

1

Overview

In this chapter, we introduce the major topics covered in this book. These topics include collections, abstract data types, and the analysis of algorithms. In addition, we give a brief overview of object-oriented programming and the software development process.

1.1 Collections

The principal topic of this book is collections. A collection, as the name implies, is a group of items that we wish to treat as a conceptual unit. Nearly every nontrivial piece of software involves the use of collections, and while much of what you learn in computer science comes and goes with changes in technology, the basic principles of organizing collections endure. In your first computer science course, you undoubtedly worked extensively with arrays, which are the most common and fundamental type of collection. Other important types of collections include stacks, lists, queues, binary search trees, heaps, graphs, maps, sets, and bags. Collections can be homogeneous, meaning that all items in the collection must be of the same type, or heterogeneous, meaning the items can be of different types. In most languages, arrays are homogeneous. In some collections, the items are restricted to being objects, whereas in others the items can be objects or primitive types such as integer and double.

1.1.1 Categories of Collections

An important distinguishing characteristic of collections is the manner in which they are organized. In this book, we explore four main categories of collections.

Linear Collections

The items in a linear collection are, like people in a line, ordered by position. Each item except the first has a unique predecessor, and each item except the last has a unique successor. As shown in Figure 1.1, D2's predecessor is D1, and its successor is D3.

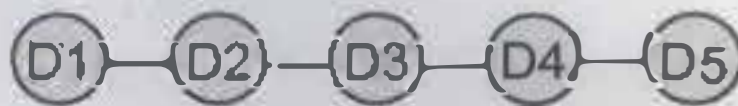


Figure 1.1 A linear collection

Everyday examples of linear collections are grocery lists, stacks of dinner plates, and a line of customers waiting at a bank.

Hierarchical Collections

Data items in hierarchical collections are ordered in a structure reminiscent of an upside-down tree. Each data item, except the one at the top, has just one predecessor, its **parent**, but potentially many successors, called its **children**. As shown in Figure 1.2, D3's predecessor (parent) is D1, and its successors (children) are D4, D5, and D6.

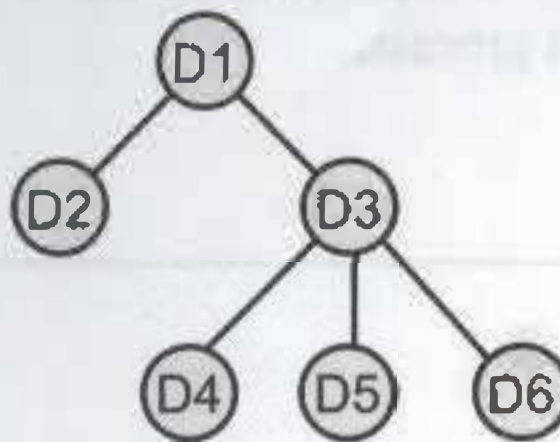


Figure 1.2 A hierarchical collection

A company's organization tree and a book's table of contents are examples of hierarchical collections.

Graphs or Networks

A collection in which each data item can have many predecessors and many successors is called a graph. As shown in Figure 1.3, all elements connected to D3 are considered to be both its predecessors and successors.

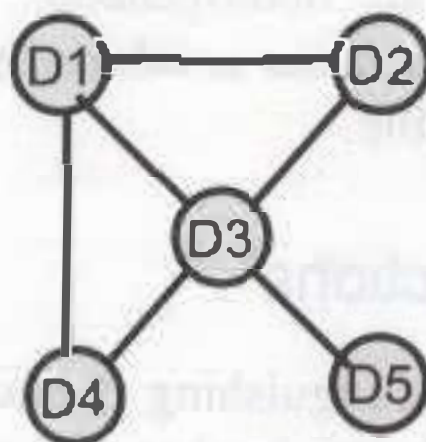


Figure 1.3 A graph collection

Examples of graphs are maps of airline routes between cities and electrical wiring diagrams for buildings.

Unordered Collections

As the name implies, items in an unordered collection are not in any particular order, and one cannot meaningfully speak of an item's predecessor or successor. Figure 1.4 shows such a structure.

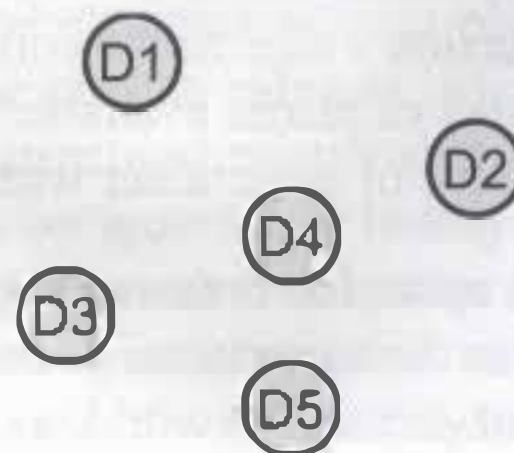


Figure 1.4 An unordered collection

A bag of marbles is an example of an unordered collection. Although one can put marbles into and take marbles out of a bag, once in the bag, the marbles are in no particular order.

1.1.2 Operations on Collections

Collections are typically dynamic rather than static, and their contents change throughout the course of a program. The actual manipulations that can be performed on a collection vary with the type of collection being used, but generally, the operations fall into several broad categories:

Search and retrieval: These operations search a collection for a given target item or for an item at a given position. If the item is found, either it or its position is returned. If the item is not found, a distinguishing value such as `null` or `-1` is returned.

Removal: This operation deletes a given item or the item at a given position.

Insertion: This operation adds an item to a collection and usually at a particular position within the collection.

Replacement: Replacement combines removal and insertion into a single operation.

Traversal: This operation visits each item in a collection. Depending on the type of collection, the order in which the items are visited can vary. During a traversal, items can be accessed or modified. Some traversals also permit the insertion or removal of items from the underlying collection.

| | |
|-------------------------------|--|
| Test for equality: | If items can be tested for equality, then the collections containing them can also be tested for equality. To be equal, two collections must contain equal items at corresponding positions. For unordered collections, of course, the requirement of corresponding positions can be ignored. Some collections, such as strings, can also be tested for their position in a natural ordering using the comparisons less than and greater than. |
| Determination of size: | Every collection contains a finite number of items. This number is the collection's size. Some collections also have a maximum capacity, or number of places available for storing items. An egg carton is a familiar example of a container with a maximum capacity. |
| Cloning: | This operation creates a copy of an existing collection. The copy usually shares the same items as the original, a feat that is impossible in the real world. In the real world, a copy of a bag of marbles could not contain the same marbles as the original bag, given a marble's inability to be in two places at once. However, the rules of cyberspace are more flexible, and there are many situations in which we make these strange copies of collections. What we are copying is the structure of the collection, not the elements it contains. However, it is possible, and sometimes useful, to produce a <i>deep copy</i> of a collection in which both the structure and the items are copied. |

1.2 Abstract Data Types

From the foregoing discussion, we see that a collection consists of data organized in a particular manner together with methods for manipulating the data. However, those who use collections in their programs have a rather different perspective on collections than those who are responsible for implementing them in the first place.

Users of collections need to know how to declare and use each type of collection. From their perspective, a collection is seen as a means for storing and accessing data in some predetermined manner, without concern for the details of the collection's implementation. From the users' perspective a collection is an *abstraction*, and for this reason, in computer science, collections are called *abstract data types*.

Developers of collections, on the other hand, are concerned with implementing a collection's behavior in the most efficient manner possible, with the goal of providing the best performance to users of the collections. Throughout this book, we consider collections from both angles. For each category of collections (linear, hierarchical, etc.), we define one or more abstract data types and one or more implementations of that type.

The idea of abstraction is not unique to a discussion of collections. It is an important principle in many endeavors both in and out of computer science. For example, when studying the effect of gravity on a falling object, we try to create an experimental situation in which we can ignore incidental details such as the color and taste of the object (what sort of apple was it anyway that hit Newton on the head?). When studying mathematics, we do not concern ourselves with what the numbers might be used to count, fishhooks or arrowheads, but try to discover abstract and enduring principles of numbers. A house plan is an abstraction of the physical house, and it allows us to focus on structural elements without being overwhelmed by incidental details such as the color of the kitchen cabinets, which is important to the eventual house but not to the house's structural relationships. In computer science, we also use abstraction as a technique for ignoring or hiding details that are for the moment inessential, and we often build up a software system layer by layer, each layer being treated as an abstraction by the layers above that utilize it. Without abstraction, we would need to consider all aspects of a software system simultaneously, an impossible task. Of course, the details must be considered eventually, but in a small and manageable context.

In Java, methods are the smallest unit of abstraction, classes are the next, and packages the largest. We will implement abstract data types as classes and gather them together in a package called `lamborne`. The `lamborne` package in turn supplements and extends the package `java.util`, which is a key component of the Java development library.

1.3 Algorithm Analysis

As we are going to be implementing abstract data types in several different ways, we need some basis for comparing their implementations. We must ask: What are the qualities that make one implementation preferable to another, assuming that each implementation is correct to begin with? More generally, how can we choose between competing algorithms for any computer process? There are several criteria for measuring algorithms, the most important of which are simplicity, clarity, and efficiency.

