DRAFT 11: What's it worth to write a short program for polynomial multiplication?

Richard Fateman

December 8, 2010

Abstract

One of the major advantages of writing computer programs in a higher-level language is that the programs tend to be shorter, easier to write, read, modify, and debug. Numerous programs for multiplication of univariate polynomials over the integers are available in open-source packages. What is the smallest expression in a programming language for such a program? (Assuming that the language does not have "multiplication of polynomials" as a primitive!) Is it possible that this program runs acceptably fast? How much elaboration is required to make it faster? How much elaboration is required to make it work for multivariate polynomials? Or coefficient domains other than integers? For concreteness, we provide programs in Lisp, and we also assume a certain representation that may or may not be best for any given application.

1 The task

Abstractly we can consider a polynomial as a list of coefficient-exponent pairs. For example, $7 + 9x + 34x^5$ might be `((0 . 7)(1 . 9)(5 . 34))` For concreteness we have decided that the list will be in descending order of exponent, and the pairs will be ordered as (exponent . coefficient). Furthermore, in order to avoid a situation in which two "equal" polynomials will have different representations, we require that only non-zero coefficients are recorded. The value 0 is represented by an empty list.

The task is to write a program that takes two such polynomials and returns a new polynomial which is their product, in the same form, fast.

Complicating the mapping of this abstraction into a conventional linked list is that one does not ordinarily have easy $O(1)$ access to the length of the list[1] Writing in Lisp, where a linked list is easily manipulated, a programmer would ordinarily use lists for all intermediate and final results. As we shall see, this is not necessarily efficient in time or space. Fortunately, lists can usually be converted to (and from) other useful representations in time linear in the length of the inputs. One representation is as a pair of vectors: we separate the coefficient vector and the exponent vector, considering that the exponents will all be integers, but the coefficients might be any sort of number. In the case of the example above, still ordering the terms from low to high degree, would be ([0 1 5], [7 9 34]).

Among other plausible representations, we can store the coefficients *only* and let the exponents be implicit in the order. We then have to represent zero coefficients. For our example: [7 9 0 0 0 34]. If there are large gaps, this representation requires large sequences of zeros and is potentially very uneconomical. This hazard of "dense" representation must be balanced against the simplicity and speed of accessing coefficients: it is trivial in such a representation to retrieve any coefficient or store a newly-computed replacement coefficient: Just index into that vector.

At this point let us elaborate on the nature of coefficients in a computer algebra system intending to "do arithmetic right." It is necessary to consider the possibility that the set of coefficients may include very large numbers that do not fit into a single computer "word". Indeed, some tasks require the manipulation of exact integers of hundreds or thousands of digits. It may also be useful to consider other types of coefficients, typically rational numbers, hardware or software floating-point numbers, or elements from a finite field.

[1] Unless one associates a length with the list and it is updated as necessary.

Since it is also possible to construct cases where the exponents are larger than can be stored in a single word, any representation must allow for this, unlikely as it may seem at first glance. This may happen as a consequence of "packing" multivariate polynomials into univariate ones. One technique we programmed is to assume that exponents fit in single words, and in the case of larger exponents, construct a superstructure which maps the problem into collections of problems with smaller exponents, with a "scaling factor".

2 Solutions

There is a long history of trying different methods for solving this problem "best". See for example, the paper by Robert Moenck [7]. The ground has shifted somewhat since then, partly because people are testing larger problems in which they believe that the time for multiplication of sparse polynomials of size N and M is not so much dominated by the NM coefficient multiplications (typically arbitrary precision integers), but by the time to coalesce and sort the $K = NM$ coefficients into a sorted collection. This time is asymptotically larger: $K \log K$, and can potentially dominate the calculation for large polynomials with simple coefficients.

In these larger problems and using newer computer designs, cache misses related to patterns of memory access may affect execution time more than operation counts. Furthermore, even in a conventional memory model, finding space for the coefficients may routinely cost more than computing them. For example, the time for cache-resident arithmetic instructions to compute multiple-precision multiplication and addition needed for a coefficient might be 50 or 100 cycles, a substantially shorter time than the perioid needed to find an appropriate new storage location for the result.

Furthermore, some attention may be paid (at least lip service) to parallelism, now that multi-core machines are more common.

3 How likely is it that a polynomial product will be sparse?

First of all, let us say what we think of as sparse. Start with two univariate polynomials f and g with N and M non-zero terms respectively. We know that their product will have a maximum degree that is the sum of the degrees of f and g, d_f and d_g respectively, but with some of the intermediate terms zero. A non-sparse answer will have size (that is, number of non-zero terms) not much larger than $N + M$, and perhaps even less if cancellations occur. A sparse answer will have size close to NM. It can't have more than $d_f \times d_g + 1$ terms, so that means, for a sparse output, the degrees f_d and g_d must be (extravagantly) higher than the number of terms N and M.

For a concrete example, take two polynomials f and g each with $N = 1000$ terms and compute fg. If the result fg is to have nearly the maximum possible number of distinct terms $(1,000,000)$ then the $d_f + d_g$ must be at least $1,000,000$. Say each is $500,000$. That means that only about one in 500 coefficients in f will be non-zero, and similarly for g. Mighty sparse, it seems. Indeed, a little testing on randomly generated polynomials suggests that this is an underestimate, and the degree of each had better be at least $5,000,000$ and perhaps even $50,000,000$. This means the input polynomials will contain a non-zero coefficient for every $50,000$ zeros. This is our estimate if you want to get 99.3 percent of the total possible number of terms in the result.

Our conclusions are based on tests where we in fact generate two polynomials and multiply them. We first choose a specified spacing s such that the spacing between coefficients in f and g is a random number between 1 and s, generate two polynomials, multiply them, and look at the result.

```
size(f)=size(g)=1000
         s  deg(f)      deg f*g size f*g

         1       999       1998   1999    dense
        10      5493      11009  10970
       100     49789      99659  94751
       500    247878     493993 373392   250 zeros for every non-zero coef
      1000    499820    1003671 569750   500 zeros for every non-zero coef
      5000   2371811    4812134 875713  2500 zeros for every non-zero coef
```

```
    10000    5001026    9916978 935760  5000 zeros. Close to fully sparse answer.
   100000   49257894   98668403 993174
  1000000  496930601  990887951 999345
 10000000 4931905803 9815952362 999936
```

What this means is that it is probably acceptable to do extra work which is proportional to the number of terms in the input, or, often, proportional to the number of terms in the output, as a preliminary step to doing the multiplications. This will burn you only in extraordinarily sparse situations. Fortunately it is probably easy to detect those cases if you care to look: compare the degree and the number of terms. This is not linear but even better: $O(1)$ in cost.

3.1 Dense arrays for internal multiplication

```
;; convert a list to a dense array of coefficients
(defun list2array(L)           ;; ((low-exponent . coeff) ...)
  ;;(list2array '((4 . 3.14d0)(10 . 1/2 )(12 . 300)) )
  ;;#(0 0 0 0 3.14d0 0 0 0 0 0 1/2 0 300)
  (let ((ans (if (null L) #()
              (make-array
               (+ 1 (caar (last L))) ;how many items in list?
               :initial-element 0))))
    (dolist (i L ans) (setf (aref ans (car i))(cdr i)))))

(defun array2list(A) ;; inverse of above, convert array to list
  (let ((ans nil))
      (do ((i (1- (length A))(1- i))) ;for each element in array, store in list
          ((< i 0) ans)
      (unless (= 0 (aref A i))  (push (cons i (aref A i)) ans)))))

(defun ptimes (L1 L2)(array2list (ptimesA (list2array L1)(list2array L2))))

(defun ptimesA(A1 A2)
  (let ((ans (make-array (+ (length A1)(length A2) -1) :initial-element 0)))
    (dotimes (i (length A1) ans)
      (dotimes (j (length A2))
        (incf (aref ans (+ i j)) (* (aref A1 i)(aref A2 j)))))))

;;test
 (ptimes '(( 0 . -1) (1 . 1)) '((0 . 1) (1 . 1)))
;; answer is (should be ..)
((0 . -1) (2 . 1))
```

This solution is not bad if the input data is relatively small and low degree[2], and the built-in Lisp arithmetic for arbitrary-precision arithmetic, rational numbers, floats, etc. is needed for generality, and the arithmetic implementation is good, and array referencing is compiled nicely. Some optimizations might be worthwhile in `ptimesA` such as declaring the index variables as fixnums. If they are not, it is unlikely that you would be able to allocate the temporary array, or if you could do so, iterate through the task in a reasonable time. A simple improvement is to take `(aref A1 i)` out of the inner loop, if the compiler has not already figured this out.

If the input polynomials are sufficiently large and dense, and the input coefficients are of modest size as well as the answer coefficients: that is, they easily fit (exactly) in a double-precision word, then it can pay off to replace the program `ptimesA` with a more elaborate program based on a numerical "convolution"

[2]We have not quantified "relatively small and low degree" but at least in some tests it appears that we could use the phrase "less than ginormous" ... thousands of terms. That is, arrays are often fast.

implemented via a Fast Fourier Transform (FFT). We have discussed this elsewhere [3], and have programs (in Lisp) to support this. Calling an FFT library written in some (primarily numerical) language like Fortran or C is an alternative to using Lisp as an implementation language; competitions for the cleverest, perhaps parallel, FFT are more commonly held using these languages. Convolution is useful in various contexts, and is, for example, a primitive function in Matlab (`conv`).

With an FFT-based multiplication algorithm one computes two FFTs of size sufficient to represent the answer, with coefficients in a sufficiently-large computation structure. We compute the FFT of each input, then multiply these values point-wise, and then compute the inverse FFT of this product. The details of this are shown in an appendix for multiplying by FFT, taking in total less than 80 non-comment lines. The program `ptimesfft` is a direct replacement for `ptimesA`, but will get exactly correct answers *only if the computation of the FFT in floating double-precision is sufficiently accurate that the answer rounds to the correct integer result.* This may be likely for modest sizes [3]. If it is not sufficiently accurate, it still may be worth doing in order to find the approximate value (and magnitude) of the coefficients, which would then give us valuable information (in particular, *the precision needed to re-do the FFT to get the answer rounded exactly to the nearest integer.*) It may also be the case that the inputs or the results overflow the floating-point format, although this may be excluded by prior analysis if numbers exceeding 308 decimal digits cannot occur. An alternative to higher precision is a version of the FFT in a discrete domain that is large enough to encode the problem; alternatively we can compute a collection of FFTs in different finite fields, each with a single-word modulus. These can be combined via the Chinese Remainder Algorithm, for fully accurate results.

There are other schemes which do not go so far afield from basic arithmetic but still provide an advantage, especially for multipling mostly-dense nearly-equal-size polynomials. The simplest of these is the Karatsuba algorithm[3]. Since these do not require, as does the FFT, the computation of sine or cosine, or roots of unity, these can be done directly in exact arithmetic, rather than floating-point or using finite-field remaindering.

