Draft. Aimed at Buhler-Stevenhagen MSRI conference proceedings.

FAST MULTIPLICATION AND ITS APPLICATIONS

DANIEL J. BERNSTEIN

Abstract. This survey explains how some useful arithmetic operations can be sped up from quadratic time to essentially linear time.

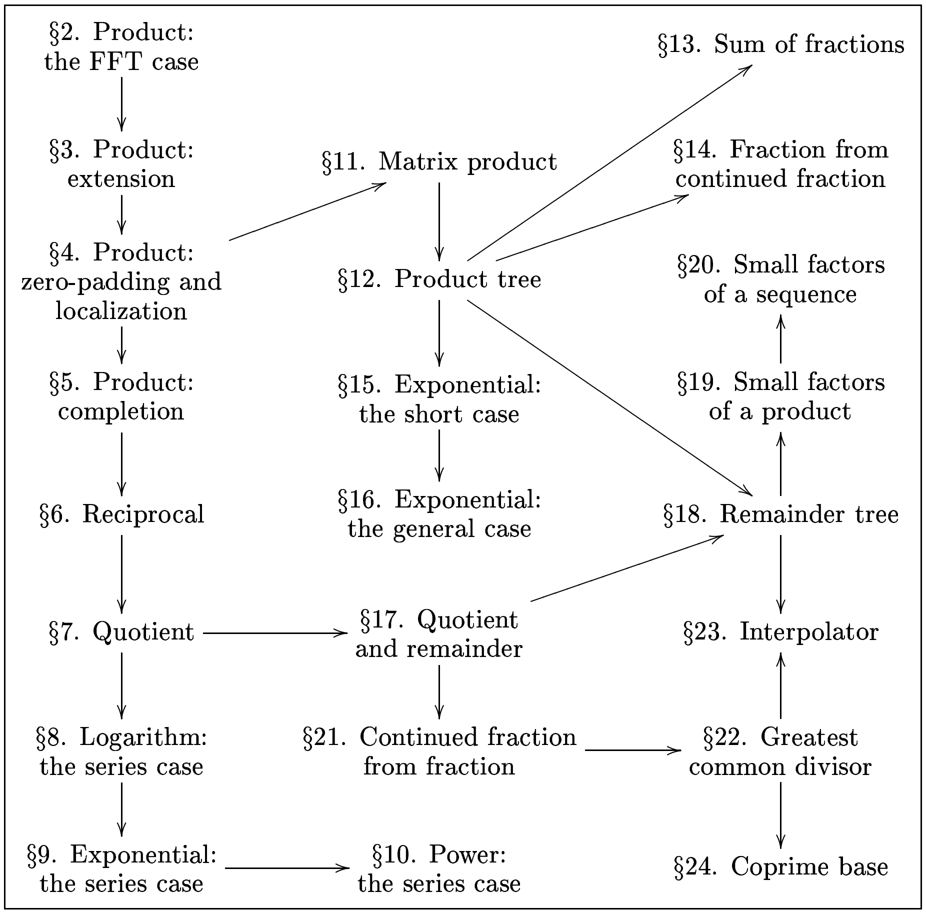
§2. Product: the FFT case
§3. Product: extension
§4. Product: zero-padding and localization
§5. Product: completion
§6. Reciprocal
§7. Quotient
§8. Logarithm: the series case
§9. Exponential: the series case
§11. Matrix product
§12. Product tree
§15. Exponential: the short case
§16. Exponential: the general case
§17. Quotient and remainder
§21. Continued fraction from fraction
§10. Power: the series case
§13. Sum of fractions
§14. Fraction from continued fraction
§20. Small factors of a sequence
§19. Small factors of a product
§18. Remainder tree
§23. Interpolator
§22. Greatest common divisor
§24. Coprime base

Figure 1. Outline of the paper. A vertex "§N. F" here means that Section N describes an algorithm to compute the function F. Arrows here indicate prerequisite algorithms.

Date: 20030119.

1991 *Mathematics Subject Classification.* Primary 68–02. Secondary 11Y16, 12Y05, 65D20, 65T50, 65Y20, 68W30.

1. Introduction

This paper presents fast algorithms for several useful arithmetic operations on polynomials, power series, integers, real numbers, and 2-adic numbers.

Each section focuses on one algorithm for one operation, and describes seven features of the algorithm:

- Input: What numbers are provided to the algorithm? Sections 2, 3, 4, and 5 explain how various mathematical objects are represented as inputs.
- Output: What numbers are computed by the algorithm?
- Speed: How many coefficient operations does the algorithm use to perform a polynomial operation? The answer is at most $n^{1+o(1)}$, where n is the problem size; each section states a more precise upper bound, often using the function μ defined in Section 4.
- How it works: What is the algorithm? The algorithm may use previous algorithms as subroutines, as shown in (the transitive closure of) Figure 1.
- The integer case (except in Section 2): The inputs were polynomials (or power series); what about the analogous operations on integers (or real numbers)? What difficulties arise in adapting the algorithm to integers? How much time does the adapted algorithm take?
- History: How were these ideas developed?
- Improvements: The algorithm was chosen to be reasonably simple (subject to the $n^{1+o(1)}$ bound) at the expense of speed; how can the same function be computed even more quickly?

Sections 2 through 5 describe fast multiplication algorithms for various rings. The remaining sections describe various applications of fast multiplication. Here is a simplified summary of the functions being computed:

- §6. Reciprocal. $f \mapsto 1/f$ approximation.
- §7. Quotient. $f, h \mapsto h/f$ approximation.
- §8. Logarithm. $f \mapsto \log f$ approximation.
- §9. Exponential. $f \mapsto \exp f$ approximation. Also §15, §16.
- §10. Power. $f, e \mapsto f^e$ approximation.
- §11. Matrix product. $f, g \mapsto fg$ for 2×2 matrices.
- §12. Product tree. $f_1, f_2, f_3, \ldots \mapsto$ tree of products including $f_1 f_2 f_3 \cdots$.
- §13. Sum of fractions. $f_1, g_1, f_2, g_2, \ldots \mapsto f_1/g_1 + f_2/g_2 + \cdots$.
- §14. Fraction from continued fraction. $q_1, q_2, \ldots \mapsto q_1 + 1/(q_2 + 1/(\cdots))$.
- §17. Quotient and remainder. $f, h \mapsto \lfloor h/f \rfloor, h \bmod f$.
- §18. Remainder tree. $h, f_1, f_2, \ldots \mapsto h \bmod f_1, h \bmod f_2, \ldots$.
- §19. Small factors of a product. $S, h_1, h_2, \ldots \mapsto S(h_1 h_2 \cdots)$ where S is a set of primes and $S(h)$ is the subset of S dividing h.
- §20. Small factors of a sequence. $S, h_1, h_2, \ldots \mapsto S(h_1), S(h_2), \ldots$.
- §21. Continued fraction from fraction. $f_1, f_2 \mapsto q_1, q_2, \ldots$ with $f_1/f_2 = q_1 + 1/(q_2 + 1/(\cdots))$.
- §22. Greatest common divisor. $f_1, f_2 \mapsto \gcd\{f_1, f_2\}$.
- §23. Interpolator. $f_1, g_1, f_2, g_2, \ldots \mapsto h$ with $h \equiv f_j \pmod{g_j}$.
- §24. Coprime base. $f_1, f_2, \ldots \mapsto$ coprime set S with $f_1, f_2, \ldots \in \langle S \rangle$.

Acknowledgments. Thanks to Alice Silverberg for her comments.

2. Product: the FFT case

Input. Let $n \ge 1$ be a power of 2. Let c be a nonzero element of $\mathbf{C}$. The algorithm described in this section is given two elements f, g of the ring $\mathbf{C}[x]/(x^n - c)$.

An element of $\mathbf{C}[x]/(x^n - c)$ is, by convention, represented as a sequence of n elements of $\mathbf{C}$: the sequence $(f_0, f_1, \ldots, f_{n-1})$ represents $f_0 + f_1 x + \cdots + f_{n-1}x^{n-1}$.

Output. This algorithm computes the product $fg \in \mathbf{C}[x]/(x^n - c)$, represented in the same way. If the input is $f_0, f_1, \ldots, f_{n-1}, g_0, g_1, \ldots, g_{n-1}$ then the output is $h_0, h_1, \ldots, h_{n-1}$, where $h_i = \sum_{0 \le j \le i} f_j g_{i-j} + c \sum_{i+1 \le j < n} f_j g_{i+n-j}$.

For example, for $n = 4$, the output is $f_0g_0 + cf_1g_3 + cf_2g_2 + cf_3g_1, f_0g_1 + f_1g_0 + cf_2g_3 + cf_3g_2, f_0g_2 + f_1g_1 + f_2g_0 + cf_3g_3, f_0g_3 + f_1g_2 + f_2g_1 + f_3g_0$.

Model of computation. Let A be a commutative ring. An **operation in** A is, by definition, one binary addition $a, b \mapsto a + b$, one binary subtraction $a, b \mapsto a - b$, or one binary multiplication $a, b \mapsto ab$. Here a is an input, a constant, or a result of a previous operation; same for b.

For example, given $a, b \in \mathbf{C}$, one can compute $10a + 11b, 9a + 10b$ with four operations in $\mathbf{C}$: add a and b to obtain $a + b$; multiply by 10 to obtain $10a + 10b$; add b to obtain $10a + 11b$; subtract a from $10a + 10b$ to obtain $9a + 10b$.

Starting in Section 19 of this paper, the definition of **operation in** A is expanded to allow equality tests. Starting in Section 21, the ring A is assumed to be a field, and the definition of **operation in** A is expanded to allow divisions (when the denominators are nonzero). Algorithms built out of additions, subtractions, multiplications, divisions, and equality tests are called **algebraic algorithms**. See [29, Chapter 4] for a precise definition of this model of computation.

Beware that the definition of an algebraic algorithm is sometimes simplified into a mere sequence of intermediate results: for example, the algorithm "add a to b, then multiply by b" is replaced by the sequence $a + b, ab + b^2$. The problem with this simplification is that standard measurements of algebraic complexity, such as the number of multiplications, are generally not determined by the sequence of intermediate results. (How many multiplications are in $2a, a^2, 2a^2$?) For example, the definition of addition-chain length in [64] is nonsense.

Speed. The algorithm in this section uses $O(n \lg n)$ operations—more precisely, $(9/2)n \lg n + 2n$ additions, subtractions, and multiplications—in $\mathbf{C}$. Here $\lg = \log_2$.

How it works. If $n = 1$ then the algorithm simply multiplies f_0 by g_0 to obtain the output f_0g_0.

The strategy for larger n is to split an n-coefficient problem into two $(n/2)$-coefficient problems, which are handled by the same method recursively. One needs $\lg n$ levels of recursion to split the original problem into n easy single-coefficient problems; each level of recursion involves $9/2$ operations per coefficient.

Consider, for any n, the functions $\varphi : \mathbf{C}[x]/(x^{2n} - c^2) \to \mathbf{C}[x]/(x^n - c)$ and $\varphi' : \mathbf{C}[x]/(x^{2n} - c^2) \to \mathbf{C}[x]/(x^n + c)$ that take $f_0 + \cdots + f_{2n-1}x^{2n-1}$ to $(f_0 + cf_n) + \cdots + (f_{n-1} + cf_{2n-1})x^{2n-1}$ and $(f_0 - cf_n) + \cdots + (f_{n-1} - cf_{2n-1})x^{2n-1}$ respectively. Given f, one can compute $\varphi(f), \varphi'(f)$ with n additions, n subtractions, and n multiplications by the constant c.

These functions φ, φ' are $\mathbf{C}[x]$-algebra morphisms. In particular, they preserve multiplication: $\varphi(fg) = \varphi(f)\varphi(g)$ and $\varphi'(fg) = \varphi'(f)\varphi'(g)$. Furthermore, $\varphi \times \varphi'$ is

$$\begin{tikzcd}
& & & \mathbf{C}[x]/(x^4+1) \arrow[dll, "x\mapsto x"'] \arrow[drr, "x\mapsto x"] & & & \\
& \mathbf{C}[x]/(x^2-i) \arrow[dl, "x\mapsto\sqrt{i}"'] \arrow[dr, "x\mapsto-\sqrt{i}"] & & & & \mathbf{C}[x]/(x^2+i) \arrow[dl, "x\mapsto\sqrt{-i}"'] \arrow[dr, "x\mapsto-\sqrt{-i}"] & \\
\mathbf{C} & & \mathbf{C} & & \mathbf{C} & & \mathbf{C}
\end{tikzcd}$$

FIGURE 2. Splitting product in $\mathbf{C}[x]/(x^4+1)$ into products in $\mathbf{C}$.

injective: one can recover f from $\varphi(f)$ and $\varphi'(f)$. It is simpler to recover $2f$: this takes n additions, n subtractions, and n multiplications by the constant $1/c$.

Here, then, is how the algorithm computes $2nfg$, given $f,g\in\mathbf{C}[x]/(x^{2n}-c^2)$:

- Compute $\varphi(f),\varphi(g),\varphi'(f),\varphi'(g)$ with $2n$ additions, $2n$ subtractions, and $2n$ multiplications by c.
- Recursively compute $n\varphi(f)\varphi(g)=\varphi(nfg)$ in $\mathbf{C}[x]/(x^n-c)$, and recursively compute $n\varphi'(f)\varphi'(g)=\varphi'(nfg)$ in $\mathbf{C}[x]/(x^n+c)$.
- Compute $2nfg$ from $\varphi(nfg),\varphi'(nfg)$ with n additions, n subtractions, and n multiplications by $1/c$.

For example, given $f=f_0+f_1x+f_2x^2+f_3x^3$ and $g=g_0+g_1x+g_2x^2+g_3x^3$, the algorithm computes $4fg$ in $\mathbf{C}[x]/(x^4+1)=\mathbf{C}[x]/(x^4-i^2)$ as follows:

- Compute $\varphi(f)=(f_0+if_2)+(f_1+if_3)x$ and $\varphi'(f)=(f_0-if_2)+(f_1-if_3)x$, and similarly compute $\varphi(g)$ and $\varphi'(g)$.
- Recursively compute $2\varphi(f)\varphi(g)$ in $\mathbf{C}[x]/(x^2-i)$, and recursively compute $2\varphi'(f)\varphi'(g)$ in $\mathbf{C}[x]/(x^2+i)$.
- Recover $4fg$.

See Figure 2.

A straightforward induction shows that the total work to compute nfg, given $f,g\in\mathbf{C}[x]/(x^n-c)$, is $(3/2)n\lg n$ additions, $(3/2)n\lg n$ subtractions, $(3/2)n\lg n$ multiplications by various constants, and n more multiplications. The algorithm then computes fg with an additional n multiplications by the constant $1/n$.

Generalization. More generally, let A be a commutative ring in which 2 is invertible, let $n\geq 2$ be a power of 2, let c be an invertible element of A, and let ζ be an $(n/2)$nd root of -1 in A.

By exactly the same method as above, one can multiply two elements of the ring $A[x]/(x^n-c^n)$ with $(9/2)n\lg n+2n$ operations in A: specifically, $(3/2)n\lg n$ additions, $(3/2)n\lg n$ subtractions, $(3/2)n\lg n+n$ multiplications by constants, and n more multiplications. The constants are $1/n$ and products of powers of c and ζ.

The assumption that A has a primitive nth root of 1 is a heavy restriction on A. If $\mathbf{Z}/t$ has a primitive nth root of 1, for example, then every prime divisor of t is in $1+n\mathbf{Z}$. (This fact is a special case of Pocklington's primality test; see page XXX of this book.) Section 3 explains how to handle more general rings A.

Variant: radix 3. Similarly, let A be a commutative ring in which 3 is invertible, let $n \geq 3$ be a power of 3, let c be an invertible element of A, and let ζ be an element of A satisfying $1 + \zeta^{n/3} + \zeta^{2n/3} = 0$. Then one can multiply two elements of the ring $A[x]/(x^n - c^n)$ with $O(n \lg n)$ operations in A.

History. Gauss in [43, pages 265–327] was the first to point out that one can quickly compute a ring isomorphism from $\mathbf{R}[x]/(x^{2n} - 1)$ to $\mathbf{R}^2 \times \mathbf{C}^{n-1}$ when n has no large prime factors. In [43, pages 308–310], for example, Gauss (in completely different language) mapped $\mathbf{R}[x]/(x^{12} - 1)$ to $\mathbf{R}[x]/(x^3 - 1) \times \mathbf{R}[x]/(x^3 + 1) \times \mathbf{C}[x]/(x^3 + i)$, then mapped $\mathbf{R}[x]/(x^3 - 1)$ to $\mathbf{R} \times \mathbf{C}$, mapped $\mathbf{R}[x]/(x^3 + 1)$ to $\mathbf{R} \times \mathbf{C}$, and mapped $\mathbf{C}[x]/(x^3 + i)$ to $\mathbf{C} \times \mathbf{C} \times \mathbf{C}$.

The **discrete Fourier transform**—this isomorphism from $\mathbf{R}[x]/(x^{2n} - 1)$ to $\mathbf{R}^2 \times \mathbf{C}^{n-1}$, or the analogous isomorphism from $\mathbf{C}[x]/(x^n - 1)$ to $\mathbf{C}^n$—was applied to many areas of scientific computation over the next hundred years. Gauss's method was reinvented several times, as discussed in [49], and finally became widely known after it was reinvented and published by Cooley and Tukey in [36]. Gauss's method is now called the **fast Fourier transform** or simply the **FFT**.

Shortly after the Cooley-Tukey paper, Sande and Stockham pointed out that one can quickly multiply in $\mathbf{C}[x]/(x^n - 1)$ by applying the FFT, multiplying in $\mathbf{C}^n$, and applying the inverse FFT. See [91, page 229] and [44, page 573].

Fiduccia in [42] was the first to point out that *each step* of the FFT is an algebra isomorphism. This fact is still not widely known, despite its tremendous expository value; most expositions of the FFT use only the *module* structure of each step. I have taken Fiduccia's idea much further in [13] and in this paper, identifying the ring morphisms behind all known multiplication methods.

Improvements. The algorithm explained above takes $15n \lg n + 8n$ operations in $\mathbf{R}$ to multiply in $\mathbf{C}[x]/(x^n - 1)$, if $\mathbf{C}$ is represented as $\mathbf{R}[i]/(i^2 + 1)$ and $n \geq 2$ is a power of 2:

- $5n \lg n$ to transform the first input from $\mathbf{C}[x]/(x^n - 1)$ to $\mathbf{C}^n$. The FFT takes $n \lg n$ additions and subtractions in $\mathbf{C}$, totalling $2n \lg n$ operations in $\mathbf{R}$, and $(1/2)n \lg n$ multiplications by various roots of 1 in $\mathbf{C}$, totalling $3n \lg n$ operations in $\mathbf{R}$.
- $5n \lg n$ to transform the second input from $\mathbf{C}[x]/(x^n - 1)$ to $\mathbf{C}^n$.
- $2n$ to scale one of the transforms, i.e., to multiply by the constant $1/n$. One can eliminate most of these multiplications by absorbing $1/n$ into other constants.
- $6n$ to multiply the two transformed inputs in $\mathbf{C}^n$.
- $5n \lg n$ to transform the product from $\mathbf{C}^n$ back to $\mathbf{C}[x]/(x^n - 1)$.

One can reduce each $5n \lg n$ to $5n \lg n - 10n + 16$ for $n \geq 4$ by recognizing roots of 1 that allow easy multiplications: multiplications by 1 can be skipped, multiplications by -1 and $\pm i$ can be absorbed into subsequent computations, and multiplications by $\pm\sqrt{\pm i}$ are slightly easier than general multiplications.

Gentleman and Sande in [44] pointed out another algorithm to map $\mathbf{C}[x]/(x^n - 1)$ to $\mathbf{C}^n$ using $5n \lg n - 10n + 16$ operations: map $\mathbf{C}[x]/(x^{2n} - 1)$ to $\mathbf{C}[x]/(x^n - 1) \times \mathbf{C}[x]/(x^n + 1)$, twist $\mathbf{C}[x]/(x^n + 1)$ into $\mathbf{C}[x]/(x^n - 1)$ by mapping $x \to \zeta x$, and handle each $\mathbf{C}[x]/(x^n - 1)$ recursively. I call this method the **twisted FFT**.

