The biggest difference between time and space is that you can't reuse time.

—Merrick Furst

Chapter 4

Multiplication without extra space

The multiplication of dense univariate polynomials is one of the most fundamental problems in mathematical computing. In the last fifty years, numerous fast multiplication algorithms have been developed, and these algorithms are used in practice today in a variety of applications. However, all the fast multiplication algorithms require a linear amount of intermediate storage space to compute the result. For each of the two most popular fast multiplication algorithms, we develop variants with the same speed but which use much less temporary storage.

We first carefully define the problem under consideration, as well as the notation we will use throughout this chapter. Consider two polynomials $f, g \in \mathsf{R}[x]$, each with degree less than n, written

$$f = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$$
$$g = b_0 + b_1 x + b_2 x^2 + \cdots + b_{n-1} x^{n-1}.$$

We consider algorithms for dense univariate polynomial multiplication in an algebraic IMM. Formally, the input and output will all be on the algebraic side. The read-only input space will contain the $2n$ coefficients of f and g, and correct algorithms must write the $2n-1$ coefficients of their product into the output space and halt. In all algorithms discussed, the cost of ring operations dominates that of integer arithmetic with machine words, and so we state all costs in terms of ring operations.

Some of the results in this chapter were presented at ISSAC 2009 and 2010 (Roche, 2009; Harvey and Roche, 2010).

4.1 Previous results

Observe that the product $f \cdot g$ can be written

$$\sum_{i=0}^{2n-2} \left(\sum_{j=\max(0,i-n+1)}^{\min(i,n-1)} a_j b_{i-j} \right) x^i .$$

Using only a single temporary ring element as an accumulator, we can compute each inner summation, one at a time, to determine all the coefficients of the product. This is called the naïve or school method for multiplication. It works over any ring, uses $O(n^2)$ ring operations, and requires only a constant amount of temporary working space.

So-called fast multiplication algorithms are those whose cost is less than quadratic. We now briefly examine a few such algorithms.

4.1.1 Karatsuba's algorithm

Karatsuba was the first to develop a sub-quadratic multiplication algorithm (1963). This is a divide-and-conquer method and works by first splitting each of f and g into two blocks with roughly half the size. Writing $k = \lfloor n/2 \rfloor$, the four resulting polynomials are

$$\begin{aligned} f_0 &= a_0 + a_1 x + \cdots + a_{k-1} x^{k-1} & f_1 &= a_k + a_{k+1} x + \cdots a_{n-1} x^{n-k-1} \\ g_0 &= b_0 + b_1 x + \cdots + b_{k-1} x^{k-1} & g_1 &= b_k + b_{k+1} x + \cdots b_{n-1} x^{n-k-1}. \end{aligned}$$

We can therefore write $f = f_0 + f_1 x^k$ and $g = g_0 + g_1 x^k$, so their product is $f_0 g_0 + (f_0 g_1 + g_1 f_0) x^k + f_1 g_1 x^{2k}$. Gauss again was the first to notice (in the context of complex number multiplication) that this product can also be written

$$f \cdot g = f_0 g_0 + \left((f_0 + f_1)(g_0 + g_1) - f_0 g_0 - f_1 g_1 \right) x^k + f_1 g_1 x^{2k}.$$

Using this formulation, the product can be computed using exactly 3 recursive calls of approximately half the size, followed by $O(n)$ additions and subtractions. The total asymptotic cost is therefore $O(n^{\log_2 3})$, which is $O(n^{1.59})$.

To be specific, label the three intermediate products as follows:

$$\alpha = f_0 \cdot g_0, \qquad \beta = f_0 \cdot f_1, \qquad \gamma = (f_0 + f_1) \cdot (g_0 + g_1). \tag{4.1}$$

So the final product of f and g is computed as

$$f \cdot g = \alpha + (\gamma - \alpha - \beta) \cdot x^k + \beta \cdot x^{2k}. \tag{4.2}$$

A straightforward implementation might allocate n units of extra storage at each recursive step to store the intermediate product γ, resulting in an algorithm that uses a linear amount of extra space and performs approximately $4n$ additions of ring elements besides the three recursive calls.

There is of course significant overlap between the three terms of (4.2). To see this more clearly, split each polynomial α, β, γ into its low-order and high-order coefficients as with f and g. Then we have (with no overlap):

$$f \cdot g = \alpha_0 + (\gamma_0 + \alpha_1 - \alpha_0 - \beta_0) x^k + (\gamma_1 + \beta_0 - \alpha_1 - \beta_1) x^{2k} + \beta_1 x^{3k} \tag{4.3}$$

Examining this expansion, we see that the difference $\alpha_1 - \beta_0$ occurs twice, so the number of additions can be reduced to $7n/2$ at each recursive step. Popular implementations of Karatsuba's algorithm already make use of this optimisation.

A few authors have also examined the problem of minimising the extra storage space required for Karatsuba's algorithm. Maeder (1993) showed how to compute tight bounds on the amount of extra temporary storage required at the top level of the algorithm, allowing temporary space to be allocated once for all recursive calls. This bound is approximately $2n$ in Maeder's formulation, where the focus was specifically on long integer multiplication.

An improvement to this for polynomial multiplication was given by Thomé (2002), who showed how to structure Karatsuba's algorithm so that only about n extra ring elements need to be stored.

It should be mentioned that both these algorithms are already working in essentially the same model as an IMM, re-using the output space repeatedly in an attempt to minimise the amount of extra space required. These approaches are also discussed by Brent and Zimmermann (2010) in a broader context. Those authors, who have extensive experience in fast implementations of multiplication algorithms, claim that "The efficiency of an implementation of Karatsuba's algorithm depends heavily on memory usage". Our experiments, given at the end of this chapter, help support this claim.

The first algorithm that we will present has the same asymptotic time complexity as Karatsuba's algorithm and the variants above, but only requires $O(\log n)$ temporary storage.

4.1.2 FFT-based multiplication

The fastest practical algorithms for integer and polynomial multiplication are based on the fast Fourier transform algorithm discussed in the previous chapter. These algorithms use the FFT to evaluate the unknown product polynomial at a number of points, then apply the inverse FFT to compute the coefficients of the product from these evaluation points.

