Complexity Analysis

2

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:

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

For small values of *n,* the last term, 1,000, is the largest. When *n* equals 10, the second (100*n*) and last (1,000) terms are on equal footing with the other terms, making a small 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 $n \geq N$.

*Bachmann introduced the notation quite casually in his discussion of an approximation of a function and defined it rather succinctly: "with the symbol $O(n)$ we express a magnitude whose order in respect to *n* does not exceed the order of *n*" (Bachmann 1894, p. 401).

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

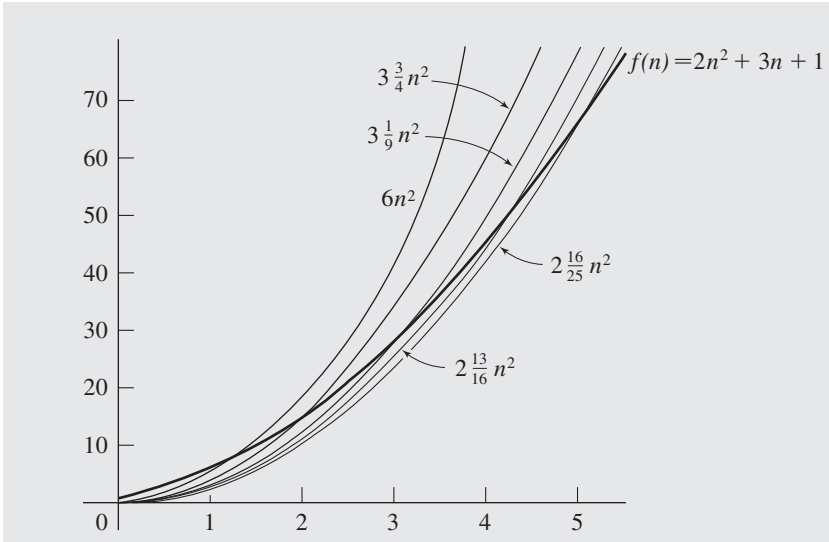
$$2 + \frac{3}{n} + \frac{1}{n^2} \leq c$$

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 $n > 1.5$. 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, . . . , n^k, . . . 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 $f(n)$ is $O(h(n))$. (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 $2n^2$, $3n^2$, $4n^2$, . . . 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 $f(n) \leq cg(n)$ 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 $f(n) \leq cg(n)$ 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 10^8n 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 $n \leq 10^7$, 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, 10^8n 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 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 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
logarithmic	$O(\lg n)$	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.

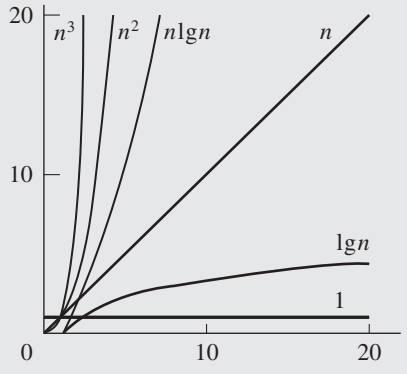


Figure 2.4 indicates that some ill-designed algorithms, or algorithms whose complexity cannot be improved, have no practical application on available computers. To process 1 million items with a quadratic algorithm, over 11 days are needed, and for a cubic algorithm, thousands of years. Even if a computer 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];
    cout<<"sum for subarray 0 through "<< i <<" is "<<sum<<endl;
}
```

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 $i \in \{1, \ldots, n-1\}$ 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];
    cout<<"sum for subarray "<<i-4<<" through "<< i <<" is ""<<sum<<endl;
}
```

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 one initialization of `i`, $n - 4$ autoincrements of `i`, and $n - 4$ initializations of `j` and `sum`, the program makes $1 + 8 \cdot (n - 4) + 3 \cdot (n - 4) = O(n)$ 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;
}
```

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 fifth 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:

```
template<class T>  // overloaded operator < is used;
int binarySearch(const T arr[], int arrSize, const T& key) {
    int lo = 0, mid, hi = arrSize-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,

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

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 satisfies two conditions: It is never negative, $p(input_i) \geq 0$, and all probabilities add up to 1, $\Sigma_i p(input_i) = 1$.

