$c_1n^2 + c_2n \leq c_3n$ for $n \leq 98$ and $c_1n^2 + c_2n > c_3n$ for $n > 98$. If $c_1 = 1$, $c_2 = 2$, and $c_3 = 1000$, then $c_1n^2 + c_2n \leq c_3n$ for $n \leq 998$.

No matter what the values of c_1, c_2, and c_3, there will be an n beyond which the algorithm with complexity c_3n will be faster than the one with complexity $c_1n^2 + c_2n$. This value of n will be called the *break-even point.* If the break-even point is zero, then the algorithm with complexity c_3n is always faster (or at least as fast). The exact break-even point cannot be determined analytically. The algorithms have to be run on a computer in order to determine the break-even point. To know that there is a break-even point, it is sufficient to know that one algorithm has complexity $c_1n^2 + c_2n$ and the other c_3n for some constants c_1, c_2, and c_3. There is little advantage in determining the exact values of c_1, c_2, and c_3.

1.3.3 Asymptotic Notation (O, Ω, Θ)

With the previous discussion as motivation, we introduce some terminology that enables us to make meaningful (but inexact) statements about the time and space complexities of an algorithm. In the remainder of this chapter, the functions f and g are nonnegative functions.

Definition 1.4 [Big "oh"] The function $f(n) = O(g(n))$ (read as "f of n is big oh of g of n") iff (if and only if) there exist positive constants c and n_0 such that $f(n) \leq c * g(n)$ for all n, $n \geq n_0$. □

Example 1.11 The function $3n + 2 = O(n)$ as $3n + 2 \leq 4n$ for all $n \geq 2$. $3n + 3 = O(n)$ as $3n + 3 \leq 4n$ for all $n \geq 3$. $100n + 6 = O(n)$ as $100n + 6 \leq 101n$ for all $n \geq 6$. $10n^2 + 4n + 2 = O(n^2)$ as $10n^2 + 4n + 2 \leq 11n^2$ for all $n \geq 5$. $1000n^2 + 100n - 6 = O(n^2)$ as $1000n^2 + 100n - 6 \leq 1001n^2$ for $n \geq 100$. $6 * 2^n + n^2 = O(2^n)$ as $6 * 2^n + n^2 \leq 7 * 2^n$ for $n \geq 4$. $3n + 3 = O(n^2)$ as $3n + 3 \leq 3n^2$ for $n \geq 2$. $10n^2 + 4n + 2 = O(n^4)$ as $10n^2 + 4n + 2 \leq 10n^4$ for $n \geq 2$. $3n + 2 \neq O(1)$ as $3n + 2$ is not less than or equal to c for any constant c and all $n \geq n_0$. $10n^2 + 4n + 2 \neq O(n)$. □

We write $O(1)$ to mean a computing time that is a constant. $O(n)$ is called *linear*, $O(n^2)$ is called *quadratic*, $O(n^3)$ is called *cubic*, and $O(2^n)$ is called *exponential.* If an algorithm takes time $O(\log n)$, it is faster, for sufficiently large n, than if it had taken $O(n)$. Similarly, $O(n \log n)$ is better than $O(n^2)$ but not as good as $O(n)$. These seven computing times–$O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$, and $O(2^n)$–are the ones we see most often in this book.

As illustrated by the previous example, the statement $f(n) = O(g(n))$ states only that $g(n)$ is an upper bound on the value of $f(n)$ for all n, $n \geq n_0$. It does not say anything about how good this bound is. Notice

that $n = O(2^n)$, $n = O(n^{2.5})$, $n = O(n^3)$, $n = O(2^n)$, and so on. For the statement $f(n) = O(g(n))$ to be informative, $g(n)$ should be as small a function of n as one can come up with for which $f(n) = O(g(n))$. So, while we often say that $3n + 3 = O(n)$, we almost never say that $3n + 3 = O(n^2)$, even though this latter statement is correct.

From the definition of O, it should be clear that $f(n) = O(g(n))$ is not the same as $O(g(n)) = f(n)$. In fact, it is meaningless to say that $O(g(n)) = f(n)$. The use of the symbol = is unfortunate because this symbol commonly denotes the equals relation. Some of the confusion that results from the use of this symbol (which is standard terminology) can be avoided by reading the symbol = as "is" and not as "equals."

Theorem 1.2 obtains a very useful result concerning the order of $f(n)$ (that is, the $g(n)$ in $f(n) = O(g(n))$) when $f(n)$ is a polynomial in n.

Theorem 1.2 If $f(n) = a_m n^m + \cdots + a_1 n + a_0$, then $f(n) = O(n^m)$.

Proof:

$$\begin{aligned} f(n) &\leq \textstyle\sum_{i=0}^{m} |a_i| n^i \\ &\leq n^m \textstyle\sum_{i=0}^{m} |a_i| n^{i-m} \\ &\leq n^m \textstyle\sum_{i=0}^{m} |a_i| \qquad \text{for } n \geq 1 \end{aligned}$$

So, $f(n) = O(n^m)$ (assuming that m is fixed). □

Definition 1.5 [Omega] The function $f(n) = \Omega(g(n))$ (read as "f of n is omega of g of n") iff there exist positive constants c and n_0 such that $f(n) \geq c * g(n)$ for all n, $n \geq n_0$. □

