3

Computing with large integers

In this chapter, we review standard asymptotic notation, introduce the formal computational model that 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.

Let f and g be real-valued functions. We shall assume that each is defined on the set of non-negative integers, or, alternatively, that each is defined on the set of non-negative reals. Actually, as we are only concerned about the behavior of $f(x)$ and $g(x)$ as $x \to \infty$, we only require that $f(x)$ and $g(x)$ are defined for all sufficiently large x (the phrase "for all sufficiently large x" means "for some x_0 and all $x \geq x_0$"). We further assume that g is **eventually positive**, meaning that $g(x) > 0$ for all sufficiently large 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 $f(x)/g(x) \to 0$ as $x \to \infty$ (read, "f is little-o of g"), and
- $f \sim g$ means that $f(x)/g(x) \to 1$ as $x \to \infty$ (read, "f is asymptotically equal to g").

Example 3.1. Let $f(x) := x^2$ and $g(x) := 2x^2 - 10x + 1$. Then $f = O(g)$ and $f = \Omega(g)$. Indeed, $f = \Theta(g)$. □

Example 3.2. Let $f(x) := x^2$ and $g(x) := x^2 - 10x + 1$. Then $f \sim g$. □

Example 3.3. Let $f(x) := 100x^2$ and $g(x) := x^3$. Then $f = o(g)$. □

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 $f = O(g)$. 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.

Example 3.4. Let $f(x) := x^3 - 2x^2 + x - 3$. One could write $f(x) = x^3 + O(x^2)$. Here, the anonymous function is $g(x) := -2x^2 + x - 3$, and clearly $g(x) = O(x^2)$. One could also write $f(x) = x^3 - (2 + o(1))x^2$. Here, the anonymous function is $g(x) := -1/x + 3/x^2$. While $g = o(1)$, it is only defined for $x > 0$. This is acceptable, since we will only regard statements such as this asymptotically, as $x \to \infty$. □

As an even further use (abuse?) of the notation, one may use the big-O, big-Omega, and big-Theta notation for functions on an arbitrary domain, in which case the relevant inequalities should hold throughout the entire domain. This usage includes functions of several independent variables, as well as functions defined on sets with no natural ordering.

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. 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. Suppose $f_1 = O(g_1)$ and $f_2 = O(g_2)$. Show that $f_1 + f_2 = O(\max(g_1, g_2))$, $f_1 f_2 = O(g_1 g_2)$, and that for every constant c, $cf_1 = O(g_1)$.

EXERCISE 3.4. Suppose that $f(x) \leq c + dg(x)$ for some positive constants c and d, and for all sufficiently large x. Show that if $g = \Omega(1)$, then $f = O(g)$.

EXERCISE 3.5. Suppose f and g are defined on the integers $i \geq k$, and that $g(i) > 0$ for all $i \geq k$. Show that if $f = O(g)$, then there exists a positive constant c such that $|f(i)| \leq cg(i)$ for all $i \geq k$.

EXERCISE 3.6. Let f and g be eventually positive functions, and assume that $f(x)/g(x)$ 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 3.7. Let $f(x) := x^\alpha (\log x)^\beta$ and $g(x) := x^\gamma (\log x)^\delta$, where $\alpha, \beta, \gamma, \delta$ are non-negative constants. Show that if $\alpha < \gamma$, or if $\alpha = \gamma$ and $\beta < \delta$, then $f = o(g)$.

EXERCISE 3.8. 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 3.9. Show that:

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

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

(c) for all eventually positive functions f, g, and every $\alpha > 0$, if $f \sim g$, then $f^\alpha \sim g^\alpha$;

(d) for all eventually positive functions f, g, and every function h such that $h(x) \to \infty$ as $x \to \infty$, if $f \sim g$, then $f \circ h \sim g \circ h$, where "$\circ$" denotes function composition.

EXERCISE 3.10. 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 3.11. Let f, g be eventually positive functions. Show that:

(a) $f = \Theta(g)$ if and only if $\log f = \log g + O(1)$;

(b) $f \sim g$ if and only if $\log f = \log g + o(1)$.

EXERCISE 3.12. 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 3.13. Suppose that f and g are piece-wise continuous on $[a, \infty)$ (see §A4), 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 3.14. Suppose that f and g are functions defined on the integers $k, k+1, \ldots$, and that both f and g 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 $F \sim G$.

