the same gcd, and one way to compute p % q is to subtract q from p until we get a number less than q.

The function `gcd()` in `euclid.py` (PROGRAM 2.3.1) is a compact recursive function whose reduction step is based on this property. The base case is when `q` is 0, with `gcd(p, 0) = p`. To see that the reduction step converges to the base case, observe that the value of the second argument strictly decreases in each recursive call since `p % q < q`. If `p < q`, then the first recursive call switches the two arguments. In fact, the value of the second argument decreases by at least a factor of 2 for every second recursive call, so the sequence of argument values quickly converges to the base case (see EXERCISE 2.3.13). This recursive solution to the problem of computing the greatest common divisor is known as *Euclid's algorithm* and is one of the oldest known algorithms—it is over 2,000 years old.

```
gcd(1440, 408)
   gcd(408, 216)
      gcd(216, 24)
         gcd(192, 24)
            gcd(24, 0)
               return 24
            return 24
         return 24
      return 24
   return 24
```

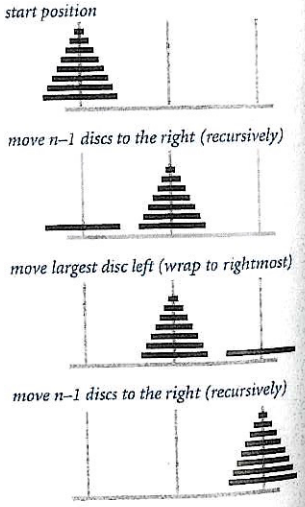
Function call trace for `gcd()`

Towers of Hanoi No discussion of recursion would be complete without the famous *towers of Hanoi* problem. In this problem, we have three poles and n discs that fit onto the poles. The discs differ in size and are initially stacked on one of the poles, in order from largest (disc n) at the bottom to smallest (disc 1) at the top. The task is to move all n discs to another pole, while obeying the following rules:

- Move only one disc at a time.
- Never place a larger disc on a smaller one.

One legend says that the world will end when a certain group of monks accomplishes this task in a temple with 64 golden discs on three diamond needles. But how can the monks accomplish the task at all, playing by the rules?

To solve the problem, our goal is to issue a sequence of instructions for moving the discs. We assume that the poles are arranged in a row, and that each instruction to move a disc specifies its number and whether to move it left or right. If a disc is on the left pole, an instruction to move left means to wrap to the right pole; if a disc is on the right pole, an instruction to move right means to wrap to the left pole. When the discs are all on one pole, there are two possible moves (move the smallest disc left or right); otherwise, there are three possible moves (move the smallest disc left or right, or make the one legal move involving the other two poles). Choosing among these possibilities on each move to achieve the goal is a challenge that requires a plan. Recursion provides just the plan that we need, based on the following idea: first we move the top $n-1$ discs to an empty pole, then we move the largest disc to the other empty pole (where it does not interfere with the smaller ones), and then we complete the job by moving the $n-1$ discs onto the largest disc. To simplify the instructions, we move discs either left or right, with *wraparound*: moving left from the leftmost pole means to move to the rightmost pole and moving right from the rightmost pole means move to the leftmost pole.

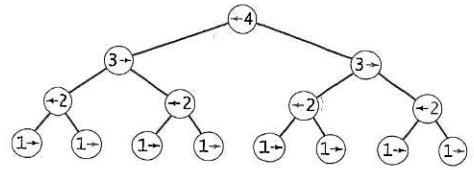


Recursive plan for towers of Hanoi

PROGRAM 2.3.2 (`towersofhanoi.py`) is a direct implementation of this strategy. It reads in a command-line argument `n` and writes the solution to the towers of Hanoi problem on `n` discs. The recursive function `moves()` writes the sequence of moves to move the stack of discs to the left (if the argument `left` is `True`) or to the right (if `left` is `False`), with wraparound. It does so exactly according to the plan just described.

Function-call trees To better understand the behavior of modular programs that have multiple recursive calls (such as `towersofhanoi.py`), we use a visual representation known as a *function-call tree*. Specifically, we represent each function call as a *tree node*, depicted as a circle labeled with the values of the arguments for that call. Below each tree node, we draw the tree nodes corresponding to each call in that use of the function (in order from left to right) and lines connecting to them. This diagram contains all the information we need to understand the behavior of the program. It contains a tree node for each function call.

