# MODULE 5

## 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) TM with more than one tape- called as multitape TM, and
(ii)TM where $\delta(q, a) = \{(p_1, y_1, D_1), (p_2, y_2, D_2), \bullet\bullet\bullet\bullet (p_r, y_r, D_r)\}$ – called as nondeterministic TM.

(i) **Multitape TM:** A multitape TM has :

- a finite set Q of states.
- an initial state $q_o$.
- 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.

assume that $\Sigma \subseteq \Gamma$ and $b \notin \Sigma$.)

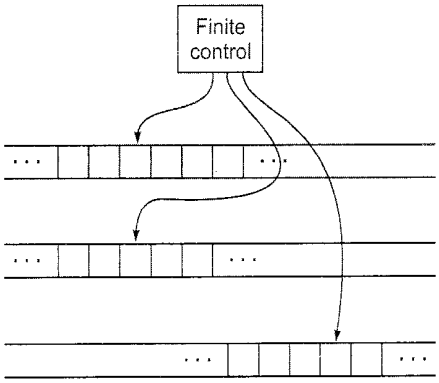



Fig. 9.8 Multitape Turing machine.

- 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 x $\Gamma^k$ into Q x $\Gamma^k$ x {L, R, S}$^k$.
  A move depends on the current state and k tape symbols under k tape heads.
- 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 - l)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 M1. 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 A2 and B5 are the current symbols to be scanned and so the head marker X is above the two symbols.

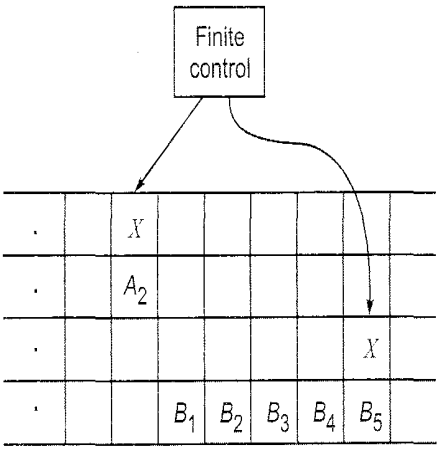



Fig. 9.9 Simulation of multitape TM.

Initially the contents of tapes 1 and 2 of *M* are stored in the second and fourth tracks of MI' The head markers of the first and third tracks are at the cells containing the first symbol.

To simulate a move of *M,* the 2k-track TM M1 has to visit the two head markers and store the scanned symbols in its control. Keeping track of the head markers visited and those to be visited is achieved by keeping a count and storing it in the finite control of M1' Note that the finite control of M1 has also the information about the states of *M* and its moves. After visiting both head markers. M1 knows the tape symbols being scanned by the two heads of *M.*

- M1 revisits each of the head markers:

(i) It changes the tape symbol in the corresponding track of M1 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 head markers to the left or right.
(iii) M1 changes the state of M in its control.

- This is the simulation of a single move of M.

**Theorem 9.2: If M1 is the single-tape TM simulating multitape TM M, then the time taken by M1 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 $O(n^2)$.

**(ii) Non Deterministic TM:**

**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. $\Gamma$ is a finite nonempty set of tape symbols
3. $b \in \Gamma$ is called the blank symbol
4. $\Sigma$ is a nonempty subset of $\Gamma$, 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. $\delta$ is a partial function from $Q \times \Gamma$ into the power set of $Q \times \Gamma \times \{L, R\}$.

**Theorem 9.3: If M is a nondeterministic TM, there is a deterministic TM M1 such that T(M) = T(M1)**

**Proof:** We construct M1 as a multitape TM. Each symbol in the input string leads to a change in ID. M1 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 M1.

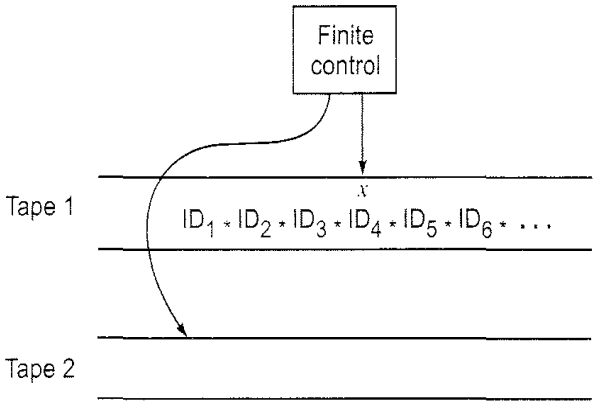



Fig. 9.10 The deterministic TM simulating *M*.