Another observation is that since A1 and A2 are referenced sequentially, we could leave them as lists, which brings us to this solution.

3.2 Dense arrays for internal multiplication, data as lists

```
;; array2list same as above.

(defun ptimes (L1 L2)(array2list (ptimesA2 L1 L2)))

(defun ptimesA2(L1 L2)
  (if (or (null L1)(null L2)) nil
  (let ((ans (make-array (+ 1(caar (last L1))(caar (last L2))) :initial-element 0))
        (e 0)(c 0))
    (dolist (i L1 ans)
       (setf e (car i) c (cdr i)) ;the compiler should do this for us
      (dolist (j L2)
        (incf (aref ans (+ e (car j)))
              (* c (cdr j))))))))
```

Note that there are only three programs needed here, the one-line `ptimes`, the 7-line `ptimesA2`, and the 4-line `array2list`. These line counts are somewhat arbitrary since line-breaks and indentation are not legislated, and the body of `ptimesA2` could just be stuffed into `ptimes`, since it is called only once. Anyway, it looks like 13 lines.

3.3 Lists only, collect terms after

In this version we just multiply everything and put the results into a big list. This has some problems because we will, in typical cases, have many terms with the same exponent that must be combined. That is,

[3]more complex versions are available under the general concept of Toom algorithms

for inputs of size n, we will produce n^2 terms, which may collapse to size n or even less if some coefficients are zero.

Nevertheless, the program is fairly small, and looks like what we had above.

```
(defun ptimes (L1 L2)(compress-terms (sort (ptimesL L1 L2) #'< :key #'car)))

(defun ptimesL(L1 L2)
  (if (or (null L1)(null L2)) nil
    (let ((ans nil))
      (dolist (i L1 ans)
        (dolist (j L2)
          (push (cons(+ (car i)(car j)) (* (cdr i)(cdr j))) ans))))))

;; here's the hard part.

(defun compress-terms(L)
  (cond ((null L) L)
        ((eql (cdar L) 0) (compress-terms (cdr L)))
        ((null(cdr L)) L)
        ((eql (caar L)(caadr L)) ;same exponent for 2 terms
         (setf (cdadr L)(+ (cdar L)(cdadr L))); add coefs in place
         (compress-terms (cdr L)))
        (t (cons (car L)(compress-terms (cdr L))))))


;; optional data abstraction to explain compress-terms

(defun exponent-of(H) (car H))
(defun coefficient-of(H) (cdr H))

;; thus (coefficient-of(first L))   turns out to be (cdr(car L)) or (cdar L)
;; and (exponent-of (second L)) turns out to be (car(cadr L)) or (caadr L)
;; and (coeffient-of (second L)) turns out to be (cdr(cadr L)) or (cdadr L)

(setf w '((0 . 1)(1 . 1) (2 . 1) (3 . 1)(4 . 1)(5 . 1)))
;;(ptimes w w);; compress-terms reduces 36 terms to 11 in the answer, which is
;;((0 . 1) (1 . 2) (2 . 3) (3 . 4) (4 . 5) (5 . 6) (6 . 5) (7 . 4)
;; (8 . 3) (9 . 2) (10 . 1))
```

The program above is 15 lines, and in some examples can be very fast, basically when `sort` and `compress-terms` do not have to do much. It avoids the problem with the dense array, an approach that allocates space that is proportional to the *degree* of the answer, rather than the number of *non-zero terms*. This program is the first of the programs that is able to, for example, multiply $x^{100000} + 1$ by itself in negligible time. A somewhat longer but more efficient version of this program interleaves the compression with the generation of terms; we can also produce terms in order so that sorting is not necessary, though not without its own costs.

3.4 Hash tables instead of arrays

Here we assume that each polynomial is a list but we should accumulate the result in a hash-table.

```
(defun ptimesLH(r s)
  (let ((ans  (make-hash-table)))
    (dolist (i r ans)
```

```
      (let ((co (cdr i)) (ex (car i)))
        (dolist (j s)
          (incf (gethash (+ ex (car j)) ans 0) (* co (cdr j))))r))  ans))
```

An alternative model supported below is that each polynomial is *already converted to a hash-table* prior to multiplication, and the result *can remain in the form of a hash-table.* In the slot indexed by exponent `ex` there is the value `co`, the coefficient.

```
(defun ptimesH(r s)
  (let ((ans (make-hash-table)))
    (maphash #'(lambda(ex co)           ; exponent, coefficient
                 (maphash #'(lambda (ex2 co2)
                              (incf (gethash (+ ex ex2) ans 0) (* co co2))) r))
             s) ans))
```

That program is 6 lines. We could leave all of our polynomials in hash-table form for internal consumption, but it is perhaps unfair given our initial task statement. The Common Lisp default output format for printing a hash-table leaves us somewhat in the dark as to its constituent elements when it responds with an answer like

```
#<eql hash-table with 11 entries @ #x20ccdf4a>.
```

We could either change the output format in the printing program, or explicitly add programs to convert from list to hash-table:

```
(defun list2hash(k) ;;k is a list of pairs, Return a hash table
  (let ((ans (make-hash-table)))
    (dolist (i k ans)(setf (gethash (car i) ans) (cdr i)))))

(defun hash2list(r)  ;; r is a hash table
  (let ((ans nil))
    (maphash #'(lambda(key val)(unless (= 0 val)(push (cons key val) ans))) r)
    (sort ans #'< :key #'car)))

(defun ptimes(L1 L2)(hash2list(ptimesH(list2hash L1)(list2hash L2))))
```

This version adds 8 lines, for a total of 14. The nice part about this program is that it scales up to enormous numbers of terms and arbitrary degrees, using memory proportional to the number of terms, not the degree. If the number of terms is about equal to the degree, the overhead of hash-tables makes this version costlier than using arrays.

3.5 Hashing with GMP arithmetic

Let's elaborate on the program as stated in the previous section, and replace the coefficient arithmetic with GMP arithmetic. GMP, GNU Multiple-precision Package is a free library that can be loaded in to Lisp[4] and provides various facilities for arbitrary precision arithmetic, duplicating but also extending the repertoire of arithmetic, while at the same time providing substantial additional efficiency. GMP uses architecture-dependent routines, some written in assembly language, to speed up processing, and (unless the Lisp system already uses GMP!) provides a boost in speed when the numbers exceed a few word-lengths.

The line `(incf (gethash (+ ex ex2) ans 0) (* co co2))`
shown in the program above needs to be changed to something more elaborate.

```
(setf (gethash (+ ex ex2) ans)
  (mp+ (gethash (+ ex ex2) ans (mpz_creat_zero)) (mp* co c2)))
```

[4]usually; details follow.

We also need to change this program slightly:

```
(defun list2hash(k)
  (let ((ans (make-hash-table :test '=)))
    (dolist (i k ans)(setf (gethash (car i) ans) (lisp2gmp (cdr i))))))

;; and a corresponding use of {\tt gmp2lisp} in {\tt hash2list}.
```

This is not entirely as good as can be done, at least with our handle on the GMP routines, which includes a routine for in-place replacement of a numeric value, that is, a "merged" operation that executes $x := x + a * b$. In particular, the GMP operation `(mpz_addmul x a b)` will update a value x (which might be an entry in a hash-table). This saves on instructions executed as well as memory allocation: temporary memory can immediately be re-used, at least when it is truly temporary and not part of an answer coefficient. The program below has an additional change in the code so the hash code is computed only once, though at the cost of storing more, namely `(exp . coef)` in the hash-table. This trick can be used with the earlier hash multiplier as well, if it appears that computing the hash code is a major component of the cost[5] The basic program has gotten longer though. We assume that the exponents are relatively small: we could declare them to fit in a single word (whose size would depend on the architecture; somewhat less than 32 or 64 bits) to generic faster code; it is certainly unlikely to be a good idea to use GMP numbers for the exponents.

```
(defun ptimesg(r s);; multiply 2 hash tables with GMP
  (let ((ans  (make-hash-table :test 'eq )))
    (maphash
     #'(lambda(ex val);; index is exponent, val is (coef. exponent)
         (maphash
          #'(lambda (ex2 val2)
              (let* ((expon (+ ex ex2)) (spot (gethash expon ans)))
                (cond (spot (mpz_addmul (car spot) (car val)(car val2)))
                      (t (setf spot (cons (create_mpz_zero) expon))
                         (mpz_mul (car spot)(car val)(car val2))
                         (setf (gethash expon ans) spot) ))))r)) s) ans))
```

Here the internal `ptimes` program is now 10 lines long, up from 6. Is it faster? Certainly it is, if the underlying Lisp has a typical arbitrary precision package (that is, not nearly as fast as GMP) and some of the coefficients are large enough (say over 30 decimal digits). Because "foreign function" interfaces (FFI) are not currently part of the ANSI standard for Common Lisp, and not every Lisp conforms to a proposed standard "universal" FFI or UFFI, the details for attaching the GMP library to Lisp still depend on the Lisp implementation. The version we use is in the generic library available online, `http://www.cs.berkeley.edu/~fateman/generic`. Its specific details are aimed at Allegro Common Lisp.

We have left unspecified the implementation of a hash-table, using the default provided by our Lisp system. As is the case for almost any general procedure, it is possible that this could be implemented in a more specialized fashion, and by eliminating unused generality, made faster. For example, we know that the keys are the term exponents: non-negative integers of a specified maximum size. The Lisp hash-table system allows for the keys to be any Lisp expression. As a kind of specialization of a hash table for integers, we wrote out a "radix-tree" structure implementation where each level in the tree represents a fixed number of bits in the index (the exponent), and the exponents are not stored explicitly, but can be inferred by position in the tree. In one experiment involving a polynomial with 5,151 terms, squared to a polynomial of 20,301 terms, the radix tree representation used slightly less memory than the hash-table (each node had up to 256 entries), and the radix tree was about 4X faster. For smaller less-dense problems the times are closer. The radix tree is, in some sense, an intermediate position between an array and a tree, since a large enough node allocation would result in all the terms in a polynomial fitting in just one dense node. In that case the terms would be indexed as though they were in an array! In the case of a sufficiently-sparse and

[5] In our tests this is trivial if the exponent which is being hashed is an integer. If the exponent is instead a vector of exponents as might be used for a multivariate polynomial, the encoding can represent a substantial improvement.

widely-scattered collection of exponents,(keys), the storage would effectively be proportional to the number of keys (times the node-size). Large portions of the (empty parts) of the polynomial might effectively use no storage even though access time would be constant. Another variant of the radix tree (which we suspect is novel) is to use different size nodes. We programmed a system which has an arbitrary specification for the size of nodes at different levels. Thus a top-level node of size 1024 will fit all polynomials of degree less than 1024 into one node. Making the next level size 32, and then size 8 for all subsequent levels means that larger polynomials may be stored more sparsely, with polynomials of degree up to 32,768 taking only 2 levels, and degree 262,144 only 3 levels. This data structure is essentially isomorphic to a hash table of size 1024 with elements that can overflow into buckets of size 32, which in turn can overflow into buckets of size 8 (etc.). The hash function is simply extracting n bits from the right end of the key via a logical AND; the key used for the rest of the search is that portion remaining after the n bits are shifted off the right.

