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

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

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:

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

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 $n = 100{,}000$.

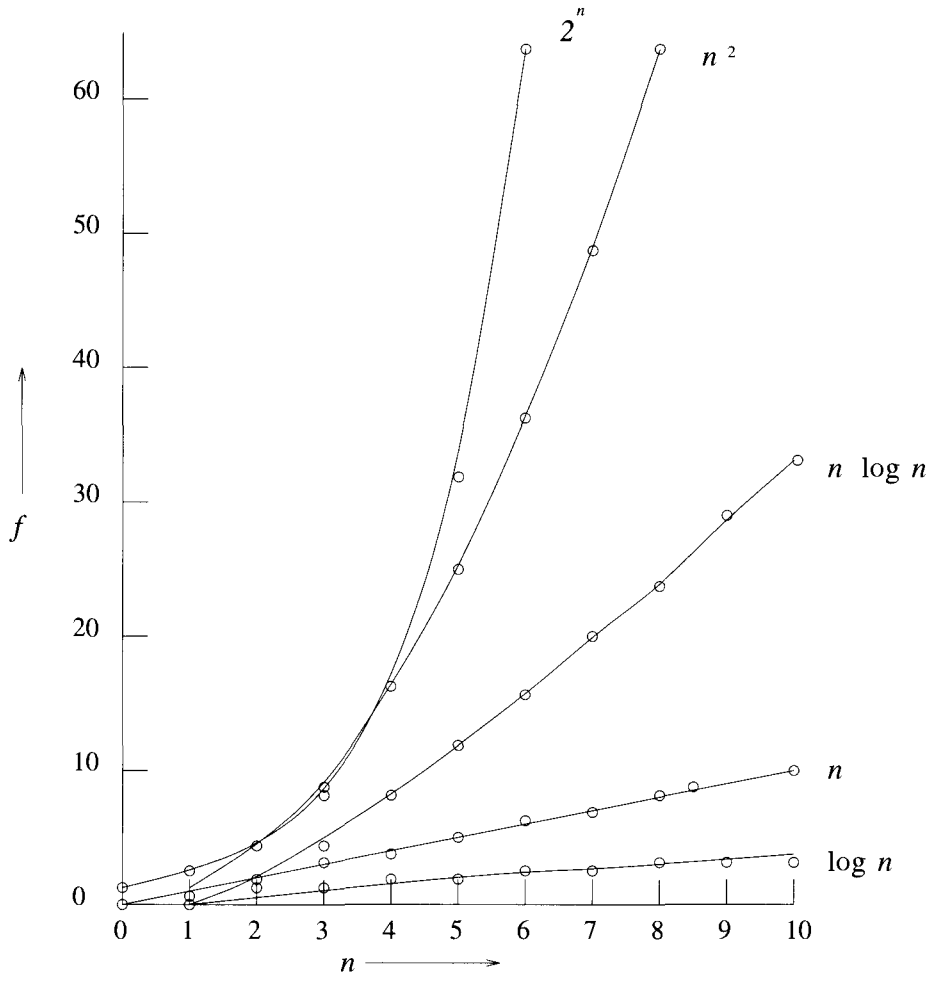

Figure 1.3 Plot of function values

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.

