On the implementation of AKS-class primality tests

R. Crandall* and J. Papadopoulos**

Abstract. Algorithms of the new "cyclotomic AKS class" determine rigorously and in polynomial time whether an input integer is prime. Herein are discussed implementation issues, with a focus on techniques such as asymptotically fast polynomial manipulations, the possible use of floating-point arithmetic, and special powering ladders. We implemented various of the ideas herein on an Apple G4, Mac OS X system, and report performance for one particular AKS variant. We note two performance issues: a) although single-precision vector floats were not employed in this test implementation, double-precision float performance was quite impressive (we quantify this), and b) there are many additional ways to invoke vectorization, if one so desires, for further speedups.

* Advanced Computation Group, Apple Computer
** University of Maryland College Park

18 Mar 2003

1. Introduction

In August 2002 the number-theoretical world was pleasantly surprised at the announcement of a paper simply titled "PRIMES is in P" [Agrawal et al. 2002]. Now known as the "cyclotomic AKS test," or just AKS test—in honor of the three authors—the idea is that the prime (or composite) character of an input integer is determined without doubt, in polynomial time. This in turn means that the time to prove or reject primality is $O(\log^\mu p)$ for some constant power μ; i.e., the run time is bounded by a polynomial function of the number of bits in p. In previous years there had appeared primality tests that "seemed to" run in polynomial time, or nearly so, or statistically so, but the AKS algorithm is the first deterministic, rigorous, polynomial-time instance. In the words of P. Leyland:

> One reason for the excitement within the mathematical community is not
> only does this algorithm settle a long-standing problem, it also does so
> in a brilliantly simple manner. Everyone is now wondering what else has
> been similarly overlooked.

We refer herein to the "AKS class" of algorithms, because in recent months there have appeared new analyses and variants, notably those of [Lenstra 2002], [Pomerance 2002], [Berrizbeitia 2003], [Cheng 2003], [Bernstein 2003a, b][Lenstra and Pomerance 2003]. The general trend is the lowering of the complexity exponent k, which began as an unconditional $k = 12 + \epsilon$ with the original AKS paper- and now stands at $k = 4 + \epsilon$ for certain input integers, or a similar low exponent for general input integers but with a possible, minuscule chance of the answer "maybe." The precise complexity exponent depends, in these various treatments, on whether one is proceeding unconditionally, or heuristically, or under the GRH, and so on. Incidentally, the current best bound—for unconditional complexity with effectively computable big-O constant—is the $O(\log^{6+\epsilon} p)$ result of [Lenstra and Pomerance 2003]; we stress that in some treatments there are either conditions or an ineffective constant—e.g. the original AKS result did not involve an effective constant. On the practical side, the speed improvements over the original AKS breakthrough are already—in just a matter of months—orders of magnitude [Bernstein 2003a] (see also Section 4 for estimates and projections).

We chose the [Lenstra 2002] variant—outlined below—for the purpose of assessing various numerical approaches and machine performance. This variant, a compromise between speed and ease of implementation, has provable runtime $O(\log^{12+\epsilon} p)$ or $O(\log^{8+\epsilon} p)$ (the former with an effective implied big-O constant) but heuristically expected to scale as $O(\log^{6+\epsilon} p)$. Indeed, with our optimizations applied, we found runtime T to behave empirically as

$$T \sim C \log^6 p,$$

where C is roughly constant over the applicable range of p (see later section for numerical values). It is understood that the $O(\log^{4+\epsilon} p)$ conditional variants—and whatever comes in future, including marriages of AKS- and elliptic-curve- methods—will be much faster; in fact we predict in loose terms how fast these would run with our kind of implementation. Moreover, the ideas presented herein can be expected to be relevant also to faster, future variants, at least as long as the cyclotomic-ring notions still apply.

2. Algorithmic issues

In the Lenstra variant of AKS, we establish that p is not a proper prime power $q^{(s>1)}$, then choose r such that the multiplicative order of p modulo r exceeds v, where

$$v = \text{round}(\lg^2 p).$$

Then we check the polynomial relation

$$(x - a)^p = x^p - a \pmod{(x^r - 1, p)}$$

for $a \in [1, \phi(r) - 1]$. Then p is prime iff the polynomial relation holds for each such a.

