Can you save time in multiplying polynomials by encoding them as integers?

Richard J. Fateman
University of California
Berkeley, CA 947220-1776

January 15, 2005

Abstract

Encoding a univariate polynomial as a long integer conceptually unifies some processing tasks on polynomials with those on integers, including multiplication. An evaluation of a polynomial at a single point is sufficient to encode and decode all its coefficients if the point is sufficiently larger than the coefficients. This is sometimes referred to as "Kronecker's trick". We show that this idea has practical implications for efficient programs.

1 Introduction

I was recently prompted[1] to re-examine the practical possibilities of packing coefficients of polynomials into long integers or "bignums" and using efficient integer library routines to do polynomial multiplication.

This kind of transformation, evaluating a polynomial to map it to an integer, is sometimes called "Kronecker's trick" (see von zur Gathen and Gerhard [11] section 8 for some references and history suggesting this dates back to at least 1882). The essential equivalence of dense polynomials and long integers is fairly widely known in the symbolic computation community, to the extent that in the first (1969) edition of Knuth's *Art of Computer Programming,* volume 2 [8] we see "polynomial arithmetic modulo b is essentially identical to multiple-precision arithmetic with radix b, except that all carries are suppressed. ... In view of this strong analogy with multiple-precision arithmetic, it is unnecessary to discuss polynomial addition, subtraction, and multiplication any further in this section [4.6.1]."

Yet during the early development of computer algebra systems starting in the mid-1960s, these possibilities, if entertained at all, were generally dismissed as of theoretical interest only. Circumstances have changed. In particular we are presented with far more powerful and complicated computers. Arguably we are faced with more challenging problems. Certainly we are provided with high-quality (free) software libraries that might relieve the computer algebra systems programmers from many concerns about implementation of multi-precision integers.

In fact, the main practical motivation here for using this technique is to provide a fast implementation with less work by (re-)using others' efficient software. Somewhat paradoxically, we reduce the polynomial arithmetic problems to integer problems which are solved (conceptually at least) by thinking of the integers as polynomials. The payoff is that we can take advantage of the elaborate hand-tuning and careful choices being made among competing programmers implementing algorithms for bignum arithmetic in GMP [6] or similar packages such as ARPREC [1]. Each package presumably has a substantial independent effort mounted to

[1]by a note from Daniel Lichtblau concerning the use of GMP for multiplying polynomials in Mathematica 5.0. DL's note is in the appendix.

optimize it in different architectures, and each claims to make a careful choice of the best multiplication variation (classical to FFT with variations between) [2].

Given such a standard library, we can view a system for computer algebra polynomial manipulation as one that is initially machine independent and stated at a high level in a common language, without concern for the underlying computer architecture. Starting from this perspective we can achieve efficiency on any problem *for each machine* because – once turned over to the corresponding bignum package – the principal operations are effectively *optimized for that architecture by the authors of that package.*

Just to be clear, let us ask and then answer this obvious question, "Would it be better to directly write an FFT program for fast polynomial multiplication, instead of using this route?" The answer: In principle, yes. In practice, probably not. We've written such programs; frankly it is laborious to get all the details right, and an endless task to continue to refine such programs in the face of changes in computer architecture and memory implementation. We are also sure that our programs are not the best possible, even for the architecture we have used for our tests: certainly they are not targetted to specific *other* architectures. They are not as good as the GMP versions. Shoup in his NTL system [12] *does* do this hard work; our technique consequently is likely to be a retrograde step for NTL. (This is confirmed by our simple benchmark.)

Another argument, only occasionally plausible but in some cases perhaps even valid for NTL, is to consider if it is possible to pack multiple small coefficients into a single machine word. Then we can use a single machine add or multiply to perform operations on several coefficients at once, so long as they do not "carry" across boundary lines. Computing with 8 coefficients packed into a 64-bit word clearly has the potential to speed up calculations. Intel's MMX architecture provides a view of the same potentiality with applications in signal processing and graphics. Vector or array super-scalar operations on some computer may make the argument plausible for larger assemblages, say if one can add two vectors of 128 words at once.

Yet another technology argument for this potentially more compact encoding: if the version of the polynomial as an integer rather than a linked list allows it to be accessed in cache memory more effectively, it may provide a substantial performance gain in a modern computer cache-based memory system. Note that the encodings of polynomials require an initial scan of the data to compute a bound on coefficient sizes prior to packing into an array, and so this overhead must be incorporated in any analysis.

2 Multiplication

Take the task of multiplying two polynomials in a single variable with non-negative integer coefficients. For simplicity of exposition we will talk about multiplying a polynomial by itself, but the method is not limited to squaring. Consider the problem of squaring $p = 34x^3 + 56x^2 + 78x + 90$. The result is this polynomial:

$$1156\,x^6 + 3808\,x^5 + 8440\,x^4 + 14856\,x^3 + 16164\,x^2 + 14040\,x + 8100$$

Now take p evaluated at $x = 10^5$ and square this integer to get 1156038080844014856161641404008100. One can just read off the coefficients in the answer, modulo 10^5, 10^{10} etc.

If we choose a power of two instead of a power of 10, and if the bignum representation is binary or some power of two (say 2^{32}), then evaluation into an integer can be done by shifting and masking: packing the coefficients padded by zeros. Unpacking the coefficients is similarly a matter of shifting and masking.

Doing this efficiently requires attention[3] *to nasty details of the representation of bignums!* Nevertheless, ignoring efficiency, it is possible to state the process very simply as arithmetic, as shown below.

We seem to be done with the task, except for figuring out how much padding is necessary.

[2]In our experience with GMP we found it is not always possible to install the latest, most-specific and efficient version on a target machine, but we suspect that a slightly more generic version, pre-compiled for a class of similar machines and available with less effort, may be nearly as efficient.

[3]But not necessarily attention by US!

An exact bound could be computed by multiplying the two polynomials, but that is exactly the work we hope to avoid. If we define max-norm $N(p)$ to be the maximum of the absolute values of the coefficients in a polynomial p, and d_p to be the degree of p in the variable x, then the largest coefficient in $p \times q$ is bounded by $(1 + \min(d_p, d_q)) \times N(p) \times N(q)$.

[Sketch of proof: if $p = \sum a_i x^i$ and $q = \sum b_k x^k$ then the largest coefficients in $r = p \times q$ for fixed $N(p)$ and $N(q)$ occur when $a_0 = a_1 = \cdots = N(p)$ and $b_0 = b_1 = \cdots = N(q)$. We can factor out $N(p)$ and $N(q)$ simplifying the question then to what is the largest coefficient in a product of two polynomials with coefficients all equal to 1? The coefficient of x^k in r is$\sum_{i+j=k} a_i b_j$. There are at most $(1+\min(d_p, d_q))$ terms in such a sum, since you run out of a_i or b_j terms beyond that point.]

This bound is tight (i.e. achievable), but we expect that for some applications of polynomials over the integers the coefficients will often be much smaller than the computed bound, and there will be substantial over-padding. For applications of polynomials with coefficients in a finite field the padding may be just adequate.

After computing this bound B, assuming we have an underlying binary representation, we choose the smallest n such that $2^n > B$, pack, multiply, and unpack.

The major question then is to see if it is faster, all aspects of the implementation considered.

3 Extensions

3.1 Negative Coefficients

Assume the leading coefficient is positive. (This is not a restriction because we can convert the problem to the negation of the polynomial and then change its sign before returning the result.) We will compute using a kind of complement notation. Consider again our decimal example, but with some negative signs: $34x^3 - 56x^2 + 78x - 90$. To evaluate at 10^5 we rewrite this polynomial as $33 * (10^5)^3 \ + \ (10^5 - 56) * (10^5)^2 \ + \ 77(10^5) \ + \ (10^5 - 90)$ where we construct each of the negative numbers by "borrowing" 10^5 from the next higher digit. If that digit is itself negative, we borrow again. There is no change in converting to the integer representation; converting back simply follows the rewriting shown above. Assuming we use a binary base, we should choose an n such that $2^n > 2B$ so that we can get both + and - coefficients reconstituted. That is, we change the padding– bumping it up by a factor of two (one bit more), and if the value (remainder) extracted is more than half the padding, it is a negative number and should be adjusted. It is sometimes suggested that negative coefficients could be dealt with by splitting polynomials into positive and negative parts [11],8.4. Using complement notation is much simpler.