In the case of very sparse polynomials, we have modified the radix tree notion and have special (terminal only) "shortkey" nodes with 2 entries: a displacement in the tree to the key, and the value. These are effectively used only if there are no elements that would be in the same regular node (nearby). That is, taking an empty tree and inserting an element with a huge key does NOT initially populate all the subtrees needed to reach that key; rather a shortkey node is generated to store (essentially) the path to that node, and its value. (Recall that in the usual case, the key is not explicitly stored: it is implicit in the path to the node.

Continuing on in this vein of trying out tree variations, we found that a re-implementation using binary search trees was terrible (200X slower) because the order of insertions turned out to be too nearly one of the well-known exactly wrong orders, resulting in the building of a very unbalanced tree. Countering this trend, we found that using a balanced tree insertion scheme (AVL tree) was approximately the same speed as the radix tree on the same example. We tried an additional alternative using yet another data structure known as a skiplist, finding that this too had no outstanding performance, and finally a heap structure, which we discuss in the next section. Another direction to pursue is an implementation of a data structure again comparable in semantics to hash-tables or search trees for exact match, but especially suitable for parallel processing. There are several issues pertinent especially to Lisp and hash-tables that may not appear in some other language contexts. Since Lisps may be (and often are) hosts for multiprocessing, the atomicity of a hash-table transaction must be assured if several processors are to be used for this task. Furthermore, insertions into the hash-table may cause a storage reallocation to grow the hash-table. If so, external pointers from data or programs directly into the hash-table itself cannot be used. Additionally, a garbage collection triggered (perhaps by another Lisp process) can move the hash-table to another location in heap-allocated storage. Any Lisp that hopes to communicate via structures with C and FORTRAN must make it possible to "lock" in memory locations, so one can, with some (non-ANSI CL) coding effort lock down hash-tables, too. In a case where the hashing overhead dominates the arithmetic, and where atomicity of transactions in a parallel-processing environment needs to be handled by the user, it may pay to write a fixed-in-memory hash-table package, perhaps distributing the hash-table itself to various processors. We have not tried to duplicate the hash-table facilities with these additional constraints.

3.6 Heaps

One alternative for storage is the "heap" made famous by "heapsort".

In this approach we assemble the answer by extracting all sub-products contributing to a particular exponent by bubbling them up through an ordered heap and adding them together. We programmed this approach, advocated by Monagan and Pearce [8]. It was close in speed to that of hashing on answers that were relatively dense, but not as fast as a method especially good for dense results (simple arrays). To its advantage: it was faster than hashing and was also more conserving of storage on very very sparse multiplications.

Our initial implementation was disappointing: Correspondence with Michael Monagan and Roman Pearce on using a heap structure for multiplication resulted in our developing a more polished implementation of this basic idea. After finding that various suggested programming tricks did not make much of a difference, we found that the way to make this approach win was to change the nature of the input data. That is, choosing the right benchmark is the key: the inputs must be extremely sparse, with the gap between exponents larger

than the number of terms (e.g. a 1,000-term polynomial with gaps of 5,000 between successive exponents.) This reflects the fact that the heap method will not be advantageous for "mostly dense" results, where hashing or a Karatsuba method or FFT, or just a carefully-coded tight loop of "naive" multiplication may be faster.[6].

Here is the argument: In a very sparse multiplication of N by M terms there can be $N \times M$ terms in the answer. If each of these is established in a separate entry in a hash-table, that hash-table will have grown to substantial size, and to very little benefit in return for the cost of hashing: no merging of coefficients with equal keys will happen, and furthermore, the results will not be sorted (if that matters). Compare this to the heap implementation where the size of the heap, by contrast, will be proportional to the smaller of N and M. As the coefficients in sub-products are generated, the components of the appropriate exponent will be pushed out of the heap, to be examined, added together, and then stored away, never to be processed again. Monagan and Pearce assert an advantage that this enables all bignum arithmetic to be done essentially with one accumulator: they accumulate the indices of the coefficients in chains, on the heap, and then only later perform the arithmetic. In our implementation storing such chains does not seem to be particularly advantageous; we just add the terms as they appear on the top of the heap. And indeed, for large super-sparse polynomials, this heap idea is faster.

There are (at least) two ways of maintaining the heap as needed. The size of the heap is equal to the number of terms of the smaller polynomial for most of the computation. As terms are exhausted, the heap shrinks. Inserting into the heap can be done from the top (as the top-most term is removed, place a successor in its place and percolate it downward), or from the bottom (after removing the top-most term and reforming the heap, insert the new term at the bottom. Somewhat to our surprise, insertion from the bottom was usually faster.

3.7 Encoding polynomials as long integers

There are other methods that we (and presumably others) have implemented and tested. It is possible to encode any polynomial $p(x)$ as a long integer by (essentially) evaluating it for some value of x that is sufficiently large. Since we can round up the value x to be a power of two, this evaluation can be done by shifting and adding. This idea is sometimes attributed to Kronecker. Decoding the polynomial can also be done simply by shifting and masking.

Here is a program to compute the product of two polynomials p and q:

```
(defun ptimesI(p q)
(let((logv (padpower2 p q))); compute the padding needed
  (int2list (* (list2int q logv)(list2int p logv)) logv)))
```

The details of the three programs to find the log of the bound, as well as the two conversions between list and integer, are given below. The proof for the size of the padding has been provided in a previous paper [3].

It is worth noting that the production of these, in general, rather long integers, as well as the solitary multiplication, can be done in ordinary Lisp, but unless the Lisp has very fast large-number arithmetic, this is not a great idea. By using Lisp to call the GMP library, we relieve the implementors of the Lisp of the burden of hacking together a truly superior bignum facility.[7] We published such linkage programs in Sept. 2003. See Appendix 2.

```
(defun list2int(p pad)
  (let ((ans 0))
    (dolist (j p ans)(incf ans (ash (cdr j)(* pad (car j)))))))
```

[6]Somewhat orthogonally to this argument, if the coefficient arithmetic is hugely expensive, and we can implement the FFT (or trivially one of these asymptotically improved algorithms) around this complication, we can reduce the number of such operations, and save time this way.

[7]Why would not every Lisp have a superfast bignum facility? It appears that there are limited "market" forces at work for this objective. The need for frequent revision in the face of new hardware variants has been more easily met by the GMP project.

```
(defun int2list(i pad)
  (let* ((ans nil)(q 0)(r 0)
         (mv (- pad))
         (vby2 (ash 1 (1- pad))) ;2^(v-1)
         (v (ash vby2 1))
         (mask (1- v))
         (signum (> i 0)))
    (setf i (abs i))
    (setf exp 0)
    (while (> i 0)
      (setf q (ash i mv))
      (setf r (logand i mask))
      (cond ((> r vby2)
             (push (cons exp (if signum (- r v)(- v r))) ans)
             (setf i (1+ q)))
            ((= r 0) (setf i q))
            (t(push (cons exp (if signum r (- r))) ans)
             (setf i q)))
      (incf exp))
    ans))

(defun maxcoef(p1)
  ;; accumulate pos and negs separately to avoid computing/storing abs.
  ;; compare at end and make positive.
  (let ((pans 0)(mans 0) temp)
    (dolist (i p1 (max pans (- mans)) )
      (setf temp (cdr i))
      (if (< temp 0)(setf mans (min mans temp))
        (setf pans (max pans temp))))))

(defun padpower2 (p1 p2)
 (* 2 (ceiling (log (*(maxcoef p1)(maxcoef p2)(1+
    (min (caar (last p1))(caar (last p2))))) 2))))
```

4 Sparse arrays

Another kind of data structure that is easily built in ANSI standard Common Lisp, and is, at least in some implementations, handled very efficiently, is an encoding optimized toward sparse polynomials, consisting of two arrays. In the exponent array of length N we have a sequence of exponents, and for each exponent a coefficient in the corresponding position in the coefficient array. Thus instead of ((10 . 20) (24 . 90)) we have arrays [10 24], [20 90], Compiled Lisp code can sequence through these arrays as fast as is done in other languages; whether this is faster than sequencing through a linked list may depend on hardware memory issues. The array version will ordinarily save space by not having to store links. Another associated positive consequence of the use of arrays is the greater likelihood that we will have localized data in a memory cache when needed.

There is a particular programming convenience in using arrays that we can use. An array can be more easily "split" into sections as compared to a linked list. For example, two "virtual" halves can be designated with each half being treated separately. The two pieces could perhaps be handed to different processors, or just used in a kind of divide-and-conquer algorithm. That is, we can multiply $A \times B$ by splitting A into two sections A_1 and A_2, and computing $A_1 \times B + A_2 \times B$. The addition is done by some kind of merge. This splitting can be done on the subproblems too, splitting B, A_1, A_2, each into smaller pieces. The idea

is to repeat the splitting until the lowest-level multiplications are small enough to fit (entirely or mostly) in some targeted memory cache (L1, L2, L3). In the case of such huge problems, merging of results should, to the extent possible, avoid examining in their entirety the potentially large coefficients; only the exponents are needed for sorting. One way to do this is to have the sparse array of coefficients consist of pointers to the coefficients, each of which may be an arbitrary-precision number, stored elsewhere than the sparse array[5]. We attempted to repeat our earlier studies using cache-measurement tools but were frustrated by the mismatch of tools (PAPI and Windows, Intel's VTUNE and Lisp, on our current hardware, a Pentium 4). We expect that at this point in the technology and size of caches, any results would be tedious to collect and hard to relate to other setups. Testing as in ATLAS [1] might be appropriate, should one try to achieve the utmost efficiency of an algorithm on a particular platform.

If memory is monolithic, the splitting into sections can be done by adjusting arrays endpoints (a book-keeping arrangement), easily done by "displaced arrays," a feature in ANSI Common Lisp.

Given an array `z` of length 2 `h`, we can, in Common Lisp, with negligible computation and no array storage allocation, rename its two halves:

```
(setq lower (make-array h :displaced-to z))
(setq upper (make-array h :displaced-to z :displaced-index-offset h))
```

To review: the scheme is to take advantage of this splitting approach is to multiply A by B by splitting one or the other into halves until the pieces are each within some target size, multiplying, and then adding all the pieces together in a tree merge where short results are merged to create longer and longer sequences, culminating in the final merge to produce an answer.