As an example, consider searching sequentially an unordered array to find a number. The best case is when the number is found in the first cell. The worst case is when the number is in the last cell or is not in the array at all. In this case, all the cells are checked to determine this fact. And the average case? We may make the assumption that there is an equal chance for the number to be found in any cell of the array; that is, the probability distribution is uniform. In this case, there is a probability equal to $\frac{1}{n}$ that the number is in the first cell, a probability equal to $\frac{1}{n}$ that it is in the second cell, . . . , and finally, a probability equal to $\frac{1}{n}$ that it is in the last, nth cell. This means that the probability of finding the number after one try equals $\frac{1}{n}$, the probability of having two tries equals $\frac{1}{n}$, . . . , and the probability of having n tries also equals $\frac{1}{n}$. Therefore, we can average all these possible numbers of tries over the number of possibilities and conclude that it takes on the average

$$\frac{1 + 2 + \ldots + n}{n} = \frac{n + 1}{2}$$

steps to find a number. But if the probabilities differ, then the average case gives a different outcome. For example, if the probability of finding a number in the first cell equals $\frac{1}{2}$, the probability of finding it in the second cell equals $\frac{1}{4}$, and the probability of locating it in any of the remaining cells is the same and equal to

$$\frac{1 - \frac{1}{2} - \frac{1}{4}}{n - 2} = \frac{1}{4(n - 2)}$$

then, on the average, it takes

$$\frac{1}{2} + \frac{2}{4} + \frac{3+\ldots n}{4(n - 2)} = 1 + \frac{n(n + 1) - 6}{8(n - 2)} = 1 + \frac{n + 3}{8}$$

steps to find a number, which is approximately four times better than $\frac{n+1}{2}$ found previously for the uniform distribution. Note that the probabilities of accessing a particular cell have no impact on the best and worst cases.

The complexity for the three cases was relatively easy to determine for a sequential search, but usually it is not that straightforward. Particularly, the complexity of the average case can pose difficult computational problems. If the computation is very complex, approximations are used, and that is where we find the big-O, Ω, and Θ notations most useful.

As an example, consider the average case for binary search. Assume that the size of the array is a power of 2 and that a number to be searched has an equal chance to be in any of the cells of the array. Binary search can locate it either after one try in the middle of the array, or after two tries in the middle of the first half of the array, or after two tries in the middle of the second half, or after three tries in the middle of the first quarter of the array; or after three tries in the middle of the fourth quarter, or after four tries in the middle of the first eighth of the array; or after four tries in the middle of the eighth eighth of the array; or after try lg n in the first cell, or after try lg n in the third cell; or, finally, after try lg n in the last cell. That is, the number of all possible tries equals

$$1\cdot 1 + 2\cdot 2 + 4\cdot 3 + 8\cdot 4 + \ldots + \frac{n}{2}\lg n = \sum_{i=0}^{\lg n-1} 2^i(i+1)$$

which has to be divided by $\frac{1}{n}$ to determine the average case complexity. What is this sum equal to? We know that it is between 1 (the best case result) and lg n (the worst case) determined in the preceding section. But is it closer to the best case—say, lg lg n—or to the worst case—for instance, $\frac{\lg n}{2}$, or lg $\frac{n}{2}$, The sum does not lend itself to a simple conversion into a closed form; therefore, its estimation should be used. Our conjecture is that the sum is not less than the sum of powers of 2 in the specified range multiplied by a half of lg n, that is,

$$s_1 = \sum_{i=0}^{\lg n-1} 2^i(i+1) \geq \frac{\lg n}{2}\sum_{i=0}^{\lg n-1} 2^i = s_2$$