Example 1.12 The function $3n + 2 = \Omega(n)$ as $3n + 2 \geq 3n$ for $n \geq 1$ (the inequality holds for $n \geq 0$, but the definition of Ω requires an $n_0 > 0$). $3n+3 = \Omega(n)$ as $3n+3 \geq 3n$ for $n \geq 1$. $100n+6 = \Omega(n)$ as $100n+6 \geq 100n$ for $n \geq 1$. $10n^2 + 4n + 2 = \Omega(n^2)$ as $10n^2 + 4n + 2 \geq n^2$ for $n \geq 1$. $6 * 2^n + n^2 = \Omega(2^n)$ as $6 * 2^n + n^2 \geq 2^n$ for $n \geq 1$. Observe also that $3n + 3 = \Omega(1)$, $10n^2 + 4n + 2 = \Omega(n)$, $10n^2 + 4n + 2 = \Omega(1)$, $6 * 2^n + n^2 = \Omega(n^{100})$, $6 * 2^n + n^2 = \Omega(n^{50.2})$, $6 * 2^n + n^2 = \Omega(n^2)$, $6 * 2^n + n^2 = \Omega(n)$, and $6 * 2^n + n^2 = \Omega(1)$. □

As in the case of the big oh notation, there are several functions $g(n)$ for which $f(n) = \Omega(g(n))$. The function $g(n)$ is only a lower bound on $f(n)$. For the statement $f(n) = \Omega(g(n))$ to be informative, $g(n)$ should be as large a function of n as possible for which the statement $f(n) = \Omega(g(n))$ is true. So, while we say that $3n + 3 = \Omega(n)$ and $6 * 2^n + n^2 = \Omega(2^n)$, we almost never say that $3n + 3 = \Omega(1)$ or $6 * 2^n + n^2 = \Omega(1)$, even though both of these statements are correct.

Theorem 1.3 is the analogue of Theorem 1.2 for the omega notation.

Theorem 1.3 If $f(n) = a_m n^m + \cdots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \Omega(n^m)$.

Proof: Left as an exercise. □

Definition 1.6 [Theta] The function $f(n) = \Theta(g(n))$ (read as "f of n is theta of g of n") iff there exist positive constants c_1, c_2, and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all n, $n \geq n_0$. □

Example 1.13 The function $3n + 2 = \Theta(n)$ as $3n + 2 \geq 3n$ for all $n \geq 2$ and $3n + 2 \leq 4n$ for all $n \geq 2$, so $c_1 = 3$, $c_2 = 4$, and $n_0 = 2$. $3n + 3 = \Theta(n)$, $10n^2 + 4n + 2 = \Theta(n^2)$, $6 * 2^n + n^2 = \Theta(2^n)$, and $10 * \log n + 4 = \Theta(\log n)$. $3n + 2 \neq \Theta(1)$, $3n + 3 \neq \Theta(n^2)$, $10n^2 + 4n + 2 \neq \Theta(n)$, $10n^2 + 4n + 2 \neq \Theta(1)$, $6 * 2^n + n^2 \neq \Theta(n^2)$, $6 * 2^n + n^2 \neq \Theta(n^{100})$, and $6 * 2^n + n^2 \neq \Theta(1)$. □

The theta notation is more precise than both the the big oh and omega notations. The function $f(n) = \Theta(g(n))$ iff $g(n)$ is both an upper and lower bound on $f(n)$.

Notice that the coefficients in all of the $g(n)$'s used in the preceding three examples have been 1. This is in accordance with practice. We almost never find ourselves saying that $3n + 3 = O(3n)$, that $10 = O(100)$, that $10n^2 + 4n + 2 = \Omega(4n^2)$, that $6 * 2^n + n^2 = O(6 * 2^n)$, or that $6 * 2^n + n^2 = \Theta(4 * 2^n)$, even though each of these statements is true.

Theorem 1.4 If $f(n) = a_m n^m + \cdots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \Theta(n^m)$.

Proof: Left as an exercise. □

Definition 1.7 [Little "oh"] The function $f(n) = o(g(n))$ (read as "f of n is little oh of g of n") iff

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

□

Example 1.14 The function $3n + 2 = o(n^2)$ since $\lim_{n \to \infty} \frac{3n+2}{n^2} = 0$. $3n + 2 = o(n \log n)$. $3n + 2 = o(n \log \log n)$. $6 * 2^n + n^2 = o(3^n)$. $6 * 2^n + n^2 = o(2^n \log n)$. $3n + 2 \neq o(n)$. $6 * 2^n + n^2 \neq o(2^n)$. □

Analogous to o is the notation ω defined as follows.

Definition 1.8 [Little omega] The function $f(n) = \omega(g(n))$ (read as "f of n is little omega of g of n") iff

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$$

□

Example 1.15 Let us reexamine the time complexity analyses of the previous section. For the algorithm Sum (Algorithm 1.6) we determined that $t_{\mathsf{Sum}}(n) = 2n + 3$. So, $t_{\mathsf{Sum}}(n) = \Theta(n)$. For Algorithm 1.7, $t_{\mathsf{RSum}}(n) = 2n + 2 = \Theta(n)$. □

Although we might all see that the O, Ω, and Θ notations have been used correctly in the preceding paragraphs, we are still left with the question, Of what use are these notations if we have to first determine the step count exactly? The answer to this question is that the asymptotic complexity (i.e., the complexity in terms of O, Ω, and Θ) can be determined quite easily without determining the exact step count. This is usually done by first determining the asymptotic complexity of each statement (or group of statements) in the algorithm and then adding these complexities. Tables 1.4 through 1.6 do just this for Sum, RSum, and Add (Algorithms 1.6, 1.7, and 1.11).

