1.2 Algorithm Efficiency and the Sorting Problem

As we pointed out in the opening paragraphs of this chapter, from a practical perspective a measure of the efficiency of an algorithm is achieved by analyzing the efficiency with which its implementation utilizes a computer's time and space. By space efficiency, we mean the amount of memory an algorithm consumes when it runs. As for time efficiency, at first glance one would expect this to mean the amount of time it takes the algorithm to execute; however, there are several reasons why such an absolute measure is not appropriate:

- The execution time of an algorithm is sensitive to the amount of data that it must manipulate and typically grows as the amount of data increases.
- The execution times for an algorithm when run with the same data set on two different computers may differ because of the execution speeds of the processors.
- Depending on how an algorithm is implemented on a particular computer (choice of programming language, use of a compiler or interpreter, and so forth), one implementation of an algorithm may run faster than another, even on the same computer and with the same data set.

In our opening remarks we observed that an algorithm is one of the *abstractions* that a computer scientist studies. Consequently, in assessing the efficiency of an algorithm's run time we want to remove all implementation considerations from our analysis and focus on those aspects of the algorithm that most critically affect this execution time. We noted that one of these is the number of data items the algorithm manipulates. Typically, the rest of the analysis consists of trying to determine how often a critical operation (e.g., a comparison, data interchange, or addition or multiplication of values) or sequence of such operations gets performed in manipulating these data items. This count, expressed as a function of a variable **n** that provides an indicator of the size of the set of data items, is what represents the "running time" of the algorithm.

Consider the following two examples of nested loops intended to sum each of the rows of an **n** × **n** array **a**, storing the row sums in a one-dimensional array **sum** and the overall total in **grandTotal**.

```
Version 1
grandTotal = 0;
for (k = 0; k < n; ++k)
{
  sum[k] = 0;
  for (j = 0; j < n; ++j)
  {
    sum[k] = sum[k] + a[k][j];
    grandTotal = grandTotal + a[k][j];
  }
}
Version 2
grandTotal = 0;
for (k = 0; k < n; ++k)
{
  sum[k] = 0;
  for (j = 0; j < n; ++j)
    sum[k] = sum[k] + a[k][j];
  grandTotal = grandTotal + sum[k];
}
```

If we analyze the number of addition operations required by these two versions, we see that Version 2 is better in this respect. Because Version 1 incorporates the accumulation of **grandTotal** into its inner loop, it requires $2\mathbf{n}^2$ additions. That is, the additions **sum[k] + a[k][j]** and **grandTotal + a[k][j]** are each executed $\mathbf{n}^2$ times, for a total of $2\mathbf{n}^2$. Version 2, on the other hand, accumulates **grandTotal** after the inner loop; hence it requires only $\mathbf{n}^2 + \mathbf{n}$ additions, which is less than $2\mathbf{n}^2$ for any **n** greater than 1. Version 2 is apparently guaranteed to execute faster than Version 1 for any nontrivial value of **n**. As a final observation, we point out that the variable **n** being used in our analysis is merely an indicator of the number of data items being manipulated (namely, $\mathbf{n}^2$) rather than giving the actual number of data items.

Note also that "faster" here may not have much significance in the real world of computing. If we assume a hypothetical computer that allows us to store a 1000 × 1000 array and that executes at a microsecond per instruction, Version 1 would require two seconds to perform its addition; Version 2 would require just over one second. On a 100,000 × 100,000 array, Version 1 would crunch numbers for slightly under six hours and Version 2 would take about three hours. Although Version 2 is certainly better from an aesthetic perspective, it may not be good enough to be appreciably different from a user's perspective. That is, in situations where one version will respond within seconds, so will the other. Conversely, when one is annoyingly slow, the other will be also. For the 1000 × 1000 array, both versions would be fast enough to allow their use in an interactive environment. For the 100,000 × 100,000 array, both versions would dictate an overnight run in batch mode since an interactive user will be no more willing to wait three hours than six hours for a response.

