Chapter 6

Arithmetic in basic domains

References in this chapter refer to references in [Win96].

6.1 Integers

For the purposes of exact algebraic computation integers have to be represented exactly. In practice, of course the size of the machine memory bounds the integers that can be represented. But it is certainly not acceptable to be limited by the word length of the machine, say 2^{32}.

Definition 6.1.1. Let $\beta \geq 2$ be a natural number. A *β–digit* is an integer b in the range $-\beta < b < \beta$. Every positive integer a can be written uniquely as $a = \sum_{i=0}^{n-1} a_i \beta^i$ for some $n \in \mathbb{N}$, $a_0, \ldots, a_{n-1}$ nonnegative β–digits and $a_{n-1} > 0$. In the *positional number system* with *basis* (or *radix*) β the number a is represented by the uniquely determined list $a_{(\beta)} = [+, a_0, \ldots, a_{n-1}]$. In an analogous way a negative integer a is represented by $a_{(\beta)} = [-, a_0, \ldots, a_{n-1}]$, where $a = \sum_{i=0}^{n-1} (-a_i)\beta^i$, $n \in \mathbb{N}$, $a_0, \ldots, a_{n-1}$ nonnegative β–digits and $a_{n-1} > 0$. The number 0 is represented by the empty list [ ].
If the integer a is represented by the list $[\pm, a_0, \ldots, a_{n-1}]$, then $L_\beta(a) := n$ is the *length of a w.r.t. β*. $L_\beta(0) := 0$. □

So, for example, in the positional number system with basis 1000, the integer 2000014720401 is represented by the list $[+, 401, 1472, 0, 2]$. By representing integers as lists, we can exactly represent integers of arbitrary size. The memory needed can be allocated dynamically, "as the need arises". For all practical purposes we will chose β such that a β–digit can be stored in a single machine word.

In measuring the complexity of arithmetic operations on integers we will often give the complexity functions as functions of L_β, the length of the integer arguments of the operations w.r.t. to the basis β of the number system. It is crucial to note that for two different radices β and γ the associated length functions are proportional. So we will often speak of the length of an integer without referring to a specific radix. You will prove the following lemma in the exercises.

Lemma 6.1.1. Let β and γ be two radices for positional number systems in $\mathbb{Z}$.
(a) $L_\beta(a) = \lfloor \log_\beta(\mid a \mid) \rfloor + 1$ for $a \neq 0$.
(b) $L_\beta \sim L_\gamma$.

For future reference we note that obviously the algorithm INT_SIGN for computing the sign ± 1 or 0 of an integer a takes constant time. So signs do not present a problem in integer arithmetic, and we will omit their computation in subsequent algorithms.

In fact, we could omit the sign bit in the representation of integers, and instead use non-positive digits for representing negative numbers. This, of course, means that we need a new algorithm for determining the sign of an integer. Such an algorithm is developed in the exercises and in fact one can prove that this determination of the sign, although in the worst case proportional to the length of the integer, in average is only proportional to a constant.

Now we describe the arithmetic operations in $\mathbb{Z}$. Throughout this section we assume that the inputs to algorithms are given in the positional number system with radix β, for some fixed β, and also the outputs are to be computed in this number system. Furthermore, in the complexity analyses we will refer to the partition $\{\mathbb{Z}_n\}_{n\in\mathbb{N}}$ of $\mathbb{Z}$, where $\mathbb{Z}_n = \{a \in \mathbb{Z} \mid L_\beta(a) = n\}$.

Addition

The "classical" addition algorithm INT_SUMC considers both inputs a and b as numbers of equal length and adds corresponding digits with carry until both inputs are exhausted. So $t_{\text{INT\_SUMC}}(m,n) \sim \max(m,n)$, where m and n are the lengths of the inputs. A closer analysis, however, reveals that after the shorter input is exhausted the carry has to be propagated only as long as the corresponding digits of the longer input are $\beta - 1$ or $-\beta + 1$, respectively. This fact is used for constructing a more efficient algorithm, whose computing time depends only linearly on the length of the shorter of the two summands. We assume that an algorithm DIGIT_SUM is available, which adds the contents of two machine words, i.e. two β–digits a, b, yielding a digit c and a digit $d \in \{-1, 0, 1\}$, such that $(a+b)_{(\beta)} = [c, d]$. Obviously the complexity function of DIGIT_SUM is constant.

First let us assume that the two integers to be added have the same sign.

---

algorithm INT_SUM1(**in:** a, b; **out:** c);
[a, b are integers, sign(a) = sign(b); $c = a + b$]
(1) [initialization] $c :=$ [ ]; $e := 0$; $a' := a$; $b' := b$;
(2) [while a' and b' are not exhausted, add successive digits of a' and b']
while $a' \neq$ [ ] and $b' \neq$ [ ] **do**
{$d_1 :=$ FIRST(a'); $a' :=$ REST(a'); $d_2 :=$ FIRST(b'); $b' :=$ REST(b');
$(d, f) :=$ DIGIT_SUM(d_1, d_2);
if $f \neq 0$
then $(d, f') :=$ DIGIT_SUM(d, e)
else $(d, f) :=$ DIGIT_SUM(d, e); $c :=$ CONS(d, c); $e := f$};
(3) [the carry is propagated, until it disappears or both numbers are exhausted]
if $a' =$ [ ] **then** $g := b'$ **else** $g := a'$;
while $e \neq 0$ and $g \neq$ [ ] **do**
{$d_1 :=$ FIRST(g); $g :=$ REST(g);
$(d, e) :=$ DIGIT_SUM(d_1, e); $c :=$ CONS(d, c)};
(4) **if** $e = 0$
then {$c :=$ INV(c); $c :=$ APPEND(c, g)}
else {$c :=$ CONS(e, c); $c :=$ INV(c)};
return □

Lemma 6.1.2. For $a, b \in \mathbb{Z}$ with $\text{sign}(a) = \text{sign}(b)$, INT_SUM1 correctly computes $a + b$.

Proof: We only consider the case $a, b > 0$. For $a = b = 0$ the correctness is obvious, and for $a, b < 0$ the correctness can be proved analogously. Let $m = L_\beta(a)$, $n = L_\beta(b)$, w.l.o.g. $m \leq n$. After the **while** loop in step (2) has been executed i times, $0 \leq i \leq m$, we have

$$a + b = \text{INV}(c) + \beta^i \cdot e + \beta^i \cdot (a' + b').$$

So when step (2) is finished,

$$a + b = \text{INV}(c) + \beta^m \cdot e + \beta^m \cdot b'.$$

After the **while** loop in step (3) has been executed j times, $0 \leq j \leq n - m$, we have

$$a + b = \text{INV}(c) + \beta^{m+j} \cdot e + \beta^{m+j} \cdot b'.$$

When the carry e becomes 0 we only need to combine $\text{INV}(c)$ and b'. Otherwise $b' = [\,]$ after $n - m$ iterations and we only need to add e as the highest digit to the result. □

Theorem 6.1.3. The maximum, minimum, and average complexity functions of INT_SUM1 are proportional to $\max(m, n), \min(m, n)$, and $\min(m, n)$, respectively, where m and n are the β–lengths of the inputs.

Proof: The maximum and minimum complexity functions for INT_SUM1 are obvious. So let us consider the average complexity.

Let a, b be the inputs of INT_SUM1, and let $m = L(a), n = L(b)$. Obviously the complexity of steps (1), (2) is proportional to $\min(m, n)$. We will show that the average complexity function of step (3) is constant. This will imply that the length of c in step (4) will be proportional to $\min(m, n)$, and therefore also the average complexity of this step is proportional to $\min(m, n)$.