EXERCISE 3.15. Suppose that f and g are piece-wise continuous on $[a, \infty)$ (see §A4), and that both f and g 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 $F \sim G$.

EXERCISE 3.16. Give an example of two non-decreasing functions f and g, each mapping positive integers to positive integers, such that $f \neq O(g)$ and $g \neq O(f)$.

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 $\gamma \leftarrow \alpha \star \beta$, where $\star$ represents one of the operations addition, subtraction, multiplication, or integer division (i.e., $\lfloor \cdot / \cdot \rfloor$). The values α and β are of the form c, $m[a]$, or $m[m[a]]$, and γ 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 $\alpha \star \beta$ to be evaluated and then stored in γ.

branching This type of instruction is of the form IF $\alpha \diamond \beta$ GOTO i, where i is the index of an instruction, and where $\diamond$ is one of the comparison operations $=, \neq, <, >, \leq, \geq$, and α and β 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 works by executing instruction I_0, and continues to execute instructions, following branching instructions as appropriate, until a HALT instruction is reached.

We do not specify input or output instructions, and instead assume that the input and output are to be found in memory cells at some prescribed locations, 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 in some fixed base, so that each digit is bounded in order to fit into a memory cell. This is discussed in more detail in the next section. The only other numbers we actually need to store in memory cells are "small" numbers representing array indices, counters, and the like, which we hope will fit into the memory cells of actual machines. Below, we shall make a more precise, formal restriction on the magnitude of numbers that may be stored in memory cells.

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 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 problem, we expect that "larger" instances of the problem will require more time to solve than "smaller" instances, and a general goal in the analysis of any algorithm is to estimate the rate of growth of the running time of the algorithm as a function of the size of its input. For this purpose, we shall simply measure the **size** of an input as the number of memory cells used to represent it. Theoretical computer scientists sometimes equate the notion of "efficient" with "polynomial time" (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 size n is at most $an^b + c$, for some constants a, b, and c (a "real" theoretical computer scientist will write this as $n^{O(1)}$). Furthermore, we also require that for a polynomial-time algorithm, all numbers stored in memory are at most $a'n^{b'} + c'$ in absolute value, for some constants a', b', and c'. Even for algorithms that are not polynomial time, we shall insist that after executing t instructions, all numbers stored in memory are at most $a'(n+t)^{b'} + c'$ in absolute value, for some constants a', b', and c'.

Note that in defining the notion of polynomial time on a RAM, it is essential that we restrict the magnitude 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 for performing arithmetic on very large integers. 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 in 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. To ensure a unique representation, if a is non-zero, then 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 that 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 itself 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 supports 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 and portable, but certainly not the fastest 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 denote by $\mathrm{QuoRem}(x, y)$ the quotient/remainder pair $(\lfloor x/y \rfloor, x \bmod y)$.

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

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
        $tmp \leftarrow a_i b_j + c_{i+j} + carry$
        $(carry, c_{i+j}) \leftarrow \text{QuoRem}(tmp, B)$
    $c_{i+\ell} \leftarrow carry$

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 theorem, 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*

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

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

Proof. We have

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

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

We also have

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

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

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

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

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

That takes care of the special case where $\ell = 1$. Now we turn to the general case $\ell \geq 1$. 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*

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

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

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

Proof. We have $x/y \leq x/y'2^n$, and so $\lfloor x/y \rfloor \leq \lfloor x/y'2^n \rfloor$, 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, we have $x/y \geq x'/(y'+1)$, which implies $x'y - xy' - x \leq 0$. Further, $2y' \geq x/y$ implies $2yy' - x \geq 0$. So we have $2yy' - x \geq 0 \geq x'y - xy' - x$, which implies $x/y \geq x'/y' - 2$, and hence $\lfloor x/y \rfloor \geq \lfloor x'/y' \rfloor - 2$. □

Based on this theorem, we first present an algorithm for division with remainder that works if we assume that b is appropriately "normalized," meaning that $b_{\ell-1} \geq 2^{w-1}$, 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.

