4
Euclid's algorithm

In this chapter, we discuss Euclid's algorithm for computing greatest common divisors. It turns out that Euclid's algorithm has a number of very nice properties, and has applications far beyond that purpose.

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 of Euclid's algorithm 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, if $b > 0$, 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 $r_0, r_1, \ldots, r_{\ell+1}$, and $q_1, \ldots, q_\ell$, where $\ell \geq 0$, as follows:*

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

Note that by definition, $\ell = 0$ if $b = 0$, and $\ell > 0$, otherwise.

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

Proof. For the first statement, one sees that for $i = 1, \ldots, \ell$, 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

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

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

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

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

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

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

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, taking as input integers a, b, such that $a \geq b \geq 0$, and producing as output $d = \gcd(a, b)$:

```
$r \leftarrow a,\ r' \leftarrow b$
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 k-bit numbers. The algorithm performs $O(k)$ divisions on numbers with at most k-bits. As each such division takes time $O(k^2)$, this leads to a bound on the running time of $O(k^3)$. However, as the following theorem shows, this cubic running time bound is well off the mark.

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

Proof. We may assume that $b > 0$. The running time is $O(\tau)$, where $\tau := \sum_{i=1}^{\ell} \mathrm{len}(r_i)\,\mathrm{len}(q_i)$. Since $r_i \leq b$ for $i = 1, \ldots, \ell$, we have

$$\tau \leq \mathrm{len}(b) \sum_{i=1}^{\ell} \mathrm{len}(q_i) \leq \mathrm{len}(b) \sum_{i=1}^{\ell} (\log_2 q_i + 1) = \mathrm{len}(b)(\ell + \log_2(\prod_{i=1}^{\ell} q_i)).$$

Note that

$$a = r_0 \geq r_1 q_1 \geq r_2 q_2 q_1 \geq \cdots \geq r_\ell q_\ell \cdots q_1 \geq q_\ell \cdots q_1.$$

We also have $\ell \leq \log b / \log \phi + 1$. Combining this with the above, we have

$$\tau \leq \mathrm{len}(b)(\log b/\log \phi + 1 + \log_2 a) = O(\mathrm{len}(a)\,\mathrm{len}(b)),$$

which proves the theorem. □

Exercise 4.1. 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 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 $\ell := \max(\text{len}(a), \text{len}(b))$.

4.2 The extended Euclidean algorithm

Let a and b be non-negative integers, and let $d := \gcd(a, b)$. We know by Theorem 1.6 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 following theorem describes the algorithm, and also states a number of important facts about the relative sizes of the numbers that arise during the computation—these size estimates 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, $r_0, r_1, \ldots, r_{\ell+1}$ and $q_1, \ldots, q_\ell$ be as in Theorem 4.1. Define integers $s_0, s_1, \ldots, s_{\ell+1}$ and $t_0, t_1, \ldots, t_{\ell+1}$ as follows:*

$$s_0 := 1, \quad t_0 := 0,$$
$$s_1 := 0, \quad t_1 := 1,$$

and for $i = 1, \ldots, \ell$,

$$s_{i+1} := s_{i-1} - s_i q_i, \quad t_{i+1} := t_{i-1} - t_i q_i.$$

Then

(i) for $i = 0, \ldots, \ell+1$, we have $s_i a + t_i b = r_i$; in particular, $s_\ell a + t_\ell b = \gcd(a, b)$;

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

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

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

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

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

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

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

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

(iii) follows directly from (ii).

For (iv), one can easily prove both statements by induction on i. The statement involving the t_i is clearly true for $i = 0$; for $i = 1, \ldots, \ell$, we have $t_{i+1} = t_{i-1} - t_i q_i$, and since 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 is the same, except that we start the induction at $i = 1$.

For (v), one considers the two equations:

$$\begin{aligned} s_{i-1}a + t_{i-1}b &= r_{i-1}, \\ s_i a + t_i b &= r_i. \end{aligned}$$

Subtracting t_{i-1} times the second equation from t_i times the first, applying (ii), and using the fact that t_i and t_{i-1} have opposite sign, we obtain

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

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

Suppose that $a > 0$ in the above theorem. Then for $i = 1, \ldots, \ell + 1$, the value r_{i-1} is a positive integer, and so part (v) of the theorem implies that $|t_i| \leq a/r_{i-1} \leq a$ and $|s_i| \leq b/r_{i-1} \leq b$. Moreover, if $a > 1$ and $b > 0$, then $\ell > 0$ and $r_{\ell-1} \geq 2$, and hence $|t_\ell| \leq a/2$ and $|s_\ell| \leq b/2$.

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

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, taking as input integers a, b, such that $a \geq b \geq 0$, and producing as output integers d, s, and t, such that $d = \gcd(a, b)$ and $as + bt = d$:

```
$r \leftarrow a,\ r' \leftarrow b$
$s \leftarrow 1,\ s' \leftarrow 0$
$t \leftarrow 0,\ t' \leftarrow 1$
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*

$$O(\operatorname{len}(a) \operatorname{len}(b)).$$

Proof. We may assume that $b > 0$. It suffices to analyze the cost of computing the sequences $\{s_i\}$ and $\{t_i\}$. Consider first the cost of computing all of the t_i, which is $O(\tau)$, where $\tau := \sum_{i=1}^{\ell} \operatorname{len}(t_i) \operatorname{len}(q_i)$. We have $t_1 = 1$ and, by part (v) of Theorem 4.3, we have $|t_i| \leq a$ for $i = 2, \ldots, \ell$. Arguing as in the proof of Theorem 4.2, we have

$$\begin{aligned}\tau &\leq \operatorname{len}(q_1) + \operatorname{len}(a) \sum_{i=2}^{\ell} \operatorname{len}(q_i) \leq \operatorname{len}(q_1) + \operatorname{len}(a)(\ell - 1 + \log_2(\prod_{i=2}^{\ell} q_i)) \\ &= O(\operatorname{len}(a) \operatorname{len}(b)),\end{aligned}$$

where we have used the fact that $\prod_{i=2}^{\ell} q_i \leq b$. An analogous argument shows that one can also compute all of the s_i in time $O(\operatorname{len}(a) \operatorname{len}(b))$, and in fact, in time $O(\operatorname{len}(b)^2)$. □

Another, instructive way to view Theorem 4.3 is as follows. For $i = 1, \ldots, \ell$, 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 for $i = 0, \ldots, \ell$,

$$\begin{pmatrix} r_i \\ r_{i+1} \end{pmatrix} = M_i \begin{pmatrix} a \\ b \end{pmatrix},$$

where for $i = 1, \ldots, \ell$, the matrix M_i is defined as

$$M_i := \begin{pmatrix} 0 & 1 \\ 1 & -q_i \end{pmatrix} \cdots \begin{pmatrix} 0 & 1 \\ 1 & -q_1 \end{pmatrix}.$$

If we define M_0 to be the 2×2 identity matrix, then it is easy to see that

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

for $i = 0, \ldots, \ell$. From this observation, 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 4.2. One can extend the binary gcd algorithm discussed in Exercise 4.1 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 runs in time $O(\ell^2)$, where $\ell := \max(\text{len}(a), \text{len}(b))$. 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 $O(\ell)$.

4.3 Computing modular inverses and Chinese remaindering

One application of the extended Euclidean algorithm is to the problem of computing multiplicative inverses in $\mathbb{Z}_n$, where $n > 1$.

