CHAPTER 21 Computational complexity

We interrupt our study of object-oriented design for a brief, informal look at the important topic of algorithmic efficiency. Measuring the efficiency of a program is in general a complicated issue. In this chapter, we try to get a handle on determining how the time required for a program to compute a solution depends on the algorithm chosen. We introduce a commonly used measure, computational complexity, that allows us to compare the efficiency of algorithms in a very fundamental way. We will apply this measure to algorithms developed in subsequent chapters.

After defining the measure, we apply it to some simple algorithms, and conclude with an example that develops three different methods for solving the same problem. These algorithms have dramatically different complexities, and should give us considerable insight into the effects of algorithm choice on program execution time.

21.1 Measuring program efficiency

When we speak of the efficiency or cost of a program, there are many issues we might be considering. We might be concerned with the time it takes for the program to produce a result, the amount of memory or other system resources required by the program, the programmer effort required to maintain the program, and so on. Here we specifically limit our attention to the execution time required for a program to solve a problem.

There are, of course, many factors that affect execution time:

- the hardware—the type and speed of the processor, size and speed of memory, disk transfer rate, network speed, *etc.*
- the operating system;
- the system environment—number of active processes, *etc.*

- the programming language and compiler;
- the run-time system or interpreter;
- the algorithms that constitute the program;
- the data on which the program is run.

We want to isolate the effects on execution time due to algorithm and data, ignoring hardware speed, operating system, *etc.* To this end, we define a measure of program efficiency called *computational complexity* or *time complexity.* This is a coarse but extremely important gauge of a program's execution time. While the definition may seem less than intuitive at first, the concept has proven to be of considerable benefit in comparing and evaluating problem solutions.

We will not attempt a complete or rigorous treatment. We give some fundamental definitions, and apply them rather informally to a few examples. Our intention is simply to gain some familiarity with the notions so that we can infer the time-cost of algorithms examined in subsequent chapters.

21.2 Time complexity for a method

Since complexity is an algorithmic notion, we limit our attention to determining the complexity of a method. We want to determine the time-cost of a method that solves a particular problem.

First of all, we assume that each instance of a problem has some inherent size, and that the execution time required by a method to solve a problem instance depends on its size. For example, if we are considering one of the sort algorithms of Chapter 13, the problem size is the length of the list to be sorted. We expect that the time required to sort the list will depend on the length of the list. If we have a method that determines whether or not a particular integer is prime, the problem size is the size of the integer. We expect the method to take longer to determine whether 9907 is prime than to determine whether 7 is prime.

Of course, a method may require a different amount of time to solve two different problem instances of the same size. For example, a sort method might be able to sort a list of 100 items that is already nearly sorted more quickly than it can sort a list of 100 items that is thoroughly scrambled. Though we are sometimes interested in an "average" time required by a method to solve a problem of a given size, for the most part we want to know the worst case behavior of an algorithm. That is, we want the maximum time-cost over all problems of a given size.

It would seem natural to compute the time-cost of a method in units of time, such as seconds, microseconds, *etc.* But the actual clock time required by a

method clearly depends on a number of factors other than the algorithm. Rather than using units of time, we'll measure the time-cost by counting the number of primitive steps the algorithm performs in solving the problem. This measure is easier to calculate and will be perfectly adequate for comparing the relative time cost of various algorithms. In fact, we don't even care to specify exactly what a primitive step is, except to say that it takes a fixed amount of time to perform regardless of the problem. For instance, we can consider it to be a single instruction executed by the Java interpreter.

We now define the *time-cost* or *time complexity* of a method M to be a non-decreasing function t_M from the natural numbers $\mathbf{N}$ (*i.e.*, $\{0, 1, 2, \ldots\}$) to the positive reals $\mathbf{R}^+$

$$t_M : \mathbf{N} \to \mathbf{R}^+$$

such that $t_M(n)$ is the maximum number of steps for method M to solve a problem of size n, over all problems of size n. For example, if M is the selection sort of Chapter 13, $t_M(100)$ is the maximum number of steps required by the method to sort a list of 100 elements.

21.3 Comparing method costs

We want to compare different algorithms for solving a problem according to their time efficiency. This amounts to comparing their corresponding cost functions. How should we do this? The following approach to comparing cost functions has proven most useful for our purposes. The definition may not seem entirely intuitive, but with a little experience it will become familiar.

Note the relation $\sqsubseteq$ defined in this section is not standard notation. It is given here to simplify the definitions of the standard class constructors O, Ω, and Θ given in the next section.

Let f and g be non-decreasing functions from the natural numbers to the positive reals

$$f, g : \mathbf{N} \to \mathbf{R}^+$$

We say f is O-*dominated by* g, and write $f \sqsubseteq g$, provided

there is a positive integer n_0, and
there is a positive real c, such that
for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$.

