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 $\mathbf{P} \neq \mathbf{NP}$. The computer scientists and mathematicians strongly believe that $\mathbf{P} \neq \mathbf{NP}$, 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 $p(n) = a_kn^k + a_{k-1}n^{k-1} + \cdots + a_1n + a_0$ is a polynomial of degree k over Z and $a_k > 0$, then $p(n) = O(n^k)$.

Proof $p(n) = a_kn^k + a_{k-1}n^{k-1} + \cdots + a_1n + a_0$. 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 $a_k \ge 1$, $\frac{p(n)}{n^k} = a_k + \frac{a_{k-1}}{n} + \cdots + \frac{a_1}{n^{k-1}} + \frac{a_0}{n^k} > 0 \qquad$ 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

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

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

So we can choose N_0 such that $z(n) \le a$ 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**

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. $d_{ij} \le d_{ik} + d_{kj}$ 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

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

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

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

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

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.

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 $P \vee (\neg P \wedge \neg Q \wedge R)$.