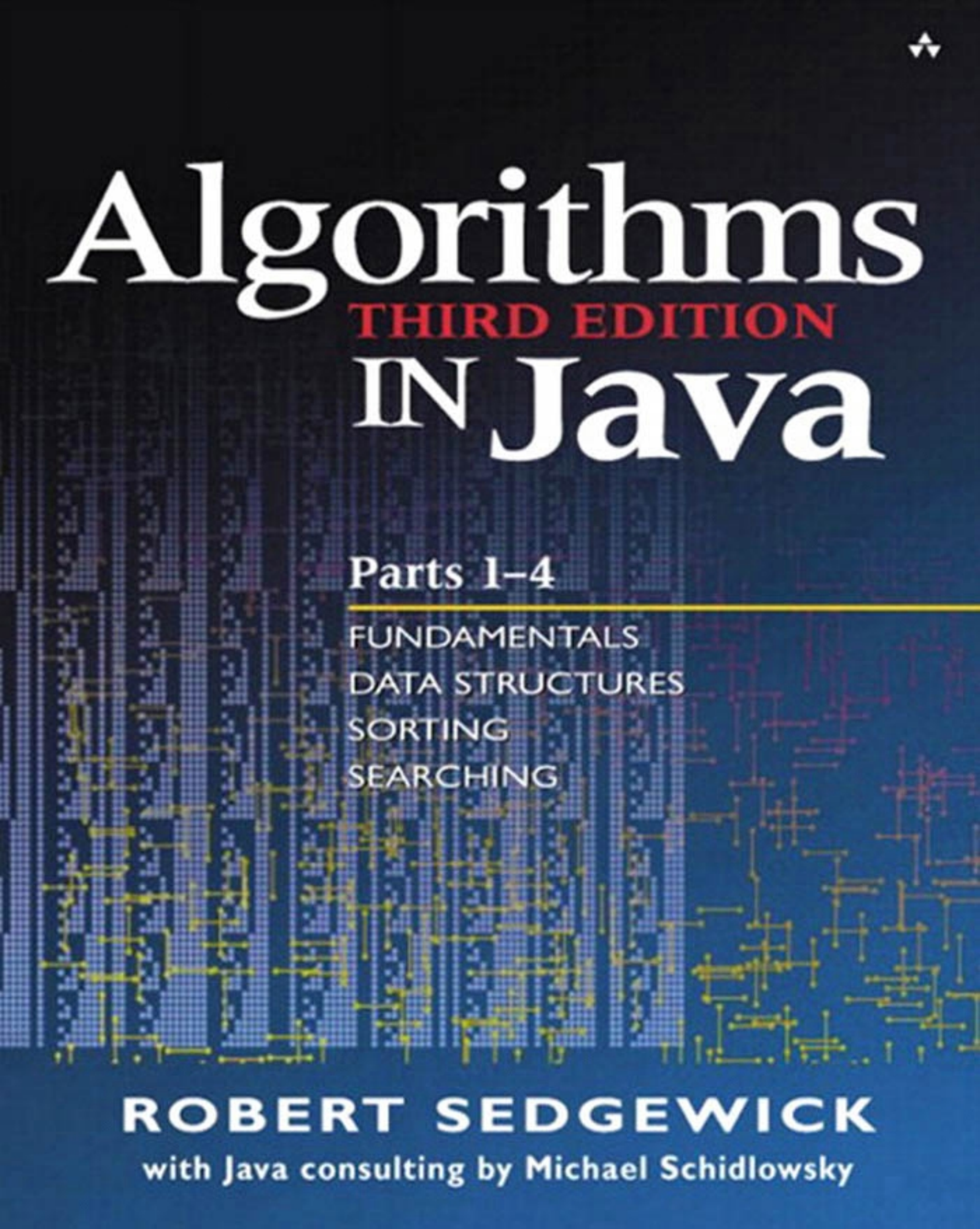

Algorithms
THIRD EDITION
IN Java
Parts 1–4
FUNDAMENTALS
DATA STRUCTURES
SORTING
SEARCHING
ROBERT SEDGEWICK
with Java consulting by Michael Schidlowsky

Algorithms
THIRD EDITION
in Java

PARTS 1–4

FUNDAMENTALS
DATA STRUCTURES
SORTING
SEARCHING

Robert Sedgewick
Princeton University

Addison-Wesley

Boston • San Francisco • New York • Toronto • Montreal
London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