This says that f is O-dominated by g if some multiple of g dominates f for all large integers, and is illustrated graphically in Figure 21.1.

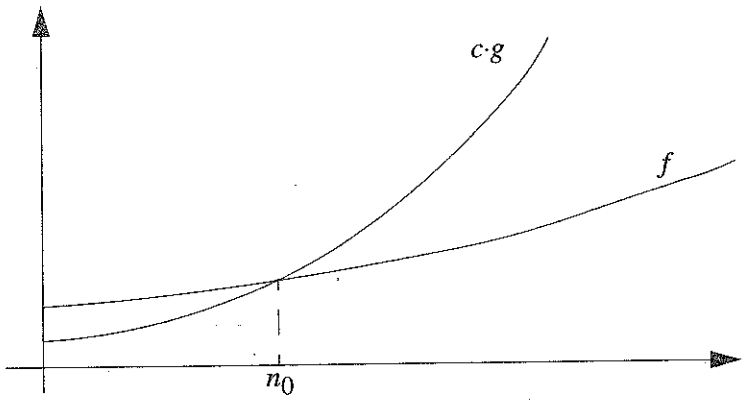


Figure 21.1 **f is O-dominated by *g*.**

Now let's see how to use this definition to compare methods. Suppose *M* and *N* are two different sort methods (selection sort and bubble sort, for instance), with time-cost functions t_M and t_N respectively. What does it mean to show that $t_M \sqsubseteq t_N$? First, we can throw out as many specific problems as we want by choosing n_0 sufficiently large. For instance, if we choose n_0 to be 100, we only look at lists that have more than 100 elements. It doesn't matter how they compare on smaller lists. Next, we can "slow down" *N* as much as we want by choosing an appropriate value of *c*. For example, if we choose *c* to be 10, we compare t_M with $10 \cdot t_N$. We can imagine running method *N* on a machine that is 10 times slower than the machine *M* is running on. Under these conditions, *M* must perform as well as *N*.

Note that we need only demonstrate that the third line of the definition holds for *one* n_0 and *one* *c*. But $t_M(n)$ must be less than or equal to $c \cdot t_N(n)$ for *all* $n \geq n_0$.

This may seem like a very weak condition. In fact, it may seem that any method can be made to run faster than any other if we slow the second down enough! To get a better handle on the definition, consider what it means if t_M is *not* O-dominated by t_M. In this case, no matter how much we slow down the machine running *N*, we can always find arbitrarily large problems for which *N* will run faster than *M*. Clearly, *M* is not as efficient as *N* in some very fundamental way.

We make a few observations about the relation $\sqsubseteq$, without proof. In the following, we assume f, g, h, f', and g' are non-decreasing functions from the natural numbers to the positive reals.

- The relation is *reflexive*: that is, for any function f, $f \sqsubseteq f$.
- The relation is *transitive*: that is, for functions f, g, and h, $f \sqsubseteq g$ and $g \sqsubseteq h$ imply $f \sqsubseteq h$.

- The relation is *not anti-symmetric*: there are functions f and g such that $f \sqsubseteq g$ and $g \sqsubseteq f$, but $f \neq g$.
- The relation is *not total*: there are functions f and g such that neither $f \sqsubseteq g$ nor $g \sqsubseteq f$.

If $f \sqsubseteq g$ and $g \sqsubseteq f$, we say f and g are said to *have the same magnitude*, and we write $f \equiv g$.

If $f \sqsubseteq g$ but it is not the case that $g \sqsubseteq f$, we write $f \sqsubset g$.

(If *exp(n)* is an expression whose value depends on n, we use *exp(n)* to denote the function f defined by $f(n) = exp(n)$. For instance, we denote the function f defined by $f(n) = n^2$ simply as n^2, and denote the constant function $f(n) = 1$ as 1.)

The *pointwise sum*, $f+g$, *pointwise product*, $f \cdot g$, and *pointwise max*, $max(f, g)$ of two functions f and g are defined as follows.

$$[f + g](n) = f(n) + g(n))$$
$$[f \cdot g](n) = f(n) \cdot g(n)$$
$$[max(f, g)](n) = max(f(n), g(n))$$

The following items are easy to show.

<21.1> If c_1 and c_2 are positive real constants, then $f \equiv c_1 f + c_2$.

<21.2> If f is a polynomial function, $f(n) = c_k n^k + c_{k-1} n^{k-1} + \ldots + c_0$, where $c_k > 0$, then $f \equiv n^k$.

<21.3> If $f \sqsubseteq g$ and $f' \sqsubseteq g'$, then $f \cdot f' \sqsubseteq g \cdot g'$.

