Factoring and Testing Irreducibility of Sparse Polynomials over Small Finite Fields

Richard P. Brent
Australian National University

joint work with

Paul Zimmermann
INRIA, Nancy
France

30 March 2009

Outline

- Introduction
  - Polynomials over finite fields
  - Irreducible and primitive polynomials
  - Fermat and Mersenne primes
- Part 1: Testing irreducibility
  - Irreducibility criteria
  - Modular composition
  - Three algorithms
  - Comparison of the algorithms
  - The "best" algorithm
  - Some computational results
- Part 2: Factoring polynomials
  - Distinct degree factorization
  - Avoiding GCDs, blocking
  - Another level of blocking
  - Average-case complexity
  - New primitive trinomials

Polynomials over finite fields

We consider univariate polynomials $P(x)$ over a finite field F. The algorithms apply, with minor changes, for any small positive characteristic, but since time is limited we assume that the characteristic is two, and $F = \mathbb{Z}/2\mathbb{Z} = \mathrm{GF}(2)$.

$P(x)$ is *irreducible* if it has no nontrivial factors. If $P(x)$ is irreducible of degree r, then [Gauss]

$$x^{2^r} = x \bmod P(x).$$

Thus $P(x)$ divides the polynomial $\mathcal{P}_r(x) = x^{2^r} - x$. In fact, $\mathcal{P}_r(x)$ is the product of all irreducible polynomials of degree d, where d runs over the divisors of r.

Let $N(d)$ be the number of irreducible polynomials of degree d. Thus

$$\sum_{d|r} dN(d) = \deg(\mathcal{P}_r) = 2^r .$$

By Möbius inversion we see that

$$rN(r) = \sum_{d|r} \mu(d) 2^{r/d} .$$

Counting irreducible polynomials

Thus, the number of irreducible polynomials of degree r is

$$N(r) = \frac{2^r}{r} + O\left(\frac{2^{r/2}}{r}\right).$$

Since there are 2^r polynomials of degree r, the probability that a randomly selected polynomial is irreducible is $\sim 1/r \to 0$ as $r \to +\infty$. In this sense, *almost all* polynomials over (fixed) finite fields are reducible (just as almost all integers are composite).

Analogy. Polynomials of degree r are analogous to prime numbers of r *digits*. By the prime number theorem, the number of r-digit primes in base b is about

$$\int_{b^{r-1}}^{b^r} \frac{dt}{\ln t} = \left(\frac{b^r - b^{r-1}}{r \ln b}\right) \left(1 + O\left(\frac{1}{r}\right)\right) .$$

The Riemann Hypothesis implies an error term $O(rb^{r/2})$ as $r \to +\infty$ for the integral on the left [von Koch].

Irreducible and primitive polynomials

Irreducible polynomials over finite fields are useful in several applications. As one example, observe that, if $P(x)$ is an irreducible polynomial of degree r over GF(2), then $\mathrm{GF}(2)[x]/P(x) \cong \mathrm{GF}(2^r)$. In other words, the ring of polynomials mod $P(x)$ gives a representation of the finite field with 2^r elements.

If, in addition, x is a generator of the multiplicative group, that is if every nonzero element of $\mathrm{GF}(2)[x]/P(x)$ can be represented as a power of x, then $P(x)$ is said to be *primitive*.

Primitive polynomials can be used to obtain linear feedback shift registers (LFSRs) with maximal period.

In general, to test primitivity, we need to know the prime factorization of $2^r - 1$.

The number of primitive polynomials of degree r over GF(2) is

$$\frac{\phi(2^r-1)}{r} \le N(r)\,,$$

with equality when $2^r - 1$ is prime.

Fermat and Mersenne primes

A *Fermat prime* is a prime of the form $2^n + 1$. There are *conjectured* to be only finitely many. For $n < 2^{33}$ the only examples are $3, 5, 17, 257, 65537$. Note that n is necessarily a power of 2, because if $n = pq$ with $p > 1$ odd, then $2^q + 1$ is a nontrivial divisor of $2^n + 1$. The converse is false, as shown by Euler:

$$2^{32} + 1 = 641 \times 6700417\,.$$

A *Mersenne prime* is a prime of the form $2^n - 1$, for example $3, 7, 31, 127, 8191, \ldots$ There are *conjectured* to be infinitely many Mersenne primes. The number for $n \le N$ is conjectured to be of order $\log N$.

The GIMPS project is searching systematically for Mersenne primes. So far 46 Mersenne primes are known, the largest being

$$2^{43112609} - 1\,.$$

If $2^n - 1$ is prime we say that n is a *Mersenne exponent.* A Mersenne exponent is necessarily prime, but not conversely ($2^{11} - 1 = 23 \times 89$).

Part 1: Testing irreducibility

Since irreducible polynomials are "rare" but useful, we are interested in algorithms for testing irreducibility.

