Factorization in $\mathbb{Z}[x]$: The Searching Phase*

John Abbott
Dipartimento di Matematica, Università di Genova, Italy
abbott@dima.unige.it

Victor Shoup
IBM Research, Zurich, Switzerland
sho@zurich.ibm.com

Paul Zimmermann
INRIA Lorraine and LORIA, France
zimmerma@loria.fr

April 24, 2000

Abstract

In this paper we describe ideas used to accelerate the Searching Phase of the Berlekamp–Zassenhaus algorithm, the algorithm most widely used for computing factorizations in $\mathbb{Z}[x]$. Our ideas do not alter the theoretical worst-case complexity, but they do have a significant effect in practice: especially in those cases where the cost of the Searching Phase completely dominates the rest of the algorithm. A complete implementation of the ideas in this paper is publicly available in the library NTL [16]. We give timings of this implementation on some difficult factorization problems.

1 Introduction

The Berlekamp–Zassenhaus algorithm (BZA) was invented about 30 years ago [2, 18], and it "solved" the factorization problem in $\mathbb{Z}[x]$. Another algorithm had been discovered centuries earlier but was practical only for very small polynomials. The main "defect" of the new algorithm is that its

*To appear, *ISSAC 2000*.

final step, the Searching Phase, has worst-case complexity exponential in the degree of the input polynomial whereas all other steps are polynomial time. Families of polynomials which exhibit this exponential cost are known explicitly [10]. However, it has also been shown that the Searching Phase has polynomial time average complexity subject to two plausible conjectures [4].

Later, Lenstra, Lenstra and Lovász [11] employed a new lattice reduction algorithm, called LLL, instead of the Searching Phase to produce a factorization algorithm completely in polynomial time. With this result, the polynomial factorization problem had been "completely solved," at least in theory; in practice, however, the original BZA is generally much faster. A fuller account of the history of polynomial factorization can be found in [7, 8, 9].

One motivation for this study is Zimmermann's collection of difficult polynomials to factorize [19], all of which have arisen during the course of solving some problem with the exception of one which was artificially created to be hard to factorize, and which we shall ignore. The harder examples have defeated factorizers distributed with the more popular general-purpose systems for algebraic computation (in the sense that no result was obtained within a reasonable period of time). These hard polynomials are problematic because they provoke the exponential behaviour of BZA since they are of fairly high degree and have many factors over any small finite field (and probably over any finite field). The ideas described in this article have led to an implementation which can tackle all of these challenging polynomials routinely in a modest length of time on a modern workstation.

Experience shows that polynomials which are troublesome for BZA, while rare amongst all polynomials, do arise quite often as inputs to factorization requests, e.g., as resolvents used during Galois group computations. On such polynomials, current lattice-based factorizers are too slow, and a naive implementation of BZA even worse. We shall explain how the Searching Phase of BZA can be speeded up sufficiently so that these "nasty" polynomials can be factorized reasonably quickly.

In the Searching Phase, one must search through a large number of candidate factors, looking for "true factors." The usual approach is to implement fast tests that can quickly rule out most candidates, employing more expensive tests only when necessary. We follow this approach as well, and describe several tests that are quite effective, and yet do not seem to be widely used. We also introduce a technique that can have a much more dramatic impact: a "pruning" technique that allows huge numbers of candidate factors to be eliminated from the search without even being enumerated.

One noteworthy success is the factorization of the polynomial P_8 which is of degree 972 and has at least 54 factors modulo any small prime. Prior to our ideas, attempting such a factorization would have been deemed prohibitive, particularly if the polynomial should be irreducible, as indeed it proved to be.

This paper is organized as follows. Immediately below we introduce our notation and terminology. In §2 we recall in outline what the Searching Phase does. In §3 we focus attention on the inner loop of the Searching Phase: the various ways of distinguishing quickly good from bad candidate combinations of modular factors. The method for pruning the search space is explained in §4. Then in §5 we give some experimental results, and make some observations on the polynomials in Zimmermann's collection.