The description as a program to do all this can be quite short, and perhaps easier to understand than our description above.

```
 /* this is Macsyma Code */
/* convert the polynomial p(x) to an integer, with padding v */
tobig(p, x, v) := subst(v, x, p);

/* notation:
  [q,r]:divide(i,v)  sets q to quotient, r to remainder of i by v */

frombig (i,x,v):=   if (i=0) then 0
               else if (i<0) then -frombig(-i,x,v)
               else block([q,r],
                     [q,r]:divide(i,v),
                     if r>v/2 then r-v+x*frombig(q+1,x,v)
                               else r +x*frombig(q,x,v));
```

e.g. if $p = 34{*}x^3+56{*}x^2+78{*}x+90$, $h = \mathrm{tobig}(p, x, 10^6)$ gives 34000056000078000090 and $\mathrm{frombig}(h, x, 10^6)$ gives $x*(x*(34*x+56)+78)+90$ which is equivalent to p.

Actually, creating this polynomial may not be as useful as creating a list of the coefficients, which is what this next program does. The first coefficient in the list is highest degree. We've also rewritten this to use iteration rather than recursion.

```
frombigL(i,v):=block([ans:[],q,r,signum:if (i<0) then -1 else 1],
                      i:i*signum,
                      [q,r]:divide(i,v),
                      while (i>0) do
                       (if r>v/2 then (push(signum*(r-v),ans),
                                       i:q+1)
                                 else (push(signum*r,ans),
                                       i:q),
                       [q,r]:divide(i,v)),
    ans);
```

3.2 More Than One Variable

It is possible to use the same trick recursively. That is, consider a polynomial p in main variable y whose coefficients are polynomials in variable x. That is, $p = \sum_{0\le i\le n} q_i(x)y^i$. Then p can be encoded by finding some sufficient uniform padding for *ALL* the coefficients within the $\{q_i\}$ to maintain their separation, as well as another padding to maintain the separation between these $\{q_i\}$. This is not a particularly appealing encoding unless the polynomials $\{q_i\}$ are dense and of uniform degree.

3.3 Other Operations

It is possible to do more operations on the bignum form as long as the number remains a faithful representation of the polynomial with respect to the operation. Clearly we must avoid any "carry" from one coefficient to another. We can try guarding these coefficents more carefully by some heuristic computation for extra padding in the case where we do not have a convenient bound, and test to see if our result is consistent. One approach is to replace x by y^2 and thus our true polynomials have only even powers of the variable; any time there is a non-zero odd coefficient, there has been some "leakage".

Addition is easily supported. If, after doing an add, there are any non-zero coefficients of the odd powers, we have detected an invalid operation. (This idea of having guard space between fields packed into a single word was used in the ALTRAN system of the 1960's. In their design only one bit between fields was allocated, since that is all that is necessary.) For other operations we must also make sure that the absence of odd powers guarantees a valid operation, as is the case with addition of two such polynomials.

There are other operations that can be reduced to integer operations *at least sometimes* or whose results can be computed with high probability, and then tested. The most prominent is the polynomial greatest common divisor (GCD) calculation which is notoriously expensive and frequently needed in simplifying rational expressions. There is a literature on the technique here, called the Heuristic GCD (GCDHeu); it is used in the Maple and MuPAD computer algebra systems [2, 10]. The checking to make sure the (heuristic) integer answer really maps to the correct polynomial is non-trivial, and a complete GCDHeu algorithm must consider that the procedure may require more padding than suggested by the initial heuristic, and that it may still fail and that another GCD algorithm entirely should be used. (Testing of the result is included in the appendix; fallback algorithms are not.)

There are other suggestions for using this kind of Kronecker evaluation in the reduction of multivariable polynomial factoring to eliminate one or more variables (e.g. to bivariate [7]).

An important difference between our proposal here and that in the GCDHeu or factoring approach is that we advocate rapid encoding of coefficients via shifting and masking of binary quantities, not evaluation of the polynomial at some "randomized" point[4].

In some earlier experiments [9], we found areas where GCDHeu was superior to other tested algorithms, even though these tests were using the built-in Lisp (Allegro CL) bignum package (classical arithmetic), and was therefore not especially speedy compared to what we know we can do with GMP. Using a basic arithmetic suite that includes *much* speedier code for much larger sizes should make GCDHeu competitive for somewhat larger problems than shown in [9], since the integer GCD calculation can be a major portion of the expense.

More recently, von zur Gathen and Gerhard [11] (in figure 6.4) show tests that using a power-of-two evaluation point for GCDHeu is (slightly) faster on their randomly generated examples compared to other points. They do not discuss the reasons for this, or the probability of the GCDHeu heuristic failing in this case.

It is beyond the scope of this paper to exhaustively test a power-of-two GCDHeu algorithm, but worth asking a feasibility question concerning how fast we can calculate the GCDHeu. Here is one data point: consider two polynomials p and q each of degree 2000 with random coefficients bounded by 10,000. To compute the correct GCD of $p \times q$ and p (The GCD is p) took 78ms including encoding and decoding. To compute the product of p and q took 31 ms using GMP encoding. To compute the product $p \times q$ by "regular" arithmetic took 343 ms. The full cost of the GCDHeu would have to include additional confirmation steps which might dominate the time quoted, so this is a "most favorable" comparison. Yet the conventional polynomial subresultant sequence GCD algorithm took 245 seconds, or over 3,000 times longer, using programs from an earlier comparison [9], leaving considerable room for testing or re-performing the heuristic GCD with different settings. (Arguably this instance of GCD calculation is implausibly large for most uses of a CAS, although much emphasis appears to be made of far larger examples in von zur Gathen and Gerhard [11] section 6.13.)

Further analysis of the GCDHeu idea and extensions may be found, for example, in a paper by Geddes, Gonnet and Smedley [5]. Although this paper is primarily concerned with GCD, it includes an algorithm for polynomial trial division. Also see Davenport/Padgett [3].

Other possible compound operations can be performed, with suitable checks for validity, on point-evaluations and single-point "interpolation" to integers. For some operations, validity is not guaranteed by the mere absence of carries. In some cases it appears that the cost of the checks may exceed the cost of doing the job in the first place. We include in the category of "possible, but be careful" the evaluation of determinants of polynomials, because algorithms of an iterative nature with many sequential additions and multiplications can be translated into integer operations. It remains to determine if we can succeed by finding a suitable "tight" padding for the whole sequence of operations, or a probable testable padding.

4 Finite Field Operations

A particularly simple way of taking advantage of the mapping to bignums is available if the arithmetic is to be performed in a finite computation structure (usually a finite field, but weaker conditions could be supported– no inverses are needed, for example.)

In the useful case of univariate polynomials over $GF[p]$, the input coefficients might as well be integers k with $0 \leq k < P$ for some odd prime P. This is particularly appealing if P is less, perhaps much less, than the maximum single-word integer, as suggest previously.

Some caution is needed: Even though we know the coefficients can all be reduced modulo P, the bignum representation needs padding. In particular, for multiplication we must allow for a maximum coefficient size

[4]A starting evaluation point for at least one version of GCDHeu is 29 + twice the minimum of the maximum absolute values of the coefficients in the input, hardly likely to be a power of two.

of $(1 + \min(d_p, d_q)) \times P^2$, and the translation back to conventional polynomial form must do a modulo P reduction for each coefficient. We know of no especially quick method here; it seems to require unpacking and computing a remainder, then re-packing coefficients.

However, knowing the domain of computation, with a finite field we do not have to look at the coefficients to find their maximum. This saving a linear-time scan through the memory storing the polynomial.

In such finite field or modular operations it is possible to take one or several such operations and through an interpolation or lifting operation, obtain results in a larger computation structure [11].