<21.4> If $f \sqsubseteq g$ and $f' \sqsubseteq g'$, then $f + f' \sqsubseteq max(g, g')$. Furthermore, $f + g \equiv max(f, g)$.

<21.5> If a and b are > 1, $\log_a n \equiv \log_b n$.[1]

<21.6> $1 \sqsubset \log_a n \sqsubset n \sqsubset n \log_a n \sqsubset n^2 \sqsubset n^3 \sqsubset n^4 \sqsubset \ldots \sqsubset 2^n$.

21.4 Complexity classes

Suppose f is a non-decreasing function from the natural numbers to the positive reals. We define the following sets of functions:

$O(f)$ is the set of functions that are O-dominated by f. That is,

$$O(f) = \{\ g \mid g \sqsubseteq f\ \}.$$

$\Omega(f)$ is the set of functions that O-dominate f. That is,

$$\Omega(f) = \{\ g \mid f \sqsubseteq g\ \}.$$

1. Of course, we fudge a bit with logarithmic functions and eliminate 0 from the domain.

$\Theta(f)$ is the set of functions that have the same magnitude as f. That is,

$$\Theta(f) = \{\ g \mid g \equiv f\ \} = \{\ g \mid f \sqsubseteq g \text{ and } g \sqsubseteq f\ \} = O(f) \cap \Omega(f).$$

$O(f)$ is pronounced "oh of f" or "big oh of f." $\Omega(f)$ is "omega of f," and $\Theta(f)$ is "theta of f."

If M is a method with time-cost t_M, then we say "M is in $O(f)$," or simply "M is $O(f)$," if $t_M \in O(f)$. Similar statements are made for Ω and Θ.

If M is a method with complexity $t_M(n) = c$, for some constant c, then M is $\Theta(1)$ and we say M is *constant*. If M is $\Theta(n)$, we say that M is *linear*. If M is $\Theta(n^2)$, M is *quadratic*. If M is not $O(n^k)$ for any k, then M is *exponential*. Most exponential methods encountered in practice are $\Omega(2^n)$. If a method is linear, we expect the run-time to increase proportionally with the size of the input; if it's quadratic, we expect run-time to increase as the square of the input; and so on.

In subsequent chapters, we categorize the methods in terms of the hierarchy shown in <21.6>. That is, we will categorize methods as being $\Theta(1)$, or $\Theta(\log n)$, or $\Theta(n)$, *etc.* The table below shows the values of a few complexity functions for several values of n.

To get some idea of the implications, Table 21.1 gives the number of microseconds in several time units. If we are counting steps that take a microsecond, a method with complexity n^3 will take about 17 minutes for a problem of size 1,000, and over 300 centuries for a problem of size 1,000,000. The implications of this are clear. If a method is to be used on large cases, it almost surely needs to be quadratic or better. A method that is exponential can be applied in practice only to very small problem instances.

Table 21.1 **Values of Some Complexity Functions**

n	$\log_2 n$	$n \cdot \log_2 n$	n^2	n^3	2^n
10	3.32	33.2	100	1,000	1,024
100	6.64	664	10,000	1,000,000	1.27×10^{30}
1,000	9.97	9,966	1,000,000	10^9	1.07×10^{301}
1,000,000	19.93	19,931,569	10^{12}	10^{18}	10^{301030}

21.4.1 Complexity of a problem

To this point, our discussion has concerned the complexity of a *method*. Sometimes we are interested in the complexity of a *problem*. That is, we want to know the complexity of the best possible method for solving the problem. If we say that the complexity of a problem is t, we mean that every method for solving the problem is $\Omega(t)$, and some method is $\Theta(t)$. That is, the best possible method for solving

Table 21.2 **Microseconds per Unit of Time.**

Unit	Number of μsecs
second	10^6
minute	6×10^7
hour	3.6×10^9
day	8.64×10^{10}
year	3.15×10^{13}
century	3.15×10^{15}

the problem is $\Theta(t)$. For instance, it is known that the complexity of the general sorting problem using comparison is $n\cdot\log n$. That is, there are methods that can sort any list of length n with $n\cdot\log n$ time-cost, and it is not possible to write a method capable of sorting every list of length n and having time-cost better than $n\cdot\log n$.

Some problems are known to be exponential, and are called *intractable*. (Some problems have no algorithmic solution at all, and are called *unsolvable*.)

Determining the complexity of a method involves analyzing the method and is often straightforward. Determining the complexity of a problem is generally much more difficult, since it involves proving something about all possible methods for solving the problem.

21.4.2 Determining the complexity of a method

We now sketch a few ideas on analyzing methods to determine their time-cost. As we have said, we do not intend to give a thorough, formal presentation.