Given $y \in \{0, \dots, n-1\}$, in time $O(\mathrm{len}(n)^2)$, we can determine if y is relatively prime to n, and if so, compute $y^{-1} \bmod n$, as follows. We run the extended Euclidean algorithm on inputs $a := n$ and $b := y$, obtaining integers d, s, and t, such that $d = \gcd(n, y)$ and $ns + yt = d$. If $d \neq 1$, then y does not have a multiplicative inverse modulo n. Otherwise, if $d = 1$, then t is a multiplicative inverse of y modulo n; however, it may not lie in the range $\{0, \dots, n-1\}$, as required. Based on Theorem 4.3 (and the discussion immediately following it), we know that $|t| \leq n/2 < n$; therefore, either $t \in \{0, \dots, n-1\}$, or $t < 0$ and $t + n \in \{0, \dots, n-1\}$. Thus, $y^{-1} \bmod n$ is equal to either t or $t + n$.

We also observe that the Chinese remainder theorem (Theorem 2.8) can be made computationally effective:

Theorem 4.5. *Given integers $n_1, \dots, n_k$ and $a_1, \dots, a_k$, where $n_1, \dots, n_k$ are pairwise relatively prime, and where $n_i > 1$ and $0 \leq a_i < n_i$ for $i = 1, \dots, k$, we can compute the integer z, such that $0 \leq z < n$ and $z \equiv a_i \pmod{n_i}$ for $i = 1, \dots, k$, where $n := \prod_i n_i$, in time $O(\mathrm{len}(n)^2)$.*

Proof. Exercise (just use the formulas in the proof of Theorem 2.8, and see Exercises 3.22 and 3.23). □

Exercise 4.3. In this exercise and the next, you are to analyze an "incremental Chinese remaindering algorithm." Consider the following algorithm, which takes as input integers z, n, z', n', such that

$$n' > 1, \quad \gcd(n, n') = 1, \quad 0 \leq z < n, \quad \text{and} \quad 0 \leq z' < n'.$$

It outputs integers z'', n'', such that

$$n'' = nn', \quad 0 \leq z'' < n'', \quad z'' \equiv z \pmod{n}, \quad \text{and} \quad z'' \equiv z' \pmod{n'}.$$

It runs as follows:

1. Set $\tilde{n} \leftarrow n^{-1} \bmod n'$.
2. Set $h \leftarrow ((z' - z)\tilde{n}) \bmod n'$.

3. Set $z'' \leftarrow z + nh$.
4. Set $n'' \leftarrow nn'$.
5. Output z'', n''.

Show that the output z'', n'' of the algorithm satisfies the conditions stated above, and estimate the running time of the algorithm.

EXERCISE 4.4. Using the algorithm in the previous exercise as a subroutine, give a simple $O(\text{len}(n)^2)$ algorithm that takes as input integers $n_1, \ldots, n_k$ and $a_1, \ldots, a_k$, where $n_1, \ldots, n_k$ are pairwise relatively prime, and where $n_i > 1$ and $0 \leq a_i < n_i$ for $i = 1, \ldots, k$, and outputs integers z and n such that $0 \leq z < n$, $n = \prod_i n_i$, and $z \equiv a_i \pmod{n_i}$ for $i = 1, \ldots, k$. The algorithm should be "incremental," in that it processes the pairs (n_i, a_i) one at a time, using time $O(\text{len}(n)\,\text{len}(n_i))$ to process each such pair.

EXERCISE 4.5. Suppose you 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 less 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 4.5) 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 $C := AB$. If the entries of A are (a_{rs}) and the entries of B are (b_{st}), then the entries (c_{rt}) of C are given by the usual rule for matrix multiplication:

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

Suppose further that H is the maximum absolute value of the entries in A and B, so that the entries in C are bounded in absolute value by $H' := H^2 m$. Then by just applying the above formula, we can compute the entries of C using m^3 multiplications of numbers of length at most $\text{len}(H)$, and m^3 additions of numbers of length at most $\text{len}(H')$, where

