Computer Algebra - with a View Toward Reliable Numeric Computation

Michael Sagraloff

December 22, 2017

Contents

Chapter 1

Basic Arithmetic

In this section, we present an efficient algorithm due to Toom and Cook for multiplying two integers, which already considerably improves upon the method that most people have learned in school. We further investigate in methods for carrying out approximate computations on fixed-point and floating-point numbers, and we derive bounds on the occurring error when using approximate instead of exact arithmetic. In addition, we introduce the concepts of interval arithmetic and box-functions and show that these concepts yield a powerful and very practical approach for carrying out approximate arithmetic. This is due to the fact that adaptive bounds on the error can directly be computed "on the fly", and that these bounds are often much better than any a priori bounds obtained by a worst-case error analysis. Finally, we give an efficient method to compute an arbitrary good approximation of the quotient of two integers or, more generally, two arbitrary complex values.

1.1 The School Method for Integer Multiplication

We represent integers $a \in \mathbb{Z}$ as digit strings with respect to a fixed base $B \in \mathbb{N}_{\geq 2}$. That is,

$$a = (-1)^s \cdot \sum_{i=0}^{n-1} a_i \cdot B^i, \text{ with } s \in \{0,1\} \text{ and } a_i \in \{0,\ldots,B-1\} \text{ for all } i = 0,\ldots,n-1.$$

We call the a_i's the *digits* and s the *sign digit* of a with respect to B. For convenience, we also write (if B is fixed)

$$a = (-1)^s \; a_{n-1}a_{n-2}\ldots a_0$$

if the base B is fixed.

Example: Important bases are $B = 2, 10, 16$, and 2^k for some $k \in \mathbb{N}$. The integer 29 writes as

$$29 = 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 + 1 \cdot 2^4 = 11101.$$

The *length* (or *bitsize* for $B = 2$) of an integer a with respect to B is defined as the number of digits needed to represent a. For convenience, we use the term *n-digit number* to denote an integer of length n. Notice that any n-digit number can always be considered as an N-digit number for arbitrary $N \geq n$. This is advantageous in the analysis of many algorithms as it allows us to assume that the length of the input is a power of 2 (or some other value k).

Algorithm 1: School Method for Addition

Input : Two non-negative n-digit integers $a = a_{n-1} \dots a_0$ and $b = b_{n-1} \dots b_0$.
Output: An $(n+1)$-digit integer $c = c_n \dots c_0$ with $c = a + b$.

$\gamma_0 := 0$
for $i = 0, \dots, n-1$ **do**
    Recursively define
    $\gamma_{i+1} \cdot B + c_i = a_i + b_i + \gamma_i$ with $c_i, \gamma_i \in \{0, \dots, B-1\}$
$c_n := \gamma_n$
return $c_n \dots c_0$

We mainly consider two different ways of measuring the efficiency of an algorithm. The first one is to count the number of additions and multiplications between integers that an algorithm needs to return a result. This is referred to as the *arithmetic complexity* of an algorithm. Notice that the arithmetic complexity might be unrelated to the actual running time of an algorithm as the involved integers can be arbitrarily large. Hence, a more meaningful and precise way of measuring the efficiency of an algorithm is to count instead the number of *primitive operations* (or bit operations if the base B equals 2) that are carried out by the algorithm, often referred to as the *bit complexity* of an algorithm. Notice that the result of a primitive operations is always a one- or two-digit number.

Example: A prominent example is Gaussian elimination for solving a linear system in n unknowns. It is easy to see that the method uses $O(n^3)$ arithmetic operations, hence the arithmetic complexity of Gaussian elimination is polynomial in the input size. However, a straight forward analysis does NOT guarantee that the intermediate results as computed by the algorithm (which are rationals if the input matrix has integer entries) have size that is polynomial in the size of the input, thus it is not obvious that Gaussian elimination actually constitutes a polynomial time algorithm for solving linear systems. A more refined argument however shows that by recursively removing common factors of the intermediate results, it can be guaranteed that all intermediate results have polynomial size. We will go into more detail in one of the exercises. Later, we will also consider a different approach based on modular computation that does not come with any of these drawbacks.

We now review and analyze the school method for adding and multiplying two non-negative n-digit integers $a = a_{n-1} \dots a_0$ and $b = b_{n-1} \dots b_0$. We first start with addition; see Algorithm 1. The γ_i's are called *carries*. Using induction, it is easy to see that $\gamma_i \in \{0, 1\}$ for all i. Further notice that γ_{i+1} is non-zero if and only if the sum of the two digits a_i and b_i and the previous carry γ_i is larger than the base B. We also remark that, for subtraction (i.e. the computation of $a - b$), we can assume that $a \geq b$. The recursion for c_i and γ is then almost identical. More specifically, we have

$$-\gamma_{i+1} \cdot B + c_i = a_i - b_i - \gamma_i \text{ with } c_i, \gamma_i \in \{0, \dots, B-1\}.$$

The proof of the following theorem is straight-forward.

Theorem 1.1.1. *The school method for adding (or subtracting) two $n-$digit numbers requires at most $2n$ primitive operations. The addition of an m-digit number and an n-digit number uses at most $m + n + 2$ primitive operations.*

Algorithm 2: School Method for Multiplication

Input : Two non-negative n-digit integers $a = a_{n-1} \dots a_0$ and $b = b_{n-1} \dots b_0$.
Output: A $2n$-digit integer $c = c_{2n-1} \dots c_0$ with $c = a \cdot b$.

$P_0 := 0$
for $j = 0, \dots, n-1$ **do**
  for $i = 0, \dots, n-1$ **do**
    Define
    $a_i \cdot b_j = c_{ij} \cdot B + d_{ij}$ with $c_{ij}, d_{ij} \in \{0, \dots, B-1\}$
  $c_j := c_{n-1,j} \dots c_{0,j} 0$
  $d_j := d_{n-1,j} \dots, d_{0,j}$
  $p_j = p_{n,j} \dots p_{0,j} := c_j + d_j$
  // * *Notice that* $p_j = a \cdot b_j$, *and thus* $a \cdot b = \sum_{j=0}^{n-1} p_j \cdot B^j$ **//*
  $P_{j+1} := P_j + p_j \cdot B$
return P_n

In the next step, we consider the school method for multiplying integers; see Algorithm 2. Let us count the number of primitive operations that Algorithm 2 needs:

- The computation of each product $a_i \cdot b_j$ requires one primitive operations, thus n^2 many primitive operations in total.
- Computing each of the integers p_j amounts for adding two $(n+1)$-digit numbers. Hence, in total, we need $2n(n+1)$ primitive operations.
- For computing P_n we need n additions each involving $2n$-digit numbers. Thus, we need $2n^2$ many primitive operations for this step.

We now obtain the following result. For the second claim on the complexity of computing the product of an m-digit number and an n-digit number, a completely analogous argument applies.

Theorem 1.1.2. *Using the school method, we need at most* $5n^2 + 2n = O(n^2)$ *primitive operations to multiply two* n*-digit numbers. Multiplication of an* n*-digit number and an* m*-digit number needs* $O(mn)$ *primitive operations.*

Exercise 1.1.3. *Let* $f = a_0 + \dots + a_d \cdot x^d \in \mathbb{Z}[x]$ *be a polynomial of degree* d *with integer coefficients of length at most* L*, and let* $m \in \mathbb{Z}$ *be an* ℓ*-digit number. Show that*

(a) $f(m)$ *is a* $O(d\ell + L)$*-digit number.*

(b) Computing $f(m)$ *using* Horner's method

$$f(m) = a_0 + m \cdot (a_1 + m \cdot (a_2 + \cdots m \cdot (a_{d-1} + m \cdot a_d))$$

and the school method for multiplication uses $O(d \cdot (d\ell^2 + \ell \cdot L)))$ *primitive operations.*

We will later see that it is even possible to compute $f(m)$ in only $\tilde{O}(d \cdot (\ell + L))$ primitive operations, where $\tilde{O}(.)$ means that poly-logarithmic factors are suppressed, that is, $\tilde{O}(T) = O(T \cdot (\log T)^c)$ for some constant c. For the special case, where f has only a few non-zero coefficients, $f(m)$ can be evaluated in a faster manner via *repeated squaring*:

Exercise 1.1.4 (Sparse Polynomial Evaluation)**.** *Let $f = \sum_{j=1}^{k} a_{i_j} \cdot x^{i_j} \in \mathbb{R}[x]$ be a so-called* sparse *polynomial (also k-*nomial*) of degree n with k non-zero coefficients and $m \in \mathbb{R}$ be an arbitrary real value. Show that $f(m)$ can be computed using $O(k \cdot \log n)$ arithmetic operations.*

Hint: Show the claim for a single monomial x^n first. For this, use *repeated squaring*

$$x^n = \prod_{i=0}^{\lceil \log n \rceil} x^{[n_i \cdot i]},$$

to compute x^n, where $n = \sum_{i=0}^{\lceil \log n \rceil} n_i \cdot 2^i$, with $n_i \in \{0, 1\}$, is the binary representation of n and $x^{[j]}$ is recursively defined as

$$x^{[0]} := 1,\ x^{[1]} := x,\ \text{and } x^{[i]} := \left(x^{[i-1]}\right)^2 \text{ for } i \geq 2.$$

Exercise 1.1.5. *Let $A = (a_{i,j})_{i,j=1,\ldots,n} \in \mathbb{Z}^{n \times n}$ be an $n \times n$-matrix with integer entries $a_{i,j}$ of length at most L.*

(a) Derive an upper bound on the number of primitive operations that are needed to compute the inverse A^{-1} of A.

(b) Show that the entries of A^{-1} are rational numbers with numerators and denominators of length $O(n(L + \log n))$.

(c) Suppose that Gaussian elimination with pivoting is used to compute the determinant of A. Further suppose that, after each iteration, we reduce all intermediate entries $a'_{i,j} = \frac{p}{q} \in \mathbb{Q}$, that is, we ensure that $\gcd(p, q) = 1$. Show that p and q can be represented using $O(n^2(L + \log n))$ digits and conclude that Gaussian elimination constitutes a polynomial time algorithm for computing determinants.*

Hints: For (a), consider Gaussian elimination to compute A^{-1} and derive a bound on the numerators and denominators of the rational entries of the matrices produced after each iteration. For (b), use Cramer's Rule to write the entries of A^{-1} as fractions of determinants of suitable $n \times n$-matrices and use the definition of the determinant to bound the size of the numerator and denominator. For (c), show that, in each iteration, the pivot element can be written as the quotient of the determinants of two sub-matrices of A.

1.2 The Toom-Cook Algorithm

We now investigate in algorithms for multiplying integers that are considerably faster than the school method. We start with a simple algorithm due to Karatsuba [AY62] (from 1960). Its running time $O(n^{\log_2 3})$ already constitutes a considerable improvement upon the running time $O(n^2)$ of the school method. Then, we show how to generalize the approach to achieve a running time $O(n^{1+\epsilon})$ for arbitrary $\epsilon > 0$.

Let $a = a_{n-1} \ldots a_0$ and $b = b_{n-1} \ldots b_0$ be integers of length n. We first write

$$a = a_{n-1} \ldots a_0 = a' \cdot B^{\lceil n/2 \rceil} + a'', \text{ and}$$
$$b = b_{n-1} \ldots b_0 = b' \cdot B^{\lceil n/2 \rceil} + b'',$$

Algorithm 3: Karatsuba Multiplication (1960)

Input : Two non-negative n-digit integers $a = a_{n-1} \dots a_0$ and $b = b_{n-1} \dots b_0$.
Output: A $2n$-digit integer $c = c_{2n-1} \dots c_0$ with $c = a \cdot b$.

if $n \leq 4$ **then**
  Compute $c = a \cdot b$ using Algorithm 2
else
  Define

$$a = a_{n-1} \dots a_0 = a' \cdot B^{\lceil n/2 \rceil} + a'', \text{ and}$$
$$b = b_{n-1} \dots b_0 = b' \cdot B^{\lceil n/2 \rceil} + b'',$$

  with integers a', a'', b', b'' of length $n/2$.
  $A := a' + a''$
  $B := b' + b''$
  Compute $P_1 := a' \cdot b'$, $P_2 := A \cdot B$, and $P_3 := a'' \cdot b''$ by recursively calling Algorithm 3.
  $P := P_1 \cdot B^{2\lceil n/2 \rceil} + (P_2 - P_1 - P_3) \cdot B^{\lceil n/2 \rceil} + P_3$
return P

with integers a', a'', b', b'' of length $\lceil n/2 \rceil$. Then, it holds that

$$\begin{aligned} a \cdot b &= (a' \cdot B^{\lceil n/2 \rceil} + a'') \cdot (b' \cdot B^{\lceil n/2 \rceil} + b'') \\ &= a'b' \cdot B^{2\lceil n/2 \rceil} + (a'b'' + a'' \cdot b') \cdot B^{\lceil n/2 \rceil} + a''b'' \\ &= \underbrace{a'b'}_{=:P_1} \cdot B^{2\lceil n/2 \rceil} + [\underbrace{(a' + a'')(b' + b'')}_{=:P_2} - (a'b' + a''b'')] \cdot B^{\lceil n/2 \rceil} + \underbrace{a''b''}_{=:P_3} \end{aligned} \tag{1.1}$$

What have we gained in the last step? They crucial point is that, when passing from the second line to the last line, we reduced the problem to three (instead of four!) multiplications and six (instead of three) additions. Notice that there are actually five multiplications, however, each of the products P_1 and P_2 appears twice, and thus only 3 different products need to be computed. So the total number of additions and multiplication has increased, however, additions are much cheaper than multiplications. We can now recursively use the above approach for multiplication until all remaining multiplications are numbers with four or less digits; see Algorithm 3

Theorem 1.2.1. *Using Karatsuba multiplication, we need $O(n^{\log 3}) = O(n^{1.58\dots})$ primitive operations to multiply two n-digit numbers.*

Proof. Let $T(n)$ denote the maximal number of operations needed to multiply two n-digit numbers using the Karatsuba algorithm. If $n \leq 4$, Theorem 1.1.2 yields that $T(n) \leq 5n^2 + 2n \leq 88$. For $n \geq 5$, it holds that

$$T(n) \leq 3 \cdot T(\lceil n/2 \rceil + 1) + 6 \cdot (4n).$$

as we need to compute 3 products involving $\lceil n/2 \rceil$- or $(\lceil n/2 \rceil + 1)$-digit numbers and 6 additions involving $2n$-digit numbers. Now, a general version of the Master Theorem (e.g. see [MS08, Sec. 2.6]) yields a total running time of size $O(n^{\log_2 3})$. □

Remark. For readers who are not familiar with the general Master Theorem, we give the following direct argument from [MS08], which also yields an explicit bound for $T(n)$. For $\ell \in \mathbb{N}_{\geq 1}$, we first prove that

$$T(2^\ell + 2) \leq 33 \cdot 3^\ell + 12 \cdot (2^{\ell+1} + 2\ell - 2)$$

using induction on ℓ. For $\ell = 1$, the claim is obviously true as $T(4) \leq 88$. For $\ell \geq 2$, we thus conclude from the induction hypothesis and the above recursive formula for $T(n)$ that

$$\begin{aligned} T(2^\ell + 2) &\leq 3 \cdot T(2^\ell + 2) + 12 \cdot (2^\ell + 2) \\ &\leq 3 \cdot [33 \cdot 3^{\ell-1} + 12 \cdot (2^\ell + 2(\ell - 1) - 2)] + 12 \cdot (2^\ell + 2) \\ &= 33 \cdot 3^\ell + 12 \cdot (2^{\ell+1} + 2\ell - 2). \end{aligned}$$

Notice that our special choice for n (i.e. $n = 2^\ell + 2$) guarantees that $\lceil n/2 \rceil + 1 = 2^{\ell-1} + 2$ is again of the same form, and thus we can recursively apply the induction hypothesis on $T(\lceil n/2 \rceil + 1)$. It remains to derive a bound on $T(n)$ for arbitrary n. Setting $\ell := \lceil \log n \rceil \leq 1 + \log n$, we have

$$\begin{aligned} T(n) \leq T(2^\ell) &\leq 33 \cdot 3^\ell + 12 \cdot (2^{\ell+1} + 2\ell - 2) \\ &\leq 33 \cdot 3 \cdot 3^{\log n} + 12 \cdot (4 \cdot 3^{\log n} + 2(1 + \log n) - 2) \\ &\leq 99 \cdot n^{\log 3} + 48 \cdot \cdot n + 24 \cdot \log n. \end{aligned}$$

We now consider the following approach due to Toom and Cook (1966),[1] which extends Karatsuba's idea; see Algorithm 4. The first step is similar as in Karatsuba's method, however, instead of splitting each of the input numbers into two almost equally sized parts, we now consider a split into k parts, where $k \in \mathbb{N}_{\geq 2}$ is an arbitrary but fixed constant. That is, with $m := \lceil n/k \rceil$, we write

$$\begin{aligned} a &= a^{(0)} + a^{(1)} \cdot B^m + \cdots + a^{(k-1)} \cdot B^{(k-1)\cdot m}, \text{ and} \\ b &= b^{(0)} + b^{(1)} \cdot B^m + \cdots + b^{(k-1)} \cdot B^{(k-1)\cdot m}, \end{aligned}$$

such that each integer $a^{(i)}$ and $b^{(i)}$ has length at most m. Now, let $f(x) := \sum_{i=0}^{k-1} a^{(i)} \cdot x^i$ and $g(x) := \sum_{i=0}^{k-1} b^{(i)} \cdot x^i$ be corresponding polynomials of degree $k - 1$ with coefficients $a^{(i)}$ and $b^{(i)}$. Then, it holds that $a \cdot b = f(B^m) \cdot g(B^m) = h(B^m)$, where

$$h(x) = \sum_{i=0}^{2k-2} c^{(i)} \cdot x^i := f(x) \cdot g(x).$$

Notice that the coefficients $c^{(i)}$ of h are integers of length at most $O(m)$. Now, suppose that we know these coefficients, then we can easily compute $a \cdot b$ by shifting each of the coefficients $c^{(i)}$ by $i \cdot m$ digits and adding up the resulting integers. The cost for these additions (there are only constantly many!) is then bounded by $O(n)$. Hence, we have reduced the problem of computing the product $a \cdot b$ of two integers of length n to the problem of computing a product $g(x) \cdot h(x)$ of polynomials of degree less than k and with coefficients of length at

[1] In his Phd Thesis (http://cr.yp.to/bib/1966/cook.html), Cook improves upon Toom's original approach [Too63] from 1963

Algorithm 4: Toom-Cook-k Algorithm

Input : Two non-negative integers a and b of length at most n.
Output: The product $c = a \cdot b$.

Write

$$a = a^{(0)} + a^{(1)} \cdot B^m + \cdots + a^{(k-1)} \cdot B^{(k-1)\cdot m}, \text{ and}$$
$$b = b^{(0)} + b^{(1)} \cdot B^m + \cdots + b^{(k-1)} \cdot B^{(k-1)\cdot m},$$

with $m := \lceil n/k \rceil$ and integers $a^{(i)}$, $b^{(i)}$ of length at most m.

$f(x) := a^{(0)} + a^{(1)} \cdot x + \cdots + a^{(k-1)} \cdot x^{k-1}$

$g(x) := b^{(0)} + b^{(1)} \cdot x + \cdots + b^{(k-1)} \cdot x^{k-1}$

for $j = 0, \ldots, 2k-2$ **do**

  Define $x_j = j$

  *// * We can also choose other values for x_j unless the x_j's are pairwise distinct and of constant length *//*

  Compute $f_j := f(x_j)$ and $g_j := g(x_j)$

  Compute $h_j := f_j \cdot g_j$ by calling the Algorithm 4 recursively.

Compute the inverse V^{-1} of the *Vandermonde Matrix*

$$V := \mathrm{Vand}(x_0, \ldots, x_{2k-2}) := \begin{pmatrix} 1 & x_0 & \cdots & x_0^{2k-2} \\ 1 & x_1 & \cdots & x_1^{2k-2} \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_{2k-2} & \cdots & x_{2k-2}^{2k-2} \end{pmatrix}$$

Compute

$$\begin{pmatrix} c^{(0)} \\ c_{(1)} \\ \vdots \\ c^{(2k-2)} \end{pmatrix} = V^{-1} \cdot \begin{pmatrix} h_0 \\ h_1 \\ \vdots \\ h_{2k-2} \end{pmatrix}$$

$C_0 = c^{(0)}$

for $j = 1, \ldots, 2k-2$ **do**

  $C_j = C_j + B^{mj} \cdot c^{(j)}$

return $C_{2k-2} = c = a \cdot b$

most $\lceil n/k \rceil$. For the latter problem, we consider an *evaluation/interpolation approach*, that is, we first evaluate f and g at $2k-1$ many different points $x_0, \ldots, x_{2k-2} \in \mathbb{Z}$ of constant length. Typically, we consider $x_j := j$ for $j = 0, \ldots, 2k-2$ but also other choices are possible. Then, the resulting integer values $f_j = f(x_j)$ and $g_j := g(x_j)$ are of length $O(m)$ according to Exercise 1.1.3. For computing the k products $h_j := f_j \cdot g_j = f(x_j) \cdot g(x_j) = h(x_j)$, we call the multiplication algorithm recursively. In the third step, we interpolate $h(x)$ from its values h_j

at the points x_j. Notice that

$$\underbrace{\begin{pmatrix} 1 & x_0 & \cdots & x_0^{2k-2} \\ 1 & x_1 & \cdots & x_1^{2k-2} \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_{2k-2} & \cdots & x_{2k-2}^{2k-2} \end{pmatrix}}_{=:V} \cdot \begin{pmatrix} c^{(0)} \\ c^{(1)} \\ \vdots \\ c^{(2k-2)} \end{pmatrix} = \begin{pmatrix} h(x_0) \\ h(x_1) \\ \vdots \\ h(x_{2k-2}) \end{pmatrix} = \begin{pmatrix} h_0 \\ h_1 \\ \vdots \\ h_{2k-2} \end{pmatrix},$$

where $V = \mathrm{Vand}(x_0, \ldots, x_{2k-2})$ is the so-called *Vandermonde-Matrix* of $x_0, \ldots, x_{2k-2}$. Hence, we can compute the coefficients $c^{(i)}$ of $h(x)$ from its values h_j at the $2k-1$ points x_j as

$$\begin{pmatrix} c^{(0)} \\ c^{(1)} \\ \vdots \\ c^{(2k-2)} \end{pmatrix} = V^{-1} \cdot \begin{pmatrix} h_0 \\ h_1 \\ \vdots \\ h_{2k-2} \end{pmatrix}$$

Since k is a constant and since each entry of V is of constant size, only a constant number of primitive operations is needed to compute V^{-1}. Computing the product of V^{-1} and the vector $(h_0, \ldots, h_{2k-2})^t$ needs $O(n)$ primitive operations as each h_j has length $O(n)$. Finally, we compute $c = a \cdot b$ as the sum of the $2k-1$ integers $c_j \cdot B^j$, for $j = 0, \ldots, 2k-2$, which also uses $O(n)$ primitive operations.

In summary, we thus obtain the following recursion for the computation time $T(n)$ of the Toom-Cook-k Algorithm:

$$T(n) \leq (2k-1) \cdot T(\lceil n/k \rceil) + O(n).$$

Again, the Master Theorem yields the following result:

Theorem 1.2.2. *For a fixed integer $k \in \mathbb{N}_{\geq 2}$, the Toom-Cook-$k$ Algorithm uses $O(n^{\frac{\log(2k-1)}{\log k}})$ primitive operations to multiply two n-digit numbers.*

From the above theorem and the fact that $\lim_{k \mapsto \infty} \frac{\log(2k-1)}{\log k} = 1$, we conclude that, for any fixed $\epsilon > 0$, there exists an algorithm with running time $O(n^{1+\epsilon})$ to multiply to n-digit numbers. In the next chapter, will discuss a method due to Schönhage and Strassen (1971) that even yields a running time of size $O(n \cdot \log^c(n))$, with some constant $c > 1$. The method is similar to the Toom-Cook approach in the sense that it considers the input integers as polynomials and then computes the product of the polynomials using an evaluation/interpolation-approach. The main difference however is that n-digits numbers are considered as polynomials of degree $n-1$ (and not k for some fixed constant k) and that the interpolation points are chosen to be the $2n$-th roots of unity. Here, the crucial point is that evaluating and interpolating a polynomial at the roots of unity can be done in a very efficient way.

Exercise 1.2.3. *Show that Karatsuba's method can be considered as a special case of Toom-Cook-2. For this, you need to choose suitable interpolation points x_0, x_1, x_2 in the Toom-Cook-2 algorithm.*

Hint: You may choose $x_0 = \infty$ as one of the interpolation points, where we define $P(\infty) := P_d$ for a polynomial $P(x) = P_0 + \cdots + P_d \cdot x^d$. For the interpolation step, you cannot use the Vandermonde matrix any more but need a more direct approach instead.

Exercise 1.2.4. *For two integers* $a = a^{(0)} + a^{(1)} \cdot B^{\lceil n \rceil} + a^{(3)} \cdot B^{2\lceil n \rceil}$ *and* $b = b^{(0)} + b^{(1)} \cdot B^{\lceil n \rceil} + b^{(3)} \cdot B^{2\lceil n \rceil}$ *of length* n*, use the Toom-Cook-3 approach to derive a relation between the values* $a^{(i)}$ *and* $b^{(i)}$ *that is similar to the relation in (1.1) as considered in Karatsuba's method.*

1.3 Approximate Computation

1.3.1 Fixed Point Arithmetic

A common approach when dealing with non-integer values a (e.g. $1/3$, $\sqrt{2}$, or π) is to approximate them by rational numbers $\tilde{a} = m \cdot B^{-\rho}$, with B the working base, $m \in \mathbb{Z}$ and $\rho \in \mathbb{N}$, such that $|a - \tilde{a}| \leq B^{-\rho+1}$. That is, $\tilde{a}$ constitutes the best approximation of a among all *fixed-point number* with base B and precision ρ:

$$\mathbb{F}_{B,\rho} := \{a = (-1)^s \cdot B^{-\rho} \cdot \sum_{i=0}^{n-1} a_i B^i \text{ with } n \in \mathbb{N},\ s \in \{0,1\},\ \text{and } a_i \in \{0, \ldots, B-1\}\}$$

If B and ρ are clear from the context, we also write $\mathbb{F} = \mathbb{F}_{B,\rho}$. For convenience, we also write

$$a = (-1)^s\ a_{n-1} \ldots a_{\rho+1} a_\rho, a_{\rho-1} \ldots a_0$$

for an arbitrary element $a = (-1)^s \cdot B^{-\rho} \cdot \sum_{i=0}^{n-1} a_i B^i \in \mathbb{F}_{B,\rho}$. The *length of* a (with respect to B) is defined as the number n of digits that is needed to represent a. It is common to consider the base $B = 2$ and to work with so called *dyadic* numbers (also called dyadic rationals). These are exactly the fixed point numbers with respect to base 2 and arbitrary but finite precision:

$$\mathbb{D} := \bigcup_{\rho=0}^{\infty} \mathbb{F}_{2,\rho} = \{p \cdot 2^{-\rho} : p \in \mathbb{Z} \text{ and } \rho \in \mathbb{N}\}.$$

In what follows, we always assume that the base B and the precision ρ is fixed. For an arbitrary real value x, we define

$$\mathrm{flu}(x) := \min\{a \in \mathbb{F} : x \leq a\}$$

and

$$\mathrm{fld}(x) := \max\{a \in \mathbb{F} : x \geq a\}.$$

the two *rounding functions* to the nearest fixed-point number that is larger/smaller than or equal to a. $\mathrm{fl}(.)$ defines the *rounding to nearest*, that is, $\mathrm{fl}(x) = \mathrm{flu}(x)$ if $|\mathrm{flu}(x) - x| < |\mathrm{fld}(x) - x|$ and $\mathrm{fl}(x) = \mathrm{fld}(x)$ if $|\mathrm{fld}(x) - x| < |\mathrm{flu}(x) - x|$. In case of ties (i.e. $|\mathrm{fld}(x) - x| = |\mathrm{flu}(x) - x|$), we round to even, that is, $\mathrm{fl}(x) = \mathrm{flu}(x)$ if the last digit of $\mathrm{flu}(x)$ is even, otherwise $\mathrm{fl}(x) = \mathrm{fld}(x)$. For each arithmetic operations $\circ \in \{+, -, \cdot\}$, we now consider a corresponding approximate variant $\tilde{\circ}$, where we use $\mathrm{fl}(.)$ to round the exact result to a nearby number in $\mathbb{F}$:

Definition 1.3.1. *For* $x, y \in \mathbb{R}$ *and* $\circ \in \{+, -, \cdot\}$*, we define*

$$x \tilde{\circ} y := \mathrm{fl}(\mathrm{fl}(x) \circ \mathrm{fl}(y)).$$

In particular, we have $x \tilde{\circ} y := \mathrm{fl}(x \circ y)$ *for* $x, y \in \mathbb{F}$*.*

Notice that the above definition yields a canonical way of approximately evaluating a polynomial $f(x) = a_0 + \cdots a_d \cdot x^d \in \mathbb{R}[x]$ at an arbitrary real value x. More precisely, we consider some evaluation method (e.g. Horner Evaluation) and replace each of the occurring arithmetic operations $\circ$ by the corresponding fixed point variant $\tilde{\circ}$. We denote the so-obtained result by $f_{\mathbb{F}}(x)$. We remark at this point that the result may crucially depend on the chosen evaluation method. That is, we might get completely different values when using Horner Evaluation instead of the "classical" way of evaluating the polynomial, that is, by first computing all powers x^i of x, then multiplying each power with the corresponding coefficient a_i, and finally summing up the obtained values. In other terms, it does not necessarily hold that

$$a_0 \mathbin{\tilde{+}} x \mathbin{\tilde{\cdot}} (a_1 \mathbin{\tilde{+}} \cdots (a_{d-1} \mathbin{\tilde{+}} x \mathbin{\tilde{\cdot}} a_d) \ldots) = a_0 \mathbin{\tilde{+}} a_1 \mathbin{\tilde{\cdot}} \tilde{x} \mathbin{\tilde{+}} \cdots \mathbin{\tilde{+}} a_d \mathbin{\tilde{\cdot}} x \mathbin{\tilde{\cdot}} x \cdots x \mathbin{\tilde{\cdot}} x$$

Exercise 1.3.2. *Give an example where Horner Evaluation and classical evaluation give different results for $f_{\mathbb{F}}(x)$.*

The above approach for approximately evaluating a univariate polynomial at a point then further extends to polynomials $F(\mathbf{x}) \in \mathbb{R}[\mathbf{x}] = \mathbb{R}[x_1, \ldots, x_n]$ in several variables. Since each complex number z can be written as $z = x + \mathbf{i} \cdot y$ with $x, y \in \mathbb{R}$, and since each addition and multiplication in $\mathbb{C}$ amounts for a constant number of additions and multiplications in $\mathbb{R}$, we may further extend the approach to polynomials with complex coefficients. In this case, the set of *complex fixed point numbers* is given as

$$\mathbb{F}_{\mathbb{C}} := \mathbb{F} + \mathbf{i} \cdot \mathbb{F},$$

and the set of *complex dyadic numbers* is given as

$$\mathbb{D}_{\mathbb{C}} := \mathbb{D} + \mathbf{i} \cdot \mathbb{D}.$$

In the next step, we investigate the error when performing a series of additions and multiplications using fixed point arithmetic. Assume that we are given approximations $\tilde{x}, \tilde{y} \in \mathbb{F}_{\mathbb{C}}$ of two complex numbers $x, y \in \mathbb{C}$ with $|x - \tilde{x}| < \epsilon_x$ and $|x - \tilde{x}| < \epsilon_y$. Then, it holds that

$$|(\tilde{x} \mathbin{\tilde{+}} \tilde{y}) - (x + y)| \leq \sqrt{2} \cdot B^{-(\rho+1)} + |(\tilde{x} + \tilde{y}) - (x + y)| < B^{-\rho} + \epsilon_x + \epsilon_y, \tag{1.2}$$

and the same error bound holds true for subtraction. For multiplication, we have

$$|\tilde{x} \mathbin{\tilde{\cdot}} \tilde{y} - x \cdot y| \leq \sqrt{2} \cdot B^{-(\rho+1)} + |(\tilde{x} \cdot \tilde{y}) - x \cdot y| < B^{-\rho} + \epsilon_x \cdot |y| + \epsilon_y \cdot |x| + \epsilon_x \cdot \epsilon_y. \tag{1.3}$$

From the above error bounds, we can now derive a bound on the error $|f(x_0) - f_{\mathbb{F}}(x_0)|$ that we obtain when using Horner evaluation and fixed point arithmetic to compute the value of a polynomial f at a complex point x_0.

