Everybody who has analyzed the logical theory of computers has come to the conclusion that the possibilities of computers are very interesting — if they could be made to be more complicated by several orders of magnitude. If they had millions of times as many elements, they could make judgments. They would have time to calculate what is the best way to make the calculation that they are about to make.

—Richard Feynman, "There's plenty of room at the bottom" (1959)

Chapter 5

Adaptive multiplication

As we have seen, the multiplication of univariate polynomials is one of the most important operations in mathematical computing, and in computer algebra systems in particular. However, existing algorithms are closely tied to the standard dense and sparse representations. In this chapter, we present new algorithms for multiplication which allow the operands to be stored in either the dense or sparse representation, and automatically adapt the algorithm to the sparse structure of the input polynomials (in some sense). The result is a method that is never significantly slower than any existing algorithm, but is significantly faster for a large class of inputs.

Preliminary progress on some of these problems was presented at the Milestones in Computer Algebra (MICA) conference in 2008 (Roche, 2008). The algorithms presented here have also been accepted to appear in the Journal of Symbolic Computation (Roche, 2010).

5.1 Background

We start by reviewing the dense and sparse representations, as well as corresponding algorithms for multiplication that have been developed. Let $f \in \mathsf{R}[x]$ with degree less than n written as

$$f = c_0 + c_1 x + c_2 x^2 + \cdots + c_{n-1} x^{n-1}, \tag{5.1}$$

for $c_0, \ldots, c_{n-1} \in \mathsf{R}$. Recall that the dense representation of f is simply an array $[c_0, c_1, \ldots, c_{n-1}]$ of length n containing every coefficient in f.

Now suppose that at most t of the coefficients are nonzero, so that we can write

$$f = a_1 x^{e_1} + a_2 x^{e_2} + \cdots + a_t x^{e_t}, \tag{5.2}$$

for $a_1, \ldots, a_t \in \mathsf{R}$ and $0 \le e_1 < \cdots < e_t$. Hence $a_i = c_{e_i}$ for $1 \le i \le t$, and in particular $e_t = \deg f$. The sparse representation of f is a list of coefficient-exponent tuples $(a_1, e_1), \ldots, (a_t, e_t)$. This always requires $O(t)$ ring elements of storage (on the algebraic side of an algebraic IMM). However, the exponents in this case could be multi-precision integers, and the total size of their storage, on the integer side, is proportional to $\sum_i (1 + \log_2 e_i)$ bits. Assuming every exponent is stored to the same precision, this is $O(t \log n)$ bits in total. Since the integer exponents are not stored in bits but in machine words, this can be reduced by a logarithmic factor for scalable IMM problems, but we will ignore that technicality for the sake of clarity in the present discussion.

The previous chapter discussed algorithms for dense polynomial multiplication in depth. Recall that the $O(n^2)$ school method was first improved by Karatsuba and Ofman (1963) to $O(n^{1.59})$ with a two-way divide-and-conquer scheme, later generalized to k-way by Toom (1963) and Cook (1966). Building on results of Schönhage and Strassen (1971) and Schönhage (1977), Cantor and Kaltofen (1991) developed an algorithm to multiply polynomials with $O(n \log n \operatorname{loglog} n)$ operations in any algebra. This stands as the best asymptotic complexity for polynomial multiplication.

In practice, all of these algorithms will be used in certain ranges, and so we employ the usual notation of a *multiplication time* function $\mathsf{M}(n)$, the cost of multiplying two dense polynomials with degrees both less than n. Also define $\delta(n) = \mathsf{M}(n)/n$. If $f, g \in \mathsf{R}[x]$ with different degrees $\deg f < n$, $\deg g < m$, and $n > m$, by splitting f into $\lceil n/m \rceil$ size-m blocks we can compute the product $f \cdot g$ with cost $O(\frac{n}{m}\mathsf{M}(m))$, or $O(n \cdot \delta(m))$.

Algorithms for sparse polynomial multiplication were briefly discussed in Chapter 2. In this case, the school method also uses $O(t^2)$ ring operations, but for sparse polynomials this cannot be improved in the worst case. However, since the degrees could be very large, the cost of exponent arithmetic becomes significant. The school method uses $O(t^3 \log n)$ word operations and $O(t^2)$ space. Yan (1998) reduces the number of word operations to $O(t^2 \log t \log n)$ with the "geobuckets" data structure. Finally, recent work by Monagan and Pearce (2007), following Johnson (1974), gets this same time complexity but reduces the space requirement to $O(t + r)$, where r is the number of nonzero terms in the product.

The algorithms we present are for univariate polynomials. They can also be used for multivariate polynomial multiplication by using *Kronecker substitution*, as described previously in section 1.2.4: Given two n-variate polynomials $f, g \in \mathsf{R}[x_1, \ldots, x_n]$ with max degrees less than d, substitute $x_i = y^{(2d)^{i-1}}$ for $1 \le i \le n$, multiply the univariate polynomials over $\mathsf{R}[y]$, then convert back. Of course there are many other ways to represent multivariate polynomials, but we will not consider them further here.

5.2 Overview of Approach

The performance of an *adaptive algorithm* depends not only on the size of the input but also on some inherent difficulty measure. Such algorithms match standard approaches in their worst-case performance, but perform far better on many instances. This idea was first applied to sorting algorithms and has proved to be useful both in theory and in practice (see Petersson and Moffat, 1995). Such techniques have also proven useful in symbolic computation, for example the early termination strategy of Kaltofen and Lee (2003).