1.3.1 Simplicity and Clarity

All things being equal, and they never are, we prefer to use algorithms that are as simple and clear as possible; however, attempts to improve space and time efficiency usually come at the cost of increased programming complexity. As people are fond of saying, there is no such thing as a free lunch.

1.3.2 Space Efficiency

When we discuss space efficiency, we are not talking about the memory occupied by the code. Rather we are concerned with memory allocated to data and to supporting method calls. The space requirements for the latter become significant whenever recursive methods are used. The memory requirements for an ideal implementation of a collection would be exactly equal to the space occupied by the data being stored in the collection. However, in practice, there is nearly always additional overhead. As we present various implementations of collections, we compare the actual space used to the ideal.

1.3.3 Time Efficiency

The amount of time it takes to perform a task can vary greatly depending on the algorithm used. For example, consider how long it takes to search a list of names. The most obvious approach is to start at the top of the list and work down until either the name or the end of the list is encountered. On average, when a name is in the list, this linear search process entails looking at half the list. However, if the list is sorted alphabetically, a much more efficient binary search algorithm can be used. First, look at the name in the middle of the list. If it equals the name desired, the search is over. If it is earlier, repeat the search in the first half of the list, and if it is later, search in the second half. At each stage, this process halves the portion of the list remaining to be examined, and it terminates quickly. Table 1.1 shows the average number of comparisons needed by both methods as a function of the list's size.

Table 1.1

| The Number of Comparisons as a Function of a List's Size | | |
|--|---|---|
| Size of List | Average Number of Comparisons Made During a Binary Search | Average Number of Comparisons Made During a Linear Search |
| 10 | 4 | 5 |
| 100 | 7 | 50 |
| 1000 | 10 | 500 |
| 10,000 | 13 | 5000 |
| n | smallest m such that $n \leq 2^m$ | $n / 2$ |

Because the two search algorithms are fundamentally different in their speed, they are said to be of a different *order*. Later, we define this term precisely, but basically, two algorithms are of a different order if there is a point beyond which one is always better than the other. For instance, consider Figure 1.5, which graphs the running time of two algorithms based on the size of the collection they are processing. For values of n less than N_1 , algorithm A is faster than algorithm B, but after that, B is always superior.

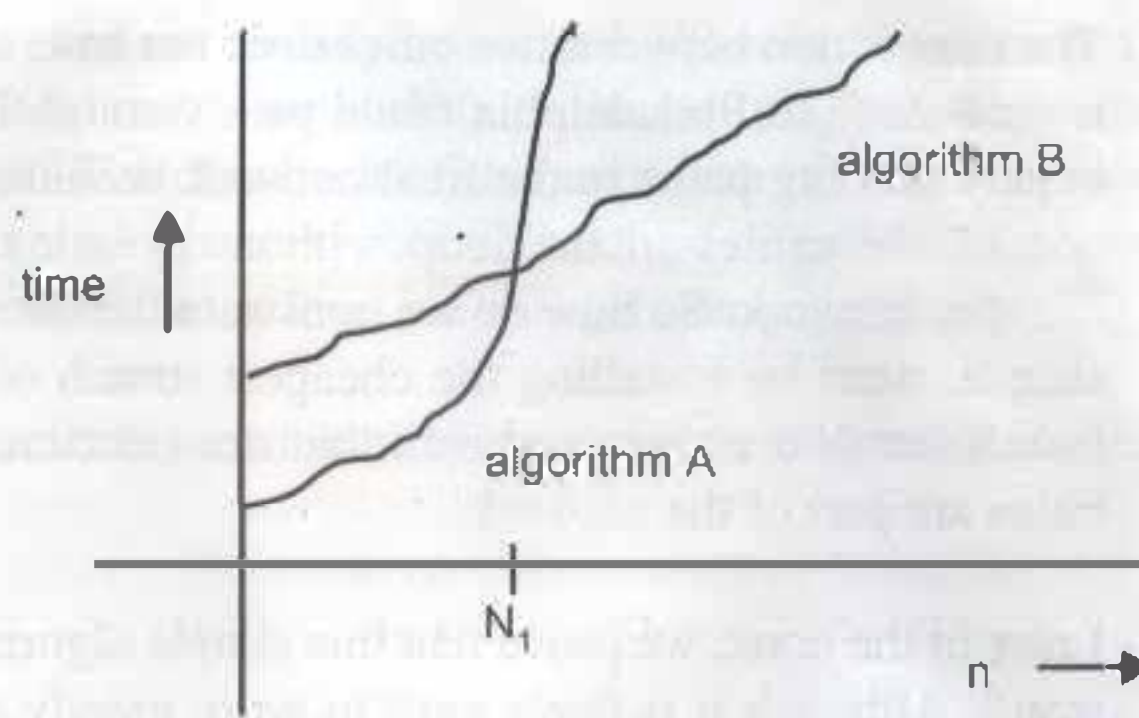


Figure 1.5 The running times of two algorithms

1.4 Algorithm Types

Computer scientists sometimes classify algorithms according to the design strategies they employ. Three of the most commonly used types of algorithms are called **greedy algorithms**, **divide-and-conquer algorithms**, and **backtracking algorithms**.

1.4.1 Greedy Algorithms

At each step, a greedy algorithm does the thing that is most profitable or most urgent or most gratifying at that instant without regard to future consequences. **A greedy algorithm repeats this process until the task has been completed.** An everyday example occurs when we work through a list of things to do. At each step, we select the most urgent item. As a more complex example, consider the task of connecting several cities with a network of fiber optic cables in the cheapest manner possible (see Figure 1.6).

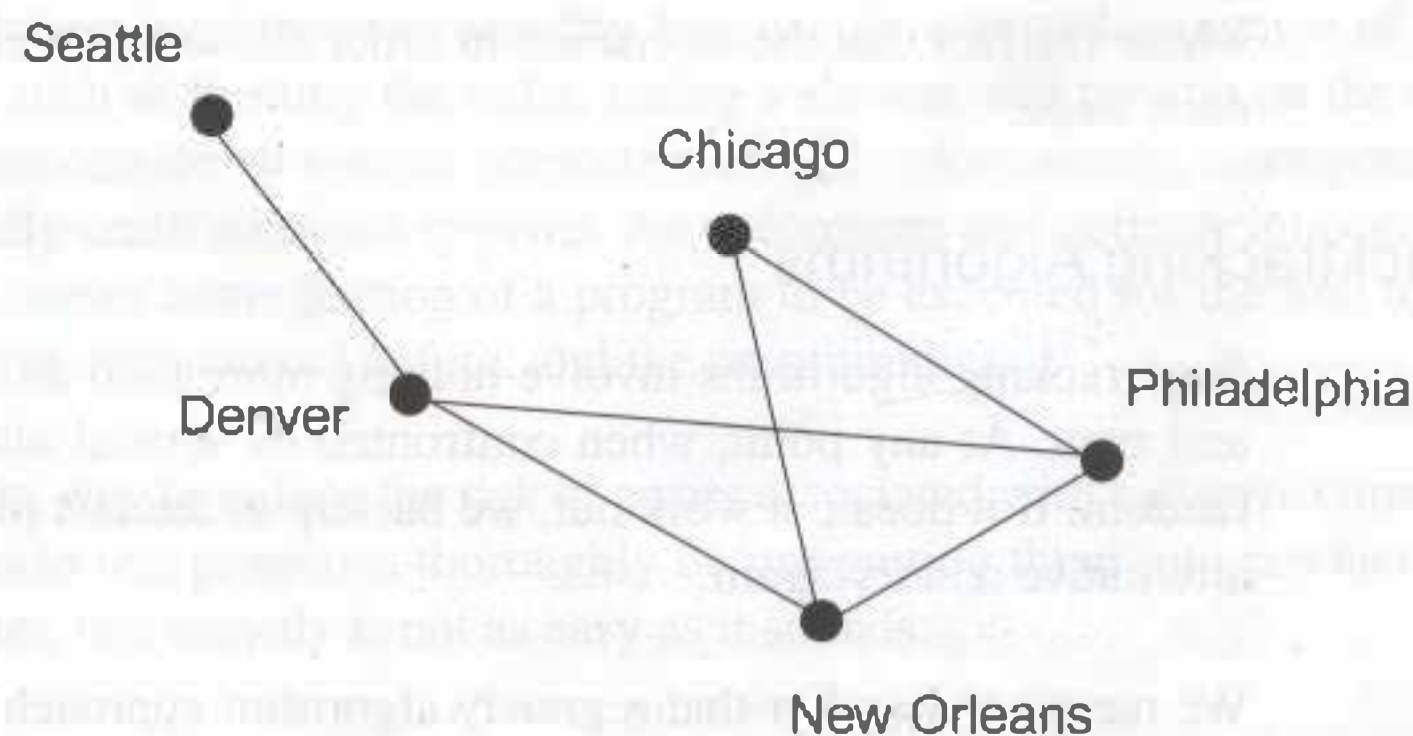


Figure 1.6 Cable routes between several cities

The connection between two cities does not have to be direct. For instance, the path from Seattle to Philadelphia could pass through Denver. Obviously, there will not be any looping paths in the final network because, if there were, we could remove one of the cables in the loop without isolating any cities, thereby obtaining a cheaper network. So how do we construct the network? The answer is surprisingly simple. Start by installing the cheapest stretch of cable. At each subsequent step, install the next cheapest stretch that does not introduce a loop. Continue until all cities are part of the network.