1.1 Notation and Assumptions

The Searching Phase is the last phase of BZA, and consequently works in a well-prepared environment which we describe here. We denote by f the polynomial being factorized. The polynomial f will have been "tamed" in various ways, in particular it is square-free, and there is no common factor dividing its coefficients. For clarity of exposition we shall also suppose that f is monic; the necessary alterations to handle non-monic polynomials are quite routine. A prime p not dividing the discriminant of f has been chosen: thus $f \bmod p$ is square-free and of full degree. Also a factorization $f \equiv \prod_{i=1}^{n} \bar{f}_i \bmod p^\kappa$ has been determined. We shall refer to these factors as **modular factors** in contrast to the **true factors** which are factors in $\mathbb{Z}[x]$. Again for clarity, we shall assume that the modular factors are monic. The exponent κ is large enough to permit recovery of all the true factors in the Searching Phase.

We shall say that a polynomial is **BZ-bad** if its factorization modulo any small prime p (not dividing the discriminant) comprises solely factors of very low degree compared to the degree of at least one true factor. It often happens that all the modular factors of a BZ-bad polynomial have the same degree, though this is not a requirement for a BZ-bad polynomial.

We shall also refer to a **degree set** for f: this is a set of integers which contains the degrees of all true factors of f. One can be determined easily [13] from the degrees of irreducible modular factors for several different primes.

We define the $d-1$ **coefficient** of a polynomial to mean the coefficient of the penultimate term: *i.e.*, where d is the degree of a polynomial g then the $d-1$ coefficient of g is the coefficient of x^{d-1} in g. Analogously, we define the $d-2$ coefficient to be the coefficient of x^{d-2} .

Our methods rely on knowledge of good bounds for certain coefficients of true factors. Most particularly for any true factor we shall want good bounding intervals for its $d-1$ and $d-2$ coefficients; good bounds for the other coefficients will be useful but they play a less crucial role. The question of bounds on factors has been extensively studied [5, 12]. Let $g = x^d + b_{d-1}x^{d-1} + b_{d-2}x^{d-2} + \cdots + b_0$ be a true factor of f of degree d then we can deduce immediately that $|b_{d-1}| \leq \rho d$ and $|b_{d-2}| \leq \rho d(d-1)/2$ where ρ is a root bound for f which can be found using the method in [5], for instance.

It will also be handy to know one or more small integers r_1, r_2, *etc.*, for which the values $f(r_1)$, $f(r_2)$, *etc.*, are all non-zero and preferably not too large. Suitable choices could be $r_1 = 1$ and $r_2 = 0$ under the assumption that any factors x or $x-1$ have been removed from f during its taming.

2 The Searching Phase

Here we recall the task which the Searching Phase accomplishes. The "input" to the Searching Phase is the environment described in §1.1.

In general the modular factorization of f is finer than the true factorization: *i.e.*, a true factor of f splits into a product of several modular factors. The task of the Searching Phase is to group the modular factors together correctly so as to recover the true factors. This is achieved by more or less blindly trying all possible combinations of the modular factors to see if they correspond to true factors. The first guidance on the order in which to search through the combinations was given in [4] where it was shown that taking combinations in increasing order of cardinality is better than in increasing order of the total degree of the combination. Henceforth we shall suppose that they are checked in order of increasing cardinality.

Usually the number of modular factors is somewhat greater than the number of true factors, so most combinations will be **bad**: that is they do not correspond to a true factor. BZ-bad polynomials are extreme in that the number of combinations is huge, but only very few yield true factors. Thus the main aim is to discard bad combinations as quickly as possible. We are interested equally in ways of disposing of individual combinations rapidly as in techniques for eliminating large blocks of combinations all at once.

```
SEARCH(n) :
    for r = 1, .., ⌊n/2⌋ do
        CARDINALITY(r, 1, n, {})

CARDINALITY(r', lo, hi, tuple_so_far) :
    if r' = 0 then { TEST(tuple_so_far); return; }
    if r' > hi − lo + 1 then return;
    CARDINALITY(r' − 1, lo + 1, hi, tuple_so_far ∪ {lo});
    CARDINALITY(r', lo + 1, hi, tuple_so_far);
```

Figure 1: Algorithm *SEARCH*

Algorithm SEARCH

To describe how various optimizations work, it is convenient to fix a particular strategy for enumerating subsets of modular factors of a given cardinality. Figure 1 shows a pair of algorithms for which a call to $SEARCH(n)$ will apply the function $TEST$ to all the subsets of $\{1, 2, \ldots, n\}$ up to size $\lfloor n/2 \rfloor$ in order of increasing cardinality. We can view these subsets as ordered tuples with the indices in each tuple sorted in increasing order. Algorithm SEARCH enumerates all tuples of a given length in lexicographic order. A good source of such combinatorial algorithms is [14]. Naturally, the corresponding piece of code in a factorizer, which could actually be iterative, will be more complicated because when a true factor is found the indices of the modular factors comprising it should be removed from further consideration.