$\mathrm{len}(H') \le 2\,\mathrm{len}(H) + \mathrm{len}(m)$. This yields a running time of

$$O(m^3\,\mathrm{len}(H)^2 + m^3\,\mathrm{len}(m)). \tag{4.1}$$

If the entries of A and B are large relative to m, specifically, if $\mathrm{len}(m) = O(\mathrm{len}(H)^2)$, then the running time is dominated by the first term above, namely

$$O(m^3\,\mathrm{len}(H)^2).$$

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

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

$$c_{rt} \equiv \sum_{s=1}^{m} a_{rs}b_{st} \pmod{n}. \tag{4.2}$$

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

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

and if we also have

$$-n/2 \le c'_{rt} < n/2 \quad \text{and} \quad n > 2H', \tag{4.4}$$

then we must have

$$c_{rt} = c'_{rt}. \tag{4.5}$$

To see why (4.5) follows from (4.3) and (4.4), observe that (4.2) and (4.3) imply that $c_{rt} \equiv c'_{rt} \pmod{n}$, which means that n divides $(c_{rt} - c'_{rt})$. Then from the bound $|c_{rt}| \le H'$ and from (4.4), we obtain

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

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 (4.5).

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 $n_1, \ldots, n_k$, relatively prime in pairs, and such that the product $n := n_1 \cdots n_k$ is just a bit larger than $2H'$. In practice, one would choose the n_i 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 such 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 any of the n_i can be done in *constant* time. This is reasonable, both from a practical and theoretical point of view, since such primes easily "fit" into a memory cell. Finally, we assume that we do not use more of the numbers n_i than are necessary, so that $\text{len}(n) = O(\text{len}(H'))$ and $k = O(\text{len}(H'))$.

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 $(c_{rt} : r, t = 1, \ldots, m)$.

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

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 (see Exercise 3.23) to be $O(m^2 \,\text{len}(H')^2)$. Under our assumption about the cost of arithmetic modulo small primes, the cost of Step 1c is $O(m^3 k)$, and since $k = O(\text{len}(H')) = O(\text{len}(H) + \text{len}(m))$, the cost of this step is $O(m^3(\text{len}(H) + \text{len}(m)))$. Finally, by Theorem 4.5, the cost of Step 2 is $O(m^2 \,\text{len}(H')^2)$. Thus, the total running time of this algorithm is easily calculated (discarding terms that are dominated by others) as

$$O(m^2 \,\text{len}(H)^2 + m^3 \,\text{len}(H) + m^3 \,\text{len}(m)).$$

Compared to (4.1), we have essentially replaced the term $m^3 \,\text{len}(H)^2$ by $m^2 \,\text{len}(H)^2 + m^3 \,\text{len}(H)$. This is a significant improvement: for example, if $\operatorname{len}(H) \approx m$, 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 4.6. 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 Rational reconstruction and applications

We next state a theorem whose immediate utility may not be entirely obvious, but we quickly follow up with several very neat applications. The general problem we consider here, called **rational reconstruction**, is as follows. Suppose that there is some rational number $\hat{y}$ that we would like to get our hands on, but the only information we have about $\hat{y}$ is the following:

- First, suppose that we know that $\hat{y}$ may be expressed as r/t for integers r, t, with $|r| \leq r^*$ and $|t| \leq t^*$—we do not know r or t, but we do know the bounds r^* and t^*.
- Second, suppose that we know integers y and n such that n is relatively prime to t, and $y = rt^{-1} \bmod n$.

It turns out that if n is sufficiently large relative to the bounds r^* and t^*, then we can virtually "pluck" $\hat{y}$ out of the extended Euclidean algorithm applied to n and y. Moreover, the restriction that n is relatively prime to t is not really necessary; if we drop this restriction, then our assumption is that $r \equiv ty \pmod{n}$, or equivalently, $r = sn + ty$ for some integer s.

Theorem 4.6. *Let r^*, t^*, n, y be integers such that $r^* > 0$, $t^* > 0$, $n \geq 4r^*t^*$, and $0 \leq y < n$. Suppose we run the extended Euclidean algorithm with inputs $a := n$ and $b := y$. Then, adopting the notation of Theorem 4.3, the following hold:*

(i) There exists a unique index $i = 1, \ldots, \ell+1$ such that $r_i \leq 2r^* < r_{i-1}$; note that $t_i \neq 0$ for this i.

