8. Ackermann's function $A(m, n)$ is defined as follows:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,\ 1) & \text{if } n = 0 \\ A(m-1,\ A(m,\ n-1)) & \text{otherwise} \end{cases}$$

This function is studied because it grows very fast for small values of m and n. Write a recursive algorithm for computing this function. Then write a nonrecursive algorithm for computing it.

9. The *pigeonhole principle* states that if a function f has n distinct inputs but less than n distinct outputs, then there exist two inputs a and b such that $a \neq b$ and $f(a) = f(b)$. Present an algorithm to find a and b such that $f(a) = f(b)$. Assume that the function inputs are $1, 2, \ldots,$ and n.

10. Give an algorithm to solve the following problem: Given n, a positive integer, determine whether n is the sum of all of its divisors, that is, whether n is the sum of all t such that $1 \leq t < n$, and t divides n.

11. Consider the function $F(x)$ that is defined by "if x is even, then $F(x) = x/2$; else $F(x) = F(F(3x + 1))$." Prove that $F(x)$ terminates for all integers x. (*Hint:* Consider integers of the form $(2i + 1)2^k - 1$ and use induction.)

12. If S is a set of n elements, the *powerset* of S is the set of all possible subsets of S. For example, if $S = (a, b, c)$, then $powerset(S) = \{(\ ), (a), (b), (c), (a, b), (a, c), (b, c), (a, b, c)\}$. Write a recursive algorithm to compute $powerset(S)$.

1.3 PERFORMANCE ANALYSIS

One goal of this book is to develop skills for making evaluative judgments about algorithms. There are many criteria upon which we can judge an algorithm. For instance:

1. Does it do what we want it to do?
2. Does it work correctly according to the original specifications of the task?
3. Is there documentation that describes how to use it and how it works?

4. Are procedures created in such a way that they perform logical subfunctions?

5. Is the code readable?

These criteria are all vitally important when it comes to writing software, most especially for large systems. Though we do not discuss how to reach these goals, we try to achieve them throughout this book with the pseudocode algorithms we write. Hopefully this more subtle approach will gradually infect your own program-writing habits so that you will automatically strive to achieve these goals.

There are other criteria for judging algorithms that have a more direct relationship to performance. These have to do with their computing time and storage requirements.

Definition 1.2 [Space/Time complexity] The *space complexity* of an algorithm is the amount of memory it needs to run to completion. The *time complexity* of an algorithm is the amount of computer time it needs to run to completion. □

Performance evaluation can be loosely divided into two major phases: (1) a priori estimates and (2) a posteriori testing. We refer to these as *performance analysis* and *performance measurement* respectively.

1.3.1 Space Complexity

Algorithm abc (Algorithm 1.5) computes $a+b+b*c+(a+b-c)/(a+b)+4.0$; Algorithm Sum (Algorithm 1.6) computes $\sum_{i=1}^{n} a[i]$ iteratively, where the $a[i]$'s are real numbers; and RSum (Algorithm 1.7) is a recursive algorithm that computes $\sum_{i=1}^{n} a[i]$.

```
1  Algorithm abc(a, b, c)
2  {
3      return a + b + b * c + (a + b − c)/(a + b) + 4.0;
4  }
```

Algorithm 1.5 Computes $a + b + b * c + (a + b - c)/(a + b) + 4.0$

The space needed by each of these algorithms is seen to be the sum of the following components:

```
1  Algorithm Sum(a, n)
2  {
3      s := 0.0;
4      for i := 1 to n do
5          s := s + a[i];
6      return s;
7  }
```

Algorithm 1.6 Iterative function for sum

```
1  Algorithm RSum(a, n)
2  {
3      if (n ≤ 0) then return 0.0;
4      else return RSum(a, n − 1) + a[n];
5  }
```

Algorithm 1.7 Recursive function for sum

1. A fixed part that is independent of the characteristics (e.g., number, size) of the inputs and outputs. This part typically includes the instruction space (i.e., space for the code), space for simple variables and fixed-size component variables (also called *aggregate*), space for constants, and so on.

