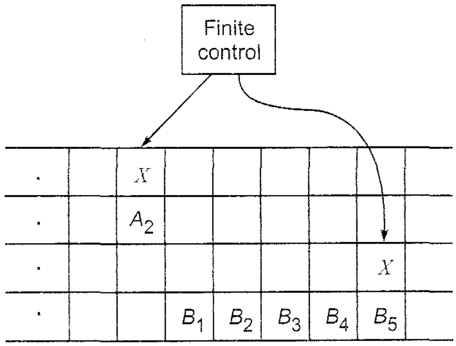

Fig. 9.9 Simulation of multitape TM.

Initially the contents of tapes 1 and 2 of M are stored in the second and fourth tracks of M_1. The headmarkers of the first and third tracks are at the cells containing the first symbol.

To simulate a move of M, the $2k$-track TM M_1 has to visit the two headmarkers and store the scanned symbols in its control. Keeping track of the headmarkers visited and those to be visited is achieved by keeping a count and storing it in the finite control of M_1. Note that the finite control of M_1 has also the information about the states of M and its moves. After visiting both head markers, M_1 knows the tape symbols being scanned by the two heads of M.

Now M_1 revisits each of the headmarkers:

(i) It changes the tape symbol in the corresponding track of M_1 based on the information regarding the move of M corresponding to the state (of M) and the tape symbol in the corresponding tape M.

(ii) It moves the headmarkers to the left or right.

(iii) M_1 changes the state of M in its control.

This is the simulation of a single move of M. At the end of this, M_1 is ready to implement its next move based on the revised positions of its headmarkers and the changed state available in its control.

M_1 accepts a string w if the new state of M, as recorded in its control at the end of the processing of w, is a final state of M.

Definition 9.3 Let M be a TM and w an input string. The running time of M on input w, is the number of steps that M takes before halting. If M does not halt on an input string w, then the running time of M on w is infinite.

Note: Some TMs may not halt on all inputs of length n. But we are interested in computing the running time, only when the TM halts.

Definition 9.4 The time complexity of TM M is the function $T(n)$, n being the input size, where $T(n)$ is defined as the maximum of the running time of M over all inputs w of size n.

Theorem 9.2 If M_1 is the single-tape *TM* simulating multitape TM M, then the time taken by M_1 to simulate n moves of M is $O(n^2)$.

Proof Let M be a k-tape TM. After n moves of M, the head markers of M_1 will be separated by $2n$ cells or less. (At the worst, one tape movement can be to the left by n cells and another can be to the right by n cells. In this case the tape headmarkers are separated by $2n$ cells. In the other cases, the 'gap' between them is less). To simulate a move of M, the TM M_1 must visit all the k headmarkers. If M starts with the leftmost headmarker, M_1 will go through all the headmarkers by moving right by at most $2n$ cells. To simulate the change in each tape, M_1 has to move left by at most $2n$ cells; to simulate changes in k tapes, it requires at most two moves in the reverse direction for each tape.

Thus the total number of moves by M_1 for simulating one move of M is atmost $4n + 2k$. ($2n$ moves to right for locating all headmarkers, $2n + 2k$ moves to the left for simulating the change in the content of k tapes.) So the number of moves of M_1 for simulating n moves of M is $n(4n + 2k)$. As the constant k is independent of n, the time taken by M_1 is $O(n^2)$.

9.7.2 NONDETERMINISTIC TURING MACHINES

In the case of standard Turing machines (hereafter we refer to this machine as deterministic TM), $\delta(q_1, a)$ was defined (for some elements of $Q \times \Gamma$) as an element of $Q \times \Gamma \times \{L, R\}$. Now we extend the definition of δ. In a nondeterministic TM, $\delta(q_1, a)$ is defined as a subset of $Q \times \Gamma \times \{L, R\}$.

Definition 9.5 A nondeterministic Turing machine is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, b, F)$ where

1. Q is a finite nonempty set of states
2. Γ is a finite nonempty set of tape symbols
3. $b \in \Gamma$ is called the blank symbol
4. Σ is a nonempty subset of Γ, called the set of input symbols. We assume that $b \notin \Sigma$.
5. q_0 is the initial state
6. $F \subseteq Q$ is the set of final states
7. δ is a partial function from $Q \times \Gamma$ into the power set of $Q \times \Gamma \times \{L, R\}$.

Note: If $q \in Q$ and $x \in \Gamma$ and $\delta(q, x) = \{(q_1, y_1, D_1), (q_2, y_2, D_2), \ldots, (q_n, y_n, D_n)\}$ then the NTM can chose any one of the actions defined by (q_i, y_i, D_i) for $i = 1, 2, \ldots, n$.

We can also express this in terms of $\vdash$ relation. If $\delta(q, x) = \{(q_i, y_i, D_i) | i = 1, 2, \ldots, n\}$ then the ID $zqxw$ can change to any one of the n IDs specified by the n-element set $\delta(q, x)$.

Suppose $\delta(q, x) = \{(q_1, y_1, L), (q_2, y_2, R), (q_3, y_3, L)\}$. Then

$$z_1z_2 \ldots z_kqxz_{k+1} \ldots z_n \vdash z_1z_2 \ldots z_{k-1}q_1z_ky_1z_{k+1} \ldots z_n$$

or

$$z_1z_2 \ldots z_kqxz_{k+1} \ldots z_n \vdash z_1z_2 \ldots z_ky_2q_2z_{k+1} \ldots z_n$$

or

$$z_1z_2 \ldots z_kqxz_{k+1} \ldots z_n \vdash z_1z_2 \ldots z_{k-1}q_3z_ky_3z_{k+1} \ldots z_n.$$

So on reading the input symbol, the NTM M whose current ID is $z_1 z_2 \ldots z_k q x z_{k+1} \ldots z_n$ can change to any one of the three IDs given earlier.

Remark When $\delta(q, x) = \{(q_i, y_i, D_i) \mid i = 1, 2, \ldots, n\}$ then NTM chooses any one of the n triples totally (that is, it cannot take a state from one triple, another tape symbol from a second triple and a third D(L or R) from a third triple, etc.

Definition 9.6 $w \in \Sigma^*$ is accepted by a nondeterministic TM M if $q_0 w \overset{*}{\vdash} x q_f y$ for some final state q_f.

The set of all strings accepted by M is denoted by $T(M)$.

Note: As in the case of NDFA, an ID of the form xqy (for some $q \notin F$) may be reached as the result of applying the input string w. But w is accepted by M as long as there is some sequence of moves leading to an ID with an accepting state. It does not matter that there are other sequences of moves leading to an ID with a nonfinal state or TM halts without processing the entire input string.

Theorem 9.3 If M is a nondeterministic TM, there is a deterministic TM M_1 such that $T(M) = T(M_1)$.

Proof We construct M_1 as a multitape TM. Each symbol in the input string leads to a change in ID. M_1 should be able to reach all IDs and stop when an ID containing a final state is reached. So the first tape is used to store IDs of M as a sequence and also the state of M. These IDs are separated by the symbol $*$ (included as a tape symbol). The current ID is known by marking an x along with the ID-separator $*$ (The symbol $*$ marked with x is a new tape symbol.) All IDs to the left of the current one have been explored already and so can be ignored subsequently. Note that the current ID is decided by the current input symbol of w.

Figure 9.10 illustrates the deterministic TM M_1.

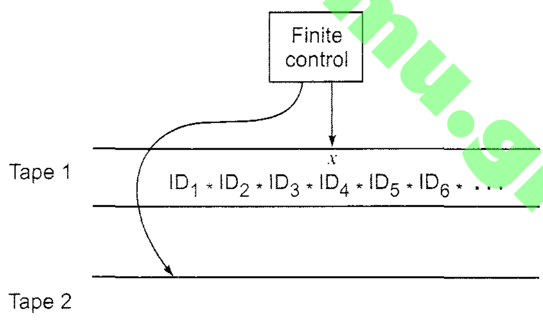

Fig. 9.10 The deterministic TM simulating M.

To process the current ID, M_1 performs the following steps.

1. M_1 examines the state and the scanned symbol of the current ID. Using the knowledge of moves of M stored in the finite control of M_1, M_1 checks whether the state in the current ID is an accepting state of M. In this case M_1 accepts and stops simulating M.

2. If the state q say in the current ID $xqay$ is not an accepting state of M_1 and $\delta(q, a)$ has k triples, M_1 copies the ID $xqay$ in the second tape and makes k copies of this ID at the end of the sequence of IDs in tape 2.
3. M_1 modifies these k IDs in tape 2 according to the k choices given by $\delta(q, a)$.
4. M_1 returns to the marked current ID, erases the mark x and marks the next ID-separator $*$ with x (to the $*$ which is to the left of the next ID to be processed). Then M_1 goes back to step 1.

M_1 stops when an accepting state of M is reached in step 1.

Now M_1 accepts an input string w only when it is able to find that M has entered an accepting state, after a finite number of moves. This is clear from the simulated sequence of moves of M_1 (ending in step 1)

We have to prove that M_1 will eventually reach an accepting ID (that is, an ID having an accepting state of M) if M enters an accepting ID after n moves. Note each move of M is simulated by several moves of M_1.

Let m be the maximum number of choices that M has for various (q, a)'s. (It is possible to find m since we have only finite number of pairs in $Q \times \Gamma$.) So for each initial ID of M, there are at most m IDs that M can reach after one move, at most m^2 IDs that M can reach after two moves, and so on. So corresponding to n moves of M, there are at most $1 + m + m^2 + \cdots + m^n$ moves of M_1. Hence the number of IDs to be explored by M_1 is at most nm^n.

We assume that M_1 explores these IDs. These IDs have a tree structure having the initial ID as its root. We can apply breadth-first search of the nodes of the tree (that is, the nodes at level 1 are searched, then the nodes at level 2, and so on.) If M reaches an accepting ID after n moves, then M_1 has to search atmost nm^n IDs before reaching an accepting ID. So, if M accepts w, then M_1 also accepts w (eventually). Hence $T(M) = T(M_1)$.

9.8 THE MODEL OF LINEAR BOUNDED AUTOMATON

This model is important because (a) the set of context-sensitive languages is accepted by the model, and (b) the infinite storage is restricted in size but not in accessibility to the storage in comparison with the Turing machine model. It is called the *linear bounded automaton* (LBA) because a linear function is used to restrict (to bound) the length of the tape.

In this section we define the model of linear bounded automaton and develop the relation between the linear bounded automata and context-sensitive languages. It should be noted that the study of context-sensitive languages is important from practical point of view because many compiler languages lie between context-sensitive and context-free languages.

A linear bounded automaton is a nondeterministic Turing machine which has a single tape whose length is not infinite but bounded by a linear function of the length of the input string. The models can be described formally by the following set format:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, b, ₵, \$, F)$$

All the symbols have the same meaning as in the basic model of Turing machines with the difference that the input alphabet Σ contains two special symbols ₵ and \$. ₵ is called the left-end marker which is entered in the left-most cell of the input tape and prevents the R/W head from getting off the left end of the tape. \$ is called the right-end marker which is entered in the right-most cell of the input tape and prevents the R/W head from getting off the right end of the tape. Both the endmarkers should not appear on any other cell within the input tape, and the R/W head should not print any other symbol over both the endmarkers.

Let us consider the input string w with $|w| = n - 2$. The input string w can be recognized by an LBA if it can also be recognized by a Turing machine using no more than kn cells of input tape, where k is a constant specified in the description of LBA. The value of k does not depend on the input string but is purely a property of the machine. Whenever we process any string in LBA, we shall assume that the input string is enclosed within the endmarkers ₵ and \$. The above model of LBA can be represented by the block diagram of Fig. 9.11. There are two tapes: one is called the input tape, and the other, working tape. On the input tape the head never prints and never moves to the left. On the working tape the head can modify the contents in any way, without any restriction.

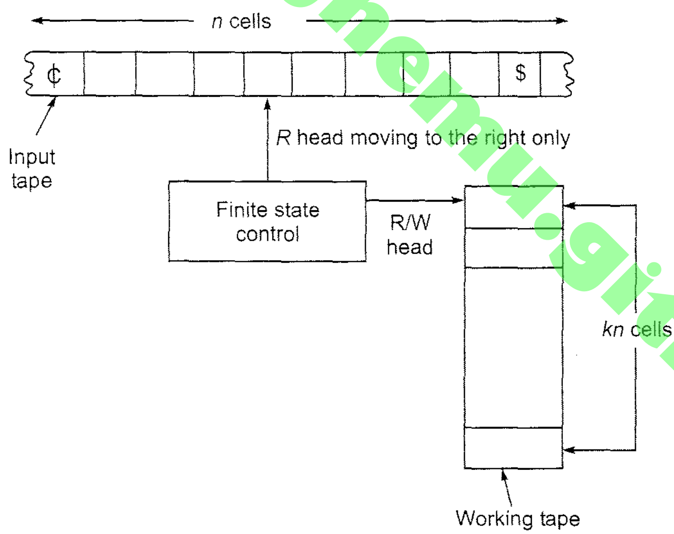

Fig. 9.11 Model of linear bounded automaton.

In the case of LBA, an ID is denoted by (q, w, k), where $q \in Q$, $w \in \Gamma$ and k is some integer between 1 and n. The transition of IDs is similar except that k changes to $k - 1$ if the R/W head moves to the left and to $k + 1$ if the head moves to the right.

The language accepted by LBA is defined as the set

$$\{w \in (\Sigma - \{¢, \$\})^* | (q_0, ¢w\$, 1) \overset{*}{\vdash} (q, \alpha, i)$$

for some $q \in F$ and for some integer i between 1 and $n\}$.

Note: As a null string can be represented either by the absence of input string or by a completely blank tape, an LBA may accept the null string.

9.8.1 Relation Between LBA and Context-sensitive Languages

The set of strings accepted by nondeterministic LBA is the set of strings generated by the context-sensitive grammars, excluding the null strings. Now we give an important result:

If L is a context-sensitive language, then L is accepted by a linear bounded automaton. The converse is also true.

The construction and the proof are similar to those for Turing machines with some modifications.

9.9 TURING MACHINES AND TYPE 0 GRAMMARS

In this section we construct a type 0 grammar generating the set accepted by a given Turing machine M. The productions are constructed in two steps. In step 1 we construct productions which transform the string $[q_1¢\, w\$]$ into the string $[q_2b]$, where q_1 is the initial state, q_2 is an accepting state, ¢ is the left-endmarker, and \$ is the right-endmarker. The grammar obtained by applying step 1 is called the *transformational grammar*. In step 2 we obtain inverse production rules by reversing the productions of the transformational grammar to get the required type 0 grammar G. The construction is in such a way that w is accepted by M if and only if w is in $L(G)$.

9.9.1 Construction of a Grammar Corresponding to TM

For understanding the construction, we have to note that a transition of ID corresponds to a production. We enclose IDs within brackets. So acceptance of w by M corresponds to the transformation of initial ID $[q_1\, ¢\, w\, \$]$ into $[q_2b]$. Also, the 'length' of ID may change if the R/W head reaches the left-end or the right-end, i.e. when the left-hand side or the right-hand side bracket is reached. So we get productions corresponding to transition of IDs with (i) no change in length, and (ii) change in length. We assume that the transition table is given.

We now describe the construction which involves two steps:

Step 1 (i) *No change in length of* IDs: (a) *Right move.* a_kRq_l corresponding to q_i-row and a_j-column leads to the production

$$q_ia_j \to a_kq_l$$

(b) *Left move.* a_kLq_l corresponding to q_i-row and a_j-column yields several productions

$$a_mq_ia_j \to q_la_ma_k \qquad \text{for all } a_m \in \Gamma$$

(ii) *Change in length of* IDs: (a) *Left-end.* a_kLq_l corresponding to q_i-row and a_j-column gives

$$[q_ia_j \to [q_lba_k$$

When b occurs next to the left-bracket, it can be deleted. This is achieved by including the production $[b \to [$.

(b) *Right-end.* When b occurs to the left of], it can be deleted. This is achieved by the production

$$a_jb] \to a_j] \qquad \text{for all } a_j \in \Gamma$$

When the R/W head moves to the right of], the length increases. Corresponding to this we have a production

$$q_i] \to q_ib] \qquad \text{for all } q_i \in Q$$

(iii) *Introduction of endmarkers.* For introducing endmarkers for the input string, the following productions are included:

$$a_i \to [q_1 ¢ a_i \qquad \text{for } a_i \in \Gamma, a_i \neq b$$

$$a_i \to a_i\$] \qquad \text{for all } a_i \in \Gamma, a_i \neq b$$

For removing the brackets from $[q_2b]$, we include the production

$$[q_2b] \to S$$

Recall that q_1 and q_2 are the initial and final states, respectively.

Step 2 To get the required grammar, reverse the arrows of the productions obtained in step 1. The productions we get can be called *inverse productions*. The new grammar is called the *generative grammar*. We illustrate the construction with an example.

EXAMPLE 9.11

Consider the TM described by the transition table given in Table 9.9. Obtain the inverse production rules.

Solution

In this example, q_1 is both initial and final.

Step 1 (i) *Productions corresponding to right moves*

$$q_1¢ \to ¢q_1, \qquad q_11 \to bq_2, \qquad q_21 \to bq_1 \tag{9.1}$$

(ii) (a) *Productions corresponding to left-end*

$$[b \rightarrow [\tag{9.2}$$

(b) *Productions corresponding to right-end*

$$bb] \rightarrow b], \quad 1b] \rightarrow 1], \quad q_1] \rightarrow q_1 b], \quad q_2] \rightarrow q_2 b] \tag{9.3}$$

(iii) $$1 \rightarrow [q_1 ¢1, \quad 1 \rightarrow 1\$], \quad [q_1 b] \rightarrow S \tag{9.4}$$

TABLE 9.9 Transition Table for Example 9.11

Present state	¢	*b*	1
$\rightarrow$ (q_1)	$¢Rq_1$		bRq_2
q_2			bRq_1

Step 2 The inverse productions are obtained by reversing the arrows of the productions (9.1)–(9.4).

$$¢q_1 \rightarrow q_1¢ \qquad bq_2 \rightarrow q_1 1, \qquad bq_1 \rightarrow q_2 1$$
$$[\rightarrow [b, \ b] \rightarrow bb], \qquad 1] \rightarrow 1b]$$
$$q_1 b \rightarrow q_1], \qquad q_2 b \rightarrow q_2], \qquad [q_1 ¢1 \rightarrow 1$$
$$1\$] \rightarrow 1, \qquad S \rightarrow [q_1 b]$$

Thus we have shown that there exists a type 0 grammar corresponding to a Turing machine. The converse is also true (we are not proving this), i.e. given a type 0 grammar G, there exists a Turing machine accepting $L(G)$. Actually, the class of recursively enumerable sets, the type 0 languages, and the class of sets accepted by TM are one and the same. We have shown that there exists a recursively enumerable set which is not a context-sensitive language (see Theorem 4.4). As a recursive set is recursively enumerable, Theorem 4.4 gives a type 0 language which is not type 1. Hence, $\mathcal{L}_{\text{csl}} \subset \mathcal{L}_0$ (cf Property 4, Section 4.3) is established.

9.10 LINEAR BOUNDED AUTOMATA AND LANGUAGES

A linear bounded automaton M accepts a string w if, after starting at the initial state with R/W head reading the left-endmarker, M halts over the right-endmarker in a final state. Otherwise, w is rejected.

The production rules for the generative grammar are constructed as in the case of Turing machines. The following additional productions are needed in the case of LBA.

$$a_i q_f \$ \rightarrow q_f \$ \qquad \text{for all } a_i \in \Gamma$$
$$¢q_f \$ \rightarrow ¢q_f, \qquad ¢q_f \rightarrow q_f$$

EXAMPLE 9.12

Find the grammar generating the set accepted by a linear bounded automaton M whose transition table is given in Table 9.10.

TABLE 9.10 Transition Table for Example 9.12

Present state	*Tape input symbol*			
	¢	\$	0	1
$\to q_1$	$\text{¢}Rq_1$		$1Lq_2$	$0Rq_2$
q_2	$\text{¢}Rq_4$		$1Rq_3$	$1Lq_1$
q_3		$\$Lq_1$	$1Rq_3$	$1Rq_3$
q_4		Halt	$0Lq_4$	$0Rq_4$

Solution

Step 1 (A) (i) *Productions corresponding to right moves.* The seven right moves in Table 9.10 give the following productions:

$$
\begin{array}{ll}
q_1\text{¢} \to \text{¢}q_1, & q_30 \to 1q_3 \\
q_11 \to 0q_2, & q_31 \to 1q_3 \\
q_2\text{¢} \to \text{¢}q_4, & q_41 \to 0q_4 \\
q_20 \to 1q_3 &
\end{array}
\tag{9.5}
$$

(ii) *Productions corresponding to left moves.* There are four left moves in Table 9.10. Each left move yields four productions (corresponding to the four tape symbols). These are:

(a) $1Lq_2$ corresponding to q_1-row and 0-column gives

$$\text{¢}q_10 \to q_2\text{¢}1,\ \$q_10 \to q_2\$1,\ 0q_10 \to q_201,\ 1q_10 \to q_211 \tag{9.6}$$

(b) $1Lq_1$ corresponding to q_1-row and 1-column yields

$$\text{¢}q_21 \to q_1\text{¢}1,\ \$q_21 \to q_1\$1,\ 0q_21 \to q_101,\ 1q_21 \to q_111 \tag{9.7}$$

(c) $\$Lq_1$ corresponding to q_3-row and \$-column gives

$$\text{¢}q_3\$ \to q_1\text{¢}\$,\ \$q_3\$ \to q_1\$\$,\ 0q_3\$ \to q_10\$,\ 1q_3\$ \to q_11\$ \tag{9.8}$$

(d) $0Lq_4$ corresponding to q_4-row and 0-column yields

$$\text{¢}q_40 \to q_4\text{¢}0,\ \$q_40 \to q_4\$0,\ 0q_40 \to q_400,\ 1q_40 \to q_410 \tag{9.9}$$

(B) There are no productions corresponding to change in length.

(C) The productions for introducing the endmarkers are

$$
\begin{array}{ll}
\text{¢} \to [q_1\text{¢}\text{¢}, & \text{¢} \to \text{¢}\$] \\
\$ \to [q_1\text{¢}\$, & \$ \to \$\$] \\
0 \to [q_1\text{¢}0, & 0 \to 0\$] \\
1 \to [q_1\text{¢}1, & 1 \to 1\$]
\end{array}
\tag{9.10}
$$

$$[q_4] \to S \tag{9.11}$$

(D) The LBA productions are

$$\begin{array}{ll} ¢q_4\$ \rightarrow q_4\$, & ¢q_4\$ \rightarrow ¢q_4 \\ \$q_4\$ \rightarrow q_4\$, & ¢q_4 \rightarrow q_4 \\ 0q_4\$ \rightarrow q_4\$, & \\ 1q_4\$ \rightarrow q_4\$ & \end{array} \tag{9.12}$$

Step 2 The productions of the generative grammar are obtained by reversing the arrows of productions given by (9.5)–(9.12).