The **split-radix FFT** is faster: it uses only $4n \lg n - 6n + 8$ operations for $n \ge 2$, so multiplication in $\mathbf{C}[x]/(x^n - 1)$ uses only $12n \lg n + O(n)$ operations. The split-radix FFT is a mixture of Gauss's FFT with the Gentleman-Sande twisted FFT: it maps $\mathbf{C}[x]/(x^{4n} - 1)$ to $\mathbf{C}[x]/(x^{2n} - 1) \times \mathbf{C}[x]/(x^{2n} + 1)$, maps $\mathbf{C}[x]/(x^{2n} + 1)$ to $\mathbf{C}[x]/(x^n - i) \times \mathbf{C}[x]/(x^n + i)$, twists each $\mathbf{C}[x]/(x^n \pm i)$ into $\mathbf{C}[x]/(x^n - 1)$ by mapping $x \to \zeta x$, and recursively handles both $\mathbf{C}[x]/(x^{2n} - 1)$ and $\mathbf{C}[x]/(x^n - 1)$.

Another method is the **real-factor FFT**: map $\mathbf{C}[x]/(x^{4n} - (2\cos 2\alpha)x^{2n} + 1)$ to $\mathbf{C}[x]/(x^{2n} - (2\cos\alpha)x^n + 1) \times \mathbf{C}[x]/(x^{2n} + (2\cos\alpha)x^n + 1)$, and handle each factor recursively. The real-factor FFT uses $4n \lg n + O(n)$ operations if one represents elements of $\mathbf{C}[x]/(x^{2n} \pm \cdots)$ using the basis $(1, x, \ldots, x^{n-1}, x^{-n}, x^{1-n}, \ldots, x^{-1})$.

It is difficult to assign credit for the bound $4n \lg n + O(n)$. Yavne announced the bound $4n \lg n - 6n + 8$ (specifically, $3n \lg n - 3n + 4$ additions and subtractions and $n \lg n - 3n + 4$ multiplications) in [106, page 117], and apparently had in mind a method achieving that bound; but nobody, to my knowledge, has ever deciphered Yavne's explanation of the method. Ten years later, Bruun in [28] published the real-factor FFT. Several years after that, Duhamel and Hollmann in [39], Martens in [72], Vetterli and Nussbaumer in [103], and Stasinski (according to [40, page 263]) independently discovered the split-radix FFT.

One can multiply in $\mathbf{R}[x]/(x^{2n} + 1)$ with $12n \lg n + O(n)$ operations in $\mathbf{R}$, if n is a power of 2: map $\mathbf{R}[x]/(x^{2n} + 1)$ to $\mathbf{C}[x]/(x^n - i)$, twist $\mathbf{C}[x]/(x^n - i)$ into $\mathbf{C}[x]/(x^n - 1)$, and apply the split-radix FFT. This is approximately twice as fast as mapping $\mathbf{R}[x]/(x^{2n}+1)$ to $\mathbf{C}[x]/(x^{2n}+1)$. An alternative is to use the real-factor FFT, with $\mathbf{R}$ in place of $\mathbf{C}$.

One can also multiply in $\mathbf{R}[x]/(x^{2n} - 1)$ with $12n \lg n + O(n)$ operations in $\mathbf{R}$, if n is a power of 2: map $\mathbf{R}[x]/(x^{2n} - 1)$ to $\mathbf{R}[x]/(x^n - 1) \times \mathbf{R}[x]/(x^n + 1)$; handle $\mathbf{R}[x]/(x^n - 1)$ by the same method recursively; handle $\mathbf{R}[x]/(x^n + 1)$ as above. This is approximately twice as fast as mapping $\mathbf{R}[x]/(x^{2n} - 1)$ to $\mathbf{C}[x]/(x^{2n} - 1)$. This speedup was announced by Bergland in [9], but it was already part of Gauss's FFT.

The general strategy of all of the above algorithms is to transform f, transform g, multiply the results, and then undo the transform to recover fg. There is some redundancy here if $f = g$: one can easily save a factor of $1.5 + o(1)$ by transforming f, squaring the result, and undoing the transform to recover f^2. (Of course, f^2 is much easier to compute if $2 = 0$ in A; this also saves time in Section 6.)

More generally, one can save the transform of each input, and reuse the transform in a subsequent multiplication if one knows (or observes) that the same input is showing up again. I call this technique **FFT caching**. FFT caching was announced in [37, Section 9], but it was already widely known; see, e.g., [76, Section 3.7].

Further savings are possible when one wants to compute a sum of products. Instead of undoing a transform to recover ab, undoing another transform to recover cd, and adding the results to obtain $ab + cd$, one can add first and then undo a single transform to recover $ab + cd$. I call this technique **FFT addition**.

There is much more to say about FFT performance, because there are much more sophisticated models of computation. Real computers have operation latency, memory latency, and instruction-decoding latency, for example; a serious analysis of constant factors takes these latencies into account. See [18] for further comments.

3. Product: extension

Input. Let A be a commutative ring in which 2 is invertible. Let $n \ge 1$ be a power of 2. The algorithm described in this section is given two elements f, g of the ring $A[x]/(x^n + 1)$.

An element of $A[x]/(x^n + 1)$ is, by convention, represented as a sequence of n elements of A: the sequence $(f_0, f_1, \ldots, f_{n-1})$ represents $f_0 + f_1 x + \cdots + f_{n-1} x^{n-1}$.

Output. This algorithm computes the product $fg \in A[x]/(x^n + 1)$.

Speed. This algorithm uses $O(n \lg n \lg\lg n)$ operations in A: more precisely, at most $(((9/2)\lg\lg(n/4) + 63/2)\lg(n/4) - 33/2)n$ operations in A if $n \ge 8$.

How it works. For $n \le 8$, use the definition of multiplication in $A[x]/(x^n + 1)$. This takes at most $n^2 + n(n-1) = (2n-1)n$ operations in A. If $n = 8$ then $\lg(n/4) = 1$ so $((9/2)\lg\lg(n/4) + 63/2)\lg(n/4) - 33/2 = 63/2 - 33/2 = 15 = 2n - 1$.

For $n \ge 16$, find the unique power m of 2 such that $m^2 \in \{2n, 4n\}$. Notice that $8 \le m < n$. Notice also that $\lg(n/4) - 1 \le 2\lg(m/4) \le \lg(n/4)$, so $\lg\lg(m/4) \le \lg\lg(n/4) - 1$ and $2\lg(2n/m) \le \lg(n/4) + 3$.

Define $B = A[x]/(x^m + 1)$. By induction, given $f, g \in B$, one can compute the product fg with at most $(((9/2)\lg\lg(m/4) + 63/2)\lg(m/4) - 33/2)m$ operations in A. One can also compute any of the following with m operations in A: the sum $f + g$; the difference $f - g$; the product cf, where c is a constant element of A; the product cf, where c is a constant power of x.

There is a $(2n/m)$th root of -1 in B, namely $x^{m^2/2n}$. Therefore one can use the algorithm explained in Section 2 to multiply quickly in $B[y]/(y^{2n/m} + 1)$—and, consequently, to multiply in $A[x, y]/(y^{2n/m} + 1)$ if each input has x-degree smaller than $m/2$. This takes $(9/2)(2n/m)\lg(2n/m) + 2n/m$ easy operations in B and $2n/m$ more multiplications in B.

Now, given $f, g \in A[x]/(x^n + 1)$, compute fg as follows. Consider the $A[x]$-algebra morphism $\varphi : A[x, y]/(y^{2n/m} + 1) \to A[x]/(x^n + 1)$ that takes y to $x^{m/2}$. Find $F \in A[x, y]/(y^{2n/m} + 1)$ such that F has x-degree smaller than $m/2$ and $\varphi(F) = f$; explicitly, $F = \sum_j \sum_{0 \le i < m/2} f_{i+(m/2)j} x^i y^j$ if $f = \sum f_i x^i$. Similarly construct G from g. Compute FG as explained above. Then compute $\varphi(FG) = fg$; this takes n additional operations in A.

One multiplication in $A[x]/(x^n + 1)$ has thus been split into

- $2n/m$ multiplications in B, i.e., at most $((9\lg\lg(m/4) + 63)\lg(m/4) - 33)n \le (((9/2)\lg\lg(n/4) + 27)\lg(n/4) - 33)n$ operations in A;
- $(9n/m)\lg(2n/m) + 2n/m \le ((9/2)\lg(n/4) + 31/2)n/m$ easy operations in B, i.e., at most $((9/2)\lg(n/4) + 31/2)n$ operations in A; and
- n additional operations in A.

The total is at most $(((9/2)\lg\lg(n/4) + 63/2)\lg(n/4) - 33/2)n$ as claimed.

For example, given $f = f_0 + f_1 x + \cdots + f_7 x^7$ and $g = g_0 + g_1 x + \cdots + g_7 x^7$ in $A[x]/(x^8 + 1)$, define $F = (f_0 + f_1 x) + (f_2 + f_3 x)y + (f_4 + f_5 x)y^2 + (f_6 + f_7 x)y^3$ and $G = (g_0 + g_1 x) + (g_2 + g_3 x)y + (g_4 + g_5 x)y^2 + (g_6 + g_7 x)y^3$ in $A[x, y]/(y^4 + 1)$. The product FG has the form

$$\begin{aligned}(h_0 + h_1 x + h_2 x^2) &+ (h_3 + h_4 x + h_5 x^2)y \\ &+ (h_6 + h_7 x + h_8 x^2)y^2 + (h_9 + h_{10} x + h_{11} x^2)y^3.\end{aligned}$$

$$\begin{tikzcd}
 & & & A[x]/(x^8+1) & & & \\
 & & & A[x,y]/(y^4+1) \arrow[u, "y\mapsto x^2"] \arrow[d] & & & \\
 & & & B[y]/(y^4+1) \arrow[dll, "y\mapsto y"] \arrow[drr, "y\mapsto y"'] & & & \\
 & B[y]/(y^2-x^2) \arrow[dl, "y\mapsto x"'] \arrow[dr, "y\mapsto -x"] & & & & B[y]/(y^2+x^2) \arrow[dl, "y\mapsto x^3"'] \arrow[dr, "y\mapsto -x^3"] & \\
B & & B & & B & & B
\end{tikzcd}$$

FIGURE 3. Splitting product in $A[x]/(x^8+1)$ into four products in $B = A[x]/(x^4+1)$, if 2 is invertible in A. Compare to Figure 2.

Compute this product in $A[x,y]/(x^4+1, y^4+1)$, and substitute $y = x^2$ to recover $fg = (h_0-h_{11})+h_1x+(h_2+h_3)x^2+h_4x^3+(h_5+h_6)x^4+h_7x^5+(h_8+h_9)x^6+h_{10}x^7$. The multiplication in $A[x,y]/(x^4+1, y^4+1)$ splits into four multiplications in $A[x]/(x^4+1)$. See Figure 3.

Variant: radix 3. Similarly, let A be a commutative ring in which 3 is invertible, and let $n \geq 3$ be a power of 3. One can multiply two elements of $A[x]/(x^{2n}+x^n+1)$ with $O(n \lg n \lg\lg n)$ operations in A.

The integer case; another model of computation. Algorithms that multiply polynomials of high degree using very few coefficient operations are analogous to algorithms that multiply integers with many bits in very little time.

There are many popular definitions of time. In this paper, **time** means number of steps on a multitape Turing machine. See [79, Section 2.3] for a precise definition of multitape Turing machines.

Let n be a power of 2. There is an algorithm, analogous to the multiplication algorithm for $A[x]/(x^n+1)$, that multiplies two elements of $\mathbf{Z}/(2^n+1)$ in time $O(n \lg n \lg\lg n)$. Here an element of $\mathbf{Z}/(2^n+1)$ is, by convention, represented as a sequence of $n+1$ bits: the sequence $(f_0, f_1, \ldots, f_n)$ represents $f_0 + 2f_1 + \cdots + 2^n f_n$. Note that most numbers have two representations.

The multiplication algorithm for $\mathbf{Z}/(2^n+1)$ performs $2n/m$ multiplications in $\mathbf{Z}/(2^m+1)$, for $n \geq 16$, where $m^2 \in \{2n, 4n\}$. Splitting a $\mathbf{Z}/(2^n+1)$ multiplication into $\mathbf{Z}/(2^m+1)$ multiplications is analogous to, but slightly more complicated than, splitting an $A[x]/(x^n+1)$ multiplication into $A[x]/(x^m+1)$ multiplications. The complication is that a sum of $2n/m$ products of $(m/2)$-bit integers generally does not quite fit into m bits. On the other hand, the sum does fit into $m+k$ bits for a small k, so it is determined by its images in $\mathbf{Z}/(2^m+1)$ and $\mathbf{Z}/2^k$. One multiplies in $\mathbf{Z}[y]/(y^{2n/m}+1)$ by multiplying recursively in $(\mathbf{Z}/(2^m+1))[y]/(y^{2n/m}+1)$ and multiplying straightforwardly in $(\mathbf{Z}/2^k)[y]/(y^{2n/m}+1)$.

History. The ideas in this section were developed first in the integer case. The crucial point is that one can multiply in $\mathbf{Z}[y]/(y^m \pm 1)$ by selecting t so that $\mathbf{Z}/t$ has an appropriate root of 1, mapping $\mathbf{Z}[y]/(y^m \pm 1)$ to $(\mathbf{Z}/t)[y]/(y^m \pm 1)$, and applying an FFT over $\mathbf{Z}/t$. This multiplication method was suggested by Pollard in [81], independently by Nicholson in [77, page 532], and independently by Schönhage and Strassen in [88]. Schönhage and Strassen suggested the choice $t = 2^m + 1$ and proved the $O(n \lg n \lg \lg n)$ time bound.

The $O(n \lg n \lg \lg n)$ operation bound in the polynomial case was published by Schönhage in [86], independently by Nussbaumer in [78], and then by Turk in [100, Section 2]. Nussbaumer's algorithm uses a different approach: it multiplies in $A[x,y]/(y^4 + 1)$, for example, by mapping to $(A[y]/(y^4 + 1))[x]/(x^4 - 1)$ and applying an FFT over $A[y]/(y^4 + 1)$.

Schönhage in [86] suggested using the radix-3 FFT to multiply polynomials over fields of characteristic 2.

Improvements. Multiplication by a constant power of x in $A[x]/(x^m + 1)$ is easier than the above analysis indicates: multiplications by 1 in A can be eliminated, and multiplications by -1 in A can be absorbed into subsequent computations. The total operation count drops from $(9/2 + o(1))n \lg n \lg \lg n$ to $(3 + o(1))n \lg n \lg \lg n$.

The constant 3 here is the best known. There is much more to say about the $o(1)$. See [13] for a survey of relevant techniques.

There is vastly more to say about the integer case, in part because Turing-machine time is a more complicated concept than algebraic complexity, and in part because real computers are more complicated than Turing machines. See [18] for upper bounds on the time needed for integer multiplication on real computers.

4. Product: zero-padding and localization

Input. Let A be a commutative ring. Let n be a positive integer. The algorithm in this section is given two elements f, g of the polynomial ring $A[x]$ such that $\deg fg < n$: e.g., such that n is the total number of coefficients in f and g.

An element of $A[x]$ is, by convention, represented as a finite sequence of elements of A: the sequence $(f_0, f_1, \ldots, f_{d-1})$ represents $f_0 + f_1x + \cdots + f_{d-1}x^{d-1}$.

Output. This algorithm computes the product $fg \in A[x]$.

Speed. The algorithm uses $O(n \lg n \lg\lg n)$ operations in A.

Equivalently: The algorithm uses at most $n\mu(n)$ operations in A, where $\mu : \mathbf{N} \to \mathbf{R}$ is a nondecreasing positive function with $\mu(n) \in O(\lg n \lg\lg n)$. The μ notation helps simplify the run-time analysis in subsequent sections of this paper.

Special case: how it works if $A = \mathbf{C}$. Given $f, g \in \mathbf{C}[x]$ such that $\deg fg < n$, one can compute fg by using the algorithm of Section 2 to compute $fg \bmod (x^m - 1)$ in $\mathbf{C}[x]/(x^m - 1)$; here m is the smallest power of 2 with $m \geq n$. This takes $O(m \lg m) = O(n \lg n)$ operations in $\mathbf{C}$.

For example, if $f = f_0 + f_1x + f_2x^2$ and $g = g_0 + g_1x + g_2x^2 + g_3x^3$, use the algorithm of Section 2 to multiply the elements $f_0 + f_1x + f_2x^2 + 0x^3 + 0x^4 + 0x^5 + 0x^6 + 0x^7$ and $g_0 + g_1x + g_2x^2 + g_3x^3 + 0x^4 + 0x^5 + 0x^6 + 0x^7$ of $\mathbf{C}[x]/(x^8 - 1)$, obtaining $h_0 + h_1x + h_2x^2 + h_3x^3 + h_4x^4 + h_5x^5 + 0x^6 + 0x^7$. Then $fg = h_0 + h_1x + h_2x^2 + h_3x^3 + h_4x^4 + h_5x^5$. Appending zeros to an input—for example, converting f_0, f_1, f_2 to $f_0, f_1, f_2, 0, 0, 0, 0, 0$—is called **zero-padding**.

In this special case $A = \mathbf{C}$, the previously mentioned bound $\mu(n) \in O(\lg n \lg\lg n)$ is unnecessarily pessimistic: one can take $\mu(n) \in O(\lg n)$. Subsequent sections of this paper use the bound $\mu(n) \in O(\lg n \lg\lg n)$, and are correspondingly pessimistic.

Similar comments apply to other rings A having appropriate roots of -1, and to nearby rings such as $\mathbf{R}$.

Intermediate generality: how it works if 2 is invertible in A. Let A be any commutative ring in which 2 is invertible. Given $f, g \in A[x]$ with $\deg fg < n$, one can compute fg by using the algorithm of Section 3 to compute $fg \bmod (x^m + 1)$ in $A[x]/(x^m + 1)$; here m is the smallest power of 2 with $m \geq n$. This takes $O(m \lg m \lg\lg m) = O(n \lg n \lg\lg n)$ operations in A.

Intermediate generality: how it works if 3 is invertible in A. Let A be any commutative ring in which 3 is invertible. The previous algorithm has a radix-3 variant that computes fg using $O(n \lg n \lg\lg n)$ operations in A.

Full generality: how it works for arbitrary rings. What if neither 2 nor 3 is invertible? Answer: Map A to the product of the localizations $2^{-\mathbf{N}}A$ and $3^{-\mathbf{N}}A$. This map is injective; 2 is invertible in $2^{-\mathbf{N}}A$; and 3 is invertible in $3^{-\mathbf{N}}A$.

In other words: Given polynomials f, g over any commutative ring A, use the technique of Section 3 to compute $2^j fg$ for some j; use the radix-3 variant to compute $3^k fg$ for some k; and then compute fg as a linear combination of $2^j fg$ and $3^k fg$. This takes $O(n \lg n \lg\lg n)$ operations in A if $\deg fg < n$.

Assume, for example, that $\deg fg < 8$. Compute $16fg$ by computing $16fg \bmod (x^8 - 1)$, and compute $9fg$ by computing $9fg \bmod (x^{18} + x^9 + 1)$; then $fg = 4(16fg) - 7(9fg)$. The numbers 16 and 9 here are the denominators produced by the algorithm of Section 3.

The integer case. An analogous algorithm computes the product of two integers in time $O(n \lg n \lg\lg n)$, if the output size is known to be at most n bits. (Given $f, g \in \mathbf{Z}$ with $|fg| < 2^n$, use the algorithm of Section 3 to compute $fg \bmod (2^m+1)$ in $\mathbf{Z}/(2^m+1)$; here m is the smallest power of 2 with $m \geq n+1$.)

Here an integer is, by convention, represented in **two's-complement notation**: a sequence of bits $(f_0, f_1, \ldots, f_{k-1}, f_k)$ represents $f_0 + 2f_1 + \cdots + 2^{k-1}f_{k-1} - 2^k f_k$.

History. Karatsuba in [55] was the first to point out that integer multiplication can be done in subquadratic time. This result is often (e.g., in [29, page 58]) credited to Karatsuba and Ofman, because [55] was written by Karatsuba and Ofman; but [55] explicitly credited the algorithm to Karatsuba alone.

Toom in [98] was the first to point out that integer multiplication can be done in essentially linear time: more precisely, time $n \exp(O(\sqrt{\log n}))$. Schönhage in [84] independently published the same observation a few years later. Cook in [35, page 53] commented that Toom's method could be used to quickly multiply polynomials over finite fields.