5 Benchmarks

It is not our intention in this paper to exhaustively explore all possible ranges of coefficients and degrees and to compare to all possible other implementations. This would be a task that[5] is simply not worthwhile. It would require comparison to systems that are obscure and special-purpose, that may or may not be available for testing, and whose characteristics may change without notice. We have considerable experience with attempting to publish such figures in a reproducible fashion, and collect comments such as "Your comparison was done with version X, but you must use version X+1 which is faster; also what about system XYZ (unknown except to the referee!)"

Instead we wish to establish a feasibility criterion. Does this technique work for a sufficiently interesting example such that implementations should be considered for other systems? That is, based on the information here, should the authors of existing systems X, Y, or Z, look at the idea here?

We choose as our first point of comparison for polynomial multiplication speed, a fixed large polynomial, say $p = (x+1)^{1000}$ and translate it to a single bignum k with suitable padding. Then we compute (and time) the multiplication r=$k \times k$. Compare that to the time to directly multiply p by itself. Another comparison would be to multiply the polynomial given by polynomials of substantially different size, say $(p^3 \times p)$.

This test is unfair to the conventional representation (CR) because the CR is far more versatile. CR allows

1. more than one variable, stored sparsely.
2. wide variation in the size of the coefficients, without much loss in efficiency. That is, some may be individually bignums, others zero.
3. the result of the operation (typically multiplication) to be immediately presented in the "normal" form which does not have to be decoded or re-converted to a representation with more (or possibly less) padding for a subsequent operation.

In fact, the notion of having to change representation depending upon what you are going to do subsequently with the object is discomforting. How would you feel about a representation for a linked list which had to be entirely recomputed depending upon how many new links would be appended to it? (We are willing to manage some discomfort for a major speed improvement, and if we can encode/decode fast.)

Thus we propose to eliminate the last of these objections in a second version where we include in the bignum algorithm the cost of encoding and decoding operations. This requires looking for the maximum coefficient in each input and finding the appropriate bound for the product.

Yet this version is somewhat unfair to the bignum program performance, since we can do multiple operations in the encoding. It would be grossly unreasonable if we used the simple-minded programs given above in Macsyma's top-level lnguage. Even if the conversions are compiled into Lisp, possible from the Macsyma language, they are far from as efficient as possible. As indicated earlier, we have written a better program for encoding (`tobig`) can use bignum shifting or masking and not really "evaluate" a polynomial. A better program for decoding (`frombig`) can use bignum operations equivalent to division with remainder,

[5]In spite of some referees' suggestions to the contrary

but really just multi-word shift and logical mask operations. As a result, the coefficients would presumably be laid out in a convenient data structure of an array or list.

We have developed an efficient version of these transformations allowing us us to test appropriateness of this approach for particular sizes (degree, coefficient bound) of polynomials. These programs, in Common Lisp, are included in an appendix.

Note that if it is to be as general as the Macsyma program given here, the polynomial encoding/decoding program must be prepared to work on coefficients *which are themselves bignums*, and so the masking and shifting can't be written trivially in a language that has only fixed-length arithmetic (such as C or Fortran). This subtlety can be avoided if the polynomials are multiplied in a finite field whose modulus is substantially less than a word-size. There are such applications, particularly in factoring of polynomials, where a simple C program might perform the task and such a speedup is potentially worthwhile.

A final consideration is the issue of multiplying polynomials in the "normal" $O(nm)$ way, where n and m are the polynomial degrees of each input. Is this just too easy a target for improvement? Is it easy to improve this without going into this proposed encoding? In fact, for a large enough polynomial multiplication problem we can run faster with a modicum of additional coding using a "divide and conquer" or "Karatsuba" style multiplication. This has asymptotic growth $O(n^{1.585})$, where n is the maximum degree of the inputs, but bookkeeping costs make it more expensive for smaller polynomials. (The asymptotically faster FFT requires more than a modicum of coding.)

Consider again the problem of multiplying $p \times p$. On our current desktop computer[6], we can perform this calculation in 23.6 seconds. Using a Karatsuba version, it takes as little as 3.7 seconds. Converting this problem to a GMP number takes 63 milliseconds. For each of two similar-sized inputs, this would be 126ms[7]. The multiplication itself takes only 485ms. to produce a number with 31,251 words. The reverse transformation, also phrased in compiled Lisp but consisting mostly of calls to special GMP *division and remainder by powers of 2*, (implemented as shifts) takes an additional 186ms, for a total time of 0.8 seconds. This total then can be compared to 3.7 seconds using Karatsuba.

We can actually do somewhat better if we revisit the use of classical polynomial multiplication, but use GMP arithmetic for coefficients rather than the built-in Lisp bignums: this reduces the time rather remarkably from 23.6 to 3.58 seconds. If we make a similar transformation to the Karatsuba multiplication, we reduce its time from 3.72 seconds to 1.05 seconds. (In each case we start from GMP input coefficients to GMP output coefficients.)

Thus encoding the whole problem input into large numbers allows us to do the internal multiplication in 485 ms versus the collection of smaller (but still multi-precision) multiplications and the associated bookkeeping in 1,050 ms. about 2.6 times faster. There are various ways one might prefer the answers to be presented. We could choose to use GMP forms throughout a system, though having them all packed into one big number is not really acceptable: the padding may have to be renegotiated periodically, at each operation!

For this rather large problem, the Lisp bignum package really falls short, and the Kronecker trick gets about a factor of 2.6 over what we would get by simply using GMP as the preferred library routines for our Lisp system's bignums.

On the other hand, converting a small problem such as $p = (x + 1)^{10}$ to a GMP number and squaring takes about 3 times longer than the naive calculation in our program trials. Such a small example has to be run numerous times, and the cost of encoding and decoding dominates the savings.

How important is it to use a power-of-two pad? Intuitively it seems that shifting should be much preferable to evaluation of a polynomial. Here is what we found from timings: the encoding from an array to a bignum is not that different. Encoding a 16,000 degree polynomial whose coefficients are all less than 256 by a 10-bit pad (2^{10}) takes 62ms, while encoding by multiply/add of an odd quantity (1023) takes 234ms. Here the advantage is less than a factor of 4. The difference in decoding time is more significant. The shift/mask

[6]2.6GHz Pentium 4, Allegro CL 6.2

[7]This conversion is written in Lisp using GMP arithmetic. It is possible it might be done even faster by direct coding in the GMP number representation closer to an assembler level.

decoding takes 63 ms, but the divide/remainder decoding takes 860ms. This represents almost a factor of 14 slowdown.

If an algorithm requires that the padding must be done with (say) powers of some large prime instead of powers of two, our testing suggests that such slower encoding may severely impact the efficiency of encoding and decoding.

A final observation: one might propose to replace *all* integer arithmetic in a Lisp implementation by GMP numbers. We think that this would likely be a mistake. Most arithmetic is likely to be on small (single-word) numbers. In the system we used here, Allegro Common Lisp (version 6.2) on a typical 32-bit computer, the fixnum data type provides 30 bits: the largest positive fixnum is 536,870,911. Counting down from this number to zero using compiled Lisp takes 0.3 seconds, but using GMP arithmetic takes 267 seconds. This factor of 890 on common small-number arithmetic is unacceptable since it is used for array indexing and almost every other integer arithmetic application. (compare this to (say) C, where there is not really an alternative in the language for fixed-length arithmetic). Given this reality, most Lisp systems, as well as computer algebra systems, have at least two *underlying* integer types. Such systems tend to have a more uniform higher-level model that covers the data type "integer" but at least in compiled Lisp, advisory implementation declarations can specify that some integers will always be small, and that compilation can produce tight code [4].

6 Conclusions

We have illustrated a case in which we can get a factor of 26 speedup by mapping our naive polynomial arithmetic into clever arithmetic on GMP bignums. This speedup is real, but we can also improve the naive (but commonly used) old algorithm. That is, we can improve the standard polynomial multiplication program which is brief and reasonably straightforward, but not the most inappropriate for very large examples.