To compute the product of f and g, two univariate polynomials with degrees less than n as above, we use a 2^m-PRU ω, where $2^m \geq 2n-1$. We need a PRU of this order because the product polynomial has size at most $2n-1$, and we need at least this many points to uniquely recover it with polynomial interpolation.

The first step is to compute $\mathrm{DFT}_\omega(f)$ and $\mathrm{DFT}_\omega(g)$. Observe that each DFT has size 2^m, which is at most $2(2n-1)-1 = 4n-3$. Since each input polynomial has degree at less than n, both coefficient vectors must be padded with zeroes to almost four times their width, at least if the usual radix-2 FFT algorithm is used. Even making use of the output space, and using in-place FFTs, this step requires extra temporary space for about $6n$ ring elements.

From the initial DFTs, we have $f(\omega^i)$ and $g(\omega^i)$ for $0 \leq i < 2^m$. Write $h = f \cdot g$ for their product in $\mathsf{R}[x]$. From the DFTs of f and g, we can easily write down $\mathrm{DFT}_\omega(h)$ using exactly 2^m multiplications in R, since $h(\omega^i) = f(\omega^i) \cdot g(\omega^i)$ for any i. Finally, we compute the inverse FFT of the evaluations of h to recover the coefficients of the product. Since 2^m could be about twice as large as $\deg h$, we may have to trim some zeros from the end of this output before copying the coefficients of h to the output space.

In summary, we compute

$$f \cdot g = \frac{1}{n} \mathrm{DFT}_{\omega^{-1}} \left(\mathrm{DFT}_{\omega}(f) * \mathrm{DFT}_{\omega}(g) \right),$$

where $*$ signifies pairwise multiplication of vector elements. The last two steps (writing down $\mathrm{DFT}_{\omega}(h)$ and computing the inverse FFT) can be computed in-place in the storage already allocated for the forward DFTs. Therefore the total amount of extra space required for the standard FFT-based multiplication algorithm is approximately $6n$ ring elements.

This basic formulation of FFT-based multiplication has been unchanged for many years, while the main focus of algorithmic development has been on the troublesome requirement that R contains an n-PRU ω. When it does not, we can construct a so-called virtual root of unity by working in a larger field and incurring a small multiplicative factor in the complexity.

For n-bit integer multiplication, Schönhage and Strassen (1971) showed how to achieve bit complexity $O(n \log n \operatorname{loglog} n)$ for this problem. This has recently been improved by Fürer (2007) and De, Kurur, Saha, and Saptharishi (2008) to $O(n \cdot \log n \cdot 2^{\log^* n})$, where $\log^* n$ indicates the *iterated logarithm*, defined as the number of times logarithm must be taken to reach a constant value. Recall from Section 1.4, however, that these subtleties of bit complexity are not meaningful in the IMM model.

Schönhage and Strassen's algorithm can also be applied to polynomial multiplication in the algebraic IMM model, giving a cost of $O(n \log n \operatorname{loglog} n)$ ring operations for degree-n polynomial multiplication, but only when the ring R admits division by 2. Schönhage (1977) used radix-3 FFTs to extend to rings of characteristic 2, and Cantor and Kaltofen (1991) further extended to arbitrary algebras, including non-commutative algebras, with the same asymptotic cost. It remains open whether the new improvements in the bit complexity of multiplication can be applied to the polynomial case.

In any case, this work explicitly dodges these issues by simply assuming that the ring R already contains an n-PRU ω. That is, we assume any of the above methods has been used to add virtual roots of unity already, and optimise from there on. The second algorithm we present reduces the amount of temporary storage required for FFT-based multiplication, over rings that already contain the proper 2^m-PRU, from $6n$ ring elements as in the standard algorithm above, to $O(1)$.

4.1.3 Lower bounds

There are a few lower bounds on the multiplication problem that merit mentioning here. First, in time complexity, Bürgisser and Lotz (2004) proved that at least $\Omega(n \log n)$ ring operations are required for degree-n polynomial multiplication over $\mathbb{C}[x]$ in the *bounded coefficients model*. This model encompasses algebraic circuits where the scalars (in our notation from Chapter 1, edge labels) are all at most 2 in absolute value. In particular, their result implies lower bounds for multiplication by *universal* IMM algorithms that contain no branch instructions.

More to the point of the current discussion, a few time-space tradeoffs for multiplication were given among the large number of such results that appeared a few decades ago. Savage and Swamy (1979) proved that $\Omega(n^2)$ time × space is required for boolean circuits computing binary integer multiplication. Here space is defined as the maximum number of nodes in the circuit that must reside in memory at any given point during the computation.

Algebraic circuits, or straight-line programs, are said to model "oblivious" algorithms, where the structure of the computation is fixed for a given input size. In fact, all the multiplication algorithms discussed above have this property. However, branching programs are more general in that they allow the computation path to change depending on the input. For binary integers, these are essentially an extension of binary decision trees. In this model, Abrahamson (1986) proved that at least $\Omega(n^2/\log^2 n)$ time × space is required for integer multiplication. In this model, space is measured as the logarithm of the number of nodes in the branching program, which corresponds to how much "state" information must be stored in an execution of the program to keep track of where it is in the computation.

The algorithms we present will break these lower bounds for time times temporary space in the algebraic IMM model. Our algorithms do not branch by examining ring elements, and therefore can be modelled by algebraic circuits or straight-line programs. They break these lower bounds by allowing both reads *and* writes to the output space. By re-using the output space, we show that these time-space tradeoffs can be broken. Observe that this follows similar results for more familiar problems such as sorting, which requires $\Omega(n^2)$ time × space in these models as well, but of course can be solved in-place in $O(n\log n)$ time, in the IMM model.

4.2 Space-efficient Karatsuba multiplication

We present an algorithm for polynomial multiplication which has the same mathematical structure as Karatsuba's, and the same time complexity, but which makes careful re-use of the output space so that only $O(\log n)$ temporary space is required for the computation. The description of the algorithm is cleanest when multiplying polynomials in $\mathsf{R}[x]$ with equal sizes that are both divisible by 2, so for simplicity we will present that case first. The corresponding methods for odd-sized operands, different-sized operands, and multiple-precision integers will follow fairly easily from this case.