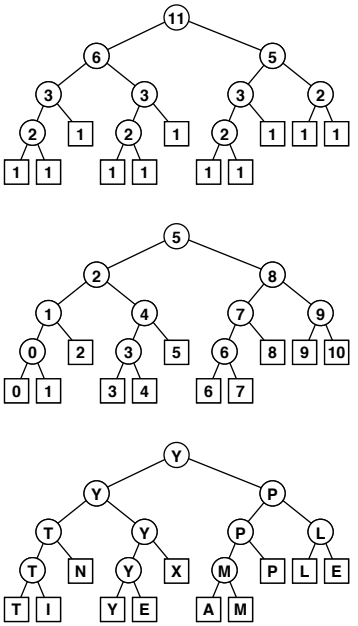


Figure 5.6
Recursive structure of find-the-maximum algorithm.

The divide-and-conquer algorithm splits a problem of size 11 into one of size 6 and one of size 5, a problem of size 6 into two problems of size 3, and so forth, until reaching problems of size 1 (top). *Each circle in these diagrams represents a call on the recursive method, to the nodes just below connected to it by lines (squares are those calls for which the recursion terminates). The diagram in the middle shows the value of the index into the middle of the file that we use to effect the split; the diagram at the bottom shows the return value.*

linear. Other divide-and-conquer algorithms may perform more work on each method invocation, as we shall see, so determining the total running time requires more intricate analysis. The running time of such algorithms depends on the precise manner of division into parts. Second, Program 5.6 is representative of divide-and-conquer algorithms for which the parts sum to make the whole. Other divide-and-conquer algorithms may divide either into smaller parts that constitute less than the whole problem or into overlapping parts that total up to more than the whole problem. These algorithms are still proper recursive algorithms because *each* part is smaller than the whole, but analyzing them is more difficult than analyzing Program 5.6. We shall consider the analysis of these different types of algorithms in detail as we encounter them.

For example, the binary-search algorithm that we studied in Section 2.6 is a divide-and-conquer algorithm that divides a problem in half, then works on just one of the halves. We examine a recursive implementation of binary search in Chapter 12.

Figure 5.5 indicates the contents of the internal stack maintained by the programming environment to support the computation in Figure 5.4. The model depicted in the figure is idealistic, but it gives useful insights into the structure of the divide-and-conquer computation. If a program has two recursive calls, the actual internal stack contains one entry corresponding to the first method invocation while that method is being executed (which contains values of arguments, local variables, and a return address), then a similar entry corresponding to the second method invocation while that method is being executed. The alternative that is depicted in Figure 5.5 is to put the two entries on the stack at once, keeping all the subtasks remaining to be done explicitly on the stack. This arrangement plainly delineates the computation and sets the stage for more general computational schemes, such as those that we examine in Sections 5.6 and 5.8.

Figure 5.6 depicts the structure of the divide-and-conquer find-the-maximum computation. It is a recursive structure: the node at the top contains the size of the input array, the structure for the left subarray is drawn at the left, and the structure for the right subarray is drawn at the right. We will formally define and discuss tree structures of this type in in Sections 5.4 and 5.5. They are useful for understanding the structure of any program involving nested method

Program 5.7 Solution to the towers of Hanoi

We shift the tower of disks to the right by (recursively) shifting all but the bottom disk to the left, then shifting the bottom disk to the right, then (recursively) shifting the tower back onto the bottom disk.

```
static void hanoi(int N, int d)
  {
    if (N == 0) return;
    hanoi(N-1, -d);
    shift(N, d);
    hanoi(N-1, -d);
  }
```

invocations—recursive programs in particular. Also shown in Figure 5.6 is the same tree, but with each node labeled with the return value for the corresponding method invocation. In Section 5.7, we shall consider the process of building explicit linked structures that represent trees like this one.