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. $\quad q_i \leftarrow \lfloor (r_{i+\ell}B + r_{i+\ell-1})/b_{\ell-1} \rfloor$
5. $\quad$ if $q_i \geq B$ then $q_i \leftarrow B-1$
6. $\quad carry \leftarrow 0$
7. $\quad$ for $j \leftarrow 0$ to $\ell-1$ do
8. $\quad\quad tmp \leftarrow r_{i+j} - q_i b_j + carry$
9. $\quad\quad (carry, r_{i+j}) \leftarrow \text{QuoRem}(tmp, B)$
10. $\quad r_{i+\ell} \leftarrow r_{i+\ell} + carry$
11. $\quad$ while $r_{i+\ell} < 0$ do
12. $\quad\quad carry \leftarrow 0$
13. $\quad\quad$ for $j \leftarrow 0$ to $\ell-1$ do
14. $\quad\quad\quad tmp \leftarrow r_{i+j} + b_i + carry$
15. $\quad\quad\quad (carry, r_{i+j}) \leftarrow \text{QuoRem}(tmp, B)$
16. $\quad\quad r_{i+\ell} \leftarrow r_{i+\ell} + carry$
17. $\quad\quad q_i \leftarrow q_i - 1$
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

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 easily verified that the running time of the above algorithm is $O(\ell \cdot (k-\ell+1))$.

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 $b' := b2^{w'}$, 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 division algorithm in Fig. 3.1. 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 3.17. 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 algorithms correctly compute the sign bit of the results, and also strip any leading zero digits from the results.