Later in the book, we prove that this simple algorithm actually produces the desired result. Although it is fairly easy to write greedy algorithms, there are many situations in which they do not work as intended. In general, we can hardly expect a sequence of locally optimal decisions to lead to a globally optimal result. For instance, when one hikes up a mountain, taking the easiest path at each juncture is as likely to lead to the base of an unscalable cliff as it is to the peak.

1.4.2 Divide-and-Conquer Algorithms

In a divide-and-conquer algorithm, we divide a large problem into smaller sub-problems. We then solve each subproblem separately and combine the results into a solution to the whole problem. The procedure is particularly attractive when each subproblem is of the same type as the original so that it can be solved by reapplying the technique, until finally sub-subproblems are reached which are trivially simple to solve. This particular approach to divide-and-conquer algorithms is called *recursion*.

As an everyday example, suppose we are asked to look for a needle in a haystack. We divide the haystack in two and look for the needle in each half separately. We reapply the technique to each half in turn and repeat until the pieces are so small that we can tell at a glance if the needle is in one of them. Despite being tediously slow, the technique should work, although the authors cannot claim to have actually tried it. However, we have all used a similar technique to find words in a dictionary, where the fact that the words are in order allows us to eliminate blocks of pages at each stage.

1.4.3 Backtracking Algorithms

Backtracking algorithms involve nothing more than an organized approach to trial and error. At any point, when confronted by several alternatives, we select one at random. If it does not work out, we backup to the last place that still has an untried alternative and try again.

We mentioned earlier that a greedy algorithm approach to mountain climbing was inadvisable; however, a backtracking algorithm is guaranteed to get us to the top provided there is at least one climbable path from the parking lot to the summit. We begin by taking any path from the parking lot. At each junction in the trail, we take

any direction that has not already been tried. Whenever we reach a location beyond which we cannot advance, we backup to the last place at which there is an untried choice and take it. The drawback of this approach is that it is probably going to be inefficient. Unless we get lucky, we will not go directly to the mountaintop on the shortest path possible, but at least we will get there eventually.

We will encounter many examples of these different types of algorithms throughout the book.

1.11 Proofs of Correctness

The difficulty of testing programs leads us to look for other approaches to establish program correctness. One of these consists of trying to prove that a program is correct in a strictly mathematical sense. The form of the statement we attempt to prove might be something like this: Given inputs of type X, then program Y produces outputs of type Z. Regrettably, proving even the simplest program correct is quite tedious, and most programmers are much worse at mathematics than they are at programming. Their proofs of correctness might be even more suspect than their code. Nonetheless, programs of significant size have been proven correct, and an understanding of the basic processes involved can help anyone reason more effectively about the programs he or she writes.

~~At this point, people sometimes wonder why we do not automate the process of determining program correctness. There are, after all, theorem proving programs. However, before investing a lot of effort in trying to write a general purpose correctness proving program, you need to keep a rather amazing fact in mind. It is a proven, incontestable, mathematical truth that it is impossible to write a program that performs the following seemingly much simpler task. To wit, write a program that takes as input the listing for an arbitrary program, X, and a list of arbitrary inputs for that program, Y, and then determines if X will stop or run forever when presented with Y. This is called the *halting problem*. And it is impossible, not just difficult, to write a program that can solve it, not just at this time and place, but at any time and in any place by any species in the universe.~~

~~But wait a minute, you say. How can one ever rule out the possibility of doing something? How can one know that some future genius will not break through the seeming barrier of impossibility? Such are the wonders of mathematics that it is possible to prove something impossible. Another task that mathematical analysis has proven impossible comes from the realm of plain old high school geometry, discovered over 2000 years ago by the early Greeks. As you may remember, it is~~

easy to bisect an angle using nothing more than a compass and straight edge. However, centuries of effort have never revealed a way to trisect an angle using the same tools. Now, thanks to a branch of mathematics called Galois theory, we know that it can never be done. The French mathematician and genius Evariste Galois, who developed the theory, died in a duel at the age of 21 in the year 1832.

The moral of this discussion of testing and proofs of correctness is not, “Hey, you can’t ever be certain your software is correct, so just put it out there and see what happens.” No, the responsible programmer must take a disciplined approach to software development. It is possible to write high-quality software, although **there are limits to the confidence we should place in complex systems.**

1.12 Some Other Aspects of the Software Development Process

We now consider two more things we can do to improve the quality of our software. First, we can follow some well-accepted coding conventions, and second, we can use preconditions and postconditions.

1.12.1 Coding Conventions

The use of good coding conventions leads to more readable, maintainable programs. Good programmers adhere to the following conventions.

Meaningful class, variable, and method names: Class and variable names should be nouns or noun phrases that suggest the purpose of the class or variable, for instance, `salesTaxRate` rather than `tax` or `str`. Method names should be verbs or verb phrases that describe the method’s action, such as `computePay` rather than just `pay` or `compute`.

Symbolic constants: A symbolic constant names a value that does not change throughout the course of a program. Using a name, such as `MAX_ACCOUNTS`, instead of a “magic number,” such as 25, greatly enhances program readability and maintainability.

Indentation: Modern programming languages are free format, and most modern programming environments come with syntax-directed editors that format code according to standard conventions. Examples of this format for Java abound throughout this book.

Capitalization conventions: The conventions Java programmers use for capitalizing identifiers are shown in Table 1.3.

Table 1.3

Java Conventions for Capitalizing Identifiers

| Kind of Identifier | Spelling Convention | Examples |
|-----------------------|---|-------------------------------|
| Constants | All uppercase. Words are separated by underscores. | MAX_VALUE, PI |
| Variables and methods | Begin with a lowercase letter. Words after the first one begin with a capital letter and are not separated by spaces. | length, getBalance |
| Classes | Begin each word with a capital letter. | MenuItem, SalariedEmployee |

Program comments: It is usually difficult to figure out what a program does just by reading the code, unless you happen to be the author and the program is fresh in your mind. Well-placed comments, on the other hand, help both the original programmer and those who must maintain the program in years to come. However, excessive comments, especially those that repeat the obvious, can be worse than none. Over time, programmers often ignore the comments and change the code without updating the comments.

What sort of comments are useful? A class's definition usually begins with an overview of the class's responsibilities. Each variable declaration can be followed by a brief comment that explains the variable's role. A tricky section of code should be preceded by comments that explains what it is doing. Methods can include a short description of what they do. This description often has a fixed format consisting of:

- ❑ a brief description of what the method does
- ❑ a list of the method's preconditions
- ❑ a list of the method's postconditions
- ❑ a description of the value returned by the method

In this book, comments in program code are italicized.

1.12.2 Preconditions and Postconditions

A precondition is a statement of what must be true before a method is invoked if the method is to run correctly. A postcondition is a statement of what will be true after the method has finished execution. One can think of preconditions and postconditions as the subject of an imaginary conversation between a method's author and user:

Author: Here are the things that you must guarantee to be true before my method is invoked. They are its preconditions.

User: Fine. And what do you guarantee will be the case if I do that?

Author: Here are the things that I guarantee to be true when my method finishes execution. They are its postconditions.

Preconditions usually describe the state of any parameters and instance variables that a method is about to access. Postconditions describe the state of any parameters and instance variables that the method has changed.

Preconditions and postconditions can also be set up as executable statements in Java. Such statements have the following form:

```
if (<condition is not satisfied>)  
    throw new <exception type> (<message string>);
```

Let's examine the use of preconditions, postconditions, and exceptions in a hypothetical square root method. If you are not yet familiar with Java, refer to Appendix A for a discussion of syntax.

```
double squareRoot (double x, double tolerance)  
// Computes the square root of x to within the specified tolerance  
//  
// Precondition: x >= 0 and tolerance > 0  
// Postcondition: If the computed value is called v, then  $|x - v*v| < \text{tolerance}$ .  
// Returns: The computed value v.  
{  
    if (x < 0)  
        throw new ArithmeticException("Error: x is negative");  
    if (tolerance <= 0)  
        throw new ArithmeticException("Error: tolerance not positive");  
  
    . . . code computing the square root goes here . . .  
  
    if (Math.abs(x - v * v) >= tolerance)  
        throw new ArithmeticException(  
            "Error: root not computed to required tolerance");  
    return v;  
}
```

In this code, each condition is enforced in the following way:

1. The logical complement of the precondition or postcondition is evaluated.
2. If that condition returns true, then the precondition or postcondition is not satisfied.
3. An exception of the appropriate type (in this case, an `ArithmeticException`) is thrown if the condition is true.