Stockham in [91, page 230] suggested zero-padding and FFT-based multiplication in $\mathbf{C}[x]/(x^n-1)$ as a way to multiply in $\mathbf{C}[x]$.

The $O(n \lg n \lg\lg n)$ time bound for integers is usually credited to Schönhage and Strassen; see Section 3. Cantor and Kaltofen in [32] used $A \to 2^{-\mathbf{N}}A \times 3^{-\mathbf{N}}A$ to prove the $O(n \lg n \lg\lg n)$ operation bound for polynomials over any ring.

Improvements. The above algorithms take

- $(m/n)(9/2 + o(1))n \lg n$ operations in $\mathbf{C}$ to multiply in $\mathbf{C}[x]$; or
- $(m/n)(12 + o(1))n \lg n$ operations in $\mathbf{R}$ to multiply in $\mathbf{C}[x]$, using the split-radix FFT or the real-factor FFT; or
- $(m/n)(6 + o(1))n \lg n$ operations in $\mathbf{R}$ to multiply in $\mathbf{R}[x]$; or
- $(m/n)(3 + o(1))n \lg n \lg\lg n$ operations in any ring A to multiply in $A[x]$, if 2 is invertible in A.

There are several ways to eliminate the m/n factor here. One good way is to compute fg modulo $x^m + 1$ for several powers m of 2 with $\sum m \geq n$, then recover fg. For example, if $n = 80000$, one can recover fg from $fg \bmod (x^{65536}+1)$ and $fg \bmod (x^{16384}+1)$. A special case of this technique was pointed out by Crandall and Fagin in [37, Section 7]. See [13, Section 8] for an older technique.

One can save time at the beginning of the FFT when the input is known to be the result of zero-padding. For example, one does not need an operation to compute $f_0 + 0$. Similarly, one can save time at the end of the FFT when the output is known to have zeros: the zeros need not be recomputed.

In the context of FFT addition—for example, computing $ab + cd$ with only five transforms—the transform size does not need to be large enough for ab and cd; it need only be large enough for $ab + cd$. This is useful in applications where $ab + cd$ is known to be small.

When f has a substantially larger degree than g (or vice versa), one can often save time by splitting f into pieces of comparable size to g, and multiplying each piece by g. Similar comments apply in Section 7. In the polynomial case, this technique is most often called the "overlap-add method"; it was introduced by Stockham in [91, page 230] under the name "sectioning." The analogous technique for integers appears in [62, answer to Exercise 4.3.3–13] with credit to Schönhage.

See [17] and [18] for further improvements, and [13] for a survey of techniques.

5. Product: completion

Input. Let A be a commutative ring. Let n be a positive integer. The algorithm in this section is given the precision-n representations of two elements f, g of the power-series ring $A[[x]]$.

The precision-n representation of a power series $f \in A[[x]]$ is, by definition, the polynomial $f \bmod x^n$. If $f = \sum_j f_j x^j$ then $f \bmod x^n = f_0 + f_1 x + \cdots + f_{n-1}x^{n-1}$. This polynomial is, in turn, represented in the usual way as its coefficient sequence $(f_0, f_1, \ldots, f_{n-1})$.

This representation does not carry complete information about f; it is only an approximation to f. It is nevertheless useful.

Output. This algorithm computes the precision-n representation of the product $fg \in A[[x]]$. If the input is $f_0, f_1, \ldots, f_{n-1}, g_0, g_1, \ldots, g_{n-1}$ then the output is $f_0g_0, f_0g_1 + f_1g_0, f_0g_2 + f_1g_1 + f_2g_0, \ldots, f_0g_{n-1} + f_1g_{n-2} + \cdots + f_{n-1}g_0$.

Speed. This algorithm uses $O(n \lg n \lg\lg n)$ operations in A: more precisely, at most $(2n-1)\mu(2n-1)$ operations in A.

How it works. Given $f \bmod x^n$ and $g \bmod x^n$, compute the polynomial product $(f \bmod x^n)(g \bmod x^n)$ by the algorithm of Section 4. Throw away the coefficients of $x^n, x^{n+1}, \ldots$ to obtain $(f \bmod x^n)(g \bmod x^n) \bmod x^n = fg \bmod x^n$.

For example, given the precision-3 representation f_0, f_1, f_2 of the series $f = f_0 + f_1x + f_2x^2 + \cdots$, and given the precision-3 representation g_0, g_1, g_2 of the series $g = g_0 + g_1x + g_2x^2 + \cdots$, first multiply $f_0 + f_1x + f_2x^2$ by $g_0 + g_1x + g_2x^2$ to obtain $f_0g_0 + (f_0g_1 + f_1g_0)x + (f_0g_2 + f_1g_1 + f_2g_0)x^2 + (f_1g_2 + f_2g_1)x^3 + f_2g_2x^4$; then throw away the coefficients of x^3 and x^4 to obtain $f_0g_0, f_0g_1 + f_1g_0, f_0g_2 + f_1g_1 + f_2g_0$.

The integer case, easy completion: $\mathbf{Q} \to \mathbf{Q}_2$. Consider the ring $\mathbf{Z}_2$ of 2-adic integers. The precision-n representation of $f \in \mathbf{Z}_2$ is, by definition, the integer $f \bmod 2^n \in \mathbf{Z}$. This representation of elements of $\mathbf{Z}_2$ as nearby elements of $\mathbf{Z}$ is analogous in many ways to the representation of elements of $A[[x]]$ as nearby elements of $A[x]$. In particular, there is an analogous multiplication algorithm: given $f \bmod 2^n$ and $g \bmod 2^n$, one can compute $fg \bmod 2^n$ in time $O(n \lg n \lg\lg n)$.

The integer case, hard completion: $\mathbf{Q} \to \mathbf{R}$. Each real number $f \in \mathbf{R}$ is, by convention, represented as a nearby element of the localization $2^{-\mathbf{N}}\mathbf{Z}$: an integer divided by a power of 2. If $|f| < 1$, for example, then there are one or two integers d with $|d| \le 2^n$ such that $|d/2^n - f| < 1/2^n$.

If another real number g with $|g| < 1$ is similarly represented by an integer e then fg is *almost* represented by the integer $\lfloor de/2^n \rfloor$, which can be computed in time $O(n \lg n \lg\lg n)$. However, the distance from fg to $\lfloor de/2^n \rfloor / 2^n$ may be somewhat larger than $1/2^n$. This effect is called **roundoff error**: the output is known to slightly less precision than the input.

History. See [62, Section 4.1] for the history of positional notation.

Improvements. Operations involved in multiplying $f \bmod x^n$ by $g \bmod x^n$ can be skipped if they are used only to compute the coefficients of $x^n, x^{n+1}, \ldots$. The number of operations skipped depends on the multiplication method; optimizing $u, v \mapsto uv$ does not necessarily optimize $u, v \mapsto uv \bmod x^n$. See [17] for further discussion. Similar comments apply to the integer case.

6. Reciprocal

Input. Let A be a commutative ring. Let n be a positive integer. The algorithm in this section is given the precision-n representation of a power series $f \in A[[x]]$ with $f(0) = 1$.

Output. This algorithm computes the precision-n representation of the reciprocal $1/f = 1 + (1-f) + (1-f)^2 + \cdots \in A[[x]]$. If the input is $1, f_1, f_2, f_3, \ldots, f_{n-1}$ then the output is $1, -f_1, f_1^2 - f_2, 2f_1f_2 - f_1^3 - f_3, \ldots, \cdots - f_{n-1}$.

Speed. This algorithm uses $O(n \lg n \lg\lg n)$ operations in A: more precisely, at most $(8n + 2k - 8)\mu(2n-1) + (2n + 2k - 2)$ operations in A if $n \le 2^k$.

How it works. If $n = 1$ then $(1/f) \bmod x^n = 1$. There are 0 operations here; and $(8n + 2k - 8)\mu(2n-1) + (2n + 2k - 2) = 2k\mu(1) + 2k \ge 0$ since $k \ge \lg n = 0$.

Otherwise define $m = \lceil n/2 \rceil$. Recursively compute $g_0 = (1/f) \bmod x^m$; note that $m < n$. Then compute $(1/f) \bmod x^n$ as $(g_0 - (fg_0 - 1)g_0) \bmod x^n$, using the algorithm of Section 5 for the multiplications by g_0. This works because the difference $1/f - (g_0 - (fg_0 - 1)g_0)$ is exactly $f(1/f - g_0)^2$, which is a multiple of x^{2m}, hence of x^n.

For example, given the precision-4 representation $1 + f_1x + f_2x^2 + f_3x^3$ of f, recursively compute $g_0 = (1/f) \bmod x^2 = 1 - f_1x$. Multiply f by g_0 modulo x^4 to obtain $1 + (f_2 - f_1^2)x^2 + (f_3 - f_1f_2)x^3$. Subtract 1 and multiply by g_0 modulo x^4 to obtain $(f_2 - f_1^2)x^2 + (f_3 + f_1^3 - 2f_1f_2)x^3$. Subtract from g_0 to obtain $1 - f_1x + (f_1^2 - f_2)x^2 + (2f_1f_2 - f_1^3 - f_3)x^3$. This is the precision-4 representation of $1/f$.

The proof of speed is straightforward. By induction, the recursive computation uses at most $(8m + 2(k-1) - 8)\mu(2m-1) + (2m + 2(k-1) - 2)$ operations in A, since $m \le 2^{k-1}$. The subtraction from g_0 and the subtraction of 1 use at most $n+1$ operations in A. The two multiplications by g_0 use at most $2(2n-1)\mu(2n-1)$ operations in A. Apply the inequalities $m \le (n+1)/2$ and $\mu(2m-1) \le \mu(2n-1)$ to see that the total is at most $(8n + 2k - 8)\mu(2n-1) + (2n + 2k - 2)$ as claimed.

The integer case, easy completion: $\mathbf{Q} \to \mathbf{Q}_2$. Let $f \in \mathbf{Z}_2$ be an odd 2-adic integer. Then f has a reciprocal $1/f = 1 + (1-f) + (1-f)^2 + \cdots \in \mathbf{Z}_2$.

One can compute $(1/f) \bmod 2^n$, given $f \bmod 2^n$, by applying the same formula as in the power-series case: first recursively compute $g_0 = (1/f) \bmod 2^{\lceil n/2 \rceil}$; then compute $(1/f) \bmod 2^n$ as $(g_0 + (1 - fg_0)g_0) \bmod 2^n$. This takes time $O(n \lg n \lg\lg n)$.

The integer case, hard completion: $\mathbf{Q} \to \mathbf{R}$. Let $f \in \mathbf{R}$ be a real number between 0.5 and 1. Then f has a reciprocal $g = 1 + (1-f) + (1-f)^2 + \cdots \in \mathbf{R}$. If g_0 is a close approximation to $1/f$, then $g_0 + (1 - fg_0)g_0$ is an approximation to $1/f$ with *nearly* twice the precision. Consequently one can compute a precision-n representation of $1/f$, given a slightly higher-precision representation of f, in time $O(n \lg n \lg\lg n)$.

The details are, thanks to roundoff error, more complicated than in the power-series case, and are not included in this paper. See [62, Algorithm 4.3.3–R] or [11, Section 8] for a complete algorithm.

History. The iteration $g \mapsto g - (fg - 1)g$ is an example of Newton's root-finding method, but according to various sources it was known for thousands of years before Newton. The fact that one can carry out the second-to-last iteration at about $1/2$ the desired precision, the third-to-last iteration at about $1/4$ the desired precision, etc., so that the total time is comparable to the time for the last iteration, was also known for thousands of years.

Cook in [35, pages 81–86] published details of a variable-precision reciprocal algorithm for $\mathbf{R}$ taking essentially linear time, using Toom's essentially-linear-time multiplication algorithm.

Sieveking in [89], apparently unaware of Cook's result, published details of an analogous reciprocal algorithm for $A[[x]]$. Kung in [69] pointed out that Sieveking's method was an example of Newton's method.

Improvements. Computing a reciprocal by the above algorithm takes $4 + o(1)$ times as many operations as computing a product. There are several ways that this constant 4 can be reduced. The following discussion focuses on $A[[x]]$; analogous comments apply to $\mathbf{Z}_2$ and $\mathbf{R}$. The ideas here and in Section 7 are variously due to Brent, Schönhage, Alan H. Karp, Peter Markstein, Robert Harley, Paul Zimmermann, and me; see [15] for historical notes and further details.

One can achieve $3 + o(1)$ as follows. By construction, $fg_0 - 1$ is divisible by x^m. To compute $((fg_0 - 1)g_0) \bmod x^n$, one can multiply $((fg_0 - 1) \bmod x^n)/x^m$ by $g_0 \bmod x^{n-m}$, and then multiply the result by x^m.

One can achieve $5/2 + o(1)$ by a similar observation. To compute $fg_0 \bmod x^n$, one multiplies $f \bmod x^n$ by $g_0 \bmod x^n$; but $\deg g_0 < m$ by construction, so this product actually has degree at most $n + m - 2$.

One can achieve $5/3 + o(1)$ by applying FFT addition to take advantage of the known structure of fg_0, and applying FFT caching to take advantage of the repetition of g_0. Assume, for example, that $A = \mathbf{C}$ and that $n \geq 2$ is a power of 2. The polynomial $((f \bmod x^n)g_0 - 1)/x^m$ has degree below n, so it is determined by its remainder modulo $x^n - 1$. One can transform $f \bmod x^n$, transform g_0, subtract the (easy) transform of 1, multiply by the (easy) transform of x^{n-m}, and untransform the result, to obtain $((f \bmod x^n)g_0 - 1)/x^m$. One can then reduce the result modulo x^{n-m} to obtain $((fg_0 - 1) \bmod x^n)/x^m$, and finally multiply by g_0, using the cached FFT of g_0. There are just five $\mathbf{C}[x]/(x^n - 1)$ transforms here, instead of the six $\mathbf{C}[x]/(x^{2n} - 1)$ transforms in the original algorithm.

The best result I know is $3/2 + o(1)$; see [15]. The point is that f and g_0 were already partially transformed as part of the computation of g_0; one can remove redundancy across several levels of recursion. I would not be surprised if further improvements are possible.

7. Quotient

Input. Let A be a commutative ring. Let n be a positive integer. The algorithm in this section is given the precision-n representations of power series $f, h \in A[[x]]$ such that $f(0) = 1$.

Output. This algorithm computes the precision-n representation of $h/f \in A[[x]]$. If the input is $1, f_1, f_2, \ldots, f_{n-1}, h_0, h_1, h_2, \ldots, h_{n-1}$ then the output is

$$\begin{aligned}
&h_0,\\
&h_1 - f_1 h_0,\\
&h_2 - f_1 h_1 + (f_1^2 - f_2) h_0,\\
&\vdots\\
&h_{n-1} - \cdots + (\cdots - f_{n-1}) h_0.
\end{aligned}$$

Speed. This algorithm uses $O(n \lg n \lg \lg n)$ operations in A: more precisely, at most $(10n + 2k - 9)\mu(2n-1) + (2n + 2k - 2)$ operations in A if $n \le 2^k$.

How it works. First compute a precision-n approximation to $1/f$ as explained in Section 6. Then multiply by h as explained in Section 5.

The integer case, easy completion: $\mathbf{Q} \to \mathbf{Q}_2$. Let h and f be elements of $\mathbf{Z}_2$ with f odd. Given $f \bmod 2^n$ and $h \bmod 2^n$, one can compute $(h/f) \bmod 2^n$ in time $O(n \lg n \lg \lg n)$ by the same method.

The integer case, hard completion: $\mathbf{Q} \to \mathbf{R}$. Let h and f be elements of $\mathbf{R}$ with $0.5 \le f \le 1$. One can compute a precision-n representation of h/f, given slightly higher-precision representations of f and h, in time $O(n \lg n \lg \lg n)$. As usual, roundoff error complicates the algorithm.

Improvements. One can improve the number of operations for a reciprocal to $3/2 + o(1)$ times the number of operations for a product, as discussed in Section 6, so one can improve the number of operations for a quotient to $5/2 + o(1)$ times the number of operations for a product.

The reader may be wondering at this point why quotient deserves to be discussed separately from reciprocal. Answer: There is no reason to believe that a quotient is as difficult as a reciprocal followed by a separate product. In fact, one can improve the number of operations for a quotient to $13/6 + o(1)$ times the number of operations for a product. See [15] for details and historical notes.

8. Logarithm: the series case

Input. Let A be a commutative ring containing $\mathbf{Q}$. Let n be a positive integer. The algorithm in this section is given the precision-n representation of a power series $f \in A[[x]]$ with $f(0) = 1$.

Output. This algorithm computes the precision-n representation of the series $\log f = -(1-f)-(1-f)^2/2-(1-f)^3/3-\cdots \in A[[x]]$. If the input is $1, f_1, f_2, f_3, \ldots$ then the output is $0, f_1, f_2 - f_1^2/2, f_3 - f_1 f_2 + f_1^3/3, \ldots$.

Define $D(\sum a_j x^j) = \sum j a_j x^j$. The reader may enjoy checking the following properties of log and D:

- $D(fg) = gD(f) + fD(g)$;
- $D(g^n) = ng^{n-1}D(g)$;
- if $f(0) = 1$ then $D(\log f) = D(f)/f$;
- if $f(0) = 1$ and $\log f = 0$ then $f = 1$;
- if $f(0) = 1$ and $g(0) = 1$ then $\log fg = \log f + \log g$;
- log is injective: i.e., if $f(0) = 1$ and $g(0) = 1$ and $\log f = \log g$ then $f = g$.

Speed. This algorithm uses $O(n \lg n \lg \lg n)$ operations in A: more precisely, at most $(10n + 2k - 9)\mu(2n - 1) + (4n + 2k - 4)$ operations in A if $n \le 2^k$.

How it works. Given $f \bmod x^n$, compute $D(f) \bmod x^n$ from the definition of D; compute $(D(f)/f) \bmod x^n$ as explained in Section 7; and recover $(\log f) \bmod x^n$ from the formula $D((\log f) \bmod x^n) = (D(f)/f) \bmod x^n$.

The integer case. This $A[[x]]$ algorithm does not have a useful analogue for $\mathbf{Z}_2$ or $\mathbf{R}$, because $\mathbf{Z}_2$ and $\mathbf{R}$ do not have adequate replacements for the differential operator D. See, however, Section 16.

History. This algorithm was published by Brent in [25, Section 13].

Improvements. See Section 7 for improved quotient algorithms. I do not know any way to compute $\log f$ more quickly than computing a generic quotient.

9. Exponential: the series case

Input. Let A be a commutative ring containing $\mathbf{Q}$. Let n be a positive integer. The algorithm in this section is given the precision-n representation of a power series $f \in A[[x]]$ with $f(0) = 0$.

Output. This algorithm computes the precision-n representation of the series $\exp f = 1 + f + f^2/2! + f^3/3! + \cdots \in A[[x]]$. If the input is $0, f_1, f_2, f_3, \ldots$ then the output is $1, f_1, f_2 + f_1^2/2, f_3 + f_1 f_2 + f_1^3/6, \ldots$.

The reader may enjoy checking the following properties of exp:

- if $f(0) = 0$ then $D(\exp f) = D(f) \exp f$;
- if $f(0) = 0$ then $\log \exp f = f$;
- if $g(0) = 1$ then $\exp \log g = g$;
- if $f(0) = 0$ and $g(0) = 0$ then $\exp(f + g) = (\exp f) \exp g$.

Speed. This algorithm uses $O(n \lg n \lg \lg n)$ operations in A: more precisely, at most $(24n + k^2 + 3k - 24)\mu(2n - 1) + (12n + k^2 + 3k - 12)$ operations in A if $n \le 2^k$.

How it works. If $n = 1$ then $(\exp f) \bmod x^n = 1$. Otherwise define $m = \lceil n/2 \rceil$. Recursively compute $g_0 = (\exp f) \bmod x^m$. Compute $(\log g_0) \bmod x^n$ as explained in Section 8. Then compute $(\exp f) \bmod x^n$ as $(g_0 + (f - \log g_0)g_0) \bmod x^n$. This works because $\exp(f - \log g_0) - 1 - (f - \log g_0)$ is a multiple of $(f - \log g_0)^2$, hence of x^{2m}, hence of x^n.

