

Arithmetic in Basic Algebraic Domains

G. E. Collins, Madison, M. Mignotte, Strasbourg, and F. Winkler, Linz

Abstract

This chapter is devoted to the arithmetic operations, essentially addition, multiplication, exponentiation, division, gcd calculation and evaluation, on the basic algebraic domains. The algorithms for these basic domains are those most frequently used in any computer algebra system. Therefore the best known algorithms, from a computational point of view, are presented. The basic domains considered here are the rational integers, the rational numbers, integers modulo m , Gaussian integers, polynomials, rational functions, power series, finite fields and p -adic numbers. Bounds on the maximum, minimum and average computing time (t^+ , t^- , t^*) for the various algorithms are given.

The Rational Integers

Most of the material for this section has been taken from [11]. There the reader may also find the details for the following computing time analyses.

Representation

For specifying the algorithms for the various operations on \mathbb{Z} we represent the elements of \mathbb{Z} in a positional number system. In order to do so we choose a natural number $\beta \geq 2$ as the base or radix of the positional system. Let $\{-(\beta - 1), \dots, -1, 0, 1, \dots, (\beta - 1)\}$ be the set of β -integers. A positive integer a is then represented (in the positional number system with radix β) by the unique sequence $a_{(\beta)} = (a_0, \dots, a_{n-1})$ of nonnegative β -integers, such that $a_{n-1} > 0$ and $a = \sum_{k=0}^{n-1} a_k \cdot \beta^k$. Similarly we represent a negative number a by the unique sequence (a_0, \dots, a_{n-1}) of nonpositive β -integers, such that $a_{n-1} < 0$. Finally, we let the empty sequence $()$ represent the integer zero. For practical implementation most systems represent only the integers a with $|a| \geq \beta$ as lists, whereas small integers are not represented as lists in order to avoid function call overhead. With this convention 0 is represented by the integer 0 and not by $()$.

We want to be a little bit more specific, at this point, about the data structure of this representation. Since almost never the lengths of the integers occurring during the execution of an algorithm are known in advance, it would be extremely wasteful to allocate the storage statically at the beginning of an algorithm. A means for avoiding this inefficiency is list representation. Thus we represent an element $a \in \mathbb{Z}$ as the list of β -digits of a , the least significant digit a_0 being the first element in the list. Relative to the alternative representation (a_{n-1}, \dots, a_0) this has both advantages and disadvantages. For addition and multiplication we have the advantage that carries propagate in the direction that the lists are scanned. But there is the

disadvantage that the determination of the sign is not immediate if some of the least significant digits are zero, and there is the further disadvantage that in division the most significant digits are required first. All in all, however, the advantages appear to significantly outweigh the disadvantages.

For $a \neq 0$, represented as (a_0, \dots, a_{n-1}) , we call n the β -length of a , denoted by $L_\beta(a)$, and we let $L_\beta(0) = 1$. The β -length of a is essentially the β -logarithm of $|a|$, precisely $L_\beta(a) = \lfloor \log_\beta |a| \rfloor + 1$ for $a \neq 0$, where $\lfloor \cdot \rfloor$ denotes the floor function. If γ is another basis, then $L_\beta(a)$ and $L_\gamma(a)$ are codominant. So we often just speak of $L(a)$, the length of a .

Though list representation is extremely helpful for minimizing the storage needed for an algorithm, it also has the disadvantage that mathematically seemingly trivial operations become nontrivial.

Conversion and Input/Output

A problem frequently occurring is that of conversion from one radix to another. Let us assume that we are converting the integer a from radix β to radix γ . Basically there are two algorithms for this purpose.

The first algorithm, CONVM, involves multiplication by β using only radix γ arithmetic. It amounts to evaluating the polynomial $a(x) = \sum_{i=0}^{n-1} a_i \cdot x^i$ at $x = \beta$ using Horner's method, where $a_{(\beta)} = (a_0, \dots, a_{n-1})$.

The second algorithm, CONVD, uses only radix β arithmetic and is essentially a repeated division by γ (the remainders being the digits of $a_{(\gamma)}$).

The computing times for both methods are proportional to $L(a)^2$.

One of the most important usages of conversion is for input and output. Here the problem is to convert from decimal to radix β representation (where β usually is a constant whose length differs only slightly from the word size of the machine) and vice versa. Since arithmetic operations will be developed for radix β representation, it is natural to employ algorithm CONVM for input and CONVD for output.

Instead of converting from decimal to radix β representation at once, it is preferable (see [23], Section 4.4) to convert first to radix 10^η ($\eta = \lfloor \log_{10} \beta \rfloor$) and to compute the radix β representation from there. Each digit in the radix 10^η representation is completely determined by η consecutive decimal digits, so the conversion of a from decimal to radix 10^η takes time proportional to $L(a)$. The conversion from radix 10^η to radix β will take time proportional to $L(a)^2$, just as would direct conversion from decimal to radix β , but the constant of proportionality for direct conversion is about η times as large. Since η will typically be approximately 10, a substantial saving is realized. A similar remark holds for conversion from radix β to decimal.

Arithmetic Operations

The major operations we want to perform are comparison, addition, subtraction, multiplication, exponentiation, division and gcd-computation.

In many algorithms we need the computation of the sign of an integer. Because of our data structure the algorithm ISIGN is not totally trivial, but still quite straightforward:

$s \leftarrow \text{ISIGN}(a)$

[integer sign. a is an integer. $s = \text{sign}(a)$, a β -integer]

Successively check a_0, a_1, \dots until the first non-zero digit a_k is found, from which the sign s can be determined. ■

Clearly, $t_{\text{ISIGN}}^+ \sim L(a)$, $t_{\text{ISIGN}}^- \sim 1$. Surprisingly, $t_{\text{ISIGN}}^* \sim 1$ (actually $< (1 - \beta^{-1})^{-2}$). This can be seen from the following consideration: the number of integers of length n ($n \geq 2$), for which k digits have to be checked, is $2(\beta - 1)$ for $k = n$ and $2(\beta - 1)^2 \beta^{n-k-1}$ for $k < n$. Since there are $2(\beta - 1)\beta^{n-1}$ integers of length n ($n \geq 2$), the average number of digits to be checked for integers of length n is

$$N_n = (\beta - 1) \sum_{k=1}^{n-1} k\beta^{-k} + n\beta^{-n+1} < \sum_{k=1}^{\infty} k\beta^{-k+1} = (1 - \beta^{-1})^{-2}.$$

Algorithms for negation, INEG, and computation of the absolute value, IABS, are straightforward. Their computing times are proportional to the length of the input integer a .

Next we give an algorithm for comparing two integers a and b . Let us consider the following example: $\beta = 10$ and we want to compare the two integers $a = -102$ and $b = 3810$. a and b are represented by the lists $(-2, 0, -1)$ and $(0, 1, 8, 3)$, respectively. First we look at the first digits of a and b . Since $b_0 = 0$, we do not know the sign of b . Thus the decision has to be delayed, but we remember that the sign of a is -1 . Inspecting the next two digits, we find that the sign of b is 1. Hence, $a < b$. As a second example consider $c = -12$ and $d = -800$, represented as the lists $(-2, -1)$ and $(0, 0, -8)$. After inspecting the first two pairs of digits a decision cannot be made. But since the end of the first list is reached we know that the absolute value of b is greater than that of a . Thus we can make the decision if we determine the sign of b . These ideas are incorporated in the following algorithm:

$s \leftarrow \text{ICOMP}(a, b)$

[integer comparison. a and b are integers. $s = \text{sign}(a - b)$]

- (1) Set u and v to zero and inspect successive digits $a_0, b_0, a_1, b_1, \dots$ until the end of one of the lists is reached. At the k th step ($k = 0, 1, \dots$) set $u \leftarrow \text{sign}(a_k)$ if $a_k \neq 0$, $v \leftarrow \text{sign}(b_k)$ if $b_k \neq 0$ and check whether $u \cdot v = -1$. If so, return $s \leftarrow u$. Otherwise set $s \leftarrow \text{sign}(a_k - b_k)$.
- (2) If both lists are finished, then return s .
- (3) If the list for a is finished then return $s \leftarrow \text{ISIGN}((b_{k+1}, \dots))$ otherwise $s \leftarrow \text{ISIGN}((a_{k+1}, \dots))$. ■

The average computing time for ICOMP is proportional to $\min(L(a), L(b))$.

Our next problem is summation of two integers a and b . We divide the problem into the following subproblems:

- either a or b is zero: in this case we simply return b or a as the sum,
 neither a nor b is zero and $\text{ISIGN}(a) = \text{ISIGN}(b)$,
 neither a nor b is zero and $\text{ISIGN}(a) \neq \text{ISIGN}(b)$.

The usual classical algorithm for the second subproblem ([23], p. 251) takes time proportional to $\max(L(a), L(b))$. If, however, a and b are not of the same length, the carry usually will not propagate far beyond the most significant digit of the shorter input. This gives us an algorithm with computing time proportional to $\min(L(a), L(b))$:

$$c \leftarrow \text{ISUM1}(a, b)$$

[integer sum, first algorithm. a and b are integers with $\text{ISIGN}(a) = \text{ISIGN}(b)$.
 $c = a + b$]

- (1) Successively add the digits of a and b with carry until one of the two lists (say b) is finished, getting c_0, \dots, c_k and a carry d .
- (2) Add the carry to the successive digits of the longer input, getting c_{k+1}, \dots, c_l , until either the carry disappears, in which case the concatenation of the two lists (c_0, \dots, c_l) and (a_{l+1}, \dots) is returned, or the longer list is also finished, in which case (c_0, \dots, c_l, d) is returned. ■

Now we would like to have an algorithm for the summation of two integers with different signs, which also makes use of the above observation in order to achieve a computing time proportional to $\min(L(a), L(b))$. Let us first consider an example: $\beta = 10$, $a = 30578$ and $b = -2095$. If we just look at the least significant digits in the representations $(8, 7, 5, 0, 3)$ and $(-5, -9, 0, -2)$ of a and b we cannot know the sign of the result c in advance. So we simply add the successive digits of a and b without carry (the partial results are guaranteed to be β -digits), getting $(3, -2, 5, -2)$ until one of the lists, in this case b , is finished. Now we look at the remaining digit of a and see that the result has to be positive. So we go back and change all digits to positive ones, getting $(3, 8, 4, 8, 2)$.

$$c \leftarrow \text{ISUM2}(a, b)$$

[integer sum, second algorithm. a and b are integers with $\text{ISIGN}(a) = \text{ISIGN}(b)$.
 $c = a + b$]

- (1) Set $u \leftarrow 0$ and successively add the digits of a and b without carry until one of the two lists (say b) is finished, getting c_0, \dots, c_k , and at each step set $u \leftarrow \text{sign}(c_i)$ if $c_i \neq 0$.
- (2) If both lists are finished and all the c_i ($0 \leq i \leq k$) are zero, then return $c \leftarrow ()$.
- (3) If $u \neq \text{sign}(a)$ then successively set $c_{k+1} \leftarrow a_{k+1}, \dots, c_l \leftarrow a_l$ until $a_l \neq 0$.
- (4) Now $\text{sign}(c) = \text{sign}(c_l)$. Go back to c_0 and propagate carries, getting new c_0, \dots, c_l .
- (5) Concatenate the two lists (c_0, \dots, c_l) and (a_{k+1}, \dots) , getting the result c . ■

Note that an average computing time proportional to $\min(L(a), L(b))$ can only be achieved by overlapping of lists, a feature not available for integers represented by arrays.

The difference of two integers a and b can be computed as $\text{ISUM}(a, \text{INEG}(b))$, the average computing time being proportional to $L(b)$.

The next problem to attack is multiplication. The "classical" algorithm ([23], p. 253) for multiplying two integers a and b has a computing time proportional to $L(a) \cdot L(b)$. For very long integers a and b , however, we can do much better than that by using an algorithm attributable to A. Karatsuba [22]. The idea is to bisect the two integers a and b into two parts each, getting $a = A_1\beta^k + A_0, b = B_1\beta^k + B_0$ where $k = \max(m, n)/2$, $m = L(a)$, $n = L(b)$ and $A_0 = (a_0, \dots, a_{k-1})$, $A_1 = (a_k, \dots, a_{m-1})$, $B_0 = (b_0, \dots, b_{k-1})$, $B_1 = (b_k, \dots, b_{n-1})$. We might view this as representing a and b in radix β^k representation. The product $c = a \cdot b$ can then be computed with only three multiplications of integers of lengths k or less plus some shiftings and additions using the formula

$$(*) \quad c = a \cdot b = (A_1 \cdot B_1)\beta^{2k} + (A_1B_0 + A_0B_1)\beta^k + (A_0B_0)$$

where $A_1B_0 + A_0B_1 = (A_1 + A_0)(B_1 + B_0) - A_1B_1 - A_0B_0$.