EXERCISE 3.18. 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 3.19. 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 3.20. 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 3.21. 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 3.22. 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 3.23. 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 $1 < \mathtt{n} < 2^{30}$, $0 \le \mathtt{a} < \mathtt{n}$, and $0 \le \mathtt{b} < \mathtt{n}$. Show that after the following code sequence is executed, the value of `r` is equal to $(\mathtt{a} \cdot \mathtt{b}) \bmod \mathtt{n}$:

```
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 its **bit length**, or simply, its **length**, which we denote by $\mathrm{len}(a)$, to be the number of bits in the binary representation of $|a|$; more precisely,

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

If $\mathrm{len}(a) = \ell$, we say that a is an **ℓ-bit integer**. Notice that if a is a positive, ℓ-bit integer, then $\log_2 a < \ell \le \log_2 a + 1$, or equivalently, $2^{\ell-1} \le a < 2^{\ell}$.

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 $O(\mathrm{len}(a) + \mathrm{len}(b))$.

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

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

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 to 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 3.20 and 3.21, 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 3.32).

We wish to stress the point that efficient algorithms on large integers should run in time bounded by a polynomial in the *bit lengths* of the inputs, rather than their *magnitudes*. For example, if the input to an algorithm is an ℓ-bit integer n, and if the algorithm runs in time $O(\ell^2)$, it will easily be able to process 1000-bit inputs in a reasonable amount of time (a fraction of a second) on a typical, modern computer. However, if the algorithm runs in time, say, $O(n^{1/2})$, this means that on 1000-bit inputs, it will take roughly 2^{500} computing steps, which even on the fastest computer available today or in the foreseeable future, will still be running long after our solar system no longer exists.

> **A note on notation: "len" and "log."** In expressing the running times of algorithms in terms of an input a, we generally prefer to write $\text{len}(a)$ rather than $\log a$. One reason is esthetic: writing $\text{len}(a)$ stresses the fact that the running time is a function of the bit length of a. Another reason is technical: for big-O estimates involving functions on an arbitrary domain, the appropriate inequalities should hold throughout the domain, and for this reason, it is very inconvenient to use functions, like log, which vanish or are undefined on some inputs.

EXERCISE 3.24. Let $a, b \in \mathbb{Z}$ with $a \geq b > 0$, and let $q := \lfloor a/b \rfloor$. Show that $\text{len}(a) - \text{len}(b) - 1 \leq \text{len}(q) \leq \text{len}(a) - \text{len}(b) + 1$.

EXERCISE 3.25. Let $n_1, \ldots, n_k$ be positive integers. Show that

$$\sum_{i=1}^{k} \mathrm{len}(n_i) - k \leq \mathrm{len}\Big(\prod_{i=1}^{k} n_i\Big) \leq \sum_{i=1}^{k} \mathrm{len}(n_i).$$

EXERCISE 3.26. Show that given integers $n_1, \ldots, n_k$, with each $n_i > 1$, we can compute the product $n := \prod_i n_i$ in time $O(\mathrm{len}(n)^2)$.

EXERCISE 3.27. Show that given integers $a, n_1, \ldots, n_k$, with each $n_i > 1$, where $0 \leq a < n := \prod_i n_i$, we can compute $(a \bmod n_1, \ldots, a \bmod n_k)$ in time $O(\mathrm{len}(n)^2)$.

EXERCISE 3.28. Show that given integers $n_1, \ldots, n_k$, with each $n_i > 1$, we can compute $(n/n_1, \ldots, n/n_k)$, where $n := \prod_i n_i$, in time $O(\mathrm{len}(n)^2)$.

EXERCISE 3.29. This exercise develops an algorithm to compute $\lfloor \sqrt{n} \rfloor$ for a given positive integer n. Consider the following algorithm:

$k \leftarrow \lfloor (\mathrm{len}(n) - 1)/2 \rfloor$, $m \leftarrow 2^k$
for $i \leftarrow k - 1$ down to 0 do
    if $(m + 2^i)^2 \leq n$ then $m \leftarrow m + 2^i$
output m

(a) Show that this algorithm correctly computes $\lfloor \sqrt{n} \rfloor$.

(b) In a straightforward implementation of this algorithm, each loop iteration takes time $O(\mathrm{len}(n)^2)$, yielding a total running time of $O(\mathrm{len}(n)^3)$. Give a more careful implementation, so that each loop iteration takes time $O(\mathrm{len}(n))$, yielding a total running time is $O(\mathrm{len}(n)^2)$.

EXERCISE 3.30. Modify the algorithm in the previous exercise so that given positive integers n and e, with $n \geq 2^e$, it computes $\lfloor n^{1/e} \rfloor$ in time $O(\mathrm{len}(n)^3/e)$.