W.l.o.g. assume that the inputs are positive and that $m < n$. Let $k = n - m$. If β is the radix of the number system, then there are $(\beta - 1)\beta^{k-1}$ possible assignments for the k highest digits of b. The carry has to be propagated exactly up to position $m + i$ of b for $i < k$, if the digits in positions $m + 1, \ldots, m + i - 1$ of b all are $\beta - 1$ and the digit in position $m + i$ is less than $\beta - 1$. So there are $(\beta - 1)^2\beta^{k-i-1}$ possible assignments of the digits in positions $m + 1, \ldots, m + k$ of b, for which exactly i iterations through step (3) are required. There are $\beta - 2$ assignments for which the carry is propagated exactly up to position $m + k$, and for one assignment the propagation is up to position $m + k + 1$. Summation over the total time for all these assignments yields

$$\sum_{i=1}^{k-1} i \cdot (\beta - 1)^2\beta^{k-i-1} + k \cdot (\beta - 2) + (k + 1) \cdot 1 = (\beta^k - \beta k + k - 1) + \beta k - k + 1 = \beta^k.$$

So the average complexity for step (3) is

$$\frac{\beta^k}{(\beta - 1)\beta^{k-1}} = \frac{\beta}{\beta - 1} \leq 2,$$

i.e. it is contant. □

By similar considerations one can develop an algorithm INT_SUM2 for adding two nonzero integers with opposite signs in maximum, minumum, and average time proportional to the maximum, minimum, and minimum of the lengths of the inputs, respectively (see Exercises). The combination of these two algorithms leads to an addition algorithm INT_SUM for adding two arbitrary integers. This proves the following theorem.

Theorem 6.1.4. There is an addition algorithm INT_SUM for integers with maximum, minimum, and average complexity functions proportional to $\max(m,n), \min(m,n)$, and $\min(m,n)$, respectively, where m and n are the β–lengths of the inputs.

The algorithm INT_NEG for computing the additive inverse $-a$ of an integer a is obviously of constant complexity. The difference $a-b$ of two integers a and b can be computed as INT_SUM$(a,$INT_NEG$(b))$. So the algorithm INT_DIFF for computing the difference of two integers has the same complexity behaviour as INT_SUM. The algorithm INT_ABS for computing the absolute value of an integer is either of constant complexity or proportional to the length of the input, depending on which representation of integers we use.

Multiplication

Now we approach the question of how fast we can multiply two integers. Here we can give only a first answer. We will come back to this question later. The "classical" multiplication algorithm INT_MULTC proceeds by multiplying every digit of the first input by every digit of the second input and adding the results after appropriate shifts. The complexity of INT_MULTC is proportional to the product of the lengths of the two inputs, and if the inputs are of the same length n, then the complexity of INT_MULTC is proportional to n^2.

A faster multiplication algorithm has been discovered in 1962 by A. Karatsuba and Yu. Ofman [Karatsuba,Ofman 62]. The basic idea in the Karatsuba algorithm is to cut the two inputs x, y of length $\leq n$ into pieces of length $\leq n/2$ such that

$$x = a \cdot \beta^{n/2} + b, \qquad y = c \cdot \beta^{n/2} + d. \tag{6.1.1}$$

A usual divide–and–conquer approach would reduce the product of two integers of length n to four products of integers of length $n/2$. The complexity of this algorithm would still be quadratic in n. Karatsuba and Ofman, however, noticed that one of the three multiplications can be dispensed with.

$$\begin{aligned} x \cdot y &= ac\beta^n + (ad+bc)\beta^{n/2} + bd \\ &= ac\beta^n + \left((a+b)(c+d) - ac - bd\right)\beta^{n/2} + bd. \end{aligned} \tag{6.1.2}$$

So three multiplications of integers of length $n/2$ and a few shifts and additions are sufficient for computing the product $x \cdot y$.

algorithm INT_MULTK(**in:** x, y; **out:** z);
$[x, y$ integers; $z = x \cdot y]$
$n := \max(\text{LENGTH}(x), \text{LENGTH}(y))$;
if $n = 1$ **then** $\{z := \text{INT\_MULTC}(x, y)$; **return**$\}$;
if n is odd **then** $n := n + 1$;
$(a, b) := (\text{DEL}(n/2, x), \text{INIT}(n/2, x)$;
$(c, d) := (\text{DEL}(n/2, y), \text{INIT}(n/2, y)$;
$u := \text{INT\_MULTK}(a + b, c + d)$;
$v := \text{INT\_MULTK}(a, c)$;
$w := \text{INT\_MULTK}(b, d)$;
$z := v\beta^n + (u - v - w)\beta^{n/2} + w$;
return $\square$

Theorem 6.1.5. The complexity of the Karatsuba algorithm INT_MULTK is proportional to $n^{\log_2 3}$, where n is the length of the inputs.

Proof: Initially we assume that n is a power of 2. Let x and y be integers of length not exceeding n, and let a, b, c, d be the parts of x, y as in (6.1.1). During the execution of the Karatsuba algorithm we have to compute the products $(a+b)(c+d)$, ac, bd. All the other operations are additions and shifts, which take time proportional to n. The factors in ac and bd are of length not exceeding $n/2$, whereas the factors in $(a + b)(c + d)$ might be of length $n/2 + 1$. We write the factors as

$$a + b = a_1\beta^{n/2} + b_1, \quad c + d = c_1\beta^{n/2} + d_1, \tag{6.1.3}$$

where a_1 and c_1 are the leading digits of $a + b$ and $c + d$, respectively. Now

$$(a + b)(c + d) = a_1c_1\beta^n + (a_1d_1 + b_1c_1)\beta^{n/2} + b_1d_1. \tag{6.1.4}$$

In the product b_1d_1 the factors are of length not exceeding $n/2$. All the other operations are multiplications by a single digit or shifts, and together their complexity is proportional to n.

So if we denote the time for multiplying two integers of length n by $M(n)$, we get the recursion equation

$$M(n) = \begin{cases} k & \text{for } n = 1 \\ 3M(n/2) + kn & \text{for } n > 1. \end{cases} \tag{6.1.5}$$

Here we have taken k to be a bound for the complexity of multiplication of digits as well as for the constant factor in the linear complexity functions of the addition and shift operations. The solution to (6.1.5) is

$$M(n) = 3kn^{\log_2 3} - 2kn, \tag{6.1.6}$$

which can easily be verified by induction. This proves the assertion for all n which are powers of 2.

Finally let us consider the general case, where n is an arbitrary positive integer. In this case we could, theoretically, increase the length of the inputs to the next higher power of 2

by adding leading zeros. The length of the multiplicands is at most doubled in this process. In the asymptotic complexity, however, the factor 2 is negligible, since $(2n)^{\log_2 3} \sim n^{\log_2 3}$. □

The Karatsuba algorithm is practically used in computer algebra systems. In fact, the idea of Karatsuba and Ofman can be generalized to yield a multiplication algorithm of complexity $n^{1+\epsilon}$ for any positive real ϵ. We do not go into details here, but rather refer to the excellent exposition in [Knuth 81], Section 4.3.3. There is even a faster method based on the fast Fourier transform and ideas by Schönhage and Strassen [Schönhage,Strassen 71]. The complexity of this multiplication algorithm is proportional to $n \log n \log\log n$. For this faster method, however, the overhead is so enormous, that a practical importance seems very unlikely.

That the complexity of multiplication depends mainly on the smaller of the two inputs is explained by the following theorem.

Theorem 6.1.6. Let IM be a multiplication algorithm for integers with complexity $t^+_{\mathrm{IM}}(m)$ for multiplying two integers of lengths not greater than m, such that $m \preceq t^+_{\mathrm{IM}}(m)$. Then there exists a multiplication algorithm IM' with

$$t^+_{\mathrm{IM}'}(m,n) \preceq \begin{cases} (m/n) \cdot t^+_{\mathrm{IM}}(n) & \text{for } m \geq n \\ (n/m) \cdot t^+_{\mathrm{IM}}(m) & \text{for } m < n \end{cases},$$

for inputs of lengths m and n, respectively.

