JOURNAL OF ALGORITHMS 15, 199–222 (1993)

Factor Refinement

ERIC BACH*

Computer Sciences Department, University of Wisconsin, Madison, Wisconsin 53706
bach@cs.wisc.edu

JAMES DRISCOLL†

Driscoll Brewing, 81 Oakwood Drive, Murray Hill, New Jersey 07974
driscoll@curacao.dartmouth.edu

AND

JEFFREY SHALLIT‡

Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1
shallit@graceland.uwaterloo.ca

Received October 1989; accepted October 28, 1992

Suppose we have obtained a partial factorization of an integer m, say $m = m_1 m_2 \cdots m_k$. Can we efficiently "refine" this factorization of m to a more complete factorization $m = \prod_{1 \le j \le l} n_j^{e_j}$, where all the $n_j \ge 2$ are pairwise relatively prime, and $l \ge 2$? A procedure to find such refinements can be used to convert a certain method for splitting integers into one that produces complete factorizations, to check whether $a^i b^j = c^k$ in polynomial time (in the lengths of a, b, c, i, j, k), to combine independently generated factorizations of a composite number, and to parallelize the generalized Chinese remainder algorithm. We present an algorithm for this problem and show that it uses $O(\log m)^2)$ bit operations, up to a constant factor the same as required for a single gcd using the naive arithmetic model. We also characterize the output of our factor refinement algorithm, showing that the result of factor refinement is actually a natural generalization of the greatest common divisor. Finally, we also show how similar results can be obtained for polynomials. As an application, we give algorithms to produce relatively prime squarefree factorizations and normal bases.

*Research supported by NSF Grants DCR-8504485 and DCR-8552596.
†Research supported by NSF Grant CCR-8809573 and a Walter Burke award from Dartmouth College.
‡Research supported by NSF Grant CCR-8817400, the Wisconsin Alumni Research Foundation, a Walter Burke award from Dartmouth College, and NSERC Canada.

0196-6774/93 $5.00

I. Introduction

Suppose we have obtained a partial factorization of a positive integer m, say $m = m_1 m_2$. Can we efficiently "refine" this factorization of m to another factorization

$$m = \prod_{1 \le j \le l} n_j^{e_j}, \tag{1}$$

where all the $n_j \ge 2$ are pairwise relatively prime, and $l \ge 2$? More generally, given a positive integer m expressed as the product of k nonunit factors $m = m_1 m_2 \cdots m_k$, can we efficiently find a set of pairwise relatively prime integers $B = \{n_1, n_2, \ldots, n_l\}$, each greater than one, such that each m_i can be written as $\prod_{1 \le j \le l} n_j^{f_{ij}}$, for appropriate nonnegative integer exponents f_{ij}? If in addition each n_j divides at least one m_i, we call such a set B a *gcd-free* basis. In this paper we will give quadratic-time algorithms for these problems.

The need for a *factor refinement* procedure was pointed out by Bach, Miller, and Shallit in 1984 [B1]. The situation in [B1] is as follows: an algorithm Split(N, M) is given to split N (i.e., write $N = ab$ with $1 < a, b < N$) in polynomial time. The algorithm works provided that $\sigma(N) | M$, where $\sigma(N)$ denotes the sum of the divisors of N. Now we wish to produce the *complete* factorization of N by repeated application of the algorithm. Merely running Split(a, M) may not succeed, however, as M is not necessarily a multiple of $\sigma(a)$. (That is, $\sigma(ab) = \sigma(a)\sigma(b)$ is guaranteed only when a and b are relatively prime.) Thus we need to *refine* the factorization of N into *relatively prime pieces* in order to reapply the splitting algorithm.

A second, perhaps more fundamental application is the following: suppose we are given six positive integers a, b, c, i, j, k. Can we determine in polynomial time (in the *lengths* of these six numbers) whether or not $a^i b^j = c^k$? We cannot simply perform the exponentiation, as that would take at least $i \log a + j \log b + k \log c$ time (and space). Using factor refinement, one can compute a gcd-free basis B for a, b, and c, and express a, b, and c as products of powers of elements from this basis. Checking if $a^i b^j = c^k$ then amounts to examining some equations in the exponents. (This was pointed out to us by H. W. Lenstra, Jr.)