In terms of the *order of magnitude* of run time involved, these versions should not be considered significantly different. Order of magnitude is an expression used by scientists to loosely compare two values. Traditionally, two positive values A and B are of the same order of magnitude if the ratio of the larger to the smaller is less than 10 : 1. On the other hand, value A would be *k* orders of magnitude greater than value B if the ratio of A to B is between 10^k : 1 and 10^{k+1} : 1. However, the reliance on powers of 10 for determining the ranges for these ratios is frequently relaxed and other ranges used instead. For example, for time measurements, seconds, minutes, hours, days, months, and years could be used to assess orders of magnitude. Under this progression, an hour would be two orders of magnitude above a second rather than the three orders the traditional method would yield.

Considerations such as these have led computer scientists to use a method of algorithm classification that makes more precise the notion of order of magnitude as it applies to time and space considerations. This method of classification, typically referred to as *big-O notation* (in reference to "on the order of"), hinges on the definition given in the following section.

Big-O Analysis

Suppose there exists a function $f(n)$ defined on the nonnegative integers such that the number of operations required by an algorithm for an input of size n, say $T(n)$, is less than some constant C times $f(n)$ for all sufficiently large values of n. That is, there is a positive integer M and a constant C such that for all $n \geq M$ we have $T(n) \leq Cf(n)$. Such an algorithm is said to be an $O(f(n))$ algorithm relative to the number of operations it requires to execute. Similarly, we could classify an algorithm

as $O(f(n))$ relative to the number of memory locations it requires to execute. The constant C is known as the *constant of proportionality* of the order relationship.

Consider the algorithms discussed earlier for summing the elements of the $n \times n$ array. For the Version 1 algorithm we calculated its run time to be $T(n) = 2n^2$ for all n; hence, with $C = 2$ in the definition of big-O, we see that $T(n)$ is $O(n^2)$. For the Version 2 algorithm, whose run time we calculated as $n^2 + n$, we note when $n \geq 1$, we have $n \leq n^2$, so that $n^2 + n \leq n^2 + n^2 = 2n^2$. Thus, this algorithm is also $O(n^2)$.

Essentially, the definition of $O(f(n))$ as applied to the run time of an algorithm states that, up to a constant factor, the function $f(n)$ gives an upper bound on how the algorithm is performing for large n; saying in effect that as n gets larger, the growth in execution time will be no worse than that shown by $f(n)$. What the definition does not say is how good this upper bound is. For example, although the Version 1 and Version 2 algorithms just analyzed were both $O(n^2)$, we could just as well have said they were $O(n^3)$, since when $n \geq 2$ we have $2n^2 \leq n^3$; similarly they are $O(n^4)$, or $O(n^2 \log_2 n)$, and so forth. If the statement "$T(n)$ is $O(f(n))$" is to be meaningful, it must be understood that f is the "smallest" such function that can be used. In practice, the manner in which f is determined in the analysis of an algorithm often ensures this minimality, but there are algorithms for which such minimal functions are currently unavailable (see the discussion of the Shell sort in Section 1.4).

Two questions merit further discussion at this stage.

How well does big-O analysis provide a way of classifying algorithms from a real-world perspective? To answer this question, consider Table 1.1. This table presents some of the typical $f(n)$ functions we will use to classify algorithms and their order of magnitude run time for an input of size 10^5 on our hypothetical computer. From this table, we can see that an $O(n^2)$ algorithm will take hours to execute for an input of size 10^5—just how many hours depends on the constant of proportionality in the definition of the big-O notation. Regardless of the value of this constant of proportionality, however, a categorization of an algorithm as an $O(n^2)$ algorithm has achieved a very practical goal: We now know that, for an input of size 10^5, we cannot expect an immediate response for such an algorithm. Moreover, we

TABLE 1.1
Some typical $f(n)$ functions and their associated run times.

$f(n)$	Order of magnitude run time for input of size 10^5 (assuming proportionality constant $k = 1$ and one operation per microsecond)
$\log_2 n$	2×10^{-5} second
n	0.1 second
$n \log_2 n$	2 seconds
n^2	3 hours
n^3	32 years
2^n	*many* centuries

know that, for a reasonably small constant of proportionality, we have an algorithm for which submission as an overnight job would be practical. That is, unlike an $O(n^3)$ algorithm, we could expect the computer to finish executing our algorithm in a time frame that would be acceptable, if it could be scheduled so as not to interfere with other uses of the machine. On the other hand, an $O(n^3)$ algorithm applied to a data set of this size would be completely impractical.