If k itself is still large then the same method can be reapplied for computing the three smaller products. Thus we get a recursive integer multiplication algorithm. The time needed to multiply two integers of lengths n or less is three times the time needed for multiplying three integers of lengths $n/2$ plus some linear term in n . The solution of the equation

$$M(n) = 3 \cdot M(n/2) + d \cdot n$$

is $M(n) = n^{\log_2 3}$ (see [1], p. 64). Thus the cost of multiplication has been reduced from $L(a) \cdot L(b)$ to $\max(L(a), L(b))^{\log_2 3}$.

The critical values for $L_\beta(a)$ and $L_\beta(b)$, for which the Karatsuba method becomes faster than the classical method, the extra additions taking less time than the saved multiplication (the so-called trade-off point), depend heavily on the machine and on the basic operations for list processing. Let us assume, for the time being, that this trade-off point is reached for $\min(L_\beta(a), L_\beta(b)) \geq N$ (a realistic assumption is that a and b have more than 200 decimal digits). Then we can describe an algorithm based on the Karatsuba method as follows:

$$c \leftarrow \text{IPRODK}(a, b)$$

[integer product, Karatsuba method. a and b are integers. $c = a \cdot b$]

- (1) For short integers (i.e. $\max(L_\beta(a), L_\beta(b)) < N$) apply the "classical" multiplication algorithm and return the result as the value of c .
- (2) Compute the partial results $A_1B_1, A_1B_0 + A_0B_1, A_0B_0$ using (*) and applying IPRODK recursively.
- (3) Combine the partial results by shifting and adding, getting the result c . ■

Algorithm IPRODK is just a slight modification of the first of an infinite sequence of algorithms M_1, M_2, M_3, \dots . In algorithm M_r , we cut a and b into $r + 1$ pieces

such that

$$a = \sum_{i=0}^r A_i \cdot \beta^{ik}, \quad b = \sum_{i=0}^r B_i \cdot \beta^{ik},$$

where $k = \lceil \max(L_\beta(a), L_\beta(b)) / (r + 1) \rceil$. a and b are just the values of the two polynomials

$$a(x) = \sum_{i=0}^r A_i x^i, \quad b(x) = \sum_{i=0}^r B_i x^i$$

at the point $x = \beta^k$. Hence, if we set $c(x) = a(x) \cdot b(x)$, then $c = a \cdot b = c(\beta^k)$. $c(x)$ is a polynomial of degree $2r$ or less. Therefore the coefficients of $c(x)$ can be computed from the values at $2r + 1$ distinct integer points, say $0, \mp 1, \mp 2, \dots, \mp r$, using interpolation. In fact, if

$$c(x) = \sum_{i=0}^{2r} c_i x^i$$

then c_i is a linear combination

$$c_i = \sum_{j=-r}^r d_{ij} \cdot c(j)$$

in which the d_{ij} are certain rational numbers independent of a and b . The computing time of M_r is dominated by $\max(L(a), L(b))^{\alpha_r}$, where $\alpha_r = \log_{r+1}(2r + 1) < 1 + \log_{r+1} 2$. Since $\log_{r+1} 2$ converges to 0 for $r \rightarrow \infty$, there is a multiplication algorithm with computing time dominated by $\max(L(a), L(b))^{1+\epsilon}$ for every $\epsilon > 0$. (Of course, the factor of proportionality grows accordingly).

Other algorithms are known (see [23], Section 4.3.3) which are faster than every algorithm M_r . The fastest currently known algorithm, due to A. Schönhage and V. Strassen [36] has a computing time dominated by $m \cdot L(m) \cdot L(L(m))$ where $m = \max(L(a), L(b))$. These results are of theoretical interest but have not yet an immediate practical significance for computer algebra because these "fast" algorithms are faster only for enormous numbers.

When a multiplication algorithm is given whose computing time for multiplying two integers of length m or less is dominated by $M(m)$, one can always design a multiplication algorithm whose computing time for multiplying two integers a and b (with $m = L(a)$, $n = L(b)$) is dominated by the function

$$M'(m, n) = \begin{cases} m \cdot n^{-1} \cdot M(n) & \text{if } m \geq n \\ n \cdot m^{-1} \cdot M(m) & \text{if } m < n \end{cases}.$$

This can be achieved by breaking a up (assuming without loss of generality that $m \geq n$) into pieces each of length n or less and then multiplying each piece by b .

These fast multiplication algorithms can be used to improve the computing times of many other algorithms. One important application is conversion. R. P. Brent [4] has shown how to make use of fast multiplication for efficiently evaluating functions such as \log , \exp and \arctan (an evaluation to n significant digits can be

achieved in time dominated by $M(n) \cdot \log(n)$ if the time needed for multiplying two n -digit numbers is dominated by $M(n)$.

A problem related to multiplication is exponentiation. The most obvious algorithm for computing a^n , $n > 1$, uses repeated multiplications by a , computing a, a^2, a^3, \dots, a^n . But we also consider the so-called left-to-right and right-to-left binary exponentiation methods.

Let $n_{(2)} = \sum_{i=0}^{k-1} b_i 2^i$ be the binary representation of n . In the left-to-right binary exponentiation method we compute successively $a^{n_1}, a^{n_2}, \dots, a^{n_k}$ for $n_i = \lfloor n/2^{k-i} \rfloor$ (note that $n_k = n$) by the formula

$$a^{n_{i+1}} = \begin{cases} (a^{n_i})^2, & \text{if } b_{k-i-1} = 0 \\ (a^{n_i})^2 a, & \text{if } b_{k-i-1} = 1 \end{cases}.$$

In the right-to-left method we compute two sequences a_0, \dots, a_k and c_0, \dots, c_{k-1} in which

$$c_0 = a, \quad c_{i+1} = c_i^2 \quad \text{and} \quad a_0 = 1, \quad a_{i+1} = \begin{cases} a_i, & \text{if } b_i = 0 \\ a_i \cdot c_i, & \text{if } b_i = 1 \end{cases}.$$

Finally $a_k = a^n$.

It turns out that the computing times of the three methods are mutually codominant. If $M(n)$ is the cost of multiplying two numbers of length n or less, then the time needed for computing a^n is proportional to $M(n \cdot L(a))$. Although fewer multiplications are needed in the binary methods, the integers to be multiplied are usually greater than in the conventional method. This remark no longer holds when we are doing modular arithmetic. In this case the binary methods are actually superior.

Let us now consider division of rational integers. For $a, b \in \mathbb{Z}$, $b \neq 0$, there exist unique integers $\text{quot}(a, b)$ and $\text{rem}(a, b)$ such that $a = \text{quot}(a, b) \cdot b + \text{rem}(a, b)$ and $0 \leq \text{rem}(a, b) < |b|$ if $a \geq 0$, $-|b| < \text{rem}(a, b) \leq 0$ if $a < 0$. In order to compute the quotient and remainder of two positive integers a and b (with $m = L_\beta(a) > L_\beta(b) = n$) we make a guess about the first quotient digit, say q_{k-1} . Having determined q_{k-1} , we replace a by $a - b \cdot q_{k-1} \cdot \beta^{k-1}$ and determine q_{k-2} in the same manner, using the new a . In determining q_j we have $0 \leq a < \beta^{j+1}$. Thus $q_j = \lfloor a/\beta^j \rfloor$ and

$$q_j^* = \lfloor \lfloor a/\beta^{n+j-1} \rfloor / \lfloor b\beta^j/\beta^{n+j-1} \rfloor \rfloor = \lfloor (a_{n+j}\beta + a_{n+j-1})/b_{n-1} \rfloor$$

should be an accurate approximation to q_j . We will have $q_j^* \geq \beta$ just in case $a_{n+j} \geq b_{n-1}$; in this case we may set $q_j^* = \beta - 1$. It turns out that $q_j \leq q_j^* \leq q_j + 2$ if $b_{n-1} \geq \beta/2$. This fact was first observed by D. A. Pope and M. L. Stein in [34]. G. E. Collins and D. R. Musser [12] later showed that the respective probabilities of $q_j^* = q_j + i$, $i = 0, 1, 2$, are approximately 0.67, 0.32, 0.01. The requirement $b_{n-1} \geq \beta/2$ can be met by normalizing b to $b' = b \cdot d$ and a to $a' = a \cdot d$ where $d = \lfloor \beta/(b_{n-1} + 1) \rfloor$. This does not change the value of the quotient q and $\text{rem}(a, b) = \text{rem}(a', b')/d$.

For describing the algorithm for division we assume that we already have an algorithm DQR for dividing an integer of β -length 2 by a β -digit. This could be achieved using the idea above. Most computers, however, have an instruction which divides a double precision integer by a single precision integer, producing a single precision quotient and a single precision remainder. Such an instruction essentially realizes DQR. Based on DQR an algorithm IDQR for dividing an integer by a digit can easily be programmed.

$$(q, r) \leftarrow \text{IQR}(a, b)$$

[integer quotient-remainder algorithm. a and b are integers, $L_\beta(a) \geq L_\beta(b) \geq 2$.
 $q = \text{quot}(a, b)$, $r = \text{rem}(a, b)$]

- (1) Compute the signs s and t of a and b , respectively, and the normalization factor $d = \beta/(|b_{n-1}| + 1)$, where $n = L_\beta(b)$.
 Normalize a and b : $a' \leftarrow \text{IPROD}(a, s \cdot d)$, $b' \leftarrow \text{IPROD}(b, t \cdot d)$.
- (2) For $j = 0$ to $m - n$ ($m = L_\beta(a)$) do steps (3)–(5).
- (3) Calculate $q^* \leftarrow \min(\beta - 1, (a'_{m-j-1} + \beta a'_{m-j})/b'_{n-1})$
 where we let $a'_m = 0$ if $L_\beta(a') = m$.
- (4) Adjust the quotient by reducing q^* by 1 as long as
 $\text{IDIF}((a'_{m-n-j}, \dots, a'_{m-j}), q^* b')$ is negative.
- (5) Set $q_{m-n-j} \leftarrow q^*$, $a' \leftarrow \text{IDIF}(a', q^* \cdot \beta^{m-n-j} \cdot b')$.
- (6) If $q_{m-n} = 0$ then set $q \leftarrow (q_0, \dots, q_{m-n-1})$.
 If $s \cdot t = -1$ then set $q \leftarrow \text{INEG}(q)$.
 $r \leftarrow a'/s \cdot d$. ■

It is clear from the above remarks that step (4) has to be done at most twice in each cycle. If we replace step (4) by

- (4') As long as $b'_{n-2} \cdot q^* > (a'_{m-j}\beta + a'_{m-j-1} - q^* b'_{n-1})\beta + a'_{m-j-2}$
 reduce q^* by 1.
 If $\text{IDIF}((a'_{m-n-j}, \dots, a'_{m-j}), q^* b')$ is negative reduce q^* by 1

we have to use IDIF only once. This correction has been suggested by D. E. Knuth in [23], Section 4.3.1.

The execution time of IQR is dominated by $L(b) \cdot (L(a) - L(b) + 1)$, which is the same as for computing $b \cdot \text{quot}(a, b)$.

Instead of computing a/b directly, we could first calculate b^{-1} and then multiply by a . In order to use this method effectively we must be able to approximate b^{-1} , i.e. the zero of the function $f(x) = 1/x - b$. This, however, can be achieved by Newton's method. If b_i is already an approximation to b^{-1} then $b_{i+1} = b_i \cdot (2 - b \cdot b_i)$ is a better one with $|b_{i+1} - b^{-1}| < \varepsilon^2$ if $\varepsilon = |b_i - b^{-1}|$. Thus we need a subalgorithm IRECIP, which takes as input an integer a and a positive β -integer n and computes a positive β -integer b such that $|b \cdot 2^{-n-1} - a^{-1} \cdot 2^m| \leq 2^{-n}$ and $b \leq 2^{n+2}$, where $m = L_2(a)$. IRECIP can now be used to obtain a fast quotient-remainder algorithm IQRN corresponding to the fast product algorithm IPRODK. To divide a by b we approximate b^{-1} with b' , using IRECIP, to $m - n + 3$ places,

where $m = L_2(a)$ and $n = L_2(b)$. We approximate a by a' to $m - n + 3$ places. Multiplying a' by b' and truncating, we obtain an approximation q' to a/b and hence to $q = \lfloor a/b \rfloor$. The following theorem (taken from [11]) shows that $q' = q$ or $q' = q + 1$, so $q = q' - 1$ just in case $a - b \cdot q' < 0$.

