Chunky and Equal-Spaced Polynomial Multiplication

Daniel S. Roche

Symbolic Computation Group
Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
droche@cs.uwaterloo.ca
http://www.cs.uwaterloo.ca/~droche/

November 29, 2008

Abstract

Finding the product of two polynomials is an essential and basic problem in computer algebra. While most previous results have focused on the worst-case complexity, we instead employ the technique of adaptive analysis to give an improvement in many "easy" cases. We present two adaptive measures and methods for polynomial multiplication, and also show how to effectively combine them to gain both advantages. One useful feature of these algorithms is that they essentially provide a gradient between existing "sparse" and "dense" methods. We prove that these approaches provide significant improvements in many cases but in the worst case are still comparable to the fastest existing algorithms.

1 Introduction

Computing the product of two polynomials is one of the most important problems in symbolic computation, and the operation is a core primitive function of any computer algebra system. With the increasing size of polynomials being used in cryptographic algorithms, and because multiplication is a subroutine of so many other operations, there is a definite need for efficient polynomial multiplication routines. We introduce new multiplication algorithms which use the technique of adaptive analysis to gain improvements compared to existing approaches both in theory and in practice.

1.1 Polynomial Multiplication

Existing algorithms for univariate polynomial multiplication fall into two categories, depending on the underlying representation chosen. The dense representation, more common for univariate polynomials, represents every coefficient of

the polynomial in an array, including zero coefficients. The sparse or "lacunary" representation, by contrast, is a list of coefficient-exponent pairs, wherein only the nonzero terms are represented.

The following two equations illustrate (respectively) the dense and lacunary representations of a univariate polynomial f over an arbitrary ring R with degree n and t nonzero terms. Here $c_0, c_1, \ldots c_n \in \mathsf{R}$, $a_1, a_2, \ldots, a_t \in \mathsf{R} \setminus \{0\}$, and $0 \leq e_1 < e_2 < \cdots < e_t = n$.

$$f(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_n x^n \tag{1.1}$$
$$f(x) = a_1 x^{e_1} + a_2 x^{e_2} + \cdots + a_t x^{e_t} \tag{1.2}$$

We will sometimes need lower bounds on the size of these representations. Assuming unit storage for ring elements, the size of the dense representation is exactly $\Theta(n)$, and (accounting for the fact that exponents could be multi-precision integers) the size of the sparse representation is bounded below by $\Omega(t \log t + \log n)$.

Here, and for the remainder, we employ the usual notion of a *multiplication time* function $\mathsf{M}(n)$, which gives the number of ring operations required to multiply two polynomials $f, g \in \mathsf{R}[x]$ with degrees both less than n (or integers with at most n bits). We also define and will make frequent use of the related function $\delta(n) = \mathsf{M}(n)/n$.

Algorithmic advances in dense polynomial multiplication have generally followed results for long integer multiplication. The first improvement over the $O(n^2)$ "school" algorithm was by Karatsuba and Ofman [1963], who give $\mathsf{M}(n) \in O(n^{1.59})$. Toom [1963] and Cook [1966] generalized this 2-way divide-and-conquer approach to develop a k-way divide-and-conquer scheme. The most commonly-used variant of this is the 3-way Toom-Cook algorithm, which reduces the asymptotic complexity of multiplication to $O(n^{1.47})$, with a higher overhead cost at each recursive step.

The use of the Fast Fourier Transform algorithm allowed Schönhage and Strassen [1971] to reduce the complexity of long-integer multiplication to $O(n \log n \operatorname{loglog} n)$; Cantor and Kaltofen [1991] later showed how to apply this technique to polynomials for the same asymptotic cost.

A lower bound of $\Omega(n \log n)$ has been proven in the bounded coefficients model over $\mathbb{C}$ [Bürgisser and Lotz, 2004], and work on reducing the complexity towards this lower bound is ongoing [Fürer, 2007].

To multiply two dense polynomials with different degrees, we can use a well-known blocking approach. Say we have $f, g \in \mathsf{R}[x]$ with $\deg f < n$, $\deg g < m$, and $n > m$. Then by splitting f into $\lceil n/m \rceil$ polynomial blocks, each with degree less than m, then we can compute fg using only $O(\frac{n}{m}\mathsf{M}(m))$, or $O(n\delta(m))$, ring operations.

For sparse polynomial multiplication, consider two lacunary polynomials as in (1.2) each with t nonzero terms. The school method requires $O(t^2)$ ring operations, and this cannot be improved in the worst case, since the result could have t^2 terms. But because lacunary polynomials can have multi-precision

integers in the exponents, we must also count word operations, and this is where most algorithmic improvements in sparse polynomial multiplication have come, as well as in the space complexity.

The school method uses $O(t^3 \log n)$ word operations and $O(t^2)$ space. Yan [1998] uses the "geobuckets" data structure to reduce the number of word operations to $O(t^2 \log t \log n)$. Finally, using some old results from Johnson [1974], the space complexity can be reduced to $O(t + r)$, where r is the number of nonzero terms in the product [Monagan and Pearce, 2007].

1.2 Adaptive Analysis

An adaptive algorithm is one whose performance depends not only on the size of the input but also on some other difficulty measure of the particular instance. An important requirement, however, is that the worst-case complexity still matches the best-known algorithms, resulting in routines which are often faster but never asymptotically slower than traditional algorithms.

This idea was originally applied to list sorting problems, and has been first credited to Mehlhorn [1984]. Adaptive algorithms for sorting algorithms work faster for "almost sorted" lists (according to various measures), and have sparked both theoretical interest and practical gain (see Petersson and Moffat [1995] for a good overview).

von zur Gathen and Gerhard [2003] point out some similarities between dense polynomial multiplication and sorting, as for both problems the "school method" has quadratic complexity, while "fast methods" reduce this to superlinear complexity. This leads to the obvious question of whether adaptive methods can be used for polynomial multiplication; this work is an initial attempt at answering that question. It should be noted, however, that adaptive analysis is not new to computer algebra, although is usually takes other names. A notable is example is "early termination" strategies for polynomial and linear algebra computation, such as in Kaltofen and Lee [2003].

1.3 Overview of Approach

We present and analyze two different strategies for adaptive polynomial multiplication. Both are based on sparsity in the input polynomials — that is, zero terms — but in fact these methods will gain the greatest advantage for polynomials which are neither sparse nor dense in the sense that neither standard representation will result in particularly efficient multiplication. These new algorithms essentially provide a finer gradient between the two existing approaches.

The algorithms we present will always proceed in three stages. First, the polynomials are read in and converted to a different representation which effectively captures the relevant measure of difficulty. Second, we multiply the two polynomials in the alternate representation. Finally, the product is converted back to the original representation.

Our aim is of course to make all steps as efficient as possible, but in particular we require that the computation cost be dominated by the second step (where the multiplication is actually performed), as this is the step whose cost will depend on the difficulty of the particular instance. The cost of the conversion steps must therefore be linear in the size of the input polynomials.

In Section 2, we give the first idea for adaptive multiplication, which is to write a polynomial as a list of dense "chunks". The second idea, presented in Section 3, is to write a polynomial with "equal spacing" between coefficients as a dense polynomial composed with a power of the indeterminate. Section 4 shows how to combine these two ideas to make one algorithm which effectively captures both difficulty measures. Finally, a few conclusions and ideas for future directions are discussed in Section 5.

Preliminary progress on some of these results was presented at the Milestones in Computer Algebra (MICA) conference held in Trinidad and Tobago in May 2008 [Roche, 2008].

2 Chunky Polynomials

The idea here is simple, and provides a natural gradient between the standard dense and sparse algorithms for univariate polynomial arithmetic. For $f \in \mathsf{R}[x]$ of degree n, we represent f as a sparse polynomial with dense "chunks" as coefficients:

$$f = f_1 x^{e_1} + f_2 x^{e_2} + \cdots + f_t x^{e_t}, \tag{2.1}$$

with each $f_i \in \mathsf{R}[x]$ and $e_i \in \mathbb{N}$. We require only that $e_{i+1} > e_i + \deg f_i$ for $i = 1, 2, \ldots, t-1$, and each f_i has nonzero constant coefficient. We do *not* insist that each f_i be completely dense; in fact, deciding how much space to allow in each chunk is the challenge of converting to this representation, as we shall see.