Two other observations on using big-O notation to classify algorithms should be mentioned. The first pertains to the use of the phrase "for all sufficiently large values of n" in our definition of big-O. This highlights the fact that there is usually little difference in the choice of an algorithm if n is reasonably small. For example, almost any sorting algorithm would sort 100 integers quickly. Second, the constant of proportionality used to establish that an algorithm's run-time efficiency is $O(f(n))$ is crucial only for comparing algorithms that share the same function $f(n)$; it makes almost no difference when comparing algorithms whose $f(n)$'s are of different magnitudes. It is therefore appropriate to say that the function $f(n)$ dominates the run-time performance of an algorithm and characterizes it in its big-O analysis. The following example should help clarify this situation.

Consider two algorithms L_1 and L_2 with run times equal to $2n^2$ and n^2, respectively. The constants of proportionality of L_1 and L_2 are 2 and 1, respectively. The dominating function $f(n)$ for both of these algorithms is n^2, but L_2 runs twice as fast as L_1 for a data set of n values. The different sizes of the two constants of proportionality indicate that L_2 is faster than L_1. Now suppose that the function $f(n)$ for L_2 is n^3. Then, even though its constant of proportionality is half of what it is for L_1, L_2 will be frustratingly slower than L_1 for large n. This latter comparison is shown in Figure 1.4.

How does one determine the function $f(n)$ that categorizes a particular algorithm? We will give an overview of that process here and later illustrate it more fully by doing actual analyses for two sorting algorithms. In general, the run-time behavior of an algorithm is dominated by its behavior in any loops it contains. Hence, by analyzing the loop structures of an algorithm, we can estimate the number of run-time operations required by the algorithm as a sum of several terms, each dependent on n, the indicator of the number of items being processed by the algorithm. That is, we are typically able to express the number of run-time operations (and, for that matter, the amount of memory) as a sum of the form

$$f_1(n) + f_2(n) + \cdots + f_k(n)$$

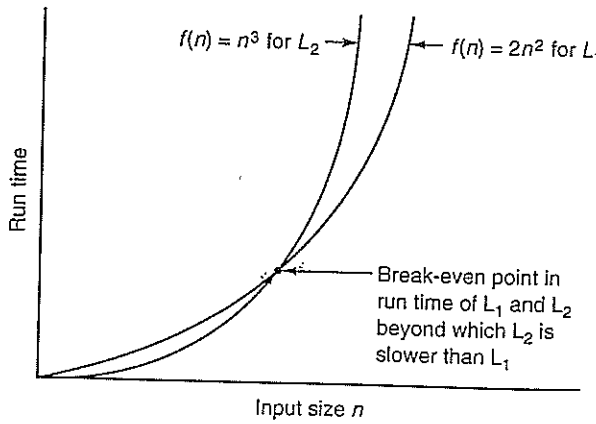


Figure 1.4
Graphical comparison of two run times.

It is also typical that we identify one of the terms in this expression as the *dominant* term. A dominant term is one that, for bigger values of n, becomes so large that it allows us to ignore all the other terms, from a big-O perspective. For instance, suppose that we had an expression involving two terms, such as $n^2 + 6n$. The n^2 term dominates the $6n$ term since, for $6 \leq n$, we have $6n \leq n^2$, whence

$$n^2 + 6n \leq n^2 + n^2 = 2n^2$$

Thus, $n^2 + 6n$ would lead to an $O(n^2)$ categorization because of the dominance of the n^2 term. In general, the problem of big-O categorization can be reduced to one of finding the dominant term in an expression representing the number of operations (or amount of memory) required by an algorithm.

Example 1.2 Use big-O analysis to determine the time efficiency of the following C++ fragment in terms of the integer **n**:

```
for (k = 0; k < n/2; ++k)
{
   .
   .
   .
   for (j = 0; j < n*n; ++j)
   {
      .
      .
      .
   }
}
```

Because these loops are nested, the critical operations for this analysis will be those within the innermost loop—they are the ones that will execute most often. Operations within the innermost loop will each execute n^2 times whenever the innermost loop executes. We see in fact that this innermost loop will execute $n/2$ times. Consequently, the critical operations for this algorithm will execute $(n/2)(n^2)$, or $n^3/2$, times. It follows then that the run-time efficiency of this algorithm is $O(n^3)$ in big-O terms, with a constant of proportionality equal to $1/2$.