Theorem. *Let a and b be integers,*

$$\begin{aligned} m &= L_2(a), & n &= L_2(b), & k &= m - n + 1 \geq 1, \\ |b' \cdot 2^{-k-3} - b^{-1} \cdot 2^n| &\leq 2^{-k-2}, & a' &= \lfloor a/2^{n-3} \rfloor, \\ q^* &= \lfloor a' \cdot b'/2^{k+4} \rfloor, & q' &= \lfloor (q^* + 3)/4 \rfloor, & q &= \lfloor a/b \rfloor. \end{aligned}$$

Then $q' = q$ or $q' = q + 1$.

The computing time of IQRN is dominated by $M(L(q)) + M'(L(b), L(q))$ for $a \geq b$, where $q = \lfloor a/b \rfloor$.

Finally, let us just state that one could design a quotient-remainder algorithm with a computing time bound comparable with that for fast multiplication. Let $m = L_2(a)$, $n = L_2(b)$, $k = m - n + 1$. If either n or k is less than some critical value then IQR is applied. Otherwise, a is expressed in the radix β^n ,

$$a = \sum_{i=0}^{h-1} a_i \beta^{ni}, \quad h = \lceil m/n \rceil.$$

The β^n -digits q_i in $q = \sum_{i=0}^{h-1} q_i \beta^{ni}$ are computed, and q is computed from the q_i s by shifting and adding. The computing time for this algorithm is dominated by $M'(L(b), L(q))$ for $|a| \geq |b|$, where $q = \lfloor a/b \rfloor$.

Finally we want to compute greatest common divisors of integers. Let a and b be integers not both equal to zero. Then we let $\text{gcd}(a, b)$ be the greatest integer that evenly divides both a and b ; $\text{gcd}(0, 0)$ is defined to be 0. If we divide a by b , getting a quotient q and a remainder c , it is immediate from the equation $a = bq + c$ that a and b have the same common divisors as b and c . If $c > 0$ the process can be repeated with (b, c) in place of (a, b) . Continuing until a zero remainder is reached, we obtain the so-called Euclidean remainder sequence $a_1, a_2, \dots, a_r, a_{r+1}$ with $a_1 = a, a_2 = b, a_{r+1} = 0$, and a_r is the greatest common divisor of a and b . From these considerations we obtain immediately the classical Euclidean greatest common divisor algorithm IGCDE. The computing time of IGCDE is proportional to $n \cdot (m - k + 1)$, where $m = \max(L(a), L(b))$, $n = \min(L(a), L(b))$, $k = L(\text{gcd}(a, b))$, for $a \neq 0, b \neq 0$. The analysis of the computing time of IGCDE, depending only on the inputs a, b , is an interesting subject in its own right. Let us just mention a theorem of G. Lamé (1845), which exhibits the relation between Euclid's algorithm and the Fibonacci numbers:

Theorem. *For $r \geq 1$, let a and b be integers with $0 < b < a$ such that Euclid's algorithm applied to a and b requires exactly r division steps, and such that a is as small as possible satisfying these conditions. Then $a = F_{r+2}$ and $b = F_{r+1}$.*

As a consequence of the theorem of Lamé we get $\lceil \log_\phi(\sqrt{5}N) \rceil - 2$ as an upper bound for the number of division steps in IGCDE, where $N = \max(a, b)$ and

$\phi = (1 + \sqrt{5})/2$. G. E. Collins [11] gives another bound for the number of division steps:

Theorem. *Let $0 < b < a$, then the number of division steps in IGCDE for computing $\gcd(a, b)$ is bounded by $(\log_{\phi} 2) \cdot L(b) + 2$.*

For lack of space, we cannot go into more details, here. The reader may find an extensive treatment in [23], Section 4.5.3 and [11].

One can extend the Euclidean algorithm in the following way: in addition to computing $\gcd(a, b)$ we also compute integers u and v such that

$$a \cdot u + b \cdot v = \gcd(a, b).$$

$$(c, u, v) \leftarrow \text{IGCDEXE}(a, b)$$

[integer greatest common divisor algorithm, extended Euclidean. a and b are integers, $c = \gcd(a, b)$, $a \cdot u + b \cdot v = c$]

- (1) Initialize the vectors u and v :
 $(u_1, u_2, u_3) \leftarrow (\text{ISIGN}(a), 0, \text{IABS}(a))$,
 $(v_1, v_2, v_3) \leftarrow (0, \text{ISIGN}(b), \text{IABS}(b))$.
- (2) Compute remainder sequence and cosequences until zero remainder is reached: as long as $v_3 \neq 0$ do the following:
 $(q, c') \leftarrow \text{IQR}(u_3, v_3)$
 $(t_1, t_2, t_3) \leftarrow (u_1, u_2, u_3) - (v_1, v_2, v_3) \cdot q$
 $(u_1, u_2, u_3) \leftarrow (v_1, v_2, v_3)$, $(v_1, v_2, v_3) \leftarrow (t_1, t_2, t_3)$.
- (3) Return $c \leftarrow u_3$, $u \leftarrow u_1$, $v \leftarrow u_2$. ■

In algorithm IGCDE (and IGCDEXE) a lot of divisions of long integers have to be performed. Since this is quite time consuming, we want to investigate whether there is a faster way of computing the gcd of two integers.

D. H. Lehmer in 1938 [25] has devised an algorithm, which uses only the most significant digits of a and b in the Euclidean division process, as long as this is possible. We want to consider here an algorithm similar to Lehmer's. This method appears to have no significant advantage or disadvantage relative to Lehmer's for rational integer greatest common divisor calculation. However, it generalizes in a more effective manner to Gaussian integers, and it also has the important advantage that one can better analyze the resulting algorithms. The idea was improved by Schönhage and carried over to polynomials by Moenck [31] and Strassen [38].

Let a and b be integers, $m = L_2(a)$, $n = L_2(b)$ ($m \geq n$) and $h < n$. Let $a' = \lfloor a/2^h \rfloor$, $b' = \lfloor b/2^h \rfloor$. The quotient of a' divided by b' will be roughly the same as that of a divided by b . Let $a'_1, a'_2, \dots, a'_{j+1}$ and q'_1, \dots, q'_{j-1} be the Euclidean remainder and quotient sequences of a' and b' ; let u'_1, \dots, u'_{j+1} and v'_1, \dots, v'_{j+1} be the associated cosequences. If $q'_i = q_i$ for $1 \leq i < j$, then $u'_i = u_i$ and $v'_i = v_i$ for $1 \leq i \leq j+1$, so we can compute a_j and a_{j+1} using the formula $u_i \cdot a + v_i \cdot b = a_i$. Since a' and b' are smaller than a and b , a considerable amount of computation will be saved if h and j are sufficiently large. The problem is to determine the largest possible value of j for

which $q'_j = q_j$. It turns out that

$$\min(a'_{i+2}, a'_{i+1} - a'_{i+2}) > 4|v'_{i+2}|$$

is a sufficient condition for $q'_i = q_i$, provided that $q'_1 = q_1, \dots, q'_{i-1} = q_{i-1}$. The correctness proof – which can be found in [11] – is too lengthy to be presented here. The following algorithm uses the above ideas:

$$c \leftarrow \text{IGCDLKS}(a, b)$$

[integer greatest common divisor algorithm, Lehmer-Knuth-Schönhage. a and b are integers, $a > b > 0$, $c = \text{gcd}(a, b)$]

- (1) Set $a_1 \leftarrow \text{IABS}(a)$, $a_2 \leftarrow \text{IABS}(b)$.
- (2) Compute the remainder sequence:
as long as $a_2 \neq 0$ do steps (3)–(9).
- (3) If a_1 is a β -integer, return $c \leftarrow \text{IGCDE}(a_1, a_2)$.
- (4) Set $m \leftarrow L_2(a_1)$, $n \leftarrow L_2(a_2)$.
- (5) If $m - n$ is "too big" compute $a_4 = \lfloor a_1/a_2 \rfloor$ by IQR and set $a_3 \leftarrow a_2$.
Go to step (9).
- (6) Choose h as large as possible so that $n > h$ and the resulting
 $a' = \lfloor a_1/2^h \rfloor$, $b' = \lfloor a_2/2^h \rfloor$ are β -digits.
- (7) Set $(u_1, u_2, v_1, v_2) \leftarrow \text{DPCC}(a', b')$.
- (8) If $v_1 = 0$ then compute $a_4 = \lfloor a_1/a_2 \rfloor$ by IQR and set $a_3 \leftarrow a_2$.
Otherwise set $a_3 \leftarrow a_1 \cdot u_1 + a_2 \cdot v_1$, $a_4 \leftarrow a_1 \cdot u_2 + a_2 \cdot v_2$ using ISUM
and IPROD.
- (9) $a_1 \leftarrow a_3$, $a_2 \leftarrow a_4$.
- (10) Return $c \leftarrow a_1$. ■

In IGCDLKS a subalgorithm DPCC is used, which we describe in the sequel:

$$(u, u', v, v') \leftarrow \text{DPCC}(a_1, a_2)$$

[digit partial cosequence calculation. a_1 and a_2 are β -integers, $a_1 \geq a_2 > 0$.
 u, u', v, v' are the last cosequence elements of a_1 and a_2 which can be guaranteed to
correspond to correct quotient digits]

- (1) Set $a \leftarrow a_1$, $a' \leftarrow a_2$, $u \leftarrow 1$, $u' \leftarrow 0$, $v \leftarrow 0$, $v' \leftarrow 1$.
- (2) Repeat steps (3)–(5) until the termination criterion (4) is met.
- (3) $q \leftarrow a/a'$, $a'' \leftarrow a - q \cdot a'$,
 $u'' \leftarrow u - q \cdot u'$, $v'' \leftarrow v - q \cdot v'$,
 $v^* \leftarrow 4 \cdot \lfloor v'' \rfloor$.
- (4) If $a'' \leq v^*$ or $(a' - a'') \leq v^*$ then return the current values of u, u', v, v' .
- (5) $a \leftarrow a'$, $a' \leftarrow a''$, $u \leftarrow u'$, $u' \leftarrow u''$, $v \leftarrow v'$, $v' \leftarrow v''$. ■

The computing time of IGCDLKS is again proportional to $n \cdot (m - k + 1)$. IGCDLKS, however, will be about j^* times as fast as the ordinary Euclidean algorithm, where j^* is the average value of j , the number of Euclidean division steps which we can carry out only on the leading digits of a and b .

Algorithm IGCDLKS can also be extended to an algorithm that computes the cosequence elements in addition to the greatest common divisor ([11]).

Let us just note that there is a gcd-algorithm with computing time dominated by $m \cdot M(\min(n, m - k + 1))$, where $m = L(a)$, $n = L(b)$, $k = L(\gcd(a, b))$. This algorithm makes appropriate use of the fast multiplication and division algorithms. The method is derived from the work of R. Moenck [30], which in turn is derived from the earlier work of D. E. Knuth [24] and A. Schönage [35].

The Rational Numbers

Now that we have algorithms for performing arithmetic operations (including gcd calculation) in \mathbb{Z} it is relatively simple to develop algorithms for arithmetic operations on arbitrary rational numbers \mathbb{Q} , the fraction field or field of quotients of the integral domain \mathbb{Z} . Van der Waerden, in [39], §13 describes how to embed an integral domain I into its field of quotients $Q(I)$.

The problem of computing canonical forms for the elements of $Q(I)$ arises at this point. An investigation and solution of this problem may be found in the chapter on algebraic simplification in this volume.

Let $>$ be a linear order on I such that $a + b > 0$ and $a \cdot b > 0$ for $a, b \in I$, $a > 0$ and $b > 0$. I is then said to be an ordered domain (relative to $>$). A linear order $>'$ on $Q(I)$ can be defined by the condition $(a, b) >' (c, d)$ in case $(bd > 0$ and $ad > bc)$ or $(bd < 0$ and $ad < bc)$. $Q(I)$ is an ordered field relative to $>'$.