The reason for this choice is that s_2 is a power series multiplied by a constant factor, and thus it can be presented in closed form very easily, namely,

$$s_2 = \frac{\lg n}{2}\sum_{i=0}^{\lg n-1} 2^i = \frac{\lg n}{2}\left(1 + 2\frac{2^{\lg n-1}-1}{2-1}\right) = \frac{\lg n}{2}(n-1)$$

which is $\Omega(n \lg n)$. Because s_2 is the lower bound for the sum s_1 under scrutiny—that is, s_1 is $\Omega(s_2)$—then so is $\frac{s_2}{n}$ the lower bound of the sought average case complexity $\frac{s_1}{n}$—that is, $\frac{s_1}{n} = \Omega(\frac{s_2}{n})$. Because $\frac{s_2}{n}$ is $\Omega(\lg n)$, so must be $\frac{s_1}{n}$. Because lg n is an assessment of the complexity of the worst case, the average case's complexity equals $\Theta(\lg n)$.

There is still one unresolved problem: Is $s_1 \geq s_2$? To determine this, we conjecture that the sum of each pair of terms positioned symmetrically with respect to the center of the sum s_1 is not less than the sum of the corresponding terms of s_2. That is,

$$2^0\cdot 1 + 2^{\lg n-1}\lg n \geq 2^0\frac{\lg n}{2} + 2^{\lg n-1}\frac{\lg n}{2}$$

$$2^1\cdot 2 + 2^{\lg n-2}(\lg n-1) \geq 2^1\frac{\lg n}{2} + 2^{\lg n-2}\frac{\lg n}{2}$$

$$\ldots$$

$$2^j(j+1) + 2^{\lg n-1-j}(\lg n-j) \geq 2^j\frac{\lg n}{2} + 2^{\lg n-1-j}\frac{\lg n}{2}$$

$$\ldots$$

where j $\leq \frac{\lg n}{2} - 1$. The last inequality, which represents every other inequality, is transformed into

$$2^{\lg n-1-j}\left(\frac{\lg n}{2} - j\right) \geq 2^j\left(\frac{\lg n}{2} - j - 1\right)$$

and then into

$$2^{\lg n-1-2j} \geq \frac{\frac{\lg n}{2}-j-1}{\frac{\lg n}{2}-j} = 1 - \frac{1}{\frac{\lg n}{2}-j} \qquad \textbf{(2.5)}$$

All of these transformations are allowed because all the terms that moved from one side of the conjectured inequality to another are nonnegative and thus do not change the direction of inequality. Is the inequality true? Because $j \leq \frac{lg\, n}{2} - 1$, $2^{\lg n - 1 - 2j} \geq 2$, and the right-hand side of the inequality (2.5) is always less than 1, the conjectured inequality is true.

This concludes our investigation of the average case for binary search. The algorithm is relatively straightforward, but the process of finding the complexity for the average case is rather grueling, even for uniform probability distributions. For more complex algorithms, such calculations are significantly more challenging.

2.9 Amortized Complexity

In many situations, data structures are subject to a sequence of operations rather than one operation. In this sequence, one operation possibly performs certain modifications that have an impact on the run time of the next operation in the sequence. One way of assessing the worst case run time of the entire sequence is to add worst case efficiencies for each operation. But this may result in an excessively large and unrealistic bound on the actual run time. To be more realistic, amortized analysis can be used to find the average complexity of a worst case sequence of operations. By analyzing sequences of operations rather than isolated operations, amortized analysis takes into account interdependence between operations and their results. For example, if an array is sorted and only a very few new elements are added, then re-sorting this array should be much faster than sorting it for the first time because, after the new additions, the array is nearly sorted. Thus, it should be quicker to put all elements in perfect order than in a completely disorganized array. Without taking this correlation into account, the run time of the two sorting operations can be considered twice the worst case efficiency. Amortized analysis, on the other hand, decides that the second sorting is hardly applied in the worst case situation so that the combined complexity of the two sorting operations is much less than double the worst case complexity. Consequently, the average for the worst case sequence of sorting, a few insertions, and sorting again is lower according to amortized analysis than according to worst case analysis, which disregards the fact that the second sorting is applied to an array operated on already by a previous sorting.

