2 Complexity Analysis

2.1 Computational and Asymptotic Complexity

The same problem can frequently be solved with algorithms that differ in efficiency. The differences between the algorithms may be immaterial for processing a small number of data items, but these differences grow with the amount of data. To compare the efficiency of algorithms, a measure of the degree of difficulty of an algorithm called *computational complexity* was developed by Juris Hartmanis and Richard E. Stearns.

Computational complexity indicates how much effort is needed to apply an algorithm or how costly it is. This cost can be measured in a variety of ways, and the particular context determines its meaning. This book concerns itself with the two efficiency criteria: time and space. The factor of time is usually more important than that of space, so efficiency considerations usually focus on the amount of time elapsed when processing data. However, the most inefficient algorithm run on a Cray computer can execute much faster than the most efficient algorithm run on a PC, so run time is always system-dependent. For example, to compare 100 algorithms, all of them would have to be run on the same machine. Furthermore, the results of run-time tests depend on the language in which a given algorithm is written, even if the tests are performed on the same machine. If programs are compiled, they execute much faster than when they are interpreted. A program written in C or Ada may be 20 times faster than the same program encoded in BASIC or LISP.

To evaluate an algorithm's efficiency, real-time units such as microseconds and nanoseconds should not be used. Rather, logical units that express a relationship between the size n of a file or an array and the amount of time t required to process the data should be used. If there is a linear relationship between the size n and time t—that is, $t_1 = cn_1$—then an increase of data by a factor of 5 results in the increase of the execution time by the same factor; if $n_2 = 5n_1$, then $t_2 = 5t_1$. Similarly, if $t_1 = \log_2 n$, then doubling n increases t by only one unit of time. Therefore, if $t_2 = \log_2(2n)$, then $t_2 = t_1 + 1$.

A function expressing the relationship between n and t is usually much more complex, and calculating such a function is important only in regard to large bodies of data; any terms that do not substantially change the function's magnitude should be eliminated from the function. The resulting function gives only an approximate measure of efficiency of the original function. However, this approximation is sufficiently close to the original, especially for a function that processes large quantities of data. This measure of efficiency is called *asymptotic complexity,* and is used when disregarding certain terms of a function to express the efficiency of an algorithm or when calculating a function is difficult or impossible and only approximations can be found. To illustrate the first case, consider the following example:

$$f(n) = n^2 + 100n + \log_{10}n + 1{,}000 \tag{2.1}$$

For small values of n, the last term, 1,000, is the largest. When n equals 10, the second ($100n$) and last (1,000) terms are on equal footing with the other terms, making the same contribution to the function value. When n reaches the value of 100, the first and the second terms make the same contribution to the result. But when n becomes larger than 100, the contribution of the second term becomes less significant. Hence, for large values of n, due to the quadratic growth of the first term (n^2), the value of the function f depends mainly on the value of this first term, as Figure 2.1 demonstrates. Other terms can be disregarded for large n.

FIGURE 2.1 The growth rate of all terms of function $f(n) = n^2 + 100n + \log_{10}n + 1{,}000$.

n	$f(n)$	n^2		$100n$		$\log_{10}n$		1,000	
	Value	Value	%	Value	%	Value	%	Value	%
1	1,101	1	0.1	100	9.1	0	0.0	1,000	90.83
10	2,101	100	4.76	1,000	47.6	1	0.05	1,000	47.60
100	21,002	10,000	47.6	10,000	47.6	2	0.001	1,000	4.76
1,000	1,101,003	1,000,000	90.8	100,000	9.1	3	0.0003	1,000	0.09
10,000	101,001,004	100,000,000	99.0	1,000,000	0.99	4	0.0	1,000	0.001
100,000	10,010,001,005	10,000,000,000	99.9	10,000,000	0.099	5	0.0	1,000	0.00

2.2 Big-O Notation

The most commonly used notation for specifying asymptotic complexity—that is, for estimating the rate of function growth—is the big-O notation introduced in 1894 by Paul Bachmann. Given two positive-valued functions f and g, consider the following definition:

Definition 1: $f(n)$ is $O(g(n))$ if there exist positive numbers c and N such that $f(n) \leq cg(n)$ for all $n \geq N$.