EXERCISE 3.31. An integer $n > 1$ is called a **perfect power** if $n = a^b$ for some integers $a > 1$ and $b > 1$. Using the algorithm from the previous exercise, design an efficient algorithm that determines if a given n is a perfect power, and if it is, also computes a and b such that $n = a^b$, where $a > 1$, $b > 1$, and a is as small as possible. Your algorithm should run in time $O(\ell^3 \,\mathrm{len}(\ell))$, where $\ell := \mathrm{len}(n)$.

EXERCISE 3.32. Show how to convert (in both directions) in time $O(\mathrm{len}(n)^2)$ between the base-10 representation and our implementation's internal representation of an integer n.

3.4 Computing in $\mathbb{Z}_n$

Let n be a positive integer. For every $\alpha \in \mathbb{Z}_n$, 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 $\mathrm{rep}(\alpha)$. 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 $O(\mathrm{len}(n))$: given $\alpha, \beta \in \mathbb{Z}_n$, to compute $\mathrm{rep}(\alpha+\beta)$, we first compute the integer sum $\mathrm{rep}(\alpha)+\mathrm{rep}(\beta)$, and then subtract n if the result is greater than or equal to n; similarly, to compute $\mathrm{rep}(\alpha-\beta)$, we compute the integer difference $\mathrm{rep}(\alpha)-\mathrm{rep}(\beta)$, adding n if the result is negative. Multiplication in $\mathbb{Z}_n$ can be performed in time $O(\mathrm{len}(n)^2)$: given $\alpha, \beta \in \mathbb{Z}_n$, we compute $\mathrm{rep}(\alpha \cdot \beta)$ as $\mathrm{rep}(\alpha)\,\mathrm{rep}(\beta) \bmod n$, 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 $a \leftarrow \mathrm{rep}(\alpha)$; 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 $O(e\ \mathrm{len}(n)^2)$. For small values of e, this is fine; however, a much faster algorithm, the **repeated-squaring algorithm**, computes α^e using just $O(\mathrm{len}(e))$ multiplications in $\mathbb{Z}_n$, thus taking time $O(\mathrm{len}(e)\ \mathrm{len}(n)^2)$.

This method is based on the following observation. 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 observation yields the following algorithm for computing α^e:

The repeated-squaring algorithm. On input α, e, where $\alpha \in \mathbb{Z}_n$ and e is a non-negative integer, do the following, where $e = (b_{\ell-1} \cdots b_0)_2$ is the binary expansion of e:

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

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 $\mathbb{Z}_n$.

Example 3.5. Suppose $e = 37 = (100101)_2$. The above algorithm performs the following operations in this case:

	// computed exponent (in binary)
$\beta \leftarrow [1]$	*// 0*
$\beta \leftarrow \beta^2, \beta \leftarrow \beta \cdot \alpha$	*// 1*
$\beta \leftarrow \beta^2$	*// 10*
$\beta \leftarrow \beta^2$	*// 100*
$\beta \leftarrow \beta^2, \beta \leftarrow \beta \cdot \alpha$	*// 1001*
$\beta \leftarrow \beta^2$	*// 10010*
$\beta \leftarrow \beta^2, \beta \leftarrow \beta \cdot \alpha$	*// 100101* . □

The repeated-squaring algorithm has numerous applications. We mention a few here, but we will see many more later on.

Computing multiplicative inverses in $\mathbb{Z}_p$. Suppose we are given a prime p and an element $\alpha \in \mathbb{Z}_p^*$, and we want to compute α^{-1}. By Euler's theorem (Theorem 2.13), we have $\alpha^{p-1} = 1$, and multiplying this equation by α^{-1}, we obtain $\alpha^{p-2} = \alpha^{-1}$. Thus, we can use the repeated-squaring algorithm to compute α^{-1} by raising α to the power $p - 2$. This algorithm runs in time $O(\text{len}(p)^3)$. While this is reasonably efficient, we will develop an even more efficient method in the next chapter, using Euclid's algorithm (which also works with any modulus, not just a prime modulus).

Testing quadratic residuosity. Suppose we are given an odd prime p and an element $\alpha \in \mathbb{Z}_p^*$, and we want to test whether $\alpha \in (\mathbb{Z}_p^*)^2$. By Euler's criterion (Theorem 2.21), we have $\alpha \in (\mathbb{Z}_p^*)^2$ if and only if $\alpha^{(p-1)/2} = 1$. Thus, we can use the repeated-squaring algorithm to test if $\alpha \in (\mathbb{Z}_p^*)^2$ by raising α to the power $(p - 1)/2$. This algorithm runs in time $O(\text{len}(p)^3)$. While this is also reasonably efficient, we will develop an even more efficient method later in the text (in Chapter 12).

Testing for primality. Suppose we are given an integer $n > 1$, and we want to determine whether n is prime or composite. For large n, searching for prime factors of n is hopelessly impractical. A better idea is to use Euler's theorem,

combined with the repeated-squaring algorithm: we know that if n is prime, then every non-zero $\alpha \in \mathbb{Z}_n$ satisfies $\alpha^{n-1} = 1$. Conversely, if n is composite, there exists a non-zero $\alpha \in \mathbb{Z}_n$ such that $\alpha^{n-1} \neq 1$ (see Exercise 2.27). This suggests the following "trial and error" strategy for testing if n is prime:

repeat k times
  choose $\alpha \in \mathbb{Z}_n \setminus \{[0]\}$
  compute $\beta \leftarrow \alpha^{n-1}$
  if $\beta \neq 1$ output "composite" and halt
output "maybe prime"