One program for this framework looks like this. (Details are available in program listing)

```
(defun mul-sac(x y)
  ;; mul-Sparse-Array-Cache(x y)
  ;; x and y are sparse polys in arrays
  (let* ((xexps (car x))   ; an array of exponents for x
         (yexps (car y)))  ; an array of exponents for y
    (cond  ((< (length xexps) *cachelim*)
            (if (< (length yexps) *cachelim*)
                ;; both x and y are small enough
                (mul-sparse-inside-cache x y);; something good for cache
              ;; x is small enough, y is not. split it.
              (multiple-value-bind (low hi)
                  (splitpol y)
                (add-sparse-array (mul low x)(mul-sac hi x)))))
           (t (multiple-value-bind (low hi)
                  (splitpol x)
                (add-sparse-array (mul-sac low y)(mul-sac hi y)))))))
```

Even if we are not trying to multiprocess, another reason exists for splitting these polynomials. This reason is we would prefer, for execution speed to have a limit on the largest exponent in a polynomial, at least in intermediate calculations. There is a speed advantage in knowing that the array of exponents is an array of numbers that fit in a single word (that is, an array of fixnums), rather than containing any multiple-word "arbitrary precision" number exponents. On the other hand, we do not want to limit the exponent range in the abstract model. Therefore, if the input polynomials have exponents that exceed the single-word size, then the multiplication can be re-cast by segmenting the inputs as $A_0 + A_1 x_1^k + A_2 x^{k_2} + \cdots + A_n x^{k_n}$ where the k_n are chosen so that two successive exponents are no further away than half of the most-positive fixnum. The exponents in the polynomials $\{A_i\}$ as well as the exponents in the product are necessarily fixnums. The total polynomial multiplication can proceed by taking two segmented polynomials using the data assumption of single-word exponents, and piecewise multiplying them (fast), and then adding up the results, suitably shifted by the $\{k_i\}$ (using arbitrary exponent sizes).

The $\{k_i\}$ are chosen so that for each A_i, the smallest exponent is 0.

5 What about multivariate polynomials?

There are at least two different styles of extension to accomodate multivariate polynomials. One extends the notion of coefficient multiplication to coefficients which are themselves polynomials in other variables. This requires a specification of order among the variables, with the "most main" variable providing a kind of scaffold upon which other polynomials hang. A mechanism must be provided to order these variables so that the multiplications of (say) $x+1$ by $y+1$ would result in $(x+1)y^1+y^0$ with y the main variable, or as an alternative, $(y+1)x^1+x^0$ with x the main variable. This *recursive* representation is used in several computer algebra systems, including Macsyma's canonical rational expression system. It is more efficient for some operations, in particular division with respect to the main variable, since accessing the highest-degree term is a constant-time operation in this representation.

Another popular approach is to store a polynomial as a collection of monomials where a monomial is a coefficient times the product of powers of variables. Ordinarily we expect most powers will be rather small, coming from terms that look like x^2y^3, so it may make sense to pack the numbers 2 and 3 into fields in a single integer. This "exponent vector" integer can represent any of the anticipated values of exponents packed together into a single sufficiently-long integer. To do this effectively we need to bound each variable's power so as to space them out in the bits of that integer. In effect we encode all the variables as suitably high powers of a single variable. For example in the expression $f = x+y+z+1$ let $x=t$, $y=t^{100}$, and $z=t^{10000}$. The expression $g = t^{10000}+t^{100}+t+1$ behaves in many ways the same as $x+y+z+1$. In this case, the 20th power of f and g has 1771 terms, and it is easy to map from one form to the other. Furthermore, g has a term $465585120t^{31005}$ which, after a moment's thought, must correspond to a term in f of $465585120x^5y^{10}z^3$. This kind of encoding was used prominently in the ALPAK and ALTRAN systems [2], where the user is required to provide, in advance, the limits of exponent range for the exponent layout. The ALTRAN system provides a mechanism that cleverly includes guard bits between the exponent fields to protect against overflow from one field to the next.

We show a program which scans each of the polynomials, finds the smallest power of two (rather than 10, as used in the illustration above) that suffices to separate the maximum exponents. Using a power of two means that binary shifting and masking can encode and extract the exponents rapidly. The program converts the multivariate polynomials each to a univariate polynomial then to a hash-table, does the multiplication, and returns the hash-table back to list form. We do not need to protect against overflow of exponent fields since we have computed the size that is sufficient. If we were to keep data in hash-tables over many operations, we could emulate ALTRAN, require user declarations, etc. In our simple case for packing exponents for a single multiplication, this takes about 40 lines of non-comment code in total. See Appendix 3.

A further extension of this approach would be to reduce the 1771 terms in our example above even further by packing all the terms into a single huge integer as suggested earlier.

Alternatively, if the hackery above seems too obscure, requiring too much hassle to pre-compute bounds, a program that just jumps in to the multiplication process can be done by using the exponents in a vector or a list, associated with each monomial. This requires only a modest change to our hash program. In practice this may be slow if run literally as shown, since the exponent hashing is now done on a list of integers rather than a single integer of the packed exponents. Furthermore, we expect that the list takes much more storage. Nevertheless, the program is 10 lines.

```
;; allow any number of powers associated with each coefficient.
;; the list of variables, e.g. x,y,z ... is implicit in the order of powers


(defun list2hash(k)  ;; k looks like ((30 . (5 4 3))(21 . (9 7 0)))  30*x^5*y^4*z^3+ ..
  (let ((ans (make-hash-table :test 'equal)))
    (dolist (i k ans)(setf (gethash (cdr i) ans) (car i)))))

(defun ptimes(r s)
  (let ((ans  (make-hash-table :test 'equal)))
    (maphash #'(lambda(ex co)            ; exponent, coefficient-list
                 (maphash #'(lambda (ex2 co2)
```

```
                        (incf (gethash (mapcar #'+ ex ex2) ans 0)
                              (* co co2))) r)) s)
    ans))
```

6 Can we make these programs faster?

Yes. It would be pointless to compare these programs as-is to make an informed choice as to which is "the best" without knowing the context of the problems to be solved, profiling the behavior of the different programs, compiling them with suitable levels of "optimization" or (lack of) error-checking, inserting type declarations when possible. (In Lisp, all declarations are optional, but they often are useful for some compilers, enabling them to produce faster or more compact code.) On some computers and with some systems it is possible to find parallel implementations of hash-tables and FFT which could be used.

As a general note, for most of these programs it seems plausible to write a collection of specific targeted programs for different scenarios and then a general "poly-algorithm" program that first tries to categorize the situation and then calls one of the specialty routines. Unless you already know the situation, it is worthwhile to do some diagnosis. That is, except for the very smallest inputs, you can spend time "linear in the size of the input" or even "linear in the size of the output" for pre- or post- processing that would help choose the right special algorithm, or even coerce the input and output into the forms needed by those routines. In spite of this extra time, the net affect can be to have an algorithm that is only slightly slower than the time for the best routine. We can build a kind of expert system to choose the right algorithm and mostly ignore the small costs for converting between integers and floats, packing and unpacking arrays, converting between lists and arrays. These are dominated by the subsequent multiplication. The diagnostic information that is readily available includes some heuristic estimates for density (comparing the degree and the number of terms), and computing some norms to see if all the exponents and/or coefficients in the input and output can be bounded by some convenient limit and packed in arrays. (this is described below in some detail).

In writing the detailed programs for these methods, as well as some other methods we have ignored in the text above because they never exhibited superior performance in our tests, we found it useful to look at the following ideas.

- Declaring all known types, and constraining the problem domain so that the types are known; writing separate versions with minor alterations (e.g. "fixnum-only coefficient" version). Lisp compilers are likely to produce better code if we promise that all *exponents* are small integers in the "fixnum" range (depending on the architecture (64- or 32- bit), this would be something less than 64 bits or 32 bits). Separately, it is advantageous if *coefficients* are all fixnums, or all double-precision floats, or all complex, rational, bytes, etc. Our multivariate encoding allows several exponents to be packed into a single word, saving space and time.

- Even if the coefficients are not known in advance to be all small, or densely arranged, it may be that in a particular problem instance they are, and it is to our advantage to scan through them so we can convert a calculation over the "arbitrary precision integers" to a calculation in fixnums, or (in our experiments, more advantageously) in double-floats. At least for some 32-bit architectures, using double-floats can improve speed while allowing for more bits for representing numbers precisely. If the integers needed are exactly representable in the floating-point fraction field, and operations preserve accuracy, then we can win with this transformation (returning the result to integers by rounding if need be).

  There are easily computed bounds on the sizes of coefficients. Here's a good one: Let maxnorm(p) be the largest coefficient in absolute value in the polynomial p.

  Let sumnorm(p) be the sum of absolute values of coefficients in p. Then a bound on the coefs in the answer is min (sumnorm(p)*maxnorm(q), sumnorm(q)*maxnorm(p)). This bound is achieved, for example, for $x^n + x^{n-1} + \cdots + 1$ squared.

As we have written earlier, if the floating-point numbers are not sufficiently precise, the calculation can be mapped to results in (several) finite fields and matched up using the Chinese Remainder Theorem (Garner's algorithm) to get higher precision results. [8]