Statement	s/e	frequency	total steps
1 **Algorithm** Sum(a, n)	0	–	$\Theta(0)$
2 {	0	–	$\Theta(0)$
3 $s := 0.0$;	1	1	$\Theta(1)$
4 **for** $i := 1$ **to** n **do**	1	$n + 1$	$\Theta(n)$
5 $s := s + a[i]$;	1	n	$\Theta(n)$
6 **return** s;	1	1	$\Theta(1)$
7 }	0	–	$\Theta(0)$
Total			$\Theta(n)$

Table 1.4 Asymptotic complexity of Sum (Algorithm 1.6)

Although the analyses of Tables 1.4 through 1.6 are carried out in terms of step counts, it is correct to interpret $t_P(n) = \Theta(g(n))$, $t_P(n) = \Omega(g(n))$, or $t_P(n) = O(g(n))$ as a statement about the computing time of algorithm P. This is so because each step takes only $\Theta(1)$ time to execute.

Statement	s/e	frequency $n=0$	frequency $n>0$	total steps $n=0$	total steps $n>0$
1 **Algorithm** RSum(a, n)	0	–	–	0	$\Theta(0)$
2 {	0	–	–	0	$\Theta(0)$
3 **if** $(n \leq 0)$ **then**	1	1	1	1	$\Theta(1)$
4 **return** 0.0;	1	1	0	1	$\Theta(0)$
5 **else return**					
6 RSum$(a, n-1) + a[n]$);	$1+x$	0	1	0	$\Theta(1+x)$
7 }	0	–	–	0	$\Theta(0)$
Total				2	$\Theta(1+x)$

$$x = t_{\mathsf{RSum}}(n-1)$$

Table 1.5 Asymptotic complexity of RSum (Algorithm 1.7).

Statement	s/e	frequency	total steps
1 **Algorithm** Add(a, b, c, m, n)	0	–	$\Theta(0)$
2 {	0	–	$\Theta(0)$
3 **for** $i := 1$ **to** m **do**	1	$\Theta(m)$	$\Theta(m)$
4 **for** $i := 1$ **to** n **do**	1	$\Theta(mn)$	$\Theta(mn)$
5 $c[i, j] := a[i, j] + b[i, j]$;	1	$\Theta(mn)$	$\Theta(mn)$
6 }	0	–	$\Theta(0)$
Total			$\Theta(mn)$

Table 1.6 Asymptotic complexity of Add (Algorithm 1.11)

After you have had some experience using the table method, you will be in a position to arrive at the asymptotic complexity of an algorithm by taking a more global approach. We elaborate on this method in the following examples.

Example 1.16 [Permutation generator] Consider Perm (Algorithm 1.4). When $k = n$, we see that the time taken is $\Theta(n)$. When $k < n$, the **else** clause is entered. At this time, the second **for** loop is entered $n - k + 1$ times. Each iteration of this loop takes $\Theta(n + t_{\mathsf{Perm}}(k+1, n))$ time. So, $t_{\mathsf{Perm}}(k,\ n) = \Theta((n-k+1)(n+t_{\mathsf{Perm}}(k+1,\ n)))$ when $k < n$. Since $t_{\mathsf{Perm}}(k+1,\ n)$ is at least n when $k+1 \leq n$, we get $t_{\mathsf{Perm}}(k,\ n) = \Theta((n-k+1)t_{\mathsf{Perm}}(k+1,\ n))$ for $k < n$. Using the substitution method, we obtain $t_{\mathsf{Perm}}(1, n) = \Theta(n(n!))$, $n \geq 1$. □

Example 1.17 [Magic square] The next example we consider is a problem from recreational mathematics. A magic square is an $n \times n$ matrix of the integers 1 to n^2 such that the sum of every row, column, and diagonal is the same. Figure 1.2 gives an example magic square for the case $n = 5$. In this example, the common sum is 65.

15	8	1	24	17
16	14	7	5	23
22	20	13	6	4
3	21	19	12	10
9	2	25	18	11

Figure 1.2 Example magic square

H. Coxeter has given the following simple rule for generating a magic square when n is odd:

> Start with 1 in the middle of the top row; then go up and left, assigning numbers in increasing order to empty squares; if you fall off the square imagine the same square as tiling the plane and continue; if a square is occupied, move down instead and continue.

The magic square of Figure 1.2 was formed using this rule. Algorithm 1.15 is for creating an $n \times n$ magic square for the case in which n is odd. This results from Coxeter's rule.

The magic square is represented using a two-dimensional array having n rows and n columns. For this application it is convenient to number the rows (and columns) from 0 to $n-1$ rather than from 1 to n. Thus, when the algorithm "falls off the square," the **mod** operator sets i and/or j back to 0 or $n-1$.

The time to initialize and output the square is $\Theta(n^2)$. The third **for** loop (in which *key* ranges over 2 through n^2) is iterated n^2-1 times and each iteration takes $\Theta(1)$ time. So, this **for** loop takes $\Theta(n^2)$ time. Hence the overall time complexity of Magic is $\Theta(n^2)$. Since there are n^2 positions in which the algorithm must place a number, we see that $\Theta(n^2)$ is the best bound an algorithm for the magic square problem can have. □

Example 1.18 [Computing x^n] Our final example is to compute x^n for any real number x and integer $n \geq 0$. A naive algorithm for solving this problem is to perform $n-1$ multiplications as follows:

power := x;
for $i := 1$ **to** $n-1$ **do** *power* := *power* $* x$;

This algorithm takes $\Theta(n)$ time. A better approach is to employ the "repeated squaring" trick. Consider the special case in which n is an integral power of 2 (that is, in which n equals 2^k for some integer k). The following algorithm computes x^n.

power := x;
for $i := 1$ **to** k **do** *power* := *power*^{2};

The value of *power* after q iterations of the **for** loop is x^{2^q}. Therefore, this algorithm takes only $\Theta(k) = \Theta(\log n)$ time, which is a significant improvement over the run time of the first algorithm.

