3
Computing with large integers

In this chapter, we review standard asymptotic notation, introduce the formal computational model we shall use throughout the rest of the text, and discuss basic algorithms for computing with large integers.

3.1 Asymptotic notation

We review some standard notation for relating the rate of growth of functions. This notation will be useful in discussing the running times of algorithms, and in a number of other contexts as well.

Suppose that x is a variable taking non-negative integer or real values, and let g denote a real-valued function in x that is positive for all sufficiently large x; also, let f denote any real-valued function in x. Then

- $f = O(g)$ means that $|f(x)| \leq cg(x)$ for some positive constant c and all sufficiently large x (read, "f is big-O of g"),
- $f = \Omega(g)$ means that $f(x) \geq cg(x)$ for some positive constant c and all sufficiently large x (read, "f is big-Omega of g"),
- $f = \Theta(g)$ means that $cg(x) \leq f(x) \leq dg(x)$, for some positive constants c and d and all sufficiently large x (read, "f is big-Theta of g"),
- $f = o(g)$ means that $f/g \to 0$ as $x \to \infty$ (read, "f is little-o of g"), and
- $f \sim g$ means that $f/g \to 1$ as $x \to \infty$ (read, "f is asymptotically equal to g").

***Example* 3.1.** Let $f(x) := x^2$ and $g(x) := 2x^2 - x + 1$. Then $f = O(g)$ and $f = \Omega(g)$. Indeed, $f = \Theta(g)$. □

***Example* 3.2.** Let $f(x) := x^2$ and $g(x) := x^2 - 2x + 1$. Then $f \sim g$. □

***Example* 3.3.** Let $f(x) := 1000x^2$ and $g(x) := x^3$. Then $f = o(g)$. □

Let us call a function in x **eventually positive** if it takes positive values for all sufficiently large x. Note that by definition, if we write $f = \Omega(g)$, $f = \Theta(g)$, or $f \sim g$, it must be the case that f (in addition to g) is eventually positive; however, if we write $f = O(g)$ or $f = o(g)$, then f need not be eventually positive.

When one writes "$f = O(g)$," one should interpret "$\cdot = O(\cdot)$" as a binary relation between f with g. Analogously for "$f = \Omega(g)$," "$f = \Theta(g)$," and "$f = o(g)$."

One may also write "$O(g)$" in an expression to denote an anonymous function f such that $f = O(g)$. As an example, one could write $\sum_{i=1}^{n} i = n^2/2 + O(n)$. Analogously, $\Omega(g)$, $\Theta(g)$, and $o(g)$ may denote anonymous functions. The expression $O(1)$ denotes a function bounded in absolute value by a constant, while the expression $o(1)$ denotes a function that tends to zero in the limit.

As an even further use (abuse?) of the notation, one may use the big-O, -Omega, and -Theta notation for functions on an arbitrary domain, in which case the relevant bound should hold throughout the entire domain.

EXERCISE 3.1. Show that

(a) $f = o(g)$ implies $f = O(g)$ and $g \neq O(f)$;

(b) $f = O(g)$ and $g = O(h)$ implies $f = O(h)$;

(c) $f = O(g)$ and $g = o(h)$ implies $f = o(h)$;

(d) $f = o(g)$ and $g = O(h)$ implies $f = o(h)$.

EXERCISE 3.2. Let f and g be eventually positive functions in x. Show that

(a) $f \sim g$ if and only if $f = (1 + o(1))g$;

(b) $f \sim g$ implies $f = \Theta(g)$;

(c) $f = \Theta(g)$ if and only if $f = O(g)$ and $f = \Omega(g)$;

(d) $f = \Omega(g)$ if and only if $g = O(f)$.

EXERCISE 3.3. Let f and g be eventually positive functions in x, and suppose f/g tends to a limit L (possibly $L = \infty$) as $x \to \infty$. Show that

(a) if $L = 0$, then $f = o(g)$;

(b) if $0 < L < \infty$, then $f = \Theta(g)$;

(c) if $L = \infty$, then $g = o(f)$.

EXERCISE 3.4. Order the following functions in x so that for each adjacent pair f, g in the ordering, we have $f = O(g)$, and indicate if $f = o(g)$, $f \sim g$, or $g = O(f)$:

$$x^3,\ e^x x^2,\ 1/x,\ \ x^2(x+100) + 1/x,\ x + \sqrt{x},\ \log_2 x,\ \log_3 x,\ 2x^2,\ x,$$
$$e^{-x},\ 2x^2 - 10x + 4,\ e^{x+\sqrt{x}},\ 2^x,\ 3^x,\ x^{-2},\ x^2(\log x)^{1000}.$$

Exercise 3.5. Suppose that x takes non-negative integer values, and that $g(x) > 0$ for all $x \geq x_0$ for some x_0. Show that $f = O(g)$ if and only if $|f(x)| \leq cg(x)$ for some positive constant c and all $x \geq x_0$.

Exercise 3.6. Give an example of two non-decreasing functions f and g, both mapping positive integers to positive integers, such that $f \neq O(g)$ and $g \neq O(f)$.

Exercise 3.7. Show that

(a) the relation "$\sim$" is an equivalence relation on the set of eventually positive functions;

(b) for eventually positive functions f_1, f_2, g_2, g_2, if $f_1 \sim f_2$ and $g_1 \sim g_2$, then $f_1 \star g_1 \sim f_2 \star g_2$, where "$\star$" denotes addition, multiplication, or division;

(c) for eventually positive functions f_1, f_2, and any function g that tends to infinity as $x \to \infty$, if $f_1 \sim f_2$, then $f_1 \circ g \sim f_2 \circ g$, where "$\circ$" denotes function composition.

Exercise 3.8. Show that all of the claims in the previous exercise also hold when the relation "$\sim$" is replaced with the relation "$\cdot = \Theta(\cdot)$."

Exercise 3.9. Let f_1, f_2 be eventually positive functions. Show that if $f_1 \sim f_2$, then $\log(f_1) = \log(f_2) + o(1)$, and in particular, if $\log(f_1) = \Omega(1)$, then $\log(f_1) \sim \log(f_2)$.

Exercise 3.10. Suppose that f and g are functions defined on the integers $k, k+1, \ldots$, and that g is eventually positive. For $n \geq k$, define $F(n) := \sum_{i=k}^{n} f(i)$ and $G(n) := \sum_{i=k}^{n} g(i)$. Show that if $f = O(g)$ and G is eventually positive, then $F = O(G)$.

Exercise 3.11. Suppose that f and g are functions defined on the integers $k, k+1, \ldots$, both of which are eventually positive. For $n \geq k$, define $F(n) := \sum_{i=k}^{n} f(i)$ and $G(n) := \sum_{i=k}^{n} g(i)$. Show that if $f \sim g$ and $G(n) \to \infty$ as $n \to \infty$, then $F \sim G$.