A third application involves consolidation of independent factorizations of a composite number m. Suppose, for example, that we run the elliptic curve method of Lenstra [L1] and the quadratic sieve of Pomerance [P] simultaneously, obtaining two different factorizations, $m = ab = cd$. H. W. Lenstra, Jr. has asked how we can efficiently combine these factorizations. One way is to use factor refinement on the product $m^2 = abcd$, obtaining $m^2 = \prod_{1 \le j \le l} n_j^{e_j}$; then we can show using our methods

that $m = \prod_{1 \le j \le l} n_j^{e_j/2}$ is a factorization that incorporates all known factors of m. Our results show that this method is essentially optimal, as it runs in the same time required for a single gcd on the inputs, up to a small constant factor.

A fourth application is as follows: suppose we wish to solve instances of the generalized Chinese remainder problem, where the moduli are not necessarily relatively prime. More precisely, we wish to find solutions to the system $x \equiv x_i \pmod{m_i}$, $1 \le i \le k$. Gauss [G5, Art. 32] gave a polynomial-time method that converts pairs of congruences to a single congruence, and so $k - 1$ applications of his method suffice to obtain x. However, suppose we need to solve many such congruences for a single set of moduli $\{m_1, m_2, \ldots, m_k\}$, and we have $O(\log m_1 m_2 \cdots m_k)$ processors. How do we find a solution efficiently in parallel? The usual method of finding idempotent elements [A1] works well in this situation if the moduli are relatively prime, but here they are not.

Our solution is to use factor refinement to preprocess the moduli $\{m_1, m_2, \ldots, m_k\}$ and convert them to a set of relatively prime numbers $\{n_1^{e_1}, n_2^{e_2}, \ldots, n_l^{e_l}\}$ such that there exist $g_1, g_2, \ldots, g_s$ with

$$\operatorname{lcm}(m_1, m_2, \ldots, m_k) = \prod_{1 \le j \le l} n_j^{g_j};$$

further, each m_i can be written as $\prod_{1 \le j \le l} n_j^{f_{ij}}$. This gives us a new system that has exactly the same set of solutions as the old system, and it can be solved efficiently with at most $\log_2 m_1 m_2 \cdots m_k$ processors.

Other applications include detecting algebraic dependencies in radicals (see the remark in the Conclusions of [S2]); computing the Galois group of a Kummer field of exponent 2 (see [L3, Problem 3.4]); distributed factorization of polynomials (see [D3]); recognizing units in number fields (see [L3, Problem 5.2]), and efficiently solving linear equations with composite moduli (see [B2]).

In this paper, we first show that we can compute the desired refinement (1) in polynomial time. Our method is essentially as follows: given $m = m_1 m_2$, we compute $d = \gcd(m_1, m_2)$ and write

$$m = (m_1/d)(d^2)(m_2/d).$$

This process is then continued until all factors are relatively prime. A similar method is used in the case where there are more than two inputs, $m = m_1 m_2 \cdots m_k$.

It is not difficult to show that our factor refinement algorithm uses $O((\log m)^3)$ bit operations. The point of the paper is to obtain a much better bound. We prove that our factor refinement algorithm actually uses

$O((\log m)^2)$ bit operations, asymptotically the same time required for a single gcd computation. We also present a characterization of the algorithm's output, which demonstrates that the decomposition (1) is actually a natural generalization of the greatest common divisor.

Finally, we show how similar results may be obtained for polynomials over finite fields. As applications, we give efficient algorithms to produce relatively prime squarefree factorizations and normal bases.

II. Previous Work

Stieltjes, in 1890 seems to have been the first to study an algorithm similar to our factor refinement algorithm [S3, Section 19]. He was interested in expressing $\text{lcm}(a, b) = AB$, where A and B are integers satisfying the conditions $\gcd(A, B) = 1$, $A|a$, and $B|b$. (A and B can also be computed efficiently via factor refinement.) Later, Dedekind studied factorization by repeated use of Euclid's algorithm [D2]. His construction, however, is exponential and it does not give as complete a factorization as ours.