Proof: Let a, b be the integers to be multiplied, and $m = L(a), n = L(b)$. W.l.o.g. assume that $m \geq n$. IM' decomposes a into pieces $a_0, \ldots, a_{l-1}$ of length $\leq n$, such that $a = \sum_{i=0}^{l-1} a_i \cdot \beta^{ni}$. The number of pieces can be chosen as $l = \lceil m/n \rceil \leq (m/n) + 1$. Now each piece a_i is multiplied by b by algorithm IM and finally these partial results are shifted and added. Thus for some positive constant c

$$t^+_{\mathrm{IM}'}(m,n) \leq (\frac{m}{n} + 1) \cdot t^+_{\mathrm{IM}}(n) + (\frac{m}{n} + 1) \cdot cn \preceq (m/n) \cdot t^+_{\mathrm{IM}}(n),$$

which completes the proof. □

Division

The problem of integer division consists of computing the uniquely determined integers $q = \mathrm{quot}(a,b)$ and $r = \mathrm{rem}(a,b)$ for $a, b \in \mathbb{Z}$, $b \neq 0$, such that

$$a = q \cdot b + r \quad \text{and} \quad \begin{cases} 0 \leq r < \mid b \mid & \text{for } a \geq 0 \\ -\mid b \mid < r \leq 0 & \text{for } a < 0 \end{cases}.$$

If $\mid a \mid < \beta^j \mid b \mid$ for $j \in \mathbb{N}$, then q has at most j digits. j will be approximately $L(a) - L(b) + 1$. For determining the highest digit in the quotient one certainly does not need more than linear time in $L(b)$, even if all the possible digits are tried. So we get

Theorem 6.1.7 There is an algorithm INT_DIV for computing the quotient and remainder of two integers a, b of lengths m, n, respectively, $m \geq n$, in time $t^+_{\mathrm{INT\_DIV}}(m,n) \sim n \cdot (m - n + 1)$.

In fact we need not really try all the possible digits of the quotient, but there is a very efficient algorithmic way of "guessing" the highest digit. Such a method has

been described in [Pope,Stein 60], where the following theorem is proved. See also [Collins,Mignotte,Winkler 83].

Theorem 6.1.8 Let $a_{(\beta)} = [+, a_0, a_1, \ldots, a_{m-1}]$, $b_{(\beta)} = [+, b_0, \ldots, b_{n-1}]$, $\beta^j b \leq a < \beta^{j+1} b$ for $j \in \mathbb{N}$, $m \geq n$ and $b_{n-1} \geq \lfloor \beta/2 \rfloor$. If $\bar{q}$ is maximal in $\mathbb{Z}$ with $\bar{q}\beta^j b \leq a$ and $q^* = \lfloor (a_{n+j}\beta + a_{n+j-1})/b_{n-1} \rfloor$ (we set $a_i = 0$ for $i \geq L(a)$), then $\bar{q} \leq q^* \leq \bar{q} + 2$.

By a successive application of Theorem 6.1.8 the digits in the quotient q of a and b can be computed. Let $m = L(a)$, $n = L(b)$, $0 \leq a, 0 < b$, and $b_{n-1} \geq \lfloor \beta/2 \rfloor$. Then $a < \beta^{m-n+1} b$, so q has at most $m-n+1$ digits. First the hightest digit q_{m-n} is determined from the guess q^*. We need at most 2 correction steps of subtracting 1 from the initial guess. Collins and Musser have shown that the probabilities of q^* in Theorem 6.1.8 being $\bar{q} + i$ for $i = 0, 1, 2$ are $0.67, 0.32$, and 0.01, respectively. Now $a - \beta^{m-n} q_{m-n} b < \beta^{m-n} b$ and the process can be continued to yield q_{m-n-1} and so on.

The condition $b_{n-1} \geq \lfloor \beta/2 \rfloor$ can be satisfied by replacing a and b by $a' = a \cdot d$, $b' = b \cdot d$, respectively, where $d = \lfloor \beta/(b_{n-1} + 1) \rfloor$. This does not change the quotient q and $\mathrm{rem}(a, b) = (a' - q \cdot b')/d$.

These considerations lead to a better division algorithm INT_DIV, the *Pope–Stein algorithm*. The theoretical complexity function of the Pope–Stein algorithm, however, is still $n(m - n + 1)$, as in Theorem 6.1.7.

In [Aho,Hopcroft,Ullman 74] the relation in complexity of integer multiplication, division, and some other operations is investigated. It is shown that the complexity functions for multiplication of integers of length $\leq n$ and division of integers of length $\leq 2n$ by integers of length $\leq n$ are proportional.

Conversion

We assume that we have the arithmetic operations for integers in β–representation available. There are two types of conversions that we need to investigate: (1) conversion of an integer a from γ–representation into β–representation, and (2) conversion of a from β–representation into γ–representation.

It is quite obvious how we can do arithmetic with radix β^j, if we can do arithmetic with radix β. So in conversion problem (1) we may assume that $\gamma < \beta$, i.e. γ is a β–digit. If $a_{(\gamma)} = [a_0, \ldots, a_{n-1}]$, then we get $a_{(\beta)}$ by Horner's rule

$$a = (\cdots((a_{n-1}\gamma + a_{n-2})\gamma + a_{n-3})\gamma + \cdots + a_1)\gamma + a_0.$$

Every multiplication by γ takes time linear in the length of the multiplicand, and every addition of a digit a_i takes constant time. So the maximum complexity of conversion of type (1) is proportional to

$$\sum_{i=1}^{n-1} i \;\sim\; n^2 \;=\; L_\gamma(a)^2.$$

Conversion problem (2) can be solved by successive division by $\gamma = \gamma_{(\beta)}$. Every such division step reduces the length of the input by a constant, and takes time proportional to the length of the intermediate result, i.e. the maximum complexity of conversion of type (2) is proportional to $L_\beta(a)^2$.

Computation of greatest common divisors

$\mathbb{Z}$ is a unique factorization domain. So for any two integers x, y which are not both equal to 0, there is a greatest common divisor (gcd) g of x and y. g is determined up to multiplication by units, i.e. up to sign. Usually we mean the positive greatest common divisor when we speak of "the greatest common divisor". For the sake of completeness let us define $\gcd(0,0) := 0$.

But in addition to mere existence of gcds in $\mathbb{Z}$, there is also a very efficient algorithm due to Euclid ($\approx$ 330 – 275 B.C.) for computing the gcd. This is probably the oldest full fledged nontrivial algorithm in the history of mathematics. In later chapters we will provide an extension of the scope of Euclid's algorithm to its proper algebraic setting. But for the time being, we are just concerned with integers.

Suppose we want to compute $\gcd(x, y)$ for $x, y \in \mathbb{N}, y \neq 0$. We divide x by y, i.e. we determine the quotient q and the remainder r of x divided by y, such that

$$x = q \cdot y + r, \qquad \text{with } r < y.$$

Now $\gcd(x, y) = \gcd(y, r)$, i.e. the size of the problem has been reduced. This process is repeated as long as $r \neq 0$. Thus we get the so–called *Euclidean remainder sequence*

$$r_1, r_2, \ldots, r_n, r_{n+1},$$

with $r_1 = x, r_2 = y$, $r_i = \text{rem}(r_{i-2}, r_{i-1})$ for $n + 1 \geq i \geq 3$ and $r_{n+1} = 0$. Clearly $\gcd(x, y) = r_n$. Associated with this remainder sequence we get a sequence of quotients

$$q_1, \ldots, q_{n-1},$$

such that

$$r_i = q_i \cdot r_{i+1} + r_{i+2} \qquad \text{for } 1 \leq i \leq n - 1.$$

Thus in $\mathbb{Z}$ greatest common divisors can be computed by the *Euclidean algorithm* INT_GCDE.

---

algorithm INT_GCDE(**in:** x, y**; out:** g)**;**
[x, y are integers; $g = \gcd(x, y)$]
(1) $r' :=$ INT_ABS(x)**;** $r'' :=$ INT_ABS(y)**;**
(2) **while** $r'' \neq 0$ **do**
{$(q, r) :=$ INT_DIV(r', r'')**;**
$r' := r''$**;** $r'' := r$}**;**
(3) $g := r'$**;**
return □

The computation of gcds of integers is an extremely frequent operation in any computation in computer algebra. So we carefully have to analyze its complexity. G. Lamé proved already in the 19th century that for positive inputs bounded by n the number of division steps in the Euclidean algorithm is at most $\lceil \log_\phi(\sqrt{5}n) \rceil - 2$, where $\phi = \frac{1}{2}(1+\sqrt{5})$. See [Knuth 81], Section 4.5.3.

Theorem 6.1.9. Let l_1, l_2 be the lengths of the inputs x, y of INT_GCDE, and let k be the length of the output. Then $t^+_{\text{INT\_GCDE}}(l_1, l_2, k) \sim \min(l_1, l_2) \cdot (\max(l_1, l_2) - k + 1)$.

