4

Euclid's algorithm

In this chapter, we discuss Euclid's algorithm for computing greatest common divisors, which, as we will see, has applications far beyond that of just computing greatest common divisors.

4.1 The basic Euclidean algorithm

We consider the following problem: given two non-negative integers a and b, compute their greatest common divisor, $\gcd(a, b)$. We can do this using the well-known **Euclidean algorithm**, also called **Euclid's algorithm**.

The basic idea is the following. Without loss of generality, we may assume that $a \geq b \geq 0$. If $b = 0$, then there is nothing to do, since in this case, $\gcd(a, 0) = a$. Otherwise, $b > 0$, and we can compute the integer quotient $q := \lfloor a/b \rfloor$ and remainder $r := a \bmod b$, where $0 \leq r < b$. From the equation

$$a = bq + r,$$

it is easy to see that if an integer d divides both b and r, then it also divides a; likewise, if an integer d divides a and b, then it also divides r. From this observation, it follows that $\gcd(a, b) = \gcd(b, r)$, and so by performing a division, we reduce the problem of computing $\gcd(a, b)$ to the "smaller" problem of computing $\gcd(b, r)$.

The following theorem develops this idea further:

Theorem 4.1. *Let a, b be integers, with $a \geq b \geq 0$. Using the division with remainder property, define the integers $r_0, r_1, \ldots, r_{\lambda+1}$ and $q_1, \ldots, q_\lambda$, where $\lambda \geq 0$, as follows:*

$$\begin{aligned}
a &= r_0, \\
b &= r_1, \\
r_0 &= r_1 q_1 + r_2 && (0 < r_2 < r_1), \\
&\vdots \\
r_{i-1} &= r_i q_i + r_{i+1} && (0 < r_{i+1} < r_i), \\
&\vdots \\
r_{\lambda-2} &= r_{\lambda-1} q_{\lambda-1} + r_\lambda && (0 < r_\lambda < r_{\lambda-1}), \\
r_{\lambda-1} &= r_\lambda q_\lambda && (r_{\lambda+1} = 0).
\end{aligned}$$

Note that by definition, $\lambda = 0$ if $b = 0$, and $\lambda > 0$, otherwise. Then we have $r_\lambda = \gcd(a, b)$. Moreover, if $b > 0$, then $\lambda \leq \log b / \log \phi + 1$, where $\phi := (1 + \sqrt{5})/2 \approx 1.62$.

Proof. For the first statement, one sees that for $i = 1, \ldots, \lambda$, we have $r_{i-1} = r_i q_i + r_{i+1}$, from which it follows that the common divisors of r_{i-1} and r_i are the same as the common divisors of r_i and r_{i+1}, and hence $\gcd(r_{i-1}, r_i) = \gcd(r_i, r_{i+1})$. From this, it follows that

$$\gcd(a, b) = \gcd(r_0, r_1) = \cdots = \gcd(r_\lambda, r_{\lambda+1}) = \gcd(r_\lambda, 0) = r_\lambda.$$

To prove the second statement, assume that $b > 0$, and hence $\lambda > 0$. If $\lambda = 1$, the statement is obviously true, so assume $\lambda > 1$. We claim that for $i = 0, \ldots, \lambda - 1$, we have $r_{\lambda-i} \geq \phi^i$. The statement will then follow by setting $i = \lambda - 1$ and taking logarithms.

We now prove the above claim. For $i = 0$ and $i = 1$, we have

$$r_\lambda \geq 1 = \phi^0 \quad \text{and} \quad r_{\lambda-1} \geq r_\lambda + 1 \geq 2 \geq \phi^1.$$

For $i = 2, \ldots, \lambda - 1$, using induction and applying the fact that $\phi^2 = \phi + 1$, we have

$$r_{\lambda-i} \geq r_{\lambda-(i-1)} + r_{\lambda-(i-2)} \geq \phi^{i-1} + \phi^{i-2} = \phi^{i-2}(1 + \phi) = \phi^i,$$

which proves the claim. □

Example 4.1. Suppose $a = 100$ and $b = 35$. Then the numbers appearing in Theorem 4.1 are easily computed as follows:

i	0	1	2	3	4
r_i	100	35	30	5	0
q_i		2	1	6	

So we have $\gcd(a, b) = r_3 = 5$. □

We can easily turn the scheme described in Theorem 4.1 into a simple algorithm:

Euclid's algorithm. On input a, b, where a and b are integers such that $a \geq b \geq 0$, compute $d = \gcd(a, b)$ as follows:

$r \leftarrow a,\ r' \leftarrow b$
while $r' \neq 0$ do
    $r'' \leftarrow r \bmod r'$
    $(r, r') \leftarrow (r', r'')$
$d \leftarrow r$
output d

We now consider the running time of Euclid's algorithm. Naively, one could estimate this as follows. Suppose a and b are ℓ-bit numbers. The number of divisions performed by the algorithm is the number λ in Theorem 4.1, which is $O(\ell)$. Moreover, each division involves numbers of ℓ bits or fewer in length, and so takes time $O(\ell^2)$. This leads to a bound on the running time of $O(\ell^3)$. However, as the following theorem shows, this cubic running time bound is well off the mark. Intuitively, this is because the cost of performing a division depends on the length of the quotient: the larger the quotient, the more expensive the division, but also, the more progress the algorithm makes towards termination.

Theorem 4.2. *Euclid's algorithm runs in time* $O(\mathrm{len}(a)\,\mathrm{len}(b))$.

Proof. We may assume that $b > 0$. With notation as in Theorem 4.1, the running time is $O(T)$, where

$$\begin{aligned} T &= \sum_{i=1}^{\lambda} \mathrm{len}(r_i)\,\mathrm{len}(q_i) \leq \mathrm{len}(b) \sum_{i=1}^{\lambda} \mathrm{len}(q_i) \\ &\leq \mathrm{len}(b) \sum_{i=1}^{\lambda} (\mathrm{len}(r_{i-1}) - \mathrm{len}(r_i) + 1) \quad \text{(see Exercise 3.24)} \\ &= \mathrm{len}(b)(\mathrm{len}(r_0) - \mathrm{len}(r_\lambda) + \lambda) \quad \text{(telescoping the sum)} \\ &\leq \mathrm{len}(b)(\mathrm{len}(a) + \log b / \log \phi + 1) \quad \text{(by Theorem 4.1)} \\ &= O(\mathrm{len}(a)\,\mathrm{len}(b)). \quad \square \end{aligned}$$

EXERCISE 4.1. With notation as in Theorem 4.1, give a direct and simple proof that for each $i = 1, \ldots, \lambda$, we have $r_{i+1} \leq r_{i-1}/2$. Thus, with every *two* division steps, the bit length of the remainder drops by at least 1. Based on this, give an alternative proof that the number of divisions is $O(\mathrm{len}(b))$.

EXERCISE 4.2. Show how to compute $\operatorname{lcm}(a, b)$ in time $O(\operatorname{len}(a) \operatorname{len}(b))$.

EXERCISE 4.3. Let $a, b \in \mathbb{Z}$ with $a \geq b \geq 0$, let $d := \gcd(a, b)$, and assume $d > 0$. Suppose that on input a, b, Euclid's algorithm performs λ division steps, and computes the remainder sequence $\{r_i\}_{i=0}^{\lambda+1}$ and the quotient sequence $\{q_i\}_{i=1}^{\lambda}$ (as in Theorem 4.1). Now suppose we run Euclid's algorithm on input $a/d, b/d$. Show that on these inputs, the number of division steps performed is also λ, the remainder sequence is $\{r_i/d\}_{i=0}^{\lambda+1}$, and the quotient sequence is $\{q_i\}_{i=1}^{\lambda}$.

EXERCISE 4.4. Show that if we run Euclid's algorithm on input a, b, where $a \geq b > 0$, then its running time is $O(\operatorname{len}(a/d) \operatorname{len}(b))$, where $d := \gcd(a, b)$.

EXERCISE 4.5. Let λ be a positive integer. Show that there exist integers a, b with $a > b > 0$ and $\lambda \geq \log b / \log \phi$, such that Euclid's algorithm on input a, b performs at least λ divisions. Thus, the bound in Theorem 4.1 on the number of divisions is essentially tight.

EXERCISE 4.6. This exercise looks at an alternative algorithm for computing $\gcd(a, b)$, called the **binary gcd algorithm**. This algorithm avoids complex operations, such as division and multiplication; instead, it relies only on subtraction, and division and multiplication by powers of 2, which, assuming a binary representation of integers (as we are), can be very efficiently implemented using "right shift" and "left shift" operations. The algorithm takes positive integers a and b as input, and runs as follows:

$r \leftarrow a,\ r' \leftarrow b,\ e \leftarrow 0$
while $2 \mid r$ and $2 \mid r'$ do $r \leftarrow r/2,\ r' \leftarrow r'/2,\ e \leftarrow e + 1$
repeat
    while $2 \mid r$ do $r \leftarrow r/2$
    while $2 \mid r'$ do $r' \leftarrow r'/2$
    if $r' < r$ then $(r, r') \leftarrow (r', r)$
    $r' \leftarrow r' - r$
until $r' = 0$
$d \leftarrow 2^e \cdot r$
output d

Show that this algorithm correctly computes $\gcd(a, b)$, and runs in time $O(\ell^2)$, where $\ell := \max(\operatorname{len}(a), \operatorname{len}(b))$.

4.2 The extended Euclidean algorithm

Let a and b be integers, and let $d := \gcd(a, b)$. We know by Theorem 1.8 that there exist integers s and t such that $as + bt = d$. The **extended Euclidean algorithm**

allows us to efficiently compute s and t. The next theorem defines the quantities computed by this algorithm, and states a number of important facts about them; these facts will play a crucial role, both in the analysis of the running time of the algorithm, as well as in applications of the algorithm that we will discuss later.

