

less of constant factors. More generally,  $O(1)$  beats  $O(\log n)$  beats  $O(n)$  beats  $O(n \log n)$  beats  $O(n^2)$  beats  $O(n^3)$  beats  $O(2^n)$ .

## 2.4 Recursive algorithms

A *recursive algorithm* is an algorithm that calls itself.

When a recursive algorithm calls itself, it performs the same steps over again. This repetition of steps is somewhat similar to the effect we get when the steps are part of a loop. Indeed, often the same algorithm can be expressed either *iteratively* (using a loop) or *recursively*.

Analogously, a *recursive method* is a method that calls itself. Indeed, a recursive algorithm is most naturally coded in Java as a recursive method.

### EXAMPLE 2.5 *Recursive simple power algorithm*

Let us return to computing the  $n$ 'th power of  $b$ , i.e.,  $b^n$ , where  $n$  is a nonnegative integer. The definition of  $b^n$  in equation (2.1) led naturally to an iterative algorithm (Algorithm 2.3).

Here now is an alternative definition of  $b^n$ :

$$b^n = 1 \quad \text{if } n = 0 \quad (2.4a)$$

$$b^n = b \times b^{n-1} \quad \text{if } n > 0 \quad (2.4b)$$

Equation (2.4b) says that we can compute the  $n$ 'th power of  $b$  by taking the  $(n-1)$ 'th power of  $b$  and multiplying that by  $b$ . On its own, equation (2.4b) would be useless, but equation (2.4a) tells us how to compute the 0'th power of  $b$  directly. For example:

$$b^3 = b \times b^2 = b \times (b \times b^1) = b \times (b \times (b \times b^0)) = b \times (b \times (b \times 1))$$

Equations (2.4a–b) together tell us all that we need to know.

These equations lead naturally to Algorithm 2.12. Step 1 deals with the easy case,  $n = 0$ , when step 1.1 directly gives the answer 1. Step 2 deals with the hard case,  $n > 0$ , when step 2.1 computes the answer with the help of a recursive call to the same algorithm. Algorithm 2.12 is therefore recursive. (In this section, we shall highlight recursive algorithm calls by underlining.)

Will Algorithm 2.12 terminate, or will it go on calling itself forever? We can reason as follows. When called to compute the  $n$ 'th power of  $b$ , with  $n > 0$ , the algorithm will call itself to compute the  $(n-1)$ 'th power of  $b$ , which is a smaller power of  $b$ . In fact it will call itself repeatedly to compute successively smaller powers of  $b$ . Eventually it must call itself to compute the 0'th power of  $b$ , and at this point it will give a direct answer without calling itself again. Thus Algorithm 2.12 will indeed terminate.

Program 2.13 shows the recursive simple power algorithm coded as a recursive Java method.

Let us now analyze the algorithm's time efficiency. The characteristic operations are multiplications. Let  $mults(n)$  be the number of multiplications required to compute  $b^n$ .