Methods encountered in practice generally have a cost that fits into the hierarchy of <21.6>. Since complexity is a function of problem size, any method that in the worst case must examine all of its data is at least linear. For instance, a method that finds the largest element on an arbitrary list must examine each element of the list. If the list has n elements, at least n steps are required to examine the elements. A method with constant time complexity, $\Theta(1)$, needs to use only a fixed amount of its data. For instance, a method to remove the first element of a list might require the same number of steps to complete regardless of the length of the list. Methods with complexity better than linear typically require that the problem data have some specific organization or structure. For instance, the binary search algorithm seen in Section 13.3 is $\Theta(\log n)$, but requires the list to be sorted. It is this property of the list that enables the method to find an item without examining each list element.

To determine the time-cost of a method, we must count the steps performed in the worst case. Considering the possible kinds of statements that can comprise a method, we observe the following.

- A simple, elementary statement, like a simple assignment, requires a constant number of steps. Similarly, evaluation of a simple expression requires a constant number of steps.
- To determine the number of steps required by a statement that invokes a method, we must determine the number of steps required by the method that is invoked.
- The worst case number of steps required by a conditional such as an *if-then-else* is the maximum of the number of steps required by each alternative. For instance, if statements `A` and `B` have time-costs t_A and t_B respectively, then

  ```
  if (simpleCondition)
     A
  else
     B
  ```

  has time-cost $max(t_A, t_B)$.
- To determine the number of steps required by a sequence of statements, we sum the number of steps required by each statement. Here, **<21.4>** simplifies our task considerably. For example, if statements `A`, `B`, and `C` have time-costs t_A, t_B, and t_C respectively, then

  ```
  A; B; C;
  ```

  has time-cost $t_A + t_B + t_C \equiv max(t_A, t_B, t_C)$.
- To determine the time-cost of a loop, we determine the number of steps required by the loop body and multiply by the number of times the loop body is executed.
- To determine the number of steps required by a recursive call, we must determine the depth of the recursion.

A little consideration of the above points should make it clear that loops and recursive calls are the syntactic elements likely to determine the complexity of a method.

An example containing a loop

As a first example, consider the method to compute the average final grade of a list of *Students*, as seen in Section 12.4. The problem size here is rather clearly the length of the argument list, `students`.

```
public double average (StudentList students) {
    int i;
    int sum;
    int count;
    count = students.size();
    sum = 0;
    i = 0;
    while (i < count) {
        sum = sum + students.get(i).finalExam();
        i = i+1;
    }
    return (double)sum / (double)count;
}
```

If we assume that the method `size` requires constant time independent of the length of the list, then all of the statements in the method except for the *while-loop* contribute only a constant amount to the time-cost. Clearly the complexity of the method depends on the loop. If we further assume that the *Student* method `finalExam` and the *StudentList* method `get` are constant, then the body of the loop requires constant time to execute, and we need only count the number of times the body of the loop is performed. But the body of the loop is performed once for each element of the list. If the length of the list `students` is n, the loop body is done n times. (The test `i < count` is actually done $n+1$ times, but this does not change the complexity of the method.) Thus the method is linear, $\Theta(n)$.

Now suppose that the method `get` is linear in its argument: in particular, suppose that `get(i)` requires ic_1+c_0 steps to complete, where c_1 and c_0 are constants. The body of the loop is no longer constant, and we need to do a little more work to determine the complexity of the method.

We can list the number of steps required by `get(i)` for each value of `i` as follows:

value of `i`:	0	1	2	...	$n-1$
steps of `get(i)`:	c_0	c_1+c_0	$2c_1+c_0$	...	$(n-1)c_1+c_0$

We sum the n terms to determine the total number of steps required by the invocations of `get`:

$$\sum_{i=0}^{n-1}(ic_1+c_0) = nc_0 + c_1\sum_{i=1}^{n-1} i$$

It is easy to see by induction that

$$\sum_{i=1}^{k} i = \frac{k(k+1)}{2} = \frac{1}{2}(k^2+k) \qquad (1)$$

Thus the total number of steps required by invocations of `get` is

$$nc_0 + \frac{1}{2}c_1(n^2 - n)$$

and the method is quadratic.

To simplify calculations, we assume for the remainder of this chapter that *List* methods such as `size` and `get` require constant time.

An example containing nested loops

As a second example we consider a method containing a pair of nested loops: a method that will determine if a List contains duplicate elements. (This is similar to the method that removed duplicate elements from a list presented in Section 12.5.1.) We can probably guess that a method containing nested loops, each of which depends on problem size, will be at least quadratic. The method is defined as follows:

```
boolean hasDuplicates (List list) {
   int i;
   int j;
   int n;
   boolean found;
   n = list.size();
   found = false;
   for (i = 0; i < n-1 && !found; i = i+1)
      for (j = i+1; j < n && !found; j = j+1)
         found = list.get(i).equals(list.get(j));
   return found;
}
```

We consider the case where the list has no duplicates. This case requires the longest execution time, since the method must examine every pair of elements.