Theorem 4.3. *Let a, b, $r_0, \ldots, r_{\lambda+1}$ and $q_1, \ldots, q_\lambda$ be as in Theorem 4.1. Define integers $s_0, \ldots, s_{\lambda+1}$ and $t_0, \ldots, t_{\lambda+1}$ as follows:*

$$\begin{aligned} s_0 &:= 1, & t_0 &:= 0, \\ s_1 &:= 0, & t_1 &:= 1, \\ s_{i+1} &:= s_{i-1} - s_i q_i, & t_{i+1} &:= t_{i-1} - t_i q_i \qquad (i = 1, \ldots, \lambda). \end{aligned}$$

Then:

(i) for $i = 0, \ldots, \lambda+1$, we have $as_i + bt_i = r_i$; in particular, $as_\lambda + bt_\lambda = \gcd(a, b)$;

(ii) for $i = 0, \ldots, \lambda$, we have $s_i t_{i+1} - t_i s_{i+1} = (-1)^i$;

(iii) for $i = 0, \ldots, \lambda + 1$, we have $\gcd(s_i, t_i) = 1$;

(iv) for $i = 0, \ldots, \lambda$, we have $t_i t_{i+1} \leq 0$ and $|t_i| \leq |t_{i+1}|$; for $i = 1, \ldots, \lambda$, we have $s_i s_{i+1} \leq 0$ and $|s_i| \leq |s_{i+1}|$;

(v) for $i = 1, \ldots, \lambda + 1$, we have $r_{i-1}|t_i| \leq a$ and $r_{i-1}|s_i| \leq b$;

(vi) if $a > 0$, then for $i = 1, \ldots, \lambda + 1$, we have $|t_i| \leq a$ and $|s_i| \leq b$; if $a > 1$ and $b > 0$, then $|t_\lambda| \leq a/2$ and $|s_\lambda| \leq b/2$.

Proof. (i) is easily proved by induction on i. For $i = 0, 1$, the statement is clear. For $i = 2, \ldots, \lambda + 1$, we have

$$\begin{aligned} as_i + bt_i &= a(s_{i-2} - s_{i-1}q_{i-1}) + b(t_{i-2} - t_{i-1}q_{i-1}) \\ &= (as_{i-2} + bt_{i-2}) - (as_{i-1} + bt_{i-1})q_{i-1} \\ &= r_{i-2} - r_{i-1}q_{i-1} \quad \text{(by induction)} \\ &= r_i. \end{aligned}$$

(ii) is also easily proved by induction on i. For $i = 0$, the statement is clear. For $i = 1, \ldots, \lambda$, we have

$$\begin{aligned} s_i t_{i+1} - t_i s_{i+1} &= s_i(t_{i-1} - t_i q_i) - t_i(s_{i-1} - s_i q_i) \\ &= -(s_{i-1}t_i - t_{i-1}s_i) \quad \text{(after expanding and simplifying)} \\ &= -(-1)^{i-1} \quad \text{(by induction)} \\ &= (-1)^i. \end{aligned}$$

(iii) follows directly from (ii).

For (iv), one can easily prove both statements by induction on i. The statement involving the t_i's is clearly true for $i = 0$. For $i = 1, \ldots, \lambda$, we have

$t_{i+1} = t_{i-1} - t_i q_i$; moreover, by the induction hypothesis, t_{i-1} and t_i have opposite signs and $|t_i| \geq |t_{i-1}|$; it follows that $|t_{i+1}| = |t_{i-1}| + |t_i|q_i \geq |t_i|$, and that the sign of t_{i+1} is the opposite of that of t_i. The proof of the statement involving the s_i's is the same, except that we start the induction at $i = 1$.

For (v), one considers the two equations:

$$as_{i-1} + bt_{i-1} = r_{i-1},$$
$$as_i + bt_i = r_i.$$

Subtracting t_{i-1} times the second equation from t_i times the first, and applying (ii), we get $\pm a = t_i r_{i-1} - t_{i-1} r_i$; consequently, using the fact that t_i and t_{i-1} have opposite sign, we obtain

$$a = |t_i r_{i-1} - t_{i-1} r_i| = |t_i| r_{i-1} + |t_{i-1}| r_i \geq |t_i| r_{i-1}.$$

The inequality involving s_i follows similarly, subtracting s_{i-1} times the second equation from s_i times the first.

(vi) follows from (v) and the following observations: if $a > 0$, then $r_{i-1} > 0$ for $i = 1, \ldots, \lambda + 1$; if $a > 1$ and $b > 0$, then $\lambda > 0$ and $r_{\lambda-1} \geq 2$. □

Example 4.2. We continue with Example 4.1. The s_i's and t_i's are easily computed from the q_i's:

i	0	1	2	3	4
r_i	100	35	30	5	0
q_i		2	1	6	
s_i	1	0	1	-1	7
t_i	0	1	-2	3	-20

So we have $\gcd(a, b) = 5 = -a + 3b$. □

We can easily turn the scheme described in Theorem 4.3 into a simple algorithm:

The extended Euclidean algorithm. On input a, b, where a and b are integers such that $a \geq b \geq 0$, compute integers d, s, and t, such that $d = \gcd(a, b)$ and $as + bt = d$, as follows:

$r \leftarrow a,\ r' \leftarrow b$
$s \leftarrow 1,\ s' \leftarrow 0$
$t \leftarrow 0,\ t' \leftarrow 1$
while $r' \neq 0$ do
    $q \leftarrow \lfloor r/r' \rfloor,\ r'' \leftarrow r \bmod r'$
    $(r, s, t, r', s', t') \leftarrow (r', s', t', r'', s - s'q, t - t'q)$
$d \leftarrow r$
output d, s, t

Theorem 4.4. *The extended Euclidean algorithm runs in time $O(\mathrm{len}(a)\,\mathrm{len}(b))$.*

Proof. We may assume that $b > 0$. It suffices to analyze the cost of computing the coefficient sequences $\{s_i\}$ and $\{t_i\}$. Consider first the cost of computing all of the t_i's, which is $O(T)$, where $T = \sum_{i=1}^{\lambda} \mathrm{len}(t_i)\,\mathrm{len}(q_i)$. We have $t_1 = 1$ and, by part (vi) of Theorem 4.3, we have $|t_i| \leq a$ for $i = 2, \ldots, \lambda$. Arguing as in the proof of Theorem 4.2, we have

$$\begin{aligned} T &\leq \mathrm{len}(q_1) + \mathrm{len}(a) \sum_{i=2}^{\lambda} \mathrm{len}(q_i) \\ &\leq \mathrm{len}(a) + \mathrm{len}(a)(\mathrm{len}(r_1) - \mathrm{len}(r_\lambda) + \lambda - 1) = O(\mathrm{len}(a)\,\mathrm{len}(b)). \end{aligned}$$

An analogous argument shows that one can also compute all of the s_i's in time $O(\mathrm{len}(a)\,\mathrm{len}(b))$, and in fact, in time $O(\mathrm{len}(b)^2)$. □

For the reader familiar with the basics of the theory of matrices and determinants, it is instructive to view Theorem 4.3 as follows. For $i = 1, \ldots, \lambda$, we have

$$\begin{pmatrix} r_i \\ r_{i+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & -q_i \end{pmatrix} \begin{pmatrix} r_{i-1} \\ r_i \end{pmatrix}.$$

Recursively expanding the right-hand side of this equation, we have

$$\begin{pmatrix} r_i \\ r_{i+1} \end{pmatrix} = \overbrace{\begin{pmatrix} 0 & 1 \\ 1 & -q_i \end{pmatrix} \cdots \begin{pmatrix} 0 & 1 \\ 1 & -q_1 \end{pmatrix}}^{M_i :=} \begin{pmatrix} a \\ b \end{pmatrix}.$$

This defines the 2×2 matrix M_i for $i = 1, \ldots, \lambda$. If we additionally define M_0 to be the 2×2 identity matrix, then it is easy to see that for $i = 0, \ldots, \lambda$, we have

$$M_i = \begin{pmatrix} s_i & t_i \\ s_{i+1} & t_{i+1} \end{pmatrix}.$$

From these observations, part (i) of Theorem 4.3 is immediate, and part (ii) follows from the fact that M_i is the product of i matrices, each of determinant -1, and the determinant of M_i is evidently $s_i t_{i+1} - t_i s_{i+1}$.

EXERCISE 4.7. In our description of the extended Euclidean algorithm, we made the restriction that the inputs a and b satisfy $a \geq b \geq 0$. Using this restricted algorithm as a subroutine, give an algorithm that works without any restrictions on its input.

EXERCISE 4.8. With notation and assumptions as in Exercise 4.3, suppose that on input a, b, the extended Euclidean algorithm computes the coefficient sequences

$\{s_i\}_{i=0}^{\lambda+1}$ and $\{t_i\}_{i=0}^{\lambda+1}$ (as in Theorem 4.3). Show that the extended Euclidean algorithm on input $a/d, b/d$ computes the same coefficient sequences.

EXERCISE 4.9. Assume notation as in Theorem 4.3. Show that:

(a) for all $i = 2, \ldots, \lambda$, we have $|t_i| < |t_{i+1}|$ and $r_{i-1}|t_i| < a$, and that for all $i = 3, \ldots, \lambda$, we have $|s_i| < |s_{i+1}|$ and $r_{i-1}|s_i| < b$;

(b) $s_i t_i \leq 0$ for $i = 0, \ldots, \lambda + 1$;

(c) if $d := \gcd(a, b) > 0$, then $|s_{\lambda+1}| = b/d$ and $|t_{\lambda+1}| = a/d$.