The following two exercises are continuous variants of the previous two exercises. To avoid unnecessary distractions, we shall only consider functions that are quite "well behaved." In particular, we restrict ourselves to piece-wise continuous functions (see §A3).

Exercise 3.12. Suppose that f and g are piece-wise continuous on $[a, \infty)$, and that g is eventually positive. For $x \geq a$, define $F(x) := \int_a^x f(t)dt$ and $G(x) := \int_a^x g(t)dt$. Show that if $f = O(g)$ and G is eventually positive, then $F = O(G)$.

Exercise 3.13. Suppose that f and g are piece-wise continuous $[a, \infty)$, both of which are eventually positive. For $x \geq a$, define $F(x) := \int_a^x f(t)dt$ and $G(x) := \int_a^x g(t)dt$. Show that if $f \sim g$ and $G(x) \to \infty$ as $x \to \infty$, then $F \sim G$.

3.2 Machine models and complexity theory

When presenting an algorithm, we shall always use a high-level, and somewhat informal, notation. However, all of our high-level descriptions can be routinely translated into the machine-language of an actual computer. So that our theorems on the running times of algorithms have a precise mathematical meaning, we formally define an "idealized" computer: the **random access machine** or **RAM**.

A RAM consists of an unbounded sequence of **memory cells**

$$m[0], m[1], m[2], \ldots$$

each of which can store an arbitrary integer, together with a **program**. A program consists of a finite sequence of instructions $I_0, I_1, \ldots$, where each instruction is of one of the following types:

arithmetic This type of instruction is of the form $\alpha \leftarrow \beta \star \gamma$, where $\star$ represents one of the operations addition, subtraction, multiplication, or integer division (i.e., $\lfloor \cdot / \cdot \rfloor$). The values β and γ are of the form c, $m[a]$, or $m[m[a]]$, and α is of the form $m[a]$ or $m[m[a]]$, where c is an integer constant and a is a non-negative integer constant. Execution of this type of instruction causes the value $\beta \star \gamma$ to be evaluated and then stored in α.

branching This type of instruction is of the form IF $\beta \diamond \gamma$ GOTO i, where i is the index of an instruction, and where $\diamond$ is one of the comparison operations $=, \neq, <, >, \leq, \geq$, and β and γ are as above. Execution of this type of instruction causes the "flow of control" to pass conditionally to instruction I_i.

halt The HALT instruction halts the execution of the program.

A RAM executes by executing instruction I_0, and continues to execute instructions, following branching instructions as appropriate, until a HALT instruction is executed.

We do not specify input or output instructions, and instead assume that the input and output are to be found in memory at some prescribed location, in some standardized format.

To determine the running time of a program on a given input, we charge 1 unit of time to each instruction executed.

This model of computation closely resembles a typical modern-day computer, except that we have abstracted away many annoying details. However, there are two details of real machines that cannot be ignored; namely, any real machine has a finite number of memory cells, and each cell can store numbers only in some fixed range.

The first limitation must be dealt with by either purchasing sufficient memory or designing more space-efficient algorithms.

The second limitation is especially annoying, as we will want to perform computations with quite large integers — much larger than will fit into any single memory cell of an actual machine. To deal with this limitation, we shall represent such large integers as vectors of digits to some fixed base, so that each digit is bounded so as to fit into a memory cell. This is discussed in more detail in the next section. Using this strategy, the only other numbers we actually need to store in memory cells are "small" numbers representing array indices, addresses, and the like, which hopefully will fit into the memory cells of actual machines.

Thus, whenever we speak of an algorithm, we shall mean an algorithm that can be implemented on a RAM, such that all numbers stored in memory cells are "small" numbers, as discussed above. Admittedly, this is a bit imprecise. For the reader who demands more precision, we can make a restriction such as the following: there exist positive constants c and d, such that at any point in the computation, if k memory cells have been written to (including inputs), then all numbers stored in memory cells are bounded by $k^c + d$ in absolute value.

Even with these caveats and restrictions, the running time as we have defined it for a RAM is still only a rough predictor of performance on an actual machine. On a real machine, different instructions may take significantly different amounts of time to execute; for example, a division instruction may take much longer than an addition instruction. Also, on a real machine, the behavior of the cache may significantly affect the time it takes to load or store the operands of an instruction. Finally, the precise running time of an algorithm given by a high-level description will depend on the quality of the translation of this algorithm into "machine code." However, despite all of these problems, it still turns out that measuring the running time on a RAM as we propose here is nevertheless a good "first order" predictor of performance on real machines in many cases. Also, we shall only state the running time of an algorithm using a big-O estimate, so that implementation-specific constant factors are anyway "swept under the rug."

If we have an algorithm for solving a certain type of problem, we expect that "larger" instances of the problem will require more time to solve than "smaller" instances. Theoretical computer scientists sometimes equate the notion of an "efficient" algorithm with that of a **polynomial-time algorithm** (although not everyone takes theoretical computer scientists very seriously, especially on this point). A polynomial-time algorithm is one whose running time on inputs of length n is bounded by $n^c + d$ for some constants c and d (a "real" theoretical computer scientist will write this as $n^{O(1)}$). To make this notion mathematically precise, one needs to define the *length* of an algorithm's input.

To define the length of an input, one chooses a "reasonable" scheme to encode all possible inputs as a string of symbols from some finite alphabet, and then defines the length of an input as the number of symbols in its encoding.

We will be dealing with algorithms whose inputs consist of arbitrary integers, or lists of such integers. We describe a possible encoding scheme using the alphabet consisting of the six symbols '0', '1', '-', ',', '(', and ')'. An integer is encoded in binary, with possibly a negative sign. Thus, the length of an integer x is approximately equal to $\log_2 |x|$. We can encode a list of integers $x_1, \ldots, x_n$ as "$(\bar{x}_1, \ldots, \bar{x}_n)$", where $\bar{x}_i$ is the encoding of x_i. We can also encode lists of lists, and so on, in the obvious way. All of the mathematical objects we shall wish to compute with can be encoded in this way. For example, to encode an $n \times n$ matrix of rational numbers, we may encode each rational number as a pair of integers (the numerator and denominator), each row of the matrix as a list of n encodings of rational numbers, and the matrix as a list of n encodings of rows.

It is clear that other encoding schemes are possible, giving rise to different definitions of input length. For example, we could encode inputs in some base other than 2 (but not unary!) or use a different alphabet. Indeed, it is typical to assume, for simplicity, that inputs are encoded as bit strings. However, such an alternative encoding scheme would change the definition of input length by at most a constant multiplicative factor, and so would not affect the notion of a polynomial-time algorithm.