This definition reads: f is big-O of g if there is a positive number c such that f is not larger than cg for sufficiently large ns; that is, for all ns larger than some number

N. The relationship between f and g can be expressed by stating either that $g(n)$ is an upper bound on the value of $f(n)$ or that, in the long run, f grows at most as fast as g.

The problem with this definition is that, first, it states only that there must exist certain c and N, but it does not give any hint of how to calculate these constants. Second, it does not put any restrictions on these values and gives little guidance in situations when there are many candidates. In fact, there are usually infinitely many pairs of cs and Ns that can be given for the same pair of functions f and g. For example, for

$$f(n) = 2n^2 + 3n + 1 = O(n^2) \tag{2.2}$$

where $g(n) = n^2$, candidate values for c and N are shown in Figure 2.2.

FIGURE 2.2 Different values of c and N for function $f(n) = 2n^2 + 3n + 1 = O(n^2)$ calculated according to the definition of big-O.

c	≥ 6	$\geq 3\frac{3}{4}$	$\geq 3\frac{1}{9}$	$\geq 2\frac{13}{16}$	$\geq 2\frac{16}{25}$	...	$\rightarrow$	2
N	1	2	3	4	5	...	$\rightarrow$	∞

We obtain these values by solving the inequality:

$$2n^2 + 3n + 1 \leq cn^2$$

or equivalently

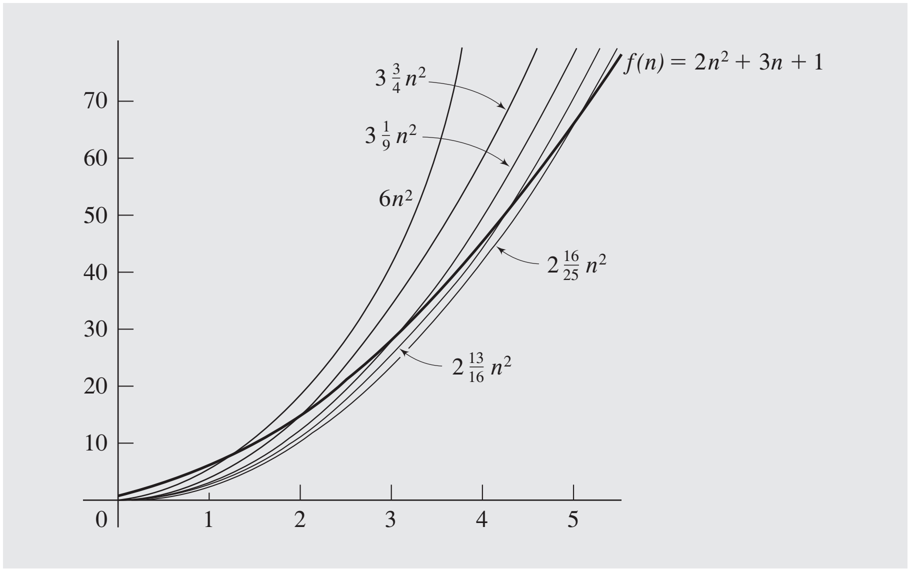
$$2 + \frac{3}{n} + \frac{1}{n^2} \leq c$$

for different ns. The first inequality results in substituting the quadratic function from Equation 2.2 for $f(n)$ in the definition of the big-O notation and n^2 for $g(n)$. Because it is one inequality with two unknowns, different pairs of constants c and N for the same function $g(= n^2)$ can be determined. To choose the best c and N, it should be determined for which N a certain term in f becomes the largest and stays the largest. In Equation 2.2, the only candidates for the largest term are $2n^2$ and $3n$; these terms can be compared using the inequality $2n^2 > 3n$ that holds for $n > 1$. Thus, $N = 2$ and $c \geq 3\frac{3}{4}$, as Figure 2.2 indicates.