Let $r' := r_i$, $s' := s_i$, and $t' := t_i$.

(ii) Furthermore, for any integers r, s, t such that

$$r = sn + ty, \quad |r| \leq r^*, \quad \text{and} \quad 0 < |t| \leq t^*, \tag{4.6}$$

we have

$$r = r'\alpha, \; s = s'\alpha, \quad \text{and} \quad t = t'\alpha,$$

for some non-zero integer α.

Proof. By hypothesis, $2r^* < n = r_0$. Moreover, since $r_0, \ldots, r_\ell, r_{\ell+1} = 0$ is a decreasing sequence, and $1 = |t_1|, |t_2|, \ldots, |t_{\ell+1}|$ is a non-decreasing sequence, the first statement of the theorem is clear.

Now let i be defined as in the first statement of the theorem. Also, let r, s, t be as in (4.6).

From part (v) of Theorem 4.3 and the inequality $2r^* < r_{i-1}$, we have

$$|t_i| \le \frac{n}{r_{i-1}} < \frac{n}{2r^*}.$$

From the equalities $r_i = s_i n + t_i y$ and $r = sn + ty$, we have the two congruences:

$$r \equiv ty \pmod{n},$$
$$r_i \equiv t_i y \pmod{n}.$$

Subtracting t_i times the first from t times the second, we obtain

$$rt_i \equiv r_i t \pmod{n}.$$

This says that n divides $rt_i - r_i t$. Using the bounds $|r| \le r^*$ and $|t_i| < n/(2r^*)$, we see that $|rt_i| < n/2$, and using the bounds $|r_i| \le 2r^*$, $|t| \le t^*$, and $4r^*t^* \le n$, we see that $|r_i t| \le n/2$. It follows that

$$|rt_i - r_i t| \le |rt_i| + |r_i t| < n/2 + n/2 = n.$$

Since n divides $rt_i - r_i t$ and $|rt_i - r_i t| < n$, the only possibility is that

$$rt_i - r_i t = 0. \tag{4.7}$$

Now consider the two equations:

$$r = sn + ty$$
$$r_i = s_i n + t_i y.$$

Subtracting t_i times the first from t times the second, and using the identity (4.7), we obtain $n(st_i - s_i t) = 0$, and hence

$$st_i - s_i t = 0. \tag{4.8}$$

From (4.8), we see that $t_i \mid s_i t$, and since from part (iii) of Theorem 4.3, we know that $\gcd(s_i, t_i) = 1$, we must have $t_i \mid t$. So $t = t_i\alpha$ for some α, and we must have $\alpha \ne 0$ since $t \ne 0$. Substituting $t_i\alpha$ for t in equations (4.7) and (4.8) yields $r = r_i\alpha$ and $s = s_i\alpha$. That proves the second statement of the theorem. □

4.5.1 Application: Chinese remaindering with errors

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

But now suppose that Alice encodes z as $(a_1, \ldots, a_k)$, and sends this encoding to Bob; however, 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 z from its corrupted encoding?

To make the problem more precise, suppose that the original, correct encoding of z is $(a_1, \ldots, a_k)$, and the corrupted encoding is $(\tilde{a}_1, \ldots, \tilde{a}_k)$. Let us define $G \subseteq \{1, \ldots, k\}$ to be the set of "good" positions i with $\tilde{a}_i = a_i$, and $B \subseteq \{1, \ldots, k\}$ to be the set of "bad" positions i with $\tilde{a}_i \neq a_i$. We shall assume that $|B| \le \ell$, where ℓ is some specified parameter.

Of course, if Bob hopes to recover z, we need to build some redundancy into the system; that is, we must require that $0 \le z \le Z$ for some Z that is somewhat smaller than n. Now, if Bob knew the location of bad positions, and if the product of the integers n_i at the good positions exceeds Z, then Bob could simply discard the errors, and reconstruct z by applying the Chinese remainder theorem to the values a_i and n_i at the good positions. However, in general, Bob will not know a priori the location of the bad positions, and so this approach will not work.

