Chapter 1

Algorithms with numbers

One of the main themes of this chapter is the dramatic contrast between two ancient problems that at first seem very similar:

> Factoring: Given a number N, express it as a product of its prime factors.
>
> Primality: Given a number N, determine whether it is a prime.

Factoring is hard. Despite centuries of effort by some of the world's smartest mathematicians and computer scientists, the fastest methods for factoring a number N take time exponential in the number of bits of N.

On the other hand, we shall soon see that we *can* efficiently test whether N is prime! And (it gets even more interesting) this strange disparity between the two intimately related problems, one very hard and the other very easy, lies at the heart of the technology that enables secure communication in today's global information environment.

En route to these insights, we need to develop algorithms for a variety of computational tasks involving numbers. We begin with basic arithmetic, an especially appropriate starting point because, as we know, the word *algorithms* originally applied only to methods for these problems.

1.1 Basic arithmetic

1.1.1 Addition

We were so young when we learned the standard technique for addition that we would scarcely have thought to ask *why* it works. But let's go back now and take a closer look.

It is a basic property of decimal numbers that

> *The sum of any three single-digit numbers is at most two digits long.*

Quick check: the sum is at most $9 + 9 + 9 = 27$, two digits long. In fact, this rule holds not just in decimal but in *any* base $b \geq 2$ (Exercise 1.1). In binary, for instance, the maximum possible sum of three single-bit numbers is 3, which is a 2-bit number.

This simple rule gives us a way to add two numbers in any base: align their right-hand ends, and then perform a single right-to-left pass in which the sum is computed digit by digit, maintaining the overflow as a carry. Since we know each individual sum is a two-digit