From [Gauss], $P(x)$ of degree $r > 1$ is irreducible iff

$$x^{2^r} = x \bmod P(x)$$

and, for all prime divisors d of r, we have

$$\mathrm{GCD}\left(x^{2^{r/d}} - x, P(x)\right) = 1\,.$$

The second condition is required to rule out the possibility that $P(x)$ is a product of irreducible factors of some degree(s) $k = r/d$, $d|r$. This condition does not significantly change anything, so let us assume that r is prime. (In our examples r is a Mersenne exponent, so necessarily prime.) Then $P(x)$ is irreducible iff

$$x^{2^r} = x \bmod P(x).$$

One more assumption

All the algorithms involve computations mod $P(x)$, that is, in the ring $\mathrm{GF}(2)[x]/P(x)$.

In the complexity analysis we assume that $P(x)$ is *sparse*, that is, the number of nonzero coefficients is small. Thus, reduction of a polynomial mod $P(x)$ can be done in linear time. The algorithms to be discussed still work without this assumption, but the complexity analysis no longer applies because more time is spent in the reductions mod $P(x)$.

In applications $P(x)$ is often a *trinomial*

$$P(x) = x^r + x^s + 1\,, \quad r > s > 0\,.$$

Irreducible and primitive trinomials

There is no known formula for the number of irreducible or primitive *trinomials* of degree r over GF(2) (unlike the case of general polynomials).

Since $N(r) \approx 2^r/r$, the probability that a randomly chosen polynomial of degree r will be irreducible is about $1/r$. It is *plausible* to assume that the same applies to trinomials. There are $r - 1$ trinomials of degree r, so we might expect $O(1)$ of them to be irreducible. More precisely, we might expect a Poisson distribution with some constant mean μ.

This plausible argument is *false*, as shown by Swan's theorem. We state a [corrected] version of Swan's Corollary 5 that is relevant to trinomials.

Historical note: Swan (1962) rediscovered results of Pellet (1878) and Stickelberger (1897), so the name of the theorem depends on your nationality.

Swan's theorem (Corollary 5)

Theorem 1 *Let $r > s > 0$, and assume $r + s$ is odd. Then $T_{r,s}(x) = x^r + x^s + 1$ has an even number of irreducible factors over* GF(2) *in the following cases:*
a) r even, $r \neq 2s$, $rs/2 = 0$ or 1 mod 4.
b) r odd, s not a divisor of $2r$, $r = \pm 3$ mod 8.
c) r odd, s divisor of $2r$, $r = \pm 1$ mod 8.
In all other cases $x^r + x^s + 1$ has an odd number of irreducible factors.

If both r and s are even, then $T_{r,s}(x)$ is a square. If both r and s are odd, apply the theorem to $T_{r,r-s}(x)$.

For r an odd prime, ignoring the easily-checked cases $s = 2$ or $r - s = 2$,
case (b) says that the trinomial has an *even* number of irreducible factors, and hence must be *reducible*, if $r = \pm 3 \bmod 8$.

For prime $r = \pm 1 \bmod 8$, the heuristic Poisson distribution does seem to apply, with mean $\mu \approx 3$. Similarly for primitive trinomials, with a correction factor $\phi(2^r - 1)/(2^r - 2)$.

First algorithm — repeated squaring

Our first and simplest algorithm for testing irreducibility is just *repeated squaring*:

> $Q(x) \leftarrow x$;
> for $j \leftarrow 1$ to r do
> $\quad Q(x) \leftarrow Q(x)^2 \bmod P(x)$;
> if $Q(x) = x$ then
> $\quad$ return **irreducible**
> else
> $\quad$ return **reducible**.

The operation $Q(x) \leftarrow Q(x)^2 \bmod P(x)$ can be performed in time $O(r)$. The constant factor is small. We recommend the fast squaring algorithm of Brent, Larvala and Zimmermann (2003). This saves both operations and memory references, and is about 2.2 times faster than the obvious squaring algorithm (as implemented in most otherwise-good software packages).

Since the irreducibility test involves r squarings, the overall time is $O(r^2)$.

Polynomial multiplication

Before describing other algorithms for irreducibility testing, we digress to discuss polynomial multiplication, matrix multiplication, and modular composition.

To multiply two polynomials $A(x)$ and $B(x)$ of degree (at most) r, the "classical" algorithm takes time $O(r^2)$. There are faster algorithms, e.g. Karatsuba, Toom-Cook, and FFT-based algorithms.

For polynomials over GF(2), the asymptotically fastest known algorithm is due to Schönhage. (The Schönhage-Strassen algorithm does not work in characteristic 2, and it is not clear whether Fürer's ideas are useful here.)

Schönhage's algorithm runs in time

$$M(r) = O(r \log r \log \log r) .$$

In practice, for $r \approx 32\,000\,000$, a multiplication takes about 480 times as long as a squaring.

Matrix multiplication

Let ω be the exponent of matrix multiplication, so we can multiply $n \times n$ matrices in time $O(n^{\omega+\varepsilon})$ for any $\varepsilon > 0$. The best result is Coppersmith and Winograd's $\omega < 2.376$, though in practice we would use the classical ($\omega = 3$) or Strassen ($\omega = \log_2 7 \approx 2.807$) algorithm.