Recall the notation introduced above of $\delta(n) = \mathsf{M}(n)/n$. A unique feature of our approach is that we will actually use this function to tune the algorithm. That is, we assume a subroutine is given to evaluate $\delta(n)$ for any chosen value n.

If n is a word-sized integer, then the computation of $\delta(n)$ must use a constant number of word operations. If n is more than word-sized, then we are asking about the cost of multiplying two dense polynomials that cannot fit in memory, so the subroutine should return ∞ in such cases. Practically speaking, the $\delta(n)$ evaluation will usually be an approximation of the actual value, but for what follows we assume the computed value is always exactly correct.

Furthermore, we require that $\delta(n)$ is an increasing function which grows more slowly than linear, meaning that for any $a, b, d \in \mathbb{N}$ with $a < b$,

$$\delta(a + d) - \delta(a) \geq \delta(b + d) - \delta(b). \tag{2.2}$$

These conditions are clearly satisfied for all the dense multiplication algorithms discussed above. In practice, classical multiplication is performed for very small

polynomials, followed by a range of degrees for which divide-and-conquer algorithms are best, and finally FFT-based methods are used for large polynomials. The piecewise δ function resulting from such an implementation would also satisfy the conditions stated above, which we propose are therefore quite reasonable.

The conversion of a sparse or dense polynomial to the chunky representation proceeds in two stages: first, we compute an "optimal chunk size" k, and then we use this computed value as a parameter in the actual conversion algorithm. The product of the two polynomials is then computed in the chunky representation, and finally the result is converted back to the original representation. The steps are presented in reverse order in the hope that the goals at each stage are more clear.

2.1 Multiplication in the chunky representation

Multiplying polynomials in the chunky representation uses sparse multiplication on the outer loop, treating the f_i's as coefficients, and dense multiplication to find each product $f_i g_j$.

Write the arguments $f, g \in \mathsf{R}[x]$ as:

$$\begin{aligned} f &= f_1 x^{e_1^{(f)}} + f_2 x^{e_2^{(f)}} + \cdots + f_{t_f} x^{e_{t_f}^{(f)}} \\ g &= g_1 x^{e_1^{(g)}} + g_2 x^{e_2^{(g)}} + \cdots + g_{t_g} x^{e_{t_g}^{(g)}} \end{aligned} \tag{2.3}$$

Assume similar conditions on each $\deg f_i$ and $\deg g_i$ as in (2.1), and without loss of generality assume $t_f \geq t_g$, that is, f has more chunks than g. To multiply f and g, we need to compute each product $f_i g_j$ and put the resulting chunks into sorted order. It is likely that some of the chunk products $f_i g_j$ will overlap, and hence some coefficients will also need to be summed.

By using heaps of pointers as in Monagan and Pearce [2007], the chunks of the result are computed in order, eliminating unnecessary additions and using little extra space. A min-heap of size t_g is filled with pairs (f_i, g_j), and ordered by the corresponding sum of exponents $e_i^{(f)} + e_j^{(g)}$. Each time we compute a new chunk product $f_i g_i$, we check the new exponent against the degree of the previous chunk, in order to determine whether to make a new chunk in the product or add to the previous one. The details of this approach are given in Algorithm 1.

After using this algortihm to multiply f and g, we can easily convert the result back to the dense or sparse representation in linear time. In fact, if the output is dense, we can preallocate space for the result and store the computed product directly in the dense array, requiring only some extra space for the heap H and a single intermediate product h_{new}.

Algorithm 1: Chunky Multiplication

Input: f, g as in (2.3)

Output: The product $fg = h_1 x^{e_1^{(h)}} + h_2 x^{e_2^{(h)}} + \cdots + h_{t_h} x^{e_{t_h}^{(h)}}$

1 $H \leftarrow$ min-heap initialized with pairs (f_1, g_i) for $i = 1, 2, \ldots, t_g$, ordered by corresponding exponent sums
2 $k \leftarrow 0; \quad c \leftarrow -1$
3 **while** H *is not empty* **do**
4 $(f_i, g_j) \leftarrow$ pair from top of H
5 $e_{\text{new}} \leftarrow e_i^{(f)} + e_j^{(g)}$
6 $h_{\text{new}} \leftarrow f_i \cdot g_j$ using dense multiplication
7 **if** $e_{\text{new}} > c$ **then**
8 $k \leftarrow k + 1$
9 $h_k \leftarrow h_{\text{new}}, \quad e_k^{(h)} \leftarrow e_{\text{new}}$
10 **else**
11 $h_k \leftarrow h_k + h_{\text{new}} x^{e_{\text{new}} - e_k^{(h)}}$
12 $c \leftarrow e_k^{(h)} + \deg h_k$
13 **if** $i < t_f$ **then**
14 Add (f_{i+1}, g_j) to H
15 $t_h \leftarrow k$

Theorem 2.1. *Algorithm 1 correctly computes the product of f and g using*

$$O\left(\sum_{\deg f_i \geq \deg g_j} (\deg f_i) \cdot \delta(\deg g_j) \quad + \sum_{\deg f_i < \deg g_j} (\deg g_j) \cdot \delta(\deg f_i)\right)$$

ring operations and $O(t_f t_g \log t_g \log(\deg fg))$ additional word operations.

Proof. Correctness is clear from the definitions. The bound on ring operations comes from Step 6 using the fact that $\delta(n) = \mathsf{M}(n)/n$. The cost of additions on Step 11 is linear and hence also within the stated bound.

The additional cost of word operations is incurred in removing from and adding to the heap on Steps 3 and 14. Because these steps are executed no more than $t_f t_g$ times, the size of the heap is never more than t_g, and each exponent sum is bounded by the degree of the product, the stated bound is correct. □

Notice that the cost of word operations is always less than the cost would be if we had multiplied f and g in the standard sparse representation. We therefore focus only on minimizing the number of ring operations in the conversion steps that follow.

2.2 Conversion given optimal chunk size

The general chunky conversion problem is, given $f, g \in \mathsf{R}[x]$, both either in the sparse or dense representation, to determine chunky representations for f and g which minimize the cost of Algorithm 1. Here we consider a simpler problem, namely determining an optimal chunky representation for f given that every chunk in g is of the same size $k \in \mathbb{N}$.

The following corollary comes directly from Theorem 2.1 and will guide our conversion algorithm on this step.

Corollary 2.2. *Given $f \in \mathsf{R}[x]$ as in (2.1), the number of ring operations required to multiply f by a single dense polynomial with degree less than k is*

$$O\left(\delta(k) \sum_{\deg f_i \geq k} \deg f_i \quad + \quad k \sum_{\deg f_i < k} \delta(\deg f_i)\right)$$

For high-degree chunks (i.e. $\deg f_i \geq k$), we see that there is no benefit to including any extra "space" — that is, zero coefficients, since the cost is proportional to the sum of the degrees of these chunks. In order to minimize the cost of multiplication, then, we should not have any chunks with degree greater than k (except possibly in the case that every coefficient of the chunk is nonzero), and we should minimize $\sum \delta(\deg f_i)$ for all chunks with size less than k.

These observations form the basis of our approach in Algorithm 2 below. For an input polynomial $f \in \mathsf{R}[x]$, each "gap" of consecutive zero coefficients in f is examined, in order. We determine the optimal chunky conversion if the polynomial ended there — that is, if the polynomial were truncated at that gap. At each gap, this can be accomplished simply by finding the previous gap of highest degree that should be included in the optimal chunky representation. We already have the conversion for the polynomial up to that gap (from a previous step), so we simply add on the last chunk and we are done. At the end, after all gaps have been examined, we have the optimal conversion for the entire polynomial.

Let $a_i, b_i \in \mathbb{Z}$ be the sizes of each consecutive "gap" of zero coefficients and "block" of nonzero coefficients, in order. Each a_i, b_i will be nonzero except possibly for a_0 (if f has a nonzero constant coefficient), and $\sum(a_i + b_i) = n + 1$. For example, the polynomial:

$$f(x) = 5x^{10} + 3x^{11} + 9x^{13} + 20x^{19} + 4x^{20} + 8x^{21}$$

has $a_0 = 10, b_0 = 2, a_1 = 1, b_1 = 1, a_2 = 5$, and $b_2 = 3$. Also define d_i to be the degree of the polynomial up to (not including) gap i, specifically $d_i = a_0 + b_0 + a_1 + b_1 + \cdots + a_{i-1} + b_{i-1}$.