As stated, this is not a fully specified algorithm: we have to specify the loop-iteration parameter k, and more importantly, we have to specify a procedure for choosing α in each loop iteration. One approach might be to just try $\alpha = [1], [2], [3], \ldots$. Another might be to choose α at random in each loop iteration: this would be an example of a *probabilistic algorithm* (a notion we shall discuss in detail in Chapter 9). In any case, if the algorithm outputs "composite," we may conclude that n is composite (even though the algorithm does not find a non-trivial factor of n). However, if the algorithm completes all k loop iterations and outputs "maybe prime," it is not clear what we should conclude: certainly, we have some reason to suspect that n is prime, but not really a proof; indeed, it may be the case that n is composite, but we were just unlucky in all of our choices for α. Thus, while this rough idea does not quite give us an effective primality test, it is not a bad start, and is the basis for several effective primality tests (a couple of which we shall discuss in detail in Chapters 10 and 21).

EXERCISE 3.33. 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.

EXERCISE 3.34. Show that given a prime p, $\alpha \in \mathbb{Z}_p$, and an integer $e \geq p$, we can compute α^e in time $O(\text{len}(e)\,\text{len}(p) + \text{len}(p)^3)$.

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

EXERCISE 3.35. 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, for some parameter t, rather than in base 2. Let $\alpha \in \mathbb{Z}_n$ and let e be a positive integer of length ℓ. Let us write e in base 2^t as $e = (e_k \cdots e_0)_{2^t}$, where $e_k \neq 0$. Consider the following algorithm:

compute a table of values $T[0 \ldots 2^t - 1]$,
    where $T[j] := \alpha^j$ for $j = 0, \ldots, 2^t - 1$
$\beta \leftarrow T[e_k]$
for $i \leftarrow k - 1$ down to 0 do
    $\beta \leftarrow \beta^{2^t} \cdot T[e_i]$

(a) Show that this algorithm correctly computes α^e, and work out the implementation details; in particular, show that it may be implemented in such a way that it uses at most ℓ squarings and $2^t + \ell/t + O(1)$ additional multiplications in $\mathbb{Z}_n$.

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

(c) Improve the algorithm so that it only uses no more than ℓ squarings and $2^{t-1} + \ell/t + O(1)$ additional multiplications in $\mathbb{Z}_n$. Hint: build a table that contains only the *odd* powers of α among $\alpha^0, \alpha^1, \ldots, \alpha^{2^t-1}$.

EXERCISE 3.36. 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 at most ℓ squarings and $\ell + 2^k$ additional multiplications in $\mathbb{Z}_n$. Your algorithm should work in two phases: the first phase uses only the values $\alpha_1, \ldots, \alpha_k$, and performs at most 2^k multiplications in $\mathbb{Z}_n$; in the second phase, the algorithm computes β, using the exponents $e_1, \ldots, e_k$, along with the data computed in the first phase, and performs at most ℓ squarings and ℓ additional multiplications in $\mathbb{Z}_n$.

EXERCISE 3.37. Suppose that we are to compute α^e, where $\alpha \in \mathbb{Z}_n$, for many exponents e of length at most ℓ, but with α fixed. Show that for every positive integer parameter k, we can make a pre-computation (depending on α, ℓ, and k) that uses at most ℓ squarings and 2^k additional multiplications in $\mathbb{Z}_n$, so that after the pre-computation, we can compute α^e for every exponent e of length at most ℓ using at most $\ell/k + O(1)$ squarings and $\ell/k + O(1)$ additional multiplications in $\mathbb{Z}_n$. Hint: use the algorithm in the previous exercise.

EXERCISE 3.38. Suppose we are given $\alpha \in \mathbb{Z}_n$, along with non-negative integers $e_1, \ldots, e_r$, where $\operatorname{len}(e_i) \leq \ell$ for $i = 1, \ldots, r$, and $r = O(\operatorname{len}(\ell))$. Using the previous exercise, show how to compute $(\alpha^{e_1}, \ldots, \alpha^{e_r})$ using $O(\ell)$ multiplications in $\mathbb{Z}_n$.

EXERCISE 3.39. Suppose we are given $\alpha \in \mathbb{Z}_n$, along with integers $m_1, \ldots, m_r$,

with each $m_i > 1$. Let $m := \prod_i m_i$. Also, for $i = 1, \ldots, r$, let $m_i^* := m/m_i$. Show how to compute $(\alpha^{m_1^*}, \ldots, \alpha^{m_r^*})$ using $O(\text{len}(r)\ell)$ multiplications in $\mathbb{Z}_n$, where $\ell := \text{len}(m)$. Hint: divide and conquer. Note that if $r = O(\text{len}(\ell))$, then using the previous exercise, we can solve this problem using just $O(\ell)$ multiplications.

EXERCISE 3.40. 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 $\text{len}(e_i) \leq \ell$ for $i = 1, \ldots, k$. Show how to compute the value $\alpha_1^{e_1} \cdots \alpha_k^{e_k}$, using at most ℓ squarings and $O(\ell / \text{len}(\ell))$ additional multiplications in $\mathbb{Z}_n$. Hint: develop a 2^t-ary version of the algorithm in Exercise 3.36.