The outer loop is performed $n - 1$ times, with i successively taking on the values 0, 1, 2, ..., $n-2$. The number of iterations of the inner loop depends on i:

value of i:	0	1	2	...	$n-3$	$n-2$
iterations of the inner loop	$n-1$	$n-2$	$n-3$	...	2	1

To compute the number of times the assignment

```
found = list.get(i).equals(list.get(j));
```

is done, we must compute the sum:

$$1 + 2 + \ldots + (n-3) + (n-2) + (n-1).$$

From equation *(1)*, this is to equal to $\frac{n^2 - n}{2}$. Thus the method is $\Theta(n^2)$.

21.5 An example: three methods

In this section, we analyze three different methods for solving the same problem. This will give us a good idea of what is involved in analyzing both iterative and recursive methods. The problem is based on [Weiss 92].

Define a *sublist* of a list l to be a list containing 0 or more contiguous elements of l. For instance, the list (1, 2, 3) has seven sublists: the empty sublist; three sublists of length 1: (1), (2), and (3); two sublists of length 2: (1, 2) and (2, 3); one sublist of length 3: (1, 2, 3).

The problem we want to solve is this: given a list of integers, find the maximum sum of the elements of a sublist. For example, the following lists have sublists with maximum sum as shown.

list	*sublist with max sum*	*max sum*
(-2, 4, -3, 5, 3, -5, 1)	(4, -3, 5, 3)	9
(2, 4, 5)	(2, 4, 5)	11
(-1, -2, 3)	(3)	3
(-1, -2, -3)	()	0

Note that the sum of the elements of the empty list is 0. Thus if a list contains only negative integers, the empty sublist is the sublist with maximum sum. Furthermore, any method to solve this problem must be at least linear, since each element of the list must be examined.

To simplify the code, assume that *IntegerList* has a method `getInt`, where

```
getInt(i) == ((Integer)get(i)).intValue()
```

We can then specify the method we want to write as follows:

```
int maxSublistSum (IntegerList list)
```

The maximum sum of a sublist of the given list.

if `list.getInt(i) > 0` for some `i`, `0 <= i < list.size()`, then

max { `list.getInt(i) + ... + list.getInt(j)` |
`0 <= i <= j < list.size()` }

else

0

The problem is admittedly artificial. But it admits several easy to understand and quite distinct solutions. We consider three methods for solving the problem, each with different time-cost.

21.5.1 A naive method

The first method we consider uses a straightforward, brute force approach. Simply sum the elements of each possible sublist, and remember the maximum. The method is given in Listing 21.1.

Listing 21.1 **A naive approach**

```
int maxSublistSum (IntegerList list) {
   int n = list.size();
   int maxSum;// max of all values of sum
   int sum;// list.getInt(i) + … + list.getInt(j)
   int i;// starting index of sublist being summed
   int j;// ending index of sublist being summed
   int k;// index of summand, i <= k <= j

   maxSum = 0;
   for (i = 0; i < n; i = i+1)
      for (j = i; j < n; j = j+1) {
         // compute list.getInt(i) + … + list.getInt(j)
         sum = 0;
         for (k = i; k <= j; k = k+1)
            sum = sum + list.getInt(k);
         if (sum > maxSum)
            maxSum = sum;
      }

   return maxSum;
}
```

The outer index `i` determines the starting point of the sublist, and the second index `j` determines the ending point. The inner loop then sums the elements `list.getInt(i) + … + list.itmeAt(j)`.

Since we have three nested loops, each of which depends on the length of the list, we expect the method to be $\Theta(n^3)$. We'll do the arithmetic this once to verify our conjecture.

The body of the inner loop (k-loop) is executed the most often. To determine the complexity of the method, we count the number of times this is done.

The outer loop is done n times, with `i` taking on the values 0, 2, …, $n - 1$. For each value of `i`, `j` iterates through the values `i` through $n - 1$. For each value of `i`

and `j`, `k` iterates from `i` through `j`. For each value of `k`, the body of the inner loop is done once. Thus we must compute the sum:

$$\sum_{i=0}^{n-1}\sum_{j=i}^{n-1}\sum_{k=i}^{j} 1$$

This requires just a little arithmetic, given equation *(1)* on page 543 and the following:

(2) $$\sum_{i=1}^{k} i^2 = \frac{k(k+1)(2k+1)}{6}$$

The inner summand

$$\sum_{k=i}^{j} 1$$

obviously equals the number of integers k such that $i \le k \le j$; *i.e.*, equals $j-i+1$. Now moving to the middle sum,

$$\sum_{j=i}^{n-1}\sum_{k=i}^{j} 1 = \sum_{j=i}^{n-1} (j-i+1);$$

substituting $j = i, i+1, \ldots, n-1$, we get

$$= 1 + 2 + \ldots + (n-i)$$

$$= \frac{(n-i+1)(n-i)}{2}, \text{ by equation (1) above;}$$

$$= \frac{1}{2}i^2 - \left(n+\frac{1}{2}\right)i + \frac{1}{2}(n^2+n), \text{ after expansion.}$$

Finally, using equations *(1)* and *(2)* and a little algebra, we compute:

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

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

$$= \frac{1}{2}\left(\frac{(n^2-n)(2n-1)}{6}\right) - \left(n+\frac{1}{2}\right)\left(\frac{n(n-1)}{2}\right) + \frac{1}{2}(n^2+n)n$$

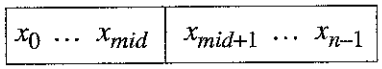
$$= \frac{1}{6}n^3 + \frac{1}{2}n^2 + \frac{1}{3}n$$

Is this method satisfactory? It is certainly easy to understand and to write correctly. If we intend only to use it a few times on relatively small lists, it is adequate. It is not critical to spend time trying to find a better approach. But if the method is part of critical operating system code or if we want to use it for large lists, we need something better.

It is relatively straightforward to improve this method to $\Theta(n^2)$ by summing in the second loop and eliminating the third. We'll leave this as an exercise, and look at two rather different approaches.

21.5.2 A recursive method

The next method we consider uses a recursive "divide and conquer" approach. Lists of length 0 and 1 are obvious base cases. If we have a longer list, we divide it in half and consider the possible cases. For the discussion, assume the list is $(x_0, \ldots, x_{n-1})$ with left half $(x_0, \ldots, x_{mid})$ and right half $(x_{mid+1}, \ldots, x_{n-1})$:



There are three possible cases:

1. the sublist with the max sum is in the left half of the list, *i.e.*, in $(x_0, \ldots, x_{mid})$;
2. the sublist with the max sum is in the right half of the list: *i.e.*, in $(x_{mid+1}, \ldots, x_{n-1})$;
3. the sublist with max sum "overlaps" the middle of the list: in particular, it includes both x_{mid} and x_{mid+1}.

As is commonly the case with recursive methods, we write an auxiliary method that takes as arguments the starting and ending indexes of the portion of the list to consider:

```
private int maxSublistSum (IntegerList list,
   int first, int last)
```

The maximum sum of a sublist in the given range. That is, the maximum sum of a sublist of `(list.get(first), ..., list.get(last))`.

require:

```
0 <= first <= last < list.size()
```

The principle method handles the empty list explicitly, and otherwise calls the auxiliary method:

```
int maxSublistSum (IntegerList list) {
   if (list.size() == 0)
      return 0;
```

```
        else
            return maxSublistSum(list,0,list.size()-1);
    }
```

The auxiliary method cannot be called with an empty range, since `first <= last` is required. The base case of a single element is explicitly handled. If there is more than one element, the maximum sums of sublists in the left and right half are recursively obtained, and the maximum sum of a sublist that overlaps the middle is computed.

To determine the maximum sum of a sublist overlapping the middle, the maximum sum of a sublist ending with x_{mid}, and the maximum sum of a sublist beginning with x_{mid+1} are computed and then added.

The largest of the three values—maximum sum of sublist in the left half, maximum sum of sublist in the right half, and maximum sum of a sublist that overlaps both halves—is then returned. The method is shown in Listing 21.2.

Listing 21.2 A divide-and-conquer approach

```
/*
 * require:
 *   0 <= first <= last < list.size()
 */
private int maxSublistSum (IntegerList list, int first,
    int last) {
    int maxSum;// return value
    int mid = (first + last)/2;
          // index of middle element;
          // first <= mid <= last
    int maxLeftSum;
          // maxSublistSum of left half of list: i.e., of
          // (list.get(first), …, list.get(mid))
    int maxRightSum;
          // maxSublistSum of right half of l: i.e., of
          // (list.get(mid+1), …, list.get(last))
    int maxMidSum;
          // max sum of a sublist containing both
          // list.get(mid) and list.get(mid+1)
    int maxEndMid;
          // max sum of a sublist ending with
          // list.get(mid)
    int maxStartMid;
          // max sum of a sublist starting with
          // list.get(mid+1)
    int endMidSum;
```