EXERCISE 4.10. One can extend the binary gcd algorithm discussed in Exercise 4.6 so that in addition to computing $d = \gcd(a, b)$, it also computes s and t such that $as + bt = d$. Here is one way to do this (again, we assume that a and b are positive integers):

$r \leftarrow a,\ r' \leftarrow b,\ e \leftarrow 0$
while $2 \mid r$ and $2 \mid r'$ do $r \leftarrow r/2,\ r' \leftarrow r'/2,\ e \leftarrow e + 1$
$\tilde{a} \leftarrow r,\ \tilde{b} \leftarrow r',\ s \leftarrow 1,\ t \leftarrow 0,\ s' \leftarrow 0,\ t' \leftarrow 1$
repeat
    while $2 \mid r$ do
        $r \leftarrow r/2$
        if $2 \mid s$ and $2 \mid t$ then $s \leftarrow s/2,\ t \leftarrow t/2$
        else $s \leftarrow (s + \tilde{b})/2,\ t \leftarrow (t - \tilde{a})/2$
    while $2 \mid r'$ do
        $r' \leftarrow r'/2$
        if $2 \mid s'$ and $2 \mid t'$ then $s' \leftarrow s'/2,\ t' \leftarrow t'/2$
        else $s' \leftarrow (s' + \tilde{b})/2,\ t' \leftarrow (t' - \tilde{a})/2$
    if $r' < r$ then $(r, s, t, r', s', t') \leftarrow (r', s', t', r, s, t)$
    $r' \leftarrow r' - r,\ s' \leftarrow s' - s,\ t' \leftarrow t' - t$
until $r' = 0$
$d \leftarrow 2^e \cdot r,$ output d, s, t

Show that this algorithm is correct and that its running time is $O(\ell^2)$, where $\ell := \max(\mathrm{len}(a), \mathrm{len}(b))$. In particular, you should verify that all of the divisions by 2 performed by the algorithm yield integer results. Moreover, show that the outputs s and t are of length $O(\ell)$.

EXERCISE 4.11. Suppose we modify the extended Euclidean algorithm so that it computes balanced remainders; that is, for $i = 1, \ldots, \lambda$, the values q_i and r_{i+1} are computed so that $r_{i-1} = r_i q_i + r_{i+1}$ and $-|r_i|/2 \leq r_{i+1} < |r_i|/2$. Assume that the s_i's and the t_i's are computed by the same formula as in Theorem 4.3. Give a detailed analysis of the running time of this algorithm, which should include an analysis of the number of division steps, and the sizes of the s_i's and t_i's.

4.3 Computing modular inverses and Chinese remaindering

An important application of the extended Euclidean algorithm is to the problem of computing multiplicative inverses in $\mathbb{Z}_n$.

Theorem 4.5. *Suppose we are given integers n, b, where $0 \leq b < n$. Then in time $O(\text{len}(n)^2)$, we can determine if b is relatively prime to n, and if so, compute $b^{-1} \bmod n$.*

Proof. We may assume $n > 1$, since when $n = 1$, we have $b = 0 = b^{-1} \bmod n$. We run the extended Euclidean algorithm on input n, b, obtaining integers d, s, and t, such that $d = \gcd(n, b)$ and $ns + bt = d$. If $d \neq 1$, then b does not have a multiplicative inverse modulo n. Otherwise, if $d = 1$, then t is a multiplicative inverse of b modulo n; however, it may not lie in the range $\{0, \ldots, n-1\}$, as required. By part (vi) of Theorem 4.3, we have $|t| \leq n/2 < n$. Thus, if $t \geq 0$, then $b^{-1} \bmod n$ is equal to t; otherwise, $b^{-1} \bmod n$ is equal to $t + n$. Based on Theorem 4.4, it is clear that all the computations can be performed in time $O(\text{len}(n)^2)$. □

Example 4.3. Suppose we are given integers a, b, n, where $0 \leq a < n$, and $0 \leq b < n$, and we want to compute a solution z to the congruence $az \equiv b \pmod{n}$, or determine that no such solution exists. Based on the discussion in Example 2.5, the following algorithm does the job:

$d \leftarrow \gcd(a, n)$
if $d \nmid b$ then
    output "no solution"
else
    $a' \leftarrow a/d,\ b' \leftarrow b/d,\ n' \leftarrow n/d$
    $t \leftarrow (a')^{-1} \bmod n'$
    $z \leftarrow tb' \bmod n'$
    output z

Using Euclid's algorithm to compute d, and the extended Euclidean algorithm to compute t (as in Theorem 4.5), the running time of this algorithm is clearly $O(\text{len}(n)^2)$. □

We also observe that the Chinese remainder theorem (Theorem 2.6) can be made computationally effective:

Theorem 4.6 (Effective Chinese remainder theorem). *Suppose we are given integers $n_1, \ldots, n_k$ and $a_1, \ldots, a_k$, where the family $\{n_i\}_{i=1}^k$ is pairwise relatively prime, and where $n_i > 1$ and $0 \leq a_i < n_i$ for $i = 1, \ldots, k$. Let $n := \prod_{i=1}^k n_i$. Then in time $O(\text{len}(n)^2)$, we can compute the unique integer a satisfying $0 \leq a < n$ and $a \equiv a_i \pmod{n_i}$ for $i = 1, \ldots, k$.*

Proof. The algorithm is a straightforward implementation of the proof of Theorem 2.6, and runs as follows:

$n \leftarrow \prod_{i=1}^{k} n_i$
for $i \leftarrow 1$ to k do
    $n_i^* \leftarrow n/n_i$, $b_i \leftarrow n_i^* \bmod n_i$, $t_i \leftarrow b_i^{-1} \bmod n_i$, $e_i \leftarrow n_i^* t_i$
$a \leftarrow \left(\sum_{i=1}^{k} a_i e_i\right) \bmod n$

We leave it to the reader to verify the running time bound. □

EXERCISE 4.12. In Example 4.3, show that one can easily obtain the quantities d, a', n', and t from the data computed in just a single execution of the extended Euclidean algorithm.

EXERCISE 4.13. In this exercise, you are to make the result of Theorem 2.17 effective. Suppose that we are given a positive integer n, two elements $\alpha, \beta \in \mathbb{Z}_n^*$, and integers ℓ and m, such that $\alpha^\ell = \beta^m$ and $\gcd(\ell, m) = 1$. Show how to compute $\gamma \in \mathbb{Z}_n^*$ such that $\alpha = \gamma^m$ in time $O(\mathrm{len}(\ell)\,\mathrm{len}(m) + (\mathrm{len}(\ell) + \mathrm{len}(m))\,\mathrm{len}(n)^2)$.

EXERCISE 4.14. In this exercise and the next, you are to analyze an "incremental Chinese remaindering algorithm." Consider the following algorithm, which takes as input integers a_1, n_1, a_2, n_2 satisfying

$$0 \le a_1 < n_1, \quad 0 \le a_2 < n_2, \quad \text{and} \quad \gcd(n_1, n_2) = 1.$$

It outputs integers a, n satisfying

$$n = n_1 n_2, \quad 0 \le a < n, \quad a \equiv a_1 \pmod{n_1}, \quad \text{and} \quad a \equiv a_2 \pmod{n_2},$$

and runs as follows:

$b \leftarrow n_1 \bmod n_2$, $t \leftarrow b^{-1} \bmod n_2$, $h \leftarrow (a_2 - a_1)t \bmod n_2$
$a \leftarrow a_1 + n_1 h$, $n \leftarrow n_1 n_2$
output a, n

Show that the algorithm correctly computes a and n as specified, and runs in time $O(\mathrm{len}(n)\,\mathrm{len}(n_2))$.

EXERCISE 4.15. Using the algorithm in the previous exercise as a subroutine, give a simple $O(\mathrm{len}(n)^2)$ algorithm that takes as input integers $n_1, \ldots, n_k$ and $a_1, \ldots, a_k$, where the family $\{n_i\}_{i=1}^k$ is pairwise relatively prime, and where $n_i > 1$ and $0 \le a_i < n_i$ for $i = 1, \ldots, k$, and outputs integers a and n such that $0 \le a < n$, $n = \prod_{i=1}^{k} n_i$, and $a \equiv a_i \pmod{n_i}$ for $i = 1, \ldots, k$. The algorithm should be "incremental," in that it processes the pairs (a_i, n_i) one at a time, using time $O(\mathrm{len}(n)\,\mathrm{len}(n_i))$ per pair.

EXERCISE 4.16. Suppose we are given $\alpha_1, \ldots, \alpha_k \in \mathbb{Z}_n^*$. Show how to compute $\alpha_1^{-1}, \ldots, \alpha_k^{-1}$ by computing *one* multiplicative inverse modulo n, and performing fewer than $3k$ multiplications modulo n. This result is useful, as in practice, if n is several hundred bits long, it may take 10–20 times longer to compute multiplicative inverses modulo n than to multiply modulo n.

4.4 Speeding up algorithms via modular computation

An important practical application of the above "computational" version (Theorem 4.6) of the Chinese remainder theorem is a general algorithmic technique that can significantly speed up certain types of computations involving long integers. Instead of trying to describe the technique in some general form, we simply illustrate the technique by means of a specific example: integer matrix multiplication.

Suppose we have two $m \times m$ matrices A and B whose entries are large integers, and we want to compute the product matrix $C := AB$. Suppose that for $r, s = 1, \ldots, m$, the entry of A at row r and column s is a_{rs}, and that for $s, t = 1, \ldots, m$, the entry of B at row s and column t is b_{st}. Then for $r, t = 1, \ldots, m$, the entry of C at row r and column t is c_{rt}, which is given by the usual rule for matrix multiplication:

$$c_{rt} = \sum_{s=1}^{m} a_{rs} b_{st}. \tag{4.1}$$

Suppose further that M is the maximum absolute value of the entries in A and B, so that the entries in C are bounded in absolute value by $M' := M^2 m$. Let $\ell := \mathrm{len}(M)$. To simplify calculations, let us also assume that $m \leq M$ (this is reasonable, as we want to consider large values of M, greater than say 2^{100}, and certainly, we cannot expect to work with $2^{100} \times 2^{100}$ matrices).

By just applying the formula (4.1), we can compute the entries of C using m^3 multiplications of numbers of length at most ℓ, and m^3 additions of numbers of length at most $\mathrm{len}(M')$, where $\mathrm{len}(M') \leq 2\ell + \mathrm{len}(m) = O(\ell)$. This yields a running time of

$$O(m^3 \ell^2). \tag{4.2}$$

Using the Chinese remainder theorem, we can actually do much better than this, as follows.