The recursive step uses at most $(24(n+1)/2 + (k-1)^2 + 3(k-1) - 24)\mu(2n-1) + (12(n+1)/2 + (k-1)^2 + 3(k-1) - 12)$ operations by induction. The computation of $(\log g_0) \bmod x^n$ uses at most $(10n + 2k - 9)\mu(2n - 1) + (4n + 2k - 4)$ operations. The subtraction from f and the addition of g_0 use at most $2n$ operations. The multiplication by g_0 uses at most $(2n-1)\mu(2n-1)$ operations. The total is at most $(24n + k^2 + 3k - 24)\mu(2n - 1) + (12n + k^2 + 3k - 12)$ as claimed.

The integer case. See Section 16.

History. The iteration $g \mapsto g + (f - \log g)g$ is an example of Newton's method. Brent in [25, Section 13] pointed out that this is a particularly efficient way to compute exp for $\mathbf{R}[[x]]$.

Improvements. Brent in [25, Section 13] stated that the number of operations for an exponential in $\mathbf{R}[[x]]$ could be improved to $22/3 + o(1)$ times the number of operations for a product. In fact, one can achieve $17/6 + o(1)$. See [15].

10. Power: the series case

Input. Let A be a commutative ring containing $\mathbf{Q}$. Let n be a positive integer. The algorithm in this section is given the precision-n representations of power series $f, e \in A[[x]]$ such that $f(0) = 1$.

Output. This algorithm computes the precision-n representation of the series $f^e = \exp(e \log f) \in A[[x]]$. If the input is $1, f_1, f_2, \ldots, e_0, e_1, \ldots$ then the output is $1, e_0 f_1, e_1 f_1 + e_0 f_2 + e_0(e_0 - 1) f_1^2/2, \ldots$.

The reader may enjoy checking the following properties of $f, e \mapsto f^e$:

- $f^0 = 1$;
- $f^1 = f$;
- $f^{d+e} = f^d \cdot f^e$, so the notation f^e for $\exp(e \log f)$ is, for positive integers e, consistent with the usual notation f^e for $\prod_{1 \le j \le e} f$;
- $f^{-1} = 1/f$;
- $(f^d)^e = f^{de}$;
- $(fg)^e = f^e g^e$;
- $D(f^e) = D(e) f^e \log f + D(f) e f^{e-1}$;
- $f^e = 1 + e(f-1) + (e(e-1)/2)(f-1)^2 + \cdots$.

Speed. This algorithm uses $O(n \lg n \lg\lg n)$ operations in A: more precisely, at most $(36n + k^2 + 5k - 34)\mu(2n-1) + (16n + k^2 + 5k - 16)$ operations in A if $n \le 2^k$.

How it works. Given $f \bmod x^n$, compute $(\log f) \bmod x^n$ as explained in Section 8; compute $(e \log f) \bmod x^n$ as explained in Section 5; compute $(\exp(e \log f)) \bmod x^n$ as explained in Section 9.

The integer case. See Section 16.

History. According to various sources, Napier introduced the functions exp and log for $\mathbf{R}$, along with the idea of using exp and log to compute products in $\mathbf{R}$. I do not know the history of exp and log for $\mathbf{Z}_2$ and $A[[x]]$.

Improvements. As in Sections 6, 7, and 9, one can remove some redundancy from the above algorithm. See [15].

Brauer in [22] pointed out that, if e is a positive integer, one can compute f^e with about $\lg e$ squarings and at most about $(\lg e)/\lg\lg e$ other multiplications. This is faster than the exp-log algorithm if e is small. See [16] for further discussion of square-and-multiply exponentiation algorithms.

One can compute f^e for any rational number e with a generalization of the algorithm of Section 6. This takes essentially linear time for fixed e, as pointed out by Cook in [35, page 86]; it is faster than the exp-log algorithm if the height of e is small, i.e., the numerator and denominator of e are small. The special case $e = 1/2$—i.e., square roots—is discussed in detail in [15].

11. Matrix product

Input. Let A be a commutative ring. The algorithm in this section is given two 2×2 matrices $F = \begin{pmatrix} F_{11} & F_{12} \\ F_{21} & F_{22} \end{pmatrix}$ and $G = \begin{pmatrix} G_{11} & G_{12} \\ G_{21} & G_{22} \end{pmatrix}$ with entries in the polynomial ring $A[x]$.

Output. This algorithm computes the 2×2 matrix product FG.

Speed. This algorithm uses $O(n \lg n \lg\lg n)$ operations in A, where n is the total number of input coefficients. More precisely, the algorithm uses at most $n(2\mu(n)+2)$ operations in A. This bound is pessimistic.

Here, and elsewhere in this paper, **number of coefficients** means the number of elements of A provided as input. Reader beware: the number of coefficients of an input polynomial is not determined by the polynomial; it depends on how the polynomial is represented. For example, the sequence $(5, 7, 0)$, with 3 coefficients, represents the same polynomial as the sequence $(5, 7)$, with 2 coefficients.

How it works. Multiply F_{11} by G_{11}, multiply F_{12} by G_{21}, add, etc., to obtain $FG = \begin{pmatrix} F_{11}G_{11} + F_{12}G_{21} & F_{11}G_{12} + F_{12}G_{22} \\ F_{21}G_{11} + F_{22}G_{21} & F_{21}G_{12} + F_{22}G_{22} \end{pmatrix}$.

The integer case. An analogous algorithm computes the product of two 2×2 matrices with entries in $\mathbf{Z}$ in time $O(n \lg n \lg\lg n)$, where n is the number of input bits.

History. The matrix concept is generally credited to Sylvester and Cayley.

Improvements. The above algorithm involves 24 transforms. FFT caching—transforming each of the input polynomials $F_{11}, F_{12}, F_{21}, F_{22}, G_{11}, G_{12}, G_{21}, G_{22}$ just once—saves 8 transforms. FFT addition—untransforming $F_{11}G_{11} + F_{12}G_{21}$, for example, rather than separately untransforming $F_{11}G_{11}$ and $F_{12}G_{21}$—saves 4 more transforms.

Strassen in [93] published a method to multiply 2×2 matrices using just 7 multiplications of entries and 18 additions or subtractions of entries, rather than 8 multiplications and 4 additions. Winograd observed that 18 could be replaced by 15; see [62, page 500].

Many applications involve matrices of particular shapes: for example, matrices F in which $F_{12} = 0$. One can often save time accordingly.

Generalization: larger matrices. Strassen in [93] published a general method to multiply $d \times d$ matrices using $O(d^\alpha)$ multiplications, additions, and subtractions of entries; here $\alpha = \log_2 7 = 2.807\ldots$. Subsequent work by Pan, Bini, Capovani, Lotti, Romani, Schönhage, Coppersmith, and Winograd showed that there is an algorithm to multiply $d \times d$ matrices using $d^{\beta+o(1)}$ multiplications and additions of entries, for a certain number $\beta < 2.38$. See [29, Chapter 15] for a detailed exposition and further references.

It is not known whether matrix multiplication can be carried out in essentially linear time, when the matrix size is a variable.

12. Product tree

Input. Let A be a commutative ring. Let t be a nonnegative integer. The algorithm in this section is given 2×2 matrices $M_1, M_2, \ldots, M_t$ with entries in $A[x]$.

Output. This algorithm computes the **product tree** of $M_1, M_2, \ldots, M_t$, which is defined as follows. The root of the tree is the 2×2 matrix $M_1 M_2 \cdots M_t$. If $t \leq 1$, that's the complete tree. If $t \geq 2$, the left subtree is the product tree of $M_1, M_2, \ldots, M_s$, and the right subtree is the product tree of $M_{s+1}, M_{s+2}, \ldots, M_t$, where $s = \lceil t/2 \rceil$.

For example, Figure 4 shows the product tree of $M_1, M_2, M_3, M_4, M_5, M_6$.

Most applications use only the root $M_1 M_2 \cdots M_t$ of the product tree. This root is often described in the language of linear recurrences as follows. Define $X_i = M_1 M_2 \cdots M_i$; then $X_i = X_{i-1} M_i$, i.e., $X_{i,j,k} = X_{i-1,j,0} M_{i,0,k} + X_{i-1,j,1} M_{i,1,k}$. The algorithm computes $X_{t,0,0}, X_{t,0,1}, X_{t,1,0}, X_{t,1,1}$, given the coefficients $M_{i,j,k}$ of the linear recurrence $X_{i,j,k} = X_{i-1,j,0} M_{i,0,k} + X_{i-1,j,1} M_{i,1,k}$, with the starting condition $(X_{0,0,0}, X_{0,0,1}, X_{0,1,0}, X_{0,1,1}) = (1, 0, 0, 1)$.

Speed. This algorithm uses $O(n(\lg n)^2 \lg\lg n)$ operations in A, where n is the total number of coefficients in $M_1, M_2, \ldots, M_t$: more precisely, at most $nk(2\mu(n) + 2)$ operations in A if k is a nonnegative integer and $t \leq 2^k$.

How it works. If $t = 0$ then the answer is the identity matrix. If $t = 1$ then the answer is M_1. Otherwise recursively compute the product tree of $M_1, M_2, \ldots, M_s$ and the product tree of $M_{s+1}, M_{s+2}, \ldots, M_t$, where $s = \lceil t/2 \rceil$. Multiply the roots $M_1 M_2 \cdots M_s$ and $M_{s+1} \cdots M_t$, as discussed in Section 11, to obtain $M_1 M_2 \cdots M_t$.

Time analysis: Define m as the total number of coefficients in $M_1, M_2, \ldots, M_s$. By induction, the computation of the product tree of $M_1, M_2, \ldots, M_s$ uses at most $m(k-1)(2\mu(n) + 2)$ operations, and the computation of the product tree of $M_{s+1}, \ldots, M_t$ uses at most $(n-m)(k-1)(2\mu(n)+2)$ operations. The final multiplication uses at most $n(2\mu(n)+2)$ operations. Add: $m(k-1)+(n-m)(k-1)+n = nk$.

The integer case. An analogous algorithm takes time $O(n(\lg n)^2 \lg\lg n)$ to compute the product tree of a sequence $M_1, M_2, \ldots, M_t$ of 2×2 matrices with entries in $\mathbf{Z}$. Here n is the total number of input bits.

Generalization: larger matrices. One can use the same method to compute a product of several $d \times d$ matrices—in other words, to compute terms in linear recurrences of any order. It is not known whether this can be done in essentially linear time for variable d; see Section 11 for further comments.

$$\begin{tikzcd}
 & & & & M_1M_2M_3M_4M_5M_6 & & & \\
 & & M_1M_2M_3 \arrow[urr] & & & & M_4M_5M_6 \arrow[ull] & \\
 & M_1M_2 \arrow[ur] & & M_3 \arrow[ul] & & M_4M_5 \arrow[ur] & & M_6 \arrow[ul] \\
M_1 \arrow[ur] & & M_2 \arrow[ul] & & M_4 \arrow[ur] & & M_5 \arrow[ul] &
\end{tikzcd}$$

FIGURE 4. Product tree of $M_1, M_2, M_3, M_4, M_5, M_6$.

History. Product trees are so simple and so widely applicable that they have been reinvented many times. They are not only a basic tool in the context of fast multiplication but also a basic tool for building low-depth parallel algorithms.

Unfortunately, most authors state product trees for particular applications, with no hint of the generality of the technique. They define ad-hoc product operations, and prove associativity of their product operations from scratch, never realizing that these operations are special cases of matrix product.

Weinberger and Smith in [105] published the "carry-lookahead adder," a low-depth parallel circuit for computing the sum of two nonnegative integers, with the inputs and output represented in the usual way as bit strings. The Weinberger-Smith algorithm computes (in different language) a product

$$\begin{pmatrix} a_1 & 0 \\ b_1 & 1 \end{pmatrix} \begin{pmatrix} a_2 & 0 \\ b_2 & 1 \end{pmatrix} \cdots \begin{pmatrix} a_t & 0 \\ b_t & 1 \end{pmatrix} = \begin{pmatrix} a_1 a_2 \cdots a_t & 0 \\ b_t + b_{t-1}a_t + b_{t-2}a_{t-1}a_t + \cdots + b_1 a_2 \cdots a_t & 1 \end{pmatrix}$$

of matrices over the Boole algebra $\{0, 1\}$ by multiplying pairs of matrices in parallel, then multiplying pairs of pairs in parallel, and so on for approximately $\lg t$ steps.

Estrin in [41] published a low-depth parallel algorithm for evaluating a one-variable polynomial. Estrin's algorithm computes (in different language) a product

$$\begin{pmatrix} a & 0 \\ b_1 & 1 \end{pmatrix} \begin{pmatrix} a & 0 \\ b_2 & 1 \end{pmatrix} \cdots \begin{pmatrix} a & 0 \\ b_t & 1 \end{pmatrix} = \begin{pmatrix} a^t & 0 \\ b_t + b_{t-1}a + b_{t-2}a^2 + \cdots + b_1 a^{t-1} & 1 \end{pmatrix}$$

by multiplying pairs, pairs of pairs, etc.

Schönhage, as reported in [59, Exercise 4.4–13], pointed out that one can convert integers from base 10 to base 2 in essentially linear time. Schönhage's algorithm computes (in different language) a product

$$\begin{pmatrix} 10 & 0 \\ b_1 & 1 \end{pmatrix} \begin{pmatrix} 10 & 0 \\ b_2 & 1 \end{pmatrix} \cdots \begin{pmatrix} 10 & 0 \\ b_t & 1 \end{pmatrix} = \begin{pmatrix} 10^t & 0 \\ b_t + 10b_{t-1} + 100b_{t-2} + \cdots + 10^{t-1}b_1 & 1 \end{pmatrix}$$

by multiplying pairs of matrices, then pairs of pairs, etc.

Knuth in [60, Theorem 1] published an algorithm to convert a continued fraction to a fraction in essentially linear time. Knuth's algorithm is (in different language) another example of the product-tree algorithm; see Section 14 for details.

Moenck and Borodin in [74, page 91] and [19, page 372] pointed out that one can compute the product tree—and thus the product—of a sequence of polynomials or a sequence of integers in essentially linear time. Beware that the Moenck-Borodin "theorems" assume that all of the inputs are "single precision"; it is unclear what this is supposed to mean for integers.

Moenck and Borodin also pointed out an algorithm to add fractions in essentially linear time. This algorithm is (in different language) yet another example of the product-tree algorithm. See Section 13 for details and further historical notes.

Meanwhile, in the context of parallel algorithms, Stone and Kogge published the product-tree algorithm in a reasonable level of generality in [92], [66], and [65], with polynomial evaluation and continued-fraction-to-fraction conversion ("tridiagonal-linear-system solution") as examples. Stone commented that linear recurrences of any order could be phrased as matrix products—see [92, page 34] and [92, page 37]—but, unfortunately, made little use of matrices elsewhere in his presentation.

Kogge and Stone in [66, page 792] credited Robert Downs, Harvard Lomax, and H. R. G. Trout for independent discoveries of general product-tree algorithms. They also stated that special cases of the algorithm were "known to J. J. Sylvester as early as 1853"; but I see no evidence that Sylvester ever formed a product tree in that context or any other context. Sylvester in [97] (cited in [60] and [92]) simply pointed out the associativity of continued fractions.

Brent in [23, Section 6] pointed out that the numerator and denominator of $1 + 1/2 + 1/3! + \cdots + 1/t! \approx \exp 1$ could be computed quickly. Brent's algorithm formed (in different language) a product tree for

$$\begin{pmatrix}1 & 0\\ 1 & 1\end{pmatrix}\begin{pmatrix}2 & 0\\ 1 & 1\end{pmatrix}\cdots\begin{pmatrix}t & 0\\ 1 & 1\end{pmatrix} = \begin{pmatrix}t! & 0\\ t! + t!/2 + \cdots + t(t-1) + t + 1 & 1\end{pmatrix}.$$

Brent also addressed exp for more general inputs, as discussed in Section 15; and π, via arctan. Brent described his method as a mixed-radix adaptation of Schönhage's base-conversion algorithm. Evidently he had in mind the product

$$\begin{pmatrix}a_1 & 0\\ b_1 & c\end{pmatrix}\begin{pmatrix}a_2 & 0\\ b_2 & c\end{pmatrix}\cdots\begin{pmatrix}a_t & 0\\ b_t & c\end{pmatrix} = \begin{pmatrix}a_1a_2\cdots a_t & 0\\ c^{t-1}b_t + c^{t-2}b_{t-1}a_t + \cdots + b_1a_2\cdots a_t & c^t\end{pmatrix}$$

corresponding to the sum $\sum_{1\le k\le t} c^{k-1}b_k/a_1\cdots a_k$. Brent and McMillan mentioned in [27, page 308] that the sum $\sum_{1\le k\le t} n^k(-1)^{k-1}/k!k$ could be handled similarly.

I gave a reasonably general statement of the product-tree algorithm in [10], with a few series and continued fractions as examples. I pointed out that computing $M_1M_2\cdots M_t$ takes time $O(t(\lg t)^3 \lg\lg t)$ in the common case that the entries of M_j are bounded by polynomials in j.

Gosper presented a wide range of illustrative examples of matrix products in [45], emphasizing their "notational, analytic, and computational virtues." Gosper briefly stated the product-tree algorithm in [45, page 263], and credited it to Rich Schroeppel.

Chudnovsky and Chudnovsky in [33, pages 115–118] stated the product-tree algorithm for matrices M_j whose entries depend rationally on j. They gave a broad class of series as examples in [33, pages 123–134]. They called the algorithm "a well-known method to accelerate the (numerical) solution of linear recurrences."

Karatsuba used product trees (in different language) to evaluate various sums in several papers starting in 1991 and culminating in [56].

See [47], [101], [21, Section 7], and [102] for further uses of product trees to evaluate sums.

Improvements. One can change s in the definition of a product tree and in the product-tree algorithm. The choice $s = \lceil t/2 \rceil$, balancing s against $t - s$, is not necessarily the fastest way to compute $M_1M_2\cdots M_t$: when $M_1, M_2, \ldots, M_t$ have widely varying degrees, it is much better to balance $\deg M_1 + \deg M_2 + \cdots + \deg M_s$ against $\deg M_{s+1} + \deg M_{s+2} + \cdots + \deg M_t$. Strassen proved in [96, Theorem 3.2] that a slightly more complicated strategy is within a constant factor of optimal.

One can further generalize the definition of a product tree by allowing vertices to have more than two children. For example, there are many ways to obtain $M_1M_2M_3$ from M_1, M_2, M_3 that do not involve first computing M_1M_2 or M_2M_3.

In some applications, $M_1, M_2, \ldots, M_t$ are known to commute. One can often permute $M_1, M_2, \ldots, M_t$ for slightly higher speed. Strassen in [96, Theorem 2.2] pointed out a particularly fast, and pleasantly simple, algorithm: find the two matrices of smallest degree, replace them by their product, and repeat. See [29, Section 2.3] for an exposition.

13. Sum of fractions

Input. Let A be a commutative ring. Let t be a positive integer. The algorithm in this section is given $2t$ polynomials $f_1, g_1, f_2, g_2, \ldots, f_t, g_t \in A[x]$.

Output. This algorithm computes $h = f_1 g_2 \cdots g_t + g_1 f_2 \cdots g_t + \cdots + g_1 g_2 \cdots f_t$, along with $g_1 g_2 \cdots g_t$.

The reader may think of this output as follows: the algorithm computes the sum $h/g_1 g_2 \cdots g_t$ of the fractions $f_1/g_1, f_2/g_2, \ldots, f_t/g_t$. The equation $h/g_1 g_2 \cdots g_t = f_1/g_1 + f_2/g_2 + \cdots + f_t/g_t$ holds in any $A[x]$-algebra where $g_1, g_2, \ldots, g_t$ are invertible: in particular, in the localization $g_1^{-\mathbf{N}} g_2^{-\mathbf{N}} \ldots g_t^{-\mathbf{N}} A[x]$.

Speed. This algorithm uses $O(n(\lg n)^2 \lg\lg n)$ operations in A, where n is the total number of coefficients in the input polynomials.

How it works. The matrix product $\begin{pmatrix} g_1 & f_1 \\ 0 & g_1 \end{pmatrix} \begin{pmatrix} g_2 & f_2 \\ 0 & g_2 \end{pmatrix} \ldots \begin{pmatrix} g_t & f_t \\ 0 & g_t \end{pmatrix}$ is exactly $\begin{pmatrix} g_1 g_2 \cdots g_t & h \\ 0 & g_1 g_2 \cdots g_t \end{pmatrix}$. Compute this product as described in Section 12.