Hacking on the naive algorithm, the improvement factor is reduced to about 6 or even less: we replace the naive polynomial arithmetic by a better polynomial multiplication program (using Karatsuba divide and conquer).

A further improvement on the naive versions is possible: Given this particular example in which a polynomial $p = (x + 1)^{1000}$ is multiplied by itself, we observe that the polynomial p has coefficients as large as 10^{299}. If we replace the Lisp bignum arithmetic with GMP arithmetic, then the factor is reduced further, to about a factor of 2 improvement.

Other (smaller) examples result in less favorable results for this encoding. These examples chosen to include very sparse or multivariate examples show that that the Kronecker encoding can be factors of 5, 10, or more slower than alternatives.

Daniel Lichtblau of Wolfram Research has suggested that a suitable implementation in Mathematica 5.0 specifically to use GMP provides advantages in certain internal algorithms even for relatively small polynomials over a finite field. This may be, in part, because Mathematica does not have a particularly efficient encoding for ordinary polynomials, lacks speedups like the Karatsuba multiplication, or like the Lisp we were using, has a relatively inefficient bignum library compared to GMP.

On the suggestion of a referee, and against our better judgment (see comments above) we include some comparison timings: In Maple 7, on the same machine: let `q:=expand((x+1)^1000). expand(q*q)` takes 11.20 seconds.

Mathematica 4.1 on the same machine: let `q=Expand[(x+1)^1000]. Expand(q*q)` takes 13.156 seconds.

GP/PARI version 2.2.7(alpha) takes 1.391 seconds. Macsyma 2.4 89.2 seconds (43 in garbage collection).

This can be contrasted with a multiplication time of about 0.485 seconds using GMP bignums, called from Lisp (Allegro Common Lisp).

In some sense the "purest" comparison may be a system like NTL, assuming you are willing to write and compile a C++ program to construct the inputs and call the arithmetic routines. Using NTL's default "LIP" arithmetic takes 0.344 seconds, faster by a factor of 1.4. Recompiling NTL to use GMP 4.2.1 arithmetic (on a fresh system) takes about 0.047 seconds This is remarkably fast, about 10.3 times faster than encoding as a single GMP number. NTL's ZZX module uses one of four multiplication methods (classical, Karatsuba, and two FFT variations); in this case an FFT. This method could obvious also be programmed in Lisp, narrowing the gap between our polynomial multiplications and integer multiplications. Even faster might be to translate the whole problem's data into NTL format, link to NTL for the polynomial arithmetic, and translate back if necessary. Or performing the calculation entirely in NTL. Or perhaps there is another system even faster than NTL, in which case that could be linked to Lisp.

Our major point remains: it appears that we obtained a substantial speedup *without having to write an FFT program ourselves.* When the GMP FFT program is improved or tuned for a new architecture (or merely debugged!), we get that benefit without recoding our own programs.

Conclusion in brief: this Kronecker technique has promise for speedups of multiplication for carefully chosen domains of large dense univariate polynomials, and requires *only* a fast arbitrary-precision integer arithmetic library such as GMP. Systems in which GMP arithmetic is easily available may achieve substantial speedups.

On the other hand if you devote extensive time and effort to specifically code an equivalently clever polynomial multiplication routine, with appropriate cut-off heuristics between methods, including an FFT routine (as in NTL), then mapping to long integers will not likely run faster.

All Lisp programs used are included in the on-line appendices to this paper. The only other programs needed are those available (free) from the GMP web site [6].

Recommendation to builders of computer algebra systems: We could spend far more time on more benchmarks and vary parameters such as denseness and size of coefficients, as well as degree. We could use implementations in C, C++, GMP, ARPREC, Lisp, Maple, Mathematica etc. On different machine architectures and even different versions of compilers under different optimizations. Rather than attempt such an inevitably fruitless comparison in the hope of providing CAS system-building advice[8], we hope that this paper provides the essential impetus to try something that is likely to be simple to implement and that will possibly be quite fast if you have a good arbitrary-precision library. (Another recommendation, if you are considering implementing a CAS from scratch, and you intend to build very fast polynomial arithmetic yet again: have you considered just using the open-code systems GP/Pari or NTL?)

7 Acknowledgments

This research was supported in part by NSF grant CCR-9901933 administered through the Electronics Research Laboratory, University of California, Berkeley. Thanks to D. Lichtblau for spurring this exercise and subsequent corrections and comments. Thanks also to (anonymous) reviewers of an earlier draft for their comments.

References

[1] D. Bailey, ARPREC. http://crd.lbl.gov/ dhbailey/mpdist/

[2] B. Char, K. O. Geddes, and G. H. Gonnet, Heuristic GCD B. Char, K. Geddes, and G. Gonnet. Gcdheu: Heuristic polynomial gcd algorithm based on integer gcd computation. *J. of Symbolic Computation, 7*:31-48, 1989. The algorithm has been modified for implementation, and the analysis cleaned up. See for example http://arxiv.org/PS_cache/cs/pdf/0206/0206032.pdf

[3] J. H. Davenport, J. A. Padget: HEUGCD: How Elementary Upperbounds Generated Cheaper Data. European Conference on Computer Algebra (2) 1985: 18-28

[8]Fruitless because it will be unlikely to tell you how much *your* precise circumstances will benefit from this technique.

[4] R. Fateman,“Importing the Gnu Multiple Precision Package (GMP) into Lisp, and implications for Functional Programming”, draft. 2003.

[5] K. O. Geddes, G. H. Gonnet, T. J. Smedley: Heuristic Methods for Operations With Algebraic Numbers. (Extended Abstract). ISSAC 98 475-480

[6] The Gnu MP Home page. http://swox.com/gmp/

[7] E. Kaltofen, “A polynomial reduction from multivariate to bivariate integral polynomial factorization,” 14th ACM Symposium on Theory of Computing 1982, p 261 – 266. ISBN:0-89791-070-2

[8] D. E. Knuth, *The Art of Computer Programming*, volume 2. (1969, 1981). Addison-Wesley.

[9] P. Liao, R. Fateman. “Evaluation of the heuristic polynomial GCD” *Proc ISSAC 1995* 240–247. http://doi.acm.org/10.1145/220346.220376

[10] “Probabilistic Computation of Integer Polynomial GCDs” *J. Algorithms, 1988* 9 365–371.

[11] J. von zur Gathen and J. Gerhard, *Modern Computer Algebra*, Cambridge Univ. Press 1999, 2003.

[12] V. Shoup. NTL, (Number Theory Library) `http://shoup.net/ntl/`.

8 Appendix

DL’s email note (excerpt)

“For univariate polynomials mod p we do what is called (I believe) binary segmentation. Based on polynomial length and size of p we create large integers appropriately zero-packed, then multiply them and recover the polynomial product coefficients. An analysis will show that this compares reasonably well with direct FFT convolution methods, especially when the modulus is large compared to polynomial length. I’m pretty sure this is discussed in von zur Gathen and Gerhard[9].

As we use GMP for integer multiplication, this becomes a case where the library software decides whether to use schoolbook, Karatsuba, Toom-Cook, or NTT. My experience is this general approach is quite efficient in practice, especially as GMP is a fairly robust library. Not perfect, but fairly robust.”