Again, very recent variants seek to lower the effective size of polynomials (in the above algorithm the polynomial degree is typically $r-1$), or reduce the size of the set of a values, and so on. Being as (very roughly, and heuristically) $r \approx \log^2 p$, the polynomial relation to be checked requires the following computational effort:

(number of polynomial squares/muls for p-th power of degree-$(r-1)$ polynomial) $\times$
   (number of ops per polynomial mul/square) $\times$
     (time per op modulo p) $\times$
      (number of loop passes, i.e. values of a)

which multiplies out respectively (again very roughly) as

$$(\log p) \cdot r \cdot (\log p) \cdot r \approx \log^6 p,$$

where we are dropping $\log \ldots \log p$ factors and so the complexity exponent here is really $(6 + \epsilon)$.

We next provide implementation notes that were valuable in optimizing our own machine tests of the Lenstra variant above (see final section of this paper for specific notes on the Apple G4 running MacOS X). However we stress that many of the observations below should apply also to the newer implementations, e.g. those of [Berrizbeitia 2002], [Cheng 2002], [Bernstein 2002, 2003].

It is to be stressed that many of the notions below are already resident in prevailing large-integer and algebra packages. However, for the AKS class there are some special considerations, such as the idea of intervening in some classical algorithm loops, the symbiosis of disparate algorithms with a view to an ultimately faster variant, and the exploitation of what has been learned in the last decade about floating-point dominance for large-integer arithmetic.

● **Detecting proper powers.** There are many ways to proceed; perhaps the most elegant is to check $\lfloor p^{1/b} \rfloor^b \neq p$ for all $2 \leq b < \log p$, but employ an *integer* Newton method for the truncated $1/b$ power, as in:

```
int_root(p, b) {  /* Returns ⌊p^(1/b)⌋. */
    x = 2^⌈B(p)/b⌉;  /* B(n) = number of bits in n. */
    while(1) {
        y = ⌊((b - 1)x + ⌊p/x^(k-1)⌋)/b⌋;
```

3

```
            if(y > x) return x;
            x = y;
        }
    }
```

This procedure can be further optimized; e.g., if int_root() returns an even value, there is no need to take the b-th power, and so on. However, for almost any—even sluggish—implementation of this proper-power checking, the time spent for this initial step in the AKS-class algorithms is negligible in practice, so we shall not analyze this procedure further.

• **Mod p operations.** The time-consuming loop over parameters a in the checking of

$$(x - a)^p = x^p - a \ (\mathrm{mod} \ (x^r - 1, p))$$

is the necessary focus of optimization. We envision polynomials of degree $(r - 1)$ with coefficients typically of size p. In the powering algorithm that resolves $(x - a)^p$ we have to reduce all coefficients modulo p, and to do so successively after each internal square or multiply within the powering ladder. One way to effect mod p operations is to avoid explicit division altogether; the Montgomery method, or the Barrett method and some recent extensions [Crandall and Pomerance 2002] allow such avoidance (except perhaps for some one-time initialization). We describe here an extension of the Barrett method, for which multiplication and shifting is all one requires to effect the mod operation. In fact, for the AKS-class algorithms there may be additional, interesting and unique reasons to use the Barrett method, as we discuss later. The Barrett idea is to forge a one-time "generalized reciprocal"

$$R = \lfloor 4^{B(p-1)}/p \rfloor,$$

where $B(n)$ is the number of bits in n. Happily, it turns out that even though the formula above implies a divide, this, too, can be avoided via a Newton-method algorithm for calculating R [Crandall and Pomerance 2002]. Next, we define $s = 2(B(R) - 1)$. Now, one effects a mod operation as:

$$n \ \mathrm{mod} \ p = n - p\lfloor (Rn) >> s \rfloor - \delta,$$

where δ has to be 0 or p, and so can be found with one simple compare (of the mod estimate to p itself). There are ways to speed all this up further, for example by splitting the shift operation here into two smaller shifts, or observing that part of the usual grammar-school multiplication parallelogram is demolished by the shift, and so said part need not be computed, and so on.

• **Special p.** This could be important in future speed-record-making, so we set it off as its own topic: Primes of special form can sometimes allow extremely rapid modular operations. The well known category of Mersenne primes $p = 2^q - 1$ allows a simple shift-add procedure for the mod. We admit, probably nothing can beat the polynomial-time Lucas-Lehmer test for these primes (after all the largest proved prime is a Mersenne of about 13 million bits), but there is also the less well known but also effective Gallot shifting scheme for Proth prime moduli $p = k2^q + c$ [Crandall and Pomerance 2002].