2.1 Memory Stacks

The fact that test tuples are enumerated in lexicographic order allows us to reduce the amount of work needed to carry out the tests described in §3. Each of these tests can be cast into the following form. With each modular factor $\bar{f}_i$, for $1 \le i \le n$, we associate a value X_i, which lies in some group (or semi-group), and to test whether a given tuple $(t_1, \ldots, t_r)$ passes the test, we check if $\prod_{i=1}^{r} X_{t_i}$ satisfies a certain predicate.[1] Instead of computing each product $\prod_{i=1}^{r} X_{t_i}$ separately, we can "amortize" the cost of computing the product over several applications of the test. More precisely, whenever

[1] We use "product notation" here for a generic group, although for some tests a "sum notation" is more natural. Actually, for the $d-2$ test (see §3.2), the value X_{t_i} depends on $(t_1, \ldots, t_i)$, but the same approach still works.

we apply the test, we compute and store the intermediate products

$$P_0 = 1,\ P_j = P_{j-1} \cdot X_{t_j}\ (1 \leq j \leq n).$$

In such a way, if the last time we applied the test it was to a tuple that shares a prefix of length l with the current test tuple, we need to update only the last $n-l$ products. This approach is quite attractive, since if a test is performed very frequently, the amortized cost for computing the product is very small; moreover, this approach is *never* more expensive than the naive approach of computing each product independently.

The idea described above is hardly new, but it seems worthy of highlighting, since many available factorization implementations do not seem to fully exploit it. We shall refer to this technique as a **memory stack**.

3 Quick Elimination Tests

The matter of eliminating quickly bad combinations has already received some attention: the use of degree sets was discussed in [13], the idea of checking the size of the penultimate coefficient (the $d-1$ test) was mentioned in [7], and testing divisibility of the constant term (the evaluation test) too was mentioned in [7] and investigated more thoroughly in [1]. We introduce here the $d-2$ test, and in the next section introduce some important improvements to both the $d-1$ and $d-2$ tests. We describe the tests in the order in which we suggest they should be applied to specific combinations; the order affects efficiency not correctness.

We define the **effectiveness** of a test to be the proportion of the bad combinations which it detects (during a particular factorization computation). Note that the effectiveness depends strongly on the example. Moreover, a test which often has low effectiveness could still be useful if it is especially cheap to apply. On the other hand a test with high effectiveness which is very costly may not be useful. Clearly no combination can be declared to give a true factor until it has passed a test which is guaranteed to have perfect effectiveness, so the last test we apply is an absolute verification.

An interesting feature to note is that the first three tests described below have cost essentially independent of the modulus p^κ provided the ideas in §3.1.1 are used. It is also clear that the effectiveness of the degree test is independent of the modulus size, and we suspect that the same is true of the effectiveness of the $d-1$ and $d-2$ tests except perhaps in very rare cases where the bounding intervals for the respective coefficients are quite uncommonly large compared to the modulus p^κ — in such cases a small amount of over-lifting (*i.e.*, using a value of κ larger than strictly necessary) may be beneficial.

3.1 The $d-1$ Test

The test simply verifies that the $d-1$ coefficient of the product of a combination of modular factors lies within the bounds permitted for a true factor of f. We recall from §1.1 that in most cases a narrow bounding interval for this coefficient can easily be found.

The attraction of this test is that it is cheap because it involves only additions: the $d-1$ coefficient of a product of monic polynomials is just the sum of their respective $d-1$ coefficients. The success of this test hinges on two crucial features: typically, the width of the bounding interval for the $d-1$ coefficient of a true factor is very small compared to the modulus p^κ, and the $d-1$ coefficients of bad combinations usually seem to behave rather like uniformly distributed random variables modulo p^κ. In other words, it is improbable that the $d-1$ coefficient of a bad combination will lie in the relatively narrow interval allowed. Nonetheless, there are some BZ-bad polynomials for which this test is completely ineffective.

Armed with this knowledge we can eliminate all combinations whose $d-1$ coefficient lies outside the bounding interval with the expectation that this test probably rejects the vast majority of bad combinations and at low cost.