We can use function-call trees to understand the behavior of any modular program, but they are particularly useful in exposing the behavior of recursive programs. For example, the tree corresponding to a call to `move()` in `towersofhanoi.py` is easy to construct. Start by drawing a tree node labeled with the values of the command-line arguments. The first argument is the number of discs in the pile to be



Function-call tree for `moves(4, true)` in `towersofhanoi.py`

Program 2.3.2 Towers of Hanoi (towersofhanoi.py)

```
import sys
import stdio

def moves(n, left):
    if n == 0: return
    moves(n-1, not left)
    if left:
        stdio.writeln(str(n) + ' left')
    else:
        stdio.writeln(str(n) + ' right')
    moves(n-1, not left)

n = int(sys.argv[1])
moves(n, True)
```

n	*number of discs*
left	*direction to move pile*

This script writes the instructions for the towers of Hanoi problem. The recursive function `moves()` writes the moves needed to move `n` discs to the left (if `left` is `True`) or to the right (if `left` is `False`).

```
% python towersofhanoi.py 1
1 left

% python towersofhanoi.py 2
1 right
2 left
1 right

% python towersofhanoi.py 3
1 left
2 right
1 left
3 left
1 left
2 right
1 left
```

```
% python towersofhanoi.py 4
1 right
2 left
1 right
3 right
1 right
2 left
1 right
4 left
1 right
2 left
1 right
3 right
1 right
2 left
1 right
```


moved (and the label of the disc to actually be moved); the second is the direction to move the pile. For clarity, we depict the direction (a boolean value) as an arrow that points left or right, since that is our interpretation of the value—the direction to move the disc. Then draw two tree nodes below with the number of discs decremented by 1 and the direction switched, and continue doing so until only nodes with labels corresponding to a first argument value 1 have no nodes below them. These nodes correspond to calls on `moves()` that do not lead to further recursive calls.

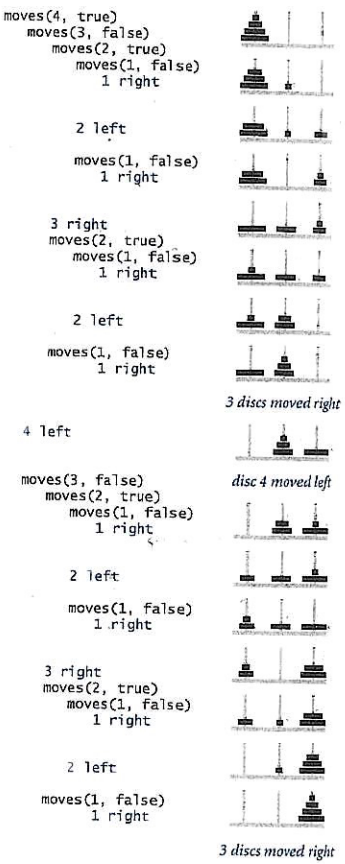
Take a moment to study the function-call tree depicted earlier in this section and to compare it with the corresponding function-call trace depicted at right. When you do so, you will see that the function-call tree is just a compact representation of the trace. In particular, reading the node labels from left to right gives the moves needed to solve the problem.

Moreover, when you study the tree, you may notice several patterns, including the following two:

- Alternate moves involve the smallest disc.
- That disc always moves in the same direction.

These observations are relevant because they give a solution to the problem that does not require recursion (or even a computer): every other move involves the smallest disc (including the first and last), and each intervening move is the only legal move at the time not involving the smallest disc. We can *prove* that this strategy produces the same outcome as the recursive program, using induction. Having started centuries ago without the benefit of a computer, perhaps our monks are using this strategy.

Trees are relevant and important in understanding recursion because the tree is the quintessential recursive object. As an abstract mathematical model, trees play an essential role in many applications, and in CHAPTER 4, we will consider the use of trees as a computational model to structure data for efficient processing.



Function-call trace for `moves(4, true)`