4.2.1 Improved algorithm: general formulation

The key to obtaining $O(\log n)$ extra space for Karatsuba-like multiplication is by solving a slightly more general problem. In particular, two extra requirements are added to the algorithm at each recursive step.

Condition 4.1. *The low-order n coefficients of the output space are pre-initialized and must be added to. That is, half of the product space is initialized with a polynomial $h \in \mathsf{R}[x]$ of degree less than n.*

With Condition 4.1, the computed result should now equal $h + f \cdot g$.

Condition 4.2. *The first operand to the multiplication f is given as two polynomials which must first be summed before being multiplied by g. That is, rather than a single polynomial $f \in \mathsf{R}[x]$, we are given two polynomials $f^{(0)}, f^{(1)} \in \mathsf{R}[x]$, each with degree less than n.*

With both these conditions, the result of the computation should be $h + (f^{(0)} + f^{(1)}) \cdot g$.

Of course, these conditions should not be made on the very first call to the algorithm, and this will be discussed in the next subsection. We are now ready to present the algorithm in the easiest case that $f, g \in \mathsf{R}[x]$ with $\deg f = \deg g = 2k - 1$ for some $k \in \mathbb{N}$. If A is an array in memory, we use the notation of $A[i..j]$ for the sub-array from indices i to j (inclusive), with $0 \le i \le j < |A|$. If array A contains a polynomial f, then the array element $A[i]$ is the coefficient of x^i in f. The three read-only input operands $f^{(0)}, f^{(1)}, g$ are stored in arrays A, B, C, respectively, and the output is written to array D. From the first condition, $D[0..2k-1]$ is initialized with h.

In the notation of an algebraic IMM, we can think of the read-only arrays A, B, C as residing in the input memory M_I on the algebraic side, and D is the output memory space M_O on the algebraic side. In some recursive calls, however, A, B, C may actually reside in the output space, although this does not change the algorithm. Incidentally, this is the first IMM problem we have seen where valid instances (I, O, O') have all three parts non-empty.

Algorithm 4.1: Space-efficient Karatsuba multiplication

```
Input: $k \in \mathbb{N}$ and $f^{(0)}, f^{(1)}, g, h \in \mathsf{R}[x]$ with degrees less than $2k$ in arrays $A, B, C, D$,
       respectively
Output: $h + (f^{(0)} + f^{(1)}) \cdot g$ stored in array $D$
1  $D[k..2k-1] \leftarrow D[k..2k-1] + D[0..k-1]$
2  $D[3k-1..4k-2] \leftarrow A[0..k-1] + A[k..2k-1] + B[0..k-1] + B[k..2k-1]$
3  Recursive call with $A' \leftarrow C[0..k-1]$, $B' \leftarrow C[k..2k-1]$, $C' \leftarrow D[3k-1..4k-2]$, and
   $D' \leftarrow D[k..3k-2]$
4  $D[3k-1..4k-2] \leftarrow D[k..2k-1] + D[2k..3k-2]$
5  Recursive call with $A' \leftarrow A[0..k-1]$, $B' \leftarrow B[0..k-1]$, $C' \leftarrow C[0..k-1]$, and
   $D' \leftarrow D[0..2k-2]$
6  $D[2k..3k-2] \leftarrow D[2k..3k-2] - D[k..2k-2]$
7  $D[k..2k-1] \leftarrow D[3k-1..4k-2] - D[0..k-1]$
8  Recursive call with $A' \leftarrow A[k..2k-1]$, $B' \leftarrow B[k..2k-1]$, $C' \leftarrow C[k..2k-1]$, and
   $D' \leftarrow D[2k..4k-2]$
9  $D[k..2k-1] \leftarrow D[k..2k-1] - D[2k..3k-1]$
10 $D[2k..3k-2] \leftarrow D[2k..3k-2] - D[3k..4k-2]$
```

Table 4.1 summarizes the computation by showing the actual values (in terms of the input polynomials and intermediate products) stored in each part of the output array D after each step of the algorithm. Between the recursive calls on Steps 3, 5, and 8, we perform some additions and rearranging to prepare for the next multiplication. Notice that a few times a value is added somewhere only so that it can be cancelled off at a later point in the algorithm. An example of this is the low-order half of h, h_0, which is added to h_1 on Step 1 only to be cancelled when we subtract $(\alpha_0 + h_0)$ from this quantity later, on Step 7.

	$D[0..k-1]$	$D[k..2k-1]$	$D[2k..3k-2]^*$	$D[3k-1..4k-2]^*$
0	h_0	h_1	—	—
1	h_0	h_0+h_1	—	—
2	h_0	h_0+h_1	—	$f_0^{(0)}+f_1^{(0)}+f_0^{(1)}+f_1^{(1)}$
3	h_0	$h_0+h_1+\gamma_0$	γ_1	$f_0^{(0)}+f_1^{(0)}+f_0^{(1)}+f_1^{(1)}$
4	h_0	$h_0+h_1+\gamma_0$	γ_1	$h_0+h_1+\gamma_0+\gamma_1$
5	$h_0+\alpha_0$	α_1	γ_1	$h_0+h_1+\gamma_0+\gamma_1$
6	$h_0+\alpha_0$	α_1	$\gamma_1-\alpha_1$	$h_0+h_1+\gamma_0+\gamma_1$
7	$h_0+\alpha_0$	$h_1+\gamma_0+\gamma_1-\alpha_0$	$\gamma_1-\alpha_1$	$h_0+h_1+\gamma_0+\gamma_1$
8	$h_0+\alpha_0$	$h_1+\gamma_0+\gamma_1-\alpha_0$	$\beta_0+\gamma_1-\alpha_1$	β_1
9	$h_0+\alpha_0$	$h_1+\alpha_1+\gamma_0-\alpha_0-\beta_0$	$\beta_0+\gamma_1-\alpha_1$	β_1
10	$h_0+\alpha_0$	$h_1+\alpha_1+\gamma_0-\alpha_0-\beta_0$	$\beta_0+\gamma_1-\alpha_1-\beta_1$	β_1

*The last two sub-arrays shift by one to $D[2k..3k-1]$ and $D[3k..4k-2]$ at Step 8.