For every integer $n > 1$, and for all $r, t = 1, \ldots, m$, we have

$$c_{rt} \equiv \sum_{s=1}^{m} a_{rs} b_{st} \pmod{n}. \tag{4.3}$$

Moreover, if we compute integers c'_{rt} such that

$$c'_{rt} \equiv \sum_{s=1}^{m} a_{rs} b_{st} \pmod{n} \tag{4.4}$$

and if we also have

$$-n/2 \leq c'_{rt} < n/2 \quad \text{and} \quad n > 2M', \tag{4.5}$$

then we must have

$$c_{rt} = c'_{rt}. \tag{4.6}$$

To see why (4.6) follows from (4.4) and (4.5), observe that (4.3) and (4.4) imply that $c_{rt} \equiv c'_{rt} \pmod{n}$, which means that n divides $(c_{rt} - c'_{rt})$. Then from the bound $|c_{rt}| \leq M'$ and from (4.5), we obtain

$$|c_{rt} - c'_{rt}| \leq |c_{rt}| + |c'_{rt}| \leq M' + n/2 < n/2 + n/2 = n.$$

So we see that the quantity $(c_{rt} - c'_{rt})$ is a multiple of n, while at the same time this quantity is strictly less than n in absolute value; hence, this quantity must be zero. That proves (4.6).

So from the above discussion, to compute C, it suffices to compute the entries of C modulo n, where we have to make sure that we compute "balanced" remainders in the interval $[-n/2, n/2)$, rather than the more usual "least non-negative" remainders.

To compute C modulo n, we choose a number of small integers $n_1, \ldots, n_k$, such that the family $\{n_i\}_{i=1}^{k}$ is pairwise relatively prime, and the product $n := \prod_{i=1}^{k} n_i$ is just a bit larger than $2M'$. In practice, one would choose the n_i's to be small primes, and a table of such primes could easily be computed in advance, so that all problems up to a given size could be handled. For example, the product of all primes of at most 16 bits is a number that has more than 90,000 bits. Thus, by simply pre-computing and storing a table of small primes, we can handle input matrices with quite large entries (up to about 45,000 bits).

Let us assume that we have pre-computed appropriate small primes $n_1, \ldots, n_k$. Further, we shall assume that addition and multiplication modulo each n_i can be done in *constant* time. This is reasonable from a practical (and theoretical) point of view, since such primes easily "fit" into a machine word, and we can perform modular addition and multiplication using a constant number of built-in machine operations. Finally, we assume that we do not use more n_i's than are necessary, so that $\text{len}(n) = O(\ell)$ and $k = O(\ell)$.

To compute C, we execute the following steps:

1. For each $i = 1, \ldots, k$, do the following:
   (a) compute $\hat{a}_{rs}^{(i)} \leftarrow a_{rs} \bmod n_i$ for $r, s = 1, \ldots, m$,
   (b) compute $\hat{b}_{st}^{(i)} \leftarrow b_{st} \bmod n_i$ for $s, t = 1, \ldots, m$,
   (c) for $r, t = 1, \ldots, m$, compute
   $$\hat{c}_{rt}^{(i)} \leftarrow \sum_{s=1}^{m} \hat{a}_{rs}^{(i)} \hat{b}_{st}^{(i)} \bmod n_i.$$
2. For each $r, t = 1, \ldots, m$, apply the Chinese remainder theorem to $\hat{c}_{rt}^{(1)}, \hat{c}_{rt}^{(2)}, \ldots, \hat{c}_{rt}^{(k)}$, obtaining an integer c_{rt}, which should be computed as a balanced remainder modulo n, so that $-n/2 \leq c_{rt} < n/2$.
3. Output the matrix C, whose entry in row r and column t is c_{rt}.

Note that in step 2, if our Chinese remainder algorithm happens to be implemented to return an integer a with $0 \leq a < n$, we can easily get a balanced remainder by just subtracting n from a if $a \geq n/2$.

The correctness of the above algorithm has already been established. Let us now analyze its running time. The running time of steps 1a and 1b is easily seen to be $O(m^2\ell^2)$. Under our assumption about the cost of arithmetic modulo small primes, the cost of step 1c is $O(m^3 k)$, and since $k = O(\ell)$, the cost of this step is $O(m^3\ell)$. Finally, by Theorem 4.6, the cost of step 2 is $O(m^2\ell^2)$. Thus, the total running time of this algorithm is

$$O(m^2\ell^2 + m^3\ell).$$

This is a significant improvement over (4.2); for example, if $\ell \approx m$, then the running time of the original algorithm is $O(m^5)$, while the running time of the modular algorithm is $O(m^4)$.

EXERCISE 4.17. Apply the ideas above to the problem of computing the product of two polynomials whose coefficients are large integers. First, determine the running time of the "obvious" algorithm for multiplying two such polynomials, then design and analyze a "modular" algorithm.

4.5 An effective version of Fermat's two squares theorem

We proved in Theorem 2.34 (in §2.8.4) that every prime $p \equiv 1 \pmod 4$ can be expressed as a sum of two squares of integers. In this section, we make this theorem computationally effective; that is, we develop an efficient algorithm that takes as input a prime $p \equiv 1 \pmod 4$, and outputs integers r and t such that $p = r^2 + t^2$.

One essential ingredient in the proof of Theorem 2.34 was Thue's lemma (Theorem 2.33). This lemma asserts the existence of certain numbers, and we proved it using the "pigeonhole principle," which unfortunately does not translate directly into an efficient algorithm to actually find these numbers. However, we can show that these numbers arise as a "natural by-product" of the extended Euclidean algorithm. To make this more precise, let us introduce some notation. For integers a, b, with $a \geq b \geq 0$, let us define

$$\mathrm{EEA}(a, b) := \big\{(r_i, s_i, t_i)\big\}_{i=0}^{\lambda+1},$$

where r_i, s_i, and t_i, for $i = 0, \ldots, \lambda + 1$, are defined as in Theorem 4.3.

Theorem 4.7 (Effective Thue's lemma). *Let $n, b, r^*, t^* \in \mathbb{Z}$, with $0 \leq b < n$ and $0 < r^* \leq n < r^* t^*$. Further, let $\mathrm{EEA}(n, b) = \{(r_i, s_i, t_i)\}_{i=0}^{\lambda+1}$, and let j be the smallest index (among $0, \ldots, \lambda + 1$) such that $r_j < r^*$. Then, setting $r := r_j$ and $t := t_j$, we have*

$$r \equiv bt \pmod{n}, \quad 0 \leq r < r^*, \quad \text{and} \quad 0 < |t| < t^*.$$

Proof. Since $r_0 = n \geq r^* > 0 = r_{\lambda+1}$, the value of the index j is well defined; moreover, $j \geq 1$ and $r_{j-1} \geq r^*$. It follows that

$$\begin{aligned} |t_j| &\leq n/r_{j-1} \quad \text{(by part (v) of Theorem 4.3)} \\ &\leq n/r^* \\ &< t^* \quad \text{(since } n < r^* t^*\text{)}. \end{aligned}$$

Since $j \geq 1$, by part (iv) of Theorem 4.3, we have $|t_j| \geq |t_1| > 0$. Finally, since $r_j = ns_j + bt_j$, we have $r_j \equiv bt_j \pmod{n}$. □

What this theorem says is that given n, b, r^*, t^*, to find the desired values r and t, we run the extended Euclidean algorithm on input n, b. This generates a sequence of remainders $r_0 > r_1 > r_2 > \cdots$, where $r_0 = n$ and $r_1 = b$. If r_j is the first remainder in this sequence that falls below r^*, and if s_j and t_j are the corresponding numbers computed by the extended Euclidean algorithm, then $r := r_j$ and $t := t_j$ do the job.

The other essential ingredient in the proof of Theorem 2.34 was Theorem 2.31, which guarantees the existence of a square root of -1 modulo p when p is a prime congruent to 1 modulo 4. We need an effective version of this result as well. Later, in Chapter 12, we will study the general problem of computing square roots modulo primes. Right now, we develop an algorithm for this special case.

Assume we are given a prime $p \equiv 1 \pmod{4}$, and we want to compute $\beta \in \mathbb{Z}_p^*$ such that $\beta^2 = -1$. By Theorem 2.32, it suffices to find $\gamma \in \mathbb{Z}_p^* \setminus (\mathbb{Z}_p^*)^2$, since then $\beta := \gamma^{(p-1)/4}$ (which we can efficiently compute via repeated squaring) satisfies

$\beta^2 = -1$. While there is no known efficient, deterministic algorithm to find such a γ, we do know that half the elements of $\mathbb{Z}_p^*$ are squares and half are not (see Theorem 2.20), which suggests the following simple "trial and error" strategy to compute β:

repeat
  choose $\gamma \in \mathbb{Z}_p^*$
  compute $\beta \leftarrow \gamma^{(p-1)/4}$
until $\beta^2 = -1$
output β

As an algorithm, this is not fully specified, as we have to specify a procedure for selecting γ in each loop iteration. A reasonable approach is to simply choose γ *at random*: this would be an example of a *probabilistic algorithm*, a notion that we will study in detail in Chapter 9. Let us assume for the moment that this makes sense from a mathematical and algorithmic point of view, so that with each loop iteration, we have a 50% chance of picking a "good" γ, that is, one that is not in $(\mathbb{Z}_p^*)^2$. From this, it follows that with high probability, we should find a "good" γ in just a few loop iterations (the probability that after k loop iterations we still have not found one is $1/2^k$), and that the *expected* number of loop iterations is just 2. The running time of each loop iteration is dominated by the cost of repeated squaring, which is $O(\text{len}(p)^3)$. It follows that the *expected running time* of this algorithm (we will make this notion precise in Chapter 9) is $O(\text{len}(p)^3)$.

Let us now put all the ingredients together to get an algorithm to find r, t such that $p = r^2 + t^2$.

1. Find $\beta \in \mathbb{Z}_p^*$ such that $\beta^2 = -1$, using the above "trial and error" strategy.
2. Set $b \leftarrow \text{rep}(\beta)$ (so that $\beta = [b]$ and $b \in \{0, \ldots, p-1\}$).
3. Run the extended Euclidean algorithm on input p, b to obtain $\text{EEA}(p, b)$, and then apply Theorem 4.7 with $n := p$, b, and $r^* := t^* := \lfloor \sqrt{p} \rfloor + 1$, to obtain the values r and t.
4. Output r, t.