Since we are working over GF(2), our matrices have single-bit entries. This means that the classical algorithm can be implemented very efficiently using full-word operations (32 or 64 bits at a time). Nevertheless, Strassen's algorithm is faster if n is larger than about 1000.

Good in practice is the "Four Russians" algorithm [Arlazarov, Dinic, Kronod & Faradzev, 1970]. It computes $n \times n$ Boolean matrix multiplication in time $O(n^3/\log n)$.

We can use the Four Russians' algorithm up to some threshold, say $n = 1024$, and Strassen's recursion for larger n, combining the advantages of both.

Modular composition

The *modular composition* problem is: given polynomials $A(x)$, $B(x)$, $P(x)$, compute

$$C(x) = A(B(x)) \bmod P(x).$$

If $\max(\deg(A), \deg(B)) < r = \deg(P)$, then we could compute $A(B(x))$, a polynomial of degree at most $(r-1)^2$, and reduce it modulo $P(x)$. However, this wastes both time and space.

Better is to compute

$$C(x) = \sum_{j \le \deg(A)} a_j (B(x))^j \bmod P(x)$$

by Horner's rule, reducing mod $P(x)$ as we go, in time $O(rM(r))$ and space $O(r)$. Using Schönhage's algorithm for the polynomial multiplications, we can compute $C(x)$ in time $O(r^2 \log r \log\log r)$.

Faster modular composition

Using an algorithm of Brent & Kung (1978), based on an idea of Paterson and Stockmeyer, we can reduce the modular composition problem to a problem of matrix multiplication. If the degrees of the polynomials are at most r, and $m = \lceil r^{1/2} \rceil$, then we have to perform m multiplications of $m \times m$ matrices. The matrices are over the same field as the polynomials (that is, GF(2) here).

The Brent-Kung modular composition algorithm takes time

$$O(r^{(\omega+1)/2}) + O(r^{1/2}M(r)),$$

where the first term is for the matrix multiplications and the second term is for computing the relevant matrices.

Assuming Strassen's matrix multiplication, the first term is $O(r^{1.904})$ and the second term is $O(r^{1.5} \log r \log\log r)$. Thus, the second term is asymptotically negligible (but maybe not in practice).

Using modular composition

Recall that our problem is to compute $x^{2^r} \bmod P(x)$. Repeated squaring is not the only way to do this.

Let $A_k(x) = x^{2^k} \bmod P(x)$. Then a modular composition algorithm can be used to compute $A_k(A_m(x)) \bmod P(x)$. Since

$$A_k(A_m(x)) = \left(x^{2^m}\right)^{2^k} \bmod P(x) = A_{m+k}(x),$$

we can compute $x^{2^r} \bmod P(x)$ with about $\log_2(r)$ modular compositions instead of r squarings.

For example, if $r = 17$, we have (all computations in $\mathrm{GF}(2)[x]/P(x)$):

$$\begin{array}{ll}
A_1(x) = x^2, & \text{(trivial)} \\
A_2(x) = A_1(A_1(x)) = x^4, & (\equiv 1 \text{ squaring}) \\
A_4(x) = A_2(A_2(x)) = x^{16}, & (\equiv 2 \text{ squarings}) \\
A_8(x) = A_4(A_4(x)) = x^{256}, & (\equiv 4 \text{ squarings}) \\
A_{16}(x) = A_8(A_8(x)) = x^{2^{16}}, & (\equiv 8 \text{ squarings}) \\
A_{17}(x) = A_{16}(x)^2 = x^{2^{17}}, & \text{(1 squaring)}
\end{array}$$

using only 4 modular composition steps.

Second algorithm

To summarise, we can compute $A_r(x) = x^{2^r} \bmod P(x)$ by the following recursive algorithm that uses the binary representation of r (*not* that of 2^r):

> if $r = 0$ then
>   return x
> else if r even then
> $\{U(x) \leftarrow A_{r/2}(x);$
>   return $U(U(x)) \bmod P(x)\}$
> else
>   return $A_{r-1}(x)^2 \bmod P(x)$.

The algorithm takes about $\log_2(r)$ modular compositions. Hence, if Strassen's algorithm is used in the Brent-Kung modular composition algorithm, we can test irreducibility in time $O(r^{1.904} \log r)$.

Third algorithm

Recently, Kedlaya and Umans (2008) proposed an asymptotically fast modular composition algorithm that runs in time $O_\varepsilon(r^{1+\varepsilon})$ for any $\varepsilon > 0$.

The algorithm is complicated, involving iterated reductions to multipoint multivariate polynomial evaluation, multidimensional FFTs, and the Chinese remainder theorem.
See the papers on Umans's web site
`www.cs.caltech.edu/~umans/research.htm`

Using the Kedlaya-Umans fast modular composition instead of the Brent-Kung reduction to matrix multiplication,
we can test irreducibility in time $O_\varepsilon(r^{1+\varepsilon})$.

Warning: the "$O_\varepsilon(\cdots)$" notation indicates that the implicit constant depends on ε. In this case, it is a rather large and rapidly increasing (probably exponential) function of $1/\varepsilon$.