- Parallel implementation of the FFT, perhaps using a program from FFTW, http://www.fftw.org/ . (Not used by us)
- Parallel implementation of maphash, get/put hash, encoding/decoding multivariate polynomials. (Not used by us)
- Packing multiple (short) coefficients in words. (Not used by us)
- For polynomials in arrays, or perhaps even in lists, consider blocking to improve cache performance.Hida [5] was able to demonstrate up to a 47% improvement on large polynomials with simple (floating-point) coefficients, using cache-aware memory blocking . (A thorough examination of these techniques is considerably aided by specialized software accessing hardware-specific registers, something we were not able to do with our latest testing done for this paper.)
- hash-tables are just containers. Other container designs with similar operations can be used. These include binary search trees, B-trees, AVL trees, heaps [8, 6], skiplists, etc. These can, from one perspective be seen as efforts to solve the "sorting X+Y" [4] problem (practically) fast, by (asymptotically) faster algorithms, or by using less storage, or storage in some disciplined fashion that may improve memory cache performance. Efforts to implement a program with an algorithmic complexity better than O(n) where n=size(X)*size(Y), seem to run into problems; $O(n \log(n))$ is achievable in simple ways[6]. In view of today's computer designs, improvements seem to depend on cache-performance techniques. See `http://maven.smith.edu/~orourke/TOPP/P41.html`.
- For sparse multivariate polynomials especially, or polynomials which have been reduced to a sparse univariable polynomial by packing exponents, we can consider another approach: We can compute the density of a result heuristically (or indeed we can compute an exact layout) by performing a kind of virtual multiplication. In effect, we replace all the coefficients by 1, and redefine multiply and add operations by binary logic. This produces a skeleton of the non-zero coefficients in the product (though some of these coefficients may turn out to be zero from cancellation when the real product is computed). This skeleton can be used as the basis for various ideas that might improve performance. If we use a hash-table, now the maximum number of entries in the hash-table is known, as well as all the hash keys. We could, if we wished, store at each hash location a recipe for exactly which products to sum up to obtain the necessary coefficient corresponding to that exponent. This would be expensive in memory, and can be done in stages by using a heap; we did not, however, find this to be as good as a hash-table.
- Another indexing structure could be manufactured from this information, including perhaps some distributed hash-table for parallel processing. (All pre-packaged DHT facilities we found assume the entries in the hash-table are more substantial, not just numbers, so we don't expect they would be of direct use). Another thought, specifically for multivariate polynomials, is that we could use this information to go back to the original polynomials, and use modular evaluation. Rather than conventional interpolation and the Chinese Remainder Algorithm, we could use sparse interpolation[9] to compute the coefficients that are not zero. This technique works well for greatest common divisor calculations, but it would be quite a stretch to expect it to work fast for multiplications. (Not used)
- A stream version of multiplication in which P and Q of size $n \leq m$ respectively, can be arranged, implementing the heap idea but thinking about the computation functionally, rather than with arrays. In effect, the program sets up n computations, each of which can be represented by afunctional stream: as a practical matter, a short record of information storing a state of a computation which can be quickly picked up and run to get the next output from that stream. If $P = p_0x^{e_0} + p_1x^{e_1} + ...$ $Q = q_0x^{f_0} + q_1x^{f_1} + ...$ Stream 0 delivers the sparse exponent and coefficient pairs: $[e_0 + f_0, p_0q_0]$,

[8] see Appendix 4

$[e_0 + f_1, p_0 q_1]$, etc. while stream $n-1$ delivers $[e_{n-1} + f_0, p_{n-1} q_0]$, etc. These streams can be allocated on up to n different processors. The main multiplication is done by arranging pointers to these streams in a heap so that the top of the heap has (one of) the lowest-valued streams judging by the $e_i + f_j$ exponent of any of the streams. As items are removed from the top of the heap, all equal exponents have their coefficients added together and then sent out to the answer. Each time we use one of the pairs from a stream we advance the stream and (if there is another monomial on it) insert it back into the heap. Several tricks can be used: the insertion into the heap can sometimes be done faster "from the top" rather than the usual "from the bottom." Also it may pay not to compute $p_i \times q_j$ ahead of time, but to defer the multiplication until the accumulation is being done. This is because the cost of computing the update $c := c + p_i \times q_j$ may be lower if it is done in a single "multiply-add". Monagan and Pearce [8] advocate storing a list of the $[i, j]$ pairs in this activity and not doing *any* coefficient arithmetic until they are all assembled; this allows them to execute the high-precision arithmetic for the accumulation all at one time (per coefficient in the answer) the result stored as a part of the answer.

After considerable twiddling, the best stream multiplication / heap program is about as good as the hash-table version for dense polynomial answers (though neither is nearly as good as a dense method) but heap is superior to hash for very sparse answers. A common-sense reason for this is that in such a situation the hash-table program is mostly wasting time allocating unique locations for each exponent in the table, never to be re-visited except when repeatedly enlarging the hash-table to accomodate the growing number of terms.

7 Benchmarks

Before running benchmarks, it is appropriate to see to what extent each of the programs should be specialized for the particular class of inputs being tested. In particular, it seemed, initially, pretty safe to assume that in programming in Lisp, we could declare exponents to be "fixnums" in ordinary polynomial cases (when do we have exponents exceeding 2^{30})? Where would such polynomials occur?

On the other hand, if we are encoding multivariate polynomials (say in 30 variables) packing them all in one exponent, then the resulting single exponent may be rather large.

We can write the program out twice, and apply the appropriate one based on the size of the exponents in the particular instance.

A program where the coefficients are arbitrary Lisp numbers can be specialized in several ways. Without further specification, a Lisp number can be a limited-precision integer, an exact arbitrary precision integer, a rational number, a single- or double-precision floating point number, a complex number (with rational or float components), or with some additional effort in our case, a quad-precision float, or an arbitrary multiple-precision (MPFR) float, or a GMP arbitrary-precision integer. The last of these duplicates the built-in Lisp facility for integers, but using an external library that could be both more efficient and more highly tuned to the particular hardware. Given all these choices, the operations of "+" and "*" are subroutine calls whose execution time can be dominated by type-dispatch decision-making rather than arithmetic. Declaring that all coefficients are (say) double-float, changes the nature of the generated code, and makes the benchmarking more elaborate: 10 kinds of coefficients, and within some classes of coefficients we should consider the contents: e.g. arbitrary-precision shortish numbers, and longish numbers.

For some of our tests we set up a generator for random polynomials. The input was n, the integer number of distinct monomials, m the (maximum) integer space between successive exponents: $m = 1$ means completely dense. $m = 100$ means that n different times a random number k between 1 and 100 is generated, and if the previous exponent was e, the next exponent is $e + k$. The third parameter c is the coefficient size. For each of n different monomials, the coefficient is set to a random integer between 0 and $c - 1$.

Setting $m = 1$ gives dense polynomials. We can produce a sparse polynomial with $n = 100,\ m = 5000$, for example.

Taking one of these random polynomials and squaring it provides a dramatically different distribution of spaces between exponents. There is a distinct clustering of monomial exponents in the "central portion" of the exponent range of the polynomial.

We also consider squaring polynomials along the lines that we have proposed for testing in earlier papers, but also picked up by others:

$p = (1 + x + y + z)^{20}$ converted to a univariate polynomial with $y = x^{41}$ and $z = x^{41^2}$
$q = x^{20000} + 1$
$r = (x^{10} + 2^{64})^{100}$
$t = 1 + x + \cdots + x^{1000}$

If we can predict the sparsity of the answer, we can choose between algorithms appropriate for dense and sparse results. That is, using a direct naive method is good for anything that is small enough. If the coefficients are small enough, methods compiled for declared types that are single-precision integers "fixnums" are considerably faster, even using an "n^2" method, carefully coded. For larger coefficients and dense enough, especially where the inputs are of about equal degree, problems can be done with a dense method such as Karatsuba or (if the coefficient domain is suitable, or suitably hacked, and the degree is large enough, FFT). However, if only half or a quarter of the coefficients are non-zero, or if the inputs have widely different degrees, the FFT loses its advantage in our tests.

A good very sparse method for large size polynomials, not necessarily of similar size, where the answer is expected to be sparse (meaning the inputs are probably very very sparse), seems to be a carefully implemented heap algorithm.

What measurement and computation on the inputs might work to predict the sparsity of the answer, at least as far as needed to choose the best among a set of existing algorithms?

It must be inexpensive to compute (no worse than linear in the size of the polynomials to be multiplied together), and not be far wrong on large sizes. (For small sizes it almost does not matter since all algorithms are fast enough.)

Here's one. Define DegreeSpan of a polynomial P: the difference in degree between the maximum monomial exponent and the minimum in P. This is one less than the degree of a polynomial if there is a non-zero constant term. Let $\#f$ be the count of monomial terms in f.

Define spacing $S(f)$ as a ratio of $\#f$/DegreeSpan(f).

Say that "Sparse" means $S < 0.2$. Note: $S = 1$ means 100 percent dense. $S = 0.2$ means that of all the possible terms up to that given degree, only 1/5 of them have non-zero coefficients.

Alternatively the existing coefficients are (on average) the sum of 5 terms in the product.

Consider polynomials $f,\ g$ with random spacing between exponents.

For $n > 1$, if the degree-span of either f or g exceeds $n\#f \times \#g$ then $f \times g$ usually has spacing $S < 1/n$. [Proof? We don't have a proof; it is not even always true. Just some test data.]

So in particular if we use 0.2 as a cutoff, the degree-span of f or g should exceed $5 * \#f * \#g$.

Whether 0.2 is a critical cutoff point for anything depends on your algorithms, your computer, your polynomials, your Lisp compiler.

In our experience precise benchmarks become mostly irrelevant as a choice of language, compiler and implementation can change a balance by a factor of up to 20; the choice of a hardware platform with different size cache and perhaps alternative instruction sets can create substantial changes as well. Memory sizes affect the benchmarks as well: some algorithms use less memory; this becomes critical when they are compared to algorithms which use slightly more memory: enough to require paging. Versions of operating systems also affect testing data.

Substantial timing data, mostly reproducible on one machine, have been generated. Because of the caveats above, we include only a very small selection; instead the discussion below attempts to summarize.

A summary of the algorithms, each of which takes two polynomials and returns their product; the polynomials are pairs of arrays containing exponents and coefficients.

dense Allocates an array large enough for all the answer coefficients, fills with zeros. Computes the product, then runs through the array counting non-zero coefficients. Allocates two fixed-length arrays and stores exponents and coefficients in them, returning the pair.

dense fixed coefs Same as dense but assumes coefficients are fixnums. Not really a good assumption, but it makes a more fair comparison with FFT, which makes assumption that coefficients are exactly representable as double-floats.

heap Allocates a heap the size of the smaller input; cycles all subproducts through the heap collecting results into a stack. Rewrites the stack into two fixed-length arrays.

naive Collects terms as generated into a sorted linked list. Rewrites into two fixed-length arrays.

hashing Allocates a hash table which grows as required to be large enough for all the answer coefficients. Computes the product, then collects data (key, value) in the hash table into an array which is then sorted and rewritten into two fixed-length arrays.

fft Like dense, except allocates larger arrays suitable for FFT, and multiplies using a floating-point double-precision FFT. Rounds the results to integers and rewrites the non-zero terms into two fixed-length arrays.

radix tree Allocates a radix tree which grows as required to be large enough for all the answer coefficients. Computes the product, then collects data (key, value) in the tree into an array which is then sorted and rewritten into two fixed-length arrays.

karatsuba, toom Like dense, except multiplies by subdividing arrays in halves recursively (until small enough that a $O(n^2)$ is faster), and using an asymptotically faster method. Then rewrites non-zero terms into two fixed-length arrays.

```
polynomials q and r have 10,000 terms, spaced 50 apart, with coefficients
between 1 and 5. Time the computation of q*r.

 Algorithm                      Time in seconds

  radix-tree                    42.7
  naive fixed coefs             50.2
  heap                          64.2
  other algorithms ran out of memory

polynomials q and r have 5,000 terms, and 1,000 terms respectively,
spaced 50 apart, with coefficients between 1 and 5.


  dense-fixed coefs              0.09
  dense                          0.22
  radix-tree                     1.62
  hashing                        1.68
  fft                            2.08
  heap                           2.53
  naive                          4.03

polynomials q and r have 5,000 terms, spaced 500 apart, with coefficients
between 1 and 5.

  dense-fixed coefs              0.20
  dense                          0.37
  heap                           3.37
  radix-tree                     8.4
  hashing                        8.1
  naive                        129.19
  fft                            out of space

polynomials q and r have 5,000 terms, spaced 1 apart, with coefficients
between 1 and 5.  (Fully dense inputs)

  fft                            0.02
```

```
  dense-fixed coefs              0.03
  dense                          0.19
  toom                           0.22
  karatsuba                      0.50
  radix-tree                     0.67
  hashing                        0.81
  heap                           2.17
  naive                          2.76
  fft                            0.02

polynomials q and r have 5,000 terms, spaced 10,000 apart, with coefficients
between 1 and 5.  (very sparse)
  heap                            4.70
  radix-tree                     22.88 ;; 6.85 without sorting answer
  hashing                        22.88
  naive                         320.81
```

8 Conclusions

A really small program won't be the fastest, but given a selection of a few, one of them will be pretty efficient. If we can find an easily computed characterization of the problem (especially an estimated description of the density of the result) we can probably decide, in a particular hardware/software situation, which of several polynomial multiplication programs will be fastest, incorporating costs of sorting of exponents. Automatically tuning can probably help with choosing the cutoff parameter for spacing versus degree, as well as dividing up large polynomials into sections for improving cache performance[5], an exercise we do not pursue here. We expect that the tradeoff will generally be between a simple array-based algorithm (for nearly all problems) and the use of a heap-based multiplication for extravagantly large and sparse cases.