When this algorithm terminates, we have $r^2 + t^2 = p$, as required: as we argued in the proof of Theorem 2.34, since $r \equiv bt \pmod{p}$ and $b^2 \equiv -1 \pmod{p}$, it follows that $r^2 + t^2 \equiv 0 \pmod{p}$, and since $0 < r^2 + t^2 < 2p$, we must have $r^2 + t^2 = p$. The (expected) running time of step 1 is $O(\text{len}(p)^3)$. The running time of step 3 is $O(\text{len}(p)^2)$ (note that we can compute $\lfloor \sqrt{p} \rfloor$ in time $O(\text{len}(p)^2)$, using the algorithm in Exercise 3.29). Thus, the total (expected) running time is $O(\text{len}(p)^3)$.

Example 4.4. One can check that $p := 1009$ is prime and $p \equiv 1 \pmod{4}$. Let us express p as a sum of squares using the above algorithm. First, we need to find a

square root of -1 modulo p. Let us just try a random number, say 17, and raise this to the power $(p-1)/4 = 252$. One can calculate that $17^{252} \equiv 469 \pmod{1009}$, and $469^2 \equiv -1 \pmod{1009}$. So we were lucky with our first try. Now we run the extended Euclidean algorithm on input $p = 1009$ and $b = 469$, obtaining the following data:

i	r_i	q_i	s_i	t_i
0	1009		1	0
1	469	2	0	1
2	71	6	1	-2
3	43	1	-6	13
4	28	1	7	-15
5	15	1	-13	28
6	13	1	20	-43
7	2	6	-33	71
8	1	2	218	-469
9	0		-469	1009

The first r_j that falls below the threshold $r^* = \lfloor\sqrt{1009}\rfloor + 1 = 32$ is at $j = 4$, and so we set $r := 28$ and $t := -15$. One verifies that $r^2 + t^2 = 28^2 + 15^2 = 1009 = p$. □

It is natural to ask whether one can solve this problem without resorting to randomization. The answer is "yes" (see §4.8), but the only known deterministic algorithms for this problem are quite impractical (albeit polynomial time). This example illustrates the utility of randomization as an algorithm design technique, one that has proved to be invaluable in solving numerous algorithmic problems in number theory; indeed, in §3.4 we already mentioned its use in connection with primality testing, and we will explore many other applications as well (after putting the notion of a probabilistic algorithm on firm mathematical ground in Chapter 9).

4.6 Rational reconstruction and applications

In the previous section, we saw how to apply the extended Euclidean algorithm to obtain an effective version of Thue's lemma. This lemma asserts that for given integers n and b, there exists a pair of integers (r, t) satisfying $r \equiv bt \pmod{n}$, and contained in a prescribed rectangle, provided the area of the rectangle is large enough, relative to n. In this section, we first prove a corresponding uniqueness theorem, under the assumption that the area of the rectangle is not too large; of course, if $r \equiv bt \pmod{n}$, then for any non-zero integer q, we also have $rq \equiv b(tq) \pmod{n}$, and so we can only hope to guarantee that the *ratio* r/t is unique. After proving this uniqueness theorem, we show how to make this theorem computationally effective, and then develop several very neat applications.

The basic uniqueness statement is as follows:

Theorem 4.8. *Let $n, b, r^*, t^* \in \mathbb{Z}$ with $r^* \geq 0$, $t^* > 0$, and $n > 2r^*t^*$. Further, suppose that $r, t, r', t' \in \mathbb{Z}$ satisfy*

$$r \equiv bt \pmod{n}, \quad |r| \leq r^*, \quad 0 < |t| \leq t^*, \tag{4.7}$$

$$r' \equiv bt' \pmod{n}, \quad |r'| \leq r^*, \quad 0 < |t'| \leq t^*. \tag{4.8}$$

Then $r/t = r'/t'$.

Proof. Consider the two congruences

$$r \equiv bt \pmod{n},$$
$$r' \equiv bt' \pmod{n}.$$

Subtracting t times the second from t' times the first, we obtain

$$rt' - r't \equiv 0 \pmod{n}.$$

However, we also have

$$|rt' - r't| \leq |r||t'| + |r'||t| \leq 2r^*t^* < n.$$

Thus, $rt' - r't$ is a multiple of n, but less than n in absolute value; the only possibility is that $rt' - r't = 0$, which means $r/t = r'/t'$. □

Now suppose that we are given $n, b, r^*, t^* \in \mathbb{Z}$ as in the above theorem; moreover, suppose that there exist $r, t \in \mathbb{Z}$ satisfying (4.7), but that these values are *not* given to us. Note that under the hypothesis of Theorem 4.8, Thue's lemma cannot be used to ensure the existence of such r and t, but in our eventual applications, we will have other reasons that will guarantee this. We would like to find $r', t' \in \mathbb{Z}$ satisfying (4.8), and if we do this, then by the theorem, we know that $r/t = r'/t'$. We call this the **rational reconstruction problem**. We can solve this problem efficiently using the extended Euclidean algorithm; indeed, just as in the case of our effective version of Thue's lemma, the desired values of r' and t' appear as "natural by-products" of that algorithm. To state the result precisely, let us recall the notation we introduced in the last section: for integers a, b, with $a \geq b \geq 0$, we defined

$$\mathrm{EEA}(a, b) := \big\{(r_i, s_i, t_i)\big\}_{i=0}^{\lambda+1},$$

where r_i, s_i, and t_i, for $i = 0, \ldots, \lambda + 1$, are defined as in Theorem 4.3.

Theorem 4.9 (Rational reconstruction). *Let $n, b, r^*, t^* \in \mathbb{Z}$ with $0 \leq b < n$, $0 \leq r^* < n$, and $t^* > 0$. Further, let $\mathrm{EEA}(n, b) = \{(r_i, s_i, t_i)\}_{i=0}^{\lambda+1}$, and let j be the smallest index (among $0, \ldots, \lambda + 1$) such that $r_j \leq r^*$, and set*

$$r' := r_j, \quad s' := s_j, \quad \text{and} \quad t' := t_j.$$

Finally, suppose that there exist $r, s, t \in \mathbb{Z}$ *such that*

$$r = ns + bt, \quad |r| \leq r^*, \quad \text{and} \quad 0 < |t| \leq t^*.$$

Then we have:

(i) $0 < |t'| \leq t^*$;

(ii) if $n > 2r^*t^*$, *then for some non-zero integer* q,

$$r = r'q, \quad s = s'q, \quad \text{and} \quad t = t'q.$$

Proof. Since $r_0 = n > r^* \geq 0 = r_{\lambda+1}$, the value of j is well defined, and moreover, $j \geq 1$, and we have the inequalities

$$0 \leq r_j \leq r^* < r_{j-1}, \quad 0 < |t_j|, \quad |r| \leq r^*, \quad \text{and} \quad 0 < |t| \leq t^*, \tag{4.9}$$

along with the identities

$$r_{j-1} = ns_{j-1} + bt_{j-1}, \tag{4.10}$$

$$r_j = ns_j + bt_j, \tag{4.11}$$

$$r = ns + bt. \tag{4.12}$$

We now turn to part (i) of the theorem. Our goal is to prove that

$$|t_j| \leq t^*. \tag{4.13}$$

This is the hardest part of the proof. To this end, let

$$\varepsilon := s_j t_{j-1} - s_{j-1} t_j, \quad \mu := (t_{j-1}s - s_{j-1}t)/\varepsilon, \quad \nu := (s_j t - t_j s)/\varepsilon.$$

Since $\varepsilon = \pm 1$, the numbers μ and ν are integers; moreover, one may easily verify that they satisfy the equations

$$s_j \mu + s_{j-1}\nu = s, \tag{4.14}$$

$$t_j \mu + t_{j-1}\nu = t. \tag{4.15}$$

We now use these identities to prove (4.13). We consider three cases:

(i) Suppose $\nu = 0$. In this case, (4.15) implies $t_j \mid t$, and since $t \neq 0$, this implies $|t_j| \leq |t| \leq t^*$.

(ii) Suppose $\mu\nu < 0$. In this case, since t_j and t_{j-1} have opposite sign, (4.15) implies $|t| = |t_j\mu| + |t_{j-1}\nu| \geq |t_j|$, and so again, we have $|t_j| \leq |t| \leq t^*$.

(iii) The only remaining possibility is that $\nu \neq 0$ and $\mu\nu \geq 0$. We argue that this is impossible. Adding n times (4.14) to b times (4.15), and using the identities (4.10), (4.11), and (4.12), we obtain

$$r_j \mu + r_{j-1}\nu = r.$$

If $\nu \neq 0$ and μ and ν had the same sign, we would have $|r| = |r_j\mu| + |r_{j-1}\nu| \geq r_{j-1}$, and hence $r_{j-1} \leq |r| \leq r^*$; however, this contradicts the fact that $r_{j-1} > r^*$.

That proves the inequality (4.13). We now turn to the proof of part (ii) of the theorem, which relies critically on this inequality. Assume that

$$n > 2r^*t^*. \tag{4.16}$$

From (4.11) and (4.12), we have

$$r_j \equiv bt_j \pmod{n} \quad \text{and} \quad r \equiv bt \pmod{n}.$$

Combining this with the inequalities (4.9), (4.13), and (4.16), we see that the hypotheses of Theorem 4.8 are satisfied, and so we may conclude that

$$rt_j - r_jt = 0. \tag{4.17}$$

Subtracting t_j times (4.12) from t times (4.11), and using the identity (4.17), we obtain $n(st_j - s_jt) = 0$, and hence

$$st_j - s_jt = 0. \tag{4.18}$$

From (4.18), we see that $t_j \mid s_jt$, and since $\gcd(s_j, t_j) = 1$, we must have $t_j \mid t$. So $t = t_jq$ for some q, and we must have $q \neq 0$ since $t \neq 0$. Substituting t_jq for t in equations (4.17) and (4.18) yields $r = r_jq$ and $s = s_jq$. That proves part (ii) of the theorem. □

In our applications in this text, we shall only directly use part (ii) of this theorem; however, part (i) has applications as well (see Exercise 4.18).