2. A variable part that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, the space needed by referenced variables (to the extent that this depends on instance characteristics), and the recursion stack space (insofar as this space depends on the instance characteristics).

The space requirement $S(P)$ of any algorithm P may therefore be written as $S(P) = c + S_P(\text{instance characteristics})$, where c is a constant.

When analyzing the space complexity of an algorithm, we concentrate solely on estimating $S_P(\text{instance characteristics})$. For any given problem, we need first to determine which instance characteristics to use to measure the space requirements. This is very problem specific, and we resort to examples to illustrate the various possibilities. Generally speaking, our choices are limited to quantities related to the number and magnitude of the inputs to and outputs from the algorithm. At times, more complex measures of the interrelationships among the data items are used.

Example 1.4 For Algorithm 1.5, the problem instance is characterized by the specific values of a, b, and c. Making the assumption that one word is adequate to store the values of each of a, b, c, and the result, we see that the space needed by abc is independent of the instance characteristics. Consequently, $S_P(\text{instance characteristics}) = 0$. □

Example 1.5 The problem instances for Algorithm 1.6 are characterized by n, the number of elements to be summed. The space needed by n is one word, since it is of type *integer*. The space needed by a is the space needed by variables of type array of floating point numbers. This is at least n words, since a must be large enough to hold the n elements to be summed. So, we obtain $S_{\mathsf{Sum}}(n) \geq (n+3)$ (n for $a[\,]$, one each for n, i, and s). □

Example 1.6 Let us consider the algorithm RSum (Algorithm 1.7). As in the case of Sum, the instances are characterized by n. The recursion stack space includes space for the formal parameters, the local variables, and the return address. Assume that the return address requires only one word of memory. Each call to RSum requires at least three words (including space for the values of n, the return address, and a pointer to $a[\,]$). Since the depth of recursion is $n+1$, the recursion stack space needed is $\geq 3(n+1)$. □

1.3.2 Time Complexity

The time $T(P)$ taken by a program P is the sum of the compile time and the run (or execution) time. The compile time does not depend on the instance characteristics. Also, we may assume that a compiled program will be run several times without recompilation. Consequently, we concern ourselves with just the run time of a program. This run time is denoted by t_P(instance characteristics).

Because many of the factors t_P depends on are not known at the time a program is conceived, it is reasonable to attempt only to estimate t_P. If we knew the characteristics of the compiler to be used, we could proceed to determine the number of additions, subtractions, multiplications, divisions, compares, loads, stores, and so on, that would be made by the code for P. So, we could obtain an expression for $t_P(n)$ of the form

$$t_P(n) = c_a ADD(n) + c_s SUB(n) + c_m MUL(n) + c_d DIV(n) + \cdots$$

where n denotes the instance characteristics, and c_a, c_s, c_m, c_d, and so on, respectively, denote the time needed for an addition, subtraction, multiplication, division, and so on, and ADD, SUB, MUL, DIV, and so on, are functions whose values are the numbers of additions, subtractions, multiplications, divisions, and so on, that are performed when the code for P is used on an instance with characteristic n.

Obtaining such an exact formula is in itself an impossible task, since the time needed for an addition, subtraction, multiplication, and so on, often depends on the numbers being added, subtracted, multiplied, and so on. The value of $t_P(n)$ for any given n can be obtained only experimentally. The program is typed, compiled, and run on a particular machine. The execution time is physically clocked, and $t_P(n)$ obtained. Even with this experimental approach, one could face difficulties. In a multiuser system, the execution time depends on such factors as system load, the number of other programs running on the computer at the time program P is run, the characteristics of these other programs, and so on.

Given the minimal utility of determining the exact number of additions, subtractions, and so on, that are needed to solve a problem instance with characteristics given by n, we might as well lump all the operations together (provided that the time required by each is relatively independent of the instance characteristics) and obtain a count for the total number of operations. We can go one step further and count only the number of program steps.