The point is that adding fractions a/b and c/d to obtain $(ad+bc)/bd$ is the same as multiplying matrices $\begin{pmatrix} b & a \\ 0 & b \end{pmatrix}$ and $\begin{pmatrix} d & c \\ 0 & d \end{pmatrix}$ to obtain $\begin{pmatrix} bd & ad+bc \\ 0 & bd \end{pmatrix}$.

Alternate proof, using the language of recurrences: the quantities $p_j = g_1 g_2 \cdots g_j$ and $q_j = (f_1/g_1 + \cdots + f_j/g_j)p_j$ satisfy the recurrences $p_j = p_{j-1} g_j$ and $q_j = q_{j-1} g_j + p_{j-1} f_j$, i.e., $\begin{pmatrix} p_j & q_j \\ 0 & p_j \end{pmatrix} = \begin{pmatrix} p_{j-1} & q_{j-1} \\ 0 & p_{j-1} \end{pmatrix} \begin{pmatrix} g_j & f_j \\ 0 & g_j \end{pmatrix}$.

The reader may prefer to describe this algorithm without matrices: for $t \ge 2$, recursively compute $f_1/g_1 + \cdots + f_s/g_s$ and $f_{s+1}/g_{s+1} + \cdots + f_t/g_t$, and then add to obtain $f_1/g_1 + \cdots + f_t/g_t$. Here $s = \lceil t/2 \rceil$.

The integer case. An analogous algorithm, given integers $f_1, g_1, f_2, g_2, \ldots, f_t, g_t$, computes $f_1 g_2 \ldots g_t + g_1 f_2 \ldots g_t + \cdots + g_1 g_2 \ldots f_t$. It takes time $O(n(\lg n)^2 \lg\lg n)$, where n is the total number of input bits.

History. Horowitz in [52] published an algorithm to compute the polynomial

$$\left(\frac{b_1}{x-a_1} + \frac{b_2}{x-a_2} + \cdots + \frac{b_t}{x-a_t}\right)(x-a_1)(x-a_2)\cdots(x-a_t)$$

within a $\lg t$ factor of the time for polynomial multiplication. Horowitz's algorithm is essentially the algorithm described above, but it splits t into $t/2, t/4, t/8, \ldots$ rather than $t/2, t/2$.

Borodin and Moenck in [19, Section 7] published a more general algorithm to add fractions, in both the polynomial case and the integer case, for the application described in Section 23.

Improvements. See Section 12 for improved product-tree algorithms.

14. Fraction from continued fraction

Input. Let A be a commutative ring. Let t be a nonnegative integer. The algorithm in this section is given t polynomials $q_1, q_2, \ldots, q_t \in A[x]$ such that, for each i, at least one of q_i, q_{i+1} is nonzero.

Output. This algorithm computes the polynomials $F(q_1, q_2, \ldots, q_t) \in A[x]$ and $G(q_1, q_2, \ldots, q_t) \in A[x]$ defined recursively by $F() = 1$, $G() = 0$, $F(q_1, q_2, \ldots, q_t) = q_1 F(q_2, \ldots, q_t) + G(q_2, \ldots, q_t)$ for $t \geq 1$, and $G(q_1, q_2, \ldots, q_t) = F(q_2, \ldots, q_t)$ for $t \geq 1$.

For example, $F(q_1, q_2, q_3, q_4) = q_1 q_2 q_3 q_4 + q_1 q_2 + q_1 q_4 + q_3 q_4 + 1$. In general, $F(q_1, q_2, \ldots, q_t)$ is the sum of all products of subsequences of $(q_1, q_2, \ldots, q_t)$ obtained by deleting any number of non-overlapping adjacent pairs.

The reader may think of this output as the numerator and denominator of a continued fraction:

$$\frac{F(q_1, q_2, \ldots, q_t)}{G(q_1, q_2, \ldots, q_t)} = q_1 + \cfrac{1}{\cfrac{F(q_2, \ldots, q_t)}{G(q_2, \ldots, q_t)}} = q_1 + \cfrac{1}{q_2 + \cfrac{1}{\ddots + \cfrac{1}{q_t}}}.$$

As in Section 13, these equations hold in any $A[x]$-algebra where all the divisions make sense.

Speed. This algorithm uses $O(n(\lg n)^2 \lg\lg n)$ operations in A, where n is the total number of coefficients in the input polynomials.

How it works. The matrix product $\begin{pmatrix} q_1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} q_2 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} q_3 & 1 \\ 1 & 0 \end{pmatrix} \cdots \begin{pmatrix} q_t & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ is exactly $\begin{pmatrix} F(q_1, q_2, \ldots, q_t) \\ G(q_1, q_2, \ldots, q_t) \end{pmatrix}$ by definition of F and G. Compute this product as described in Section 12.

The assumption that no two consecutive q's are 0 ensures that the total number of coefficients in these matrices is in $O(n)$.

The integer case. An analogous algorithm, given integers $q_1, q_2, \ldots, q_t$, computes $F(q_1, q_2, \ldots, q_t)$ and $G(q_1, q_2, \ldots, q_t)$. It takes time $O(n(\lg n)^2 \lg\lg n)$, where n is the total number of input bits.

History. See Section 12.

Improvements. See Section 12 for improved product-tree algorithms.

15. Exponential: the short case

Input. Let A be a commutative ring containing $\mathbf{Q}$. Let m and n be positive integers. The algorithm in this section is given a polynomial $f \in A[x]$ with $\deg f < 2m$ and $f \bmod x^m = 0$. For example, if $m = 2$, the input is a polynomial of the form $f_2x^2 + f_3x^3$.

Output. This algorithm computes the precision-n representation of the series $\exp f \in A[[x]]$ defined in Section 9.

Speed. This algorithm uses $O(n(\lg n)^2 \lg\lg n)$ operations in A. It is usually slower than the algorithm of Section 9; its main virtue is that the same idea also works for $\mathbf{Z}_2$ and $\mathbf{R}$.

How it works. Define $k = \lceil n/m - 1 \rceil$. Compute the matrix product $\begin{pmatrix} u & v \\ 0 & w \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} f & f \\ 0 & 1 \end{pmatrix} \begin{pmatrix} f & f \\ 0 & 2 \end{pmatrix} \cdots \begin{pmatrix} f & f \\ 0 & k \end{pmatrix}$ as described in Section 12. Then $(\exp f) \bmod x^n = (v/w) \bmod x^n$. Note that w is simply the integer $k!$, so the division by w is a multiplication by the constant $1/k!$.

The point is that $(u, v, w) = (f^k, k!(1 + f + f^2/2 + \cdots + f^k/k!), k!)$ by induction, so $(\exp f) - v/w = f^{k+1}/(k+1)! + f^{k+2}/(k+2)! + \cdots$; but k was chosen so that f^{k+1} is divisible by x^n.

The integer case, easy completion: $\mathbf{Q} \to \mathbf{Q}_2$. One can use the same method to compute a precision-n representation of $\exp f \in \mathbf{Z}_2$, given an integer $f \in \{0, 2^m, (2)2^m, \ldots, (2^m - 1)2^m\}$, in time $O(n(\lg n)^2 \lg\lg n)$, for $m \geq 2$. Note that k must be chosen somewhat larger in this case, because the final division of v by $w = k!$ loses approximately k bits of precision.

The integer case, hard completion: $\mathbf{Q} \to \mathbf{R}$. One can compute a precision-n representation of $\exp f$, given a real number f such that $|f| < 2^{-m}$ and f is a multiple of 2^{-2m}, in time $O(n(\lg n)^2 \lg\lg n)$. As usual, roundoff error complicates the algorithm.

History. See Section 12.

Improvements. See Section 16.

16. Exponential: the general case

Input. Let A be a commutative ring containing $\mathbf{Q}$. Let n be a positive integer. The algorithm in this section is given the precision-n representation of a power series $f \in A[[x]]$ with $f(0) = 0$.

Output. This algorithm computes the precision-n representation of the series $\exp f \in A[[x]]$ defined in Section 9.

Speed. This algorithm uses $O(n(\lg n)^3 \lg\lg n)$ operations in A. It is usually much slower than the algorithm of Section 9; its main virtue is that the same idea also works for $\mathbf{Z}_2$ and $\mathbf{R}$.

How it works. Write f as a sum $f_1 + f_2 + f_4 + f_8 + \cdots$ where $f_m \bmod x^m = 0$ and $\deg f_m < 2m$. In other words, put the coefficient of x^1 into f_1; the coefficients of x^2 and x^3 into f_2; the coefficients of x^4 through x^7 into f_4; and so on.

Compute precision-n approximations to $\exp f_1, \exp f_2, \exp f_4, \ldots$ as described in Section 15. Multiply to obtain $\exp f$.

The integer case. Similar algorithms work for $\mathbf{Z}_2$ and $\mathbf{R}$.

History. This method of computing exp is due to Brent. See [23, Theorem 6.2]. Brent also pointed out that, starting from a fast algorithm for exp, one can use Newton's method to quickly compute log and various other functions.

Improvements. There is a completely different method of computing log and exp for $\mathbf{R}$, using the "arithmetic-geometric mean." This method takes time only $O(n(\lg n)^2 \lg\lg n)$. See [20] for a detailed exposition. This application of the arithmetic-geometric mean was introduced by Salamin and independently by Brent; see [8, Item 143], [83], [24], and [25, Section 9].

17. Quotient and remainder

Input. Let A be a commutative ring. Let d and e be nonnegative integers. The algorithm in this section is given two elements f, h of the polynomial ring $A[x]$ such that f is monic, $\deg f = d$, and $\deg h < e$.

Output. This algorithm computes $q, r \in A[x]$ such that $h = qf + r$ and $\deg r < d$. In other words, this algorithm computes $r = h \bmod f$ and $q = (h - r)/f$.

For example, say $d = 2$ and $e = 5$. Given $f = f_0 + f_1 x + x^2$ and $h = h_0 + h_1 x + h_2 x^2 + h_3 x^3 + h_4 x^4$, this algorithm computes

$$q = (h_2 - h_3 f_1 + h_4(f_1^2 - f_0)) + (h_3 - h_4 f_1)x + h_4 x^2$$

and

$$\begin{aligned} r = h \bmod f &= h - qf \\ &= (h_0 - h_2 f_0 + h_3 f_1 f_0 + h_4(f_1^2 - f_0)f_0) \\ &\quad + (h_1 - h_2 f_1 + h_3(f_1^2 - f_0) + h_4((f_1^2 - f_0)f_1 + f_1 f_0))x. \end{aligned}$$

Speed. This algorithm uses $O(e \lg e \lg\lg e)$ operations in A.

More precisely, the algorithm uses at most $(10(e-d) + 2k - 9)\mu(2(e-d) - 1) + (2(e-d) + 2k - 2) + e\mu(e) + e$ operations in A if $1 \le e - d \le 2^k$. The algorithm uses no operations if $e \le d$.

For simplicity, subsequent sections of this paper use the relatively crude upper bound $12(e+1)(\mu(2e) + 1)$.

How it works: $A(x) \to A((x^{-1}))$. The point is that polynomial division in $A[x]$ is division in $A((x^{-1}))$; $A((x^{-1}))$, in turn, is isomorphic to $A((x))$.

If $e \le d$, the answer is $q = 0$ and $r = h$. Assume from now on that $e > d$.

Reverse the coefficient order in $f = \sum_j f_j x^j$ to obtain $F = \sum_j f_{d-j} x^j \in A[x]$; in other words, define $F = x^d f(x^{-1})$. Then $\deg F \le d$ and $F(0) = 1$. For example, if $d = 2$ and $f = f_0 + f_1 x + x^2$, then $F = 1 + f_1 x + f_0 x^2$.

Similarly, reverse $h = \sum_j h_j x^j$ to obtain $H = \sum_j h_{e-1-j} x^j \in A[x]$; in other words, define $H = x^{e-1} h(x^{-1})$. Then $\deg H < e$. For example, if $e = 5$ and $h = h_0 + h_1 x + h_2 x^2 + h_3 x^3 + h_4 x^4$, then $H = h_4 + h_3 x + h_2 x^2 + h_1 x^3 + h_0 x^4$.

Now compute $Q = (H/F) \bmod x^{e-d}$ as explained in Section 7. Then $\deg Q < e - d$. Reverse $Q = \sum_j q_{e-d-1-j} x^j$ to obtain $q = \sum_j q_j x^j \in A[x]$; in other words, define $q = x^{e-d-1} Q(x^{-1})$.

Compute $r = h - qf \in A[x]$ as explained in Section 4. Then $\deg r < d$. Indeed, $x^{e-1} r(x^{-1}) = H - QF$ is a multiple of x^{e-d} by construction of Q.

The x-adic case: $A(x) \to A((x))$. Omit the reversal of coefficients in the above algorithm. The resulting algorithm, given two polynomials f, h with $f(0) = 1$, $\deg f \le d$, and $\deg h < e$, computes polynomials q, r such that $h = qf + x^{\max\{e-d,0\}} r$ and $\deg r < d$.

The integer case, easy completion: $\mathbf{Q} \to \mathbf{Q}_2$. An analogous algorithm, given integers f, h with f odd, $|f| \le 2^d$, and $|h| < 2^e$, computes integers q, r such that $h = qf + 2^{\max\{e-d,0\}} r$ and $|r| < 2^d$. The algorithm takes time $O(n \lg n \lg\lg n)$, where n is the total number of input bits.

The integer case, hard completion: $\mathbf{Q} \to \mathbf{R}$. An analogous algorithm, given integers f, h with $f \neq 0$, computes integers q, r such that $h = qf + r$ and $0 \leq r < |f|$. The algorithm takes time $O(n \lg n \lg \lg n)$, where n is the total number of input bits.

It is often convenient to change the sign of r when f is negative; in other words, to replace $0 \leq r < |f|$ with $0 \leq r/f < 1$; in other words, to take $q = \lfloor h/f \rfloor$. The time remains $O(n \lg n \lg \lg n)$.

History. See Section 7 for historical notes on fast division in $A[[x]]$ and $\mathbf{R}$.

The use of $x \mapsto x^{-1}$ for computing quotients dates back to at least 1973: Strassen commented in [94, page 240] (translated) that "the division of two formal power series can easily be used for the division of two polynomials with remainder." I have not attempted to trace the earlier history of the x^{-1} valuation.

Improvements. Another way to divide h by f is to recursively divide the top half of h by the top half of f, then recursively divide what's left. Moenck and Borodin in [74] published this algorithm (in the polynomial case), and observed that it takes time $O(n(\lg n)^2 \lg \lg n)$. Unfortunately, Borodin and Moenck omitted the algorithm from [19], apparently believing that the extra $\lg n$ factor removed it from competition. This belief has not been adequately investigated. Many years later, integer versions of the Moenck-Borodin algorithm were independently discovered by Jebelean in [54], by Daniel Ford (according to email I received from John Cannon in June 1998), and by Burnikel and Ziegler in [30].

One can often save some time, particularly in the integer case, by changing the problem, allowing a slightly wider range of remainders. Most applications do not need the smallest possible remainder of h modulo f; any reasonably small remainder is adequate.

Many applications of division in $\mathbf{R}$ can work equally well with division in $\mathbf{Z}_2$. This fact—widely known to number theorists since Hensel's introduction of $\mathbf{Z}_2$ (and more general completions) in the early 1900s—has frequently been applied to computations; replacing $\mathbf{R}$ with $\mathbf{Z}_2$ usually saves a little time and a considerable amount of effort. See, e.g., [67], [48], [46], [38] (using $\mathbf{Z}_p$ where, for simplicity, p is chosen to not divide an input), and [75]. Often $\mathbf{Z}_2$ division is called "Montgomery reduction," but this gives far too much credit to [75].

In some applications, one knows in advance that a division will be exact, i.e., that the remainder will be zero. Jebelean in [53] suggested computing the top half of the quotient with division in $\mathbf{R}$, and the bottom half of the quotient with division in $\mathbf{Z}_2$. These two half-size computations are faster than one full-size computation, because computation speed is not exactly linear. Similarly, for polynomials, one can combine x-adic division with the usual division.

18. Remainder tree

Input. Let A be a commutative ring. Let t be a nonnegative integer. The algorithm in this section is given a polynomial $h \in A[x]$ and monic polynomials $f_1, f_2, \ldots, f_t \in A[x]$.

Output. This algorithm computes $h \bmod f_1, h \bmod f_2, \ldots, h \bmod f_t$. Actually, the algorithm computes more: the **remainder tree** of $h, f_1, f_2, \ldots, f_t$.

The remainder tree is defined as follows: for each vertex v in the product tree of $f_1, f_2, \ldots, f_t$, there is a corresponding vertex $h \bmod v$ in the remainder tree of $h, f_1, f_2, \ldots, f_t$. In particular, the leaves of the product tree are $f_1, f_2, \ldots, f_t$, so the leaves of the remainder tree are $h \bmod f_1, h \bmod f_2, \ldots, h \bmod f_t$.

For example, Figure 5 shows the remainder tree of $h, f_1, f_2, f_3, f_4, f_5, f_6$.

In other words: The root of the remainder tree is $h \bmod f_1 f_2 \ldots f_t$. That's the complete tree if $t \le 1$. Otherwise the left subtree is the remainder tree of $h, f_1, f_2, \ldots, f_s$, and the right subtree is the remainder tree of $h, f_{s+1}, f_{s+2}, \ldots, f_t$, where $s = \lceil t/2 \rceil$.

Speed. This algorithm uses $O(n(\lg n)^2 \lg\lg n)$ operations in A, where n is the total number of coefficients in $h, f_1, f_2, \ldots, f_t$.

More precisely: Assume that d, m, k are nonnegative integers, that $\deg h < m$, that $f_1, f_2, \ldots, f_t$ together have at most d coefficients, and that $t \le 2^k$. Then the algorithm uses at most $(12m + 26dk + 24 \cdot 2^k - 12)(\mu(2\max\{d, m\}) + 1)$ operations in A.

How it works: $A(x) \to A((x^{-1}))$. Here is a recursive algorithm that, given h and the product tree P of $f_1, \ldots, f_t$, computes the remainder tree R of $h, f_1, f_2, \ldots, f_t$. This algorithm uses at most $12(m + 2dk + 2^{k+1} - 1)(\mu(2\max\{d, m\}) + 1)$ operations in A; add $2dk(\mu(2d)+1)$ operations in A to compute P in the first place as explained in Section 12.

The root of P is $f_1 f_2 \cdots f_t$. Compute $g = h \bmod f_1 f_2 \cdots f_t$ as explained in Section 17; this is the root of R. This uses at most $12(m + 1)(\mu(2m) + 1) \le 12(m + 1)\mu(2\max\{d, m\}) + 1$ operations in A.

The strategy is to compute each remaining vertex in R by reducing its parent vertex modulo the corresponding vertex in P. For example, the algorithm computes $h \bmod f_1 f_2 \cdots f_s$ as $(h \bmod f_1 f_2 \cdots f_t) \bmod f_1 f_2 \cdots f_s$.

If $t \le 1$, stop. There are no operations here; and $k \ge 0$ so $12(m + 1) \le 12(m + 2dk + 2^{k+1} - 1)$.

Otherwise define $s = \lceil t/2 \rceil$. Apply this algorithm recursively to g and the left subtree of P to compute the remainder tree of $g, f_1, f_2, \ldots, f_s$, which is exactly the left subtree of R. Apply this algorithm recursively to g and the right subtree of P to compute the remainder tree of $g, f_{s+1}, f_{s+2}, \ldots, f_t$, which is exactly the right subtree of R.