Proof: Steps (1) and (3) take constant time. So it remains to investigate the complexity behaviour of step (2), $t^+_2(l_1, l_2, k)$.

Let $r_1, r_2, \ldots, r_{n+1}$ be the remainder sequence and $q_1, \ldots, q_{n-1}$ the quotient sequence computed by INT_GCDE for the inputs x, y. If $\mid x \mid < \mid y \mid$ then the first iteration through the loop in (2) results in a reversal of the input pair. In this case the first iteration through the loop takes time proportional to $\min(l_1, l_2)$. So in the sequel we assume that $\mid x \mid \geq \mid y \mid > 0$. By Theorem 6.1.7

$$t^+_2(l_1, l_2, k) \preceq \sum_{i=1}^{n-1} L(q_i)L(r_{i+1}) \leq L(r_2) \cdot (\sum_{i=1}^{n-2} L(q_i + 1) + L(q_{n-1})). \tag{6.1.7}$$

$q_i \geq 1$ for $1 \leq i \leq n-2$ and $q_{n-1} \geq 2$. So by Exercise 2.1.5 in [Win96]

$$\sum_{i=1}^{n-2} L(q_i + 1) + L(q_{n-1}) \sim L(q_{n-1} \cdot \prod_{i=1}^{n-2} (q_i + 1)). \tag{6.1.8}$$

For $1 \leq i \leq n-2$ we have $r_{i+2}(q_i + 1) < r_{i+1}q_i + r_{i+2} = r_i$, and therefore $q_i + 1 < r_i / r_{i+2}$. Furthermore $q_{n-1} = r_{n-1}/r_n$. Thus

$$q_{n-1} \cdot \prod_{i=1}^{n-2} (q_i + 1) < \frac{r_{n-1} \cdot r_1 \cdot r_2}{r_n \cdot r_{n-1} \cdot r_n} \leq \left(\frac{r_1}{r_n}\right)^2. \tag{6.1.9}$$

Joining (6.1.7), (6.1.8), and (6.1.9) we finally arrive at

$$t^+_2(l_1, l_2, k) \preceq \min(l_1, l_2) \cdot L(\left(\frac{r_1}{r_n}\right)^2) \sim \min(l_1, l_2) \cdot (\max(l_1, l_2) - k + 1).$$

So $t^+_{\text{INT\_GCDE}}(l_1, l_2, k) \preceq \min(l_1, l_2) \cdot (\max(l_1, l_2) - k + 1)$.
From this it is easily shown that $t^+_{\text{INT\_GCDE}}(l_1, l_2, k) \sim \min(l_1, l_2) \cdot (\max(l_1, l_2) - k + 1)$. □

The greatest common divisor g of x and y generates ideal(x, y) in $\mathbb{Z}$. So in particular g can be written as a linear combination of x and y,

$$g = u \cdot x + v \cdot y.$$

These linear coefficients can be computed by a straighforward extension of INT_GCDE, the *extended Euclidean algorithm* INT_GCDEE. Throughout the algorithm INT_GCDEE the invariant

$$r' = u' \cdot x + v' \cdot y \quad \text{and} \quad r'' = u'' \cdot x + v'' \cdot y$$

is preserved.

```
algorithm INT_GCDEE(in: x, y; out: g, u, v);
[x, y are integers; g = gcd(x, y) = u · x + v · y]
(1) (r', u', v') := (INT_ABS(x), INT_SIGN(x),0);
    (r'', u'', v'') := (INT_ABS(y), 0, INT_SIGN(x));
(2) while r'' ≠ 0 do
        {q := INT_QUOT(r', r'');
        (r, u, v) := (r', u', v') − q · (r'', u'', v'');
        (r', u', v') := (r'', u'', v'');
        (r'', u'', v'') := (r, u, v); }
(3) (g, u, v) := (r', u', v');
    return □
```

6.2 Polynomials

Before we can design algorithms on polynomials, we need to introduce some notation and suitable representations.

Representations

A representation of polynomials can be either recursive or distributive, and it can be either dense or sparse. Thus, there are four basically different representations of multivariate polynomials.

In a recursive representation a nonzero polynomial $p(x_1, \ldots, x_n)$ is viewed as an element of $(R[x_1, \ldots, x_{n-1}])[x_n]$, i.e. as a univariate polynomial in the main variable x_n,

$$p(x_1, \ldots, x_n) = \sum_{i=0}^{m} p_i(x_1, \ldots, x_{n-1}) x_n^i, \quad \text{with } p_m \neq 0.$$

In the dense recursive representation p is represented as the list

$$p_{(dr)} = [(p_m)_{(dr)}, \ldots, (p_0)_{(dr)}],$$

where $(p_i)_{(dr)}$ is in turn the dense representation of the coefficient $p_i(x_1, \ldots, x_{n-1})$. If $n = 1$ then the coefficients p_i are elements of the ground ring R and are represented as such. The dense representation makes sense if many coeffients are different from zero. On the other hand, if the set of support of a polynomial is sparse, then a sparse recursive representation is better suited, i.e.

$$p(x_1, \ldots, x_n) = \sum_{i=0}^{k} p_i(x_1, \ldots, x_{n-1}) x_n^{e_i}, \quad e_0 > \cdots > e_k \text{ and } p_i \neq 0 \text{ for } 1 \leq i \leq k,$$

is represented as the list

$$p_{(sr)} = [e_1, (p_1)_{(sr)}, \ldots, e_k, (p_k)_{(sr)}].$$

In a distributive representation a nonzero polynomial $p(x_1, \ldots, x_n)$ is viewed as an element of $R[x_1, \ldots, x_n]$, i.e. a function from the set of power products in $x_1, \ldots, x_n$ into R. In a dense distributive representation we need a bijection $e : \mathbb{N} \longrightarrow \mathbb{N}^n$. A polynomial

$$p(x_1, \ldots, x_n) = \sum_{i=0}^{r} a_i x^{e(i)}, \quad \text{with } a_r \neq 0$$

is represented as the list

$$p_{(dd)} = [a_r, \ldots, a_0].$$

A sparse representation of

$$p(x_1, \ldots, x_n) = \sum_{i=0}^{s} a_i x^{e(j_i)}, \quad \text{with } a_i \neq 0 \text{ for } 1 \leq i \leq s$$

is the list

$$p_{(sd)} = [e(j_0), a_0, \ldots, e(j_s), a_s].$$

Which representation is actually employed depends of course on the algorithms that are to be applied. In later chapters we will see examples for algorithms that depend crucially on a recursive representation and also for algorithms that need a distributive representation. However, only very rarely will there be a need for dense representations in computer algebra. If the set of support of multivariate polynomials is dense, then the number of terms even in polynomials of modest degree is so big, that in all likelyhood no computations are possible any more.

In the sequel we will mainly analyze the complexity of operations on polynomials in recursive representation. So if not explicitly stated otherwise, the representation of polynomials is assumed to be recursive.

Addition and subtraction

The algorithms for addition and subtraction of polynomials are obvious: the coefficients of like powers have to be added or subtracted, respectively. If p and q are n–variate polynomials in dense representation, with $\max(\deg_{x_i}(p), \deg_{x_i}(q)) \leq d$ for $1 \leq i \leq n$, then the complexity of adding p and q is dominated by $A(p,q) \cdot (d+1)^n$, where $A(p,q)$ is the maximal time needed for adding two coefficients of p and q in the ground ring R. If p and q are in sparse representation, and t is a bound for the number of terms x_i^m with nonzero coefficient in p and q, for $1 \leq i \leq n$, then the complexity of adding p and q is dominated by $A(p,q) \cdot t^n$.

Multiplication

In the classical method for multiplying polynomials $p(x) = \sum_{i=0}^{m} p_i x^i$ and $q(x) = \sum_{j=0}^{n} q_j x^j$ the formula

$$p(x) \cdot q(x) = \sum_{l=0}^{m+n} \Big(\sum_{i+j=l} p_i \cdot q_j \Big) x^l$$

is employed. If p and q are n–variate polynomials in dense representation with d as above, then the complexity of multiplying p and q is dominated by $M(p,q) \cdot (d+1)^{2n}$, where $M(p,q)$ is the maximal time needed for multiplying two coefficients of p and q in the ground ring R. Observe that $(d+1)^n$ is a good measure of the size of the polynomials, when the size of the coefficients is neglected.