~~A client program in Java can catch these exceptions and handle them gracefully, or they can be allowed to propagate to the Java Virtual Machine, which terminates the program.~~

~~Preconditions and postconditions are especially helpful when used in conjunction with the many methods needed to implement an abstract data type. Preconditions and postconditions form a contract between~~

- ~~□ the designer trying to develop a set of manipulative methods that are consistent and complete,~~
- ~~□ the programmer trying to write the code, and~~
- ~~□ users of the ADT trying to determine how it works~~

4

Complexity

Algorithms are implemented as programs that run on real computers with finite resources. Computer programs consume two resources: processing time and memory. Obviously, when run with the same problems or data sets, programs that consume less of these two resources are in some sense “better” than programs that consume more. In this chapter, we introduce tools for analyzing the computational complexity or efficiency of algorithms. These tools are used throughout the book to assess the trade-offs in using different implementations of ADTs.

4.1 Measuring the Efficiency of Algorithms

4.1.1 Algorithms and the Resources They Use

Some algorithms consume an amount of time or memory that is below a threshold of tolerance. For example, most users are happy with any algorithm that loads a file in less than 1 second. Then, any algorithm that meets this requirement is as good as any other. Other algorithms take an amount of time or memory that is totally impractical (1000s of years) with large data sets. In such cases, we try to avoid using these algorithms and find others that perform better.

There often is a space/time trade-off in that an algorithm can be designed to gain speed at the cost of using extra memory or the other way around. Some clients might be willing to pay for more memory to get a faster algorithm, whereas others would rather settle for a slower algorithm that economizes on memory. In any case, because efficiency is a desirable feature of algorithms, it is important to pay attention to the potential of some algorithms for poor behavior.

4.1.2 Measuring Execution Time of an Algorithm

One way to measure the time cost of an algorithm is to use the computer's clock to obtain an actual running time. One can compute the average time for several different data sets of the same size. Then one obtains the running time of the algorithm for larger and larger data sets. After several such tests, the programmer has enough data to generalize and predict how an algorithm will behave for a data set of any size.

Consider a simple, if unrealistic, example. The following program implements an algorithm that counts from 1 to a given number. Thus, the problem size is the number. We start with the number 1,000,000, time the algorithm for that problem size, and output the running time to the terminal window. We then double the size of this number and repeat this process. After five such increases, we have a set of results from which we can generalize. Here is the code for the tester program:

```
long problemSize = 1000000;
int work = 0;

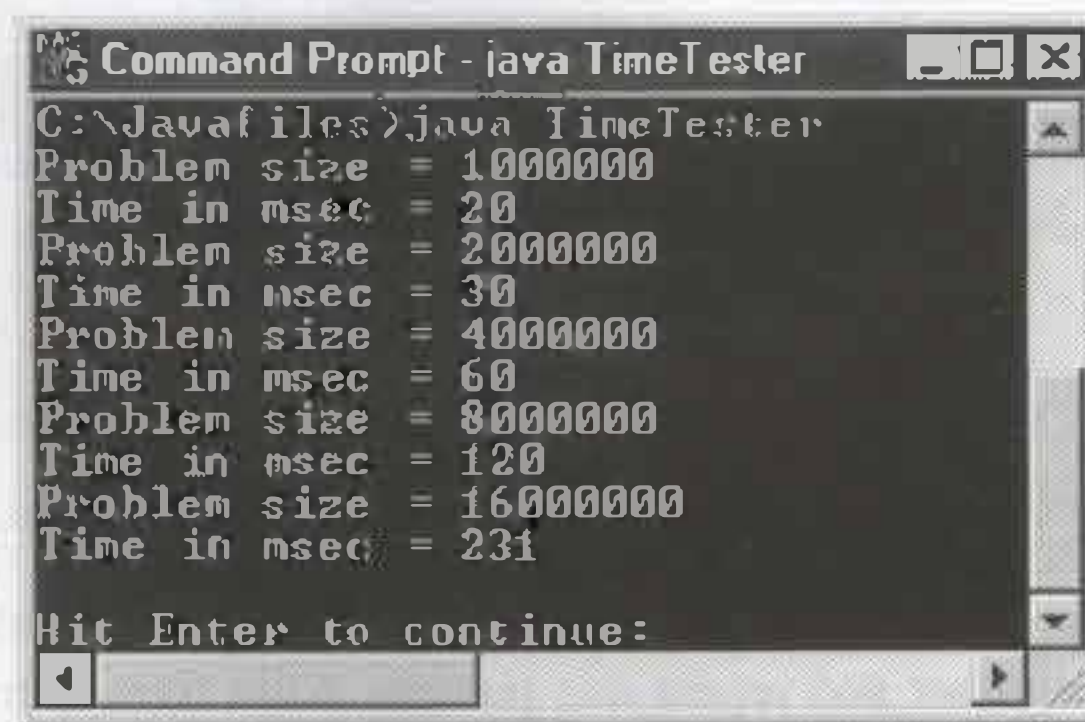
for (int i = 1; i <= 5; i++){
    Date d1 = new Date();

    // The start of "the algorithm"
    for (int j = 1; j <= problemSize; j++){
        work++;
        work--;
    }
    // The end of "the algorithm"

    Date d2 = new Date();
    long msec = d2.getTime() - d1.getTime();
    System.out.println("Problem size = " + problemSize + "\n" +
        "Time in msec = " + msec);
    problemSize = problemSize * 2;
}
```

The tester program uses the Date class defined in the **custom library** to track the running time. The constructor for Date returns a Date object containing the time at which it was created to the nearest millisecond. The Date method getTime returns the number of milliseconds between that time and January 1, 1970. Thus, the difference between the times of the two dates represents the elapsed time in milliseconds. Note also that the program does a constant amount of “work” on each pass through the loop. This work causes enough time to be consumed on each pass so that

the total running time is significant but has no other impact on our results. Figure 4.1 shows the output of the program.



```
Command Prompt - java TimeTester
C:\Javafiles>java TimeTester
Problem size = 1000000
Time in msec = 20
Problem size = 2000000
Time in msec = 30
Problem size = 4000000
Time in msec = 60
Problem size = 8000000
Time in msec = 120
Problem size = 16000000
Time in msec = 231
Hit Enter to continue:
```

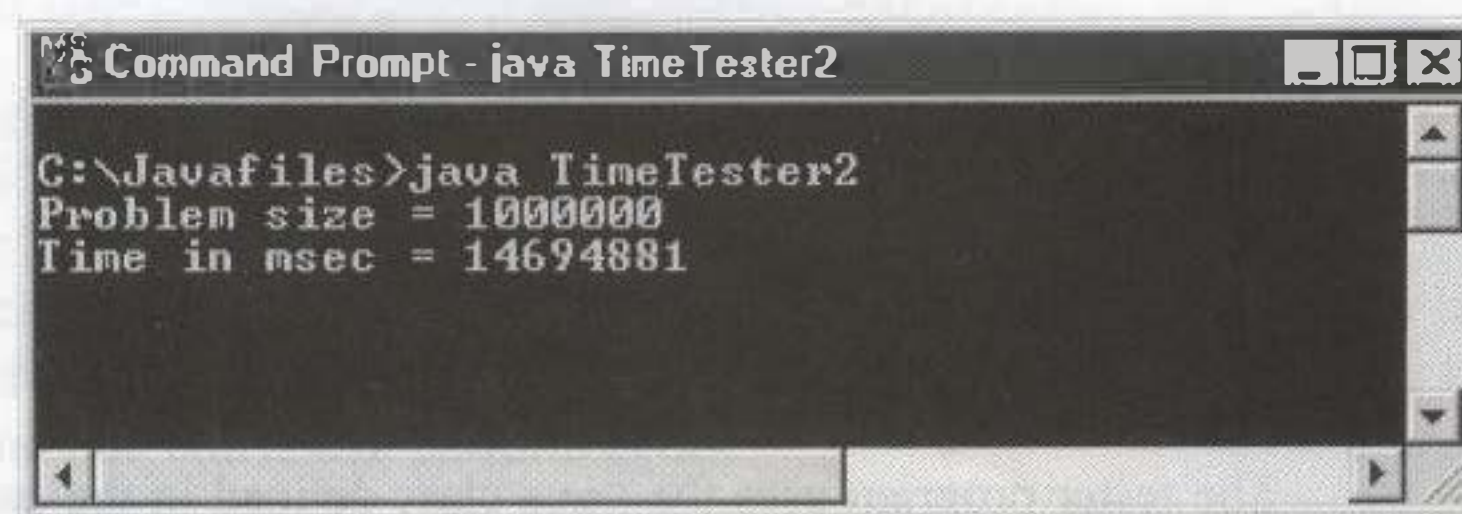
Figure 4.1 The output of the tester program

A quick glance at the results reveals that the running time more or less doubles when the size of the problem doubles. Thus, one might predict that the running time for a problem of size 32,000,000 is approximately 460 milliseconds.