Time analysis: Define c as the number of coefficients of $f_1, \ldots, f_s$. By induction, the recursion for the left subtree uses at most $12(d + 2c(k - 1) + 2^k - 1)$ times $\mu(2d) + 1 \le \mu(2\max\{d, m\}) + 1$ operations, since $\deg g < d$. The recursion for the right subtree uses at most $12(d + 2(d - c)(k - 1) + 2^k - 1)$ times $\mu(2\max\{d, m\}) + 1$ operations. Add: $m + 1 + d + 2c(k - 1) + 2^k - 1 + d + 2(d - c)(k - 1) + 2^k - 1 = m + 2dk + 2^{k+1} - 1$.

$$\begin{tikzcd}
 & & & & h \bmod f_1f_2f_3f_4f_5f_6 \arrow[dll] \arrow[drr] & & & \\
 & & h \bmod f_1f_2f_3 \arrow[dl] \arrow[dr] & & & & h \bmod f_4f_5f_6 \arrow[dl] \arrow[dr] & \\
 & h \bmod f_1f_2 \arrow[dl] \arrow[dr] & & h \bmod f_3 & & h \bmod f_4f_5 \arrow[dl] \arrow[dr] & & h \bmod f_6 \\
h \bmod f_1 & & h \bmod f_2 & & h \bmod f_4 & & h \bmod f_5 &
\end{tikzcd}$$

FIGURE 5. Remainder tree of $h, f_1, f_2, f_3, f_4, f_5, f_6$.

The x-adic case: $A(x) \to A((x))$. One obtains a simpler algorithm by omitting the reversals described in Section 17. The simpler algorithm, given polynomials $h, f_1, f_2, \ldots, f_t$ where $f_1(0) = f_2(0) = \cdots = f_t(0) = 1$, computes small polynomials $r_1, r_2, \ldots, r_t$ such that h is congruent modulo f_j to a certain power of x times r_j. The algorithm uses $O(n(\lg n)^2 \lg\lg n)$ operations in A.

The integer case, easy completion: $\mathbf{Q} \to \mathbf{Q}_2$. An analogous algorithm, given an integer h and odd integers $f_1, f_2, \ldots, f_t$, computes small integers $r_1, r_2, \ldots, r_t$ such that h is congruent modulo f_j to a certain power of 2 times r_j. The algorithm takes time $O(n(\lg n)^2 \lg\lg n)$, where n is the total number of input bits.

The integer case, hard completion: $\mathbf{Q} \to \mathbf{R}$. An analogous algorithm, given an integer h and nonzero integers $f_1, f_2, \ldots, f_t$, computes integers $r_1, r_2, \ldots, r_t$, with $0 \le r_j < |f_j|$, such that h is congruent modulo f_j to r_j. The algorithm takes time $O(n(\lg n)^2 \lg\lg n)$, where n is the total number of input bits.

History. This algorithm was published by Borodin and Moenck in [74] and [19, Sections 4–6] for "single-precision" moduli $f_1, f_2, \ldots, f_t$.

Improvements. The computation of reciprocals, as described in Section 6, can be sped up inside this application. Consider, for example, dividing by f_1f_2, then by f_1, then by f_2. Section 6 computes approximate reciprocals of f_1, f_2, f_1f_2 with separate Newton iterations, starting from 1. It is better to start the Newton iteration for f_1f_2 with the product of the approximate reciprocals of f_1 and f_2. I mentioned this idea in [14].

19. Small factors of a product

Input. Let A be a commutative ring. Let s be a positive integer, and let t be a nonnegative integer. The algorithm in this section is given polynomials $h_1, h_2, \ldots, h_s \in A[x]$ and monic polynomials $f_1, f_2, \ldots, f_t \in A[x]$.

Output. This algorithm figures out which f_i's divide $h_1 h_2 \cdots h_s$: it computes the subsequence $g_1, g_2, \ldots$ of $f_1, \ldots, f_t$ consisting of each f_i that divides $h_1 h_2 \cdots h_s$.

The name "small factors" comes from the following important special case. Let A be a finite field, and let $f_1, f_2, \ldots$ be all the small primes in $A[x]$, i.e., all the low-degree monic irreducible polynomials in $A[x]$. Then this algorithm computes the small factors of $h_1 h_2 \cdots h_s$.

For example, say $A = \mathbf{Z}/2$, $s = 4$, $t = 5$, $h_1 = 101111 = 1 + x^2 + x^3 + x^4 + x^5$, $h_2 = 1101011$, $h_3 = 00001011$, $h_4 = 0001111$, $f_1 = 01$, $f_2 = 11$, $f_3 = 111$, $f_4 = 1101$, and $f_5 = 1011$. This algorithm finds all the factors of $h_1 h_2 h_3 h_4$ among f_1, f_2, f_3, f_4, f_5. Its output is $(01, 11, 111, 1011)$: the product $h_1 h_2 h_3 h_4$ is divisible by 01, 11, 111, and 1011, but not by 1101.

Speed. This algorithm uses $O(n(\lg n)^2 \lg\lg n)$ operations in A, where n is the total number of coefficients in $h_1, h_2, \ldots, h_s, f_1, f_2, \ldots, f_t$.

More precisely: Assume that d, m, j, k are nonnegative integers, that $h_1, \ldots, h_s$ together have at most m coefficients, that $f_1, \ldots, f_t$ together have at most d coefficients, that $s \le 2^j$, and that $t \le 2^k$. Then the algorithm uses at most $(2mj + 12m + 26dk + 24 \cdot 2^k - 12)(\mu(2\max\{d, m\}) + 1) + d$ operations in A.

How it works. Compute $h = h_1 h_2 \cdots h_s$ as explained in Section 12. This uses at most $2mj(\mu(m) + 1) \le 2mj(\mu(2\max\{d, m\}) + 1)$ operations in A.

Compute $h \bmod f_1, h \bmod f_2, \ldots, h \bmod f_t$ as explained in Section 18. This uses at most $(12m + 26dk + 24 \cdot 2^k - 12)(\mu(2\max\{d, m\}) + 1)$ operations in A.

Check whether $h \bmod f_1 = 0$, $h \bmod f_2 = 0$, $\ldots$, $h \bmod f_t = 0$. This uses at most d equality tests in A.

The integer case. An analogous algorithm, given integers $h_1, h_2, \ldots, h_s$ and given nonzero integers $f_1, f_2, \ldots, f_t$, figures out which f_i's divide $h_1 h_2 \cdots h_s$. The algorithm takes time $O(n(\lg n)^2 \lg\lg n)$, where n is the total number of input bits.

History. See Section 20. This algorithm is a stepping-stone to the algorithm of Section 20.

Improvements. See Section 12 for improved product-tree algorithms, and Section 18 for improved remainder-tree algorithms.

As discussed in Section 17, Jebelean in [53] combined $\mathbf{Z}_2$ and $\mathbf{R}$ to compute a quotient more quickly when the remainder was known in advance to be 0. A similar technique can be used to check more quickly whether a remainder is 0.

20. Small factors of a sequence

Input. Let A be a commutative ring. Let s be a positive integer, and let t be a nonnegative integer. The algorithm in this section is given nonzero polynomials $h_1, h_2, \ldots, h_s \in A[x]$ and monic coprime polynomials $f_1, f_2, \ldots, f_t \in A[x]$ with $\deg f_i \geq 1$ for each i.

Here **coprime** means that $A[x] = f_i A[x] + f_j A[x]$ for every i, j with $i \neq j$; in other words, there exist $u, v \in A[x]$ with $f_i u + f_j v = 1$. (Beware that many authors instead say **pairwise coprime**, and reserve "coprime" for the less important notion that $A[x] = f_1 A[x] + f_2 A[x] + \cdots + f_t A[x]$.)

The importance of coprimality is the **Chinese remainder theorem**: the $A[x]$-algebra morphism from $A[x]/f_1 f_2 \cdots f_t$ to $A[x]/f_1 \times A[x]/f_2 \times \cdots \times A[x]/f_t$ is an isomorphism. In particular, if each of $f_1, f_2, \ldots, f_t$ divides a polynomial h, then the product $f_1 f_2 \cdots f_t$ divides h. This is crucial for the speed of this algorithm.

Output. This algorithm figures out which f_i's divide h_1, which f_i's divide h_2, which f_i's divide h_3, etc.

As in Section 19, the name "small factors" comes from the important special case that A is a finite field and $f_1, f_2, \ldots$ are all of the small primes in $A[x]$. Then this algorithm computes the small factors of h_1, the small factors of h_2, etc.

For example, say $A = \mathbf{Z}/2$, $s = 4$, $t = 5$, $h_1 = 101111 = 1 + x^2 + x^3 + x^4 + x^5$, $h_2 = 1101011$, $h_3 = 00001011$, $h_4 = 0001111$, $f_1 = 01$, $f_2 = 11$, $f_3 = 111$, $f_4 = 1101$, and $f_5 = 1011$. This algorithm finds all the factors of h_1, h_2, h_3, h_4 among f_1, f_2, f_3, f_4, f_5. Its output is $(), (111), (01, 1011), (01, 11)$.

Speed. This algorithm uses $O(n(\lg n)^3 \lg\lg n)$ operations in A, where n is the total number of coefficients in $h_1, h_2, \ldots, h_s, f_1, f_2, \ldots, f_t$.

More precisely: Assume, as in Section 19, that d, m, j, k are nonnegative integers, that $h_1, \ldots, h_s$ together have at most m coefficients, that $f_1, \ldots, f_t$ together have at most d coefficients, that $s \leq 2^j$, and that $t \leq 2^k$. Then the algorithm uses at most $((104jk + j^2 + 109j + 12)m + 26dk + 24 \cdot 2^k)(\mu(2\max\{d, m\}) + 1) + d + 4mj$ operations in A.

How it works. Figure out which of $f_1, f_2, \ldots, f_t$ divide $h_1 h_2 \cdots h_s$, as explained in Section 19. Write $(g_1, g_2, \ldots)$ for this subsequence of $(f_1, f_2, \ldots, f_t)$. This uses at most $(2mj + 12m + 26dk + 24 \cdot 2^k)(\mu(2\max\{d, m\}) + 1) + d$ operations in A, leaving $(104jk + j^2 + 107j)m(\mu(2\max\{d, m\}) + 1) + 4mj$ operations for the remaining steps in the algorithm.

If $s = 1$, the answer is $(g_1, g_2, \ldots)$. There are no further operations in this case, and $(104jk + j^2 + 107j)m(\mu(2\max\{d, m\}) + 1) + 4mj$ is nonnegative.

Otherwise apply the algorithm recursively to $h_1, h_2, \ldots, h_r$ and $g_1, g_2, \ldots$, and then apply the algorithm recursively to $h_{r+1}, h_{r+2}, \ldots, h_s$ and $g_1, g_2, \ldots$, where $r = \lceil s/2 \rceil$. This works because any f's that divide h_i also divide $h_1 h_2 \cdots h_s$ and are therefore included among the g's.

The central point in the time analysis is that $\deg(g_1 g_2 \cdots) < m$. Indeed, $g_1, g_2, \ldots$ are coprime divisors of $h_1 \cdots h_s$, so their product is a divisor of $h_1 \cdots h_s$; but $h_1 \cdots h_s$ is a nonzero polynomial of degree smaller than m. Thus there are at most $\min\{m, t\}$ polynomials in $g_1, g_2, \ldots$, and the total number of coefficients in $g_1, g_2, \ldots$ is at most $\min\{2m, d\}$.

Define ℓ as the number of coefficients in $h_1, h_2, \ldots, h_r$, and define e as the smallest nonnegative integer with $\min\{m,t\} \le 2^e$. Then $\ell \le m$; $e \le k$; and $2^e \le 2m$, since $m \ge 1$. By induction, the recursive computation for $h_1, \ldots, h_r, g_1, g_2, \ldots$ uses at most

$$\begin{aligned}&((104(j-1)e + (j-1)^2 + 109(j-1) + 12)\ell + 26\min\{2m,d\}e + 24\cdot 2^e)\\ &\qquad(\mu(2\max\{\min\{2m,d\},\ell\}) + 1) + \min\{2m,d\} + 4\ell(j-1)\\ &\le ((104(j-1)k + (j-1)^2 + 109(j-1) + 12)\ell + 52mk + 48m)\\ &\qquad(\mu(2\max\{d,m\}) + 1) + 2m + 4\ell(j-1)\end{aligned}$$

operations in A. Similarly, the recursive computation for $h_{r+1}, \ldots, h_s, g_1, g_2, \ldots$ uses at most

$$\begin{aligned}&((104(j-1)k + (j-1)^2 + 109(j-1) + 12)(m-\ell) + 52mk + 48m)\\ &\qquad(\mu(2\max\{d,m\}) + 1) + 2m + 4(m-\ell)(j-1)\end{aligned}$$

operations in A. The total is exactly $(104jk + j^2 + 107j)m(\mu(2\max\{d,m\}) + 1) + 4mj$ as desired.

Here is an example of how the algorithm works. To factor $h_1 = 101111$, $h_2 = 1101011$, $h_3 = 00001011$, and $h_4 = 0001111$ over $A = \mathbf{Z}/2$ using the primes $01, 11, 111, 1101, 1011$, the algorithm first finds the factors $01, 11, 111, 1011$ of $h_1h_2h_3h_4$ as explained in Section 19. It then recursively factors h_1, h_2 using $01, 11, 111, 1011$, and recursively factors h_3, h_4 using $01, 11, 111, 1011$.

At the first level of recursion in the same example: To factor h_3, h_4 using $01, 11, 111, 1011$, the algorithm finds the factors $01, 11, 1011$ of h_3h_4 as explained in Section 19. It then recursively factors h_3 using $01, 11, 1011$, and recursively factors h_4 using $01, 11, 1011$.

The integer case. An analogous algorithm, given nonzero integers $h_1, h_2, \ldots, h_s$ and coprime integers $f_1, f_2, \ldots, f_t \ge 2$, figures out which f_j's divide h_1, which f_j's divide h_2, which f_j's divide h_3, etc. The algorithm takes time $O(n(\lg n)^3 \lg\lg n)$, where n is the total number of input bits.

An important special case is that $f_1, f_2, \ldots, f_t$ are the first t prime numbers. Then this algorithm computes the small factors of h_1, the small factors of h_2, etc.

History. I introduced this algorithm in [12, Section 21] and [14]. The version in [12] is slower but more general: it relies solely on multiplication, exact division, and greatest common divisors.

Improvements. There are many previous algorithms to find small factors: for example, Legendre's root-finding method (when the inputs are polynomials over a finite field), sieving (when the inputs are successive values of a polynomial), Pollard's $p-1$ method, and Lenstra's elliptic-curve method. One of my current projects is to combine and optimize these algorithms. See [14, Section 2] for a more detailed survey, and [14, Section 3] for a survey of applications.

21. Continued fraction from fraction

Input. Let A be a field. Let d be a nonnegative integer. The algorithm in this section is given polynomials $f_1, f_2 \in A[x]$, not both zero.

Output. This algorithm computes polynomials $M_{11}, M_{12}, M_{21}, M_{22}$ in $A[x]$ such that $\deg M_{11}, \deg M_{12}, \deg M_{21}, \deg M_{22}$ are all at most d; $M_{11}M_{22} - M_{12}M_{21}$ is in $\{-1, 1\}$; and $\deg(M_{21}f_1 + M_{22}f_2) < \max\{\deg f_1, \deg f_2\} - d$. In particular, $M_{21}f_1 + M_{22}f_2 = 0$ in the important special case $d = \max\{\deg f_1, \deg f_2\}$.

This algorithm also computes a factorization of the matrix $M = \begin{pmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{pmatrix}$: polynomials $q_1, q_2, \ldots, q_t \in A[x]$ with $M = \begin{pmatrix} 0 & 1 \\ 1 & -q_t \end{pmatrix} \cdots \begin{pmatrix} 0 & 1 \\ 1 & -q_2 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & -q_1 \end{pmatrix}$. In particular, $\begin{pmatrix} f_1 \\ f_2 \end{pmatrix} = \begin{pmatrix} q_1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} q_2 & 1 \\ 1 & 0 \end{pmatrix} \cdots \begin{pmatrix} q_t & 1 \\ 1 & 0 \end{pmatrix} M \begin{pmatrix} f_1 \\ f_2 \end{pmatrix}$. The reader may think of this equation as expanding f_1/f_2 into a continued fraction

$$q_1 + \cfrac{1}{q_2 + \cfrac{1}{\ddots + \cfrac{1}{q_t + \cfrac{g_2}{g_1}}}}$$

with $g_1 = M_{11}f_1 + M_{12}f_2$ and $g_2 = M_{21}f_1 + M_{22}f_2$; see Section 14.

Note for future reference that if $\deg f_1 \ge \deg f_2$ then $\deg g_1 = \deg f_1 - \deg M_{22} \ge \deg f_1 - d > \deg g_2$; if also $M_{21} \ne 0$ then $\deg g_1 = \deg f_2 - \deg M_{21}$. (Proof: $\deg M_{12}g_2 < d + \deg f_1 - d = \deg f_1$, and $M_{22}g_1 = M_{12}g_2 \pm f_1$, so $\deg M_{22}g_1 = \deg f_1$. If $M_{21} \ne 0$ then $\deg M_{21}f_1 \ge \deg f_1 \ge \deg f_1 - d > \deg g_2$, and $M_{22}f_2 = g_2 - M_{21}f_1$, so $\deg M_{22}f_2 = \deg M_{21}f_1$.)

Speed. This algorithm uses $O(n(\lg n)^2 \lg\lg n)$ operations in A, where n is the total number of coefficients in f_1, f_2.

How it works. The following description explains how to compute the desired matrix M. The desired factorization of M is visible from the construction of M, as is the (consequent) fact that $\det M \in \{-1, 1\}$.

There are several recursive calls in this algorithm. Most of the recursive calls reduce d; the other recursive calls preserve d and reduce $\deg f_2$.

Check whether $\deg f_1 < \deg f_2$. If so, apply the algorithm recursively to d, f_2, f_1 to find a matrix C, of degree at most d, such that $\deg(C_{21}f_2 + C_{22}f_1) < \deg f_2 - d$. Compute $M = C \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$. The answer is M.

Check whether $\deg f_1 < d$. If so, apply the algorithm recursively to $\deg f_1, f_1, f_2$ to find a matrix M, of degree at most $\deg f_1 < d$, such that $M_{21}f_1 + M_{22}f_2 = 0$. The answer is M.

Check whether $\deg f_2 < \deg f_1 - d$. If so, the answer is $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$.

From now on, $0 \le \deg f_1 - d \le \deg f_2 \le \deg f_1$.

The next step in the algorithm will focus on the top $2d$ coefficients of f_1 and the corresponding coefficients of f_2. It will turn out that these coefficients determine the answer. Consequently, what matters most for the speed of the algorithm is d:

large sizes of f_1 and f_2 make a difference only for the equality tests involved in checking $\deg f_1$ and $\deg f_2$.

Check whether $\deg f_1 > 2d$. If so, define $i = \deg f_1 - 2d$ and apply the algorithm recursively to $d, \lfloor f_1/x^i \rfloor, \lfloor f_2/x^i \rfloor$ to find a matrix M, of degree at most d, such that $\deg(M_{21}\lfloor f_1/x^i \rfloor + M_{22}\lfloor f_2/x^i \rfloor) < \deg(\lfloor f_1/x^i \rfloor) - d = \deg f_1 - i - d$. The answer is M. Indeed, $x^i(M_{21}\lfloor f_1/x^i \rfloor + M_{22}\lfloor f_2/x^i \rfloor)$ has degree below $\deg f_1 - d$; $M_{21}(f_1 \bmod x^i)$ and $M_{22}(f_2 \bmod x^i)$ have degree below $d + i = \deg f_1 - d$; add to see that $M_{21}f_1 + M_{22}f_2$ has degree below $\deg f_1 - d$.

From now on, $0 \le \deg f_1 - d \le \deg f_2 \le \deg f_1 \le 2d$.

If $d = 0$ then $\deg f_1 = \deg f_2 = 0$; the answer is $\begin{pmatrix} 0 & 1 \\ 1 & -f_1/f_2 \end{pmatrix}$.

From now on, $d \ge 1$. The strategy here is to split a d problem into two $\lfloor d/2 \rfloor$ problems: compute the first half of the continued fraction of f_1/f_2, compute one more quotient q, and then compute the rest of the continued fraction of f_1/f_2.

Apply the algorithm recursively to $\lfloor d/2 \rfloor, f_1, f_2$ to find a matrix C, of degree at most $\lfloor d/2 \rfloor$, such that $\deg(C_{21}f_1 + C_{22}f_2) < \deg f_1 - \lfloor d/2 \rfloor$. Compute $g_2 = C_{21}f_1 + C_{22}f_2$. Check whether $\deg g_2 < \deg f_1 - d$; if so, the answer is C.

From now on, $\deg f_1 - d \le \deg g_2 < \deg f_1 - \lfloor d/2 \rfloor$.