Theorem 1.3.3. *For any $x_0 \in \mathbb{C}$ and any polynomial $f \in \mathbb{C}[x]$ of degree d with coefficients of absolute value less than 2^L, with $L \in \mathbb{Z}_{\geq 0}$, it holds that*

$$|f(x_0) - f_{\mathbb{F}}(x_0)| < 4(d+1)^2 \cdot 2^L \cdot B^{-\rho} \cdot \max(1, |x_0|)^d.$$

if Horner Evaluation and fixed point arithmetic with a precision $\rho \geq \log d$ is used for the evaluation of f at x_0.

Proof. We argue by induction on the degree d of $f = a_0 + \cdots + a_d \cdot x^d$. Obviously, the error bound is true for $d = 0$ as

$$|a_0 - \mathrm{fl}(a_0)| \leq \sqrt{2} \cdot B^{-(\rho+1)},$$

When using Horner evaluation to evaluate a polynomial f of degree $d \geq 1$ at x_0, we first evaluate $\hat{f} := a_1 + a_2 \cdot x + \cdots + a_d \cdot x^{d-1}$ at x_0, then multiply the result by x_0 and eventually add a_0. Using fixed point arithmetic with precision ρ, our induction hypotheses yields that

$$|\hat{f}_{\mathbb{F}}(x_0) - \hat{f}(x_0)| < \epsilon := 4d^2 \cdot 2^L \cdot B^{-\rho} \cdot \max(1, |x_0|)^{d-1}$$

Since $|\hat{f}(x_0)| \leq d \cdot 2^L \cdot \max(1, |x_0|)^{d-1}$ and $|x_0 - \mathrm{fl}(x_0)| \leq \sqrt{2} \cdot B^{-(\rho+1)} < B^{-\rho}$, we conclude from (1.3) that

$$\begin{aligned}
|x_0 \cdot \hat{f}(x_0) - \mathrm{fl}(x_0) \mathbin{\tilde{\cdot}} \hat{f}_{\mathbb{F}}(x_0))| &< \sqrt{2} \cdot B^{-(\rho+1)} + \epsilon \cdot |x_0| + B^{-\rho} \cdot |\hat{f}(x_0)| + B^{-\rho} \cdot \epsilon \\
&< B^{-\rho} + \epsilon \cdot \max(1, |x_0|) + B^{-\rho} \cdot |\hat{f}(x_0)| + \frac{\epsilon \cdot \max(1, |x_0|)}{d} \\
&< B^{-\rho} \cdot [1 + 5d \cdot 2^L \cdot \max(1, |x_0|)^{d-1} + 4d^2 \cdot 2^L \cdot \max(1, |x_0|)^d]. \\
&\leq B^{-\rho} \cdot \max(1, |x_0|)^d \cdot 2^L \cdot (1 + 5d + 4d^2)
\end{aligned}$$

Adding the constant a_0 increases the error by less than $2 \cdot B^{-\rho}$ due to (1.2). Hence, the total error is bounded by

$$B^{-\rho} \cdot 2^L \cdot (3 + 5d + 4d^2) \cdot \max(1, |x_0|)^d \leq 4(d+1)^2 \cdot 2^L \cdot B^{-\rho} \cdot \max(1, |x_0|)^d.$$

Hence, the claim follows. □

1.3.2 Interval Arithmetic

Instead of computing an approximation of the value $f(x_0)$ that a function $f : \mathbb{R} \mapsto \mathbb{R}$ (or more general, $f : \mathbb{C} \mapsto \mathbb{C}$) takes at a specific point $x_0 \in \mathbb{R}$ (or $x_0 \in \mathbb{C}$), it is often useful to compute an approximation of the image $f([a, b])$ (or $f([a, b] + \mathbf{i} \cdot [c, d])$) of an interval $[a, b]$ (rectangle $[a, b] + \mathbf{i} \cdot [c, d]$) under the mapping f.

Definition 1.3.4 (Interval Extensions and Box Functions)**.** *Let $f : \mathbb{R} \mapsto \mathbb{R}$ be an arbitrary function. An* interval extension *$\square f : H \mapsto H$ of f is a function from the halfplane $H := \{[a, b] : a, b \in \mathbb{R}$ with $a \leq b\}$ of intervals $X = [a, b]$ to itself such that $f(x) \in \square f(X)$ for all $x \in X$. For continuous f, $\square f$ is a* continuous interval extension *(or* box-function*) if*

$$\bigcap_{i=1}^{\infty} \square f(X_i) = f(x_0)$$

for any sequence $X_1 \supset X_2 \supset \cdots$ such that $\bigcap_{i=1}^{\infty} X_i$ contains only a single point x_0.

In simpler terms, an interval extension $\square f$ of f is a function that maps an interval $[a, b]$ to an interval $[A, B]$ such that $f(x) \in [A, B]$ for any $x \in [a, b]$. Notice that this is not a very restricting condition as we can simply choose $\square f$ as the function that maps any interval to $(-\infty, +\infty)$. However, for a box function, it must also hold that $[A, B]$ shrinks to one point $(f(x_0))$ if $[a, b]$ shrinks to one point (x_0).

We further remark that Definition 1.3.4 further generalizes to complex valued functions $f : \mathbb{C} \mapsto \mathbb{C}$. Then, an interval extension $\square f : H_{\mathbb{C}} \mapsto H_{\mathbb{C}}$ computes for each rectangle

$R = [a,b] + \mathbf{i} \cdot [c,d] \in H_{\mathbb{C}} := H + \mathbf{i} \cdot H$ a rectangle $\square f(B) \in H_{\mathbb{C}}$ with $f(B) \subset \square f(B)$. The definition of a box function is also completely analogous to the real case. We now show how to compute a box-function for a polynomial. For this, we introduce the concept of interval-arithmetic.

Definition 1.3.5 (Interval Arithmetic)**.** *Let $[a,b]$ and $[c,d]$ be arbitrary intervals and λ a non-negative real number. Then, we define*

$$\begin{aligned}
\lambda \cdot [a,b] &:= [\lambda \cdot a, \lambda \cdot b] \\
-[a,b] &:= [-b,-a] \\
[a,b] \boxplus [c,d] &:= [a+c, b+d] \\
[a,b] \boxminus [c,d] &:= [a,b] \boxplus [-c,-d], \text{ and} \\
[a,b] \boxdot [c,d] &:= [\min(ab,bd,ad,bc), \max(ab,bd,ad,bc)]
\end{aligned}$$

The above rules then extend to arithmetic operations on rectangles in $\mathbb{C}$ in a straight forward way. In particular, for $R = [a,b] + \mathbf{i} \cdot [c,d]$ and $R' := [a',b'] + \mathbf{i} \cdot [c',d']$, we have

$$\begin{aligned}
R \boxplus R' &:= [a,b] \boxplus [a',b'] + \mathbf{i} \cdot ([c,d] \boxplus [c',d']), \\
R \boxdot R' &:= [a,b] \boxdot [a',b'] \boxminus [c,d] \boxdot [c',d'] + \mathbf{i} \cdot ([a,b] \boxdot [c',d'] \boxplus [a',b'] \boxdot [c,d]).
\end{aligned}$$

Often, we have to restrict to fixed point arithmetic instead of exact arithmetic. Similar to the definition of fl(.), which rounds a real (or complex) value to its best approximation in $\mathbb{F}$ (or $\mathbb{F}_{\mathbb{C}}$), we introduce the following rounding function for intervals (rectangles in $\mathbb{C}$):

$$\mathrm{Fl} : H_{\mathbb{C}} \mapsto H_{\mathbb{C}} : \mathrm{Fl}([a,b] + \mathbf{i} \cdot [c,d]) := [\mathrm{fld}(a), \mathrm{flu}(b)] + \mathbf{i} \cdot [\mathrm{fld}(c), \mathrm{flu}(d)]$$

Hence, Fl(.) rounds each of the vertices of a rectangle B to the nearest corresponding approximations in $\mathbb{F}_{\mathbb{C}}$ such that $\mathrm{Fl}(B)$ contains B. We can now define arithmetic operations on intervals (rectangles) using fixed point arithmetic.

Definition 1.3.6 (Fixed Point Interval Arithmetic)**.** *Let $[a,b]$ and $[c,d]$ be arbitrary intervals with $a,b,c,d \in \mathbb{F}$ and $\lambda \in \mathbb{R}$ a non-negative real number. Then, we define*

$$\begin{aligned}
[a,b] \tilde{\boxplus} [c,d] &:= \mathrm{Fl}([a,b] \boxplus [c,d]) \\
[a,b] \tilde{\boxminus} [c,d] &:= \mathrm{Fl}([a,b] \boxplus [-d,-c]), \\
[a,b] \tilde{\boxdot} [c,d] &:= \mathrm{Fl}([a,b] \boxdot [c,d]), \text{ and} \\
\lambda \tilde{\boxdot} [a,b] &:= [\mathrm{fld}(\lambda), \mathrm{flu}(\lambda)] \tilde{\boxdot} [a,b]
\end{aligned}$$

Again, the above rules for arithmetic operations on intervals extend in a straight forward manner to rectangles in $\mathbb{C}$. In addition, they induce interval extensions $\square f$ and $\tilde{\square} f$ for a polynomial $f \in \mathbb{R}[x]$. For this, we replace each arithmetic operation $\circ$ in the evaluation of f (e.g. when using Horner Evaluation) by the corresponding interval variant $\boxdot$ and $\tilde{\boxdot}$, respectively. Notice that $\square f$ is a box-function, whereas this is not true for $\tilde{\square} f$.

Exercise 1.3.7. *For any $x \in \mathbb{R}$ with $0 \leq x \leq 1$ and $k \in \mathbb{N}$, there exists a $\xi \in [0,x]$ such that*

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \cdots + \frac{x^{4k}}{(4k)!} \cdot \cos(\xi) \quad \text{(Taylor Series Expansion with Remainder Term)}$$

Use the above formula to derive a box function $\square \cos$ for $\cos$ for intervals $[a,b] \subset [0,1]$! Can you extend your approach to derive a box function for $\sin x$ and e^x.

Exercise 1.3.8. *Let $f(x) = a_0 + a_1 x + \cdots + a_d x^d \in \mathbb{Z}[x]$ be an arbitrary polynomial with integer coefficients. Our goal is to count all real roots of f, provided that f has only simple roots.*

(a) Show that all real roots of f have absolute value bounded by $M := 1 + \max_{0 \le i < d} |\frac{a_i}{a_n}|$.[2]

(b) Use box functions for f and its derivative f' to derive a method that allows you to decide whether a certain interval I contains no root or exactly one root. Your method may fail (with the output "I don't know"); however, it should succeed for sufficiently small intervals I.

(c) Formulate an algorithm to determine the number of real roots of f.

(Hint: By Rolle's theorem, any interval I which contains more than one root of f also contains a root of its derivative f'.)

We now investigate a bound on the size of the intervals (rectangles) that are obtained when performing a series of additions and multiplication according to the above rules. Notice that there are similarities to our considerations in the previous section, where we derived bounds on the error that occurs when adding or multiplying numbers using fixed point arithmetic. Namely, you might think of two rectangles $R := [a,b] + \mathbf{i} \cdot [c,d]$ and $R' := [a',b'] + \mathbf{i} \cdot [c',d']$ as approximations of its centers $m_R := \frac{a+b}{2} + \mathbf{i} \cdot \frac{c+d}{2}$ and $m_{R'} := \frac{a'+b'}{2} + \mathbf{i} \cdot \frac{c'+d'}{2}$ up to an error of size at most $\epsilon := \sqrt{2} \cdot w(R)$ and $\epsilon' := \sqrt{2} \cdot w(R')$, respectively, where $w(R) = \max(b-a, d-c)$ and $w(R') = \max(b'-a', d'-c')$ are defined as the *width* of R and R'. Then, the output of an arithmetic operation between R and R' can again be considered as an approximation of the corresponding arithmetic operation between m_R and $m_{R'}$. Hence, similarly to the bounds in (1.2) and (1.3), we obtain for any two rectangles R and R' with vertices in $\mathbb{F} + \mathbf{i} \cdot \mathbb{F}$ that

$$w(R \boxplus R') \le w(R \tilde{\boxplus} R') \le w(R) + w(R') + 2 \cdot B^{-\rho} \tag{1.4}$$

and

$$w(R \boxdot R') \le w(R) \cdot w(R') + |m_R| \cdot w(R') + |m_{R'}| \cdot w(R) + 2 \cdot B^{-\rho}. \tag{1.5}$$

Exercise 1.3.9. *Prove correctness of the inequalities in (1.4) and (1.5).*

Exercise 1.3.10. *Let $f \in \mathbb{C}[x]$ be a polynomial of degree d with coefficients of absolute value less than 2^L, with $L \in \mathbb{Z}_{\ge 0}$, let $\rho \in \mathbb{N}$ be a precision with $\rho > \log d$, and let $\mathbb{F}$ the corresponding set of fixed point numbers with precision ρ. Let $R = [a,b] + \mathbf{i} \cdot [c,d]$ be a rectangle of width $w(R) < \frac{1}{d}$ with vertices in $\mathbb{F} + \mathbf{i} \cdot \mathbb{F}$, and suppose that we compute $\square f$ (and $\tilde{\square} f$) using Horner Evaluation and fixed point interval arithmetic with a precision ρ. Then, it holds*

$$w(\square f(R)) \le w(\tilde{\square} f(R)) < 8 \cdot (d+1)^2 \cdot 2^L \cdot \max(1, |m_R|)^d \cdot w(R). \tag{1.6}$$

Hint: Consider a similar argument as in the proof of Theorem 1.3.3.

Notice that the bound (1.6) on $w(\square f(R))$ and $w(\tilde{\square} f(R))$ tends to zero if we consider a rectangle (square) R of width $c \cdot B^{-\rho}$, for some constant c, and the precision ρ tends to ∞. Hence, in order to compute an approximation of $f(x_0)$ for some complex value x_0, we may

[2] The bound M is also called *Cauchy's Root Bound* in the literature.

first approximate x_0 by some fixed point number $\tilde{x}_0 = \tilde{x}_{0,\Re} + \mathbf{i} \cdot \tilde{x}_{0,\Im} \in \mathbb{F}_{B,\rho} + \mathbf{i} \cdot \mathbb{F}_{B,\rho}$ such that $|x_0 - \tilde{x}_0| \leq B^{-\rho}$ and consider a rectangle

$$R := [\tilde{x}_{0,\Re} - B^{-\rho}, \tilde{x}_{0,\Re} + B^{-\rho}] + \mathbf{i} \cdot [\tilde{x}_{0,\Im} - B^{-\rho}, \tilde{x}_{0,\Im} + B^{-\rho}]$$

of width $2B^{-\rho}$ whose vertices are obtained by adding and subtracting $B^{-\rho}$ from the real and complex part of $\tilde{x}_0$. Then, R contains x_0 and we can use use interval arithmetic to compute the rectangle $\tilde{\square} f(R)$, which contains $f(x_0)$. Its center m constitutes an approximation of $f(x_0)$ with $|m - f(x_0)| < w(\tilde{\square} f(R))$. Hence, for computing an approximation m with $|m - f(x_0)| < \epsilon$, we can iteratively compute $\tilde{\square} f(R)$ with increasing precision $\rho = 1, 2, 4, 8, \ldots$ until $w(\tilde{\square} f(R)) < \epsilon$, and then return the center of $\tilde{\square} f(R)$. Exercise 1.3.10 guarantees that we must succeed as soon as the precision ρ fulfills the inequality

$$\rho > \rho_\epsilon := \log_B[16(d+1)^2 \cdot 2^L \cdot \max(1, |x_0|)^d \cdot \epsilon^{-1}] = O(\log d + d \log \max(1, |x_0|) + L + |\log \epsilon|),$$

where we used that

$$\max(1, |m_R|)^d \leq \max(1, |x_0| + B^{-\rho}) \leq \max(1, |x_0|)^d \cdot (1 + 1/d^2)^d \leq 2 \max(1, |x_0|)^d$$

for any $\rho > 2 \log d$. Since we double ρ in each step, this shows that we succeed for a precision $\rho < 2\rho_\epsilon$. We fix this result, which will turn out to be useful at several places in the following considerations.

Theorem 1.3.11. *Let $f \in \mathbb{C}[x]$ be a polynomial of degree d with coefficients of absolute value less than 2^L, with $L \in \mathbb{Z}_{\geq 0}$, and let x_0 be an arbitrary complex value. For any non-negative integer ℓ, we can compute an approximation $\tilde{y}_0$ of $y_0 = f(x_0)$ with $|y_0 - \tilde{y}_0| < 2^{-\ell}$ using fixed point interval arithmetic with a precision ρ bounded by*

$$O(\log d + d \log \max(1, |x_0|) + L + \ell).$$

Notice that the above bound on ρ that is needed in the worst-case is also a (worst-case) bound on the input precision as, in each iteration, we need approximations of the coefficients of f as well as of x_0 to an error less than $B^{-\rho}$. We further remark that, as an alternative to the above approach, one could also use fixed point arithmetic directly to compute an approximation of $f(x_0)$, and to estimate the occurring error using Theorem 1.3.3. This yields a comparable bound on the needed precision in the worst case. However, the main drawback of this approach is that one has to work with an a priori computed worst-case error bound, which means that the needed precision is always of size $\Omega(\log d + d \log \max(1, |x_0|) + L + \ell)$. In contrast, when using interval arithmetic with increasing precision, we might already succeed with a much smaller precision.

Exercise 1.3.12. *Suppose that a polynomial $f \in \mathbb{R}[x]$ as well as a real value x_0 is given by means of an oracle that returns arbitrary good dyadic approximations of the coefficients of f and x_0. Under the assumption that $f(x_0) \neq 0$, formulate an algorithm that computes an $\ell \in \mathbb{Z}$ such that $2^{-\ell} < |f(x_0)| < 2^{\ell+2}$. How does its running time depend on $|f(x_0)|$?*

1.3.3 Floating point arithmetic (Under construction)

When actually implementing algorithms, the standard approach for the approximate computation with real (complex) numbers is NOT fixed point arithmetic but *floating point arithmetic*.

However, a corresponding error analysis is more delicate, and thus, for the seek of simplicity, we decided to use fixed point arithmetic as our main tool for approximate computation. Nevertheless, we give a self-contained introduction for the interested reader. It originally appeared in the appendix of [MOS11].

Hardware floating point arithmetic is standardized in the IEEE floating point standard[3]. A floating point number is specified by a sign s, a mantissa m, and an exponent e. The sign is $+1$ or -1. The mantissa consists of ρ bits $m_1, \ldots, m_\rho$, and e is an integer in the range $[e_{min}, e_{max}]$. The range of possible exponents contains zero and $e_{min} \leq -\rho - 2$. The number represented by the triple (s, m, e) is as follows:

- If $e_{min} < e \leq e_{max}$, the number is $s \cdot (1 + \sum_{1 \leq i \leq \rho} m_i 2^{-i}) \cdot 2^e$. This is called a *normalized* number.
- If $e = e_{min}$, then the number is $s \cdot \sum_{1 \leq i \leq \rho} m_i 2^{-i} 2^{e_{min}+1}$. This is called a *subnormal* number. Observe that the exponent is $e_{min} + 1$. This is to guarantee that the distance between the largest subnormal number $(1 - 2^{-\rho})2^{e_{min}+1}$ and the smallest normalized number $1 \cdot 2^{e_{min}+1}$ is small.
- In addition, there are the special numbers $-\infty$ and $+\infty$ and a symbol NaN which stands for not-a-number. It is used as an error indicator, e.g., for the result of a division by zero.

Let $\mathbb{F} = \mathbb{F}(\rho, e_{min}, e_{max})$ be the set of real numbers (including $+\infty$ and $-\infty$) that can be represented as above.[4] A real number in $\mathbb{F}$ is called *representable*, a number in $\mathbb{R} \setminus \mathbb{F}$ is called *non-representable*. The largest positive representable number (except for ∞) is $max_{\mathbb{F}} = (2 - 2^{-\rho}) \cdot 2^{e_{max}}$, the smallest positive representable number is $min_{\mathbb{F}} = 2^{-\rho} \cdot 2^{e_{min}+1} = 2^{-\rho+e_{min}+1}$, and the smallest positive normalized representable number is $mnorm_F = 1 \cdot 2^{e_{min}+1} = 2^{e_{min}+1}$.

$\mathbb{F}$ is a discrete subset of $\mathbb{R}$. For any real x, let $\mathrm{fl}(x)$ be a floating point number closest[5] to x. By convention, if $x > max_{\mathbb{F}}$, $\mathrm{fl}(x) = \infty$, and if $x < -max_{\mathbb{F}}$, $\mathrm{fl}(x) = -\infty$. As for fixed point arithmetic, arithmetic on floating point numbers is only approximate. Again, we distinguish between a mathematical operation $\circ \in \{-, +, \cdot\}$ and the corresponding floating point implementation $\tilde{\circ}$. We further use $^{1/2}$ for the square-root operation and $\sqrt{\ }$ for its floating point implementation. *The floating point implementations of the operations $+$, $-$, $\cdot$, and $^{1/2}$ yield the best possible result.* This is an axiom of floating point arithmetic. That is, if $x, y \in \mathbb{F}$ and $\circ \in \{+, -, \cdot\}$, then

$$x \tilde{\circ} y = \mathrm{fl}(x \circ y)$$

and

$$\sqrt{x} = \mathrm{fl}(x^{1/2}).$$

We need bounds on the error in the floating point evaluation of simple arithmetic expressions. Any real constant or variable is an arithmetic expression, and if A and B are arithmetic

[3] IEEE standard 754-1985 for binary floating-point arithmetic, 1987.

[4] Double precision floating point numbers are represented in 64 bits. One bit is used for the sign, 52 bits for the mantissa ($\rho = 52$) and 11 bits for the exponent. These 11 bits are interpreted as an integer $f \in [0...2^{11} - 1] = [0...2047]$. The exponent e equals $f - 1023$; $f = 2047$ is used for the special values, and hence $e_{min} = -1023$ and $e_{max} = 1023$. The rules for $f = 2047$ are: If all m_i are zero and $f = 2047$, then the number is $+\infty$ or $-\infty$ depending on s. If $f = 2047$ and some m_i is nonzero, the triple represents NaN (= not a number).

[5] The IEEE-standard also specifies how to break ties. This is of no concern here.

E	condition	$\tilde{E}$	m_E	ind_E	c_E	$\deg E$
a	constant in $\mathbb{R} \setminus \mathbb{F}$	$\mathrm{fl}(a)$	$\max(mnorm_F, \lvert\mathrm{fl}(a)\rvert)$	1	$\max(1, \lvert\mathrm{fl}(a)\rvert)$	0
a	constant in $\mathbb{F}$	a	$\max(mnorm_F, \lvert a\rvert)$	0	$\max(1, \lvert a\rvert)$	0
x	var. ranging over $\mathbb{R}$	$\mathrm{fl}(x)$	$\max(mnorm_F, \lvert\mathrm{fl}(x)\rvert)$	1	1	1
x	var. ranging over $\mathbb{F}$	x	$\max(mnorm_F, \lvert x\rvert)$	0	1	1
$A + B$		$\tilde{A} \oplus \tilde{B}$	$m_A \oplus m_B$	$1 + \max(ind_A, ind_B)$	$c_A + c_B$	$\max(\deg A, \deg B)$
$A - B$		$\tilde{A} \ominus \tilde{B}$	$m_A \oplus m_B$	$1 + \max(ind_A, ind_B)$	$c_A + c_B$	$\max(\deg A, \deg B)$
$A \cdot B$		$\tilde{A} \odot \tilde{B}$	$\max(mnorm_F, m_A \odot m_B)$	$1 + ind_A + ind_B$	$c_A c_B$	$\deg A + \deg B$
$A^{1/2}$	$\tilde{A} < \mathbf{u} m_A$	0	$2^{(t+1)/2}\sqrt{m_A}$	$2 + ind_A$	not defined	
$A^{1/2}$	$\tilde{A} \geq \mathbf{u} m_A$	$\sqrt{\tilde{A}}$	$\max(\sqrt{\tilde{A}}, m_A \oslash \sqrt{\tilde{A}})$	$2 + ind_A$	not defined	

Table 1.1: The recursive definitions of m_E, ind_E, c_E and $\deg E$. The first two columns specify the case distinction according to the syntactic structure of E, the third column contains the rule for computing $\tilde{E}$, and the fourth to seventh columns contain the rules for computing m_E, ind_E, c_E and $\deg E$; $\oplus$, $\ominus$, and $\odot$ denote the floating point implementations of addition, subtraction, and multiplication, and $\sqrt{}$ denotes the floating point implementation of the square-root operation. Observe that $m_E = \infty$ if either $m_A = \infty$ or $m_B = \infty$.

expressions, then so are $A + B$, $A - B$, $A \cdot B$, and $A^{1/2}$. The latter assumes that the value of A is non-negative. For an arithmetic expression E, let $\tilde{E}$ be the result of evaluating E with floating point arithmetic. The quantity $\mathbf{u} = 2^{-\rho-1}$ is called *unit of roundoff*. Table 1.1 gives recursive definitions of quantities m_E, ind_E, c_E and $\deg E$; we bound $|E - \tilde{E}|$ in terms of them. Intuitively, m_E is an upper bound on the absolute value of E, ind_E measures the complexity of the syntactic structure of E, $\deg E$ is the degree of E when interpreted as a polynomial, and c_E bounds the coefficient size when E is interpreted as a polynomial.

Theorem 1.3.13. *If* $ind_E \leq 2^{(\rho+1)/2} - 1$, *then*

$$|E-\tilde{E}| \leq (ind_E+1)\cdot\mathbf{u}\cdot m_E \leq (ind_E+2)\odot\max(mnorm_F, m_E\odot\mathbf{u}) \leq (ind_E+3)\cdot\max(mnorm_F, m_E\cdot\mathbf{u}),$$

where ind_E *and* m_E *are defined as in Table 1.1.*

The error bound of Theorem 1.3.13 is only used for guards. For the analysis we use a simpler, but weaker bound. It applies to polynomial expressions, i.e., expressions using only constants, variables, additions, subtractions, and multiplications.

Theorem 1.3.14. *For a polynomial expression we have* $m_E \leq c_E M^{\deg E}$, *where* m_E, c_E *and* $\deg E$ *are defined as in Table 1.1 and* M *is the smallest power of two with*

$$M \geq \max(1, \max\{|x| : x \text{ is a variable in } E\}).$$

This assumes that $c_E M^{\deg E}$ *is representable.*

We next specialize the theorem above to polynomial expressions that are sums of products, i.e., that correspond to the standard representation of polynomials. We consider polynomials in k variables z_1 to z_k. For $\alpha = (\alpha_1, \ldots, \alpha_k)$ let $z^\alpha = z_1^{\alpha_1} \cdots z_k^{\alpha_k}$. Any polynomial f in $\mathbb{R}[z_1, \ldots, z_k]$ can then be written as

$$f(z_1, \ldots, z_k) = \sum_\alpha f_a z^\alpha,$$

where f_α is the coefficient of the monomial term z^α. For simplicity assume that the coefficients are representable as floating point numbers. For a monomial term, $Z = f_\alpha z^\alpha$, we have $c_Z = \max(1, |f_\alpha|)$, $\deg Z = \deg(z^\alpha) = \sum_i \alpha_i$, and $ind_Z = 2 \deg Z$. For the entire polynomial, we have $c_f = \sum_\alpha \max(1, |f_\alpha|)$ and $\deg f$ equal to the total degree of f. The index depends on the order in which we add the monomial terms. If we sum serially, as in $((((t_1+t_2)+t_3)+t_4)+t_5))$, the index is the number of monomial terms minus one plus the largest index of any monomial term. If we sum in the form of a binary tree as in $((t_1+t_2)+((t_3+t_4)+t_5))$, the index is the logarithm of the number of monomial terms rounded upwards plus the largest index of any monomial term.

Theorem 1.3.15. *Let $f(z_1, \ldots, z_k) = \sum_\alpha f_a x^\alpha$ be a polynomial of total degree N. Let $c_f = \sum_\alpha \max(1, |f_\alpha|)$ and let $m_f = |\{\alpha : f_\alpha \neq 0\}|$ be the number of monomial terms in f. Let $M \geq 1$ be a power of two and let z_1 to z_k be real values with $|z_i| \leq M$ for all i. Then*

$$|f(z_1, \ldots, z_k) - \tilde{f}(\mathrm{fl}(z_1), \ldots, \mathrm{fl}(z_k))| \leq c_f(m_f + 2N)M^N 2^{-\rho-1},$$

where $\tilde{f}$ is the floating point version of f, i.e., all operations in f are replaced by their floating point counterpart.

Proof. We use Theorems 1.3.13 and 1.3.14. The index is largest if the monomial terms are summed serially. It is then equal to $m_f + 2N - 1$. Also $m_E \leq c_f M^N$. □

The above theorem also generalizes to complex values x_i and polynomials defined over the complex numbers. The obtained error bound is comparable, that is, it only differs by a multiplicative constant from the above bound.

1.4 Division

In the previous sections, we have shown how to efficiently carry out additions and multiplications on integers. We also considered corresponding operations on fixed-point numbers and intervals and estimated the error that occurs when using approximate instead of exact arithmetic. So far, any such treatment for the division of integers or fixed-/floating-point numbers a and b is missing. We will first show how to compute an arbitrary good dyadic approximation $\tilde{q} \in \mathbb{D}$ of a rational number $q := \frac{a}{b} \in \mathbb{Q}$ using only additions and multiplications of integers. We start with the special case, where $a = 1$ and b is a positive integer of length less than n. The crucial idea underlying the approach is to consider q as the unique solution of the equation $f(x) := \frac{1}{x} - b = 0$ and to use the Newton-Raphson method to derive an approximation of q. That is, with $x_0 := 2^{-\lceil \log b \rceil} \in \mathbb{D}$, we define

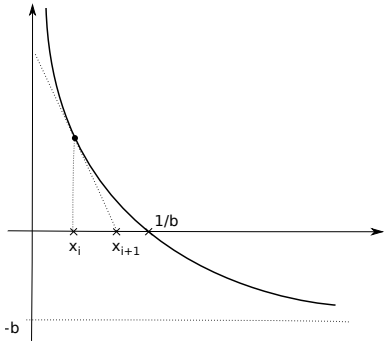

Figure 1.1: The graph of the function $f(x) = \frac{1}{x} - b$. The value x_{i+1} results from applying one step of the Newton-Raphson method to x_i.

$$x_{i+1} := x_i - \frac{f(x_i)}{f'(x_i)} = x_i - \frac{\frac{1}{x_i} - b}{-\frac{1}{x_i^2}} = 2 \cdot x_i - b \cdot x_i^2 = x_i \cdot (2 - b \cdot x_i) \in \mathbb{D} \quad \text{for } i \in \mathbb{N}_{\geq 1}. \quad (1.7)$$

Algorithm 5: Division

Input : Two non-negative n-digit integers a and b and a non-negative integer L.
Output: A dyadic number $\tilde{q} \in \mathbb{D}$ of length $O(n+L)$ such that $|\tilde{q} - a/b| < 2^{-L}$.