Note that algorithms may use data structures for representing mathematical objects that look quite different from whatever encoding scheme one might choose. Indeed, our mathematical objects may never actually be written down using our encoding scheme (either by us or our programs)—the encoding scheme is a purely conceptual device that allows us to express the running time of an algorithm as a function of the length of its input.

Also note that in defining the notion of polynomial time on a RAM, it is essential that we restrict the sizes of numbers that may be stored in the machine's memory cells, as we have done above. Without this restriction, a program could perform arithmetic on huge numbers, being charged just one unit of time for each arithmetic operation—not only is this intuitively "wrong," it is possible to come up with programs that solve some problems using a polynomial number of arithmetic operations on huge numbers, and these problems cannot otherwise be solved in polynomial time (see §3.6).

3.3 Basic integer arithmetic

We will need algorithms to manipulate integers of arbitrary length. Since such integers will exceed the word-size of actual machines, and to satisfy the formal requirements of our random access model of computation, we shall represent large integers as vectors of digits to some base B, along with a bit indicating the sign. That is, for $a \in \mathbb{Z}$, if we write

$$a = \pm \sum_{i=0}^{k-1} a_i B^i = \pm (a_{k-1} \cdots a_1 a_0)_B,$$

where $0 \leq a_i < B$ for $i = 0, \ldots, k-1$, then a will be represented in memory as a data structure consisting of the vector of base-B digits $a_0, \ldots, a_{k-1}$, along with a "sign bit" to indicate the sign of a. When a is non-zero, the high-order digit a_{k-1} in this representation should be non-zero.

For our purposes, we shall consider B to be a constant, and moreover, a power of 2. The choice of B as a power of 2 is convenient for a number of technical reasons.

A note to the reader: *If you are not interested in the low-level details of algorithms for integer arithmetic, or are willing to take them on faith, you may safely skip ahead to §3.3.5, where the results of this section are summarized.*

We now discuss in detail basic arithmetic algorithms for unsigned (i.e., non-negative) integers—these algorithms work with vectors of base-B digits, and except where explicitly noted, we do not assume the high-order digits of the input vectors are non-zero, nor do these algorithms ensure that the high-order digit of the output vector is non-zero. These algorithms can be very easily adapted to deal with arbitrary signed integers, and to take proper care that the high-order digit of the vector representing a non-zero number is non-zero (the reader is asked to fill in these details in some of the exercises below). All of these algorithms can be implemented directly in a programming language that provides a "built-in" signed integer type that can represent all integers of absolute value less than B^2, and that provides the basic arithmetic operations (addition, subtraction, multiplication, integer division). So, for example, using the *C* or *Java* programming language's `int` type on a typical 32-bit computer, we could take $B = 2^{15}$. The resulting software would be reasonably efficient, but certainly not the best possible.

Suppose we have the base-B representations of two unsigned integers a and b. We present algorithms to compute the base-B representation of $a+b$, $a-b$, $a \cdot b$, $\lfloor a/b \rfloor$, and $a \bmod b$. To simplify the presentation, for integers x, y with $y \neq 0$, we write $\operatorname{divmod}(x, y)$ to denote $(\lfloor x/y \rfloor, x \bmod y)$.

3.3.1 Addition

Let $a = (a_{k-1} \cdots a_0)_B$ and $b = (b_{\ell-1} \cdots b_0)_B$ be unsigned integers. Assume that $k \geq \ell \geq 1$ (if $k < \ell$, then we can just swap a and b). The sum $c := a+b$ is of the form $c = (c_k c_{k-1} \cdots c_0)_B$. Using the standard "paper-and-pencil" method (adapted from base-10 to base-B, of course), we can compute the base-B representation of $a+b$ in time $O(k)$, as follows:

```
$carry \leftarrow 0$
for $i \leftarrow 0$ to $\ell - 1$ do
    $tmp \leftarrow a_i + b_i + carry$,  $(carry, c_i) \leftarrow \operatorname{divmod}(tmp, B)$
for $i \leftarrow \ell$ to $k - 1$ do
    $tmp \leftarrow a_i + carry$,  $(carry, c_i) \leftarrow \operatorname{divmod}(tmp, B)$
$c_k \leftarrow carry$
```

Note that in every loop iteration, the value of *carry* is 0 or 1, and the value *tmp* lies between 0 and $2B - 1$.

3.3.2 Subtraction

Let $a = (a_{k-1} \cdots a_0)_B$ and $b = (b_{\ell-1} \cdots b_0)_B$ be unsigned integers. Assume that $k \geq \ell \geq 1$. To compute the difference $c := a - b$, we may use the same algorithm as above, but with the expression "$a_i + b_i$" replaced by "$a_i - b_i$." In every loop iteration, the value of *carry* is 0 or -1, and the value of *tmp* lies between $-B$ and $B-1$. If $a \geq b$, then $c_k = 0$ (i.e., there is no carry out of the last loop iteration); otherwise, $c_k = -1$ (and $b - a = B^k - (c_{k-1} \cdots c_0)_B$, which can be computed with another execution of the subtraction routine).

3.3.3 Multiplication

Let $a = (a_{k-1} \cdots a_0)_B$ and $b = (b_{\ell-1} \cdots b_0)_B$ be unsigned integers, with $k \geq 1$ and $\ell \geq 1$. The product $c := a \cdot b$ is of the form $(c_{k+\ell-1} \cdots c_0)_B$, and may be computed in time $O(k\ell)$ as follows:

```
for $i \leftarrow 0$ to $k + \ell - 1$ do $c_i \leftarrow 0$
for $i \leftarrow 0$ to $k - 1$ do
    $carry \leftarrow 0$
    for $j \leftarrow 0$ to $\ell - 1$ do
        $tmp \leftarrow a_i b_j + c_{i+j} + carry$
        $(carry, c_{i+j}) \leftarrow \text{divmod}(tmp, B)$
    $c_{i+\ell} \leftarrow carry$
```

Note that at every step in the above algorithm, the value of *carry* lies between 0 and $B - 1$, and the value of *tmp* lies between 0 and $B^2 - 1$.

3.3.4 Division with remainder

Let $a = (a_{k-1} \cdots a_0)_B$ and $b = (b_{\ell-1} \cdots b_0)_B$ be unsigned integers, with $k \geq 1$, $\ell \geq 1$, and $b_{\ell-1} \neq 0$. We want to compute q and r such that $a = bq + r$ and $0 \leq r < b$. Assume that $k \geq \ell$; otherwise, $a < b$, and we can just set $q \leftarrow 0$ and $r \leftarrow a$. The quotient q will have at most $m := k - \ell + 1$ base-B digits. Write $q = (q_{m-1} \cdots q_0)_B$.