Listing 21.2 **A divide-and-conquer approach** ***(continued)***

```
int startMidSum;
      // list.getInt(mid+1) + … + list.getInt(i)
int i;// index for summing

if (first == last)
   if (list.getInt(first) < 0)
      return 0;
   else
      return list.getInt(first);
else {
   // first <= mid < last
   maxLeftSum = maxSubList(list, first, mid);
   maxRightSum = maxSubList(list, mid+1, last);

   maxEndMid = list.getInt(mid);
   endMidSum = list.getInt(mid);
   for (i = mid-1; i >= first; i = i-1) {
      endMidSum = endMidSum + list.getInt(i);
      if (endMidSum > maxEndMid)
         maxEndMid = endMidSum;
   }

   maxStartMid = list.getInt(mid+1);
   startMidSum = list.getInt(mid+1);
   for (i = mid+2; i <= last; i = i+1) {
      startMidSum = startMidSum + list.getInt(i);
      if (startMidSum > maxStartMid)
         maxStartMid = startMidSum;
   }

   maxMidSum = maxEndMid + maxStartMid;

   maxSum = maxMidSum;
   if (maxLeftSum > maxSum)
      maxSum = maxLeftSum;
   if (maxRightSum > maxSum)
      maxSum = maxRightSum;

   return maxSum;
}
```

This method is much more involved than the previous, and care must be taken to ensure correctness. For instance, we must make sure that preconditions are satisfied for each of the recursive calls.

The analysis of this recursive method is a little different from those that we've looked at previously. We give a very informal analysis.

The complexity of the method is clearly the complexity of the auxiliary method. Let t be the time-cost function of the recursive auxiliary method. If there is only one element to consider, the method requires a fixed amount of time, which we'll call c_0.

(3) $$t(1) = c_0$$

If there are $n > 1$ elements to consider, the method twice recursively calls itself, with half as many elements each time:

$$t(n) = 2 \cdot t(n/2) + \ldots$$

It then performs two loops, iterating each essentially $n/2$ times. If we let c_1 be the time required for an iteration, we have:

$$t(n) = 2 \cdot t(n/2) + c_1 \cdot n + \ldots$$

Finally, there are a number of other steps requiring constant time independent of n. Call this time c_2:

(4) $$t(n) = 2 \cdot t(n/2) + c_1 \cdot n + c_2$$

An equation like *(4)* that describes a function at one point in terms of its value at another point is called a *recurrence relation*. Recurrence relations are often the key to analyzing recursive methods.

If we substitute $n/2$ for n in *(4)*, we get:

(5) $$t(n/2) = 2 \cdot t(n/4) + c_1 \cdot (n/2) + c_2$$

Substituting the right side of *(5)* for $t(n/2)$ in *(4)*, we have:

(6) $$\begin{aligned} t(n) &= 2 \cdot (2 \cdot t(n/4) + c_1 \cdot (n/2) + c_2) + c_1 \cdot n + c_2 \\ &= 4 \cdot t(n/4) + 2 \cdot c_1 \cdot n + 3 \cdot c_2 \end{aligned}$$

Repeating the process for $n/4$ gives:

$$t(n/4) = 2 \cdot t(n/8) + c_1 \cdot (n/4) + c_2$$

and from *(6)*:

$$t(n) = 8 \cdot t(n/8) + 3 \cdot c_1 \cdot n + 7 \cdot c_2$$

It is not hard to show that this generalizes to

$$t(n) = 2^k \cdot t(n/2^k) + k \cdot c_1 \cdot n + (2^k - 1) \cdot c_2$$

We can solve the equation if we can determine $t(n/2^k)$ for some particular k. But we know

$$t(1) = c_0,$$

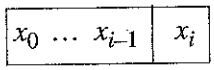
and $n/2^k = 1$ when $n = 2^k$, or when $k = \log_2 n$. Thus for $k = \log_2 n$, $n/2^k = 1$ and

$$\begin{aligned} t(n) &= 2^k \cdot t(1) + k \cdot c_1 \cdot n + (2^k - 1) \cdot c_2 \\ &= 2^{\log_2 n} \cdot c_0 + \log_2 n \cdot c_1 \cdot n + (2^{\log_2 n} - 1) \cdot c_2 \\ &= n \cdot c_0 + c_1 \cdot n \cdot \log_2 n + (n - 1) \cdot c_2 \end{aligned}$$

and the complexity of the method is $\Theta(n \cdot \log_2 n)$. This is a considerable improvement over the previous method, but we can do even better.

21.5.3 An iterative method

Finally, we consider an iterative approach. Suppose we have examined the first i elements of the list, $(x_0, \ldots, x_{i-1})$, and have found the sublist with maximum sum in the first i elements. We now look at the $i+1$st element.