Table 4.1: Values stored in D through the steps of Algorithm 4.1

The final value stored in D after Step 10 is

$$\begin{aligned}(h_0+\alpha_0)&+(h_1+\alpha_1+\gamma_0-\alpha_0-\beta_0)x^k\\ &+(\beta_0+\gamma_1-\alpha_1-\beta_1)x^{2k}+\beta_1\cdot x^{3k},\end{aligned} \tag{4.4}$$

which we notice is exactly the same as (4.3) with the addition of $h_0+h_1x^k$ as specified by Condition 4.1.

The base case of the algorithm will be to switch over to the classical algorithm for multiplication; the exact size at which the classical algorithm should be preferred will depend on the implementation. Even with the extra conditions of our problem, the classical algorithm can still be implemented with $O(1)$ space, using a single accumulator in temporary memory, as discussed earlier. This proves the correctness of the following theorem:

Theorem 4.3. *Let $f^{(0)}, f^{(1)}, g, h \in \mathsf{R}[x]$ be polynomials each with degree one less than $n = 2^b$ for some $b \in \mathbb{N}$, with all polynomials stored in read-only input memory except h, which is stored initially in the output space. Algorithm 4.1 correctly computes $h + (f^{(0)} + f^{(1)}) \cdot g$ using $O(1)$ temporary storage on the algebraic side and $O(\log n)$ temporary storage of word-sized integers. The time complexity is $O(n^{\log_2 3})$, or $O(n^{1.59})$.*

Proof. The size of the input polynomials n must be a power of 2 so that each recursive call is on even-sized arguments. To be more precise, the algorithm should be modified so that it initially checks whether the input size is below a certain (constant) threshold, in which case the space-efficient classical algorithm is called. Correctness follows from the discussion above. We can see that there are exactly three recursive calls on input of one-half the size of the original input, and this gives the stated time complexity bounds.

Finally, observe that the temporary storage required by the algorithm on a single recursive call consists of a constant number of pointers, stored on the integer side of the algebraic

IMM, and a single accumulator on the algebraic side. This accumulator can be re-used by the recursive calls, but the pointers must remain, so that the algorithm may proceed when the recursive calls terminate. Since the depth of recursion is $\log_2 n$, this means that $O(\log n)$ temporary space is required on the integer side, as stated. □

4.2.2 Initial calls

The initial call to compute the product of two polynomials f and g will not satisfy Conditions 4.1 and 4.2 above. So there must be top-level versions of the algorithm which do *not* solve the more general problem of $h + (f^{(0)} + f^{(1)}) \cdot g$.

First, denote by `SE_KarMult_1+2` Algorithm 4.1, indicating that both conditions 1 and 2 are satisfied by this version.

Working backwards, we then denote by `SE_KarMult_1` an algorithm that is similar to Algorithm 4.1, but which does not satisfy Condition 4.2. Namely, the input will be just three polynomials $f, g, h \in \mathsf{R}[x]$, with h stored in the output space, and the algorithm computes $h + f \cdot g$. In this version, two of the additions on Step 2 are eliminated. The function call on Step 3 is still to `SE_KarMult_1+2`, but the other two on Steps 5 and 8 are recursive calls to `SE_KarMult_1`.

Similarly, `SE_KarMult` will be the name of the top-level call which does not satisfy either Condition 4.1 or 4.2, and therefore simply computes a single product $f \cdot g$ into an uninitialized output space. Here again we save two array additions on Step 2, and in addition Step 1 is replaced with the instruction:

$$D[0..k-1] \leftarrow C[0..k-1] + C[k..2k-1].$$

This allows the first two function calls on Steps 3 and 5 to be recursive calls to `SE_KarMult`, and only the last one on Step 8 is a call to `SE_KarMult_1`.

The number of additions (and subtractions) at each recursive step determine the hidden constant in the $O(n^{1.59})$ time complexity measure. We mentioned that a naïve implementation of Karatsuba's algorithm uses $4n$ additions at each recursive step, and it is easy to improve this to $7n/2$. By inspection, Algorithm 4.1 uses $9n/2 + O(1)$ additions at each recursive step, and therefore both `SE_KarMult_1` and `SE_KarMult` use only $7n/2 + O(1)$ additions at each recursive step. So `SE_KarMult_1` and `SE_KarMult` match the best known existing algorithms in the number of additions required, and `SE_KarMult_1+2` uses only $n + O(1)$ more additions and subtractions than this.

Asymptotically, the cost of calls to `SE_KarMult_1+2` will eventually dominate, incurring a slight penalty in extra arithmetic operations. However, most of the initial calls, particularly for smaller values of n (where Karatsuba's algorithm is actually used), will be to `SE_KarMult_1` or `SE_KarMult`, and should not be any more costly in time than a good existing implementation. This gives us hope that our space-efficient algorithm might be useful in practice; this hope is confirmed in Section 4.4.

4.2.3 Unequal and odd-sized operands

So far our algorithm only works when both input polynomials have the same degree, and that degree is exactly one less than a power of two. This is necessary because the size of both operands at each recursive call to Algorithm 4.1, as stated, must be even. Special cases to handle the cases when the input polynomials have even degree (and therefore an odd size) or different degrees will resolve these issues and give a general-purpose multiplication algorithm.