It is important to stress that amortized analysis is analyzing sequences of operations, or if single operations are analyzed, it is done in view of their being part of the sequence. The cost of operations in the sequence may vary considerably, but how frequently particular operations occur in the sequence is important. For example, for the sequence of operations op_1, op_2, op_3, . . . , the worst case analysis renders the computational complexity for the entire sequence equal to

$$C(op_1, op_2, op_3, \ldots) = C_{worst}(op_1) + C_{worst}(op_2) + C_{worst}(op_3) + \ldots$$

whereas the average complexity determines it to be

$$C(op_1, op_2, op_3, \ldots) = C_{avg}(op_1) + C_{avg}(op_2) + C_{avg}(op_3) + \ldots$$

Although specifying complexities for a sequence of operations, neither worst case analysis nor average case analysis was looking at the position of a particular operation in the sequence. These two analyses considered the operations as executed in isolation and the sequence as a collection of isolated and independent operations. Amortized analysis changes the perspective by looking at what happened up until a particular point in the sequence of operations and then determines the complexity of a particular operation,

$$C(op_1, op_2, op_3, \ldots) = C(op_1) + C(op_2) + C(op_3) + \ldots$$

where C can be the worst, the average, the best case complexity, or very likely, a complexity other than the three depending on what happened before. To find amortized complexity in this way may be, however, too complicated. Therefore, another approach is used. The knowledge of the nature of particular processes and possible changes of a data structure is used to determine the function C, which can be applied to each operation of the sequence. The function is chosen in such a manner that it considers quick operations as slower than they really are and time-consuming operations as quicker than they actually are. It is as though the cheap (quick) operations are charged more time units to generate credit to be used for covering the cost of expensive operations that are charged below their real cost. It is like letting the government charge us more for income taxes than necessary so that at the end of the fiscal year the overpayment can be received back and used to cover the expenses of something else. The art of amortized analysis lies in finding an appropriate function C so that it overcharges cheap operations sufficiently to cover expenses of undercharged operations. The overall balance must be nonnegative. If a debt occurs, there must be a prospect of paying it.

Consider the operation of adding a new element to the vector implemented as a flexible array. The best case is when the size of the vector is less than its capacity because adding a new element amounts to putting it in the first available cell. The cost of adding a new element is thus $O(1)$. The worst case is when size equals capacity, in which case there is no room for new elements. In this case, new space must be allocated, the existing elements are copied to the new space, and only then can the new element be added to the vector. The cost of adding a new element is $O(\text{size}(vector))$. It is clear that the latter situation is less frequent than the former, but this depends on another parameter, capacity increment, which refers to how much the vector is increased when overflow occurs. In the extreme case, it can be incremented by just one cell, so in the sequence of m consecutive insertions, each insertion causes overflow and requires $O(size(vector))$ time to finish. Clearly, this situation should be delayed. One solution is to allocate, say, 1 million cells for the vector, which in most cases does not cause an overflow, but the amount of space is excessively large and only a small percentage of space allocated for the vector may be expected to be in actual use. Another solution to the problem is to double the space allocated for the vector if overflow occurs. In this case, the pessimistic $O(size(vector))$ performance of the insertion operation may be expected to occur only infrequently. By using this estimate, it may be claimed that, in the best case, the cost of inserting m items is $O(m)$, but it is impossible to claim that, in the worst case, it is $O(m \cdot size(vector))$. Therefore, to see better what impact this performance has on the sequence of operations, the amortized analysis should be used.