Hybrid algorithms combine multiple different approaches to the same problem to effectively handle more cases (e.g., Duran, Saunders, and Wan, 2003). Our algorithms are also hybrid in the sense that they provide a smooth gradient between existing sparse and dense multiplication algorithms. The adaptive nature of the algorithms means that in fact they will be faster than existing algorithms in many cases, while never being (asymptotically) slower.

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.

The second step is where the multiplication is actually performed, and it is this step alone that depends on the difficulty of the particular instance. Therefore this step should be the dominating cost of the entire algorithm, and in particular the cost of the conversion steps must be linear in the size of the input polynomials.

In Section 5.3, 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 5.4, is to write a polynomial with "equal spacing" between coefficients as a dense polynomial composed with a power of the indeterminate. Section 5.5 shows how to combine these two ideas to make one algorithm which effectively captures both difficulty measures. We examine how our algorithms perform in practice with some benchmarking against existing algorithms in Section 5.6. Finally, a few conclusions and ideas for future directions are discussed in Section 5.7.

5.3 Chunky Polynomials

The basic idea of chunky multiplication is a straightforward combination of the standard sparse and dense representations, providing a natural gradient between the two approaches for multiplication. We note that a similar idea was noticed (independently) around the same time by Fateman (2008, page 11), although the treatment here is much more extensive.

For $f \in \mathsf{R}[x]$ of degree n, the *chunky representation* of f is a sparse polynomial with dense polynomial "chunks" as coefficients:

$$f = f_1 x^{e_1} + f_2 x^{e_2} + \cdots + f_t x^{e_t}, \tag{5.3}$$

with $f_i \in \mathsf{R}[x]$ and $e_i \in \mathbb{N}$ for each $1 \le i \le t$. We require only that $e_{i+1} > e_i + \deg f_i$ for $1 \le i \le t-1$, and each f_i has nonzero constant coefficient.

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 $\delta(n)$ to be an increasing function which grows more slowly than linearly, meaning that for any $a, b, d \in \mathbb{N}$ with $a < b$,

$$\delta(a+d) - \delta(a) \geq \delta(b+d) - \delta(b). \tag{5.4}$$

These conditions are clearly satisfied for all the dense multiplication algorithms and corresponding $\mathsf{M}(n)$ functions discussed above, including the piecewise function used in practice.

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.

5.3.1 Multiplication in the chunky representation

Multiplying polynomials in the chunky representation uses sparse multiplication on the outer loop, treating each dense polynomial chunk as a coefficient, and dense multiplication to find each product of two chunks.

For $f, g \in \mathsf{R}[x]$ to be multiplied, write f as in (5.3) and g as

$$g = g_1 x^{d_1} + g_2 x^{d_2} + \cdots + g_s x^{d_s}, \tag{5.5}$$

with $s \in \mathbb{N}$ and similar conditions on each $g_i \in \mathsf{R}[x]$ and $d_i \in \mathbb{N}$ as in (5.3). Without loss of generality, assume also that $t \geq s$, that is, f has more chunks than g. To multiply f and g, we need to compute each product $f_i g_j$ for $1 \leq i \leq t$ and $1 \leq j \leq s$ and put the resulting chunks into sorted order. It is likely that some of the chunk products 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 s is filled with pairs (i, j), for $i, j \in \mathbb{N}$, and ordered by the corresponding sum of exponents $e_i + d_j$. Each time we compute a new chunk product $f_i \cdot g_j$, 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 5.1.

After using this algorithm 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 5.1: Chunky Multiplication

```
Input: $f, g$ as in (5.3) and (5.5)
Output: The product $f \cdot g = h$ in the chunky representation
1  $\alpha \leftarrow f_1 \cdot g_1$ using dense multiplication
2  $b \leftarrow e_1 + d_1$
3  $H \leftarrow$ min-heap with pairs $(1, j)$ for $j = 2, 3, \ldots, s$, ordered by exponent sums
4  if $i \geq 2$ then insert $(2, 1)$ into $H$
5  while $H$ is not empty do
6      $(i, j) \leftarrow$ extract-min$(H)$
7      $\beta \leftarrow f_i \cdot g_j$ using dense multiplication
8      if $b + \deg \alpha < e_i + d_j$ then
9          write $\alpha x^b$ as next term of $h$
10         $\alpha \leftarrow \beta$;    $b \leftarrow e_i + d_j$
11     else $\alpha \leftarrow \alpha + \beta x^{e_i + d_j - b}$ stored as a dense polynomial
12     if $i < t$ then insert $(i + 1, j)$ into $H$
13 write $\alpha x^b$ as final term of $h$
```

Theorem 5.1. *Algorithm 5.1 correctly computes the product of f and g using*

$$O\Bigg(\sum_{\substack{\deg f_i \geq \deg g_j \\ 1 \leq i \leq t,\ 1 \leq j \leq s}} (\deg f_i) \cdot \delta(\deg g_j) \quad + \sum_{\substack{\deg f_i < \deg g_j \\ 1 \leq i \leq t,\ 1 \leq j \leq s}} (\deg g_j) \cdot \delta(\deg f_i) \Bigg)$$

ring operations and $O(ts \cdot \log s \cdot \log(\deg fg))$ *word operations.*

Proof. Correctness is clear from the definitions. The bound on ring operations comes from Step 7 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 cost of word operations is incurred in removing from and inserting to the heap on Steps 6 and 12. 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.