What is the practical significance of the pairs of constants just listed? All of them are related to the same function $g(n) = n^2$ and to the same $f(n)$. For a fixed g, an infinite number of pairs of cs and Ns can be identified. The point is that f and g grow at the same rate. The definition states, however, that g is almost always greater than or equal to f if it is multiplied by a constant c. "Almost always" means for all ns not less than a constant N. The crux of the matter is that the value of c depends on which N is chosen, and vice versa. For example, if 1 is chosen as the value of N—that is, if g is multiplied by c so that $cg(n)$ will not be less than f right away—then c has to be equal to 6 or greater. If $cg(n)$is greater than or equal to $f(n)$ starting from $n = 2$, then it is enough that c is equal to 3.75. The constant c has to be at least $3\frac{1}{9}$ if $cg(n)$ is not less than $f(n)$ starting from

$n = 3$. Figure 2.3 shows the graphs of the functions f and g. The function g is plotted with different coefficients c. Also, N is always a point where the functions $cg(n)$ and f intersect each other.

FIGURE 2.3 Comparison of functions for different values of c and N from Figure 2.2.



The inherent imprecision of the big-O notation goes even further, because there can be infinitely many functions g for a given function f. For example, the f from Equation 2.2 is big-O not only of n^2, but also of $n^3, n^4, \ldots, n^k, \ldots$ for any $k \geq 2$. To avoid this embarrassment of riches, the smallest function g is chosen, n^2 in this case.

The approximation of function f can be refined using big-O notation only for the part of the equation suppressing irrelevant information. For example, in Equation 2.1, the contribution of the third and last terms to the value of the function can be omitted (see Equation 2.3).

$$f(n) = n^2 + 100n + O(\log_{10} n) \tag{2.3}$$

Similarly, the function f in Equation 2.2 can be approximated as

$$f(n) = 2n^2 + O(n) \tag{2.4}$$

2.3 Properties of Big-O Notation

Big-O notation has some helpful properties that can be used when estimating the efficiency of algorithms.

Fact 1. (transitivity) If $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$, then $f(n)$ if $O(h(n))$. (This can be rephrased as $O(O(g(n)))$ is $O(g(n))$.)

Proof: According to the definition, $f(n)$ is $O(g(n))$ if there exist positive numbers c_1 and N_1 such that $f(n) \le c_1 g(n)$ for all $n \ge N_1$, and $g(n)$ is $O(h(n))$ if there exist positive numbers c_2 and N_2 such that $g(n) \le c_2 h(n)$ for all $n \ge N_2$. Hence, $c_1 g(n) \le c_1 c_2 h(n)$ for $n \ge N$ where N is the larger of N_1 and N_2. If we take $c = c_1 c_2$, then $f(n) \le ch(n)$ for $n \ge N$, which means that f is $O(h(n))$.

Fact 2. If $f(n)$ is $O(h(n))$ and $g(n)$ is $O(h(n))$, then $f(n) + g(n)$ is $O(h(n))$.

Proof: After setting c equal to $c_1 + c_2$, $f(n) + g(n) \le ch(n)$.

Fact 3. The function an^k is $O(n^k)$.

Proof: For the inequality $an^k \le cn^k$ to hold, $c \ge a$ is necessary.

Fact 4. The function n^k is $O(n^{k+j})$ for any positive j.

Proof: The statement holds if $c = N = 1$.

It follows from all these facts that every polynomial is big-O of n raised to the largest power, or

$$f(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 \text{ is } O(n^k)$$

It is also obvious that in the case of polynomials, $f(n)$ is $O(n^{k+j})$ for any positive j.

One of the most important functions in the evaluation of the efficiency of algorithms is the logarithmic function. In fact, if it can be stated that the complexity of an algorithm is on the order of the logarithmic function, the algorithm can be regarded as very good. There are an infinite number of functions that can be considered better than the logarithmic function, among which only a few, such as $O(\lg \lg n)$ or $O(1)$, have practical bearing. Before we show an important fact about logarithmic functions, let us state without proof:

Fact 5. If $f(n) = cg(n)$, then $f(n)$ is $O(g(n))$.

Fact 6. The function $\log_a n$ is $O(\log_b n)$ for any positive numbers a and $b \ne 1$.

This correspondence holds between logarithmic functions. Fact 6 states that regardless of their bases, logarithmic functions are big-O of each other; that is, all these functions have the same rate of growth.

Proof: Letting $\log_a n = x$ and $\log_b n = y$, we have, by the definition of logarithm, $a^x = n$ and $b^y = n$.