Example 1.3 Use big-O analysis to determine the time efficiency of the following C++ fragment in terms of the integer **n**:

```
for (k = 0; k < n/2; ++k)
{
   .
   .
   .
}
for (j = 0; j < n*n; ++j)
{
   .
   .
   .
}
```

Since one loop follows the other, the number of operations executed by both of them is the sum of the individual loop efficiencies. Hence, the efficiency is $n/2+n^2$, or $O(n^2)$ in big-O terms.

Example 1.4 Use big-O analysis to determine the time efficiency of the following C++ fragment in terms of the integer n:

```
k = n;
while (k > 1)
{
   .
   .
   .
   k /= 2;     // integer division; equivalent to k = k/2;
}
```

Since the loop control variable is cut in half each time through the loop, the number of times that statements inside the loop will be executed is measured by $\log_2 n$. For instance, if n is 64, then the loop will be executed for the following values of k:

64
32
16
8
4
2

Note that this yields six loop iterations, that is, $\log_2 64$. For values of n that are not precisely a power of 2, the number of loop iterations will be the smallest integer greater than $\log_2 n$. In any case, the run-time efficiency of this algorithm is $O(\log_2 n)$.

Table 1.2, which lists frequently occurring dominant terms, will prove helpful in our future big-O analyses of algorithms. It is worthwhile to characterize briefly some of the classes of algorithms that arise from the dominant terms listed in Table 1.2. Algorithms with an efficiency function that is dominated by the $\log_a n$ term—and hence are categorized as $O(\log_a n)$—are often called *logarithmic algorithms*. Since $\log_a n$ will increase more slowly than n itself, logarithmic algorithms are generally very efficient. Algorithms with an efficiency function that can be expressed in terms of a polynomial of the form

$$a_m n^m + a_{m-1} n^{m-1} + \cdots + a_2 n^2 + a_1 n + a_0$$

are called *polynomial algorithms*. Since the highest power of n will dominate such a polynomial, these algorithms are $O(n^m)$. The only polynomial algorithms we will discuss in this book have $m = 1$, 2, or 3; they are called *linear, quadratic,* and *cubic* algorithms, respectively.

Algorithms with an efficiency function that is dominated by a term of the form a^n are called *exponential algorithms*. These algorithms are of more theoretical than practical interest because for moderate or large values of n they cannot reasonably be run on typical computers. We will not encounter algorithms in the exponential

TABLE 1.2
Common dominant terms in expressions for algorithmic efficiency based on the variable n.

n dominates $\log_a n$; a is often 2
$n \log_a n$ dominates n; a is often 2
n^m dominates n^k when $m > k$
a^n dominates n^m for any values of a and m greater than 1

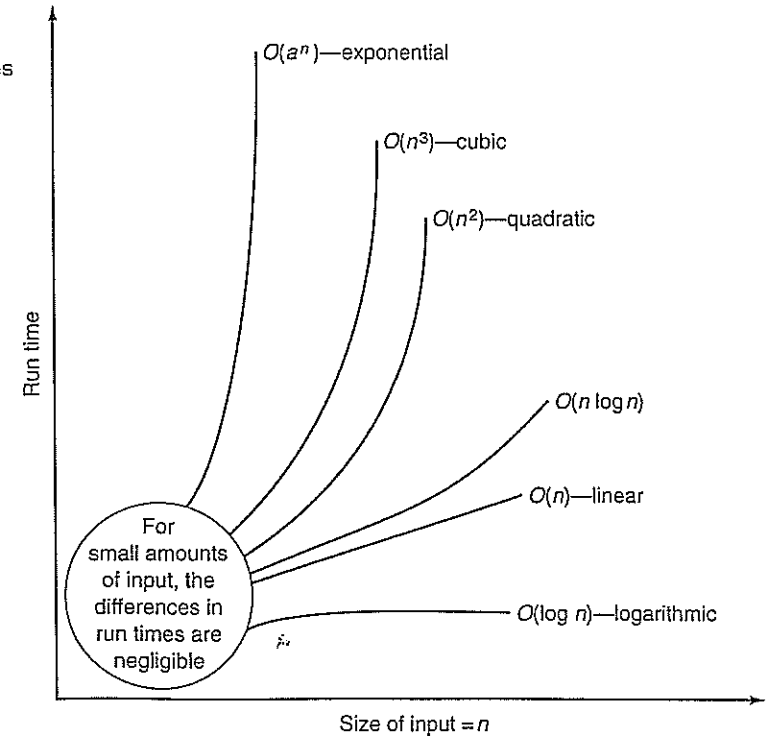
and $n \log_a n$ categories until we study recursion in Chapter 6. At that time, we will develop some additional principles for analyzing such algorithms. The relationships between all of these various classes of algorithms are graphically summarized in Figure 1.5.