5.3.2 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 5.1. Here we consider a simpler problem, namely determining an optimal chunky representation for f given that g has only one chunk of size k.

The following corollary comes directly from Theorem 5.1 and will guide our conversion algorithm on this step.

Corollary 5.2. *Given $f \in \mathsf{R}[x]$ as in (5.3), 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 any high-degree chunk (i.e., $\deg f_i \geq k$), we see that there is no benefit to making the chunk any larger, as 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 5.2 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 were truncated at that gap. This is accomplished 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 $a_i, b_i \in \mathbb{Z}$ for $0 \leq i \leq m$ be the sizes of each consecutive "gap" of zero coefficients and "block" of nonzero coefficients, in order. Each a_i and b_i will be nonzero except possibly for a_0 (if f has a nonzero constant coefficient), and $\sum_{0 \leq i \leq m}(a_i + b_i) = \deg f + 1$. For example, the polynomial

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

has $a_0 = 10, b_0 = 2, a_1 = 1, b_1 = 1, a_2 = 5$, and $b_2 = 3$.

Finally, reusing notation from the previous subsection, define d_i to be the size of the polynomial up to (not including) gap i, i.e., $d_i = \sum_{0 \leq j < i}(a_j + b_j)$. And define e_i to be the size *including* gap i, i.e., $e_i = d_i + a_i$.

For the gap at index ℓ, for $1 \leq \ell \leq m$, we store the optimal chunky conversion of $f \bmod x^{d_\ell}$ 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, in c_ℓ, the cost in ring operations of multiplying $f \bmod x^{d_\ell}$ (in this optimal representation) by a single chunk of size k, scaled by $1/k$. In particular, since from the discussion above we can conclude that every chunk in this optimal representation has size at most k, and writing s_i for the size of the i'th chunk in the optimal chunky representation of $f \bmod x^{d_\ell}$, c_ℓ is equal to $\sum \delta(s_i)$.

The conversion algorithm works by considering, for each ℓ, the gap of highest degree that should be included in the optimal chunky representation of $f \bmod x^{d_\ell}$. If this gap has index

i, then from above we have $c_\ell = c_i + \delta(d_\ell - e_i)$. This is because the optimal chunky representation will consist exactly of the chunks in $f \bmod x^{d_i}$, followed by the last chunk, which goes from gap i to gap ℓ and has size $d_\ell - e_i$.

Hence to determine the optimal chunky conversion of $f \bmod x_{d_\ell}$ we must find an index $i < \ell$ that minimizes $c_i + \delta(d_\ell - e_i)$. To see how this search is performed efficiently, first observe that, from the discussion above, we need not consider chunks of size greater than k, and therefore at this stage indices i where $d_\ell - e_i > k$ may be excluded. Furthermore, from (5.4), we know that, if $1 \le i < j < \ell$ and $c_i + \delta(d_\ell - e_i) < c_j + \delta(d_\ell - e_j)$, then this same inequality continues to hold as ℓ 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.

Thus we can essentially precompute the values of $\min_{i<\ell}(c_i + \delta(d_\ell - e_i))$ by maintaining a stack of index-index pairs. A pair (i, j) of indices on the top of the stack indicates that $c_i + \delta(d_\ell - e_i)$ is minimal as long as $\ell \le j$. The second pair of indices indicates the minimal value from gap j 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. Observe that the first index decreases and the second index increases as we move from the top of the stack to the bottom. The initial pair at the bottom of the stack is $(0, m+1)$, indicating that gap 0, corresponding to the power of x that divides f (if any), is always beneficial to take in the chunky representation.

The details of this rather complicated algorithm are given in Algorithm 5.2.

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

The **while** loop on Step 5 removes all gaps from the stack which are no longer relevant, either because their cost is now beaten by a previous gap (when $j < \ell$), or because the size of the resulting chunk would be greater than k and therefore unnecessary to consider.

If the condition of Step 7 is true, then there is no index at which gap $(\ell - 1)$ should be used, so we discard it.

Otherwise, the gap at index $\ell - 1$ is good at least some of the time, so we proceed to the task of determining the largest gap index v at which gap $(\ell - 1)$ might still be useful. First, in Steps 14–16, we repeatedly check whether gap $(\ell - 1)$ always beats the gap at the top of the stack S, and if so remove it. After this process, either no gaps remain on the stack, or we have a range $r \le v \le j$ in which binary search can be performed to determine v.

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 of course just f itself.

Theorem 5.3. *Algorithm 5.2 returns the optimal chunky representation for multiplying f by a dense size-k chunk. The running time of the algorithm is linear in the size of the input representation of f.*

Proof. Correctness follows from the discussions above.