Taking ln of both sides results in

$$x \ln a = \ln n \quad\text{and}\quad y \ln b = \ln n$$

Thus

$$x \ln a = y \ln b,$$

$$\ln a \log_a n = \ln b \log_b n,$$

$$\log_a n = \frac{\ln b}{\ln a} \log_b n = c \log_b n$$

which proves that $\log_a n$ and $\log_b n$ are multiples of each other. By Fact 5, $\log_a n$ is $O(\log_b n)$.

Because the base of the logarithm is irrelevant in the context of big-O notation, we can always use just one base and Fact 6 can be written as

Fact 7. $\log_a n$ is $O(\lg n)$ for any positive $a \neq 1$, where $\lg n = \log_2 n$.

2.4 Ω and Θ Notations

Big-O notation refers to the upper bounds of functions. There is a symmetrical definition for a lower bound in the definition of big-Ω:

Definition 2: The function $f(n)$ is $\Omega(g(n))$ if there exist positive numbers c and N such that $f(n) \geq cg(n)$ for all $n \geq N$.

This definition reads: f is Ω (big-omega) of g if there is a positive number c such that f is at least equal to cg for almost all ns. In other words, $cg(n)$ is a lower bound on the size of $f(n)$, or, in the long run, f grows at least at the rate of g.

The only difference between this definition and the definition of big-O notation is the direction of the inequality; one definition can be turned into the other by replacing "≥" with "≤." There is an interconnection between these two notations expressed by the equivalence

$$f(n) \text{ is } \Omega(g(n)) \text{ iff } g(n) \text{ is } O(f(n))$$

Ω notation suffers from the same profusion problem as does big-O notation: There is an unlimited number of choices for the constants c and N. For Equation 2.2, we are looking for such a c, for which $2n^2 + 3n + 1 \geq cn^2$, which is true for any $n \geq 0$, if $c \leq 2$, where 2 is the limit for c in Figure 2.2. Also, if f is an Ω of g and $h \leq g$, then f is an Ω of h; that is, if for f we can find one g such that f is an Ω of g, then we can find infinitely many. For example, the function 2.2 is an Ω of n^2 but also of n, $n^{1/2}$, $n^{1/3}$, $n^{1/4}$, . . . , and also of $\lg n$, $\lg \lg n$, . . . , and of many other functions. For practical purposes, only the closest Ωs are the most interesting, (i.e., the largest lower bounds). This restriction is made implicitly each time we choose an Ω of a function f.

There are an infinite number of possible lower bounds for the function f; that is, there is an infinite set of gs such that $f(n)$ is $\Omega(g(n))$ as well as an unbounded number of possible upper bounds of f. This may be somewhat disquieting, so we restrict our attention to the smallest upper bounds and the largest lower bounds. Note that there is a common ground for big-O and Ω notations indicated by the equalities in the definitions of these notations: Big-O is defined in terms of "≤" and Ω in terms of "≥"; "=" is included in both inequalities. This suggests a way of restricting the sets of possible lower and upper bounds. This restriction can be accomplished by the following definition of Θ (theta) notation:

Definition 3: $f(n)$ is $\Theta(g(n))$ if there exist positive numbers c_1, c_2, and N such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq N$.

This definition reads: f has an order of magnitude g, f is on the order of g, or both functions grow at the same rate in the long run. We see that $f(n)$ is $\Theta(g(n))$ if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.

The only function just listed that is both big-O and Ω of the function 2.2 is n^2. However, it is not the only choice, and there are still an infinite number of choices, because the functions $2n^2, 3n^2, 4n^2, \ldots$ are also Θ of function 2.2. But it is rather obvious that the simplest, n^2, will be chosen.

When applying any of these notations (big-O, Ω, and Θ), do not forget that they are approximations that hide some detail that in many cases may be considered important.

2.5 Possible Problems

All the notations serve the purpose of comparing the efficiency of various algorithms designed for solving the same problem. However, if only big-Os are used to represent the efficiency of algorithms, then some of them may be rejected prematurely. The problem is that in the definition of big-O notation, f is considered $O(g(n))$ if the inequality $f(n) \le cg(n)$ holds in the long run for all natural numbers with a few exceptions. The number of ns violating this inequality is always finite. It is enough to meet the condition of the definition. As Figure 2.2 indicates, this number of exceptions can be reduced by choosing a sufficiently large c. However, this may be of little practical significance if the constant c in $f(n) \le cg(n)$ is prohibitively large, say 10^8, although the function g taken by itself seems to be promising.