1. M1 examines the state and the scanned symbol of the current ID. Using the knowledge of moves of *M stored in the finite control of M1, it* checks whether the state in the current ID is an accepting state of *M*. In this case M1 accepts and stops simulating *M*.
2. If the state q say in the current ID xqay, is not an accepting state of M1 and δ(q, a) has k triples, M1 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. M1 modifies these k IDs in tape 2 according to the k choices given by δ(q, a).
4. M1 returns to the marked current ID. erases the mark x and marks the next ID-separator * with x. Then M1 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 LINEAR BOUNDED AUTOMATON

This model is important because:
(a) the set of context-sensitive languages is accepted by the model.
(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 L contains two special symbols ¢ and $. ¢ is called the left-end marker which is entered in the leftmost 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 rightmost cell of the input tape and prevents the R/W head from getting off the right end of the tape. Both the end markers should not appear on any other cell within the input tape, and the RIW head should not print any other symbol over both the end markers.

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 end markers ¢ 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.

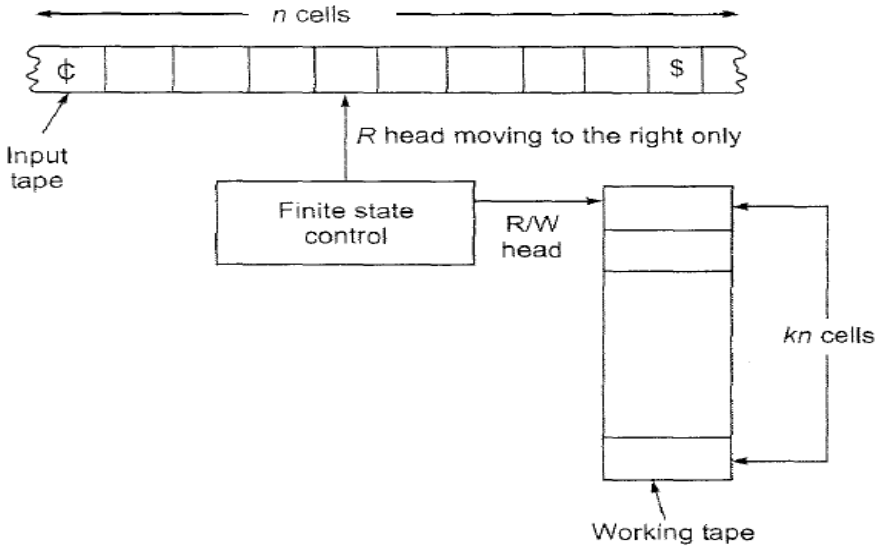



Fig. 9.11 Model of linear bounded automaton.

In the case of LEA, an ID is denoted by *(q, w. k),* where *q* € Q. *w* € Γ and *k* is some integer between 1 and *n.* The transition of IDs is similar except that *k* changes to *k* - 1 if the RIW 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

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

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

**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.

# CHAPTER 10

# DECIDABILITY/RECURSIVELY ENUMERABLE LANGUAGES

## 10.1 THE DEFINITION OF AN ALGORITHM

Algorithm is 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. 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 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

- 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 δ(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.

**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.

## 10.3 DECIDABLE LANGUAGES

**Definition 10.5**

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

**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. 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 H' in the TM *M.*
3. If the simulation ends in an accepting state of *B.* then *M* accepts *ω.*
If it ends in a non accepting 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, ∑, δ, $q_o$, *f* by strings of ∑* and input string W € ∑*. *M* checks whether *(B,* w) is a valid input. If not, it rejects *(B, ω)* and halts. If *(B,* w) is a valid input, *M* writes the initial state $q_o$ and the leftmost

input symbol of *w*. It updates the state using *0* 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**

$A_{CFG} = \{(G, w) \mid$ the context-free grammar $G$ 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. 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.