3.1.1 The Fixed-Point Trick

A simple trick which nevertheless proves to be of great practical benefit is to implement the $d-1$ test using machine arithmetic, as we now describe. Oddly enough this simple idea does not appear to have been used before.

The test as outlined above performs addition modulo p^κ, but that is too expensive: an addition modulo p^κ costs $O(\kappa \log p)$ against $O(1)$ for adding two machine integers. Instead of using arithmetic modulo p^κ we shall use "fixed point" arithmetic: recall that in the C programming language addition of `unsigned` integers automatically reduces modulo 2^w where w is the wordsize of the computer.

Let $c_1, c_2, \ldots, c_n$ be the $d-1$ coefficients of the individual modular factors; for a given candidate r-tuple $(t_1, t_2, \ldots, t_r)$ the $d-1$ test amounts to computing $\sum_{i=1}^{r} c_{t_i} \bmod p^\kappa$ and checking that the symmetric remainder lies within the small bounding interval. Now let $c_i^* = c_i/p^\kappa$ for each i, then the above test is equivalent to $\sum_{i=1}^{r} c^*_{t_i} \bmod 1$ being small using a symmetric remainder (*i.e.*, it should lie within the bounding interval divided by p^κ).

We shall compute an approximation to the sum of the $c^*_{t_i}$ using "fixed-point" arithmetic. Since addition of machine integers is automatically reduced modulo 2^w, a natural choice is to work with the quantities $2^w c^*_i$ which can be approximated well using machine integers. Define $C_i = \lfloor 2^w c^*_i \rfloor$ for each i; so each C_i is small enough to fit in a machine integer, and we have $C_i \leq 2^w c^*_i < C_i + 1$. From this, it follows that

$$\sum_{i=1}^{r} C_{t_i} \leq 2^w \sum_{i=1}^{r} c^*_{t_i} < r + \sum_{i=1}^{r} C_{t_i}.$$

So if this interval containing $2^w \sum_{i=1}^{r} c^*_{t_i}$ is disjoint from the bounding interval (rescaled by $2^w/p^\kappa$) then we can be certain that the tuple $(t_1, t_2, \ldots, t_r)$ is bad, otherwise we cannot be sure. Note that in testing whether these intervals overlap, it suffices to compute modulo 2^w, and so we can use the built-in machine arithmetic directly.

This technique weakens the test slightly: it is equivalent to loosening the bounds on the $d-1$ coefficient. In practice, though, this weakening is more than compensated for by the increase in speed.

3.2 The $d-2$ Test

The $d-1$ test is fast and usually effective at weeding out bad combinations though in rare cases it can be quite ineffective. To handle these cases reasonably swiftly we propose a "$d-2$" test. Even if the $d-1$ test is fairly effective, the $d-2$ test can still have a significant impact on the overall running time, since it is usually significantly faster than some of the subsequent tests.

The idea is completely analogous: we just verify that the $d-2$ coefficient of the product of a combination of modular factors lies within the bounds permitted for a true factor of f. We recall from §1.1 that in most cases a narrow bounding interval for this coefficient can easily be found.

Those keys to the success of the $d-1$ test are valid also for the $d-2$ test: a narrow bounding interval can easily be obtained, and we can determine the value of the $d-2$ coefficient of a product of monic modular factors just using addition. For this latter claim to be true we assume the existence of a table of products of all the possible pairs of $d-1$ coefficients of the modular factors, a table which can be pre-computed at modest cost. Moreover we must allow a number of additions quadratic in the number of terms in the product. Conveniently, the "fixed-point" trick used for the $d-1$ test works just as well here, and the use of a memory stack for this test will reduce significantly the number of additions per combination if the test is performed frequently.

3.3 Degree Test

We can discard any combination whose total degree lies outside the degree set since it cannot possibly yield a true factor as its degree differs from that of any true factor. This test is as cheap as the $d-1$ test using the "fixed point" idea, but can often be totally ineffective at weeding out bad combinations for BZ-bad polynomials. Customarily the degree test is used as the first test because of its speed, but with our new "fixed-point" trick the $d-1$ test is just as rapid and with generally better effectiveness. The $d-2$ test is slower than the degree test, but frequently much more effective, so it is perhaps better to perform the $d-2$ test before the degree test (but it probably does not really matter much).

We mention three points which are helpful in extracting the most from performing degree tests.