Consider that there are two algorithms to solve a certain problem and suppose that the number of operations required by these algorithms is $10^8 n$ and $10n^2$. The first function is $O(n)$ and the second is $O(n^2)$. Using just the big-O information, the second algorithm is rejected because the number of steps grows too fast. It is true but, again, in the long run, because for $n \le 10^7$, which is 10 million, the second algorithm performs fewer operations than the first. Although 10 million is not an unheard-of number of elements to be processed by an algorithm, in many cases the number is much lower, and in these cases the second algorithm is preferable.

For these reasons, it may be desirable to use one more notation that includes constants which are very large for practical reasons. Udi Manber proposes a double-O (OO) notation to indicate such functions: f is $OO(g(n))$ if it is $O(g(n))$ and the constant c is too large to have practical significance. Thus, $10^8 n$ is $OO(n)$. However, the definition of "too large" depends on the particular application.

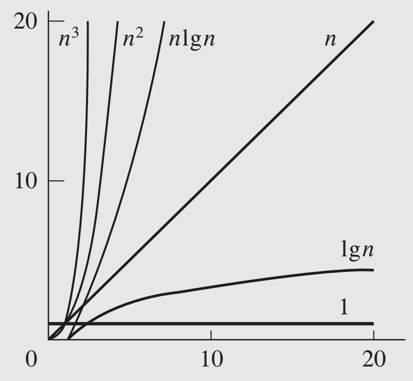
2.6 Examples of Complexities

Algorithms can be classified by their time or space complexities, and in this respect, several classes of such algorithms can be distinguished, as Figure 2.4 illustrates. Their growth is also displayed in Figure 2.5. For example, an algorithm is called *constant* if its execution time remains the same for any number of elements; it is called *quadratic* if its execution time is $O(n^2)$. For each of these classes, a number of operations is

FIGURE 2.4 Classes of algorithms and their execution times on a computer executing 1 million operations per second (1 sec = 10^6 µsec = 10^3 msec).

Class	**Complexity**	**Number of Operations and Execution Time (1 instr/µsec)**					
n		**10**		**10^2**		**10^3**	
constant	$O(1)$	1	1 µsec	1	1 µsec	1	1 µsec
logarithimic	$O(\lg n)$	3.32	3 µsec	6.64	7 µsec	9.97	10 µsec
linear	$O(n)$	10	10 µsec	10^2	100 µsec	10^3	1 msec
O(n lg n)	$O(n \lg n)$	33.2	33 µsec	664	664 µsec	9970	10 msec
quadratic	$O(n^2)$	10^2	100 µsec	10^4	10 msec	10^6	1 sec
cubic	$O(n^3)$	10^3	1 msec	10^6	1 sec	10^9	16.7 min
exponential	$O(2^n)$	1024	10 msec	10^{30}	$3.17 * 10^{17}$ yrs	10^{301}	
n		**10^4**		**10^5**		**10^6**	
constant	$O(1)$	1	1 µsec	1	1 µsec	1	1 µsec
logarithmic	$O(\lg n)$	13.3	13 µsec	16.6	7 µsec	19.93	20 µsec
linear	$O(n)$	10^4	10 msec	10^5	0.1 sec	10^6	1 sec
$O(n \lg n)$	$O(n \lg n)$	133×10^3	133 msec	166×10^4	1.6 sec	199.3×10^5	20 sec
quadratic	$O(n^2)$	10^8	1.7 min	10^{10}	16.7 min	10^{12}	11.6 days
cubic	$O(n^3)$	10^{12}	11.6 days	10^{15}	31.7 yr	10^{18}	31,709 yr
exponential	$O(2^n)$	10^{3010}		10^{30103}		10^{301030}	

FIGURE 2.5 Typical functions applied in big-O estimates.