A *program step* is loosely defined as a syntactically or semantically meaningful segment of a program that has an execution time that is independent of the instance characteristics. For example, the entire statement

$$\textbf{return } a + b + b * c + (a + b - c)/(a + b) + 4.0;$$

of Algorithm 1.5 could be regarded as a step since its execution time is independent of the instance characteristics (this statement is not strictly true, since the time for a multiply and divide generally depends on the numbers involved in the operation).

The number of steps any program statement is assigned depends on the kind of statement. For example, comments count as zero steps; an assignment statement which does not involve any calls to other algorithms is counted as one step; in an iterative statement such as the **for**, **while**, and **repeat-until** statements, we consider the step counts only for the control part of the statement. The control parts for **for** and **while** statements have the following forms:

for $i :=$ ⟨*expr*⟩ **to** ⟨*expr*1⟩ **do**

while (⟨*expr*⟩) **do**

Each execution of the control part of a **while** statement is given a step count equal to the number of step counts assignable to ⟨*expr*⟩. The step count for each execution of the control part of a **for** statement is one, unless the counts attributable to ⟨*expr*⟩ and ⟨*expr*1⟩ are functions of the instance characteristics. In this latter case, the first execution of the control part of the **for** has a step count equal to the sum of the counts for ⟨*expr*⟩ and ⟨*expr*1⟩ (note that these expressions are computed only when the loop is started). Remaining executions of the **for** statement have a step count of one; and so on.

We can determine the number of steps needed by a program to solve a particular problem instance in one of two ways. In the first method, we introduce a new variable, *count*, into the program. This is a global variable with initial value 0. Statements to increment *count* by the appropriate amount are introduced into the program. This is done so that each time a statement in the original program is executed, *count* is incremented by the step count of that statement.

Example 1.7 When the statements to increment *count* are introduced into Algorithm 1.6, the result is Algorithm 1.8. The change in the value of *count* by the time this program terminates is the number of steps executed by Algorithm 1.6.

Since we are interested in determining only the change in the value of *count*, Algorithm 1.8 may be simplified to Algorithm 1.9. For every initial value of *count*, Algorithms 1.8 and 1.9 compute the same final value for *count*. It is easy to see that in the **for** loop, the value of *count* will increase by a total of $2n$. If *count* is zero to start with, then it will be $2n + 3$ on termination. So each invocation of Sum (Algorithm 1.6) executes a total of $2n + 3$ steps. □

```
1   Algorithm Sum(a, n)
2   {
3       s := 0.0;
4       count := count + 1; // count is global; it is initially zero.
5       for i := 1 to n do
6       {
7           count := count + 1; // For for
8           s := s + a[i]; count := count + 1; // For assignment
9       }
10      count := count + 1; // For last time of for
11      count := count + 1; // For the return
12      return s;
13  }
```

Algorithm 1.8 Algorithm 1.6 with count statements added

```
1   Algorithm Sum(a, n)
2   {
3       for i := 1 to n do count := count + 2;
4       count := count + 3;
5   }
```

Algorithm 1.9 Simplified version of Algorithm 1.8

Example 1.8 When the statements to increment *count* are introduced into Algorithm 1.7, Algorithm 1.10 is obtained. Let $t_{\mathsf{RSum}}(n)$ be the increase in the value of *count* when Algorithm 1.10 terminates. We see that $t_{\mathsf{RSum}}(0) = 2$. When $n > 0$, *count* increases by 2 plus whatever increase results from the invocation of RSum from within the **else** clause. From the definition of t_{RSum}, it follows that this additional increase is $t_{\mathsf{RSum}}(n-1)$. So, if the value of *count* is zero initially, its value at the time of termination is $2+t_{\mathsf{RSum}}(n-1)$, $n > 0$.

```
1   Algorithm RSum(a, n)
2   {
3       count := count + 1; // For the if conditional
4       if (n ≤ 0) then
5       {
6           count := count + 1; // For the return
7           return 0.0;
8       }
9       else
10      {
11          count := count + 1;  // For the addition, function
12                               // invocation and return
13          return RSum(a, n − 1) + a[n];
14      }
15  }
```