In amortized analysis, the question is asked: What is the expected efficiency of a sequence of insertions? We know that the best case is $O(1)$ and the worst case is

O(*size*(*vector*)), but also we know that the latter case occurs only occasionally and leads to doubling the size of the vector. In this case, what is the expected efficiency of one insertion in the series of insertions? Note that we are interested only in sequences of insertions, excluding deletions and modifications, to have the worst case scenario. The outcome of amortized analysis depends on the assumed amortized cost of one insertion. It is clear that if

$$amCost(push(x)) = 1$$

where 1 represents the cost of one insertion, then we are not gaining anything from this analysis because easy insertions are paying for themselves right away, and the insertions causing overflow and thus copying have no credit to use to make up for their high cost. Is

$$amCost(push(x)) = 2$$

a reasonable choice? Consider the table in Figure 2.6a. It shows the change in vector capacity and the cost of insertion when size grows from 0 to 18; that is, the table indicates the changes in the vector during the sequence of 18 insertions into an initially empty vector. For example, if there are four elements in the vector (size = 4), then

FIGURE 2.6 Estimating the amortized cost.

(a)

Size	Capacity	Amortized Cost	Cost	Units Left
0	0			
1	1	2	0 + 1	1
2	2	2	1 + 1	1
3	4	2	2 + 1	0
4	4	2	1	1
5	8	2	4 + 1	–2
6	8	2	1	–1
7	8	2	1	0
8	8	2	1	1
9	16	2	8 + 1	–6
10	16	2	1	–5
:	:	:	:	:
16	16	2	1	1
17	32	2	16 + 1	–14
18	32	2	1	–13
:	:	:	:	:

(b)

Size	Capacity	Amortized Cost	Cost	Units Left
0	0			
1	1	3	0 + 1	2
2	2	3	1 + 1	3
3	4	3	2 + 1	3
4	4	3	1	5
5	8	3	4 + 1	3
6	8	3	1	5
7	8	3	1	7
8	8	3	1	9
9	16	3	8 + 1	3
10	16	3	1	5
:	:	:	:	:
16	16	3	1	17
17	32	3	16 + 1	3
18	32	3	1	5
:	:	:	:	:

before inserting the fifth element, the four elements are copied at the cost of four units and then the new fifth element is inserted in the newly allocated space for the vector. Hence, the cost of the fifth insertion is 4 + 1. But to execute this insertion, two units allocated for the fifth insertion are available plus one unit left from the previous fourth insertion. This means that this operation is two units short to pay for itself. Thus, in the Units Left column, – 2 is entered to indicate the debt of two units. The table indicates that the debt decreases and becomes zero, one cheap insertion away from the next expensive insertion. This means that the operations are almost constantly executed in the red, and more important, if a sequence of operations finishes before the debt is paid off, then the balance indicated by amortized analysis is negative, which is inadmissible in the case of algorithm analysis. Therefore, the next best solution is to assume that

$$amCost(push(x)) = 3$$

The table in Figure 2.6b indicates that we are never in debt and that the choice of three units for amortized cost is not excessive because right after an expensive insertion, the accumulated units are almost depleted.

In this example, the choice of a constant function for amortized cost is adequate, but usually it is not. A function is defined as *potential* when it assigns a number to a particular state of a data structure *ds* that is a subject of a sequence of operations. The amortized cost is defined as a function

$$amCost(op_i) = cost(op_i) + potential(ds_i) - potential(ds_{i-1})$$

which is the real cost of executing the operation op_i plus the change in potential in the data structure *ds* as a result of execution of op_i. This definition holds for one single operation of a sequence of *m* operations. If amortized costs for all the operations are added, then the amortized cost for the sequence

$$\begin{aligned} amCost(op_1,\ldots,op_m) &= \sum_{i=1}^{m}(cost(op_i) + potential(ds_i) - potential(ds_{i-1})) \\ &= \sum_{i=1}^{m}(cost(op_i) + potential(ds_m) - potential(ds_0)) \end{aligned}$$

In most cases, the potential function is initially zero and is always nonnegative so that amortized time is an upper bound of real time. This form of amortized cost is used later in the book.