$L' := \lceil \log a \rceil + L + 1$
$N := \lceil \log L' \rceil$
$x_0 := 2^{-\lceil \log b \rceil} \cdot (2 - b \cdot 2^{-\lceil \log b \rceil})$
for $i = 1, \ldots, N-1$ **do**
    Recursively define

$$x_{i+1} := \mathrm{fl}(x_i \cdot (2 - b \cdot x_i)),$$

    where fl(.) is defined as "rounding to the nearest element" in $\mathbb{F}_{2,\rho_i}$ and $\rho_i := 2^{i+1} + 2n$.
Compute $\tilde{q} := \mathrm{fl}(a \cdot x_{N-1})$, where fl(.) is defined as rounding to the nearest in $\mathbb{F}_{2,L}$.
return $\tilde{q}$

The first part of the following exercise shows that the sequence x_i converges *quadratically* to q. Roughly speaking, this means that the number of correct digits doubles in each iteration. We then conclude that, after $\lceil \log L \rceil$ iterations, we have computed a dyadic approximation $\tilde{q}$ of $q = 1/b$ with $|q - \tilde{q}| < 2^{-L}$. However, there is a small problem with this approach, namely, the lengths of the dyadic numbers x_i double in each iteration, and since x_0 has length $\lceil \log B \rceil \leq n$, we end up with dyadic numbers of length $O(nL)$ after $\lceil \log L \rceil$ iterations. In Part (c) of the exercise, we show that we can improve upon this approach by rounding the result obtained in the i-th iteration to the ρ_i-th digit after the binary point, with $\rho_i := 2^{i+1} + 2n$. As a result, we can reduce the length of the occurring numbers from $O(nL)$ to $O(n+L)$.

Exercise 1.4.1. *Let $(x_i)_i$ be defined as above and L be an arbitrary positive number. Show that, for all i, it holds that*

(a) $\left|x_{i+1} - \frac{1}{b}\right| \leq b \cdot \left|x_i - \frac{1}{b}\right|^2$ *and*

(b) $\left|x_i - \frac{1}{b}\right| < \frac{1}{b} \cdot 2^{-2^i}$. *In particular, it holds that* $\left|x_i - \frac{1}{b}\right| < 2^{-L}$ *for all* $i \geq \log L$.

(c) Suppose now that we start with $y_0 := x_1 = 2^{-\lceil \log b \rceil} \cdot (2 - b \cdot 2^{-\lceil \log b \rceil})$ *and define*

$$y_{i+1} := \mathrm{fl}(y_i \cdot (2 - b \cdot y_i)) \quad \text{for } i \in \mathbb{N}_{\geq 1},$$

where we consider rounding to the nearest fixed-point number of precision $\rho_i := 2^{i+1} + 2n$. *Then, it holds* $\left|y_i - \frac{1}{b}\right| < \frac{1}{b+1} \cdot 2^{-2^i}$ *for all* i.

Hint: For (c), use that the error $2^{-\rho_i - 1}$ that is induced by the rounding in the $(i+1)$-st iteration is smaller than $\frac{2^{-2^{i+1}}}{(b+1)^2}$. Then, use induction on i to prove the claim.

From the above consideration, we conclude that we can compute a dyadic number $\tilde{q}$ with $|\tilde{q} - 1/b| < 2^{-L}$ using $O(\log L)$ additions and multiplications of integers of length $O(L+n)$. Now, computing a corresponding approximation $\tilde{q}$ of $q := \frac{a}{b}$, with integers a and b of length less than n, is straightforward; see Algorithm 5. Namely, we first compute a dyadic q' of length $O(L+n)$ such that $|q' - 1/b| < 2^{-L - \lceil a \rceil - 1}$ and then determine the product $a \cdot q'$. The result

is eventually rounded to the L-th digit after the binary point. The so-obtained $\tilde{q} = \mathrm{fl}(a \cdot q')$ has length $O(n+L)$ and it holds that $|\tilde{q} - q| < 2^{-L}$. We fix this result:

Theorem 1.4.2. *Let a and b be integers of length n. For any non-negative L, Algorithm 5 computes a dyadic approximation $\tilde{q} \in \mathbb{D}$ of length $O(n+L)$ such that $|\tilde{q} - q| < 2^{-L}$. For this, it uses $O(\log(n+L))$ additions and multiplications of $O(n+L)$-digit integers.*

We can now go one step further and derive a bound on the cost for computing an approximation of the quotient of two arbitrary complex numbers $a = a_0 + \mathbf{i} \cdot a_1$ and $b = b_0 + \mathbf{i} \cdot b_1$. Here, we assume that, for any $L' \in \mathbb{N}$, we can ask for dyadic approximations $\tilde{a}, \tilde{b} \in \mathbb{D}$ such that $|a - \tilde{a}|, |b - \tilde{b}| < 2^{-L'}$. Notice that

$$\frac{a}{b} = \frac{a_0 + \mathbf{i} \cdot a_1}{b_0 + \mathbf{i} \cdot b_1} = \frac{(a_0 + \mathbf{i} \cdot a_1) \cdot (b_0 + \mathbf{i} \cdot b_1)}{(b_0 + \mathbf{i} \cdot b_1) \cdot (b_0 - \mathbf{i} \cdot b_1)} = \frac{(a_0 b_0 - a_1 b_1) + \mathbf{i} \cdot (a_1 b_0 + a_0 b_1)}{|b|^2},$$

thus we can restrict to quotients of real numbers $a, b \in \mathbb{R}_{\neq 0}$. Suppose that dyadic approximations $\tilde{a}, \tilde{b} \in \mathbb{R}_{\neq 0}$ with $|a - \tilde{a}|, |b - \tilde{b}| < 2^{-L'} < |b|/2$ are given. Then, we have

$$\left| \frac{\tilde{a}}{\tilde{b}} - \frac{a}{b} \right| = \left| \frac{b\tilde{a} - a\tilde{b}}{b\tilde{b}} \right| = \frac{\left| b(\tilde{a} - a) - a(b - \tilde{b}) \right|}{\left| b^2 + b(b - \tilde{b}) \right|} < 2^{-L'+1} \cdot \frac{|a| + |b|}{|b|^2} \leq 2^{-L'+2} \cdot \frac{\max(|a|, |b|)}{\min(1, |b|)^2}.$$

For $L' > L + \lceil \log \max(1, |a|) \rceil + 3\lceil |\log b| \rceil + 3$, this implies that $\left| \frac{\tilde{a}}{\tilde{b}} - \frac{a}{b} \right| < 2^{-L-1}$. Hence, we may first consider L'-digit approximations $\tilde{a}, \tilde{b} \in \mathbb{D}$ of a and b, and then compute an $(L+1)$-digit approximation $\tilde{q} \in \mathbb{D}$ of their quotient $q = \frac{\tilde{a}}{\tilde{b}}$ using the method from above. Then, it holds that $|\tilde{q} - a/b| < 2^{-L}$. We fix this result:

Theorem 1.4.3. *Let $a, b \in \mathbb{C}$ be arbitrary complex numbers and $L \in \mathbb{N}$. Then, there exists a positive integer L' of size*

$$L' := O(L + \lceil \log \max(1, |a|) \rceil + \lceil |\log |b|| \rceil)$$

such that we can compute a fixed point number $\tilde{q} \in \mathbb{F} + \mathbf{i} \cdot \mathbb{F}$ of length L' with $|\tilde{q} - q| < 2^{-L}$ using $O(\log L')$ additions and multiplications of $O(L')$-digit integers. The values a and b need to be approximated to an error of size $2^{-L'}$.

Exercise 1.4.4. *For arbitrary $x \in \mathbb{R}$ with $0 \leq x \leq 1$, it holds that*

$$\arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \cdots \tag{1.8}$$

Now, for given $L \in \mathbb{N}$, use the above formula and the fact (due to Euler) that

$$\pi = 20 \cdot \arctan(1/7) + 8 \cdot \arctan(3/79)$$

to derive an efficient algorithm (i.e. with a running time polynomial in L) for computing a fixed point approximation $\tilde{\pi}$ (wrt. base 2) of π to an error less than 2^{-L}.

Hint: Estimate the error when considering only the first k summands in (1.8). Then, proceed with a suitably truncated series.

Exercise 1.4.5. *For arbitrary $x \in \mathbb{R}$ with $0 \leq x \leq 1$, we have*

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \cdots$$

For fixed $n \in \mathbb{N}_{\geq 8}$ and arbitrary $L \in \mathbb{N}$, formulate an efficient method to compute an L-digit approximation $\tilde{\omega}$ of $\omega := \cos(2\pi/n)$.

Hint: Proceed similar as in Exercise 1.4.4 and use a sufficiently good approximation $\tilde{\pi}$ of π. For the evaluation of the truncated series at $x = \tilde{\pi}$, use Theorem 1.3.3.

Chapter 2

The Fast Fourier Transform and Fast Polynomial Arithmetic

2.1 Schönhage-Strassen Multiplication

In the previous chapter, we have seen that the cost $M(n)$ for computing the product of two integers of length n is bounded by $O(n^{1+\epsilon})$, where ϵ is a an arbitrary but fixed positive real value. For sufficiently large k, this bound is achieved by the Toom-Cook-k algorithm. In this section, we present a method [SS71] due to Schönhage and Strassen whose running time is bounded by[1] $O(n \log n \cdot M(\log n)) = O(n \log^{2+\epsilon} n)$. Before we go into detail, we give an overview of the main steps.

2.1.1 The Algorithm in a Nutshell

In the first step, we split a and b into n blocks $a^{(i)}$ and $b^{(i)}$, that is, we write

$$\begin{aligned} a &= a^{(0)} + a^{(1)} \cdot B + \cdots + a^{(n-1)} \cdot B^{n-1}, \quad \text{and} \\ b &= b^{(0)} + b^{(1)} \cdot B + \cdots + b^{(n-1)} \cdot B^{n-1} \end{aligned}$$

with one-digit numbers a^i, $b^{(i)} \in \{0, \ldots, B-1\}$. Notice the difference to the Toom-Cook algorithm, where we split a and b into only constantly many (i.e. k) blocks of size $\lceil n/k \rceil$. Similar to the Toom-Cook method, we now consider corresponding polynomials

$$\begin{aligned} f(x) &:= a^{(0)} + a^{(1)} \cdot x + \cdots + a^{(n-1)} \cdot x^{n-1}, \quad \text{and} \\ g(x) &:= b^{(0)} + b^{(1)} \cdot x + \cdots + b^{(n-1)} \cdot x^{n-1} \end{aligned} \tag{2.1}$$

of degree $n-1$ (instead of k as in the Toom-Cook method) with coefficients $a^{(i)}$ and $b^{(i)}$, and reduce the computation of $a \cdot b$ to the problem of computing the product $h = \sum_{i=0}^{2n-2} c^{(i)} \cdot x^i := f \cdot g$ of the polynomials f and g, followed by the evaluation of h at $x = B$. For the computation of h, we again use an evaluation/interpolation approach, that is, we first evaluate f and g at $2n$ points $x_0, \ldots, x_{2n-1}$, compute each of the products $f(x_i) \cdot g(x_i) = h(x_i)$, and then

[1] We remark that there exists a slightly more involved variant of the Schönhage-Strassen method that needs only $O(n \log n \log\log n)$ primitive operations. For the seek of simplicity, we decided to only present the variant with slightly worse running time but hint to the faster approach when discussing the corresponding steps in more detail.

reconstruct h from its values at the points x_i. The crucial part of the algorithm is the special choice of the points x_i, that is, instead of considering arbitrary distinct values for the points x_i, we now choose $x_i = \omega^i$ for $i = 0, \ldots, 2n-1$, where $\omega \in \mathbb{C}$ is a *primitive* $2n$-th root of unity. That is, ω is a solution of the equation $x^{2n} - 1 = 0$, and it holds that $\omega^i \neq 1$ for any integer i with $1 \leq i < 2n$. For convenience, we choose $\omega := e^{\frac{\pi \mathbf{i}}{n}} = \cos(\pi/n) + \mathbf{i} \cdot \sin(\pi/n)$, even though other choices are possible. We will see that, for n a power of two, there exists a very efficient method, called *Fast Fourier Transform* (FFT for short) due to Cooley and Tukey (1965), that needs only $O(n \log n)$ additions and multiplications of complex numbers in order compute the so-called *Discrete Fourier Transform* (DFT for short)

$$\mathrm{DFT}_\omega(f) := (f(1), f(\omega), \ldots, f(\omega^{2n-1})).$$

The efficiency of the method is based on the fact that there are only $2n$ different values for x_i^j for any i, j if $x_i = \omega$, whereas, for a general choice of x_i, there are $2n^2$ different values for x_i^j. We will further show that the fast convolution method can also be used to interpolate h from the values $h(x_i)$ in a comparably efficient manner.

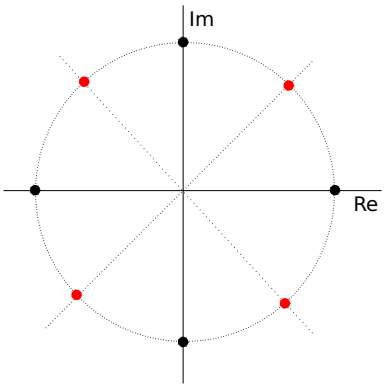

Figure 2.1: The dots on the unit circle are the 8-th roots of unity. The red dots are primitive.

One problem of the approach is that, since ω is not a rational number in general, the computations involving ω can only be carried out with approximate arithmetic. However, we will show that the total (absolute) error that occurs during the computation is less than $1/2$ if we use fixed point arithmetic with a precision $\rho > \rho_0$ in each step, where ρ_0 is some computable number of size $O(\log n)$. In addition, we will show that all occurring numbers in the intermediate results have length bounded by $O(\log n)$, and thus we may conclude that, using $O(n \log n)$ arithmetic operations on fixed-point numbers of length $O(\log n)$, we can compute approximations $\tilde{c}^{(i)}$ of the coefficients $c^{(i)}$ of h with $|c^{(i)} - \tilde{c}^{(i)}| < 1/2$. Since each coefficient $c^{(i)}$ is an integer, we can thus derive the exact value $c^{(i)}$ from its approximation $\tilde{c}^{(i)}$. We give the following example to illustrate the last step: Suppose that our approach yields the approximation

$$\tilde{h} = 2.34 \cdot x^{10} - 0.14 \cdot x^9 + 0.98 \cdot x^8 + \cdots + 0.67 \cdot x + 1.11 \tag{2.2}$$

for the product $h = f \cdot g$ of two integer polynomials f and g. In addition, according to the choice of our precision ρ, we can guarantee that the absolute error is less than $1/2$. Now, since the coefficients of h are integers and since they differ from the corresponding approximations by less than $1/2$, we conclude that $h = 2 \cdot x^{10} + x^8 + \cdots + x + 1$.

It remains to show how to recover the product $c = a \cdot b$ from the polynomial h. For this, we evaluate h at $x = B$, which amounts for shifting each coefficient $c^{(i)}$ by i digits and summing up the so obtained numbers. Here, it is crucial that each $c^{(i)}$ has length $O(\log n)$, and thus each summation uses only $O(\log n)$ primitive operations. We conclude that the total cost is bounded by $O(n \log n \cdot M(\log n))) = O(n(\log n)^{2+\epsilon})$ primitive operations, where ϵ is an arbitrary fixed positive number. Instead of using the Toom-Cook algorithm for the occurring multiplications in the Schönhage-Strassen method, we could instead call the Schönhage-Strassen method recursively. This yields the running time

$$O(n \log n M(n)) = O(n(\log n)(\log\log n) M(\log\log n)) = O(n(\log n)^2 (\log\log n)^2 M(\log\log\log n))$$

and so on. As already mentioned above, it is possible to slightly improve upon this approach. This is achieved by splitting the initial numbers not into $\approx n/\log n$ blocks of size $\approx \log n$. Then, recursively calling the algorithm even yields the complexity bound $O(nM(\log n)))$. We now give details in the following two sections.

2.1.2 Fast Fourier Transform

Even though we are mainly interested in solving problems defined over the real or complex numbers, it will turn out to be useful to work over an arbitrary ring R (or a field $\mathbb{K}$). In what follows, we always assume that R is a commutative ring with $1 = 1_R$.

We start with the following definition:

Definition 2.1.1 (Convolution)**.** *Let $f = a_0 + \cdots + a_{N-1} \cdot x^{N-1}$ and $g = b_0 + \cdots + b_{N-1} \cdot x^{N-1}$ be two polynomials of degree less than N in $R[x]$. We define*

$$f \star_N g := \sum_{k=0}^{N-1} c_k \cdot x^k := \sum_{k=0}^{N-1} \left(\sum_{i,j:i+j=k \mod N} a_i \cdot b_j \right) \cdot x^k$$

as the convolution *of f and g.*

Example. Let $f = 1 + x + x^2 \in \mathbb{Z}[x]$ and $g := 2 - x$, then $f \cdot g = 2 + x + x^2 - x^3$, and

$$f \star_3 g = (2 - 1) + 1 \cdot x + 1 \cdot x^2 = 1 - x + x^2.$$

Notice that, in general, $f \star_N g = f \cdot g \mod (x^N - 1)$. In particular, if we consider two polynomials f and g of degree less than n as polynomials of degree less than $2n - 1$ (by setting $a_n = \cdots = a_{2n-1} = b_n = \cdots = b_{2n-1} = 0$), then it holds that $f \star_{2n} g = f \cdot g$.

In our overview of the Schönhage-Strassen multiplication for n-digit numbers, we mentioned that the method considers an evaluation/interpolation approach using the $2n$-th complex roots of unity. Again, we generalize this approach to arbitrary rings.

Definition 2.1.2 (Root of Unity and Discrete Fourier Transform (DFT))**.** *Let $\omega \in R$, and $N \in \mathbb{N}$. We call ω an N-th root of unity if $\omega^N = 1$. We further call ω primitive if $\omega^{N/i} - 1$ is not a zero-divisor[2] in R for any divisor i of N. For fixed ω, the* Discrete Fourier Transform *of a polynomial $f \in R[x]$ is defined as*

$$\mathrm{DFT}_\omega(f) := (f(1), f(\omega), \ldots, f(\omega^{N-1})).$$

For a vector $a = (a_0, \ldots, a_{N-1})^t \in R^N$, we define $\mathrm{DFT}_\omega(a) := \mathrm{DFT}_\omega(\sum_{i=0}^{N-1} a_i x^i)$.

We remark that there does not always exist a primitive N-th root of unity in a ring R. For instance, this is the case for $R = \mathbb{Z}$ or $R = \mathbb{R}$. The following exercise (taken from [GG03, Sec. 8]) gives a necessary and sufficient condition on the existence of a primitive root of unity in the finite field $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$.

[2] An element $a \in R$ is a zero divisor if there exists an $r \in R$ with $a \cdot r = 0 = 0_R$ or $r \cdot a = 0$. A zero-divisor does not have to be zero. For instance, $a = \bar{3} \in R = \mathbb{Z}/6\mathbb{Z}$ is a zero divisor in R as $\bar{2} \cdot \bar{3} = \bar{0}$.

Exercise 2.1.3. *Denote by $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$ the finite field with p elements for some prime p, and let $N \in \{1, \ldots, p-1\}$. Show that $\mathbb{F}_p$ contains a primitive N-th root of unity if and only if N divides $p-1$, and conclude that the multiplicative group $\mathbb{F}_p^\times$ of $\mathbb{F}_p$ is cyclic.*

Hints:

1. *Use (without proof)* ***Fermat's little theorem:*** For arbitrary $a \in \mathbb{Z}$ arbitrary, it holds

$$a^p \equiv a \mod p.$$

   In particular, if $a \in \{1, \ldots, p-1\}$, then

$$a^{p-1} \equiv 1 \mod p.$$

2. *Let $q \in \mathbb{N}$ be a divisor of $p-1$ and $q = q_1^{e_1} \cdots q_r^{e_r}$ its prime factorization. For $a \in \mathbb{F}_p^\times$, we denote by $\mathrm{ord}(a) := \min\{i \in \mathbb{N}_{>0} : a^i = 1\}$ the order of a in $\mathbb{F}_p^\times$.*

   Prove the following facts:

   - $\mathrm{ord}(a) = q$ *if and only if* $a^q = 1$ *and* $a^{q/q_i} \neq 1$ *for* $i = 1, \ldots, r$.
   - *For each i, $\mathbb{F}_p^\times$ contains an element a_i with $q_i^{e_i} \mid \mathrm{ord}(a_i)$. Conclude that there is an element b_i with $\mathrm{ord}(b_i) = q_i^{e_i}$.*
   - *If $a, b \in \mathbb{F}_p^\times$ are elements of coprime orders, then $\mathrm{ord}(ab) = \mathrm{ord}(a)\,\mathrm{ord}(b)$.*
   - $\mathbb{F}_p^\times$ *contains an element of order q.*

Lemma 2.1.4. *For $N \in \mathbb{N}$, suppose that there exists a primitive N-root of unity ω in R. For any two polynomials $f, g \in R[x]$ of degree less than N, it holds that*

$$\mathrm{DFT}_\omega(f \star_N g) = \mathrm{DFT}_\omega(f) \cdot \mathrm{DFT}_\omega(g) = (f(1) \cdot g(1), f(\omega) \cdot g(\omega), \ldots, f(\omega^{N-1}) \cdot g(\omega^{N-1})).$$

Proof. There exists a polynomial $q \in R[x]$ with $f \star_N g = f \cdot g + q \cdot (x^N - 1)$. Thus, we have

$$\begin{aligned}(f \star_N g)(\omega^i) &= f(\omega^i) \cdot g(\omega^i) + q(\omega^i) \cdot ((\omega^i)^N - 1) = f(\omega^i) \cdot g(\omega^i) + q(\omega^i) \cdot ((\omega^N)^i - 1) = \\ &= f(\omega^i) \cdot g(\omega^i) + q(\omega^i) \cdot (1^i - 1) = f(\omega^i) \cdot g(\omega^i).\end{aligned}$$

□

In our overview of the Schönhage-Strassen method, one step is to compute the Discrete Fourier Transforms $\mathrm{DFT}_\omega(f)$ and $\mathrm{DFT}_\omega(g)$ of two polynomials of degree at most $n-1$, where ω is an N-th root of unity in $\mathbb{C}$, with $N := 2n$. Now from the above lemma and the fact that $f \star_N g = f \cdot g = h$, we conclude that

$$\mathrm{DFT}_\omega(h) = \mathrm{DFT}_\omega(f \cdot g) = \mathrm{DFT}_\omega(f \star_N g) = \mathrm{DFT}_\omega(f) \cdot \mathrm{DFT}_\omega(g). \tag{2.3}$$

Notice that the mapping $\mathrm{DFT}_\omega : R^N \mapsto R^N$ is given by the Vandermonde matrix

$$V_\omega := \mathrm{Vand}(1, \omega, \ldots, \omega^{N-1}) = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ 1 & \omega & \cdots & \omega^{N-1} \\ \vdots & \vdots & \vdots & \vdots \\ 1 & \omega^{N-1} & \cdots & \omega^{N(N-1)} \end{pmatrix}.$$

That is, the coefficient vector $a := (a_0, \ldots, a_{N-1})^t$ of a polynomial $f = \sum_{i=0}^{N-1} a_i \cdot x^i \in R[x]$ is mapped to the vector $v := (f(1), f(\omega), \ldots, f(\omega^{N-1}))^t = V_\omega \cdot a$. Vice versa, if v is known, then the coefficients a_i of f can be reconstructed as $a = V_\omega^{-1} \cdot v$. It turns out that a multiple of V_ω^{-1} can be easily computed.

Theorem 2.1.5. *Let ω be a primitive N-th root in R. Then, $\omega^{N-1} = \omega^{-1}$ is also a primitive N-th root of unity and $V_\omega \cdot V_{\omega^{-1}} = N \cdot \mathrm{Id}_N$, with Id_N the $N \times N$-identity matrix.*

Proof. We split the proof into four parts:

(1) $\omega^{n-1} = \omega^{-1}$ is a primitive N-th root of unity: Since

$$(\omega^{N-1})^N = (\omega^N)^{N-1} = 1^{N-1} = 1,$$

it follows that ω^{N-1} is an root of unity. Now suppose that there exists a divisor t of N and a $b \in R$ with $((\omega^{N-1})^{N/t} - 1)$. Then, multiplication with $\omega^{N/t}$ implies that

$$0 = \omega^{N/t} \cdot ((\omega^{N-1})^{N/t} - 1) = [(\omega \cdot \omega^{N-1})^{N/t} - \omega^{N/t}) \cdot b = (1 - \omega^{N/t}) \cdot b,$$

and thus $\omega^{N/t} - 1$ is a zero-divisor in R, which contradicts our assumption.

(2) $\omega^\ell - 1$ is not a zero divisor for all $\ell \in \mathbb{N}$ with $1 \leq \ell < N$: Let $g := \gcd(\ell, N)$ be the greatest common divisor of ℓ and $\mathbb{N}$. Then, there exist integers[3] s and t with $s \cdot \ell + t \cdot N = g$. Since $g < n$, there exists a prime divisor p of N that divides N/g, and thus g divides N/p. Hence, we obtain

$$\omega^{N/p} - 1 = (\omega^g)^{\frac{N}{pg}} - 1 = (\omega^g - 1) \cdot \underbrace{\sum_{i=0}^{\frac{N}{pg}-1} \omega^{i \cdot g}}_{=:r}.$$

Now, suppose that there exists a $b \in R$ with $b \cdot (\omega^g - 1) = 0$, then we also have $b \cdot (\omega^{N/p} - 1) = 0$, and thus $b = 0$ as ω is not a zero divisor. This shows that $\omega^g - 1$ is not a zero divisor as well. Notice that $\omega^\ell - 1$ divides $\omega^{s\ell} - 1 = (\omega^\ell - 1) \cdot \sum_{i=0}^{s-1} \omega^{i\ell}$, and since

$$\omega^{s\ell} - 1 = \omega^{s\ell} \cdot (\omega^N)^t - 1 = \omega^{s\ell + tN} - 1 = \omega^g - 1$$

we conclude that $\omega^\ell - 1$ also divides $\omega^g - 1$. It follows that $\omega^\ell - 1$ is not a zero divisor as $b \cdot (\omega^\ell - 1) = 0$ implies that $b \cdot (\omega^g - 1) = 0$, and thus $b = 0$.

(3) It holds that $\sum_{0 \leq j < N} \omega^{\ell j} = 0$ for any $\ell \in \mathbb{N}$ with $1 \leq \ell < N$: It holds that

$$(\omega^\ell - 1) \cdot \sum_{j=0}^{N-1} \omega^{\ell j} = \omega^{\ell N} - 1 = 0,$$

and thus $\sum_{j=0}^{N-1} \omega^{\ell j} = 0$ as $\omega^\ell - 1$ is not a zero divisor.

(4) $V_\omega \cdot V_{\omega^{-1}} = N \cdot \mathrm{Id}_N$: The (i,k)-th entry c_{ij} of $V_\omega \cdot V_{\omega^{-1}}$ is given as

$$c_{ij} = \sum_{j=0}^{N-1} \omega^{ij} \omega^{-jk} = \sum_{j=0}^{N-1} \omega^{(i-k)j} = \begin{cases} N & \text{if } i = k \\ 0 & \text{if } i \neq k, \end{cases}$$

where we used (3) for the case $i \neq k$. □

Algorithm 6: Fast Fourier Transform

Input : A polynomial $f = a_0 + \cdots + a_{N-1} \cdot x^{N-1} \in R[x]$, with $N = 2^k$ and $k \in \mathbb{N}_0$, and a primitive N-th root of unity $\omega \in R$.

Output: $\mathrm{DFT}_\omega(f)$.

if *N=1* **then**
  return a_0
Compute $\omega_i := \omega^i$ for $i = 0, \ldots, N-1$
$f^{\mathrm{ev}} := \sum_{i=0}^{N/2-1} a_{2i} \cdot x^i$ and $f^{\mathrm{odd}} := \sum_{i=0}^{N/2-1} a_{2i+1} \cdot x^i$
Call Algorithm 6 recursively to compute

$$(d_0^{\mathrm{ev}}, \ldots, d_{N/2-1}^{\mathrm{ev}}) := \mathrm{DFT}_{\omega^2}(f^{\mathrm{ev}})$$

and

$$(d_0^{\mathrm{odd}}, \ldots, d_{N/2-1}^{\mathrm{odd}}) := \mathrm{DFT}_{\omega^2}(f^{\mathrm{odd}}).$$

for $i = 1, \ldots, N-1$ **do**
  Let $j = i \mod N/2$. Compute

$$d_i := d_j^{\mathrm{ev}} + \omega_i \cdot d_j^{\mathrm{odd}}.$$

return $(d_0, \ldots, d_{N-1})$

Exercise 2.1.6. *Let* $\mathbb{F} = \mathbb{Z}/29\mathbb{Z}$.

1. *Find a primitive* 4*-th root of unity* $\omega \in \mathbb{F}$ *and compute its inverse* $\omega^{-1} \in \mathbb{F}$.
2. *Check that the product of the two matrices* DFT_ω *and* $\mathrm{DFT}_{\omega^{-1}}$ *equals* $4 \cdot \mathrm{Id}_4$.

Theorem 2.1.5 shows that polynomial interpolation is essentially the same as polynomial evaluation when considering the N-th roots of unity as interpolation points. In particular, applying $\mathrm{DFT}_{\omega^{-1}}$ to both sides of (2.3), we obtain for the coefficient vector $c := (c_0, \ldots, c_{N-1})^t$ of $h = \sum_{i=0}^{N-1} c_i x^i$ that

$$N \cdot c = \mathrm{DFT}_{\omega^{-1}}(\mathrm{DFT}_\omega(h)) = \mathrm{DFT}_{\omega^{-1}}(\mathrm{DFT}_\omega(f) \cdot \mathrm{DFT}_\omega(g)). \tag{2.4}$$

Hence, for the evaluation/interpolation step in the Schönhage-Strassen algorithm, we need to carry out three computations of a DFT plus one pointwise multiplication of two DFTs. We next describe an efficient method [CT65] due to Cooley und Tukey (from 1965) for computing the discrete Fourier Transform $\mathrm{DFT}_\omega(f)$ for some polynomial f of degree less than $N-1$ and ω a primitive N-th root of unity.[4] In what follows, we assume that R *supports the* FFT, that is, it contains an N-th root of unity for any $N = 2^k$, with $k \in \mathbb{N}$. In the following considerations, we further assume that N is such a power of two. We can now write a

[3] This follows from the extended Euclidean Algorithm, which we will treat in detail in the next chapter.

[4] In fact, it was Gauss who invented the algorithm already 160 years earlier. Cooley and Tukey rediscovered and popularized the method. The algorithm has a series of applications in engineering, applied mathematics, and the natural sciences. The original paper from 1965 has more than 13400 citations!