First, recall that a degree set is obtained by combining information about the factorization pattern of the given polynomial modulo several small primes. If the Searching Phase seems to be going slowly, we can from time to time factor the polynomial modulo additional small primes, thus possibly refining the degree set.

Second, one should compute degree sets for individual cardinalities, so that during the search phase, one can potentially rule out entire cardinalities without having to generate any tuples of that cardinality.

Third, one should be sure to completely update the degree set information whenever a true factor is found.

3.4 Evaluation Test

Let $g, h \in \mathbb{Z}[x]$. Now if $g|h$ then there is $m \in \mathbb{Z}[x]$ such that $h = gm \in \mathbb{Z}[x]$, and consequently $h(r) = g(r) \cdot m(r)$ for any $r \in \mathbb{Z}$. In other words $g(r)|h(r)$. Conversely, if $g(r) \nmid h(r)$ for some $r \in \mathbb{Z}$ then clearly $g \nmid h$. We can use this converse for detecting bad combinations. In the special case $r = 0$ we see that $g(r)$ is just the constant term of g.

Let r be any integer for which $|f(r)| < p^\kappa/2$; it is possible that no such r can be found, in which case either the evaluation test must be skipped or some over-lifting must be performed. Assume that the values $\bar{f}_i(r)$ modulo

p^κ have been pre-computed and stored. Then the value at r of a product of some of the $\bar{f}_i$ can be found by multiplication of the stored values modulo p^κ; viewing the symmetric remainder as an integer we can then test whether it divides $f(r)$, if not then the associated combination is surely bad.

Notes: several different evaluation points can be used, we suggest using $r = 1$ first and then $r = 0$; this test is generally far more costly than the tests above since multiplications modulo p^κ must be performed (at a cost of $O(\kappa^2 \log^2 p)$ for each multiplication using classical methods); the overall cost can be reduced significantly using a memory stack. Normally this test is very effective at detecting bad combinations: *i.e.*, almost no bad combinations pass.

3.5 Ultimate Verification

All the tests described here detect some proportion of the bad combinations. Ultimately, a combination which passes these tests must be verified absolutely, this we do by trial division.

To effect the final trial division check it is necessary to convert the product of the modular factors into a single polynomial over $\mathbb{Z}$. The cost of multiplying out the modular factors modulo p^κ is relatively high, and use of a memory stack makes little difference since this stage is hardly ever reached with a bad combination, hence there is not normally any common prefix with the last combination which reached this stage. Nonetheless, there is a chance of having a bad combination, and the potentially enormous cost of a failing trial division (in $\mathbb{Z}[x]$) makes further checks worthwhile: e.g., the quotient and remainder produced by applying long division to, say, $x^{100} + 100x + 1$ and $x^2 + 100x + 1$ would each contain coefficients with almost 200 decimal digits. Thus we check that the putative factor is plausible by ensuring that its coefficients all lie within their respective bounding intervals — the tighter the bounding intervals, the more stringent this test is. Similarly we check each coefficient of the quotient as it is produced: if ever a coefficient does not lie within the bounds permitted for a factor of f, we can abort immediately.

4 Speeding up the $d - 1$ Test using Tables

In this section we wish to focus attention on the $d - 1$ test which can be developed into an extremely rapid way of pruning the search space, eliminating many candidate tuples from consideration without even enumerating them. Provided the $d - 1$ test is itself effective (which is typically, but not always, the case), this pruning technique can dramatically reduce the overall running time. Our idea is related to methods used in solving the knapsack problem in combinatorics. To use the ideas here it is essential that the "fixed-point" trick of §3.1.1 be employed.

The idea is to build an oracle which can decide quickly whether a given r-tuple prefix can be extended to an r-tuple that would pass the $d-1$ test. A perfect oracle would cost too much to make, so we shall make an imperfect, but good, oracle which can answer in one of two ways: either "No, the prefix cannot be extended" or "Maybe the prefix can be extended". A good oracle should only rarely respond "Maybe" when a perfect oracle would have responded "No". We implement this oracle by table lookup. The size of the table is controlled by a parameter k, and the resulting tables have size about $k2^{k-1}$ bytes.

