Algorithms
THIRD EDITION
IN C++

Parts 1–4

FUNDAMENTALS
DATA STRUCTURES
SORTING
SEARCHING

ROBERT SEDGEWICK
with C++ consulting by Christopher J. Van Wyk

Algorithms
THIRD EDITION
in C++

PARTS 1–4

FUNDAMENTALS
DATA STRUCTURES
SORTING
SEARCHING

Robert Sedgewick
Princeton University

ADDISON-WESLEY

An imprint of Addison Wesley Longman, Inc.

Reading, Massachusetts • Harlow, England • Menlo Park, California
Berkeley, California • Don Mills, Ontario • Sydney • Bonn • Amsterdam
Tokyo • Mexico City

CHAPTER FIVE

Recursion and Trees

THE CONCEPT OF recursion is fundamental in mathematics and computer science. The simple definition is that a recursive program in a programming language is one that calls itself (just as a recursive function in mathematics is one that is defined in terms of itself). A recursive program cannot call itself always, or it would never stop (just as a recursive function cannot be defined in terms of itself always, or the definition would be circular); so a second essential ingredient is that there must be a *termination condition* when the program can cease to call itself (and when the mathematical function is not defined in terms of itself). All practical computations can be couched in a recursive framework.

The study of recursion is intertwined with the study of recursively defined structures known as *trees*. We use trees both to help us understand and analyze recursive programs and as explicit data structures. We have already encountered an application of trees (although not a recursive one), in Chapter 1. The connection between recursive programs and trees underlies a great deal of the material in this book. We use trees to understand recursive programs; we use recursive programs to build trees; and we draw on the fundamental relationship between both (and recurrence relations) to analyze algorithms. Recursion helps us to develop elegant and efficient data structures and algorithms for all manner of applications.

Our primary purpose in this chapter is to examine recursive programs and data structures as practical tools. First, we discuss the relationship between mathematical recurrences and simple recursive

programs, and we consider a number of examples of practical recursive programs. Next, we examine the fundamental recursive scheme known as *divide and conquer*, which we use to solve fundamental problems in several later sections of this book. Then, we consider a general approach to implementing recursive programs known as *dynamic programming*, which provides effective and elegant solutions to a wide class of problems. Next, we consider trees, their mathematical properties, and associated algorithms in detail, including basic methods for *tree traversal* that underlie recursive tree-processing programs. Finally, we consider closely related algorithms for processing graphs—we look specifically at a fundamental recursive program, *depth-first search*, that serves as the basis for many graph-processing algorithms.

As we shall see, many interesting algorithms are simply expressed with recursive programs, and many algorithm designers prefer to express methods recursively. We also investigate nonrecursive alternatives in detail. Not only can we often devise simple stack-based algorithms that are essentially equivalent to recursive algorithms, but also we can often find nonrecursive alternatives that achieve the same final result through a different sequence of computations. The recursive formulation provides a structure within which we can seek more efficient alternatives.

A full discussion of recursion and trees could fill an entire book, for they arise in many applications throughout computer science, and are pervasive outside of computer science as well. Indeed, it might be said that *this* book is filled with a discussion of recursion and trees, for they are present, in a fundamental way, in every one of the book's chapters.

5.1 Recursive Algorithms

A *recursive algorithm* is one that solves a problem by solving one or more smaller instances of the same problem. To implement recursive algorithms in C++, we use *recursive functions*—a recursive function is one that calls itself. Recursive functions in C++ correspond to recursive definitions of mathematical functions. We begin our study of recursion by examining programs that directly evaluate mathematical functions. The basic mechanisms extend to provide a general-purpose programming paradigm, as we shall see.