At a high level, the strategy we shall use to compute q and r is the following:

```
$r \leftarrow a$
for $i \leftarrow m - 1$ down to 0 do
    $q_i \leftarrow \lfloor r/B^i b \rfloor$
    $r \leftarrow r - B^i \cdot q_i b$
```

One easily verifies by induction that at the beginning of each loop iteration, we have $0 \leq r < B^{i+1}b$, and hence each q_i will be between 0 and $B - 1$, as required.

Turning the above strategy into a detailed algorithm takes a bit of work.

In particular, we want an easy way to compute $\lfloor r/B^i b \rfloor$. Now, we could in theory just try all possible choices for q_i—this would take time $O(B\ell)$, and viewing B as a constant, this is $O(\ell)$. However, this is not really very desirable from either a practical or theoretical point of view, and we can do much better with just a little effort.

We shall first consider a special case; namely, the case where $\ell = 1$. In this case, the computation of the quotient $\lfloor r/B^i b \rfloor$ is facilitated by the following, which essentially tells us that this quotient is determined by the two high-order digits of r:

Theorem 3.1. *Let x and y be integers such that*

$$0 \le x = x'2^n + s \quad \text{and} \quad 0 < y = y'2^n$$

for some integers n, s, x', y', with $n \ge 0$ and $0 \le s < 2^n$. Then $\lfloor x/y \rfloor = \lfloor x'/y' \rfloor$.

Proof. We have

$$\frac{x}{y} = \frac{x'}{y'} + \frac{s}{y'2^n} \ge \frac{x'}{y'}.$$

It follows immediately that $\lfloor x/y \rfloor \ge \lfloor x'/y' \rfloor$.

We also have

$$\frac{x}{y} = \frac{x'}{y'} + \frac{s}{y'2^n} < \frac{x'}{y'} + \frac{1}{y'} \le \left(\left\lfloor \frac{x'}{y'} \right\rfloor + \frac{y'-1}{y'} \right) + \frac{1}{y'}.$$

Thus, we have $x/y < \lfloor x'/y' \rfloor + 1$, and hence, $\lfloor x/y \rfloor \le \lfloor x'/y' \rfloor$. □

From this theorem, one sees that the following algorithm correctly computes the quotient and remainder in time $O(k)$ (in the case $\ell = 1$):

```
$carry \leftarrow 0$
for $i \leftarrow k-1$ down to 0 do
    $tmp \leftarrow carry \cdot B + a_i$
    $(carry, q_i) \leftarrow \text{divmod}(tmp, b_0)$
output the quotient $q = (q_{k-1} \cdots q_0)_B$ and the remainder $carry$
```

Note that in every loop iteration, the value of *carry* lies between 0 and $b_0 \le B-1$, and the value of *tmp* lies between 0 and $B \cdot b_0 + (B-1) \le B^2 - 1$.

That takes care of the special case where $\ell = 1$. Now we turn to the general case $\ell \ge 1$. In this case, we cannot so easily get the digits q_i of the quotient, but we can still fairly easily estimate these digits, using the following:

Theorem 3.2. *Let x and y be integers such that*

$$0 \le x = x'2^n + s \quad \text{and} \quad 0 < y = y'2^n + t$$

for some integers n, s, t, x', y' with $n \ge 0$, $0 \le s < 2^n$, and $0 \le t < 2^n$. Further suppose that $2y' \ge x/y$. Then we have

$$\lfloor x/y \rfloor \le \lfloor x'/y' \rfloor \le \lfloor x/y \rfloor + 2.$$

Proof. For the first inequality, note that $x/y \le x/(y'2^n)$, and so $\lfloor x/y \rfloor \le \lfloor x/(y'2^n) \rfloor$, and by the previous theorem, $\lfloor x/(y'2^n) \rfloor = \lfloor x'/y' \rfloor$. That proves the first inequality.

For the second inequality, first note that from the definitions, $x/y \ge x'/(y'+1)$, which is equivalent to $x'y - xy' - x \le 0$. Now, the inequality $2y' \ge x/y$ is equivalent to $2yy' - x \ge 0$, and combining this with the inequality $x'y - xy' - x \le 0$, we obtain $2yy' - x \ge x'y - xy' - x$, which is equivalent to $x/y \ge x'/y' - 2$. It follows that $\lfloor x/y \rfloor \ge \lfloor x'/y' \rfloor - 2$. That proves the second inequality. □

Based on this theorem, we first present an algorithm for division with remainder that works assuming that b is appropriately "normalized," meaning that $b_{\ell-1} \ge 2^{w-1}$, where $B = 2^w$. This algorithm is shown in Fig. 3.1.

Some remarks are in order:

1. In line 4, we compute q_i, which by Theorem 3.2 is greater than or equal to the true quotient digit, but exceeds this value by at most 2.
2. In line 5, we reduce q_i if it is obviously too big.
3. In lines 6–10, we compute

   $$(r_{i+\ell} \cdots r_i)_B \leftarrow (r_{i+\ell} \cdots r_i)_B - q_i b.$$

   In each loop iteration, the value of *tmp* lies between $-(B^2 - B)$ and $B - 1$, and the value *carry* lies between $-(B - 1)$ and 0.
4. If the estimate q_i is too large, this is manifested by a negative value of $r_{i+\ell}$ at line 10. Lines 11–17 detect and correct this condition: the loop body here executes at most twice; in lines 12–16, we compute

   $$(r_{i+\ell} \cdots r_i)_B \leftarrow (r_{i+\ell} \cdots r_i)_B + (b_{\ell-1} \cdots b_0)_B.$$

   Just as in the algorithm in §3.3.1, in every iteration of the loop in lines 13–15, the value of *carry* is 0 or 1, and the value *tmp* lies between 0 and $2B - 1$.

It is quite easy to see that the running time of the above algorithm is $O(\ell \cdot (k - \ell + 1))$.

```
1.  for $i \leftarrow 0$ to $k-1$ do $r_i \leftarrow a_i$
2.  $r_k \leftarrow 0$
3.  for $i \leftarrow k-\ell$ down to 0 do
4.      $q_i \leftarrow \lfloor (r_{i+\ell}B + r_{i+\ell-1})/b_{\ell-1} \rfloor$
5.      if $q_i \geq B$ then $q_i \leftarrow B-1$
6.      $carry \leftarrow 0$
7.      for $j \leftarrow 0$ to $\ell-1$ do
8.          $tmp \leftarrow r_{i+j} - q_i b_j + carry$
9.          $(carry, r_{i+j}) \leftarrow \mathrm{divmod}(tmp, B)$
10.     $r_{i+\ell} \leftarrow r_{i+\ell} + carry$
11.     while $r_{i+\ell} < 0$ do
12.         $carry \leftarrow 0$
13.         for $j \leftarrow 0$ to $\ell-1$ do
14.             $tmp \leftarrow r_{i+j} + b_i + carry$
15.             $(carry, r_{i+j}) \leftarrow \mathrm{divmod}(tmp, B)$
16.         $r_{i+\ell} \leftarrow r_{i+\ell} + carry$
17.         $q_i \leftarrow q_i - 1$
18. output the quotient $q = (q_{k-\ell} \cdots q_0)_B$
        and the remainder $r = (r_{\ell-1} \cdots r_0)_B$
```