Algorithm 5.2: 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  $L_0 \leftarrow$ (empty);   $L_1 \leftarrow 0$
2  $c_0 \leftarrow 0$;   $c_1 \leftarrow \delta(b_0)$
3  $S \leftarrow (0, m+1)$
4  for $\ell = 2, 3, \ldots, m+1$ do
5      while top pair $(i, j)$ from $S$ satisfies $j < \ell$ or $d_\ell - e_i > k$ do
6          Remove $(i, j)$ from $S$
7      if $S$ is not empty and the top pair $(i, j)$ from $S$ satisfies
       $c_i + \delta(d_\ell - e_i) \le c_{\ell-1} + \delta(d_\ell - e_{\ell-1})$ then
8          $L_\ell \leftarrow L_i,\ i$
9          $c_\ell \leftarrow c_i + \delta(d_\ell - e_i)$
10     else
11         $L_\ell \leftarrow L_{\ell-1},\ (\ell - 1)$
12         $c_\ell \leftarrow c_{\ell-1} + \delta(d_\ell - e_{\ell-1})$
13         $r \leftarrow \ell$
14         while top pair $(i, j)$ from $S$ satisfies $c_i + \delta(d_j - e_i) > c_{\ell-1} + \delta(d_j - e_{\ell-1})$ do
15             $r \leftarrow j$
16             Remove $(i, j)$ from $S$
17         if $S$ is empty then $(i, j) \leftarrow (0, m+1)$
18         else $(i, j) \leftarrow$ top pair from $S$
19         $v \leftarrow$ least index with $r \le v < j$ s.t. $c_{\ell-1} + \delta(d_v - e_{\ell-1}) > c_i + \delta(d_v - e_i)$
20         Push $(\ell - 1, v)$ onto the top of $S$
21 return $L_{m+1}$
```

For the complexity analysis, first note 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 in f. Clearly m is less than the number of nonzero terms in f, so this is bounded above by the sparse or dense representation size. If the lists L_i are implemented as singly-linked lists, sharing nodes, then the total extra storage for the algorithm is $O(m)$.

The total number of iterations of the two **while** loops corresponds to the number of gaps that are removed from the stack S at any step. Since at most one gap is pushed onto S at each step, the total number of removals, and hence the total cost of these **while** loops over all iterations, is $O(m)$.

Now consider the cost of Step 19 at each iteration. If the input is given in the sparse representation, we just perform a binary search on the interval from r to j, for a total cost of $O(m \log m)$ over all iterations. Because m is at most the number of nonzero terms in f, $m \log m$ is bounded above by the sparse representation size, so the theorem is satisfied for sparse input.

When the input is given in the dense representation, a binary search is again used on Step 19, but we start with a one-sided binary search, also called "galloping" search. This search starts from either r or j, depending on which interval endpoint v is closest to. The cost of this search is at a single iteration is $O(\log \min\{v - r, i_2 - v\})$. Notice that the interval (r, j) 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.

More precisely, a loose upper bound in the worst case of the total cost over all iterations is $O(\sum_{i=1}^{u} 2^i \cdot (u - i + 1))$, where $u = \lceil \log_2 m \rceil$. This is less than 2^{u+2}, which is $O(m)$, giving linear cost in the size of the dense representation. □

5.3.3 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.

If f is written in the chunky representation as in (5.3), there are many possible choices for the number of chunks t, depending on how large the chunks are. So define $t(k)$ to be the least number of chunks if each chunk has size at most k, i.e., $\deg f_i < k$ for $1 \le i \le t(k)$. Similarly define $s(k)$ for $g \in \mathsf{R}[x]$ written as in (5.5).

Therefore, from the cost of multiplication in Theorem 5.1, in this part we want to compute the value of k that minimizes

$$t(k) \cdot s(k) \cdot k \cdot \delta(k). \tag{5.6}$$

Say $\deg f = n$. After $O(n)$ preprocessing work (making pointers to the beginning and end of each "gap"), $t(k)$ 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 (5.6) above: simply compute (5.6) for each possible $k = 1, 2, \ldots, \max\{\deg f, \deg g\}$. This naïve approach is too costly for our purposes, but underlies the basic idea of our algorithm.

Rather than explicitly computing each $t(k)$ and $s(k)$, 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 gives a tight approximation to the actual values of $t(k)$ and $f(k)$, while achieving 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 $3 + 35 + 5 = 43$, More precisely, if f is written as in (5.3), then the i^{th} gap has key value

$$\deg f_{i+1} + e_{i+1} - e_i + 1. \tag{5.7}$$

Each gap in the priority queue also contains pointers to the two (or fewer) neighboring gaps in the current chunky representation. Removing a gap from the queue corresponds to 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 (5.6).

Since the purpose here 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. Algorithm 5.3 gives the details of this computation. Note in particular that Q_f and Q_g initially contain *every* gap — even empty ones — so that $\left|Q_f\right| = t_f - 1$ and $\left|Q_g\right| = t_g - 1$ initially.

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 might imagine encountering chunk sizes that are larger than the largest word-sized integer. But as discussed previously, such a chunk size would be meaningless 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

Algorithm 5.3: Optimal Chunk Size Computation