Amortized cost of including new elements in a vector can now be phrased in terms of the potential function defined as

$$potential(vector_i) = \begin{cases} 0 & \text{if } size_i = capacity_i \text{ (vector is full)} \\ 2size_i - capacity_i & \text{otherwise} \end{cases}$$

To see that the function works as intended, consider three cases. The first case is when a cheap pushing follows cheap pushing (vector is not extended right before the current push and is not extended as a consequence of the current push) and

$$amCost(push_i()) = 1 + 2size_{i-1} + 2 - capacity_{i-1} - 2size_{i-1} + capacity_i = 3$$

because the capacity does not change, $size_i = size_{i-1} + 1$, and the actual cost equals 1. For expensive pushing following cheap pushing,

$$amCost(push_i()) = size_{i-1} + 2 + 0 - 2size_{i-1} + capacity_{i-1} = 3$$

because $size_{i-1} + 1 = capacity_{i-1}$ and the actual cost equals $size_i + 1 = size_{i-1} + 2$, which is the cost of copying the vector elements plus adding the new element. For cheap pushing following expensive pushing,

$$amCost(push_i()) = 1 + 2size_i - capacity_i - 0 = 3$$

because $2(size_i - 1) = capacity_i$ and actual cost equals 1. Note that the fourth case, expensive pushing following expensive pushing, occurs only twice, when capacity changes from zero to one and from one to zero. In both cases, amortized cost equals 3.

2.10 NP-Completeness

A *deterministic* algorithm is a uniquely defined (determined) sequence of steps for a particular input; that is, given an input and a step during execution of the algorithm, there is only one way to determine the next step that the algorithm can make. A *nondeterministic* algorithm is an algorithm that can use a special operation that makes a guess when a decision is to be made. Consider the nondeterministic version of binary search.

If we try to locate number k in an unordered array of numbers, then the algorithm first accesses the middle element m of the array. If $m = k$, then the algorithm returns m's position; if not, the algorithm makes a guess concerning which way to go to continue: to the left of m or to its right. A similar decision is made at each stage: If number k is not located, continue in one of the two halves of the currently scrutinized subarray. It is easy to see that such a guessing very easily may lead us astray, so we need to endow the machine with the power of making correct guesses. However, an implementation of this nondeterministic algorithm would have to try, in the worst case, all the possibilities. One way to accomplish it is by requiring that the decision in each iteration is in reality this: if $m \neq k$, then go both to the right and to the left of m. In this way, a tree is created that represents the decisions made by the algorithm. The algorithm solves the problem, if any of the branches allows us to locate k in the array that includes k and if no branch leads to such a solution when k is not in the array.

A *decision problem* has two answers, call them "yes" and "no." A decision problem is given by the set of all instances of the problem and the set of instances for which the answer is "yes." Many optimization problems do not belong to that category ("find the minimum x for which . . ."") but in most cases they can be converted to decision problems ("is x, for which . . . , less than k?").

Generally, a nondeterministic algorithm solves a decision problem if it answers it in the affirmative and there is a path in the tree that leads to a yes answer, and it answers it in the negative if there is no such path. A nondeterministic algorithm is considered polynomial if a number of steps leading to an affirmative answer in a decision tree is $O(n^k)$, where n is the size of the problem instance.

Most of the algorithms analyzed in this book are polynomial-time algorithms; that is, their running time in the worst case is $O(n^k)$ for some k. Problems that can be solved with such algorithms are called *tractable* and the algorithms are considered *efficient*.

A problem belongs to the class of P problems if it can be solved in polynomial time with a deterministic algorithm. A problem belongs to the class of NP problems if it can be solved in polynomial time with a nondeterministic algorithm. P problems are obviously tractable. NP problems are also tractable, but only when nondeterministic algorithms are used.

Clearly, P $\subseteq$ NP, because deterministic algorithms are those nondeterministic algorithms that do not use nondeterministic decisions. It is also believed that P $\neq$ NP; that is, there exist problems with nondeterministic polynomial algorithms that cannot be solved with deterministic polynomial algorithms. This means that on deterministic Turing machines they are executed in nonpolynomial time and thus they are intractable. The strongest argument in favor of this conviction is the existence of NP-complete problems. But first we need to define the concept of reducibility of algorithms.