shown along with the real time needed for executing them on a machine able to perform 1 million operations per second, or one operation per microsecond (μsec). The table in Figure 2.4 indicates that some ill-designed algorithms, or algorithms whose complexity cannot be improved, have no practical application on available computers. To process 1 million items with a quadratic algorithm, over 11 days are needed, and for a cubic algorithm, thousands of years. Even if a computer can perform one operation per nanosecond (1 billion operations per second), the quadratic algorithm finishes in only 16.7 seconds, but the cubic algorithm requires over 31 years. Even a 1,000-fold improvement in execution speed has very little practical bearing for this algorithm. Analyzing the complexity of algorithms is of extreme importance and cannot be abandoned on account of the argument that we have entered an era when, at relatively little cost, a computer on our desktop can execute millions of operations per second. The importance of analyzing the complexity of algorithms, in any context but in the context of data structures in particular, cannot be overstressed. The impressive speed of computers is of limited use if the programs that run on them use inefficient algorithms.

2.7 Finding Asymptotic Complexity: Examples

Asymptotic bounds are used to estimate the efficiency of algorithms by assessing the amount of time and memory needed to accomplish the task for which the algorithms were designed. This section illustrates how this complexity can be determined.

In most cases, we are interested in time complexity, which usually measures the number of assignments and comparisons performed during the execution of a program. Chapter 9, which deals with sorting algorithms, considers both types of operations; this chapter considers only the number of assignment statements.

Begin with a simple loop to calculate the sum of numbers in an array:

```
for (i = sum = 0; i < n; i++)
    sum += a[i];
```

First, two variables are initialized, then the `for` loop iterates n times, and during each iteration, it executes two assignments, one of which updates `sum` and the other of which updates `i`. Thus, there are $2 + 2n$ assignments for the complete run of this `for` loop; its asymptotic complexity is $O(n)$.

Complexity usually grows if nested loops are used, as in the following code, which outputs the sums of all the subarrays that begin with position 0:

```
for (i = 0; i < n; i++) {
    for (j = 1, sum = a[0]; j <= i; j++)
        sum += a[j];
    System.out.println ("sum for subarray 0 through "+i+" is" + sum);
}
```

Before the loops start, `i` is initialized. The outer loop is performed n times, executing in each iteration an inner `for` loop, print statement, and assignment statements for `i`, `j`, and `sum`. The inner loop is executed i times for each $i \in \{1, . . . , n - 1\}$

with two assignments in each iteration: one for `sum` and one for `j`. Therefore, there are $1 + 3n + \sum_{i=1}^{n-1} 2i = 1 + 3n + 2(1 + 2 + \cdots + n - 1) = 1 + 3n + n(n - 1) = O(n) + O(n^2) = O(n^2)$ assignments executed before the program is completed.

Algorithms with nested loops usually have a larger complexity than algorithms with one loop, but it does not have to grow at all. For example, we may request printing sums of numbers in the last five cells of the subarrays starting in position 0. We adopt the foregoing code and transform it to

```
for (i = 4; i < n; i++) {
    for (j = i-3, sum = a[i-4]; j <= i; j++)
        sum += a[j];
    System.out.println ("sum for subarray "+(i - 4)+" through "+i+" is"+ sum);
}
```

The outer loop is executed $n - 4$ times. For each i, the inner loop is executed only four times: For each iteration of the outer loop, there are eight assignments in the inner loop, and this number does not depend on the size of the array. With initialization of `i`, $n - 4$ autoincrements of `i`, and $n - 4$ initializations of `j` and `sum`, the program makes $1 + 8 \cdot (n - 4) = O(n)$ assignments.

Analysis of these two examples is relatively uncomplicated because the number of times the loops executed did not depend on the ordering of the arrays. Computation of asymptotic complexity is more involved if the number of iterations is not always the same. This point can be illustrated with a loop used to determine the length of the longest subarray with the numbers in increasing order. For example, in [1 8 1 2 5 0 11 12], it is three, the length of subarray [1 2 5]. The code is

```
for (i = 0, length = 1; i < n-1; i++) {
    for (i1 = i2 = k = i; k < n-1 && a[k] < a[k+1]; k++, i2++);
    if (length < i2 - i1 + 1)
      length = i2 - i1 + 1;
    System.out.println ("the length of the longest ordered subarray is" + length);
}
```

Notice that if all numbers in the array are in decreasing order, the outer loop is executed $n - 1$ times, but in each iteration, the inner loop executes just one time. Thus, the algorithm is $O(n)$. The algorithm is least efficient if the numbers are in increasing order. In this case, the outer `for` loop is executed $n - 1$ times, and the inner loop is executed $n - 1 - i$ times for each $i \in \{0, \ldots, n - 2\}$. Thus, the algorithm is $O(n^2)$. In most cases, the arrangement of data is less orderly, and measuring the efficiency in these cases is of great importance. However, it is far from trivial to determine the efficiency in the average cases.