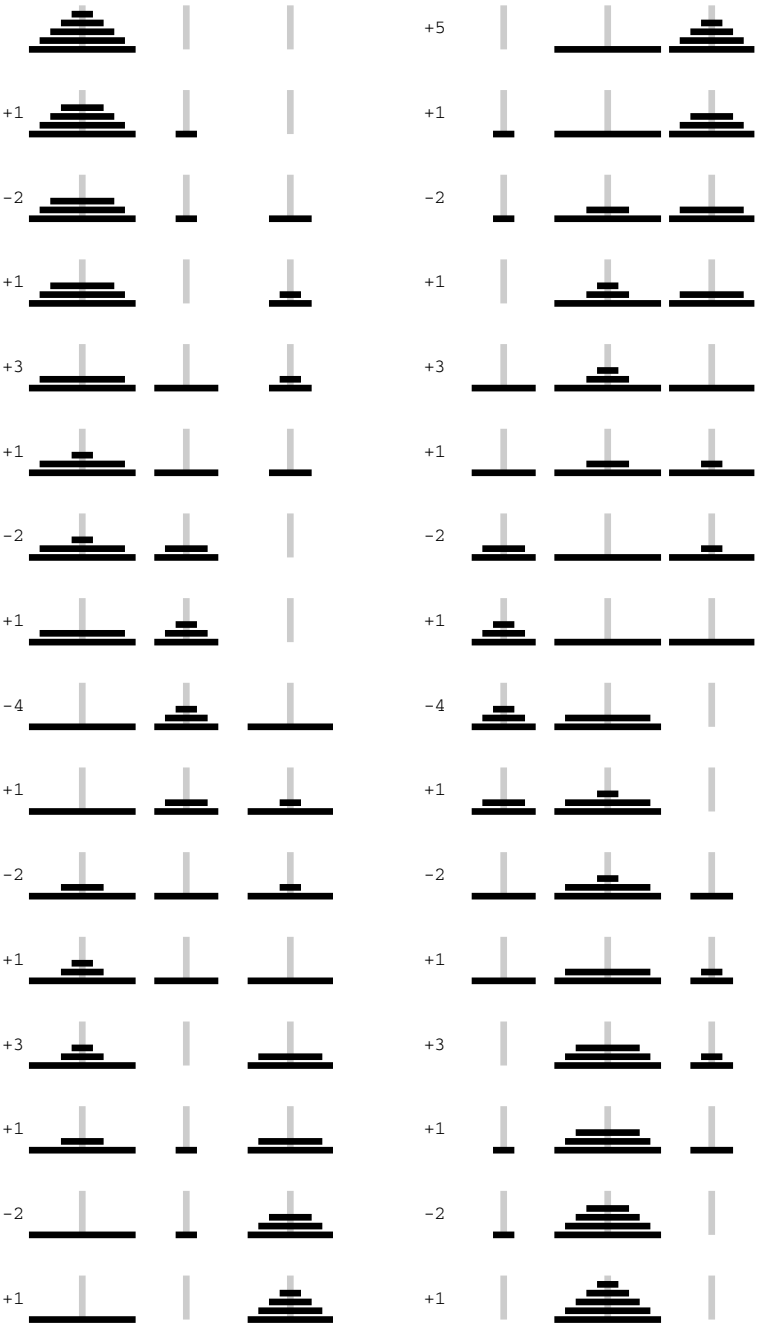
No discussion of recursion would be complete without the ancient *towers of Hanoi* problem. We have three pegs and N disks that fit onto the pegs. The disks differ in size and are initially arranged on one of the pegs, in order from largest (disk N) at the bottom to smallest (disk 1) at the top. The task is to move the stack of disks to the right one position (peg), while obeying the following rules: (*i*) only one disk may be shifted at a time; and (*ii*) no disk may be placed on top of a smaller one. One legend says that the world will end when a certain group of monks accomplishes this task in a temple with 40 golden disks on three diamond pegs.

Program 5.7 gives a recursive solution to the problem. It specifies which disk should be shifted at each step, and in which direction (+ means move one peg to the right, cycling to the leftmost peg when on the rightmost peg; and - means move one peg to the left, cycling to the rightmost peg when on the leftmost peg). The recursion is based on the following idea: To move N disks one peg to the right, we first move the top $N-1$ disks one peg to the left, then shift disk N one peg to the right, then move the $N-1$ disks one more peg to the left (onto disk N). We can verify that this solution works by induction. Figure 5.7

