Factoring Polynomials over Finite Fields: Asymptotic Complexity vs. Reality*

Victor Shoup
Dept. of Computer Science
University of Toronto
Toronto, Canada M5S 1A4

Abstract

Several algorithms for factoring polynomials over finite fields are compared from the point of view of asymptotic complexity, and from a more realistic point of view: how well actual implementations perform on "moderately" sized inputs.

1 Introduction

The purpose of this paper is to examine several algorithms for factoring polynomials over finite fields, from both the point of view of asymptotic complexity, and from a more realistic point of view: how well actual implementations perform on "moderately" sized inputs.

We restrict our attention to factoring in $\mathbf{Z}_p[x]$, where p is prime. The algorithms we consider are the algorithms of Berlekamp [B], Cantor & Zassenhaus [CZ], and von zur Gathen & Shoup [GS].

2 Asymptotic Complexity

Let n be the degree of the polynomial $f \in \mathbf{Z}_p[x]$ to be factored. It is natural to measure the running times of factorization algorithms in terms of the number of operations in $\mathbf{Z}_p$ (additions, subtractions, multiplications, divisions, and zero-tests). All of the algorithms we consider are probabilistic, and so the operation count is a random variable—it is the expected value of this random variable that we are most interested in.

Berlekamp's algorithm can be implemented so as to use an expected number of

$$O(n^\omega + n(\log n \ \mathrm{loglog}\, n) \log p)$$

operations in $\mathbf{Z}_p$. The constant ω represents the exponent of some matrix multiplication algorithm; that is, we assume that two $k \times k$ matrices can be multiplied using $O(k^\omega)$ arithmetic operations, $2 < k \leq 3$. Also, it is assumed that two polynomials of degree k can me multiplied using $O(k(\log k \ \mathrm{loglog}\, k))$ arithmetic operations.

*Appeared in *Proc. IMACS Symposium*, Lille, France, 1993.

The Cantor/Zassenhaus algorithm can be implemented so as to use an expected number of

$$O(n^2(\log n \,\mathrm{loglog}\, n)\,(\log p + \log n))$$

operations in $\mathbf{Z}_p$.

The von zur Gathen/Shoup algorithm can be implemented so as to use an expected number of

$$O(n^2((\log n)^2 \,\mathrm{loglog}\, n) + n(\log n \;\mathrm{loglog}\, n)\; \log p)$$

operations in $\mathbf{Z}_p$.

The operation count for these algorithms depends on both n and p, and so it is a bit difficult to compare the running times directly. If we let n and $\log p$ tend to infinity at the same rate, then Berlekamp's algorithm uses $O(n^\omega)$ operations, Cantor/Zassenhaus uses $O(n^3 \log n \,\mathrm{loglog}\, n)$, and von zur Gathen/Shoup uses $O(n^2(\log n)^2 \,\mathrm{loglog}\, n)$. So in this case, von zur Gathen/Shoup is asymptotically fastest. Moreover, for any choice of n and p, the operation count for von zur Gathen/Shoup is no worse than a constant factor times the operation count of either of the other two algorithms.

Another complexity consideration is space. A natural measure of space is the maximum number of elements of $\mathbf{Z}_p$ that must be stored at any given time.

Berlekamp's algorithm uses space for $O(n^2)$ elements of $\mathbf{Z}_p$. A major advantage of the Cantor/Zassenhaus algorithm is that it uses space for only $O(n)$ elements. The von zur Gathen/Shoup algorithm can be implemented using space for $O(n^{3/2})$ elements.

3 An Implementation Experiment

To assess these algorithms from a more realistic perspective, we discuss the results of an implementation experiment.

All of our code was written in `C++`, is relatively machine independent, and is available from the author via e-mail (`shoup@cs.toronto.edu`).

An important limitation in this experiment is that we restricted the inputs to polynomials of the form $f = f_1 f_2$, where f_1 and f_2 are nonassociate irreducible polynomials of degree $d = n/2$.

We maintain that in the general factoring case, where no information about the factorization of f is known *a priori*, neither the Cantor/Zassenhaus or von zur Gathen/Shoup algorithms can compete with Berlekamp's algorithm on moderately sized inputs (say, $n \leq 1000$) in terms of running time (although space could be an important issue). Making the above restriction makes the competition more sporting and more interesting.

We implemented all three algorithms in light of this restriction, and we did not hesitate to fine tune the algorithms to take advantage of this restriction.