For the gap at index i, where $i > 0$, we store the optimal chunky conversion of $f \bmod x^{d_i}$ by a linked list of indices of all gaps in f that should also be gaps between chunks in the optimal chunky representation. We also store the cost (in ring operations) of multiplying $f \bmod x^{d_i}$ (in this optimal representation)

by a single chunk of size k. Actually, the cost divided by k is saved; call these values $c_1, c_2, \ldots$.

When examining the gap at index j, in order to determine the previous gap of highest degree to be included in the optimal chunky representation if the polynomial ended at gap j, we need to find the index i that minimizes $c_i + \delta(d_j - d_i)$ (indices i where $d_j - d_i > k$ need not be considered, as discussed above). From the condition on $\delta(n)$ in (2.2), we know that, if $c_{i_1} + \delta(d_j - d_{i_1}) < c_{i_2} + \delta(d_j - d_{i_2})$ and $i_1 < i_2$, then the inequality continues to hold as j increases. That is, as soon as an earlier gap results in a smaller cost than a later one, that earlier gap will continue to beat the later one.

This means that we can essentially precompute the values of $\min_i c_i + \delta(d_j - d_i)$ by maintaining a stack of index-index pairs. A pair (i_1, i_2) of indices indicates that $c_{i_1} + \delta(d_j - d_{i_1})$ is minimal as long as $j \leq i_2$. The second pair of indices indicates the minimal value from gap i_2 to the gap of the second index of the second pair, and so forth up to the bottom of the stack and the last gap.

The details of this rather complicated algorithm are given in Algorithm 2.

For an informal justification of the correctness of this algorithm, consider a single iteration through the main **for** loop. At this point, we have computed all optimal costs $c_1, c_2, \ldots, c_{j-1}$, and the lists of gaps to achieve those costs $L_1, L_2, \ldots, L_{j-1}$. We also have computed the stack S, which indicate which of the gaps up to index $j-2$ is optimal and when.

The **while** loop on Step 3 removes all gaps from the stack which are no longer relevant, either because their cost is now beaten by a previous gap (when $i_2 < j$), or because the size of the resulting chunk would be greater than k and therefore unnecessary to consider.

Next, we examine values of $c_{j-1} + \delta(d_i - d_{j-1})$ for various values of i, starting with j and increasing until a lower cost for a previous gap is found. This corresponds to the cost of having the $(j-1)$'th gap be the gap of highest degree when representing $f \bmod x^{d_i}$. If the condition of Step 5 is true, then there is no index at which the $(j-1)$'th gap should be used, so we discard it.

Otherwise, we find indices p and i_2 with $p < i_2$ such that the $(j-1)$'th gap should be the highest-degree one in $f \bmod x^{d_p}$, but not in $f \bmod x^{d_{i_2}}$. This means that, of the gaps up to $j-1$, this gap is optimal up to index v, for some $r \leq v < i_2$. The corresponding pair is then added to the stack and we continue.

From the definitions, $d_{m+1} = \deg f + 1$, and so the list of gaps L_{m+1} returned on the final step gives the optimal list of gaps to include in $f \bmod x^{\deg f + 1}$, which is obviously just f itself.

Theorem 2.3. *Algorithm 2 returns the list of chunks which minimizes the cost when multiplying by a dense size-k chunk, and has linear cost in the size of the input representation of f.*

Proof. Correctness is clear from the discussions above.

For complexity, first notice that the maximal size of S, as well as the number of saved values a_i, b_i, d_i, s_i, L_i, is m, the number of "gaps" of consecutive zero coefficients in f. Clearly m is less than or equal to the number of nonzero terms in f (call this t), which in turn is bounded by the degree of f (call this n). We

Algorithm 2: Chunky Conversion Algorithm

Input: $k \in \mathbb{N}$, $f \in \mathsf{R}[x]$, and integers a_i, b_i, d_i for $i = 0, 1, 2, \ldots, m$ as above

Output: A list L of the indices of gaps to include in the optimal chunky representation of f when multiplying by a single chunk of size k

```
1  L_1 ← 0;   c_1 ← δ(b_0);   S ← empty
2  for j = 2, 3, ..., m + 1 do
3      while top pair (i_1, i_2) from S satisfies i_2 < j or d_j − d_{i_1} > k do
4          Remove (i_1, i_2) from S
5      if top pair (i_1, i_2) from S satisfies
       c_{i_1} + δ(d_j − d_{i_1}) ≤ c_{j−1} + δ(d_j − d_{j−1}) then
6          L_j ← L_{j−1}
7      else
8          L_j ← (j − 1), L_{j−1}
9          r ← j
10         while top pair (i_1, i_2) from S satisfies
           c_{i_1} + δ(d_{i_2} − d_{i_1}) > c_{j−1} + δ(d_{i_2} − d_{j−1}) do
11             r ← i_2
12             Remove (i_1, i_2) from S
13         if S is empty then
14             S ← (j − 1, m)
15         else
16             (i_1, i_2) ← top pair from S
17             v ← least index s.t. r ≤ v < i_2 s.t.
               c_{j−1} + δ(d_v − d_{j−1}) > c_{i_1} + δ(d_v − d_{i_1})
18             S ← (j, v), S
19     c_j ← c_{i_i} + δ(d_j − d_{i_1})        (where (i_1, i_2) is top pair from S)
20 return L_{m+1}
```

store the lists L_i as singly-linked lists, and notice that creating each L_i requires the creation of at most one new node in the linked list (some linked list nodes will be shared in different lists L_i). Therefore the total extra storage is $O(t)$, which is linear in either the sparse or dense representation.

Next, we see that the number of pairs (i_1, i_2) from S examined at each iteration through the main loop is at most one more than the number of pairs that are removed from S in that iteration. Since at most one new pair is added to S on each iteration, the total cost for operations on S is $O(t)$.

Now consider the cost of Step 17 at each iteration. If the input is given in the sparse representation, we just perform a binary search on the interval from r to i_2. Since $i_2 - r < m$, and this step is executed at most m times, the total cost is $O(m \log m)$, which is $O(t \log t)$, linear in the size of the sparse representation.

When the input is given in the dense representation, we also use a binary

search for Step 17, but we start with a one-sided search, sometimes called a "galloping" search. First determine whether v is closer to r or to i_2 by testing the median value $\lfloor (i_2 + r)/2 \rfloor$. If we determine that v is closer to r, then a one-sided binary search from r starts by checking $r + 1$, then $r + 2$, then $r + 4$, $r + 8$, etc., until an index $v = r + 2^i$ is found (not necessarily with v minimal) that satisfies the stated condition. Then perform a normal binary search on the interval from $r + 2^{i-1}$ to $r + 2^i$ to find the smallest value of v, as required. We do the opposite if v is closer to i_2.

The cost of this kind of binary search is thus proportional to $O(\log \min\{v - r, i_2 - v\})$ at each iteration. Notice that the interval (r, i_2) in the stack is then effectively split at the index v, so intuitively whenever more work is required through one iteration of this step, the size of intervals is reduced, so future iterations should have lower cost.

To be more precise, we can see that the worst case for the algorithm is that v lies exactly between r and i_2 at each iteration, and $i_2 - r$ is maximal. Ignoring the fact that the index j increases at each iteration, and assuming the worst case that $k > \deg f$, a rough upper bound on the total cost would then be $O(\sum_{i=1}^{\ell} 2^i \cdot (\ell - i + 1))$, where $\ell = \lceil \log_2 m \rceil$. This sum can be rewritten as $\sum_{i=1}^{\ell} \sum_{\hat{i}=1}^{i} 2^{\hat{i}}$, which is easily seen to be less than $2^{\ell+2}$, or $O(m)$, giving linear cost in the size of the dense representation.