Compute $g_1 = C_{11}f_1 + C_{12}f_2$. Compute polynomials $q, r \in A[x]$ such that $g_1 = qg_2 + r$ and $\deg r < \deg g_2$, as explained in Section 17. Problem: Section 17 considers division by monic polynomials; g_2 is not necessarily monic. Solution: Divide g_2 by its leading coefficient, and adjust q accordingly.

Observe that the matrix $\begin{pmatrix} 0 & 1 \\ 1 & -q \end{pmatrix} C = \begin{pmatrix} C_{21} & C_{22} \\ C_{11} - qC_{21} & C_{12} - qC_{22} \end{pmatrix}$ has degree at most $\deg f_1 - \deg g_2$. (Proof: Recall that $\deg C_{22} = \deg f_1 - \deg g_1$; so $\deg qC_{22} = \deg q + \deg C_{22} = (\deg g_1 - \deg g_2) + (\deg f_1 - \deg g_1) = \deg f_1 - \deg g_2$. Recall also that $\deg C_{21} \le \deg f_2 - \deg g_1 \le \deg f_1 - \deg g_1$; so $\deg qC_{21} \le \deg f_1 - \deg g_2$. Finally, all of $C_{11}, C_{12}, C_{21}, C_{22}$ have degree at most $\lfloor d/2 \rfloor < \deg f_1 - \deg g_2$.)

Apply the algorithm recursively to $\deg g_2 - (\deg f_1 - d), g_2, r$ to find a matrix D, of degree at most $\deg g_2 - (\deg f_1 - d)$, such that $\deg(D_{21}g_2 + D_{22}r) < \deg f_1 - d$.

Compute $M = D\begin{pmatrix} 0 & 1 \\ 1 & -q \end{pmatrix} C$. Observe that $M_{21}f_1 + M_{22}f_2 = \begin{pmatrix} 0 & 1 \end{pmatrix} M \begin{pmatrix} f_1 \\ f_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \end{pmatrix} D \begin{pmatrix} 0 & 1 \\ 1 & -q \end{pmatrix} C \begin{pmatrix} f_1 \\ f_2 \end{pmatrix} = \begin{pmatrix} D_{21} & D_{22} \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & -q \end{pmatrix} \begin{pmatrix} g_1 \\ g_2 \end{pmatrix} = \begin{pmatrix} D_{21} & D_{22} \end{pmatrix} \begin{pmatrix} g_2 \\ r \end{pmatrix} = D_{21}g_2 + D_{22}r$.

The answer is M. Indeed, the degree of D is at most $\deg g_2 - \deg f_1 + d$, and the degree of $\begin{pmatrix} 0 & 1 \\ 1 & -q \end{pmatrix} C$ is at most $\deg f_1 - \deg g_2$, so the degree of M is at most d; and $\deg(M_{21}f_1 + M_{22}f_2) = \deg(D_{21}g_2 + D_{22}r) < \deg f_1 - d$.

The integer case. A more complicated algorithm, given a nonnegative integer d and given integers f_1, f_2 not both zero, computes a (factored) 2×2 integer matrix M with entries not much larger than 2^d in absolute value, with determinant in $\{-1, 1\}$, and with $|M_{21}f_1 + M_{22}f_2| < \max\{|f_1|, |f_2|\}/2^d$. It takes time $O(n(\lg n)^2 \lg\lg n)$, where n is the total number of input bits.

The main complication here is that the answer for the top $2d$ bits of f_1 and f_2 is, in general, not exactly the answer for f_1 and f_2. One has to check whether $|M_{21}f_1 + M_{22}f_2|$ is too large, and divide a few more times if it is.

History. Almost all of the ideas in this algorithm were published by Lehmer in [70] in the integer case. Lehmer made the crucial observation that the top $2d$ bits of f_1 and f_2 determined approximately d bits of the continued fraction for f_1/f_2. Lehmer suggested computing the continued fraction for f_1/f_2 by computing a small part of the continued fraction, computing another quotient, and then computing the rest of the continued fraction.

Shortly after fast multiplication was widely understood, Knuth in [60] suggested replacing "a small part" with "half" in Lehmer's algorithm. Knuth proved that the continued fraction for f_1/f_2 could be computed within a $O((\lg n)^4)$ factor of multiplication time. Schönhage in [85] streamlined the Lehmer-Knuth algorithm and proved that the continued fraction for f_1/f_2 could be computed within a $O(\lg n)$ factor of multiplication time.

The $(\lg n)^3$ disparity between [60] and [85] arose as follows. Knuth lost one $\lg n$ factor from continually re-multiplying matrices $\begin{pmatrix} 0 & 1 \\ 1 & -q \end{pmatrix}$ instead of reusing their products M; another $\lg n$ factor from doing a binary search, again with no reuse of partial results, to determine how much of the continued fraction of f_1/f_2 matched the continued fraction of $\lfloor f_1/2^i \rfloor / \lfloor f_2/2^i \rfloor$; and one more $\lg n$ factor from an unnecessarily crude analysis.

Moenck in [73] claimed to have a simplified algorithm covering both the integer case and the polynomial case. In fact, Moenck's algorithm does not work in the integer case, does not work in the "nonnormal" situation of a quotient having degree different from 1, and does not work except when $\deg f_1$ is a power of 2. The errors in [73] begin on page 143, where the "degree" function mapping an integer A to $\lfloor \lg |A| \rfloor$ is claimed to be a homomorphism.

Brent, Gustavson, and Yun in [26, Section 3] outlined a simplified algorithm for the polynomial case. Strassen in [96, page 16] stated the remarkably clean algorithm shown above (under the assumption $\deg f_1 \geq \deg f_2$), with one omission: Strassen's algorithm recurses forever when $d = \deg f_1 = \deg f_2 = 0$.

Improvements. One can replace $\lfloor d/2 \rfloor$ in this algorithm by any integer between 0 and $d-1$. The optimal choice depends heavily on the exact speed of multiplication.

One can often skip some of the degree tests in this algorithm. For example, the recursive calls always have $\deg f_1 \geq \deg f_2$.

It is often helpful (for applications, and for recursion inside this algorithm) to compute $M_{21}f_1 + M_{22}f_2$, and sometimes $M_{11}f_1 + M_{12}f_2$, along with M. One can often save time by incorporating these computations into the recursion. For example, when M is constructed from D, q, C, one can compute $M_{21}f_1 + M_{22}f_2$ as $D_{21}g_2 + D_{22}r$, and one can compute $M_{11}f_1 + M_{12}f_2$ as $D_{11}g_2 + D_{12}r$.

One can often save time by skipping M_{11} and M_{21}, and working solely with $M_{12}, M_{22}, M_{11}f_1 + M_{12}f_2, M_{21}f_1 + M_{22}f_2$. Applications that need M_{11} and M_{21} can use formulas such as $M_{11} = ((M_{11}f_1 + M_{12}f_2) - M_{12}f_2)/f_1$. In [62, page 343] this observation is credited to Gordon H. Bradley.

Some applications need solely $M_{11}f_1+M_{12}f_2$ and $M_{21}f_1+M_{22}f_2$. The algorithm, and some of its recursive calls, can be sped up accordingly.

Often f_1 and f_2 have an easily detected common factor, such as x or $x - 1$. Dividing out this factor speeds up the algorithm, perhaps enough to justify the cost of checking for the factor in the first place.

22. Greatest common divisor

Input. Let A be a field. The algorithm in this section is given polynomials $f_1, f_2 \in A[x]$.

Output. This algorithm computes $\gcd\{f_1, f_2\}$: in other words, a polynomial g such that $gA[x] = f_1A[x] + f_2A[x]$ and such that g is monic if it is nonzero.

This algorithm also computes polynomials $h_1, h_2 \in A[x]$, each of degree at most $\max\{\deg f_1, \deg f_2\}$, such that $g = f_1h_1 + f_2h_2$.

In particular, if f_1 and f_2 are coprime, then $g = 1$; h_1 is a reciprocal of f_1 modulo f_2; and h_2 is a reciprocal of f_2 modulo f_1.

Speed. This algorithm uses $O(n(\lg n)^2 \lg\lg n)$ operations in A, where n is the total number of coefficients in f_1, f_2.

How it works. If $f_1 = f_2 = 0$ then the answer is $0, 0, 0$. Assume from now on that at least one of f_1 and f_2 is nonzero.

Define $d = \max\{\deg f_1, \deg f_2\}$. Apply the algorithm of Section 21 to compute $M_{11}, M_{12}, M_{21}, M_{22}$ in $A[x]$, of degree at most d, with $M_{11}M_{22} - M_{12}M_{21} = \pm 1$ and $M_{21}f_1 + M_{22}f_2 = 0$.

Compute $u = M_{11}f_1 + M_{12}f_2$. Note that $\pm f_1 = M_{22}u$ and $\mp f_2 = M_{21}u$, so $uA[x] = f_1A[x] + f_2A[x]$. In particular, $u \neq 0$.

Compute $g = u/c$, $h = M_{11}/c$, and $h_2 = M_{12}/c$, where c is the leading coefficient of u. Then g is monic, and $gA[x] = f_1A[x] + f_2A[x]$, so $g = \gcd\{f_1, f_2\}$. The answer is g, h_1, h_2.

The integer case. An analogous algorithm, given integers f_1 and f_2, computes $\gcd\{f_1, f_2\}$ and reasonably small integers h_1, h_2 with $\gcd\{f_1, f_2\} = f_1h_1 + f_2h_2$. It takes time $O(n(\lg n)^2 \lg\lg n)$, where n is the total number of input bits.

History. See Section 21. This application has always been one of the primary motivations for studying the problem of Section 21.

Improvements. See Section 21.

See [62, pages 338–341; Exercise 4.5.2–38; Exercise 4.5.2–39; Exercise 4.5.2–40] and [90] for a 2-adic ("binary") approach to greatest common divisors of integers. Here is a typical example of this approach. Assume that f_1 and f_2 are odd multiples of 2^k. The odd part of $\gcd\{f_1, f_2\}$ is not changed by the following procedure:

- If $|f_2| > |f_1|$: Exchange f_1 and f_2. (This is important only when $|f_2|$ is substantially larger than $|f_1|$; one can keep track of very-low-precision approximations to f_1 and f_2, and compare those approximations.)
- If $(f_1 + f_2) \bmod 2^{k+2} = 0$, replace f_1 with $f_1 + f_2$; otherwise replace f_1 with $f_1 - f_2$.
- If $f_1 = 0$, stop. (Now $\gcd\{f_1, f_2\} = f_2$.)
- Double f_2. Add 1 to k. Repeat this step as long as $f_1 \bmod 2^{k+1} = 0$.

One can repeat this procedure until $f_1 = 0$. The total number of steps is at most linear in the input size. Furthermore, one can perform a fraction of the steps using only a fraction of the bits of f_1 and f_2, as in Section 21.

23. Interpolator

Input. Let A be a field. Let t be a nonnegative integer. The algorithm in this section is given polynomials $f_1, f_2, \dots, f_t \in A[x]$ and nonzero coprime polynomials $g_1, g_2, \dots, g_t \in A[x]$.

Output. This algorithm computes $h \in A[x]$, with $\deg h < \deg g_1 g_2 \cdots g_t$, such that $h \equiv f_1 \pmod{g_1}$, $h \equiv f_2 \pmod{g_2}$, $\dots$, $h \equiv f_t \pmod{g_t}$.

In particular, consider the special case that each g_j is a monic linear polynomial $x - c_j$. The answer h is a polynomial of degree below t such that $h(c_1) = f_1(c_1)$, $h(c_2) = f_2(c_2)$, $\dots$, $h(c_t) = f_t(c_t)$. Finding h is usually called **interpolation** in this case, and I suggest using the same name for the general case. Another common name is **Chinese remaindering**.

Speed. This algorithm uses $O(n(\lg n)^2 \lg\lg n)$ operations in A, where n is the total number of input coefficients.

How it works. For $t = 0$: The answer is 0.

Compute $G = g_1 \cdots g_t$ as explained in Section 12.

Compute $G \bmod g_1^2, \dots, G \bmod g_t^2$ as explained in Section 18.

Divide each $G \bmod g_j^2$ by g_j, as explained in Section 17, to obtain $(G/g_j) \bmod g_j$. Note that G/g_j and g_j are coprime; thus $((G/g_j) \bmod g_j)$ and g_j are coprime.

Compute a (reasonably small) reciprocal p_j of $((G/g_j) \bmod g_j)$ modulo g_j, as explained in Section 22. Compute $q_j = f_j p_j \bmod g_j$ as explained in Section 17.

Now compute $h = (q_1/g_1 + \cdots + q_t/g_t)G$ as explained in Section 13. (Proof that, modulo g_j, this works: $h \equiv q_j(G/g_j) \equiv f_j p_j (G/g_j) \equiv f_j p_j (G/g_j \bmod g_j) \equiv f_j$.)

The integer case. An analogous algorithm, given integers $f_1, f_2, \dots, f_t$ and given nonzero coprime integers $g_1, g_2, \dots, g_t$, computes a reasonably small integer h such that $h \equiv f_1 \pmod{g_1}$, $h \equiv f_2 \pmod{g_2}$, $\dots$, $h \equiv f_t \pmod{g_t}$. The algorithm takes time $O(n(\lg n)^2 \lg\lg n)$, where n is the total number of input bits.

History. Horowitz in [52] published most of the above algorithm, in the special case that each g_j is a monic linear polynomial. Horowitz did not have a fast method (for large t) to compute $(G/g_1) \bmod g_1, \dots, (G/g_t) \bmod g_t$ from G. Moenck and Borodin in [74, page 95] and [19, page 381] suggested the above solution in the ("single-precision") integer case.

The special case $t = 2$ was published first by Heindel and Horowitz in [50], along with a different essentially-linear-time interpolation algorithm for general t. The Heindel-Horowitz algorithm is summarized below; it takes time $O(n(\lg n)^3 \lg\lg n)$.

Improvements. When g_j is a linear polynomial, $(G/g_j) \bmod g_j$ has degree 0, so p_j is simply $1/((G/g_j) \bmod g_j)$, and q_j is $(f_j \bmod g_j)/((G/g_j) \bmod g_j)$. More generally, whenever g_j is very small, the algorithm of this section provides very small inputs to the modular-reciprocal algorithm of Section 22.

When g_j is a monic linear polynomial, $(G/g_j) \bmod g_j$ is the same as $G' \bmod g_j$, where G' is the derivative of G. Borodin and Moenck in [19, Sections 8–9] suggested computing $G' \bmod g_1, \dots, G' \bmod g_t$ as explained in Section 18, instead of computing $G \bmod g_1^2, \dots, G \bmod g_t^2$.

More generally, if g_j and its derivative g_j' are coprime, then $(G/g_j) \bmod g_j$ is the same as $(g_j')^{-1} G' \bmod g_j$. One can compute $G' \bmod g_1, \dots, G' \bmod g_t$; compute each reciprocal $(G')^{-1} \bmod g_j$; and then compute $q_j = f_j g_j' (G')^{-1} \bmod g_j$.

When $t = 2$, one can use the algorithm of Section 22 to simultaneously compute a reciprocal p_2 of $g_1 = G/g_2$ modulo g_2 and a reciprocal p_1 of $g_2 = G/g_1$ modulo g_1. The answer is then $(f_1 p_1 \bmod g_1)g_2 + (f_2 p_2 \bmod g_2)g_1$. It might be faster to compute $(f_1 p_1 g_2 + f_2 p_2 g_1) \bmod g_1 g_2$.

One can skip the computation of p_1 when $f_1 = 0$. One can reduce the general case to this case: interpolate $0, f_2 - f_1, f_3 - f_1, \dots, f_t - f_1$ and then add f_1 to the result. In particular, for $t = 2$, the answer is $f_1 + ((f_2 - f_1)p_2 \bmod g_2)g_1$, if f_1 is small enough for that answer to be in the right range.

The Heindel-Horowitz algorithm interpolates pairs, then pairs of pairs, etc. This may be better than the Horowitz-Borodin-Moenck algorithm for small t.

One can cache the reciprocals p_j for subsequent interpolations involving the same $g_1, \dots, g_t$.

24. Coprime base

Input. Let A be a field. Let t be a nonnegative integer. The algorithm in this section is given monic polynomials $f_1, f_2, \ldots, f_t \in A[x]$.

Output. This algorithm computes a **coprime base** for $\{f_1, f_2, \ldots, f_t\}$: coprime monic polynomials $g_1, g_2, \ldots \in A[x]$ such that each f_j can be factored as a product of powers of $g_1, g_2, \ldots$.

In fact, the algorithm computes the **natural coprime base** for $\{f_1, f_2, \ldots, f_t\}$: the unique coprime base that does not contain 1 and that can be obtained from $f_1, f_2, \ldots, f_t$ by multiplication, exact division, and greatest common divisors.

In most applications, one also wants the factorization of each f_j over $g_1, g_2, \ldots$. These factorizations can be computed quickly by an extension of the algorithm of Section 20.

Speed. This algorithm uses $O(n(\lg n)^7 \lg\lg n)$ operations in A, where n is the total number of input coefficients.

How it works. See [12]. The algorithm is too complicated to be stated in this paper, as the reader might guess from the exponent 7.

The integer case. An analogous algorithm computes the natural coprime base of a set of positive integers $f_1, f_2, \ldots, f_t$. It takes time $O(n(\lg n)^7 \lg\lg n)$, where n is the total number of input bits.

History. I published this algorithm in [12]. No previous essentially-linear-time algorithms were known, even in the case $t = 2$.

Improvements. One of my current projects is to streamline this algorithm.

References

[1] —, *AFIPS conference proceedings, volume 17: 1960 Western Joint Computer Conference*, 1960.

[2] —, *AFIPS conference proceedings, volume 28: 1966 Spring Joint Computer Conference*, Spartan Books, Washington, 1966.

[3] —, *AFIPS conference proceedings, volume 29: 1966 Fall Joint Computer Conference*, Spartan Books, Washington, 1966.

[4] —, *AFIPS conference proceedings, volume 33, part one: 1968 Fall Joint Computer Conference, December 9–11, 1968, San Francisco, California*, Thompson Book Company, Washington, 1968.

[5] —, *Actes du congrès international des mathématiciens, tome 3*, Gauthier-Villars Éditeur, Paris, 1971. MR 54:5.

[6] Alfred V. Aho (chairman), *Proceedings of fifth annual ACM symposium on theory of computing: Austin, Texas, April 30–May 2, 1973*, Association for Computing Machinery, New York, 1973.

[7] Robert S. Anderssen, Richard P. Brent (editors), *The complexity of computational problem solving*, University of Queensland Press, Brisbane, 1976. ISBN 0–7022–1213–X. Available from `http://web.comlab.ox.ac.uk/oucl/work/richard.brent/pub/pub031.html`.

[8] M. Beeler, R. W. Gosper, R. Schroeppel, *HAKMEM*, Artificial Intelligence Memo No. 239, Massachusetts Institute of Technology, 1972. Available from `http://www.inwap.com/pdp10/hbaker/hakmem/hakmem.html`.

[9] Glenn D. Bergland, *A fast Fourier transform algorithm for real-valued series*, Communications of the ACM **11** (1968), 703–710. ISSN 0001–0782. Available from `http://cr.yp.to/bib/entries.html#1968/bergland-real`.

[10] Daniel J. Bernstein, *New fast algorithms for π and e*, paper for the Westinghouse competition, distributed widely at the Ramanujan Centenary Conference (1987). Available from `http://cr.yp.to/bib/entries.html#1987/bernstein`.

[11] Daniel J. Bernstein, *Detecting perfect powers in essentially linear time*, Mathematics of Computation **67** (1998), 1253–1283. ISSN 0025–5718. MR 98j:11121. Available from `http://cr.yp.to/papers.html`.

[12] Daniel J. Bernstein, *Factoring into coprimes in essentially linear time*, to appear, Journal of Algorithms. ISSN 0196–6774. Available from `http://cr.yp.to/papers.html`.

[13] Daniel J. Bernstein, *Multidigit multiplication for mathematicians*. Available from `http://cr.yp.to/papers.html`.

[14] Daniel J. Bernstein, *How to find small factors of integers*, to appear, Mathematics of Computation. ISSN 0025–5718. Available from `http://cr.yp.to/papers.html`.

[15] Daniel J. Bernstein, *Removing redundancy in high-precision Newton iteration*, draft.

[16] Daniel J. Bernstein, *Pippenger's exponentiation algorithm*, draft. Available from `http://cr.yp.to/papers.html`.