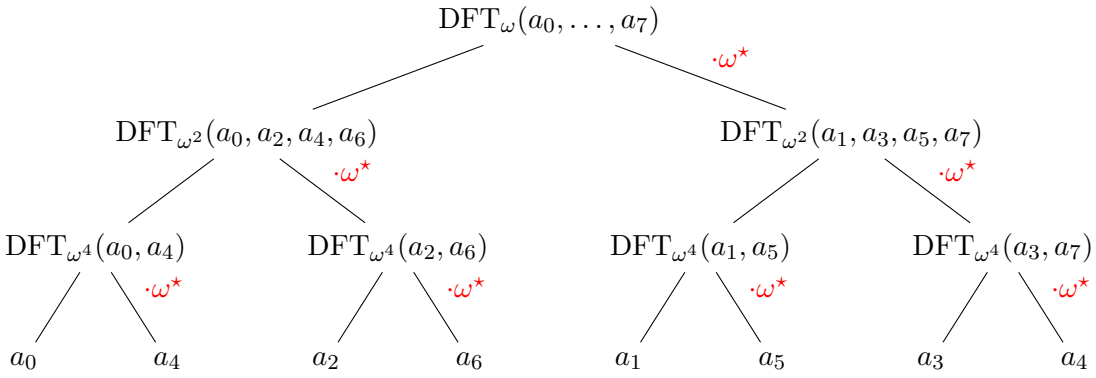

Figure 2.2: Starting with the coefficients $a_i = \mathrm{DFT}_{\omega^8}(a_i)$ of f, we iteratively compute four DFT's of length 2, two DFT's of length 4, and eventually $\mathrm{DFT}_\omega(f)$, which has length 8. In Step ℓ, the i-th entry of a Discrete Fourier Transform of size $N/2^\ell$ is computed as the the sum of the j-th entry of the left child and the j-th entry of the right child multiplied by ω^i (illustrated by the edge labelling "$\cdot\omega^\star$" in the above picture), where $j = i \mod N/2^{\ell+1}$.

polynomial $f(x) = a_0 + \cdots + a_{N-1} \cdot x^{N-1} \in R[x]$ as

$$f(x) = \sum_{i=0}^{N/2-1} a_{2i} \cdot x^{2i} + \sum_{i=0}^{N/2-1} a_{2i+1} \cdot x^{2i+1} = f^{\mathrm{ev}}(x^2) + x \cdot f^{\mathrm{odd}}(x^2),$$

with $f^{\mathrm{ev}} := \sum_{i=0}^{N/2-1} a_{2i} \cdot x^i$ and $f^{\mathrm{odd}} := \sum_{i=0}^{N/2-1} a_{2i+1} \cdot x^i$. Plugging $x = \omega^i$ into the above equation then yields that

$$f(\omega^i) = f^{\mathrm{ev}}(\omega^{2i}) + \omega^i \cdot f^{\mathrm{odd}}(\omega^{2i}). \tag{2.5}$$

Notice that ω^2 is a primitive $N/2$-root, hence the computation of $\mathrm{DFT}_\omega(f) = (d_0,\ldots,d_{N-1})$ can be reduced to the computation of the two Discrete Fourier Transforms $\mathrm{DFT}_{\omega^2}(f^{\mathrm{ev}}) = (d_0^{\mathrm{ev}},\ldots,d_{N/2-1}^{\mathrm{ev}})$ and $\mathrm{DFT}_{\omega^2}(f^{\mathrm{odd}}) = (d_0^{\mathrm{odd}},\ldots,d_{N/2-1}^{\mathrm{odd}})$ followed by the computation of $d_i := d_j^{\mathrm{ev}} + \omega^i \cdot d_j^{\mathrm{odd}}$ for all $i = 0,\ldots,N$ and $j = i \mod N/2$; see Algorithm 6.

In terms of complexity, this means that we can compute a Discrete Fourier Transform of size N by computing two Discrete Fourier Transforms of size $N/2$ plus $3N$ additional additions and multiplications (by powers of ω). If we use $T(N)$ to denote the number of arithmetic operations in R that are needed in the worst case to compute the Discrete Fourier Transform $\mathrm{DFT}_\omega(f)$ for a polynomial f of degree less than N and a primitive N-th root of unity ω, the above consideration implies that

$$T(N) \leq 2 \cdot T(N/2) + 3 \cdot N.$$

Hence, we obtain the following result:

Theorem 2.1.7. *Let $f \in R[x]$ be a polynomial of degree less than N and ω be a primitive N-th root of unity ω in R, then Algorithm 6 computes $\mathrm{DFT}_\omega(f)$ using $O(N \log N)$ arithmetic operations in R.*

For an illustration of the FFT Algorithm when applied to a polynomial $f = a_0 + \cdots + a_7 \cdot x^7 \in R[x]$ of degree 7 and ω a primitive 8-th root of unity, see Figure 2.2.

Algorithm 7: Fast Convolution

Input : A commutative ring R, two polynomials $f, g \in R[x]$ of degree less than $N = 2^k$, with $k \in \mathbb{N}_0$, and a primitive N-th root of unity $\omega \in R$.

Output: $f \star_N g$.

Compute:
$\omega^{-1} = \omega^{N-1}$.
$D_f := \mathrm{DFT}_\omega(f)$ and $D_g := \mathrm{DFT}_\omega(g)$
$D_h := D_f \cdot D_g$
$E := \frac{\mathrm{DFT}_{\omega^{-1}}(D_h)}{N}$
return E

From (2.4) and the FFT algorithm, we can can now directly derive an efficient algorithm for computing the convolution $f \star_N g$ of two polynomials $f, g \in R[x]$ of degree less than N. Namely, we first compute $\mathrm{DFT}_\omega(f)$ and $\mathrm{DFT}_\omega(g)$ and their pointwise product P. Then, we compute $\mathrm{DFT}_{\omega^{-1}}(P)$ and divide each of its entries by N; see Algorithm 7. Notice that all but the last operation use $O(n \log n)$ arithmetic operations in R. According to Section 1.4, the division by N is relatively cheap in the special case where $R = \mathbb{C}$, however, it might be an entirely non-trivial task for a different ring.

Theorem 2.1.8. *Let $f, g \in R[x]$ be polynomials of degree less than $N = 2^k$ with $k \in \mathbb{N}$. Suppose that a primitive N-th root of unity ω in R is given. Then, Algorithm 7 computes $f \star_N g$ using $O(N \log N)$ arithmetic operations in R plus N divisions by N.*

For two polynomial $f, g \in R[x]$ of degree n or less, it holds that $f \cdot g = f \star_N g$, with $N := 2^{\lceil \log n \rceil + 1}$. Hence, if a primitive N-th root of unity is given, then Algorithm 7 computes the product of f and g using $O(n \log n)$ arithmetic operations in R plus N divisions by N.

Corollary 2.1.9. *Let $f, g \in R[x]$ be polynomials of degree less than n, and $N := 2^{\lceil \log n \rceil + 1}$. If a primitive N-th root of unity ω in R is given, then Algorithm 6 computes $f \cdot g$ using $O(N \log N) = O(n \log n)$ arithmetic operations in R plus N divisions by N.*

2.1.3 Fast Multiplication in $\mathbb{Z}$ and $\mathbb{Z}[x]$.

We are now coming back to our original problem of computing the product of two integer polynomials $f, g \in \mathbb{Z}[x]$ of degree less than n. We further assume that the coefficients of f and g have absolute value less than 2^L. Since $\mathbb{Z}$ does not contain a primitive N-th root of unity for any integer $N > 2$, we cannot directly apply the above approach (with $R = \mathbb{Z}$) to compute the product $f \cdot g$. However, since f, g can also be considered as polynomials with complex coefficients and since $\mathbb{C}$ supports the FFT, Corollary 2.1.9 implies that we can compute the product using $O(n \log n)$ arithmetic operations in $\mathbb{C}$ plus N divisions by N, where $N := 2^{\lceil \log n \rceil + 1}$. As already mentioned in our overview of the Schönhage-Strassen method, we need to address the problem that these operations can only be carried out with approximate arithmetic. Now, suppose that we use fixed point arithmetic with base 2 and a fixed precision ρ in each step of Algorithm 7. Then, we aim to answer the question how large ρ needs to be chosen such that the final error is smaller than $1/2$, which would allow us to derive the exact coefficients of $f \cdot g$ from the computed approximations; see (2.2) for the example we gave at

the beginning of the chapter. Before running Algorithm 7, we first compute an approximation $\tilde{\omega} \in \mathbb{F} = \mathbb{F}_{2,\rho}$ of the N-th root of unity $\omega = \cos(2\pi/N) + \mathbf{i} \cdot \sin(2\pi/N)$ such that $|\tilde{\omega} - \omega| < 2^{-\rho}$. According to Exercise 1.4.4 and Exercise 1.4.5, the cost for this computation is bounded by $O(\rho^c)$ for some constant c. From Theorem 1.3.3, we further conclude that

$$|P(\omega) - P_{\mathbb{F}}(\omega)| < 4N^2 \cdot 2^{-\rho} \cdot \max(1, |\omega|)^{N-1} = 4N^2 \cdot 2^{-\rho}$$

for $P(x) := x^i$ and an arbitrary $i \in \{0, \ldots, N-1\}$. Hence, recursively taking powers of the approximation $\tilde{\omega}_1 := \tilde{\omega}$ and using fixed point arithmetic in each step yields approximations $\tilde{\omega}_i$ of $\omega_i := \omega^i$ with $|\tilde{\omega}_i - \omega_i| < 4N^2 \cdot 2^{-\rho}$.

In the Fast Fourier Transform, the entries of $\mathrm{DFT}_\omega(f) = (c_0, \ldots, c_{N-1})$ are recursively computed from the coefficients of $f = a_0 + \cdots + a_{N-1} \cdot x^{N-1}$. That is, at the highest level of the recursion, we start with a suitable permutation of the coefficients a_i and recursively compute corresponding DFT's of size $2, 4, 8, \ldots$ until we obtain $\mathrm{DFT}_\omega(f)$. More specifically, at level ℓ of the recursion, the i-th entry d_i of each DFT of size $N/2^{\ell-1}$ is computed as

$$d_i = d_j^{\mathrm{ev}} + \omega^i \cdot d_j^{\mathrm{odd}}$$

where d_j^{ev} and d_j^{odd} are the j-th entries of previously computed DFT's of size $N/2^\ell$ and $j = i \mod N/2^\ell$. Now suppose that we use a precision $\rho > 2(\log N + 1)$ and that we have already computed approximations $\tilde{d}_j^{\mathrm{ev}}$ and $\tilde{d}_j^{\mathrm{odd}}$ of the entries d_j^{ev} and d_j^{odd}, respectively, with $|\tilde{d}_j^{\mathrm{ev}} - d_j^{\mathrm{ev}}|, |\tilde{d}_j^{\mathrm{odd}} - d_j^{\mathrm{odd}}| < \epsilon$. Then $\tilde{d}_i := \tilde{d}_j^{\mathrm{ev}} \mathbin{\tilde{+}} \tilde{w}_i \mathbin{\tilde{\cdot}} \tilde{d}_j^{\mathrm{odd}}$ constitutes an approximation of d_i with

$$\begin{aligned} |d_i - \tilde{d}_i| &< 2^{-\rho+1} + \epsilon + \epsilon \cdot |\omega_i| + 4N^2 \cdot 2^{-\rho} \cdot |d_j^{\mathrm{odd}}| + 4N^2 \cdot 2^{-\rho} \cdot \epsilon \\ &= \epsilon \cdot (2 + 4N^2 \cdot 2^{-\rho}) + 2^{-\rho} \cdot (2 + 4N^2 \cdot d_j^{\mathrm{odd}}) \\ &< 3\epsilon + 4N^2 \cdot 2^{-\rho} \cdot (1 + |d_j^{\mathrm{odd}}|), \end{aligned} \tag{2.6}$$

where we used our bounds (1.2) and (1.3) for the error that occurs when using fixed point arithmetic. Further notice that d_j^{odd} is an entry of $\mathrm{DFT}_{\omega^{N/2^\ell}}(\hat{f})$, where $\hat{f}$ is an integer polynomial of degree less than $N/2^\ell$, whose coefficients form a subset of the set of coefficients of f. Hence, we have $d_j^{\mathrm{odd}} < \frac{N}{2^\ell} \cdot 2^L < \frac{N}{2} \cdot 2^L$, and thus (2.6) yields

$$|d_i - \tilde{d}_i| < 8 \cdot \max(\epsilon, 4N^3 \cdot 2^L \cdot 2^{-\rho})$$

Since there are $\log N$ steps in the recursion, we conclude that the computed approximations of the entries of $\mathrm{DFT}_\omega(f)$ differ from the exact values by at most $8^{\log N}$ times the maximum of the input error[5] for the coefficients a_i and the value $4N^3 \cdot 2^L \cdot 2^{-\rho}$. Hence, the total error is bounded by $4N^6 \cdot 2^L \cdot 2^{-\rho}$. The same bound then also applies to the error that we obtain when computing $\mathrm{DFT}_\omega(g)$ with fixed point arithmetic.

We may now assume that we have computed approximations $\tilde{D}_f = (\tilde{f}_0, \ldots, \tilde{f}_{N-1})$ and $\tilde{D}_g = (\tilde{g}_0, \ldots, \tilde{g}_{N-1})$ of

$$D_f = (f_0, \ldots, f_{N-1}) := \mathrm{DFT}_\omega(f)$$

and

$$D_g = (g_0, \ldots, g_{N-1}) := \mathrm{DFT}_\omega(g)$$

[5]Here, the coefficients are given exactly, and thus the input error is zero. However, our analysis also applies to the case where only approximations $\tilde{a}_i$ of the coefficients a_i are given. Then the total error is bounded by $8^{\log N} \cdot \max(4N^3 \cdot 2^L \cdot 2^{-\rho}, \max_i |a_i - \tilde{a}_i|)$.

to an absolute error bounded by $4N^6 \cdot 2^L \cdot 2^{-\rho}$. Pointwise multiplication of $\tilde{D}_f$ and $\tilde{D}_g$ (again using fixed point arithmetic with precision ρ) then yields an approximation $\tilde{D}_h = (\tilde{h}_0, \ldots, \tilde{h}_{N-1}) := \tilde{D}_f \mathbin{\tilde{\cdot}} \tilde{D}_g$ of $D_h = \mathrm{DFT}_\omega(h) = (h_0, \ldots, h_{N-1})$, and according to (1.3), the absolute error $|h_i - \tilde{h}_i|$ is bounded by

$$2^{-\rho} + 4N^6 \cdot 2^L \cdot 2^{-\rho} \cdot N \cdot 2^L \cdot (|f_i| + |g_i|) + (4N^6 \cdot 2^L \cdot 2^{-\rho})^2 < 32N^{12} \cdot 2^{2L} \cdot 2^{-\rho}$$

as $|f_i|, |g_i| \leq N \cdot 2^L$ for all $i = 0, \ldots, N-1$.

It remains to estimate the error when computing $\frac{1}{N} \cdot \mathrm{DFT}_{\omega^{-1}}(\tilde{D}_h)$ with fixed point arithmetic. In completely analogous manner as above, one shows that the output error of the computation of $\mathrm{DFT}_{\omega^{-1}}(\tilde{D}_h)$ is bounded by $8^{\log N} \cdot \max_i |h_i - \tilde{h}_i| < 32N^{15} \cdot 2^{2L} \cdot 2^{-\rho}$. The final division by N amounts for a shift by $\log N$ bits as N is a power of two, which shows that the total error is at most $32N^{14} \cdot 2^{2L} \cdot 2^{-\rho}$. Hence, in order to guarantee an output error of less than $1/2$, it suffices to consider a precision

$$\rho > \rho_0 := \log(64N^{14} \cdot 2^{2L}) = 6 + 14 \log N + 2L = O(\log n + L).$$

Each of the intermediate results is an approximation of an entry of some $\mathrm{DFT}_{\omega^{N/2^\ell}}(\hat{f})$, where $\ell \in \{0, \ldots, \log N\}$ and $\hat{f}$ is a polynomial of degree at most N with integer coefficients that form a subset of the set of coefficients of f, g, or $f \cdot g$. Hence, each of these coefficients has absolute value less than $N \cdot 2^L$. It follows that each intermediate result is a fixed point number of length $2^{O(\log N + L + \rho)}$. Since we succeed for $\rho = 2\rho_0$, it follows that the computation of $f \cdot g$ uses $O(n \log n)$ arithmetic operations of fixed numbers of length $O(\log n + L)$. The following result then follows directly.

Theorem 2.1.10. *Let $f, g \in \mathbb{Z}[x]$ be polynomials of degree less than n and with one-digit integer coefficients. Then, the product $f \cdot g$ can be computed using $O(n \log n \cdot M(\log n))$ primitive operations.*

From the above theorem, we can now derive the following result on the cost for multiplying two integers of length less than n:

Theorem 2.1.11. *Given two integers a and b of length less than n, the product $a \cdot b$ can be computed using $O(n \log n \cdot M(\log n)) = O(n(\log n)^{2+\epsilon})$ primitive operations, where ϵ is an arbitrary but fixed constant. Furthermore, we can compute a dyadic approximation $\tilde{q}$ with $|\tilde{q} - a/b| < 2^{-L}$ using $O((n+L) \cdot (\log(n+L))^{3+\epsilon})$ primitive operations.*

Proof. The polynomials $f = a^{(0)} + \cdots + a^{(n-1)} \cdot x^{n-1}$ and $f = b^{(0)} + \cdots + b^{(n-1)} \cdot x^{n-1}$ in (2.1) have one digit coefficients, hence we can compute the product $h = f \cdot g$ using $O(n \log n \cdot M(\log n))$ primitive operations according to Theorem 2.1.10. The computation of $a \cdot b = h(B)$ is bounded by $O(n \log n)$ primitive operations as this step requires $O(n)$ additions, each involving an integer of length $O(n)$ and an integer of length $O(\log n)$. The bound on the cost for the approximate division then follows directly from Theorem 1.4.2. □

You might wonder why we have not given a more general bound in Theorem 2.1.10 that applies to polynomials with integer coefficients of arbitrary length. Namely, if the length of the coefficients is bounded by L, then our above considerations show that the cost for multiplying f and g is bounded by $O(n \log n \cdot M(\log n + L))$ primitive operations *if* a sufficiently good approximation $\tilde{\omega}$ of ω with $|\omega - \tilde{\omega}| = 2^{-\Omega(L + \log n)}$ is already computed. But this is actually

critical as we have only shown that the cost for this step is bounded by $O((\log n + L)^c)$. Hence, in order to derive a bound on the total running time that is near-linear in L, we need a different approach.[6] Here, we consider an approach known as *Kronecker substitution*. The crucial idea is that if an upper bound on the length of the coefficients of a polynomial $F(x) = c_0 + c_1 \cdot x + \cdots c_n \cdot x^n$ is known, then one can recover the coefficients from the value of F at a single point. Namely, suppose that each c_i has length less than L (with respect to some base B), then evaluating F at $x = B^L$ yields

$$F(B^L) = c_0 + B^L \cdot c_1 + B^{2L} \cdot c_2 + \cdot + B^{nL} \cdot c_n.$$

Since each c_i has length less than L and since multiplication by B^{iL} yields a shift of c_i by iL digits, the coefficients can directly be read off the value $F(B^L)$ as there is no overlap. As an example, consider the polynomial $F(x) = 12 + 34 \cdot x + 45 \cdot x^2 + 67 \cdot x^3 + 8x^4$, where we have $f(1000) = \mathbf{8067045034012}$. *Kronecker substitution* now allows us to reduce the problem of computing the product $h = f \cdot g$ of two polynomials $f, g \in \mathbb{Z}[x]$ with coefficients of length less than L to the problem of multiplying two integers of length $O(n(L + \log n))$. This works as follows: Each coefficient of h has length less than $L' := \lceil 2L + \log n \rceil$. Hence, we can directly derive the coefficients of h from the value $h(B^{L'}) = f(B^{L'}) \cdot g(B^{L'})$. Evaluating f (or g) at $x = B^{L'}$ amounts for shifting the corresponding coefficients a_i (or b_i) by iL' digits and summing up the so obtained numbers. This step uses $O(n(L + \log n))$ primitive operations. The values $f(B^{L'})$ and $g(B^{L'})$ are integers of length $O(n(L+\log n))$, and thus we can compute their product using $\tilde{O}(nL)$ primitive operations, where the $\tilde{O}$ - notation indicates that we are omitting poly-logarithmic factors in the input. That is, $\tilde{O}(N) = O(N \cdot \log^c N)$ for some constant c. We fix this result:

Theorem 2.1.12. *Let $f, g \in \mathbb{Z}[x]$ be polynomials of degree less than n and with integer coefficients of length less than L. Then, the product $f \cdot g$ can be computed using $\tilde{O}(nL)$ primitive operations.*

We also state the following complexity bound for the evaluation of a polynomial $f \in \mathbb{Z}[x]$ at a rational point.

Theorem 2.1.13. *Let $f \in \mathbb{Z}[x]$ be a polynomial of degree n with coefficients of length less than 2^L, and let $x_0 = p/q$ be a rational point with integers p, q of length less than ℓ. Then, using Horner Evaluation, we can compute the value $f(x_0)$ using $\tilde{O}(n^2(\ell + L))$ primitive operations.*

Proof. We define $f_0 := a_n$, and

$$f_{i+1} = x \cdot f_i + a_{n-i-1} \in \mathbb{Z}[x] \text{ for } i = 0, \ldots, n-1.$$

Notice that, when using Horner Evaluation, we recursively compute the values $v_i := f_i(x_0)$. Since f_i is a polynomial of degree i, we conclude that $v_i = \frac{p_i}{q_i}$ is a rational number with denominator $q_i = q^i$ and numerator p_i of length less than $\log n + L + i \cdot \ell$. Hence, computing $f_{i+1}(x_0)$ from f_i amount for a constant number of arithmetic operations of integers of length $O(\log n + L + i \cdot \ell) = O(L + n \cdot \ell)$. Each such operations uses $\tilde{O}(L + n \cdot \ell)$ primitive operations, thus the claimed bound follows. □

[6] In fact, one can show that a such an approximation of ω can be computed in a number of primitive operations that is near linear in n and L. However, this requires to introduce some additional tools that we will treat only in one of the following chapters.

In the following exercise, we present a different evaluation method that yields a complexity bound that is near-optimal.

Exercise 2.1.14 (Estrin Evaluation)**.** *You already know Horner's method for polynomial evaluation. An alternative method is due to* Estrin: *In order to evaluate a polynomial* $f(x) = a_0 + \cdots + a_n \cdot x^n$*, let* $m := 2^{\lceil \log n \rceil - 1}$ *and write* f *as*

$$f(x) = \underbrace{(a_n x^m + a_{n-1} x^{m-1} + \cdots a_m)}_{=:f_H(x)} \cdot x^m + \underbrace{a_{m-1} x^{m-1} + a_{m-2} x^{m-2} + \cdots + a_0}_{=:f_L(x)},$$

where f_H *and* f_L *are polynomials of degree at most* m*. Recursively evaluate* f_H *and* f_L *and reconstruct* $f(x) = f_H(x) \cdot x^m + f_L(x)$*.*

Show that Estrin's method uses only $\tilde{O}(n(L+\ell))$ *primitive operations to compute* $f(x_0)$ *if* f *has integer coefficients of length* L *and* $x_0 = p/q \in \mathbb{Q}$ *is a rational point with integers* p, q *of length less than* ℓ*.*

Exercise 2.1.15 (Computing Euler's Number e)**.** *Show that*

$$\begin{pmatrix} \frac{1}{1} & \frac{1}{1} \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \frac{1}{2} & \frac{1}{2} \\ 0 & 1 \end{pmatrix} \cdots \begin{pmatrix} \frac{1}{n} & \frac{1}{n} \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} \frac{1}{n!} & \sum_{i=1}^{n} \frac{1}{i!} \\ 0 & 1 \end{pmatrix}.$$

Derive an algorithm with running time $\tilde{O}(L)$ *for computing a rational approximation* $\tilde{e}$ *of Euler's number* e *with* $|\tilde{e} - e| < 2^{-L}$*!*

Remark. Instead of using fixed-point arithmetic in each step of the Fast Convolution algorithm, we could have used fixed-point interval arithmetic. A corresponding analysis then yields comparable bounds on the needed precision. However, we might again profit from the fact that each interval approximation of some value carries a canonical adaptive bound on the approximation error (i.e. the width of the computed interval), whereas we have to work with a worst-case error bound if fixed point arithmetic is used. For instance, for the Schönhage-Strassen method, this means that we can iteratively increase the precision until the final interval approximations of the coefficients of h have width less than $1/2$ or, alternatively, until they contain only one integer.

2.1.4 Fast Multiplication over arbitrary Rings*

We have already shown that if R supports the FFT and if division by 2 can be carried out in an efficient manner in R, then computing the product of two polynomials $f, g \in R[x]$ of degree less than n uses only $O(n \log n)$ arithmetic operations in R. Can we also give a comparable bound for arbitrary commutative rings that do not support the FFT? The answer is yes, however, we will not give the details here, but only a rough idea of the approach. There are certain cases that need to be distinguished and the actual approach is slightly more involved than what we describe below. The interested reader should have a look into Section 8.3 of the textbook [GG03] "Modern Computer Algebra" from von zur Gathen und Gerhard, which contains a comprehensive description of the algorithm and its analysis.

The crucial idea underlying the approach is to adjoin a so-called *virtual root of unity*. For this, suppose that 2 is a unit in R and that $N = 2^k$ is a power of two. Then, we define $D_N := R[x]/\langle x^N + 1 \rangle$, an extension of the ring R. Since $x^{2N} = (x^N)^2 = 1 \mod (x^N + 1)$, we conclude that $\omega := x \mod (x^N + 1)$ is a $2N$-th root of unity in D_N. Suppose that, for some

divisor ℓ of $2N$ and some $b \in D$, we have $b \cdot (\omega^{2N/\ell} - 1) \cdot b = 0$. Since N is a power of two, the same holds for ℓ, and thus we may write $\omega^{2N/\ell} - 1$ as $\omega^{N/\ell'} - 1$ with $\ell' = \ell/2$. Hence, we obtain

$$b \cdot (\omega^N - 1) = b \cdot (\omega^{N/\ell'-1}) \cdot \sum_{i=0}^{\ell'-1} \omega^{in/\ell'} = 0.$$

Since $\omega^N - 1 = -2$ is a unit in R, it is also a unit in D_N, and thus we must have $b = 0$. This shows that ω is a primitive $2N$-th root of unity. Now, how does this help to multiply two polynomials $f, g \in R[x]$ of degree less than n? Remember that, when multiplying two integers of length n using either the Toom-Cook approach or the Schönhage-Strassen method, we first partitioned each integer into k blocks of size n/k and derived corresponding polynomials of degree k whose coefficients are integers of length n/k. We now proceed in a similar way with a suitably chosen k. More specifically, we first partition the coefficients of $f = \sum_{i=0}^{n-1} a_i x^i$ and $g = \sum_{i=0}^{n-1} b_i x^i$ into blocks of size $\sqrt{N}$, where $N := 2^{\lceil \log 2n \rceil}$. That is, we write

$$f(x) = \sum_{j=0}^{\sqrt{N}-1} f_j(x) \cdot x^{\sqrt{N} \cdot j} \text{ and } g(x) = \sum_{j=0}^{\sqrt{N}-1} g_j(x) \cdot x^{\sqrt{N} \cdot j},$$

with polynomials f_j and g_j of degree less than $\sqrt{N}$. Then, we consider polynomials F and G in $R[x][y]$ of degree less than $\sqrt{N}$ (in the variable y) with coefficients in $R[x]$ of degree less than $\sqrt{N}$:

$$F(x) := \sum_{j=0}^{\sqrt{N}-1} f_j(x) \cdot y^j \text{ and } G(x) := \sum_{j=0}^{\sqrt{N}-1} g_j(x) \cdot y^j$$

such that $f(x) = F(x, x^{\sqrt{N}})$ and $g(x) = G(x, x^{\sqrt{N}})$. We now consider the coefficients of F and G as elements in the ring $D_{2\sqrt{N}}$. Computationally, nothing happens at this step, however, in order to distinguish the polynomials F and G, which are contained in $R[x][y]$, from their corresponding images in $D_{2\sqrt{N}}[y]$, we use F^* and G^* to denote these images. Notice that since the coefficients of the product $H := F \cdot G \in R[x][y]$ are polynomials of degree less than $2\sqrt{D}$, they coincide with the corresponding coefficients of the product $H^* := F^* \cdot G^* \in D_{2\sqrt{N}}[y]$. This shows that we can reduce the computation of H (and thus also that of $h = f \cdot g$) to that of H^*. What we have gained with this approach is that since $D_{2\sqrt{N}}$ supports the DFT, we can use the fast convolution algorithm to compute H^*. For the latter computation, we need three FFT computations of size $2\sqrt{N}$ over the ring $D_{2\sqrt{N}}$ plus $2\sqrt{N}$ *essential multiplications* in $D_{2\sqrt{N}}$. Notice that the remaining multiplications in the FFT's are easy as each such multiplication just amounts for a multiplication by x^i modulo $x^{2\sqrt{N}} + 1$. Each essential multiplication amounts for computing the product of two polynomials in $R[x]$ of degree less than $2\sqrt{N}$. For these multiplications, we then call the algorithm recursively. A careful analysis then yields the claimed complexity bound as given in Theorem 2.1.16.

You may notice that $\sqrt{N}$ may not always be an integer, in particular, when calling the algorithm recursively for an N that is different from the initial one. In this case, one has to consider a corresponding rounding to the next power of two. We further remark that there is a variant of the approach that also works for rings, where 3 is a unit. One can further combine the latter two methods to an algorithm to compute the product of $f, g \in R[x]$, where R is an arbitrary commutative ring with 1. Details can be found in [GG03, Sec. 8.3].

Theorem 2.1.16. *Let R be a commutative ring with 1 and $f, g \in R[x]$ polynomials of degree less than n. The product of f and g can be computed using $O(n \log n \log\log n)$ arithmetic operations in R.*

The following Exercise gives an idea of the approach sketched above. Therein, we describe a simplified variant of one of the two algorithms for integer multiplication that Schönhage and Strassen published in their original paper from 1971.

Exercise 2.1.17. *Let $n = 2^{2^k}$ with $k \in \mathbb{N}$.*

(a) Show that $\omega := 8$ is a primitive $\sqrt{n}$-th root of unity in $R := \mathbb{Z}/(2^{3\sqrt{n}}+1)$.

(b) Let $a = a_{n-1}a_{n-2}\dots a_0$ and $b = b_{n-1}b_{n-2}\dots b_0$ be two integers of length n. Consider the integer polynomials

$$f(x) := \sum_{i=0}^{\sqrt{n}-1} (a_{(i+1)\sqrt{n}-1} \dots a_{i\sqrt{n}+1} a_{i\sqrt{n}}) \cdot x^i$$
$$g(x) := \sum_{i=0}^{\sqrt{n}-1} (b_{(i+1)\sqrt{n}-1} \dots b_{i\sqrt{n}+1} b_{i\sqrt{n}}) \cdot x^i,$$

and their images $f^* := f \mod (2^{3\sqrt{n}}+1)$ and $g^* := f \mod (2^{3\sqrt{n}}+1)$ in $R[x]$. Show that the coefficients of $h^* = f^* \star_{2\sqrt{n}} g^* \in R[x]$ equal the coefficients of $f \cdot g \in \mathbb{Z}[x]$, and conclude that h can be computed with $O(n \log n)$ arithmetic operations in R.

(c) Notice that, for computing h^*, we need only $2\sqrt{n}$ essential multiplications in R, whereas the remaining multiplications are multiplications by powers of ω. Which complexity bound can you derive for the computation of $a \cdot b$ when using the approach recursively for the essential multiplications?

Hint: You should first prove that each of these essential multiplications can be reduced to a constant number of additions and multiplications of integers of length $\sqrt{n}$.

2.2 Fast Polynomial Division and Applications

We start with the following definition of a Euclidean domain.

Definition 2.2.1. *A Euclidean domain is an integral domain*[7] *R together with a function $d : R \mapsto \mathbb{N} \cup \{-\infty\}$ if for all $a, b \in R$, with $b \neq 0$, there exist $q, r \in R$ with $a = q \cdot b + r$ and $d(r) < d(b)$. We call q and r the* quotient *and* remainder *of a and b, respectively, and write $q = \mathrm{quo}(a, b)$ and $r = \mathrm{rem}(a, b)$.*

Exercise 2.2.2. *For $R = \mathbb{Z}$ and $R = F[x]$, with F an arbitrary field, give a function $d : R \mapsto \mathbb{N} \cup \{-\infty\}$ such that R together with d is a Euclidean domain. Does there exist such a function d such that $R = \mathbb{Z}[x]$ is a Euclidean domain?*

In what follows, we now assume that R is an integral domain and that $R[x]$ together with the degree function $d := \deg$ is a Euclidean domain. Hence, for two polynomials $f = \sum_{i=0}^{n} a_i x^i$ and $g = \sum_{i=0}^{m} b_i x^i$ in $R[x]$, with $n \geq m$, there exist polynomials $q, r \in R[x]$ with

$$f(x) = q(x) \cdot g(x) + r(x) \text{ and } \deg(r) < m. \tag{2.7}$$

[7] An integral domain is a commutative ring with 1 that contains no zero-divisor.