Program 5.1 Factorial function (recursive implementation)

This recursive function computes the function $N!$, using the standard recursive definition. It returns the correct value when called with N nonnegative and sufficiently small that $N!$ can be represented as an `int`.

```
int factorial(int N)
  {
    if (N == 0) return 1;
    return N*factorial(N-1);
  }
```

Recurrence relations (see Section 2.5) are recursively defined functions. A recurrence relation defines a function whose domain is the nonnegative integers either by some initial values or (recursively) in terms of its own values on smaller integers. Perhaps the most familiar such function is the *factorial* function, which is defined by the recurrence relation

$$N! = N \cdot (N-1)!, \qquad \text{for } N \geq 1 \text{ with } 0! = 1.$$

This definition corresponds directly to the recursive C++ function in Program 5.1.

Program 5.1 is equivalent to a simple loop. For example, the following `for` loop performs the same computation:

```
for ( t = 1, i = 1; i <= N; i++) t *= i;
```

As we shall see, it is always possible to transform a recursive program into a nonrecursive one that performs the same computation. Conversely, we can express without loops any computation that involves loops, using recursion, as well.

We use recursion because it often allows us to express complex algorithms in a compact form, without sacrificing efficiency. For example, the recursive implementation of the factorial function obviates the need for local variables. The cost of the recursive implementation is borne by the mechanisms in the programming systems that support function calls, which use the equivalent of a built-in pushdown stack. Most modern programming systems have carefully engineered mechanisms for this task. Despite this advantage, as we shall see, it is all too easy to write a simple recursive function that is extremely inefficient,

Program 5.2 A questionable recursive program

If the argument N is odd, this function calls itself with $3N + 1$ as an argument; if N is even, it calls itself with $N/2$ as an argument. We cannot use induction to prove that this program terminates, because not every recursive call uses an argument smaller than the one given.

```
int puzzle(int N)
  {
    if (N == 1) return 1;
    if (N % 2 == 0)
         return puzzle(N/2);
    else return puzzle(3*N+1);
  }
```

and we need to exercise care to avoid being burdened with intractable implementations.

Program 5.1 illustrates the basic features of a recursive program: it calls itself (with a smaller value of its argument), and it has a termination condition in which it directly computes its result. We can use mathematical induction to convince ourselves that the program works as intended:

- It computes $0!$ (basis).
- Under the assumption that it computes $k!$ for $k < N$ (inductive hypothesis), it computes $N!$.

Reasoning like this can provide us with a quick path to developing algorithms that solve complex problems, as we shall see.

In a programming language such as C++, there are few restrictions on the kinds of programs that we write, but we strive to limit ourselves in our use of recursive functions to those that embody inductive proofs of correctness like the one outlined in the previous paragraph. Although we do not consider formal correctness proofs in this book, we are interested in putting together complicated programs for difficult tasks, and we need to have some assurance that the tasks will be solved properly. Mechanisms such as recursive functions can provide such assurances while giving us compact implementations. Practically speaking, the connection to mathematical induction tells us that we should ensure that our recursive functions satisfy two basic properties:

```
puzzle(3)
  puzzle(10)
    puzzle(5)
      puzzle(16)
        puzzle(8)
          puzzle(4)
            puzzle(2)
              puzzle(1)
```

Figure 5.1 Example of a recursive call chain

This nested sequence of function calls eventually terminates, but we cannot prove that the recursive function in Program 5.2 does not have arbitrarily deep nesting for some argument. We prefer recursive programs that always invoke themselves with smaller arguments.

Program 5.3 Euclid's algorithm

One of the oldest-known algorithms, dating back over 2000 years, is this recursive method for finding the greatest common divisors of two integers.

```
int gcd(int m, int n)
  {
    if (n == 0) return m;
    return gcd(n, m % n);
  }
```

- They must explicitly solve a basis case.
- Each recursive call must involve smaller values of the arguments.

These points are vague—they amount to saying that we should have a valid inductive proof for each recursive function that we write. Still, they provide useful guidance as we develop implementations.