9 Macsyma Programs, Some Repeated, with tests

```
/*padpower2 computes the power k such that 2^k is an adequate
pad for the multiplication of polynomials p1 and p2. Double it
if you expect positive and negative numbers. */

(pad(p1,p2,x):=(1+min(hipow(p1,x),hipow(p2,x)))*maxcoef(p1,x)*maxcoef(p2,x),
padpower2(p1,p2,x):= ceiling(log(bfloat(pad(p1,p2,x)))/log(2)))$

/* Maxcoef is a kludge to compute max abs value of coefficients in a
 polynomial.  A simple lisp program does this much faster */

maxcoef(p,x):=max (map (abs, subst(1,x,
                       substinpart("[", ratdisrep(p),0))))$
```

[9]in section 8.4

```
/* For comparison with the result of frombigL, here is
a program to make a list of the coefficients in a polynomial, zeros too.
They should be the same values if the multiplication worked. */

allcoeflist(p,x):=block([h:hipow(p,x),ans:[],head,rest],
         for k:h thru 1 step -1
                do ([head,rest]: bothcoef(p,x^k),
                                 push(head,ans),
                              p: rest),
          push(p,ans), /* last piece is the constant */
          reverse(ans))$

tobig(p, x, v) := subst(v, x, p);
/* [q,r]:divide(i,v)  sets q to quotient, r to remainder of i by v */
frombigL(i,v):=block([ans:[],q,r,signum:if (i<0) then -1 else 1],
                     i:i*signum,
                     [q,r]:divide(i,v),
                     while (i>0) do
                      (if r>v/2 then (push(signum*(r-v),ans),
                                      i:q+1)
                                else (push(signum*r,ans),
                                      i:q),
                      [q,r]:divide(i,v)),
                    ans);
```