Notice that the polynomials q and r in the above representation are uniquely defined if b_m is a unit in R. Namely, $f = q \cdot g + r = q^* \cdot g + r^*(x)$ implies that $r - r^* = g \cdot (q^* - q)$, and thus $r = r^*$ and $q^* = q$ as otherwise $\deg(g \cdot (q^* - q)) > \deg(r - r^*)$. Hence, we can assume that g is monic, that is, $b_m = 1$. We now give an efficient method for computing q and r. If $f = q \cdot g + r$, then

$$f(1/x) = q(1/x) \cdot g(1/x) + r(1/x)$$

and thus

$$\underbrace{x^n \cdot f(1/x)}_{=:\hat{f}(x)} = \underbrace{x^{n-m} \cdot q(1/x)}_{=:\hat{q}(x)} \cdot \underbrace{x^m \cdot g(1/x)}_{=:\hat{g}(x)} + x^{n-m+1} \cdot (x^{m-1} \cdot r(1/x)).$$

Notice that $\hat{f}, \hat{g}$, and $\hat{q}$ are obtained by just reversing the coefficients of f, g, and q, respectively. In addition, since r has degree less than m, $x^{m-1} \cdot r(1/x)$ is a polynomial. Hence, we obtain

$$\hat{f}(x) = \hat{q}(x) \cdot \hat{g}(x) \mod x^{n-m+1},$$

which shows that, in order to compute $\hat{q}(x)$ (and thus $q(x)$), we can alternatively compute the product of $\hat{f}(x)$ and an inverse of $\hat{g}(x)$ modulo x^{n-m+1}. This does not sound much easier, however, there is a simple way of recursively computing an inverse $\hat{h}_i \in R[x]/\langle x^{2^i} \rangle$ of $\hat{g}(x)$ mod x^{2^i} such that $\hat{h}_i \cdot \hat{g} = 1 \mod x^{2^i}$. Notice that $\hat{g}$ has constant coefficient $\hat{g}_0 = 1$ as g is monic, and thus $\hat{h}_0 := 1$ fulfills the equation $\hat{h}_0 \cdot \hat{g}_0 \mod x$. Now, for $i \geq 0$, we recursively define:

$$\hat{h}_{i+1} := 2\hat{h}_i - \hat{g} \cdot \hat{h}_i^2 \mod x^{2^{i+1}}. \tag{2.8}$$

You might remember that we have already used a similar recursion in (1.7) to compute an approximation of $1/b$ for some integer b based on Newton iteration. The following computation now shoes that $\hat{h}_i$ has indeed the desired property. Using induction, we may assume that $\hat{h}_i \cdot \hat{g} = 1 \mod x^{2^i}$, and thus $\hat{h}_i \cdot \hat{g} = 1 + s_i \cdot x^{2^i}$ for some $s_i \in R[x]$. From (2.8), we further conclude that there exists a polynomial $s \in R[x]$ with $\hat{h}_{i+1} := 2\hat{h}_i - \hat{g} \cdot \hat{h}_i^2 + s \cdot x^{2^{i+1}}$. Hence we obtain

$$\begin{aligned}
\hat{h}_{i+1} \cdot \hat{g} &= [\hat{h}_i \cdot (2 - \hat{g} \cdot \hat{h}_i) + s \cdot x^{2^{i+1}}] \cdot \hat{g} \\
&= \hat{h}_i \cdot \hat{g} \cdot (2 - \hat{g} \cdot \hat{h}_i) && \mod x^{2^{i+1}} \\
&= (1 + s_i \cdot x^{2^i}) \cdot (2 - s_i \cdot x^{2^i}) && \mod x^{2^{i+1}} \\
&= 1 - s^2 \cdot x^{2^{i+1}} && \mod x^{2^{i+1}} \\
&= 1 && \mod x^{2^{i+1}}
\end{aligned}$$

It follows that, for $i_0 := \lceil \log(n - m + 1) \rceil$, we have

$$\hat{h}_{i_0} \cdot \hat{g} = 1 \mod x^{n-m+1}.$$

Since $\hat{q}$ has degree at most $n - m$, we can now immediately compute $\hat{q}$ from $\hat{h}_{i_0}$ as

$$\hat{q} = \hat{f} \cdot \hat{h}_{i_0} \mod x^{n-m+1}.$$

Algorithm 8: Fast Polynomial Division

Input : A Euclidean ring $R[x]$, a polynomial $f \in R[x]$ of degree n, and a monic polynomial $g \in R[x]$ of degree m, with $m \leq n$.

Output: Polynomials $q, r \in R[x]$ with $f = q \cdot g + r$ and $\deg r < \deg g$.

1 $\hat{f} := x^n \cdot f(1/x)$ and $\hat{g} := x^m \cdot g(1/x)$.
2 $\hat{h}_0 := 1$
3 $i_0 := \lceil \log(n - m + 1) \rceil$
4 **for** $i = 1, \ldots, i_0$ **do**
5 Recursively define
$$\hat{h}_{i+1} := 2\hat{h}_i - \hat{g} \cdot \hat{h}_i^2 \mod x^{2^{i+1}}$$
6 $\hat{q} := \hat{f} \cdot \hat{h}_{i_0} \mod x^{n-m+1}$
7 $q := x^{n-m} \cdot \hat{q}(1/x)$
8 $r := f - q \cdot g$
9 **return** q, r

This further yields the polynomial $q(x) = x^{n-m} \cdot \hat{q}(1/x)$, and eventually the remainder

$$r(x) = f(x) - q(x) \cdot g(x).$$

We now estimate the computational cost of the above approach. The computation of $\hat{h}_i$ amounts for two multiplications and one addition of polynomials in $R[x]$ of degree $2^{2^{i+1}}$. Hence, we conclude that the cost for computing all polynomials $\hat{h}_i$ for $i = 0, \ldots, i_0$ is bounded by

$$4 \cdot [M_P(0) + M_P(2) + M_P(4) + \cdots + M_P(n)] < 8 \cdot M_P(n),$$

where $M_P(N)$ denotes the cost for adding or multiplying two polynomials in $R[x]$ of degree at most N. According to Theorem 2.1.16, we have $M_P(n) = O(n \log n \log\log n)$. The cost for the last two steps is comparable as there are two multiplications and one addition of polynomials of degree n or less. We fix this result:

Theorem 2.2.3. *Let $f \in R[x]$ be a polynomial of degree n, and g a monic polynomial of degree m, with $m \leq n$. Then, we can compute polynomials $q, r \in R[x]$ with*

$$f(x) = q(x) \cdot g(x) + r(x) \text{ and } \deg(r) < m$$

in a number of arithmetic operations in R bounded by $8 \cdot M_P(n) = O(n \log n \log\log n)$.

Exercise 2.2.4. *Let*

$$f = 30x^7 + 31x^6 + 32x^5 + 33x^4 + 34x^3 + 35x^2 + 36x + 37$$

and

$$g = 17x^3 + 18x^2 + 19x + 20$$

be two polynomials in $\mathbb{Z}/_{101}[x]$.

(i) Compute $f^{-1} \bmod x^4$.

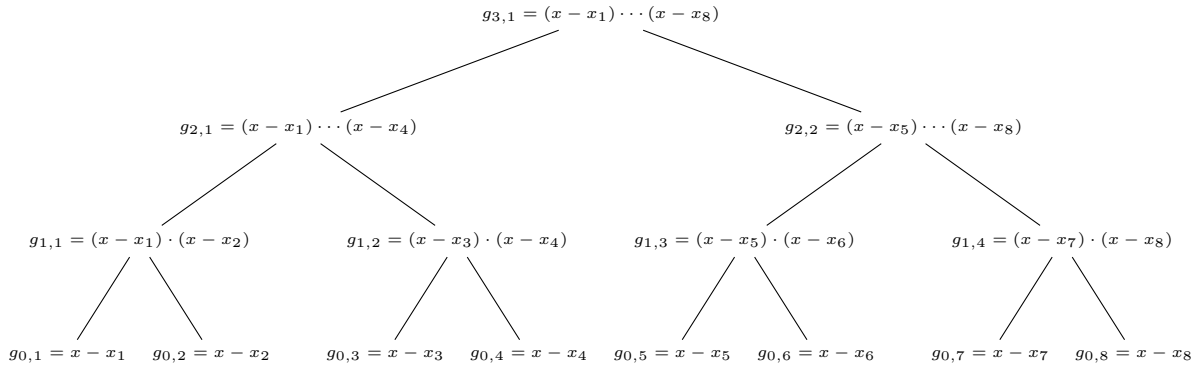

Figure 2.3: Illustration for the computation of all polynomials $g_{i,j}$ for $n = 8$.

(ii) Compute q and r in $\mathbb{Z}/101[x]$ with $f = q \cdot g + r$ and $\deg r < 3 = \deg g$.

Exercise 2.2.5. *Let p be an arbitrary prime and a an integer that is not divisible by p.*

- *Derive an algorithm to compute an integer $b \in \{1, \ldots, p^\ell - 1\}$ with $a \cdot b \equiv 1 \bmod p^\ell$, where $\ell \neq 0$ is an arbitrary given integer.*

  Hint: Use Newton iteration.

- *Compute* $97^{-1} \bmod 4096$.

Exercise 2.2.6. *Let $f, g \in \mathbb{Q}[x]$ be polynomials of degrees m and n, respectively, and $m \geq n$. If the length of the numerators and denominators of the coefficients of f and g are less than L, then the coefficients of q and r, with*

$$f = q \cdot g + r \text{ and } \deg r < \deg g,$$

have bitsize $O(nL)$.

There are a series of applications of the fast division algorithm. We start with an algorithm [MB72] due to Moenck and Borodin (from 1972) that allows us to evaluate a polynomial $f \in R[x]$ of degree n at n points $x_1, \ldots, x_n \in R$ in only $O(M_P(n) \cdot \log n) = \tilde{O}(n)$ primitive operations. This can be considered as a generalization of the FFT algorithm. For the seek of a simplified presentation, we again assume that $n = 2^k$ is a power of two. Starting with linear forms $g_{0,j}(x) := x - x_j$, we recursively compute

$$g_{i,j}(x) := g_{i-1,2j-1}(x) \cdot g_{i-1,2j} = (x - x_{(j-1)\cdot 2^i+1}) \cdots (x - x_{j \cdot 2^i}) \text{ for } i = 1, \ldots, k \text{ and } j = 1, \ldots, \frac{n}{2^i}.$$

Notice that each $g_{i,j}$ is a product of 2^i linear forms, and that $g_{k,1}(x) = \prod_{i=1}^{n}(x - x_i)$; see also Figure 2.3 for an illustration in the case $n = 8$.

In the second step, we start with $r_{k,1} := f$, and recursively compute

$$\begin{aligned} r_{k-i,j} &:= f(x) \mod g_{k-i,j} \text{ for } i = 1, \ldots, k \text{ and } j = 1, \ldots, \frac{n}{2^{k-i}} \\ &= r_{k-i+1,\lceil j/2 \rceil} \mod g_{k-i,j}, \end{aligned}$$

Algorithm 9: Fast Multipoint Evaluation

Input : A Euclidean ring $R[x]$, a polynomial $f \in R[x]$ of degree $n = 2^k$, with $k \in \mathbb{N}$, and $x_1, \dots, x_n \in R$

Output: $(f(x_1), \dots, f(x_n))$

$g_{0,j}(x) := x - x_j$ for $j = 1, \dots, n$

for $i = 1, \dots, k$ **do**

  Recursively define

$$g_{i,j}(x) := g_{i-1,2j-1}(x) \cdot g_{i-1,2j} \text{ for } j = 1, \dots, 2^{k-i}.$$

$r_{k,1} := f$

for $i = 1, \dots, k$ **do**

  Recursively define

$$r_{k-i,j} := r_{k-i+1,\lceil j/2 \rceil} \mod g_{k-i,j} \text{ for } j = 1, \dots, 2^i.$$

return $(r_{0,1}, \dots, r_{0,n})$

where the latter equality follows from the fact that $g_{k-i,j}$ divides $g_{k-i+1,\lceil j/2 \rceil}$ and

$$\begin{aligned} f(x) &= q_{k-i+1,\lceil j/2 \rceil}(x) \cdot g_{k-i+1,\lceil j/2 \rceil} + r_{k-i+1,\lceil j/2 \rceil} \\ &= \left[q_{k-i+1,\lceil j/2 \rceil}(x) \cdot \frac{g_{k-i+1,\lceil j/2 \rceil}}{g_{k-i,j}} \right] \cdot g_{k-i,j} + r_{k-i+1,\lceil j/2 \rceil} \end{aligned}$$

for some $q_{k-i+1,j} \in R[x]$. Since each $r_{i,j}$ is the remainder of a polynomial division by some $g_{i,j'}$, it follows that $r_{i,j}$ has degree less than 2^i. Further notice that

$$r_{0,j} = f(x) \mod g_{0,j}(x) = f(x) \mod (x - x_j) = f(x_j),$$

thus we have computed all values $f(x_1), \dots, f(x_n)$; see Algorithm 9 for pseudocode.

It remains to bound the cost for running Algorithm 9. The computation of each $g_{i,j}$ amounts for multiplying two polynomials in $R[x]$ of degree 2^{i-1}. For the computation of each $r_{i,j}$, we need to carry out one division with remainder between a polynomial of degree less than 2^{i+1} and a polynomial of degree 2^i. Hence, from Theorem 2.1.16 and 2.2.3, we conclude that the total cost is bounded by

$$\sum_{i=1}^{k} 2^{k-i} \cdot 8M_P(2^i)) = \sum_{i=1}^{k} 8M_P(n) = 8 \log n \cdot M_P(n).$$

We fix this result:

Theorem 2.2.7. *Let $f \in R[x]$ be a polynomial of degree n, and $x_1, \dots, x_n \in R$. Then, Algorithm 9 computes all values $f(x_i)$, for $i = 1, \dots, n$, using at most $6 \log n \cdot M_P(n) = O(n \log^2 n \log \log n)$ arithmetic operations in R.*

In the next step, we focus on the inverse problem, that is, given n distinct elements $x_1, \dots, x_n \in R$, with $n = 2^k$, and corresponding values $v_1, \dots, v_n \in R$, determine a polynomial

Algorithm 10: Fast Polynomial Interpolation

Input : A Euclidean ring $R[x]$, points $x_1, \dots, x_n \in R$, with $n = 2^k$ and $k \in \mathbb{N}$, such that $x_i - x_j$ is a unit in R for all $i \neq j$, and values $v_1, \dots, v_n \in R$.

Output: A polynomial $f \in R[x]$ of degree less than n such that $f(x_i) = v_i$ for all $i = 1, \dots, n$.

1 Compute all polynomials $g_{i,j}$, with $i = 0, \dots, k$ and $j = 1, \dots, n/2^i$.
2 $G := \frac{\partial}{\partial x} g_{k,1}$
3 Use Algorithm 9 to compute $\lambda_i := G(x_i)$ for all $i = 1, \dots, n$.
4 Compute $f_{0,j} := \mu_j := v_j / \lambda_j$ for $j = 1, \dots, n$.
5 **for** $i = 1, \dots, k$ **do**
6 Recursively define

$$f_{i,j}(x) := g_{i-1,2j-1}(x) \cdot f_{i-1,2j-1} + g_{i-1,2j}(x) \cdot f_{i-1,2j} \text{ for } j = 1, \dots, 2^{k-i}.$$

7 **return** $f_{k,1}$

$f(x) \in R[x]$ of degree less than n such that $f(x_i) = v_i$ for all i. We will now give a very efficient method for interpolation problem under the additional assumption that $x_i - x_j$ is a unit in R for all pairs i, j with $i \neq j$. Using *Lagrange interpolation*, we have

$$f(x) = \sum_{i=1}^{n} v_i \cdot \prod_{j \neq i} \frac{x - x_j}{x_i - x_j} = \sum_{i=1}^{n} v_i \cdot \underbrace{\frac{1}{\prod_{j \neq i}(x_i - x_j)}}_{=:\lambda_i} \cdot \underbrace{\prod_{j \neq i}(x - x_j)}_{=:g_i(x)}.$$

Notice that $\lambda_i' := \lambda_i^{-1} = \prod_{j \neq i}(x_i - x_j) = g_{k,1}'(x_i)$, where $\frac{\partial}{\partial x} g_{k,1}(x)$ is the (formal) derivative of $g_{k,1} = \prod_{j=1}^{n}(x - x_j)$ as defined in the fast multipoint evaluation algorithm above.[8] Hence, we may first compute $g_{k,1}$ and its derivative $g_{k,1}'$, and then use the fast multipoint evaluation algorithm to evaluate $g_{k,1}'$ at the points x_i to compute the values λ_i'. Then, dividing v_i by λ_i' yields the values $\mu_i := v_i / \lambda_i'$. The cost for this step is bounded by $O(\log n \cdot M_P(n))$ arithmetic operations in R plus n divisions in R. Now, in order to compute $f_{k,1}(x) := f(x) = \sum_{i=1}^{n} \mu_i \cdot \prod_{j \neq i}(x - x_j)$, we write

$$f_{k,1}(x) = g_{k-1,1}(x) \cdot \underbrace{\sum_{i=n/2+1}^{n} \mu_i \cdot \prod_{j=n/2+1, j \neq i}^{n} (x - x_j)}_{=:f_{k-1,1(x)}} + g_{k-1,2}(x) \cdot \underbrace{\sum_{i=1}^{n/2} \mu_i \cdot \prod_{j=1, j \neq i}^{n/2} (x - x_j)}_{=:f_{k-1,2}(x)}.$$

Hence, we can recursively compute the polynomial f from the values μ_i and the polynomials $g_{i,j}$; see Algorithm 10. A completely analogous analysis as for the fast multipoint evaluation then yields the following result:

[8] For a polynomial $f = \sum_{i=0}^{n} a_i \cdot x^i \in R[x]$, the formal derivative is defined as $\frac{\partial}{\partial x} f := \sum_{i=1}^{n} i \cdot a_i \cdot x^{i-1}$. Then, for any two polynomials $f, g \in R[x]$, it holds that $\frac{\partial}{\partial x}(f \cdot g) = \frac{\partial}{\partial x} f \cdot g + \frac{\partial}{\partial x} g \cdot f$, and thus $g_{k,1}'(x) = \prod_{j \neq i}(x - x_i) + (x - x_i) \cdot \left(\prod_{j \neq i}(x - x_i)\right)'$. It follows that $\frac{\partial}{\partial x} g_{k,1}(x_i) = \prod_{j \neq i}(x_i - x_j)$.

Theorem 2.2.8. *Let $x_1, \ldots, x_n \in R$ be arbitrary points in R such that $x_i - x_j$ is a unit for all $i \neq j$, and let $v_1, \ldots, v_n \in R$ be arbitrary points in R. Then, computing the unique polynomial $f \in R[x]$ of degree less than n with $f(x_i) = v_i$ for all i uses $O(\log n \cdot M_P(n)) = O(n \log^2 n \log\log n)$ additions and multiplication plus n divisions in R.*

We give a final application of the fast division algorithm, that is, the computation of a Taylor shift $x \mapsto m + x$ for a polynomial $f = \sum_{i=0}^{n} a_i \cdot x^i \in R[x]$ of degree n. Given the coefficients of f and a point $m \in R$, we aim to compute the coefficients of $\hat{f}(x) := f(m+x) = \sum_{i=0}^{n} \hat{a}_i \cdot x^i$. The idea is to reduce the problem to a fast multipoint evaluation followed by an interpolation. Suppose that there exist points $\hat{x}_1, \ldots, \hat{x}_n$ such that $\hat{x}_i - \hat{x}_j$ is a unit in R for all $i \neq j$. Then, we evaluate f at the points $x_i := m + \hat{x}_i$, and eventually interpolate $\hat{f}$ from its values $\hat{f}(\hat{x}_i) = f(x_i)$ at the points $\hat{x}_i$. Notice that, if R supports the FFT, we may also choose $\hat{x}_i = \omega^i$, with ω a $2n$-th root of unity. Then, the interpolation step amounts for a single FFT computation. The following result immediately follows from Theorem 2.2.7 and Theorem 2.2.8.

Theorem 2.2.9. *Suppose that R contains elements $x_1, \ldots, x_n$ such that $x_i - x_j$ is a unit in R for all $i \neq j$ (or R supports the FFT). Then, for an arbitrary polynomial $f \in R[x]$ and a point $m \in R$, we can compute the coefficients of $f(m + x)$ using $O(\log n \cdot M_P(n)) = O(n \log^2 n \log\log n)$ additions and multiplication plus n divisions in R.*

2.3 Fast Polynomial Arithmetic in $\mathbb{C}[x]$

We finally investigate in fast numerical variants of the algorithms presented in the previous two sections. Here, we assume that the coefficients of the input polynomial $f \in \mathbb{C}[x]$ (or any other input points in $\mathbb{C}$) are only given up to a certain precision, that is, for arbitrary $\rho \in \mathbb{N}$, we may ask for a dyadic approximation in $\mathbb{F} = \mathbb{F}_{2,\rho}$ of each coefficient (or of each point) to an error less than $2^{-\rho}$. For short, we call a corresponding approximation $\tilde{f}$ of f an *(absolute) ρ-bit approximation of f*. We start with a method for the approximate computation of a product of two polynomials.

Theorem 2.3.1. *Let $f, g \in \mathbb{C}[x]$ be polynomials of degree less than n and with coefficients of absolute value less than 2^L. Then, an ℓ - bit approximation $\tilde{h} = \sum_{i=0}^{2n-2} \tilde{h}_i \cdot x^i$ of the product $h = \sum_{i=0}^{2n-2} h_i \cdot x^i := f \cdot g$ (i.e. $|h_i - \tilde{h}_i| < 2^{-\ell}$ for all i) can be computed using $O(n(L + \ell))$ primitive operations. For this, we need ρ-bit approximations of f and g for some ρ of size $O(\log n + \ell + L)$.*

Proof. We reduce the multiplication of f and g to that of integer polynomials. For a non-negative integer ρ, consider ρ-bit approximations $\tilde{f} = \sum_{i=0}^{n-1} \tilde{a}_i \cdot x^i$ and $\tilde{g} = \sum_{i=0}^{n-1} \tilde{b}_i \cdot x^i$ of f and g. Then, $\tilde{h} = \sum_{i=0}^{2n-2} \tilde{c}_i \cdot x^i := \tilde{f} \cdot \tilde{g}$ constitutes an approximation of $h = \sum_{i=0}^{2n-2} c_i x^i := f \cdot g$ with

$$|h_i - \tilde{h}_i| < n \cdot [2^{L+1} \cdot 2^{-\rho} + 2^{-2\rho}] < 2^{L+2+\lceil \log n \rceil - \rho},$$

which follows from the fact that $|\tilde{a}_i \tilde{b}_j - a_i b_j| < (|a_i| + |b_j|) \cdot 2^{-\rho} + 2^{-2\rho}$ for all i, j. Hence, in order to guarantee that $\tilde{h}$ approximate h to an error less than $2^{-\ell}$, it suffices to choose $\rho := L + 2 + \lceil \log n \rceil + \ell$. In order to compute the product $\tilde{f} \cdot \tilde{g}$, we first compute the product $(2^\rho \cdot \tilde{f}) \cdot (2^\rho \cdot \tilde{g})$ of integer polynomials and then shift the coefficients of the result by 2ρ bits. The latter product can be computed in $\tilde{O}(n(\ell + L))$ primitive operations according to Theorem 2.1.12. □

For a corresponding numerical variant of the fast polynomial division, we have to work harder. We start with following lemma:

Lemma 2.3.2. *Let $f \in \mathbb{C}[x]$ be a polynomial of degree n, $g \in \mathbb{C}[x]$ a monic polynomial of degree m, with $m \leq n$, and $q, r \in \mathbb{C}[x]$ with $f = q \cdot g + r$ and $\deg r < m$. Then, it holds that*

$$\log \max(\|q\|_\infty, \|r\|_\infty) = O(L + n + (n-m) \cdot \Gamma),$$

where L and Γ are non-negative integers with $\max(\|f\|_\infty, \|g\|_\infty) < 2^L$ and $|z| < 2^\Gamma$ for any complex root z of g.

Proof. Let $f(x) = a_0 + \cdots + a_n \cdot x^n$ and $g(x) = b_0 + \cdots + b_m \cdot x^m$, then we have

$$\begin{aligned} \frac{f(x)}{x^{n-m} \cdot g(x)} &= \frac{q(x) \cdot g(x) + r(x)}{x^{n-m} \cdot g(x)} = \frac{q(x)}{x^{n-m}} + \frac{r(x)}{x^{n-m} \cdot g(x)} \\ &= q_{n-m} + \frac{q_{n-m-1}}{x} + \cdots + \frac{q_0}{x^{n-m}} + \cdots \end{aligned} \tag{2.9}$$

where $q(x) = q_0 + \cdots + q_{n-m} \cdot x^{n-m}$. Here, we use the fact that $r(x)/g(x)$ is a holomorphic function in the domain $D := \{x \in \mathbb{C} : |x| > 2^\Gamma\}$. Using a corresponding result from Complex Analysis, we thus conclude that it can be written as a Laurent series $\sum_{i=-\infty}^{\infty} c_i \cdot x^i$, which converges for all $x \in D$. We further remark that $c_i = 0$ for all $i \geq 0$ as, otherwise, $\lim_{x \mapsto \infty} \frac{|x \cdot r(x)|}{|g(x)|} = \infty$, which contradicts the fact that $\deg r < m$. Now, from (2.9), we conclude that

$$\frac{f}{x^{n-m} \cdot g(x)} \cdot x^{i-1} = q_{n-m} \cdot x^{i-1} + \cdots + \frac{q_{n-m-i}}{x} + \cdots,$$

and thus the Residue Theorem yields that

$$\frac{1}{2\pi \mathbf{i}} \oint_{|x|=2^{\Gamma+1}} \frac{f(x)}{x^{n-m-i+1} \cdot g(x)} dx = q_{n-m-i} \text{ for all } i = 0, \ldots, n-m$$

or

$$\frac{1}{2\pi \mathbf{i}} \oint_{|x|=2^{\Gamma+1}} \frac{f(x)}{x^{j+1} \cdot g(x)} dx = q_j \text{ for all } j = 0, \ldots, n-m.$$

For $|x| = 2^{\Gamma+1}$, we have $|f(x)| \leq (n+1) \cdot 2^L \cdot 2^{n(\Gamma+1)}$ and $|g(x)| \geq 2^{m\Gamma}$. Hence, it follows that the absolute value of the integrand is upper bounded by $B = (n+1) \cdot 2^{L+n+(n-m)\Gamma}$. We conclude that each coefficient q_j of q is bounded by $B \cdot 2^{\Gamma+1} = 2^{O(L+n+(n-m)\Gamma)}$. The claim regarding the size of the coefficients of r then immediately follows from the bound on $\|q\|_\infty$ and the fact that $r = f - q \cdot g$. □

Using the above lemma, we can now derive a bound on the polynomials $\hat{h}_i \in \mathbb{C}[x]$ as computed in Algorithm 8. Namely, let $\hat{h}_i$ be a polynomial of degree less than $2^i - 1$ such that $\hat{h}_i \cdot \hat{g} = 1 \mod x^{2^i}$, with $g(x) = x^m \cdot g(1/x)$ and $g \in \mathbb{C}[x]$ a monic polynomial of degree m. Then, there exists a polynomial $s_i \in \mathbb{C}[x]$ of degree less than m such that

$$\hat{g}(x) \cdot \hat{h}_i = 1 + x^{2^i} \cdot s_i(x) \Rightarrow \underbrace{x^m \cdot \hat{g}(1/x)}_{=g(x)} \cdot \underbrace{x^{2^i} \cdot \hat{h}_i(1/x)}_{=:h_i} = x^{m+2^i} + \underbrace{x^m \cdot s_i(1/x)}_{\hat{s}_i}.$$

Hence, the polynomials $h_i := x^{2^i} \cdot \hat{h}_i(1/x)$ and $\hat{s}_i := x^m \cdot s_i(1/x)$ are the quotient and remainder obtained dividing x^{m+2^i} by $g(x)$. Lemma 2.3.2 then yields that

$$\log \|\hat{h}_i\|_\infty = \log \|h_i\|_\infty = O(\log \|g\|_\infty + n + n\Gamma),$$

with $\Gamma \geq 0$ a bound on $\log |z_i|$ for every complex root of g. In the $(i+1)$ - st iteration step in Algorithm 8, we compute $\hat{h}_{i+1} = 2\hat{h}_i - \hat{g} \cdot \hat{h}_i^2 \mod x^{2^{i+1}}$. Hence, if we use ρ-bit approximations of $\hat{h}_i$ and $\hat{g}$ instead of the exact polynomials h_i and g, then Theorem 2.3.1 shows that we obtain an approximation of $\hat{h}_{i+1}$ to an error less than $2^{-\rho+O(\log \|g\|_\infty + n + n\Gamma)}$. In other words the precision loss in each iteration is bounded by $O(\log \|g\|_\infty + n + n\Gamma)$. Since there are at most $\lceil \log n \rceil$ many iterations, the total precision loss is bounded by $\tilde{O}(\log \|g\|_\infty + n + n\Gamma)$. Hence, we can use fixed point arithmetic with precision $\rho = \ell + \tilde{O}(\log \|g\|_\infty + n + n\Gamma)$ to guarantee an output error of less than $2^{-\ell}$.

Theorem 2.3.3. *Let f and g be polynomials as in Lemma 2.3.2. Then, computing ℓ-bit approximations $\tilde{q}$ and $\tilde{r}$ of q and r uses $\tilde{O}(n(\ell + L + n + n\Gamma))$ primitive operations. For this, we need ρ-bit approximations of the polynomials f and g for some ρ of size $\ell + \tilde{O}(L + n + n\Gamma)$.*

We briefly summarize out findings from Theorems 2.3.1 and 2.3.3: A multiplication of two polynomials $f, g \in \mathbb{C}[x]$ using fixed point arithmetic with precision ρ yields a loss in precision bounded by $O(\log n + \log \max(\|f\|_\infty, \|g\|_\infty))$, whereas the precision loss of a corresponding division with remainder is bounded by $\tilde{O}(n + \log \max(\|f\|_\infty, \|g\|_\infty + n\Gamma)$. Now, what can we conclude about the precision loss in the fast multipoint evaluation algorithm? The polynomials $g_{i,j}$ are products of linear forms $x - x_s$, hence $\log \|g_{i,j}\|_\infty$ is bounded by $O(n\Gamma^*)$ with $\Gamma^* := \max(1, \log \max_{i=1,\ldots,n} |x_i|)$. Since the depth of the recursion is $\log n$, we conclude that the precision loss is bounded by $O(n\Gamma^* \cdot \log n)$. Now, for the divisions in the algorithm, notice that we start with $r_{k,1} = f$. In each step of the recursion, we divide a previously computed remainder $r_{i,j}$ by some $g_{i',j'}$. Further notice that $r_{i,j} = f \mod g_{i,j}$, and thus $\log \|r_{i,j}\|_\infty = O(L + n\Gamma^*)$ according to Lemma 2.3.2. It follows that the precision loss in each of the considered divisions is bounded by $\tilde{O}(L + n\Gamma^*)$. Now, since the depth of the recursion is bounded by $O(\log n)$, we conclude that the total loss in precision is bounded by $\tilde{O}((L + n\Gamma^*))$. Thus, in order to guarantee an output error of size less than $2^{-\ell}$, it suffices to use fixed point arithmetic with a precision of size $\ell + \tilde{O}(L + n\Gamma^*)$.

