

The most satisfactory way to measure an algorithm's time efficiency is to count *characteristic operations*. Which operations are characteristic depends on the problem to be solved. For an arithmetic algorithm it is natural to count arithmetic operations. For example, Algorithm 2.2(a) takes two additions, three subtractions, three multiplications, one division, and one square root; Algorithm 2.2(b) takes exactly the same number of arithmetic operations. In this book we shall see many examples of algorithms where comparisons or copies or other characteristic operations are the natural choice.

EXAMPLE 2.2 Power algorithms

Given a nonnegative integer n , the n th power of a number b , written b^n , is defined by:

$$b^n = b \times \cdots \times b \quad (2.1)$$

(where n copies of b are multiplied together). For example:

$$\begin{aligned} b^3 &= b \times b \times b \\ b^2 &= b \times b \\ b^1 &= b \\ b^0 &= 1 \end{aligned}$$

Algorithm 2.3 (the 'simple' power algorithm) is based directly on definition (2.1). The variable p successively takes the values 1, b , b^2 , b^3 , and so on – in other words, the successive powers of b . Program 2.4 is a Java implementation of Algorithm 2.3.

Let us now analyze Algorithm 2.3. The characteristic operations are obviously multiplications. The algorithm performs one multiplication for each iteration of the loop, and there will be n iterations, therefore:

$$\text{No. of multiplications} = n \quad (2.2)$$

Algorithm 2.3 is fine if we want to compute small powers like b^2 and b^3 , but it is very time-consuming if we want to compute larger powers like b^{20} and b^{100} .

Fortunately, there is a better algorithm. It is easy to see that $b^{20} = b^{10} \times b^{10}$. So once we know b^{10} , we can compute b^{20} with only one more multiplication, rather than ten more multiplications. This shortcut is even more effective for still larger powers: once we know b^{50} , we can compute b^{100} with only one more multiplication, rather than fifty.

Likewise, it is easy to see that $b^{21} = b^{10} \times b^{10} \times b$. So once we know b^{10} , we can compute b^{21} with only two more multiplications, rather than eleven.

Algorithm 2.5 (the 'smart' power algorithm) takes advantage of these observations. The variable q successively takes the values b , b^2 , b^4 , b^8 , and so on. At the same time, the variable m successively takes the values n , $n/2$, $n/4$, and so on (neglecting any remainders) down to 1. Whenever m has an odd value, p is multiplied by the current value of q . Program 2.6 is a Java implementation of Algorithm 2.5.

This algorithm is not easy to understand, but that is not the issue here. Instead, let us focus on analyzing its efficiency.

First of all, note that steps 2.1 and 2.2 each performs a multiplication, but the multiplication in step 2.1 is conditional. Between them, these steps perform *at most* two multiplications.