10 Appendix: Some Common Lisp Code

```
;;; Experimentation in encoding polynomials for faster multiplication.
;;; Richard Fateman
;;; September, 2003

(eval-when (compile load)
  (declaim (optimize (speed 3) (safety 0) (space 0) (compilation-speed 0)))
  (declaim (inline svref = eql make-array)))

#| Polynomials encoded as bignum representation: a normal polynomial
 in one variable x will be represented as an array or vector.
 Example: #(x 10 30 21) means 10+30*x+21*x^2 = 10+x*(30+x*21)

 How to encode zero? #(0) works better, I think, than #().
 A Normal polynomials should have no trailing zeros. Normality not enforced.
 Without normalization,   p-p  might be #(0 0 0 0...). |#

;; tobig converts a polynomial p(x) to an integer, with padding logv
;; bits wide This essentially evaluates p at the point 2^logv.

(defun tobig(p logv)
  (declare (optimize (speed 3)(safety 0)(debug 0)))
  (declare (fixnum logv))
  (assert (arrayp p))
  (assert (integerp logv))
  (let ((res 0))
    (declare (fixnum len))
    (do ((i (1- (length p)) (1- i)))
((< i 0) res)
      (declare(fixnum i))
      (setf res (+ (aref p i) (ash res logv))))))

;; frombig converts an integer i with padding logv into the polynomial
;; that it was originally.  (Assuming that padding is big enough)
;; This uses ash and masking instead  of division with remainder.

(defun frombig(i logv)
  (declare (optimize (speed 3)(safety 0)(debug 0)) )
  (let* ((ans nil) (q 0)(r 0)
 (mv (- logv)) ;minus v needed for ash
 (vby2 (ash 1 (1- logv))) ;2^(v-1)
 (v (ash vby2 1))
 (mask (1- v))
 (signum (> i 0)))
    (setf i (abs i))
    (while (> i 0)
   (setf q (ash i mv))
   (setf r (logand i mask))
   (cond ((> r vby2)
  (push (if signum (- r v)(- v r)) ans)
  (setf i (1+ q)))
 (t
  (push(if signum r (- r)) ans)
  (setf i q))))
    (coerce (nreverse ans)'array)))

;;padpower2 computes the power k such that 2^k is an adequate pad
;;to support the multiplication of polynomials p1 and p2 in variable x.

(defun padpower2(p1 p2)  ;; p1 and p2 are arrays of lisp numbers
  (assert (arrayp p1))
  (assert (arrayp p2))
  (+ (integer-length  (min (length p1) (length p2)))
     (integer-length (maxcoef p1))
     (integer-length (maxcoef p2))))

;; maxcoef is needed for computing padpower2. It returns the abs val of
;; the largest (in absolute value) of the coefficients in a polynomial.

(defun maxcoef(p1)
  ;; accumulate pos and negs separately to avoid computing/storing abs.
  ;; compare at end and make positive.
  (declare (optimize (speed 3)(safety 0)(debug 0))
   (type (array t) p1 ))
  (let ((pans 0)(mans 0) temp)
    (do ((i (1- (length p1))  (1- i)))
((< i 0)(max pans (- mans)))
 (declare (fixnum i))
 (setf temp (aref p1 i))
 (if (< temp 0)(setf mans (min mans temp))
   (setf pans (max pans temp))))))

;; round trip test to see if tobig and frombig work

(defun rt(p)(let((logv (padpower2 p p)))
      (frombig (tobig p logv) logv)))

 ;; use LISP bignum arithmetic to multiply polys with Lisp coefs

(defun polymultb(p q)
  (let((logv (padpower2 p q )))
    (frombig (* (tobig q logv)(tobig p logv)) logv)))

;; use LISP  bignum arithmetic to compute a power of p
(defun ppowerb(p n) (if (= n 1) p (polymultb p (ppowerb p (1- n)))))

;;Now try the same thing with GMP arithmetic

(defun tobigG(p logv)
  (declare (optimize (speed 3)(safety 0)(debug 0)))
  (declare (fixnum logv))
  (assert (arrayp p))
  (assert (integerp logv))
  (let ((res (create_mpz_zero)))
    (declare (fixnum len))
    (do ((i (1- (length p)) (1- i)))
((< i 0) res)
      (declare(fixnum i))
      (mpz_mul_2exp res res logv) ;; res <- res*2^logv
      (mpz_add res res (togmp (aref p i))))))

(defun frombigG(i logv)
  (declare (optimize (speed 3)(safety 0)(debug 0)) )
  (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
  (assert (arrayp p1))
  (assert (arrayp p2))
  (+ (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.
  (declare (optimize (speed 3)(safety 0)(debug 0))
   (type (array t) p1))
  (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
 (declare (fixnum i))
 (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)))))


;;; .........................some GMP/lisp utilities

;; convert numbers to gmp numbers if they aren't already
(defun togmp(x)(if (numberp x)(lisp2gmp x) x))
;; gmp sequence to lisp sequence
(defun gs2l (s)(map 'array #'gmp2lisp s))
;; lisp sequence to gmp sequence
(defun ls2g (s)(map 'array #'lisp2gmp s))

;;Here are programs to deal with bignums. I don't know how specifically
;; it is dependent on implementation, but it works on intel allegro 6.2.
;;Convert a bignum into an array of 16-bit fixnums

(eval-when (compile load)(require :llstructs))

(defun nth-bigit (x n) (sys:memref x -14 (* 2 n) :unsigned-word))
(defun bignum-size(x) ;; in 16-bit quantities
  (excl::bm_size x))

;; btoa works for any bignum or fixnum.  It converts the
;; MAGNITUDE of the number into a sequence of 16-bit integers stored
;; in an array.
;; I suspect this would be a no-op in C...
;; (Re-typing an existing structure)

(defun btoa(x) ;;bignum or fixnum to array of bigits, 16 bits
  (if (fixnump x) (vector (abs x))
  (let* ((size (bignum-size x))
 (ans (make-array size :element-type '(unsigned-byte 16))))
    (declare (optimize (speed 3)(safety 0)(debug 0))(fixnum size))
    (do ((i 0 (1+ i)))
((= i size) ans)
      (declare (fixnum i))
      (setf (aref ans i)(nth-bigit x i))))))

(defun atob(a) ; array to POSITIVE integer (bignum)
  (let ((num 0))
    (declare (optimize (speed 3)(safety 0)(debug 0)))
    (do ((i (1- (length a))  (1- i)))
((< i  0) num)
      (declare (fixnum i))
    (setf num (+ (aref a i) (ash num 16))))))

;; signum works on bignums or fixnums.

;; convert array of unsigned-byte-32 to a bignum
```

```
(defun atob32(a)
  (let ((num 0))
    (do ((i (1- (length a))  (1- i)))
((< i  0) num)
    (setf num (+ (aref a i) (ash num 32))))))


;; convert lisp integer to gmp-like stuff

(defun btoa32(x)
  ;;bignum or fixnum to array of unsigned-byte-32 bigits.
  ;;This is, I think, the same as the contents of GMP integer.
  ;; (other parts of GMP integer are k, number of bigits*sign,
  ;; and allocated size >=k)

  (if (fixnump x) (vector (abs x))
    (let* ((size (bignum-size x))
   (newlen (ceiling size 2))
   (a32 (make-array newlen :element-type '(unsigned-byte 32)
    :initial-element 0)))
    (do ((i 0 (+ 2 i))) ;; collect low order 16 bits of each bigit
((>= i size) nil)
      (setf (aref a32 (truncate i 2))
(nth-bigit x i)))
    (do ((i 1 (+ 2 i)))
;; collect high order. [We do this order so even/odd sizes work]
((>= i size) a32)
      (incf (aref a32 (truncate i 2))  (ash (nth-bigit x i) 16))))
    ))

;; convert from a lisp number to gmp.
(defun lisp2gmp(x)
  (let ((r (create_mpz_zero)))
    ;; returns a gmp number equal to any  lisp integer
    (cond ((fixnump x) (mpz_set_si r x));; works faster for fixnums
  ((bignump x) (let ((a (btoa x)) ;; bignum to array.
     (h (create_mpz_zero)))
 (do ((i (1- (length a))(1- i)))
     ((< i 0) r)
   (declare (fixnum i))
   (mpz_set_si h (aref a i))
   (mpz_mul_2exp r r 16)
   (mpz_add r r h))
 (if (< x 0) (setf (aref r 1)(- (aref r 1))))
 ))
  (t (error "can't convert ~s to gmp" x) ))
    r))

;; convert a gmp number to lisp
```

```
(defun gmp2lisp(g);;
  (let* ((size (abs (aref g 1)));; number of limbs
 (signum (signum (aref g 1)))
 (ans 0)) ; the bignum result
    (do ((i (1- size) (1- i)))
((< i 0) (* signum ans))
      (declare (fixnum i))
      (setf ans (+ (let ((k (mpz_limbn g i)))
     (if (< k 0)(+ k #.(ash 1 32)) k));compensate for unsigned
   (ash ans 32))))))


(defun rtg(x) ;round trip test for number x
  (if (= x  (gmp2lisp(lisp2gmp x))) 'true 'false))

;; append two arrays
(defun append-seq(r s)(let* ((rlen (length r))
    (slen (length s))
    (new (make-array (+ rlen slen))))
(do ((i 0 (1+ i)))
    ((= i  rlen))
  (setf (aref new i)(aref r i)))
(do ((i 0 (1+ i)))
    ((= i  slen) new)
  (setf (aref new (+ i rlen ))(aref s i)))))

;;;;;;;;;;;;;now some regular polynomial arithmetic (n^2 method)

(defun polymult(p q) ;; by ordinary poly multiplication.
  (assert (arrayp p))
  (assert (arrayp q))
  (let* ((plim (length p)) ;; degree is plim-1
 (qlim (length q))
 (ans(make-array (+ plim qlim -1) :initial-element 0)))
    (declare(fixnum qlim plim)(type (array t) ans))
    (do ((i 0 (1+ i)))
((= i plim) ans)
      (declare (fixnum i))
      (do ((j 0 (1+ j)))
  ((= j qlim) nil)
(declare (fixnum j))
(incf (svref ans (+ i j))
      (* (svref p i)(svref q j)))))))

;;  compute a power of p
(defun ppower(p n) (if (= n 1) p (polymult p (ppower p (1- n)))))


;;An implementation of Karatsuba multiplication
```

```
#|
Assume p is of degree 2n-1,  i.e.  p= a*x^n+ b  where a is degree n-1 and b is degree n-1.
 assume q is also of degree 2n-1,  i.e.  q= c*x^n+d

compute U= (a+b)*(c+d)  =  a*c+a*d+b*c+b*d,  a polynomial of degree k=2*(n-1)
compute S= a*c   of degree k
compute T= b*d   of degree k
compute U = U-S-T = a*d+b*c of degree k

Then V=p*q = S*x^(2n)+U*x^n + T.

From first principles, V =p*q should be of degree 2*(2n-1) = 4n-2.
from the form above, k+2n= 4n-2.  Note that U*x^n is of degree
2*(n-1)+n = 3n-1 so that it OVERLAPS the S*x^(2n) component.  Note
that T is of degree 2*(n-1) = 2n-2 so that it OVERLAPS the U*x^(n)
component.


  (the trick is that, used recursively to multiply a*c etc,
    this saves coefficient mults, instead of nm, max(n,m)^1.585)

If the smallest of p and q is of some small degree *karatlim* 16 or so,
we will use conventional arithmetic. There may be better characterizations
of the switch-over size, but some fiddling around gave us this guess.

Is it critical that the size be odd?  What if it is not 2n-1?
Find the max degree of p, q.  If it is odd, we are set.  If it is
even, 2n, do this: p=a*x^n+ b where b is of degree n-1, but a is of
degree n. Same deal works.  |#

(defvar *karatlim* 16) ;; an estimate -- don't use karatsuba for
small sub-problems.

(defun polymultk(pp qq);; by Karatsuba.
  (assert (arrayp pp))
  (assert (arrayp qq))
  (let* ((plim (length pp))
 (qlim (length qq)))
    (declare (fixnum qlim plim)(type (array t) aa pp qq))
    (if (< (min plim qlim) *karatlim*)(polymult pp qq)
     ; don't use for small polys
      (let* (aa
     (h (truncate (max plim qlim) 2))