Fig. 3.1. Division with Remainder Algorithm

Finally, consider the general case, where b may not be normalized. We multiply both a and b by an appropriate value $2^{w'}$, with $0 \leq w' < w$, obtaining $a' := a2^{w'}$ and $b' := 2^{w'}$, where b' is normalized; alternatively, we can use a more efficient, special-purpose "left shift" algorithm to achieve the same effect. We then compute q and r' such that $a' = b'q + r'$, using the above division algorithm for the normalized case. Observe that $q = \lfloor a'/b' \rfloor = \lfloor a/b \rfloor$, and $r' = r2^{w'}$, where $r = a \bmod b$. To recover r, we simply divide r' by $2^{w'}$, which we can do either using the above "single precision" division algorithm, or by using a special-purpose "right shift" algorithm. All of this normalizing and denormalizing takes time $O(k + \ell)$. Thus, the total running time for division with remainder is still $O(\ell \cdot (k - \ell + 1))$.

Exercise 3.14. Work out the details of algorithms for arithmetic on *signed* integers, using the above algorithms for unsigned integers as subroutines. You should give algorithms for addition, subtraction, multiplication, and division with remainder of arbitrary signed integers (for division with remainder, your algorithm should compute $\lfloor a/b \rfloor$ and $a \bmod b$). Make sure your algorithm correctly computes the sign bit of the result, and also strips leading zero digits from the result.

EXERCISE 3.15. Work out the details of an algorithm that compares two *signed* integers a and b, determining which of $a < b$, $a = b$, or $a > b$ holds.

EXERCISE 3.16. Suppose that we run the division with remainder algorithm in Fig. 3.1 for $\ell > 1$ without normalizing b, but instead, we compute the value q_i in line 4 as follows:

$$q_i \leftarrow \lfloor (r_{i+\ell}B^2 + r_{i+\ell-1}B + r_{i+\ell-2})/(b_{\ell-1}B + b_{\ell-2}) \rfloor.$$

Show that q_i is either equal to the correct quotient digit, or the correct quotient digit plus 1. Note that a limitation of this approach is that the numbers involved in the computation are larger than B^2.

EXERCISE 3.17. Work out the details for an algorithm that shifts a given unsigned integer a to the left by a specified number of bits s (i.e., computes $b := a \cdot 2^s$). The running time of your algorithm should be linear in the number of digits of the output.

EXERCISE 3.18. Work out the details for an algorithm that shifts a given unsigned integer a to the right by a specified number of bits s (i.e., computes $b := \lfloor a/2^s \rfloor$). The running time of your algorithm should be linear in the number of digits of the output. Now modify your algorithm so that it correctly computes $\lfloor a/2^s \rfloor$ for *signed* integers a.

EXERCISE 3.19. This exercise is for *C/Java* programmers. Evaluate the *C/Java* expressions

```
(-17) % 4;  (-17) & 3;
```

and compare these values with $(-17) \bmod 4$. Also evaluate the *C/Java* expressions

```
(-17) / 4;  (-17) >> 2;
```

and compare with $\lfloor -17/4 \rfloor$. Explain your findings.

EXERCISE 3.20. This exercise is also for *C/Java* programmers. Suppose that values of type `int` are stored using a 32-bit 2's complement representation, and that all basic arithmetic operations are computed correctly modulo 2^{32}, even if an "overflow" happens to occur. Also assume that double precision floating point has 53 bits of precision, and that all basic arithmetic operations give a result with a relative error of at most 2^{-53}. Also assume that conversion from type `int` to `double` is exact, and that conversion from `double` to `int` truncates the fractional part. Now, suppose we are given `int` variables `a`, `b`, and `n`, such that $1 < \texttt{n} < 2^{30}$, $0 \leq \texttt{a} < \texttt{n}$, and $0 \leq \texttt{b} < \texttt{n}$. Show that after the following code sequence is executed, the value of `r` is equal to $(\texttt{a} \cdot \texttt{b}) \bmod \texttt{n}$:

```
int q;
q  = (int) ((((double) a) * ((double) b)) / ((double) n));
r = a*b - q*n;
if (r >= n)
   r = r - n;
else if (r < 0)
   r = r + n;
```

3.3.5 Summary

We now summarize the results of this section. For an integer a, we define $\text{len}(a)$ to be the number of bits in the binary representation of $|a|$; more precisely,

$$\text{len}(a) := \begin{cases} \lfloor \log_2 |a| \rfloor + 1 & \text{if } a \neq 0, \\ 1 & \text{if } a = 0. \end{cases}$$

Notice that for $a > 0$, if $\ell := \text{len}(a)$, then we have $\log_2 a < \ell \leq \log_2 a + 1$, or equivalently, $2^{\ell-1} \leq a < 2^{\ell}$.

Assuming that arbitrarily large integers are represented as described at the beginning of this section, with a sign bit and a vector of base-B digits, where B is a constant power of 2, we may state the following theorem.

Theorem 3.3. *Let a and b be arbitrary integers.*

(i) We can compute $a \pm b$ in time $O(\text{len}(a) + \text{len}(b))$.

(ii) We can compute $a \cdot b$ in time $O(\text{len}(a)\,\text{len}(b))$.

(iii) If $b \neq 0$, we can compute the quotient $q := \lfloor a/b \rfloor$ and the remainder $r := a \bmod b$ in time $O(\text{len}(b)\,\text{len}(q))$.

Note the bound $O(\text{len}(b)\,\text{len}(q))$ in part (iii) of this theorem, which may be significantly less than the bound $O(\text{len}(a)\,\text{len}(b))$. A good way to remember this bound is as follows: the time to compute the quotient and remainder is roughly the same as the time to compute the product bq appearing in the equality $a = bq + r$.

This theorem does not explicitly refer to the base B in the underlying implementation. The choice of B affects the values of the implied big-O constants; while in theory, this is of no significance, it does have a significant impact in practice.

From now on, we shall (for the most part) not worry about the implementation details of long-integer arithmetic, and will just refer directly this theorem. However, we will occasionally exploit some trivial aspects of our data structure for representing large integers. For example, it is clear that in constant time, we can determine the sign of a given integer a, the bit length of a, and any particular bit of the binary representation of a; moreover, as discussed in Exercises 3.17 and 3.18, multiplications and divisions by powers of 2 can be computed in linear time via "left shifts" and "right shifts." It is also clear that we can convert between the base-2 representation of a given integer and our implementation's internal representation in linear time (other conversions may take longer—see Exercise 3.25).