First consider the case that $\deg f^{(0)}$, $\deg f^{(1)}$, $\deg g$, and $\deg h$ are all equal to $2k$ for some $k \in \mathbb{N}$, so that each polynomial has an odd number of coefficients. In order to use Algorithm 4.1 in this case, we first pull off the low-order coefficients of each polynomial, writing

$$\begin{aligned}
f^{(0)} &= a_0 + x\hat{f}^{(0)}\\
f^{(1)} &= b_0 + x\hat{f}^{(1)}\\
g &= c_0 + x\hat{g}\\
h &= d_0 + d_1x + x^2\hat{h}
\end{aligned}$$

Now we can rewrite the desired result as follows:

$$\begin{aligned}
h + (f^{(0)} + f^{(1)})\cdot g &= d_0 + d_1x + x^2\hat{h} + (a_0 + b_0 + x\hat{f}^{(0)} + x\hat{f}^{(1)})\cdot(c_0 + x\hat{g})\\
&= d_0 + d_1x + x(a_0+b_0)\hat{g} + xc_0\hat{f}^{(0)} + xc_0\hat{f}^{(1)} + x^2(\hat{h} + (\hat{f}^{(0)} + \hat{f}^{(1)})\cdot\hat{g}).
\end{aligned}$$

Therefore this result can be computed with a single call to Algorithm 4.1 with even-length arguments, $\hat{h} + (\hat{f^{(0)}} + \hat{f^{(1)}})\cdot\hat{g}$, followed by three additions of a scalar times a polynomial, and a few more scalar products and additions. Since we can multiply a polynomial by a scalar and add to a pre-initialized result without using any extra space, this still achieves the same asymptotic time and space complexity as before, albeit with a few extra arithmetic operations.

In fact, our implementation in the MVP library uses only $n/2 + O(1)$ more operations at each recursive step than the power-of-two algorithm analysed earlier, by making careful use of additional versions of these routines when the size of the operands differ by exactly one. These "one-different" versions have the same structure and asymptotic cost as the routines already discussed, so we will not give further details here.

If the degrees of f and g differ by more than one, we use the well-known trick of blocking the larger polynomial into sub-arrays whose length equal the degree of the smaller one. That is, given $f, g \in \mathsf{R}[x]$ with $n = \deg f$, $m = \deg g$ and $n > m$, we write $n = qm + r$ and reduce the problem to computing q products of a degree-m by a degree-$(m-1)$ polynomial and one product of a degree-m by a degree-$(r-1)$ polynomial. The m by $m-1$ products are handled with the "one-different" versions of the algorithms mentioned above.

Each of the q initial products overlap in exactly m coefficients (half the size of the output), so Condition 4.1 actually works quite naturally here, and the q m-by-$(m-1)$ products are calls to a version of `SE_KarMult_1`. The single m-by-$(r-1)$ product is performed *first* (so that the entire output is uninitialized), and this is done by a recursive call to this same procedure multiplying arbitrary unequal-length polynomials.

All these special cases give the following result, which is the first algorithm for dense univariate multiplication to achieve sub-quadratic time times space.

Theorem 4.4. *For any $f, g \in \mathsf{R}[x]$ with degrees less than n, the product $f \cdot g$ can be computed using $O(n^{\log_2 3})$ ring operations, $O(1)$ temporary storage of ring elements, and $O(\log n)$ temporary storage of word-sized integers.*

4.2.4 Rolling a log from space to time

In fact, the $O(\log n)$ temporary storage of integers can be avoided entirely, at the cost of an extra logarithmic factor in the time cost. The idea is to store the current position of the execution in the recursion tree, by a list of integers $(r_0, r_1, \ldots)$. Each r_i is 0, 1, or 2, indicating which of the three recursive calls to the algorithm has been taken on level i of the recursion. Since the recursion depth is at most $\log_2 n$, this list can be encoded into a single integer R with at most $(\log_2 3)\log_2 n$ bits. This is bounded by $O(\log n)$ bits, and therefore the IMM algorithm can choose the word size w appropriately to store R in a single word. Each time the ith recursive call is made, R is updated to $3R + i$, and each time a call returns, R is reset to $\lfloor R/3 \rfloor$. With the position in the recursion tree thus stored in a single machine word, each recursive call overwrites the pointers from the parent call, using only $O(1)$ temporary space throughout the computation.

But when each recursive call returns, the algorithm will not know what parts of memory to operate on, since the parent call's pointers have been overwritten! This is solved by using the encoded list of r_i's, simulating the same series of calls from the top level to recover the pointers used by the parent call. Such a simulation can always be performed, even though the values in the memory locations are different than at the start of the algorithm, since the algorithm is "oblivious". That is, its recursive structure does not depend on the values in memory, but only on their size. The cost of this simulation is $O(\log n)$ at each stage, increasing the time complexity to $O(n^{\log_2 3} \log n)$. Somewhat dishonestly, we point out that this is still actually $O(n^{1.59})$, since 1.59 is strictly greater than $\log_2 3$.

4.3 Space-efficient FFT-based multiplication

We now show how to perform FFT-based multiplication in $O(n \log n)$ time and using only $O(1)$ temporary space, when the coefficient ring R already contains the necessary primitive root of unity. A crucial subroutine will be our in-place truncated Fourier transform algorithm from Chapter 3.

Recall the notation for PRUs introduced in subsection 3.1.6 of the previous chapter: $\omega_{[k]}$ is a 2_k-PRU in R for any k small enough that R actually contains such a PRU, and ω_i is defined as $\omega_{[k]}^{\mathrm{rev}_k\, i}$.

For polynomials $f, g \in \mathsf{R}[x]$, write $n = \deg f + \deg g - 1$ for the number of coefficients in their product. The multiplication problem we consider is as follows. Given f, g, and a 2^k-PRU $\omega_{[k]}$, with $k \geq \log_2 n$, compute the coefficients of the product $h = f \cdot g$. As usual, the coefficients of f and g are stored in read-only input space along with the PRU $\omega_{[k]}$, and the coefficients of h must be written to the size-n output space.

A straightforward implementation of the standard FFT-based multiplication algorithm outlined at the start of this chapter would require $O(n)$ temporary memory to compute this product. Our approach avoids the need for this temporary memory by computing entirely in the output space. In summary, we first compute at least one-third of DFT($f \cdot g$), then at least one-third of what remains, and so forth until the operation is complete. A single in-place inverse TFT then completes the algorithm.

4.3.1 Folded polynomials

The initial stages of the algorithm compute partial DFTs of each of f and g, then multiply them to obtain a partial DFT of h. A naïve approach to computing these partial transforms would be to compute the entire DFT and then discard the unneeded parts, but this would violate both the time and space requirements of our algorithm.

Instead, we define the *folded polynomials* as smaller polynomials determined from each of f, g, h whose ordinary, full-length DFTs correspond to contiguous subsequences of the DFTs of f, g, h.