*(G,* w) is represented by representing the four components $V_N$, $\sum$, *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 $\{w_o, w_1 \ldots w_k\}$ of subsets of $(V \cup \sum)^*$, that terminates after a finite number of iterations. The given string $w \in \sum^*$ 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 $(G, w)$. ∎

## 10.4 UNDECIDABLE LANGUAGES

**Theorem 10.4** There exists a language over 2: that is not recursively enumerable.

***Proof*** A language $L$ is recursively enumerable if there exists a TM $M$ such that $L = T(M)$. As $\Sigma$ is finite, $\Sigma^*$ is countable (that is, there exists a one-to-one correspondence between $\Sigma^*$ 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 $\Sigma$. Then a member of $\mathcal{L}$ is a subset of $\Sigma^*$ (Note that $\Sigma^*$ is infinite even though $\Sigma$ 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 $\Sigma^*$ 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.

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

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

We define a subset $L$ of $\Sigma^*$ 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 $\Sigma^*$ 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 $\Sigma$ tnat is not recursively enumerable. ∎

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

**Theorem 10.5** $A_{\text{TM}}$ is undecidable.

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

$(M, w)$ is an input to $U$. Simulate $M$ on $w$. If $M$ enters an accepting state, $U$ accepts $(M, w)$. Hence $A_{\text{TM}}$ is recursively enumerable. We prove that $A_{\text{TM}}$ is undecidable by contradiction. We assume that $A_{\text{TM}}$ 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 TM

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 $x^4 - 3x^2 + 2 = 0$.

**Theorem 10.6** $HALT_{TM} = \{(M, w) \mid$ The Turing machine $M$ 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, $T(M_2) = \{(M, w) \mid$ 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 $\Sigma$ 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 $1 \leq i_j \leq n$, 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

| $bab^3$ | $b$ | $b$ | $ba$ | = | $ba$ | $b^3$ | $b^3$ | $a$ |
|---|---|---|---|---|---|---|---|---|
| $x_2$ | $x_1$ | $x_1$ | $x_3$ | | $y_2$ | $y_1$ | $y_1$ | $y_3$ |

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.

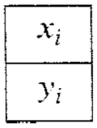
***Note:*** If the first substring used in PCP is always $x_1$ and $y_1$, then the PCP is known as the *Modified Post Correspondence Problem.*

**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




upper-half

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.

**Theorem 10.7** The PCP over $\Sigma$ 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 $\Sigma$ 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 $\Sigma$, show that $\bar{L}$ ( $\bar{L}$ 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 $T(M_1) = \bar{L}$ 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 $I$ is recursive.

### EXAMPLE 10.5

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

#### *Solution*

Let $M_1$ and $M_2$ be two TMs such that $L = T(M_1)$ and $\bar{L} = T(M_2)$. 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 $w \in \bar{L}$, 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 $\bar{L}$ 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 $\bar{A}_{TM}$ is not recursively enumerable.

#### *Solution*

We have already seen that $A_{TM}$ is recursively enumerable (by Theorem 10.5). If $\bar{A}_{TM}$ 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 $\bar{A}_{TM}$ 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.

# CHAPTER 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 nondeterministic polynomial. Another important class is the class of NP-complete problems which is a subclass of NP.

## 12.1 GROWTH RATE OF FUNCTIONS

**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

$$f(n) \leq Cg(n) \qquad \text{for all } n \geq N_0.$$

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

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

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, $a_k \geq 1$.

As $a_{k-1}, a_{k-2}, \ldots, a_1, a_0$ and $k$ are fixed integers, choose $N_0$ such that for all $n \geq N_0$ 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 $a_k \geq 1$, $\dfrac{p(n)}{n^k} = a_k + \dfrac{a_{k-1}}{n} + \cdots + \dfrac{a_1}{n^{k-1}} + \dfrac{a_0}{n^k} > 0 \qquad$ for all $n \geq N_0$

Also,

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

$$\leq a_k + 1 \qquad \text{by } (*)$$

So,

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

Hence,

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

**Definition 12.2** An exponential function is a function $q : N \rightarrow N$ 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 $\infty$.)

**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, $\frac{\log n}{n}$ tends to 0 as $n \rightarrow \infty$. (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, $k\left(\frac{\log n}{n}\right)$ tends to 0 and hence $z(n)$ tends to 0.

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

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

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

## 12.2 CLASSES P AND NP

**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 $T(n)$.

**Ex: Construct the time complexity *T(n)* for the Turing machine *M ={0$^n$1$^n$:n>=1}***

### *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*.

**Ex:** 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 $T(n) = nO(n) = O(n^2)$.

## 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

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

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 $\alpha$ and $\beta$ 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

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

The matrix $\begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}$ is a unitary matrix. (A matrix $A$ is unitary if $A$ 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

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

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

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

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**

*An 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).

**********