Figure 5.7
Towers of Hanoi

This diagram depicts the solution to the towers of Hanoi problem for five disks. We shift the top four disks left one position (left column)*, then move disk 5 to the right, then shift the top four disks left one position* (right column)*. The sequence of method invocations that follows constitutes the computation for three disks. The computed sequence of moves is* `+1 -2 +1 +3 +1 -2 +1`*, which appears four times in the solution (for example, the first seven moves).*

```
hanoi(3, +1)
  hanoi(2, -1)
    hanoi(1, +1)
      hanoi(0, -1)
      shift(1, +1)
      hanoi(0, -1)
    shift(2, -1)
    hanoi(1, +1)
      hanoi(0, -1)
      shift(1, +1)
      hanoi(0, -1)
  shift(3, +1)
  hanoi(2, -1)
    hanoi(1, +1)
      hanoi(0, -1)
      shift(1, +1)
      hanoi(0, -1)
    shift(2, -1)
    hanoi(1, +1)
      hanoi(0, -1)
      shift(1, +1)
      hanoi(0, -1)
```



shows the moves for $N = 5$ and the recursive calls for $N = 3$. An underlying pattern is evident, which we now consider in detail.

First, the recursive structure of this solution immediately tells us the number of moves that the solution requires.

Property 5.2 *The recursive divide-and-conquer algorithm for the towers of Hanoi problem produces a solution that has $2^N - 1$ moves.*

As usual, it is immediate from the code that the number of moves satisfies a recurrence. In this case, the recurrence satisfied by the number of disk moves is similar to Formula 2.5:

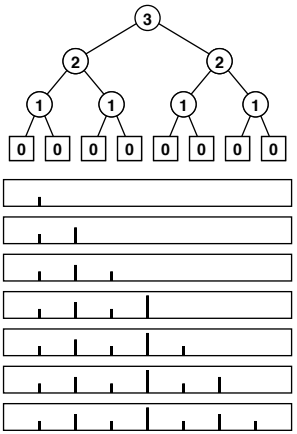
$$T_N = 2T_{N-1} + 1, \qquad \text{for } N \geq 2 \text{ with } T_1 = 1.$$

We can verify the stated result directly by induction: we have $T(1) = 2^1 - 1 = 1$; and, if $T(k) = 2^k - 1$ for $k < N$, then $T(N) = 2(2^{N-1} - 1) + 1 = 2^N - 1$. ■

If the monks are moving disks at the rate of one per second, it will take at least 348 centuries for them to finish (see Figure 2.1), assuming that they do not make a mistake. The end of the world is likely to be even further off than that because those monks presumably never have had the benefit of being able to use Program 5.7, and might not be able to figure out so quickly which disk to move next. We now consider an analysis of the method that leads to a simple (nonrecursive) method that makes the decision easy. While we may not wish to let the monks in on the secret, it is relevant to numerous important practical algorithms.

To understand the towers of Hanoi solution, let us consider the simple task of drawing the markings on a ruler. Each inch on the ruler has a mark at the $\frac{1}{2}$ inch point, slightly shorter marks at $\frac{1}{4}$ inch intervals, still shorter marks at $\frac{1}{8}$ inch intervals, and so forth. Our task is to write a program to draw these marks at any given resolution, assuming that we have at our disposal a procedure `mark(x, h)` to make a mark `h` units high at position `x`.

If the desired resolution is $\frac{1}{2^n}$ inches, we rescale so that our task is to put a mark at every point between 0 and 2^n, endpoints not included. Thus, the middle mark should be n units high, the marks in the middle of the left and right halves should be $n-1$ units high, and so forth. Program 5.8 is a straightforward divide-and-conquer algorithm to accomplish this objective; Figure 5.8 illustrates it in operation on a




```
rule(0, 8, 3)
  rule(0, 4, 2)
    rule(0, 2, 1)
      rule(0, 1, 0)
      mark(1, 1)
      rule(1, 2, 0)
    mark(2, 2)
    rule(2, 4, 1)
      rule(2, 3, 0)
      mark(3, 1)
      rule(3, 4, 0)
  mark(4, 3)
  rule(4, 8, 2)
    rule(4, 6, 1)
      rule(4, 5, 0)
      mark(5, 1)
      rule(5, 6, 0)
    mark(6, 2)
    rule(6, 8, 1)
      rule(6, 7, 0)
      mark(7, 1)
      rule(7, 8, 0)
```

Figure 5.8 Ruler-drawing method invocations

This sequence of method invocations constitutes the computation for drawing a ruler of length 8, resulting in marks of lengths 1, 2, 1, 3, 1, 2, and 1.