Comparison of the algorithms

> So the last shall be first,
> and the first last
>
> Matthew 20:16

The theoretical time bounds predict that the third algorithm should be the fastest, and the first algorithm the slowest. However, this is only for *sufficiently large* degrees r.

In practice, for r up to at least 4.3×10^7, the situation is reversed! The first algorithm is the fastest, and the third algorithm is the slowest.

A minor drawback of the first (squaring) algorithm is that it is hard to speed up on a parallel machine. The other algorithms are much easier to parallelise. However, this is not so relevant when we are considering many trinomials, as we can let different processors of a parallel machine work on different trinomials in parallel.

Example, $r = 32\,582\,657$

Following are actual or estimated times on a 2.2 Ghz AMD Opteron 275 for $r = 32\,582\,657$ (a Mersenne exponent).

1. Squaring (actual): 64 hours
2. Brent-Kung (estimates):
   - classical: 265 hours (19% mm)
   - Strassen: 254 hours (15% mm)
   - Four Russians: 239 hours (10% mm) (plus Strassen for $n > 1024$)
3. Kedlaya-Umans (estimate): $> 10^{10}$ years

The Brent-Kung algorithm would be the fastest if the matrix multiplication were dominant; unfortunately the $O(r^{1/2}M(r))$ overhead term dominates.
Since the overhead scales roughly as $r^{1.5}$, we estimate that the Brent-Kung algorithm would be faster than the squaring algorithm for $r > 7 \times 10^8$ (approximately).

Note on Kedlaya-Umans

Éric Schost writes:

> The Kedlaya-Umans algorithm reduces modular composition to the multipoint evaluation of a multivariate polynomial, assuming the base field is large enough.
>
> The input of the evaluation is over F_p; the algorithm works over $\mathbb{Z}$ and reduces mod p in the end. The evaluation over $\mathbb{Z}$ is done by CRT modulo a bunch of smaller primes, and so on. At the end-point of the recursion, we do a naive evaluation on all of F_{p^m}, where p is the current prime and m the number of variables. So the cost here is $\geq p^m$.
>
> [Now he considers choices of m in the case $r = 32\,582\,657$; all give $p^m \geq 1.36 \times 10^{27}$.]

Our estimate of $> 10^{10}$ years is based on a time of 1 nsec per evaluation (very optimistic).

The "best" algorithm

Comparing the second algorithm with the first, observe that the modular compositions do not all save equal numbers of squarings. In fact the *last* modular composition saves $\lfloor r/2 \rfloor$ squarings, the *second-last* saves $\lfloor r/4 \rfloor$ squarings, etc.

Each modular composition has the same cost. Thus, if we can use only one modular composition, *it should be the one that saves the most squarings.*

If we use $\lfloor r/2 \rfloor$ squarings to compute $x^{2^{\lfloor r/2 \rfloor}} \bmod P(x)$, then use one modular composition (and one further squaring, if r is odd), we can compute $x^{2^r} \bmod P(x)$ faster than with any of the algorithms considered so far, provided r exceeds a certain threshold.

In the example, the time would be reduced from 64 hours to 44 hours, a saving of 31%.

Doing two modular compositions would reduce the time to 40 hours, a saving of 37%.

Computational results

In 2007-8 Paul Zimmermann and I conducted a search for irreducible trinomials $x^r + x^s + 1$ whose degree r is a (known) Mersenne exponent. Since $2^r - 1$ is prime, *irreducible* implies *primitive.* The previous record degree of a primitive trinomial was $r = 6\,972\,593$.

r	s
24 036 583	8 412 642, 8 785 528
25 964 951	880 890, 4 627 670, 4 830 131, 6 383 880
30 402 457	2 162 059
32 582 657	5 110 722, 5 552 421, 7 545 455

Table 1: Ten new primitive trinomials $x^r + x^s + 1$ of degree a Mersenne exponent, for $s \leq r/2$.

We used the first algorithm to test irreducibility of the most difficult cases. Most of the time was spent discarding the vast majority of trinomials that have a small factor, using a new factoring algorithm with good average-case behaviour (the topic of the second half of this talk).

Part 2: Factoring

The problem of factoring a univariate polynomial $P(x)$ over a finite field F often arises in computational algebra. An important case is when F has small characteristic and $P(x)$ has high degree but is *sparse* (has only a small number of nonzero terms).

Since time is limited, I will make the same assumptions as in Part 1: $F = \mathrm{GF}(2)$ and $P(x)$ is sparse, typically a trinomial

$$P(x) = x^r + x^s + 1, \quad r > s > 0,$$

although the ideas apply more generally.

The aim is to give an algorithm with good *amortized complexity*, that is, one that works well *on average.* Since we are restricting attention to trinomials, we average over all trinomials of fixed degree r.

Equivalently, we can use probabilistic language, and assume a uniform distribution over all trinomials of fixed degree r.

Distinct degree factorization

I will only consider *distinct degree factorization*. That is, if $P(x)$ has several factors of the same degree d, the algorithm will produce the product of these factors. The Cantor-Zassenhaus algorithm can be used to split this product into distinct factors. This is usually cheap because in most cases the product has small degree or consists of just one factor.