Finally, computing each value of $\delta(u)$ required has constant cost, and the number of such computations is bounded by the number of operations performed on S and the cost of Step 17, so this is also linear in the size of the input representation. □

2.3 Determining the optimal chunk size

All that remains is to compute the optimal chunk size k that will be used in the conversion algorithm from the previous section. This is accomplished by finding the value of k that minimizes the cost of multiplying two polynomials $f, g \in \mathsf{R}[x]$, under the restriction that every chunk of f and of g has size k.

Again, we require more notation. For a polynomial $f \in \mathsf{R}[x]$ and integer k, define $c_f(k)$ to be the least number of chunks in the chunky representation of f, if each chunk has size at most k. So if f is written as in (2.1), $c_f(k)$ is the smallest possible value of t under the restriction that each $\deg f_i$ is less than k.

Therefore, from the cost of multiplication in Theorem 2.1, in this part we want to compute the value of k that minimizes

$$c_f(k) \cdot c_g(k) \cdot k \cdot \delta(k) \tag{2.4}$$

Say $\deg f = n$. After $O(n)$ preprocessing work (making pointers to the beginning and end of each "gap"), $c_f(k)$ could be computed using $O(n/k)$ word operations, for any value k. This leads to one possible approach to computing the value of k that minimizes (2.4) above: simply compute (2.4) for each possible $k = 1, 2, \ldots, \max\{\deg f, \deg g\}$, and remember the minimum. Because $\sum_{i=1}^{n} 1/k \in O(\log k)$, this gives a total cost of $O((\deg f) \log(\deg f) + (\deg g) \log(\deg g))$ ring

operations, which is obviously not linear in either the dense or sparse sizes of g and f, and hence unacceptable for our purposes here.

Rather than explicitly computing each $c_f(k)$ and $c_g(k)$, we essentially maintain chunky representations of f and g with all chunks having size less than k, starting with $k = 1$. As k increases, we count the number of chunks in each representation, which we prove gives a very good approximation to the actual values of $c_f(k)$ and $c_g(k)$, and has the advantage of linear complexity in the size of either the sparse or dense representation.

To facilitate the "update" step, a minimum priority queue Q (whose specific implementation depends on the input polynomial representation) is maintained containing all gaps in the current chunky representations of f and g. For each gap, the key value (on which the priority queue is ordered) is the size of the chunk that would result from merging the two chunks adjacent to the gap into a single chunk.

So for example, if we write f in the chunky representation as

$$f = (4 + 0x + 5x^2) \cdot x^{12} + (7 + 6x + 0x^2 + 0x^3 + 8x^4) \cdot x^{50},$$

then the single gap in f will have key value $3 + 35 + 5 = 43$, which is the size of the two adjacent chunks plus the size of the gap itself. If f is written as in (2.1), then the i^{th} gap has key value

$$1 + e_{i+1} + \deg f_{i+1} - e_i. \tag{2.5}$$

Each gap in the priority queue also contains pointers to the two (or fewer) neighboring gaps in the current chunky representation. Whenever a gap is removed from the queue, this means that we are merging the two chunks adjacent to that gap, so we will need to update (by increasing) the key values of any neighboring gaps accordingly.

At each iteration through the main loop in the algorithm, the smallest key value in the priority queue is examined, and k is increased to this value. Then gaps with key value k are repeatedly removed from the queue until no more remain. This means that each remaining gap, if removed, would result in a chunk of size strictly greater than k. Finally, we compute $\delta(k)$ and an approximation of (2.4) using the number of chunks in the current representations of f and g.

Since the purpose of this step is only to compute an "optimal chunk size" k, and not actually to compute chunky representations of f and g, we do not have to maintain chunky representations of the polynomials as the algorithm proceeds, but merely counters for the number of chunks in each one.

The details of this computation are given in Algorithm 3. Because the initial configuration of f and g, with $k = 1$, is exactly equal to the standard sparse representation, the algorithm states that f and g be input in the sparse representation. This is really only for notational convenience; if the input polynomials are in the dense representation, the value and exponent of all nonzero terms in the sparse representation can obviously be computed in linear time, so the algorithm works in either case.

Algorithm 3: Optimal Chunk Size Computation

Input: $f, g \in \mathsf{R}[x]$ written in the sparse representation as $f = a_1^{(f)} x^{e_1^{(f)}} + a_2^{(f)} x^{e_2^{(f)}} + \cdots + a_{t_f}^{(f)} x^{e_{t_f}^{(f)}}$ and $g = a_1^{(g)} x^{e_1^{(g)}} + a_2^{(g)} x^{e_2^{(g)}} + \cdots + a_{t_g}^{(g)} x^{e_{t_g}^{(g)}}$

Output: $k \in \mathbb{N}$ that minimizes $c_f(k) c_g(k) k\delta(k)$

$Q_f, Q_g \leftarrow$ empty minimum priority queues
for $i = 1, 2, \ldots, t_f - 1$ **do**
    Insert gap i from f into Q_f with key $e_{i+1}^{(f)} - e_i^{(f)} + 1$
for $i = 1, 2, \ldots, t_g - 1$ **do**
    Insert gap i from g into Q_g with key $e_{i+1}^{(g)} - e_i^{(g)} + 1$
$k, k_{\min} \leftarrow 1$; $c_{\min} \leftarrow t_f t_g$
while *Q_f and Q_g are not both empty* **do**
    $k \leftarrow$ smallest key value from Q_f or Q_g
    while *Q_f has an element with key value $\leq k$* **do**
        Remove a k-valued gap from Q_f and update neighbors
    while *Q_g has an element with key value $\leq k$* **do**
        Remove a k-valued gap from Q_g and update neighbors
    $c_{\text{current}} \leftarrow (|Q_f| + 1) \cdot (|Q_g| + 1) \cdot k\delta(k)$
    if $c_{\text{current}} < c_{\min}$ **then**
        $k_{\min} \leftarrow k$; $c_{\min} \leftarrow c_{\text{current}}$
return $k_{\min}$

All that remains is the specification of the data structures used to implement the priority queues Q_f and Q_g. If the input polynomials are in the sparse representation, we simply use standard binary heaps, which give logarithmic cost for each removal and update. Because the exponents in this case are multi-precision integers, we could in theory encounter chunk sizes that are larger than the largest word-sized integer. But, as discussed previously, such a chunk size would be impossible since a dense polynomial with that size cannot be represented in memory. So our priority queues may discard any gaps whose key value is larger than word-sized. This guarantees all keys in the queues are word-size integers, which is necessary for the complexity analysis later.

If the input polynomials are dense, we need a structure which can perform removals and updates in constant time. The advantage is that we can use $O(\deg f + \deg g)$ time and space. For Q_f, we use an array with length $\deg f$ of (possibly empty) linked lists, where the list at index i in the array contains all elements in the queue with key i. (An array of this length is sufficient because each key value in Q_f is at least 2 and at most $1 + \deg f$.) We use the same data structure for Q_g, and this clearly gives constant time for each remove and update operation.

To find the smallest key value in either queue at each iteration through

Step 8, we simply start at the beginning of the array and search forward in each position until a non-empty list is found. Because each queue element update only results in the key values *increasing*, we can start the search at each iteration at the point where the previous search ended. Hence the total cost of Step 8 for all iterations is $O(\deg f + \deg g)$.

The following lemma proves that our approximations of $c_f(k)$ and $c_g(k)$ are reasonably tight, and will be crucial in proving the correctness of the algorithm.

Lemma 2.4. *At any iteration through Step 14 in Algorithm 3, $|Q_f| < 2c_f(k)$ and $|Q_g| < 2c_g(k)$.*

Proof. First consider only the polynomial f. There are two chunky representations with each chunk of degree less than k to consider: the optimal one with $c_f(k)$ chunks and the implicitly computed one with $|Q_f| + 1$ chunks. Denote these by $\bar{f}$ and $\hat{f}$, respectively, and write:

$$\begin{aligned} \bar{f} &= \bar{f}_1 x^{\bar{e}_1} + \bar{f}_2 x^{\bar{e}_2} + \cdots + \bar{f}_{c_f(k)} x^{\bar{e}_{c_f(k)}} \\ \hat{f} &= \hat{f}_1 x^{\hat{e}_1} + \hat{f}_2 x^{\hat{e}_2} + \cdots + \hat{f}_{|Q_f|+1} x^{\hat{e}_{|Q_f|+1}} \end{aligned}$$

By way of contradiction, assume that the low-order terms in three chunks of $\hat{f}$ all lie within a single chunk of $\bar{f}$. That is, for some $i, j \in \mathbb{N}$,

$$\bar{e}_i \le \hat{e}_j < \hat{e}_{j+1} < \hat{e}_{j+2} \le \bar{e}_i + \deg \bar{f}_i.$$

Now consider the size of the chunk formed by eliminating the gap between $\hat{f}_j$ and $\hat{f}_{j+1}$. This value is exactly $1 + \hat{e}_{j+1} + \deg \hat{f}_{j+1} - \hat{e}_j$, from (2.5), and it is the key value of the corresponding gap in Q_f. Because $\hat{e}_{j+2} > \hat{e}_{j+1} + \deg \hat{f}_{j+1}$, this value is at most $\hat{e}_{j+2} - \hat{e}_j$. Applying the inequalities above, we finally see that the key value is less than $\deg \bar{f}_i$, which by definition is at most k.