```
Input: $f, g \in \mathsf{R}[x]$
Output: $k \in \mathbb{N}$ that minimizes $t(k) \cdot s(k) \cdot k \cdot \delta(k)$
1  $Q_f, Q_g \leftarrow$ minimum priority queues initialized with all gaps in $f$ and $g$, respectively
2  $k, k_{\min} \leftarrow 1$;    $c_{\min} \leftarrow t_f t_g$
3  while $Q_f$ and $Q_g$ are not both empty do
4      $k \leftarrow$ smallest key value from $Q_f$ or $Q_g$
5      while $Q_f$ has an element with key value $\leq k$ do
6          Remove a $k$-valued gap from $Q_f$ and update neighbors
7      while $Q_g$ has an element with key value $\leq k$ do
8          Remove a $k$-valued gap from $Q_g$ and update neighbors
9      $c_{\text{current}} \leftarrow (|Q_f| + 1) \cdot (|Q_g| + 1) \cdot k \cdot \delta(k)$
10     if $c_{\text{current}} < c_{\min}$ then
11         $k_{\min} \leftarrow k$;    $c_{\min} \leftarrow c_{\text{current}}$
12 return $k_{\min}$
```

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, using $O(\deg f + \deg g)$ 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 4, 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 4 for all iterations is $O(\deg f + \deg g)$.

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

Lemma 5.4. *At any iteration through Step 10 in Algorithm 5.3, $|Q_f| < 2t(k)$ and $|Q_g| < 2s(k)$.*

Proof. First consider f. There are two chunky representations with each chunk of degree less than k to consider: the optimal having $t(k)$ chunks and the one implicitly computed by Algorithm 5.3 with $|Q_f| + 1$ chunks. Call these $\bar{f}$ and $\hat{f}$, respectively.

We claim that any single chunk of the optimal $\bar{f}$ contains at most three constant terms of chunks in the implicitly-computed $\hat{f}$. If this were not so, then two chunks in $\hat{f}$ could be combined to result in a single chunk with degree less than k. But this is impossible, since all such pairs of chunks would already have been merged after the completion of Step 5.

Therefore every chunk in $\bar{f}$ contains at most two constant terms of distinct chunks in $\hat{f}$. Since each constant term of a chunk is required to be nonzero, the number of chunks in $\hat{f}$ is at most twice the number in $\bar{f}$. Hence $|Q_f| + 1 \leq 2t(k)$. An identical argument for g gives the stated result. □

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

Theorem 5.5. *Algorithm 5.3 computes a chunk size k such that $t(k) \cdot s(k) \cdot k \cdot \delta(k)$ is at most 4 times the minimum value. The worst-case cost of the algorithm is linear 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 (5.6), 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 5.4. So the first statement of the theorem holds.

Write c for the total number of nonzero terms in f and g. The initial sizes of the queues Q_f and Q_g is $O(c)$. Since gaps are only removed from the queues (after they are initialized), the total cost of all queue operations is bounded above by $O(c)$, which in turn is bounded above by the sparse and dense sizes of the input polynomials.

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

5.3.4 Chunky Multiplication Overview

Now we are ready to examine the whole process of chunky polynomial conversion and multiplication. First we need the following easy corollary of Theorem 5.3.

Corollary 5.6. *Let $f \in \mathsf{R}[x]$, $k \in \mathbb{N}$, and $\hat{f}$ be any chunky representation of f where all chunks have degree at least k, and $\bar{f}$ be the representation returned by Algorithm 5.2 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 5.2 on input ℓ. We know from Theorem 5.3 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 5 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 5.3.
2. Compute chunky representations of f and g using Algorithm 5.2 with input k.
3. Multiply the two chunky representations using Algorithm 5.1.
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 5.7. *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 5.3, the values of $t(k) \cdot s(k) \cdot k \cdot \delta(k)$ 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 5.3, and $\bar{f}$ and $\bar{g}$ the representations returned from Algorithm 5.2 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 5.3. Finally, since every chunk in $\bar{f}$ has size at most k, Corollary 5.6 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. □

5.4 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).

One motivating example suggested by Michael Monagan is that of homogeneous polynomials. Recall that a multivariate polynomial $h \in \mathsf{R}[x_1, \ldots, x_n]$ is *homogeneous of degree d* if every nonzero term of h has total degree d. It is well-known that the number of variables in a homogeneous polynomial can be effectively reduced by one by writing $y_i = x_i/x_n$ for $1 \le i < n$ and $h = x_n^d \cdot \hat{h}$, for $\hat{h} \in \mathsf{R}[y_1, \ldots, y_{n-1}]$ an $(n-1)$-variate polynomial with max degree d. This leads to efficient schemes for homogeneous polynomial arithmetic.

But this is only possible if (1) the user realizes this structure in their polynomials, and (2) every polynomial used is homogeneous. Otherwise, a more generic approach will be used, such as the Kronecker substitution mentioned in the introduction. Choosing some integer $\ell > d$, we evaluate $h(y, y^\ell, y^{\ell^2}, \ldots, y^{\ell^{n-1}})$, and then perform univariate arithmetic over $\mathsf{R}[y]$. But if h is homogeneous, a special structure arises: every exponent of y is of the form $d+i(\ell-1)$ for some integer $i \ge 0$. Therefore we can write $h(y, \ldots, y^{\ell^{n-1}}) = (\bar{h} \circ y^{\ell-1}) \cdot y^d$, for some $\bar{h} \in \mathsf{R}[y]$ with much smaller degree. The algorithms presented in this section will automatically recognize this structure and perform the corresponding optimization to arithmetic.

The key idea is *equal-spaced representation*, which corresponds to writing a polynomial $f \in \mathsf{R}[x]$ as

$$f = (f_D \circ x^k) \cdot x^d + f_S, \tag{5.8}$$

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 $ik + d$ for some $i \ge 0$.

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 5.5, 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.

5.4.1 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 (5.8). 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 efficiently computed using dense multiplication. 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 5.4 gives the details of this method.

Algorithm 5.4: Equal Spaced Multiplication