A note on notation: "len**" and "**log**."** In expressing the running times of algorithms, we generally prefer to write, for example, $O(\text{len}(a)\,\text{len}(b))$, rather than $O((\log a)(\log b))$. There are two reasons for this. The first is esthetic: the function "len" stresses the fact that running times should be expressed in terms of the bit length of the inputs. The second is technical: big-O estimates involving expressions containing several independent parameters, like $O(\text{len}(a)\,\text{len}(b))$, should be valid for *all* possible values of the parameters, since the notion of "sufficiently large" does not make sense in this setting; because of this, it is very inconvenient to have functions, like log, that vanish or are undefined on some inputs.

Exercise 3.21. Let $n_1, \ldots, n_k$ be positive integers. Show that

$$\sum_{i=1}^{k} \text{len}(n_i) - k \leq \text{len}\left(\prod_{i=1}^{k} n_i\right) \leq \sum_{i=1}^{k} \text{len}(n_i).$$

Exercise 3.22. Show that the product n of integers $n_1, \ldots, n_k$, with each $n_i > 1$, can be computed in time $O(\text{len}(n)^2)$. Do not assume that k is a constant.

Exercise 3.23. Show that given integers $n_1, \ldots, n_k$, with each $n_i > 1$, and an integer z, where $0 \leq z < n$ and $n := \prod_i n_i$, we can compute the k integers $z \bmod n_i$, for $i = 1, \ldots, k$, in time $O(\text{len}(n)^2)$.

Exercise 3.24. Consider the problem of computing $\lfloor n^{1/2} \rfloor$ for a given non-negative integer n.

(a) Using binary search, give an algorithm for this problem that runs in time $O(\text{len}(n)^3)$. Your algorithm should discover the bits of $\lfloor n^{1/2} \rfloor$ one at a time, from high- to low-order bit.

(b) Refine your algorithm from part (a), so that it runs in time $O(\text{len}(n)^2)$.

EXERCISE 3.25. Show how to convert (in both directions) between the base-10 representation and our implementation's internal representation of an integer n in time $O(\text{len}(n)^2)$.

3.4 Computing in $\mathbb{Z}_n$

Let $n > 1$. For $\alpha \in \mathbb{Z}_n$, there exists a unique integer $a \in \{0, \ldots, n-1\}$ such that $\alpha = [a]_n$; we call this integer a the **canonical representative of** α, and denote it by $\text{rep}(\alpha)$. For computational purposes, we represent elements of $\mathbb{Z}_n$ by their canonical representatives.

Addition and subtraction in $\mathbb{Z}_n$ can be performed in time $O(\text{len}(n))$: given $\alpha, \beta \in \mathbb{Z}_n$, to compute $\text{rep}(\alpha + \beta)$, we simply compute the integer sum $\text{rep}(\alpha) + \text{rep}(\beta)$, subtracting n if the result is greater than or equal to n; similarly, to compute $\text{rep}(\alpha - \beta)$, we compute the integer difference $\text{rep}(\alpha) - \text{rep}(\beta)$, adding n if the result is negative. Multiplication in $\mathbb{Z}_n$ can be performed in time $O(\text{len}(n)^2)$: given $\alpha, \beta \in \mathbb{Z}_n$, we compute $\text{rep}(\alpha \cdot \beta)$ as $\text{rep}(\alpha)\,\text{rep}(\beta) \bmod n$, using one integer multiplication and one division with remainder.

A note on notation: "rep," "mod," and "$[\cdot]_n$." In describing algorithms, as well as in other contexts, if α, β are elements of $\mathbb{Z}_n$, we may write, for example, $\gamma \leftarrow \alpha + \beta$ or $\gamma \leftarrow \alpha\beta$, and it is understood that elements of $\mathbb{Z}_n$ are represented by their canonical representatives as discussed above, and arithmetic on canonical representatives is done modulo n. Thus, we have in mind a "strongly typed" language for our pseudo-code that makes a clear distinction between integers in the set $\{0, \ldots, n-1\}$ and elements of $\mathbb{Z}_n$. If $a \in \mathbb{Z}$, we can convert a to an object $\alpha \in \mathbb{Z}_n$ by writing $\alpha \leftarrow [a]_n$, and if $a \in \{0, \ldots, n-1\}$, this type conversion is purely conceptual, involving no actual computation. Conversely, if $\alpha \in \mathbb{Z}_n$, we can convert α to an object $a \in \{0, \ldots, n-1\}$, by writing $a \leftarrow \text{rep}(\alpha)$; again, this type conversion is purely conceptual, and involves no actual computation. It is perhaps also worthwhile to stress the distinction between $a \bmod n$ and $[a]_n$—the former denotes an element of the set $\{0, \ldots, n-1\}$, while the latter denotes an element of $\mathbb{Z}_n$.

Another interesting problem is exponentiation in $\mathbb{Z}_n$: given $\alpha \in \mathbb{Z}_n$ and a non-negative integer e, compute $\alpha^e \in \mathbb{Z}_n$. Perhaps the most obvious way to do this is to iteratively multiply by α a total of e times, requiring time $O(e \operatorname{len}(n)^2)$. A much faster algorithm, the **repeated-squaring algorithm**, computes α^e using just $O(\operatorname{len}(e))$ multiplications in $\mathbb{Z}_n$, thus taking time $O(\operatorname{len}(e) \operatorname{len}(n)^2)$.

This method works as follows. Let $e = (b_{\ell-1} \cdots b_0)_2$ be the binary expansion of e (where b_0 is the low-order bit). For $i = 0, \ldots, \ell$, define $e_i := \lfloor e/2^i \rfloor$; the binary expansion of e_i is $e_i = (b_{\ell-1} \cdots b_i)_2$. Also define $\beta_i := \alpha^{e_i}$ for $i = 0, \ldots, \ell$, so $\beta_\ell = 1$ and $\beta_0 = \alpha^e$. Then we have

$$e_i = 2e_{i+1} + b_i \text{ and } \beta_i = \beta_{i+1}^2 \cdot \alpha^{b_i} \text{ for } i = 0, \ldots, \ell - 1.$$

This idea yields the following algorithm:

```
$\beta \leftarrow [1]_n$
for $i \leftarrow \ell - 1$ down to 0 do
    $\beta \leftarrow \beta^2$
    if $b_i = 1$ then $\beta \leftarrow \beta \cdot \alpha$
output $\beta$
```

It is clear that when this algorithm terminates, we have $\beta = \alpha^e$, and that the running-time estimate is as claimed above. Indeed, the algorithm uses ℓ squarings in $\mathbb{Z}_n$, and at most ℓ additional multiplications in $\mathbb{Z}_n$.

The following exercises develop some important efficiency improvements to the basic repeated-squaring algorithm.