Theorem 2.3.4. *Let $f \in \mathbb{C}[x]$ be a polynomial of degree n with coefficients of absolute value bounded by 2^L, with $L \in \mathbb{N}_{\geq 1}$, and let $x_1, \ldots, x_n \in \mathbb{C}$ be arbitrary points of absolute value less than 2^{Γ^*}, with $\Gamma^* \geq 1$. For an arbitrary non-negative number ℓ, we can compute ℓ-bit approximations $\tilde{v}_i$ of all values $v_i := f(x_i)$ using $\tilde{O}(n \cdot (\ell + L + \Gamma^*))$ primitive operations. For this, we need ρ-bit approximations of f and the points x_i for some ρ of size $\ell + \tilde{O}(L + n\Gamma^*)$.*

Corollary 2.3.5. *Let $f \in \mathbb{C}[x]$ be a polynomial of degree n with coefficients of absolute value bounded by 2^L, with $L \in \mathbb{N}_{\geq 1}$, and $m \in \mathbb{C}$ be an arbitrary point of absolute value less than 2^{Γ^*}, with $\Gamma^* \geq 1$. For an arbitrary non-negative number ℓ, we can compute an ℓ-bit approximations of $\hat{F}(x) = F(m + x)$ using $\tilde{O}(n \cdot (\ell + L + \Gamma^*))$ primitive operations. For this, we need ρ-bit approximations of f and m for some ρ of size $\ell + \tilde{O}(L + n\Gamma^*)$.*

Proof. The proof is left as an exercise. □

Exercise 2.3.6. *Let $f \in \mathbb{Z}[x]$ be an integer polynomial of degree less than n with coefficients of absolute value less than 2^L. Furthermore, let $x_1, \ldots, x_n$ be n be distinct rational points in $[0, 1]$ of bitsize ℓ (i.e., $x_i = p_i/q_i \in [0, 1]$ with integers p_i and q_i of absolute value less than 2^ℓ).*

We say that the point x_i is large *for f among $X := \{x_1, \ldots, x_n\}$ if*

$$4 \cdot |f(x_i)| \geq \max_{1 \leq j \leq n} |f(x_j)| =: \lambda.$$

- *Determine the cost of finding a large point in a naive way, that is, by evaluating f at all points x_j exactly.*
- *Show how to find a large point in $\tilde{O}(n(L + \log \max(1, \lambda^{-1})))$ bit operations.*

  Hint: Use approximate multipoint evaluation with increasing precision.

Exercise 2.3.7 (Sparse Approximate Polynomial Evaluation)**.** *Let $f = \sum_{j=1}^{k} a_{i_j} \cdot x^{i_j} \in \mathbb{C}[x]$ be a k-*nomial *of degree n with k non-zero coefficients of absolute value bounded by 2^L, with $L \in \mathbb{N}_{\geq 1}$, and let $x_0 \in \mathbb{C}$ be an arbitrary point of absolute value less than 2^{Γ^*}, with $\Gamma^* \geq 1$. For an arbitrary non-negative number ℓ, we can compute an ℓ-bit approximations of $v := f(x_0)$ using $\tilde{O}(k \cdot (\ell + L + \Gamma^*))$ primitive operations. For this, we need ρ-bit approximations of f and x_0 for some ρ of size $\ell + \tilde{O}(L + n\Gamma^*)$.*

Hint: Use Exercise 1.1.4

Chapter 3

The Extended Euclidean Algorithm and (Sub-) Resultants

3.1 Gauss' Lemma

In what follows, we assume that R is a commutative ring with 1.

Definition 3.1.1 (Integral Domain)**.** *R is called an* integral domain *if it does not contain a zero-divisor, that is, if there exists no $a, b \in R \setminus \{0\}$ with $a \cdot b = 0$. We further use R^* to denote the set of invertible elements in R.*

Examples. we give several examples for commutative rings that are (not) integral domains:

1. $\mathbb{Z}$ is an integral domain and $\mathbb{Z}^* = \{-1, 1\}$.
2. $\mathbb{Z}/q$ is an integral domain if and only if q is a prime.
3. If R is an integral domain, then $R[x_1, \ldots, x_n]$ is an integral domain.

Definition 3.1.2. *Let R be an integral domain and $a, b \in R$.*

1. *a is a* divisor *of b iff there exists a $c \in R$ with $a \cdot c = b$. We write $a|b$.*
2. *a, b are* associated *iff there exists a $u \in R^*$ with $a = u \cdot b$. We write $a \sim b$.*
3. *$q \in R \setminus R^*$ is* irreducible *if $q = a \cdot b$, with $a, b, \in R$, implies that $a \in R^*$ or $b \in R^*$.*
4. *$p \in R \setminus R^*$ is* prime *if $p|a \cdot b$, with $a, b \in R$, implies that $p|a$ or $p|b$.*

It holds that

Theorem 3.1.3. *In an integral domain R, it holds that*

$$p \in R \text{ is prime } \Rightarrow p \text{ is irreducible.}$$

Proof. Suppose that p is prime and that $p = a \cdot b$ with $a, b \in R$. Hence, p divides a or b. W.l.o.g. we may assume that p divides a, hence there exists a $c \in R$ with $p = p \cdot c \cdot b$, or equivalently $p \cdot (1 - b \cdot c) = 0$. Since R is an integral domain, we must have $1 - b \cdot c = 0$, and thus $b \in R^*$ with $b^{-1} = c$. □

Definition 3.1.4 (Ideal)**.** *A subset $I \subset R$ in a ring R is called an* ideal *if, for all $a, b \in I$ and all $r \in R$, we have*

$$a + b \in I \text{ and } r \cdot a \in I.$$

If there exist elements $a_1, \ldots, a_n \in R$ such that each $a \in I$ can be written as

$$a = r_1 \cdot a_1 + \cdots + r_n \cdot a_n, \text{ with } r_i \in R,$$

then we say that I is generated *by $a_1, \ldots, a_n$. For short, we write $I = \langle a_1, \ldots, a_n \rangle$. If I is generated by only one element, we say that I is a* principal ideal*. R is called a* principal ideal ring *(or just principal) if each ideal in R is a principal ideal.*

Examples.

1. Each polynomial $f(x) \in \mathbb{Z}[x]$ that has a root at $x = 0$ of multiplicity k is contained in $I := \langle x^k \rangle$.
2. Each polynomial $f(x, y) \in \mathbb{Z}[x, y]$ with $f(1, 2) = 0$ is contained in $I := \langle x - 1, y - 2 \rangle$.
3. $\mathbb{Z}$ is principal but $\mathbb{Z}[x]$ is not principal.
4. $\mathbb{Q}[x]$ is principal.

Exercise 3.1.5. *Show that every Euclidean domain is principal.*

Definition 3.1.6 (Factorial Ring)**.** *An integral domain R is called a* factorial ring *if, for all $a \in R \setminus R^*$, there exists a factorization*

$$a = p_1 \cdots p_r$$

of a into primes $p_1, \ldots, p_r$.

We remark that the above factorization of a into primes is unique.

Theorem 3.1.7. *In a factorial ring R, the factorization $a = p_1 \cdots p_r$ of an element $a \in R \setminus R^*$ into primes p_i is unique up to ordering and a unit in R.*

Proof. Suppose that $a = p_1 \cdots p_r = q_1 \cdots q_s$ with primes p_i, q_j. Since p_1 is prime, there must be a q_j with $p | q_j$. W.l.o.g. we assume that $q_1 = w_1 \cdot p_1$ for some $w \in R$. Since q_1 is irreducible, we further conclude that $w \in R^*$. Hence, we get $p_2 \cdots p_r = w \cdot q_2 \cdots q_s$. Notice that p_2 does not divide w as otherwise, p_2 would be also invertible. Hence, the claim follows by induction. □

Definition 3.1.8 (Noetherian Ring)**.** *A ring R is* Noether *if each ideal of R is finitely generated.*

Examples. We give examples of rings that are (not) Noether:

1. $\mathbb{Z}, \mathbb{Q}, \mathbb{R}, \mathbb{C}$ are Noether.
2. $\mathbb{Q}[x]$ is Noether. This follows from the extended Euclidean algorithm, which shows that $\langle f, g \rangle = \langle \gcd(f, g) \rangle$ for any two polynomials $f, g \in \mathbb{Q}[x]$. We give an independent proof of a more general result in the following theorem.

3. $\mathbb{Q}[x_1, x_2, \ldots]$ is not Noether.

4. The ring $\mathrm{Int}(\mathbb{Z}) := \{f \in \mathbb{Q}[x] : f(x) \in \mathbb{Z} \text{ for all } x \in \mathbb{Z}\}$ of so-called integer-valued polynomials is not Noether. One can show (the proof is non-trivial) that $I := \langle x, x(x-1)/2, x(x-1)(x-2)/3, \ldots, \rangle$ is not finitely generated.

The crucial fact is that a Noetherian ring R is that it passes this property to its corresponding polynomial ring $R[x]$.

Theorem 3.1.9 (Hilbert's Basis Theorem)**.** *If R is Noether, then $R[x]$ is Noether as well. In particular, $R[x_1, \ldots, x_n]$ is Noether for $R = \mathbb{Z}, \mathbb{Q}, \mathbb{R}, \mathbb{C}$.*

Proof. For a polynomial $f(x) = a_0 + \cdots a_n \cdot x^n \in R[x]$, we use $\mathrm{LT}(f) = a_n \cdot x^n$ to denote the *leading term* of f, and $\mathrm{LC}(f) = a_n$ to denote the *leading coefficient* of f. We call f monic if $\mathrm{LC}(f) = 1$. Now, suppose that $R[x]$ is not Noether, then there exists an ideal $I \subset R[x]$ that is not finitely generated. Let $f_1 \in I$ with $\deg f_1$ be an element in I of minimal degree, f_2 be an element in $I_2 \setminus \langle f_1 \rangle$ of minimal degree, etc. Then, it follows that

$$\deg(f_1) \leq \deg(f_2) \leq \cdots \leq \deg(f_k) \leq \cdots,$$

and thus we obtain an ascending chain of ideals in R

$$\underbrace{\langle \mathrm{LC}(f_1) \rangle}_{=:J_1} \subset \underbrace{\langle \mathrm{LC}(f_1), \mathrm{LC}(f_2) \rangle}_{=:J_2} \subset \cdots$$

We first show that the above chain is strictly ascending, that is, $J_k \neq J_{k+1}$ for all k. Namely, if $J_k = J_{k+1}$, then there exist $b_j \in R$ with $\mathrm{LC}(f_{k+1}) = \sum_{j=1}^{k} b_j \cdot \mathrm{LC}(f_j)$, and thus it follows that

$$g := \sum_{j=1}^{k} b_j \cdot x^{\deg(f_{k+1}) - \deg(f_k)} \cdot f_j$$

is contained in $\langle f_1, \ldots, f_k \rangle$ and that it has the same leading coefficient as f_{k+1}. Since $f_{k+1} \notin \langle f_1, \ldots, f_k \rangle$, we also have $g - f_{k+1} \notin \langle f_1, \ldots, f_k \rangle$. In addition, $g - f_{k+1}$ has lower degree than f_{k+1}, which contradicts our choice of f_{k+1}.

Now, let $J := \bigcup_{k=1}^{\infty} J_k$ be the union of all J_k. J is an ideal in R, hence finitely generated by elements $a_1, \ldots, a_r$. Since each a_i is contained in some J_{k_i}, J must be contained in the union of all J_{k_i}, with $i = 1, \ldots, r$. However, the fact that the sequence of the ideals J_k is strictly increasing. □

Theorem 3.1.10. *Let R be Noether, then each $a \in R \setminus R^*$ can be written as*

$$a = q_1 \cdots q_r$$

with irreducible $q_1, \ldots, q_r \in R$.

Proof. If a is irreducible, then there is nothing to prove. Otherwise, there exist $a_1, b_1 \notin R^*$ with $a = a_1 \cdot b_1$. If a_1 and b_1 are both irreducible, we are done. Otherwise, we may assume that $a_1 = a_2 \cdot b_2$ with $a_2, b_2 \in R^*$. Continuing with this approach, we obtain a sequence of principal ideals

$$\langle a \rangle \subset \langle a_1 \rangle \subset \langle a_2 \rangle \subset \cdots$$

Since R is Noether, the ideal generated by all elements a_i is finitely generated, and thus the above sequence must become stationary. □

We have already seen that $\mathbb{Z}$ is factorial. Our goal is to prove that $\mathbb{Z}[x_1, \ldots, x_n]$ is factorial as well. Since $\mathbb{Z}[x_1, \ldots, x_n]$ is Noether, Theorem 3.1.10 guarantees that each element $f \in \mathbb{Z}[x_1, \ldots, x_n]$ can be written as a product of irreducible factors. It remains to answer the question whether this factors are also prime and whether they are unique (up to ordering).

Theorem 3.1.11. *In a factorial ring R, we have*

$$q \in R \text{ irreducible } \Leftrightarrow q \text{ prime.}$$

Proof. Theorem 3.1.3 already shows that q prime implies that q is irreducible. For the counter direction, write q as product $q = p_1 \cdots p_r$ with primes p_i. Since q is irreducible, we must have $r = 1$ and $q = p_1$. □

Exercise 3.1.12. *In a principal ideal domain R, it holds that*

$$q \in R \text{ irreducible } \Leftrightarrow q \text{ prime.}$$

Definition 3.1.13 (Primitive Polynomials)**.** *Let R be factorial, and $f = \sum_{i=0}^{n} a_i \cdot x^i \in R[x]$. Then, f is called* pimitive *if there exists no $a \in R \setminus R^*$ that divides each coefficient of f. We call $\operatorname{cont}(f) \in R$ a content[1] of f if $\operatorname{cont}(f)$ divides each coefficient of f and $f/\operatorname{cont}(f)$ is primitive.*

Example. The polynomial $f(x) = 7x^2 + 3x + 6 \in \mathbb{Z}[x]$ is primitive, however, $g(x) = 12x^2 + 3x + 6 \in \mathbb{Z}[x]$ is not primitive.

Lemma 3.1.14 (Gauss'Lemma)**.** *Let R be a factorial ring and*

$$F := \left\{ \frac{a}{b} : a, b \in R \text{ and } b \neq 0 \right\}$$

its quotient field[2]*. Then, it holds:*

1. *The product of two primitive polynomials $f, g \in R[x]$ is again primitive.*
2. *A polynomial $f \in R[x]$ is irreducible (over R) if and only if it is irreducible over F.*

Proof. For simplicity, we assume that $R = \mathbb{Z}$ and $F = \mathbb{Q}$. The argument for the general case is completely analogous.

For (1), suppose that there exists a prime p that divides each coefficients of $f \cdot g$. Then, let i and j minimal such that p does not divide a_i and p does not divide b_j. The coefficient c_{i+j} of x^{i+j} in the product $f \cdot g$ is given as

$$c_{i+j} = a_i \cdot b_j + a_{i-1} \cdot b_{j+1} + \cdots + a_{i+1} \cdot b_{j-1} + \cdots$$

Since p divides each term in the above sum except $a_i \cdot b_j$, we conclude that p does not divide c_{i+j}, which contradicts our assumption that $f \cdot g$ is not primitive.

For (2), it obviously suffices to show that a polynomial is irreducible over $\mathbb{Z}$ is also irreducible over $\mathbb{Q}$. Hence, suppose that $f = g \cdot h$ with polynomials $g, h \in \mathbb{Q}[x] \setminus \mathbb{Q}$. We can now choose integers $a, b \in \mathbb{Z}$ such that $a \cdot g$ and $b \cdot h$ are both primitive polynomials in $\mathbb{Z}[x]$. Then, part (1) implies that the product $(ab) \cdot f = (ag) \cdot (bh)$ is primitive as well. Thus, we obtain $a \cdot b = \pm 1$, which shows that $g, h \in \mathbb{Z}[x]$. □

[1] Notice that the content is unique up to a factor in R^*.

[2] Addition and multiplication in F is defined as for rational numbers, that is, $\frac{a}{b} + \frac{a'}{b'} := \frac{ab'+a'b}{bb'}$ and $\frac{a}{b} \cdot \frac{a'}{b'} = \frac{aa'}{bb'}$. Two elements $\frac{a}{b}$ and $\frac{a'}{b'}$ are equal if and only if $ab' = a'b$.

We can now prove that $R[x]$ is factorial if R is factorial:

Theorem 3.1.15. *If R is factorial, then $R[x]$ is factorial as well.*

Proof. Let F be the quotient ring of R. For simplicity, we again assume that $R = \mathbb{Z}$ and $F = \mathbb{Q}$. Since $\mathbb{Q}[x]$ is a principal domain, $\mathbb{Q}[x]$ is also factorial. Namely, each f can be written as $q_1 \cdots q_s$ with irreducible $q_i \in \mathbb{Q}[x]$ according to Theorem 3.1.10, and the q_i's are also prime according to Exercise 3.1.12. Now, let $f \in \mathbb{Z}[x]$ be a polynomial. We aim to show that there exists a factorization of f into prime factors $q_i \in \mathbb{Z}[x]$. We may assume that f is primitive as, otherwise, there exists a common divisor $r \in \mathbb{Z}$ of all coefficients such that f/r is primitive, and since $\mathbb{Z}$ is factorial, r can be written as a product of primes. Now, since $\mathbb{Q}[x]$ is factorial, there exists a factorization

$$f(x) = q_1 \cdots q_s$$

of f into prime factors $q_i \in \mathbb{Q}[x]$. We can now choose $r_1, \ldots, r_s \in \mathbb{Z}$ such that $r_i \cdot q_i \in \mathbb{Z}[x]$ is primitive. This implies that

$$(r_1 \cdots r_s) \cdot f = (r_1 q_1) \cdots (r_s q_s)$$

is primitive as well, and thus $r_1 \cdots r_s = \pm 1$. Hence, we have $r_i = \pm 1$ for all i. Since each q_i is irreducible in $\mathbb{Q}[x]$, it is also irreducible in $\mathbb{Z}[x]$. It remains to show that the above factorization of f into irreducible polynomials is unique. For this, suppose that we have $f(x) = \bar{q}_1 \cdots \bar{q}_{s'}$ with irreducible polynomials $\bar{q}_i \in \mathbb{Z}[x]$. Since the factorization into irreducible polynomials in $\mathbb{Q}[x]$ is (unique up to ordering and a unit in $\mathbb{Q}$), we have $s = s'$ and we may assume that $\frac{a_i}{b_i} \cdot \bar{q}_i = q_i$ with integers $a_i, b_i \in \mathbb{Z} \setminus \{0\}$. Since $\bar{q}_i$ is irreducible in $\mathbb{Z}[x]$, it must be primitive as well. Since q_i is also primitive, we thus conclude from $a_i \bar{q}_i = b_i q_i$ that $a_i = b_i$. This shows that the factorization is unique. We conclude that q_i is prime as, in every integral domain that yields a unique factorization into irreducibles, an element is prime if and only if it is irreducible. □

From the above theorem, we conclude that $\mathbb{Z}[x_1, \ldots, x_n]$ is a factorial ring. The same holds true for $F[x_1, \ldots, x_n]$, where F is an arbitrary field.

Definition 3.1.16 (GCD and LCM)**.** *Let R be an integral domain and $a, b, c \in R$. Then, c is a* greatest common divisor *of a and b ($c = \gcd(a, b)$ for short) if*

- *c divides a and b, and*
- *for all $d \in R$, it holds that if d divides a and b, then d divides c.*

We further define $c = \operatorname{lcm}(a, b)$ a least common multiple *of a and b if*

- *a and b divide c, and*
- *for all $d \in R$, it holds that if a and b divide d, then c divides d.*

Notice that we do not use the article "the" in definitions of a greatest common divisor and a least common multiple. The reason is that, in general, $\gcd(a, b)$ and $\operatorname{lcm}(a, b)$ are not uniquely defined. For instance, 2 as well as -2 are greatest common divisors of the two integers 4 and 14. Also, for $a = x^2 - 1 \in \mathbb{Q}[x]$ and $b = x^2 + 2x - 1$, both of the two polynomials $(x + 1) \cdot (x^2 - 1) = x^3 + x^2 - x - 1$ and $\frac{1}{2} \cdot (x^3 + x^2 - x - 1)$ are least common multiples of a and b. Hence, it makes sense to normalize the polynomials, which allows us to speak about "the" greatest common divisor and the least common multiple.

Definition 3.1.17 (Normal Form)**.** *Let R be an integral domain, then we call a function* normal : $R \mapsto R$ *a* normal form *if $\text{normal}(a) \sim a$ for all $a \in R$ and the following two properties are fulfilled:*

- $\text{normal}(0) = 0$,
- $a \sim b \Rightarrow \text{normal}(a) = \text{normal}(b)$, *and*
- $\text{normal}(a \cdot b) = \text{normal}(a) \cdot \text{normal}(b)$.

We call the unique $e \in R^*$ with $e \cdot \text{normal}(a) = a$ the leading coefficient *of a ($\text{LC}(a) = e$ for short). For $a = 0$, we define $\text{LC}(0) = 1$.*

In the special case, where $R = F[x]$ with F a field, it is easy to see that $\text{normal}(f) := \text{LC}(f)^{-1} \cdot f$ is a normal form.

3.2 The Extended Euclidean Algorithm

In this section, we study the *extended Euclidean algorithm* (EEA for short) to compute the gcd of two polynomials $f, g \in F[x]$, where F a field. We further show that the algorithm has a polynomial bit complexity when applied to compute the gcd of two polynomials f, g with integer coefficients. The proof of the latter fact is non-trivial at all (even though it seems like this) and requires a deeper understanding of the method.

Before we formulate the algorithm in its general form, we first review the Euclidean algorithm for computing the gcd of two integers $a, b \in \mathbb{Z}$. For this, we consider a simple example: In order to compute the $c := \gcd(a, b)$ of $a := 130$ and $b := 56$, we first divide $r_0 := a = 130$ by $r_1 := b = 56$:

$$130 = 2 \cdot 56 + 18 \text{ or } 18 = 1 \cdot 130 - 2 \cdot 56.$$

This yields the remainder $r_2 = 18$. Since c divides r_0 and r_1, it must also divide r_2. Vice versa, each divisor of r_1 and r_2 divides r_0, and thus it follows that $c = \gcd(r_0, r_1) = \gcd(r_1, r_2)$. This shows that we recursively continue with r_1 and r_2 (instead of r_0 and r_1) in this way in order to compute c. Dividing r_1 by r_2 yields a remainder $r_3 := 2$:

$$56 = 3 \cdot 18 + 2 \text{ or } 2 = 56 - 3 \cdot 18 = 56 - 3 \cdot (130 - 2 \cdot 56) = (-3) \cdot 130 + 7 \cdot 56.$$

Finally, we divide r_2 by r_3, which yields the remainder $r_4 = 0$:

$$18 = 9 \cdot 2 + 0 \text{ or } 0 = 18 - 9 \cdot 2 = (130 - 2 \cdot 56) - 9 \cdot ((-3) \cdot 130 + 7 \cdot 56) = 28 \cdot 130 - 65 \cdot 56.$$

We conclude that $\gcd(130, 56) = 2$. Further notice that, in each step of the recursion, we expressed the remainder r_i as a linear combination of a and b. In particular, this holds for $r_3 := \gcd(130, 56)$:

$$2 = (-3) \cdot 130 + 7 \cdot 56.$$

Exercise 3.2.1. *Show that, for two integers $a, b \in \mathbb{Z}$ of length at most L, the Euclidean algorithm uses $O(L)$ iterations. Further show that this bound is optimal, and derive a bound on the bit complexity of the Euclidean algorithm!*

Hint: Show first that $r_{i-1} > 2 \cdot r_{i+1}$, where r_i is the remainder obtained in the i-th iteration of the algorithm.

Algorithm 11: Extended Euclidean Algorithm

Input : Polynomials $f, g \in F[x]$, with $\deg f \geq \deg g$ and F a field.

Output: An integer $\ell \in \mathbb{N}$, and $\rho_i, r_i, s_i, t_i \in R$ such that $r_i = s_i \cdot a + t_i \cdot b$ for all $i \in \{0, 1, \ldots, \ell + 1\}$ and $r_\ell = \gcd(a, b)$.

$\rho_0 := \mathrm{LC}(f)$, $r_0 := \mathrm{normal}(a)$, $s_0 := \rho_0^{-1}$, and $t_0 := 0$.
$\rho_1 := \mathrm{LC}(g)$, $r_1 := \mathrm{normal}(b)$, $s_1 := 0$, and $t_1 := \rho_1^{-1}$.
$i := 1$
while $r_i \neq 0$ **do**
    Define
    $q_i := \mathrm{quo}(r_{i-1}, r_i)$
    $\rho_{i+1} := \mathrm{LC}(\mathrm{rem}(r_{i-1}, r_i))$
    $r_{i+1} := \mathrm{normal}(\mathrm{rem}(r_{i-1}, r_i))$
    $s_{i+1} := (s_{i-1} - q_i \cdot s_i) \cdot \rho_{i+1}^{-1}$
    $t_{i+1} := (t_{i-1} - q_i \cdot t_i) \cdot \rho_{i+1}^{-1}$
    $i := i + 1$
$\ell := i - 1$
return $(\ell, (\rho_i, s_i, t_i, r_i)_{i=0,\ldots,\ell+1})$

We can now formulate the EEA in its general form; see Algorithm 11. The steps are essentially the same as in the integer case, however, after each iteration, the computed remainders are normalized. Termination of the algorithm follows directly from the fact that $d(r_i)$ is strictly decreasing. Hence, we are left to prove that $s_i \cdot f + t_i \cdot g = r_i$ for all i, in particular,

$$s_\ell \cdot f + t_\ell \cdot g = r_\ell = \gcd(f, g).$$

The elements s_ℓ and t_ℓ are called the *Bézout coefficients* of a and b. Before we prove correctness of the algorithm, we first give an example to illustrate the approach.

Example. Let $R = \mathbb{Q}[x]$, and $f = 12x^3 - 28x^2 + 20x - 4$, $g = -12x^2 + 10x - 2$ polynomials in $\mathbb{Q}[x]$. Algorithm 11 recursively computes $h := \gcd(f, g)$:

i	q_i	ρ_i	r_i	s_i	t_i
0		12	$x^3 - \frac{7}{3}x^2 + \frac{5}{3}x - \frac{1}{3}$	$\frac{1}{12}$	0
1	$x - \frac{3}{2}$	-12	$x^2 - \frac{5}{6}x + \frac{1}{6}$	0	$-\frac{1}{12}$
2	$x - \frac{1}{2}$	$\frac{1}{4}$	$x - \frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}x - \frac{1}{2}$
3		1	0	$-\frac{1}{3}x + \frac{1}{6}$	2

Hence, from Row 2, we conclude that

$$\gcd(f, g) = x - \frac{1}{3} = \frac{1}{3} \cdot f + \left(\frac{1}{3} \cdot x - \frac{1}{2}\right) \cdot g.$$

Exercise 3.2.2. *Trace the Extended Euclidean Algorithm to compute the GCD of*

$f = 77400x^7 + 29655x^6 - 153746x^5 + 37585x^4 + 91875x^3 - 130916x^2 - 21076x + 51183$ *and*
$g = -5040x^6 + 27906x^5 + 44950x^4 - 66745x^3 + 69052x^2 + 111509x - 98208,$

considered as polynomials in $\mathbb{Q}[x]$ with rational coefficients. What do you observe?

Exercise 3.2.3 (Sturm Sequences)**.** *A* Sturm Sequence $\mathcal{S}$ *is a sequence of polynomials $f_0, \ldots, f_\ell \in \mathbb{R}[x]$ such that the following conditions are fulfilled:*

- $\deg f_0 > \deg f_1 > \cdots > \deg f_\ell = 0$,
- f_0 *has no multiple roots,*
- *If* $f_0(\xi) = 0$, *then* $\operatorname{sign}(f_1(\xi)) = \operatorname{sign}(f_0'(\xi))$, *and*
- *if* $f_i(\xi) = 0$ *for* $i \in \{1, \ldots, \ell - 1\}$, *then* $\operatorname{sign}(f_{i-1}(\xi)) = -\operatorname{sign}(f_{i+1}(\xi))$

For an arbitrary $\xi \in \mathbb{R}$, we define

$$\operatorname{var}(\mathcal{S}, \xi) = \#\{i : \exists j > i \text{ with } f_{i+1}(\xi) = \cdots = f_{j-1}(\xi) = 0 \text{ and } f_i(\xi) \cdot f_j(\xi) < 0\}$$

as the number of sign changes (ignoring zeroes) in the sequence $f_0(\xi), \ldots, f_\ell(\xi)$.

(a) Show that, for arbitrary $a, b \in \mathbb{R}$ with $a < b$, it holds that*

$$\#\{\text{roots of } f_0 \text{ in } (a, b]\} = \operatorname{var}(\mathcal{S}, a) - \operatorname{var}(\mathcal{S}, b).$$

(b) Let $f \in \mathbb{R}[x]$ be a polynomial, and let $r_0, \ldots, r_{\ell+1} \in \mathbb{R}[x]$ and $\rho_0, \ldots, \rho_{\ell+1} \in \mathbb{R}$ be as computed by the EEA with input f and $g = f'$. We recursively define $\sigma_0 := \operatorname{sign}(\rho_0)$, $\sigma_1 := \operatorname{sign}(\rho_1)$, and $\sigma_i := -\operatorname{sign}(\sigma_{i-1} \cdot \rho_{i+1})$ for $i > 1$. Show that the sequence $\mathcal{S} := \{\bar{r}_i := \sigma_i \cdot r_i\}_{i=0,\ldots,\ell}$, is a Sturm sequence if f has no multiple roots.[3]

(c) Derive an algorithm to compute all real roots of a polynomial $f \in \mathbb{Z}[x]$ within a given interval $[a, b]$!

Hint: For (a), use the fact that the number $\operatorname{var}(\mathcal{S}, \xi)$ can only change at a root ξ of one of the polynomials f_i. Further show that each root of f_0 is not a root of any other polynomial f_i. Finally, show that each such root of f_0 yields a change of $\operatorname{var}(\mathcal{S}, \xi)$, whereas each root of f_i, with $i \neq 0$, does not yield a change.

Lemma 3.2.4. *Let f and g be polynomials in $F[x]$ and let ρ_i, s_i, t_i as computed in the EEA with input f, g, then*

(a) $\gcd(f, g) = \gcd(r_i, r_{i+1}) = r_\ell$

(b) $s_i \cdot f + t_i \cdot g = r_i$ *for all* $i = 0, \ldots, \ell + 1$. *In particular,* $s_\ell \cdot f + t_\ell \cdot g = r_\ell = \gcd(a, b)$.

(c) $\gcd(s_i, t_i) = 1$ *for all* $i = 0, \ldots, \ell$

(d) $\deg s_i = \sum_{2 \leq j < i} \deg q_j = \deg g - \deg r_{i-1}$ *for all i with* $2 \leq i \leq \ell + 1$

(e) $\deg t_i = \sum_{1 \leq j < i} \deg q_j = \deg f - \deg r_{i-1}$ *for all i with* $1 \leq i \leq \ell + 1$.

[3] One can even show that, also for polynomials f with multiple roots, the sequence $\mathcal{S}$ has the property that $\operatorname{var}(\mathcal{S}, a) - \operatorname{var}(\mathcal{S}, b)$ equals the number of distinct roots of f in $(a, b]$.

Proof. From the definition of the ρ_i, r_i, and q_i, we obtain for $i = 1, \dots, \ell$:

$$\begin{aligned} \rho_i \cdot r_{i+1} &= r_{i-1} - q_i \cdot r_i \\ \rho_{i+1} \cdot s_{i+1} &= s_{i-1} - q_i \cdot s_i \\ \rho_{i+1} \cdot t_{i+1} &= t_{i-1} - q_i \cdot t_i. \end{aligned}$$

Hence, with $Q_i := \begin{pmatrix} 0 & 1 \\ \rho_{i+1}^{-1} & -q_i \cdot \rho_{i+1}^{-1} \end{pmatrix}$, we have

$$Q_i \cdot \begin{pmatrix} r_{i-1} \\ r_i \end{pmatrix} = \begin{pmatrix} r_i \\ (r_{i-1} - q_i \cdot r_i) \cdot \rho_{i+1}^{-1} \end{pmatrix} = \begin{pmatrix} r_i \\ r_{i+1} \end{pmatrix}$$

for $i = 1, \dots, \ell$. Hence, with $Q_0 := \begin{pmatrix} s_0 & t_0 \\ s_1 & t_1 \end{pmatrix}$ and $R_i := Q_i \cdots Q_1 Q_0$, we conclude that

$$R_i \cdot \begin{pmatrix} f \\ g \end{pmatrix} = Q_i \cdots Q_1 \cdot \begin{pmatrix} f \\ g \end{pmatrix} = Q_i \cdots Q_1 \cdot \begin{pmatrix} \rho_0^{-1} & 0 \\ 0 & \rho_1^{-1} \end{pmatrix} \cdot \begin{pmatrix} f \\ g \end{pmatrix} = Q_i \cdots Q_1 \cdot \begin{pmatrix} r_0 \\ r_1 \end{pmatrix} = \begin{pmatrix} r_i \\ r_{i+1} \end{pmatrix}.$$

Furthermore, we have

$$Q_i \cdot \begin{pmatrix} s_{i-1} & t_{i-1} \\ s_i & t_i \end{pmatrix} = \begin{pmatrix} s_i & t_i \\ s_{i+1} & t_{i+1} \end{pmatrix}, \text{ and thus } R_i = \begin{pmatrix} s_i & t_i \\ s_{i+1} & t_{i+1} \end{pmatrix}.$$

We are now ready to prove (a)-(e):

We have

$$\begin{pmatrix} r_\ell \\ 0 \end{pmatrix} = \begin{pmatrix} r_\ell \\ r_{\ell+1} \end{pmatrix} = Q_\ell \cdots Q_0 \cdot \begin{pmatrix} f \\ g \end{pmatrix} = Q_\ell \cdots Q_{i+1} \cdot \begin{pmatrix} r_i \\ r_{i+1} \end{pmatrix},$$

from which we conlude that r_ℓ can be written as a linear combination of r_i and r_{i+1}. It follows that $\gcd(r_i, r_{i+1})$ divides r_ℓ for all i. In addition, since Q_i is invertible and $Q_i^{-1} = \begin{pmatrix} q_i & \rho_{i+1} \\ 1 & 0 \end{pmatrix}$, we have

$$\begin{pmatrix} r_i \\ r_{i+1} \end{pmatrix} = Q_{i+1}^{-1} \cdots Q_\ell^{-1} \cdot \begin{pmatrix} r_\ell \\ 0 \end{pmatrix},$$

and thus r_ℓ divides r_i as well as r_{i+1}. Hence, (a) follows.