A problem P_1 is reducible to another problem P_2 if there is a way of encoding instances x of P_1 as instances $y = r(x)$ of P_2 using a *reduction function* r executed with a *reduction algorithm*; that is, for each x, x is an instance of P_1 if $y = r(x)$ is an instance of P_2. Note that reducibility is not a symmetric relation: P_1 can be reducible to P_2 but not necessarily vice versa; that is, each instance x of P_1 should have a counterpart y of P_2, but there may be instances y of P_2 onto which no instances x of P_1 are mapped with the function r. Therefore, P_2 can be considered a harder problem than P_1.

The reason for the reduction is that if the value $r(x)$ for any x can be found efficiently (in polynomial time), then an efficient solution for y can be efficiently transformed into an efficient solution of x. Also, if there is no efficient algorithm for x, then there is no efficient solution for y.

A problem is called *NP-complete* if it is NP (it can be solved efficiently by a nondeterministic polynomial algorithm) and every NP problem can be polynomially reduced to this problem. Because reducibility is a transitive relation, we can also say that an NP problem P_1 is NP-complete if there is an NP-complete problem P_2 that is polynomially reducible to P_1. In this way, all NP-complete problems are computationally equivalent; that is, if an NP-complete problem can be solved with a deterministic polynomial algorithm, then so can all NP-complete problems, and thus P = NP. Also, if any problem in NP is intractable, then all NP-complete problems are intractable.

The reduction process uses an NP-complete problem to show that another problem is also NP-complete. There must be, however, at least one problem directly proven to be NP-complete by other means than reduction to make the reduction process possible. A problem that was shown by Stephen Cook to be in that category is the satisfiability problem.

The *satisfiability problem* concerns Boolean expressions in conjunctive normal form (CNF). An expression is in CNF if it is a conjunction of alternatives where each alternative involves Boolean variables and their negations, and each variable is either true or false. For example,

$$(x \vee y \vee z) \wedge (w \vee x \vee \neg y \vee z) \wedge (\neg w \vee \neg y)$$

is in CNF. A Boolean expression is *satisfiable* if there exists an assignment of values true and false that renders the entire expression true. For example, our expression is satisfiable for x = false, y = false, and z = true. The satisfiability problem consists of determining whether a Boolean expression is satisfiable (the value assignments do not have to be given). The problem is NP, because assignments can be guessed and then the expression can be tested for satisfiability in polynomial time.

Cook proves that the satisfiability problem is NP-complete by using a theoretical concept of the Turing machine that can perform nondeterministic decisions (make good guesses). Operations of that machine are then described in terms of Boolean expressions, and it is shown that the expression is satisfiable if the Turing machine terminates for a particular input (for the proof, see Appendix C).

To illustrate the reduction process, consider the *3-satisfiability problem,* which is the satisfiability problem in the case when each alternative in a Boolean expression in CNF includes only three different variables. We claim that the problem is NP-complete. The problem is NP, because a guessed assignment of truth values to variables in a Boolean expression can be verified in polynomial time. We show that the 3-satisfiability problem is NP-complete by reducing it to the satisfiability problem. The reduction process involves showing that an alternative with any number of Boolean variables can be converted into a conjunction of alternatives, each alternative with three Boolean variables only. This is done by introducing new variables. Consider an alternative

$$A = (p_1 \vee p_2 \vee \ldots \vee p_k)$$

for $k \geq 4$ where $p_i \in \{x_i, \neg x_i\}$. With new variables $y_1, \ldots, y_{k-3}$, we transform A into

$$A' = (p_1 \vee p_2 \vee y_1) \wedge (p_3 \vee \neg y_1 \vee y_2) \wedge (p_4 \vee \neg y_2 \vee y_3) \wedge \ldots \wedge (p_{k-2} \vee \neg y_{k-4} \vee y_{k-3}) \wedge (p_{k-1} \vee p_k \vee \neg y_{k-3})$$