As for integer multiplication one can apply the Karatsuba method. I.e. the multiplicands p and q are decomposed as

$$p(x) = p_1(x) \cdot x^{[d/2]} + p_0(x), \quad q(x) = q_1(x) \cdot x^{[d/2]} + q_0(x),$$

and the product is computed as

$$p(x) \cdot q(x) = p_1 \cdot q_1 \cdot x^{2[d/2]} + \big((p_1 + p_0) \cdot (q_1 + q_0) - p_1 \cdot q_1 - p_0 \cdot q_0\big) \cdot x^{[d/2]} + p_0 \cdot q_0.$$

Neglecting the complexity of operations on elements of the ground ring R, we get $(d+1)^{n \log_2 3}$ as a dominating function for the complexity of mulitplying p and q.

For multiplying the sparsely represented polynomials $p(x) = \sum_{i=1}^{t} p_i x^{e_i}$ and $q(x) = \sum_{j=1}^{t} q_j x^{f_i}$, one basically has to (1) compute $p \cdot q_j x^{f_j}$ for $j = 1, \ldots, t$, and (2) add this to

the already computed partial result, which has roughly $(i-1)t$ terms, if $t \ll \deg(q), \deg(q)$. So the overall time complexity of multiplying polynomials in sparse representation is

$$\sum_{i=1}^{t} \big(\underbrace{M(p,q) \cdot t}_{(1)} + \underbrace{(i-1)t}_{(2)} \big) \sim M(p,q) \cdot t^3.$$

Division

First let us assume that we are dealing with univariate polynomials over a field K. If $b(x)$ is a nonzero polynomial in $K[x]$, then every other polynomial $a(x) \in K[x]$ can be divided by $b(x)$ in the sense that one can compute a *quotient* $q(x) = \mathrm{quot}(a,b)$ and a *remainder* $r(x) = \mathrm{rem}(a,b)$ such that

$$a(x) = q(x) \cdot b(x) + r(x) \quad \text{and} \quad (r(x) = 0 \text{ or } \deg(r) < \deg(b)). \tag{6.2.1}$$

The quotient q and remainder r in (6.2.1) are unique. The algorithm POL_DIVK computes the quotient and remainder for densely represented polynomials. It can easily be modified for sparsely represented polynomials.

---

algorithm POL_DIVK(**in:** a, b**; out:** q, r)**;**
[$a, b \in K[x]$, $b \neq 0$; $q = \mathrm{quot}(a,b), r = \mathrm{rem}(a,b)$. a and b are assumed to be in dense representation, the results q and r are likewise in dense representation]
(1) $q := [\,]$**;** $a' := a$**;** $c := \mathrm{lc}(b)$ **;** $m := \deg(a')$**;** $n := \deg(b)$**;**
(2) **while** $m \geq n$ **do**
$\{d := \mathrm{lc}(a')/c$ **;** $q := \mathrm{CONS}(d,q)$ **;** $a' := a' - d \cdot x^{m-n} \cdot b$ **;**
for $i = 1$ **to** $\min\{m - \deg(a') - 1, m - n\}$ **do** $q := \mathrm{CONS}(0,q)$ **;**
$m := \deg(a')\}$ **;**
(3) $q := \mathrm{INV}(q)$ **;** $r := a'$**; return** □

Theorem 6.2.1. Let $a(x), b(x) \in K[x]$, $b \neq 0$, $m = \deg(a), n = \deg(b)$, $m \geq n$. The maximal number of field operations in executing POL_DIVK on the inputs a and b is proportional to $n(m-n+1)$.

Proof: The **while**–loop is executed $m - n + 1$ times. The number of field operations in one pass through the loop is proportional to n. □

The case $K = \mathbb{Q}$ is of special importance in computer algebra. Assuming that the length of coefficients, i.e. the lengths of the numerators and the denominators, is bounded by d, then in the i–th iteration through the loop in POL_DIVK the length of the coefficients of a' and the length of the new coefficient added to q will be proportional to $i \cdot d$. So if the classical multiplication algorithm on the coefficients is used, the complexity of the i–th iteration is proportional to $n \cdot (id) \cdot d$. For the overall complexity of POL_DIVK over $\mathbb{Q}$ we get

$$nd^2 \sum_{i=1}^{m-n+1} i \;\sim\; nd^2(m-n+1)^2.$$

The algorithm POL_DIVK is not applicable any more, if the underlying domain of coefficients is not a field. In this case, the leading coefficient cannot be divided. Important

examples of such polynomial rings are $\mathbb{Z}[x]$ or multivariate polynomial rings. In fact, there are no quotient and remainder satisfying equation (6.2.1). However, it is possible to satisfy (6.2.1) if we allow to normalize the polynomial a by a certain power of the leading coefficient of b.

Theorem 6.2.2. Let R be an integral domain, $a(x), b(x) \in R[x]$, $b \neq 0$, and $m = \deg(a) \geq n = \deg(b)$. There are uniquely defined polynomials $q(x), r(x) \in R[x]$ such that

$$\mathrm{lc}(b)^{m-n+1} \cdot a(x) = q(x) \cdot b(x) + r(x) \quad \text{and} \quad (r(x) = 0 \text{ or } \deg(r) < \deg(b)). \tag{6.2.2}$$

Proof: R being an integral domain guarantees that multiplication of a polynomial by a non-zero constant does not change the degree.

For proving the existence of q and r we proceed by induction on $m-n$. For $m-n = 0$ the polynomials $q(x) = \mathrm{lc}(a)$, $r(x) = \mathrm{lc}(b) \cdot a - \mathrm{lc}(a) \cdot b$ obviously satisfy (6.2.2).

Now let $m - n > 0$. Let

$$c(x) := \mathrm{lc}(b) \cdot a(x) - x^{m-n} \cdot lc(a) \cdot b(x) \quad \text{and} \quad m' := \deg(c).$$

Then $m' < m$. For $m' < n$ we can set $q' := 0$, $r := \mathrm{lc}(b)^{m-n} \cdot c$ and we get $\mathrm{lc}(b)^{m-n} \cdot c(x) = q'(x) \cdot b(x) + r(x)$. For $m' \geq n$ we can use the induction hypothesis on c and b, yielding q_1, r_1 such that

$$\mathrm{lc}(b)^{m'-n+1} \cdot c = q_1 \cdot b + r_1 \quad \text{and} \quad (r_1 = 0 \text{ or } \deg(r_1) < \deg(b)).$$

Now we can multiply both sides by $\mathrm{lc}(b)^{m-m'-1}$ and we get

$$\mathrm{lc}(b)^{m-n} \cdot c(x) = q'(x) \cdot b(x) + r(x), \text{where } r = 0 \text{ or } \deg(r) < \deg(b).$$

Backsubstitution for c yields (6.2.2).

For establishing the uniqueness of q and r, we assume to the contrary that both q_1, r_1 and q_2, r_2 satisfy (6.2.2). Then $q_1 \cdot b + r_1 = q_2 \cdot b + r_2$, and $(q_1 - q_2) \cdot b = r_2 - r_1$. For $q_1 \neq q_2$ we would have $\deg((q_1 - q_2) \cdot b) \geq \deg(b) > \deg(r_1 - r_2)$, which is impossible. Therefore $q_1 = q_2$ and consequently also $r_1 = r_2$. □

---

algorithm POL_DIVP(**in:** a, b**; out:** q, r)**;**
[$a, b \in R[x]$, $b \neq 0$; $q = \mathrm{pquot}(a, b), r = \mathrm{prem}(a, b)$. a and b are assumed to be in dense representation, the results q and r are likewise in dense representation]
(1) $q := [\,]$**;** $a' := a$**;** $c := \mathrm{lc}(b)$ **;** $m := \deg(a')$**;** $n := \deg(b)$**;**
(2) **while** $m \geq n$ **do**
$\{d := \mathrm{lc}(a') \cdot c^{m-n}$ **;** $q := \mathrm{CONS}(d, q)$ **;** $a' := c \cdot a' - \mathrm{lc}(a') \cdot x^{m-n} \cdot b$ **;**
for $i = 1$ **to** $\min\{m - \deg(a') - 1, m - n\}$ **do**
$\{\ q := \mathrm{CONS}(0, q)$ **;** $a' := c \cdot a'\}$**;**
$m := \deg(a')\}$ **;**
(3) $q := \mathrm{INV}(q)$ **;** $r := a'$**; return** □

Definition 6.2.1. Let $R, a(x), b(x), m, n$ be as in Theorem 6.2.2. Then the uniquely defined polynomials $q(x)$ and $r(x)$ satisfying (6.2.2) are called the *pseudoquotient* and the *pseudoremainder*, respectively, of a and b. We write $q = \mathrm{pquot}(a, b), r = \mathrm{prem}(a, b)$. □

The algorithm POL_DIVP computes the pseudoquotient and pseudoremainder of two polynomials over an integral domain R.