Algorithm 1.10 Algorithm 1.7 with count statements added

When analyzing a recursive program for its step count, we often obtain a recursive formula for the step count, for example,

$$t_{\mathsf{RSum}}(n) = \begin{cases} 2 & \text{if } n = 0 \\ 2 + t_{\mathsf{RSum}}(n-1) & \text{if } n > 0 \end{cases}$$

These recursive formulas are referred to as *recurrence relations*. One way of solving any such recurrence relation is to make repeated substitutions for each occurrence of the function t_{RSum} on the right-hand side until all such occurrences disappear:

$$\begin{aligned} t_{\mathsf{RSum}}(n) &= 2 + t_{\mathsf{RSum}}(n-1) \\ &= 2 + 2 + t_{\mathsf{RSum}}(n-2) \\ &= 2(2) + t_{\mathsf{RSum}}(n-2) \\ &\vdots \\ &= n(2) + t_{\mathsf{RSum}}(0) \\ &= 2n + 2, \qquad\qquad n \geq 0 \end{aligned}$$

So the step count for RSum (Algorithm 1.7) is $2n + 2$. □

The step count is useful in that it tells us how the run time for a program changes with changes in the instance characteristics. From the step count for Sum, we see that if n is doubled, the run time also doubles (approximately); if n increases by a factor of 10, the run time increases by a factor of 10; and so on. So, the run time grows *linearly* in n. We say that Sum is a linear time algorithm (the time complexity is linear in the instance characteristic n).

Definition 1.3 [Input size] One of the instance characteristics that is frequently used in the literature is the *input size*. The input size of any instance of a problem is defined to be the number of words (or the number of elements) needed to describe that instance. The input size for the problem of summing an array with n elements is $n + 1$, n for listing the n elements and 1 for the value of n (Algorithms 1.6 and 1.7). The problem tackled in Algorithm 1.5 has an input size of 3. If the input to any problem instance is a single element, the input size is normally taken to be the number of bits needed to specify that element. Run times for many of the algorithms presented in this text are expressed as functions of the corresponding input sizes. □

Example 1.9 [Matrix addition] Algorithm 1.11 is to add two $m \times n$ matrices a and b together. Introducing the *count*-incrementing statements leads to Algorithm 1.12. Algorithm 1.13 is a simplified version of Algorithm 1.12 that computes the same value for *count*. Examining Algorithm 1.13, we see that line 7 is executed n times for each value of i, or a total of mn times; line 5 is executed m times; and line 9 is executed once. If *count* is 0 to begin with, it will be $2mn + 2m + 1$ when Algorithm 1.13 terminates.

From this analysis we see that if $m > n$, then it is better to interchange the two **for** statements in Algorithm 1.11. If this is done, the step count becomes $2mn+2n+1$. Note that in this example the instance characteristics are given by m and n and the input size is $2mn + 2$. □

The second method to determine the step count of an algorithm is to build a table in which we list the total number of steps contributed by each statement. This figure is often arrived at by first determining the number of

```
1   Algorithm Add(a, b, c, m, n)
2   {
3       for i := 1 to m do
4           for j := 1 to n do
5               c[i, j] := a[i, j] + b[i, j];
6   }
```

Algorithm 1.11 Matrix addition

```
1   Algorithm Add(a, b, c, m, n)
2   {
3       for i := 1 to m do
4       {
5           count := count + 1; // For 'for i'
6           for j := 1 to n do
7           {
8               count := count + 1; // For 'for j'
9               c[i, j] := a[i, j] + b[i, j];
10              count := count + 1; // For the assignment
11          }
12          count := count + 1;// For loop initialization and
13                             // last time of 'for j'
14      }
15      count := count + 1;    // For loop initialization and
16                             // last time of 'for i'
17  }
```