```
   Input: $f = (f_D \circ x^k) \cdot x^d + f_S$,   $g = (g_D \circ x^\ell) \cdot x^e + g_S$,
   with $f_D = a_0 + a_1 x + a_2 x^2 + \cdots$,   $g_D = b_0 + b_1 x + b_2 x^2 + \cdots$
   Output: The product $f \cdot g$
1  $r \leftarrow \gcd(k, \ell)$,   $s \leftarrow \mathrm{lcm}(k, \ell)$
2  for $i = 0, 1, \ldots, s/k - 1$ do
3      $f_i \leftarrow a_i + a_{s+i} x + a_{2s+i} x^2 + \cdots$
4  for $i = 0, 1, \ldots, s/\ell - 1$ do
5      $g_i \leftarrow b_i + b_{s+i} x + b_{2s+i} x^2 + \cdots$
6  $h_D \leftarrow 0$
7  for $i = 0, 1, \ldots, s/k - 1$ do
8      for $j = 0, 1, \ldots, s/\ell - 1$ do
9          Compute $f_i \cdot g_j$ by dense multiplication
10         $h_D \leftarrow h_D + ((f_i \cdot g_j) \circ x^s) \cdot x^{ik + j\ell}$
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 $h_D \cdot x^{e+d} + (f_D \circ x^k) \cdot g_S \cdot x^d + (g_D \circ x^\ell) \cdot f_S \cdot x^e + f_S \cdot g_S$
```

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 5.8. *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 5.4 correctly computes the product $f \cdot g$ using*

$$O\Big((n/r) \cdot \delta(m/s) + n t_g/k + m t_f/\ell + t_f t_g\Big)$$

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 $O(n t_g/k + m t_f/\ell + t_f t_g)$ ring operations, giving the last three terms in the complexity measure.

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 any additions were performed, we would have

$$i_1 k + j_1 \ell \equiv i_2 k + j_2 \ell \pmod{s}$$

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 $\ell | s$, 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 the equal-spaced multiplication algorithm to standard dense multiplication, and will be used to guide the approach to conversion below.

Theorem 5.9. *Let f, g, m, n, t_f, t_g be as before. Algorithm 5.4 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 5.4 is

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

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. □

5.4.2 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 5.9 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 $t \le \deg f + 1$ 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 5.10. *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*

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

Proof. Without loss of generality, order the e_i's so that $0 \le e_1 < e_2 < \cdots < e_t \le n$. 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 $e_j - e_i \ge k$.

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 $(t - 2\log_2 t - 1) \cdot k \le n$. The stated result follows. $\square$

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 (1991) 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. In practice, this is accomplished without actually modifying the list.

Algorithm 5.5: Equal Spaced Conversion