Definition 4.5. *For any $u \in \mathsf{R}[x]$, and any $i, j \in \mathbb{N}$, the* folded polynomial $u_{a,b} \in \mathsf{R}[x]$ *is the polynomial with degree less than 2^b given by*

$$u_{a,b}(x) = u(\omega_a \cdot x) \operatorname{rem} x^{2^b} - 1.$$

To see how to compute the folded polynomials, first write $u = \sum_{i \ge 0} c_i x^i$. Then

$$u_{a,b} = u(\omega_a x) \operatorname{rem} x^{2^b} - 1 = \sum_{i \ge 0} c_i \omega_a^i x^{i \operatorname{rem} 2^b}.$$

Using this formulation, we can compute $u_{a,b}$ using $O(\deg u)$ ring operations, and using only the storage space for the result, plus a single temporary ring element, an accumulator for the powers of ω_a.

The usefulness of the folded polynomials is illustrated in the following lemma, which shows the relationship between DFT(u) and DFT($u_{a,b}$).

Lemma 4.6. *For any $u \in \mathsf{R}[x]$ and $a, b \in \mathbb{N}$ such that $2^b \mid a$, the elements of the discrete Fourier transform of $u_{a,b}$ at $\omega_{[b]}$ are exactly*

$$\mathrm{DFT}_{\omega_{[b]}}(u_{a,b}) = u(\omega_a), u(\omega_{a+1}), \ldots, u(\omega_{a+2^b-1}).$$

Proof. Let $s \in \{0, 1, \ldots, 2^b - 1\}$. Then ω_s can be written as $\omega_{[b]}^{\mathrm{rev}_b s}$, from the definitions. Therefore, we see that ω_s is a 2^b-PRU, and furthermore, $\omega_s^{2^b} - 1 = 0$ in R. This implies that $u_{a,b}(\omega_s) = u(\omega_a \omega_s)$.

Now since $2^b \mid a$, a and s have no bits in common, so for any $k > \log_2 a$, we have that $\mathrm{rev}_k(a+s) = \mathrm{rev}_k\, a + \mathrm{rev}_k\, s$, and therefore, from the definitions,

$$\omega_a \omega_s = \omega_{[k]}^{\mathrm{rev}_k\, a + \mathrm{rev}_k\, s} = \omega_{a+s}.$$

Putting this together, we see that

$$\begin{aligned}\mathrm{DFT}_{\omega_{[b]}}(u_{a,b}) &= u_{a,b}(\omega_0), u_{a,b}(\omega_1), \ldots, u_{a,b}(\omega_{2^b-1}) \\ &= u(\omega_a), u(\omega_{a+1}), \ldots, u(\omega_{a+2^b-1}).\end{aligned}$$ □

4.3.2 Constant-space algorithm

Our strategy for FFT-based multiplication with $O(1)$ extra space is then to compute

$$\mathrm{DFT}_{\omega_{[b_i]}}(h_{a_i,b_i})$$

for a sequence of indices a_i and b_i satisfying $a_0 = 0$ and $a_i = a_{i-1} + 2^{b_{i-1}}$ for $i \geq 1$. From Lemma 4.6 above, this sequence will give the DFT of h itself after enough iterations. The b_i's at each step will be chosen so that the computation of that step can be performed entirely in the output space.

This approach is presented in Algorithm 4.2. The input consists of two arrays $A, B \in \mathsf{R}^*$, where $A[i]$ is the coefficient of x^i in f and $B[i]$ is the coefficient of x^i in g. The algorithm uses the output space of size $\deg f + \deg g + 1$ and ultimately writes the coefficients of the product $h = fg$ in that space. The subroutine `FFT` indicates the usual radix-2 in-place FFT algorithm on a power-of-two input size, and the subroutine `InplaceITFT` is Algorithm 3.2 from the previous chapter.

Theorem 4.7. *Algorithm 4.2 correctly computes the product $h = f \cdot g$, using $O(n \log n)$ ring operations and $O(1)$ temporary storage of ring elements and word-sized integers.*

Proof. The main loop terminates since q is strictly increasing. Let m be the number of iterations, and let $q_0 > q_1 > \cdots > q_{m-1}$ and $L_0 \geq L_1 \geq \cdots \geq L_{m-1}$ be the values of q and L on each iteration. By construction, the intervals $[q_i, q_i + L_i)$ form a partition of $[0, r-1)$, and L_i is the largest power of two such that $q_i + 2L_i \leq r$. Therefore each L can appear at most twice (i.e., if $L_i = L_{i-1}$ then $L_{i+1} < L_i$) Furthermore, $m \leq 2 \lg r$, and we have $L_i \mid q_i$ for each i.

At each iteration, lines 7–8 compute the coefficients of the folded polynomial $f_{q,\ell}$, placing the result in $M_O[q, \ldots, q+L-1]$. From Lemma 4.6, we know that the FFT on Line 9 then computes $M_O[q+i] = f(\omega_{q+i})$ for $0 \leq i < L$. The next two lines similarly compute $M_O[q+L+i] = g(\omega_{q+i})$ for $0 \leq i < L$. (The condition $q + 2L \leq r$ ensures that both of these transforms fit into the output space.) Lines 13–14 then compute $M_O[q+i] = f(\omega_{q+i}) \cdot g(\omega_{q+i}) = h(\omega_{q+i})$ for $0 \leq i < L$. These are the corresponding elements of the discrete Fourier transform of h, written in reverted binary order.

After line 16 we finally have $M_O[i] = h(\omega_i)$ for all $0 \leq i < n$. Observe that the last element is handled specially since the output space does not have room for both evaluations. The call to Algorithm 3.2 on Line 17 then recovers the coefficients of h, in the normal order.