9.11 SUPPLEMENTARY EXAMPLES

EXAMPLE 9.13

Design a TM that copies strings of 1's.

Solution

We design a TM so that we have ww after copying $w \in \{1\}^*$. Define M by

$$M = (\{q_0, q_1, q_2, q_3\}, \{1\}, \{1, b\}, \delta, q_0, b, \{q_3\})$$

where δ is defined by Table 9.11.

TABLE 9.11 Transition Table for Example 9.13

Present state	*Tape symbol*		
	1	*b*	*a*
q_0	q_0aR	q_1bL	—
q_1	q_11L	q_3bR	q_21R
q_2	q_21R	q_11L	—
q_3	—	—	—

The procedure is simple.

M replaces every 1 by the symbol a. Then M replaces the rightmost a by 1. It goes to the right end of the string and writes a 1 there. Thus M has added a 1 for the rightmost 1 in the input string w. This process can be repeated.

M reaches q_1 after replacing all 1's by a's and reading the blank at the end of the input string. After replacing a by 1, M reaches q_2. M reaches q_3 at the end of the process and halts. If $w = 1^n$, than we have 1^{2n} at the end of the computation. A sample computation is given below.

$$\begin{aligned} q_011 &\vdash aq_01 \vdash aaq_0b \vdash aq_1a \\ &\vdash a1q_2b \vdash aq_111 \vdash q_1a11 \\ &\vdash 1q_211 \vdash 11q_21 \vdash 111q_2b \\ &\vdash 11q_211 \vdash 1q_1111 \\ &\vdash q_11111 \vdash q_1b1111 \vdash q_31111 \end{aligned}$$

EXAMPLE 9.14

Construct a TM to accept the set L of all strings over $\{0,1\}$ ending with 010.

Solution

L is certainly a regular set and hence a deterministic automaton is sufficient to recognize L. Figure 9.12 gives a DFA accepting L.

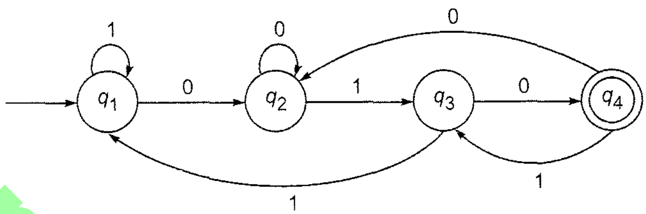

Fig. 9.12 DFA for Example 9.14.

Converting this DFA to a TM is simple. In a DFA M, the move is always to the right. So the TM's move will always be to the right. Also M reads the input symbol and changes state. So the TM M_1 does the same; it reads an input symbol, does not change the symbol and changes state. At the end of the computation, the TM sees the first blank b and changes to its final state. The initial ID of M_1 is q_0w. By defining $\delta(q_0, b) = (q_1, b, R)$, M_1 reaches the initial state of M. M_1 can be described by Fig. 9.13.

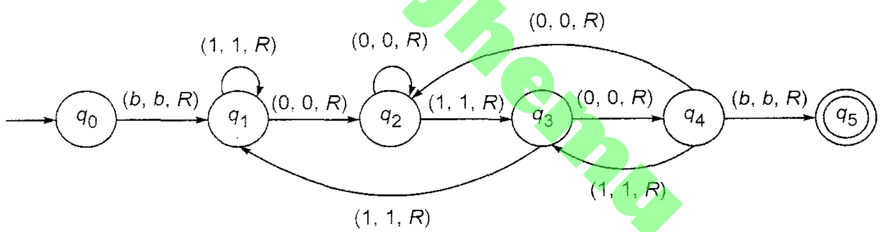

Fig. 9.13 TM for Example 9.14.

Note: q_5 is the unique final state of M_1. By comparing Figs. 9.12 and 9.13 it is easy to see that strings of L are accepted by M_1.

EXAMPLE 9.15

Design a TM that reads a string in $\{0, 1\}^*$ and erases the rightmost symbol.

Solution

The required TM M is given by

$$M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0, 1\}, \{0, 1, b\}, \delta, q_0, b, \{q_4\})$$

where δ is defined by

$$\delta(q_0, 0) = (q_1, 0, R) \qquad \delta(q_0, 1) = (q_1, 1, R) \qquad (R_1)$$
$$\delta(q_1, 0) = (q_1, 0, R) \qquad \delta(q_1, 1) = (q_1, 1, R) \qquad (R_2)$$
$$\delta(q_1, b) = (q_2, b, L) \qquad (R_3)$$
$$\delta(q_2, 0) = (q_3, b, L) \qquad \delta(q_2, 1) = (q_3, b, L) \qquad (R_4)$$
$$\delta(q_3, 0) = (q_3, 0, L) \qquad \delta(q_3, 1) = (q_3, 1, L) \qquad (R_5)$$
$$\delta(q_3, b) = (q_4, b, R) \qquad (R_6)$$

Let w be the input string. By (R_1) and (R_2), M reads the entire input string w. At the end, M is in state q_1. On seeing the blank to the right of w, M reaches the state q_2 and moves left. The rightmost string in w is erased (by (R_4)) and the state becomes q_3. Afterwards M moves to the left until it reaches the left-end of w. On seeing the blank b to the right of w, M changes its state to q_4, which is the final state of M. From the construction it is clear that the rightmost symbol of w is erased.

EXAMPLE 9.16

Construct a TM that accepts $L = \{0^{2^n} \mid n \geq 0\}$.

Solution

Let w be an input string in $\{0\}^*$. The TM accepting L functions as follows:

1. It writes b (blank symbol) on the leftmost 0 of the input string w. This is done to mark the left-end of w.
2. M reads the symbols of w from left to right and replaces the alternate 0's with x's.
3. If the tape contains a single 0 in step 2, M accepts w.
4. If the tape contains more than one 0 and the number of 0's is odd in step 2, M rejects w.
5. M returns the head to the left-end of the tape (marked by blank b in step 1).
6. M goes to step 2.

Each iteration of step 2 reduces w to half its size. Also whether the number of 0's seen is even or odd is known after step 2. If that number is odd and greater than 1, w cannot be 0^{2^n} (step 4). In this case M rejects w. If the number of 0's seen is 1 (step 3), M accepts w (In this case 0^{2^n} is reduced to 0 in successive stages of step 2).

We define M by

$$M = (\{q_0, q_1, q_2, q_3, q_4, q_f, q_t\}, \{0\}, \{0, x, b\}, \delta, q_0, b, \{q_f\})$$

where δ is defined by Table 9.12.

TABLE 9.12 Transition Table for Example 9.16

Present state	*Tape symbol*		
	0	b	x
q_0	bRq_1	bRq_t	xRq_t
q_1	xRq_2	bRq_f	xRq_1
q_2	$0Rq_3$	bRq_4	xRq_2
q_3	xRq_2	bRq_6	xRq_3
q_4	$0Lq_4$	bRq_1	xLq_4
q_f	—	—	—
q_t	—	—	—

From the construction, it is apparent that the states are used to know whether the number of 0's read is odd or even.

We can see how M processes 0000.

$$\begin{aligned} q_00000 &\vdash bq_1000 \vdash bxq_200 \vdash bxq_30 \vdash bx0xq_2b \\ &\vdash bx0q_4xb \vdash bxq_40xb \vdash bq_4x0xb \vdash q_4bx0xb \\ &\vdash bq_1x0xb \vdash bxq_10xb \vdash bxxq_2xb \vdash bxxxq_2b \\ &\vdash bxxq_4xb \vdash bxq_4xxb \vdash bq_4xxxb \vdash q_4bxxxb \\ &\vdash bq_1xxxb \vdash bxq_1xxb \vdash bxxq_1xb \vdash bxxxq_1b \\ &\vdash bxxxbq_f. \end{aligned}$$

Hence M accepts w.

Also note that M always halts. If M reaches q_f, the input string w is accepted by M. If M reaches q_t, w is not accepted by M; in this case M halts in the trap state.

EXAMPLE 9.17

Let $M = (\{q_0, q_1, q_2\}, \{0, 1\}, \{0, 1, b\}, \delta, q_0, \{q_2\})$

where δ is given by

$$\delta(q_0, 0) = (q_1, 1, R) \qquad (R_1)$$

$$\delta(q_1, 1) = (q_0, 0, R) \qquad (R_2)$$

$$\delta(q_1, b) = (q_2, b, R) \qquad (R_3)$$

Find $T(M)$.

Solution

Let $w \in T(M)$. As $\delta(q_0, 1)$ is not defined, w cannot start with 1. From (R_1) and (R_2), we can conclude that M starts from q_0 and comes back to q_0 after reaching 01.

So, $q_0(01)^n \overset{*}{\vdash} (10)^n q_0$. Also, $q_00b \vdash 1q_1b \vdash 1bq_2$.

So, $(01)^n0 \in T(M)$. Also, $(01)^n0$ is the only string that makes M move from q_0 to q_2. Hence, $T(M) = \{(01)^n0 \mid n \geq 0\}$.

SELF-TEST

Choose the correct answer to Questions 1–10:

1. For the standard TM:
   (a) $\Sigma = \Gamma$
   (b) $\Gamma \subseteq \Sigma$
   (c) $\Sigma \subseteq \Gamma$
   (d) Σ is a proper subset of Γ.

2. In a standard TM, $\delta(q, a)$, $q \in Q$, $a \in \Gamma$ is
   (a) defined for all $(q, a) \in Q \times \Gamma$
   (b) defined for some, not necessarily for all $(q, a) \in Q \times \Gamma$
   (c) defined for no element (q, a) of $Q \times \Gamma$
   (d) a set of triples with more than one element.

3. If $\delta(q, x_i) = (p, y, L)$, then
   (a) $x_1x_2 \ldots x_{i-1}qx_i \ldots x_n \vdash x_1x_2 \ldots x_{i-2}px_{i-1}yx_{i+1} \ldots x_n$
   (b) $x_1x_2 \ldots x_{i-1}qx_i \ldots x_n \vdash x_1x_2 \ldots x_{i-1}ypx_{i+1} \ldots x_n$
   (c) $x_1x_2 \ldots x_{i-1}qx_i \ldots x_n \vdash x_1 \ldots x_{i-3}px_{i-2}x_{i-1}yx_{i+1} \ldots x_n$
   (d) $x_1x_2 \ldots x_{i-1}qx_i \ldots x_n \vdash x_1 \ldots x_{i+1}pyx_{i+2} \ldots x_n$

4. If $\delta(q, x_i) = (p, y, R)$, then
   (a) $x_1x_2 \ldots x_{i-1}qx_i \ldots x_n \vdash x_1x_2 \ldots x_{i-1}ypx_{i+1} \ldots x_n$
   (b) $x_1x_2 \ldots x_{i-1}qx_i \ldots x_n \vdash x_1x_2 \ldots x_ipx_{i+1} \ldots x_n$
   (c) $x_1x_2 \ldots x_{i-1}qx_i \ldots x_n \vdash x_1x_2 \ldots x_{i-1}px_ix_{i+1} \ldots x_n$
   (d) $x_1x_2 \ldots x_{i-1}qx_i \ldots x_n \vdash x_1x_2 \ldots x_{i-1}ypx_{i+1} \ldots x_n$

5. If $\delta(q, x_1) = (p, y, L)$, then
   (a) $qx_1x_2 \ldots x_n \vdash pyx_2 \ldots x_n$
   (b) $qx_1x_2 \ldots x_n \vdash ypx_2 \ldots x_n$
   (c) $qx_1x_2 \ldots x_n \vdash pbx_1 \ldots x_n$
   (d) $qx_1x_2 \ldots x_n \vdash pbx_2 \ldots x_n$

6. If $\delta(q, x_n) = (p, y, R)$, then
   (a) $x_1 \ldots x_{n-1}qx_n \vdash pyx_2x_3 \ldots x_n$
   (b) $x_1 \ldots x_{n-1}qx_n \vdash^* pyx_2x_3 \ldots x_n$
   (c) $x_1 \ldots x_{n-1}qx_n \vdash x_1x_2 \ldots x_{n-1}ypb$
   (d) $x_1 \ldots x_{n-1}qx_n \vdash^* x_1x_2 \ldots x_{n-1}ypb$

7. For the TM given in Example 9.6:
   (a) $q_01b11 \vdash^* bq_f11bb1$
   (b) $q_01b11 \vdash bq_f11bb1$
   (c) $q_01b11 \vdash 1q_0b111$
   (d) $q_01b11 \vdash q_3b11bb1$

8. For the TM given in Example 9.4:
   (a) 011 is accepted by M
   (b) 001 is accepted by M
   (c) 00 is accepted by M
   (d) 0011 is accepted by M.

9. For the TM given in Example 9.5:
   (a) 1 is accepted by M
   (b) 11 is accepted by M
   (c) 111 is accepted by M
   (d) 11111 is accepted by M

10. In a standard TM $(Q, \Sigma, \Gamma, \delta, q_0, b, F)$ the blank symbol b is
   (a) in $\Sigma - \Gamma$
   (b) in $\Gamma - \Sigma$
   (c) $\Gamma \cap \Sigma$
   (d) none of these

EXERCISES

9.1 Draw the transition diagram of the Turing machine given in Table 9.1.

9.2 Represent the transition function of the Turing machine given in Example 9.2 as a set of quintuples.

9.3 Construct the computation sequence for the input $1b11$ for the Turing machine given in Example 9.5.

9.4 Construct the computation sequence for strings 1213, 2133, 312 for the Turing machine given in Example 9.8.

9.5 Explain how a Turing machine can be considered as a computer of integer functions (i.e. as one that can compute integer functions; we shall discuss more about this in Chapter 11).

9.6 Design a Turing machine that converts a binary string into its equivalent unary string.

9.7 Construct a Turing machine that enumerates $\{0^n1^n \mid n \geq 1\}$.

9.8 Construct a Turing machine that can accept the set of all even palindromes over $\{0, 1\}$.

9.9 Construct a Turing machine that can accept the strings over $\{0, 1\}$ containing even number of 1's.

9.10 Design a Turing machine to recognize the language $\{a^nb^nc^m \mid n, m \geq 1\}$.

9.11 Design a Turing machine that can compute proper subtraction, i.e. $m \dot{-} n$, where m and n are positive integers. $m \dot{-} n$ is defined as $m - n$ if $m > n$ and 0 if $m \leq n$.

10 Decidability and Recursively Enumerable Languages

In this chapter the formal definition of an algorithm is given. The problem of decidability of various class of languages is discussed. The theorem on halting problem of Turing machine is proved.

10.1 THE DEFINITION OF AN ALGORITHM

In Section 4.4, we gave the definition of an algorithm as a procedure (finite sequence of instructions which can be mechanically carried out) that terminates after a finite number of steps for any input. The earliest algorithm one can think of is the Euclidean algorithm, for computing the greatest common divisor of two natural numbers. In 1900, the mathematician David Hilbert, in his famous address at the International congress of mathematicians in Paris, averred that every definite mathematical problem must be susceptible for an exact settlement either in the form of an exact answer or by the proof of the impossibility of its solution. He identified 23 mathematical problems as a challenge for future mathematicians; only ten of the problems have been solved so far.

Hilbert's tenth problem was to devise 'a process according to which it can be determined by a finite number of operations', whether a polynomial over Z has an integral root. (He did not use the word 'algorithm' but he meant the same.) This was not answered until 1970.

The formal definition of algorithm emerged after the works of Alan Turing and Alanzo Church in 1936. The Church–Turing thesis states that any algorithmic procedure that can be carried out by a human or a computer, can also be carried out by a Turing machine. Thus the Turing machine arose as an ideal theoretical model for an algorithm. The Turing machine provided a machinery to mathematicians for attacking the Hilberts' tenth problem. The problem can be restated as follows: does there exist a TM that can accept a polynomial over n variables if it has an integral root and reject the polynomial if it does not have one.

In 1970, Yuri Matijasevic, after studying the work of Martin Davis, Hilary Putnam and Julia Robinson showed that no such algorithm (Turing machine) exists for testing whether a polynomial over n variables has integral roots. Now it is universally accepted by computer scientists that Turing machine is a mathematical model of an algorithm.

10.2 DECIDABILITY

We are familiar with the recursive definition of a function or a set. We also have the definitions of recursively enumerable sets and recursive sets (refer to Section 4.4). The notion of a recursively enumerable set (or language) and a recursive set (or language) existed even before the dawn of computers.

Now these terms are also defined using Turing machines. When a Turing machine reaches a final state, it 'halts.' We can also say that a Turing machine M halts when M reaches a state q and a current symbol a to be scanned so that $\delta(q, a)$ is undefined. There are TMs that never halt on some inputs in any one of these ways. So we make a distinction between the languages accepted by a TM that halts on all input strings and a TM that never halts on some input strings.

Definition 10.1 A language $L \subseteq \Sigma^*$ is recursively enumerable if there exists a TM M, such that $L = T(M)$.

Definition 10.2 A language $L \subseteq \Sigma^*$ is recursive if there exists some TM M that satisfies the following two conditions.

(i) If $w \in L$ then M accepts w (that is, reaches an accepting state on processing w) and halts.
(ii) If $w \notin L$ then M eventually halts, without reaching an accepting state.

Note: Definition 10.2 formalizes the notion of an 'algorithm'. An algorithm, in the usual sense, is a well-defined sequence of steps that always terminates and produces an answer. The Conditions (i) and (ii) of Definition 10.2 assure us that the TM always halts, accepting w under Condition (i) and not accepting under Condition (ii). So a TM, defining a recursive language (Definition 10.2) always halts eventually just as an algorithm eventually terminates.

A problem with only two answers Yes/No can be considered as a language L. An instance of the problem with the answer 'Yes' can be considered as an element of the corresponding language L; an instance with answer 'No' is considered as an element not in L.

Definition 10.3 A problem with two answers (Yes/No) is decidable if the corresponding language is recursive. In this case, the language L is also called *decidable*.

Definition 10.4 A problem/language is undecidable if it is not decidable.

Note: A decidable problem is called a solvable problem and an undecidable problem an unsolvable problem by some authors.

10.3 DECIDABLE LANGUAGES

In this section we consider the decidability of regular and context-free languages.

First of all, we consider the problem of testing whether a deterministic finite automaton accepts a given input string w.

Definition 10.5

$$A_{\text{DFA}} = \{(B, w) \mid B \text{ accepts the input string } w\}$$

Theorem 10.1 A_{DFA} is decidable.

Proof To prove the theorem, we have to construct a TM that always halts and also accepts A_{DFA}. We describe the TM M using high level description (refer to Section 9.5). Note that a DFM B always ends in some state of B after n transitions for an input string of length n.

We define a TM M as follows:

1. Let B be a DFA and w an input string. (B, w) is an input for the Turing machine M.
2. Simulate B and input w in the TM M.
3. If the simulation ends in an accepting state of B, then M accepts w. If it ends in a nonaccepting state of B, then M rejects w.

We can discuss a few implementation details regarding steps 1, 2 and 3 above. The input (B, w) for M is represented by representing the five components Q, Σ, δ, q_0, f by strings of Σ^* and input string $w \in \Sigma^*$. M checks whether (B, w) is a valid input. If not, it rejectes (B, w) and halts. If (B, w) is a valid input, M writes the initial state q_0 and the leftmost input symbol of w. It updates the state using δ and then reads the next symbol in w. This explains step 2.

If the simulation ends in an accepting state w, then M accepts (B, w). Otherwise, M rejects (B, w). This is the description of step 3.

It is evident that M accepts (B, w) if and only if w is accepted by the DFA B. ▌

Definition 10.6

$$A_{\text{CFG}} = \{(G, w) \mid \text{the context-free grammar } G \text{ accepts the input string } w\}$$

Theorem 10.2 A_{CFG} is decidable.

Proof We convert a CFG into Chomsky normal form. Then any derivation of w of length k requires $2k - 1$ steps if the grammar is in CNF (refer to Example 6.18). So for checking whether the input string w of length k is in $L(G)$, it is enough to check derivations in $2k - 1$ steps. We know that there are only finitely many derivations in $2k - 1$ steps. Now we design a TM M that halts as follows.

1. Let G be a CFG in Chomsky normal form and w an input string. (G, w) is an input for M.
2. If $k = 0$, list all the single-step derivations. If $k \neq 0$, list all the derivations with $2k - 1$ steps.
3. If any of the derivations in step 2 generates the given string w, M accepts (G, w). Otherwise M rejects.

The implementation of steps 1–3 is similar to the steps in Theorem 10.1. (G, w) is represented by representing the four components V_N, Σ, P, S of G and input string w. The next step of the derivation is got by the production to be applied.

M accepts (G, w) if and only if w is accepted by the CFG G.

In Theorem 4.3, we proved that a context-sensitive language is recursive. The main idea of the proof of Theorem 4.3 was to construct a sequence $\{W_0, W_1, \ldots, W_k\}$ of subsets of $(V_N \cup \Sigma)^*$, that terminates after a finite number of iterations. The given string $w \in \Sigma^*$ is in $L(G)$ if and only if $w \in W_k$. With this idea in mind we can prove the decidability of the context-sensitive language. ▮

Definition 10.7 $A_{CSG} = \{(G, w) \mid$ the context-sensitive grammar G accepts the input string $w\}$.

Theroem 10.3 A_{CSG} is decidable.

Proof The proof is a modification of the proof of Theorem 10.2. In Theorem 10.2, we considered derivations with $2k - 1$ steps for testing whether an input string of length k was in $L(G)$. In the case of context-sensitive grammar we construct $W_i = \{\alpha \in (V_N \cup \Sigma)^* \mid S \overset{*}{\underset{G}{\Rightarrow}} \alpha$ in i or fewer steps and $|\alpha| \leq n\}$. There exists a natural number k such that $W_k = W_{k+1} = W_{k+2} = \ldots$ (refer to proof of Theorem 4.3).