Factor of smallest degree

To simplify the complexity analysis and speed up the algorithm in the common application of searching for irreducible polynomials, I only consider the time required to find *one* nontrivial factor (it will be a factor of smallest degree) or output "irreducible".

Certificates of reducibility

A nontrivial factor (preferably of smallest degree) gives a "reducibility certificate" that can quickly be checked.

Factorization in $\mathrm{GF}(2)[x]$

From now on we write "+" instead of "−" (they are equivalent in $\mathrm{GF}(2)[x]$).

As we already mentioned, $x^{2^d} + x$ is the product of all irreducible polynomials of degree dividing d. For example,

$$x^{2^3} + x = x(x+1)(x^3+x+1)(x^3+x^2+1) \ .$$

Thus, a simple (but slow) algorithm to find a factor of smallest degree of $P(x)$ is to compute $\mathrm{GCD}(x^{2^d} + x, P(x))$ for $d = 1, 2, \ldots$. The first time that the GCD is nontrivial, it contains a factor of minimal degree d. If the GCD has degree $> d$, it must be a product of factors of degree d.

If no factor has been found for $d \le r/2$, where $r = \deg(P(x))$, then $P(x)$ must be irreducible.

Note that x^{2^d} should not be computed explicitly; instead compute $x^{2^d} \bmod P(x)$ by repeated squaring.

Application to trinomials

Some simplifications are possible when $P(x) = x^r + x^s + 1$ is a trinomial.

- We can skip the case $d = 1$ because a trinomial can not have a factor of degree 1.
- Since $x^r P(1/x) = x^r + x^{r-s} + 1$, we only need consider $s \le r/2$.
- By applying Swan's theorem, we can usually show that the trinomial under consideration has an odd number of factors; in this case we only need check $d \le r/3$.

Complexity of squares and muls

In Part 1 we already considered the complexity of computing squares and products in $\mathrm{GF}(2)[x]/P(x)$. Recall that, with our usual assumption that $P(x)$ is sparse, squaring can be performed in time

$$S(r) = \Theta(r) \ll M(r)$$

and multiplication can be performed in time

$$M(r) = O(r \log r \log\log r) \ .$$

In the complexity estimates we assume that $M(r)$ is a sufficiently smooth and well-behaved function.

Complexity of GCD

For GCDs we use a sub-quadratic algorithm that runs in time $G(r) = O(M(r) \log r)$.

More precisely,

$$G(2r) = 2G(r) + O(M(r)) \; ,$$

so

$$M(r) = O(r \log r \log \log r) \Rightarrow G(r) = \Theta(M(r) \log r) \; .$$

In practice, for $r \approx 2.4 \times 10^7$ and our implementation on a 2.2 Ghz Opteron,

$$S(r) \approx 0.005 \; \text{seconds},$$

$$M(r) \approx 2 \; \text{seconds},$$

$$G(r) \approx 80 \; \text{seconds},$$

$$M(r)/S(r) \approx 400 \; ,$$

$$G(r)/M(r) \approx 40 \; .$$

Avoiding GCD computations

In the context of integer factorization, Pollard (1975) suggested a blocking strategy to avoid most GCD computations and thus reduce the amortized cost; von zur Gathen and Shoup (1992) applied the same idea to polynomial factorization.

The idea of blocking is to choose a parameter $\ell > 0$ and, instead of computing

$$\mathrm{GCD}(x^{2^d} + x, P(x)) \;\; \text{for} \;\; d \in [d', d' + \ell) \; ,$$

compute

$$\mathrm{GCD}(p_\ell(x^{2^{d'}}, x), P(x)) \; ,$$

where the *interval polynomial* $p_\ell(X, x)$ is defined by

$$p_\ell(X, x) = \prod_{j=0}^{\ell-1} \left(X^{2^j} + x \right) \; .$$

In this way we replace ℓ GCDs by one GCD and $\ell - 1$ multiplications mod $P(x)$.

Backtracking

The drawback of blocking is that we may have to backtrack if $P(x)$ has more than one factor with degrees in $[d', d' + \ell)$, so ℓ should not be too large. The optimal strategy depends on the expected size distribution of factors and the ratio of times for GCDs and multiplications.

New idea - multi-level blocking

We introduce a finer level of blocking to replace most multiplications by squarings, which speeds up the computation in $\mathrm{GF}(2)[x]/P(x)$ of the interval polynomials $p_m(x^{2^d}, x)$, where

$$p_m(X, x) = \prod_{j=0}^{m-1} \left(X^{2^j} + x \right) = \sum_{j=0}^{m} x^{m-j} s_{j,m}(X) \; ,$$

$$s_{j,m}(X) = \sum_{0 \le k < 2^m, \; w(k) = j} X^k \; ,$$

and $w(k)$ denotes the Hamming weight of k.

Note that $s_{j,m}(X^2) = s_{j,m}(X)^2$ in $\mathrm{GF}(2)[x]/P(x)$. Thus, $p_m(x^{2^d}, x)$ can be computed with cost $m^2 S(r)$ if we already know $s_{j,m}(x^{2^{d-m}})$ for $0 < j \le m$.