But this is a contradiction, since no key values less than or equal to k remain in Q_f after the while loop on Step 9 completes. Therefore every chunk in $\bar{f}$ contains at most two low-order terms of distinct chunks in $\hat{f}$. Since each low-order term of a chunk in $\hat{f}$ is nonzero, it must be in some chunk of $\bar{f}$, and hence the number of chunks in $\hat{f}$ is at most twice the number in $\bar{f}$. It follows immediately that $|Q_f| < 2c_f(k)$. An identical argument for g gives the stated result. □

Now we are ready for the main result of this subsection.

Theorem 2.5. *Algorithm 3 computes a chunk size k such that $c_f(k) \cdot c_g(k) \cdot k \cdot \delta(k)$ is at most 4 times the minimum value, and has worst-case linear cost in the size of the input representations.*

Proof. If k is the value returned from the algorithm and k^* is the value which actually minimizes (2.4), the worst that can happen is that the algorithm computes the actual value of $c_f(k)\, c_g(k)\, k\, \delta(k)$, but overestimates the value of $c_f(k^*)\, c_g(k^*)\, k^*\, \delta(k^*)$. This overestimation can only occur in $c_f(k^*)$ and $c_g(k^*)$,

and each of those by only a factor of 2 from Lemma 2.4. Hence the value of (2.4) at k is at most 4 times more than the actual minimum.

If f and g have s_f and s_g nonzero terms, respectively, then the initial sizes of Q_f and Q_g are $s_f - 1$ and $s_g - 1$. Since gaps are only removed from the queues (after they are initialized), there are a total of $O(s_f + s_g)$ insert, remove, and update operations for the entire algorithm.

If the input is sparse and we use a binary heap, the cost of each queue operation is $O(\log t)$, for a total cost of $O(t \log t)$, which is a lower bound on the size of the sparse representation. If the input is in the dense representation, then each queue operation has constant cost. Since $s_f + s_g \in O(\deg f + \deg g)$, the total cost is $O(\deg f + \deg g)$, which is of course linear in the size of the dense representation. □

2.4 Chunky Multiplication Overview

Now we are ready to examine the whole process chunky polynomial conversion and multiplication. First we need the following easy corollary of Theorem 2.3.

Corollary 2.6. *Let $f \in \mathsf{R}[x]$, $k \in \mathbb{N}$, and $\hat{f}$ be a chunky representation of f where all chunks have degree at least k, and $\bar{f}$ be the representation returned by Algorithm 2 on input k. The cost of multiplying $\bar{f}$ by a single chunk of size $\ell < k$ is then less than the cost of multiplying $\hat{f}$ by the same chunk.*

Proof. Consider the result of Algorithm 2 on input ℓ. We know from Theorem 2.3 that this gives the optimal chunky representation for multiplication of f with a size-ℓ chunk. But the only difference in the algorithm on input ℓ and input k is that more pairs are removed at each iteration on Step 3 on input ℓ.

This means that every gap included in the representation $\bar{f}$ is also included in the optimal representation. We also know that all chunks in $\bar{f}$ have degree less than k, so that $\hat{f}$ must have fewer gaps that are in the optimal representation than $\bar{f}$. It follows that multiplication of a size-ℓ chunk by $\bar{f}$ is more efficient than multiplication by $\hat{f}$. □

To review, the entire process to multiply $f, g \in \mathsf{R}[x]$ using the chunky representation is as follows:

1. Compute k from Algorithm 3
2. Compute chunky representations of f and g using Algorithm 2 with input k
3. Multiply the two chunky representations using Algorithm 1
4. Convert the chunky result back to the original representation

Because each step is optimal (or within a constant bound of the optimal), we expect this approach to yield the most efficient chunky multiplication of f and g. In any case, we know it will be at least as efficient as the standard sparse or dense algorithm.

Theorem 2.7. *Computing the product of $f, g \in \mathsf{R}[x]$ never uses more ring operations than either the standard sparse or dense polynomial multiplication algorithms.*

Proof. In Algorithm 3, the values of $c_f(k) \cdot c_g(k) \cdot k \cdot \delta(k)$ for $k = 1$ and $k = \min\{\deg f, \deg g\}$ correspond to the costs of the standard sparse and dense algorithms, respectively. Furthermore, it is easy to see that these values are never overestimated, meaning that the k returned from the algorithm which minimizes this formula gives a cost which is not greater than the cost of either standard algorithm.

Now call $\hat{f}$ and $\hat{g}$ the implicit representations from Algorithm 3, and $\bar{f}$ and $\bar{g}$ the representations returned from Algorithm 2 on input k. We know that the multiplication of $\hat{f}$ by $\hat{g}$ is more efficient than either standard algorithm from above. Since every chunk in $\hat{g}$ has size k, multiplying $\bar{f}$ by $\hat{g}$ will have an even lower cost, from Theorem 2.3. Finally, since every chunk in $\bar{f}$ has size at most k, Corollary 2.6 tells us that the cost is further reduced by multiplying $\bar{f}$ by $\bar{g}$.

The proof is complete from the fact that conversion back to either original representation takes linear time in the size of the output. □

3 Equal-Spaced Polynomials

Next we consider an adaptive representation which is in some sense orthogonal to the chunky representation. This representation will be useful when the coefficients of the polynomial are not grouped together into dense chunks, but rather when they are spaced evenly apart.

Let $f \in \mathsf{R}[x]$ with degree n, and suppose the exponents of f are all divisible by some integer k. Then we can write $f = a_0 + a_1 x^k + a_2 x^{2k} + \cdots$. So by letting $f_D = a_0 + a_1 x + a_2 x^2 + \cdots$, we have $f = f_D \circ x^k$ (where the symbol $\circ$ indicates functional composition).

The generalization of this idea is the equal-spaced representation, which corresponds to writing f as

$$f = (f_D \circ x^k) \cdot x^d + f_S, \tag{3.1}$$

with $k, d \in \mathbb{N}$, $f_D \in \mathsf{R}[x]$ dense with degree less than $n/k - d$, and $f_S \in \mathsf{R}[x]$ sparse with degree less than n. The polynomial f_S is a "noise" polynomial which contains the comparatively few terms in f whose exponents are not of the form $ik + d$.

Unfortunately, converting a sparse polynomial to the best equal-spaced representation seems to be difficult. To see why this is the case, consider the much simpler problem of verifying that a sparse polynomial f can be written as $(f_D \circ x^k) \cdot x^d$. For each exponent e_i of a nonzero term in f, this means confirming that $e_i \equiv d \bmod k$. But the cost of computing each $e_i \bmod k$ is roughly $O(\sum(\log e_i)\delta(\log k))$, which is a factor of $\delta(\log k)$ greater than the size of the input. Since k could be as large as the exponents, we see that even verifying a

proposed k and d takes too much time for the conversion step. Surely computing such a k and d would be even more costly!

Therefore, for this subsection, we will always assume that the input polynomials are given in the dense representation. In Section 4, we will see how by combining with the chunky representation, we effectively handle equal-spaced sparse polynomials without ever having to convert a sparse polynomial directly to the equal-spaced representation.

3.1 Multiplication in the equal-spaced representation

Let $g \in \mathsf{R}[x]$ with degree less than m and write $g = (g_D \circ x^\ell) \cdot x^e + g_S$ as in (3.1). To compute $f \cdot g$, simply sum up the four pairwise products of terms. All these except for the product $(f_D \circ x^k) \cdot (g_D \circ x^\ell)$ are performed using standard sparse multiplication methods.