So $w \in L(G)$ if and only if $w \in W_k$. The construction of W_k is the key idea used in the construction of a TM accepting A_{CSG}. Now we can design a Turing machine M as follows:

1. Let G be a context-sensitive grammar and w an input string of length n. Then (G, w) is an input for TM.
2. Construct $W_0 = \{S\}$. $W_{i+1} = W_i \cup \{\beta \in (V_N \cup \Sigma)^* \mid$ there exists $\alpha_i \in W_i$ such that $\alpha \Rightarrow \beta$ and $|\beta| \leq n\}$. Continue until $W_k = W_{k+1}$ for some k. (This is possible by Theorem 4.3.)
3. If $w \in W_k$, $w \in L(G)$ and M accepts (G, w); otherwise M rejects (G, w). ▮

Note: If $\mathcal{L}_d$ denotes the class of all decidable languages over Σ, then

$$\mathcal{L}_{rl} \subseteq \mathcal{L}_{cfl} \subseteq \mathcal{L}_{csl} \subseteq \mathcal{L}_d$$

10.4 UNDECIDABLE LANGUAGES

In this section we prove the existence of languages that are not recursively enumerable and address the undecidability of recursively enumerable languages.

Theorem 10.4 There exists a language over Σ that is not recursively enumerable.

Proof A language L is recursively enumerable if there exists a TM M such that $L = T(M)$. As Σ is finite, Σ^* is countable (that is, there exists a one-to-one correspondence between Σ^* and N).

As a Turing machine M is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, b, F)$ and each member of the 7-tuple is a finite set, M can be encoded as a string. So the set I of all TMs is countable.

Let $\mathcal{L}$ be the set of all languages over Σ. Then a member of $\mathcal{L}$ is a subset of Σ^* (Note that Σ^* is infinite even though Σ is finite). We show that $\mathcal{L}$ is uncountable (that is, an infinite set not in one-to correspondence with N).

We prove this by contradiction. If $\mathcal{L}$ were countable then $\mathcal{L}$ can be written as a sequence $\{L_1, L_2, L_3, \ldots\}$. We write Σ^* as a sequence $\{w_1, w_2, w_3, \ldots\}$. So L_i can be represented as an infinite binary sequence $x_{i1}x_{i2}x_{i3}\ldots$ where

$$x_{ij} = \begin{cases} 1 & \text{if } w_j \in L_i \\ 0 & \text{otherwise} \end{cases}$$

Using this representation we write L_i as an infinite binary sequence.

$$\begin{array}{l} L_1 : x_{11}x_{12}x_{13} \ \ldots\ x_{1j}\ \ldots \\ L_2 : x_{21}x_{22}x_{23} \ \ldots\ x_{2j}\ \ldots \\ \quad\vdots \qquad\qquad\qquad \vdots \\ L_i : x_{i1}x_{i2}x_{i3} \ \ldots\ x_{ij}\ \ldots \end{array}$$

Fig. 10.1 Representation of $\mathcal{L}$.

We define a subset L of Σ^* by the binary sequence $y_1y_2y_3 \ldots$ where $y_i = 1 - x_{ii}$. If $x_{ii} = 0$, $y_i = 1$ and if $x_{ii} = 1$, $y_i = 0$. Thus according to our assumption the subset L of Σ^* represented by the infinite binary sequence $y_1y_2y_3 \ldots$ should be L_k for some natural number k. But $L \neq L_k$, since $w_k \in L$ if and only if $w_k \notin L_k$. This contradicts our assumption that $\mathcal{L}$ is countable. Therefore $\mathcal{L}$ is uncountable. As I is countable, $\mathcal{L}$ should have some members not corresponding to any TM in I. This proves the existence of a language over Σ that is not recursively enumerable. ∎

Definition 10.8 $A_{\text{TM}} = \{(M, w) \mid \text{The TM } M \text{ accepts } w\}$.

Theorem 10.5 A_{TM} is undecidable.

Proof We can prove that A_{TM} is recursively enumerable. Construct a TM U as follows:

$\langle M, w\rangle$ is an input to U. Simulate M on w. If M enters an accepting state, U accepts $\langle M, w\rangle$. Hence A_{TM} is recursively enumerable. We prove that A_{TM} is undecidable by contradiction. We assume that A_{TM} is decidable by a TM H that eventually halts on all inputs. Then

$$H(M, w) = \begin{cases} \text{accept} & \text{if } M \text{ accepts } w \\ \text{reject} & \text{if } M \text{ does not accept } w \end{cases}$$

We construct a new TM D with H as subroutine. D calls H to determine what M does when it receives the input $\langle M\rangle$, the encoded description of M as a string. Based on the received information on $(M, \langle M\rangle)$, D rejects M if M accepts $\langle M\rangle$ and accepts M if M rejects $\langle M\rangle$. D is described as follows:

1. $\langle M\rangle$ is an input to D, where $\langle M\rangle$ is the encoded string representing M.
2. D calls H to run on $(M, \langle M\rangle)$
3. D rejects $\langle M\rangle$ if H accepts $(M, \langle M\rangle)$ and accepts $\langle M\rangle$ if H rejects $(M, \langle M\rangle)$.

Now step 3 can be described as follows:

$$D(\langle M\rangle) = \begin{cases} \text{accept} & \text{if } M \text{ does not accept } \langle M\rangle \\ \text{reject} & \text{if } M \text{ accepts } \langle M\rangle \end{cases}$$

Let us look at the action of D on the input $\langle D\rangle$. According to the construction of D,

$$D(\langle D\rangle) = \begin{cases} \text{accept} & \text{if } D \text{ does not accept } \langle D\rangle \\ \text{reject} & \text{if } D \text{ accepts } \langle D\rangle \end{cases}$$

This means D accepts $\langle D\rangle$ if D does not accept $\langle D\rangle$, which is a contradiction. Hence ATM is undecidable. ∎

The Turing machine U used in the proof of Theorem 10.5 is called the *universal Turing machine*. U is called universal since it is simulating any other Turing machine.

10.5 HALTING PROBLEM OF TURING MACHINE

In this section we introduce the reduction technique. This technique is used to prove the undecidability of halting problem of Turing machine.

We say that problem A is reducible to problem B if a solution to problem B can be used to solve problem A.

For example, if A is the problem of finding some root of $x^4 - 3x^2 + 2 = 0$ and B is the problem of finding some root of $x^2 - 2 = 0$, then A is reducible to B. As $x^2 - 2$ is a factor of $x^4 - 3x^2 + 2$, a root of $x^2 - 2 = 0$ is also a root of $x^4 - 3x^2 + 2 = 0$.

Note: If A is reducible to B and B is decidable then A is decidable. If A is reducible to B and A is undecidable, then B is undecidable.

Theorem 10.6 $HALT_{TM} = \{(M, w) \mid \text{The Turing machine } M \text{ halts on input } w\}$ is undecidable.

Proof We assume that $HALT_{TM}$ is decidable, and get a contradiction. Let M_1 be the TM such that $T(M_1) = HALT_{TM}$ and let M_1 halt eventually on all (M, w). We construct a TM M_2 as follows:

1. For M_2, (M, w) is an input.
2. The TM M_1 acts on (M, w).
3. If M_1 rejects (M, w) then M_2 rejects (M, w).
4. If M_1 accepts (M, w), simulate the TM M on the input string w until M halts.
5. If M has accepted w, M_2 accepts (M, w); otherwise M_2 rejects (M, w).

When M_1 accepts (M, w) (in step 4), the Turing machine M halts on w. In this case either an accepting state q or a state q' such that $\delta(q', a)$ is undefined till some symbol a in w is reached. In the first case (the first alternative of step 5) M_2 accepts (M, w). In the second case (the second alternative of step 5) M_2 rejects (M, w).

It follows from the definition of M_2 that M_2 halts eventually.

Also, $$T(M_2) = \{(M, w) \mid \text{The Turing machine accepts } w\}$$
$$= A_{TM}$$

This is a contradiction since A_{TM} is undecidable. ∎

10.6 THE POST CORRESPONDENCE PROBLEM

The Post Correspondence Problem (PCP) was first introduced by Emil Post in 1946. Later, the problem was found to have many applications in the theory of formal languages. The problem over an alphabet Σ belongs to a class of yes/no problems and is stated as follows: Consider the two lists $x = (x_1 \ldots x_n)$, $y = (y_1 \ldots y_n)$ of nonempty strings over an alphabet $\Sigma = \{0, 1\}$. The PCP is to determine whether or not there exist $i_1, \ldots, i_m$, where $1 \le i_j \le n$, such that

$$x_{i_1} \ldots x_{i_m} = y_{i_1} \ldots y_{i_m}$$

Note: The indices i_j's need not be distinct and m may be greater than n. Also, if there exists a solution to PCP, there exist infinitely many solutions.

EXAMPLE 10.1

Does the PCP with two lists $x = (b, bab^3, ba)$ and $y = (b^3, ba, a)$ have a solution?

Solution

We have to determine whether or not there exists a sequence of substrings of x such that the string formed by this sequence and the string formed by the sequence of corresponding substrings of y are identical. The required sequence is given by $i_1 = 2$, $i_2 = 1$, $i_3 = 1$, $i_4 = 3$, i.e. (2, 1, 1,3), and $m = 4$. The corresponding strings are

$\boxed{bab^3}$	$\boxed{b}$	$\boxed{b}$	$\boxed{ba}$	=	$\boxed{ba}$	$\boxed{b^3}$	$\boxed{b^3}$	$\boxed{a}$
x_2	x_1	x_1	x_3		y_2	y_1	y_1	y_3

Thus the PCP has a solution.

EXAMPLE 10.2

Prove that PCP with two lists $x = (01, 1, 1)$, $y = (01^2, 10, 1^1)$ has no solution.

Solution

For each substring $x_i \in x$ and $y_i \in y$, we have $|x_i| < |y_i|$ for all i. Hence the string generated by a sequence of substrings of x is shorter than the string generated by the sequence of corresponding substrings of y. Therefore, the PCP has no solution.

Note: If the first substring used in PCP is always x_1 and y_1, then the PCP is known as the *Modified Post Correspondence Problem.*

EXAMPLE 10.3

Explain how a Post Correspondence Problem can be treated as a game of dominoes.

Solution

The PCP may be thought of as a game of dominoes in the following way: Let each domino contain some x_i in the upper-half, and the corresponding substring of y in the lower-half. A typical domino is shown as

x_i	upper-half
y_i	lower-half

The PCP is equivalent to placing the dominoes one after another as a sequence (of course repetitions are allowed). To win the game, the same string should appear in the upper-half and in the lower-half. So winning the game is equivalent to a solution of the PCP.

We state the following theorem by Emil Post without proof.

Theorem 10.7 The PCP over Σ for $|\Sigma| \geq 2$ is unsolvable.

It is possible to reduce the PCP to many classes of two outputs (yes/no) problems in formal language theory. The following results can be proved by the reduction technique applied to PCP.

1. If L_1 and L_2 are any two context-free languages (type 2) over an alphabet Σ and $|\Sigma| \geq 2$, there is no algorithm to determine whether or not
   (a) $L_1 \cap L_2 = \emptyset$,
   (b) $L_1 \cap L_2$ is a context-free language,
   (c) $L_1 \subseteq L_2$, and
   (d) $L_1 = L_2$.
2. If G is a context-sensitive grammar (type 1), there is no algorithm to determine whether or not
   (a) $L(G) = \emptyset$,
   (b) $L(G)$ is infinite, and
   (c) $x_0 \in L(G)$ for a fixed string x_0.
3. If G is a type 0 grammar, there is no algorithm to determine whether or not any string $x \in \Sigma^*$ is in $L(G)$.

10.7 SUPPLEMENTARY EXAMPLES

EXAMPLE 10.4

If L is *a* recursive language over Σ, show that $\overline{L}$ ($\overline{L}$ is defined as $\Sigma^* - L$) is also recursive.

Solution

As L is recursive, there is a Turing machine M that halts and $T(M) = L$. We have to construct a TM M_1, such that $T(M_1) = \overline{L}$ and M_1 eventually halts. M_1 is obtained by modifying M as follows:

1. Accepting states of M are made nonaccepting states of M_1.
2. Let M_1 have a new state q_f. After reaching q_f, M_1 does not move in further transitions.
3. If q is a nonaccepting state of M and $\delta(q, x)$ is not defined, add a transition from q to q_f for M_1.

As M halts, M_1 also halts. (If M reaches an accepting state on w, then M_1 does not accept w and halts and conversely.)

Also M_1 accepts w if and only if M does not accept w. So $\overline{L}$ is recursive.

EXAMPLE 10.5

If L and $\overline{L}$ are both recursively enumerable, show that L and $\overline{L}$ are recursive.

Solution

Let M_1 and M_2 be two TMs such that $L = T(M_1)$ and $\overline{L} = T(M_2)$. We construct a new two-tape TM M that simulates M_1 on one tape and M_2 on the other.

If the input string w of M is in L, then M_1 accepts w and we declare that M accepts w. If $w \in \overline{L}$, then M_2 accepts w and we declare that M halts without accepting. Thus in both cases, M eventually halts. By the construction of M it is clear that $T(M) = T(M_1) = L$. Hence L is recursive. We can show that $\overline{L}$ is recursive, either by applying Example 10.4 or by interchanging the roles of M_1 and M_2 in defining acceptance by M.

EXAMPLE 10.6

Show that $\overline{A}_{TM}$ is not recursively enumerable.

Solution

We have already seen that A_{TM} is recursively enumerable (by Theorem 10.5). If $\overline{A}_{TM}$ were also recursively enumerable, then A_{TM} is recursive (by Example 10.5). This is a contradiction since A_{TM} is not recursive by Theorem 10.5. Hence $\overline{A}_{TM}$ is not recursively enumerable.

EXAMPLE 10.7

Show that the union of two recursively enumerable languages is recursively enumerable and the union of two recursive languages is recursive.

Solution

Let L_1 and L_2 be two recursive languages and M_1, M_2 be the corresponding TMs that halt. We design a TM M as a two-tape TM as follows:

1. w is an input string to M.
2. M copies w on its second tape.
3. M simulates M_1 on the first tape. If w is accepted by M_1, then M accepts w.
4. M simulates M_2 on the second tape. If w is accepted by M_2, then M accepts w.

M always halts for any input w.

Thus $L_1 \cup L_2 = T(M)$ and hence $L_1 \cup L_2$ is recursive.

If L_1 and L_2 are recursively enumerable, then the same conclusion gives a proof for $L_1 \cup L_2$ to be recursively enumerable. As M_1 and M_2 need not halt, M need not halt.

SELF-TEST

1. What is the difference between a recursive language and a recursively enumerable language?
2. The DFA M is given by

$$M = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \delta, q_0, \{q_0\})$$

where δ is defined by the transition Table 10.1.

TABLE 10.1 Transition Table for Self-Test 2

State	0	1
$\rightarrow q_0$	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

Answer the following:
(a) Is $(M, 001101)$ in A_{DFA}?
(b) Is $(M, 01010101)$ in A_{DFA}?
(c) Does $M \in A_{\text{DFA}}$?
(d) Find w such that $(M, w) \notin A_{\text{DFA}}$.

3. What do you mean by saying that the halting problem of TM is undecidable?
4. Describe A_{DFA}, A_{CFG}, A_{CSG}, A_{TM}, and $HALT_{\text{TM}}$.
5. Give one language from each of $\mathcal{L}_{\text{rl}}$, $\mathcal{L}_{\text{cfl}}$, $\mathcal{L}_{\text{csl}}$.
6. Give a language
   (a) which is in $\mathcal{L}_{\text{csl}}$ but not in $\mathcal{L}_{\text{rl}}$
   (b) which is in $\mathcal{L}_{\text{cfl}}$ but not in $\mathcal{L}_{\text{csl}}$
   (c) which is in $\mathcal{L}_{\text{cfl}}$ but not in $\mathcal{L}_{\text{rl}}$.

EXERCISES

10.1 Describe the Euclid's algorithm for finding the greatest common divisor of two natural numbers.

10.2 Show that $A_{\text{NDFA}} = \{(B, w) \mid B \text{ is an } N_{\text{DFA}} \text{ and } B \text{ accepts } w\}$ is decidable.

10.3 Show that $E_{\text{DFA}} = \{M \mid M \text{ is a } D_{\text{FA}} \text{ and } T(M) = \emptyset\}$ is decidable.

10.4 Show that $EQ_{\text{DFA}} = \{(A, B) \mid A \text{ and } B \text{ are DFAs and } T(A) = T(B)\}$ is decidable

10.5 Show that E_{CFG} is decidable (E_{CFG} is defined in a way similar to that of E_{DFA}).

10.6 Give an example of a language that is not recursive but recursively enumerable.

10.7 Do there exist languages that are not recursively enumerable?

10.8 Let L be a language over Σ. Show that only one of the following are possible for L and $\overline{L}$.
(a) Both L and $\overline{L}$ are recursive.
(b) Neither L nor $\overline{L}$ is recursive.
(c) L is recursively enumerable but $\overline{L}$ is not.
(d) $\overline{L}$ is recursively enumerable but L is not.

10.9 What is the difference between A_{TM} and $HALT_{\text{TM}}$?

10.10 Show that the set of all real numbers between 0 and 1 is uncountable. (A set S is uncountable if S is infinite and there is no one-to-one correspondence between S and the set of all natural numbers.)

10.11 Why should one study undecidability?

10.12 Prove that the recursiveness problem of type 0 grammar is unsolvable.

10.13 Prove that there exists a Turing machine M for which the halting problem is unsolvable.

10.14 Show that there exists a Turing machine M over $\{0, 1\}$ and a state q_m such that there is no algorithm to determine whether or not M will enter the state q_m when it begins with a given ID.

10.15 Prove that the problem of determining whether or not a TM over $\{0,1\}$ will ever print the symbol 1, with a given tape configuration, is unsolvable.

10.16 (a) Show that $\{x \mid x \text{ is a set and } x \notin x\}$ is not a set. (Note that this seems to be well-defined. This is one version of Russell's paradox.)
(b) A village barber shaves those who do not shave themselves but no others. Can he achieve his goal? For example, who is to shave the barber? (This is a popular version of Russell's paradox.)

Hints: (a) Let $S = \{x \mid x \text{ be a set and } x \notin x\}$. If S were a set, then $S \in S$ or $S \notin S$. If $S \notin S$ by the 'definition' of S, then $S \in S$. On the other hand, if $S \in S$ by the 'definition' of S, then $S \notin S$. Thus we can neither assert that $S \notin S$ nor $S \in S$. (This is Russell's paradox.) Therefore, S is not a set.

(b) Let $S = \{x \mid x \text{ be a person and } x \text{ does not shave himself}\}$. Let b denote the barber. Examine whether $b \in S$. (The argument is similar to that given for (a).) It will be instructive to read the proof of HP of Turing machines and this example, in order to grasp the similarity.

10.17 Comment on the following: "We have developed an algorithm so complicated that no Turing machine can be constructed to execute the algorithm no matter how much (tape) space and time is allowed."

10.18 Prove that PCP is solvable if $|\Sigma| = 1$.

10.19 Let $x = (x_1 \ldots x_n)$ and $y = (y_1 \ldots y_n)$ be two lists of nonempty strings over Σ and $|\Sigma| \geq 2$. (i) Is PCP solvable for $n = 1$? (ii) Is PCP solvable for $n = 2$?

10.20 Prove that the PCP with $\{(01, 011), (1, 10), (1, 11)\}$ has no solution. (Here, $x_1 = 01$, $x_2 = 1$, $x_3 = 1$, $y_1 = 011$, $y_2 = 10$, $y_3 = 11$.)

10.21 Show that the PCP with $S = \{(0, 10), (1^20, 0^3), (0^21, 10)\}$ has no solution. [*Hint:* No pair has common nonempty initial substring.]

10.22 Does the PCP with $x = (b^3, ab^2)$ and $y = (b^3, bab^3)$ have a solution?

10.23 Find at least three solutions to PCP defined by the dominoes:

1	10	10111
111	0	10

10.24 (a) Can you simulate a Turing machine on a general-purpose computer? Explain.

(b) Can you simulate a general-purpose computer on a Turing machine? Explain.

11 Computability

In this chapter we shall discuss the class of primitive recursive functions—a subclass of partial recursive functions. The Turing machine is viewed as a mathematical model of a partial recursive function.

11.1 INTRODUCTION AND BASIC CONCEPTS

In Chapters 5, 7 and 9, we considered automata as the accepting devices. In this chapter we will study automata as the computing machines. The problem of finding out whether a given problem is 'solvable' by automata reduces to the evaluation of functions on the set of natural numbers or a given alphabet by mechanical means.

We start with the definition of partial and total functions.

A partial function f from X to Y is a rule which assigns to every element of X at most one element of Y.

A total function from X to Y is a rule which assigns to every element of X a unique element of Y. For example, if R denotes the set of all real numbers, the rule f from R to itself given by $f(r) = +\sqrt{r}$ is a partial function since $f(r)$ is not defined as a real number when r is negative. But $g(r) = 2r$ is a total function from R to itself. (Note that all the functions considered in the earlier chapters were total functions.)

In this chapter we consider total functions from X^k to X, where $X = \{0, 1, 2, 3, \ldots\}$ or $X = \{a, b\}^*$. Throughout this chapter we denote $(0, 1, 2, \ldots)$ by N and (a, b) by Σ. (Recall that X^k is the set of all k-tuples of elements of X.) For example, $f(m, n) = m - n$ defines a partial function from N to itself as $f(m, n)$ is not defined when $m - n < 0$; $g(m, n) = m + n$ defines a total function from N to itself.

Remark A partial or total function f from X^k to X is also called a function of k variables and denoted by $f(x_1, x_2, \ldots, x_k)$. For example, $f(x_1, x_2) = 2x_1 + x_2$ is a function of two variables: $f(1, 2) = 4$, 1 and 2 are called arguments and 4 is called a value. $g(w_1, w_2) = w_1w_2$ is a function of two variables ($w_1w_2 \in \Sigma^*$); $g(ab, aa) = abaa$, ab, aa are called arguments and $abaa$ is a value.

11.2 PRIMITIVE RECURSIVE FUNCTIONS

In this section we construct primitive recursive functions over N and Σ. We define some initial functions and declare them as primitive recursive functions. By applying certain operations on the primitive recursive functions obtained so far, we get the class of primitive recursive functions.

11.2.1 INITIAL FUNCTIONS

The initial functions over N are given in Table 11.1. In particular,

$$S(4) = 5, \quad Z(7) = 0$$

$$U_2^3(2, 4, 7) = 4, \quad U_1^3(2, 4, 7) = 2, \quad U_3^3(2, 4, 7) = 7$$

TABLE 11.1 Initial Functions Over N

Zero function Z defined by $Z(x) = 0$.
Successor function S defined by $S(x) = x + 1$.
Projection function U_i^n defined by $U_i^n(x_1, \ldots, x_n) = x_i$.

Note: As $U_1^1(x) = x$ for every x in N, U_1^1 is simply the identity function. So U_i^n is also termed a generalized identity function.

The initial functions over Σ are given in Table 11.2. In particular,

$$\text{nil }(abab) = \Lambda$$

$$\text{cons } a(abab) = aabab$$

$$\text{cons } b(abab) = babab$$

Note: We note that cons $a(x)$ and cons $b(x)$ simply denote the concatenation of the 'constant' string a and x and the concatenation of the constant string b and x.