Program 5.2 is an amusing example that illustrates the need for an inductive argument. It is a recursive function that violates the rule that each recursive call must involve smaller values of the arguments, so we cannot use mathematical induction to understand it. Indeed, it is not known whether or not this computation terminates for every N, if there are no bounds on the size of N. For small integers that can be represented as `int`s, we can check that the program terminates (see Figure 5.1 and Exercise 5.4), but for large integers (64-bit words, say), we do not know whether or not this program goes into an infinite loop.

Program 5.3 is a compact implementation of *Euclid's algorithm* for finding the greatest common divisor of two integers. It is based on the observation that the greatest common divisor of two integers x and y with $x > y$ is the same as the greatest common divisor of y and $x \bmod y$ (the remainder when x is divided by y). A number t divides both x and y if and only if t divides both y and $x \bmod y$, because x is equal to $x \bmod y$ plus a multiple of y. The recursive calls made for an example invocation of this program are shown in Figure 5.2. For Euclid's algorithm, the depth of the recursion depends on arithmetic properties of the arguments (it is known to be logarithmic).

Program 5.4 is an example with multiple recursive calls. It is another expression evaluator, performing essentially the same compu-

```
gcd(314159, 271828)
  gcd(271828, 42331)
    gcd(42331, 17842)
      gcd(17842, 6647)
        gcd(6647, 4458)
          gcd(4458, 2099)
            gcd(2099, 350)
              gcd(350, 349)
                gcd(349, 1)
                  gcd(1, 0)
```

Figure 5.2 Example of Euclid's algorithm

This nested sequence of function calls illustrates the operation of Euclid's algorithm in discovering that 314159 and 271828 are relatively prime.

Program 5.4 Recursive program to evaluate prefix expressions

To evaluate a prefix expression, we either convert a number from ASCII to binary (in the `while` loop at the end), or perform the operation indicated by the first character in the expression on the two operands, evaluated recursively. This function is recursive, but it uses a global array containing the expression and an index to the current character in the expression. The index is advanced past each subexpression evaluated.

```
char *a; int i;
int eval()
  { int x = 0;
    while (a[i] == ' ') i++;
    if (a[i] == '+')
      { i++; return eval() + eval(); }
    if (a[i] == '*')
      { i++; return eval() * eval(); }
    while ((a[i] >= '0') && (a[i] <= '9'))
      x = 10*x + (a[i++]-'0');
    return x;
  }
```

```
eval() * + 7 * * 4 6 + 8 9 5
  eval() + 7 * * 4 6 + 8 9
    eval() 7
    eval() * * 4 6 + 8 9
      eval() * 4 6
        eval() 4
        eval() 6
        return 24 = 4*6
      eval() + 8 9
        eval() 8
        eval() 9
        return 17 = 8 + 9
      return 408 = 24*17
    return 415 = 7+408
  eval() 5
  return 2075 = 415*5
```

Figure 5.3 Prefix expression evaluation example

This nested sequence of function calls illustrates the operation of the recursive prefix-expression–evaluation algorithm on a sample expression. For simplicity, the expression arguments are shown here. The algorithm itself never explicitly decides the extent of its argument string: rather, it takes what it needs from the front of the string.

tations as Program 4.2, but on prefix (rather than postfix) expressions, and letting recursion take the place of the explicit pushdown stack. In this chapter, we shall see many other examples of recursive programs and equivalent programs that use pushdown stacks. We shall examine the specific relationship between several pairs of such programs in detail.

Figure 5.3 shows the operation of Program 5.4 on a sample prefix expression. The multiple recursive calls mask a complex series of computations. Like most recursive programs, this program is best understood inductively: Assuming that it works properly for simple expressions, we can convince ourselves that it works properly for complex ones. This program is a simple example of a *recursive descent parser*—we can use the same process to convert C++ programs into machine code.

A precise inductive proof that Program 5.4 evaluates the expression properly is certainly much more challenging to write than are the proofs for functions with integer arguments that we have been