4.6.1 Application: recovering fractions from their decimal expansions

It should be a familiar fact to the reader that every real number has a decimal expansion, and that this decimal expansion is unique, provided one rules out those expansions that end in an infinite run of 9's (e.g., $1/10 = 0.1000\cdots = 0.0999\cdots$).

Now suppose that Alice and Bob play a game. Alice thinks of a rational number $z := s/t$, where s and t are integers with $0 \leq s < t$, and tells Bob some of the high-order digits in the decimal expansion of z. Bob's goal in the game is to determine z. Can he do this?

The answer is "yes," provided Bob knows an upper bound M on t, and provided Alice gives Bob enough digits. Of course, Bob probably remembers from grade school that the decimal expansion of z is ultimately periodic, and that given enough digits of z so that the periodic part is included, he can recover z; however, this technique is quite useless in practice, as the length of the period can be huge—

$\Theta(M)$ in the worst case (see Exercises 4.21–4.23 below). The method we discuss here requires only $O(\text{len}(M))$ digits.

Suppose Alice gives Bob the high-order k digits of z, for some $k \geq 1$. That is, if

$$z = 0 \,.\, z_1 z_2 z_3 \cdots \tag{4.19}$$

is the decimal expansion of z, then Alice gives Bob $z_1, \ldots, z_k$. Now, if 10^k is much smaller than M^2, the number z is not even uniquely determined by these digits, since there are $\Omega(M^2)$ distinct rational numbers of the form s/t, with $0 \leq s < t \leq M$ (see Exercise 1.33). However, if $10^k > 2M^2$, then not only is z uniquely determined by $z_1, \ldots, z_k$, but using Theorem 4.9, Bob can efficiently compute it.

We shall presently describe efficient algorithms for both Alice and Bob, but before doing so, we make a few general observations about the decimal expansion of z. Let e be an arbitrary non-negative integer, and suppose that the decimal expansion of z is as in (4.19). Observe that

$$10^e z = z_1 \cdots z_e \,.\, z_{e+1} z_{e+2} \cdots .$$

It follows that

$$\lfloor 10^e z \rfloor = z_1 \cdots z_e \,.\, 0 \,. \tag{4.20}$$

Since $z = s/t$, if we set $r := 10^e s \bmod t$, then $10^e s = \lfloor 10^e z \rfloor t + r$, and dividing this by t, we have $10^e z = \lfloor 10^e z \rfloor + r/t$, where $r/t \in [0, 1)$. Therefore,

$$\frac{10^e s \bmod t}{t} = 0 \,.\, z_{e+1} z_{e+2} z_{e+3} \cdots . \tag{4.21}$$

Next, consider Alice. Based on the above discussion, Alice may use the following simple, iterative algorithm to compute $z_1, \ldots, z_k$, for arbitrary $k \geq 1$, after she chooses s and t:

$x_1 \leftarrow s$
for $i \leftarrow 1$ to k do
    $y_i \leftarrow 10 x_i$
    $z_i \leftarrow \lfloor y_i / t \rfloor$
    $x_{i+1} \leftarrow y_i \bmod t$
output $z_1, \ldots, z_k$

Correctness follows easily from the observation that for each $i = 1, 2, \ldots$, we have $x_i = 10^{i-1} s \bmod t$; indeed, applying (4.21) with $e = i - 1$, we have $x_i/t = 0 \,.\, z_i z_{i+1} z_{i+2} \cdots$, and consequently, by (4.20) with $e = 1$ and x_i/t in the role of z, we have $\lfloor 10 x_i / t \rfloor = z_i$. The total time for Alice's computation is $O(k \,\text{len}(M))$, since each loop iteration takes time $O(\text{len}(M))$.

Finally, consider Bob. Given the high-order digits $z_1, \ldots, z_k$ of $z = s/t$, along with the upper bound M on t, he can compute z as follows:

1. Compute $n \leftarrow 10^k$ and $b \leftarrow \sum_{i=1}^{k} z_i 10^{k-i}$.
2. Run the extended Euclidean algorithm on input n, b to obtain $\mathrm{EEA}(n, b)$, and then apply Theorem 4.9 with n, b, and $r^* := t^* := M$, to obtain the values r', s', t'.
3. Output the rational number $-s'/t'$.

Let us analyze this algorithm, assuming that $10^k > 2M^2$.

For correctness, we must show that $z = -s'/t'$. To prove this, observe that by (4.20) with $e = k$, we have $b = \lfloor nz \rfloor = \lfloor ns/t \rfloor$. Moreover, if we set $r := ns \bmod t$, then we have

$$r = ns - bt,\ 0 \le r < t \le r^*,\ 0 < t \le t^*, \text{ and } n > 2r^*t^*.$$

It follows that the integers s', t' from Theorem 4.9 satisfy $s = s'q$ and $-t = t'q$ for some non-zero integer q. Thus, $s/t = -s'/t'$, as required. As a bonus, since the extended Euclidean algorithm guarantees that $\gcd(s', t') = 1$, not only do we obtain z, but we obtain z expressed as a fraction in lowest terms.

We leave it to the reader to verify that Bob's computation may be performed in time $O(k^2)$.

We conclude that both Alice and Bob can successfully play this game with k chosen so that $k = O(\mathrm{len}(M))$, in which case, their algorithms run in time $O(\mathrm{len}(M)^2)$.

Example 4.5. Alice chooses integers s, t, with $0 \le s < t \le 1000$, and tells Bob the high-order seven digits in the decimal expansion of $z := s/t$, from which Bob should be able to compute z. Suppose $s = 511$ and $t = 710$. Then $s/t = 0.7197183098591549\cdots$. Bob receives the digits $7, 1, 9, 7, 1, 8, 3$, and computes $n = 10^7$ and $b = 7197183$. Running the extended Euclidean algorithm on input n, b, Bob obtains the data in Fig. 4.1. The first r_j that meets the threshold $r^* = 1000$ is at $j = 10$, and Bob reads off $s' = 511$ and $t' = -710$, from which he obtains $z = -s'/t' = 511/710$.

Another interesting phenomenon to observe in Fig. 4.1 is that the fractions $-s_i/t_i$ are very good approximations to the fraction $b/n = 7197183/10000000$; indeed, if we compute the error terms $b/n + s_i/t_i$ for $i = 1, \ldots, 5$, we get (approximately)

$$0.72, \quad -0.28, \quad 0.053, \quad -0.03, \quad 0.0054.$$

Thus, we can approximate the "complicated" fraction $7197183/10000000$ by the "very simple" fraction $5/7$, introducing an absolute error of less than 0.006. Exercise 4.18 explores this "data compression" capability of Euclid's algorithm in more generality. □

i	r_i	q_i	s_i	t_i
0	10000000		1	0
1	7197183	1	0	1
2	2802817	2	1	-1
3	1591549	1	-2	3
4	1211268	1	3	-4
5	380281	3	-5	7
6	70425	5	18	-25
7	28156	2	-95	132
8	14113	1	208	-289
9	14043	1	-303	421
10	70	200	511	-710
11	43	1	-102503	142421
12	27	1	103014	-143131
13	16	1	-205517	285552
14	11	1	308531	-428683
15	5	2	-514048	714235
16	1	5	1336627	-1857153
17	0		-7197183	10000000

Fig. 4.1. Bob's data from the extended Euclidean algorithm

4.6.2 Application: Chinese remaindering with errors

One interpretation of the Chinese remainder theorem is that if we "encode" an integer a, with $0 \leq a < n$, as the sequence $(a_1, \ldots, a_k)$, where $a_i = a \bmod n_i$ for $i = 1, \ldots, k$, then we can efficiently recover a from this encoding. Here, of course, $n = n_1 \cdots n_k$, and the family $\{n_i\}_{i=1}^k$ is pairwise relatively prime.

Suppose that Alice encodes a as $(a_1, \ldots, a_k)$, and sends this encoding to Bob over some communication network; however, because the network is not perfect, during the transmission of the encoding, some (but hopefully not too many) of the values $a_1, \ldots, a_k$ may be corrupted. The question is, can Bob still efficiently recover the original a from its corrupted encoding?

To make the problem more precise, suppose that the original, correct encoding of a is $(a_1, \ldots, a_k)$, and the corrupted encoding is $(b_1, \ldots, b_k)$. Let us define $G \subseteq \{1, \ldots, k\}$ to be the set of "good" positions i with $a_i = b_i$, and $B \subseteq \{1, \ldots, k\}$ to be the set of "bad" positions i with $a_i \neq b_i$. We shall assume that $|B| \leq \ell$, where ℓ is some specified parameter.

Of course, if Bob hopes to recover a, we need to build some redundancy into the system; that is, we must require that $0 \leq a \leq M$ for some bound M that is

somewhat smaller than n. Now, if Bob knew the location of bad positions, and if the product of the n_i's at the good positions exceeds M, then Bob could simply discard the errors, and reconstruct a by applying the Chinese remainder theorem to the a_i's and n_i's at the good positions. However, in general, Bob will not know a priori the locations of the bad positions, and so this approach will not work.

Despite these apparent difficulties, Theorem 4.9 may be used to solve the problem quite easily, as follows. Let P be an upper bound on the product of any ℓ of the integers $n_1, \ldots, n_k$ (e.g., we could take P to be the product of the ℓ largest numbers among $n_1, \ldots, n_k$). Further, let us assume that $n > 2MP^2$.

Now, suppose Bob obtains the corrupted encoding $(b_1, \ldots, b_k)$. Here is what Bob does to recover a:

1. Apply the Chinese remainder theorem, obtaining the integer b satisfying $0 \leq b < n$ and $b \equiv b_i \pmod{n_i}$ for $i = 1, \ldots, k$.
2. Run the extended Euclidean algorithm on input n, b to obtain $\mathrm{EEA}(n, b)$, and then apply Theorem 4.9 with n, b, $r^* := MP$ and $t^* := P$, to obtain values r', s', t'.
3. If $t' \mid r'$, output the integer r'/t'; otherwise, output "error."