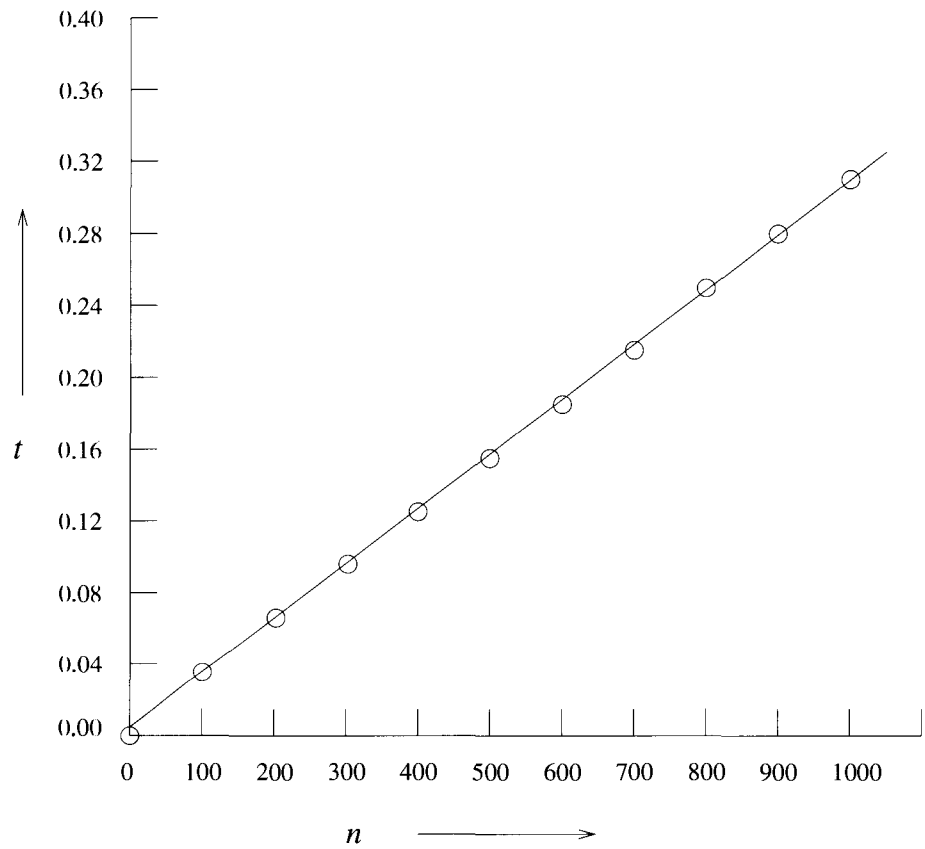

Figure 1.4 Plot of the data in Table 1.10

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.

```
1   Algorithm D(x, n)
2   {
3       i := 1;
4       repeat
5       {
6           x[i] := x[i] + 2; i := i + 2;
7       } until (i > n);
8       i := 1;
9       while (i ≤ ⌊n/2⌋) do
10      {
11          x[i] := x[i] + x[i + 1]; i := i + 1;
12      }
13  }
```

Algorithm 1.23 Example algorithm

```
1   Algorithm Transpose(a, n)
2   {
3       for i := 1 to n − 1 do
4           for j := i + 1 to n do
5           {
6               t := a[i, j]; a[i, j] := a[j, i]; a[j, i] := t;
7           }
8   }
```

Algorithm 1.24 Matrix transpose

```
1   Algorithm Mult(a, b, c, n)
2   {
3       for i := 1 to n do
4           for j := 1 to n do
5           {
6               c[i, j] := 0;
7               for k := 1 to n do
8                   c[i, j] := c[i, j] + a[i, k] * b[k, j];
9           }
10  }
```

Algorithm 1.25 Matrix multiplication

7. (a) Do Exercise 4 for Algorithm 1.26. This algorithm multiplies two matrices a and b, where a is an $m \times n$ matrix and b is an $n \times p$ matrix.

```
1   Algorithm Mult(a, b, c, m, n, p)
2   {
3       for i := 1 to m do
4           for j := 1 to p do
5           {
6               c[i, j] := 0;
7               for k := 1 to n do
8                   c[i, j] := c[i, j] + a[i, k] * b[k, j];
9           }
10  }
```

Algorithm 1.26 Matrix multiplication

(b) Under what conditions is it profitable to interchange the two outermost **for** loops?

8. Show that the following equalities are correct:

(a) $5n^2 - 6n = \Theta(n^2)$

(b) $n! = O(n^n)$

(c) $2n^2 2^n + n \log n = \Theta(n^2 2^n)$

(d) $\sum_{i=0}^{n} i^2 = \Theta(n^3)$

(e) $\sum_{i=0}^{n} i^3 = \Theta(n^4)$.

(f) $n^{2^n} + 6 * 2^n = \Theta(n^{2^n})$

(g) $n^3 + 10^6 n^2 = \Theta(n^3)$

(h) $6n^3/(\log n + 1) = O(n^3)$

(i) $n^{1.001} + n \log n = \Theta(n^{1.001})$

(j) $n^{k+\epsilon} + n^k \log n = \Theta(n^{k+\epsilon})$ for all fixed k and ϵ, $k \geq 0$ and $\epsilon > 0$

(k) $10n^3 + 15n^4 + 100n^2 2^n = O(100n^2 2^n)$

(l) $33n^3 + 4n^2 = \Omega(n^2)$

(m) $33n^3 + 4n^2 = \Omega(n^3)$

9. Show that the following equalities are incorrect:

   (a) $10n^2 + 9 = O(n)$

   (b) $n^2 \log n = \Theta(n^2)$

   (c) $n^2/\log n = \Theta(n^2)$

   (d) $n^3 2^n + 6n^2 3^n = O(n^3 2^n)$

10. Prove Theorems 1.3 and 1.4.

11. Analyze the computing time of SelectionSort (Algorithm 1.2).

12. Obtain worst-case run times for SelectionSort (Algorithm 1.2). Do this for suitable values of n in the range [0, 100]. Your report must include a plan for the experiment as well as the measured times. These times are to be provided both in a table and as a graph.

13. Consider the algorithm Add (Algorithm 1.11).

    (a) Obtain run times for $n = 1, 10, 20, \ldots, 100$.

    (b) Plot the times obtained in part (a).

14. Do the previous exercise for matrix multiplication (Algorithm 1.26).

15. A complex-valued matrix X is represented by a pair of matrices (A, B), where A and B contain real values. Write an algorithm that computes the product of two complex-valued matrices (A, B) and (C, D), where $(A, B) * (C, D) = (A + iB) * (C + iD) = (AC - BD) + i(AD + BC)$. Determine the number of additions and multiplications if the matrices are all $n \times n$.