TABLE 11.2 Initial Functions Over $\{a, b\}$

nil $(x) = \Lambda$
cons $a(x) = ax$
cons $b(x) = bx$

In the following definition, we introduce an operation on functions over X.

Definition 11.1 If $f_1, f_2, \ldots, f_k$ are partial functions of n variables and g is a partial function of k variables, then the composition of g with $f_1, f_2, \ldots, f_k$ is a partial function of n variables defined by

$$g(f_1(x_1, x_1, \ldots, x_n), f_2(x_1, x_2, \ldots, x_n), \ldots, f_k(x_1, x_2, \ldots, x_n))$$

If, for example, f_1, f_2 and f_3 are partial functions of two variables and g is a partial function of three variables, then the composition of g with f_1, f_2, f_3 is given by $g(f_1(x_1, x_2), f_2(x_1, x_2), f_3(x_1, x_2))$.

EXAMPLE 11.1

Let $f_1(x, y) = x + y$, $f_2(x, y) = 2x$, $f_3(x, y) = xy$ and $g(x, y, z) = x + y + z$ be functions over N. Then

$$g(f_1(x, y), f_2(x, y), f_3(x, y)) = g(x + y, 2x, xy)$$
$$= x + y + 2x + xy$$

Thus the composition of g with f_1, f_2, f_3 is given by a function h:

$$h(x, y) = x + y + 2x + xy$$

Note: Definition 11.1 generalizes the composition of two functions. The concept is useful where a number of outputs become the inputs for a subsequent step of a program.

The composition of g with $f_1, \ldots, f_n$ is total when $g, f_1, f_2, \ldots, f_n$ are total. The function given in Example 11.1 is total as f_1, f_2, f_3 and g are total.

EXAMPLE 11.2

Let $f_1\ (x, y) = x - y$, $f_2(x, y) = y - x$ and $g(x, y) = x + y$ be functions over N. The function f_1 is defined only when $x \geq y$ and f_2 is defined only when $y \geq x$. So f_1 and f_2 are defined only when $x = y$. Hence when $x = y$,

$$g(f_1(x, y), f_2(x, y)) = g(x - x, x - x) = g(0, 0) = 0$$

Thus the composition of g with f_1 and f_2 is defined only for (x, x), where $x \in N$.

EXAMPLE 11.3

Let $f_1(x_1, x_2) = x_1x_2$, $f_2(x_1, x_2) = \Lambda$, $f_3(x_1, x_2) = x_1$, and $g(x_1, x_2, x_3) = x_2x_3$ be functions over Σ. Then

$$g(f_1(x_1, x_2), f_2(x_1, x_2), f_3(x_1, x_2)) = g(x_1x_2, \Lambda, x_1) = \Lambda x_1 = x_1$$

So the composition of g with f_1, f_2, f_3 is given by a function h, where $h(x_1, x_2) = x_1$.

The next definition gives a mechanical process of computing a function.

Definition 11.2 A function $f(x)$ over N is defined by recursion if there exists a constant k (a natural number) and a function $h(x, y)$ such that

$$f(0) = k, \qquad f(n + 1) = h(n, f(n)) \tag{11.1}$$

By induction on n, we can define $f(n)$ for all n. As $f(0) = k$, there is basis for induction. Once $f(n)$ is known, $f(n + 1)$ can be evaluated by using (11.1).

EXAMPLE 11.4

Define $n!$ by recursion.

Solution

$f(0) = 1$ and $f(n + 1) = h(n, f(n))$, where $h(x, y) = S(x) * y$.

The above definition can be generalized for $f(x_1, x_2, \ldots, x_n, x_{n+1})$. We fix n variables in $f(x_1, x_2, \ldots, x_{n+1})$, say, $x_1, x_2, \ldots, x_n$. We apply Definition 11.2 to $f(x_1, x_2, \ldots, x_n, y)$. In place of k we get a function $g(x_1, x_2, \ldots, x_n)$ and in place of $h(x, y)$, we obtain $h(x_1, x_2, \ldots, x_n, y, f(x_1, \ldots, x_n, y))$.

Definition 11.3 A function f of $n + 1$ variables is defined by recursion if there exists a function g of n variables, and a function h of $n + 2$ variables, and f is defined as follows:

$$f(x_1, x_2, \ldots, x_n, 0) = g(x_1, x_2, \ldots, x_n) \tag{11.2}$$

$$f(x_1 \ldots x_n, y + 1) = h(x_1, x_2, \ldots, x_n, y, f(x_1, x_2, \ldots, x_n, y)) \tag{11.3}$$

We may note that f can be evaluated for all arguments $(x_1, x_2, \ldots, x_n, y)$ by induction on y for fixed $x_1, x_2, \ldots, x_n$. The process is repeated for every $x_1, x_2, \ldots, x_n$.

Now we can define the primitive recursive functions over N.

11.2.2 PRIMITIVE RECURSIVE FUNCTIONS OVER *N*

Definition 11.4 A total function f over N is called primitive recursive (i) if it is any one of the three initial functions, or (ii) if it can be obtained by applying composition and recursion a finite number of times to the set of initial functions.

EXAMPLE 11.5

Show that the function $f_1(x, y) = x + y$ is primitive recursive.

Solution

f_1 is a function of two variables. If we want f_1 to be defined by recursion, we need a function g of a single variable and a function h of three variables.

$$f_1(x, 0) = x + 0 = x$$

By comparing $f_1(x, 0)$ with L.H.S. of (11.2), we see that g can be defined by

$$g(x) = x = U_1^1(x)$$

Also, $f_1(x, y + 1) = x + (y + 1) = (x + y) + 1 = f_1(x, y) + 1$
By comparing $f_1(x, y + 1)$ with L.H.S. of (11.3), we have

$$h(x, y, f_1(x, y)) = f_1(x, y) + 1 = S(f_1(x, y)) = S(U_3^3(x, y, f_1(x, y)))$$

Define $h(x, y, z) = S(U_3^3(x, y, z))$. As $g = U_1^1$, it is an initial function. The function h is obtained from the initial functions U_3^3 and S by composition, and by recursion using g and h. Thus f_1 is obtained by applying composition and recursion a finite number of times to initial functions U_1^1, U_3^3 and S. So f_1 is primitive recursive.

Note: A total function is primitive recursive if it can be obtained by applying composition and recursion a finite number of times to primitive recursive functions $f_1, f_2, \ldots, f_m$. This is clear as each f_i is obtained by applying composition and recursion a finite number of times to initial functions.

EXAMPLE 11.6

The function $f_2(x, y) = x * y$ is primitive recursive.

Solution

As multiplication of two natural numbers is simply repeated addition, f_2 has to be primitive recursive. We prove this as follows:

$$f_2(x, 0) = 0, \qquad f_2(x, y + 1) = x * (y + 1) = f_2(x, y) + x$$

i.e. $f_2(x, y + 1) = f_1(f_2(x, y), x)$. Comparing these with (11.2) and (11.3), we can write

$$f_2(x, 0) = Z(x) \text{ and } f_2(x, y + 1) = f_1(U_3^3(x, y, f_2(x, y)), U_1^3(x, y, f_2(x, y))))$$

By taking $g = Z$ and h defined by

$$h(x, y, z) = f_1(U_3^3(x, y, z), U_1^3(x, y, z))$$

we see that f_2 is defined by recursion. As g and h are primitive recursive, f_2 is primitive recursive (by the above note).

EXAMPLE 11.7

Show that $f(x, y) = x^y$ is a primitive recursive function.

Solution

We define

$$f(x, 0) = 1$$
$$f(x, y + 1) = x * f(x, y)$$
$$= U_1^3(x, y, f(x, y)) * U_3^3(x, y, f(x, y))$$

Therefore, $f(x, y)$ is primitive recursive.

EXAMPLE 11.8

Show that the following functions are primitive recursive:

(a) The predecessor function $p(x)$ defined by

$$p(x) = x - 1 \quad \text{if } x \neq 0, \qquad p(x) = 0 \quad \text{if } x = 0.$$

(b) The proper subtraction function $\dot{-}$ defined by

$$x \dot{-} y = x - y \quad \text{if } x \geq y \quad \text{and} \quad x \dot{-} y = 0 \quad \text{if } x < y.$$

(c) The absolute value function $|\ |$ given by

$$|x| = x \quad \text{if } x \geq 0, \qquad |x| = -x \quad \text{if } x < 0.$$

(d) min (x, y), i.e. minimum of x and y.

Solution

(a) $p(0) = 0$ and $p(y + 1) = U_1^2(y, p(y))$

(b) $x \dot{-} 0 = x$ and $x \dot{-} (y + 1) = p(x \dot{-} y)$

(c) $|x - y| = (x \dot{-} y) + (y \dot{-} x)$

(d) $\min(x, y) = x \dot{-} (x \dot{-} y)$

The first function is defined by recursion using an initial function. So it is primitive recursive.

The second function is defined by recursion using the primitive recursive function p and so it is primitive recursive. Similarly, the last two functions are primitive recursive.

11.2.3 PRIMITIVE RECURSIVE FUNCTIONS OVER $\{a, b\}$

For constructing the primitive recursive function over $\{a, b\}$, the process is similar to that of function over N except for some minor modifications. It should be noted that Λ plays the role of 0 in (11.2) and ax or bx plays the role of $y + 1$ in (11.3). Recall that Σ denotes $\{a, b\}$.

Definition 11.5 A function $f(x)$ over Σ is defined by recursion if there exists a 'constant' string $w \in \Sigma^*$ and functions $h_1(x, y)$ and $h_2(x, y)$ such that

$$f(\Lambda) = w \tag{11.4}$$

$$f(ax) = h_1(x, f(x)) \tag{11.5}$$

$$f(bx) = h_2(x, f(x))$$

(h_1 and h_2 may be functions in one variable.)

Definition 11.6 A function $f(x_1, x_2, \ldots, x_n)$ over Σ is defined by recursion if there exist functions $g(x_1, \ldots, x_{n-1})$, $h_1(x_1, \ldots, x_{n+1})$, $h_2(x_1, \ldots, x_{n+1})$, such that

$$f(\Lambda, x_2, \ldots, x_n) = g(x_2, \ldots, x_n) \tag{11.6}$$

$$f(ax_1, x_2, \ldots, x_n) = h_1(x_1, x_2, \ldots, x_n, f(x_1, x_2, \ldots, x_n)) \tag{11.7}$$

$$f(bx_1, x_2, \ldots, x_n) = h_2(x_1, x_2, \ldots, x_n, f(x_1, x_2, \ldots, x_n))$$

(h_1 and h_2 may be functions of m variables, where $m < n + 1$.)

Now we can define the class of primitive recursive functions over Σ.

Definition 11.7 A total function f is primitive recursive (i) if it is any one of the three initial functions (given in Table 11.2), or (ii) if it can be obtained by applying composition and recursion a finite number of times to the initial functions.

In Example 11.9 we give some primitive recursive functions over Σ.

Note: As in the case of functions over N, a total function over Σ is primitive recursive if it is obtained by applying composition and recursion a finite number of times to primitive recursive function $f_1, f_2, \ldots, f_m$.

EXAMPLE 11.9

Show that the following functions are primitive recursive:

(a) Constant functions a and b (i.e. $a(x) = a$, $b(x) = b$)
(b) Identity function
(c) Concatenation
(d) Transpose
(e) Head function (i.e. head $(a_1a_2 \ldots, a_n) = a_1$)
(f) Tail function (i.e. tail $(a_1a_2 \ldots a_n) = a_2 \ldots, a_n$)
(g) The conditional function "if $x_1 \neq \Lambda$, then x_2 else x_3."

Solution

(a) As $a(x) =$ cons a (nil (x)), the function $a(x)$ is the composition of the initial function cons a with the initial function nil and is hence primitive recursive.

(b) Let us denote the identity function by id. Then,

$$\text{id}(\Lambda) = \Lambda$$

$$\text{id}(ax) = \text{cons } a(x)$$

$$\text{id}(bx) = \text{cons } b(x)$$

So id is defined by recursion using cons a and cons b. Therefore, the identity function is primitive recursive.

(c) The concatenation function can be defined by

$$\text{concat}(x_1, x_2) = x_1x_2$$

$$\text{concat}(\Lambda, x_2) = \text{id}(x_2)$$

$$\text{concat}(ax_1, x_2) = \text{cons } a \ (\text{concat}(x_1, x_2))$$

$$\text{concat}(bx_1, x_2) = \text{cons } b \ (\text{concat}(x_1, x_2))$$

So concat is defined by recursion using id, cons a and cons b. Therefore, concat is primitive recursive.

(d) The transpose function can be defined by trans(x) = x^T. Then

$$\text{trans}(\Lambda) = \Lambda$$

$$\text{trans}(ax) = \text{concat}(\text{trans}(x), a(x))$$

$$\text{trans}(bx) = \text{concat}(\text{trans}(x), b(x))$$

Therefore, trans(x) is primitive recursive.

(e) The head function head(x) satisfies

$$\text{head}(\Lambda) = \Lambda$$

$$\text{head}(ax) = a(x)$$

$$\text{head}(bx) = b(x)$$

Therefore, head(x) is primitive recursive.

(f) The tail function tail(x) satisfies

$$\text{tail}(\Lambda) = \Lambda$$

$$\text{tail}(ax) = \text{id}(x)$$

$$\text{tail}(bx) = \text{id}(x)$$

Therefore, tail(x) is primitive recursive.

(g) The conditional function can be defined by

$$\text{cond}(x_1, x_2, x_3) = \text{"if } x_1 \neq \Lambda \text{ then } x_2 \text{ else } x_3\text{"}$$

Then,

$$\text{cond}(\Lambda, x_2, x_3) = \text{id}(x_3)$$

$$\text{cond}(ax_1, x_2, x_3) = \text{id}(x_2)$$

$$\text{cond}(bx_1, x_2, x_3) = \text{id}(x_2)$$

Therefore, id(x_1, x_2, x_3) is primitive recursive.

11.3 RECURSIVE FUNCTIONS

By introducing one more operation on functions, we define the class of recursive functions, which includes the class of primitive recursive functions.

Definition 11.8 Let $g(x_1, x_2, \ldots, x_n, y)$ be a total function over N. g is a regular function if there exists some natural number y_0 such that $g(x_1, x_2, \ldots, x_n, y_0) = 0$ for all values $x_1, x_2, \ldots, x_n$ in N.

For instance, $g(x, y) = \min(x, y)$ is a regular function since $g(x, 0) = 0$ for all x in N. But $f(x, y) = |x - y|$ is not regular since $f(x, y) = 0$ only when $x = y$, and so we cannot find a fixed y such that $f(x, y) = 0$ for all x in N.

Definition 11.9 A function $f(x_1, x_2, \ldots, x_n)$ over N is defined from a total function $g(x_1, x_2, \ldots, x_n, y)$ by minimization if

(a) $f(x_1, x_2, \ldots, x_n)$ is the least value of all y's such that $g(x_1, x_2, \ldots, x_n, y) = 0$ if it exists. The least value is denoted by $\mu_y(g(x_1, x_2, \ldots, x_n, y) = 0)$.
(b) $f(x_1, x_2, \ldots, x_n)$ is undefined if there is no y such that $g(x_1, x_2 \ldots x_n, y) = 0$.

Note: In general, f is partial. But, if g is regular then f is total.

Definition 11.10 A function is recursive if it can be obtained from the initial functions by a finite number of applications of composition, recursion and minimization over regular functions.

Definition 11.11 A function is partial recursive if it can be obtained from the initial functions by a finite number of applications of composition, recursion and minimization.

EXAMPLE 11.10

$f(x) = x/2$ is a partial recursive function over N.

Solution

Let $g(x, y) = |2y - x|$, where $2y - x = 0$ for some y only when x is even. Let $f_1(x) = \mu_y(|2y - x| = 0)$. Then $f_1(x)$ is defined only for even values of x and is equal to $x/2$. When x is odd, $f_1(x)$ is not defined. f_1 is partial recursive. As $f(x) = x/2 = f_1(x)$, f is a partial recursive function.

The following example gives a recursive function which is not primitive recursive.

EXAMPLE 11.11

The Ackermann's function is defined by

$$A(0, y) = y + 1 \tag{11.8}$$

$$A(x + 1, 0) = A(x, 1) \tag{11.9}$$

$$A(x + 1, y + 1) = A(x, A(x + 1, y)) \tag{11.10}$$

$A(x, y)$ can be computed for every (x, y), and hence $A(x, y)$ is total. The Ackermann's function is not primitive recursive but recursive.

EXAMPLE 11.12

Compute $A(1, 1)$, $A(2, 1)$, $A(1, 2)$, $A(2, 2)$.

Solution

$$\begin{aligned}
A(1, 1) &= A(0 + 1, 0 + 1) & \\
&= A(0, A(1, 0)) & \text{by } (11.10) \\
&= A(0, A(0, 1)) & \text{by } (11.9) \\
&= A(0, 2) & \text{by } (11.8) \\
&= 3 & \text{by } (11.8) \\
A(1, 2) &= A(0 + 1, 1 + 1) & \\
&= A(0, A(1, 1)) & \text{by } (11.10) \\
&= A(0, 3) & \\
&= 4 & \text{by } (11.8) \\
A(2, 1) &= A(1 + 1, 0 + 1) & \\
&= A(1, A(2, 0)) & \text{by } (11.10) \\
&= A(1, A(1, 1)) & \text{by } (11.9) \\
&= A(1, 3) & \\
&= A(0 + 1, 2 + 1) & \\
&= A(0, A(1, 2)) & \text{by } (11.10) \\
&= A(0, 4) & \\
&= 5 & \\
A(2, 2) &= A(1 + 1, 1 + 1) & \\
&= A(1, A(2, 1)) & \text{by } (11.10) \\
&= A(1, 5) & \\
A(1, 5) &= A(0 + 1, 4 + 1) & \\
&= A(0, A(1, 4)) & \text{by } (11.10) \\
&= 1 + A(1, 4) & \text{by } (11.8) \\
&= 1 + A(0 + 1, 3 + 1) & \\
&= 1 + A(0, A(1, 3)) & \\
&= 1 + 1 + A(1, 3) & \\
&= 1 + 1 + 1 + A(1, 2) = 1 + 1 + 1 + 4 & \\
&= 7 &
\end{aligned}$$

As $A(2, 2) = A(1, 5)$, we have $A(2, 2) = 7$

So far we have dealt with recursive and partial recursive functions over N. We can define partial recursive functions over Σ using the primitive recursive predicates and the minimization process. As the process is similar, we will discuss it here.

The concept of recursion occurs in some programming languages when a procedure has a call to the same procedure for a different parameter. Such a procedure is called a recursive procedure. Certain programming languages like C, C++ allow recursive procedures.

11.4 PARTIAL RECURSIVE FUNCTIONS AND TURING MACHINES

In this section we prove that partial recursive functions introduced in the earlier sections are Turing-computable.

11.4.1 COMPUTABILITY

In mid 1930s, mathematicians and logicians were trying to rigorously define computability and algorithms. In 1934 Kurt Gödel pointed out that primitive recursive functions can be computed by a finite procedure (i.e. an algorithm). He also hypothesized that any function computable by a finite procedure can be specified by a recursive function. Around 1936, Turing and Church independently designed a 'computing machine' (later termed *Turing machine*) which can carry out a finite procedure.

For formalizing computability, Turing assumed that, while computing, a person writes symbols on a one-dimensional paper (instead of a two-dimensional paper as is usually done) which can be viewed as a tape divided into cells. He scans the cells one at a time and usually performs one of the three simple operations, namely (i) writing a new symbol in the cell he is scanning, (ii) moving to the cell left of the present cell, and (iii) moving to the cell right of the present cell. These observations led Turing to propose a computing machine. The Turing machine model we have introduced in Chapter 9 is based on these three simple operations but with slight variations. In order to introduce computability, we consider the Turing machine model due to Post. In the present model the transition function is represented by a set of quadruples (i.e. 4-tuples), whereas the transition function of the model we have introduced in Chapter 9 can be represented by a set of quintuples (5-tuples). For example, $\delta(q_i, a) = (q_j, \alpha, \beta)$ is represented by the quintuple $q_j a \alpha \beta q_j$. Using the model specifying the transition function in terms of quadruples, we define Turing-computable functions and prove that partially recursive functions are Turing-computable.

11.4.2 A Turing Model for Computation

As in the model introduced in Chapter 9, Q, q_0 and Γ denote the set of states, the initial state, and the set of tape symbols, respectively. The blank symbol b is in Γ. The only difference is in the transition function. In the present model the transition function represents only one of the following three basic operations:

(i) Writing a new symbol in the cell scanned
(ii) Moving to the left cell
(iii) Moving to the right cell

Each operation is followed by a change of state. Suppose the Turing machine M is in state q and scans a_i. If a_j is written and M enters q', then this basic operation is represented by the quadruple qa_ia_jq'. Similarly, the other two operations are represented by the quadruples qa_iLq' and qa_iRq'. Thus the transition function can be specified by a set P of quadruples. As in Chapter 9, we can define instantaneous descriptions, i.e. IDs.

Each quadruple induces a change of IDs. For example, qa_ia_jq' induces

$$\alpha qa_i\beta \vdash \alpha q' a_j\beta$$

The quadruple qa_iLq' induces

$$a_1a_2 \ldots a_{i-1}qa_i \ldots a_n \vdash a_1a_2 \ldots a_{i-2}q'a_{i-1}a_i \ldots a_n$$

and qa_iRq' induces

$$a_1 \ldots a_{i-1}qa_i \ldots a_n \vdash a_1 \ldots a_iq'a_{i+1} \ldots a_n$$

When we require M to perform some computation, we 'feed' the input by initial tape expression denoted by X. So q_0X is the initial ID for the given input. For computing with the given input X, the Turing machine processes X using appropriate quadruples in P. As a result, we have $q_0X = \text{ID}_i \vdash \text{ID}_2 \vdash \ldots$. When an ID, say ID_n, is reached, which cannot be changed using any quadruple in P, M halts. In this case, ID_n is called a terminal ID. Actually, $aq_1\alpha\beta$ is a terminal ID if there is no quadruple starting with q_ia. The terminal ID is called the result of X and denoted by Res(X). The computed value corresponding to input X can be obtained by deleting the state appearing in it as also some more symbols from Res(X).