Part (b) follows directly from the fact that

$$Q_i \cdots Q_0 = \begin{pmatrix} s_i & t_i \\ s_{i+1} & t_{i+1} \end{pmatrix} \text{ and } Q_i \cdots Q_0 \cdot \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} r_i \\ r_{i+1} \end{pmatrix}.$$

For (c), we use that

$$s_i \cdot t_{i+1} + s_{i+1} \cdot t_i = \det \begin{pmatrix} s_i & t_i \\ s_{i+1} & t_{i+1} \end{pmatrix} = \det Q_i \cdots \det Q_1 \cdot \det \begin{pmatrix} s_0 & t_0 \\ s_1 & t_1 \end{pmatrix} = (-1)^i \cdot (\rho_0 \cdots \rho_{i+1})^{-1},$$

which implies that $\gcd(s_i, t_i) = 1$.

For (d), we first show by induction that $\deg s_{i-1} < \deg s_i$ for all i with $2 \le i \le \ell + 1$. For $i = 2$, we have

$$s_2 = \rho_2^{-1} \cdot (s_0 - q_1 \cdot s_1) = (\rho_0^{-1} - q_1 \cdot 0) \cdot \rho_2^{-1} = \rho_2^{-1} \cdot \rho_0^{-1},$$

and thus $\deg s_1 = -\infty < 0 = \deg s_2$. Now, suppose that for i with $2 \leq i \leq i_0$, the claim is already proven. Then, we have

$$\deg s_{i_0-1} < \deg s_{i_0} < \deg r_{i_0-1} - \deg r_{i_0} + \deg s_{i_0} = \deg q_{i_0} + \deg s_{i_0} = \deg q_{i_0} s_{i_0},$$

where we used that $q_{i_0} = \operatorname{quo}(r_{i_0-1}, r_{i_0})$, and thus $\deg r_{i_0-1} - \deg r_{i_0} = \deg q_{i_0}$. From the above inequality, we conclude that

$$\deg s_{i_0+1} = \deg(s_{i_0-1} - q_{i_0} \cdot s_{i_0}) = \deg(q_{i_0} \cdot s_{i_0}) = \deg q_{i_0} + \deg s_{i_0} > \deg s_{i_0},$$

and

$$\deg s_{i_0+1} = \deg q_{i_0} + \deg s_{i_0} = \sum_{2 \leq j < i_0} \deg q_j + \deg q_{i_0} = \sum_{1 \leq j \leq i} \deg q_j.$$

The proof for (e) is completely analogous to that of (d). □

The algorithm is called *extended* Euclidean Algorithm as it does not only return the gcd of a and b, but also its Bézout representation $s_\ell \cdot f + t_\ell \cdot g = \gcd(f, g)$. Obviously, the algorithm uses at most $m := \deg g$ many iterations, and each iteration uses $\tilde{O}(n)$ arithmetic operations in F, where $n = \deg f$. Hence, the total arithmetic complexity is bounded by $\tilde{O}(nm)$. We will study a variant of the algorithm (called *Half-GCD*) that uses only $\tilde{O}(n)$ arithmetic operations. However, this does not directly imply that the bit complexity of the algorithm is also polynomial when applied to polynomials $f, g \in \mathbb{Q}[x]$ with rational coefficients. Namely, it is non-trivial to prove that the bitsizes of the intermediate results do not grow exponentially in n. For this, a deeper understanding of the algorithm is necessary. Before we give details, we give some applications of the EEA.

Definition 3.2.5 (and Lemma)**.** *Let R be a factorial ring and $f = a_0 + \cdots + a_n x^n \in R[x]$. We call f* square-free *if there exists no polynomial $g \in R[x] \setminus R$ such that g^2 divides f. There exists a unique factorization*

$$f = \operatorname{cont}(f) \cdot \prod_{i=1}^{k} g_i^i \tag{3.1}$$

of f into square-free and primitive polynomials $g_i \in R[x]$ that are pairwise coprime. We call a factorization as above the square-free factorization *of f. The polynomial $f^* := \prod_{i=1}^{k} g_i = f / \gcd(f, f')$ is called the* square-free part *of f.*

Proof. Since R is factorial, $R[x]$ is factorial as well. Hence, there exists a unique factorization

$$f = \operatorname{cont}(f) \cdot \prod_{j=1}^{k'} f_j^{d_j}$$

of f into irreducible, primitive, and distinct polynomials f_j. Then, $\operatorname{cont}(f) \cdot \prod_{i=1}^{k} g_i^i$, with $g_i := \prod_{j:d_j=i} f_j$, is the unique square-free factorization of f. In addition, we have

$$f' = \operatorname{cont}(f) \cdot \sum_{j=1}^{k'} \frac{d_j \cdot f}{f_j} \cdot f_j',$$

Algorithm 12: Yun's Square-Free Factorization Algorithm

Input : $f \in R[x]$ primitive, with R a factorial ring.
Output: A square-free factorization as in (3.1).

$u := \gcd(f, f')$, $v_1 := \frac{f}{u}$, $w_1 := \frac{f'}{u}$, $i = 1$
while $v_i \neq 1$ **do**
    Recursively define
    $g_i := \gcd(v_i, w_i - v_i')$
    $v_{i+1} := \frac{v_i}{g_i}$
    $w_{i+1} := \frac{w_i - v_i'}{g_i}$
    $i = i + 1$
$m := i - 1$
return $g_1 \dots g_m$

and thus $f_j^{d_j - 1}$ divides f' for all j. Suppose that $f_i^{d_i}$ divides f for some i. Then, since $f_i^{d_i}$ divides $\frac{d_j \cdot f}{f_j}$ for all $j \neq i$, it must also divide $\frac{d_i \cdot f}{f_i}$, which is impossible. Hence, it follows that

$$f' = h \cdot \prod_{j=1}^{k'} f_j^{d_j - 1},$$

with some polynomial $h \in R[x]$ that is not divisible by any f_j. It thus follows that $\gcd(f, f') = \operatorname{cont}(f) \cdot \prod_{j=1}^{k'} f_j^{d_j - 1}$ and $f^* = f / \gcd(f, f')$. □

Exercise 3.2.6 (Yun's Square-Free Factorization Algorithm)**.** *Show that Yun's algorithm computes a square-free factorization of a polynomial $f \in R[x]$!*

Exercise 3.2.7. *Let $f \in F[x]$ be a polynomial, with F a field, and let ℓ be defined as in the Extended Euclidean Algorithm when applied to f and f'; that is,*

$$s_\ell \cdot f + t_\ell \cdot f' = \gcd(f, f').$$

Show that the polynomial $t_{\ell+1}$ as computed in the next iteration of the algorithm equals the square-free part f^* of f.

3.3 The Half-GCD Algorithm (under construction)

The main goal of this section is to prove the following theorem:

Theorem 3.3.1. *Let f and g be polynomials in $F[x]$ of degree m and n, respectively, where $m \geq n$. Using $\tilde{O}(m)$ arithmetic operations in F, we can compute*

- *the greatest common divisor $r_\ell := \gcd(f, g)$ of f and g,*
- *the polynomials s_ℓ and t_ℓ as computed in the EEA such that $s_\ell \cdot f + t_\ell + g = r_\ell$, and*
- *the polynomials s_i, t_i, and r_i for an arbitrary index $i \in \{0, \dots, \ell + 1\}$.*

3.4 The Resultant

In what follows, we always assume that R is a factorial ring. Given two polynomials $f = a_0 + \cdots + a_m \cdot x^m$ and $g = b_0 + \cdots + b_n \cdot x^n$ in $R[x]$, we can always write

$$u \cdot f + v \cdot g = 0,$$

with $u := \frac{g}{\gcd(f,g)}$ and $v := -\frac{f}{\gcd(f,g)}$. If the greatest common divisor of f and g is non-trivial (i.e. $\gcd(f,g) \in R[x] \setminus R$), then we have $\deg u < n$ and $\deg v < m$. Vice versa, if $0 = u' \cdot f + v \cdot g$ for polynomials $u', v' \in R[x]$ of degrees less than n and m, respectively, then f and g must share a non-trivial common factor. This gives a necessary and sufficient condition for $\gcd(f,g)$ to be non-trivial:

Lemma 3.4.1. *Let $f, g \in R[x]$ be two polynomials of degrees m and n, respectively. Then, f and g share a non-trivial divisor if and only if there exists polynomials $u, v \in R[x]$ with*

$$u \cdot f + v \cdot g = 0 \text{ and } \deg u < n,\ \deg v < m. \tag{3.2}$$

The above lemma now allows us to reformulate the problem of deciding whether $\gcd(f,g)$ is non-trivial in terms of linear algebra. Namely, considering polynomials $u = u_0 + \cdots + u_{n-1} \cdot x^{n-1}$ and $v = v_0 + \cdots + v_{m-1} \cdot x^{m-1}$ of degrees less than n and m, respectively, and with indeterminate coefficients. Then, the condition (3.2) is equivalent to

$$\begin{pmatrix} u_{n-1} & \cdots & u_0 & v_{m-1} & \cdots & v_0 \end{pmatrix} \cdot \underbrace{\begin{pmatrix} a_m & \cdots & a_0 & & & \\ & a_m & \cdots & a_0 & & \\ & & \ddots & & \ddots & \\ & & & a_m & \cdots & a_0 \\ b_n & \cdots & b_0 & & & \\ & b_n & \cdots & b_0 & & \\ & & \ddots & & \ddots & \\ & & & b_n & \cdots & b_0 \end{pmatrix}}_{=:\mathrm{Syl}(f,g)} = 0$$

Here, $\mathrm{Syl}(f,g)$ is an $(m+n) \times (m+n)$-matrix, which is called the *Sylvester Matrix* of f and g. Notice that the above equality can only be fulfilled if the rows of $\mathrm{Syl}(f,g)$ are linear dependent, hence we must have $\det \mathrm{Syl}(f,g) = 0$. Vice versa, if the determinant of the Sylvester Matrix vanishes, then there exists a coefficient vector $(u_{n-1}, \ldots, u_0, v_{m-1}, \ldots, v_0)$ such that the above equality holds. We call $\mathrm{Res}(f,g) := \det \mathrm{Syl}(f,g)$ the *Resultant* of f and g. Notice that the definition of Syl as well as Res crucially depends on the degrees of f and g.

Theorem 3.4.2. *Let $f, g \in R[x]$ be polynomials of degrees m and n, respectively. It holds:*

(a) $\gcd(f,g) \in R[x] \setminus R \Leftrightarrow \mathrm{Res}(f,g) = 0$

(b) There exist polynomials $u, v \in R[x]$ of degrees less than n and m, respectively, such that

$$\mathrm{Res}(f,g) = u \cdot f + v \cdot g.$$

(c) $\mathrm{Res}(f,c) = c^m$ *for an arbitrary constant $c \in R$*

(d) $\operatorname{Res}(f,g) = (-1)^{mn} \cdot \operatorname{Res}(g,f)$

(e) For R a field, $m \geq n$, and $r(x) := \operatorname{rem}(f,g)$, we have

$$\operatorname{Res}(f,g) = (-1)^{mn} \cdot \operatorname{LC}(g)^{m-\deg r} \cdot \operatorname{Res}(g,r).$$

Proof. Part (a) follows from our considerations above. For (b), we distinguish the two cases $\operatorname{Res}(f,g) = 0$ and $\operatorname{Res}(f,g) \neq 0$. In the first case, the claim follows directly from Lemma 3.4.1. For $\operatorname{Res}(f,g) \neq 0$, consider the matrix

$$\operatorname{Syl}^*(f,g) := \begin{pmatrix} a_m & \cdots & a_0 & & & x^{n-1} \cdot f \\ & a_m & \cdots & a_0 & & x^{n-2} \cdot f \\ & & \ddots & & \ddots & \vdots \\ & & & a_m & \cdots & x^0 \cdot f \\ b_n & \cdots & b_0 & & & x^{m-1} \cdot g \\ & b_n & \cdots & b_0 & & x^{m-2} \cdot g \\ & & \ddots & & \ddots & \vdots \\ & & & b_n & \cdots & x^0 \cdot g \end{pmatrix}$$

obtained by replacing the last column of $\operatorname{Syl}(f,g)$ by $(x^{n-1} \cdot f, \ldots, x^0 \cdot f, x^{m-1} \cdot g, \ldots, x^0 \cdot g)^t$. Using linearity of the determinant, we obtain

$$\det \operatorname{Syl}^*(f,g) = \operatorname{Res}(f,g) + \sum_{j=1}^{m+n-1} \det(S_j) \cdot x^j,$$

with

$$S_j := \begin{pmatrix} a_m & \cdots & a_0 & & & a_{j-(n-1)} \\ & a_m & \cdots & a_0 & & a_{j-(n-2)} \\ & & \ddots & & \ddots & \vdots \\ & & & a_m & \cdots & a_j \\ b_n & \cdots & b_0 & & & b_{j-(m-1)} \\ & b_n & \cdots & b_0 & & b_{j-(m-2)} \\ & & \ddots & & \ddots & \vdots \\ & & & b_n & \cdots & b_j \end{pmatrix},$$

where we define $a_i = b_i = 0$ for $i < 0$. Now, notice that the $(m+n-j)$-th column of D_j coincides with the last column of D_j, and thus we have $\operatorname{Res}(f,g) = \det \operatorname{Syl}^*(f,g)$. Hence, using Laplace expansion for the computation of $\det \operatorname{Syl}^*(f,g)$ yields that $\operatorname{Res}(f,g) = \det \operatorname{Syl}^*(f,g) = u \cdot f + v \cdot g$ with polynomials u and v of degrees less than n and m, respectively.

Parts (c) and (d) follow immediately from the definition of Res and the fact that the determinant switches sign if two rows are switched. It remains to prove (e): For this, let

$q = q_0 + \cdots q_{m-n} \cdot x^{m-n}$ with $f = q \cdot g + r$. We then write the Sylvester Matrix $\mathrm{Syl}(f, g)$ as

$$\mathrm{Syl}(g, f) = \begin{pmatrix} b_n & \cdots & b_0 & & & \\ & b_n & \cdots & b_0 & & \\ & & \ddots & & \ddots & \\ & & & b_n & \cdots & b_0 \\ a_m & \cdots & a_0 & & & \\ & a_m & \cdots & a_0 & & \\ & & \ddots & & \ddots & \\ & & & a_m & \cdots & a_0 \end{pmatrix} = \begin{pmatrix} B \\ A \end{pmatrix}$$

with matrices A and B of size $n \times (m+n)$ and $m \times (m+n)$, respectively. Our goal is to transform $\mathrm{Syl}(g, f)$ via suitable row operations into an $(m+n) \times (m+n)$ matrix

$$T = \begin{pmatrix} b_n & \cdots & b_k & \cdots & b_0 & & \\ & b_n & \cdots & b_k & \cdots & b_0 & \\ & & \ddots & & \ddots & & \ddots \\ & & & b_n & \cdots & b_k & \cdots & b_0 \\ & & r_k & \cdots & r_0 & & \\ & & & r_k & \cdots & r_0 & \\ & & & & \ddots & & \ddots \\ & & & & & r_k & \cdots & r_0 \end{pmatrix} = \begin{pmatrix} B \\ 0 \; R \end{pmatrix},$$

where the rows of R correspond to the coefficients of the remainder $r(x) = r_0 + \cdots + r_k \cdot x^k$, which has degree $k < n$. This can be achieved by subtracting $q_{n+m+1-i}$ - times the i-th row of $\mathrm{Syl}(f, g)$ from its $(m+i)$-th row for all $i = 1, \ldots, m$. Here, we use that

$$\begin{pmatrix} q_{m-n} & \cdots & q_0 & 0 & \cdots & 0 \end{pmatrix} \cdot \begin{pmatrix} b_n & \cdots & b_0 & & & \\ & b_n & \cdots & b_0 & & \\ & & \ddots & & \ddots & \\ & & & b_n & \cdots & b_0 \end{pmatrix} = \begin{pmatrix} a_m \\ \vdots \\ a_{k+1} \\ a_k - r_k \\ \vdots \\ a_0 - r_0 \end{pmatrix}$$

Since the above row operations do not change the value of the determinant of $\mathrm{Syl}(g, f)$, it follows that

$$\mathrm{Res}(f, g) = (-1)^{mn} \cdot \det \mathrm{Syl}(g, f) = \det T = (-1)^{mn} \cdot b_n^{m-k} \cdot \mathrm{Res}(g, r).$$

□

Exercise 3.4.3 (Computing Resultants via the Euclidean Algorithm)**.** *Use the Euclidean Algorithm and Theorem 3.4.2 (e) to compute the resultant of the polynomials*

$$f := x^4 + 2 \cdot x^3 - 3 \cdot x + 1 \in \mathbb{Z}[x] \text{ and } g := x^2 + x + 1 \in \mathbb{Z}[x].$$

Exercise 3.4.4 (Specialization Property of Resultants)**.** *An important property of the resultant is that it is compatible with respect to specialization. More specifically, let* $\phi : R \mapsto R'$ *be a ring homomorphism*[4] *between factorial rings* R *and* R'*, and* $\bar{\phi} : R[x] \mapsto R'[x]$ *its canonical extension to the corresponding polynomial rings (i.e.* $\bar{\phi}(a_0 + \cdots + a_m \cdot x^m) = \phi(a_0) + \cdots + \phi(a_m) \cdot x^m$*). Suppose that* $\deg \bar{\phi}(f) = \deg f$ *and* $\deg \bar{\phi}(g) = \deg g$ *for polynomials* $f, g \in R[x]$*, then it holds that*

$$\phi(\mathrm{Res}(f, g)) = \mathrm{Res}(\bar{\phi}(f), \bar{\phi}(g)).$$

Give an example of two polynomials $f, g \in \mathbb{Z}[x]$ *and a prime* p *such that* $\mathrm{Res}(f, g) \neq \mathrm{Res}(\bar{f}, \bar{g})$*, where we define* $\bar{f}, \bar{g} \in \mathbb{Z}/_p[x]$ *as the canonical images of* f *and* g *in* $\mathbb{Z}/_p[x]$*.*

Exercise 3.4.5. *Let* $f := y^2 + 2 \cdot x^2 + x \cdot y - 4 \cdot x - 2 \cdot y + 2$ *and* $g := 3 \cdot x^2 + y^2 - 4 \cdot x$ *be two polynomials in* $\mathbb{Z}[x]$*. Show that* $f = g = 0$ *has exactly one real solution and determine this solution.*

Hint: *Consider* f *and* g *as polynomials in* $R[y]$*, with* $R = \mathbb{Z}[x]$*. Then, use Exercise 3.4.4 with the ring homomorphism* $\phi : R \mapsto \mathbb{R}$ *that maps an* $h \in \mathbb{Z}[x]$ *to its value* $h(x_0)$ *at some fixed point* $x_0 \in \mathbb{R}$*. You should also use the fact that* $f(x_0, y)$ *and* $g(x_0, y)$ *have a common (complex) root if and only if their greatest common divisor is non-trivial.*

Exercise 3.4.6 (The Field of Algebraic Numbers)**.** *We aim to show that the set of algebraic numbers*

$$\bar{\mathbb{Q}} := \{\alpha \in \mathbb{C} : \textit{there exists an } f \in \mathbb{Q}[x] \textit{ such that } f(\alpha) = 0\} \subset \mathbb{C}$$

over $\mathbb{Q}$ *is a field.*

(a) Let $\alpha, \beta \in \mathbb{C}$ *and* f *and* g *be polynomials in* $\mathbb{Q}[x]$ *such that* $f(\alpha) = 0$ *and* $g(\beta) = 0$*. Show how to construct polynomials* $h \in \mathbb{Q}[x]$ *that satisfy*

- $h(-\alpha) = 0$*, or*
- $h(\alpha + \beta) = 0$*, or*
- $h(\alpha \cdot \beta) = 0$*, or*
- $h(1/\alpha) = 0$*, or*
- $h(\sqrt[k]{\alpha}) = 0$ *for some* $k \in \mathbb{N}_{\geq 2}$*, respectively.*

Hint: *Use resultants to show that the coordinates of any solution of a bivariate system* $F(x, y) = G(x, y) = 0$*, with* $F, G \in \mathbb{Z}[x, y]$*, is a root of a polynomial with integer coefficients. Then, derive a corresponding bivariate system in* α *and* γ*, where* $\gamma = \alpha + \beta, \alpha \cdot \beta, 1/\alpha$*, etc.*

(b) Determine a polynomial $f \in \mathbb{Z}[x]$ *with* $f(\sqrt{3} - \sqrt[3]{3} + 1) = 0$*.*

[4] A mapping $\phi : R \mapsto R'$ is a ring homomorphism if $\phi(1_R) = 1_{R'}$, and $\phi(a + b) = \phi(a) + \phi(b)$ and $\phi(a \cdot b) = \phi(a) \cdot \phi(b)$ for all $a, b \in R$.

Theorem 3.4.7. *Let $f, g \in R[x]$ be polynomials of degrees m and n, respectively, and α an arbitrary element in R. Then, it holds that*

$$\mathrm{Res}((x-\alpha)\cdot f, g) = g(\alpha)\cdot \mathrm{Res}(f,g).$$

For polynomials $f, g \in \mathbb{C}[x]$ with complex roots $\alpha_1, \ldots, \alpha_m$ and $\beta_1, \ldots, \beta_n$, respectively, it holds that:

$$\mathrm{Res}(f,g) = \mathrm{LC}(f)^n \cdot \prod_{i=1}^{m} g(\alpha_i) = (-1)^{mn}\cdot \mathrm{LC}(g)^m \cdot \prod_{i=1}^{n} f(\beta_i) = \mathrm{LC}(f)\cdot \mathrm{LC}(g)\cdot \prod_{i=1}^{m}\prod_{j=1}^{n}(\alpha_i - \beta_j).$$

Proof. Write $f = a_0 + \cdots + a_m \cdot x^m$ and $g = b_0 + \cdots + b_n \cdot x^n$. Now, we define

$$f^* := (x-\alpha)\cdot f = -\alpha \cdot a_0 + \sum_{i=1}^{m} (\alpha_{i-1} - \alpha a_i)\cdot x^i + a_m \cdot x^{m+1},$$

and consider the Sylvester Matrix of f^* and g:

$$\mathrm{Syl}(f^*, g) = \begin{pmatrix} a_m & a_{m-1}-\alpha a_m & \cdots & -\alpha a_0 & & & \\ & a_m & a_{m-1}-\alpha a_m & \cdots & -\alpha a_0 & & \\ & & \ddots & & & \ddots & \\ & & & a_m & a_{m-1}-\alpha a_m & \cdots & -\alpha a_0 \\ b_n & b_{n-1} & \cdots & b_0 & & & \\ & b_n & b_{n-1} & \cdots & b_0 & & \\ & & \ddots & & & \ddots & \\ & & & b_n & b_{n-1} & \cdots & b_0 \end{pmatrix}.$$

Our goal is to transform the above matrix into a matrix of the form

$$\mathrm{Syl}(f^*, g) = \begin{pmatrix} & \mathrm{Syl}(f,g) & & 0 \\ & & & 0 \\ & & & 0 \\ * & * & * & g(\alpha) \end{pmatrix}. \tag{3.3}$$

For this, we start with $\mathrm{Syl}(f^*, g)$ and add the first column multiplied by α to the second column. Then, the second column multiplied by α is added to the third column, and so on. This yields the matrix

$$S := \begin{pmatrix} a_m & a_{m-1} & \cdots & a_0 & & & & \\ & a_m & a_{m-1} & \cdots & a_0 & & & \\ & & \ddots & & & \ddots & & \\ & & & a_m & a_{m-1} & \cdots & a_0 & 0 \\ b_n & b_{n-1}+\alpha b_n & b_{n-2}+\alpha b_{n-1}+\alpha^2 b_{n-2} & \cdots & & & & \\ & b_n & b_{n-1}+\alpha b_n & \cdots & \cdots & & & \\ & & \ddots & & & & & \\ & & & b_n & b_{n-1}+\alpha b_n & \cdots & \cdots & b_0+\cdots+b_n\alpha^n \end{pmatrix}.$$

In a second step, we subtract α-times the $(n+2)$-nd row of the above matrix from the $(n+1)$-st row. Then, we subtract the $(n+3)$-rd row multiplied by α from the $(n+2)$-nd,

and so on. Following this approach, we obtain a matrix as in (3.3), whose determinant equals $g(\alpha) \cdot \mathrm{Syl}(f,g)$. This proves the first part.

For the second part, notice that $f(x) = \mathrm{LC}(f) \cdot \prod_{i=1}^{m}(x-\alpha_i)$ and $g(x) = \mathrm{LC}(g) \cdot \prod_{i=1}^{n}(x-\beta_i)$. Now, recursive application of the first part and Theorem 3.4.2 (c) yields that

$$\mathrm{Res}(f,g) = \mathrm{LC}(f)^n \cdot \mathrm{Res}(\prod_{i=1}^{m}(x-\alpha_i), g) = \mathrm{LC}(f)^n \cdot \prod_{i=1}^{m} g(\alpha_i).$$

Since $\mathrm{Res}(f,g) = (-1)^{mn} \cdot \mathrm{Res}(g,f)$, we further conclude that

$$\mathrm{Res}(f,g) = (-1)^{mn} \cdot \mathrm{LC}(g)^m \cdot \prod_{i=1}^{m} f(\beta_i) = (-1)^{mn} \cdot \mathrm{LC}(g)^m \cdot \mathrm{LC}^n \cdot \prod_{i=1}^{m}\prod_{j=1}^{n}(\beta_i - \alpha_j).$$

□

As a consequence of the above result, we are now ready to prove some useful bounds on the absolute values of the roots of a polynomial $f \in \mathbb{Z}[x]$ as well as on the distances between distinct roots.

Theorem 3.4.8 (and Definition)**.** *Let $f = a_0 + \cdots + a_n x^n \in \mathbb{C}[x]$ be a polynomial of degree n with coefficients of absolute value less than 2^L, and let $\alpha_1, \ldots, \alpha_n$ be the complex roots of f. Then, it holds*

(a) The Mahler Measure

$$\mathrm{Mea}(f) := \mathrm{LC}(f) \cdot \prod_{i=1}^{n} \max(1, |\alpha_i|)$$

is upper bounded by the 2-norm $\|f\|_2 := \sqrt{a_0^2 + \cdots + a_n^2} \leq \sqrt{n+1} \cdot 2^L$ *of f.*

(b) if f has integer coefficients of length less than L and if the roots α_i are pairwise distinct, then the separation

$$\mathrm{sep}(\alpha_i, f) := \min_{j \neq i} |\alpha_i - \alpha_j|$$

of each root α_i is lower bounded by $2^{-O(n(\log n + L))}$. We call $\mathrm{sep}(f) := \min_i \mathrm{sep}(\alpha_i, f)$ the separation of f.

Proof. For (a), we first show that

$$\|(x-z) \cdot f\|_2 = \|(\bar{z}x - 1) \cdot f\|_2$$

for arbitrary $z = a + \mathbf{i} \cdot b \in \mathbb{C}$ and its cojugate $\bar{z} = a - \mathbf{i} \cdot b$. Namely, with $f(x) = a_0 + \cdots + a_n \cdot x^n$ and $a_{-1} = a_{n+1} = 0$, we have $(x-z) \cdot f = \sum_{i=0}^{n+1}(a_{i-1} - z \cdot a_i) \cdot x^i$, and thus

$$\begin{aligned}
\|(x-z) \cdot f\|_2 &= \sum_{i=0}^{n+1}(a_{i-1} - z a_i) \cdot (\bar{a}_{i-1} - \bar{z}\bar{a}_i) \\
&= \sum_{i=0}^{n+1} \left[(|a_{i-1}|^2 + |z|^2 |a_i|^2) - (z a_i \bar{a}_{i-1} + \bar{z} a_{i-1} \bar{a}_i)\right] \\
&= (1 + |z|^2) \cdot \sum_{i=0}^{n} |a_i|^2 - \sum_{i=0}^{n}(z a_i \bar{a}_{i-1} + \bar{z}\bar{a}_i a_{i-1})
\end{aligned}$$

In completely analogous manner, we can expand $\|(\bar{z}x-1)\cdot f\|_2$, which yields exactly the same expression. Hence, we conclude that

$$\begin{aligned}\|f\|_2 &= \|a_n \cdot \prod_{i=1}^{n}(x-\alpha_i)\|_2 \\ &= \|a_n \cdot \prod_{i:|\alpha_i|\geq 1}(x-\alpha_i)\cdot \prod_{i:|\alpha_i|<1}(x-\alpha_i)\|_2 \\ &= \|a_n \cdot \prod_{i:|\alpha_i|\geq 1}(x\bar{\alpha}_i-1)\cdot \prod_{i:|\alpha_i|<1}(x-\alpha_i)\|_2\end{aligned}$$

Since the leading coefficient of $f^* := a_n \cdot \prod_{i:|\alpha_i|\geq 1}(x\bar{\alpha}_i-1)\cdot\prod_{i:|\alpha_i|<1}(x-\alpha_i)$ equals the Mahler measure of f, it follows that $\mathrm{Mea}(f) \leq \|f^*\|_2 = \|f\|_2$.