Notice that if $k = \ell$, then $(f_D \circ x^k) \cdot (g_D \circ x^\ell)$ is simply $(f_D \cdot g_D) \circ x^k$, and hence is easily computed using standard dense methods. However, if k and ℓ are relatively prime, then almost any term in the product can be nonzero.

This indicates that the gcd of k and ℓ is very significant. Write r and s for the greatest common divisor and least common multiple of k and ℓ, respectively. To multiply $(f_D \circ x^k)$ by $(g_D \circ x^\ell)$, we perform a transformation similar to the process of finding common denominators in the addition of fractions. First split $f_D \circ x^k$ into s/k (or ℓ/r) polynomials, each with degree less than n/s and right composition factor x^s, as follows:

$$f_D \circ x^k = (f_0 \circ x^s) + (f_1 \circ x^s) \cdot x^k + (f_2 \circ x^s) \cdot x^{2k} \cdots + (f_{s/k-1} \circ x^s) \cdot x^{s-k}$$

Similarly split $g_D \circ x^\ell$ into s/ℓ polynomials $g_0, g_1, \ldots, g_{s/\ell-1}$ with degrees less than m/s and right composition factor x^s. Then compute all pairwise products $f_i \cdot g_j$, and combine them appropriately to compute the total sum (which will be equal-spaced with right composition factor x^r).

Algorithm 4 gives the details of this method.

As with chunky multiplication, this final product is easily converted to the standard dense representation in linear time. The following theorem gives the complexity analysis for equal-spaced multiplication.

Theorem 3.1. *Let f, g be as above such that $n > m$, and write t_f, t_g for the number of nonzero terms in f_S and g_S, respectively. Then Algorithm 4 correctly computes the product $f \cdot g$ using*

$$O\left((n/r) \cdot \delta(m/s) + nt_g/k + mt_f/\ell + t_f t_g\right)$$

ring operations.

Proof. Correctness follows from the preceding discussion.

The polynomials f_D and g_D have at most n/k and m/ℓ nonzero terms, respectively. So the cost of computing the three products in Step 11 by using standard sparse multiplication is $O(nt_g/k + mt_f/\ell + t_f t_g)$ ring operations, giving the last three terms in the complexity measure.

Algorithm 4: Equal Spaced Multiplication

Input: $f = (f_D \circ x^k) \cdot x^d + f_S, \quad g = (g_D \circ x^\ell) \cdot x^e + g_S,$
with $f_D = a_0 + a_1 x + a_2 x^2 + \cdots, \quad g_D = b_0 + b_1 x + b_2 x^2 + \cdots$
Output: The product $f \cdot g$

1 $r \leftarrow \gcd(k, \ell), \quad s \leftarrow \operatorname{lcm}(k, \ell)$
2 **for** $i = 0, 1, \ldots, s/k - 1$ **do**
3 $\quad f_i \leftarrow a_i + a_{s+i} x + a_{2s+i} x^2 + \cdots$
4 **for** $i = 0, 1, \ldots, s/\ell - 1$ **do**
5 $\quad g_i \leftarrow b_i + b_{s+i} x + b_{2s+i} x^2 + \cdots$
6 $h_D \leftarrow 0$
7 **for** $i = 0, 1, \ldots, s/k - 1$ **do**
8 $\quad$ **for** $j = 0, 1, \ldots, s/\ell - 1$ **do**
9 $\quad\quad$ Compute $f_i \cdot g_j$ by dense multiplication
10 $\quad\quad h_D \leftarrow h_D + ((f_i \cdot g_j) \circ x^s) \cdot x^{ik + j\ell}$
11 Compute $(f_D \circ x^k) \cdot g_S, (g_D \circ x^\ell) \cdot f_S$, and $f_S \cdot g_S$ by sparse multiplication
12 **return** $h_D \cdot x^{e+d} + (f_D \circ x^k) \cdot g_S \cdot x^d + (g_D \circ x^\ell) \cdot f_S \cdot x^e + f_S \cdot g_S$

The initialization in Steps 2–5 and the additions in Steps 10 and 12 all have cost bounded by $O(n/r)$, and hence do not dominate the complexity.

All that remains is the cost of computing each product $f_i \cdot g_j$ by dense multiplication on Step 9. From the discussion above, $\deg f_i < n/s$ and $\deg g_j < m/s$, for each i and j. Since $n > m$, $(n/s) > (m/s)$, and therefore this product can be computed using $O((n/s)\delta(m/s))$ ring operations. The number of iterations through Step 9 is exactly $(s/k)(s/\ell)$. But $s/\ell = k/r$, so the number of iterations is just s/r. Hence the total cost for this step is $O((n/r)\delta(m/s))$, which gives the first term in the complexity measure. □

It is worth noting that no additions of ring elements are actually performed through each iteration of Step 10. The proof is as follows. If this were not true, then we would have

$$i_1 k + j_1 \ell \equiv i_2 k + j_2 \ell \mod s$$

for distinct pairs (i_1, j_1) and (i_2, j_2). Without loss of generality, assume $i_1 \neq i_2$, and write

$$(i_1 k + j_1 \ell) - (i_2 k + j_2 \ell) = qs$$

for some $q \in \mathbb{Z}$. Rearranging gives

$$(i_1 - i_2)k = (j_2 - j_1)\ell + qs.$$

Because $\ell | s$ by definition, the left hand side is a multiple of both k and ℓ, and therefore by definition must be a multiple of s, their lcm. Since $0 \leq i_1, i_2 < s/k$, $|i_1 - i_2| < s/k$, and therefore $|(i_1 - i_2)k| < s$. The only multiple of s with this property is of course 0, and since $k \neq 0$ this means that $i_1 = i_2$, a contradiction.

The following theorem compares the cost of equal-spaced multiplication to standard dense multiplication, and will be used to guide the approach to conversion below.

Theorem 3.2. *Let f, g, m, n, t_f, t_g be as before. Algorithm 4 does not use asymptotically more ring operations than standard dense multiplication to compute the product of f and g as long as $t_f \in O(\delta(n))$ and $t_g \in O(\delta(m))$.*

Proof. Assuming again that $n > m$, the cost of standard dense multiplication is $O(n\delta(m))$ ring operations, which is the same as $O(n\delta(m) + m\delta(n))$.

Using the previous theorem, the number of ring operations used by Algorithm 4 is

$$O\left((n/r)\delta(m/s) + n\delta(m)/k + m\delta(n)/\ell + \delta(n)\delta(m)\right).$$

Because all of k, ℓ, r, s are at least 1, and since $\delta(n) < n$, every term in this complexity measure is bounded by $n\delta(m)+m\delta(n)$. The stated result follows. □

3.2 Converting to equal-spaced

The only question when converting a polynomial f to the equal-spaced representation is how large we should allow t_S (the number of nonzero terms in of f_S) to be. From Theorem 3.2 above, clearly we need $t_S \in \delta(\deg f)$, but we can see from the proof of the theorem that having this bound be tight will often give performance that is equal to the standard dense method (not worse, but not better either).

Let t be the number of nonzero terms in f. Since the goal of any adaptive method is to in fact be faster than the standard algorithms, we use the lower bound of $\delta(n) \in \Omega(\log n)$ and $t \leq \deg f + 1$ and require that $t_S < \log_2 t$.

As usual, let $f \in \mathsf{R}[x]$ with degree less than n and write

$$f = a_1 x^{e_1} + a_2 x^{e_2} + \cdots + a_t x^{e_t},$$

with each $a_i \in \mathsf{R} \setminus \{0\}$. The reader will recall that this corresponds to the sparse representation of f, but keep in mind that we are assuming f is given in the dense representation; f is written this way only for notational convenience.

The conversion problem is then to find the largest possible value of k such that all but at most $\log_2 t$ of the exponents e_j can be written as $ki + d$, for any nonnegative integer i and a fixed integer d. Our approach to computing k and d will be simply to check each possible value of k, in decreasing order. To make this efficient, we need a bound on the size of k.