For describing algorithms for addition and multiplication in the field of the rational numbers $\mathbb{Q} = Q(\mathbb{Z})$ one need only to apply the definitions

$$(*) \quad \frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd} \quad \text{and} \quad \frac{a}{b} \cdot \frac{c}{d} = \frac{ac}{bd}$$

and reduce the result to lowest terms by a canonical simplifier. P. Henrici, in [17], however, introduced less obvious algorithms which are in general more efficient if the inputs are reduced. His method can be stated in the general context of a unique factorization domain I with effective gcd. Let $r_1, r_2, s_1, s_2 \in I$, and $\gcd(r_1, r_2) = \gcd(s_1, s_2) = 1$. If we let $d = \gcd(r_2, s_2)$, $r'_2 = r_2/d$, $s'_2 = s_2/d$ then

$$\gcd(r_1 s'_2 + s_1 r'_2, r_2 s'_2) = \gcd(r_1 s'_2 + s_1 r'_2, d).$$

This suggests an algorithm for adding two fractions:

$$t \leftarrow \text{RNSUM}(r, s)$$

[Sum of rational numbers. $r = (r_1, r_2)$ and $s = (s_1, s_2)$ are rational numbers in canonical form. $t = r + s$, $t = (t_1, t_2)$ is in canonical form.]

- (1) $[r = 0$ or $s = 0]$ If $r = 0$ then set $t \leftarrow s$ and return.
If $s = 0$ then set $t \leftarrow r$ and return.

- (2) $d \leftarrow \text{IGCD}(r_2, s_2)$.
- (3) [$d = 1$] If $d = 1$ then set $t_1 \leftarrow r_1 \cdot s_2 + r_2 \cdot s_1$,
 $t_2 \leftarrow r_2 \cdot s_2$ and return.
- (4) [$d \neq 1$] $r'_2 \leftarrow r_2/d, s'_2 \leftarrow s_2/d$.
 $t'_1 \leftarrow r_1 \cdot s'_2 + s_1 \cdot r'_2$.
 $t'_2 \leftarrow r_2 \cdot s'_2$.
 If $t'_1 = 0$ then set $t_1 \leftarrow 0, t_2 \leftarrow 1$ and return.
 $e \leftarrow \text{IGCD}(t'_1, d)$.
 If $e = 1$ then set $t_1 \leftarrow t'_1, t_2 \leftarrow t'_2$ and return.
 $t_1 \leftarrow t'_1/e, t_2 \leftarrow t'_2/e$.
 Return. ■

The computing time for RNSUM is dominated by $m \cdot n^{\alpha-1} \cdot L(m)$ where $m = \max(L(r), L(s))$, $n = \min(L(r), L(s))$, and $\alpha = \log_2 3$ for Karatsuba multiplication and $\alpha = 2$ in the classical case. Here $L(r)$ is defined as $\max(L(r_1), L(r_2))$. The improvement in computing time by the use of RNSUM instead of an algorithm based on (*) is a factor of about 3 for $d = 1$ and 1.52 for $d \neq 1$ (see [11]). But for other choices of the domain I (compare the section on rational functions) the improvement may be much more significant.

One could write a subtraction algorithm analogous to RNSUM, or simply compute $r - s$ as $r + (-s)$ where it is assumed that an algorithm RNNEG for negation is given (using INEG).

For multiplication of fractions we use the following fact: if $r_1, r_2, s_1, s_2 \in I$,

$$\gcd(r_1, r_2) = \gcd(s_1, s_2) = 1, \quad d_1 = \gcd(r_1, s_2), \quad d_2 = \gcd(s_1, r_2),$$

$$r'_1 = r_1/d_1, \quad r'_2 = r_2/d_2, \quad s'_1 = s_1/d_2, \quad s'_2 = s_2/d_1$$

then $\gcd(r'_1 s'_1, r'_2 s'_2) = 1$. From this fact we have the following multiplication algorithm:

- $t \leftarrow \text{RNPROD}(r, s)$
 [Product of rational numbers. $r = (r_1, r_2)$ and $s = (s_1, s_2)$ are rational numbers in canonical form. $t = r \cdot s, t = (t_1, t_2)$ is in canonical form.]
- (1) [$r = 0$ or $s = 0$] If $r = 0$ or $s = 0$ then set $t_1 \leftarrow 0, t_2 \leftarrow 1$ and return.
- (2) $d_1 \leftarrow \text{IGCD}(r_1, s_2), d_2 \leftarrow \text{IGCD}(s_1, r_2)$.
- (3) [compute quotients]
 If $d_1 = 1$ then $r'_1 \leftarrow r_1, s'_2 \leftarrow s_2$
 else $r'_1 \leftarrow r_1/d_1, s'_2 \leftarrow s_2/d_1$.
 If $d_2 = 1$ then $s'_1 \leftarrow s_1, r'_2 \leftarrow r_2$
 else $s'_1 \leftarrow s_1/d_2, r'_2 \leftarrow r_2/d_2$.
- (4) [compute products] $t_1 \leftarrow r'_1 \cdot s'_1, t_2 \leftarrow r'_2 \cdot s'_2$.
 Return. ■

The computing time t_{RNPROD} is dominated by $m \cdot n^{\alpha-1} \cdot L(m)$.

The inverse r^{-1} of the non-zero fraction $r = (r_1, r_2)$ is (r_2, r_1) if r_1 is in the ample set A and (ur_2, ur_1) otherwise, where $ur_1 = \gcd(r_1, 0)$. The quotient of two rational numbers r, s can readily be computed as $r \cdot s^{-1}$.

It is clear how to design an algorithm for comparing two rational numbers r and s : if the signs of r_1 and s_1 are different, then we can immediately decide which one is the greater. Otherwise we have to compute and compare the integers r_1s_2 and s_1r_2 .

Certain best approximative representations of rational numbers, the so-called "fixed slash" and "floating slash" representations, have been suggested by D. W. Matula [27] and have been further developed in [28, 29]. These papers also address the issue of reconstructing fractions from given approximations.

Modular Arithmetic

When we are doing arithmetic (especially a lot of multiplications and only a few divisions and comparisons, see [23], p. 269, 275) on large integers, it is often preferable to work instead modulo several moduli m_1, \dots, m_k . So let us just review the basic facts about modular arithmetic (see also the chapter on computing by homomorphic images).

Arithmetic modulo a positive integer m is best understood as working in the residue class ring of \mathbb{Z} modulo the ideal generated by m , i.e. in $\mathbb{Z}/(m)$ (such ideals are the only ones in \mathbb{Z} , since it is well known that \mathbb{Z} is a principal ideal domain. See, for example, [39], §17). The only residue classes in this case are $\bar{0}, \bar{1}, \dots, \bar{m-1}$ (\bar{a} denotes the residue class of a , i.e. $\bar{a} = \{a + bm \mid b \in \mathbb{Z}\}$). Addition and multiplication are defined by

$$\bar{a} + \bar{b} = \overline{a + b}, \quad \bar{a} \cdot \bar{b} = \overline{a \cdot b}.$$

The additive identity is $\bar{0}$, the multiplicative identity is $\bar{1}$.

There are two natural choices of a set of representatives for $\mathbb{Z}/(m)$, namely $\{0, 1, \dots, m-1\}$ and $\{a \mid -m/2 < a \leq m/2\}$. If m is odd, the second set is symmetric with respect to zero, while if m is even it contains $m/2$ but not $-m/2$. Each of these choices has certain advantages and disadvantages; we choose the first and denote it by \mathbb{Z}_m . By H_m we shall denote the canonical simplifier associated with \mathbb{Z}_m .

\mathbb{Z}_m is itself a ring isomorphic with $\mathbb{Z}/(m)$ in which addition and multiplication are defined by

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

(of course we also have $a -_m b = H_m(a - b)$).

An algorithm MIHOM that realizes the modular integer homomorphism H_m takes m and a as input and essentially applies one of the division algorithms discussed in the section on integer arithmetic for computing $b = \text{rem}(a, m)$.

MISUM, MIDIF, and MIPROD for computing $a +_m b$, $a -_m b$, and $a \cdot_m b$ can be based directly on (*), using MIHOM as a subalgorithm. This approach, however, is rather inefficient for MISUM and MIDIF, since the quotient computed by the

application of MIHOM can only be zero or one. So it is preferable to subtract the modulus m from the result until the desired representative is reached.

For the computing times we have the relations $t_{\text{MISUM}}^+ \sim t_{\text{MISUM}}^* \sim t_{\text{MIDIF}}^+ \sim t_{\text{MIDIF}}^* \sim L(m)$. t_{MIPROD} is dominated by $L(m)^2$.

Modular exponentiation has many applications in number theory. For integer exponentiation all three methods (repeated multiplication, left-to-right and right-to-left binary exponentiation) have computing times proportional to each other. Not so for modular exponentiation, because the modulus m is a bound for all integers which are multiplied. The time to compute $a^n \pmod{m}$ by repeated multiplication will be proportional to $n \cdot L(m)^2$ as opposed to $L(n) \cdot L(m)^2$ for any of the binary methods.

Division of a by b in a commutative ring R is defined just in case there is a unique c such that $a = b \cdot c$, and then c is the quotient a/b . In the case of $R = \mathbb{Z}_m$, b is a zero divisor (and therefore c is not uniquely determined) if $\gcd(m, b) \neq 1$. On the other hand if $\gcd(m, b) = 1$, then b is a unit and we can compute b^{-1} (thus the quotient $c = a/b$ is $a \cdot b^{-1}$). If $\gcd(m, b) = 1$ then there exist u, v such that $m \cdot u + b \cdot v = 1$, thus $b^{-1} = H_m(v)$. So an algorithm MIDIV for dividing a by b modulo m will essentially compute $d = \gcd(m, b)$ together with the cofactors u and v (e.g. by algorithm IGCDEXE). If $d \neq 1$ then a is not divisible by b , otherwise $a/b = a \cdot H_m(v)$. The computing time of MIDIV is dominated by $L(m)^2 \cdot L(L(m))$.

For the case in which m is a prime p an alternative for computing the inverse b^{-1} would be the use of Fermat's theorem (see [39], §43), according to which $b^{-1} = b^{p-1}$ in \mathbb{Z}_p . It follows from the time bound for modular exponentiation that the computing time of an algorithm using Fermat's theorem would be dominated by $L(m)^2 \cdot L(m)$ as opposed to the much better bound $L(m)^2 \cdot L(L(m))$ for an algorithm computing $\gcd(p, b)$. The following table compares both methods on a DEC10. The length of the prime p is close to the word size. The times are given in milliseconds.

Fermat	Euclid
3.795	0.042
3.787	0.053
3.797	0.050
3.800	0.058
3.773	0.034
3.786	0.052
3.782	0.043
3.783	0.029
3.789	0.024
3.768	0.042

The usual situation for the application of modular arithmetic is that one wants to replace a computation in \mathbb{Z} (with possibly huge intermediate results) by a number of computations with respect to relatively prime moduli. The important question now is how we can convert fast into and out of the modular representation. Conversion

from integer to modular representation is handled by the homomorphism H_m (computed by algorithm MIHOM). Next we shall consider algorithm MICRA1 for converting back from modular representation to the rational integers \mathbb{Z} . The mathematical background for the solution of this problem is provided by the Chinese remainder theorem which we need here for the Euclidean domain \mathbb{Z} . See the chapter on computing by homomorphic images in this volume, where in Section 2.2 the algorithm is presented.

The computing time t_{MICRA1} of an algorithm MICRA1 (modular integer Chinese remainder algorithm 1) based on these ideas is dominated by $L(m'_1) \cdot L(m'_2)^{\alpha-1} \cdot L(L(m'_2))$ where $m'_1 = \max(m_1, m_2)$ and $m'_2 = \min(m_1, m_2)$.