A last example used to determine the computational complexity is the *binary search algorithm,* which is used to locate an element in an ordered array. If it is an array of numbers and we try to locate number k, then the algorithm accesses the middle element of the array first. If that element is equal to k, then the algorithm returns its position; if not, the algorithm continues. In the second trial, only half of the original array is considered: the first half if k is smaller than the middle element, and the second otherwise. Now, the middle element of the chosen subarray is accessed and compared to k. If it is the same, the algorithm completes successfully. Otherwise, the subarray is divided into two halves, and if k is larger than this middle element, the first half is discarded; otherwise, the first half is retained. This process of halving and comparing continues until k is found or the array can no longer be divided into two subarrays. This relatively simple algorithm can be coded as follows:

```
int binarySearch(int[] arr, int key) {
    int lo = 0, mid, hi = arr.length-1;
    while (lo <= hi) {
          mid = (lo + hi)/2;
          if (key < arr[mid])
             hi = mid - 1;
          else if (arr[mid] < key)
             lo = mid + 1;
          else return mid;  // success: return the index of
    }                       //   the cell occupied by key;
    return -1;              // failure: key is not in the array;
}
```

If `key` is in the middle of the array, the loop executes only one time. How many times does the loop execute in the case where `key` is not in the array? First the algorithm looks at the entire array of size n, then at one of its halves of size $\frac{n}{2}$, then at one of the halves of this half, of size $\frac{n}{2^2}$, and so on, until the array is of size 1. Hence, we have the sequence $n, \frac{n}{2}, \frac{n}{2^2}, \ldots, \frac{n}{2^m}$, and we want to know the value of m. But the last term of this sequence $\frac{n}{2^m}$ equals 1, from which we have $m = \lg n$. So the fact that k is not in the array can be determined after $\lg n$ iterations of the loop.

2.8 The Best, Average, and Worst Cases

The last two examples in the preceding section indicate the need for distinguishing at least three cases for which the efficiency of algorithms has to be determined. The *worst case* is when an algorithm requires a maximum number of steps, and the *best case* is when the number of steps is the smallest. The *average case* falls between these extremes. In simple cases, the average complexity is established by considering possible inputs to an algorithm, determining the number of steps performed by the algorithm for each input, adding the number of steps for all the inputs, and dividing by the number of inputs. This definition, however, assumes that the probability of occurrence of each input is the same, which is not always the case. To consider the probability explicitly, the average complexity is defined as the average over the number of steps executed when processing each input weighted by the probability of occurrence of this input, or,

$$C_{avg} = \sum_i p(input_i)steps(input_i)$$

This is the definition of expected value, which assumes that all the possibilities can be determined and that the probability distribution is known, which simply determines a probability of occurrence of each input, $p(input_i)$. The probability function p satis-

2.11 EXERCISES

1. Explain the meaning of the following expressions:

a. $f(n)$ is $O(1)$.

b. $f(n)$ is $\Theta(1)$.

c. $f(n)$ is $n^{O(1)}$.

2. Assuming that $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, prove the following statements:

a. $f_1(n) + f_2(n)$ is $O(\max(g_1(n),g_2(n)))$.

b. If a number k can be determined such that for all $n > k$, $g_1(n) \leq g_2(n)$, then $O(g_1(n)) + O(g_2(n))$ is $O(g_2(n))$.

c. $f_1(n) * f_2(n)$ is $O(g_1(n) * g_2(n))$ (rule of product).

d. $O(cg(n))$ is $O(g(n))$.

e. c is $O(1)$.