Algorithm 1.12 Matrix addition with counting statements

```
1   Algorithm Add(a, b, c, m, n)
2   {
3       for i := 1 to m do
4       {
5           count := count + 2;
6           for j := 1 to n do
7               count := count + 2;
8       }
9       count := count + 1;
10  }
```

Algorithm 1.13 Simplified algorithm with counting only

steps per execution (s/e) of the statement and the total number of times (i.e., frequency) each statement is executed. *The s/e of a statement is the amount by which the count changes as a result of the execution of that statement.* By combining these two quantities, the total contribution of each statement is obtained. By adding the contributions of all statements, the step count for the entire algorithm is obtained.

In Table 1.1, the number of steps per execution and the frequency of each of the statements in Sum (Algorithm 1.6) have been listed. The total number of steps required by the algorithm is determined to be $2n + 3$. It is important to note that the frequency of the **for** statement is $n + 1$ and not n. This is so because i has to be incremented to $n + 1$ before the **for** loop can terminate.

Table 1.2 gives the step count for RSum (Algorithm 1.7). Notice that under the s/e (steps per execution) column, the **else** clause has been given a count of $1 + t_{\text{RSum}}(n - 1)$. This is the total cost of this line each time it is executed. It includes all the steps that get executed as a result of the invocation of RSum from the **else** clause. The frequency and total steps columns have been split into two parts: one for the case $n = 0$ and the other for the case $n > 0$. This is necessary because the frequency (and hence total steps) for some statements is different for each of these cases.

Table 1.3 corresponds to algorithm Add (Algorithm 1.11). Once again, note that the frequency of the first **for** loop is $m + 1$ and not m. This is so as i needs to be incremented up to $m + 1$ before the loop can terminate. Similarly, the frequency for the second **for** loop is $m(n + 1)$.

When you have obtained sufficient experience in computing step counts, you can avoid constructing the frequency table and obtain the step count as in the following example.

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

Table 1.1 Step table for Algorithm 1.6

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

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

Table 1.2 Step table for Algorithm 1.7

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

Table 1.3 Step table for Algorithm 1.11

Example 1.10 [Fibonacci numbers] The Fibonacci sequence of numbers starts as

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

Each new term is obtained by taking the sum of the two previous terms. If we call the first term of the sequence f_0, then $f_0 = 0$, $f_1 = 1$, and in general

$$f_n = f_{n-1} + f_{n-2}, \quad n \geq 2$$

Fibonacci (Algorithm 1.14) takes as input any nonnegative integer n and prints the value f_n.

To analyze the time complexity of this algorithm, we need to consider the two cases (1) $n = 0$ or 1 and (2) $n > 1$. When $n = 0$ or 1, lines 4 and 5 get executed once each. Since each line has an s/e of 1, the total step count for this case is 2. When $n > 1$, lines 4, 8, and 14 are each executed once. Line 9 gets executed n times, and lines 11 and 12 get executed $n - 1$ times each (note that the last time line 9 is executed, i is incremented to $n+1$, and the loop exited). Line 8 has an s/e of 2, line 12 has an s/e of 2, and line 13 has an s/e of 0. The remaining lines that get executed have s/e's of 1. The total steps for the case $n > 1$ is therefore $4n + 1$. □

Summary of Time Complexity

The time complexity of an algorithm is given by the number of steps taken by the algorithm to compute the function it was written for. The number of steps is itself a function of the instance characteristics. Although any specific instance may have several characteristics (e.g., the number of inputs, the number of outputs, the magnitudes of the inputs and outputs), the number

```
1   Algorithm Fibonacci(n)
2   // Compute the nth Fibonacci number.
3   {
4       if (n ≤ 1) then
5           write (n);
6       else
7       {
8           fnm2 := 0; fnm1 := 1;
9           for i := 2 to n do
10          {
11              fn := fnm1 + fnm2;
12              fnm2 := fnm1; fnm1 := fn;
13          }
14          write (fn);
15      }
16  }
```