EXERCISE 3.26. The goal of this exercise is to develop a "2^t-ary" variant of the above repeated-squaring algorithm, in which the exponent is effectively treated as a number in base 2^t, rather than in base 2.

(a) Show how to modify the repeated squaring so as to compute α^e using $\ell + O(1)$ squarings in $\mathbb{Z}_n$, and an additional $2^t + \ell/t + O(1)$ multiplications in $\mathbb{Z}_n$. As above, $\alpha \in \mathbb{Z}_n$ and $\operatorname{len}(e) = \ell$, while t is a parameter that we are free to choose. Your algorithm should begin by building a table of powers $[1], \alpha, \ldots, \alpha^{2^t-1}$, and after that, it should process the bits of e from left to right in blocks of length t (i.e., as base-2^t digits).

(b) Show that by appropriately choosing the parameter t, we can bound the number of additional multiplications in $\mathbb{Z}_n$ by $O(\ell/\operatorname{len}(\ell))$. Thus, from an asymptotic point of view, the cost of exponentiation is essentially the cost of ℓ squarings in $\mathbb{Z}_n$.

(c) Improve your algorithm from part (a), so that it only uses $\ell + O(1)$ squarings in $\mathbb{Z}_n$, and an additional $2^{t-1} + \ell/t + O(1)$ multiplications in $\mathbb{Z}_n$. Hint: build a table that contains only the *odd* powers of α among $[1], \alpha, \ldots, \alpha^{2^t-1}$.

EXERCISE 3.27. Suppose we are given $\alpha_1, \ldots, \alpha_k \in \mathbb{Z}_n$, along with non-negative integers $e_1, \ldots, e_k$, where $\operatorname{len}(e_i) \leq \ell$ for $i = 1, \ldots, k$. Show how to compute

$$\beta := \alpha_1^{e_1} \cdots \alpha_k^{e_k}$$

using $\ell + O(1)$ squarings in $\mathbb{Z}_n$ and an additional $\ell + 2^k + O(1)$ multiplications in $\mathbb{Z}_n$. Your algorithm should work in two phases: in the first phase, the algorithm uses just the values $\alpha_1, \ldots, \alpha_k$ to build a table of all possible products of subsets of $\alpha_1, \ldots, \alpha_k$; in the second phase, the algorithm computes β, using the exponents $e_1, \ldots, e_k$, and the table computed in the first phase.

EXERCISE 3.28. Suppose that we are to compute α^e, where $\alpha \in \mathbb{Z}_n$, for many ℓ-bit exponents e, but with α fixed. Show that for any positive integer parameter k, we can make a pre-computation (depending on α, ℓ, and k) that uses $\ell + O(1)$ squarings in $\mathbb{Z}_n$ and $2^k + O(1)$ multiplications in $\mathbb{Z}_n$, so that after the pre-computation, we can compute α^e for any ℓ-bit exponent e using just $\ell/k + O(1)$ squarings and $\ell/k + O(1)$ multiplications in $\mathbb{Z}_n$. Hint: use the algorithm in the previous exercise.

EXERCISE 3.29. Let k be a *constant*, positive integer. Suppose we are given $\alpha_1, \ldots, \alpha_k \in \mathbb{Z}_n$, along with non-negative integers $e_1, \ldots, e_k$, where $\operatorname{len}(e_i) \leq \ell$ for $i = 1, \ldots, k$. Show how to compute

$$\beta := \alpha_1^{e_1} \cdots \alpha_k^{e_k}$$

using $\ell + O(1)$ squarings in $\mathbb{Z}_n$ and an additional $O(\ell / \operatorname{len}(\ell))$ multiplications in $\mathbb{Z}_n$. Hint: develop a 2^t-ary version of the algorithm in Exercise 3.27.

EXERCISE 3.30. Let $m_1, \ldots, m_r$ be integers, each greater than 1, and let $m := m_1 \cdots m_r$. Also, for $i = 1, \ldots, r$, define $m_i' := m/m_i$. Given $\alpha \in \mathbb{Z}_n$, show how to compute all of the quantities

$$\alpha^{m_1'}, \ldots, \alpha^{m_r'}$$

using a total of $O(\operatorname{len}(r) \operatorname{len}(m))$ multiplications in $\mathbb{Z}_n$. Hint: divide and conquer.

EXERCISE 3.31. The repeated-squaring algorithm we have presented here processes the bits of the exponent from left to right (i.e., from high order to low order). Develop an algorithm for exponentiation in $\mathbb{Z}_n$ with similar complexity that processes the bits of the exponent from right to left.

3.5 Faster integer arithmetic (∗)

The quadratic-time algorithms presented in §3.3 for integer multiplication and division are by no means the fastest possible. The next exercise develops a faster multiplication algorithm.

Exercise 3.32. Suppose we have two positive, ℓ-bit integers a and b such that $a = a_1 2^k + a_0$ and $b = b_1 2^k + b_0$, where $0 \le a_0 < 2^k$ and $0 \le b_0 < 2^k$. Then

$$ab = a_1 b_1 2^{2k} + (a_0 b_1 + a_1 b_0) 2^k + a_0 b_0.$$

Show how to compute the product ab in time $O(\ell)$, given the products $a_0 b_0$, $a_1 b_1$, and $(a_0 - a_1)(b_0 - b_1)$. From this, design a recursive algorithm that computes ab in time $O(\ell^{\log_2 3})$. (Note that $\log_2 3 \approx 1.58$.)

The algorithm in the previous is also not the best possible. In fact, it is possible to multiply ℓ-bit integers *on a RAM* in time $O(\ell)$, but we do not explore this any further here (see §3.6).

The following exercises explore the relationship between integer multiplication and related problems. We assume that we have an algorithm that multiplies two integers of at most ℓ bits in time $M(\ell)$. It is convenient (and reasonable) to assume that M is a **well-behaved complexity function**. By this, we mean that M maps positive integers to positive real numbers, and

- for all positive integers a and b, we have $M(a + b) \ge M(a) + M(b)$, and
- for all real $c > 1$ there exists real $d > 1$, such that for all positive integers a and b, if $a \le cb$, then $M(a) \le dM(b)$.

Exercise 3.33. Let $\alpha > 0$, $\beta \ge 1$, $\gamma \ge 0$, $\delta \ge 0$ be real constants. Show that

$$M(\ell) := \alpha \ell^{\beta} \operatorname{len}(\ell)^{\gamma} \operatorname{len}(\operatorname{len}(\ell))^{\delta}$$

is a well-behaved complexity function.

Exercise 3.34. Give an algorithm for Exercise 3.22 that runs in time

$$O(M(\operatorname{len}(n)) \operatorname{len}(k)).$$

Hint: divide and conquer.