     (A (subseq pp (min h plim) plim))
     (B (subseq pp 0 (min h plim)))
     (C (subseq qq (min h qlim) qlim))
     (D (subseq qq 0 (min h qlim)))
     (U (polymultk(polyadd A B) (polyadd C D)))
     (TT (polymultk A C))
     (S (polymultk B D)))
(polysub-into S U 0) ; change U to  U-S
(polysub-into TT U 0) ; change U to  U-S-T
(setf aa (polyshiftleft TT (ash h 1))) ;aa = T*x^(2h)
(polyadd-into U aa h) ; change aa to T*x^(2h)+U*x^h
(polyadd-into S aa 0) ; change aa to T*x^(2h)+U*x^h +S
(pnorm aa) ;remove trailing 0's if any
))))


(defun polyshiftleft(p k) ; like multiplying by x^k
  ;; k is positive integer, p is a raw polynomial, no header
  (let* ((oldlen (length p))
 (newlen (+ oldlen k))
 (newpoly (make-array newlen :initial-element 0)))
    (do ((i  (1- oldlen) (1- i)))
((< i 0) newpoly)
      (setf (svref newpoly (+ k i))(svref p i)))))


(defun polyadd(p q)
  (let* ((plim (length p))(qlim (length q))
(aa (make-array (max plim qlim) :initial-element 0)))
      (if (> plim qlim)
  (polyadd-1 p q plim qlim aa)
(polyadd-1 q p qlim plim aa))
      ;; We could normalize here, removing all trailing zeros but one.
      ;; We would rather wait until we know we care about it.
      ;;(pnorm aa)
      ))

(defun polyadd-1 (p q pl ql aa)
  ;; p is longer, length pl. same length as array aa, filled with zeros.
  ;; p and q are unchanged.
  (do ((i 0 (1+ i)))
      ((= i pl))
    (setf (svref aa i)(svref p i))) ; copy longer input to answer
  (do ((i 0 (1+ i)))
      ((= i ql) aa)
    (incf (svref aa i)(svref q i)))); add other arg

(defun polyadd-into (p aa start)
  ;; p is longer than answer aa
  ;; p is unchanged.  aa is changed to aa+ p*x^start
  (do ((i (+ (length p) start -1) (1- i)))
      ((< i start) aa)
    (incf (svref aa i)(svref p (- i start)))))
```

```
(defun polysub-into (p aa start)
  ;; p is longer than answer aa
  ;; p is unchanged.  aa is changed to aa+ p*x^start
  (do ((i (+ (length p) start -1) (1- i)))
      ((< i start) aa)
    (decf (svref aa i)(svref p (- i start)))))

;; remove trailing zeros.
(defun pnorm (x) ; x is a vector
  (declare (optimize (speed 3)(safety 0)(debug 0)))
      (let ((x x) pos)
(declare (simple-vector x) (fixnum pos)
 (inline position-if-not coefzerofunp delete-if))
(setq pos (position-if-not #'zerop x :from-end t))
(cond
 ((null pos) #(0))  ; change if we want #() to be zero!
 ((= pos (1- (length x))) x) ; already normal
 (t (subseq x 0 (1+ pos) )))))

;; Whew! That simple idea results in a lot of programming.

;;  compute a power of p using Karatsuba
(defun ppowerk(p n) (if (= n 1) p (polymultk p (ppowerk p (1- n)))))


;;; return to the previous classical multiplication but assume
;;; that Lisp bignum arithmetic is slow; use GMP instead.


;; USE GMP coefficients with classical and with karatsuba multiplication

(defun polymultG(p q) ;; by ordinary poly multiplication on GMP numbers
  (declare (optimize (speed 3)(safety 0)(debug 0)))
  (assert (arrayp p))
  (assert (arrayp q))
  (let* ((plim (length p)) ;; degree is plim-1
 (qlim (length q))
 (ans(make-array (+ plim qlim -1) :initial-element 0)))
        (declare(fixnum qlim plim)(type (array t) ans))
    (setf ans (ls2g ans)); convert to gmp zeros
    (do ((i 0 (1+ i)))
((= i plim) ans)
      (declare (fixnum i))
      (do ((j 0 (1+ j)))
  ((= j qlim) nil)
(declare (fixnum j))
(mpz_addmul  (svref ans (+ i j))  (svref p i)(svref q j))))))
```

```
;; Karatsuba version with GMP numbers

(defun polymultkG(pp qq);; by Karatsuba.
  (assert (arrayp pp))
  (assert (arrayp qq))
  (let* ((plim (length pp))
 (qlim (length qq)))
    (declare (fixnum qlim plim)(type (array t) aa pp qq))
    (if (< (min plim qlim) *karatlim*)(polymultG pp qq)
      (let* (aa
     (h (truncate (max plim qlim) 2))
     (A (subseq pp (min h plim) plim))
     (B (subseq pp 0 (min h plim)))
     (C (subseq qq (min h qlim) qlim))
     (D (subseq qq 0 (min h qlim)))
     (U (polymultkG(polyaddG A B) (polyaddG C D)))
     (TT (polymultkG A C))
     (S (polymultkG B D)))
(polysub-intoG S U 0) ; change U to  U-S
(polysub-intoG TT U 0) ; change U to  U-S-T
(setf aa (polyshiftleftG TT (ash h 1))) ;aa = T*x^(2h)
(polyadd-intoG U aa h) ; change aa to T*x^(2h)+U*x^h
(polyadd-intoG S aa 0) ; change aa to T*x^(2h)+U*x^h +S
(pnormG aa) ;remove trailing 0's if any
))))


(defun polyshiftleftG(p k) ; like multiplying by x^k
  ;; k is positive integer, p is a raw polynomial, no header
  (let* ((oldlen (length p))
 (newlen (+ oldlen k))
 (newpoly (make-array newlen :initial-element 0)))
    (setf newpoly (ls2g newpoly))
    (do ((i  (1- oldlen) (1- i)))
((< i 0) newpoly)
      (declare (fixnum i))
      (setf (svref newpoly (+ k i))(svref p i)))))

(defun polyaddG(p q)
  (let* ((plim (length p))(qlim (length q))
 (aa (make-array (max plim qlim) :initial-element 0)))
    (setf aa (ls2g aa))
    (if (> plim qlim)
(polyadd-1G p q plim qlim aa)
      (polyadd-1G q p qlim plim aa))
    ;; We could normalize here, removing all trailing zeros but one.
    ;; We would rather wait until we know we care about it.
    ;;(pnorm aa)
    ))
```

```
(defun polyadd-1G (p q pl ql aa)
  ;; p is longer, length pl. same length as array aa, filled with zeros.
  ;; p and q are unchanged.
  (do ((i 0 (1+ i)))
      ((= i pl))
    (mpz_add (svref aa i)(svref aa i)(svref p i))) ; copy longer input to answer
  (do ((i 0 (1+ i)))
      ((= i ql) aa)
    (mpz_add (svref aa i)(svref aa i)(svref q i)))); add other arg

(defun polyadd-intoG (p aa start)
  ;; p is longer than answer aa
  ;; p is unchanged.  aa is changed to aa+ p*x^start
  (do ((i (+ (length p) start -1) (1- i)))
      ((< i start) aa)
    (mpz_add (svref aa i)(svref aa i)(svref p (- i start)))))

(defun polysub-intoG (p aa start)
  ;; p is longer than answer aa
  ;; p is unchanged.  aa is changed to aa+ p*x^start
  (do ((i (+ (length p) start -1) (1- i)))
      ((< i start) aa)
    (mpz_sub (svref aa i) (svref aa i)(svref p (- i start)))))

;; remove trailing zeros.
(defun pnormG (x) ; x is a vectora
  (declare (optimize (speed 3)(safety 0)(debug 0)))
  (let ((x x) pos)
    (declare (simple-vector x) (fixnum pos))
    (setq pos (position-if-not #'(lambda(r)(= (aref r 1) 0))  x :from-end t))
    (cond
     ((null pos) (vector (create_mpz_zero)))  ; change if we want #() to be zero!
     ((= pos (1- (length x))) x) ; already normal
     (t (subseq x 0 (1+ pos) )))))

;;  compute a power of p using Karatsuba and GMP
(defun ppowerkG(p n) (if (= n 1) p (polymultkG p (ppowerkG p (1- n)))))


;; some comparisons for timing of GMP bignums vs fixnums

(defun test1(k)  ;;count from k down to 0
  (let ((kg (lisp2gmp k))
(one (lisp2gmp 1)))
    (declare (optimize (speed 3)(safety 0)(debug 0)))
    (do ((i kg (progn (mpz_sub kg kg one) kg) ))
((= (aref i 1) 0))))) ;test for zero.  exit when i=0
```

```
(defun test2(k) ;; this is 900 times faster, if k = most-positive-fixnum
  (do ((i k (1- i)))
      ((= i 0))
    (declare (optimize (speed 3)(safety 0)(debug 0)) (fixnum i))))



;; benchmarks
#|
(setf ones (make-array 1000 :initial-element 1)) ;1
(setf bigs(make-array 1000 :initial-element (expt 2 40))); 1,000
(setf moreones(make-array 10000 :initial-element 1)) ;10,000
(setf short #(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30))
(time (prog nil (polymult ones ones))) ;;62ms
(time (prog nil (polymultk ones ones)));; 16ms

(time (prog nil (polymult bigs bigs))) ;; 410ms
(time (prog nil (polymultk bigs bigs)));; 109ms
(time (prog nil (polymultb bigs bigs)));; 188ms

(time (prog nil (setf m (ls2g bigs)))) ;; 0ms
(time (prog nil (setf ans (polymultg m m)))) ;;15 ms
(time (prog nil (gs2l ans))) ;;16 ms
                                             ;; total 31 ms

(time (prog nil (polymult moreones moreones))) ;; 5484ms
(time (prog nil (polymultk moreones moreones)));;  656ms *karatlim* is 20.. 922ms if *karatlim*=10
(time (prog nil (polymultb moreones moreones)));; 4017ms

(time (prog nil (setf m (ls2g moreones)))) ;; 16ms ;twice
(time (prog nil (setf ans (polymultg m m)))) ;; 203 ms
(time (prog nil (gs2l ans)));; 32 ms
                                            ;; total  267 ms.

(time (prog nil (ppower short 50))) ;;1140ms
(time (prog nil (ppowerk short 50)));;1159ms
(time (prog nil (ppowerb short 50)));;9884ms
(time (prog nil (gs2l(ppowerg (ls2g short) 50)))) ;; 687ms
(time (prog nil (gs2l(ppowerkG (ls2g short) 50)))) ;; 1874 ms


(setf p1000 (ppower #(1 1) 1000)) ;(x+1)^100
(time (prog nil (polymult p1000 p1000)));; 23,611ms
(time (prog nil (polymultk p1000 p1000)));; 3,723ms with *karatlim*=3
(time (prog nil (polymultk p1000 p1000)));; 5,423ms with *karatlim*=16
(time (prog nil (polymultk p1000 p1000)));; 7,000ms with *karatlim*=20
(time (prog nil (polymultb p1000 p1000))) ;; fails: creates too large a bignum for Lisp
(time (prog nil (gs2l (polymultg (ls2g p1000)(ls2g p1000))))) ;785ms
```

```
;;;; suggested by WKC: use GMP with classical or Karatsuba multiplication.
(time (prog nil (gs2l (polymultkG (ls2g p1000)(ls2g p1000))))); 1,266 ms

;;; how much is conversion, how much arithmetic?
(time (setf m (ls2g p1000))) ;47 ms
(time (prog nil (setf ans (polymultkG m m)))) ;;1,050 ms ;*karatlim*=16
(time (prog nil (gs2l ans))) ;;200 ms
(time (prog nil (setf ans (polymultGMP m m)))) ;; 3,578 ms

We made minor modifications to the classical polynomial mult
(polymult) creating polymultGMP. The changes were to to (a) assume all
inputs were arrays of GMP numbers, and (b) do all the arithmetic in
GMP data. This reduced the time from 23.6 seconds to 3.578
seconds. Turning then to converting the Karatsuba multiplication to
use GMP numbers gets the time is down to 1.05 seconds.  Thus encoding
it all into one number allows us to do the internal multiplication in
485 ms versus the collection of smaller (but still multi-precision)
multiplications and the associated bookkeeping in 1,05 ms. about 2.6
times faster.  There are various ways one might prefer the answers to
be presented.  We could choose to use GMP forms throughout a system,
though having them all packed into one big number is not really
acceptable because the padding has to be renegotiated periodically.

So for this problem, the Lisp bignum package really falls short, and
the Kronecker trick gets about a factor of 2.6.


;; times on 2.6GHz Pentium 4 Allegro CL 6.2 512 MB RAM Sept 2003.
|#
```