In the recent past, most authors have been concerned with factor refinement algorithms for polynomials rather than integers, although the underlying techniques are essentially the same. In 1974, Collins [C2] gave a factor refinement algorithm for polynomials, and stated results analogous to our Theorem 1, parts (b) and (c), without proof. (Such results were also noted by von zur Gathen [G2].) Collins also gave a description of the output, similar to our Theorem 3, but did not prove uniqueness. In 1980, Wang [W2] discussed factor refinement for multivariate polynomials; in a 1984 paper dealing with parallel computation, von zur Gathen mentioned a refinement method that is exponential in the number of input polynomials [G1]. The 1984 paper of Bach, Miller, and Shallit [B1] gave a factor refinement algorithm for integers. In 1985, Kaltofen [K1, K2] defined the closely related notion of gcd-free basis and proved uniqueness in the context of a general unique factorization domain.

None of these authors gave explicit running times for factor refinement, beyond stating that their methods ran in polynomial time. Explicit running times are known in some cases, however, for a related algorithm on polynomials that might be called "squarefree factor refinement." (Unlike our algorithm, the factors produced by this method are all squarefree.) Collins [C3] analyzed such an algorithm for univariate polynomials with integer coefficients, and Epstein [E] extended the analysis to Gaussian integer polynomials. Ben-Or, Kozen, and Reif [B4] showed that a squarefree factor refinement of univariate polynomials over a field of characteristic zero can be computed with an NC algorithm, assuming that a field

operation takes one time unit. This was later extended to fields of characteristic p by Kaltofen, Krishnamoorthy, and Saunders [K3, K4], under a further assumption that pth roots in the field may be computed in unit time.

There is also a connection between factor refinement and a 1985 algorithm of Lüneburg [L4]. The latter method takes as input two elements a, b from some principal ideal domain, and finds another element r such that $r|a$, $\gcd(r, b) = 1$, and each prime divisor of a/r divides b. Lüneburg gave several applications of his algorithm, including the two problems we will discuss in Section VI; he did not analyze its complexity.

III. Nondeterministic Factor Refinement Algorithm

In this section, we describe an algorithm that successively refines a partial factorization $m = m_1 m_2 \cdots m_k$ into relatively prime pieces. Here and elsewhere, we sometimes use a *product* such as $m = m_1 m_2 \cdots m_k$ to denote the *list* of factors $\{m_1, m_2, \ldots, m_k\}$. We first discuss the complexity model used throughout the paper.

Define

$$\lg n = \begin{cases} 1, & \text{if } n = 0; \\ 1 + \lfloor \log_2 |n| \rfloor, & \text{if } n > 0. \end{cases}$$

Thus $\lg n$ counts the number of bits in the binary representation of n.

We use the "naive bit complexity" model popularized by Collins [C1]. In this model, we can multiply m by n in $O((\lg m)(\lg n))$ bit operations, and we can express $m = qn + r$, $0 \le |r| < |n|$, in $O((\lg q)(\lg n))$ bit operations. From this, it is not difficult to show that we can compute $d = \gcd(m, n)$ in $O((\lg m)(\lg n))$ bit operations. Now we state our first factor refinement algorithm:

Algorithm Refine.

Input. k integers $m_1, m_2, \ldots, m_k \ge 2$.
Output. List of pairs $L = \{(n_1, e_1), (n_2, e_2), \ldots, (n_l, e_l)\}$ such that
(1) $\prod_{1 \le j \le l} n_j^{e_j} = m$, where $m = \prod_{1 \le i \le k} m_i$.
(2) $\gcd(n_i, n_j) = 1$ for all $i \ne j$;

comment The algorithm maintains a list L of pairs (n_j, e_j) such that at all times $m = \prod_j n_j^{e_j}$.
initialize $n_i \leftarrow m_i, e_i \leftarrow 1$, for $1 \le i \le k$.
while there remain i, j with $\gcd(n_i, n_j) \ne 1$ **do**
begin
$d \leftarrow \gcd(n_i, n_j)$;

remove pairs $(n_i, e_i), (n_j, e_j)$ from L;
add the pairs $(n_i/d, e_i), (d, e_i + e_j), (n_j/d, e_j)$ to L, except for those pairs containing 1 as their first entry;
end;
output List of pairs $L = \{(n_1, e_1), (n_2, e_2), \ldots, (n_l, e_l)\}$.