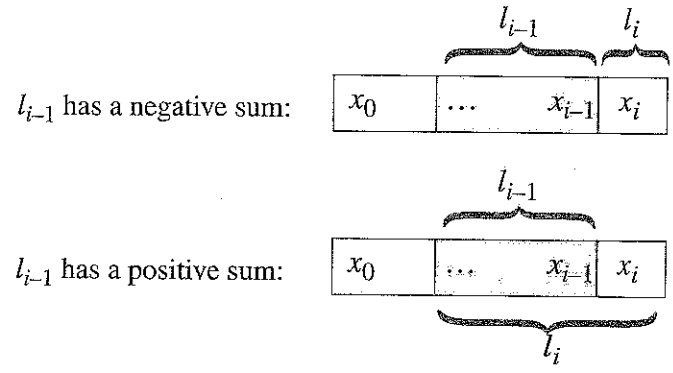
There are two possibilities for the maximum sum sublist of the first $i+1$ elements:

1. it doesn't include x_i, in which case it is the same as the maximum sum sublist of the first i elements; or
2. it includes x_i.

If we look at the maximum sum sublist ending with x_i, there are again two cases:

1. it consists only of x_i; or
2. it includes x_{i-1}, in which case it is x_i appended to the maximum sum sublist ending with x_{i-1}.

It is, in fact, rather easy to distinguish these two cases. Let l_{i-1} denote the maximum sum sublist ending with x_{i-1}, l_i the maximum sum sublist ending with x_i. If l_{i-1} has a negative sum, then l_i consists of just x_i. Otherwise, l_i is l_{i-1} with x_i appended.



This suggests that we write a method that iterates through the list, keeping track of the maximum sublist sum, and the maximum sum of a sublist ending with the current element. The method is shown in Listing 21.3. The method is $\Theta(n)$, which is the best we can hope for. It is also rather straightforward and easy to understand.

Listing 21.3 An iterative approach

```
int maxSublistSum (IntegerList list) {
   int n = list.size();
   int i;// index for the list
   int maxSum;// max sum of a sublist of
         // (list.get(0), …, list.get(i-1))
   int maxTail;// max sum of a sublist ending with
         // list.get(i-1)

   maxSum = 0;
   maxTail = 0;
   for (i = 0; i < n; i = i+1) {
      if (maxTail > 0)
         maxTail = maxTail + list.getInt(i);
      else
         maxTail = list.getInt(i);
      if (maxTail > maxSum)
         maxSum = maxTail;
   }

   return maxSum;
}
```

21.6 Summary

In this chapter, we considered a fundamental measure of algorithm efficiency called time complexity. We will use this measure to evaluate algorithms introduced in subsequent chapters.

The time-cost of a method is a function that specifies the maximum number of steps required by the method to solve a problem of a specified size. Time-cost functions can be compared with a relation O-*dominated by*. If f and g are time-cost functions, f is O-dominated by g when there exist positive numbers n_0 and c such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$. If f is O-dominated by g, then $f \in O(g)$ and $g \in \Omega(f)$. If $f \in O(g)$ and $f \in \Omega(g)$, then f and g have the same magnitude, and $f \in \Theta(g)$. Comparing the time-cost of a method to a collection of standard functions enables us to categorize efficiency of the method in a very fundamental way.

We saw how to compute the time-cost for some simple examples, and noted that loops and recursive invocations played a principle role in determining the complexity of a method. Finally, we did a case study in which three different methods were presented to solve the same problem, with substantially different complexities.

EXERCISES

21.1 Compute the time-cost of the selection sort and bubble sort algorithms of Chapter 13.

21.2 How many steps are required by the modified bubble sort algorithm (Listing 13.3) to sort a list of length n in the best possible case?

21.3 Modify the "naive" algorithm for maximum sum of a sublist (Listing 21.1) so that it is $\Theta(n^2)$.

21.4 Compute the time-cost of the insertion sort algorithm of exercise 13.5.

21.5 Compute the time-cost of the recursive binary search algorithm of exercise 17.1. Compute the time-cost of the binary search algorithm of Section 13.3.

21.6 Compute the time-cost of the towers puzzle-solving algorithm given in Section 17.2.

21.7 Compute the time-cost of the merge-sort algorithm of exercise 17.3.

21.8 Compute the time-cost of the *String*-reversing method of exercise 17.4.

21.9 Verify that the relation $\sqsubseteq$ introduced in Section 21.3 is transitive.

21.10 Verify assertions <21.1> through <21.6> from page 539.

GLOSSARY

O-dominated by: a relation between time-cost functions; f is O-dominated by g provided there is a positive integer n_0 and a positive real c, such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

$O(f)$: the set of functions that are O-dominated by f.

$\Omega(f)$: the set of functions that f is O-dominated by.

$\Theta(f)$: the set of functions that have the same magnitude as f; that is $\Theta(f) = O(f) \cap \Omega(f)$.

intractable: a problem for which the best possible algorithm is exponential.

time-complexity: the time-cost of a method.

time-cost: a function that gives the maximum number of steps required by a method to solve a problem of a specified size.