As another example, consider the following change in the tester program's algorithm:

```
for (int j = 1; j <= problemSize; j++)
    for (int k = 1; k <= problemSize; k++){
        work++;
        work--;
    }
```

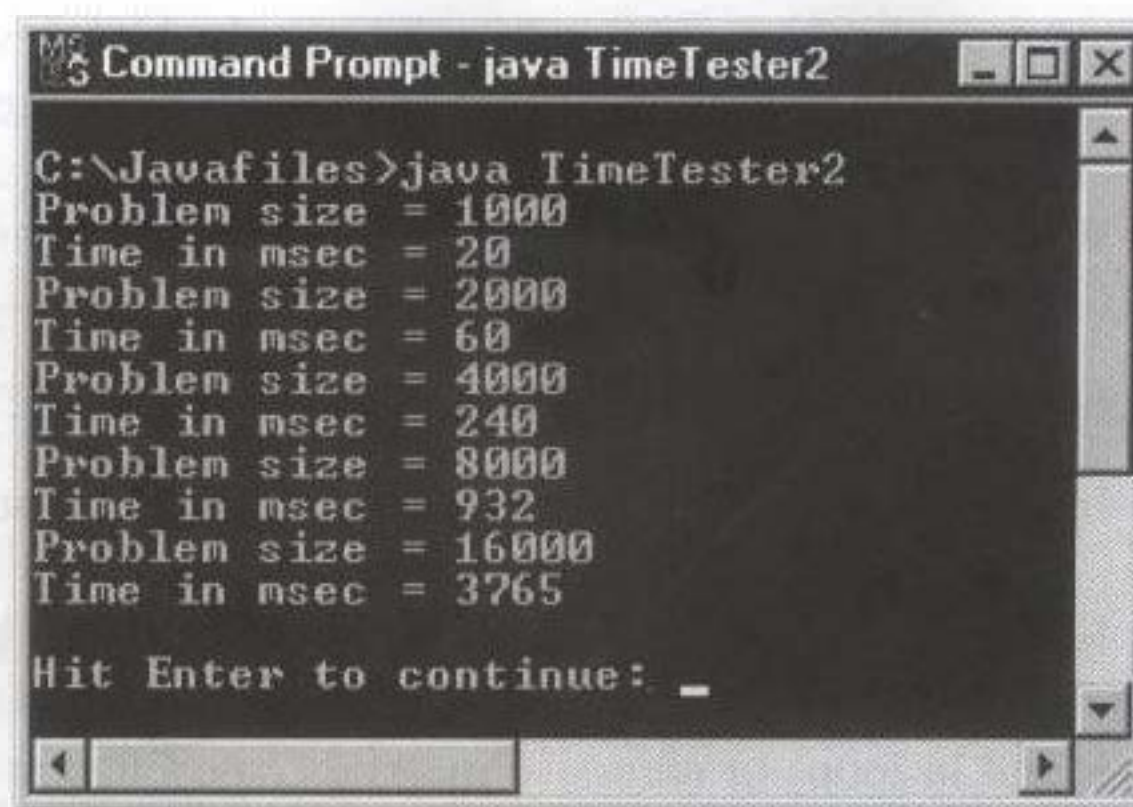
We have moved the “work” instructions into a nested loop. This loop also iterates through the size of the problem. One of the authors left this program running overnight and returned the next morning to find that it had processed just the first data set, as shown in Figure 4.2.



```
Command Prompt - java TimeTester2
C:\Javafiles>java TimeTester2
Problem size = 1000000
Time in msec = 14694881
```

Figure 4.2 The output of the tester program with a nested loop and initial size of 1,000,000

He then terminated the program and reduced the size of the initial data set to 1000. Figure 4.3 shows the results.



```
Command Prompt - java TimeTester2
C:\Javafiles>java TimeTester2
Problem size = 1000
Time in msec = 20
Problem size = 2000
Time in msec = 60
Problem size = 4000
Time in msec = 240
Problem size = 8000
Time in msec = 932
Problem size = 16000
Time in msec = 3765
Hit Enter to continue: _
```

Figure 4.3 The output of the tester program with a nested loop and initial size of 1000

Note that when the problem size doubles, the number of milliseconds of running time more or less quadruples. At this rate, it would take 175 days to process just the largest data set in the previous version!

Recording running times allows one to make accurate predictions of the running times of many algorithms. However, there are two major problems with this technique:

1. Different hardware platforms have different processing speeds, so the running times of an algorithm differ from machine to machine. Also, the running time of a program varies with the type of operating system that lies between it and the hardware. Finally, different programming languages and compilers produce code whose performance varies. For example, an algorithm coded in C usually runs faster than the same algorithm in Java byte code.
2. It is impractical to time some algorithms with very large data sets. For some algorithms, it doesn't matter how fast the compiled code or the computer processor is. They are impractical to run with very large data sets on any computer.

As with all testing, timing might still have a critical role to play. But we would also want an estimate of the efficiency of an algorithm that is independent of a particular hardware or software platform. This estimate tells us how well or how poorly the algorithm performs on any platform.

4.1.3 Counting Instructions

Another technique used to estimate the efficiency of an algorithm is to count the instructions executed with different problem sizes. These counts provide a good indicator of the amount of abstract “work” one can expect from an algorithm no matter what platform the algorithm runs on. Of course, when we count instructions,

we are referring to the instructions in the high-level code in which the algorithm is written, not instructions in the executable machine language program.

When analyzing an algorithm in this way, one distinguishes two classes of instructions:

1. Instructions that execute the same number of times regardless of the problem size
2. Instructions whose execution count varies with the problem size

For now, we ignore instructions in the first class because they do not figure significantly in this kind of analysis. The instructions in the second class normally are found in loops or recursive methods (see Chapter 10). In the case of loops, we also zero in on instructions performed in any nested loops. For example, let us wire the algorithm of the previous program to track and display the number of times the inner loop executes with the different data sets:

```
for (int i = 1; i <= 5; i++){
    int count = 0;
    for (int j = 1; j <= problemSize; j++){
        for (int k = 1; k <= problemSize; k++){
            count++;
            work++;
            work--;
        }
        System.out.println("Problem size = " + problemSize + "\n" +
                           "Instr. count = " + count);
        problemSize = problemSize * 2;
    }
}
```

As you can see from the results, the number of instructions executed is the square of the problem size (Figure 4.4).

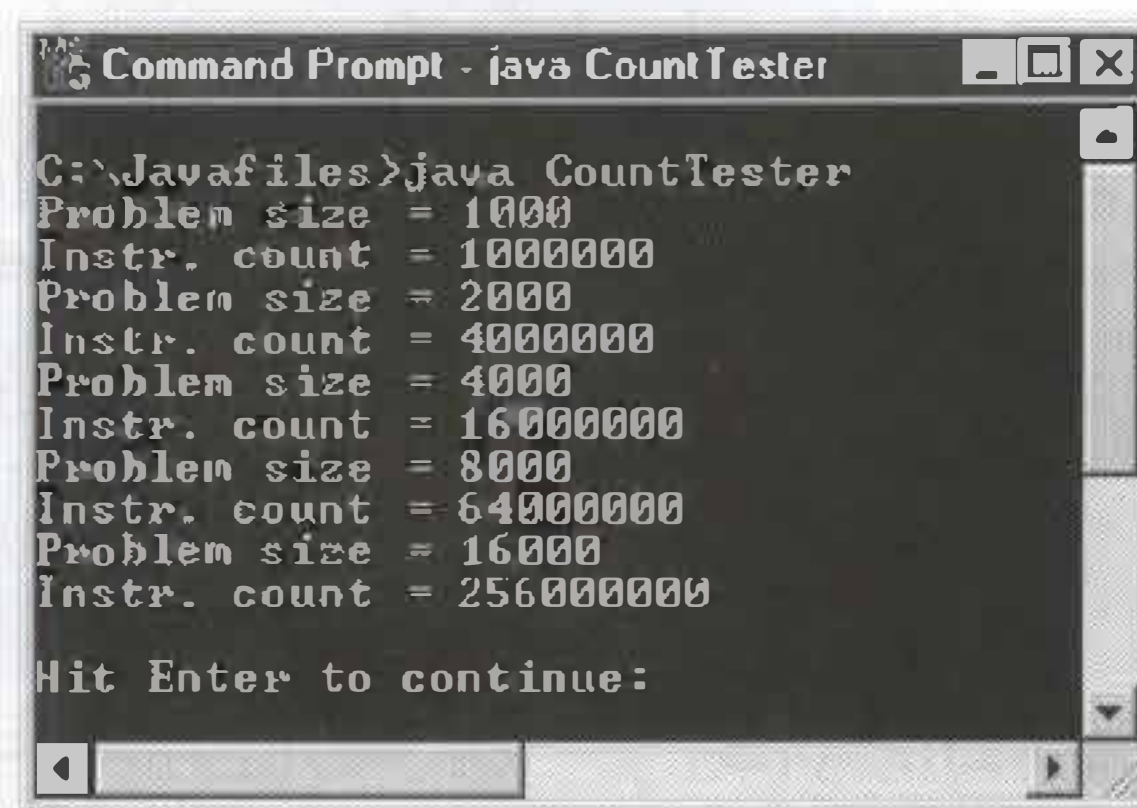


Figure 4.4 The output of a tester program that counts instructions

Here is a similar program that tracks the number of calls of a recursive fibonacci method for several problem sizes:


```

public class CountTester2{

    private static int count = 0;

    public static void main(String[] args){
        int problemSize = 2;

        for (int i = 1; i <= 5; i++){
            count = 0;
            fibonacci(problemSize);
            System.out.println("Problem size = " + problemSize + "\n" +
                               "Instr. count = " + count);
            problemSize = problemSize * 2;
        }
        Console.pause();
    }

    private static int fibonacci(int n){
        count++;
        if (n < 3)
            return 1;
        else
            return fibonacci(n - 1) + fibonacci(n - 2);
    }
}

```

The output of this program is shown in Figure 4.5.

```

Command Prompt - java CountTester2
C:\Javafiles>java CountTester2
Problem size = 2
Instr. count = 1
Problem size = 4
Instr. count = 5
Problem size = 8
Instr. count = 41
Problem size = 16
Instr. count = 1973
Problem size = 32
Instr. count = 4356617
Hit Enter to continue:

```

Figure 4.5 The output of a tester program that runs the Fibonacci method

As the problem size doubles, the instruction count (number of recursive calls) grows slowly at first and then very rapidly. At first, the instruction count is less than the square of the problem size, but the instruction count of 1973 is significantly larger than the square of the problem size 16. (For a thorough discussion of the design and analysis of recursive algorithms, see Chapter 10.)