11.4.3 Turing-computable Functions

Before developing the concept of Turing-computable functions, let us recall Example 9.6. The TM developed in Example 9.6 concatenates two strings α and β. Initially, α and β appear on the input tape separated by a blank b. Finally, the concatenated string $\alpha\beta$ appears on the input tape. The same method can be adopted with slight modifications for computing $f(x_1, \ldots, x_m)$. Suppose we want to construct a TM which can compute $f(x_1, \ldots, x_m)$ over

N for given arguments $a_1, \ldots, a_m$. Initially, the input $a_1, a_2, \ldots, a_m$ appears on the input tape separated by markers $x_1, \ldots, x_m$. The computed value $f(a_1, \ldots, a_m)$, say, c appears on the input tape , once the computation is over. To locate c we need another marker, say y. The value c appears to the right of x_m and to the left of y. To make the construction simpler, we use the tally notation to represent the elements of N. In the tally notation, 0 is represented by a string of b's. A positive integer n is represented by a string consisting of n 1's. So the initial tape expression takes the form $1^{a_1}x_1 1^{a_2}x_2 \ldots 1^{a_m}x_m by$. As a result of computation, the initial ID $q_0 1^{a_1}x_1 1^{a_2}x_2 \ldots 1^{a_m}x_m\, by$ is changed to a terminal ID of the form $1^{a_1}x_1 1^{a_2}x_2 \ldots 1^{a_m}x_m 1^c q' y$ for some $q' \in Q$. In fact, the position of q' in a terminal ID is immaterial and it can appear anywhere in Res(X). The computed value is found between x_m and y. Sometimes we may have to omit the leading b's.

We say that a function $f(x_1, \ldots, x_m)$ is Turing-computable for arguments $a_1, \ldots, a_m$ if there exists a Turing machine for which

$$q_0 1^{a_1}x_1 1^{a_2}x_2 \ldots 1^{a_m}x_m by \overset{*}{\vdash} \text{ID}_n$$

where ID_n is a terminal ID containing $f(a_1, \ldots, a_m)$ to the left of y.

Our ultimate aim is to prove that partial recursive functions are Turing-computable. For this purpose, first of all we prove that the three initial primitive recursive functions are Turing-computable.

11.4.4 Construction of the Turing Machine That Can Compute the Zero Function Z

The zero function Z is defined as $Z(a_1) = 0$ for all $a_1 \geq 0$. So the initial tape expression can be taken as $X = 1^{a_1}x_1 by$. As we require the computed value $Z(a_1)$, namely 0, to appear to the left of y, we require the machine to halt without changing the input. (Note that 0 is represented by b in the tally notation.)

Thus we define a TM by taking $Q = \{q_0, q_1\}$, $\Gamma = \{b, 1, x_1, y\}$, $X = 1^{a_1}x_1 by$. P consists of $q_0 bRq_0$, $q_0 1Rq_0$, $q_0 x_1 x_1 q_1$. $q_0 bRq_0$ and $q_0 1Rq_0$ are used to move to the right until x_1 is encountered. $q_0 x_1 x_1 q_1$ enables the TM to enter the state q_1. M enters q_1 without altering the tape symbol. In terms of change of IDs, we have

$$q_0 1^{a_1}x_1 by \overset{*}{\vdash} 1^{a_1}q_0 x_1 by \vdash 1^{a_1}q_1 x_1 by$$

As there is no quadruple starting with q_1, M halts and Res(X) = $1^{a_1}q_1 x_1 by$. By deleting q_1 in Res(X), we get $1^{a_1}x_1 by$ (which is the same as X) yielding 0 (given by b).

Note: We can also represent the quadruples in a tabular form which is similar to the transition table obtained in Chapter 9. In this case we have to specify (i) the new symbol written, or (ii) the movement to the left (denoted by L), or (iii) the movement to the right (denoted by R). So we get Table 11.3.

TABLE 11.3 Representation of Quadruples

State	*b*	*1*	x_1	*y*
q_0	(R, q_0)	(R, q_0)	(x_1, q_1)	
q_1				

11.4.5 Construction of the Turing Machine for Computing—The Successor Function

The successor function S is defined by $S(a_1) = a_1 + 1$ for all $a_1 \geq 0$. So the initial tape expression can be taken as $X = 1^{a_1}x_1by$ (as in the case of the zero function). At the end of the computation, we require 1^{a_1+1} to appear to the left of y. Hence we define a TM by taking

$$Q = \{q_0, \ldots, q_9\}, \qquad \Gamma = \{b, 1, x_1, y\}, \qquad X = 1^{a_1}x_1\,by$$

where P consists of

(i) q_0bRq_0, q_01bq_1, $q_0x_1Rq_6$
(ii) q_1bRq_1, q_11Rq_1, $q_1x_1Rq_1$, q_1y1q_2
(iii) q_21Rq_2, q_2byq_3,
(iv) q_3bLq_3, q_31Lq_3, q_3yLq_3, $q_3x_1Lq_4$
(v) q_41Lq_4, q_4b1q_5,
(vi) q_51Rq_0.
(vii) q_6bRq_6, q_61Rq_6, $q_6x_1Rq_6$, q_6yLq_7
(viii) q_71Lq_7, q_7b1q_8,
(ix) q_8bLq_8, q_81Lq_8, q_8yLq_8, $q_8x_1x_1q_9$.

The corresponding operations can be explained as follows:

(i) If M starts from the initial ID, the head replaces the first 1 it encounters by b. Afterwards the head moves to the right until it encounters y (as a result of q_01bq_1, q_1bRq_1, q_11Rq_1, $q_1x_1Rq_1$).
(ii) y is replaced by 1 and M enters q_2. Once the end of the input tape is reached, y is added to the next cell. M enters q_3 (q_1y1q_2, q_21Rq_2, q_2byq_3).
(iii) Then the head moves to the left and the state is not changed until x_1 is encountered (q_3yLq_3, q_3yLq_3, q_3bLq_3).
(iv) On encountering x_1, the head moves to the left and M enters q_4. Once again the head moves to the left till the left end of the input string is reached ($q_3x_1Lq_4$, q_11Lq_4).
(v) The leftmost blank (written in point (i)) is replaced by 1 and M enters q_5 (q_4b1q_5).

Thus at the end of operations (i)–(v), the input part remains unaffected but the first 1 is added to the left of y.

(vi) Then the head scans the second 1 of the input string and moves right, and M enters q_0 ($q_5 1 R q_0$).

Operations (i)–(vi) are repeated until all the 1's of the input part (i.e. in 1^{a_1}) are exhausted and $11 \ldots 1$ (a_1 times) appear to the left of y. Now the present state is q_0, and the current symbol is x_1.

(vii) M in state q_0 scans x_1, moves right, and enters q_6. It continues to move to the right until it encounters y ($q_0 x_1 R q_6$, $q_6 b R q_6$, $q_6 1 R q_6$, $q_6 x_1 R q_6$).

(viii) On encountering y, the head moves to the left and M enters q_7, after which the head moves to the left until it encounters b appearing to the left of 1^{a_1} of the output part. This b is changed to 1, and M enters q_8 ($q_6 y L q_7$, $q_1 1 L q_7$, $q_7 b_1 q_8$).

(ix) Once M is in q_8, the head continues to move to the left and on scanning x_1, M enters q_9. As there is no quadruple starting with q_9, M halts ($q_8 b L q_8$, $q_8 1 L q_8$, $q_8 x_1 x_1 q_9$).

The machine halts, and the terminal ID is $1^{a_1} q_9 x_1 1^{a_1+1} y$. For example, let us compute $S(1)$. In this case the initial ID is $q_0 1 x_1 b y$. As a result of the computation, we have the following moves:

$$
\begin{aligned}
& q_0 1 x_1 b y \vdash q_1 b x_1 b y \vdash b q_1 x_1 b y \\
\vdash\; & b x_1 q_1 b y \vdash b x_1 b q_1 y \vdash b x_1 b q_2 1 \\
\vdash\; & b x_1 b 1 q_2 b \vdash b x_1 b 1 q_3 y \vdash b x_1 b q_3 1 y \\
\overset{*}{\vdash}\; & b q_3 x_1 b 1 y \vdash q_4 b x_1 b 1 y \vdash q_5 1 x_1 b 1 y \\
\overset{*}{\vdash}\; & 1 q_6 x_1 b 1 y \overset{*}{\vdash} 1 x_1 b 1 q_6 y \vdash 1 x_1 b q_7 1 y \\
\vdash\; & 1 x_1 q_7 b 1 y \vdash 1 x_1 q_8 1 1 y \vdash 1 q_8 x_1 1 1 y \\
\vdash\; & 1 q_9 x_1 1 1 y
\end{aligned}
$$

Thus, M halts and $S(1) = 2$ (given by 11 to the left of y).

11.4.6 CONSTRUCTION OF THE TURING MACHINE FOR COMPUTING THE PROJECTION U_i^m

Recall $U_i^m(a_1, \ldots, a_m) = a_i$. The initial tape expression can be taken as

$$X = 1^{a_1} x_1 1^{a_2} x_2 \ldots 1^{a_m} x_m\, b y$$

We define a Turing machine by taking $Q = \{q_0, \ldots, q_8\}$

$\Gamma = \{b, 1, x_1, \ldots, x_m, y\}$. P consists of

$$
\begin{array}{ll}
q_0 z R q_0 & \text{for all } z \in \Gamma - \{x_i\} \\
q_0 x_i L q_1, & q_1 b b q_8, \quad q_1 1 b q_2 \\
q_2 z R q_2 & \text{for all } z \in \Gamma - \{y\} \\
q_2 y 1 q_3, & q_3 1 R q_3, \quad q_3 b y q_4
\end{array}
$$

$$\begin{array}{lllll} q_4 zLq_4 & \text{for all } z \in \Gamma - \{x_i\} & & & \\ q_4 x_i Lq_5, & q_5 1Lq_5, & q_5 b1q_6, & q_6 1Lq_7, & q_7 1bq_2 \\ q_7 zRq_8 & \text{for all } z \in \Gamma - \{1\} & & & \end{array}$$

The operations of M are as follows:

(i) M starts from the initial ID and the head moves to the right until it encounters x_i $(q_0 zRq_0)$.

(ii) On seeing x_i, the head moves to the left $(q_0 x_i Lq_1)$.

(iii) The head replaces 1 (the rightmost 1 in 1^{a_i}) by b $(q_1 1bq_2)$.

(iv) The head moves to the right until it encounters y and replaces y by 1 $(q_2 zRq_2,\ z \in \Gamma - \{y\}$ and $q_2 y1q_3)$.

(v) On reaching the right end, the head scans b and replaces this b by y $(q_3 byq_4)$.

(vi) The head moves to the left until it scans the symbol b. This b is replaced by 1 $(q_4 zLq_4,\ z \in \Gamma - \{x_i\},\ q_4 x_i Lq_5,\ q_5 b1q_6)$.

(vii) The head moves to the left and one of the 1's in 1^{a_i} is replaced by b. M reaches q_2 $(q_6 1Lq_7,\ q_7 1bq_2)$.
As a result of (i)–(vii), one of the 1's in 1^{a_i} is replaced by b and 1 is added to the left of y. Steps (iv)–(vii) are repeated for all 1's in 1^{a_i}.

(viii) On scanning x_{i-1}, the head moves to the right and M enters q_8 $(q_7 x_{i-1} Rq_8)$.

As there are no quadruples starting with q_8, the Turing machine M halts. When $i \neq 1$ and $a_i \neq 0$, the terminal ID is $1^{a_1}x_1 \ldots x_{i-1} q_8 1^{a_i} x_i \ldots x_n b 1^{a_i} y$. For example, let us compute $U_2^3(1, 2, 1)$:

$$\begin{array}{llll} & q_0 1x_1 11x_2 1x_3 by & \overset{*}{\vdash} & 1x_1 11 q_0 x_2 1x_3 by \\ \vdash & 1x_1 1q_1 1x_2 1x_3 by & \vdash & 1x_1 1q_2 bx_2 1x_3 by \\ \overset{*}{\vdash} & 1x_1 1bx_2 1x_3 bq_2 y & \vdash & 1x_1 1bx_2 1x_3 bq_3 1 \\ \vdash & 1x_1 1bx_2 1x_3 b1q_3 b & \vdash & 1x_1 1bx_2 1x_3 b1q_4 y \\ \overset{*}{\vdash} & 1x_1 1bq_4 x_2 1x_3 b1y & \vdash & 1x_1 1q_5 bx_2 1x_3 b1y \\ \vdash & 1x_1 1q_6 1x_2 1x_3 b1y & \vdash & 1x_1 q_7 11x_2 1x_3 b1y \\ \vdash & 1x_1 q_2 b1x_2 1x_3 b1y & & \end{array}$$

From the above derivation, we see that

$$1x_1 1q_2 bx_2 1x_3 by \overset{*}{\vdash} 1x_1 q_2 b1x_2 1x_3 b1y$$

Repeating the above steps, we get

$$1x_1 q_2 b1x_2 1x_3 b1y \overset{*}{\vdash} 1x_1 q_8 11x_2 1x_3 b11y$$

It should be noted that this construction is similar to that for the successor function. While computing U_i^m, the head skips the portion of the input corresponding to a_j, $j \neq i$. For every 1 in 1^{a_i}, 1 is added to the left of y.

Thus we have shown that the three initial primitive recursive functions are Turing-computable. Next we construct Turing machines that can perform composition, recursion, and minimization.

11.4.7 CONSTRUCTION OF THE TURING MACHINE THAT CAN PERFORM COMPOSITION

Let $f_1(x_1, x_2, \ldots, x_m), \ldots, f_k(x_1, \ldots, x_m)$ be Turing-computable functions. Let $g(y_1, \ldots, y_k)$ be Turing-computable. Let $h(x_1, \ldots, x_m) = g(f_1(x_1, \ldots, x_m) \ldots, f_k(x_1, \ldots, x_m))$. We construct a Turing machine that can compute $h(a_1, \ldots, a_m)$ for given arguments $a_1, \ldots, a_m$. This involves the following steps:

Step 1 Construct Turing machines $M_1, \ldots, M_k$ which can compute $f_1, \ldots, f_k$, respectively. For the TMs $M_1, \ldots, M_k$, let $\Gamma = \{1, b, x_1, x_2, \ldots, x_n, y\}$ and $X = 1^{a_1}x_1 \ldots 1^{a_m} x_m\ by$. But the number of states for these TMs will vary. Let $n_1 + 1, \ldots, n_k + 1$ be the number of states for $M_1, \ldots, M_k$, respectively. As usual, the initial state is q_0 and the states for M_i are $q_0, \ldots, q_{n_i}$. As in the earlier constructions, the set P_i of quadruples for M_i is constructed in such a way that there is no quadruple starting with q_{n_i}.

Step 2 Let $f_i(a_1, \ldots, a_m) = b_i$ for $i = 1, 2, \ldots, k$. At the end of step 1, we have M_i's and the computed values b_i's. As g is Turing-computable, we can construct a TM M_{k+1} which can compute $g(b_1, \ldots, b_k)$. For M_{k+1},

$$\Gamma = \{1, b, x'_1, \ldots, x'_m, y\}, \qquad X' = 1^{b_1}x'_1 \ldots 1^{a_m}x'_m by$$

(We use different markers for M_{k+1} so that the TM computing h to be constructed need not scan the inputs $a_1, \ldots, a_m$.) Let $n_{k+1} + 1$ be the number of states of M_{k+1}. As in the earlier constructions, M_{k+1} has no quadruples starting with q_{k+1}.

Step 3 At the end of step 2, we have TMs $M_1, \ldots, M_k, M_{k+1}$ which give $b_1, \ldots, b_m$ and $g(b_1, \ldots, b_k) = c$ (say), respectively. So we are able to compute $h(a_1, \ldots, a_m)$ using $k + 1$ Turing machines. Our objective is to construct a single TM M_{k+2} which can compute $h(a_1, \ldots, a_m)$. We outline the construction of M without giving the complete details of the encoding mechanism. For M, let

$$\Gamma = \{1, b, x_1, \ldots, x_m, x'_1, \ldots, x'_m, y\}$$
$$X = 1^{a_1}x_11^{a_2}x_2 \ldots 1^{a_m}x_m by$$

(i) In the beginning, M simulates M_1. As a result, the value $b_1 = f_1(a_1, \ldots, a_m)$ is obtained as output. Thus we get the tape expression $1^{a_1}x_11^{a_2}x_2 \ldots 1^{a_m}x_m1^{b_1}y$ which is the same as that obtained by M_1 while halting. M does not halt but changes y to x'_1 and adds by to the right of x'_1. The head moves to the left to reach the beginning of X.

(ii) The tape expression obtained at the end of (i) is

$$1^{a_1}x_1 1^{a_2}x_2 \ldots 1^{a_m}x_m 1^{b_1}x'_1 by$$

The construction given in (i) is repeated, i.e. M simulates $M_2, \ldots, M_k$, changes y to x'_i, and adds by to the right of x'_i. After simulating M_k, the tape expression is

$$X' = 1^{a_1}x_1 \ldots 1^{a_m}x_m 1^{b_1}x'_1 \ldots 1^{b_{k-1}}x_{k-1} 1^{b_k}x'_k by$$

Then the head moves to the left until it is positioned at the cell having 1 just to the right of x_m.

(iii) M simulates M_{k+1}. M_{k+1} with initial tape expression X' halts with the tape expression $1^{b_1}x'_1 \ldots 1^{b_k}x'_m 1^c y$. As a result, the corresponding tape expression for M is obtained as

$$1^{a_1}x_1 1^{a_2}x_2 \ldots 1^{a_m}x_m 1^{b_1}x'_1 \ldots 1^{b_k}x'_k 1^c y$$

(iv) The required value is obtained to the left of y, but $1^{b_1}x'_1 \ldots 1^{b_k}x'_k$ also appears to the left of c. M erases all these symbols and moves $1^c y$ just to the right of x_m. The head moves to the cell having x_m and M halts. The final tape expression is $1^{a_1}x_1 1^{a_2}x_2 \ldots 1^{a_m}x_m 1^c y$.

11.4.8 Construction of the Turing Machine That Can Perform Recursion

Let $g(x_1, \ldots, x_m)$, $h(y_1, y_2, \ldots, y_{m+2})$ be Turing-computable. Let $f(x_1, \ldots, x_{m+1})$ be defined by recursion as follows:

$$f(x_1, \ldots, x_m, 0) = g(x_1 \ldots x_m)$$

$$f(x_1, \ldots, x_m, y + 1) = h(x_1, \ldots, x_m, y, f(x_1, \ldots, x_m, y))$$

For the Turing machine M, computing $f(a_1, \ldots, a_m, c)$, (say k), X is taken as

$$1^{a_1}x_1 \ldots 1^{a_m}x_m 1^c x_{m+1} by$$

As the construction is similar to the construction for computing composition, we outline below the steps of the construction.

Step 1 Let M simulate the Turing machine M' which computes $g(a_1, \ldots, a_m)$. The computed value, namely $g(a_1, \ldots, a_m)$, is placed to the left of y. If $c = 0$, then the computed value $g(a_1, \ldots, a_m)$ is $f(a_1, \ldots, a_m, 0)$. The head is placed to the right of x_m and M halts.

Step 2 If c is not equal to zero, 1^c to the left of x_{m+1} is replaced by b^c. The marker y is changed to x_{m+2} and by is added to the right of x_{m+2}. The head moves to the left of 1^{a_1}.

Step 3 h is computable. M is allowed to compute h for the arguments $a_1, \ldots, a_m$, 0, $g(a_1, \ldots, a_m)$ which appear to the left of $x_1, \ldots, x_m, x_{m+1}, x_{m+2}$, respectively. The computed value is $f(a_1, \ldots, a_m, 1)$. And $f(a_1, \ldots, a_m, 2)$ $\ldots f(a_1, \ldots, a_m, c)$ are computed successively by replacing the rightmost b and computing h for the respective arguments.

The computation stops with a terminal ID. namely

$$b1^{a_1}x_11^{a_2} \ldots q_f1^c x_{n+1}1^k y, \quad k = f(a_1, \ldots, a_m, c)$$

11.4.9 CONSTRUCTION OF THE TURING MACHINE THAT CAN PERFORM MINIMIZATION

When $f(x_1, \ldots, x_m)$ is defined from $g(x_1, \ldots, x_m, y)$ by minimization, $f(x_1, \ldots, x_m)$ is the least of all k's such that $g(x_1, \ldots, x_m, k) = 0$. So the problem reduces to computing $g(a_1, \ldots, a_m, k)$ for given arguments $a_1, \ldots, a_m$ and for values of k starting from 0. $f(a_1, \ldots, a_m)$ is the first k for which $g(a_1, \ldots, a_m, k) = 0$. Hence as soon as the computed value of $g(a_1, \ldots, a_m, y)$ is zero, the required Turing machine M has to halt. Of course, when no such y exists, M never halts, and $f(a_1, \ldots, a_m)$ is not defined.

Thus the construction of M is in such a way that it simulates the TM that computes $g(a_1, \ldots, a_m, k)$ for successive values of k. Once the computed value $g(a_1, \ldots, a_m, k) = 0$ for the first time, M erases by and changes x_{m+1} to y. The head moves to the left of x_m and M halts.

As partial recursive functions are obtained from the initial functions by a finite number of applications of composition, recursion and minimization (Definition 11.11) by the various constructions we have made in this section, the partial recursive functions become Turing-computable.

Using Godel numbering which converts operations of Turing machines into numeric quantities, it can be proved that Turing-computable functions are partial recursive. (For proof, refer Mendelson (1964).)

11.5 SUPPLEMENTARY EXAMPLES

EXAMPLE 11.13

Show that the function $f(x_1, x_2, \ldots, x_n) = 4$ is primitive recursive.

Solution

$$\begin{aligned} 4 &= S^4(0) \\ &= S^4(Z(x_1)) \\ &= S^4(Z(U_1^n(x_1, x_2, \ldots, x_n))) \end{aligned}$$

i.e.

$$f(x_1, x_2, \ldots, x_n) = S^4(Z(U_1^n(x_1, x_2, \ldots, x_n))).$$

As f is the composition of initial functions, f is primitive recursive.

EXAMPLE 11.14

If $f(x_1, x_2)$ is primitive recursive, show that $g(x_1, x_2, x_3, x_4) = f(x_1, x_4)$ is primitive recursive.

Solution

$$\begin{aligned} & g(x_1, x_2, x_3, x_4) \\ & = f(x_1, x_4) \\ & = f(U_1^4(x_1, x_2, x_3, x_4), U_4^4(x_1, x_2, x_3, x_4)) \end{aligned}$$

U_1^4 and U_4^4 are initial functions and hence primitive recursive. f is primitive recursive. As the function g is obtained by applying composition to primitive recursive functions, g is primitive recursive (by the Note appearing at the end of Example 11.5).

EXAMPLE 11.15

If $f(x, y)$ is primitive recursive, show that $g(x, y) = f(4, y)$ is primitive recursive.