4

• **Polynomial arithmetic.** Various means exist for the multiplication of two polynomials. The one we selected for our own tests of the Lenstra variant may not be the fastest, but it is quite straightforward and leads to some intriguing future possibilities for the AKS class. The selected method is binary-segmentation multiply. To multiply two polynomials $f = \sum_i f_i x^i$ and $g = \sum_i g_i x^i$ each of degree $(r-1)$, one may forge two (typically large) integers:

$$F = 0 \ldots 0 f_{r-1} 0 \ldots 0 f_{r-2} 0 \ldots \ldots 0 f_0,$$

$$G = 0 \ldots 0 g_{r-1} 0 \ldots 0 g_{r-2} 0 \ldots \ldots 0 g_0,$$

where the "$0 \ldots 0$" parts indicate zero-pads of a length one may rigorously calculate so that the coefficients of the polynomial product fg can be read directly off the *integer* product FG. Said integer product will appear as a set of "coefficient cells" with small zero-pads between each coefficient. Note that a typical coefficient cell will have on the order of $\lg r + 2 \lg p$ bits plus any remaining zero-pad bits, and this is the starting observation for designing the integer sizes and zero-pad lengths (see [Crandall and Pomerance 2002] for details on the binary-segmentation algorithm). The zero-padding prevents coefficient sums from spilling over and interfering with each other during the integer multiply. This intriguing reduction of a polynomial operation to one involving large-integer arithmetic has various advantages such as: Ease of implementation, referral of software optimization issues to the excellent, prevailing integer packages, and true exploitation of reduced complexity for squaring. On this last point, if $f = g$ then we simply do a large-integer square F^2, and even if FFT multiplication is used, there is a gain of $3/2$ for squares over general multiplies.

• **Reduction modulo** $x^r - 1$. An interesting issue is whether to perform the (mod $x^r - 1$) reduction after all modulo p coefficient reduction, or to operate *on the integer FG* to effect (mod $x^r - 1$) reduction first, and then reduce the coefficients. One can obtain the reduced integer in a fashion reminiscent of fast Mersenne-mod: Extract the first r coefficient cells from the integer FG (think of this as an "and" operation with $2^{kr} - 1$ where k is the bit-width of a coefficient cell). Then shift the discarded part of FG to the right by kr bits and add to the extracted part. Keep shifting and adding to exhaust all cells of the integer FG.

• **Possible algorithm symbiosis.** Our implementation used an extended-Barrett mod and binary-segmentation multiply. There are advantages to the Barrett scheme for mod p reductions; for example, if yet more zero-padding is used in the binary-segmentation scheme, then one could in principle multiply the *integer FG* by the generalized reciprocal R and then shift compartmentally to obtain the desired coefficients modulo p. One could argue that processors do not like compartmental shifts; however that is not so for the newer vector processors that can perform Barrett shifting by load/stores of coefficient cells with intermediate shifting-off of bits.

And here is yet another chance for algorithm symbiosis: Consider invoking not an integer multiply for binary segmentation products, but an integer multiply modulo $2^{kr} - 1$ (to take care of the mod $x^r - 1$ operation) so that shorter run lengths in whatever transform based multiply can be effected, much in the same way that the irrational-base discrete weighted transform (IBDWT) has been used for Mersenne prime searches [Crandall and Fagin 1994]. Modern extensions of the basic ideas are found in [Percival 2003].

By "algorithm symbiosis" in this context, we refer to the notion of getting as much of the calculation as possible into large-integer format by combining various large-integer expedients of the past. Some of the newer AKS variants use different polynomial mod operations (i.e. not quite as simple as mod $x^r - 1$) and these could also allow symbiosis.

• **The big multiply.** If one uses binary segmentation and other ideas mentioned previously, one depends completely on a good integer multiply. We have already mentioned FFT methods (including the IBDWT) but then we live in a state of sin, in the sense that the FFT has roundoff errors and yet the whole point of AKS-class algorithms is rigorous proving. Here is a splendid option: Use Nussbaumer (pure-integer) convolution, in fact Nussbaumer *cyclic*, which is in practice faster than negacyclic, modulo $2^{kr} - 1$ as above. There is an extra bonus for such a choice: The small negacyclic (of say order 4, at the bottom of Nussbaumer recursion) which takes 16 ring operations, can be changed to a spectacular 9 operations in the case of negacyclic *squaring*. Other options include Schönhage multiplication (for which the usual transform speedups for squaring are allowed), fast Galois transform (FGT), and so on (see [Crandall and Pomerance 2002] for pseudocode).