Can the same algorithm be used when n is not an integral power of 2? Fortunately, the answer is yes. Let $b_k b_{k-1} \cdots b_1 b_0$ be the binary representation of the integer n. This means that $n = \sum_{q=0}^{k} b_q 2^q$. Now,

$$x^n = x^{\sum_{q=0}^{k} b_q 2^q} = (x)^{b_0} * (x^2)^{b_1} * (x^4)^{b_2} * \cdots * (x^{2^k})^{b_k}$$

Also observe that b_0 is nothing but n **mod** 2 and that $\lfloor n/2 \rfloor$ is $b_k b_{k-1} \cdots b_1$ in binary form. These observations lead us to Exponentiate (Algorithm 1.16) for computing x^n.

```
1   Algorithm Magic(n)
2   // Create a magic square of size n, n being odd.
3   {
4       if ((n mod 2) = 0) then
5       {
6           write ("n is even"); return;
7       }
8       else
9       {
10          for i := 0 to n − 1 do  // Initialize square to zero.
11              for j := 0 to n − 1 do  square[i, j] := 0;
12          square[0, (n − 1)/2] := 1; // Middle of first row
13          // (i, j) is the current position.
14          j := (n − 1)/2;
15          for key := 2 to n^2 do
16          {
17              // Move up and left. The next two if statements
18              // may be replaced by the mod operator if
19              // −1 mod n has the value n − 1.
20              if (i ≥ 1) then k := i − 1; else k := n − 1;
21              if (j ≥ 1) then l := j − 1; else l := n − 1;
22              if (square[k, l] ≥ 1) then i := (i + 1) mod n;
23              else // square[k,l] is empty.
24              {
25                  i := k; j := l;
26              }
27              square[i, j] := key;
28          }
29          // Output the magic square.
30          for i := 0 to n − 1 do
31              for j := 0 to n − 1 do write (square[i, j]);
32      }
33  }
```

Algorithm 1.15 Magic square

```
1   Algorithm Exponentiate(x, n)
2   // Return x^n for an integer n ≥ 0.
3   {
4       m := n; power := 1; z := x;
5       while (m > 0) do
6       {
7           while ((m mod 2) = 0) do
8           {
9               m := ⌊m/2⌋; z := z^2;
10          }
11          m := m − 1; power := power * z;
12      }
13      return power;
14  }
```

Algorithm 1.16 Computation of x^n

Proving the correctness of this algorithm is left as an exercise. The variable m starts with the value of n, and after every iteration of the innermost **while** loop (line 7), its value decreases by a factor of at least 2. Thus there will be only $\Theta(\log n)$ iterations of the **while** loop of line 7. Each such iteration takes $\Theta(1)$ time. Whenever control exits from the innermost **while** loop, the value of m is odd and the instructions $m := m - 1$; $power := power * z$; are executed once. After this execution, since m becomes even, either the innermost **while** loop is entered again or the outermost **while** loop (line 5) is exited (in case $m = 0$). Therefore the instructions $m := m - 1$; $power := power * z$; can only be executed $O(\log n)$ times. In summary, the overall run time of Exponentiate is $\Theta(\log n)$. □

1.3.4 Practical Complexities

We have seen that the time complexity of an algorithm is generally some function of the instance characteristics. This function is very useful in determining how the time requirements vary as the instance characteristics change. The complexity function can also be used to compare two algorithms P and Q that perform the same task. Assume that algorithm P has complexity $\Theta(n)$ and algorithm Q has complexity $\Theta(n^2)$. We can assert that algorithm P is faster than algorithm Q for sufficiently large n. To see the validity of this assertion, observe that the computing time of P is bounded

from above by cn for some constant c and for all n, $n \geq n_1$, whereas that of Q is bounded from below by dn^2 for some constant d and all n, $n \geq n_2$. Since $cn \leq dn^2$ for $n \geq c/d$, algorithm P is faster than algorithm Q whenever $n \geq \max\{n_1, n_2, c/d\}$.

You should always be cautiously aware of the presence of the phrase "sufficiently large" in an assertion like that of the preceding discussion. When deciding which of the two algorithms to use, you must know whether the n you are dealing with is, in fact, sufficiently large. If algorithm P runs in $10^6 n$ milliseconds, whereas algorithm Q runs in n^2 milliseconds, and if you always have $n \leq 10^6$, then, other factors being equal, algorithm Q is the one to use.

To get a feel for how the various functions grow with n, you are advised to study Table 1.7 and Figure 1.3 very closely. It is evident from Table 1.7 and Figure 1.3 that the function 2^n grows very rapidly with n. In fact, if an algorithm needs 2^n steps for execution, then when $n = 40$, the number of steps needed is approximately $1.1 * 10^{12}$. On a computer performing one billion steps per second, this would require about 18.3 minutes. If $n = 50$, the same algorithm would run for about 13 days on this computer. When $n = 60$, about 310.56 years are required to execute the algorithm and when $n = 100$, about $4 * 10^{13}$ years are needed. So, we may conclude that the utility of algorithms with exponential complexity is limited to small n (typically $n \leq 40$).