```
Input: Exponents $e_1, e_2, \dots, e_t \in \mathbb{N}$ and $n \in \mathbb{N}$ such that $0 \le e_1 < e_2 < \cdots < e_t = n$
Output: $k, d \in \mathbb{N}$ and $S \subseteq \{e_1, \dots, e_t\}$ such that $e_i \equiv d \bmod k$ for all exponents $e_i$ not in $S$,
        and $|S| \le \log_2 t$.
1  if $t < 32$ then $k \leftarrow n$
2  else $k \leftarrow \lfloor n/(t-1-2\log_2 t) \rfloor$
3  while $k \ge 2$ do
4      $d \leftarrow e_1 \bmod k$;   $j \leftarrow 1$
5      for $i = 2, 3, \dots, t$ do
6          if $e_i \equiv d \bmod k$ then $j \leftarrow j+1$
7          else if $j > 0$ then $j \leftarrow j-1$
8          else $d \leftarrow e_i \bmod k$;   $j \leftarrow 1$
9      $S \leftarrow \{e_i : e_i \not\equiv d \bmod k\}$
10     if $|S| \le \log_2 t$ then return $k$, $d$, $S$
11     $k \leftarrow k-1$
12 return 1, 0, $\emptyset$
```

Given k, d, S from the algorithm, in one more pass through the input polynomial, f_D and f_S are constructed such that $f = (f_D \circ x^k) \cdot x^d + f_S$. After performing separate conversions for two polynomials $f, g \in \mathsf{R}[x]$, they are multiplied using Algorithm 5.4.

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

Theorem 5.11. *Given integers $e_1, \dots, e_t$ and n, with $t > 4$, Algorithm 5.5 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 $\{e_i \bmod k : i = 1, 2, \dots, t\}$, if there is one. 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 5.10, 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 analysis, 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, and this happens $O(t)$ times in each iteration.

If $t < 32$, each of the $O(n)$ iterations has constant cost, for total cost $O(n)$.

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 5.5 can be implemented using only $O(t)$ space for the storage of the exponents $e_1,\ldots,e_t$, which is linear in the size of the output, plus the space required for the returned set S.

5.5 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. As before, we seek an *adaptive* combination of previous algorithms, so that the combination is never asymptotically worse than either original idea.

An obvious approach would be to first perform chunky polynomial conversion, and then equal-spaced conversion on each of the dense chunks. Unfortunately, this would be asymptotically less efficient than equal-spaced multiplication alone in a family of instances, and therefore is not acceptable as a proper adaptive algorithm.

The algorithm presented here does in fact perform chunky conversion first, but instead of performing equal-spaced conversion on each dense chunk independently, Algorithm 5.5 is run simultaneously on all chunks in order to determine, for each polynomial, a single spacing parameter k that will be used for every chunk.

Let $f = f_1x^{e_1}+f_2x^{e_2}+\cdots+f_tx^{e_t}$ in the optimal chunky representation for multiplication by another polynomial g. We first compute the smallest bound on the spacing parameter k for any of the chunks f_i, using Lemma 5.10. Starting with this value, we execute the **while** loop of Algorithm 5.5 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 (recycling the variables f_i and e_i) 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 5.5 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 5.1 to compute the product $f \cdot g$, with the difference that each product $f_i \cdot g_j$ is computed by Algorithm 5.4 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 5.12. *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 5.5.*

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 5.8 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 5.5 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 5.5.

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 5.8.

Now recall the cost equation (5.6) used for Algorithm 5.3:

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

where b is the size of all dense chunks in f and g. By definition, $c_f(n) = 1$, and $c_g(n) \le m/n$, so we know that $c_f(n)\, c_g(n)\, n\, \delta(n) \le m\, \delta(n)$. Because the chunk sizes d_i were originally chosen by Algorithm 5.3, we must therefore have $m \sum_{i=1}^{t} \delta(d_i) \le m\delta(n)$. 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 $m \le n$, the number of ring operations to compute the product using the combined method, again by Theorem 5.8, is

$$O\left(\delta(m/s)\sum_{d_i\ge m}(d_i/r)+(m/r)\sum_{d_i<m}\delta(d_i/s)\right), \tag{5.9}$$

compared with $O((n/r)\delta(m/s))$ for the standard equal-spaced algorithm. Because we always have $\sum_{i=1}^{t} d_i \le n$, the first term of (5.9) is $O((n/r)\delta(m/s))$. Using again the inequality $m\sum_{i=1}^{t}\delta(d_i) \le m\delta(n)$, along with the fact that $m\delta(n) \in O(n\delta(m))$ because $m \le n$, we see that the second term of (5.9) 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. □

5.6 Implementation and benchmarking

We implemented the full chunky conversion and multiplication algorithm for dense polynomials in MVP. We compare the results against the standard dense and sparse multiplication algorithms in MVP. Previously, in sections 2.5 and 4.4, we saw that the multiplication routines in MVP are competitive with existing software. Therefore our comparison against these routines is fair and will provide useful information on the utility of the new algorithms.

Recall that the chunky conversion algorithm requires a way to approximate $\delta(n)$ for any integer n. For this, we actually used the timing results reported in Section 4.4 for dense polynomial multiplication in MVP on our testing machine. Using this data and the known crossover points, we did a least-squares fit to the three curves

$$\begin{aligned}
n\delta_1(n) &= a_1 n^2 + b_1 n + c_1\\
n\delta_2(n) &= a_2 n^{\log_2 3} + b_2 n + c_2\\
n\delta_3(n) &= a_3 2^{\lceil \log_2 n\rceil}\lceil \log_2 n\rceil + b_3 n + c_3,
\end{aligned}$$

for the ranges of classical, Karatsuba, and FFT-based multiplication, respectively. Clearly this could be automated as part of a tuning phase, but this has not yet been implemented. However, observe that as only the *relative* times are important, it is possible that some default timing data could still be useful on different machines.

Choosing benchmarks for assessing the performance of these algorithms is a challenging task. Our algorithms adapts to chunkiness in the input, but this measure is difficult to quantify. Furthermore, while we can (and do) generate examples that are more or less chunky, it is difficult to justify whether such polynomials actually appear in practice.

With these reservations in mind, we proceed to describe the benchmarking trials we performed to compare chunky multiplication against the standard dense and sparse algorithms. First, to generate random polynomials with some varying "chunkiness" properties, we used the following approach. The parameters to the random polynomial generation are the usual degree n and desired sparsity t, as well as a third parameter c, which should be a small positive integer. The polynomials were then generated by randomly choosing t exponents, successively, and for each exponent choosing a random nonzero coefficient. To avoid trivially easy cases, we always set the constant coefficient and the coefficient of x^d to be nonzero.

Chunkiness c	Standard sparse	Chunky multiplication
0	.981	1.093
10	.969	1.059
20	.984	1.075
25	.899	.971
30	.848	1.262
32	.778	.771
34	.749	.313
36	.729	.138
38	.646	.084
40	.688	.055

Table 5.1: Benchmarks for adaptive multiplication with varying chunkiness