Generally speaking, the fastest floating-point FFT-based multiplies are significantly faster than their pure-integer counterparts. (See Section 3 for some timing data.)

Incidentally there is a cultural scenario in which the "nonrigor" of floating-point FFT is not so troublesome. Namely, if you really want to resolve a large prime, say for cryptographic purposes, use the FFT to get a positive AKS output, and *then* invoke a pure-integer convolution, which may be slower but one is now focused on a *very* likely prime. This kind of idea—to use slower, rigorous-integer programs to "chase and verify" a "floating-point wavefront" was used to advantage in tie resolution of F_{24} as composite [Crandall et al. 2003]. In the wavefront regard, it might even be appropriate to have pure-integer machinery recreating, in parallel fashion, various parts of the painful loop over a values. That is, floating-point machinery could deposit partial results at periodic junctures within the big loop, whence the integer machinery would check these partials. The whole point is that the floating-point wavefront may finish way ahead of the integer-verification apparatus. (Indeed, in the actual F_{24} resolution the integer-checking procedure ran, literally, months longer than the overall floating-point (wavefront) run.)

• **Windowed power ladders.** For AKS-class algorithms, being as polynomial powering as in: $(y - a)^p$ is involved, it turns out to be a substantial improvement to use so-called windowed power ladders. One reason for this is that by windowing, one drives the computational work more toward squaring as opposed to general multiplication. There are many variants of windowed power ladders [Crandall and Pomerance 2002] but we did find a good compromise between complexity of coding and performance (not unlike the classical compromise in differential-equation solving presented by 4-th order Runge–Kutta). Say we have a 12-bit exponent for a construct:

$$H = h^{w_3 w_2 w_1 w_0},$$

where each word w_i is 3 bits. Then

$$H = (((h^{w_3})^8 \cdot h^{w_2})^8 \cdot h^{w_1})^8 \cdot h^{w_0},$$

so that only 9 squarings and 3 multiplies are required if one has stored h^w for all $w \in [0, 7]$ in a table. This compares quite favorably with simpler ladders. It turns out that one only need store powers for *odd* values of w, whence one modifies slightly the pattern of squaring. Such windowing allows one to exploit the relative simplicity of squaring, which is a factor of 2 in speed over grammar-school multiply, or a factor of $3/2$ for FFT squaring, or a factor greater than this for pure-integer transform squaring, say.

These is one other feature of interest: If one is invoking FFT multiply/square and were to choose a very wide window (much wider than the 3 bits above) and have a great deal of available memory, one might be tempted to store the *transforms* of h^w to render the occasional multiplies yet faster.

• **AKS-class is "embarrallel."** It is an obvious observation that AKS-class algorithms, what with their loops on some a parameter, are "embarrallel," meaning embarrassingly parallel, meaning in turn that M equivalent machines can each work independently toward an answer that will thus arise in $1/M$ of the time required by 1 machine. In fact, one can just dole out a set of a values over which a given machine will loop. What is less clear is how to dole out loops in such a way that memory can also be reduced; that is, we see immediately how to go parallel in time, but going parallel in space appears difficult.

3. Implementation

We implemented many of the techniques expressed in Section 2, including enhanced Barrett mod and binary-segmentation polynomial multiply/square. Virtually all of the runtime for an AKS-class test is spent performing modular exponentiation operations on polynomials of high degree, and so in our implementation almost all of the time is spent multiplying (and squaring) large integers.

On an Apple G4, Mac OS X platform, gcc compiler, we did *not* employ vectorization (note, though that many of the ideas of Section 2 could certainly leverage Altivec). Instead, our work was an experiment into the world of large-array double-precision computation, at which the G4 does well (and vectorization can indeed be used to enhance what we have done, as explained below). We used the "glucas" package with its double-precision-float FFT multiply, written by G. Ballester-Vamor (see [glucas homepage]) and being first and foremost a tool for performing Lucas-Lehmer primality tests. The package includes a large and complex library for the large-integer FFT arithmetic such tests require, and also includes a generic interface that lets applications use this library. More specifically, the FFT multiply code in glucas is specialized for real signals that need not be a power of two in size, needs no scratch space or bit-reversals, accounts for the cache and memory latency effects of modern microprocessors, includes machine-specific and compiler-specific code for fast arithmetic- rounding and prefetching, and incorporates balanced representation [Crandall and Fagin 1994] to reduce FFT roundoff error. No other publicly available integer multiplication library incorporates so many features in such a portable (mostly ANSI C) form, and so glucas is a natural choice for accelerating our AKS implementation. We note that various OS X libraries may also be invoked for further speedups. Technically, vecLib/vDSP has optimized functions for such as scalar-vector dyadic multiply (vamD function) and more implementation labor would certainly exploit such library calls to

advantage. Beyond this, one might contemplate single-precision, vectorized FFT—the present authors settled on the glucas package because of its having been tuned to the kinds of gargantuan integer sizes in question.