In this way we replace m multiplications and m squarings by one multiplication and m^2 squarings. Choosing $m \approx \sqrt{M(r)/S(r)}$ (about 20 if $M(r)/S(r) \approx 400$), the speedup over single-level blocking is about $m/2 \approx 10$.

Fast initialization

The polynomials

$$s_{j,m}(x) = \sum_{0 \le k < 2^m,\, w(k)=j} x^k$$

satisfy a "Pascal triangle" recurrence relation

$$s_{j,m}(x) = s_{j,m-1}(x^2) + x \times s_{j-1,m-1}(x^2)$$

with boundary conditions

$$s_{j,m}(x) = 0 \;\text{ if }\; j > m\ ,$$

$$s_{0,m}(x) = 1\ .$$

Thus, we can compute

$$\{s_{j,m}(x) \bmod P(x) \mid 0 \le j \le m\}$$

in time $O(m^2 r)$, even though the definition of $s_{j,m}(x)$ involves $O(2^m)$ terms.

Question: Have the polynomials $s_{j,m}(x)$ been studied before? It seems probable but I have not found any references to them.

Recapitulation

To summarize, we use two levels of blocking:

- The outer level replaces most GCDs by multiplications.
- The inner level replaces most multiplications by squarings.
- The blocking parameter $m \approx \sqrt{M(r)/S(r)}$ is used for the inner level of blocking.
- A different parameter $\ell = km$ is used for the outer level of blocking.

Example

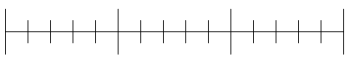

Figure 1: $\ell = 15$, $m = 5$

In the example, $S = 1/25$, $M = 1$, $G = 10$

No blocking: cost $15G + 15S = 150.6$

1-level blocking: $G + 14M + 15S = 24.6$

2-level blocking: $G + 2M + 75S = 15.0$

More realistically, suppose $\ell = 80$, $m = 20$, $S = 1/400$, $M = 1$, $G = 40$

No blocking: cost $80G + 80S = 3200.2$

1-level blocking: $G + 79M + 80S = 119.2$

2-level blocking: $G + 3M + 1600S = 47.0$

Sieving

A *small* factor is one with degree $d < \frac{1}{2}\log_2 r$, so $2^d < \sqrt{r}$.

It would be inefficient to find small factors in the same way as large factors. Instead, let $d' = 2^d - 1$, $r' = r \bmod d'$, $s' = s \bmod d'$. Then

$$P(x) = x^r + x^s + 1 = x^{r'} + x^{s'} + 1 \bmod (x^{d'} - 1)\ ,$$

so we only need compute

$$\mathrm{GCD}(x^{r'} + x^{s'} + 1, x^{d'} - 1)\ .$$

The cost of finding small factors is negligible (both theoretically and in practice), so will be ignored.

In the definition, the fraction $\frac{1}{2}$ is rather arbitrary; it can be replaced by $1 - \varepsilon$ for any $\varepsilon > 0$.

Distribution of degrees of factors

In order to predict the expected behaviour of our algorithm, we need to know the expected distribution of degrees of irreducible factors. Our complexity estimates here are based on the assumption that trinomials of degree r behave like the set of all polynomials of the same degree, up to a constant factor:

Assumption 1 *Over all trinomials $x^r + x^s + 1$ of degree r over* GF(2)*, the probability π_d that a trinomial has no nontrivial factor of degree $\le d$ is at most c/d, where c is a constant and $1 < d \le r$.*

This assumption is plausible and in agreement with experiments, though not proven. Under the assumption, we use an amortized model to obtain the total complexity over all trinomials of degree r.

From Assumption 1, the probability that a trinomial does not have a small factor is $O(1/\log r)$.

Simpler approximation

Let $p_d = \pi_{d-1} - \pi_d$ be the probability that the smallest nontrivial factor of a randomly chosen trinomial has degree $d \ge 2$. Although not strictly correct, the following is a good approximation.

Assumption 2 *p_d is of order $1/d^2$, provided d is not too large.*

I will use Assumption 2 because it simplifies the amortized complexity analysis, but the same results can be obtained from Assumption 1 using summation by parts.

Some empirical evidence for Assumptions 1–2 in the case $r = 6\,972\,593$ is given in Table 2 (next slide). Results for other large Mersenne exponents are similar.

Table 2: Statistics for $r = 6\,972\,593$

d	$d\pi_d$	$d^2 p_d$
2	1.33	1.33
3	1.43	1.71
4	1.52	1.52
5	1.54	1.84
6	1.60	1.47
7	1.60	1.85
8	1.67	1.29
9	1.64	2.10
10	1.65	1.73
100	1.77	
1000	1.76	
10000	1.88	
100000	1.62	
226887	2.08	
$r-1$	2.00	

Evidence for Assumptions 1–2.