Our test suite of factorization problems consisted of various primes p, with roughly 50, 100, and 200 bits, and polynomials (of the restricted form) of degrees roughly 25, 50, 100, 250, 500, and 1000.

The test data was generated randomly in the following way. Given n and p, irreducible polynomials f_1 and f_2 were chosen uniformly at random from among all irreducibles in $\mathbf{Z}_p[x]$ of degree d. The selection of n and p is not entirely random—they are selected to facilitate the efficient generation of f_1 and f_2; however, no special properties of n and p are exploited by the factoring algorithms under consideration. In particular, for example, it is not assumed that $\mathbf{Z}_p$ supports the FFT.

3.1 Polynomial Arithmetic

Multi-precision integer arithmetic was implemented using code available from Arjen Lenstra.

A critical component in all of these factoring algorithms is multiplication in the ring $\mathbf{Z}_p[x]/(f)$. This was done using a "modular" FFT algorithm. Suppose two polynomials $g, h \in \mathbf{Z}[x]$ are given, with each coefficient in the range $0 \ldots p-1$, and the degree of each polynomial bounded by $n-1$. To compute $g \cdot h \in \mathbf{Z}[x]$, we first compute the product modulo several single-precision primes q such that $\mathbf{Z}_q$ supports the FFT. We then use Chinese remaindering to recover the coefficients of the product over $\mathbf{Z}$.

Once the product is obtained over $\mathbf{Z}$, the coefficients are reduced mod p.

We also need to compute the remainder of degree $2(n-1)$ polynomials modulo f. By pre-conditioning on f, this can be done very efficiently—essentially at the cost of multiplying two polynomials of degree at most $n-1$. Note that without pre-conditioning, the best-known algorithms for division with remainder cost several polynomial multiplications. This pre-conditioning was critical in our implementation.

If the number of machine words required to represent p is w, then multiplications in $\mathbf{Z}_p$ take $O(w^2)$ single-precision machine operations. Although w is related to $\log p$ by a constant factor, to make our "O" estimates of single-precision operation counts more meaningful, we use either w or $\log_2 p$, whichever most accurately reflects the running time—for example, note that $(\log_2 p)^2$ is about 900 times as large as w^2 for the 30-bit radix used in our implementation.

Using the above FFT implementation, the cost of multiplication in $\mathbf{Z}_p[x]/(f)$ is $O(n \log n \, w + n\, w^2)$ single-precision operations.

Polynomial GCD computations were implemented using the classical algorithm, which uses $O(n^2 w^2)$ single-precision operations.

3.2 A Generic Factoring Algorithm

All three factoring algorithms that we implemented can be viewed as specializations of the following "generic" algorithm.

1. Find $h \in \mathbf{Z}_p[x]$, such that $h^p \equiv h \pmod{f}$ and $0 < \deg(h) < n$.

2. Compute constants $c_0, c_1 \in \mathbf{Z}_p$ such that $h^2 + c_1 h + c_0 \equiv 0 \pmod{f}$.

3. Compute a solution $t \in \mathbf{Z}_p$ to the equation $t^2 + c_1 t + c_0 = 0$.

4. Compute $\gcd(h-t, f)$. This splits f.

The only difference in the three algorithms under consideration is how step 1 of this generic algorithm is implemented. For each algorithm, the computation in step 1 overwhelmingly dominates the running time.

Step 2 is trivial to implement. Step 3 is implemented using a probabilistic algorithm for computing square roots modulo p.

The correctness of this algorithm follows from the fact that $\mathbf{Z}_p[x]/(f)$ is isomorphic to $\mathbf{F}_{p^d} \oplus \mathbf{F}_{p^d}$ and that $(h \bmod f)$ lies in the subring isomorphic to $\mathbf{F}_p \oplus \mathbf{F}_p$.

Step 2 uses $O(nw(\log n + w))$ single-precision operations. Step 3 uses an expected number of $O(\log_2 p \, w^2)$ single-precion operations. Step 4 uses $O(n^2 w^2)$ single-precision operations.

The space requirement for steps 2–4 is $O(nw)$ machine words.

3.3 Berlekamp's algorithm

We now describe how step 1 of the generic algorithm is implemented using Berlekamp's method.

The basic idea is to build the $n \times n$ matrix representing the $\mathbf{Z}_p$-linear map $h \mapsto h^p - h \bmod f$, with respect to the basis $1, x, \ldots, x^{n-1}$, and then compute a nontrivial vector in the null space of this matrix.