The Sorting Problem

Stated generally, the sorting problem requires that one take a given sequence of items (perhaps presented in an array or a sequential file) and reorder them so that an item and its successor satisfy a prescribed ordering relationship (for example, greater than or less than). Here we apply the concepts of big-O analysis to determine the efficiency of two sorting algorithms.

For the purposes of discussion, we can use the following C++ interface to specify the pre- and postconditions for the sorting problem when the items to be sorted appear in an array.

Figure 1.5
Relationships in run time between commonly occurring classes of algorithms.



```
//-----------------------------------------------------------------
// Interface for function sort
// GIVEN:   a -- an array of ElementType values;
//          numvals -- number of values in the array,
//                     assume numvals >= 0;
//          precedes -- a function to compare ElementType values:
//            GIVEN:   x and y -- values of type ElementType to
//                                compare
//            RETURN as value of function:
//                     TRUE   if x precedes y;
//                     FALSE  if x and y are equal, or
//                            if y precedes x
// RETURN:  The array a with its values arranged in order by the
//          precedes relation
// RETURN as value of function: void

template <class ElementType>
void sort(ElementType a[], int numvals,
          BOOLEAN (*precedes)(const ElementType &x, const ElementType &y))
```

Selection Sort

A simple solution to the sorting problem just specified is the *selection sort* algorithm. This algorithm repeatedly finds the position of the entry that should come first from among array positions **k**, **k** + 1, ..., **numvals** − 1. This entry is then exchanged with the entry in position **k**. Hence, as **k** runs from 0 to **numvals** − 2, we *select* the entry that belongs in index zero and swap it there, then the entry which belongs in index 1 and swap it there, and so on (thus the name *selection sort*). The action of this sorting algorithm on a small array is illustrated in Figure 1.6. In this figure we assume that the *precedes* relation is the usual *less than* relation between integers. Hence, the five-element array is arranged in ascending numerical order after four passes.

It is apparent from Figure 1.6 that sorting an array of **numvals** elements will require **numvals** − 1 passes through the array, with one less comparison needed to select the appropriate entry on each successive pass. We may bury the internal logic of each pass in a call to the function **findChampionIndex** discussed in Section 1.2, thereby enhancing the function's degree of abstraction.

```
//-----------------------------------------------------------------
// Interface for function sort
// GIVEN:   a -- an array of ElementType values;
//          numvals -- number of values in the array,
//                     assume numvals >= 0
//          precedes -- a function to compare ElementType values:
//            GIVEN:   x and y -- values to compare
//            RETURN as value of function:
//                     TRUE     if x precedes y;
//                     FALSE    if x and y are equal, or
//                              if y precedes x
// RETURN:  The array a with its values arranged in order by the precedes
//          relation
// RETURN as value of function: void

template <class ElementType>
void sort(ElementType a[], int numvals,
          BOOLEAN (*precedes)(const ElementType &x, const ElementType &y))
```

successor. If the two values are not in the proper order relative to each other, they are swapped. The array will be completely sorted when no swaps are made during a pass. Write a C++ version of this algorithm. Then subject your algorithm to a big-O efficiency analysis as we did for the selection and insertion sorts. Compare bubble sort's best, average, and worst case performance to those of the other two algorithms.

5. Do a big-O analysis for those statements inside each of the following nested loop constructs.

a.
```
for (k = 0; k < n; ++k)
  for (j = 6; j < n; ++j)
  {
    ...
  }
```

b.
```
for (k = 0; k < n; ++k)
{
  j = n;
  while (j > 0)
  {
    ...
    j /= 2;      // integer division
  }
}
```

c.
```
k = 1;
do
{
  j = 1;
  do
  {
    ...
    j *= 2;
  }
  while (j < n);
  ++k;
}
while (k < n);
```

6. An algorithm has an efficiency $O(n^2|\sin(n)|)$. Is it any better than $O(n^2)$ for large integer n? Explain why, or why not.