Program 5.8 Divide and conquer to draw a ruler

To draw the marks on a ruler, we draw the marks on the left half, then draw the longest mark in the middle, then draw the marks on the right half. This program is intended to be used with $r - l$ equal to a power of 2—a property that it preserves in its recursive calls (see Exercise 5.27).

```
static void rule(int l, int r, int h)
  { int m = (l+r)/2;
    if (h > 0)
      {
        rule(l, m, h-1);
        mark(m, h);
        rule(m, r, h-1);
      }
  }
```

small example. Recursively speaking, the idea behind the method is the following: To make the marks in an interval, we first divide the interval into two equal halves. Then, we make the (shorter) marks in the left half (recursively), the long mark in the middle, and the (shorter) marks in the right half (recursively). Iteratively speaking, Figure 5.8 illustrates that the method makes the marks in order, from left to right—the trick lies in computing the lengths. The recursion tree in the figure helps us to understand the computation: Reading down, we see that the length of the mark decreases by 1 for each recursive method invocation. Reading across, we get the marks in the order that they are drawn, because, for any given node, we first draw the marks associated with the method invocation on the left, then the mark associated with the node, then the marks associated with the method invocation on the right.

We see immediately that the sequence of lengths is precisely the same as the sequence of disks moved for the towers of Hanoi problem. Indeed, a simple proof that they are identical is that the recursive programs are the same. Put another way, our monks could use the marks on a ruler to decide which disk to move.

Moreover, both the towers of Hanoi solution in Program 5.7 and the ruler-drawing program in Program 5.8 are variants of the basic divide-and-conquer scheme exemplified by Program 5.6. All

three solve a problem of size 2^n by dividing it into two problems of size 2^{n-1}. For finding the maximum, we have a linear-time solution in the size of the input; for drawing a ruler and for solving the towers of Hanoi, we have a linear-time solution in the size of the output. For the towers of Hanoi, we normally think of the solution as being *exponential* time, because we measure the size of the problem in terms of the number of disks, n.

It is easy to draw the marks on a ruler with a recursive program, but is there some simpler way to compute the length of the ith mark, for any given i? Figure 5.9 shows yet another simple computational process that provides the answer to this question. The ith number printed out by both the towers of Hanoi program and the ruler program is nothing other than the number of trailing 0 bits in the binary representation of i. We can prove this property by induction by correspondence with a divide-and-conquer formulation for the process of printing the table of n-bit numbers: Print the table of $(n-1)$-bit numbers, each preceded by a 0 bit, then print the table of $(n-1)$-bit numbers, each preceded by a 1-bit (see Exercise 5.25).

For the towers of Hanoi problem, the implication of the correspondence with n-bit numbers is a simple algorithm for the task. We can move the pile one peg to the right by iterating the following two steps until done:

- Move the small disk to the right if n is odd (left if n is even).
- Make the only legal move not involving the small disk.

That is, after we move the small disk, the other two pegs contain two disks, one smaller than the other. The only legal move not involving the small disk is to move the smaller one onto the larger one. Every other move involves the small disk for the same reason that every other number is odd and that every other mark on the rule is the shortest. Perhaps our monks *do* know this secret, because it is hard to imagine how they might be deciding which moves to make otherwise.

A formal proof by induction that every other move in the towers of Hanoi solution involves the small disk (beginning and ending with such moves) is instructive: For $n = 1$, there is just one move, involving the small disk, so the property holds. For $n > 1$, the assumption that the property holds for $n-1$ implies that it holds for n by the recursive construction: The first solution for $n-1$ begins with a small-disk move, and the second solution for $n-1$ ends with a small-disk move,

0	0	0	0	1	
0	0	0	1	0	1
0	0	0	1	1	
0	0	1	0	0	2
0	0	1	0	1	
0	0	1	1	0	1
0	0	1	1	1	
0	1	0	0	0	3
0	1	0	0	1	
0	1	0	1	0	1
0	1	0	1	1	
0	1	1	0	0	2
0	1	1	0	1	
0	1	1	1	0	1
0	1	1	1	1	
1	0	0	0	0	4
1	0	0	0	1	
1	0	0	1	0	1
1	0	0	1	1	
1	0	1	0	0	2
1	0	1	0	1	
1	0	1	1	0	1
1	0	1	1	1	
1	1	0	0	0	3
1	1	0	0	1	
1	1	0	1	0	1
1	1	0	1	1	
1	1	1	0	0	2
1	1	1	0	1	
1	1	1	1	0	1
1	1	1	1	1	

Figure 5.9
Binary counting and the ruler function

Computing the ruler function is equivalent to counting the number of trailing zeros in the even N-bit numbers.