The oracle can be used in Algorithm SEARCH (see Figure 1) by inserting the following step between steps (2) and (3) in the auxiliary Algorithm CARDINALITY:

(2.5) if $lo > hi - k$ and $ORACLE(r', lo, hi, tuple\_so\_far) =$ "no" then return;

We now describe the implementation of the oracle in a bit more detail. We refer the reader to §3.1 for notation. Suppose that $tuple\_so\_far = (t_1, \ldots, t_i)$ where $i = r - r'$, and further that we maintain the sum $X = \sum_{j=1}^{i} C_{t_j} \bmod 2^w$ throughout the computation. We then use the values lo, r', and the high-order m bits of X as indices into a three-dimensional table of bits. The value m is a parameter whose choice is described below. The entry in the table will be '1' if there exists a tuple suffix $(t_{i+1}, \ldots, t_r)$ of cardinality r' with $t_{i+1} \geq lo$ such that $X + \sum_{j=i+1}^{r} C_{t_j} \bmod 2^w$ lies in the permitted interval. Given $C_1, \ldots, C_n$, it is straightforward to construct such a table. Of course, an entry may be '1' even if there is no such suffix; to control the number of such "false hits," for a given lo and r', the value m is chosen so that the density of '1'-entries in the table (for these values of lo and r') is between $1/(2k)$ and $1/k$. This particular density level was determined experimentally—it seemed to give the best time/space tradeoff on a number of examples.

This particular way of implementing the oracle seemed to be a very good practical compromise among a number of competing goals:

- keeping the tables small,
- making table lookup fast, and
- minimizing the number of "false hits."

In practice, if the $d-1$ test is itself effective, this pruning technique can have a quite dramatic impact on the running time, leading to a speed up by a factor on the order of 2^k. Thus, we obtain a tradeoff between the size of the tables ($k2^{k-1}$ bytes) and speed (a factor on the order of 2^k).

5 Experimental Results

5.1 Origin of the test polynomials

The polynomials P_1, P_2 and P_3 are contributed by Fabrice Rouillier; they come from the Rational Univariate Representation (RUR) [15] of the "Cyclic 6" system for P_1, of "Cyclic 7" for P_2 and P_3. P_4 was contributed by A. Hulpke and H. Matzat: it is the 5-set resolvent of the polynomial

$$\begin{aligned} f = {} & x^{11} + 101x^{10} + 4151x^9 + 87851x^8 + 976826x^7 \\ & + 4621826x^6 - 5948674x^5 - 113111674x^4 - 12236299x^3 \\ & + 1119536201x^2 - 1660753125x - 332150625, \end{aligned}$$

and its factorization proves that f has Galois group M_{11}. P_5 is the Swinnerton-Dyer polynomial for $2, 3, 5, 7, 11, 13$, *i.e.*, the product of all 64 monomials of the form $x \pm \sqrt{2} \pm \sqrt{3} \pm \cdots \pm \sqrt{13}$, which once expanded has only integer coefficients. P_6 is related to Galois group computations, and was contributed by Frédéric Lehobey (University of Rennes) and Nicolas Rennert (LIP6, University Paris 6); it is the resultant with respect to x of the polynomials $p(x)$ and $p(y-2x)$, where $p(x) =$

$$\begin{aligned} & x^{12} - 308x^{10} + 43109x^8 - 3312297x^6 \\ & \quad + 134697056x^4 - 2566974800x^2 + 1142440000. \end{aligned}$$

The polynomial P_8, contributed by Jean-Charles Faugère (LIP6, University Paris 6) [6], comes from the decomposition into irreducible primes of the ideal generated by the equations of the "Cyclic 9" algebraic system. During the computations in one (not yet irreducible) component Faugère found a non square-free polynomial of degree 2745 of the form $Q(x^9)$. He then called NTL first on $Q(x)$, then on $F(x^3)$ for all factors F of Q, then on $G(x^9)$ for all the factors of $F(x^3)$. At the end he found a list of 33 factors, which were all easy to factorize using NTL, except one of degree 972, which is precisely the polynomial P_8.

5.2 Comparison with existing software

For want of a better test suite we have used the polynomials in Zimmermann's collection to construct the table below; it did not seem useful to

Poly.	Deg.	Height	n	NTL	Maple
P_1	156	10^{423}	60	0.2	0.6
P_2	196	10^{418}	20	1.1	1.8
P_3	336	10^{596}	28	2.2	2.4
P_4	462	10^{755}	42	28	> 50000
P_5	64	10^{39}	32	43	> 50000
P_6	144	10^{164}	48	0.7	48
P_8	972	10^{212}	54	1950	> 50000

Table 1: Table of Timings

include artificially created examples. All times given are in seconds, and tests were conducted on a Compaq computer whose processor is a 500Mhz Digital Alpha 21264 (EV6). The software used was NTL version 4.0a; for the factorization of P_8 the oracle parameter was set manually to $k = 26$, in all other cases the default setting used.