However, to introduce "chunkiness" as c grows larger, the exponents of successive terms were not always chosen independently. With probability proportional to $1/c$, the next exponent was chosen independently and randomly from $\{1, 2, \ldots, d-1\}$. However, with high probability proportional to $(c-1)/c$, the next successive exponent was chosen to be "close" to the previous one. Specifically, in this case the next exponent was chosen randomly from a set of size proportional to $n/2^c$ surrounding the previously-chosen exponent. This deviation from independence has the effect of creating clusters of nonzero coefficients, when the parameter c is sufficiently large.

We make no claims that this randomization has any basis in a particular application, but point out that it does provide more of a challenge to our algorithms than the simplistic approach of clustering nonzero terms into completely dense chunks with no intervening gaps. Also observe that when $c = 0$, we simply have a sparse polynomial with randomly-chosen support and no particular chunky structure. Furthermore, when c is on the order of $\log_2 n$, the generated polynomial is likely to consist of only a few very dense chunks. So by varying c in this range, we see the performance of the chunky multiplication algorithm on a range of cases.

Specifically, for our benchmarks we fixed the degree d at one million and the sparsity t at 3000. With these parameters, the usual dense and sparse algorithms in MVP have roughly the same cost. We then ran a number of tests varying the chunkiness parameter c between 0 and 40. In every case, the dense algorithm used close to 1.91 CPU seconds for a single operation. The timings reported in Table 5.1 are relative to this time, the cost of the dense multiplication algorithm. As usual, a number less than one means that algorithm was faster than the dense algorithm for the specified choice of c. Observe that although the sparsity t was invariant for all test cases, the normal sparse algorithm still derives some benefit from polynomials with high chunkiness, simply because of the reduced number of terms in the output.

With the slight anomaly at $c = 30$ (probably due to some algorithmic "confusion" and inaccuracy in $\delta(n)$), we see that the algorithm performs quite well, and as expected according to our theoretical results. In particular, the chunky multiplication algorithm is always either very nearly the same as existing methods, or much better than them. We also report, perhaps surprisingly, that the overhead associated with the chunky conversion is negligible in nearly every case. In fact, we can see this in the first few benchmarks with small c, where the chunky representation produced is simply the dense representation; the performance hit for the unnecessary conversion, at least in these cases, is less than 10 percent.

5.7 Conclusions and Future Work

Two methods for adaptive polynomial multiplication have been given where we can compute optimal representations (under some set of restrictions) in linear time in the size of the input. Combining these two ideas into one algorithm inherently captures both measures of difficulty, and will 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 5.12. 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.

On the subject of word size and exponents, we assumed at the beginning that $O(t \log n)$ words are required to store the sparse representation of a polynomial. This fact is used to justify that some of the conversion algorithms with $O(t \log t)$ complexity were in fact linear-time in the size of the input. However, the original assumption is not very good for many practical cases, e.g., where all exponents fit into single machine words. In such cases, the input size is $O(t)$, but the conversion algorithms still use $O(t \log t)$ operations, and hence are not linear-time. We would argue that they are still useful, especially since the sparse multiplication algorithms are quadratic, but there is certainly room for improvement in the sparse polynomial conversion algorithms.

Yet another area for further development is multivariate polynomials. We have mentioned the usefulness of Kronecker substitution, but developing an adaptive algorithm to choose the optimal variable ordering would give significant improvements. An interesting observation is that using the chunky conversion under a dense Kronecker substitution seems similar in many ways to the recursive dense representation of a multivariate polynomial. Examining the connections here more carefully would be interesting, and could give more justification to the practical utility of the algorithms we have presented.

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.

Bibliography

Robert S. Boyer and J. Strother Moore. MJRTY — a fast majority vote algorithm. In Robert S. Boyer, editor, *Automated Reasoning: Essays in Honor of Woody Bledsoe*, Automated Reasoning, pages 105–117. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1991. URL http://www.cs.utexas.edu/~moore/best-ideas/mjrty/index.html. Referenced on page 86.

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 70.

Stephen Arthur Cook. *On the minimum computation time of functions*. PhD thesis, Harvard University, 1966. Referenced on page 70.

Ahmet Duran, B. David Saunders, and Zhendong Wan. Hybrid algorithms for rank of sparse matrices. In *Proc. SIAM Conf. on Appl. Linear Algebra*, 2003. Referenced on page 71.

Richard Fateman. Draft: What's it worth to write a short program for polynomial multiplication? Online, December 2008. URL http://www.cs.berkeley.edu/~fateman/papers/shortprog.pdf. Referenced on page 71.

M. J. Fischer and S. L. Salzberg. Finding a majority among n votes: Solution to problem 81-5. *Journal of Algorithms*, 3(4):376–379, 1982. ISSN 0196-6774. doi: 10.1016/0196-6774(82)90031-1. Referenced on page 86.

Stephen C. Johnson. Sparse polynomial arithmetic. *SIGSAM Bull.*, 8:63–71, August 1974. ISSN 0163-5824. doi: 10.1145/1086837.1086847. Referenced on page 70.

Erich Kaltofen and Wen-shin Lee. Early termination in sparse interpolation algorithms. *Journal of Symbolic Computation*, 36(3-4):365–400, 2003. ISSN 0747-7171. doi: 10.1016/S0747-7171(03)00088-9. ISSAC 2002. Referenced on page 71.

A. A. Karatsuba and Yu. Ofman. Multiplication of multidigit numbers on automata. *Doklady Akademii Nauk SSSR*, 7:595–596, 1963. Referenced on page 70.

Michael Monagan and Roman Pearce. Polynomial division using dynamic arrays, heaps, and packed exponent vectors. In Victor Ganzha, Ernst Mayr, and Evgenii Vorozhtsov, editors,

Computer Algebra in Scientific Computing, volume 4770 of *Lecture Notes in Computer Science*, pages 295–315. Springer Berlin / Heidelberg, 2007.
doi: 10.1007/978-3-540-75187-8_23. Referenced on pages 70 and 72.

Ola Petersson and Alistair Moffat. A framework for adaptive sorting. *Discrete Applied Mathematics*, 59(2):153–179, 1995. ISSN 0166-218X.
doi: 10.1016/0166-218X(93)E0160-Z. Referenced on page 70.

Daniel S. Roche. Adaptive polynomial multiplication. In *Proc. Milestones in Computer Algebra (MICA)*, pages 65–72, 2008. Referenced on page 69.

Daniel S. Roche. Chunky and equal-spaced polynomial multiplication. *Journal of Symbolic Computation*, In Press, Accepted Manuscript:–, 2010. ISSN 0747-7171.
doi: 10.1016/j.jsc.2010.08.013. Referenced on page 69.

A. Schönhage and V. Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7:281–292, 1971. ISSN 0010-485X.
doi: 10.1007/BF02242355. Referenced on page 70.

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 70.

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 70.

Thomas Yan. The geobucket data structure for polynomials. *Journal of Symbolic Computation*, 25(3):285–293, 1998. ISSN 0747-7171.
doi: 10.1006/jsco.1997.0176. Referenced on page 70.