[17] Daniel J. Bernstein, *The complexity of complex convolution*, draft.

[18] Daniel J. Bernstein, *Faster multiplication of integers*, draft.

[19] Allan Borodin, Robert T. Moenck, *Fast modular transforms*, Journal of Computer and System Sciences **8** (1974), 366–386; older version, not a subset, in [74]. ISSN 0022–0000. MR 51:7365. Available from `http://cr.yp.to/bib/entries.html#1974/borodin`.

[20] Jonathan M. Borwein, Peter B. Borwein, *Pi and the AGM*, Wiley, New York, 1987. ISBN 0–471–83138–7. MR 89a:11134.

[21] Jonathan M. Borwein, David M. Bradley, Richard E. Crandall, *Computational strategies for the Riemann zeta function*, Journal of Computational and Applied Mathematics **121** (2000), 247–296. ISSN 0377–0427. MR 2001h:11110. Available from `http://www.sciencedirect.com/science/article/B6TYH-4118GDF-F/1/64371ba75fa0e923ba6b231779fb0673`.

[22] Alfred Brauer, *On addition chains*, Bulletin of the American Mathematical Society **45** (1939), 736–739. ISSN 0273–0979. MR 1,40a. Available from `http://cr.yp.to/bib/entries.html#1939/brauer`.

[23] Richard P. Brent, *The complexity of multiple-precision arithmetic*, in [7] (1976), 126–165. Available from `http://web.comlab.ox.ac.uk/oucl/work/richard.brent/pub/pub032.html`.

[24] Richard P. Brent, *Fast multiple-precision evaluation of elementary functions*, Journal of the ACM **23** (1976), 242–251. ISSN 0004–5411. MR 52:16111. Available from `http://web.comlab.ox.ac.uk/oucl/work/richard.brent/pub/pub034.html`.

[25] Richard P. Brent, *Multiple-precision zero-finding methods and the complexity of elementary function evaluation*, in [99] (1976), 151–176. MR 54:11843. Available from `http://web.comlab.ox.ac.uk/oucl/work/richard.brent/pub/pub028.html`.

[26] Richard P. Brent, Fred G. Gustavson, David Y. Y. Yun, *Fast solution of Toeplitz systems of equations and computation of Padé approximants*, Journal of Algorithms **1** (1980), 259–295. ISSN 0196–6774. MR 82d:65033. Available from `http://web.comlab.ox.ac.uk/oucl/work/richard.brent/pub/pub059.html`.

[27] Richard P. Brent, Edwin M. McMillan, *Some new algorithms for high-precision computation of Euler's constant*, Mathematics of Computation **34** (1980), 305–312. ISSN 0025–5718. MR 82g:10002. Available from `http://web.comlab.ox.ac.uk/oucl/work/richard.brent/pub/pub049.html`.

[28] Georg Bruun, *z-transform DFT filters and FFTs*, IEEE Transactions on Acoustics, Speech, and Signal Processing **26** (1978), 56–63. ISSN 0096–3518. Available from `http://cr.yp.to/bib/entries.html#1978/bruun`.

[29] Peter Bürgisser, Michael Clausen, Mohammed Amin Shokrollahi, *Algebraic complexity theory*, Springer-Verlag, Berlin, 1997. ISBN 3–540–60582–7. MR 99c:68002.

[30] Christoph Burnikel, Joachim Ziegler, *Fast recursive division*, MPI research report I-98-1-022, 1998. Available from `http://data.mpi-sb.mpg.de/internet/reports.nsf/NumberView/1998-1-022`.

[31] Jacques Calmet (editor), *Computer algebra: EUROCAM '82*, Lecture Notes in Computer Science, 144, Springer-Verlag, Berlin, 1982. ISBN 3–540–11607–9. MR 83k:68003.

[32] David G. Cantor, Erich Kaltofen, *On fast multiplication of polynomials over arbitrary algebras*, Acta Informatica **28** (1991), 693–701. ISSN 0001–5903. MR 92i:68068. Available from `http://www.math.ncsu.edu/~kaltofen/bibliography/`.

[33] David V. Chudnovsky, Gregory V. Chudnovsky, *Computer algebra in the service of mathematical physics and number theory*, in [34] (1990), 109–232. MR 92g:11122.

[34] David V. Chudnovsky, Richard D. Jenks (editors), *Computers in mathematics*, Lecture Notes in Pure and Applied Mathematics, 125, Marcel Dekker, New York, 1990. ISBN 0–8247–8341–7. MR 91e:00020.

[35] Stephen A. Cook, *On the minimum computation time of functions*, Ph.D. thesis, Department of Mathematics, Harvard University, 1966. Available from `http://cr.yp.to/bib/entries.html#1966/cook`.

[36] James W. Cooley, John W. Tukey, *An algorithm for the machine calculation of complex Fourier series*, Mathematics of Computation **19** (1965), 297–301. ISSN 0025–5718. MR 31:2843. Available from `http://cr.yp.to/bib/entries.html#1965/cooley`.

[37] Richard Crandall, Barry Fagin, *Discrete weighted transforms and large-integer arithmetic*, Mathematics of Computation **62** (1994), 305–324. ISSN 0025–5718. MR 94c:11123. Available from `http://cr.yp.to/bib/entries.html#1994/crandall`.

[38] John D. Dixon, *Exact solution of linear equations using p-adic expansions*, Numerische Mathematik **40** (1982), 137–141. ISSN 0029–599X. MR 83m:65025.

[39] Pierre Duhamel, H. Hollmann, *Split-radix FFT algorithm*, Electronics Letters **20** (1984), 14–16. ISSN 0013–5194. Available from `http://cr.yp.to/bib/entries.html#1984/duhamel`.

[40] Pierre Duhamel, Martin Vetterli, *Fast Fourier transforms: a tutorial review and a state of the art*, Signal Processing **19** (1990), 259–299. ISSN 0165–1684. MR 91a:94004. Available from `http://cr.yp.to/bib/entries.html#1990/duhamel`.

[41] Gerald Estrin, *Organization of computer systems—the fixed plus variable structure computer*, in [1] (1960), 33–40.

[42] Charles M. Fiduccia, *Polynomial evaluation via the division algorithm: the fast Fourier transform revisited*, in [82] (1972), 88–93. Available from `http://cr.yp.to/bib/entries.html#1972/fiduccia-fft`.

[43] Carl F. Gauss, *Werke, Band 3*, Königlichen Gesellschaft der Wissenschaften, Göttingen, 1866. Available from `http://134.76.163.65/agora_docs/41929TABLE_OF_CONTENTS.html`.

[44] W. Morven Gentleman, Gordon Sande, *Fast Fourier transforms—for fun and profit*, in [3] (1966), 563–578. Available from `http://cr.yp.to/bib/entries.html#1966/gentleman`.

[45] William Gosper, *Strip mining in the abandoned orefields of nineteenth century mathematics*, in [34] (1990), 261–284. MR 91h:11154. Available from `http://cr.yp.to/bib/entries.html#1990/gosper`.

[46] Robert T. Gregory, *Error-free computation: why it is needed and methods for doing it*, Robert E. Krieger Publishing Company, New York, 1980. ISBN 0–89874–240–4. MR 83f:65061.

[47] Bruno Haible, Thomas Papanikolaou, *Fast multiprecision evaluation of series of rational numbers*, Technical Report TI-7/97, Darmstadt University of Technology, 1997. Available from `http://www.informatik.tu-darmstadt.de/TI/Mitarbeiter/papanik/Welcome.html`.

[48] Eric C. R. Hehner, R. Nigel Horspool, *A new representation of the rational numbers for fast easy arithmetic*, SIAM Journal on Computing **8** (1979), 124–134. ISSN 1095–7111. MR 80h:68027.

[49] Michael T. Heideman, Don H. Johnson, C. Sidney Burrus, *Gauss and the history of the fast Fourier transform*, Archive for History of Exact Sciences **34** (1985), 265–277. ISSN 0003–9519. MR 87f:01018. Available from `http://cr.yp.to/bib/entries.html#1985/heideman`.

[50] Lee E. Heindel, Ellis Horowitz, *On decreasing the computing time for modular arithmetic*, in [51] (1971), 126–128. Available from `http://cr.yp.to/bib/entries.html#1971/heindel`.

[51] Fred C. Hennie (chairman), *12th annual symposium on switching and automata theory*, IEEE Computer Society, Northridge, 1971.

[52] Ellis Horowitz, *A fast method for interpolation using preconditioning*, Information Processing Letters **1** (1972), 157–163. ISSN 0020–0190. MR 47:4413. Available from `http://cr.yp.to/bib/entries.html#1972/horowitz`.

[53] Tudor Jebelean, *An algorithm for exact division*, Journal of Symbolic Computation **15** (1993), 169–180. ISSN 0747–7171. MR 93m:68092.

[54] Tudor Jebelean, *Practical integer division with Karatsuba complexity*, in [68] (1997), 339–341.

[55] Anatoly A. Karatsuba, Y. Ofman, *Multiplication of multidigit numbers on automata*, Soviet Physics Doklady **7** (1963), 595–596. ISSN 0038–5689. Available from `http://cr.yp.to/bib/entries.html#1963/karatsuba`.

[56] Ekatharine A. Karatsuba, *Fast evaluation of hypergeometric functions by FEE*, in [80] (1999), 303–314. MR 2000e:65030. Available from `http://cr.yp.to/bib/entries.html#1999/karatsuba`.

[57] Richard M. Karp (chairman), *13th annual symposium on switching and automata theory*, IEEE Computer Society, Northridge, 1972.

[58] Donald E. Knuth, *The art of computer programming, volume 2: seminumerical algorithms*, 1st edition, 1st printing, Addison-Wesley, Reading, 1969; see also newer version in [59]. MR 44:3531.

[59] Donald E. Knuth, *The art of computer programming, volume 2: seminumerical algorithms*, 1st edition, 2nd printing, Addison-Wesley, Reading, 1971; see also older version in [58]; see also newer version in [61]. MR 44:3531.

[60] Donald E. Knuth, *The analysis of algorithms*, in [5] (1971), 269–274. MR 54:11839.

[61] Donald E. Knuth, *The art of computer programming, volume 2: seminumerical algorithms*, 2nd edition, Addison-Wesley, Reading, 1981; see also older version in [59]; see also newer version in [62]. ISBN 0–201–03822–6. MR 83i:68003.

[62] Donald E. Knuth, *The art of computer programming, volume 2: seminumerical algorithms*, 3rd edition, Addison-Wesley, Reading, 1997; see also older version in [61]. ISBN 0–201–89684–2.

[63] Donald E. Knuth (editor), *Selected papers on analysis of algorithms*, CSLI Publications, Stanford, 2000. ISBN 1–57586–212–3. MR 2001c:68066.

[64] Donald E. Knuth, Christos H. Papadimitriou, *Duality in addition chains*, Bulletin of the European Association for Theoretical Computer Science **13** (1981), 2–4; reprinted in [63, chapter 31]. ISSN 0252–9742.

[65] Peter M. Kogge, *Parallel solution of recurrence problems*, IBM Journal of Research and Development **18** (1974), 138–148. ISSN 0018–8646. MR 49:6552.

[66] Peter M. Kogge, Harold S. Stone, *A parallel algorithm for the efficient solution of a general class of recurrence equations*, IEEE Transactions on Computers **22** (1973), 786–793. ISSN 0018–9340. Available from `http://cr.yp.to/bib/entries.html#1973/kogge`.

[67] E. V. Krishnamurthy, *Matrix processors using p-adic arithmetic for exact linear computations*, IEEE Transactions on Computers **26** (1977), 633–639. ISSN 0018–9340. MR 57:7963.

[68] Wolfgang Kuechlin (editor), *Symbolic and algebraic computation: ISSAC '97*, Association for Computing Machinery, New York, 1997. ISBN 0–89791–875–4.

[69] H. T. Kung, *On computing reciprocals of power series*, Numerische Mathematik **22** (1974), 341–348. ISSN 0029–599X. MR 50:3536. Available from `http://cr.yp.to/bib/entries.html#1974/kung`.

[70] Derrick H. Lehmer, *Euclid's algorithm for large numbers*, American Mathematical Monthly **45** (1938), 227–233. ISSN 0002–9890. Available from `http://links.jstor.org/sici?sici=0002-9890%28193804%2945%3A4%3C227%3AEAFLN%3E2.0.CO%3B2-Y`.

[71] Hendrik W. Lenstra, Jr., Robert Tijdeman (editors), *Computational methods in number theory I*, Mathematical Centre Tracts, 154, Mathematisch Centrum, Amsterdam, 1982. ISBN 90–6196–248–X. MR 84c:10002.

[72] Jean-Bernard Martens, *Recursive cyclotomic factorization—a new algorithm for calculating the discrete Fourier transform*, IEEE Transactions on Acoustics, Speech, and Signal Processing **32** (1984), 750–761. ISSN 0096–3518. MR 86b:94004. Available from `http://cr.yp.to/bib/entries.html#1984/martens`.

[73] Robert T. Moenck, *Fast computation of GCDs*, in [6] (1973), 142–151. Available from `http://cr.yp.to/bib/entries.html#1973/moenck`.

[74] Robert T. Moenck, Allan Borodin, *Fast modular transforms via division*, in [57] (1972), 90–96; newer version, not a superset, in [19]. Available from `http://cr.yp.to/bib/entries.html#1972/moenck`.

[75] Peter L. Montgomery, *Modular multiplication without trial division*, Mathematics of Computation **44** (1985), 519–521. ISSN 0025–5718. MR 86e:11121.

[76] Peter L. Montgomery, *An FFT extension of the ellpitic curve method of factorization*, Ph.D. thesis, University of California at Los Angeles, 1992.

[77] Peter J. Nicholson, *Algebraic theory of finite Fourier transforms*, Journal of Computer and System Sciences **5** (1971), 524–547. ISSN 0022–0000. MR 44:4112.

[78] Henri J. Nussbaumer, *Fast polynomial transform algorithms for digital convolution*, IEEE Transactions on Acoustics, Speech, and Signal Processing **28** (1980), 205–215. ISSN 0096–3518. MR 80m:94004. Available from `http://cr.yp.to/bib/entries.html#1980/nussbaumer`.

[79] Christos M. Papadimitriou, *Computational complexity*, Addison-Wesley, Reading, Massachusetts, 1994. ISBN 0201530821. MR 95f:68082.

[80] Nicolas Papamichael, Stephan Ruscheweyh, Edward B. Saff (editors), *Computational methods and function theory 1997: proceedings of the third CMFT conference, 13–17 October 1997, Nicosia, Cyprus*, Series in Approximations and Decompositions, 11, World Scientific, Singapore, 1999. ISBN 9810236263. MR 2000c:00029.

[81] John M. Pollard, *The fast Fourier transform in a finite field*, Mathematics of Computation **25** (1971), 365–374. ISSN 0025–5718. MR 46:1120. Available from `http://cr.yp.to/bib/entries.html#1971/pollard`.

[82] Arnold L. Rosenberg (chairman), *Fourth annual ACM symposium on theory of computing*, Association for Computing Machinery, New York, 1972. MR 50:1553.

[83] Eugene Salamin, *Computation of π using arithmetic-geometric mean*, Mathematics of Computation **30** (1976), 565–570. ISSN 0025–5718. MR 53:7928.

[84] Arnold Schönhage, *Multiplikation großer Zahlen*, Computing **1** (1966), 182–196. ISSN 0010–485X. MR 34:8676. Available from `http://cr.yp.to/bib/entries.html#1966/schoenhage`.

[85] Arnold Schönhage, *Schnelle Berechnung von Kettenbruchentwicklugen*, Acta Informatica **1** (1971), 139–144. ISSN 0001–5903. Available from `http://cr.yp.to/bib/entries.html#1971/schoenhage-gcd`.

[86] Arnold Schönhage, *Schnelle Multiplikation von Polynomen über Körpern der Charakteristik* 2, Acta Informatica **7** (1977), 395–398. ISSN 0001–5903. MR 55:9604. Available from `http://cr.yp.to/bib/entries.html#1977/schoenhage`.

[87] Arnold Schönhage, *Asymptotically fast algorithms for the numerical multiplication and division of polynomials with complex coefficients*, in [31] (1982), 3–15. MR 83m:68064.

[88] Arnold Schönhage, Volker Strassen, *Schnelle Multiplikation großer Zahlen*, Computing **7** (1971), 281–292. ISSN 0010–485X. MR 45:1431. Available from `http://cr.yp.to/bib/entries.html#1971/schoenhage-mult`.

[89] Malte Sieveking, *An algorithm for division of powerseries*, Computing **10** (1972), 153–156. ISSN 0010–485X. MR 47:1257. Available from `http://cr.yp.to/bib/entries.html#1972/sieveking`.

[90] Jonathan Sorenson, *Two fast GCD algorithms*, Journal of Algorithms **16** (1994), 110–144. ISSN 0196–6774. MR 94k:11135.

[91] Thomas G. Stockham, Jr., *High-speed convolution and correlation*, in [2] (1966), 229–233. Available from `http://cr.yp.to/bib/entries.html#1966/stockham`.

[92] Harold S. Stone, *An efficient parallel algorithm for the solution of a tridiagonal linear system of equations*, Journal of the ACM **20** (1973), 27–38. ISSN 0004–5411. MR 48:12792.

[93] Volker Strassen, *Gaussian elimination is not optimal*, Numerische Mathematik **13** (1969), 354–356. ISSN 0029–599X. MR 40:2223. Available from `http://cr.yp.to/bib/entries.html#1969/strassen`.

[94] Volker Strassen, *Die Berechnungskomplexität von elementarsymmetrischen Funktionen und von Interpolationskoeffizienten*, Numerische Mathematik **20** (1973), 238–251. ISSN 0029–599X. MR 48:3296.

[95] Volker Strassen, *The computational complexity of continued fractions*, in [104] (1981), 51–67; see also newer version in [96].

[96] Volker Strassen, *The computational complexity of continued fractions*, SIAM Journal on Computing **12** (1983), 1–27; see also older version in [95]. ISSN 1095–7111. MR 84b:12004. Available from `http://cr.yp.to/bib/entries.html#1983/strassen`.

[97] James J. Sylvester, *On a fundamental rule in the algorithm of continued fractions*, Philosophical Magazine **6** (1853), 297–299.

[98] Andrei L. Toom, *The complexity of a scheme of functional elements realizing the multiplication of integers*, Soviet Mathematics Doklady **3** (1963), 714–716. ISSN 0197–6788.

[99] Joseph F. Traub, *Analytic computational complexity*, Academic Press, New York, 1976. MR 52:15938.

[100] Johannes W. M. Turk, *Fast arithmetic operations on numbers and polynomials*, in [71] (1982), 43–54. MR 84f:10006. Available from `http://cr.yp.to/bib/entries.html#1982/turk`.

[101] Joris van der Hoeven, *Fast evaluation of holonomic functions*, Theoretical Computer Science **210** (1999), 199–215. ISSN 0304–3975. MR 99h:65046. Available from `http://www.math.u-psud.fr/~vdhoeven/`.

[102] Joris van der Hoeven, *Fast evaluation of holonomic functions near and in regular singularities*, Journal of Symbolic Computation **31** (2001), 717–743. ISSN 0747–7171. MR 2002j:30037. Available from `http://www.math.u-psud.fr/~vdhoeven/`.

[103] Martin Vetterli, Henri J. Nussbaumer, *Simple FFT and DCT algorithms with reduced number of operations*, Signal Processing **6** (1984), 262–278. ISSN 0165–1684. MR 85m:65128.

[104] Paul S. Wang (editor), *SYM-SAC '81: proceedings of the 1981 ACM Symposium on Symbolic and Algebraic Computation, Snowbird, Utah, August 5–7, 1981*, Association for Computing Machinery, New York, 1981. ISBN 0–89791–047–8.

[105] Arnold Weinberger, J. L. Smith, *A logic for high-speed addition*, National Bureau of Standards Circular **591** (1958), 3–12. ISSN 0096–9648.

[106] R. Yavne, *An economical method for calculating the discrete Fourier transform*, in [4] (1968), 115–125. Available from `http://cr.yp.to/bib/entries.html#1968/yavne`.

Department of Mathematics, Statistics, and Computer Science (M/C 249), The University of Illinois at Chicago, Chicago, IL 60607–7045

E-mail address: `djb@cr.yp.to`