As we will see later, pseudoremainders can be used in a generalization of Euclid's algorithm. The following is an important technical requirement for this generalization.

Lemma 6.2.3. Let $R, a(x), b(x), m, n$ be as in Theorem 6.2.2. Let $\alpha, \beta \in R$. Then $\mathrm{pquot}(\alpha \cdot a, \beta \cdot b) = \beta^{m-n} \cdot \alpha \cdot \mathrm{pquot}(a, b)$ and $\mathrm{prem}(\alpha \cdot a, \beta \cdot b) = \beta^{m-n+1} \cdot \alpha \cdot \mathrm{prem}(a, b)$.

Evaluation

Finally we consider the problem of evaluating polynomials. Let $p(x) = p_n x^n + \cdots + p_0 \in R[x]$ for a ring R and $a \in R$. We want to compute $p(a)$.

Successive computation and addition of $p_0, p_1 x, \ldots, p_n x^n$ requires $2n-1$ multiplications and n additions in R. A considerable improvement is obtained by *Horner's rule*, which evaluates p at a according to the scheme

$$p(a) = (\cdots(p_n \cdot a + p_{n-1}) \cdot a + \cdots) \cdot a + p_0,$$

requiring n multiplications and n additions in R. One get's Horner's rule from the computation of $\mathrm{rem}(p, x-a)$, by using the relation $p(a) = \mathrm{rem}(p, x-a)$. In fact, $p(a) = \mathrm{rem}(p, f)(a)$ for every polynomial f with $f(a) = 0$. In particular, for $f(x) = x^2 - a^2$ one gets the *2nd order Horner's rule*, which evaluates the polynomial

$$p(x) = \underbrace{\sum_{j=0}^{\lfloor n/2 \rfloor} p_{2j} x^{2j}}_{p^{(even)}} + \underbrace{\sum_{j=0}^{\lceil n/2 \rceil - 1} p_{2j+1} x^{2j+1}}_{p^{(odd)}}$$

at a as

$$\begin{aligned} p^{(even)} &= (\cdots(p_{2\lfloor n/2 \rfloor} \cdot a^2 + p_{2(\lfloor n/2 \rfloor - 1)}) \cdot a^2 + \cdots) \cdot a^2 + p_0, \\ p^{(odd)} &= ((\cdots(p_{2\lceil n/2 \rceil - 1} \cdot a^2 + p_{2\lceil n/2 \rceil - 3}) \cdot a^2 + \cdots) \cdot a^2 + p_1) \cdot a. \end{aligned}$$

The second order Horner's rule requires $n+1$ multiplications and n additions in R, which is no improvement over the 1st order Horner's rule. However, if both $p(a)$ and $p(-a)$ are needed, then the second evaluation can be computed by just one more addition.

6.3 Quotient fields

Let I be an integral domain and $Q(I)$ its quotient field. The arithmetic operations in $Q(I)$ can be based on (6.4.7). If I is actually a Euclidean domain, then we can compute normal forms of quotients by eliminating the gcd of the numerator and the denominator. We say that $r \in Q(I)$ is *in lowest terms* if numerator and denominator of r are relatively prime.

The rational numbers $\mathbb{Q}$ and the rational functions $K(x)$, for a field K, are important examples of such domains.

The efficiency of arithmetic depends on a clever choice of when exactly the gcd is eliminated in the result. In a classical approach numerators and denominators are computed according to (6.4.7) and afterwards the result is transformed into lowest terms. P. Henrici [Henrici 56] has devised the fastest known algorithms for arithmetic in such quotients fields. The so-called *Henrici algorithms* for addition and multiplication of r_1/r_2 and s_1/s_2 in $Q(I)$ rely on the following facts.

Theorem 6.3.1. *Let I be a Euclidean domain, $r_1, r_2, s_1, s_2 \in I$,* $\gcd(r_1, r_2) = \gcd(s_1, s_2) = 1$.

(a) If $d = \gcd(r_2, s_2), r_2' = r_2/d, s_2' = s_2/d,$
then $\gcd(r_1 s_2' + s_1 r_2', r_2 s_2') = \gcd(r_1 s_2' + s_1 r_2', d)$.

(b) If $d_1 = \gcd(r_1, s_2), d_2 = \gcd(s_1, r_2), r_1' = r_1/d_1, r_2' = r_2/d_2, s_1' = s_1/d_2, s_1' = s_2/d_1,$
then $\gcd(r_1' s_1', r_2' s_2') = 1$.

---

algorithm QF_SUMH(**in:** $r = (r_1, r_2), s = (s_1, s_2)$**; out:** $t = (t_1, t_2)$)**;**
[$r, s \in Q(I)$ in lowest terms. t is a representation of $r + s$ in lowest terms.]
(1) **if** $r_1 = 0$ **then** $\{t := s$**; return**$\}$**;**
if $s_1 = 0$ **then** $\{t := r$**; return**$\}$**;**
(2) $d := \gcd(r_2, s_2)$**;**
(3) **if** $d = 1$
then $\{t_1 := r_1 s_2 + r_2 s_1$**;** $t_2 := r_2 s_2\}$
else
$\{r_2' := r_2/d$**;** $s_2' := s_2/d$**;** $t_1' := r_1 s_2' + s_1 r_2'$**;** $t_2' := r_2 s_2'$**;**
if $t_1' = 0$
then $\{t_1 := 0$**;** $t_2 := 1\}$
else $\{e := \gcd(t_1', d)$**;**
if $e = 1$
then $\{t_1 := t_1'$**;** $t_2 := t_2'\}$
else $\{t_1 := t_1'/e$**;** $t_2 := t_2'/e\}$ $\}$ $\}$
return $\square$

Since the majority of the computing time is spent in extracting the gcd from the result, the Henrici algorithms derive their advantage from replacing one gcd computation of large inputs by several gcd computations for smaller inputs.

algorithm QF_MULTH(**in:** $r = (r_1, r_2), s = (s_1, s_2)$**; out:** $t = (t_1, t_2)$)**;**
[$r, s \in Q(I)$ in lowest terms. t is a representation of $r \cdot s$ in lowest terms.]
(1) **if** $r_1 = 0$ **or** $s_1 = 0$ **then** $\{t_1 := 0$**;** $t_2 := 1$**; return**$\}$**;**
(2) $d_1 := \gcd(r_1, s_2)$**;** $d_2 := \gcd(s_1, r_2)$**;**
(3) **if** $d_1 = 1$
then $\{r_1' := r_1$**;** $s_2' := s_2\}$
else $\{r_1' := r_1/d_1$**;** $s_2' := s_2/d_1\}$**;**
if $d_2 = 1$
then $\{s_1' := s_1$**;** $r_2' := r_2\}$
else $\{s_1' := s_1/d_2$**;** $r_2' := r_2/d_2\}$**;**
(4) $t_1 := r_1's_1'$**;** $t_2 := r_2's_2'$**;**
return □

Let us compare the complexities of the classical algorithm QF_SUMC versus the Henrici algorithm for addition in $\mathbb{Q}$. We will only take into account the gcd computations, since they are the most expensive operations in any algorithm.

Suppose $r = r_1/r_2, s = s_1/s_2 \in \mathbb{Q}$ and the numerators and denominators are bounded by n in length. In QF_SUMC we have to compute a gcd of 2 integers of length $2n$ each. In QF_SUMH we first compute a gcd of 2 integers of length n each, and, if $d = \gcd(r_2, s_2) \neq 1$ and $k = L(d)$, a gcd of integers of length $2n-k$ and k, respectively. We will make use of the complexity function for gcd computation stated in Theorem 6.1.9, i.e. $t^{+}_{\text{INT\_GCD}}(l_1, l_2, k) \sim \min(l_1, l_2) \cdot (\max(l_1, l_2) - k + 1)$.