Analogies

The following have similar distributions in the limit as $n \to \infty$:

1. Degree of smallest irreducible factor of a random monic polynomial of degree n over a finite field (say GF(2)).
2. Size of smallest cycle in a random permutation of n objects.
3. Size (in base-b digits) of smallest prime factor in a random integer of n digits.

More precisely, let P_d be the limiting probability that the smallest irreducible factor has degree $> d$, that the smallest cycle has length $> d$, or that the smallest prime factor has $> d$ digits, in cases 1–3 respectively. Then

$$P_d \sim c/d \ \text{ as } \ d \to \infty$$

(the constant c is different in each case).
For example, in case 3, let $x = b^d$; then

$$P_d = \prod_{\text{prime } p<x} \left(1 - \frac{1}{p}\right) \sim \frac{e^{-\gamma}}{\ln x} = \left(\frac{e^{-\gamma}}{\ln b}\right)\frac{1}{d}$$

by the theorem of Mertens.

Outer level blocking strategy

The blocksize in the outer level of blocking is $\ell = km$. We take an increasing sequence

$$k = k_0 j \text{ for } j = 1, 2, 3, \ldots ,$$

where $k_0 m$ is of order $\log r$ (since small factors will have been found by sieving). This leads to a quadratic polynomial for the interval bounds.

There is nothing magic about a quadratic polynomial, but it is simple to implement and experiments show that it is reasonably close to optimal.

Using the data that we have obtained on the distribution of degrees of smallest factors of trinomials, and assuming that this distribution is insensitive to the degree r, we could obtain a strategy that is close to optimal. However, the choice $k_0 j$ with suitable k_0 is simple and not too far from optimal. The number of GCD and sqr/mul operations is usually within a factor of 1.5 of the minimum possible.

Expected cost of sqr/mul

Recall that the inner level of blocking replaces m multiplications by m^2 squarings and one multiplication, where $m \approx \sqrt{M(r)/S(r)}$ makes the cost of squarings about equal to the cost of multiplications.

For a smallest factor of degree d, the expected number of squarings is $m(d + O(\sqrt{d}))$. Averaging over all trinomials of degree r, the expected number is

$$O\left(m \sum_{d \le r/2} \frac{d + O(\sqrt{d})}{d^2}\right) = O\left(m \log r\right) .$$

Thus, the expected cost of sqr/mul operations per trinomial is

$$\begin{aligned} & O\left(S(r) \log r \sqrt{M(r)/S(r)}\right) \\ = & O\left(\log r \sqrt{M(r)S(r)}\right) \\ = & O\left(r (\log r)^{3/2} (\log\log r)^{1/2}\right) . \end{aligned}$$

Expected cost of GCDs

Suppose that $P(x)$ has smallest factor of degree d. The number of GCDs required to find the factor, using our (quadratic polynomial) blocking strategy, is $O(\sqrt{d})$. By Assumption 2, the expected number of GCDs for a trinomial with no small factor is

$$1 + O\left(\sum_{(\lg r)/2 < d \le r/2} \frac{\sqrt{d}}{d^2}\right) = 1 + O\left(\frac{1}{\sqrt{\log r}}\right) .$$

Thus the expected cost of GCDs per trinomial is

$$O(G(r)/\log r) = O(M(r)) = O(r \log r \log\log r) .$$

This is asymptotically $\ll$ expected cost of sqr/mul operations

In practice, for $r \approx 4.3 \times 10^7$, GCDs take about 65% of the time versus 35% for sqr/mul. Once again, the asymptotic analysis is misleading, because the function

$$\sqrt{\frac{\log r}{\log\log r}}$$

is a very slowly growing function of r.

Comparison with classical algorithms

For simplicity I will use the $\widetilde{O}$ notation which ignores log factors.

The "classical" algorithm takes an expected time $\widetilde{O}(r^2)$ per trinomial, or $\widetilde{O}(r^3)$ to cover all trinomials of degree r.

The new algorithm takes expected time $\widetilde{O}(r)$ per trinomial, or $\widetilde{O}(r^2)$ to cover all trinomials of degree r.

In practice, the new algorithm is faster by a factor of about 160 for $r = 6\,972\,593$, and by a factor of about 1000 for $r = 43\,112\,609$.

Thus, comparing the computation for $r = 43\,112\,609$ with that for $r = 6\,972\,593$: using the classical algorithm would take about 240 times longer (impractical), but using the new algorithm saves a factor of 1000.

Generally, our search for different Mersenne exponents $r \in \{6\,972\,593, 24\,036\,583, 25\,964\,951, 30\,402\,457, 32\,582\,657, 43\,112\,609\}$ took less time for larger r, due to incremental improvements in the search program!

Recent computational results

Since Sept 2008 we have been searching for primitive trinomials of degree 43 112 609 (the largest known Mersenne exponent).

Dan Bernstein and Tanja Lange have joined in the search and contributed CPU cycles.

So far we completed about 98% of the search and found four new primitive trinomials $x^r + x^s + 1$, $r = 43\,112\,609$:

$$s = 3\,569\,337,\ 4\,463\,337,\ 17\,212\,521,\ 21\,078\,848$$

Testing irreducibility took about 119 hours per trinomial on a 2.2 Ghz AMD Opteron, using our first algorithm. The "best" algorithm would take about 69 hours (saving 42%).

