(a)

	n					
Function	10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

(b)

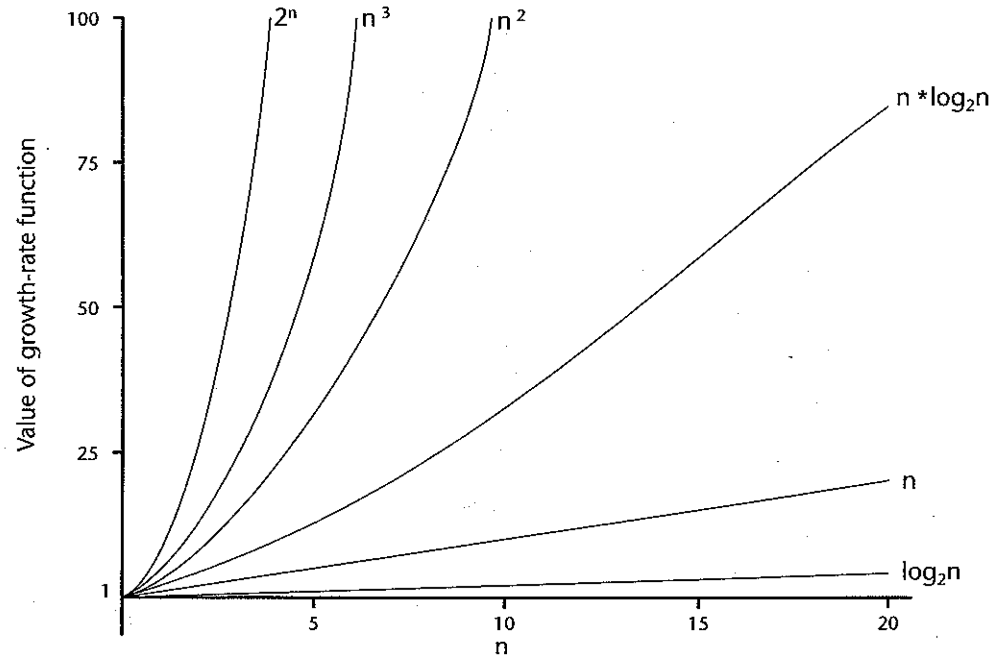

Figure 9-3

A comparison of growth-rate functions: (a) in tabular form; (b) in graphical form

To dramatize further the significance of an algorithm's proportional growth rate, consider the table and graph in Figure 9-3. The table (Figure 9-3a) gives, for various values of *n*, the approximate values of some common growth-rate functions, which are listed in order of growth:

$$O(1) < O(\log_2 n) < O(n) < O(n * \log_2 n) < O(n^2) < O(n^3) < O(2^n)$$

Order of growth of some common functions

The table demonstrates the relative speed at which the values of the functions grow. (Figure 9-3b represents the growth-rate functions graphically.[3])

These growth-rate functions have the following intuitive interpretations:

Intuitive interpretations of growth-rate functions

1	A growth-rate function of 1 implies a problem whose time requirement is constant and, therefore, independent of the problem's size n.
$\log_2 n$	The time requirement for a **logarithmic** algorithm increases slowly as the problem size increases. If you square the problem size, you only double its time requirement. Later you will see that the recursive binary search algorithm that you studied in Chapter 2 has this behavior. Recall that a binary search halves an array and then searches one of the halves. Typical logarithmic algorithms solve a problem by solving a smaller constant fraction of the problem. The base of the log does not affect a logarithmic growth rate, so you can omit it in a growth-rate function. Exercise 6 at the end of this chapter asks you to show why this is true.
n	The time requirement for a **linear** algorithm increases directly with the size of the problem. If you square the problem size, you also square its time requirement.
$n * \log_2 n$	The time requirement for an $n * \log_2 n$ algorithm increases more rapidly than a linear algorithm. Such algorithms usually divide a problem into smaller problems that are each solved separately. You will see an example of such an algorithm—the mergesort—later in this chapter.
n^2	The time requirement for a **quadratic** algorithm increases rapidly with the size of the problem. Algorithms that use two nested loops are often quadratic. Such algorithms are practical only for small problems. Later in this chapter, you will study several quadratic sorting algorithms.
n^3	The time requirement for a **cubic** algorithm increases more rapidly with the size of the problem than the time requirement for a quadratic algorithm. Algorithms that use three nested loops are often cubic, and are practical only for small problems.
2^n	As the size of a problem increases, the time requirement for an **exponential** algorithm usually increases too rapidly to be practical.

[3]The graph of $f(n) = 1$ is omitted because the scale of the figure makes it difficult to draw. It would, however, be a straight line parallel to the x axis through $y = 1$.