Despite these apparent difficulties, Theorem 4.6 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 n_i). Further, let us assume that $n \ge 4P^2Z$.

Now, suppose Bob obtains the corrupted encoding $(\tilde{a}_1, \ldots, \tilde{a}_k)$. Here is what Bob does to recover z:

1. Apply the Chinese remainder theorem, obtaining an integer y, with $0 \le y < n$ and $y \equiv \tilde{a}_i \pmod{n_i}$ for $i = 1, \ldots, k$.
2. Run the extended Euclidean algorithm on $a := n$ and $b := y$, and let r', t' be the values obtained from Theorem 4.6 applied with $r^* := ZP$ and $t^* := P$.
3. If $t' \mid r'$, output r'/t'; otherwise, output "error."

We claim that the above procedure outputs z, 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 $1 \le t \le P$. Also, let $r := tz$, and note that

$0 \le r \le r^*$ and $0 < t \le t^*$. We claim that

$$r \equiv ty \pmod{n}. \tag{4.9}$$

To show that (4.9) holds, it suffices to show that

$$tz \equiv ty \pmod{n_i} \tag{4.10}$$

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 $a_i = \tilde{a}_i$, and therefore,

$$tz \equiv ta_i \equiv t\tilde{a}_i \equiv ty \pmod{n_i}.$$

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

$$tz \equiv 0 \equiv ty \pmod{n_i}.$$

Thus, (4.10) holds for all $i = 1, \ldots, k$, and so it follows that (4.9) holds. Therefore, the values r', t' obtained from Theorem 4.6 satisfy

$$\frac{r'}{t'} = \frac{r}{t} = \frac{tz}{t} = z.$$

One easily checks that both the procedures to encode and decode a value z run in time $O(\mathrm{len}(n)^2)$. If one wanted a practical implementation, one might choose $n_1, \ldots, n_k$ to be, say, 16-bit primes, so that the encoding of a value z consisted of a sequence of k 16-bit words.

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

4.5.2 Application: recovering fractions from their decimal expansions

Suppose Alice knows a rational number $z := s/t$, where s and t are integers with $0 \le s < t$, and tells Bob some of the high-order digits in the decimal expansion of z. Can Bob determine z? The answer is yes, provided Bob knows an upper bound T on t, and provided Alice gives Bob enough digits. Of course, from grade school, Bob probably remembers that the decimal expansion of z is ultimately periodic, and that given enough digits of z so as to include the periodic part, he can recover z; however, this technique is quite useless in practice, as the length of the period can be huge—$\Theta(T)$ in the worst case (see Exercises 4.8–4.10 below). The method we discuss here requires only $O(\mathrm{len}(T))$ digits.

To be a bit more general, suppose that Alice gives Bob the high-order k

digits in the d-ary expansion of z, for some base $d > 1$. Now, we can express z in base d as

$$z = z_1 d^{-1} + z_2 d^{-2} + z_3 d^{-3} + \cdots ,$$

and the sequence of digits $z_1, z_2, z_3, \ldots$ is uniquely determined if we require that the sequence does not terminate with an infinite run of $(d-1)$-digits. Suppose Alice gives Bob the first k digits $z_1, \ldots, z_k$. Define

$$y := z_1 d^{k-1} + \cdots + z_{k-1} d + z_k = \lfloor z d^k \rfloor .$$

Let us also define $n := d^k$, so that $y = \lfloor zn \rfloor$.

Now, if n is much smaller than T^2, the number z is not even uniquely determined by y, since there are $\Omega(T^2)$ distinct rational numbers of the form s/t, with $0 \leq s < t \leq T$ (see Exercise 1.18). However, if $n \geq 4T^2$, then not only is z uniquely determined by y, but using Theorem 4.6, we can compute it as follows:

1. Run the extended Euclidean algorithm on inputs $a := n$ and $b := y$, and let s', t' be as in Theorem 4.6, using $r^* := t^* := T$.
2. Output s', t'.