We emphasize that the algorithm is *nondeterministic*: the order in which the pairs n_i, n_j are examined is left unspecified. We will prove below in Theorem 3 that the output of the algorithm does not depend on the order in which the pairs are examined.

THEOREM 1. *The algorithm Refine terminates, and on input* $m_1, m_2, \ldots, m_k$ *produces as output a list of pairs* $L = \{(n_1, e_1), (n_2, e_2), \ldots, (n_l, e_l)\}$ *such that*

(a) $\prod_{1 \le j \le l} n_j^{e_j} = m$;

(b) $\gcd(n_i, n_j) = 1$ *for all* $i \ne j$;

(c) *Each* m_i *is a multiplicative combination of the* n_j*; that is, there exist nonnegative integers* f_{ij}, $1 \le i \le k$, $1 \le j \le l$, *such that* $m_i = \prod_{1 \le j \le l} n_j^{f_{ij}}$.

Proof. Let us refer to the process of replacing the pairs

$$(n_i, e_i), (n_j, e_j) \tag{2}$$

with

$$(n_i/d, e_i), (d, e_i + e_j), (n_j/d, e_j) \tag{3}$$

as a *refinement step*. We prove that the algorithm terminates after at most $\log_2 m$ refinement steps.

The list of pairs L changes with each execution of a refinement step. To keep track of how L changes throughout the algorithm, let us denote the list L after t refinement steps as L_t. Thus $L_0 = \{(m_1, 1), (m_2, 1), \ldots, (m_k, 1)\}$. We also write

$$L_t = \left\{(n_1, e_1)^{(t)}, (n_2, e_2)^{(t)}, \ldots, (n_s, e_s)^{(t)}\right\},$$

where the upper subscript (t) is meant to indicate that we have executed t refinement steps. Note that s is a function of t.

We claim that the sequence

$$S_t = \sum_i \left(e_i^{(t)} - 1\right) \tag{4}$$

is strictly increasing. To prove this, we show that $S_{t+1} - S_t > 0$. It clearly suffices to consider how a refinement step changes the contribution of the terms corresponding to i and j. The pairs in (2) above contribute $e_i + e_j - 2$ to S_t. They are replaced by the pairs in (3) above. If neither m_i/d nor

m_j/d is equal to unity, the contribution corresponding to (3) in S_{t+1} is $2e_i + 2e_j - 3$, so $S_{t+1} - S_t = e_i + e_j - 1 \geq 1$. If one of $m_i/d, m_j/d$ equals unity (let us say m_i/d), the contribution to S_{t+1} is $e_i + 2e_j - 2$, so $S_{t+1} - S_t = e_j \geq 1$. And finally, if both m_i/d and m_j/d equal unity, the contribution to S_{t+1} is $e_i + e_j - 1$, so $S_{t+1} - S_t = 1$.

Now it is clear that $S_t \leq \log_2 m$, so we have proved that the algorithm terminates after at most $\log_2 m$ refinement steps, independent of the order in which the refinement is done. It is also clear that conditions (a) and (c) hold, for they hold at the start of the algorithm and they are preserved by each refinement step. Condition (b) holds, since the algorithm can terminate only if it is true. □

It will follow from Theorem 7 of Section V that we can compute the f_{ij} such that $m_i = \prod_{1 \leq j \leq l} n_j^{f_{ij}}$ using $O((\lg m)^2)$ bit operations.

We also observe that the bound of $\log_2 m$ refinement steps is tight, as the algorithm performs r refinement steps on input $m_1 = 2^r, m_2 = 2$. This example also shows that the running time for our refinement algorithm is quadratic, even if one uses the asymptotically fast gcd algorithm of Schönhage [S0]. It would be interesting to find a subquadratic refinement algorithm.