In order to obtain a "fast" Chinese remainder algorithm MICRA2, one must use a recursive algorithm, which divides its problem into two roughly equal subproblems. This is done by dividing the list of moduli m_1, \dots, m_k into two sublists $m_1, \dots, m_{\lfloor k/2 \rfloor}$ and $m_{\lfloor k/2 \rfloor + 1}, \dots, m_k$. MICRA2 is applied to both sublists (if their lengths are greater than 2, otherwise MICRA1 is applied) and the solution of the given problem is computed from the solutions of the two subproblems by the algorithm MICRA1. (Note that if $a \equiv b_1 \pmod{m_1 \cdots m_{\lfloor k/2 \rfloor}}$ and $a \equiv b_2 \pmod{m_{\lfloor k/2 \rfloor + 1} \cdots m_k}$ and $b_1 \equiv a_i \pmod{m_i}$ for $1 \leq i \leq \lfloor k/2 \rfloor$ and $b_2 \equiv a_i \pmod{m_i}$ for $\lfloor k/2 \rfloor + 1 \leq i \leq k$ then $a \equiv a_i \pmod{m_i}$ for $1 \leq i \leq k$.)

The computing time of MICRA2 is dominated by $L(m)^\alpha \cdot L(L(m))^2$ where $m = m_1 \cdots m_k$.

When one is doing modular arithmetic, it is essential to be able to generate a large supply of prime numbers. Actually we only need relatively prime moduli. Knuth (in [23], p. 272) considers moduli of the form $2^e - 1$, for which relative primality can be checked by the simple rule $\gcd(2^e - 1, 2^f - 1) = 2^{\gcd(e, f)} - 1$. Whenever possible one is interested in single-precision moduli, because that makes the computation more efficient. If one used double-precision moduli, only half as many would be required for the solution of any given problem, but the calculations in \mathbb{Z}_m for each modulus m would take more than twice as long.

An algorithm SPGEN (small prime generator) for generating prime numbers p in the range between m and $m + 4k$ for given m and k could be based on the following ideas: provide an array A of length k as workspace, $A[i]$ corresponding to the integer $n_i = m_1 + 4(i - 1)$ for $1 \leq i \leq k$ where $m_1 = m + 3 - \text{rem}(m, 4)$. Now successively consider all the possible divisors d of the integers n_i , $1 \leq i \leq k$, and mark each element of A which corresponds to a proper multiple of d . The integers whose corresponding fields in A remain unmarked by this process are the prime numbers in the given range. It suffices to consider as the possible divisors only the numbers $d = 3$ and $d = 6n \mp 1$, $n \geq 1$, since this set of numbers includes all odd primes. An upper bound for the divisors to be considered is $\sqrt{m_2}$, where $m_2 = m_1 + 4(k - 1)$. $d \leq \sqrt{m_2}$ can be decided by $d \leq \lfloor m_2/d \rfloor$.

In [11] $40\sqrt{n} + 250(\ln n)^2 + 1000 \ln n + 1500$ microseconds is given as an estimate of the execution time of SPGEN for generating 100 primes. The following table gives approximate values, in seconds, of this computing time function. In [11]

it is also observed that for a Telefunken TR-440 computer the times of this table agree closely with observed times.

n	Time for computing 100 primes
2^{20}	0.1
2^{25}	0.3
2^{30}	1.4
2^{35}	7.6
2^{40}	42
2^{45}	238

Thus SPGEN is feasible for $n \leq 2^{48}$ but not for generating larger primes, where maybe probabilistic algorithms are acceptable ([23], Section 4.5.4).

Gaussian Integer Arithmetic

A Gaussian integer a is a number of the form $a_1 + a_2i$ (a_1, a_2 rational integers, $i = \sqrt{-1}$). The components a_1 and a_2 of a are respectively called the real and imaginary part of a . The domain \mathbb{G} of Gaussian integers constitutes a subring of the field of complex numbers. In fact, \mathbb{G} is an integral domain. Addition and subtraction are defined component-wise. The ordinary algorithm for multiplication

$$(a_1 + a_2i) \cdot (b_1 + b_2i) = (a_1b_1 - a_2b_2) + (a_1b_2 + a_2b_1)i$$

can be speeded-up by Karatsuba's method (see the section on integer arithmetic), i.e. one multiplication can be traded for three additions/subtractions (see [7]).

It is clear that

$$t_{\text{GIADD}}(a, b) \leq L(a) + L(b),$$

$$t_{\text{GISUB}}(a, b) \leq L(a) + L(b),$$

$$t_{\text{GIMULT}}(a, b) \leq M'(L(a), L(b)).$$

(t_{GIADD} , t_{GISUB} , t_{GIMULT} times to add, subtract and multiply Gaussian integers; $L(a_1 + a_2i) = L(a_1) + L(a_2)$ length of a Gaussian integer $a_1 + a_2i$, M and M' are the dominating functions for multiplication of rational integers, see the section on integer arithmetic.)

For $a, b \in \mathbb{G}$, " b divides a " may be decided by forming the quotient a/b according to the well-known formula

(DIV) $a/b = (a \cdot \bar{b}) \cdot (1/N(b))$ or, equivalently,

$$(a_1 + a_2i)/(b_1 + b_2i) = (a_1b_1 + a_2b_2)/(b_1^2 + b_2^2) \\ + ((a_2b_1 - a_1b_2)/(b_1^2 + b_2^2))i$$

$$(\bar{b} = \overline{b_1 + b_2i} = b_1 - b_2i \text{ the conjugate of } b,$$

$$N(b) = N(b_1 + b_2i) := b \cdot \bar{b} = b_1^2 + b_2^2 \text{ the norm of } b),$$

and testing whether the real and imaginary part of a/b are both in \mathbb{G} .

There is a method known to save 2 multiplications/divisions which is not known to be optimal (Alt van Leeuwen, private communication).

The units of \mathbb{G} are $1 = i^0$, $i = i^1$, $-1 = i^2$ and $-i = i^3$. $S(a) := i^{5-j} \cdot a$ (j number of quadrant in which a lies) defines a canonical simplifier S for the equivalence relation " a and a' are associates" (compare the chapter on simplification in this volume). \mathbb{G} is a Euclidean domain: the norm N is a function from \mathbb{G} into \mathbb{N}_0 satisfying

(EUCL1) for all $a, b \in \mathbb{G}$: $N(a \cdot b) = N(a) \cdot N(b)$ and

(EUCL2) for all $a, b \in \mathbb{G}$, $b \neq 0$ there exist $q, r \in \mathbb{G}$ such that $a = q \cdot b + r$,
 $r = 0$ or $N(r) < N(b)$.

By (EUCL2), q and r are not uniquely determined. In fact, exactly all those $q \in \mathbb{G}$ that satisfy $N(q - a/b) < 1$ together with $r := a - q \cdot b \in \mathbb{G}$ are suitable in (EUCL2). In particular $q := \{a/b\}$ is such that $N(q - a/b) < 1$.

(For real numbers c_1, c_2 , $\{c_1 + c_2 i\}$ is the nearest Gaussian integer to $c_1 + c_2 i$:

$$\{c_1 + c_2 i\} := \{c_1\} + \{c_2\}i,$$

where

$$\{c\} := [c - 1/2] \text{ when } c \text{ is real.})$$

Hence, $q := \{a/b\}$ (the nearest quotient of a and b) and $r := a - q \cdot b$ (the least remainder of a and b) is a suitable definition of a Euclidean division procedure in \mathbb{G} , on which a Euclidean algorithm for the computation of the greatest common divisor may be based in the usual way ([39]). If the computation of a/b in $q := \{a/b\}$ is based on formula (DIV), i.e. $q := \{c_1/d\} + \{c_2/d\}i$, where $c_1 + c_2 i = a \cdot \bar{b}$ and $d := N(b)$, and Newton's method is used for computing $\{c_1/d\}$ and $\{c_2/d\}$, then it can be shown, [7], that

$$t_{\text{GINQR}}(a, b) \leq M(L(b)) + M'(L(b), L(q))$$

(where t_{GINQR} is the time to compute q and r according to this method). The term $M(L(b))$ in this formula reflects the time to compute the norm of b . In case $L(q) < L(b)$, it cannot be absorbed by $M'(L(b), L(q))$. B. F. Caviness and G. E. Collins [7] have presented an alternative method for division in \mathbb{G} that avoids computing $N(b)$ in a large percentage of cases when $N(q) < N(b)$. The time t_{GINQR} to compute q and r according to their method behaves as follows

$$t_{\text{GINQR}}(a, b) \leq L(\max(L(b), L(q))) \cdot M(\max(L(b), L(q))),$$

$$t_{\text{GINQR}}^*(a, b) \leq M'(\max(L(b), L(q))), \min(L(b), L(q)),$$

i.e., roughly, in the average computing time the term $M(L(b))$ disappeared. The main idea of this method is trying to obtain $q := \{a/b\}$ from $q^* := \{a^*/b^*\}$ where a^*, b^* are approximations to a, b having only a few more bits than q . By a detailed case analysis it can be seen that the number $L^*(q)$ of bits in q can be estimated in advance:

$$q = 0, \quad \text{if} \quad L^*(a) - L^*(b) \leq -3,$$

and

$$L^*(a) - L^*(b) - 2 \leq L^*(q) \leq L^*(a) - L^*(b) + 3, \quad \text{otherwise.}$$

(Exact definition of L^* :

$$L^*(a_1 + a_2i) := \max(L^*(a_1), L^*(a_2)), \text{ for } a_1 + a_2i \in \mathbb{G},$$

$$L^*(a) := \begin{cases} \lceil \log_2 |a| \rceil + 1, & \text{if } a \neq 0 \\ 0, & \text{otherwise.} \end{cases}$$

In fact, it turns out that in case $L^*(a) - L^*(b) > -3$ and $k \geq 6$

(APPDIV) $q = \{q^*/2^k\}$, with probability greater than $1 - 2^{-k+2}$
 (where $q^* = \{a^*/b^*\}$,
 $a^* = \lceil a/2^{m-2h} \rceil$, $b^* = \lceil b/2^{n-h} \rceil$,
 $m = L^*(a)$, $n = L^*(b)$, $h = m - n + k$).

For computing $q^* = \{a^*/b^*\}$ the ordinary division method is applied. The following equivalence is an effective test for determining whether q is indeed equal to $\{q^*/2^k\}$:

$$q = \{q^*/2^k\} \Leftrightarrow |d_i| \neq 2^{k-1} \quad \text{for } i = 1, 2$$

(where $d_i := q_i^* - c_i \cdot 2^k$, $c_i := \{q_i^*/2^k\}$).

Using this test (APPDIV) may be used iteratively for $k = k_0 (\geq 6), 2k_0, 4k_0, 8k_0, \dots$ until the test reveals that $q = \{q^*/2^k\}$ or $(m - n) + k \geq n$, in which case $q = \{a/b\}$ is completed by the ordinary method.

When Euclid's algorithm for computing the greatest common divisor of $a, b \in \mathbb{G}$ is based on the above improved division algorithm a speed-up ratio of approximately two is reported in [7]. A Lehmer-type algorithm for Gaussian integer gcd [6] yields a further speed-up of two. The best Gaussian integer gcd algorithm known so far is in [7], p. 41. It again improves the computation time by a factor of two as has been shown by extensive test computations, [7], p. 42. It uses the idea of the original Lehmer algorithm (approximation of the quotient sequence by using the high-order, single-precision parts of the inputs), but avoids complex interval arithmetic. Very roughly for $a, b \in \mathbb{G}$ this algorithm proceeds as follows:

$$c \leftarrow \text{GIGCD}(a, b)$$

[Gaussian integer greatest common divisor. $a, b \in \mathbb{G}$, $c = \text{gcd}(a, b)$].

- (1) If a and b are single-precision or one of them is zero then skip (2).
- (2) If $\{a/b\}$ may be multiple-precision
 [this can be effectively checked by considering the lengths of a and b]
 then
 $(a, b) \leftarrow (b, \text{rem}(a, b))$
 else
 $(a^*, b^*) \leftarrow$ high-order single-precision parts of a and b .
 Apply extended single-precision Euclidean algorithm to (a^*, b^*) until a certain termination criterion is satisfied. This yields u, u', v, v' as last elements of the first and second consequence.

$(a, b) \leftarrow (u \cdot a + v \cdot b, u' \cdot a + v' \cdot b)$
Go to step 1.

- (3) $c \leftarrow \text{GCD}(a, b)$ using a single-precision gcd algorithm
(or $c \leftarrow S(a)$ if, for instance, $b = 0$). ■

A suitable termination criterion in the loop in step 2 is

$$\frac{1}{6} \max(|a_1|, |a_2|) < |u_1| + |u_2| + |v_1| + |v_2|,$$

where $a_1 + a_2i, u_1 + u_2i, v_1 + v_2i$ are respectively the last element in the remainder sequence and first and second consequences starting from a^*, b^* . Normally, in step 2, the else-branch will be chosen. Essentially, this is the reason why the algorithm achieves a speed-up.