We claim that the above procedure outputs a, under our assumption that the set B of bad positions is of size at most ℓ. To see this, let $t := \prod_{i \in B} n_i$. By construction, we have $1 \leq t \leq P$. Also, let $r := at$, and note that $0 \leq r \leq r^*$ and $0 < t \leq t^*$. We claim that

$$r \equiv bt \pmod{n}. \tag{4.22}$$

To show that (4.22) holds, it suffices to show that

$$at \equiv bt \pmod{n_i} \tag{4.23}$$

for all $i = 1, \ldots, k$. To show this, for each index i we consider two cases:

Case 1: $i \in G$. In this case, we have $a_i = b_i$, and therefore,

$$at \equiv a_i t \equiv b_i t \equiv bt \pmod{n_i}.$$

Case 2: $i \in B$. In this case, we have $n_i \mid t$, and therefore,

$$at \equiv 0 \equiv bt \pmod{n_i}.$$

Thus, (4.23) holds for all $i = 1, \ldots, k$, and so it follows that (4.22) holds. Therefore, the values r', t' obtained from Theorem 4.9 satisfy

$$\frac{r'}{t'} = \frac{r}{t} = \frac{at}{t} = a.$$

One easily checks that both the procedures to encode and decode a value a run in time $O(\text{len}(n)^2)$.

The above scheme is an example of an **error correcting code**, and is actually the integer analog of a **Reed–Solomon code**.

Example 4.6. Suppose we want to encode a 1024-bit message as a sequence of 16-bit blocks, so that the above scheme can correct up to 3 corrupted blocks. Without any error correction, we would need just $1024/16 = 64$ blocks. However, to correct this many errors, we need a few extra blocks; in fact, 7 will do.

Of course, a 1024-bit message can naturally be viewed as an integer a in the set $\{0, \ldots, 2^{1024} - 1\}$, and the ith 16-bit block in the encoding can be viewed as an integer a_i in the set $\{0, \ldots, 2^{16} - 1\}$. Setting $k := 71$, we select k primes, $n_1, \ldots, n_k$, each 16-bits in length. In fact, let us choose $n_1, \ldots, n_k$ to be the *largest* k primes under 2^{16}. If we do this, then the smallest prime among the n_i's turns out to be 64717, which is greater than $2^{15.98}$. We may set $M := 2^{1024}$, and since we want to correct up to 3 errors, we may set $P := 2^{3 \cdot 16}$. Then with $n := \prod_i n_i$, we have

$$n > 2^{71 \cdot 15.98} = 2^{1134.58} > 2^{1121} = 2^{1+1024+6 \cdot 16} = 2MP^2.$$

Thus, with these parameter settings, the above scheme will correct up to 3 corrupted blocks. This comes at a cost of increasing the length of the message from 1024 bits to $71 \cdot 16 = 1136$ bits, an increase of about 11%. □

4.6.3 Applications to symbolic algebra

Rational reconstruction also has a number of applications in symbolic algebra. We briefly sketch one such application here. Suppose that we want to find the solution v to the equation $vA = w$, where we are given as input a non-singular square integer matrix A and an integer vector w. The solution vector v will, in general, have rational entries. We stress that we want to compute the *exact* solution v, and not some floating point approximation to it. Now, we could solve for v directly using Gaussian elimination; however, the intermediate quantities computed by that algorithm would be rational numbers whose numerators and denominators might get quite large, leading to a rather lengthy computation (however, it is possible to show that the overall running time is still polynomial in the input length).

Another approach is to compute a solution vector modulo n, where n is a power of a prime that does not divide the determinant of A. Provided n is large enough, one can then recover the solution vector v using rational reconstruction. With this approach, all of the computations can be carried out using arithmetic on integers not too much larger than n, leading to a more efficient algorithm. More of the details of this procedure are developed later, in Exercise 14.18.

EXERCISE 4.18. Let $n, b \in \mathbb{Z}$ with $0 \leq b < n$, and let $\mathrm{EEA}(n, b) = \{(r_i, s_i, t_i)\}_{i=0}^{\lambda+1}$. This exercise develops some key properties of the fractions $-s_i/t_i$ as approximations to b/n. For $i = 1, \ldots, \lambda + 1$, let $\varepsilon_i := b/n + s_i/t_i$.

(a) Show that $\varepsilon_i = r_i/t_i n$ for $i = 1, \ldots, \lambda + 1$.

(b) Show that successive ε_i's strictly decrease in absolute value, and alternate in sign.

(c) Show that $|\varepsilon_i| < 1/t_i^2$ for $i = 1, \ldots, \lambda$, and $\varepsilon_{\lambda+1} = 0$.

(d) Show that for all $s, t \in \mathbb{Z}$ with $t \neq 0$, if $|b/n - s/t| < 1/2t^2$, then $s/t = -s_i/t_i$ for some $i = 1, \ldots, \lambda + 1$. Hint: use part (ii) of Theorem 4.9.

(e) Consider a fixed index $i \in \{2, \ldots, \lambda + 1\}$. Show that for all $s, t \in \mathbb{Z}$, if $0 < |t| \leq |t_i|$ and $|b/n - s/t| \leq |\varepsilon_i|$, then $s/t = -s_i/t_i$. In this sense, $-s_i/t_i$ is the unique, best approximation to b/n among all fractions of denominator at most $|t_i|$. Hint: use part (i) of Theorem 4.9.

EXERCISE 4.19. Using the decimal approximation $\pi \approx 3.141592654$, apply Euclid's algorithm to calculate a rational number of denominator less than 1000 that is within 10^{-6} of π. Illustrate the computation with a table as in Fig. 4.1.

EXERCISE 4.20. Show that given integers s, t, k, with $0 \leq s < t$, and $k > 0$, we can compute the kth digit in the decimal expansion of s/t in time $O(\mathrm{len}(k)\,\mathrm{len}(t)^2)$.

For the following exercises, we need a definition. Let $\Psi = \{z_i\}_{i=1}^{\infty}$ be a sequence of elements drawn from some arbitrary set. For integers $k \geq 0$ and $\ell \geq 1$, we say that Ψ is (k, ℓ)-**periodic** if $z_i = z_{i+\ell}$ for all $i > k$; in addition, we say that Ψ is **ultimately periodic** if it is (k, ℓ)-periodic for some (k, ℓ).

EXERCISE 4.21. Show that if a sequence Ψ is ultimately periodic, then it is (k^*, ℓ^*)-periodic for some uniquely determined pair (k^*, ℓ^*) for which the following holds: for every pair (k, ℓ) such that Ψ is (k, ℓ)-periodic, we have $k^* \leq k$ and $\ell^* \mid \ell$.

The value ℓ^* in the above exercise is called the **period** of Ψ, and k^* is called the **pre-period** of Ψ. If its pre-period is zero, then Ψ is called **purely periodic**.

EXERCISE 4.22. Let z be a real number whose decimal expansion is an ultimately periodic sequence. Show that z is rational.

EXERCISE 4.23. Let $z = s/t \in \mathbb{Q}$, where s and t are relatively prime integers with $0 \leq s < t$. Show that:

(a) there exist integers k, k' such that $0 \leq k < k'$ and $s10^k \equiv s10^{k'} \pmod{t}$;

(b) for all integers k, k' with $0 \leq k < k'$, the decimal expansion of z is $(k, k' - k)$-periodic if and only if $s10^k \equiv s10^{k'} \pmod{t}$;

(c) if $\gcd(10, t) = 1$, then the decimal expansion of z is purely periodic with period equal to the multiplicative order of 10 modulo t;

(d) more generally, if k is the smallest non-negative integer such that 10 and $t' := t/\gcd(10^k, t)$ are relatively prime, then the decimal expansion of z is ultimately periodic with pre-period k and period equal to the multiplicative order of 10 modulo t'.

A famous conjecture of Artin postulates that for every integer d, not equal to -1 or to the square of an integer, there are infinitely many primes t such that d has multiplicative order $t - 1$ modulo t. If Artin's conjecture is true, then by part (c) of the previous exercise, there are infinitely many primes t such that the decimal expansion of s/t, for every s with $0 < s < t$, is a purely periodic sequence of period $t - 1$. In light of these observations, the "grade school" method of computing a fraction from its decimal expansion using the period is hopelessly impractical.

4.7 The RSA cryptosystem

One of the more exciting uses of number theory in recent decades is its application to cryptography. In this section, we give a brief overview of the RSA cryptosystem, named after its inventors Rivest, Shamir, and Adleman. At this point in the text, we already have the concepts and tools at our disposal necessary to understand the basic operation of this system, even though a full understanding of the system will require other ideas that will be developed later in the text.

Suppose that Alice wants to send a secret message to Bob over an insecure network. An adversary may be able to eavesdrop on the network, and so sending the message "in the clear" is not an option. Using older, more traditional cryptographic techniques would require that Alice and Bob share a secret key between them; however, this creates the problem of securely generating such a shared secret. The RSA cryptosystem is an example of a **public key cryptosystem**. To use the system, Bob simply places a "public key" in the equivalent of an electronic telephone book, while keeping a corresponding "private key" secret. To send a secret message to Bob, Alice obtains Bob's public key from the telephone book, and uses this to encrypt her message. Upon receipt of the encrypted message, Bob uses his private key to decrypt it, obtaining the original message.

Here is how the RSA cryptosystem works. To generate a public key/private key pair, Bob generates two very large, random primes p and q, with $p \neq q$. To be secure, p and q should be quite large; in practice, they are chosen to be around 512 bits in length. Efficient algorithms for generating such primes exist, and we shall discuss them in detail later in the text (that there are sufficiently many primes of a given bit length will be discussed in Chapter 5; algorithms for generating them will

be discussed at a high level in §9.4, and in greater detail in Chapter 10). Next, Bob computes $n := pq$. Bob also selects an integer $e > 1$ such that $\gcd(e, \varphi(n)) = 1$, where φ is Euler's phi function. Here, $\varphi(n) = (p-1)(q-1)$. Finally, Bob computes $d := e^{-1} \bmod \varphi(n)$, using the extended Euclidean algorithm. The public key is the pair (n, e), and the private key is the pair (n, d). The integer e is called the "encryption exponent" and d is called the "decryption exponent." In practice, the integers n and d are about 1024 bits in length, while e is usually significantly shorter.