$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4,096	65,536
5	32	160	1,024	32,768	4,294,967,296

Table 1.7 Function values

Algorithms that have a complexity that is a polynomial of high degree are also of limited utility. For example, if an algorithm needs n^{10} steps, then using our 1-billion-steps-per-second computer, we need 10 seconds when $n = 10$, 3171 years when $n = 100$, and $3.17 * 10^{13}$ years when $n = 1000$. If the algorithm's complexity had been n^3 steps instead, then we would need one second when $n = 1000$, 110.67 minutes when $n = 10{,}000$, and 11.57 days when $n = 100{,}000$.

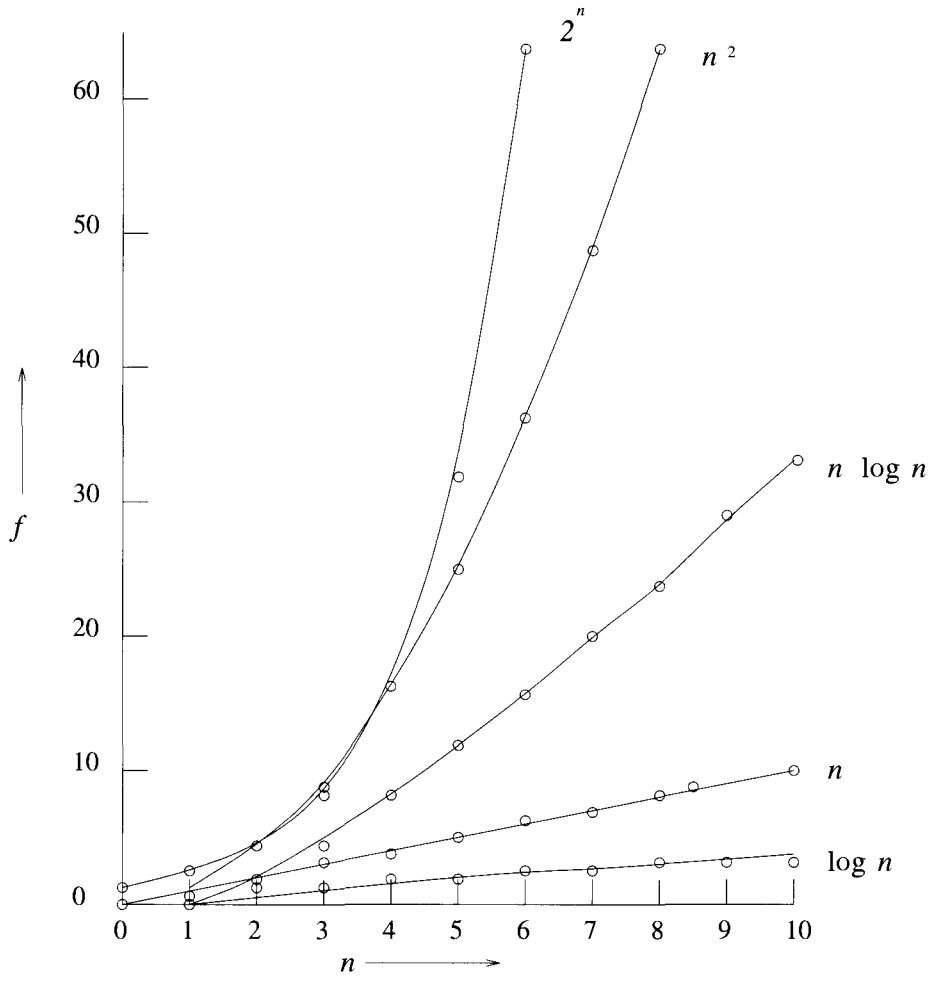

Figure 1.3 Plot of function values

Table 1.8 gives the time needed by a one-billion-steps-per-second computer to execute an algorithm of complexity $f(n)$ instructions. You should note that currently only the fastest computers can execute about 1 billion instructions per second. From a practical standpoint, it is evident that for reasonably large n (say $n > 100$), only algorithms of small complexity (such as n, $n \log n$, n^2, and n^3) are feasible. Further, this is the case even if you could build a computer capable of executing 10^{12} instructions per second. In this case, the computing times of Table 1.8 would decrease by a factor of 1000. Now, when $n = 100$, it would take 3.17 years to execute n^{10} instructions and $4 * 10^{10}$ years to execute 2^n instructions.

	Time for $f(n)$ instructions on a 10^9 instr/sec computer						
n	$f(n) = n$	$f(n) = n \log_2 n$	$f(n) = n^2$	$f(n) = n^3$	$f(n) = n^4$	$f(n) = n^{10}$	$f(n) = 2^n$
10	.01 μs	.03 μs	.1 μs	1 μs	10 μs	10 s	1 μs
20	.02 μs	.09 μs	.4 μs	8 μs	160 μs	2.84 hr	1 ms
30	.03 μs	.15 μs	.9 μs	27 μs	810 μs	6.83 d	1 s
40	.04 μs	.21 μs	1.6 μs	64 μs	2.56 ms	121.36 d	18.3 min
50	.05 μs	.28 μs	2.5 μs	125 μs	6.25 ms	3.1 yr	13 d
100	.1 μs	.66 μs	10 μs	1 ms	100 ms	3171 yr	$4*10^{13}$ yr
1,000	1 μs	9.96 μs	1 ms	1 s	16.67 min	$3.17*10^{13}$ yr	$32*10^{283}$ yr
10,000	10 μs	130 μs	100 ms	16.67 min	115.7 d	$3.17*10^{23}$ yr	
100,000	100 μs	1.66 ms	10 s	11.57 d	3171 yr	$3.17*10^{33}$ yr	
1,000,000	1 ms	19.92 ms	16.67 min	31.71 yr	$3.17*10^{7}$ yr	$3.17*10^{43}$ yr	

Table 1.8 Times on a 1-billion-steps-per-second computer

1.3.5 Performance Measurement

Performance measurement is concerned with obtaining the space and time requirements of a particular algorithm. These quantities depend on the compiler and options used as well as on the computer on which the algorithm is run. Unless otherwise stated, all performance values provided in this book are obtained using the Gnu C++ compiler, the default compiler options, and the Sparc 10/30 computer workstation.