Other choices, including asymptotically faster methods (like dense FFT, dense Karatsuba) may pay off, even by a factor of 10. While the FFT has a significant leg up, in some circumstances, the implementation we timed requires that approximate double-precision floating-point number coefficients are sufficient for accuracy. Regarding the FFT, note that if other programs are compiled to be similarly type-restricted to fixed-length coefficients, and also given dense problems, the advantage of the FFT on any problems that fit in memory seems to be more like a factor of two. Given a substantially sparse problem, the FFT or Karatsuba style algorithms are quite slow, and may in fact run out of memory when other algorithms are feasible, because they use memory proportional to the degree, not the number of terms in the answer.

A rude, but nevertheless plausible question to ask is: What purpose could possibly be served by multiplying polynomials of size 10,000 or more? We don't address it, but merely remark that we have discussed the question with other researchers who have written up algorithms recently. They don't seem to care.

9 About Lisp

We have unapologetically provided Lisp programs without explaining the details of the Lisp language usage. There are excellent books as well as online guides to Lisp. While we do not claim these Lisp programs are "self explanatory" each of them uses Lisp in normal (though occasionally somewhat sophisticated) idiomatic usage. If the reader is unfamiliar with Lisp, perhaps seeing the brevity of these programs would be an incentive to gain familiarity.

10 Appendix: FFT multiplication

In the spirit of providing explicit programs in ANSI Standard Common Lisp, we provide a Lisp FFT program.

This program, which can easily be improved in various ways (as show in the generic arithmetic package from which it is extracted) can be run with no changes, in any supported (floating point) precision, which

usually means single or double precision in Lisp. A "generic" version can be compiled from the same program using quad-doubles, from which coefficients can reconstructed that are 60 decimal digits or so. Furthermore, arbitrary other precisions can be set using MPFR or other software packages, but these involve some subtleties, like the need for computing π to sufficient precision.

Details in `http://www.cs.berkeley.edu/~fateman/generic`.

```
(defun ptimesfft(r s) ; r and s are arrays compute the size Z of the
  ;; answer. Z is a power of 2.  compute complex array r, increased to
  ;; size Z compute complex array s, increased to size Z compute two
  ;; FFTs, then multiply pointwise, then compute inverse FFT * 1/n
  ;; finally convert back to array and return answer.
  (let* ((lr (length r))
         (ls (length s))
         (lans (+ lr ls -1))
         (z (ash 1 (ceiling (log lans 2)))) ; round up to power of 2
         (rfft (four1 (v2dfa r z) z))
         (sfft (four1 (v2dfa s z) z)))
    (dfa2v(four1 (prodarray rfft sfft z ) z :isign -1) lans)))

(defun v2dfa(a &optional (m (length a)) )
  ;; coerce a vector of numbers of length m to an array of length
  ;; 2m, since here complex numbers are stored in 2 adjacent
  ;; locations.
  (let ((k (length a))
        (ans (make-array (* 2 m) :initial-element 0.0d0))
        (zz  0.0d0))
    (dotimes (i k)
      (setf (aref ans (* 2 i)) (aref a i)))
    ans))

(defun dfa2v(a &optional (m (/ (length a)2)))
  ;; Coerce real parts back to integers, more or less.  a is an an
  ;; array of even length.  If you know that there are trailing zeros,
  ;; set the actual length with the optional second parameter m.
  (let* ((k (/ (length a) 2))
         (ans (make-array m)))
    (dotimes (i m ans)
      (setf (aref ans i)(round  (aref a (* 2 i)) k)))))

;; a simple fft, in-place,  not optimized
(defun four1 (data nn &key (isign 1))
  (let ((wr 0.0d0)       (wi 0.0d0)
        (wpr 0.0d0)      (wpi 0.0d0)
        (wtemp 0.0d0)    (theta 0.0d0)
        (tempr 0.0d0)    (tempi 0.0d0)
        (j 1) (n (* 2 nn)) (m 0) (mmax 0) (istep 0))
    (do ((i 1 (+ i 2)))
        ((> i n) t)
      (when (> j i)
        (setf tempr (aref data (1- j)))
        (setf tempi (aref data j))
        (setf (aref data (1- j)) (aref data (1- i)))
        (setf (aref data j) (aref data i))
```

```
        (setf (aref data (1- i)) tempr)
        (setf (aref data i) tempi))
      (setf m (floor (/ n 2)))
   label1
      (when (and (>= m 2) (> j m))
        (setf j (- j m)) (setf m (floor (/ m 2)))
        (go label1))
      (setf j (+ j m))) ;end do
    (setf mmax 2)
  label2
    (when (> n mmax)
      (setf istep (* 2 mmax))
      (setf theta  (/ #.(* 2 pi) (* isign mmax)))
      (setf wpr  (* -2 (expt (sin (* 1/2 theta)) 2)))
      (setf wpi (sin theta)) (setf wr 1.0d0) (setf wi 0.0d0)
      (do ((m 1 (+ m 2)))
          ((> m mmax) t)
        (do ((i m (+ i istep)))
            ((> i n) t)
          (setf j (+ i mmax))
          (setf tempr  (- (* wr (aref data (1- j)))
                          (* wi (aref data j))))
          (setf tempi (+ (* wr (aref data j))
                         (* wi (aref data (1- j)))))
          (setf (aref data (1- j)) (- (aref data (1- i)) tempr))
          (setf (aref data j)  (- (aref data i) tempi))
          (setf (aref data (1- i)) (+ (aref data (1- i)) tempr))
          (setf (aref data i) (+ (aref data i) tempi)))
        (setf wtemp wr)
        (setf wr (+ (* wr wpr) (* -1 wi wpi) wr))
        (setf wi (+ (* wi wpr) (* wtemp wpi) wi)))
      (setf mmax istep)
      (go label2))
    (return data)))

(defun prodarray(r s len)
  ;; r and s are the same length arrays
  ;; compute, for i=0, 2, ..., len-2
  ;; ans[i]:=  r[i]*s[i]-r[i+1]*s[i+1] ;; real part
  ;; ans[i+1]:=r[i]*s[i+1]+s[i]*r[i+1] ;; imag part
  (let ((ans (make-array (* 2 len))))
  (dotimes (i len ans)
    (let* ((ind (* 2 i))
           (ind1 (1+ ind))
           (a (aref r ind))
           (b (aref r ind1))
           (c (aref s ind))
           (d (aref s ind1)))
      (setf (aref ans ind) (- (* a c)(* b d)))
      (setf (aref ans ind1) (+ (* a d)(* b c)))))))
```

11 Appendix 2: polynomials mapped to numbers

```
;;; source for polysbyGMP.lisp  September 2003
;;; Using arrays of gmps.

(defun tobigG(p logv)
  (let ((res (create_mpz_zero)))
    (do ((i (1- (length p)) (1- i)))
        ((< i 0) res)
      (mpz_mul_2exp res res logv);; res <- res*2^logv
      (mpz_add res res (togmp (aref p i))))))

(defun frombigG(i logv)
  (let* ((ans nil)
         (q (create_mpz_zero))
         (r (create_mpz_zero))
         (one (create_mpz_zero))        ; will be one
         (vby2 (create_mpz_zero))       ; will be v/2
         (v (create_mpz_zero))          ; will be v
         (signum (>= (signum (aref i 1)) 0))); t if i non-neg
    (mpz_set_si one 1)                  ; set to one
    (mpz_mul_2exp vby2 one (1- logv))   ; set to v/2
    (mpz_mul_2exp v one logv)           ; set to v
    (if signum nil (mpz_neg i i))       ; make i positive
    (while (> (aref i 1) 0)             ; will be 0 when i is zero
      (setf r (create_mpz_zero))
      (mpz_fdiv_r_2exp r i logv)        ; set r to remainder
      (mpz_fdiv_q_2exp q i logv)        ; set q to quotient
      (cond ((> (mpz_cmp r vby2) 0)     ; it is a negative coef.
             (if signum (mpz_sub r r v) (mpz_sub r v r))
             (push r ans)
             (mpz_add i q one) )
            (t
             (if signum nil (mpz_neg r r))
             (push r ans)
             (setf i q))))
    (coerce (nreverse ans)array)))

(defun padpower2g(p1 p2);; p1 and p2 are arrays of gmps
  (+ (integer-length (min (length p1) (length p2)))
     (mpz_sizeinbase(maxcoefg p1) 2)
     (mpz_sizeinbase (maxcoefg p2) 2)))

(defun maxcoefg(p1)
  ;; assume coefs are gmp numbers, use gmp arith
  ;; accumulate pos and negs separately to avoid computing/storing abs.
  ;; compare at end and make positive.
  (let ((pans (create_mpz_zero))(mans (create_mpz_zero)) temp)
    (do ((i (1- (length p1)) (1- i)))
        ((< i 0)(mpz_neg mans mans)     ; make minus ans into pos
         (if (> (mpz_cmp mans pans) 0) mans pans)); return larger
```

```
      (setf temp (aref p1 i))
      (if (< (aref temp 1) 0)
          (if (< (mpz_cmp mans temp)0) nil (setf mans temp))
        (if (> (mpz_cmp pans temp) 0)nil (setf pans temp))))))

(defun polymultg(p q);; use GMP bignum stuff to multiply polys with gmp coefs
  (let((logv (padpower2g p q)))
    (frombigG (mp* (tobigG q logv)(tobigG p logv)) logv)))

(defun ppowerg(p n) (if (= n 1) p (polymultg p (ppowerg p (1- n)))))
```