Solution

Let $h(x, y) = 4$. h is primitive recursive by Example 11.13.

$$\begin{aligned} & g(x, y) \\ & = f(4, y) \\ & = f(h(x, y), U_2^2(x, y)) \end{aligned}$$

As f and g are primitive recursive and U_2^2 is an initial function, g is primitive recursive.

EXAMPLE 11.16

Show that $f(x, y) = x^2y^4 + 7xy^3 + 4y^5$ is primitive recursive.

Solution

As $f_1(x, y) = x + y$ is primitive recursive (Example 9.5), it is enough to prove that each summand of $f(x, y)$ is primitive recursive.

But,

$$x^2y^4 = U_1^2(x, y) * U_1^2(x, y) * U_2^2(x, y) * U_2^2(x, y) * U_2^2(x, y) * U_2^2(x, y)$$

As multiplication is primitive recursive, $g(x, y) = x^2y^4$ is primitive recursive.

As $h(x, y) = xy^3$ is primitive recursive, $7xy^3 = xy^3 + \cdots + xy^3$ is primitive recursive. Similarly, $4y^5$ is primitive recursive.

SELF-TEST

Choose the correct answer to Questions 1–10.

1. $S(Z(6))$ is equal to
   (a) $U_1^3(1, 2, 3)$
   (b) $U_2^3(1, 2, 3)$
   (c) $U_3^3(1, 2, 3)$
   (d) none of these.

2. Cons $a(y)$ is equal to
   (a) $\wedge$
   (b) ya
   (c) ay
   (d) a

3. $\min(x, y)$ is equal to
   (a) $x \dot{-} (x \dot{-} y)$
   (b) $y \dot{-} (y \dot{-} x)$
   (c) $x - y$
   (d) $y - x$

4. $A(1, 2)$ is equal to
   (a) 3
   (b) 4
   (c) 5
   (d) 6

5. $f(x) = x/3$ over N is
   (a) total
   (b) partial
   (c) not partial
   (d) total but not partial.

6. $\psi_{\{4\}}(3)$ is equal to
   (a) 0
   (b) 3
   (c) 4
   (d) none of these.

7. sgn(x) takes the value 1 if
   (a) $x < 0$
   (b) $x \leq 0$
   (c) $x > 0$
   (d) $x \geq 0$

8. $\psi_A + \psi_B = \psi_{A \cup B}$ if
   (a) $A \cup B = A$
   (b) $A \cup B = B$
   (c) $A \cap B = A$
   (d) $A \cap B = \emptyset$

9. $U_2^4(S(4), S(5), S(6), Z(7))$ is
(a) 6
(b) 5
(c) 4
(d) 0

10. If $g(x, y) = \min(x, y)$ and $h(x, y) = |x - y|$, then:
(a) Both functions are regular functions.
(b) The first function is regular and the second is not regular.
(c) Neither of the functions is regular.
(d) The second function is not regular.

State whether the Statements 11–15 are true or false.

11. $f(x, y) = x + y$ is primitive recursive.

12. $3 \dot{-} 4 = 0$.

13. The transpose function is not primitive recursive.

14. The Ackermann's function is recursive but not primitive recursive.

15. $A(2, 2) = 7$.

EXERCISES

11.1 Test which of the following functions are total. If a function is not total, specify the arguments for which the function is defined.
(a) $f(x) = x/3$ over N
(b) $f(x) = 1/(x - 1)$ over N
(c) $f(x) = x^2 - 4$ over N
(d) $f(x) = x + 1$ over N
(e) $f(x) = x^2$ over N

11.2 Show that the following functions are primitive recursive:

(a) $\chi_{\{0\}}(x) = \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{if } x \neq 0 \end{cases}$

(b) $f(x) = x^2$

(c) $f(x, y) =$ maximum of x and y

(d) $f(x) = \begin{cases} x/2 & \text{when } x \text{ is even} \\ (x-1)/2 & \text{when } x \text{ is odd} \end{cases}$

(e) The sign function defined by
$\text{sgn}(0) = 0, \quad \text{sgn}(x) = 1 \quad \text{if} \quad x > 0.$

(f) $L(x, y) = \begin{cases} 1 & \text{if } x > y \\ 0 & \text{if } x \le y \end{cases}$

(g) $E(x, y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{if } x \ne y \end{cases}$

11.3 Compute $A(3, 2)$, $A(2, 3)$, $A(3, 3)$.

11.4 Show that the following functions are primitive recursive:

(a) $q(x, y)$ = the quotient obtained when x is divided by y
(b) $r(x, y)$ = the remainder obtained when x is divided by y

(c) $f(x) = \begin{cases} 2x & \text{if } x \text{ is a perfect square} \\ 2x + 1 & \text{otherwise} \end{cases}$

11.5 Show that $f(x)$ = integral part of $\sqrt{x}$ is partial recursive.

11.6 Show that the Fibonacci numbers are generated by a primitive recursive function.

11.7 Let $f(0) = 1$, $f(1) = 2$, $f(2) = 3$ and $f(x + 3) = f(x) + f(x + 1)^2 + f(x + 2)^3$. Show that $f(x)$ is primitive recursive.

11.8 The characteristic function χ_A of a given set A is defined as

$$\chi_A(a) = \begin{cases} 0 & \text{if } a \notin A \\ 1 & \text{if } a \in A \end{cases}$$

If A, B are subsets of N and χ_A, χ_B are recursive, show that χ_{A^c}, $\chi_{A \cup B}$, $\chi_{A \cap B}$ are also recursive.

11.9 Show that the characteristic function of the set of all even numbers is recursive. Prove that the characteristic function of the set of all odd integers is recursive.

11.10 Show that the function $f(x, y) = x - y$ is partial recursive.

11.11 Show that a constant function over N, i.e. $f(n) = k$ for all n in N where k is a fixed number, is primitive recursive.

11.12 Show that the characteristic function of a finite subset of N is primitive recursive.

11.13 Show that the addition function $f_1(x, y)$ is Turing-computable. (Represent x and y in tally notation and use concatenation.)

11.14 Show that the Turing machine M in the Post notation (i.e. the transition function specified by quadruples) can be simulated by a Turing machine M (as defined in Chapter 9).

[*Hint:* The transition given by a quadruple can be simulated by two quintuples of M' by adding new states to M'.]

11.15 Compute $Z(4)$ using the Turing machine constructed for computing the zero function.

11.16 Compute $S(3)$ using the Turing machine which computes S.

11.17 Compute $U_1^3(2, 1, 1)$, $U_2^3(1, 2, 1)$, $U_3^3(1, 2, 1)$ using the Turing machines which can compute the projection functions.

11.18 Construct a Turing machine which can compute $f(x) = x + 2$.

11.19 Construct a Turing machine which can compute $f(x_1, x_2) = x_1 + 2$ for the arguments 1, 2 (i.e. $x_1 = 1$, $x_2 = 2$).

11.20 Construct a Turing machine which can compute $f(x_1, x_2) = x_1 + x_2$ for the arguments 2, 3 (i.e. $x_1 = 2$, $x_2 = 3$).

12 Complexity

When a problem/language is decidable, it simply means that the problem is computationally solvable in principle. It may not be solvable in practice in the sense that it may require enormous amount of computation time and memory. In this chapter we discuss the computational complexity of a problem. The proofs of decidability/undecidability are quite rigorous, since they depend solely on the definition of a Turing machine and rigorous mathematical techniques. But the proof and the discussion in complexity theory rests on the assumption that **P** ≠ **NP**. The computer scientists and mathematicians strongly believe that **P** ≠ **NP**, but this is still open.

This problem is one of the challenging problems of the 21st century. This problem carries a prize money of \$1M. **P** stands for the class of problems that can be solved by a deterministic algorithm (i.e. by a Turing machine that halts) in polynomial time; **NP** stands for the class of problems that can be solved by a nondeterministic algorithm (that is, by a nondeterministic TM) in polynomial time; **P** stands for polynomial and **NP** for nondeterminisitc polynomial. Another important class is the class of *NP*-complete problems which is a subclass of **NP**.

In this chapter these concepts are formalized and Cook's theorem on the *NP*-completeness of SAT problem is proved.

12.1 GROWTH RATE OF FUNCTIONS

When we have two algorithms for the same problem, we may require a comparison between the running time of these two algorithms. With this in mind, we study the growth rate of functions defined on the set of natural numbers.

In this section, N denotes the set of natural numbers.

Definition 12.1 Let $f, g : N \to R^+$ (R^+ being the set of all positive real numbers). We say that $f(n) = O(g(n))$ if there exist positive integers C and N_0 such that

$$f(n) \le Cg(n) \qquad \text{for all } n \ge N_0.$$

In this case we say f is of the order of g (or f is 'big oh' of g)

Note: $f(n) = O(g(n))$ is not an equation. It expresses a relation between two functions f and g.

EXAMPLE 12.1

Let $f(n) = 4n^3 + 5n^2 + 7n + 3$. Prove that $f(n) = O(n^3)$.

Solution

In order to prove that $f(n) = O(n^3)$, take $C = 5$ and $N_0 = 10$. Then

$$f(n) = 4n^3 + 5n^2 + 7n + 3 \le 5n^3 \qquad \text{for } n \ge 10$$

When $n = 10$, $5n^2 + 7n + 3 = 573 < 10^3$. For $n > 10$, $5n^2 + 7n + 3 < n^3$. Then, $f(n) = O(n^3)$.

Theorem 12.1 If $p(n) = a_k n^k + a_{k-1}n^{k-1} + \cdots + a_1 n + a_0$ is a polynomial of degree k over Z and $a_k > 0$, then $p(n) = O(n^k)$.

Proof $p(n) = a_k n^k + a_{k-1}n^{k-1} + \cdots + a_1 n + a_0$. As a_k is an integer and positive, $a_k \ge 1$.

As $a_{k-1}, a_{k-2}, \ldots, a_1, a_0$ and k are fixed integers, choose N_0 such that for all $n \ge N_0$ each of the numbers

$$\frac{|a_{k-1}|}{n}, \frac{|a_{k-2}|}{n^2}, \ldots, \frac{|a_1|}{n^{k-1}}, \frac{|a_0|}{n^k} \text{ is less than } \frac{1}{k} \qquad (*)$$

Hence,

$$\left| \frac{a_{k-1}}{n} + \frac{a_{k-2}}{n^2} + \cdots + \frac{a_0}{n^k} \right| < 1$$

As $a_k \ge 1$, $\dfrac{p(n)}{n^k} = a_k + \dfrac{a_{k-1}}{n} + \cdots + \dfrac{a_1}{n^{k-1}} + \dfrac{a_0}{n^k} > 0 \qquad$ for all $n \ge N_0$

Also,

$$\frac{p(n)}{n^k} = a_k + \left(\frac{a_{k-1}}{n} + \cdots + \frac{a_1}{n^{k-1}} + \frac{a_0}{n^k} \right)$$
$$\le a_k + 1 \qquad \text{by } (*)$$

So,

$$p(n) \le Cn^k, \qquad \text{where} \qquad C = a_k + 1$$

Hence,

$$p(n) = O(n^k). \qquad \blacksquare$$

Corollary The order of a polynomial is determined by its degree.

Definition 12.2 An exponential function is a function $q : N \to N$ defined by

$$q(n) = a^n \quad \text{for some fixed } a > 1.$$

When n increases, each of n, n^2, 2^n increases. But a comparison of these functions for specific values of n will indicate the vast difference between the growth rate of these functions.

TABLE 12.1 Growth Rate of Polynomial and Exponential Functions

n	$f(n) = n^2$	$g(n) = n^2 + 3n + 9$	$q(n) = 2^n$
1	1	13	2
5	25	49	32
10	100	139	1024
50	2500	2659	$(1.13)10^{15}$
100	10000	10309	$(1.27)10^{30}$
1000	1000000	1003009	$(1.07)10^{301}$

From Table 12.1, it is easy to see that the function $q(n)$ grows at a very fast rate when compared to $f(n)$ or $g(n)$. In particular the exponential function grows at a very fast rate when compared to any polynomial of large degree. We prove a precise statement comparing the growth rate of polynomials and exponential function.

Definition 12.3 We say $g \neq O(f)$, if for any constant C and N_0, there exists $n \geq N_0$ such that $g(n) > Cf(n)$.

Definition 12.4 If f and g are two functions and $f = O(g)$, but $g \neq O(f)$, we say that the growth rate of g is greater than that of f. (In this case $g(n)/f(n)$ becomes unbounded as n increases to ∞.)

Theorem 12.2 The growth rate of any exponential function is greater than that of any polynomial.

Proof Let $p(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$ and $q(n) = a^n$ for some $a > 1$.

As the growth rate of any polynomial is determined by its term with the highest power, it is enough to prove that $n^k = O(a^n)$ and $a^n \neq O(n^k)$. By L'Hospital's rule, $\dfrac{\log n}{n}$ tends to 0 as $n \to \infty$. (Here $\log n = \log_e n$.) If

$$z(n) = \left[e^{k\left(\frac{\log n}{n}\right)}\right]$$

then,

$$(z(n))^n = \left[e^{k\left(\frac{\log n}{n}\right)}\right]^n = e^{k \log n} = e^{\log n^k} = n^k$$

As n gets large, $k\left(\dfrac{\log n}{n}\right)$ tends to 0 and hence $z(n)$ tends to 0.

So we can choose N_0 such that $z(n) \le a$ for all $n \ge N_0$. Hence $n^k = z(n)^n \le a^n$, proving $n^k = O(a^n)$.

To prove $a^n \ne O(n^k)$, it is enough to show that a^n/n^k is unbounded for large n. But we have proved that $n^k \le a^n$ for large n and any positive integer k and hence for $k + 1$. So $n^{k+1} \le a^n$ or $\dfrac{a^n}{n^{k+1}} \ge 1$.

Multiplying by n, $n\left(\dfrac{a^n}{n^{k+1}}\right) \ge n$, which means $\dfrac{a^n}{n^k}$ is unbounded for large values of n. ▌

Note: The function $n^{\log n}$ lies between any polynomial function and a^n for any constant a. As $\log n \ge k$ for a given constant k and large values of n, $n^{\log n} \ge n^k$ for large values of n. Hence $n^{\log n}$ dominates any polynomial. But $n^{\log n} = (e^{\log n})^{\log n} = e^{(\log n)^2}$. Let us calculate $\lim\limits_{x\to\infty} \dfrac{(\log x)^2}{cx}$. By L'Hospital's rule, $\lim\limits_{x\to\infty} \dfrac{(\log x)^2}{cx} = \lim\limits_{x\to\infty} (2\log x)\dfrac{1/x}{c} = \lim\limits_{x\to\infty} \dfrac{2\log x}{cx} = \lim\limits_{x\to\infty} \dfrac{2}{cx} = 0$.

So $(\log n)^2$ grows more slowly than cn. Hence $n^{\log n} = e^{(\log n)^2}$ grows more slowly than 2^{cn}. The same holds good when logarithm is taken over base 2 since $\log_e n$ and $\log_2 n$ differ by a constant factor.

Hence there exist functions lying between polynomials and exponential functions.

12.2 THE CLASSES *P* AND *NP*

In this section we introduce the classes **P** and **NP** of languages.

Definition 12.5 A Turing machine M is said to be of time complexity $T(n)$ if the following holds: Given an input w of length n, M halts after making at most $T(n)$ moves.

Note: In this case, M eventually halts. Recall that the standard TM is called a deterministic TM.

Definition 12.6 A language L is in class **P** if there exists some polynomial $T(n)$ such that $L = T(M)$ for some deterministic TM M of time complexity $T(n)$.

EXAMPLE 12.2

Construct the time complexity $T(n)$ for the Turing machine M given in Example 9.7.

Solution

In Example 9.7, the step (i) consists of going through the input string (0^n1^n) forward and backward and replacing the leftmost 0 by x and the leftmost 1 by y. So we require at most $2n$ moves to match a 0 with a 1. Step (ii) is repetition of step (i) n times. Hence the number of moves for accepting a^nb^n is at most $(2n)(n)$. For strings not of the form a^nb^n, TM halts with less than $2n^2$ steps. Hence $T(M) = O(n^2)$.

We can also define the complexity of algorithms. In the case of algorithms. $T(n)$ denotes the running time for solving a problem with an input of size n, using this algorithm.

In Example 12.2. we use the notation $\leftarrow$ which is used in expressing algorithm. For example. $a \leftarrow b$ means replacing a by b.

$\lceil a \rceil$ denotes the smallest integer greater than or equal to a. This is called the *ceiling function*.

EXAMPLE 12.3

Find the running time for the Euclidean algorithm for evaluating gcd(a, b) where a and b are positive integers expressed in binary representation.

Solution

The Euclidean algorithm has the following steps:

1. The input is (a, b)
2. Repeat until $b = 0$
3. Assign $a \leftarrow a \bmod b$
4. Exchange a and b
5. Output a.

Step 3 replaces a by $a \bmod b$. If $a/2 \geq b$, then $a \bmod b < b \leq a/2$. If $a/2 < b$, then $a < 2b$. Write $a = b + r$ for some $r < b$. Then $a \bmod b = r < b < a/2$. Hence $a \bmod b \leq a/2$. So a is reduced by at least half in size on the application of step 3. Hence one iteration of step 3 and step 4 reduces a and b by at least half in size. So the maximum number of times the steps 3 and 4 are executed is $\min\{\lceil \log_2 a \rceil, \lceil \log_2 b \rceil\}$. If n denotes the maximum of the number of digits of a and b, that is $\max\{\lceil \log_2 a \rceil, \lceil \log_2 b \rceil\}$ then the number of iterations of steps 3 and 4 is $O(n)$. We have to perform step 2 at most $\min\{\lceil \log_2 a \rceil, \lceil \log_2 b \rceil\}$ times or n times. Hence $T(n) = nO(n) = O(n^2)$.

Note: The Euclidean algorithm is a polynomial algorithm.

Definition 12.7 A language L is in class **NP** if there is a nondeterministic TM M and a polynomial time complexity $T(n)$ such that $L = T(M)$ and M executes at most $T(n)$ moves for every input w of length n.

We have seen that a deterministic TM M_1 simulating a nondeterministic TM M exists (refer to Theorem 9.3). If $T(n)$ is the complexity of M, then the complexity of the equivalent deterministic TM M_1 is $2^{O(T(n))}$. This can be justified as follows. The processing of an input string w of length n by M is equivalent to a 'tree' of computations by M_1. Let k be the maximum of the number of choices forced by the nondeterministic transition function. (It is $\max|\delta(q, x)|$, the maximum taken over all states q and all tape symbol X.) Every branch of the computation tree has a length $T(n)$ or less. Hence the total number of leaves is atmost $kT(n)$. Hence the complexity of M_1 is at most $2^{O(T(n))}$.

It is not known whether the complexity of M_1 is less than $2^{O(T(n))}$. Once again an answer to this question will prove or disprove **P** $\neq$ **NP**. But there do exist algorithms where $T(n)$ lies between a polynomial and an exponential function (refer to Section 12.1).

12.3 POLYNOMIAL TIME REDUCTION AND *NP*-COMPLETENESS

If P_1 and P_2 are two problems and $P_2 \in$ **P**, then we can decide whether $P_1 \in$ **P** by relating the two problems P_1 and P_2. If there is an algorithm for obtaining an instance of P_2 given any instance of P_1, then we can decide about the problem P_1. Intuitively if this algorithm is a polynomial one, then the problem P_1 can be decided in polynomial time.

Definition 12.8 Let P_1 and P_2 be two problems. A reduction from P_1 to P_2 is an algorithm which converts an instance of P_1 to an instance of P_2. If the time taken by the algorithm is a polynomial $p(n)$, n being the length of the input of P_1, then the reduction is called a polynomial reduction P_1 to P_2.

Theorem 12.3 If there is a polynomial time reduction from P_1 to P_2 and if P_2 is in **P** then P_1 is in **P**.

Proof Let m denote the size of the input of P_1. As there is a polynomial-time reduction of P_1 to P_2, the corresponding instance of P_2 can be got in polynomial-time. Let it be $O(m^j)$, So the size of the resulting input of P_2 is atmost cm^j for some constant c. As P_2 is in **P**, the time taken for deciding the membership in P_2 is $O(n^k)$, n being the size of the input of P_2. So the total time taken for deciding the membership of m-size input of P_1 is the sum of the time taken for conversion into an instance of P_2 and the time for decision of the corresponding input in P_2. This is $O[m^j + (cm^j)^k]$, which is the same as $O(m^{jk})$. So P_1 is in **P**. ∎

Definition 12.9 Let L be a language or problem in **NP**. Then L is *NP*-complete if

1. L is in **NP**

2. For every language L' in **NP** there exists a polynomial-time reduction of L' to L.

Note: The class of *NP*-complete languages is a subclass of **NP**.

The next theorem can be used to enlarge the class of *NP*-complete problems provided we have some known *NP*-complete problems.

Theorem 12.4 If P_1 is *NP*-complete, and there is a polynomial-time reduction of P_1 to P_2, then P_2 is *NP*-complete.

Proof If L is any language in **NP**, we show that there is a polynomial-time reduction of L to P_2. As P_1 is *NP*-complete, there is a polynomial-time reduction of L to P_1. So the time taken for converting an n-size input string w in L to a string x in P_1 is at most $p_1(n)$ for some polynomial p_1. As there is a polynomial-time reduction of P_1 to P_2, there exists a polynomial p_2 such that the input x to P_1 is transferred into input y to P_2 in at most $p_2(n)$ time. So the time taken for transforming w to y is at most $p_1(n) + p_2(p_1(n))$. As $p_1(n) + p_2(p_1(n))$ is a polynomial, we get a polynomial-time reduction of L to P_2. Hence P_2 is *NP*-complete. ∎

Theorem 12.5 If some *NP*-complete problem is in **P**, then **P** = **NP**.

Proof Let P be an *NP*-complete problem and $P \in$ **P**. Let L be any *NP*-complete problem. By definition, there is a polynomial-time reduction of L to P. As P is in **P**, L is also in **P** by Theorem 12.3. Hence **NP** = **P**.

12.4 IMPORTANCE OF *NP*-COMPLETE PROBLEMS

In Section 12.3, we proved theorems regarding the properties of *NP*-complete problems. At the beginning of this chapter we noted that the computer scientists and mathematicians strongly believe that **P** ≠ **NP**. At the same time, no problem in **NP** is proved to be in **P**. The entire complexity theory rests on the strong belief that **P** ≠ **NP**.

Theorem 12.4 enables us to extend the class of *NP*-complete problems, while Theorem 12.5 asserts that the existence of one *NP*-complete problem admitting a polynomial-time algorithm will prove **P** = **NP**. More than 2500 *NP*-complete problems in various fields have been found so far.