In Table 1 the column headed "Poly." contains the names in Zimmermann's collection; the degree and the order of magnitude of the largest coefficient are in the columns headed "Deg." and "Height"; the column headed n contains the smallest number of modular factors modulo any prime — not dividing the discriminant — up to 100; the column headed "NTL" is the time (in seconds) taken by NTL 4.0a to complete the factorization; finally, the column headed "Maple" is the time (in seconds) taken by the computer algebra system Maple (version V Release 5.1) to factorize the polynomial, in some cases the computation was stopped because it was taking too long.

Our implementation in NTL was able to factorize P_8, a polynomial having at least 54 modular factors (for primes up to 1000), in under an hour. With the oracle parameter $k = 26$ the tables occupied a total of about 700Mbytes and the computation was completed in about 2000 seconds; with $k = 25$ the time rose to about 2800 seconds and the tables occupied about 300Mbytes. A full search through all the combinations would have entailed examining $2^{53} \approx 9 \times 10^{15}$ cases; it would take years of CPU time merely to generate all those combinations, let alone process them in any way.

Exploiting the structure $g(x^m)$. When a polynomial $f(x)$ has the special form $g(x^m)$ for $m > 1$, one can exploit this structure by first factoring $g(x)$, then substituting x by x^m in the factors $h(x)$ of $g(x)$, and finally factoring $h(x^m)$. The last step is required since $h(x)$ is irreducible when $h(x^m)$ is, but the converse is false: $h(x^2) = x^2 - 1$ is a counter-example. This trick dramatically speeds up the factorization of polynomials of the form $g(x^m)$

when $g(x)$ factors; if not, the overhead is rather small since the degree of the latter is at least twice as small as that of the former. Version 4.0a of NTL automatically exploits this special form.

5.3 Observations about Zimmermann's Collection

Here we make some comments and observations about factorizing the polynomials Zimmermann has collected.

The first three polynomials, P_1, P_2 and P_3 are not BZ-bad: though they do have many modular factors they also have many true factors, so the Searching Phase does not take long. In fact, in NTL, most of the time was spent in the Hensel Lifting Phase. Probably the performance of NTL could be improved in these cases by employing an "early factor detection" technique for very small cardinalities [3, 17].

The polynomial P_4 is the first interesting case. It is BZ-bad having only 2 true factors but 42 or more modular factors (certainly for all primes up to 1000). Unusually, we find that the $d-1$ test is completely ineffective. The rapid completion of the factorization is due to the combined effects of the $d-2$ test and the degree test; the latter proved highly effective at eliminating entire cardinalities — this polynomial was the motivation for the idea of refining the degree set every so often during searching. Indeed, once the search in cardinality 6 is complete, the remaining unfactored part can be proved irreducible by computing a degree set for all primes up to and including 73.

The polynomial P_5 is a Swinnerton-Dyer polynomial, *i.e.*, a member of the first family of BZ-bad polynomials to be discovered. It is irreducible but has modular factors of degree at most 2. The $d-1$ and $d-2$ tests are both quite effective. The degree test is totally ineffective (as for all Swinnerton-Dyer polynomials).

The polynomial P_6 is also BZ-bad: the modular factors have degree at most 3 (for primes up to 1000). The difficulty arises from the need to search up to cardinality 16 before finding the two largest true factors. Again the $d-1$ and $d-2$ tests prove quite effective. As soon as information from the prime 83 is included in the degree set, all odd cardinalities are instantly ruled out. Taking advantage of the structure $P_6(x) = g(x^2)$ led to a significantly shorter time.

The last polynomial P_8 is of high degree and is BZ-bad having at least 54 factors modulo any prime up to 1000. It is also irreducible: the worst case for the searching phase. Fortunately the $d-1$ test is extremely effective, and together with the table-based pruning scheme permits the factorization to be completed within a reasonable time. The $d-2$ test is highly effective too. P_8 is of the form $g(x^9)$, but since P_8 is irreducible, all time spent trying to factorize $g(x)$ and $g(x^3)$ is just wasted — in this case we waste about 15 seconds. P_8 is also self-reverse (*i.e.*, if the order of the coefficients is reversed then the result is the same polynomial); it may be possible to use this special structure to obtain a faster certification of irreducibility, but we have not done so.