The problem with tracking counts in this way is that, with some algorithms, the computer still cannot run fast enough to show the counts for very large problem sizes. Counting instructions is the right idea, but we need to turn to pencil and paper for a complete method of analysis.

4.1.4 Measuring the Memory Used by an Algorithm

A complete analysis of the resources used by an algorithm includes the amount of memory required. Once again, we focus on rates of potential growth. Some algorithms require the same amount of memory to solve any problem. Other algorithms require more memory as the problem size gets larger. We consider several of these algorithms in later chapters, particularly in Chapter 10. For now, we simply assume that the ideal data structure contains a reference to each object in it and nothing more. We thus ignore the space requirements for the actual objects. For example, an array of n objects requires n references (these might be four bytes each).

4.2 Big-O Analysis

4.2.1 Orders of Complexity

We would like to have a way of assessing an algorithm for its efficiency with pencil and paper rather than relying on platform-dependent timings or impractical counts of instructions. Consider the two counting loops discussed earlier. The first loop executes n times for a problem of size n , whereas the second loop, which contains a nested loop, executes n^2 times. The amount of work that these two algorithms do is similar for small values of n but is very different for large values of n . Figure 4.6 and Table 4.1 illustrate this divergence.

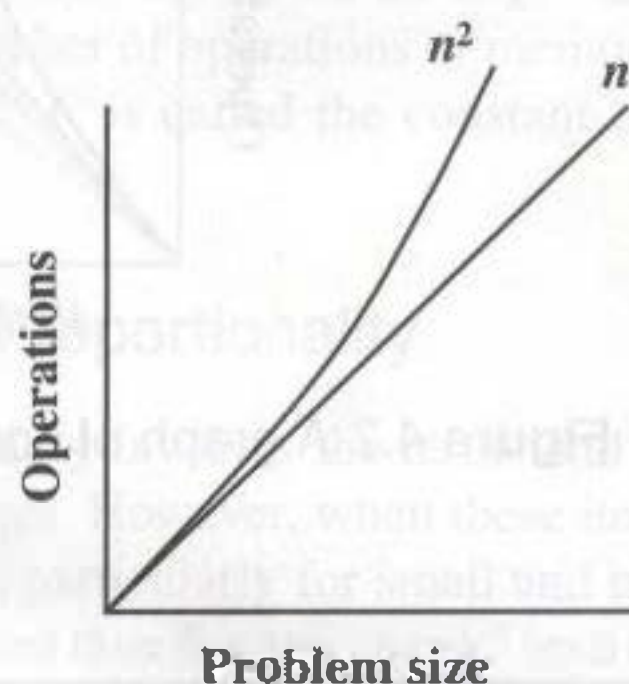


Figure 4.6 A graph of the amounts of work done in the tester programs

Table 4.1

The Amounts of Work in the Tester Programs

| Problem Size | Work of the First Algorithm | Work of the Second Algorithm |
|--------------|-----------------------------|------------------------------|
| 2 | 2 | 4 |
| 10 | 10 | 100 |
| 1000 | 1000 | 1,000,000 |

The behaviors of these algorithms differ by what we call an order of complexity. The behavior of the first algorithm is linear in that its work grows in direct proportion to the size of the problem. The behavior of the second algorithm is quadratic in that its work grows as a function of the square of the problem size. As you can see from the graph and the table, algorithms with linear behavior are better than algorithms with quadratic behavior, especially for large n .

A number of other orders of complexity are commonly used in the analysis of algorithms. An algorithm has constant behavior if it takes the same number of operations

for any problem size. Array indexing is a good example of a constant-time algorithm. Another order of complexity that is better than linear is called logarithmic. The amount of work of a logarithmic algorithm is proportional to the \log_2 of the problem size. An order of complexity that is worse than quadratic is called exponential. Exponential algorithms are impractical to run with large problem sizes. The common orders of complexity used in the analysis of algorithms are summarized in Figure 4.7 and Table 4.2.

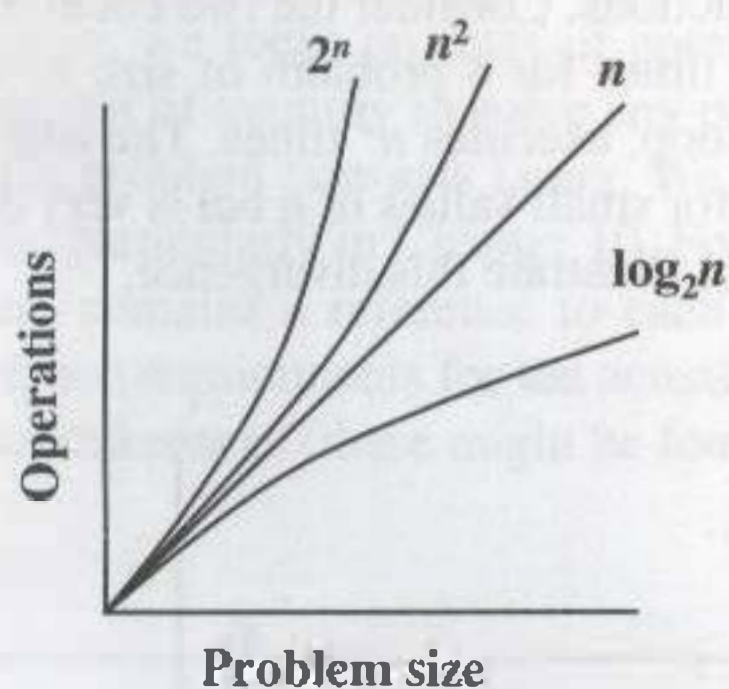


Figure 4.7 A graph of some sample orders of complexity

Table 4.2

| Some Sample Orders of Complexity | | | | |
|----------------------------------|----------------------------|----------------|---------------------|-----------------------|
| n | Logarithmic ($\log_2 n$) | Linear (n) | Quadratic (n^2) | Exponential (2^n) |
| 100 | 7 | 100 | 10,000 | Off the charts |
| 1000 | 10 | 1000 | 1,000,000 | Off the charts |
| 1,000,000 | 14 | 1,000,000 | 1,000,000,000,000 | Really off the charts |

4.2.2 Big-O Notation

An algorithm rarely performs a number of operations exactly equal to n or n^2 or whatever. For example, there usually is other work performed in the body of a loop. This work can be figured in as a constant factor, so we might say more exactly that an algorithm performs $2n$ or $2n^2$ operations. Moreover, in the case of a nested loop, the inner loop might run one less pass after each pass through the outer loop, so the amount of work might be more like $\frac{1}{2}n^2 - \frac{1}{2}n$. Thus, the amount of work typically is the sum of several terms in a polynomial expression. Whenever the amount of work is expressed as a polynomial, we focus on one term as dominant. As n becomes large, this term becomes so large that the amount of work represented by the other terms can be ignored. Thus, for example, in the polynomial expression $\frac{1}{2}n^2 - \frac{1}{2}n$, we focus on the quadratic term, in effect dropping the linear term from consideration. We can also drop the coefficient $\frac{1}{2}$ because the difference between

$\frac{1}{2}n^2$ and n^2 is not significant for very large values of n . Complexity analysis with big-O notation is sometimes called asymptotic analysis because the value of a polynomial asymptotically approaches or approximates the value of its largest term as n becomes very large.

The notation that computer scientists use to express the efficiency or computational complexity of an algorithm is called big-O notation. “O” stands for “on the order of,” a reference to the order of complexity of the work of the algorithm. Big-O notation formalizes our discussion of orders of complexity.

Suppose there exists a function $f(n)$ such that the number of operations for problem size n is less than a constant C times $f(n)$ for all but finitely many n . Then the number of operations is proportional to $f(n)$ for all large values of n . Such an algorithm is $O(f(n))$ relative to the number of operations or memory cells required for problem of size n , or $O(n)$ for short. C is called the constant of proportionality, which we now discuss.

4.2.3 The Role of the Constant of Proportionality

The constant of proportionality involves the terms and coefficients that are usually ignored during big-O analysis. However, when these items are large, they may have an impact on the algorithm, particularly for small and medium-size data sets. In the example algorithms discussed thus far, the “work” instructions that execute within a loop are also part of the constant of proportionality. When analyzing an algorithm, one must be careful to determine that any “work” instructions do not hide a loop that depends on a variable problem size. If that is the case, then the analysis must move down into the nested loop, as we saw in the last example.