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 3.41. Suppose we have two positive integers a and b, each of length at most ℓ, such that $a = a_1 2^k + a_0$ and $b = b_1 2^k + b_0$, where $0 \leq a_0 < 2^k$ and $0 \leq b_0 < 2^k$. 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 exercise is also not the best possible. In fact, it is possible to multiply two integers of length at most ℓ *on a RAM* in time $O(\ell)$, but we do not explore this any further for the moment (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 length at most ℓ in time at most $M(\ell)$. 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, such that for some constant $\gamma \geq 1$, and all positive integers a and b, we have

$$1 \leq \frac{M(a+b)}{M(a) + M(b)} \leq \gamma.$$

EXERCISE 3.42. Show that if M is a well-behaved complexity function, then it is strictly increasing.

EXERCISE 3.43. Show that if $N(\ell) := M(\ell)/\ell$ is a non-decreasing function, and $M(2\ell)/M(\ell) = O(1)$, then M is a well-behaved complexity function.

EXERCISE 3.44. Let $\alpha > 0$, $\beta \geq 1$, $\gamma \geq 0$, $\delta \geq 0$ 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 3.45. Show that given integers $n > 1$ and $e > 1$, we can compute n^e in time $O(M(\operatorname{len}(n^e)))$.

EXERCISE 3.46. Give an algorithm for Exercise 3.26 whose running time is $O(M(\operatorname{len}(n)) \operatorname{len}(k))$. Hint: divide and conquer.

EXERCISE 3.47. In the previous exercise, suppose all the inputs n_i have the same length, and that $M(\ell) = \alpha \ell^{\beta}$, where α and β are constants with $\alpha > 0$ and $\beta > 1$. Show that your algorithm runs in time $O(M(\operatorname{len}(n)))$.

EXERCISE 3.48. 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 $\operatorname{len}(a)$ 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 $\hat{z} = (1 + \varepsilon)z$ for some $|\varepsilon| \leq 2^{-k+1}$. Show that given positive integers b and k, we can compute 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 3.49. Using the result of the previous exercise, show that, given positive integers a and b of bit length at most ℓ, we can 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 than multiplication.

EXERCISE 3.50. Using the result of the previous exercise, give an algorithm for Exercise 3.27 that runs in time $O(M(\operatorname{len}(n)) \operatorname{len}(k))$. Hint: divide and conquer.

EXERCISE 3.51. Give an algorithm for Exercise 3.29 whose running time is $O(M(\operatorname{len}(n)))$. Hint: Newton iteration.

EXERCISE 3.52. Suppose we have an algorithm that computes the square of an ℓ-bit integer in time at most $S(\ell)$, where S is a well-behaved complexity function.

Show how to use this algorithm to compute the product of two arbitrary integers of length at most ℓ in time $O(S(\ell))$.

EXERCISE 3.53. Give algorithms for Exercise 3.32 whose running times are $O(M(\ell)\,\mathrm{len}(\ell))$, where $\ell := \mathrm{len}(n)$. Hint: divide and conquer.

3.6 Notes

Shamir [89] 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 `gmplib.org`). 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 assembly language. 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, in order to avoid overflow. However, if one codes in assembly language, one can take B to be much closer, 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 3.41, which was originally invented by Karatsuba [54] (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 integers of length at most ℓ in time

$O(\ell^{\log_2 3})$. The fastest known algorithm for multiplying such integers on a RAM runs in time $O(\ell)$, and is due to Schönhage. It actually works on a very restricted type of RAM called a "pointer machine" (see Exercise 12, Section 4.3.3 of Knuth [56]). See Exercise 17.25 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 that are bit strings 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. For many years, the smallest known Boolean circuit that multiplies two integers of length at most ℓ was of size $O(\ell \operatorname{len}(\ell) \operatorname{len}(\operatorname{len}(\ell)))$. This result was due to Schönhage and Strassen [86]. More recently, Fürer showed how to reduce this to $O(\ell \operatorname{len}(\ell) 2^{O(\log^* \ell)})$ [38]. Here, the value of $\log^* n$ is defined as the minimum number of applications of the function $\log_2$ to the number n required to obtain a number that is less than or equal to 1. The function $\log^*$ is an extremely slow growing function, and is a constant for all practical purposes.

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. These have the advantages of being simple and of being 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, are superior 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 [56].