3. Prove the following statements:

a. $\Sigma^n_{i=1} i^2$ is $O(n^3)$ and more generally, $\Sigma^n_{i=1} i^k$ is $O(n^{k+1})$.

b. $an^k/\lg n$ is $O(n^k)$ but $an^k/\lg n$ is not $\Theta(n^k)$.

c. $n^{1.1} + n \lg n$ is $\Theta(n^{1.1})$.

d. 2^n is $O(n!)$ and $n!$ is not $O(2^n)$.

e. 2^{n+a} is $O(2^n)$.

f. 2^{2n+a} is not $O(2^n)$.

g. $2^{\sqrt{\lg n}}$ is $O(n^a)$.

4. Make the same assumptions as in Exercise 2 and, by finding counterexamples, refute the following statements:

a. $f_1(n) - f_2(n)$ is $O(g_1(n) - g_2(n))$.

b. $f_1(n)/f_2(n)$ is $O(g_1(n)/g_2(n))$.

5. Find functions f_1 and f_2 such that both $f_1(n)$ and $f_2(n)$ are $O(g(n))$, but $f_1(n)$ is not $O(f_2)$.

6. Is it true that

a. if $f(n)$ is $\Theta(g(n))$, then $2^{f(n)}$ is $\Theta(2^{g(n)})$?

b. $f(n) + g(n)$ is $\Theta(\min(f(n),g(n)))$?

c. 2^{na} is $O(2^n)$?

7. The algorithm presented in this chapter for finding the length of the longest subarray with the numbers in increasing order is inefficient, because there is no need to continue to search for another array if the length already found is greater than the length of the subarray to be analyzed. Thus, if the entire array is already in order, we can discontinue the search right away, converting the worst case into the best. The change needed is in the outer loop, which now has one more test:

```
for (i = 0, length = 1; i < n-1 && length < n==i; i++)
```

What is the worst case now? Is the efficiency of the worst case still $O(n^2)$?

8. Find the complexity of the function used to find the *k*th smallest integer in an unordered array of integers

```
int selectkth(int a[], int k, int n) {
    int i, j, mini, tmp;
    for (i = 0; i < k; i++) {
        mini = i;
        for (j = i+1; j < n; j++)
            if (a[j]<a[mini])
                mini = j;
        tmp = a[i];
        a[i] = a[mini];
        a[mini] = tmp;
    }
    return a[k-1];
}
```

9. Determine the complexity of the following implementations of the algorithms for adding, multiplying, and transposing $n \times n$ matrices:

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        a[i][j] = b[i][j] + c[i][j];

for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        for (k = a[i][j] = 0; k < n; k++)
            a[i][j] += b[i][k] * c[k][j];

for (i = 0; i < n - 1; i++)
    for (j = i+1; j < n; j++) {
        tmp = a[i][j];
        a[i][j] = a[j][i];
        a[j][i] = tmp;
    }
```

10. Find the computational complexity for the following four loops:

a.
```
for (cnt1 = 0, i = 1; i <= n; i++)
    for (j = 1; j <= n; j++)
          cnt1++;
```

b.
```
for (cnt2 = 0, i = 1; i <= n; i++)
      for (j = 1; j <= i; j++)
          cnt2++;
```

c.
```
for (cnt3 = 0, i = 1; i <= n; i *= 2)
    for (j = 1; j <= n; j++)
        cnt3++;
```

d.
```
for (cnt4 = 0, i = 1; i <= n; i *= 2)
    for (j = 1; j <= i; j++)
        cnt4++;
```

11. Find the average case complexity of sequential search in an array if the probability of accessing the last cell equals $\frac{1}{2}$, the probability of the next to last cell equals $\frac{1}{4}$, and the probability of locating a number in any of the remaining cells is the same and equal to $\frac{1}{4(n-2)}$.

12. Consider a process of incrementing a binary n-bit counter. An increment causes some bits to be flipped: Some 0s are changed to 1s, and some 1s to 0s. In the best case, counting involves only one bit switch; for example, when 000 is changed to 001, sometimes all the bits are changed, as when incrementing 011 to 100.

Number	**Flipped Bits**
000	
001	1
010	2
011	1
100	3
101	1
110	2
111	1

Using worst case assessment, we may conclude that the cost of executing $m = 2^n - 1$ increments is $O(mn)$. Use amortized analysis to show that the cost of executing m increments is $O(m)$.

13. How can you convert a satisfiability problem into a three-satisfiability problem for an instance when an alternative in a Boolean expression has two variables? One variable?