Most of the time (about 22 processor-years) was spent eliminating reducible trinomials at an average rate of about 32 sec per trinomial ($\times 43112609/2$ trinomials).

Conclusion

The new double-blocking strategy works well and, combined with fast multiplication and GCD algorithms, has allowed us to find new primitive trinomials of record degree. This would have been impractical using the classical algorithms.

The same ideas work over finite fields GF(p) for small prime $p > 2$, and for factoring sparse polynomials $P(x)$ that are not necessarily trinomials: all we need is that the time for p-th powers (mod $P(x)$) is much less than the time for multiplication (mod $P(x)$).

Acknowledgement

Thanks to Éric Schost for his comments on the work of Kedlaya and Umans; to Dan Bernstein and Tanja Lange for contributing computer time to the search for degree 43 112 609; to Alan Steel for independently verifying our new primitive trinomials using Magma; and to Victor Shoup for his package NTL, which was used to debug our software and check our certificates.

References

[1] V. L. Arlazarov, E. A. Dinic, M. A. Kronod & I. A. Faradzev, On economical construction of the transitive closure of an oriented graph, *Soviet Math. Dokl.* **11** (1975), 1209–1210.

[2] W. Bosma & J. Cannon, *Handbook of Magma Functions*, School of Mathematics and Statistics, University of Sydney, 1995. `http://magma.maths.usyd.edu.au/`

[3] R. P. Brent, P. Gaudry, E. Thomé & P. Zimmermann, Faster multiplication in $GF(2)[x]$, *Proc. ANTS VIII 2008*, *Lecture Notes in Computer Science* **5011**, 153–166. `http://wwwmaths.anu.edu.au/~brent/pub/pub232.html`

[4] R. P. Brent & H. T. Kung, Fast algorithms for manipulating formal power series, *J. ACM* **25** (1978), 581–595. `.../pub045.html`

[5] R. P. Brent, S. Larvala & P. Zimmermann, A fast algorithm for testing reducibility of trinomials mod 2 and some new primitive trinomials of degree 3021377, *Math. Comp.* **72** (2003), 1443–1452. `.../pub199.html`

[6] R. P. Brent & P. Zimmermann, A multi-level blocking distinct-degree factorization algorithm, *Finite Fields and Applications: Contemporary Mathematics* **461** (2008), 47–58. `.../pub230.html`

[7] R. P. Brent & P. Zimmermann, Ten new primitive binary trinomials, *Math. Comp.* **78** (2009), 1197–1199. `.../pub233.html`

[8] D. G. Cantor and H. Zassenhaus, A new algorithm for factoring polynomials over finite fields, *Math. Comp.* **36** (1981), 587–592.

[9] D. Coppersmith & W. Winograd, Matrix multiplication via arithmetic progressions, *J. Symb. Comput.* **9** (1980), 251–280.

[10] M. Fürer, Faster integer multiplication, *Proc. 48th STOC Conference*, 2007, 57–66.

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

[12] J. von zur Gathen and J. Gerhard, Polynomial factorization over F_2, *Math. Comp.* **71** (2002), 1677–1698.

[13] J. von zur Gathen and V. Shoup, Computing Frobenius maps and factoring polynomials, *Computational Complexity* **2** (1992), 187–224. `http://www.shoup.net/papers/`

[14] K. Kedlaya & C. Umans, Fast modular composition in any characteristic, *Proc. FOCS 2008*, 146–155. `http://www.cs.caltech.edu/~umans/research.htm`

[15] T. Kumada, H. Leeb, Y. Kurita and M. Matsumoto, New primitive t-nomials ($t = 3, 5$) over GF(2) whose degree is a Mersenne exponent, *Math. Comp.* **69** (2000), 811–814. Corrigenda: *ibid* **71** (2002), 1337–1338.

[16] A.-E. Pellet, Sur la décomposition d'une fonction entière en facteurs irréductibles suivant un module premier p, *Comptes Rendus de l'Académie des Sciences Paris* **86** (1878), 1071–1072.

[17] J. M. Pollard. A Monte Carlo method for factorization, *BIT* **15** (1975), 331–334,

[18] A. Schönhage, Schnelle Multiplikation von Polynomen über Körpern der Charakteristik 2, *Acta Inf.* **7** (1977), 395–398.

[19] É. Schost, *Fast irreducibility test*, personal communication, 4 June 2008.

[20] V. Shoup, NTL: A library for doing number theory. `http:www.shoup.net/ntl/`

[21] L. Stickelberger, Über eine neue Eigenschaft der Diskriminanten algebraischer Zahlkörper, *Verhandlungen des ersten Internationalen Mathematiker-Kongresses*, Zürich, 1897, 182–193.

[22] R. G. Swan, Factorization of polynomials over finite fields, *Pacific J. Math.* **12** (1962), 1099–1106.

[23] G. Woltman *et al*, GIMPS, The Great Internet Mersenne Prime Search. `http://www.mersenne.org/`