We will prove the existence of an *NP*-complete problem in Section 12.5. We will give a list of *NP*-complete problems in Section 12.6. Thousands of *NP*-complete problems in various branches such as Operations Research, Logic, Graph Theory, Combinatorics, etc. have been constructed so far. A polynomial-time algorithm for any one of there problems will yield a proof of **P** = **NP**. But such multitude of *NP*-complete problems only strengthens the belief of the computer scientists that **P** ≠ **NP**. We will discuss more about this in Section 12.7.

12.5 SAT IS *NP*-COMPLETE

In this section, we prove that the satisfiability problem for boolean expressions (whether a boolean expression is satisfiable) is *NP*-complete. This is the first problem to be proved *NP*-complete. Cook proved this theorem in 1971.

12.5.1 BOOLEAN EXPRESSIONS

In Section 1.1.2, we defined a well-formed formula involving propositional variables. A boolean expression is a well-formed formula involving boolean variables x, y, z replacing propositions P, Q, R and connectives $\vee$, $\wedge$ and $\neg$. The truth value of a boolean expression in x, y, z is determined from the truth values of x, y, z and the truth tables for $\vee$, $\wedge$ and $\neg$. For example, $\neg x \wedge \neg (y \vee z)$ is a boolean expression. The expression $\neg x \wedge \neg (y \vee z)$ is true when x is false, y is false and z is false.

Definition 12.10 (a) A truth assignment t for a boolean expression E is the assignment of truth values T or F to each of the variables in E. For example, $t = (F, F, F)$ is a truth assignment for (x, y, z) where x, y, z are the variables in a boolean expression $E(x, y, z) = \neg x \wedge \neg (y \vee z)$.

The value $E(t)$ of the boolean expression E given a truth assignment t is the truth value of the expression of E, if the truth values give by t are assigned to the respective variables.

If $t = (F, F, F)$ then the truth values of $\neg x$ and $\neg (y \vee z)$ are T and T. Hence the value of $E = \neg x \wedge \neg (y \vee z)$ is T. So $E(t) = T$.

Definition 12.11 A truth assignment t satisfies a boolean expression E if the truth value of $E(t)$ is T. In other words, the truth assignment t makes the expression E true.

Definition 12.12 A boolean expression E is satisfiable if there exists at least one truth assignment t that satisfies E (that is $E(t) = T$). For example, $E = \neg x \wedge \neg (y \vee z)$ is satisfiable since $E(t) = T$ when $t = (F, F, F)$.

12.5.2 CODING A BOOLEAN EXPRESSION

The symbols in a boolean expression are the variables x, y, z, etc. the connectives $\vee$, $\wedge$, $\neg$, and parantheses (and). Thus a boolean expression in three variables will have eight distinct symbols. The variables are written as x_1, x_2, x_3, etc. Also we use x_n only after using $x_1, x_2, \ldots, x_{n-1}$ for variables.

We encode a boolean expression as follows:

1. The variables $x_1, x_2, x_3, \ldots$ are written as $x1$, $x10$, $x11$, . . . etc. (The binary representation of the subscript is written after x.)
2. The connectives $\vee$, $\wedge$, $\neg$, (, and) are retained in the encoded expression.

For example, $\neg x \wedge \neg (y \vee z)$ is encoded as $\neg x1 \wedge \neg (x10 \vee x11)$, (where x, y, z are represented by x_1, x_2, x_3).

Note: Any boolean expression is encoded as a string over $\Sigma = \{x, 0, 1, \vee, \wedge, \neg, c\ (,)\}$

Consider a boolean expression having m occurrences of variables, connectives and parantheses. The variable x_m can be represented using $1 + \log_2 m$ symbols (x together with the digits in the binary representation of m). The other occurrences require less symbols. So any occurrence of a variable, connective or a parenthesis requires at most $1 + \log_2 m$ symbols over Σ. So the length of the encoded expression is at most $O(m \log m)$.

As our interest is only in deciding whether a problem can be solved in polynomial-time, we need not distinguish between the length of the coded expression and the number of occurrences of variables etc. in a boolean expression.

12.5.3 COOK'S THEOREM

In this section we define the SAT problem and prove the Cook's theorem that SAT is *NP*-complete.

Definition 12.13 The satisfiability problem (SAT) is the problem:

Given a boolean expression, is it satisfiable?

Note: The SAT problem can also be formulated as a language. We can define SAT as the set of all coded boolean expressions that are satisfiable. So the problem is to decide whether a given coded boolean expression is in SAT.

Theorem 12.6 (Cook's theorem) SAT is *NP*-complete.

Proof PART I: SAT $\in$ **NP**.

If the encoded expression E is of length n, then the number of variables is $\lceil n/2 \rceil$. Hence, for guessing a truth assignment t we can use multitape TM for E. The time taken by a multitape NTM M is $O(n)$. Then M evaluates the value of E for a truth assignment t. This is done in $O(n^2)$ time. An equivalent single-tape TM takes $O(n^4)$ time. Once an accepting truth assignment is found, M accepts E and M and halts. Thus we have found a polynomial time NTM for SAT. Hence SAT $\in$ **NP**.

PART II: POLYNOMIAL-TIME REDUCTION OF ANY L IN **NP** TO SAT.

1. Construction of NTM for L

Let L be any language in **NP**. Then there exists a single-tape NTM M and a polynomial $p(n)$ such that the time taken by M for an input of length n is at most $p(n)$ along any branch. We can further assume that this M never writes a blank on any move and never moves its head to the left of its initial tape position (refer to Example 12.6).

If M accepts an input w and $|w| = n$, then there exists a sequence of moves of M such that

1. α_0 is the initial ID of M with input w.
2. $\alpha_0 \vdash \alpha_1 \vdash \cdots \vdash \alpha_k$, $k \le p(n)$.
3. α_k is an ID with an accepting state.
4. Each α_i is a string of nonblanks, its leftmost symbol being the leftmost symbol of w (the only exception occurs when the processing of w is complete, in which case the ID is qb).

2. Representation of Sequence of Moves of M

As the maximum number of steps on w is $p(n)$ we need not bother about the contents beyond $p(n)$ cells. We can write α_i as a sequence of $p(n) + 1$ symbols (one symbol for the state and the remaining symbols for the tape symbols). So $a_i = X_{i0}X_{i1} \ldots X_{i,\ p(n)}$.

By assuming $Q \cap \Gamma = \emptyset$, we can locate the state in α_i and hence the position of the tape head. The length of some ID may be less than $p(n)$. In this case we pad the ID on the right with blank symbols, so that all IDs are of the same length $p(n) + 1$. Also the acceptance may happen earlier. If α_m is an accepting ID in the course of processing w, then we write $\alpha_0 \vdash \ldots \vdash \alpha_m \vdash \alpha_m \ldots \vdash \alpha_m = \alpha_{i,p(n)}$.

Thus all IDs have $p(n) + 1$ symbols and any computation has $p(n)$ moves.

TABLE 12.2 Array of IDs

ID	0	1	...	$j-1$	j	$j+1$	...	$p(n)$
α_0	X_{00}	X_{01}		...				$X_{0,p(n)}$
α_1	X_{10}	X_{11}						$X_{1,p(n)}$
α_i	X_{i0}	X_{i1}	...	$X_{i,j-1}$	X_{ij}	$X_{i,j+1}$	...	$X_{i,p(n)}$
α_{i+1}	$X_{i+1,0}$	$X_{i+1,1}$		$X_{i+1,j-1}$	$X_{i+1,j}$	$X_{i+1,j+1}$	...	$X_{i+1,p(n)}$
$\alpha_{p(n)}$				...				$X_{p(n),p(n)}$

So we can represent any computation as an $(p(n) + 1) \times (p(n) + 1)$ array as in Table 12.2.

3. Representation of IDs in Terms of Boolean Variables

We define a boolean variable y_{ijA} corresponding to (i, j)th entry in the ith ID. The variable y_{ijA} represents the proposition that $x_{ij} = A$, where A is a state or tape symbol and $0 \le i, j \le p(n)$.

We simulate the sequence of IDs leading to the acceptance of an input string w by a boolean expression. This is done in such a way that M accepts w if an only if the simulated boolean expression $E_{M,w}$ is satisfiable.

4. Polynomial Reduction of M to SAT

In order to check that the reduction of M to SAT is correct, we have to ensure the correctness of

(a) the initial ID,
(b) the accepting ID, and
(c) the intermediate moves between successive IDs.

(a) Simulation of initial ID

X_{00} must start with the initial state q_0 of M followed by the symbols of $w = a_1 a_2 \ldots a_n$ of length n and ending with b's (blank symbol). The corresponding boolean expression S is defined as

$$S = y_{00q_0} \wedge y_{01a_1} \wedge y_{01a_2} \wedge \ldots \wedge y_{0na_n} \wedge y_{0,n+1,b} \wedge \ldots \wedge y_{0,p(n),b}$$

Thus given an encoding of M and w, we can write S in a tape of a multiple TM M_1. This takes $O(p(n))$ time.

(b) Simulation of accepting ID

$\alpha_{p(n)}$ is the accepting ID. If $p_1, p_2, \ldots, p_k$ are the accepting states of M, then $\alpha_{p(n)}$ contains one of p_i's, $1 \le i \le k$ in any place j. If $\alpha_{p(n)}$ contains an accepting state p_i in jth position, then $x_{p(n),j}$ is the accepting state p_i. The corresponding boolean expression covering all the cases $(0 \le j \le p(n), 1 \le i \le k)$ is given by

$$F = F_0 \vee F_1 \vee \ldots \vee F_{p(n)}$$

where

$$F_j = y_{p(n),j,p_1} \vee y_{p(n),j,p_2} \vee \ldots \vee y_{p(n),j,p_k}$$

Each F_j has k variables and hence has constant number of symbols depending on M but not on n. The number of F_j's in F is $p(n)$. Thus given an encoding of M and w, F can be written in $O(p(n))$ time on the multiple TM M_1.

(c) Simulation of intermediate moves

We have to simulate valid moves $\alpha_i \vdash \alpha_{i+1}$, $i = 0, 1, 2, \ldots p(n)$. Corresponding to each move, we have to define a boolean variable N_i. Hence the entire sequence of IDs leading to acceptance of w is

$$N = N_0 \wedge N_1 \wedge \ldots \wedge N_{p(n)-1}$$

First of all note that the symbol $X_{i+1,j}$ can be determined from $X_{i,j-1}$, X_{ij}, $X_{i,j+1}$ by the move (if there is one changing α_i to a different α_{i+1}). For every position (i, j), we have two cases:

Case 1 The state of α_i is at position j.
Case 2 The state of α_i is not in any of the $(j-1)$th, jth and $(j+1)$th positions.

Case 1 is taken care of by a variable A_{ij} and Case 2 by a variable B_{ij}.

The variable N_i will be designed in such a way that it gurantees that ID α_{i+1} is one of the IDs that follows the ID α_i.

$X_{i+1,j}$ can be determined from

(i) the three symbols $X_{i,j-1}$, X_{ij}, $X_{i,j+1}$ above it

(ii) the move chosen by the nondeterministic TM M when one of the three symbols (in (i)) is a state.

If the state of α_i is not X_{ij}, $X_{i,j-1}$ or $X_{i,j+1}$, then $X_{i+1,j} = X_{ij}$. This is taken care of by the variable B_{ij}.

If X_{ij} is the state of α_i, then $X_{i,j+1}$ is being scanned by the state X_{ij}. The move corresponding to the state-tape symbol pair $(X_{ij}, X_{i,j+1})$ will determine the sequence $X_{i+1,j-1}\, X_{i+1,j}\, X_{i+1,j+1}$. This is taken care of by the variable A_{ij}.

We write $N_i = \wedge_j (A_{ij} \vee B_{ij})$, where $\wedge$ is taken over all j's, $0 \le j \le p(n)$.

(i) Formulation of B_{ij} When the state of α_i is none of $X_{i,j-1}$, X_{ij}, $X_{i,j+1}$, then the transition corresponding to $\alpha_i \vdash \alpha_{i+1}$ will not affect $X_{i,j+1}$. In this case $X_{i+1,j} = X_{ij}$

Denote the tape symbols by $Z_1, Z_2, \ldots, Z_r$. Then $X_{i,j-1}$, $X_{i,j}$ and $X_{i,j+1}$ are the only tape symbols. So we write B_{ij} as

$$
\begin{aligned}
B_{ij} = {} & (y_{i,j-1,Z_1} \vee y_{i,j-1,Z_2} \vee \ldots \vee y_{i,j-1,Z_r}) \wedge \\
& (y_{i,j,Z_1} \vee y_{i,j,Z_2} \cdots \vee y_{i,j,Z_r}) \wedge \\
& (y_{i,j+1,Z_1}, \vee y_{i,j+1,Z_2} \cdots \vee y_{i,j+1,Z_r}) \wedge \\
& (y_{i,j,Z_1} \wedge y_{i+1,j,Z_1}) \vee (y_{i,j,Z_2} \wedge y_{i+1,j,Z_2} \vee \ldots \vee (y_{i,j,Z_r} \wedge y_{i+1,j,Z_r})
\end{aligned}
$$

This first line of B_{ij} says that $X_{i,j-1}$ is one of the tape symbols $Z_1, Z_2, \ldots, Z_r$. The second and third lines are regarding $X_{i,j}$ and $X_{i,j+1}$. The fourth line says that X_{ij} and $X_{i,j+1}$ are the same and the common value is any one of $Z_1, Z_2, \ldots, Z_r$.

Recall that the head of M never moves to the left of 0-cell and does not have to move to the right of the $p(n)$-cell. So B_{i0} will not have the first line and $B_{i,p(n)}$ will not have the third line.

(ii) Formulation of A_{ij} This step corresponds to the correctness of the 2×3 array (see Table 12.3).

TABLE 12.3 Valid Computation

$X_{i,j-1}$	X_{ij}	$X_{i,j+1}$
$X_{i+1,j-1}$	$X_{i+1,j}$	$X_{i+1,j+1}$

The expression B_{ij} takes care of the case when the state of α_i is not at the position $X_{i,j-1}$, $X_{i,j}$ or $X_{i,j+1}$. The A_{ij} corresponds to the case when the state of α_i is at the position X_{ij}. In this case we have to assign boolean variables to six positions given in Table 12.3 so that the transition corresponding to $\alpha_i \vdash \alpha_{i+1}$ is described by the variables in the box correctly.

We say that that an assignment of symbols to the six variables in the box is valid if

1. X_{ij} is a state but $X_{i,j-1}$ and $X_{i,j+1}$ are tape symbols.
2. Exactly one of $X_{i+1,j-1}$, $X_{i+1,j}$, $X_{i+1,j+1}$ is a state.
3. There is a move which explains how $(X_{i,j-1}, X_{i,j}, X_{i,j+1})$ changes to $(X_{i+1,j-1}, X_{i+1,j}, X_{i+1,j+1})$ in $\alpha_i \vdash \alpha_{i+1}$.

There are only a finite number of valid assignments and A_{ij} is obtained by applying OR (that is $\vee$) to these valid assignments. A valid assignment corresponds to one of the following four cases:

Case A $\quad (p, C, L) \in \delta(q, A)$
Case B $\quad (p, C, R) \in \delta(q, A)$
Case C $\quad \alpha_i = \alpha_{i+1}$ (when α_i and α_{i+1} contain an accepting state)
Case D $\quad j = 0$ and $j = p(n)$

Case A Let D be some tape symbol of M. Then $X_{i,j-1}X_{ij}X_{i,j+1} = DqA$ and $X_{i+1,j-1}X_{i+1,j}X_{i+1,j+1} = pDC$. This can be expressed by the boolean variable.

$$y_{i,j-1,D} \wedge y_{i,j,q} \wedge y_{i,j+1,A} \wedge y_{i+1,j-1,p} \wedge y_{i+1,j,D} \wedge y_{i+1,j+1,C}$$

Case B As in case A, let D be any tape symbol. In this case $X_{i,j-1}X_{ij}X_{i,j+1} = DqA$ and $X_{i+1,j-1}X_{i+1,j}X_{i+1,j+1} = DCp$. The corresponding boolean expression is

$$y_{i,j-1,D} \wedge y_{i,j,q} \wedge y_{i,j+1,A} \wedge y_{i+1,j-1,D} \wedge y_{i+1,j,C} \wedge y_{i+1,j+1,p}$$

Case C In this case $X_{i,j-1}X_{ij}X_{i,j+1} = X_{i+1,j-1}X_{i+1,j}X_{i+1,j+1}$.

In this case the same tape symbol say D appears in $X_{i,j-1}$ and $X_{i+1,j-1}$; some other tape symbol say D' in $X_{i,j+1}$ and $X_{i+1,j+1}$. $X_{i,j}$ and $X_{i+1,j}$ contain the same state. One typical boolean expression is

$$y_{i,j-1,Z_k} \wedge y_{i,j,q} \wedge y_{i,j+1,Z_l} \wedge y_{i+1,j-1,Z_k} \wedge y_{i+1,j,q} \wedge y_{i+1,j+1,Z_l}$$

Case D When $j = 0$, we have only $X_{i0}X_{i1}$ and $X_{i+1,0}X_{i+1,1}$. This is a special case of Case B. $j = p(n)$ corresponds to a special case of Case A.

So, A_{ij} is defined as the OR of all valid terms obtained in Case A to Case D.

(iii) Definition of N_i and N We define N_i and N by

$$N_i = (A_{i0} \vee B_{i0}) \wedge (A_{i1} \vee B_{i1}) \wedge \ldots \wedge (A_{i,p(n)} \vee B_{i,p(n)})$$

$$N = N_0 \wedge N_1 \wedge N_2 \wedge \ldots \wedge N_{p(n)-1}$$

(iv) Time taken for writing N The time taken to write B_{ij} is a constant depending on the number $|\Gamma|$ of tape symbols. (Actually the number of variables in B_{ij} is $5|\Gamma|$). The time taken to write A_{ij} depends only on the number of moves of M. As N_i is obtained by applying OR to $A_{ij} \wedge B_{ij}$, $0 \le i \le p(n) - 1$, $0 \le j \le p(n) - 1$, the time taken to write on N_i is $O(p(n))$. As N is obtained by applying $\wedge$ to $N_0, N_1, \ldots, N_{p(n)-1}$, the time taken to write N is $p(n)O(p(n)) = O(p^2(n))$.

5. Completion of Proof

Let $E_{M,w} = S \wedge N \wedge F$.

We have seen that the time taken to write S and F are $O(p(n))$ and the time taken for N is $O(p^2(n))$. Hence the time taken to write $E_{M,w}$ is $O(p^2(n))$.

Also M accepts w if and only if $E_{M,w}$ is satisfiable.

Hence the deterministic multitape TM M_1 can convert w to a boolean expression $E_{M,w}$ in $O(p^2(n))$ time. An equivalent single tape TM takes $O(p^4(n))$ time. This proves the Part II of the Cook's theorem, thus completing the proof of this theorem. ∎

12.6 OTHER *NP*-COMPLETE PROBLEMS

In the last section, we proved the *NP*-completeness of SAT. Actually it is difficult to prove the *NP*-completeness of any problem. But after getting one *NP*-complete problem such as SAT, we can prove the *NP*-completeness of problem P' by obtaining a polynomial reduction of SAT to P'. The polynomial reduction of SAT to P' is relatively easy. In this section we give a list of *NP*-complete problems without proving their *NP*-completeness. Many of the *NP*-complete problems are of practical interest.

1. CSAT—Given a boolean expression in CNF (conjunctive normal form—Definition 1.10), is it satisfiable?

   We can prove that CSAT is *NP*-complete by proving that CSAT is in **NP** and getting a polynomial reduction from SAT to CSAT.
2. Hamiltonian circuit problem—Does G have a Hamiltonian circuit (i.e. a circuit passing through each edge of G exactly once)?
3. Travelling salesman problem (TSP)—Given n cities, the distance between them and a number D, does there exist a tour programme for a salesman to visit all the cities exactly once so that the distance travelled is at most D?
4. Vertex cover problem—Given a graph G and a natural number k, does there exist a vertex cover for G with k vertices? (A subsets C of vertices of G is a vertex cover for G if each edge of G has an odd vertex in C.)

5. Knapsack problem—Given a set $A = \{a_1, a_2, \ldots, a_n\}$ of nonnegative integers, and an integer K, does there exist a subset B of A such that $\sum_{b_j \in B} b_j = K$?

This list of *NP*-complete problems can be expanded by having a polynomial reduction of known *NP*-complete problems to the problems which are in **NP** and which are suspected to be *NP*-complete.

12.7 USE OF *NP*-COMPLETENESS

One practical use in discovering that problem is *NP*-complete is that it prevents us from wasting our time and energy over finding polynomial or easy algorithms for that problem.

Also we may not need the full generality of an *NP*-complete problem. Particular cases may be useful and they may admit polynomial algorithms. Also there may exist polynomial algorithms for getting an approximate optimal solution to a given *NP*-complete problem.

For example, the travelling salesman problem satisfying the triangular inequality for distances between cities (i.e. $d_{ij} \leq d_{ik} + d_{kj}$ for all i, j, k) has approximate polynomial algorithm such that the ratio of the error to the optimal values of total distance travelled is less than or equal to 1/2.

12.8 QUANTUM COMPUTATION

In the earlier sections we discussed the complexity of algorithm and the dead end was the open problem **P** = **NP**. Also the class of *NP*-complete problems provided us with a class of problems. If we get a polynomial algorithm for solving one *NP*-complete problem we can get a polynomial algorithm for any other *NP*-complete problem.

In 1982, Richard Feynmann, a Nobel laurate in physics suggested that scientists should start thinking of building computers based on the principles of quantum mechanics. The subject of physics studies elementary objects and simple systems and the study becomes more intersting when things are larger and more complicated. Quantum computation and information based on the principles of Quantum Mechanics will provide tools to fill up the gulf between the small and the relatively complex systems in physics. In this section we provide a brief survey of quantum computation and information and its impact on complexity theory.

Quantum mechanics arose in the early 1920s, when classical physics could not explain everything even after adding ad hoc hypotheses. The rules of quantum mechanics were simple but looked counterintuitive, and even Albert Einstein reconciled himself with quantum mechanics only with a pinch of salt.

Quantum Mechanics is real black magic calculus.

—A. Einstein

12.8.1 QUANTUM COMPUTERS

We know that a bit (a 0 or a 1) is the fundamental concept of classical computation and information. Also a classical computer is built from an electronic circuit containing wires and logical gates. Let us study quantum bits and quantum circuits which are analogous to bits and (classical) circuits.