Polynomial Arithmetic

In this section we consider polynomials over a ring R in finitely many variables x_1, \dots, x_v . We get univariate polynomials for $v = 1$, multivariate polynomials for $v > 1$, and elements of the ground ring R for $v = 0$. Since $R[x_1, \dots, x_v]$ is isomorphic with $(\dots(R[x_1])\dots)[x_v]$ a polynomial $p \in R[x_1, \dots, x_v]$ can be considered as a polynomial in x_v with coefficients in the ring $R[x_1, \dots, x_{v-1}]$.

For

$$p(x_1, \dots, x_v) = \sum_{i=1}^m p_i(x_1, \dots, x_{v-1})x_v^{d_i},$$

where $d_1 > \dots > d_m$ and $p_m \neq 0$, the degree of p (in the main variable x_v), written $\text{deg}(p)$, is d_1 . The degree of the zero polynomial is zero. For $1 \leq j \leq v$ the degree of p in x_j is defined as $\text{deg}_j(p) = d_1$ for $j = v$ and $\text{deg}_j(p) = \max_{1 \leq i \leq m} \text{deg}_j(p_i)$ for $j < v$. $p_1(x_1, \dots, x_{v-1})$ is the leading coefficient of p , written $\text{ldcf}(p)$. By convention $\text{ldcf}(0) = 0$. The trailing coefficient of p is $\text{ldcf}(x_v^n \cdot p(1/x_v))$. The leading term of p is the polynomial

$$\text{ldt}(p) = p_1(x_1, \dots, x_{v-1})x_v^{d_1}$$

and the reductum of p is the polynomial

$$\text{red}(p) = \sum_{i=2}^m p_i(x_1, \dots, x_{v-1})x_v^{d_i}.$$

By convention, $\text{ldt}(0) = \text{red}(0) = 0$. The leading base coefficient of p belongs to the ring R and is defined as $\text{lbcf}(p) = \text{ldcf}(p)$ if $v = 1$ and $\text{lbcf}(p) = \text{lbcf}(\text{ldcf}(p))$ if $v > 1$.

There are four major forms of representation for polynomials in $R[x_1, \dots, x_v]$, depending on whether we represent the polynomials in a recursive or a distributive way and whether the polynomials are dense or sparse. In computer algebra multivariate polynomials are usually sparse.

Let $p \neq 0$ be a polynomial in $R[x_1, \dots, x_v]$ in which the power products $x_1^{e_{1,1}} \cdots x_v^{e_{1,v}}, \dots, x_1^{e_{m,1}} \cdots x_v^{e_{m,v}}$ have the respective coefficients a_1, \dots, a_m and all the other coefficients are zero then the sparse distributive representation of p is the list $(e_1, a_1, \dots, e_m, a_m)$ where $e_i = (e_{i,1}, \dots, e_{i,v})$ for $1 \leq i \leq m$. For most applications one wants the e_i to be ordered, e.g. according to the inverse lexicographical ordering. This representation makes nearly no distinction between the variables. We might, however, view the polynomial p as a polynomial in only one variable, say x_v , with coefficients in the ring $R[x_1, \dots, x_{v-1}]$ (the same applies for the coefficients of p until the ground ring R is reached), since $R[x_1, \dots, x_v]$ is isomorphic with $(\cdots(R[x_1])\cdots)[x_v]$. Thus if

$$p(x_1, \dots, x_v) = \sum_{i=1}^k p_i(x_1, \dots, x_{v-1}) \cdot x_v^{e_i},$$

where $e_1 > \cdots > e_k$ and $p_i \neq 0$ for $1 \leq i \leq k$, the sparse recursive representation of p is the list $(e_1, \pi_1, \dots, e_k, \pi_k)$ where π_i is the sparse recursive representation of p_i for $1 \leq i \leq k$ (assuming that a representation for the coefficient ring R is available).

Let us just briefly mention possible corresponding dense representations. If a linear ordering is defined on the exponent vectors such that for each exponent vector e there are only finitely many other exponent vectors f with $f < e$ (e.g. the ordering which orders the exponent vectors according to their degree and inverse lexicographically within the same degree), $e_0 < e_1 < \cdots$, and

$$p(x_1, \dots, x_v) = \sum_{i=0}^h a_i x^{e_i},$$

where $a_h \neq 0$ and x^{e_i} stands for $x_1^{e_{i,1}} \cdots x_v^{e_{i,v}}$, then the dense distributive representation of p may be the list (e_h, a_h, \dots, a_0) .

If

$$p(x_1, \dots, x_v) = \sum_{i=0}^j q_i(x_1, \dots, x_{v-1}) \cdot x_v^i \quad \text{and} \quad q_j \neq 0$$

then the dense recursive representation of p may be the list (j, q_j^*, \dots, q_0^*) , where q_i^* is the dense recursive representation of q_i for $1 \leq i \leq j$.

Addition and subtraction of polynomials is done by adding or subtracting the coefficients of like powers. One only has to be careful to delete "leading zeros" in the resulting polynomial. Polynomial arithmetic is similar to arithmetic on large integers, indeed simpler since there are no carries. The computing time for adding or subtracting two polynomials p, q by this method is dominated by

$$(L(p_0) + L(q_0)) \cdot \prod_{i=1}^v (\delta_i(p) + \delta_i(q) + 1),$$

where p_0 denotes the biggest coefficient in p and $\delta_i(p)$ is the degree of p in x_i .

The classical method for multiplying polynomials uses the formula

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

where

$$p(x) = \sum_{i=0}^m p_i x^i, \quad q(x) = \sum_{j=0}^n q_j x^j.$$

The computing time required by this classical algorithm applied to sparse polynomials is proportional to t^3 , where t is the maximum number of terms in p and q . It can, however, be reduced to $t^2 \cdot \log_2 t$ (see [20, 21]) by the use of better sorting algorithms.

For the case of dense polynomials "fast" algorithms for multiplication have been developed. The computing time t_{MPC} of a recursive version of the classical method for multiplying two dense polynomials of degree n in each of v variables (a polynomial p in v variables is dense if every power product $x_1^{d_1} \cdots x_v^{d_v}$ has a non-zero coefficient in p for $d_1 + \cdots + d_v$ less or equal some degree d) is

$$t_{\text{MPC}}(n, v) = (n + 1)^2 \cdot t_{\text{MPC}}(n, v - 1) + n^2 \cdot (2n + 1)^{v-1},$$

where the first term comes from multiplying the coefficients and the second term from the required additions. Thus $t_{\text{MPC}}(n, v)$ is proportional to n^{2v} (more precisely, the time for multiplying two polynomials p, q is dominated by

$$L(p_0) \cdot L(q_0) \cdot \prod_{i=1}^v (\delta_i(p) + 1)(\delta_i(q) + 1)).$$

A first fast algorithm is based on the Karatsuba method: in order to multiply two polynomials p and q of degree n , p and q are represented as

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

Then

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

The computing time $t_{\text{MPK}}(n, v)$ for multiplying two dense polynomials of degree n in v variables by an algorithm based on the Karatsuba method is codominant with $n^{v \cdot \log_2 3}$.

A second fast algorithm is based on the fast Fourier transform. For this purpose we consider only polynomials with coefficients in a finite field \mathbb{Z}_p . This restriction can be removed by the use of the Chinese remainder theorem.

For a detailed description of the fast Fourier transform we refer to [33]. A precise description of an implementation is given in [3].

The computing time $t_{\text{MPF}}(n, v)$ for multiplying two polynomials of degree n in each of v variables by an algorithm using the FFT is proportional to $v \cdot n^v \cdot \log_2 n$.

For further details on fast multiplication algorithms for dense polynomials see [32]. The trade-off points for all these "fast" algorithms are rather high. In [32] R. T. Moenck presents a hybrid mixed basis FFT algorithm that achieves a trade-off point at degree 25 for the univariate case.

When we are given two polynomials $a(x), b(x)$ over a field, with $b(x) \neq 0$, division of a by b is possible, i.e. we can obtain a unique quotient q and remainder r such that

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

The classical algorithm for this problem is given in [23], p. 402.

In many applications, however, (e.g. gcd computation of polynomials in $\mathbb{Z}[x]$) the underlying ring of coefficients is not a field and therefore the classical algorithm does not apply. So let I be a unique factorization domain, ufd, (e.g. $\mathbb{Z}, \mathbb{Z}[x]$ are ufds and $I[x]$ is a ufd if I is a ufd). For polynomials $a(x), b(x) \neq 0$ over I there exists a unique pseudoquotient $q(x) = \text{pquo}(a(x), b(x))$ and pseudoremainder $r(x) = \text{prem}(a(x), b(x))$ such that

$$\text{lcf}(b)^{\delta+1} \cdot a(x) = q(x) \cdot b(x) + r(x) \quad \text{and} \quad \deg(r) < \deg(b),$$

where $\delta = \deg(a) - \deg(b)$. An algorithm for pseudodivision is given in [23], p. 408.

Now assume that we are given polynomials a and $b \neq 0$ in $I[x]$ for some integral domain I . We want to compute the quotient $c = a/b$, if it exists. This is the problem of trial division. An algorithm for trial division is

$$c \leftarrow \text{PTDIV}(a, b)$$

[polynomial trial division. a and b are polynomials over some integral domain. If $b|a$ then $c = a/b$, otherwise an error is reported]

- (1) Set $c \leftarrow 0$.
- (2) If $a = 0$ then return c .
- (3) If $m < n$, where $m = \deg(a)$ and $n = \deg(b)$, or $\text{lcf}(b) \nmid \text{lcf}(a)$ then b does not divide a and an error is reported.
- (4) Otherwise, $c_{m-n} \leftarrow \text{lcf}(a)/\text{lcf}(b)$ is the coefficient of x^{m-n} in the quotient c , if it exists.
- (5) $a \leftarrow a - c_{m-n}x^{m-n} \cdot b$. Go to step (2). ■

In PTDIV we assume a trial division algorithm for I . Such an algorithm exists for $I = \mathbb{Z}$ and therefore we get trial division algorithms for $\mathbb{Z}[x], \mathbb{Z}[x, y]$ etc.

G. E. Collins [10] observes that the algorithm PTDIV is efficient for cases in which the quotient exists. An analysis of the computing time, however, is very difficult.

Now let us consider the problem of computing the n th power of a polynomial p . The algorithms which can be efficiently used to solve this problem depend heavily on the "density" of the polynomial p . Investigations of algorithm complexity are carried out for the two extreme cases, where p is completely sparse or p is dense.

R. J. Fateman [13] gives comparative analyses of algorithms for the computation of integer powers of sparse polynomials. More precisely, all the following analyses are carried out for polynomials which are completely sparse to power N (for the exact definition we refer to [13], p. 145), where $N \geq n$, and n is the power to which p should be raised. All the polynomials considered have coefficients in a finite field.

Thereby we avoid the consideration of coefficient growth. By the use of the Chinese Remainder Algorithm this restriction can be removed (see [19]).

The most obvious algorithm, RMUL, successively computes $p^2 = p \cdot p$, $p^3 = p \cdot p^2, \dots, p^n = p \cdot p^{n-1}$. The number of coefficient multiplications needed is

$$t \cdot \binom{t+n-1}{t} - t,$$

where t is the number of non-zero terms in p .

A second algorithm, RSQ, for computing p^n uses the binary exponentiation method, based on the binary expansion of n (compare the corresponding algorithm for integer exponentiation). However, it turns out that RSQ is more expensive than RMUL. The reason for this complexity behaviour is, that it is less costly to multiply a large polynomial by a small one, than to multiply two "mediumsize" polynomials. For a more detailed comparison of RMUL and RSQ we refer to [15].