Lemma 3.3. *Let $n \in \mathbb{N}$ and $e_1, \ldots, e_t$ be distinct integers in the range $[0, n]$. If at least $t - \log_2 t$ of the integers e_i are congruent to the same value modulo k, for some $k \in \mathbb{N}$, then*

$$k \leq \frac{n}{t - 2\log_2 t - 1}.$$

Proof. Without loss of generality, order the e_i's so that $0 \leq e_1 < e_2 < \cdots < e_t \leq n$. Now consider the telescoping sum $(e_2 - e_1) + (e_3 - e_2) + \cdots + (e_t - e_{t-1})$. Every term in the sum is at least 1, and the total is $e_t - e_1$, which is at most n.

Let $S \subseteq \{e_1, \ldots, e_t\}$ be the set of at most $\log_2 t$ integers not congruent to the others modulo k. Then for any $e_i, e_j \notin S$, $e_i \equiv e_j \bmod k$. Therefore $k|(e_j - e_i)$. If $j > i$, this means that $e_j - e_i \geq k$.

Returning to the telescoping sum above, each $e_j \in S$ is in at most two of the sum terms $e_i - e_{i-1}$. So all but at most $2\log_2 t$ of the terms are at least k. Since there are exactly $t-1$ terms, and the total sum is at most n, we conclude that $(t - 2\log_2 t - 1) \cdot k \leq n$. The stated result follows. □

We now employ this lemma to develop an algorithm to determine the best values of k and d, given a dense polynomial f. Starting from the largest possible value from the bound, for each candidate value k, we compute each $e_i \bmod k$, and find the majority element — that is, a common modular image of more than half of the exponents.

To compute the majority element, we use a now well-known approach first credited to Boyer and Moore [1981] and Fischer and Salzberg [1982]. Intuitively, pairs of different elements are repeatedly removed until only one element remains. If there is a majority element, this remaining element is it; only one extra pass through the elements is required to check whether this is the case. Finding the majority element does not actually require removing items from the list, but just making a single pass through the list and maintaining a current candidate majority and a count.

Algorithm 5: Equal Spaced Conversion

Input: Exponents $e_1, e_2, \ldots, e_t \in \mathbb{N}$ and integer n such that $0 \leq e_1 < e_2 < \cdots < e_t = n$

Output: $k, d \in \mathbb{N}$ and $S \subseteq \{e_1, \ldots, e_t\}$ such that $e_i \equiv d \bmod k$ for all exponents e_i not in S, and $|S| \leq \log_2 t$.

```
1  if t < 32 then k ← n
2  else k ← ⌊n/(t − 1 − 2 log_2 t)⌋
3  while k ≥ 2 do
4      d ← e_1 mod k;   j ← 1
5      for i = 2, 3, ..., t do
6          if e_i ≡ d mod k then  j ← j + 1
7          else if j > 0 then  j ← j − 1
8          else  d ← e_i mod k;   j ← 1
9      S ← {e_i : e_i ≢ d mod k}
10     if |S| ≤ log_2 t then  return k, d, S
11     k ← k − 1
12 return 1, 0, ∅
```

Once we have the values of k, d, S from the algorithm, it is easy to construct f_D and f_S such that $f = (f_D \circ x^k) \cdot x^d + f_S$ by making one more pass through

the polynomial. After performing separate conversions for two polynomials $f, g \in \mathsf{R}[x]$, they can of course be multiplied using Algorithm 4.

The following theorem proves correctness when $t > 4$. If $t \leq 4$, we can always trivially set $k = e_t - e_1$ and $d = e_1 \bmod k$ to satisfy the stated conditions.

Theorem 3.4. *Given integers $e_1, \ldots, e_t$ and n, with $t > 4$, Algorithm 5 computes the largest integer k such that at least $t - \log_2 t$ of the integers e_i are congruent modulo k, and uses $O(n)$ word operations.*

Proof. In a single iteration through the **while** loop, we compute the majority element of the set $\{e_i \bmod k : i = 1, 2, \ldots, t\}$, if there is one. For the details of the correctness of this part, see Boyer and Moore [1981]. Because $t > 4$, $\log_2 t < t/2$. Therefore any element which occurs at least $t - \log_2 t$ times in a t-element set is a majority element, which proves that any k returned by the algorithm is such that at least $t - \log_2 t$ of the integers e_i are congruent modulo k.

From Lemma 3.3, we know that the initial value of k on Step 1 or 2 is greater than the optimal k value. Since we start at this value and decrement to 1, the largest k satisfying the stated conditions is returned.

For the complexity, first consider the cost of a single iteration through the main **while** loop. Since each integer e_i is word-sized, computing each $e_i \bmod k$ has constant cost. Computing the set S on Step 9 involves checking each $e_i \bmod k$ once more, so in total each iteration through the loop requires iterating through the values of $e_i \bmod k$ twice. Hence the total cost for each loop iteration is $O(t)$ word operations.

If $t < 32$, then $t \in O(1)$, so the $O(n)$ iterations through the loop result in $O(n)$ total cost.

Otherwise, we start with $k = \lfloor n/(t - 1 - 2\log_2 t) \rfloor$ and decrement. Because $t \geq 32$, $t/2 > 1 + 2\log_2 t$. Therefore $(t - 1 - 2\log_2 t) > t/2$, so the initial value of k is less than $2n/t$. This gives an upper bound on the number of iterations through the **while** loop, and so the total cost is $O(n)$ word operations, as required. □

Algorithm 5 can be implemented using only a constant amount of space (not counting the space for the returned set S). However, this is misleading as the exponents $e_1, \ldots, e_t$ must first be extracted from the dense polynomial f and stored in a list before calling this algorithm. So the total conversion to the equal-spaced representation uses $O(t)$ extra space, not including the space for the output polynomials f_D and f_S. This is at most linear in the size of the output.

4 Chunks with Equal Spacing

The next question is whether the ideas of chunky and equal-spaced polynomial multiplication can be effectively combined into a single algorithm. To accomplish this, an obvious approach would be to first perform chunky polynomial conversion, and then equal-spaced conversion on each of the dense chunks. This

approach should work very well in practice, but unfortunately it may be less efficient than just performing equal-spaced multiplication in some extreme cases, and hence is not acceptable.

We will in fact perform chunky conversion first, but instead of performing equal-spaced conversion on each dense chunk separately, we run Algorithm 5 in parallel in order to determine a single spacing parameter k that will be the same for all chunks of each polynomial.

Let $f = f_1 x^{e_1} + f_2 x^{e_2} + \cdots + f_t x^{e_t}$ in the optimal chunky representation for multiplication by another polynomial g. We first compute the smallest bound on k for any of the chunks f_i, using Lemma 3.3. Starting with this value, we execute the **while** loop of Algorithm 5 for each polynomial f_i, stopping at the largest value of k such that the total size of all sets S on Step 9 for all chunks f_i is at most $\log_2 t_f$, where t_f is the total number of nonzero terms in f.

The polynomial f can then be rewritten (changing the values of f_i and e_i from above) as

$$f = (f_1 \circ x^k) \cdot x^{e_1} + (f_2 \circ x^k) \cdot x^{e_2} + \cdots + (f_t \circ x^k) \cdot x^{e_t} + f_S,$$

where f_S is in the sparse representation and has $O(\log t_f)$ nonzero terms.

Let k^* be the value returned from Algorithm 5 on input of the entire polynomial f. Using k^* instead of k, f could still be written as above with f_S having at most $\log_2 t_f$ terms. Therefore the value of k computed in this way is always greater than or equal to k^* if the initial bounds are correct. This will be the case except when every chunk f_i has few nonzero terms (and therefore t is close to t_f). However, this reduces to the problem of converting a sparse polynomial to the equal-spaced representation, which seems to be intractable, as discussed above. So our cost analysis will be predicated on the assumption that the computed value of k is never smaller than k^*.

We perform the same equal-spaced conversion for g, and then use Algorithm 1 to compute the product $f \cdot g$, with the difference that each product $f_i \cdot g_j$ is computed by Algorithm 4 rather than standard dense multiplication. As with equal-spaced multiplication, the products involving f_S or g_S are performed using standard sparse multiplication.