12 Appendix 3: Multivariate polynomials mapped to univariate

```
;;; Simple code for multiplying multivariate polynomials by mapping to univariate

(defun max-expons-multivar(p)
  ;; p is a polynomial in the form below. Return a list of the max exponents
  ;; (#((5 4 3) (9 7 0)) .   ;;exponents
  ;;  #(30 21)  )            ;;coefs
  ;;  30*x^5*y^4*z^3+  21*x^9*y^7*z^0
  (let ((h (map 'array #'(lambda(z)0) (aref (car p) 0)))) ;zero out an array
    (map nil #'(lambda(r) (setf h (map 'array #'max h r)))(car p))
    h))
(defun bound-expons-product(p1 p2)
  ;; returns a list of the maximum exponents that will occur in p1*p2, polynomials
  (map 'list #'+ (max-expons-multivar p1)(max-expons-multivar p2)))

(defun sumlist(h n)(cond((null h) nil) ;; (sumlist '(a b c) 0) returns list (a a+b a+b+c).
                        (t (let((z(+ (car h) n)))
                             (cons  z (sumlist (cdr h) z))))))

(defun mul-multivar-direct (p q)
  ;; use hash tables, multiply directly WITHOUT packing exponents
  (let* ((ans  (make-hash-table :test 'equal ))
         (ep (car p))    (cp (cdr p))
         (eq (car q))    (cq (cdr q))
         (cpi 0)         (epi 0))

    (dotimes (i (length ep)
                ;; this answer is not in the right form.
                ;; needs to be sorted and converted to pair of arrays.
                 (l2pam
                  (sort
                   (loop for x being the hash-key in ans
                         and y being the hash-value in ans unless (= y 0)
                         collect (cons x y))
                   #'lexgp :key #'car)
                  (hash-table-count ans)))
      (setf epi (aref ep i) cpi (aref cp i))
      (dotimes (j (length eq))
        (incf (gethash (map 'list #'+ epi(aref eq j)) ans 0)
```

```
              (* cpi(aref cq j)))))))

(defun l2pam  (z &optional (n (length z)))
    ;; list to pair of arrays, multivar
  (let ((anse (make-array n))
        (ansc (make-array n))
        (exponsize (length (caar z)))
        (j (1- n)))

    (declare (fixnum j n exponsize))
    (loop (if (< j 0) (return (cons anse ansc)))
      (setf (aref anse j) (make-array exponsize :initial-contents (caar z)))
      (setf (aref ansc j) (cdar z))
      (decf j)
      (pop z))))

(defun lexgp(a b)
  (cond((null b) nil)
       ((> (car a)(car b)) t)
       (t (and (= (car a)(car b))(lexgp (cdr a)(cdr b))))))

;; this version encodes multivariate into univariate, then multiplies

(defun mul-multivar(r s &optional (mulprog #'mul-naive))
  ;; multiply r and s; mulprog is the univariate mult prog to use
  (multiple-value-bind (in out)(make-coders r s) ; set up the encoder and decoder
    (let((prod
          (funcall mulprog
                   (cons (map 'array in (car r))(cdr r))
                   (cons (map 'array in (car s))(cdr s)))))
      (cons ;; the re-formed exponents
       (map 'array  out (car prod)) (cdr prod)))))

(defun make-coders(p1 p2);; returns 2 programs for encoding and decoding
  (let* ((z (nreverse(bound-expons-product p1 p2)))
         (fields (mapcar #'integer-length z))
         (sumfields (cons 0(sumlist fields 0)))
         (maskfields  (mapcar #'(lambda(r)(1- (ash 1 r))) fields)))
    ;; encoder  vector to integer
    (values
     #'(lambda(v)
         (reduce #'+ (map 'list  #'ash (reverse v)  sumfields)))
     ;; decoder  integer to vector
     #'(lambda(i)(let ((q (make-array 5 :adjustable t :fill-pointer 0)))
                   (do ((u fields (cdr u))
                        (masks maskfields (cdr masks)))
                       ((null u) (nreverse q))
                     (vector-push-extend (boole boole-and (car masks) i) q)
                     (setf i (ash i (- (car u))))) )))))
```

