

Next, note that these steps are contained within a loop, which is iterated as often as we can halve the value of n (neglecting any remainder) until we reach zero. It can be shown (see Appendix A.2) that the number of iterations is $\text{floor}(\log_2 n) + 1$, where $\text{floor}(r)$ is the function that converts a real number r to an integer by discarding its fractional part.

Putting these points together:

$$\begin{aligned} \text{Maximum no. of multiplications} &= 2 (\text{floor}(\log_2 n) + 1) \\ &= 2 \text{floor}(\log_2 n) + 2 \end{aligned} \quad (2.3)$$

The *exact* number of multiplications depends on the value of n in a rather complicated way. For $n = 15$ the actual number of multiplications corresponds to (2.3), since halving 15 repeatedly gives a series of odd numbers; while for $n = 16$ the actual number of multiplications is smaller, since halving 16 repeatedly gives a series of even numbers. Equation (2.3) gives us the *maximum* number of multiplications for any given n , which is a pessimistic estimate.

Figure 2.7 plots (2.2) and (2.3) for comparison. The message should be clear. For small values of n , there is little to choose between the two algorithms. For larger values of n , the smart power algorithm is clearly better; indeed, its advantage grows as n grows.

2.3 Complexity of algorithms

If we want to understand the efficiency of an algorithm, we first choose characteristic operations, and then analyze the algorithm to determine the number of characteristic

To compute b^n (where n is a nonnegative integer):

1. Set p to 1.
2. For $i = 1, \dots, n$, repeat:
 - 2.1. Multiply p by b .
3. Terminate with answer p .

Algorithm 2.3 Simple power algorithm.

```
static int power (int b, int n) {
// Return the value of b raised to the n'th power (where n is a nonnegative
// integer).
    int p = 1;
    for (int i = 1; i <= n; i++)
        p *= b;
    return p;
}
```

Program 2.4 Java implementation of the simple power algorithm.