In keeping with the discussion of the preceding section, we do not concern ourselves with the space and time needed for compilation. We justify this by the assumption that each program (after it has been fully debugged) is compiled once and then executed several times. Certainly, the space and time needed for compilation are important during program testing, when more time is spent on this task than in running the compiled code.

We do not consider measuring the run-time space requirements of a program. Rather, we focus on measuring the computing time of a program. To obtain the computing (or run) time of a program, we need a clocking procedure. We assume the existence of a program GetTime() that returns the current time in milliseconds.

Suppose we wish to measure the worst-case performance of the sequential search algorithm (Algorithm 1.17). Before we can do this, we need to (1) decide on the values of n for which the times are to be obtained and (2) determine, for each of the above values of n, the data that exhibit the worst-case behavior.

```
1  Algorithm SeqSearch(a, x, n)
2  // Search for x in a[1 : n]. a[0] is used as additional space.
3  {
4      i := n; a[0] := x;
5      while (a[i] ≠ x) do i := i − 1;
6      return i;
7  }
```

Algorithm 1.17 Sequential search

The decision on which values of n to use is based on the amount of timing we wish to perform and also on what we expect to do with the times once they are obtained. Assume that for Algorithm 1.17, our intent is simply to predict how long it will take, in the worst case, to search for x, given the size n of a. An asymptotic analysis reveals that this time is $\Theta(n)$. So, we expect a plot of the times to be a straight line. Theoretically, if we know the times for any two values of n, the straight line is determined, and we can obtain the time for all other values of n from this line. In practice, we need the times for more than two values of n. This is so for the following reasons:

1. Asymptotic analysis tells us the behavior only for sufficiently large values of n. For smaller values of n, the run time may not follow the asymptotic curve. To determine the point beyond which the asymptotic curve is followed, we need to examine the times for several values of n.

2. Even in the region where the asymptotic behavior is exhibited, the times may not lie exactly on the predicted curve (straight line in the case of Algorithm 1.17) because of the effects of low-order terms that are discarded in the asymptotic analysis. For instance, an algorithm with asymptotic complexity $\Theta(n)$ can have time complexity $c_1 n + c_2 \log n + c_3$ or, for that matter, any other function of n in which the highest-order term is $c_1 n$ for some constant c_1, $c_1 > 0$.

It is reasonable to expect that the asymptotic behavior of Algorithm 1.17 begins for some n that is smaller than 100. So, for $n > 100$, we obtain the

run time for just a few values. A reasonable choice is $n = 200, 300, 400, \ldots, 1000$. There is nothing magical about this choice of values. We can just as well use $n = 500, 1,000, 1,500, \ldots, 10,000$ or $n = 512, 1,024, 2,048, \ldots, 2^{15}$. It costs us more in terms of computer time to use the latter choices, and we probably do not get any better information about the run time of Algorithm 1.17 using these choices.

For n in the range [0, 100] we carry out a more-refined measurement, since we are not quite sure where the asymptotic behavior begins. Of course, if our measurements show that the straight-line behavior does not begin in this range, we have to perform a more-detailed measurement in the range [100, 200], and so on, until the onset of this behavior is detected. Times in the range [0, 100] are obtained in steps of 10 beginning at $n = 0$.

Algorithm 1.17 exhibits its worst-case behavior when x is chosen such that it is not one of the $a[i]$'s. For definiteness, we set $a[i] = i$, $1 \leq i \leq n$, and $x = 0$. At this time, we envision using an algorithm such as Algorithm 1.18 to obtain the worst-case times.

```
1   Algorithm TimeSearch()
2   {
3       for j := 1 to 1000 do  a[j] := j;
4       for j := 1 to 10 do
5       {
6           n[j] := 10 * (j - 1); n[j + 10] := 100 * j;
7       }
8       for j := 1 to 20 do
9       {
10          h := GetTime();
11          k := SeqSearch(a, 0, n[j]);
12          h1 := GetTime();
13          t := h1 - h;
14          write (n[j], t);
15      }
16  }
```

Algorithm 1.18 Algorithm to time Algorithm 1.17

The timing results of this algorithm is summarized in Table 1.9. The times obtained are too small to be of any use to us. Most of the times are zero; this indicates that the precision of our clock is inadequate. The nonzero times are just noise and are not representative of the time taken.

n	time	n	time
0	0	100	0
10	0	200	0
20	0	300	1
30	0	400	0
40	0	500	1
50	0	600	0
60	0	700	0
70	0	800	1
80	0	900	0
90	0	1000	0

Table 1.9 Timing results of Algorithm 1.18. Times are in milliseconds.

To time a short event, it is necessary to repeat it several times and divide the total time for the event by the number of repetitions.

Since our clock has an accuracy of about one-tenth of a second, we should not attempt to time any single event that takes less than about one second. With an event time of at least ten seconds, we can expect our observed times to be accurate to one percent.