13 Appendix 4 - Radix Tree multiplication

```
s;;; A multi-way tree for storing sparse indexed items. Although I just
;;; made this up, it looks like it is basically around in the
;;; literature, sometimes called a radix tree or a crit bit tree.  It
;;; is an alternative to a hash table or an array, and is, in some
;;; sense, a data structure that can be made more array-like or more
;;; tree-like. The index or key is always an integer, perhaps a
;;; bignum.

;;; The size of the nodes is parameterized: each internal node is a
;;; small array, say of length 8 =2^logsize Then <key, data> is
;;; stored in a node by decomposing the key into 3 =logsize bits at
;;; a time per layer in tree.  thus 30-bit keys will have depth 10
;;; max.  The initial sub-keys are the rightmost bits, and the remaining
;;; keys are computed by shifting the keys by logsize at each level.

;;; The key is never in fact stored in the tree, but computed by position.
;;; There is one subtlety worth noting.

;;; Note that with logsize 3, we can store items with keys
;;; [0,1,...,7] in the topmost node.  If we need to store an item with
;;; key 9, it has the same trailing 3 bits as 1. To accomodate, we
;;; move the 1 down exactly one level.  That is, the topmost node
;;; looks like [0,p,...,7] where p=[1,9,nil,nil...nil].

;;; so an key like 1  be at the top level (or maybe 1 down).

;;; CON: Keys must be integers or mapped to integers.  Traversing the
;;; tree "in order" is tricky. Numerically adjacent keys will not be
;;; adjacent in the tree, except in the trivial case where all keys
;;; fit in one node.  Nodes may be largely empty.  A key with N bits
;;; will take about (N/logsize) probes.

;;; PRO: we need not know how long the longest key is, which would be
;;; the case if we were using a string-type decomposition starting
;;; from the left.  Short keys will be located near the top of the
;;; tree. The keys are not stored, saving space. By increasing logsize
;;; we can make the performance closer to that of an array.  We never
;;; compare keys; we only extract sub-keys and use them as indexes
;;; into arrays.

;;; author RJF June 13, 2008
;;; made faster, Oct 22, 2008, RJF


;;;; THIS SHOULD BE REPLACED ENTIRELY WITH CODE FROM Nov 21, 2008

(eval-when (:compile-toplevel :load-toplevel :execute)
  (declaim (optimize (speed 3)(safety 1)))
 ;;   (defconstant logsize 5)           ; intermediate node will have 2^logsize slots
  (defconstant logsize 3)              ; best for debugging, not bad for running some tests.
```

```
)

(defmacro init-rtree()`(make-array #.(expt 2 logsize) :initial-element 0))
(defmacro leafp(x) `(numberp ,x))
(defmacro nodep(x) `(arrayp ,x))

(defun update-rtree(key val tree &aux node)
  ;; we use this for multiplication or addition of polynomials
  ;; but we could do anything with the update operation below
  ;; Declarations for speed.
  (declare (optimize (speed 3)(safety 0))
           (type (simple-array t (#.(expt 2 logsize))) tree))

  (cond  ((fixnump key)(update-rtree-fix key val tree))
         ;;above, optimized for fixnum key
         ((< key #.(expt 2 logsize))
          ;; we have found the level, or level-1 for the key.
          (cond ((nodep (setf node (aref tree key)))
                 ;; if a node here, must descend one level further in tree
                 (update-rtree-fix 0 val node)); and update that node's location 0.
                (t(setf (aref tree key) (+ node val)))));; put it here.

         ;;The key has too many bits to insert at this level.
         ;;Compute the subkey and separate the rest of the key by masking and shifting.

         (t (let ((ind(logand key #.(1- (expt 2 logsize)))))
              ;; mask off the relevant bits for subkey
              (declare (fixnum ind))
              (setf node (aref tree ind))
              (cond ((arrayp node)
                     (update-rtree (ash key #.(- logsize)) val node))
                    ;;descend into tree with rest of key.
                    (t (setf (aref tree ind)
                             (update-rtree
                              (ash key #.(- logsize)) val
                              (if (leafp node)
                                  (update-rtree-fix 0 node (init-rtree))
                                (init-rtree)))))))))
  tree)

(defun query-rtree(key tree)            ;works. though we don't use it..
  (declare (optimize (speed 3)(safety 0))
           (type (simple-array t (#.(expt 2 logsize))) tree))
  (cond  ((< key #.(expt 2 logsize))
          (if (arrayp (aref tree key)) (query-rtree 0 (aref tree key))
            (aref tree key)))
         (t (let* ((ind(logand key #.(1- (expt 2 logsize))))
                   (h (aref tree ind)))
              (if h (query-rtree (ash key #.(- logsize)) h) nil)))))

(defun maptree (fn mt);; order appears jumbled, actually reversed-radix order
  ;; apply fn, a function of 2 arguments, to key and val, for each entry in tree.
  ;; order is "radix reversed"
```

```
  (declare (optimize (speed 3)(safety 0)))
  (labels
      ((tt1 (path displace mt);; path = path to the key's location
            (cond ((null mt) nil)
                  ((leafp mt) (funcall fn path mt));; function applied to key  and val.
                  (t;; must be an array
                   (do ((i 0 (1+ i)))
                       ((= i #.(expt 2 logsize)))
                     (declare (fixnum i))
                     (tt1 (+ (ash i displace) path) (+ displace logsize)(aref mt i)))))))
    (tt1 0 0  mt)))

#| example
(setf mt (init-rtree))
(dotimes (i (expt 8 3))(update-rtree i i mt))
mt  ;;; a fully populated b-rtree with value n at location n, 0<= n < 8^3
(ordertree-nz mt)
|#

(defun mul-multivar-rtree(r s)
  (declare(optimize (speed 3)(safety 0)))
  (multiple-value-bind (in out)(make-coders r s)
    (labels ((incode (z)(mapcar #'(lambda(h)(cons (car h)(funcall in (cdr h)))) z)))
        (setf r (ordertree-nz (mul-rtree (incode r) (incode s))))
        (dolist (h r r)(setf (car h)(funcall out (car h)))))))

(defun A2PA(A) ;; array of (exp . coef) to 2 arrays.
  (let* ((LA (length A))
         (V nil)
         (anse (make-array LA))
         (ansc (make-array LA)))
    (dotimes (i LA (cons anse ansc))
      (setf V (aref A i))
      (setf (aref anse i) (car V))
      (setf (aref ansc i) (cdr V)))))



(defun mul-rtree (r s)                  ;multiply two polynomials, in arrays
  (declare (optimize (speed 3)(safety 0)))
  (let* ((er (car r)) (es (car s))
         (cr (cdr r)) (cs (cdr s))
         (cri 0) (eri 0)
         (ler (length er))
         (les (length es))
         (ans (init-rtree)) )
    (declare (type (simple-array integer (*)) cr cs er es)
             (fixnum ler les))          ;maybe some others are fixnums too..
    ;; do the NXM multiplies into the answer tree
    (dotimes (i ler (A2PA (ordertree-nz ans)))
      (declare (fixnum i))
      (setf cri (aref cr i) eri(aref er i))
      (dotimes (j les)
```

```
        (declare (fixnum j))
        (update-rtree (+ eri(aref es j)) (* cri(aref cs j)) ans)))))

(defun mul-rtree-nosort (r s);multiply two polynomials, in arrays, result in tree
  (declare (optimize (speed 3)(safety 0)))
  (let* ((er (car r)) (es (car s))
         (cr (cdr r)) (cs (cdr s))
         (cri 0) (eri 0)
         (ler (length er))
         (les (length es))
         (ans (init-rtree)) )
    (declare (type (simple-array integer (*)) cr cs er es)
             (fixnum ler les))          ;maybe some others are fixnums too..
    ;; do the NXM multiplies into the answer tree
    (dotimes (i ler ans                 ;(L2PA (ordertree-nz ans))
               )
      (declare (fixnum i))
      (setf cri (aref cr i) eri(aref er i))
      (dotimes (j les)
        (declare (fixnum j))
        (update-rtree (+ eri(aref es j)) (* cri(aref cs j)) ans)
        ))))

;;; The program update-rt1 below is a more general update function.
;;; First we show how we could use it for multiplication or addition.

(defun update-rtree+ (key val tree) ;; same as update-rtree, using rt1
  (update-rt1 key val tree #'(lambda(val oldval)(+ val (or oldval 0)))))

(defun update-rt1(key val tree fn)
  (labels((update-rtree
           (key val tree)
           ;; we use this for multiplication or addition of polynomials
           ;; but we could do anything with the update operation below
           ;; carefully, perhaps overly, declared for speed.
           (declare (optimize (speed 3)(safety 0))
                    (type (simple-array t (#.(expt 2 logsize))) tree))

           (cond  ((fixnump key)(update-rt1-fix key val tree fn))
                  ((< key #.(expt 2 logsize)) ; we have found the level, or level-1 for the key.
                   (cond ((nodep (aref tree (the fixnum key))); if a node here, must descend one level
                          (update-rtree 0 val (aref tree (the fixnum key)))); and update that node's lo
                         (t
                          ;; we have a leaf node to update.
                          (setf (aref tree (the fixnum key))
                                (funcall fn val (aref tree (the fixnum key)))))))
                  ;;The key is still too big. Compute the subkey and the rest of the key by masking and
                  (t (let* ((ind(logand key #.(1- (expt 2 logsize)))); mask off the relevant bits for su
                            (h (aref tree ind)))
                       (declare (type (simple-array t (#.(expt 2 logsize))) tree)(fixnum ind))
                       (cond ((arrayp h)
                              (update-rtree (ash key #.(- logsize)) val h));descend into tree with rest
                             ((leafp h) ; leaf, not link. We make a subtree and move the leaf down one
```

```
                              (setf (aref tree ind)
                                    (update-rtree (ash key #.(- logsize)) val (update-rtree 0 h (init-r
                             ;; h is nil means the node was empty. Just make a new subtree and insert.
                             (t (setf (aref tree ind)(update-rtree (ash key #.(- logsize)) val (init-rt
          tree))
    (update-rtree key val tree)))


(defun ordertree-nz(mt);;   order-tree the tree removing zeros.  sort after the whole tree is traversed
  (declare (optimize (speed 3)(safety 0))
           (type (simple-array t (#.(expt 2 logsize))) tree))
  (let ((ans (make-array 10 :adjustable t :fill-pointer 0)))
     (declare   (type (array t (*)) ans))
  (labels
      ((tt1 (path displace mt);; path = path to the key's location
            (cond ((null mt) nil)
                  ;; line below changed to return nil if mt is zero
                  ((leafp mt)(unless (zerop mt)
               (vector-push-extend (cons path mt) ans)))
                  (t;; mt must be an array
                   (do ((i #.(1- (expt 2 logsize)) (1- i)))
                       ((< i 0) ans)
                     (declare (fixnum i) (type (array t (*)) mt))
                    (tt1 (+ (ash i displace) path) (+ displace logsize)(aref mt i)))))))
    (sort (tt1 0 0 mt) #'< :key #'car))))


(defun update-rtree-fix(key val tree &aux node)  ;;  optimization for  key being fixnum; minor hacks
  (declare (optimize (speed 3)(safety 0))
           (type (simple-array t (#.(expt 2 logsize))) tree)
           (fixnum key))

  (cond  ((< key #.(expt 2 logsize))    ; we have found the level, or level-1 for the key.
          (cond ((nodep (setf node (aref tree key))); if a node here, must descend one level further in
                 (update-rtree-fix 0 val node)); and update that node's location 0.
                (t(setf (aref tree key) (+ node val)))));; put it here.

          (t (let ((ind(logand key #.(1- (expt 2 logsize)))))); mask off the relevant bits for subkey
              (declare (fixnum ind))
              (setf node (aref tree ind))
              (cond ((arrayp node)
                     (update-rtree-fix (ash key #.(- logsize)) val node));descend into tree with rest o
                    (t (setf (aref tree ind)
                             (update-rtree-fix (ash key #.(- logsize)) val
                                               (if (leafp node)
                                                   (update-rtree-fix 0 node (init-rtree))
                                                 (init-rtree)))))))))
  tree)

(defun update-rt1-fix(key val tree fn)
  (labels((update-rtree;;internally, use fixnum keys
           (key val tree)
           (declare (optimize (speed 3)(safety 0))
```

```
                (type (simple-array t (#.(expt 2 logsize))) tree)
                (fixnum key))

        (cond  ((< key #.(expt 2 logsize)); we have found the level, or level-1 for the key.
                (cond ((nodep (aref tree (the fixnum key))); if a node here, must descend one level
                       (update-rtree 0 val (aref tree (the fixnum key)))); and update that node's lo
                      (t
                       (setf (aref tree (the fixnum key))
                             (funcall fn val (aref tree (the fixnum key)))))))


               (t (let* ((ind(logand key #.(1- (expt 2 logsize)))); mask off the relevant bits for s
                         (h (aref tree ind)))
                    (declare (type (simple-array t (#.(expt 2 logsize))) tree)(fixnum ind))
                    (cond ((arrayp h)
                           (update-rtree (ash key #.(- logsize)) val h));descend into tree with rest
                          ((leafp h) ; leaf, not link. We make a subtree and move the leaf down one
                           (setf (aref tree ind)
                                 (update-rtree (ash key #.(- logsize)) val (update-rtree 0 h (init-r
                          (t (setf (aref tree ind)(update-rtree (ash key #.(- logsize)) val (init-rt
        tree))
  (update-rtree key val tree)))
```

14 Appendix 4: Chinese Remainder Algorithm

How does this work? Let $N = n_0 n_1 ... n_k$. Then the Chinese Remainder Theorem tells us that we can represent any number x in the range $-(N-1)/2$ to $+(N-1)/2$ by its residues modulo n_0, n_1, n_2, ..., n_k.

Conversion to Normal Form (Garner's Alg.) Converting to normal rep. takes k^2 steps.

Beforehand, compute inverse of n_1 mod n_0, inverse of n_2 mod $n_0 n_1$, and also the products $n_0 n_1$, etc.

Aside: how to compute these inverses: Extended Euclidean Algorithm.

Input: x as a list of residues $\{u_i\}$: $u_i = x \bmod n_i$

Output: x as an integer in $[-(N-1)/2, (N-1)/2]$. (Other possibilities include x in another range, also x as a rational fraction!)

Consider the mixed radix representation

$$x = v_0 + v_1 * n0 + v - 2 * (n_0 * n_1) + ... + v_k * (n_0 * ... n_{k-1}) \qquad (*)$$

if we find the v_i, we are done, after k more mults. $v_0 = u_0$, each being x mod n_0

Next, computing remainder mod n_1 of each side of (*)

$u_1 = v_0 + v_1 * n_0 \bmod n_1$, with everything else dropping out

so $v_0 = (u_1 - v_0) * n_0^{-1}$ all mod n_1

in general,

$v_k = (u_k - [v_0 + v_1 * n_0 + ... + v_{k-1} * (n_0 ... n_{k-2})]) * (n_0 * ... * n_{k-1})^{-1} \bmod n_k$.

Note that all the v_k are small numbers and the products of $\{n_i\}$ are precomputed.

Cost: If we find all these values in sequence, we have k^2 multiplication operations, where k is the number of primes needed. In practice we pick the k largest primes we can, subject to the other constraints (e.g. we would like the arithmetic to be single-precision, and fast). If we can do 31-bit signed arithmetic fast, and L is the largest number in the answer, k is about $2\log(L/2^{31})$

15 Appendix 5: Heap management

Code is provide in the shortprog.tex file. Needs cleaning up. Also testing data.

References

[1] ATLAS, Automatically Tuned Linear Algebra Software, `http://math-atlas.sourceforge.net/`

[2] Brown, W.S. "A language and system for symbolic algebra on a digital computer", in Kuo, F. F. and J. F. Kaiser (eds.), *System Analysis by Digital Computer*, John Wiley, New York, 1966 349–369

[3] Richard Fateman. "Comparing the speed of programs for sparse polynomial multiplication," SIGSAM Bulletin 37, 1 March 2003.

[4] Fredman, M. L. " How good is the information theory bound in sorting?" *Theoret. Comput. Sci.*, 1:355-361, 1976.

[5] Yozo Hida, "Data Structures and Cache Behavior of Sparse Polynomial Multiplication," Class project CS282, UC Berkeley, May, 2002. `http://www.cs.berkeley.edu/~fateman/papers/yozocache.pdf`

[6] Johnson, S.C., "Sparse Polynomial Arithmetic", *SIGSAM Bull.*, vol 8 issue 3 (August 1974) 63—71.

[7] Moenck, R. "Practical Fast Polynomial Multiplication," ISSAC 1976 (SYMSAC 1976). ACM.

[8] Michael Monagan, Roman Pearce. "Polynomial Division Using Dynamic Arrays, Heaps, and Packed Exponent Vectors" in *Computer Algebra in Scientific Computing*, Lecture Notes in Computer Science Volume 4770/2007, 2007 295–315

[9] Zippel, R. "Interpolating polynomials from their values," *J. Symb. Comput. 9*, 3, 375–403, 1990