If the alternative A is satisfiable, then at least one term p_i is true, so the values of y_j's can be so chosen that A' is true: if p_i is true, then we set $y_1, \ldots, y_{i-2}$ to true and the remaining $y_{i-1}, \ldots, y_{k-3}$ to false. Conversely, if A' is satisfiable, then at least one p_i must be true, because if all p_i's are false, then the expression

$$A' = (\text{false} \vee \text{false} \vee y_1) \wedge (\text{false} \vee \neg y_1 \vee y_2) \wedge (\text{false} \vee \neg y_2 \vee y_3) \wedge \ldots \wedge (\text{false} \vee \text{false} \vee \neg y_{k-3})$$

has the same truth value as the expression

$$(y_1) \wedge (\neg y_1 \vee y_2) \wedge (\neg y_2 \vee y_3) \wedge \ldots \wedge (\neg y_{k-3})$$

which cannot be true for any choice of values for y_j's, thus it is not satisfiable.

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. $\sum_{i=1}^{n} i^2$ is $O(n^3)$ and more generally, $\sum_{i=1}^{n} i^k$ 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 3-satisfiability problem for an instance when an alternative in a Boolean expression has two variables? One variable?

BIBLIOGRAPHY

Computational Complexity

Arora, Sanjeev, and Barak, Boaz, *Computational Complexity: A Modern Approach,* Cambridge: Cambridge University Press, 2009.

Fortnow, Lance, "The Status of the P versus NP Problem," *Communications of the ACM* 52 (2009), No. 9, 78–86.

Hartmanis, Juris, and Hopcroft, John E., "An Overview of the Theory of Computational Complexity," *Journal of the ACM* 18 (1971), 444–475.

Hartmanis, Juris, and Stearns, Richard E., "On the Computational Complexity of Algorithms," *Transactions of the American Mathematical Society* 117 (1965), 284–306.

Preparata, Franco P., "Computational Complexity," in Pollack, S. V. (ed.), *Studies in Computer Science,* Washington, DC: The Mathematical Association of America, 1982, 196–228.

Big-O, Ω, and Θ Notations

Bachmann, Paul G.H., *Zahlentheorie,* vol. 2: *Die analytische Zahlentheorie,* Leipzig: Teubner, 1894.

Brassard, Gilles, "Crusade for a Better Notation," *SIGACT News* 17 (1985), No. 1, 60–64.

Gurevich, Yuri, "What does O(n) mean?" *SIGACT News* 17 (1986), No. 4, 61–63.

Knuth, Donald, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms,* Reading, MA: Addison-Wesley, 1998.

Knuth, Donald, "Big Omicron and Big Omega and Big Theta," *SIGACT News,* 8 (1976), No. 2, 18–24.

Vitanyi, Paul M. B., and Meertens, Lambert, "Big Omega versus the Wild Functions," *SIGACT News* 16 (1985), No. 4, 56–59.

OO Notation

Manber, Udi, *Introduction to Algorithms: A Creative Approach,* Reading, MA: Addison-Wesley, 1989.

Amortized Analysis

Heileman, Gregory L., *Discrete Structures, Algorithms, and Object-Oriented Programming,* New York: McGraw-Hill, 1996, chs. 10–11.

Tarjan, Robert E., "Amortized Computational Complexity," *SIAM Journal on Algebraic and Discrete Methods* 6 (1985), 306–318.

NP-Completeness

Cook, Stephen A., "The Complexity of Theorem-Proving Procedures," *Proceedings of the Third Annual ACM Symposium on Theory of Computing,* 1971, 151–158.

Garey, Michael R., and Johnson, David S., *Computers and Intractability: A Guide to the Theory of NP-Completeness,* San Francisco: Freeman, 1979.

Johnson, David S., and Papadimitriou, Christos H., Computational Complexity, in Lawler, E. L., Lenstra, J. K., Rinnoy, Kan A. H. G., and Shmoys, D. B. (eds.), *The Traveling Salesman Problem,* New York: Wiley, 1985, 37–85.

Karp, Richard M., "Reducibility Among Combinatorial Problems," in R. E. Miller and J. W. Thatcher (eds.), *Complexity of Computer Computations,* New York: Plenum Press, 1972, 85–103.