To build the matrix, we first compute $g = x^p \bmod f$. This is done using a repeated squaring algorithm. A significant savings is obtained by using the left to right (i.e., high order to low order) repeated squaring algorithm, since multiplication by $x \bmod f$ is essentially free.

Once we have g, we fill the matrix by computing $g^2, g^3, \ldots, g^{n-1} \bmod f$. A significant savings is obtained by pre-computing the point-wise FFT representation of g once, and then using this point-wise representation to compute g^{i+1} from g^i.

Once we have the matrix, we compute a nontrivial vector in its null space using Gaussian elimination to put the matrix in row echelon form and then back-substitute. Curiously, the null-space algorithm in Knuth [K, §4.6.2] computes the *reduced* row echelon form of the matrix, which is significantly more expensive than computing the row echelon form and then back-substituting: $(1/2)n^3(1+o(1))$ multiplications in $\mathbf{Z}_p$ vs. $(1/3)n^3(1+o(1))$.

A final implementation note: in performing Gaussian elimination over $\mathbf{Z}_p$ for large p, a speed-up factor of about two in the running time is obtained by taking a "lazy" approach to reducing mod p—that is, we reduce integers mod p only occasionally.

The cost of computing h by this method is

$$O(n^3 w^2 + \log_2 p \, nw(\log n + w))$$

single-precision operations. The space requirement is $O(n^2 w)$ machine words.

3.4 The Cantor/Zassenhaus Algorithm

The basic idea in the Cantor/Zassenhaus algorithm is to choose a polynomial $g \in \mathbf{Z}_p[x]$ of degree less than n at random, and then compute

$$h = g^{(p^d-1)/(p-1)} \bmod f.$$

The polynomial h is computed using repeated squaring.

h will always satisfy $h^p \equiv h \bmod p$, and with probability $1 - 1/p$ it will not be a constant.

A significant computational savings (almost a factor of two) is obtained by choosing $g = x + r$, where $r \in \mathbf{Z}_p$ is chosen at random. The reason for the savings is that in using a left to right repeated squaring algorithm, multiplication by $x + r \bmod f$ is essentially free.

With this modification, it turns out that the probability that h is constant is still quite small. If $f = f_1 f_2$, then one sees that $h \equiv (-1)^d f_i(-r) \pmod{f_i}$, for $i = 1, 2$. Therefore, the probability that h is constant is at most $n/(2p)$, which for the large p that we are considering is quite negligible.

This second variant is the one we actually implemented. In the test cases we report on later, a "good" h was always found on the first try (as is to be expected).

The cost of computing h by this method is

$$O(\log_2 p \, n^2 w(\log n + w))$$

single precision operations. The space requirement is $O(nw)$ machine words.

3.5 The von zur Gathen/Shoup algorithm

Just like the Cantor/Zassenhaus algorithm, this algorithm chooses $r \in \mathbf{Z}_p$ at random, sets $g = x + r$, and then computes

$$h = g^{(p^d-1)/(p-1)} \bmod f.$$

However, the method of raising g to this power is entirely different. For $k \geq 1$, let

$$N(k) = (p^k - 1)/(p-1) = 1 + p + \cdots + p^{k-1}.$$

Then we have the following recursive formulation of $N(k)$, for $k \geq 1$:

$$\begin{aligned} N(2k) &= N(k) + N(k) \cdot p^k, \\ N(k+1) &= N(k) + p^k. \end{aligned}$$

The algorithm for computing $g^{N(d)} \bmod f$ scans the bits of d from left to right, maintain the polynomials $T_k = x^{p^k} \bmod f$ and $S_k = g^{N(k)} \bmod f$. The algorithms computes S_{2k} by computing $S_k \cdot S_k(T_k) \bmod f$, and computes T_{2k} by computing $T_k(T_k) \bmod f$. The algorithm computes S_{k+1} by computing $S_k \cdot (T_k + r) \bmod f$ (this exploits the special form of g), and computes T_{k+1} by computing $T_k(T_1) \bmod f$.

To get the process started, we have to compute $T_1 = x^p \bmod f$ by repeated squaring.

From this, we see that h can be computed using $O(\log n)$ multiplications mod f, and $O(\log n)$ "modular composition" problems; i.e., computing $S(T) \bmod f$, where S and T are polynomials of degree less than n.