Theorem 4.1. *The algorithm described above to multiply polynomials with equal-spaced chunks never uses more ring operations than either chunky or equal-spaced multiplication, provided that the computed "spacing parameters" k and ℓ are not smaller than the values returned from Algorithm 5.*

Proof. Let n, m be the degrees of f, g respectively and write t_f, t_g for the number of nonzero terms in f, g respectively. The sparse multiplications involving f_S and g_S use a total of $t_g \log t_f + t_f \log t_g + (\log t_f)(\log t_g)$ ring operations. Both the chunky or equal-spaced multiplication algorithms always require $O(t_g \delta(t_f) + t_f \delta(t_g))$ ring operations in the best case, and since $\delta(n) \in \Omega(\log n)$, the cost of these sparse multiplications is never more than the cost of the standard chunky or equal-spaced method.

The remaining computation is that to compute each product $f_i \cdot g_j$ using equal-spaced multiplication. Write k and ℓ for the powers of x in the right

composition factors of f and g respectively. Theorem 3.1 tells us that the cost of computing each of these products by equal-spaced multiplication is never more than computing them by standard dense multiplication, since k and ℓ are both at least 1. Therefore the combined approach is never more costly than just performing chunky multiplication.

To compare with the cost of equal-spaced multiplication, assume that k and ℓ are the actual values returned by Algorithm 5 on input f and g. This is the worst case, since we have assumed that k and ℓ are never smaller than the values from Algorithm 5.

Now consider the cost of multiplication by a single equal-spaced chunk of g. This is the same as assuming g consists of only one equal-spaced chunk. Write $d_i = \deg f_i$ for each equal-spaced chunk of f, and r, s for the gcd and lcm of k and ℓ, respectively. If $m > n$, then of course m is larger than each d_i, so multiplication using the combined method will use $O((m/r)\sum \delta(d_i/s))$ ring operations, compared to $O((m/r)\delta(n/s))$ for the standard equal-spaced algorithm, by Theorem 3.1.

Now recall the cost equation (2.4) used for Algorithm 3:

$$c_f(b) \cdot c_g(b) \cdot b \cdot \delta(b),$$

where b is the size of all dense chunks in f and g. By definition, $c_f(n) = 1$, and $c_g(n) \leq m/n$, so we know that $c_f(n)\, c_g(n)\, n\, \delta(n) \leq m\, \delta(n)$. Because the chunk sizes d_i were originally chosen by Algorithm 3, we must therefore have $m \sum_{i=1}^{t} \delta(d_i) \leq m\delta(n)$. The restriction that the δ function grows more slowly than linear then implies that $(m/r)\sum \delta(d_i/s) \in O((m/r)\delta(n/s))$, and so the standard equal-spaced algorithm is never more efficient in this case.

When $m \leq n$, the number of ring operations to compute the product using the combined method, again by Theorem 3.1, is

$$O\left(\delta(m/s) \sum_{d_i \geq m} (d_i/r) + (m/r) \sum_{d_i < m} \delta(d_i/s)\right), \tag{4.1}$$

compared with $O((n/r)\delta(m/s))$ for the standard equal-spaced algorithm. Because we always have $\sum_{i=1}^{t} d_i \leq n$, the first term of (4.1) is $O((n/r)\delta(m/s))$. Using again the inequality $m \sum_{i=1}^{t} \delta(d_i) \leq m\delta(n)$, along with the fact that $m\delta(n) \in O(n\delta(m))$ because $m \leq n$, we see that the second term of (4.1) is also $O((n/r)\delta(m/s))$. Therefore the cost of the combined method is never more than the cost of equal-spaced multiplication alone. □

5 Conclusions

We have seen two methods for adaptive polynomial multiplication where we can now achieve optimal representations (under some set of restrictions) in linear time in the size of the input. We have also seen how to combine these two ideas into one algorithm which inherently captures both measures of difficulty, and

should in fact have significantly better performance than either the chunky or equal-spaced algorithm in many cases.

However, converting a sparse polynomial to the equal-spaced representation in linear time is still out of reach, and this problem is the source of the restriction of Theorem 4.1. Some justification for the impossibility of such a conversion algorithm was given, due to the fact that the exponents could be long integers. However, we still do not have an algorithm for sparse polynomial to equal-spaced conversion under the (probably reasonable) restriction that all exponents be word-sized integers. A linear-time algorithm for this problem would be useful and would make our adaptive approach more complete, though slightly more restricted in scope.

Some early results from a trial implementation indicate that the algorithms we present are quite good at computing efficient adaptive representations, even in the presence of "noise" in the input polynomials, and although the conversion does sometimes have a measurable cost, it is almost always significantly less than the cost of the actual multiplication. This gives some evidence that our theoretical results hold in practice, but much more work on a more efficient implementation and much more testing is needed to investigate this claim. There are countless small "tricks" and tweaks to the algorithms that are likely to cause significant improvements in practice (without affecting the asymptotic complexities).

Another area for further development would be multivariate polynomials. The ideas we present here could be trivially extended to multivariate polynomials given a term ordering, but a "smarter" adaptive algorithm should be able to choose the best such ordering by investigating the structure of the input.

Finally, even though we have proven that our algorithms produce optimal adaptive representations, it is always under some restriction of the way that choice is made (for example, requiring to choose an "optimal chunk size" k first, and then compute optimal conversions given k). These results would be significantly strengthened by proving lower bounds over all available adaptive representations of a certain type, but such results have thus far been elusive.

Acknowledgement

The author would like to thank his supervisors, Mark Giesbrecht and Arne Storjohann, for their continuing support and guidance.

References

R. Boyer and J. Moore. A fast majority vote algorithm. Technical Report 1981-32, Institute for Computing Science, University of Texas, Austin, 1981.

Peter Bürgisser and Martin Lotz. Lower bounds on the bounded coefficient complexity of bilinear maps. *J. ACM*, 51(3):464–482 (electronic), 2004. ISSN 0004-5411.

David G. Cantor and Erich Kaltofen. On fast multiplication of polynomials over arbitrary algebras. *Acta Inform.*, 28(7):693–701, 1991. ISSN 0001-5903.

Stephen A. Cook. *On the Mininum Computation Time of Functions*. PhD thesis, Harvard University, 1966.

M. J. Fischer and S. L. Salzberg. Finding a majority among n votes: Solution to problem 81-5. *J. Algorithms*, 3(4):376–379, 1982.

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

Joachim von zur Gathen and Jürgen Gerhard. *Modern computer algebra*. Cambridge University Press, Cambridge, second edition, 2003. ISBN 0-521-82646-2.

Stephen C. Johnson. Sparse polynomial arithmetic. *SIGSAM Bull.*, 8(3):63–71, 1974. ISSN 0163-5824. doi: http://doi.acm.org/10.1145/1086837.1086847.

Erich Kaltofen and Wen-shin Lee. Early termination in sparse interpolation algorithms. *J. Symbolic Comput.*, 36(3-4):365–400, 2003. ISSN 0747-7171. International Symposium on Symbolic and Algebraic Computation (ISSAC'2002) (Lille).

A. Karatsuba and Yu. Ofman. Multiplication of multidigit numbers on automata. *Dokl. Akad. Nauk SSSR*, 7:595–596, 1963.

Kurt Mehlhorn. *Data structures and algorithms. 1*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1984. ISBN 3-540-13302-X. Sorting and searching.

Michael B. Monagan and Roman Pearce. Polynomial division using dynamic arrays, heaps, and packed exponent vectors. *Lecture Notes in Computer Science*, 4770:295–315, 2007. Computer Algebra in Scientific Computing (CASC'07).

Ola Petersson and Alistair Moffat. A framework for adaptive sorting. *Discrete Appl. Math.*, 59(2):153–179, 1995. ISSN 0166-218X.

Daniel S. Roche. Adaptive polynomial multiplication. In *Proc. Milestones in Computer Algebra (MICA '08)*, pages 65–72, 2008.

A. Schönhage and V. Strassen. Schnelle Multiplikation grosser Zahlen. *Computing (Arch. Elektron. Rechnen)*, 7:281–292, 1971.

A. L. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. *Dokl. Adad. Nauk. SSSR*, 150(3):496–498, 1963.

Thomas Yan. The geobucket data structure for polynomials. *J. Symbolic Comput.*, 25(3):285–293, 1998. ISSN 0747-7171.