We now prove (b): We first show that

$$2^{-4n(\log n+L)} < \prod_{i\in I}|f'(\alpha_i)| < 2^{4n(\log n+L)}$$

for any subset I of $\{1,\ldots,n\}$. For the right inequality, we use that $|f'(\alpha_i)| < n^2\cdot 2^L \cdot \max(1,\alpha_i)^{n-1}$, and thus

$$\prod_{i\in I}|f'(\alpha_i)| < n^{2n}\cdot 2^{nL}\cdot \mathrm{Mea}(f)^{n-1} < (n+1)^{\frac{n-1}{2}}\cdot n^{2n}\cdot 2^{2nL} < 2^{3n(\log n+L)}.$$

Since f and f' do not share a common factor (f has only simple roots), $\mathrm{Res}(f,f')$ is non-zero. Since $\mathrm{Res}(f,f')$ is the determinant of an integer matrix, we further conclude that $\mathrm{Res}(f,f')$ is a non-zero integer, which implies that

$$1 \leq |\operatorname{Res}(f,f')| = |\operatorname{LC}(f)|^{n-1}\cdot\prod_{i=1}^{n}|f'(\alpha_i)| < 2^{4n(\log n+L)},$$

and thus

$$\prod_{i\in I}|f'(\alpha_i)| = \frac{\prod_{i=1}^{n}|f'(\alpha_i)|}{\prod_{i\notin I}|f'(\alpha_i)|} > \frac{|\operatorname{Res}(f,f')|\cdot\operatorname{LC}(f)^{-(n-1)}}{\prod_{i\notin I}|f'(\alpha_i)|} > \frac{2^{-(n-1)L}}{2^{3n(\log n+L)}} > 2^{-4n(\log n+L)}.$$

In order to estimate the separation of a specific root α_i, consider a root $\alpha_{j_i}\neq\alpha_i$ that minimizes the distance between α_i and any other root such that $\mathrm{sep}(\alpha_i,f) = |\alpha_{j_i}-\alpha_i|$. Then, since $|f'(\alpha_i)| = |a_n|\cdot\prod_{j\neq i}|\alpha_i-\alpha_j|$, we obtain

$$\begin{aligned}|f'(\alpha_i)| &= \mathrm{sep}(\alpha_i,f)\cdot|a_n|\cdot\prod_{j\neq i,i_j}|\alpha_j-\alpha_i| \\ &< \mathrm{sep}(\alpha_i,f)\cdot\mathrm{Mea}(f(x+\alpha_i)) \\ &< \mathrm{sep}(\alpha_i,f)\cdot\sqrt{n+1}\cdot 2^{2n+L}\cdot\max(1,|\alpha_i|)^n,\end{aligned}$$

where the latter two inequalities follow from the fact that $f(x+\alpha_i)$ has the roots $\alpha_j-\alpha_i$, with $j=1,\ldots,n$, and that the coefficients of $f(x+\alpha_i)$ are of absolute value less than $(n+1)\cdot 2^L\cdot 2^n\cdot\max(1,|\alpha_i|)^n < 2^{2n+L}\cdot\max(1,|\alpha_i|)^n$. We thus obtain

$$\mathrm{sep}(\alpha_i,f) > \frac{|f'(\alpha_i)|}{2^{2n+L}\cdot\sqrt{n+1}\cdot\max(1,|\alpha_i|)^n} > \frac{2^{-4n(\log n+L)}}{2^{3n+L}\cdot 2^{n(L+1)}} > 2^{-8n(\log n+L)}.$$

For the product $\prod_{i\in I} \operatorname{sep}(\alpha_i, f)$ over an arbitrary subset I of $\{1, \dots, n\}$, we obtain:

$$\prod_{i\in I} \operatorname{sep}(\alpha_i, f) > \prod_{i\in I} \frac{|f'(\alpha_i)|}{2^{2n+L} \cdot \sqrt{n+1} \cdot \max(1, |\alpha_i|)^n} > \frac{2^{-4n(\log n + L)}}{2^{n(3n+L)} \cdot \operatorname{Mea}(f)^n} > 2^{-8n(n+L)}$$

□

Exercise 3.4.9. *For two polynomials $f, g \in \mathbb{C}[x]$ and a disk Δ in complex space, Rouché's Theorem states that if*

$$|f(z)| > |f(z) - g(z)| \text{ for all } z \in \partial\Delta,$$

with $\partial\Delta$ the boundary of Δ, then f and g have the same number of roots in Δ. Use Rouché's Theorem to show that, for $n \geq 8$, the so-called Mignotte polynomial

$$f(x) = x^n - (2^L \cdot x - 1)^2$$

has two distinct real roots x_1 and x_2 with $|x_1 - x_2| < 2^{-\frac{nL}{2}+1}$.

Hint: *Use the fact that $g := -(2^L \cdot x - 1)^2$ has a root of multiplicity 2 at $m = 2^{-L}$. Then, consider a disc Δ centered at m and of suitable radius, and compare the values of $|f|$ and $|f - g|$ at the boundary of Δ.*

Without proof, we state the following theorem that extends the results from Theorem 3.4.8 to the general case, where f is allowed to have multiple roots. It further provides amortized bounds on the (weighted) product of all separations. Notice that the bound in (a) also constitutes an improvement upon the bound $\sum_{i=1}^{n} |\log \operatorname{sep}(\alpha_i, f)| = O(n(n+L))$ that we have already derived in the proof of Theorem 3.4.8. For proofs of Theorem 3.4.10, consider [MSW15, Thm. 5] and [KS15, Thm. 9].

Theorem 3.4.10. *Let $f \in \mathbb{Z}[x]$ be a polynomial of degree n with integer coefficients of length less than L, and let $\alpha_1, \dots, \alpha_m$ be the distinct complex roots of f with corresponding multiplicities $\mu_i := \mu(\alpha_i, f)$. Then, for an arbitrary subset I of $\{1, \dots, m\}$, it holds that*

(a) $\sum_{i\in I} |\log \operatorname{sep}(\alpha_i, f)| = O(n(\log n + L))$,

(b) $\sum_{i\in I} \mu_i \cdot |\log \operatorname{sep}(\alpha_i, f)| = O(n(n + L))$, *and*

(c) $\sum_{i\in I} |\log \frac{\partial^{\mu_i} f}{\partial x^{\mu_i}}(\alpha_i)| = O(n(\log n + L))$.

Another application of Theorem 3.4.8 (a) is a bound on the length of the coefficients of a divisor $g \in \mathbb{Z}[x_1, \dots, x_n]$ of a multivariate polynomial $f \in \mathbb{Z}[x_1, \dots, x_n]$ with integer coefficients.

Theorem 3.4.11. *Let $f \in \mathbb{Z}[x_1, \dots, x_n]$ be an integer polynomial of total degree d with integer coefficients of length less than 2^L. Then, each divisor $g \in \mathbb{Z}[x_1, \dots, x_n]$ of f has coefficients of length $O(d \log d + L)$.*

Proof. We prove the claim via induction over n. For a univariate $f \in \mathbb{Z}[x_1]$, we remark that $\operatorname{Mea}(g) \leq \operatorname{Mea}(f) \leq \|f\|_2 \leq (d+1) \cdot 2^L$, and thus the absolute value of each coefficient of g is bounded by $2^d \operatorname{Mea}(g) \leq (d+1) \cdot 2^{d+L}$.

For the general case, we write

$$f(x_1, \ldots, x_n) = \sum_{\lambda=(\lambda_1,\ldots,\lambda_{n-1})} a_\lambda(x_n) \cdot x_1^{\lambda_1} \cdots x_{n-1}^{\lambda_{n-1}}, \text{ with } a_\lambda \in \mathbb{Z}[x_n]$$

and

$$g(x_1, \ldots, x_n) = \sum_{\lambda=(\lambda_1,\ldots,\lambda_{n-1})} b_\lambda(x_n) \cdot x_1^{\lambda_1} \cdots x_{n-1}^{\lambda_{n-1}}, \text{ with } b_\lambda \in \mathbb{Z}[x_n].$$

For a fixed $\bar{x}_n \in \{0, \ldots, d\}$, the polynomial $g(x_1, \ldots, x_{n-1}, \bar{x}_n) \in \mathbb{Z}[x_1, \ldots, x_{n-1}]$ is a divisor of $f(x_1, \ldots, x_{n-1}, \bar{x}_n) \in \mathbb{Z}[x_1, \ldots, x_{k-1}]$. In addition, since $|\bar{x}_n|^d \leq d^d = 2^{d \log d}$ and since $a_\lambda(x_n)$ has degree d or less, it follows that $f(x_1, \ldots, x_{n-1}, \bar{x}_n)$ has coefficients of length $O(d \log d + L)$. Hence, from the induction hypothesis, we conclude that the polynomial $g(x_1, \ldots, x_{n-1}, \bar{x}_n)$ has coefficients of length $O(L + d \log d)$. It thus follows that $b_\lambda(\bar{x}_n) \in \mathbb{Z}$ has length bounded by $\tilde{O}(L + d \log d))$ for all $\bar{x}_n \in \{0, \ldots, d\}$ Since $b_\lambda(x_n)$ is a polynomial of degree at most d, we further conclude that $b_\lambda(x_n)$ is uniquely determined by its values at $x_n = 0, \ldots, n$. Hence, Lagrange interpolation yields

$$b_\lambda(x) = \sum_{i=0}^{d} b_\lambda(i) \cdot \frac{x \cdot (x-1) \cdots (x-i+1)(x-i-1) \cdots (x-d)}{i \cdot (i-1) \cdots 1 \cdot (-1) \cdots (i-d)}$$

Expanding the numerator of the fraction yields an integer polynomial with coefficients of length $O(d \log d)$. The denominator is a non-zero integer, and thus each coefficient of $b_\lambda(x_n)$ has length $O(L + d \log d)$ because $b_\lambda(i)$ has length $O(L + d \log d)$ and there are $d+1$ summands. This proves the claim. □

3.5 Subresultants

We have seen in the previous section that the problem of deciding whether two polynomials $f, g \in R[x]$ share a common non-trivial factor can be reduced to the computation of the determinant of a matrix whose entries are the coefficients of the given polynomials. We now extend this approach to determine the actual degree $k_0 = \deg h$ of the greatest common divisor $h := \gcd(f, g)$ of f and g. We will further show how to obtain h as the determinant of a Sylvester-like matrix. For this, we start with a generalization of Lemma 3.4.1:

Lemma 3.5.1. *Let $f, g \in R[x]$ be two polynomials of degrees m and n, respectively, and let $k_0 = \deg h$ be the degree of $h := \gcd(f, g)$. Then, k_0 is the minimal integer k such that*

$$\forall u, v \in R[x] \text{ with } \deg u < n - k \text{ and } 0 \leq \deg v < m - k \text{ it hold that } \deg(u \cdot f + v \cdot g) \geq k. \tag{3.4}$$

Proof. Let k^* be the minimal k such that (3.4) holds. We first show that $k_0 \leq k^*$: Define $u := g/h$ and $t = -f/h$, then $\deg u = n - k_0 < n - (k_0 - 1)$, $\deg v = m - k_0 < m - (k_0 - 1)$, and $\deg(uf + vg) = -\infty$. Hence, it follows that $k_0 - 1 < k^*$.

It remains to show that $k_0 \geq k^*$: Consider polynomials $u, v \in R[x]$ with $\deg u < n - k_0$ and $0 \leq \deg v < m - k_0$. Since $u \cdot f + v \cdot g$ is a multiple of h, we either have $\deg(u \cdot f + v \cdot g) \geq k_0$ or $u \cdot f + v \cdot g = 0$. Since f/h and g/h are coprime, $u \cdot f = -v \cdot g$ implies that g/h divides u and that f/h divides v. However, since $\deg f/h = m - k_0$ and $\deg g/h = n - k_0$, this is not possible because of the degree bounds on u and v. This shows that $\deg(u \cdot f + v \cdot g) \geq k_0$, and thus $k_0 \geq k^*$. □

In order to reformulate the above lemma in terms of linear algebra, we consider the contrapositive of (3.4): If k_0 is the degree of $h = \gcd(f, g)$, then, for all $k < k_0$, there exist polynomials $u = u_0 + \cdots + u_{n-k-1} \cdot x^{n-k-1}$ and $v = v_0 + \cdots + v_{n-k-1} \cdot x^{n-k-1} \neq 0$ such that $\deg(u \cdot f + v \cdot g) < k$. This is equivalent to the existence of a non-trivial solution of the following linear system, where we use $a_0, \ldots, a_m$ and $b_0, \ldots, b_n$ to denote the coefficients of the polynomials f and g, respectively:

$$\begin{pmatrix} u_{n-k-1} & \cdots & u_0 & v_{m-k-1} & \cdots & v_0 \end{pmatrix} \cdot \underbrace{\begin{pmatrix} a_m & \cdots & a_0 & & & \\ & a_m & \cdots & a_0 & & \\ & & \ddots & & \ddots & \\ & & & a_m & \cdots & a_k \\ b_n & \cdots & b_0 & & & \\ & b_n & \cdots & b_0 & & \\ & & \ddots & & \ddots & \\ & & & b_n & \cdots & b_k \end{pmatrix}}_{=:\mathrm{Syl}_k(f,g)} = 0 \tag{3.5}$$

Here, $\mathrm{Syl}_k(f, g)$ is an $(m + n - 2k) \times (m + n - 2k)$-matrix, which is called the k-th *Sylvester Submatrix* of f and g. It can be obtained from the corresponding Sylvester matrix by removing the last $2k$ columns and the rows numbered from $n - k + 1$ until n and from $n + m - k + 1$ until $n + m$. Now, similar to the definition of the resultant, we introduce the following more general definition:

Lemma 3.5.2 (and Definition of Subresultants)**.** *The k-th* (polynomial) subresultant of f and g *is defined as* $\mathrm{Sres}_k(x) := \det \mathrm{Syl}_k^* \in R[x]$*, with*

$$\mathrm{Syl}_k^* := \begin{pmatrix} a_m & \cdots & a_0 & & & & x^{n-k-1} \cdot f \\ & a_m & \cdots & a_0 & & & x^{n-2} \cdot f \\ & & \ddots & & \ddots & & \vdots \\ & & & a_m & \cdots & a_{k+1} & x^0 \cdot f \\ b_n & \cdots & b_0 & & & & x^{m-k-1} \cdot g \\ & b_n & \cdots & b_0 & & & x^{m-2} \cdot g \\ & & \ddots & & \ddots & & \vdots \\ & & & b_n & \cdots & b_{k+1} & x^0 \cdot g \end{pmatrix}.$$

$\mathrm{Sres}_k = c_{k,0} + \cdots + c_{k,k} \cdot x^k$ *is a polynomial of degree at most k and $c_{k,k} = \det \mathrm{Syl}_k(f, g)$. We call* $\mathrm{sres}_k := \det \mathrm{Syl}_k(f, g)$ *the* k-th leading subresultant coefficient *of f and g. It further holds that*

$$\mathrm{Sres}_k(f, g) = u_k \cdot f + v_k \cdot g,$$

with polynomials $u_k, v_k \in R[x]$ of degree less than $n - k$ and $m - k$, respectively.

Proof. The proof is similar to the one of Theorem 3.4.2 (b). Namely, using linearity of the determinant, we obtain

$$\det \mathrm{Syl}_k^*(f, g) = \sum_{j=0}^{m+n-k-1} \det(S_{k,j}) \cdot x^j,$$

with

$$S_{k,j} := \begin{pmatrix} a_m & \cdots & a_0 & & & a_{j-(n-k-1)} \\ & a_m & \cdots & a_0 & & a_{j-(n-k-2)} \\ & & \ddots & & \ddots & \vdots \\ & & & a_m & \cdots\ a_{k+1} & a_j \\ b_n & \cdots & b_0 & & & b_{j-(m-k-1)} \\ & b_n & \cdots & b_0 & & b_{j-(m-k-2)} \\ & & \ddots & & \ddots & \vdots \\ & & & b_n & \cdots\ b_{k+1} & b_j \end{pmatrix},$$

where we define $a_i = b_i = 0$ for $i < 0$, $a_i = 0$ for $i > m$, and $b_j = 0$ for $j > n$. Now, notice that, for $j > k$, the $(m+n-k-j)$-th column of $S_{k,j}$ coincides with the last column of $S_{k,j}$, and thus we have $\det S_{k,j}(f,g) = 0$ for all $j > k$. Furthermore, since $S_{k,k} = \mathrm{Syl}_k(f,g)$, the coefficient of x^k of Sres_k equals $\det \mathrm{Syl}_k(f,g)$. The last claim follows directly from using Laplace expansion for the computation of $\det \mathrm{Syl}_k^*(f,g)$. □

Notice that $\mathrm{Sres}_0(f,g)$ equals the determinant of $\mathrm{Syl}_0(f,g)$, and that $\mathrm{Syl}_0(f,g)$ is just the Sylvester matrix of f and g. Hence, we have $\mathrm{Sres}_0(f,g) = \mathrm{sres}_0(f,g) = \mathrm{Res}(f,g)$. In the above theorem, we have shown that there exists polynomials $u_k, v_k \in R[x]$ of respective degrees less than $n-k$ and $m-k$ such that $\mathrm{Sres}_k(f,g) = u_k \cdot f + v_k \cdot g$. According to the following exercise, the cofactors u_k and v_k can be written as determinants of Sylvester-like matrices.

Exercise 3.5.3. *Show that*

$$u_k := \begin{vmatrix} a_m & \cdots & a_0 & & & & x^{n-k-1} \\ & a_m & \cdots & a_0 & & & x^{n-2} \\ & & \ddots & & \ddots & & \vdots \\ & & & a_m & \cdots & a_{k+1} & x^0 \\ b_n & \cdots & b_0 & & & & 0 \\ & b_n & \cdots & b_0 & & & 0 \\ & & \ddots & & \ddots & & \vdots \\ & & & b_n & \cdots & b_{k+1} & 0 \end{vmatrix}, v_k := \begin{vmatrix} a_m & \cdots & a_0 & & & & 0 \\ & a_m & \cdots & a_0 & & & 0 \\ & & \ddots & & \ddots & & \vdots \\ & & & a_m & \cdots & a_{k+1} & 0 \\ b_n & \cdots & b_0 & & & & x^{m-k-1} \\ & b_n & \cdots & b_0 & & & x^{m-2} \\ & & \ddots & & \ddots & & \vdots \\ & & & b_n & \cdots & b_{k+1} & x^0 \end{vmatrix}$$

are polynomials of respective degrees less than $n-k$ and $m-k$ such that

$$u_k \cdot f + v_k \cdot g = \mathrm{Sres}_k(f,g).$$

Combining Lemma 3.5.1 and 3.5.2 now yields the following result, which allows us to read off the degree of the gcd of f and g directly from the subresultants of f and g.

Corollary 3.5.4. *For $f, g \in R[x]$, it holds that*

$$k_0 := \deg \gcd(f,g) = \min\{k \in \mathbb{N} : \mathrm{Sres}_k(f,g) \not\equiv 0\} = \min\{k \in \mathbb{N} : \mathrm{sres}_k(f,g) \neq 0\}.$$

For R a field, we further have $\mathrm{Sres}_{k_0}(f,g) \sim \gcd(f,g)$.

Proof. Since $u_k \cdot f + v_k \cdot g = \mathrm{Sres}_k(f,g) = 0$, it follows that $h := \gcd(f,g)$ divides $\mathrm{Sres}_k(f,g)$. Hence, since $\deg \mathrm{Sres}_k(f,g) \leq k$ for all k, it follows that $\mathrm{Sres} \equiv 0$ for all $k < k_0$. For $k = k_0$, Lemma 3.5.1 guarantees that there does not exist a solution of (3.5), and thus $\mathrm{sres}_{k_0}(f,g) \neq 0$. If R is a field, then we must have $\mathrm{Sres}_{k_0}(f,g) \sim h$ as h divides $\mathrm{Sres}_{k_0}(f,g)$ and has the same degree as $\mathrm{Sres}_{k_0}(f,g)$. □

In the next step, we aim to show that the polynomials s_i, t_i, r_i as computed in the Extended Euclidean Algorithm are associated to polynomials u_{k_i}, v_{k_i}, and Sres_{k_i}. For this, we make use of the following result:

Lemma 3.5.5. *Let F be a field and $r, s, t, f, g \in F[x]$ be polynomials such that*

$$r = s \cdot f + t \cdot g,\ t \neq 0,\ \textit{and}\ \deg r + \deg t < \deg f = m$$

Let $r_0, \ldots, r_{\ell+1}$ be the remainders as computed by the EEA with input f and g, and let $j \in \{0, \ldots, \ell+1\}$ be the unique value with $\deg r_j \leq \deg r < \deg r_{j-1}$. Then, there exists a $\lambda \in F[x]$ such that

$$r = \lambda \cdot r_j,\ , s = \lambda \cdot s_j,\ \textit{and}\ t = \lambda \cdot t_j.$$

Proof. We first argue by contradiction that $s_j \cdot t = s \cdot t_j$: Suppose that $s_j \cdot t \neq s \cdot t_j$, then the matrix $\begin{pmatrix} s_j & t_j \\ s & t \end{pmatrix}$ is invertible. Hence, using Cramer's Rule, we obtain

$$\begin{pmatrix} s_j & t_j \\ s & t \end{pmatrix} \cdot \begin{pmatrix} f \\ g \end{pmatrix} = \begin{pmatrix} r_j \\ r \end{pmatrix} \Rightarrow f = \frac{\begin{vmatrix} r_j & t_j \\ r & t \end{vmatrix}}{\begin{vmatrix} s_j & t_j \\ s & t \end{vmatrix}}.$$

However, this is not possible as

$$\begin{aligned} \deg(r_j \cdot t - r \cdot t_j) &\leq \max(\deg r_j + \deg t, \deg r + \deg t_j) \\ &= \max(\deg r_j + \deg t, \deg r + m - \deg r_{j-1}) \\ &< \max(m, \deg r_{j-1} + m - \deg r_{j-1}) \\ &= \deg f. \end{aligned}$$

Here, we used Lemma 3.2.4 (e) to show that $\deg t_j = m - \deg r_{j-1}$. In Lemma 3.2.4, we have further shown that s_j and t_j are coprime, and thus s_j divides s and t_j divides t. It follows that there exist polynomials $\lambda, \lambda' \in F[x]$ with $s = \lambda \cdot s_j$ and $t = \lambda' \cdot t_j$, and since $s_j \cdot t = s \cdot t_j$, we further conclude that $\lambda = \lambda'$. Finally, we have

$$r = s \cdot f + t \cdot g = \lambda \cdot (s_j \cdot f + t_j \cdot g) = \lambda \cdot r_j.$$

□

We are now ready to prove one of the main results in this section, namely that each remainder as computed by the EEA coincide with the corresponding subresultant polynomial of the same degree up to a factor in F.

Theorem 3.5.6. *Let $n_i := \deg r_i$ be the degree of the remainder r_i as computed by the EEA with input $f, g \in F[x]$. Then, we have $r_i \sim \mathrm{Sres}_{n_i}(f, g)$. Furthermore, $\mathrm{sres}_k(f, g)$ vanishes if and only if k does not appear in the degree sequence $n_0, n_1, \ldots, n_\ell$.*

Proof. We have already shown that there exist polynomials u_k and v_k of respective degree less than $n - k$ and $m - k$ such that

$$u_k \cdot f + v_k \cdot g = \mathrm{Sres}_k(f, g).$$

Now, let i, with $2 \leq j \leq \ell+1$, be the unique index such that $n_i \leq k^* := \deg \mathrm{Sres}_k(f,g) < n_{i-1}$. Then, $s := u_k$ and $t := v_k$ fulfill the conditions in Lemma 3.5.5, and thus there exists a $\lambda \in F[x]$ such that $u_k = \lambda \cdot s_i$, $v_k = \lambda \cdot t_i$, and $\lambda \cdot r_i = \lambda \cdot \mathrm{Sres}_k(f,g)$. It further holds that

$$m - n_{i-1} < \deg t_j \leq \deg v_k < m - k \Rightarrow n_{i-1} > k,$$

and thus $n_i \leq k^* \leq k < n_{i-1}$. Hence, k cannot appear in the degree sequence if $k^* \neq k$. Vice versa, if k does not appear in the degree sequence, then the equality

$$s_i \cdot f + t_i \cdot g = r_i$$

implies that $\mathrm{sres}_k(f,g) = 0$ as $\deg s_i = n - r_{i-1} < n - k$, $\deg t_i = m - r_{i-1} < m - k$, and $\deg r_i = n_i < k$. We thus conclude that k appears in the degree sequence if and only if $\mathrm{sres}_k(f,g) \neq 0$.

It remains to show that $\mathrm{Sres}_{n_i} \sim r_i$. In this case, there exist a $\lambda \in F[x]$ with $u_{n_i} = \lambda \cdot s_i$, $v_{n_i} = \lambda \cdot t_i$, and $\lambda \cdot r_i = \lambda \cdot \mathrm{Sres}_{n_i}(f,g)$. Since both polynomials r_i and $\mathrm{Sres}_{n_i}(f,g)$ have degree n_i, we must have $\lambda \in F$, hence the claim follows. □

We can now bound the bitsize of the coefficients of the polynomials r_i, s_i, and t_i for input polynomials $f, g \in \mathbb{Z}[x]$.

Theorem 3.5.7. *Let f and g be polynomials of respective degrees m and n, with $m \geq n$, and integer coefficients of length less than L. Then, the polynomials r_i, s_i, and t_i computed by the EEA with input f and g, have rational coefficients with numerators and denominators of length $O(m(\log m + L))$.*

Proof. Let u_k and v_k be the polynomials in $\mathbb{Z}[x]$ of respective degrees less than $n-k$ and $m-k$ such that $u_k \cdot f + v_k \cdot g = \mathrm{Sres}_k(f,g)$. Each coefficient of each of the polynomials u_k, v_k, and $\mathrm{Sres}_k(f,g)$ can be computed as the determinant of a square matrix $M = (m_{i,j})_{i,j}$ of size $N \leq (m+n) \times (m+n)$ with integer entries of length at most L. The determinant of M is given as

$$\det M = \sum_{\sigma \in S_N} \mathrm{sign}(\sigma) \cdot m_{1,\sigma(1)} \cdots m_{N,\sigma(N)},$$

where we sum over all permutations σ of the the integers $1, \ldots, N$. Hence, $\det M$ is an integer of absolute value less than $N! \cdot 2^{NL}$, which shows that the polynomials u_k, v_k, and $\mathrm{Sres}_k(f,g)$ have coefficients of length bounded by $O(m(\log m + L))$. According to Theorem 3.5.6, there exists a rational λ with $r_i = \lambda \cdot \mathrm{Sres}_{n_i}(f,g)$, $s_i = \lambda \cdot u_{n_i}$ and $t_i = \lambda \cdot v_{n_i}$, with $n_i = \deg r_i$. Since r_i is monic, we thus conclude that $\lambda := \mathrm{LC}(\mathrm{Sres}_{n_i}(f,g))^{-1} = \mathrm{sres}_{n_i}(f,g)^{-1}$, which proves our claim. □

Notice that we can now use Exercise 2.2.6 to bound the bitsize of the coefficients of the quotients q_i and of the leading coefficient of the remainders $\mathrm{rem}(r_{i-1}, r_i)$ as computed in Step 6 of the EEA. Namely, since

$$r_{i-1} = q_i \cdot r_i + \rho_i \cdot r_{i+1}$$

it follows from the above bound on the bitsize of the coefficients of the r_k's that the coefficients of q_i as well as the leading coefficient ρ_i of the remainder $\mathrm{rem}(r_{i-1}, r_i)$ have bitsize bounded by $\tilde{O}(m^2 L)$. In fact, we can derive a bound that is by a factor n better:

Exercise 3.5.8. *Let r_i be the remainders as computed in the EEA, and let*

$$r_{i-1} = q_i \cdot r_i + \rho_{i+1} \cdot r_{i+1}.$$

Show that there exist integers μ_i of length $O(m(\tau + \log m))$ such that $\mu_i \cdot \rho_i$ and $\mu_i \cdot q_i$ are integers (integer polynomials) of length (with coefficients of length) $O(m(\tau + \log m))$!

Proceed as follows:

1. *Use that a comparable result has already been shown for s_i, t_i, and r_i!*
2. *Recall that*

$$R_i = \begin{pmatrix} s_i & t_i \\ s_{i+1} & t_{i+1} \end{pmatrix} = R_0 \cdot \prod_{j=1}^{i} Q_j, \quad \text{where}$$
$$R_0 = \begin{pmatrix} s_0 & t_0 \\ s_1 & t_1 \end{pmatrix} \quad \text{and} \quad Q_j = \begin{pmatrix} 0 & 1 \\ \rho_{j+1}^{-1} & -q_j \rho_{j+1}^{-1} \end{pmatrix}$$

   and, in particular, $\begin{vmatrix} s_i & t_i \\ s_{i+1} & t_{i+1} \end{vmatrix} = (-1)^{i-1}(\rho_0 \cdots \rho_i)^{-1}$. *Use these identities to derive a bound on the length of the numerator and denominator of ρ_i.*
3. *Show that $f = q \cdot g$ with $f, g \in \mathbb{Z}[x]$ polynomials of degree less than N and with integer coefficients of length less than L, and $q \in \mathbb{Q}[x]$ implies that there exists a $\lambda \in \mathbb{Z}$ with $|\lambda| < 2^L$ such that $\lambda \cdot q$ is a polynomial with integer coefficients of length $O(n + L)$.*

Notice that the bounds on the bitsizes of the polynomials r_i, s_i, and t_i as derived above imply that the EEA runs in polynomial time. Namely, since the intermediate results have bitsize bounded by $O(L(m + \log m))$, it follows that the cost for the division of r_{i-1} by r_i in the i-th iteration is bounded by $\tilde{O}(m^2 L)$. Hence, the total cost is bounded by $\tilde{O}(m^3 L)$. In the following section, we will see that is possible to reduce the cost to $\tilde{O}(m^2 L)$ using a more efficient variant of the EEA.

Exercise 3.5.9. *Let $f, g \in \mathbb{Z}[x]$ be integer polynomials of degree bounded by n and coefficients of absolute value less than 2^τ, let p be prime such that $p \nmid \mathrm{LC}(f)$ and $p \nmid \mathrm{LC}(g)$, and define $d \coloneqq \deg \gcd(f, g)$ to be the degree of the GCD of f and g.*

1. *Show that*

$$\gcd(f, g) \equiv \gcd(\bar{f}, \bar{g}) \mod p \quad \text{if and only if} \quad p \nmid \mathrm{sres}_d(f, g),$$

   where $\bar{f}$ and $\bar{g}$ are the modular images of f and g in $\mathbb{Z}/p\mathbb{Z}[x]$.
2. *Develop a modular algorithm to compute* under guarantee *the degree d of* $\gcd(f, g) \in \mathbb{Z}[x]$ *and determine its bit complexity in terms of n and τ.*

Exercise 3.5.10. *(a) Let*

$$f = x^3 + 4x^2 - 2ax - a^2 \quad \text{and}$$
$$g = x^2 - 2a^2.$$

Choose a such that $\deg \gcd(f, g) = 1$.

(b) Determine the gcd *of*

$$f = x^2 + \Big(\frac{1}{10}\sqrt{5} - \frac{3}{10}\Big)x + \Big(\frac{3}{50}\sqrt{5} - \frac{7}{50}\Big) \quad \textit{and}$$
$$g = 4x^2 + \Big(-\frac{1}{10}\sqrt{5} + \frac{3}{10}\Big)x + \Big(\frac{1}{25}\sqrt{5} - \frac{4}{25}\Big).$$

Bibliography

[AY62] Karatsuba A. and Ofman Y. "Multiplication of Many-Digital Numbers by Automatic Computers". In: *Doklady Akademii Nauk SSSR* 14 (1962), pp. 293–294 (cit. on p. 5).

[CT65] James W. Cooley and John W. Tukey. "An Algorithm for the Machine Calculation of Complex Fourier Series". In: *Mathematics of Computation* 19.90 (1965), pp. 297–301. ISSN: 00255718, 10886842 (cit. on p. 27).

[GG03] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, 2003. ISBN: 9780521826464 (cit. on pp. 24, 33, 34).

[KS15] Alexander Kobel and Michael Sagraloff. "On the complexity of computing with planar algebraic curves". In: *J. Complexity* 31.2 (2015), pp. 206–236 (cit. on p. 63).

[MB72] Robert T. Moenck and Allan Borodin. "Fast modular transforms via division". In: *13th*. 1972, pp. 90–96 (cit. on p. 38).

[MOS11] Kurt Mehlhorn, Ralf Osbild, and Michael Sagraloff. "A general approach to the analysis of controlled perturbation algorithms". In: *Comput. Geom.* 44.9 (2011), pp. 507–528 (cit. on p. 16).

[MS08] K. Mehlhorn and P. Sanders. *Algorithms and Data Structures: The Basic Toolbox*. SpringerLink: Springer e-Books. Springer, 2008. ISBN: 9783540779773 (cit. on pp. 6, 7).

[MSW15] Kurt Mehlhorn, Michael Sagraloff, and Pengming Wang. "From approximate factorization to root isolation with application to cylindrical algebraic decomposition". In: *J. Symb. Comput.* 66 (2015), pp. 34–69 (cit. on p. 63).

[SS71] A. Schönhage and V. Strassen. "Schnelle Multiplikation großer Zahlen". In: *Computing* 7.3 (1971), pp. 281–292. ISSN: 1436-5057 (cit. on p. 22).

[Too63] Andrei Toom. "The Complexity of a Scheme of Functional Elements Realizing the Multiplication of Integers". In: *Soviet Mathematics-Doklady* 7 (1963), pp. 714–716 (cit. on p. 7).