6 Conclusions

In view of the recent developments reported here there is good reason to reconsider the widely held views that methods for factorization in $\mathbb{Z}[x]$ have been developed as far as possible, and that they are generally rather costly. Indeed, we have observed that a correct and careful implementation of our ideas for the Searching Phase can have an enormous impact on computation time for the worst cases for BZA (without any measurable penalty in the "normal" case).

The two key ideas were the use of "fixed-point" arithmetic in §3.1.1, and then using tables to prune the search for plausible combinations of modular factors in §4. The greater impact came from the latter idea.

The authors believe that there is still scope for significant improvement to current implementations of BZA, at least for certain classes of polynomial. To guide further practical developments it would be useful to know what sorts of polynomial are generally given to factorizers. Zimmermann's collection has already proved useful in this respect, but is only a small database. Nevertheless it is quite clear that these polynomials are very far removed from "random polynomials" (which are generally irreducible).

Acknowledgements. We thank Jean-Charles Faugère who helped motivate some of this work by asking us to factorize his huge polynomial P_8 of degree 972. The experiments of §5 were possible thanks to the Médicis Center at École Polytechnique (Palaiseau, France).

References

[1] Abbott, J., Bradford, R., and Davenport, J. A remark on factorization. *SIGSAM Bulletin 19*, 2 (1985), 31–33 & 37.

[2] Berlekamp, E. R. Factoring polynomials over large finite fields. *Mathematics of Computation 24*, 111 (1970), 713–735.

[3] Collins, G., and Encarnación, M. Improved techniques for factoring univariate polynomials. *Journal of Symbolic Computation 21* (1996), 313–327.

[4] Collins, G. E. Factoring univariate integral polynomials in polynomial average time. In *Proceedings of EUROSAM'79* (1979), pp. 317–329.

[5] Davenport, J., and Mignotte, M. On finding the largest root of a polynomial. *Modélisation Mathématique et Analyse Numérique 24*, 6 (1990), 693–696.

[6] Faugère, J.-C. How my computer find all the solutions of Cyclic 9. Tech. Rep. 007, Rapport LIP6, 2000.

[7] Kaltofen, E. Polynomial factorization. In *Computer Algebra*, B. Buchberger et *alii*, Ed., 2 ed. Springer Verlag, 1982, pp. 95–113.

[8] Kaltofen, E. Polynomial factorization 1982–1986. In *Computers in Mathematics*, Chudnovsky and Jenks, Eds., vol. 125 of *Lecture Notes in Pure and Applied Mathematics*. Marcel Dekker, 1990, pp. 285–309.

[9] Kaltofen, E. Polynomial factorization 1987–1991. Tech. Rep. 1, Rensselaer Polytechnic Institute, 1992. Also Proc. Latin'92, Springer LNCS 583, pages 294–313.

[10] Kaltofen, E., Musser, D., and Saunders, B. D. A generalized class of polynomials that are hard to factor. *SIAM J. Comput. 12*, 3 (1983), 473–483.

[11] Lenstra, A. K., Lenstra, H. W., and Lovász, L. Factoring polynomials with rational coefficients. *Mathematische Annalen 261* (1982), 515–534.

[12] Mignotte, M. An inequality about factors of polynomials. *Mathematics of Computation 28*, 128 (1974), 1153–1157.

[13] Musser, D. Multivariate polynomial factorization. *J. ACM 22*, 2 (1975), 291–308.

[14] Nijenhuis, A., and Wilf, H. S. *Combinatorial Algorithms*, second ed. Academic Press, 1978.

[15] Rouillier, F. Solving zero-dimensional systems through the Rational Univariate Representation. *Journal of Applicable Algebra in Engineering, Communication and Computing 9*, 5 (1999), 433–461.

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

[17] Wang, P. Early detection of true factors in univariate polynomial factorization. In *Proceedings of EUROCAL'83* (1983), Springer, Ed., vol. 162 of *Lecture Notes in Computer Science*, pp. 225–235.

[18] Zassenhaus, H. On Hensel factorization I. *Journal of Number Theory 1* (1969), 291–311.

[19] Zimmermann, P. Polynomial factorization challenges: a collection of polynomials difficult to factor. `http://www.loria.fr/~zimmerma/mupad`.