4.2.4 Best, Worst, and Average Behavior

The behavior of some algorithms depends on the placement of the data that are processed. For example, a linear search algorithm does less work to find a target at the beginning of an array than at the end of the array. For such algorithms, one can determine the best-case behavior, the worst-case behavior, and the average behavior. In general, we worry more about average and worst-case behaviors than about best-case behavior. In the examples that follow, we show how to analyze algorithms for all three cases.

Exercises

1. Assume that each of the following expressions indicates the number of operations performed by an algorithm for a problem size of n . Point out the dominant term of each algorithm and use big-O notation to classify it.

a. $2^n - 4n^2 + 5n$

b. $3n^2 + 6$

c. $n^3 + n^2 - n$

2. For problem size n , algorithms A and B perform exactly n^2 and $\frac{1}{2}n^2 + \frac{1}{2}n$ instructions, respectively. Which algorithm does more work? Are there particular problem sizes for which one algorithm performs significantly better than the other? Are there particular problem sizes for which both algorithms perform approximately the same amount of work?
3. Suppose there are three algorithms that have n^4 , n^3 , and n^2 behaviors, respectively. Explain why the three algorithms do basically the same amount of work.

4.3 Search Algorithms

We now present and discuss several typical algorithms for searching and sorting arrays. We first discuss the design of an algorithm, then show its implementation as a Java method, and finally provide an analysis of the algorithm's computational complexity. To keep things simple, each method processes a filled array of integers. Arrays of different sizes can be passed as parameters to the methods. The methods are declared as `public` and `static`, so they can be included in a class that can be used like Java's `Math` class. We use this package of methods in the Case Study later in this chapter.

4.3.1 Linear Search of an Array

Assume that items in the array are in random order. Then the only way to search for a target item is to begin with the item at the first position and compare it to the target. If the items are the same, we return the position of the current item. Otherwise, we move on to the next position. If we arrive at the last position and still cannot find the target, we return `-1`. This kind of search is called linear search or sequential search. Here is the Java code for the linear search method:

```
public static int linearSearch(int target, int[] array){
    for (int i = 0; i < array.length; i++){
        if (array[i] == target)
            return i;
    }
    return -1;
}
```


The comparison in the body of the loop is the privileged instruction for our analysis. Our analysis considers three cases:

1. In the worst case, the target item is at the end of the list or not in the list at all. Then the algorithm must make n comparisons for an array of size n . Thus, the worst-case complexity of linear search is $O(n)$.
2. In the best case, the algorithm finds the target at the first position, after making one comparison, for an $O(1)$ complexity.
3. In the average case, the algorithm performs $(n + n - 1 + n - 2 + \dots + 1) / n = (n + 1) / 2$ comparisons. For large n , the constant factor of $/2$ is insignificant, so the average complexity is still $O(n)$.

Clearly, the best-case behavior of linear search is rare when compared with the average and worst-case behaviors, which are essentially the same.

When the data in the array are objects rather than values of a primitive type, one must alter the comparison instruction. In Java, the operator `==` returns *true* only if the two objects are identical, and this is hardly ever the case in a search situation. We must use the `equals` method instead, as shown in the following code:

```
public static int linearSearch(Object target, Object[] array){
    for (int i = 0; i < array.length; i++){
        if (array[i].equals(target))
            return i;
    }
    return -1;
}
```

The effect of `equals` from an efficiency standpoint is to increase the running time of each comparison. For example, `equals` compares each pair of characters within two strings rather than the references to the string objects. However, this change simply increases the constant of proportionality and has no effect on the order of complexity of the search.

4.3.2 Binary Search of an Array

When you look up a person's number in a phone book, you don't do a linear search. Instead, you estimate the portion of pages to pull over based on the name's alphabetical position in the book, and open the book there. If the names on that page come before the target name, you look on later pages; if the names on that page come after the target name, you look on earlier pages. You repeat this process until you find the name or find that it's not in the book. You search this way rather than using linear search because the items in the phone book are in sorted order.

Let us assume that the items in the array are sorted in ascending order. The search algorithm now goes directly to the middle position in the array and compares the target to the item. If there is a match, we return the position. Otherwise, if the current

item is less than the target, we search the portion of the array after the middle position. If it's more, we search the portion of the array before the middle position. The search process stops when we find the target or we cannot compute a middle position from the current beginning and end positions. Here is the code for the binary search method:

```
public static int binarySearch(int target, int[] array){
    int low = 0;
    int high = array.length - 1;
    while (low <= high){
        int middle = (low + high) / 2;
        if (array[middle] == target)
            return middle;
        else if (array[middle] < target)
            low = middle + 1;           // Search to right of middle
        else
            high = middle - 1;         // Search to left of middle
    }
    return -1;
}
```

The first comparison within the loop is the privileged instruction. Once again, the worst case occurs when the target is not in the array. How many times does the loop run in the worst case? This is equal to the number of times the size of the array can be divided by 2. For an array of size n , we perform

$$n / 2 / 2 \dots / 2 = 1$$

k times. To solve for k , we have $n / 2^k = 1$, and $n = 2^k$, and $k = \log_2 n$. Thus, the worst-case complexity of binary search is $O(\log_2 n)$.

Figure 4.8 shows the portions of the array being searched in a binary search with an array of 9 items and a target item, 10, that is not in the array. The items compared to the target are shaded.

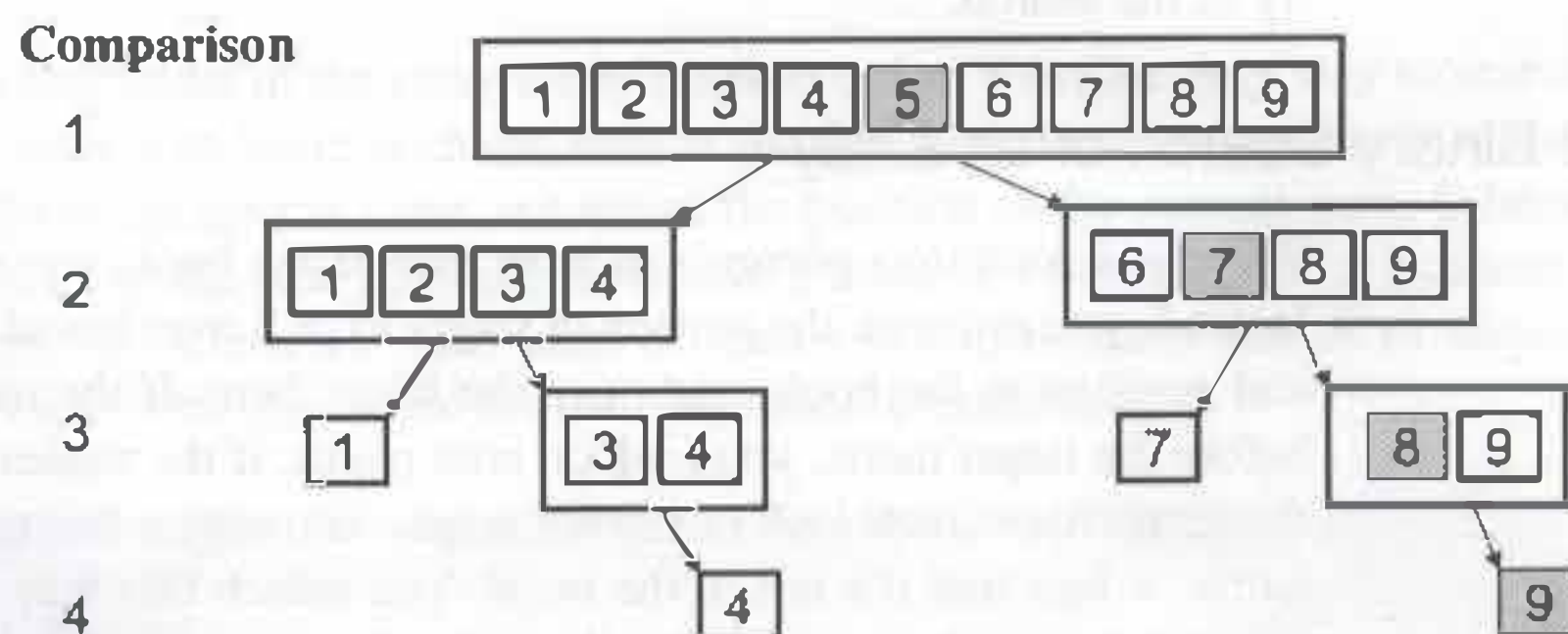


Figure 4.8 The items of an array visited during a binary search for 10

The binary search for the target item 10 requires four comparisons, whereas a linear search would have required ten comparisons.

As was the case with linear search, when the data in the array are objects rather than values of a primitive type, one must alter the way in which the comparisons are performed. In Java, the programmer must satisfy two requirements:

1. The objects in the array must implement the `Comparable` interface.
2. The search method must use the method `compareTo` to perform the comparisons. This method returns 0 if the objects are equal (using method `equals`), a positive integer if the first object is greater than the second object, or a negative integer if the first object is less than the second object.