A quantum bit, or simply qubit can be described mathematically as

$$|\psi\rangle = \alpha|0\rangle + \beta|0\rangle$$

The qubit can be explained as follows. A classical bit has two states, a 0 and a 1. Two possible states for a qubit are the states $|0\rangle$ and $|1\rangle$. (The notation $|.\rangle$ is due to Dirac.) Unlike a classical bit, a qubit can be in infinite number of states other than $|0\rangle$ and $|1\rangle$. It can be in a state $|\psi\rangle = \alpha|0\rangle + \beta|0\rangle$, where α and β are complex numbers such that $|\alpha|^2 + |\beta|^2 = 1$. The 0 and 1 are called the computational basis states and $|\psi\rangle$ is called a superposition. We can call $|\psi\rangle = \alpha|0\rangle + \beta|0\rangle$ a quantum state.

In the classical case, we can observe it as a 0 or a 1. But it is not possible to determine the quantum state on observation. When we measure/observe a qubit, we get either the state $|0\rangle$ with probability $|\alpha|^2$ or the state $|1\rangle$ with probability $|\beta|^2$.

This is difficult to visualize, using our 'classical thinking' but this is the source of power of the quantum computation.

Multiple qubits can be defined in a similar way. For example, a two-qubit system has four computational basis states, $|00\rangle$, $|01\rangle$, $|10\rangle$ and $|11\rangle$ and quantum states $|\psi\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle$ with $|\alpha_{00}|^2 + |\alpha_{01}|^2 + |\alpha_{10}|^2 + |\alpha_{11}|^2 = 1$.

Now we define the qubit gates. The classical NOT gate interchanges 0 and 1. In the case of the qubit the NOT gate, $\alpha|0\rangle + \beta|1\rangle$, is changed to $\alpha|1\rangle + \beta|0\rangle$.

The action of the qubit NOT gate is linear on two-dimensional complex vector spaces. So the qubit NOT gate can be described by

$$\begin{bmatrix} \alpha \\ \beta \end{bmatrix} \to \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \beta \\ \alpha \end{bmatrix}$$

The matrix $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ is a unitary matrix. (A matrix A is unitary if $A \operatorname{adj}A = I$.)

We have seen earlier that {NOR} is functionally complete (refer to Exercises of Chapter 1). The qubit gate corresponding to NOR is the controlled-NOT or CNOT gate. It can be described by

$$|A, B\rangle \to |A, B \oplus A\rangle$$

where $\oplus$ denotes addition modulo 2. The action on computational basis is $|00\rangle \to |00\rangle$, $|01\rangle \to |01\rangle$, $|10\rangle \to |11\rangle$, $|11\rangle \to |10\rangle$. It can be described by the following 4×4 unitary matrix:

$$U_{CN} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Now, we are in a position to define a quantum computer:

A quantum computer is a system built from quantum circuits, containing wires and elementary quantum gates, to carry out manipulation of quantum information.

12.8.2 CHURCH–TURING THESIS

Since 1970s many techniques for controlling the single quantum systems have been developed but with only modest success. But an experimental prototype for performing quantum cryptography, even at the initial level may be useful for some real-world applications.

Recall the Church–Turing thesis which asserts that any algorithm that can be performed on any computing machine can be performed on a Turing machine as well.

Miniaturization of chips has increased the power of the computer. The growth of computer power is now described by Moore's law, which states that the computer power will double for constant cost once in every two years. Now it is felt that a limit to this doubling power will be reached in two or three decades, since the quantum effects will begin to interfere in the functioning of electronic devices as they are made smaller and smaller. So efforts are on to provide a theory of quantum computation which will compensate for the possible failure of the Moore's law.

As an algorithm requiring polynomial time was considered as an efficient algorithm, a strengthened version of the Church–Turing thesis was enunciated.

Any algorithmic process can be simulated efficiently by a Turing machine. But a challenge to the strong Church–Turing thesis arose from analog computation. Certain types of analog computers solved some problems efficiently whereas these problems had no efficient solution on a Turing machine. But when the presence of noise was taken into account, the power of the analog computers disappeared.

In mid-1970s, Robert Solovay and Volker Strassen gave a randomized algorithm for testing the primality of a number. (A deterministic polynomial algorithm was given by Manindra Agrawal, Neeraj Kayal and Nitein Saxena of IIT Kanpur in 2003.) This led to the modification of the Church thesis.

Strong Church-Turing Thesis

Any algorithmic process can be simulated efficiently using a nondeterministic Turing machine.

In 1985, David Deutsch tried to build computing devices using quantum mechanics.

> *Computers are physical objects, and computations are physical processes. What computers can or cannot compute is determined by the law of physics alone, and not by pure mathematics*
>
> —David Deutsch

But it is not known whether Deutsch's notion of universal quantum computer will efficiently simulate any physical process. In 1994, Peter Shor proved that finding the prime factors of a composite number and the discrete logarithm problem (i.e. finding the positive value of s such that $b = a^s$ for the given positive integers a and b) could be solved efficiently by a quantum computer. This may be a pointer to proving that quantum computers are more efficient than Turing machines (and classical computers).

12.8.3 Power of Quantum Computation

In classical complexity theory, the classes **P** and **NP** play a major role, but there are other classes of interest. Some of them are given below:

L—The class of all decision problems which may be decided by a TM running in logarithmic space.

PSPACE—The class of decision problems which may be decided on a Turing machine using a polynomial number of working bits, with no limitation on the amount of time that may be used by the machine.

EXP—The class of all decision problems which may be decided by a TM in exponential time, that is, $O(2^{n^k})$, k being a constant.

The hierarchy of these classes is given by

$$\mathbf{L} \subseteq \mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} \subseteq \mathbf{EXP}$$

The inclusions are strongly believed to be strict but none of them has been proved so far in classical complexity theory.

We also have two more classes.

BPP—The class of problems that can be solved using the randomized algorithm in polynomial time, if a bounded probability of error (say 1/10) is allowed in the solution of the problem.

BQP—The class of all computational problems which can be solved efficiently (in polynomial time) on a quantum computer where a bounded probability of error is allowed. It is easy to see that **BPP** $\subseteq$ **BQP**. The class **BQP** lies somewhere between **P** and **PSPACE**, but where exactly it lies with respect to **P**, **NP** and **PSPACE** is not known.

It is easy to give non-constructive proofs that many problems are in **EXP**, but it seems very hard to prove that a particular class of problems is in **EXP** (the possibility of a polynomial algorithm of these problems cannot be ruled out).

As far as quantum computation is concerned, two important classes are considered. One is **BQP**, which is analogous to **BPP**. The other is **NPI** (**NP** intermediate) defined by

NPI — The class of problems which are neither in **P** nor *NP*-complete

Once again, no problem is shown to be in **NPI**. In that case **P** ≠ **NP** is established.

Two problems are likely to be in **NPI**, one being the factoring problem (i.e. given a composite number n to find its prime factors) and the other being the graph isomorphism problems (i.e. to find whether the given undirected graphs with the same set of vertices are isomorphic).

A quantum algorithm for factoring has been discovered. Peter Shor announced a quantum order-finding algorithm and proved that factoring could be reduced to order-finding. This has motivated a search for a fast quantum algorithm for other problems suspected to be in **NPI**.

Grover developed an algorithm called the quantum search algorithm. A loose formulation of this means that a quantum computer can search a particular item in a list of N items in $O(\sqrt{N})$ time and no further improvement is possible. If it were $O(\log N)$, then a quantum computer can solve an *NP*-complete problem in an efficient way. Based on this, some researchers feel that the class **BQP** cannot contain the class of *NP*-complete problems.

If it is possible to find some structure in the class of *NP*-complete problems then a more efficient algorithm may become possible. This may result in finding efficient algorithms for *NP*-complete problems. If it is possible to prove that quantum computers are strictly more powerful than classical computers, then it will follow that **P** is properly contained in **PSPACE**. Once again, there is no proof so far for $\mathbf{P} \subsetneq \mathbf{PSPACE}$.

12.8.4 CONCLUSION

Deutsch proposed the first blueprint of a quantum computer. As a single qubit can store two states 0 and 1 in quantum superposition, adding more qubits to the memory register will increase the storage capacity exponentially. When this happens, exponential complexity will reduce to polynomial complexity. Peter Shor's algorithm led to the hope that quantum computer may work efficiently on problems of exponential complexity.

But problems arise at the implementation stage. When more interacting qubits are involved in a circuit, the surrounding environment is affected by those interactions. It is difficult to prevent them. Also quantum computation will spread outside the computational unit and will irreversibly dissipate useful information to the environment. This process is called *decoherence*. The problem is to make qubits interact with themselves but not with the environment. Some physicists are pessimistic and conclude that the efforts cannot go beyond a few simple experiments involving only a few qubits.

But some researchers are optimistic and believe that efforts to control decoherence will bear fruit in a few years rather than decades.

It remains a fact that optimism, however overstretched, makes things happen. The proof of Fermat's last theorem and the four colour problem are examples of these. Thomas Watson, the Chairman of IBM, predicted in 1943, "I think there is a world market for maybe five computers". But the growth of computers has very much surpassed his prediction.

Charles Babbage (1791–1871) conceived of most of the essential elements of a modem computer in his analytical engine. But there was not sufficient technology available to implement his ideas. In 1930s, Alan Turing and John von Neumann thought of a theoretical model. These developments in 'Software' were matched by 'Hardware' support, resulting in the first computer in the early 1950s. Then, the microprocessors in 1970s led to the design of smaller computers with more capacity and memory.

But computer scientists realized that hardware development will improve the power of a computer only by a multiplicative constant factor. The study of **P** and **NP** led to developing approximate polynomial algorithms to *NP*-complete problems. Once again the importance of software arose. Now the quantum computers may provide the impetus to the development of computers from the hardware side.

The problem of developing quantum computers seems to be very hard but the history of sciences indicates that quantum computers may rule the universe in a few decades.

12.9 SUPPLEMENTARY EXAMPLES

EXAMPLE 12.4

Suppose that there is an *NP*-complete problem P that has a deterministic solution taking $O(n^{\log n})$ time (here $\log n$ denotes $\log_2 n$). What can you say about the running time of any other *NP*-complete problem Q?

Solution

As $Q \in$ **NP**, there exists a polynomial $p(n)$ such that the time for reduction of Q to P is atmost $p(n)$. So the running time for Q is $O(p(n) + p(n)^{\log p(n)})$. As $p(n)^{\log p(n)}$ dominates $p(n)$, we can omit $p(n)$ in $p(n) + p(n)^{\log p(n)}$. If the degree of $p(n)$ is k, then $p(n) = O(n^k)$. So we can replace $p(n)$ by n^k. So $p(n)^{\log p(n)} = O((n^k)^{k \log n}) = O(n^{k^2 \log n})$. Hence the running time of Q is $O^{(c \log n)}$ for some constant c.

EXAMPLE 12.5

Show that **P** is closed under (a) union, (b) concatenation, and (c) complementation.

Solution

Let L_1 and L_2 be two languages in **P**. Let w be an input of length n.

(a) To test whether $w \in L_1 \cup L_2$, we test whether $w \in L_1$. This takes polynomial time $p(n)$. If $w \notin L_1$, test another $w \in L_2$. This takes polynomial time $q(n)$. The total time taken for testing whether $w \in L_1 \cup L_2$ is $p(n) + q(n)$, which is also a polynomial in n. Hence $L_1 \cup L_2 \in$ **P**.

(b) Let $w = x_1x_2 \ldots x_n$. For each k, $1 \le k \le n - 1$, test whether $x_1x_2 \ldots x_k \in L_1$ and $x_{k+1}x_{k+2} \ldots x_n \in L_2$. If this happens, $w \in L_1L_2$. If the test fails for all k, $w \notin L_1L_2$. The time taken for this test for a particular k is $p(n) + q(n)$, where $p(n)$ and $q(n)$ are polynomials in n. Hence the total time for testing for all k's is at most n times the polynomial $p(n) + q(n)$. As $n(pn) + q(n)$ is a polynomial, $L_1L_2 \in$ **P**.

(c) Let M be the polynomial time TM for L_1. We construct a new TM M_1 as follows:

1. Each accepting state of M is a nonaccepting state of M_1 from which there are no further moves. So if M accepts w, M_1 on reading w will halt without accepting.
2. Let q_f be a new state, which is the accepting state of M_1. If $\delta(q, a)$ is not defined in M, define $\delta_{M_1}(q, a) = (q_f, a, R)$. So, $w \notin L$ if and only if M accepts w and halts. Also M_1 is a polynomial-time TM. Hence $L_1^c \in$ **P**.

EXAMPLE 12.6

Show that every language accepted by a standard TM M is also accepted by a TM M_1 with the following conditions:

1. M_1's head never moves to the left of its initial position.
2. M_1 will never write a blank.

Solution

It is easy to implement Condition 2 on the new machine. For the new TM, create a new blank b'. If the blank is written by M, the new Turing machine writes b'. The move of this new TM on seeing b' is the same as the move of M for b. The new TM satisfies the Condition 2. Denote the modified TM by M itself. Define the modified M by

$$M = (Q, \Sigma, \Gamma, \delta, q_2, b, F)$$

Define a new TM M_1 as

$$M_1 = (Q_1, \Sigma \times \{b\}, \Gamma_1, \delta_1, q_0, [b, b], F_1)$$

where

$$Q_1 = \{q_0, q_1\} \cup (Q \times \{U, L\})$$

$$\Gamma_1 = (\Gamma \times \Gamma) \cup \{[x, *] \mid x \in \Gamma\}$$

q_0 and q_1 are used to initiate the initial move of M. The two-way infinite tape of M is divided into two tracks as in Table 12.4. Here $*$ is the marker for the leftmost cell of the lower track. The state $[q, U]$ denotes that M_1 simulates M on the upper track. $[q, L]$ dentoes that M_1 simulates M on the lower track. If M moves to the left of the cell with $*$, M_1 moves to the right of the lower track.

TABLE 12.4 Folded Two-way Tape

X_0	X_1	X_2	.	. .	
$*$	X_{-1}	X_{-2}	X_{-3}	.	. .

We can define F_1 of M_1 by

$$F_1 = F \times \{U, L\}$$

We can describe δ as follows:

1. $\delta_1(q_0, [a, b]) = (q_1, [a, *], R)$

$$\delta(q_1, [X, b]) = ([q_2, U], [X, b], L)$$

By Rule 1, M_1 marks the leftmost cell in the lower track with $*$ and initiates the initial move of M.

2. If $\delta(q, X) = (p, Y, D)$ and $Z \in \Gamma$, then:

(i) $\delta_1([q, U], [X, Z]) = ([p, U], [Y, Z], D)$ and

(ii) $\delta_1([q, L], [Z, X]) = ([p, L], [Z, Y], \bar{D})$

where $\bar{D} = L$ if $D = R$ and $\bar{D} = R$ if $D = L$.

By Rule 2, M_1 simulates the moves of M on the appropriate track. In (i) the action is on the upper track and Z on the lower track is not changed. In (ii) the action is on the lower track and hence the movement is in the opposite direction $\bar{D}$; the symbol in the upper track is not changed.

3. If $\delta(q, X) = (p, Y, R)$ then

$$\delta_1([q, L], [X, *]) = \delta_1([q, U], [X, *]) = ([p, U], [Y, *], R)$$

When M_1 see $*$ in the lower track, M moves right and simulates M on the upper track.

4. If $\delta(q, X) = (p, Y, L)$, then

$$\delta_1([q, L], [X, *]) = \delta_1([q, U], [X, *]) = ([p, L], [Y, *], R)$$

When M_1 sees $*$ in the lower track and M's movement is to the left of the cell of the two-way tape corresponding to the $*$ cell in the lower track, the M's movement is to X_{-1} and the M_1's movement is also to X_{-1} but towards the right. As the tape of M is folded on the cell with $*$. the movement of M to the left of the $*$ cell is equivalent to the movement of M_1 to the right.

M reaches q in F if and only if M_1 reaches $[q, L]$ or $[q, R]$. Hence $T(M) = T(M_1)$.

EXAMPLE 12.7

We can define the 2SAT problem as the satisfiability problem for boolean expressions written as $\wedge$ of clauses having two or fewer variables. Show that 2SAT is in **P**.

Solution

Let the boolean expression E be an instance of the 2SAT problem having n variables.

Step 1 Let E have clauses consisting of a single variable (x_i or $\overline{x}_i$). If (x_i) appears as a clause in E, then x_i has to be assigned the truth value T in order to make E satisfiable. Assign the truth value T to x_i. Once x_i has the truth value T, then $(x_i \vee x_j)$ has the truth value T irrespective of the truth value of x_j (Note that x_j can also be $\overline{x}_j$). So $(x_i \vee x_j)$ or $(x_i \vee \overline{x}_j)$ can be deleted from E. If E contains $(\overline{x}_i \vee x_j)$ as a clause, then x_j should be assigned the truth value T in order to make E satisfiable. Hence we replace $(\overline{x}_i \vee x_j)$ by x_j in E so that x_j should be assigned the truth value T is order to make E satisfiable. Hence we replace $(\overline{x}_i \vee x_j)$ by x_j in E so that x_j can be assigned the truth value T later. If we repeat this process of eliminating clauses with a single variable (or its negation), we end up in two cases.

Case 1 We end up with $(x_i) \wedge (\overline{x}_i)$. In this case E is not satisfiable for any assignment of truth values. We stop.

Case 2 In this case all clauses of E have two variables. (A typical clause is $x_i \vee x_j$ or $x_i \vee \overline{x}_j$.)

Step 2 We have the apply step 2 only in Case 2 of step 1. We have already assigned truth values for variables not appearing in the reduced expression E. Choose one of the remaining variables appearing in E. If we have chosen x_i, assign the truth value T to x_i. Delete $x_i \vee x_j$ or $x_i \vee \overline{x}_j$ from E. If $\overline{x}_i \vee x_j$ appears in E, delete $\overline{x}_i$ to get (x_j). Repeat step 1 for clauses consisting of a single variable. If Case 1 occurs, assign the truth value F for x_i and proceed with E that we had before applying step 1.

Proceeding with these iterations, we end up either in unsatisfiability of E or satisfiability of E.

Step 2 consists of repetition of step 1 at most n times and step 1 requires $O(n)$ basic steps.

Let n be the number of clauses in E. Step 1 consists of deleting $(x_i \vee x_j)$ from E or deleting $\overline{x}_i$ from $(\overline{x}_i \vee x_j)$. This is done at most n times for each clause. In step 2, step 1 is applied at most two times, one for x_i and the second for $\overline{x}_i$. As the number of variables appearing in E is less than or equal to n, we delete $(x_i \vee x_j)$ or delete $\overline{x}_i$ from $(\overline{x}_i \vee x_j)$ at most $O(n)$ times while applying steps 1 and 2 repeatedly. Hence 2SAT is in **P**.

SELF-TEST

Choose the correct answer to Questions 1–7:

1. If $f(n) = 2n^3 + 3$ and $g(n) = 10000n^2 + 1000$, then:
   (a) the growth rate of g is greater than that of f.
   (b) the growth rate of f is greater than that of g.
   (c) the growth rate of f is equal to that of g.
   (d) none of these.

2. If $f(n) = n^3 + 4n + 7$ and $g(n) = 1000n^2 + 10000$, then $f(n) + g(n)$ is
   (a) $O(n^2)$
   (b) $O(n)$
   (c) $O(n^3)$
   (d) $O(n^5)$

3. If $f(n) = O(n^k)$ and $g(n) = O(n^l)$, then $f(n)g(n)$ is
   (a) $\max\{k, l\}$
   (b) $k + l$
   (c) kl
   (d) none of these.

4. The gcd of (1024, 28) is
   (a) 2
   (b) 4
   (c) 7
   (d) 14

5. $\lceil 10.7 \rceil + \lceil 9.9 \rceil$ is equal to
   (a) 19
   (b) 20
   (c) 18
   (d) none of these.

6. $\log_2 1024$ is equal to
   (a) 8
   (b) 9
   (c) 10
   (d) none of these.

7. The truth value of $f(x, y, z) = (x \vee \neg y) \wedge (\neg x \vee y) \wedge z$ is T if x, y, z have the truth values
   (a) T, T, T
   (b) F, F, F
   (c) T, F, F
   (d) F, T, F

State whether the following Statements 8–15 are true or false.

8. If the truth values of x, y, z are T, F, F respectively, then the truth value of $f(x, y, z) = x \wedge \neg(y \vee z)$ is T.
9. The complexity of a k-tape TM and an equivalent standard TM are the same.
10. If the time complexity of a standard TM is polynomial, then the time complexity of an equivalent k-tape TM is exponential.
11. If the time complexity of a standard TM is polynomial, then the time complexity of an equivalent NTM is exponential.
12. $f(x, y, z) = (x \vee y \vee z) \wedge (\neg x \wedge \neg y \wedge \neg z)$ is satisfiable.
13. $f(x, y, z) = (x \vee y) \wedge (\neg x \wedge \neg y)$ is satisfiable.
14. If f and g are satisfiable expressions, then $f \vee g$ is satisfiable.
15. If f and g are satisfiable expressions, then $f \wedge g$ is satisfiable.

EXERCISES

12.1 If $f(n) = O(n^k)$ and $g(n) = O(n^l)$, then show that $f(n) + g(n) = O(n^t)$ where $t = \max\{k, l\}$ and $f(n)g(n) = O(n^{k+l})$.

12.2 Evaluate the growth rates of (i) $f(n) = 2n^2$, (ii) $g(n) = 10n^2 + 7n \log n + \log n$, (iii) $h(n) = n^2 \log n + 2n \log n + 7n + 3$ and compare them.

12.3 Use the O-notation to estimate (i) the sum of squares of first n natural numbers, (ii) the sum of cubes of first n natural numbers, (iii) the sum of the first n terms of a geometric progression whose first term is a and the common ratio is r, and (iv) the sum of the first n terms of the arithmetic progression whose first term is a and the common difference is d.

12.4 Show that $f(n) = 3n^2 \log_2 n + 4n \log_3 n + 5 \log_2 \log_2 n + \log n + 100$ dominates n^2 but is dominated by n^3.

12.5 Find the gcd (294, 15) using the Euclid's algorithm.

12.6 Show that there are five truth assignments for (P, Q, R) satisfying $P \vee (\neg P \wedge \neg Q \wedge R)$.

12.7 Find whether $(P \wedge Q \wedge R) \wedge \neg Q$ is satisfiable.

12.8 Is $f(x, y, z, w) = (x \vee y \vee z) \wedge (\overline{x} \vee \overline{y} \vee \overline{z})$ satisfiable?

12.9 The set of all languages whose complements are in **NP** is called **CO-NP**. Prove that **NP** = **CO-NP** if and only if there is some *NP*-complete problem whose complement is in **NP**.

12.10 Prove that a boolean expression E is a tautology if and only if $\neg E$ is unsatisfiable (refer to Chapter 1 for the definition of tautology).