The body of Algorithm 1.18 needs to be changed to that of Algorithm 1.19. In this algorithm, $r[i]$ is the number of times the search is to be repeated when the number of elements in the array is $n[i]$. Notice that rearranging the timing statements as in Algorithm 1.20 or 1.21 does not produce the desired results. For instance, from the data of Table 1.9, we expect that with the structure of Algorithm 1.20, the value output for $n = 0$ will still be 0. This is because there is a chance that in every iteration of the **for** loop, the clock does not change between the two times GetTime() is called. With the structure of Algorithm 1.21, we expect the algorithm never to exit the **while** loop when $n = 0$ (in reality, the loop will be exited because occasionally the measured time will turn out to be a few milliseconds).

Yet another alternative is shown in Algorithm 1.22. This approach can be expected to yield satisfactory times. It cannot be used when the timing procedure available gives us only the time since the last invocation of GetTime. Another difficulty is that the measured time includes the time needed to read the clock. For small n, this time may be larger than the time to run SeqSearch. This difficulty can be overcome by determining the time taken by the timing procedure and subtracting this time later.

```
1   Algorithm TimeSearch()
2   {
3       // Repetition factors
4       r[21] := {0, 200000, 200000, 150000, 100000, 100000, 100000,
5           50000, 50000, 50000, 50000, 50000, 50000, 50000, 50000,
6           50000, 50000, 25000, 25000, 25000, 25000};
7       for j := 1 to 1000 do a[j] := j;
8       for j := 1 to 10 do
9       {
10          n[j] := 10 * (j - 1); n[j + 10] := 100 * j;
11      }
12      for j := 1 to 20 do
13      {
14          h := GetTime();
15          for i := 1 to r[j] do k := SeqSearch(a, 0, n[j]);
16          h1 := GetTime();
17          t1 := h1 - h;
18          t := t1; t := t/r[j];
19          write (n[j], t1, t);
20      }
21  }
```

Algorithm 1.19 Timing algorithm

```
1   t := 0;
2   for i := 1 to r[j] do
3   {
4       h := GetTime();
5       k := SeqSearch(a, 0, n[j]);
6       h1 := GetTime();
7       t := t + h1 - h;
8   }
9   t := t/r[j];
```

Algorithm 1.20 Improper timing construct

```
1   t := 0;
2   while (t < DESIRED_TIME) do
3   {
4       h := GetTime();
5       k := SeqSearch(a, 0, n[j]);
6       h1 := GetTime();
7       t := t + h1 − h;
8   }
```

Algorithm 1.21 Another improper timing construct

```
1   h := GetTime(); t := 0;
2   while (t < DESIRED_TIME) do
3   {
4       k := SeqSearch(a, 0, n[j]);
5       h1 := GetTime();
6       t := h1 − h;
7   }
```

Algorithm 1.22 An alternate timing construct

Timing results of Algorithm 1.19, is given in Table 1.10. The times for n in the range [0, 1000] are plotted in Figure 1.4. Values in the range [10, 100] have not been plotted. The linear dependence of the worst-case time on n is apparent from this graph.

n	$t1$	t	n	$t1$	t
0	308	0.002	100	1683	0.034
10	923	0.005	200	3359	0.067
20	1181	0.008	300	4693	0.094
30	1087	0.011	400	6323	0.126
40	1384	0.014	500	7799	0.156
50	1691	0.017	600	9310	0.186
60	999	0.020	700	5419	0.217
70	1156	0.023	800	6201	0.248
80	1306	0.026	900	6994	0.280
90	1460	0.029	1000	7725	0.309

Times are in milliseconds

Table 1.10 Worst-case run times for Algorithm 1.17

The graph of Figure 1.4 can be used to predict the run time for other values of n. We can go one step further and get the equation of the straight line. The equation of this line is $t = c + mn$, where m is the slope and c the value for $n = 0$. From the graph, we see that $c = 0.002$. Using the point $n = 600$ and $t = 0.186$, we obtain $m = (t - c)/n = 0.184/600 = 0.0003067$. So the line of Figure 1.4 has the equation $t = 0.002 + 0.0003067n$, where t is the time in milliseconds. From this, we expect that when $n = 1000$, the worst-case search time will be 0.3087 millisecond, and when $n = 500$, it will be 0.155 millisecond. Compared to the observed times of Table 1.10, we see that these figures are very accurate!

Summary of Running Time Calculation

To obtain the run time of a program, we need to plan the experiment. The following issues need to be addressed during the planning stage:

1. What is the accuracy of the clock? How accurate do our results have to be? Once the desired accuracy is known, we can determine the length of the shortest event that should be timed.