Here is the modified code for the binary search of an array of objects:

```
// Assumes that the objects implement Comparable
public static int binarySearch(Object target, Object[] array){
    int low = 0;
    int high = array.length - 1;
    while (low <= high){
        int middle = (low + high) / 2;
        int result = array[middle].compareTo(target);
        if (result == 0)
            return middle;
        else if (result < 0)
            low = middle + 1;
        else
            high = middle - 1;
    }
    return -1;
}
```

Binary search is obviously more efficient than linear search. However, the kind of search algorithm we choose depends on the organization of the data in the array. There is some additional cost in the overall system if we use binary search. That cost lies in maintaining a sorted array. In the next section, we examine several strategies for sorting an array, and we analyze their complexity.

Exercises

1. Suppose that an array contains the values

20 44 48 55 62 66 74 88 93 99

2. The method we usually use to look up an entry in a phone book is not exactly the same as a binary search because we don't always go to the midpoint of the sublist being searched. Instead, we estimate the position of the target based on the alphabetical position of the first letter of the person's last name. For example, when looking up a number for Smith, we look toward the middle of the second half of the phone book first, instead of the middle of the entire book. Suggest a modification of the binary search algorithm that emulates this strategy for an array of names. Is its computational complexity any better than that of the standard binary search?

4.4 Sort Algorithms

Computer scientists have devised many ingenious strategies for sorting an array of items. We won't consider all of them here. In this chapter, we examine some algorithms that are easy to write but are inefficient. In Chapter 10, we look at some algorithms that are harder to write but are more efficient. Each of the Java sort methods that we develop here operates on an array of integers and uses a swap method to exchange the positions of two items in the array. Here is the code for that method:

```
public static void swap(int[] array, int i, int j){
    int temp = array[i];
    array[i] = array[j];
    array[j] = temp;
}
```

4.4.1 Selection Sort

Perhaps the simplest strategy is to search the entire array for the position of the smallest item. If that position does not equal the first position, we swap the items at those positions. We then start at the second position and repeat this process, swapping the smallest item with the item at the second position if necessary. When we reach the last position in this overall process, the array is sorted. The algorithm is called *selection sort* because each pass through the main loop selects a single item to be moved. Table 4.3 shows the states of an array of five items after each search and swap pass of selection sort. The two items just swapped on each pass have asterisks next to them, and the sorted portion of the array is shaded.

Table 4.3

A Trace of the Data During a Selection Sort

| Unsorted Array | After 1st Pass | After 2nd Pass | After 3rd Pass | After 4th Pass |
|----------------|----------------|----------------|----------------|----------------|
| 5 | 1* | 1 | 1 | 1 |
| 3 | 3 | 2* | 2 | 2 |
| 1 | 5* | 5 | 3* | 3 |
| 2 | 2 | 3* | 5* | 4* |
| 4 | 4 | 4 | 4 | 5* |

Here is the Java method for selection sort:

```
public static void selectionSort(int[] array){
    for (int i = 0; i < array.length - 1; i++){
        int minIndex = i;
        for (int j = i + 1; j < array.length; j++){
            if (array[j] < array[minIndex])
                minIndex = j;
        }
        if (minIndex != i)
            swap(array, minIndex, i);
    }
}
```

Selection sort has a nested loop within which we find our privileged instruction, the comparison, for analysis. For an array of size n , the outer loop executes $n - 1$ times. On the first pass through the outer loop, the inner loop executes $n - 1$ times. On the second pass through the outer loop, the inner loop executes $n - 2$ times. On the last pass through the outer loop, the inner loop executes once. Thus, the total number of comparisons for an array of size n is

$$(n - 1) + (n - 2) + \dots + 1 = n(n - 1) / 2 = 1/2n^2 - 1/2n$$

For large n , we can pick the term with the largest degree and drop the coefficient, so selection sort is $O(n^2)$ in all cases. For large data sets, the cost of swapping items might also be significant. Because data items are swapped only in the outer loop, this additional cost for selection sort is linear in the worst and average cases.

As was the case for binary search, when the array contains objects, the objects must implement the `Comparable` interface, and the sort method must use the `compareTo` method.

4.4.2 Bubble Sort

Another sort algorithm that is easy to conceive and code is called *bubble sort*. The strategy is to start at the beginning of the array and compare pairs of data items as we move down to the end. Each time the items in the pair are out of order, we swap

them. This process has the effect of “bubbling” the largest items down to the end of the array. We then repeat the process from the beginning of the array and go to the next to last item, and so on, until we begin with the last item. At that point, the array is sorted.

Table 4.4 shows a trace of the bubbling process through an array of five items. This process makes four passes to bubble the largest item down to the end of the array. Once again, the items just swapped are marked with asterisks, and the sorted portion is shaded.

Table 4.4

| A Trace of the Data During a Bubble Sort | | | | |
|--|----------------|----------------|----------------|----------------|
| Unsorted Array | After 1st Pass | After 2nd Pass | After 3rd Pass | After 4th Pass |
| 5 | 4* | 4 | 4 | 4 |
| 3 | 5* | 2* | 2 | 2 |
| 1 | 2 | 5* | 1* | 1 |
| 2 | 1 | 1 | 5* | 3* |
| 4 | 3 | 3 | 3 | 5* |

Here is the Java method for bubble sort:

```
public static void bubbleSort(int[] array){
    for (int i = 0; i < array.length - 1; i++)
        for (int j = 0; j < array.length - i - 1; j++)
            if (array[j] > array[j + 1])
                swap(array, j, j + 1);
}
```

As with selection sort, bubble sort has a nested loop within which we find our privileged instruction, a comparison. The sorted portion of the array now grows from the end of the array up to the beginning, but the behavior of bubble sort is quite similar to the behavior of selection sort: The inner loop executes $\frac{1}{2}n^2 - \frac{1}{2}n$ times for an array of size n . Thus, bubble sort is $O(n^2)$. Like selection sort, bubble sort won't perform any swaps if the array is already sorted. However, bubble sort's worst-case behavior for exchanges is greater than linear. The proof of this is left as an exercise.

We can make a minor adjustment to bubble sort to improve its best-case behavior to linear. If no swaps occur during a pass through the main loop, then the array is sorted. This can happen on any pass and in the best case will happen on the first pass. We can track the presence of swapping with a Boolean flag and return from the method when the inner loop does not set this flag. Here is the modified bubble sort method:


```

public static void bubbleSort(int[] array){
    for (int i = 0; i < array.length - 1; i++){
        boolean swapped = false;
        for (int j = 0; j < array.length - i - 1; j++){
            if (array[j] > array[j + 1]){
                swap(array, j, j + 1);
                swapped = true;
            }
        }
        if (!swapped)
            return;
    }
}

```

Note that this modification only improves best-case behavior. On the average, the behavior of bubble sort is still $O(n^2)$.

4.4.3 Insertion Sort

Our modified bubble sort performs better than selection sort for arrays that are already sorted but can still perform poorly if one or two items are out of order in the array. Another algorithm called *insertion sort* attempts to exploit the partial ordering of the array in a different way. The strategy is as follows:

- ❑ On the i th pass through the array, the i th item should be inserted into its proper place among the first i items array.
- ❑ After the i th pass, the first i items should be in sorted order.
- ❑ This process is analogous to the way in which many people organize playing cards in their hands. That is, if one holds the first $i - 1$ cards in order, one picks the i th card and compares it to these cards until its proper spot is found.
- ❑ As with our other sort algorithms, insertion sort consists of two loops. The outer loop traverses the positions from 1 to $n - 1$. For each position i in this loop, we save the item and start the inner loop at position $i - 1$. For each position j in this loop, we move the item to position $j + 1$ until we find the insertion point for the saved (i th) item.

Here is the code for the insertionSort method:

```

public static void insertionSort(int[] array){
    for (int i = 1; i < array.length; i++){
        int itemToInsert = array[i];
        int j = i - 1;
        while (j >= 0){
            if (itemToInsert < array[j]){
                array[j + 1] = array[j];
                j--;
            }
            else
                break;
        }
        array[j + 1] = itemToInsert;
    }
}

```


Table 4.5 shows the states of an array of five items after each major pass in an insertion sort. The item to be inserted on the next pass is marked with an arrow; after it is inserted, this item is marked with an asterisk.

Table 4.5

| A Trace of the Data During an Insertion Sort | | | | |
|--|------------------|----------------|----------------|----------------|
| Unsorted Array | After 1st Pass | After 2nd Pass | After 3rd Pass | After 4th Pass |
| 2 | 2 | 1* | 1 | 1 |
| 5 ← | 5 (no insertion) | 2 | 2 | 2 |
| 1 | 1 ← | 5 | 4* | 3* |
| 4 | 4 | 4 ← | 5 | 4 |
| 3 | 3 | 3 | 3 ← | 5 |

Once again, analysis focuses on the comparison in the nested loop as the privileged instruction. The outer loop executes $n - 1$ times. In the worst case, when all of the data are out of order, the inner loop executes once on the first pass, twice on the second pass, and so on, for a total of $\frac{1}{2}n^2 - \frac{1}{2}n$ times. Thus, the worst-case behavior of insertion sort is $O(n^2)$.

The more items in the array are in order, the better insertion sort gets until, in the best case of a sorted array, the algorithm is linear. In the average case, however, insertion sort is still quadratic.