7. Suppose that each of the following expressions represents the number of logical operations in an algorithm as a function of n, the number of data items being manipulated. For each expression, determine the dominant term and then classify the algorithm in big-O terms.

a. $n^3 + n^2 \log_2 n + n^3 \log_2 n$

b. $n + 4n^2 + 4^n$

c. $48n^4 + 16n^2 + \log_8 n + 2n$

8. Consider the following nested loop construct. Categorize its efficiency in terms of the variable n using big-O notation. Suppose the statements represented by the ellipsis require four main memory accesses (each requiring one microsecond) and two disk file accesses (each requiring one millisecond). Express in milliseconds the amount of time this construct would require to execute if n were 1000.

```
x = 1;
do
{
  y = n;
  while (y > 0)
  {
    ...
    --y;
  }
  x *= 2;
}
while (x < n*n);
```

9. A sorting algorithm is called *stable* if it does not change the relative order of array elements that are equal. For example, a stable sorting algorithm will not place 13_1 after 13_2 in the array

18	13	6	12	13	9

Are insertion sort and selection sort stable? If not, provide an example of an array with at least two equal elements that change in their order relative to each other.

1.3 Algorithm Efficiency—The Search Problem

In the previous section, we introduced big-O notation and used it to analyze two algorithmic solutions to the sorting problem. In this section we examine a general strategy for solving the search problem—finding a particular value in an ordered array. As in Section 1.2, we may state this problem more formally as a C++ interface expressed in precondition/postcondition form.

```
//-----------------------------------------------------------------
// Interface for function search:
// GIVEN:    a -- an array of values of type ElementType arranged
//                in order by precedes function;
//           numvals -- number of values in the array,
//                      assume numvals >= 0
//           target -- value being sought in the array;
//           precedes -- a function to compare ElementType values:
//             GIVEN:    x and y -- values to compare
//             RETURN as value of function:
//                       TRUE       if x precedes y,
//                       FALSE      if x and y are equal, or
//                                  if y precedes x
//           split -- a function to determine how the array
//                    should be split
//             GIVEN: lo and hi, integer values with
//                    0 <= lo <= hi <= numvals-1 for numvals > 0
//             RETURN as value of function:
//                       An integer between 0 and numvals-1 inclusive
// RETURN: place -- If the return value is TRUE, place contains the
//                  index position of target in a; otherwise,
//                  place is unreliable.
// RETURN as value of function:
//           TRUE    if target can be found in the array a;
//           FALSE   otherwise

template <class ElementType>
BOOLEAN search(ElementType a[], int numvals, const ElementType &target,
               BOOLEAN (*precedes)(const ElementType &x, const ElementType &y),
               int (*split)(int lo, int hi), int &place)
```

The **precedes** and **split** parameters in the **search** interface require some explanation. If we do not assume that the array being searched is ordered, these two parameters are not needed. However, we prefer to work under that assumption because it allows us to develop search strategies that are more efficient than a sequential search strategy. Recall that such a strategy merely examines successive locations (starting with the first) until it finds the **target** or advances beyond index **numvals** − 1.

By assuming an array that is ordered according to the **precedes** relationship, we may develop a *divide-and-conquer* search strategy similar to that used when searching for a name in a phone book. We do not search for "Smith, Sam" in a phone book by starting with the first page and examining successive pages. Rather, because the phone book is alphabetically ordered, we make a guess that the location of "Smith, Sam" will be roughly three-quarters of the way into the phone book. We open to the page designated by this guess. If "Smith, Sam" appears on this page, we are done. If the names on this page precede "Smith, Sam," we iterate this process with the portion of the phone book that follows. If the names on the page follow "Smith, Sam," we iterate with the front portion of the book. This divide-and-conquer strategy is highlighted in Figure 1.9. The **split** function parameter that appears in our **search** function interface represents the method on which our divide-and-conquer guess is based.

In the code for **search** you will see that we use the C++ operator "==" to test for equality as the way to determine whether **target** appears in the array **a** or not. While this may be appropriate for certain data types (especially **int**, **char**, or **enum**)