Exercise 3.35. We can represent a "floating point" number $\hat{z}$ as a pair (a, e), where a and e are integers — the value of $\hat{z}$ is the rational number

$a2^e$, and we call $\mathrm{len}(a)$ the **precision** of $\hat{z}$. We say that $\hat{z}$ is a **k-bit approximation** of a real number z if $\hat{z}$ has precision k and $\hat{z} = (1+\epsilon)z$ for some $|\epsilon| \leq 2^{-k+1}$. Show how to compute—given positive integers b and k—a k-bit approximation of $1/b$ in time $O(M(k))$. Hint: using Newton iteration, show how to go from a t-bit approximation of $1/b$ to a $(2t-2)$-bit approximation of $1/b$, making use of just the high-order $O(t)$ bits of b, in time $O(M(t))$. **Newton iteration** is a general method of iteratively approximating a root of an equation $f(x) = 0$ by starting with an initial approximation x_0, and computing subsequent approximations by the formula $x_{i+1} = x_i - f(x_i)/f'(x_i)$, where $f'(x)$ is the derivative of $f(x)$. For this exercise, apply Newton iteration to the function $f(x) = x^{-1} - b$.

EXERCISE 3.36. Using the result of the previous exercise, given positive integers a and b of bit length at most ℓ, show how to compute $\lfloor a/b \rfloor$ and $a \bmod b$ in time $O(M(\ell))$. From this, we see that up to a constant factor, division with remainder is no harder that multiplication.

EXERCISE 3.37. Using the result of the previous exercise, give an algorithm for Exercise 3.23 that runs in time $O(M(\mathrm{len}(n))\,\mathrm{len}(k))$. Hint: divide and conquer.

EXERCISE 3.38. Give an algorithm for Exercise 3.24 that runs in time $O(M(\mathrm{len}(n)))$. Hint: Newton iteration.

EXERCISE 3.39. Give algorithms for Exercise 3.25 that run in time $O(M(\ell)\,\mathrm{len}(\ell))$, where $\ell := \mathrm{len}(n)$. Hint: divide and conquer.

EXERCISE 3.40. Suppose we have an algorithm that computes the square of an ℓ-bit integer in time $S(\ell)$, where S is a well-behaved complexity function. Show how to use this algorithm to compute the product of two arbitrary integers of at most ℓ bits in time $O(S(\ell))$.

3.6 Notes

Shamir [84] shows how to factor an integer in polynomial time on a RAM, but where the numbers stored in the memory cells may have exponentially many bits. As there is no known polynomial-time factoring algorithm on any realistic machine, Shamir's algorithm demonstrates the importance of restricting the sizes of numbers stored in the memory cells of our RAMs to keep our formal model realistic.

The most practical implementations of algorithms for arithmetic on large integers are written in low-level "assembly language," specific to a particular machine's architecture (e.g., the GNU Multi-Precision library GMP, available at `www.swox.com/gmp`). Besides the general fact that such hand-crafted code is more efficient than that produced by a compiler, there is another, more important reason for using such code. A typical 32-bit machine often comes with instructions that allow one to compute the 64-bit product of two 32-bit integers, and similarly, instructions to divide a 64-bit integer by a 32-bit integer (obtaining both the quotient and remainder). However, high-level programming languages do not (as a rule) provide any access to these low-level instructions. Indeed, we suggested in §3.3 using a value for the base B of about half the word-size of the machine, so as to avoid overflow. However, if one codes in assembly language, one can take B to be much closer to, or even equal to, the word-size of the machine. Since our basic algorithms for multiplication and division run in time quadratic in the number of base-B digits, the effect of doubling the bit-length of B is to decrease the running time of these algorithms by a factor of *four*. This effect, combined with the improvements one might typically expect from using assembly-language code, can easily lead to a five- to ten-fold decrease in the running time, compared to an implementation in a high-level language. This is, of course, a significant improvement for those interested in serious "number crunching."

The "classical," quadratic-time algorithms presented here for integer multiplication and division are by no means the best possible: there are algorithms that are asymptotically faster. We saw this in the algorithm in Exercise 3.32, which was originally invented by Karatsuba [52] (although Karatsuba is one of two authors on this paper, the paper gives exclusive credit for this particular result to Karatsuba). That algorithm allows us to multiply two ℓ-bit integers in time $O(\ell^{\log_2 3})$. The fastest known algorithm for multiplying two ℓ-bit integers on a RAM runs in time $O(\ell)$. This algorithm is due to Schönhage, and actually works on a very restricted type of RAM called a "pointer machine" (see Problem 12, Section 4.3.3 of Knuth [54]). See Exercise 18.27 later in this text for a much simpler (but heuristic) $O(\ell)$ multiplication algorithm.

Another model of computation is that of **Boolean circuits**. In this model of computation, one considers families of Boolean circuits (with, say, the usual "and," "or," and "not" gates) that compute a particular function—for every input length, there is a different circuit in the family that computes the function on inputs of that length. One natural notion of complexity for such circuit families is the **size** of the circuit (i.e., the number of gates and wires in the circuit), which is measured as a function of the input length. The smallest known Boolean circuit that multiplies two ℓ-bit numbers has size $O(\ell \operatorname{len}(\ell) \operatorname{len}(\operatorname{len}(\ell)))$. This result is due to Schönhage and Strassen [82].

It is hard to say which model of computation, the RAM or circuits, is "better." On the one hand, the RAM very naturally models computers as we know them today: one stores small numbers, like array indices, counters, and pointers, in individual words of the machine, and processing such a number typically takes a single "machine cycle." On the other hand, the RAM model, as we formally defined it, invites a certain kind of "cheating," as it allows one to stuff $O(\operatorname{len}(\ell))$-bit integers into memory cells. For example, even with the simple, quadratic-time algorithms for integer arithmetic discussed in §3.3, we can choose the base B to have $\operatorname{len}(\ell)$ bits, in which case these algorithms would run in time $O((\ell/\operatorname{len}(\ell))^2)$. However, just to keep things simple, we have chosen to view B as a constant (from a formal, asymptotic point of view).

In the remainder of this text, unless otherwise specified, we shall always use the classical $O(\ell^2)$ bounds for integer multiplication and division, which have the advantage of being both simple and reasonably reliable predictors of actual performance for small to moderately sized inputs. For relatively large numbers, experience shows that the classical algorithms are definitely not the best—Karatsuba's multiplication algorithm, and related algorithms for division, start to perform significantly better than the classical algorithms on inputs of a thousand bits or so (the exact crossover depends on myriad implementation details). The even "faster" algorithms discussed above are typically not interesting unless the numbers involved are truly huge, of bit length around 10^5–10^6. Thus, the reader should bear in mind that for serious computations involving very large numbers, the faster algorithms are very important, even though this text does not discuss them at great length.

For a good survey of asymptotically fast algorithms for integer arithmetic, see Chapter 9 of Crandall and Pomerance [30], as well as Chapter 4 of Knuth [54].