In [13] a number of further algorithms for computing the powers of a polynomial are considered, which rely on the multinomial or binomial expansion. The relation of the computing times for these algorithms to the computing time for RMUL is better than (or equal to) $t/(t+n-1)$. The best of this class of algorithms seems to be BINB, an algorithm which can be described as follows:

$$q \leftarrow \text{BINB}(p, n)$$

[algorithm for raising the polynomial p to the n th power. Binomial expansion with half splitting. p is a polynomial with t non-zero terms, n is an integer, $q = p^n$]

- (1) Split p into $p_1 + p_2$, where the number of terms in p_1 is $\lfloor t/2 \rfloor$.
- (2) Compute p_1^2, \dots, p_1^n as well as p_2^2, \dots, p_2^n .
- (3) Use the binomial theorem to compute

$$q = p^n = \sum_{i=0}^n \binom{n}{i} p_1^i \cdot p_2^{n-i}. \quad \blacksquare$$

The cost of BINB is

$$\binom{t+n-1}{n} + t \cdot \binom{t/2+n-1}{n-1} - 2 \cdot \binom{t/2+n-1}{n} \\ - (t/2) \cdot (t/4 - n - \log_2(t-1) + 4).$$

For large n , and larger t BINB approaches $t^n/n! + O(t^{n-1}/2(n-2)!)$.

The other extreme case is p being a dense polynomial, i.e. every power product $x_1^{d_1} \cdots x_v^{d_v}$ has a non-zero coefficient in p for $d_1 + \cdots + d_v$ less or equal to some degree d . In [14] R. F. Fateman gives an overview of exponentiation of dense polynomials. The number of coefficient multiplications for the method of repeated multiplication is bounded by $(d+1)^{2^v} \cdot n^{v+1}/(v+1)$. Again there is the binary exponentiation method. If we use the Karatsuba algorithm for computing the intermediary results, the cost for the binary method is $O((nd)^{v \cdot \log_2 3})$.

Another algorithm, EVAL, computes the n th power of the polynomial p by computing $p(b_i)^n$ for $nd + 1$ integers $b_1 = 0, \dots, b_{nd+1} = nd$, (where d is the degree of p) and then uses interpolation to compute $q = p^n$. For several variables $p(b_i)$ is a polynomial in one less variables and EVAL is applied recursively to compute $p(b_i)^n$. The number of multiplications required for computing p^n by the algorithm EVAL is $O((nd)^{v+1})$, which is, asymptotically, more efficient than the previous algorithms.

The binomial method can also be used in this case. It turns out that it is best to split p into p_1 and p_2 such that p_1 is a monomial. In this case the number of multiplications is $O(n^{2v}d^{2v-1} + n^{v+1}d^{2v})$. So the binomial method will be inferior to repeated multiplication and EVAL for large problems. Timings given in [14], however, show that the binomial method is quite good for many interesting problems.

Finally p^n may be calculated by the fast Fourier transform. The cost for exponentiation by the use of the FFT is $O(v \cdot (nd)^v \cdot \log_2(nd))$, which is the best known asymptotic bound for the dense multivariate case.

Now let us turn to the problem of evaluating a polynomial $p = \sum_{i=0}^n p_i x^i$ at some point x_0 . An overview is given in [23], Section 4.6.4. The most obvious algorithm, which successively computes $p_0, p_1 x_0, \dots, p_n x_0^n$ uses $2n - 1$ multiplications and n additions. An essential improvement in complexity is Horner's rule, by which $p(x_0)$ is computed as

$$((\dots (p_n x_0 + p_{n-1})x_0 + \dots)x_0 + p_0.$$

n multiplications and n additions are necessary for evaluating a polynomial p of degree n at some point x_0 by the Horner's rule. D. E. Knuth in [23] points out that the computation of $p(x_0)$ by Horner's rule is essentially the same as dividing $p(x)$ by $(x - x_0)$. Dividing $p(x)$ by another polynomial which has a root at x_0 , namely $x^2 - x_0^2$, yields the second-order Horner's rule

$$p(x_0) = (\dots (p_{2\lfloor n/2 \rfloor} x_0^2 + p_{2\lfloor n/2 \rfloor - 2} x_0^2 + \dots)x_0^2 + p_0 \\ + ((\dots (p_{2\lfloor n/2 \rfloor - 1} x_0^2 + p_{2\lfloor n/2 \rfloor - 3} x_0^2 + \dots)x_0^2 + p_1)x_0.$$

$n + 1$ multiplications and n additions are necessary to evaluate p at x_0 by the second-order Horner's rule. So it is no improvement over the first-order Horner's rule, but if we have to evaluate $p(x_0)$ and $p(-x_0)$ at the same time, this can be achieved by just one more addition.

Polynomial evaluation is essential for performing the Taylor shift of a polynomial p , i.e. from given coefficients of p the coefficients of $p(x + x_0)$ should be computed:

$$p(x + x_0) = p(x_0) + p'(x_0)x + (p''(x_0)/2!)x^2 + \dots + (p^{(n)}(x_0)/n!)x^n.$$

M. Shaw and J. F. Traub [37] investigate the complexity of algorithms for solving this problem. Successive use of Horner's rule for computing the coefficients of $p(x + x_0)$ requires $n(n + 1)/2$ multiplications and the same number of additions. In [37], p. 162 an algorithm for computing all coefficients of $p(x + x_0)$ by only $2n - 1$ multiplications and $n - 1$ divisions is presented:

$$q \leftarrow \text{TST}(p, x_0)$$

[Taylor shift. Algorithm of Shaw and Traub. p is a polynomial, x_0 a point in the range of p , $q = p(x + x_0)$]

- (1) Compute $t_i^{-1} \leftarrow p_{n-i-1} x_0^{n-i-1}$ for $i = n - 1, \dots, 1, 0$.
- (2) Compute $t_j^j \leftarrow p_n x_0^n$ for $j = 0, 1, \dots, n$.
- (3) For $j = 0, 1, \dots, n - 1$, $i = j + 1, \dots, n$
set $t_i^i \leftarrow t_{i-1}^{j-1} + t_{i-1}^j$.
- (4) $q_j \leftarrow t_n^j / x_0^j$ for $j = 0, 1, \dots, n$. ■

TST is a special case of a family of splitting algorithms for computing the first m normalized derivatives $p^{(j)}/j!$ of a polynomial p , which is presented in [37].

A balance and conquer strategy due to Schönhage (private communication) splits $p(x) = p_1(x) \cdot x^r + p_0(x)$ similar to the Karatsuba split. Then the Taylor shift operator T is applied recursively over the degree

$$Tp = Tp_1 \cdot (x + x_0)^r + Tp_0,$$

where the basis is obvious and the powers $(x + x_0)^r$ are precomputed up to $r = \lfloor n/2 \rfloor$. For the multiplication a fast method can be useful.

The Rational Functions

In the section on rational numbers we have developed the basic concepts in a context general enough to contain rational functions as a subcase. Whenever I is an integral domain then $I[x_1, \dots, x_r]$ is also an integral domain and therefore one can embed $I[x_1, \dots, x_r]$ into its field of quotients $I(x_1, \dots, x_r)$ (the elements of $I(x_1, \dots, x_r)$ are called rational functions or quopolynomials).

We are especially interested in the case where $I = \mathbb{Z}$. There exists a gcd-algorithm for $\mathbb{Z}[x_1, \dots, x_r]$ (see the chapter on remainder sequences in this volume) and therefore unique representatives for the equivalence classes in $\mathbb{Z}(x_1, \dots, x_r)$ can be computed.

Addition, subtraction, multiplication, inversion and division algorithms for $\mathbb{Z}(x_1, \dots, x_r)$ are straightforward generalizations of the corresponding algorithms on rational numbers. The method of P. Henrici [17] and W. S. Brown [5] is used. For a detailed description of the algorithms see [9]. By carefully analyzing the points at which to compute gcd's it is also possible to design a Henrici-Brown algorithm for differentiating rational functions (see [9]).

A thorough computing time analysis of the Henrici-Brown algorithms for addition, multiplication and differentiation would be a considerable task. Let us just coarsely compare the Henrici-Brown multiplication algorithms to a classical algorithm, where the two rational functions to be multiplied are univariate. Most of the computing time for either algorithm will be devoted to gcd calculation, since the time to compute the gcd of two n th degree polynomials is approximately proportional to n^4 by [8]. Thus it follows easily that the Henrici-Brown algorithm will be faster than the classical algorithm by a factor of approximately 8. It is fairly

clear that the advantage of the Henrici-Brown algorithms will increase as the number of variables increases.

Evaluation of rational functions reduces to evaluation of polynomials.

Power Series

A power series is a formal sum $U(x) = \sum_{n=0}^{\infty} u_n x^n$ whose coefficients u_n belong to a field. Of course, it is impossible to store all the infinitely many coefficients of a power series in a computer memory. As in the case of algebraic numbers, an initial segment u_0, \dots, u_N may be considered as an approximation of the power series, but conceptually an algorithm to compute for arbitrary N the next higher coefficients of the series is considered as the algebraic part of the power series representation.

Addition and subtraction are defined (and computed) component-wise. Multiplication of power series is done by the familiar Cauchy product rule. The quotient $Q(x) = U(x)/V(x)$ (for $v_0 \neq 0$) is the solution of $U = Q \cdot V$ for given U and V . The coefficients of Q are obtained by comparing like powers of x on both sides of this equation.

For computing the results of these operations it is necessary to have exact arithmetic in the underlying field, or, if the coefficients of the inputs are given only symbolically, to be able to simplify the occurring expressions. This establishes the connection between arithmetic on power series and other topics in computer algebra. Algebraic computations of power series coefficients are frequently used to increase the order of an approximation symbolically for subsequent numerical treatment (e.g. Runge-Kutta formulas).

D. E. Knuth, in [23], Section 4.7, presents algorithms for performing these operations. Furthermore he considers the problems of exponentiation (by a method suggested by J. C. P. Miller, see [18]) and reversion.

Finite Fields

Representations

If $q > 1$ is the power of some prime p there exists a finite field with q elements, (this field is commutative and unique up to an isomorphism, it is denoted \mathbb{F}_q or $\text{GF}(q)$). Inversely if a finite field has q elements then q is a power of some prime number, say $q = p^k$. For our point of view the best source of information about finite fields is Berlekamp's book [2], Chap. 4. It is known that the nonzero elements of \mathbb{F}_q are equal to the powers $\alpha^0, \alpha^1, \alpha^2, \dots, \alpha^{q-2}$ of some element α called a primitive element ([2], Theorem 4.24).

The field \mathbb{F}_q can be viewed in different ways:

- (i) $\mathbb{F}_q = \{0\} \cup \{1, \alpha, \alpha^2, \dots, \alpha^{q-2}\}$,
- (ii) \mathbb{F}_q is a vector space of dimension k over \mathbb{F}_p with a given basis,
- (iii) $\mathbb{F}_q = \mathbb{F}_p[X]/Q(X)$, where $Q(X)$ is a polynomial of $\mathbb{F}_p[X]$ of degree k and irreducible over \mathbb{F}_p .

See [2], Table 4.1, for an example of several representations of \mathbb{F}_{16} . Even if equivalent from the mathematical point of view, these representations are quite different in computer algebra because the algorithms of the arithmetical operations are very dependent on the representation, as we shall see below. Of course, representation (iii) is a special case of representation (ii).

Addition

In representation (ii) (and (iii)), if (e_1, \dots, e_k) is a basis of \mathbb{F}_q as a vector field over \mathbb{F}_p and if $u = u_1e_1 + \dots + u_ke_k, v = v_1e_1 + \dots + v_ke_k$ (where $u_1, \dots, u_k, v_1, \dots, v_k$ are integers modulo p) then

$$w = u + v = w_1e_1 + \dots + w_ke_k$$

where

$$w_j = (u_j + v_j) \bmod p, \quad j = 1, \dots, k.$$

In this case addition is very easy. But this is not at all the same in representation (i), if $u = \alpha^m$ and $v = \alpha^n$ then

$$w = u + v = \alpha^m + \alpha^n.$$

To obtain an integer l such that $w = \alpha^l$ there is no simple law and a table of the integers a_0, a_1, \dots such that

$$1 + \alpha^j = \alpha^{a_j}, \quad j = 0, 1, \dots, q-2$$

is needed.

Multiplication

For this operation representation (i) is very practical:

$$\alpha^m \cdot \alpha^n = \alpha^l, \quad \text{with} \quad l = (m + n) \bmod (q - 1).$$

But now representation (ii) is not very good. To compute

$$(u_1e_1 + \dots + u_ke_k) \cdot (v_1e_1 + \dots + v_ke_k)$$

one needs a table of the products $e_i \cdot e_j, 0 \leq i \leq j \leq k$. The special case of representation (iii) is better, then $e_i = X^{i-1}, i = 1, \dots, k$ (more precisely, e_i is the image of X^{i-1} by the natural homomorphism $\mathbb{F}_p[X] \rightarrow \mathbb{F}_p[X]/Q(X)$!) and we need only to compute $X^i \bmod Q(X)$ for $i = k, \dots, 2k-2$.