The modular composition problems are solved using an adaptation of an algorithm of Brent & Kung [BK, Algorithm 2.1]. One modular composition problem can be solved with this method using $O(n^{1/2})$ multiplications in $\mathbf{Z}_p[x]/(f)$, plus $O(n^2)$ multiplications and additions in $\mathbf{Z}_p$. The space required for Brent & Kung's method is $O(n^{3/2})$ elements of $\mathbf{Z}_p$.

The cost of computing h by this method is

$$O(\log_2 p \, nw(\log n + w) + n^2 w^2 + n^{3/2}(\log n)^2 w)$$

single-precision operations. Asymptotically, the third term is dominated by the second; however, we include it here because in the range of n we are considering, the third term dominates the second.

The space requirement is $O(n^{3/2} w)$ machine words.

3.6 Timing results

The code we developed was compiled using the Gnu `C++` compiler and executed on a Silicon Graphics 4D/240S. All running times given below are in CPU seconds, as measured by the `getrusage` UNIX library routine.

Table 1 illustrates the effectiveness of our FFT-based polynomial multiplication, modulo a 100-bit prime. It shows the running time of the classical multiplication (CM) and division with remainder algorithms (CD), as well as the FFT-based algorithms (FM) and (FD).

Table 2 shows the running times of Berlekamp's algorithm (B), the Cantor/Zassenhaus algorithm (C/Z) and the von zur Gathen/Shoup (G/S) algorithm for a 50-bit prime modulus p.

Tables 3 and 4 give the same information for 100-bit, and 200-bit primes p, respectively.

In Table 3, the Cantor/Zassenhaus algorithm was not attempted for $n = 1004$. Table 3 also shows the running time of the factorizer implemented in Maple V (M) for $n = 24$, 56, and 104. This algorithm is a Cantor/Zassenhaus-type algorithm. To be fair, we ran only the final part of Maple's factorizer, which assumes that all irreducible factors of f have the same degree.

In Table 4, the Cantor/Zassenhaus algorithm was not attempted for $n = 500$ or $n = 1004$.

The largest polynomial that was factored by any of these algorithms was of degree 1500, modulo a 200-bit prime. The von zur Gathen/Shoup algorithm finished in $27,978$ seconds. Berlekamp's algorithm was attempted, but failed as the process attempted to allocate more memory than the standard limits imposed by the operating system. The Cantor/Zassenhaus algorithm was not attempted at all.

We attempted to factor a polynomial of degree about 2000 modulo a 200-bit prime, but in this case, even the von zur Gathen/Shoup algorithm exceeded the memory limits, and the factorization failed.

Finally, we report that all of these algorithms typically spent more than 98% of their time in step 1 of the "generic" algorithm.

4 Discussion

We now attempt to draw some conclusions from the results of this experiment.

1. It is clear that for polynomial multiplication and pre-conditioned division with remainder over $\mathbf{Z}_p$, FFT-based algorithms are far superior to the classical algorithms, and the crossover point is very low—near $n = 25$, depending on the size of p. Table 1 clearly illustrates this.

   We would conjecture the main reason that Maple's factoring algorithm does so poorly (see Table 3) is the fact that no fast polynomial arithmetic is utilized in these routines. Other reasons for the poor performance are the facts that the factoring code is interpreted rather than compiled, and that a significant amount of time is spent on storage management (even though only linear space is required!). However, we would speculate that these issues are of secondary importance.

2. It is also clear that the Cantor/Zassenhaus algorithm is uniformly much slower than either Berlekamp or von zur Gathen/Shoup, at least for the restricted form of the factoring problem we are looking at.

   However, from our data we can draw some more general conclusions about Berlekamp versus Cantor/Zassenhaus. We would assert that the running time of a general purpose factorizer based on Berlekamp would not be considerably slower than the special one we implemented here. On the other hand, a general purpose factorizer based on Cantor/Zassenhaus would be at least as slow as the special purpose one we implemented here. If one accepts these assertions, then our data strongly suggests that for large p, the general purpose Cantor/Zassenhaus algorithm is horrendously slow compared to Berlekamp. The only advantage of Cantor/Zassenhaus is its more economical use of space.