Algorithm 1.14 Fibonacci numbers

of steps is computed as a function of some subset of these. Usually, we choose those characteristics that are of importance to us. For example, we might wish to know how the computing (or run) time (i.e., time complexity) increases as the number of inputs increase. In this case the number of steps will be computed as a function of the number of inputs alone. For a different algorithm, we might be interested in determining how the computing time increases as the magnitude of one of the inputs increases. In this case the number of steps will be computed as a function of the magnitude of this input alone. Thus, before the step count of an algorithm can be determined, we need to know exactly which characteristics of the problem instance are to be used. These define the variables in the expression for the step count. In the case of Sum, we chose to measure the time complexity as a function of the number n of elements being added. For algorithm Add, the choice of characteristics was the number m of rows and the number n of columns in the matrices being added.

Once the relevant characteristics $(n, m, p, q, r, \ldots)$ have been selected, we can define what a step is. A *step* is any computation unit that is independent of the characteristics $(n, m, p, q, r, \ldots)$. Thus, 10 additions can be one step; 100 multiplications can also be one step; but n additions cannot. Nor can $m/2$ additions, $p + q$ subtractions, and so on, be counted as one step.

A systematic way to assign step counts was also discussed. Once this has been done, the time complexity (i.e., the total step count) of an algorithm can be obtained using either of the two methods discussed.

The examples we have looked at so far were sufficiently simple that the time complexities were nice functions of fairly simple characteristics like the number of inputs and the number of rows and columns. For many algorithms, the time complexity is not dependent solely on the number of inputs or outputs or some other easily specified characteristic. For example, the searching algorithm you wrote for Exercise 4 in Section 1.2, may terminate in one step if x is the first element examined by your algorithm, or it may take two steps (this happens if x is the second element examined), and so on. In other words, knowing n alone is not enough to estimate the run time of your algorithm.

We can extricate ourselves from the difficulties resulting from situations when the chosen parameters are not adequate to determine the step count uniquely by defining three kinds of step counts: best case, worst case, and average. The *best-case step count* is the minimum number of steps that can be executed for the given parameters. The *worst-case step count* is the maximum number of steps that can be executed for the given parameters. The *average step count* is the average number of steps executed on instances with the given parameters.

Our motivation to determine step counts is to be able to compare the time complexities of two algorithms that compute the same function and also to predict the growth in run time as the instance characteristics change.

Determining the exact step count (best case, worst case, or average) of an algorithm can prove to be an exceedingly difficult task. Expending immense effort to determine the step count exactly is not a very worthwhile endeavor, since the notion of a step is itself inexact. (Both the instructions $x := y$; and $x := y + z + (x/y) + (x * y * z - x/z)$; count as one step.) Because of the inexactness of what a step stands for, the exact step count is not very useful for comparative purposes. An exception to this is when the difference between the step counts of two algorithms is very large, as in $3n + 3$ versus $100n + 10$. We might feel quite safe in predicting that the algorithm with step count $3n+3$ will run in less time than the one with step count $100n+10$. But even in this case, it is not necessary to know that the exact step count is $100n + 10$. Something like, "it's about $80n$ or $85n$ or $75n$," is adequate to arrive at the same conclusion.

For most situations, it is adequate to be able to make a statement like $c_1n^2 \leq t_P(n) \leq c_2n^2$ or $t_Q(n,m) = c_1n + c_2m$, where c_1 and c_2 are nonnegative constants. This is so because if we have two algorithms with a complexity of $c_1n^2 + c_2n$ and c_3n respectively, then we know that the one with complexity c_3n will be faster than the one with complexity $c_1n^2 + c_2n$ for sufficiently large values of n. For small values of n, either algorithm could be faster (depending on c_1, c_2, and c_3). If $c_1 = 1$, $c_2 = 2$, and $c_3 = 100$, then

$c_1n^2 + c_2n \leq c_3n$ 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