Reciprocal

Again representation (i) is very practical

$$(\alpha^m)^{-1} = \alpha^{q-1-m} \quad \text{for} \quad m = 1, 2, \dots, q-2.$$

Representation (ii) is very bad. And if

$$u = u_1 + u_2X + \dots + u_kX^{k-1} = U(X)$$

the extended Euclidean algorithm in $\mathbb{F}_p[X]$ gives a relation

$$U(X)V(X) + Q(X)W(X) = 1$$

for some polynomials V and W over \mathbb{F}_q and

$$u^{-1} = V(X) \bmod Q(X).$$

Some computational details of arithmetic in finite fields may be found in the chapter on computing in extensions in this volume.

p -adic Numbers

Let p be a prime number. On the set \mathbb{Z} of rational integers we define the function

$$|a|_p = p^{-r} \quad \text{if } p^r | a \text{ and } p^{r+1} \nmid a, \quad a \neq 0,$$

and

$$|0|_p = 0.$$

This is a distance and \mathbb{Z}_p , the set of p -adic integers, is the completion of \mathbb{Z} with respect to this distance. In \mathbb{Z}_p a series

$$\sum_{i=0}^{\infty} c_i p^i, \quad c_i \in \mathbb{Z}$$

is convergent. Moreover each element α of \mathbb{Z}_p can be written uniquely in the canonical form

$$\alpha = \sum_{i=0}^{\infty} a_i p^i, \quad a_i \in \{0, 1, \dots, p-1\}, \quad i \geq 0.$$

If

$$\beta = \sum_{i=0}^{\infty} b_i p^i$$

then

$$\alpha + \beta = \sum_{i=0}^{\infty} (a_i + b_i) p^i$$

(of course, in general this formula is not the canonical form but the canonical form can be obtained easily from it) and

$$\alpha\beta = \sum_{n=0}^{\infty} \left(\sum_{i+j=n} a_i b_j \right) p^n.$$

The formula to compute the canonical form of $-\beta$ is the following: suppose

$$\beta = b_k g^k + b_{k+1} g^{k+1} + \dots, \quad b_k \neq 0,$$

then

$$-\beta = 0 - \beta = (g \cdot g^k + (g-1)g^{k+1} + (g-1)g^{k+2} + \dots) - \beta$$

so

$$-\beta = (g - b_k)g^k + (g - 1 - b_{k+1})g^{k+1} + (g - 1 - b_{k+2})g^{k+2} + \dots$$

and this is a canonical expansion. Of course

$$\alpha - \beta = \alpha + (-\beta).$$

The ring \mathbb{Z}_p is integral and its quotient field is called the field of p -adic numbers, \mathbb{Q}_p . Each non-zero element α of \mathbb{Q}_p can be written uniquely in the form

$$\alpha = \sum_{n=r}^{\infty} a_n p^n, \quad a_r \neq 0,$$

for some rational integer r . The formulas for addition, subtraction and multiplication in \mathbb{Q}_p are similar to those for the same operations in \mathbb{Z}_p .

The formula for division in \mathbb{Q}_p is the same as the formula for division of formal power series (this was also the case for addition and multiplication). Suppose we compute $\gamma = \alpha/\beta$, where

$$\alpha = a_0 + a_1 p + \cdots, \quad \beta = b_0 + b_1 p + \cdots, \quad b_0 \neq 0$$

(this normalization causes no loss of generality), then

$$\gamma = c_0 + c_1 p + c_2 p^2 + \cdots$$

with $c_0 = a_0/b_0$,

$$c_n = (a_n - c_0 b_n - c_1 b_{n-1} - \cdots - c_{n-1} b_1)/b_0, \quad n \geq 1.$$

But in the previous expansion the c_i 's are not rational integers in the general case, except if $b_0 = \pm 1$. To obtain the canonical expansion of α/β one can proceed as follows:

there is a digit $c'_0 \neq 0$ such that

$$b_0 c'_0 \equiv a_0 \pmod{p},$$

form

$$\alpha - c'_0 \beta = a'_1 p + a'_2 p^2 + \cdots,$$

next there is a digit c'_1 such that

$$c'_1 \beta \equiv a'_1 \pmod{p},$$

form

$$\alpha - c'_0 \beta - c'_1 p \beta = a''_2 p^2 + a''_3 p^3 + \cdots$$

etc. . . .

A very interesting fact about \mathbb{Q}_p is that some algebraic equations can be solved. Let us take an example. We try to solve the equation $x^2 + 1 = 0$ in \mathbb{Q}_5 . Suppose $\alpha = a_0 + 5a_1 + 5^2 a_2 + \cdots$ is such a solution then

$$a_0^2 \equiv -1 \pmod{5}, \quad \text{i.e.} \quad a_0 = 2 \text{ or } 3.$$

Choose $a_0 = 2$. Then

$$0 = \alpha^2 + 1 = 5 + 4a_1 \cdot 5 + (4a_2 + a_1^2) \cdot 5^2 + \cdots.$$

These relations imply $1 + 4a_1 \equiv 0 \pmod{5}$, hence $a_1 = 1$. Similarly we get $a_2 = 2$, $a_3 = 1, \dots$ and

$$\alpha = 2 + 1 \cdot 5 + 2 \cdot 5^2 + 1 \cdot 5^3 + 2 \cdot 5^4 + \dots$$

For a detailed presentation of p -adic numbers we refer the reader to Mahler's book [26].

References

- [1] Aho, A. V., Hopcroft, J. E., Ullman, J. D.: *The Design and Analysis of Computer Algorithms*. Reading, Mass.: Addison-Wesley 1974.
- [2] Berlekamp, E. R.: *Algebraic Coding Theory*. New York: McGraw-Hill 1968.
- [3] Bonneau, R. J.: *Polynomial Operations Using the Fast Fourier Transform*. Cambridge, Mass., MIT Dept. of Math., 1974.
- [4] Brent, R. P.: Fast Multiple Precision Evaluation of Elementary Functions. *J. ACM* **23**, 242–251 (1976).
- [5] Brown, W. S., Hyde, J. P., Tague, B. A.: The ALPAK System for Nonnumerical Algebra on a Digital Computer-II: Rational Functions of Several Variables and Truncated Power Series with Rational Function Coefficients. *Bell Syst. Tech. J.* **43**, No. 1, 785–804 (1964).
- [6] Caviness, B. F.: A Lehmer-Type Greatest Common Divisor Algorithm for Gaussian Integers. *SIAM Rev.* **15**, No. 2, Part 1, 414 (April 1973).
- [7] Caviness, B. F., Collins, G. E.: Algorithms for Gaussian Integer Arithmetic. *SYMSAC* **1976**, 36–45.
- [8] Collins, G. E.: Computing Time Analyses of Some Arithmetic and Algebraic Algorithms. *Univ. of Wisconsin, Madison, Comp. Sci. Techn. Rep.* 36 (1968).
- [9] Collins, G. E.: The SAC-1 Rational Function System. *Univ. of Wisconsin, Madison, Comp. Sci. Techn. Rep.* 135 (1971).
- [10] Collins, G. E.: Computer Algebra of Polynomials and Rational Functions. *Am. Math. Mon.* **80**, No. 2, 725–755 (1973).
- [11] Collins, G. E.: *Lecture Notes in Computer Algebra*. Univ. of Wisconsin, Madison.
- [12] Collins, G. E., Musser, D. R.: Analysis of the Pope-Stein Division Algorithm. *Inf. Process. Lett.* **6**, 151–155 (1977).
- [13] Fateman, R. F.: On the Computation of Powers of Sparse Polynomials. *Stud. Appl. Math.* **LIII**, No. 2, 145–155 (1974).
- [14] Fateman, R. F.: Polynomial Multiplication, Powers and Asymptotic Analysis: Some Comments. *SIAM J. Comput.* **3**, No. 3, 196–213 (1974).
- [15] Gentleman, W. M.: Optimal Multiplication Chains for Computing a Power of a Symbolic Polynomial. *Math. Comput.* **26**, No. 120 (1972).
- [16] Hardy, G. H., Wright, E. M.: *An Introduction to the Theory of Numbers*, 5th ed. Oxford: Clarendon Press 1979.
- [17] Henrieci, P.: A Subroutine for Computations with Rational Numbers. *J. ACM* **3**, 6–9 (1956).
- [18] Henrieci, P.: Automatic Computations with Power Series. *J. ACM* **3**, 10–15 (1956).
- [19] Horowitz, E.: The Efficient Calculation of Powers of Polynomials. 13th Annual Symp. on Switching and Automata Theory (IEEE Comput. Soc.) (1972).
- [20] Horowitz, E.: A Sorting Algorithm for Polynomial Multiplication. *J. ACM* **22**, No. 4, 450–462 (1975).
- [21] Johnson, S. C.: Sparse Polynomial Arithmetic. *EUROSAM* **1974**, 63–71.
- [22] Karatsuba, A., Ofman, Yu.: *Dokl. Akad. Nauk SSSR* **145**, 293–294 (1962) [English translation: *Multiplication of Multidigit Numbers on Automata. Sov. Phys., Dokl.* **7**, 595–596 (1963)].
- [23] Knuth, D. E.: *The Art of Computer Programming*, Vol. 2, 2nd ed. Reading, Mass.: Addison-Wesley 1981.
- [24] Knuth, D. E.: *The Analysis of Algorithms*. Proc. Internat. Congress Math. (Nice, 1970), Vol. 3, pp. 269–274. Paris: Gauthier-Villars 1971.
- [25] Lehmer, D. H.: Euclid's Algorithm for Large Numbers. *Am. Math. Mon.* **45**, 227–233 (1938).
- [26] Mahler, K.: *Introduction to p -adic Numbers and Their Functions*. Cambridge Univ. Press 1973.

- [27] Matula, D. W.: Fixed-Slash and Floating-Slash Rational Arithmetic. Proc. 3rd Symp. on Comput. Arith. (IEEE), 90–91 (1975).
- [28] Matula, D. W., Kornerup, P.: A Feasibility Analysis of Binary Fixed-Slash and Floating-Slash Number Systems. Proc. 4th Symp. on Comput. Arith. (IEEE), 29–38 (1978).
- [29] Matula, D. W., Kornerup, P.: Approximate Rational Arithmetic Systems: Analysis of Recovery of Simple Fractions During Expression Evaluation. EUROSAM 1979, 383–397.
- [30] Moenck, R. T.: Studies in Fast Algebraic Algorithms. Ph.D. Thesis, Univ. of Toronto, 1973.
- [31] Moenck, R. T.: Fast Computations of GCD's. Proc. 5th Symp. on Theory of Comput. (ACM), 142–151 (1973).
- [32] Moenck, R. T.: Practical Fast Polynomial Multiplication. SYMSAC 1976, 136–148.
- [33] Pollard, J. M.: The Fast Fourier Transform in a Finite Field. Math. Comput. **25**, 365–374 (1971).
- [34] Pope, D. A., Stein, M. L.: Multiple Precision Arithmetic. Commun. ACM **3**, 652–654 (1960).
- [35] Schönhage, A.: Schnelle Berechnung von Kettenbruchentwicklungen. Acta Inf. **1**, 139–144 (1971).
- [36] Schönhage, A., Strassen, V.: Schnelle Multiplikation großer Zahlen. Computing **7**, 281–292 (1971).
- [37] Shaw, M., Traub, J. F.: On the Number of Multiplications for the Evaluation of a Polynomial and Some of its Derivatives. J. ACM **21**, No. 1, 161–167 (1974).
- [38] Strassen, V.: The Computational Complexity of Continued Fractions. SYMSAC 1981, 51–67.
- [39] van der Waerden, B. L.: Modern Algebra, Vol. 1. New York: Frederick Ungar 1948.

Prof. Dr. G. E. Collins
Computer Science Department
University of Wisconsin – Madison
1210 West Dayton Street
Madison, WI 53706, USA

Prof. Dr. M. Mignotte
Centre de Calcul de l'Esplanade
Université Louis Pasteur
5, rue René Descartes
F-67084 Strasbourg Cédex
France

Dipl.-Ing. F. Winkler
Institut für Mathematik
Johannes-Kepler-Universität Linz
Altenbergerstrasse 69
A-4040 Linz
Austria