3. The data suggest that for the special factoring problem we are considering, the von zur Gathen/Shoup algorithm is at least competitive with Berlekamp's algorithm for moderately

n	CM	CD	FM	FD
24	.033	.037	.054	.052
56	.183	.192	.121	.118
104	.628	.657	.246	.240
252	3.74	3.85	.557	.556
500	14.9	15.4	1.16	1.16
1004	62.0	63.9	2.44	2.43

Table 1: polynomial multiplication and division

n	B	C/Z	G/S
24	2	18	3
56	12	132	14
104	43	512	42
252	326	2,837	172
500	2,048	11,897	589
1004	14,339	49,995	2,545

Table 2: factoring with 50-bit primes

n	B	C/Z	G/S	M
24	8	68	10	2,152
56	39	609	42	4,770
104	115	2,311	114	58,216
252	716	13,013	438	—
500	4,030	53,736	1,399	—
1004	27,482	—	4,887	—

Table 3: factoring with 100-bit primes

n	B	C/Z	G/S
24	30	301	37
56	141	2,854	151
104	366	10,685	367
252	1,863	60,872	1,254
500	9,504	—	3,700
1004	61,036	—	12,024

Table 4: factoring with 200-bit primes

sized inputs, and perhaps even faster. Indeed, in our implementation, the cross-over point was near $n = 100$, and this was relatively insensitive to the size of p.

A possible objection to this conclusion is that the implementor of these algorithms (the author) was biased in favor of the von zur Gathen/Shoup algorithm (for obvious reasons). To this objection, we can only offer our assurances that great care was taken to ensure fairness.

A more substantive objection is that perhaps our diagonalization procedure is not the best possible. First, perhaps the diagonalization could be made faster by utilizing a modular technique with single-precision primes, yielding a diagonalization algorithm that uses $O(n^3w + n^2w^2)$ single-precision operations, instead of $O(n^3w^2)$ as in the current implementation. Second, perhaps for the large matrices being diagonalized, Gaussian elimination should not be used, and instead, a method based on asymptotically fast matrix multiplication should be used. At present, we have not investigated the efficacy of either approach.

Even if one accepts our conclusion, it is still quite weak. The speed-up over Berlekamp obtained by von zur Gathen/Shoup was quite modest (a factor of about 6, at most). This speed-up could conceivably be eliminated with a faster diagonalization procedure, as mentioned above. Also, we would assert that the general purpose version of von zur Gathen/Shoup is substantially slower than the special purpose one we implemented. Because of this, for the general factoring problem, we do not believe the general purpose von zur Gathen/Shoup algorithm is at all competitive with the general purpose version of Berlekamp's algorithm in the range of input sizes considered in our test suite. Indeed, devising a practical, general purpose version of the von zur Gathen/Shoup algorithm is an interesting open problem.

4. Finally, in light of our results, we attempt to answer the question: if one must implement a general purpose factorizer, what algorithm should be used?

   If this factorizer is only going to be used to factor polynomials of degree up to 12 modulo 8-bit primes, then it does not matter too much which algorithm is used. In posing this question, we are intending that this factorizer is expected to perform reasonably well on, say, polynomials of degree up to 1000 or so, and for large primes with up to several hundred bits or so.

   First, no matter what algorithm is used, if large polynomials are going to be factored, the FFT should be used for polynomial multiplication and division.

   Second, some variant of our "generic" algorithm should probably be used. If the number of irreducible factors of f is very small (which is typical for random input polynomials) then a reduction to root-finding via a minimal polynomial computation (generalizing steps 2–4 of our "generic" algorithm) is especially effective.

   Third, if nothing is known about the degrees of the factors of f, then Berlekamp seems like the best choice, provided the space requirement of Berlekamp does not become a problem.

   In situations where space is a problem for Berlekamp's algorithm, some adaptation of the techniques in [GS], perhaps in conjunction with a sparse linear system solver such as Wiedemann's [W], may be a practical alternative.

   Constructing a well-engineered general purpose factorizer is no easy task. Such a factorizer would select a strategy based on several factors: the relative sizes of n and p, the size of p relative to the machine word size, the space limitations of the machine, and the number of irreducible factors (as discovered early on in the algorithm).

References

[B] E. R. Berlekamp. Factoring polynomials over large finite fields. *Math. Comp.* **24**, 713–735 (1970).

[BK] R. P. Brent and H. T. Kung. Fast algorithms for manipulating formal power series. *J. Assoc. Comput. Mach.* **25**, 581–595 (1978).

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

[GS] J. von zur Gathen and V. Shoup. Computing Frobenius maps and factoring polynomials. *Comput. Complexity* **2**, 187–224 (1992).

[K] D. E. Knuth. *The Art of Computer Programming*, vol. 2, second edition. Addison-Wesley (1981).

[W] D. Wiedemann. Solving sparse linear systems over finite fields. *IEEE Trans. Inf. Theory* **IT-32**, 54–62 (1986).