If $d = 1$, then the computing time for the gcd in QF_SUMC is proportional to $4n^2$, whereas the gcd in QF_SUMH takes time proportional to n^2. So QF_SUMH is faster than QF_SUMC by a factor of 4.

Now let us assume that $d \neq 1$, $k = n/2$, and $e = 1$. In this case the gcd in QF_SUMC is proportional to $2n(2n-n/2) = 3n^2$. The times for the gcd computations in QF_SUMH are proportional to $n(n - n/2) = n^2/2$ and $(n/2)(3n/2) = 3n^2/4$. So in this case QF_SUMH is faster than QF_SUMC by a factor of 12/5.

The advantage of the Henrici algorithms over the classical ones becomes even more pronounced with increasing costs of gcd computations, e.g. in multivariate function fields like $\mathbb{Q}(x_1, \ldots, x_n) = Q(\mathbb{Q}[x_1, \ldots, x_n])$.

6.4 Algebraic extension fields

Let K be a field and α algebraic over K. Let $f(x) \in K[x]$ be the minimal polynomial of α and $m = \deg(f)$. For representing the elements in the algebraic extension field $K(\alpha)$ of K we use the isomorphism $K(\alpha) \cong K[x]_{/\langle f(x)\rangle}$. Every polynomial $p(x)$ can be reduced modulo $f(x)$ to some $q(x)$ with $\deg(q) < m$. On the other hand, two different polynomials $p(x), q(x)$ with $\deg(p), \deg(q) < m$ cannot be congruent modulo $f(x)$, since otherwise $p-q$, a non-zero polynomial of degree less than m, would be a multiple of f. Thus, every element $a \in K(\alpha)$ has a unique representation

$$a = \underbrace{a_{m-1}x^{m-1} + \cdots + a_1 x + a_0}_{a(x)} + \langle f(x)\rangle, \qquad a_i \in K.$$

We call $a(x)$ the *normal representation* of a, and sometimes we also write $a(\alpha)$.

From this unique normal representation we can immediately deduce that $K(\alpha)$ is a vector space over K of dimension m and with a basis $1, \alpha, \alpha^2, \ldots, \alpha^{m-1}$.

Consider, for instance, the field $\mathbb{Q}$ and let α be a root of $x^3 - 2$. $\mathbb{Q}(\alpha) = \mathbb{Q}[x]_{/\langle x^3-2\rangle}$ is an algebraic extension field of $\mathbb{Q}$, in which $x^3 - 2$ has a root, namely α, whose normal representation is x. So $\mathbb{Q}(\alpha) = \mathbb{Q}(\sqrt[3]{2})$ can be represented as $\{a_2x^2 + a_1x + a_0 | a_i \in \mathbb{Q}\}$. In $\mathbb{Q}(\alpha)$ the polynomial $x^3 - 2$ factors into $x^3 - 2 = (x - \alpha)(x^2 + \alpha x + \alpha^2)$.

Addition and subtraction in $K(\alpha)$ can obviously be carried out by simply adding and subtracting the normal representations of the arguments. The result is again a normal representation. Multiplication of normal representations and subsequent reduction (i.e. remainder computation) by the minimal polynomial $f(x)$ yields the normal representation of the product of two elements in $K(\alpha)$.

If we assume that the complexity of field operations in K is proportional to 1, then the complexity for addition and subtraction in $K(\alpha)$ is proportional to m. The complexity of multiplication is dominated by the complexity of the reduction modulo $f(x)$. A polynomial of degree $< 2m$ has to be divided by a polynomial of degree m, i.e. the complexity of multiplication in $K(\alpha)$ is proportional to m^2.

If we take into account the increase of coefficients in K, as indeed we must for instance in $\mathbb{Q}$, we get higher and usually more complicated complexity functions. In $\mathbb{Q}(\alpha)$ the complexity of addition and multiplication turn out to be proportional to md and m^3d^2, respectively, where the lengths of coefficients of the operands and the minimal polynomial are proportional to d.

The inverse a^{-1} of $a \in K(\alpha)$ can be computed by an application of the extended Euclidean algorithm E_EUCLID (see Chapter 3) to the minimal polynomial $f(x)$ and the normal representation $a(x)$ of a. Since $f(x)$ and $a(x)$ are relatively prime, the extended Euclidean algorithm yields the gcd 1 and linear factors $u(x), v(x) \in K[x]$ such that

$$u(x)f(x) + v(x)a(x) = 1 \qquad \text{and } \deg(v) < m.$$

So $v(x)$ is the normal representation of a^{-1}.

If we use the modular gcd algorithm, then the complexity of computing the inverse in $\mathbb{Q}(\alpha)$ is proportional to m^3d^3.

For example, let $\mathbb{Q}(\alpha)$ be as above, i.e. α a root of x^3-2. Let $a, b \in \mathbb{Q}(\alpha)$ with normal representations $a(x) = 2x^2 - x + 1$ and $b(x) = x + 2$, respectively. Then $a + b = 2x^2 + 3$, $a \cdot b = \mathrm{rem}(2x^3+3x^2-x+2, x^3-2) = 3x^2-x+6$. For computing a^{-1} we apply E_EUCLID to $x^3 - 2$ and $a(x)$, getting

$$\frac{1}{43}(2x - 19)(x^3 - 2) + \frac{1}{43}(-x^2 + 9x + 5)a(x) = 1.$$

So a^{-1} has the normal representation $(-x^2 + 9x + 5)/43$.

An algebraic extension $K(\alpha)$ over K with minimal polynomial $f(x)$ is separable if and only if $f(x)$ has no multiple roots or, in other words, $f'(x) \neq 0$. In characteristic 0 every algebraic extension is separable. Let $K(\alpha_1)\ldots(\alpha_n)$ be a multiple algebraic extension of K, so α_i is the root of an irreducible polynomial $f_i(x) \in K(\alpha_1)\ldots(\alpha_{i-1})$. For every such multiple separable algebraic field extension there exists an algebraic element γ over K such that

$$K(\alpha_1)\ldots(\alpha_n) = K(\gamma),$$

i.e. every multiple separable algebraic extension can be rewritten as a simple algebraic extension. γ is a *primitive element* of this algebraic extension.

6.5 Finite fields

Modular arithmetic in residue class rings

Every integer m generates an ideal $\langle m \rangle$ in $\mathbb{Z}$. Two integers a, b are *congruent* modulo $\langle m \rangle$ iff $a - b \in \langle m \rangle$, or in other words, $m|a - b$. In this case we write

$$a \equiv b \bmod m \quad \text{or} \quad a \equiv_{\bmod\ m} b.$$

So obviously a and b are congruent modulo m if and only if they have the same residue modulo m. Since $\equiv_{\bmod\ m}$ is an equivalence relation, we get a decomposition of the integers into eqivalence classes, the *residue classes* modulo m.

Let us consider the residue classes of integers modulo any positive integer m. In general, this is not a field, not even an integral domain, but just a commutative ring. This commutative ring is called the *residue class ring modulo* m and it is denoted by $\mathbb{Z}_{/\langle m \rangle}$ or just $\mathbb{Z}_{/m}$. The residue class ring is a field if and only if the modulus m is a prime number.

For the purpose of computation in such a residue class ring we need to choose representations of the elements. There are two natural representations for $\mathbb{Z}_{/m}$, namely as the residue classes corresponding to

$$\{0, 1, \ldots, m-1\} \qquad (\text{least non} - \text{negative representation})$$

or the residue classes corresponding to

$$\{a \in \mathbb{Z} | -\frac{m}{2} < a \leq \frac{m}{2}\}. \qquad (\text{zero} - \text{centered representation})$$

Both representations are useful in specific applications. Of course a change of representations is trivial.

The canonical homomorphism H_m which maps an integer a to its representation in $\mathbb{Z}_{/m}$ is simply the computation of the remainder of a w.r.t. m. So according to Theorem 6.1.7 it takes time proportional to $L(m)(L(a) - L(m) + 1)$.