Integrating the glucas library was relatively simple. The only subtlety stemmed from the fact that our proof-of-concept AKS implementation used a large-integer library whose internal format stored numbers as arrays of 16-bit integers. This was incompatible with the glucas interface, which expected input arrays of at least 32-bit integers and so had to be modified.

4. Performance

As mentioned in the opening section, we found empirically that

$$T \sim C \log^6 p,$$

and a working value is $C \approx 1000$ clocks. A 30-decimal-digit prime thus took, on a 1 GHz. Apple G4 machine, "about a day" to be resolved. Of course, the C constant really should have $\log \log \ldots$ factors, but an approximate constancy happens over a wide range of prime candidates p.

The "glucas" FFT-multiply which allows this kind of C value in clocks turned out to be quite impressive indeed. Whereas the popular GMP library squares an integer in about 100 clocks per output bit, a cursory check shows the glucas code squaring an integer in about 13 clocks per output bit. Such is the possible difference between floating-point and pure-integer convolution schemes (though we do recognize that the glucas code has been optimized in many ways that have nothing to do with the float/integer dichotomy).

We did not attempt to implement any newer, say $O(\log^{4+\epsilon} p)$ schemes, but casual analysis of these newest schemes would allow us to resolve, on the same 1 GHz. machine, a 700-decimal-digit prime (actually, perhaps even larger—it is difficult to estimate an implementation's performance without having implemented it) in "about a day." We admit we did not effect all of the suggestions of Section 2, and again we realize many of the suggestions may not be optimal. (Note: Very recently (Mar 2003) D. Bernstein announced the resolution of an about-300 decimal digit integer likewise in about a day on an 800 MHz. Pentium III, via a casual GMP implementation, so our extrapolation into hundreds of digits for 1-day runs on modern machinery seems reasonable.)

Incidentally, by the "mole rule"—that about 10^{24} CPU operations have been performed in all of human history (perhaps we are closing in on 10^{25}), and neglecting memory constraints but using our runtime estimates above, one of the newer AKS-class variants might have resolved a 100000-digit random prime with such CPU effort. Being as the current largest explicit prime has about 4000000 digits, we see that—the new and spectacular polynomial-time discoveries notwithstanding—there is a long way to go in the matter of algorithm development.

8

References

Agrawal M, Kayal N, aod Saxena N 2002. "PRIMES in P," pseprint, 6 Aug.

Bernstein D 2003a, "Proving primality after Agrawal-Kayal-Saxena," preprint 25 Jan., http://cr.yp.to/papers.html

Bernstein D 2003b, "Proving Primality in Essentially Quartic Expected Time," preprint 28 Jan., http://cr.yp.to/papers.html

Berrizbeitia P 2002, "Sharpening 'Primes is in P' for a large family of numbers," preprint 20 Nov.

Cheng Q 2003, "Primality proving via one round of ECPP and one iteration in AKS," http://www.cs.ou.edu/∼qcheng/.

Crandall R and Pomerance C 2001, 2002 *Prime Numbers: A Computational Perspective*, Springer-Verlag, New York.

Crandall R and Fagin B 1994, "Discrete Weighted Transforms and Large-Integer Arithmetic," *Math. Comp.* **62**, 305-324.

Crandall R, Mayer E and Papadopoulos J 2003, "The Twenty Fourth Fermat Number is Composite," *Math. Comp.*, to appear.

glucas homepage: http://glucas.sourceforge.net

Lenstra H W Jr, "Primality testing with cyclotomic rings," preprint, 14 Aug.

Lenstra H W Jr and Pomesance C, "Primality testing with Gaussian periods," manuscript Mar 2003

Percival C 2003, "Rapid multiplication modulo the sum and difference of highly composite numbers, *Math. Comp.* **72**, 387-395.

Pomerance C 2002, "The cyclotomic ring test of Agrawal, Kayal, and Saxena.", preprint.