11 Appendix II - GMP and Lisp interface

```
;; this is from file loadgmp2.cl.
;; Set up the Allegro Common Lisp foreign function interface to GMP
;; other Common Lisp implementations have similar but not identical
;; foreign function interfaces.

(eval-when (compile load eval)
  (ff:def-foreign-type mpz (* :int))

  (ff:def-foreign-call
      (mpz_init "__gmpz_init")
      ((x (* :int)))
      :returning :int
      :arg-checking nil
      :call-direct t)

  (ff:def-foreign-call
      (mpz_set_str "__gmpz_set_str") ;changes value of mpz x.
      ((x mpz)
       (s (* :char))
       (base :int))
      :strings-convert t  :returning :int
      :arg-checking nil)
  ;;   :call-direct t

  (ff:def-foreign-call
      (mpz_get_str "__gmpz_get_str")
      ((s :int);;let it allocate the space
       (base :int)
       (op mpz))
      :returning  ((* :char) )
      :arg-checking nil :call-direct t)

  (ff:def-foreign-call
      (mpz_get_d  "__gmpz_get_d") ((x mpz))
      :returning :double
      :arg-checking nil :call-direct t)

  (ff:def-foreign-call;; remove gmp number from memory
      (mpz_clear  "__gmpz_clear") ((x mpz))
      :returning :int
      :arg-checking nil :call-direct t)

  (ff:def-foreign-call;; remove gmp number from memory
      (mpz_cmp  "__gmpz_cmp") ((x mpz) (y mpz) )
      ; returns pos if x>y, zero if x=y, neg if x<y
      :returning :int
      :arg-checking nil :call-direct t)

  (ff:def-foreign-call
      (mpz_mul  "__gmpz_mul")
      ((target mpz)(op1 mpz)(op2 mpz))
      :returning :void
      :arg-checking nil :call-direct t)

  (ff:def-foreign-call
      (mpz_add  "__gmpz_add")
      ((target mpz)(op1 mpz)(op2 mpz))
      :returning :void
      :arg-checking nil :call-direct t)

  (ff:def-foreign-call
      (mpz_sub  "__gmpz_sub")
      ((target mpz)(op1 mpz)(op2 mpz))
      :returning :void
      :arg-checking nil :call-direct t)
```

```
(ff:def-foreign-call
    (mpz_addmul  "__gmpz_addmul");;rop <-rop+op1*op2
    ((rop mpz)(op1 mpz)(op2 mpz))
    :returning :void
    :arg-checking nil
    :call-direct t)

(ff:def-foreign-call;; convert fixnum y to mpz and store in x
    (mpz_set_si  "__gmpz_set_si") ((x mpz) (y :fixnum) )
    :returning :void
    :arg-checking nil
    :call-direct t )

(ff:def-foreign-call
    (mpz_mul_2exp  "__gmpz_mul_2exp");; set target to op1*2^op2
    ((target mpz)(op1 mpz)(op2 :long)) ;****
    :returning :void
    :arg-checking nil :call-direct t)

(ff:def-foreign-call
    (mpz_sign  "__gmpz_sign");; return -1, 0, 1
    ((op1 mpz))
    :returning :fixnum
    :arg-checking nil :call-direct t)

(ff:def-foreign-call
    (mpz_size  "__gmpz_size");; number of limbs
    ((op1 mpz))
    :returning :fixnum
    :arg-checking nil :call-direct t)

(ff:def-foreign-call
    (mpz_neg  "__gmpz_neg");; set targ to -op1
    ((targ mpz)(op1 mpz))
    :returning :void
    :arg-checking nil :call-direct t)

(ff:def-foreign-call
    (mpz_limbn  "__gmpz_getlimbn");; unsigned byte-32
    ((op1 mpz) (op2 :long))
    :returning  :long ;**** might be negative?? must make unsigned
    :arg-checking nil :call-direct t)

(ff:def-foreign-call
    (mpz_fdiv_q_2exp  "__gmpz_fdiv_q_2exp");; quotient of op1 div by 2^op2
    ((target mpz)(op1 mpz) (op2 :long))
    :returning  :void
    :arg-checking nil :call-direct t)
```

```
  (ff:def-foreign-call
      (mpz_fdiv_r_2exp  "__gmpz_fdiv_r_2exp");; remainder of op1 div by 2^op2
      ((target mpz)(op1 mpz) (op2 :long))
      :returning  :void
      :arg-checking nil :call-direct t)

  (ff:def-foreign-call
      (mpz_sizeinbase "__gmpz_sizeinbase") ;number of base digits in op2 or 1 more
      ((op1 mpz) (op2 :long));; e.g. ceiling log 2 would be (mpz_sizeinbase x 2)
      :returning  :fixnum
      :arg-checking nil :call-direct t)

  ;; There are about 165 gmpz  other signed integer functions.
  )

;; In this paper we actually ignore management of a free list of GMPs
;; just create numbers with indefinite lifetimes.  In an earlier paper
;; we described how to use a reference count as well as the Lisp
;; garbage collector to remove such material.

;; create space for a number and put zero in it.
(defun
  create_mpz_zero()
  (let((newobj (make-array 4
   :element-type
   '(signed-byte 32) :initial-element 0)))
    (mpz_init newobj) ;; this sets up gmp to know about this object.
    newobj))

(defun create_mpz_from_string(s)  ;; s is a string like "123"
 (let ((r (create_mpz_zero)))
   (mpz_set_str r s 10) ;; base 10 number conversion
  r))

;; given mpz number 123, return "123"
(defun create_string_from_mp(m)  (mpz_get_str 0 10 m))

(defun create_double_from_mp(m) ;;  creates a double precision version of mpz
  (mpz_get_d m))

;; this function computes and returns the GMP product x*y

(defun mp*(x y)(let ((r (create_mpz_zero)))
 ;; this leaves r around forever.
 (mpz_mul r x y) r))
;; similarly, addition

(defun mp+(x y)(let ((r (create_mpz_zero)))
(mpz_add r x y) r))
```