Addition $+_m$ and multiplication $\cdot_m$ in $\mathbb{Z}_{/m}$ are defined as follows on the representatives

$$a +_m b = H_m(a + b), \quad a \cdot_m b = H_m(a \cdot b).$$

Using this definition and the bounds of Section 2.1, we see that the obvious algorithms MI_SUM and MI_MULT for $+_m$ and $\cdot_m$, respectively, have the complexities $t^+_{MI\_SUM} \sim t^*_{MI\_SUM} \sim L(m)$ and $t^+_{MI\_MULT} \sim L(m)^2 + L(m)(2L(m) - L(m) + 1) \sim L(m)^2$. So even if we use faster multiplication methods, the bound for MI_MULT does not decrease.

An element $a \in \mathbb{Z}_{/m}$ has an inverse if and only if $\gcd(m, a) = 1$, otherwise a is a zero-divisor. So we can decide whether a can be inverted and if so compute a^{-1} by applying INT_GCDEE to m and a. If a is invertible we will get a linear combination $u \cdot m + v \cdot a = 1$, and v will be the inverse of a. The time complexity for computing the inverse is bounded by $L(m)^2$.

An important relation of modular arithmetic is captured in Fermat's "Little" Theorem. Theoretically this provides an alternative method for computing inverses modulo primes.

Theorem 6.5.1. (Fermat's Little Theorem) *If p is a prime and a is an integer not divisible by p, then $a^{p-1} \equiv 1 \bmod p$.*

Proof: First we proof a more general fact:
(*) *In a finite abelian group $\langle A, \cdot, 1\rangle$ with n elements we have $a^n = 1$ for all $a \in A$.*
Let $A = \{a_1, \ldots, a_n\}$. For every $a \in A$ we have $aa_i = aa_j$ only if $a_i = a_j$; so also $A = \{aa_1, \ldots, aa_n\}$. We get $\prod_{i=1}^n a_i = \prod_{i=1}^n (aa_i) = a^n \prod_{i=1}^n a_i$, which implies $a^n = 1$.

Now if we consider the multiplicative subgoup of $\mathbb{Z}_p$ (with $p-1$ elements), we get the theorem. □

Arithmetic in finite fields

Now let us turn to finite fields. As we have seen above, $\mathbb{Z}_{/p}$ will be a finite field if and only if p is a prime number. In general a finite field need not have prime cardinality. However, every finite field has cardinality p^n, for p a prime. On the other hand, for every prime p and natural number n there exists a unique (up to isomorphism) finite field of order p^n. This field is usually denoted $GF(p^n)$, the *Galois field* of order p^n.

From what we have derived in previous sections, it is not difficult to construct and compute in Galois fields. Let $f(x)$ be an irreducible polynomial of degree n in $\mathbb{Z}_p[x]$. We will see later how to check irreducibility efficently, and in fact for every choice of p and n there is a corresponding f. Then f determines an algebraic field extension of $\mathbb{Z}$ of degree n, i.e.

$$GF(p^n) \cong \mathbb{Z}_p[x]_{/\langle f(x)\rangle}.$$

So the arithmetic operations can be handled as in any algebraic extension field.

For a thorough introduction to the theory of finite fields we refer to [Lidl,Niederreiter 83], [Lidl,Pilz 84]. Here we list only some facts that will be useful in subsequent chapters.
$GF(p^n)$ is the splitting field of $x^{p^n} - x$ over $\mathbb{Z}_p$, i.e.

$$x^{p^n} - x = \prod_{a \in GF(p^n)} (x - a).$$

Every $\beta \in GF(p^n)$ is algebraic over $\mathbb{Z}_p$. If s is the smallest postitive integer such that $\beta^{p^s} = \beta$, then $m_\beta(x) = \prod_{i=0}^{s-1}(x - \beta^{p^i})$ is the minimal polynomial of β over $\mathbb{Z}_p$. The multiplicative group of $GF(p^n)$ is cyclic. A generating element of this cyclic group is called a *primitive element* of the Galois field.

An important property of Galois fields is the "freshman's dream".

Theorem 6.5.2. *Let $a(x), b(x) \in GF(p^n)[x]$. Then $(a(x) + b(x))^p = a(x)^p + b(x)^p$.*

Proof: In the binomial expansion of the left hand side

$$\sum_{i=0}^{p} \binom{p}{i} a(x)^i b(x)^{p-i}$$

all the binomial coefficients except the first and the last are divisible by p and for those we have $\binom{p}{0} = 1 = \binom{p}{p}$. □

Corollary *Let $a(x) \in \mathbb{Z}_p[x]$. Then $a(x)^p = a(x^p)$.*

Proof: Let $a(x) = \sum_{i=0}^{m} a_i x^i$. By the theorem and Fermat's Little Theorem we have $a(x)^p = \sum_{i=0}^{m} (a_i x^i)^p = \sum_{i=0}^{m} a_i x^{ip} = a(x^p)$. □

Example 6.5.1. Let us carry out some of these constructions in $GF(2^4)$. The polynomial $f(x) = x^4 + x + 1$ is irreducible over $\mathbb{Z}_2$, so

$$GF(2^4) \cong \mathbb{Z}_2[x]_{/\langle x^4+x+1\rangle}.$$

Let α be a root of f. Every $\beta \in GF(2^4)$ has a unique representation as

$$\beta = b_0 + b_1\alpha + b_2\alpha^2 + b_3\alpha^3, \quad b_i \in \mathbb{Z}_2.$$

The following is a table of $GF(2^4)$:

elements $\beta = b_0 + b_1\alpha + b_2\alpha^2 + b_3\alpha^3$ of $\mathbb{Z}_2(\alpha)$, where $\alpha^4 + \alpha + 1 = 0$

β	b_0	b_1	b_2	b_3	minimal polynomial of β over $\mathbb{Z}_2$
0	0	0	0	0	x
1	1	0	0	0	$x+1$
α	0	1	0	0	x^4+x+1
α^2	0	0	1	0	x^4+x+1
α^3	0	0	0	1	$x^4+x^3+x^2+x+1$
$1+\alpha=a^4$	1	1	0	0	x^4+x+1
$\alpha+\alpha^2=\alpha^5$	0	1	1	0	x^2+x+1
$\alpha^2+\alpha^3=\alpha^6$	0	0	1	1	$x^4+x^3+x^2+x+1$
$1+\alpha+\alpha^3=\alpha^7$	1	1	0	1	x^4+x^3+1
$1+\alpha^2=\alpha^8$	1	0	1	0	x^4+x+1
$\alpha+\alpha^3=\alpha^9$	0	1	0	1	$x^4+x^3+x^2+x+1$
$1+\alpha+\alpha^2=\alpha^{10}$	1	1	1	0	x^2+x+1
$\alpha+\alpha^2+\alpha^3=\alpha^{11}$	0	1	1	1	x^4+x^3+1
$1+\alpha+\alpha^2+\alpha^3=\alpha^{12}$	1	1	1	1	$x^4+x^3+x^2+x+1$
$1+\alpha^2+\alpha^3=\alpha^{13}$	1	0	1	1	x^4+x^3+1
$1+\alpha^3=\alpha^{14}$	1	0	0	1	x^4+x^3+1
$1=\alpha^{15}$	1	0	0	0	$x+1$

For computing the minimal polynomial $m_\beta(x)$ for $\beta = \alpha^6$, we consider the powers $\beta^2 = \alpha^{12}, \beta^4 = \alpha^9, \beta^8 = \alpha^3, \beta^{16} = \alpha^6 = \beta$, so

$$m_\beta(x) = \prod_{i=0}^{3} (x - \beta^{p^i}) = x^4 + x^3 + x^2 + x + 1.$$

Every $\beta \in GF(2^4)$ is a power of α, so α is a primitive element of $GF(2^4)$. However, not every irreducible polynomial has a primitive element as a root. For instance, $g(x) = x^4 + x^3 + x^2 + x + 1$ also is irreducible over $\mathbb{Z}_2[x]$, so $GF(2^4) = \mathbb{Z}_2[x]_{/\langle g(x)\rangle}$. But β, a root of g, is not a primitive element of $GF(16)$, since $\beta^5 = 1$. □