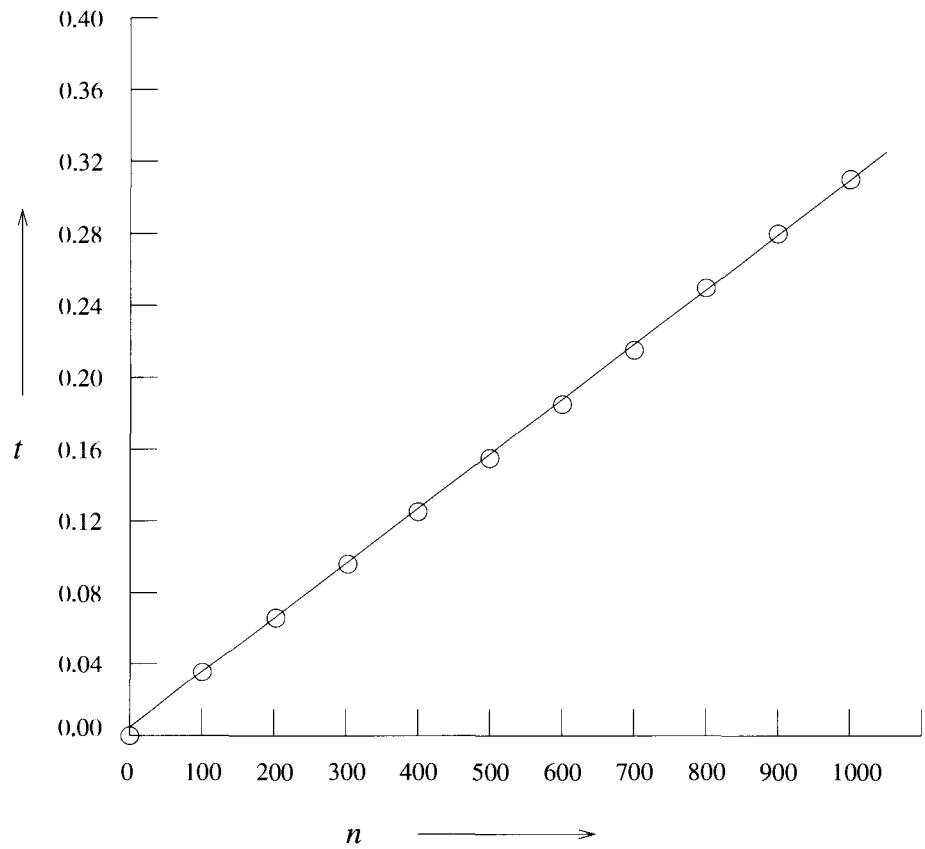

Figure 1.4 Plot of the data in Table 1.10

2. For each instance size, a repetition factor needs to be determined. This is to be chosen such that the event time is at least the minimum time that can be clocked with the desired accuracy.

3. Are we measuring worst-case or average performance? Suitable test data need to be generated.

4. What is the purpose of the experiment? Are the times being obtained for comparative purposes, or are they to be used to predict run times? If the latter is the case, then contributions to the run time from such sources as the repetition loop and data generation need to be subtracted (in case they are included in the measured time). If the former is the case, then these times need not be subtracted (provided they are the same for all programs being compared).

5. In case the times are to be used to predict run times, then we need to fit a curve through the points. For this, the asymptotic complexity should be known. If the asymptotic complexity is linear, then a least-squares straight line can be fit; if it is quadratic, then a parabola can be used (that is, $t = a_0 + a_1 n + a_2 n^2$). If the complexity is $\Theta(n \log n)$, then a least-squares curve of the form $t = a_0 + a_1 n + a_2 n \log_2 n$ can be fit. When obtaining the least-squares approximation, one should discard data corresponding to small values of n, since the program does not exhibit its asymptotic behavior for these n.

Generating Test Data

Generating a data set that results in the worst-case performance of an algorithm is not always easy. In some cases, it is necessary to use a computer program to generate the worst-case data. In other cases, even this is very difficult. In these cases, another approach to estimating worst-case performance is taken. For each set of values of the instance characteristics of interest, we generate a suitably large number of random test data. The run times for each of these test data are obtained. The maximum of these times is used as an estimate of the worst-case time for this set of values of the instance characteristics.

To measure average-case times, it is usually not possible to average over all possible instances of a given characteristic. Although it is possible to do this for sequential search, it is not possible for a sort algorithm. If we assume that all keys are distinct, then for any given n, $n!$ different permutations need to be used to obtain the average time. Obtaining average-case data is usually much harder than obtaining worst-case data. So, we often adopt the strategy outlined above and simply obtain an estimate of the average time on a suitable set of test data.

Whether we are estimating worst-case or average time using random data, the number of instances that we can try is generally much smaller than the total number of such instances. Hence, it is desirable to analyze the algorithm being tested to determine classes of data that should be generated for the experiment. This is a very algorithm-specific task, and we do not go into it here.

EXERCISES

1. Compare the two functions n^2 and $2^n/4$ for various values of n. Determine when the second becomes larger than the first.

2. Prove by induction:

   (a) $\sum_{i=1}^{n} i = n(n+1)/2,\ n \geq 1$

   (b) $\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6,\ n \geq 1$

   (c) $\sum_{i=0}^{n} x^i = (x^{n+1}-1)/(x-1),\ x \neq 1,\ n \geq 0$

3. Determine the frequency counts for all statements in the following two algorithm segments:

```
1  for i := 1 to n do
2      for j := 1 to i do
3          for k := 1 to j do
4              x := x + 1;
```

(a)

```
1  i := 1;
2  while (i ≤ n) do
3  {
4      x := x + 1;
5      i := i + 1;
6  }
```

(b)

4. (a) Introduce statements to increment *count* at all appropriate points in Algorithm 1.23.

   (b) Simplify the resulting algorithm by eliminating statements. The simplified algorithm should compute the same value for *count* as computed by the algorithm of part (a).

   (c) What is the exact value of *count* when the algorithm terminates? You may assume that the initial value of *count* is 0.

   (d) Obtain the step count for Algorithm 1.23 using the frequency method. Clearly show the step count table.

5. Do Exercise 4 for Transpose (Algorithm 1.24).

6. Do Exercise 4 for Algorithm 1.25. This algorithm multiplies two $n \times n$ matrices a and b.