Algorithm 4.2: Space-efficient FFT-based multiplication

```
**Input**: $f, g \in \mathsf{R}[x]$, stored in arrays $A$ and $B$, respectively
**Output**: The coefficients of $h = f \cdot g$, stored in output space $M_O$
1  $n \leftarrow \deg f + \deg g - 1$
2  $q \leftarrow 0$
3  **while** $q < n - 1$ **do**
4      $\ell \leftarrow \lfloor \lg(n - q) \rfloor - 1$
5      $L \leftarrow 2^\ell$
6      $M_O[q, q+1, \ldots, q+2L-1] \leftarrow (0, 0, \ldots, 0)$
7      **for** $0 \le i \le \deg f$ **do**
8          $M_O[q + (i \operatorname{rem} L)] \leftarrow M_O[q + (i \operatorname{rem} L)] + \omega_q^i A[i]$
9      FFT($M_O[q, q+1, \ldots, q+L-1]$)
10     **for** $0 \le i \le \deg g$ **do**
11         $M_O[q + L + (i \operatorname{rem} L)] \leftarrow M_O[q + L + (i \operatorname{rem} L)] + \omega_q^i B[i]$
12     FFT($M_O[q+L, q+L+1, \ldots, q+2L-1]$)
13     **for** $0 \le i < L$ **do**
14         $M_O[q+i] \leftarrow M_O[q+i] \cdot M_O[q+L+i]$
15     $q \leftarrow q + L$
16 $M_O[n-1] \leftarrow f(\omega_{n-1}) \cdot g(\omega_{n-1})$
17 InplaceITFT($M_O[0, 1, \ldots, n-1]$)
```

We now analyse the time and space complexity. Computing the folded polynomials in the loops on lines 6, 7, 10 and 13 takes $O(n)$ operations per iteration, or $O(n \log n)$ in total, since $m = O(\log n)$. The FFT calls contribute $O(L_i \log L_i)$ per iteration, for a total of

$$O(\sum_i L_i \log L_i) = O(\sum_i L_i \log L) = O(n \log n).$$

Line 16 requires $O(n)$ ring operations, and line 17 requires $O(n \log n)$. The space requirements follow directly from the fact that the `FFT` and `InplaceTFT` calls are all performed in-place. □

4.4 Implementation

These algorithms have been implemented in the MVP library for dense polynomial multiplication over prime fields $\mathbb{F}_p[x]$ with p a single machine word-sized prime. For the FFT-based multiplication method, we further require that $\mathbb{F}_p$ contains 2^k-PRU of sufficiently high order, as mentioned above.

For benchmarking, we compared our algorithms with the `C++` library NTL (Shoup, 2009). This is a useful comparison, as NTL also relies on a blend of classical, Karatsuba, and FFT-based algorithms for different ranges of input sizes, using the standard Karatsuba and FFT algorithms we presented earlier. There are some faster implementations of modular multiplication, most notably `zn_poly` (Harvey, 2008), but these use somewhat different algorithms and methods. We also used the `Mod<long>` class in MVP for the coefficient arithmetic, as opposed to the (faster) Montgomery representation mentioned earlier, because this is essentially the same as NTL's implementation for coefficient arithmetic. Our aim is not to claim the "fastest" implementation but merely to demonstrate that the space-efficient algorithms presented can compete in time cost with other methods that use more space. By comparing against a similar library with similar coefficient arithmetic, we hope to gain insight about the algorithms themselves.

Table 4.2 shows the results of some early benchmarking tests. For all cases, we multiplied two randomly-chosen dense polynomials of the given degree, and the time per iteration, in CPU seconds, is reported. NTL uses classical multiplication for degree less than 16 (and as a base case for Karatsuba), Karatsuba's algorithm for size up to 512, and FFT-based multiplication for the largest sizes. Our crossover points, determined experimentally on the target machine, are a little higher: 32 and 768 respectively. This is probably due to the slightly higher complication of our new algorithms, but there could also be other factors at play.

We should also point out that we used the radix-2 not-in-place inverse FFT for the final step of Algorithm 4.2, rather than the in-place truncated Fourier transform of the previous chapter. This choice was made because of the especially bad performance of the in-place inverse TFT in our experiments before. Observe that when the size of the product is a power of two, our algorithm is actually in-place, and even this is better than previously-known approaches in terms of space efficiency.

Size	NTL	Low-space in MVP	Ratio
100	4.00×10^{-5}	3.00×10^{-5}	.750
150	7.94×10^{-5}	6.24×10^{-5}	.786
200	1.26×10^{-4}	8.70×10^{-5}	.690
250	1.70×10^{-4}	1.17×10^{-4}	.688
300	2.35×10^{-4}	1.88×10^{-4}	.800
350	2.97×10^{-4}	2.24×10^{-4}	.754
400	3.61×10^{-4}	2.60×10^{-4}	.720
450	4.34×10^{-4}	3.10×10^{-4}	.714
500	5.02×10^{-4}	3.50×10^{-4}	.697
600	7.04×10^{-4}	5.68×10^{-4}	.807
700	8.32×10^{-4}	6.96×10^{-4}	.837
800	8.42×10^{-4}	7.78×10^{-4}	.924
900	8.48×10^{-4}	8.24×10^{-4}	.972
1000	8.74×10^{-4}	8.34×10^{-4}	.954
1200	1.70×10^{-3}	1.54×10^{-3}	.906
1400	1.77×10^{-3}	1.59×10^{-3}	.898
1600	1.80×10^{-3}	1.70×10^{-3}	.944
1800	1.90×10^{-3}	1.76×10^{-3}	.926
2000	1.95×10^{-3}	1.88×10^{-3}	.964
3000	4.07×10^{-3}	3.58×10^{-3}	.879
4000	4.22×10^{-3}	4.04×10^{-3}	.957
5000	1.05×10^{-2}	7.38×10^{-3}	.703
6000	1.06×10^{-2}	8.00×10^{-3}	.755
7000	1.09×10^{-2}	8.44×10^{-3}	.774
8000	1.10×10^{-2}	8.80×10^{-3}	.800
9000	2.28×10^{-2}	1.59×10^{-2}	.697

Table 4.2: Benchmarks versus NTL

4.5 Conclusions

The two algorithms presented in this chapter match existing "fast" algorithms in asymptotic time complexity, but need considerably less auxiliary storage space to compute the result. Our algorithms are novel theoretically as the first methods to achieve less than quadratic time $\times$ space for multiplication. This is achieved in part by working in the more permissive (and realistic) IMM model.