We claim that $z = -s'/t'$. To prove this, observe that since $y = \lfloor zn \rfloor = \lfloor (ns)/t \rfloor$, if we set $r := (ns) \bmod t$, then we have

$$r = sn - ty \quad \text{and} \quad 0 \leq r < t \leq t^* .$$

It follows that the integers s', t' from Theorem 4.6 satisfy $s = s'\alpha$ and $-t = t'\alpha$ for some non-zero integer α. Thus, $s'/t' = -s/t$, which proves the claim.

We may further observe that 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.

It is clear that the running time of this algorithm is $O(\text{len}(n)^2)$.

***Example* 4.3.** Alice chooses numbers $0 \leq s < t \leq 1000$, and tells Bob the high-order seven digits y in the decimal expansion of $z := s/t$, from which Bob should be able to compute z. Suppose $s = 511$ and $t = 710$. Then $s/t \approx 0.71971830985915492958$, and so $y = 7197183$ and $n = 10^7$. Running the extended Euclidean algorithm on inputs $a := n$ and $b := y$, Bob obtains the following data:

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

The first r_i that meets or falls below the threshold $2r^* = 2000$ is at $i = 10$, and Bob reads off $s' = 511$ and $t' = -710$, from which he obtains $z = -s'/t' = 511/710$. $\square$

EXERCISE 4.7. 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 $O(\text{len}(k)\,\text{len}(t)^2)$.

For the following exercises, we need a definition: a sequence $S := (z_1, z_2, z_3, \ldots)$ of elements drawn from some arbitrary set is called **(k, ℓ)-periodic** for integers $k \geq 0$ and $\ell \geq 1$ if $z_i = z_{i+\ell}$ for all $i > k$. S is called **ultimately periodic** if it is (k, ℓ)-periodic for some (k, ℓ).

EXERCISE 4.8. Show that if a sequence S is ultimately periodic, then it is (k^*, ℓ^*)-periodic for some uniquely determined pair (k^*, ℓ^*) for which the following holds: for any pair (k, ℓ) such that S is (k, ℓ)-periodic, we have $k^* \leq k$ and $\ell^* \leq \ell$.

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

EXERCISE 4.9. Let z be a real number whose base-d expansion is an ultimately periodic sequence. Show that z is rational.

EXERCISE 4.10. Let $z = s/t \in \mathbb{Q}$, where s and t are relatively prime integers with $0 \leq s < t$, and let $d > 1$ be an integer.

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

(b) Show that for integers k, k' with $0 \leq k < k'$, the base-d expansion of z is $(k, k' - k)$-periodic if and only if $sd^k \equiv sd^{k'} \pmod{t}$.

(c) Show that if $\gcd(t, d) = 1$, then the base-d expansion of z is purely periodic with period equal to the multiplicative order of d modulo t.

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

A famous conjecture of Artin postulates that for any 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, for any $d > 1$ that is not a square, there are infinitely many primes t such that the base-d expansion of s/t, for any $0 < s < t$, 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.

4.5.3 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 15.13.

4.6 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 $O(\ell \operatorname{len}(\ell))$ on a RAM, and the smallest Boolean circuits are of size $O(\ell \operatorname{len}(\ell)^2 \operatorname{len}(\operatorname{len}(\ell)))$. This algorithm is due to Schönhage [81]. The same complexity results also hold for the extended Euclidean algorithm, as well as Chinese remaindering 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 4.1) is due to Stein [95]. The extended binary gcd algorithm (Exercise 4.2) was first described by Knuth [54], who attributes it to M. Penk. Our formulation of both of these algorithms closely follows that of Menezes, van Oorschot, and Vanstone [62]. Experience suggests that the binary gcd algorithm is faster in practice than Euclid's algorithm.

Our exposition of Theorem 4.6 is loosely based on Bach [11]. A somewhat "tighter" result is proved, with significantly more effort, by Wang, Guy, and Davenport [97]. However, for most practical purposes, the result proved here is just as good. 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.