After Bob publishes his public key (n, e), Alice may send a secret message to Bob as follows. Suppose that a message is encoded in some canonical way as a number between 0 and $n - 1$—we can always interpret a bit string of length less than $\text{len}(n)$ as such a number. Thus, we may assume that a message is an element α of $\mathbb{Z}_n$. To encrypt the message α, Alice simply computes $\beta := \alpha^e$ using repeated squaring. The encrypted message is β. When Bob receives β, he computes $\gamma := \beta^d$, and interprets γ as a message.

The most basic requirement of any encryption scheme is that decryption should "undo" encryption. In this case, this means that for all $\alpha \in \mathbb{Z}_n$, we should have

$$(\alpha^e)^d = \alpha. \tag{4.24}$$

If $\alpha \in \mathbb{Z}_n^*$, then this is clearly the case, since we have $ed = 1 + \varphi(n)k$ for some positive integer k, and hence by Euler's theorem (Theorem 2.13), we have

$$(\alpha^e)^d = \alpha^{ed} = \alpha^{1+\varphi(n)k} = \alpha \cdot \alpha^{\varphi(n)k} = \alpha.$$

To argue that (4.24) holds in general, let α be an arbitrary element of $\mathbb{Z}_n$, and suppose $\alpha = [a]_n$. If $a \equiv 0 \pmod{p}$, then trivially $a^{ed} \equiv 0 \pmod{p}$; otherwise,

$$a^{ed} \equiv a^{1+\varphi(n)k} \equiv a \cdot a^{\varphi(n)k} \equiv a \pmod{p},$$

where the last congruence follows from the fact that $\varphi(n)k$ is a multiple of $p - 1$, which is a multiple of the multiplicative order of a modulo p (again by Euler's theorem). Thus, we have shown that $a^{ed} \equiv a \pmod{p}$. The same argument shows that $a^{ed} \equiv a \pmod{q}$, and these two congruences together imply that $a^{ed} \equiv a \pmod{n}$. Thus, we have shown that equation (4.24) holds for all $\alpha \in \mathbb{Z}_n$.

Of course, the interesting question about the RSA cryptosystem is whether or not it really is secure. Now, if an adversary, given only the public key (n, e), were able to factor n, then he could easily compute the decryption exponent d himself using the same algorithm used by Bob. It is widely believed that factoring n is computationally infeasible, for sufficiently large n, and so this line of attack is ineffective, barring a breakthrough in factorization algorithms. Indeed, while trying to factor n by brute-force search is clearly infeasible, there are much faster algorithms, but even these are not fast enough to pose a serious threat to the security of the RSA

cryptosystem. We shall discuss some of these faster algorithms in some detail later in the text (in Chapter 15).

Can one break the RSA cryptosystem without factoring n? For example, it is natural to ask whether one can compute the decryption exponent d without having to go to the trouble of factoring n. It turns out that the answer to this question is "no": if one could compute the decryption exponent d, then $ed - 1$ would be a multiple of $\varphi(n)$, and as we shall see later in §10.4, given any multiple of $\varphi(n)$, we can easily factor n. Thus, computing the decryption exponent is equivalent to factoring n, and so this line of attack is also ineffective. But there still could be other lines of attack. For example, even if we assume that factoring large numbers is infeasible, this is not enough to guarantee that for a given encrypted message β, the adversary is unable to compute β^d (although nobody actually knows how to do this without first factoring n).

The reader should be warned that the proper notion of security for an encryption scheme is quite subtle, and a detailed discussion of this is well beyond the scope of this text. Indeed, the simple version of RSA presented here suffers from a number of security problems (because of this, actual implementations of public-key encryption schemes based on RSA are somewhat more complicated). We mention one such problem here (others are examined in some of the exercises below). Suppose an eavesdropping adversary knows that Alice will send one of a few, known, candidate messages. For example, an adversary may know that Alice's message is either "let's meet today" or "let's meet tomorrow." In this case, the adversary can encrypt for himself each of the candidate messages, intercept Alice's actual encrypted message, and then by simply comparing encryptions, the adversary can determine which particular message Alice encrypted. This type of attack works simply because the encryption algorithm is deterministic, and in fact, any deterministic encryption algorithm will be vulnerable to this type of attack. To avoid this type of attack, one must use a *probabilistic* encryption algorithm. In the case of the RSA cryptosystem, this is often achieved by padding the message with some random bits before encrypting it (but even this must be done carefully).

EXERCISE 4.24. This exercise develops a method to speed up RSA decryption. Suppose that we are given two distinct ℓ-bit primes, p and q, an element $\beta \in \mathbb{Z}_n$, where $n := pq$, and an integer d, where $1 < d < \varphi(n)$. Using the algorithm from Exercise 3.35, we can compute β^d at a cost of essentially 2ℓ squarings in $\mathbb{Z}_n$. Show how this can be improved, making use of the factorization of n, so that the total cost is essentially that of ℓ squarings in $\mathbb{Z}_p$ and ℓ squarings in $\mathbb{Z}_q$, leading to a roughly four-fold speed-up in the running time.

EXERCISE 4.25. Alice submits a bid to an auction, and so that other bidders cannot

see her bid, she encrypts it under the public key of the auction service. Suppose that the auction service provides a public key for an RSA encryption scheme, with a modulus n. Assume that bids are encoded simply as integers between 0 and $n-1$ prior to encryption. Also, assume that Alice submits a bid that is a "round number," which in this case means that her bid is a number that is divisible by 10. Show how an eavesdropper can submit an encryption of a bid that exceeds Alice's bid by 10%, without even knowing what Alice's bid is. In particular, your attack should work even if the space of possible bids is very large.

EXERCISE 4.26. To speed up RSA encryption, one may choose a very small encryption exponent. This exercise develops a "small encryption exponent attack" on RSA. Suppose Bob, Bill, and Betty have RSA public keys with moduli n_1, n_2, and n_3, and all three use encryption exponent 3. Assume that $\{n_i\}_{i=1}^3$ is pairwise relatively prime. Suppose that Alice sends an encryption of the same message to Bob, Bill, and Betty — that is, Alice encodes her message as an integer a, with $0 \leq a < \min\{n_1, n_2, n_3\}$, and computes the three encrypted messages $\beta_i := [a^3]_{n_i}$, for $i = 1, \ldots, 3$. Show how to recover Alice's message from these three encrypted messages.

EXERCISE 4.27. To speed up RSA decryption, one might choose a small decryption exponent, and then derive the encryption exponent from this. This exercise develops a "small decryption exponent attack" on RSA. Suppose $n = pq$, where p and q are distinct primes with $\mathrm{len}(p) = \mathrm{len}(q)$. Let d and e be integers such that $1 < d < \varphi(n)$, $1 < e < \varphi(n)$, and $de \equiv 1 \pmod{\varphi(n)}$. Further, assume that $d < n^{1/4}/3$. Show how to efficiently compute d, given n and e. Hint: since $ed \equiv 1 \pmod{\varphi(n)}$, it follows that $ed = 1 + \varphi(n)k$ for an integer k with $0 < k < d$; let $r := nk - ed$, and show that $|r| < n^{3/4}$; next, show how to recover d (along with r and k) using Theorem 4.9.

4.8 Notes

The Euclidean algorithm as we have presented it here is not the fastest known algorithm for computing greatest common divisors. The asymptotically fastest known algorithm for computing the greatest common divisor of two numbers of bit length at most ℓ runs in time $O(\ell\,\mathrm{len}(\ell))$ on a RAM, which is due to Schönhage [85]. The same algorithm leads to Boolean circuits of size $O(\ell\,\mathrm{len}(\ell)^2\,\mathrm{len}(\mathrm{len}(\ell)))$, which using Fürer's result [38], can be reduced to $O(\ell\,\mathrm{len}(\ell)^2\,2^{O(\log^* n)})$. The same complexity results also hold for the extended Euclidean algorithm, as well as for Chinese remaindering, Thue's lemma, and rational reconstruction.

Experience suggests that such fast algorithms for greatest common divisors are not of much practical value, unless the integers involved are *very* large — at least

several tens of thousands of bits in length. The extra "log" factor and the rather large multiplicative constants seem to slow things down too much.

The binary gcd algorithm (Exercise 4.6) is due to Stein [100]. The extended binary gcd algorithm (Exercise 4.10) was first described by Knuth [56], who attributes it to M. Penk. Our formulation of both of these algorithms closely follows that of Menezes, van Oorschot, and Vanstone [66]. Experience suggests that the binary gcd algorithm is faster in practice than Euclid's algorithm.

Schoof [87] presents (among other things) a deterministic, polynomial-time algorithm that computes a square root of -1 modulo p for any given prime $p \equiv 1 \pmod 4$. If we use this algorithm in §4.5, we get a deterministic, polynomial-time algorithm to compute integers r and t such that $p = r^2 + t^2$.

Our Theorem 4.9 is a generalization of one stated in Wang, Guy, and Davenport [103]. One can generalize Theorem 4.9 using the theory of **continued fractions**. With this, one can generalize Exercise 4.18 to deal with rational approximations to irrational numbers. More on this can be found, for example, in the book by Hardy and Wright [46].

The application of Euclid's algorithm to computing a rational number from the first digits of its decimal expansion was observed by Blum, Blum, and Shub [17], where they considered the possibility of using such sequences of digits as a pseudo-random number generator — the conclusion, of course, is that this is not such a good idea.

The RSA cryptosystem was invented by Rivest, Shamir, and Adleman [82]. There is a vast literature on cryptography. One starting point is the book by Menezes, van Oorschot, and Vanstone [66]. The attack in Exercise 4.27 is due to Wiener [110]; this attack was recently strengthened by Boneh and Durfee [19].