Much work remains to be done on this topic. The most obvious question is how these algorithms can be adapted to multi-precision integer multiplication. The FFT-based algorithm can already be used directly as a subroutine, since integer multiplication algorithms that rely on the FFT generally work over modular rings that contain the required PRUs. A straightforward adaptation of the space-efficient Karatsuba multiplication to the multiplication of multi-precision integers is not difficult to imagine, but the extra challenges introduced by the presence of carries, combined with the extreme efficiency of existing libraries such as GMP (Granlund et al., 2010), mean that an even more careful implementation would be needed to gain an advantage in this case. For instance, it would probably be better to use a subtractive version of Karatsuba's algorithm to avoid some carries. This is also likely the area of greatest potential utility of our new algorithms, as routines for long integer multiplication are used in many different areas of computing.

There are also some more theoretical questions left open here. One direction for further research would be to see if a scheme similar to the one presented here for Karatsuba-like multiplication with low space requirements could also be adapted to the Toom-Cook 3-way divide-and-conquer method, or even their arbitrary k-way scheme (Toom, 1963; Cook, 1966). Some of these algorithms are actually used in practice for a range of input sizes between Karatsuba and FFT-based methods, so there is a potential practical application here. Another task for future research would be to improve the Karatsuba-like algorithm even further, either by reducing the amount of extra storage below $O(\log n)$ or (more usefully) reducing the implied constant in the $O(n^{1.59})$ time complexity.

Finally, a natural question is to ask whether polynomial or long integer multiplication can be done completely in-place. We observe that the size of the input (counting the coefficients of both multiplicands) is roughly the same as the size of the output, so asking for an in-place transformation overwriting the input with the output seems plausible. Upon further reflection, it seems that the data dependencies will make this task impossible, but we have no proof of this at the moment.

Bibliography

Karl Abrahamson. Time-space tradeoffs for branching programs contrasted with those for straight-line programs. In *Foundations of Computer Science, 1986., 27th Annual Symposium on*, pages 402–409, October 1986.
doi: 10.1109/SFCS.1986.58. Referenced on page 57.

Richard Brent and Paul Zimmermann. *Modern Computer Arithmetic*. Number 18 in Cambridge Monographs on Applied and Computational Mathematics. Cambridge Univ. Press, November 2010. Referenced on page 55.

Peter Bürgisser and Martin Lotz. Lower bounds on the bounded coefficient complexity of bilinear maps. *J. ACM*, 51:464–482, May 2004. ISSN 0004-5411.
doi: 10.1145/990308.990311. Referenced on page 56.

David G. Cantor and Erich Kaltofen. On fast multiplication of polynomials over arbitrary algebras. *Acta Informatica*, 28:693–701, 1991. ISSN 0001-5903.
doi: 10.1007/BF01178683. Referenced on page 56.

Stephen Arthur Cook. *On the minimum computation time of functions*. PhD thesis, Harvard University, 1966. Referenced on page 68.

Anindya De, Piyush P. Kurur, Chandan Saha, and Ramprasad Saptharishi. Fast integer multiplication using modular arithmetic. In *STOC '08: Proceedings of the 40th annual ACM symposium on Theory of computing*, pages 499–506, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-047-0.
doi: 10.1145/1374376.1374447. Referenced on page 56.

Martin Fürer. Faster integer multiplication. In *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, STOC '07, pages 57–66, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-631-8.
doi: 10.1145/1250790.1250800. Referenced on page 56.

Torbjörn Granlund et al. *GNU Multiple Precision Arithmetic Library, The*. Free Software Foundation, Inc., 4.3.2 edition, January 2010.
URL http://gmplib.org/. Referenced on page 68.

David Harvey. zn_poly: a C library for polynomial arithmetic in $\mathbb{Z}/n\mathbb{Z}[x]$. Online, October 2008.

URL http://cims.nyu.edu/~harvey/code/zn_poly/. Version 0.9. Referenced on page 66.

David Harvey and Daniel S. Roche. An in-place truncated Fourier transform and applications to polynomial multiplication. In *ISSAC '10: Proceedings of the 2010 International Symposium on Symbolic and Algebraic Computation*, pages 325–329, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0150-3.
doi: 10.1145/1837934.1837996. Referenced on page 53.

A. A. Karatsuba and Yu. Ofman. Multiplication of multidigit numbers on automata. *Doklady Akademii Nauk SSSR*, 7:595–596, 1963. Referenced on page 54.

Roman Maeder. Storage allocation for the Karatsuba integer multiplication algorithm. In Alfonso Miola, editor, *Design and Implementation of Symbolic Computation Systems*, volume 722 of *Lecture Notes in Computer Science*, pages 59–65. Springer Berlin / Heidelberg, 1993.
doi: 10.1007/BFb0013168. Referenced on page 55.

Daniel S. Roche. Space- and time-efficient polynomial multiplication. In *ISSAC '09: Proceedings of the 2009 international symposium on Symbolic and algebraic computation*, pages 295–302, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-609-0.
doi: 10.1145/1576702.1576743. Referenced on page 53.

John Savage and Sowmitri Swamy. Space-time tradeoffs for oblivious integer multiplication. In Hermann Maurer, editor, *Automata, Languages and Programming*, volume 71 of *Lecture Notes in Computer Science*, pages 498–504. Springer Berlin / Heidelberg, 1979.
doi: 10.1007/3-540-09510-1_40. Referenced on page 56.

A. Schönhage and V. Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7:281–292, 1971. ISSN 0010-485X.
doi: 10.1007/BF02242355. Referenced on page 56.

Arnold Schönhage. Schnelle Multiplikation von Polynomen über Körpern der Charakteristik 2. *Acta Informatica*, 7:395–398, 1977. ISSN 0001-5903.
doi: 10.1007/BF00289470. Referenced on page 56.

Victor Shoup. NTL: A Library for doing Number Theory. Online, August 2009.
URL http://www.shop.net/ntl/. Version 5.5.2. Referenced on page 66.

Emmanuel Thomé. Karatsuba multiplication with temporary space of size $\leq n$. Online, September 2002.
URL http://www.loria.fr/~thome/publis/. Referenced on page 55.

A. L. Toom. The complexity of a scheme of functional elements simulating the multiplication of integers. *Doklady Akademii Nauk SSSR*, 150:496–498, 1963. ISSN 0002-3264. Referenced on page 68.