COROLLARY 2. *Algorithm Refine runs in polynomial time.*

Proof. Above we have seen that the algorithm requires at most $\log_2 m$ refinement steps. Each refinement step involves finding a pair from the list with nontrivial gcd and then performing two divisions. The list never contains more than $\log_2 m$ entries. Let $T(m_i, m_j)$ denote the time to compute $\gcd(m_i, m_j)$; then our naive complexity bound given above implies that $T(m_i, m_j) = O((\lg m_i)(\lg m_j))$. Hence the time to find a pair with nontrivial gcd is

$$\begin{aligned}
\sum_{1<i<j\leq |L|} T(m_i, m_j) &\leq \sum_{1<i<j\leq |L|} c(\lg m_i)(\lg m_j) \\
&\leq c \sum_{1\leq i,j\leq |L|} (\lg m_i)(\lg m_j) \\
&= c\left(\sum_{1\leq i\leq |L|} \lg m_i\right)^2 \\
&\leq c\left(\sum_{1\leq i\leq |L|} 1 + \log_2 m_i\right)^2 \\
&\leq c(2\log_2 m)^2.
\end{aligned}$$

Hence a single refinement step can be done in $O((\lg m)^2)$ bit operations, and so the entire algorithm uses $O((\lg m)^3)$ bit operations. □

We will see below that this time bound of $O((\lg m)^3)$ can be improved. For now, we concentrate on giving a characterization of the output. If p is a prime and $p^a | m$ but $p^{a+1} \nmid m$, then we say p^a divides m exactly, and we write $p^a \| m$. We now generalize this familiar concept to the case where p is not necessarily prime:

DEFINITION. If $n^a | m$ and $\gcd(m/n^a, n) = 1$, then we say that n^a divides m *exactly*, and we write $n^a \| m$.

(In the literature, n is sometimes called a *unitary divisor* of m.)

Now it is easy to see that the conditions (a)–(c) in Theorem 1 do not suffice to characterize the output uniquely. For example, if $m_1 = 30, m_2 = 42$, then the algorithm produces the output $L = \{(5, 1), (6, 2), (7, 1)\}$, while the set $L' = \{(2, 2), (3, 2), (5, 1), (7, 1)\}$ also satisfies conditions (a)–(c). Note that it is unreasonable to expect that Refine could produce the output L', since $6 \| 30$ and $6 \| 42$ and thus "behaves like a prime number."

Intuitively, the factor refinement algorithm produces only those factors of m that we could "reasonably expect to find." We make this precise below. First, we extend the idea of exact divisibility to *sets* of positive integers:

DEFINITION. Let N and M be sets of positive integers and let the elements of N be pairwise relatively prime. We say that N divides M exactly (and write $N \| M$) if (i) for all $n \in N$, there exists $a > 0$ and $m \in M$ such that $n^a \| m$; and (ii) for all $n \in N$ and $m \in M$, there exists an $a \geq 0$ such that $n^a \| m$.

EXAMPLE. $\{2, 3, 5, 7\} \| \{5, 6, 7\}$. Note that $\|$ induces a partial order on sets of pairwise relatively prime integers.

We are now ready to characterize the output of the algorithm Refine.

THEOREM 3. *Let the input to Refine be* $M = \{m_1, m_2, \ldots, m_k\}$. *Then the output of Refine is the* unique *set of pairs* $L = \{(n_1, e_1), \ldots, (n_l, e_l)\}$ *such that*

(a) $\prod_{1 \leq j \leq l} n_j^{e_j} = m$, *and each* $n_j > 1$.

(b) $\gcd(n_i, n_j) = 1$ *for all* $i \neq j$;

(c) $N = \{n_1, \ldots, n_l\}$ *divides* M *exactly; and*

(d) N *is maximal* (*for the ordering defined by* $\|$) *among all sets satisfying* (a)–(c).

Proof. Note that, although earlier we claimed the output of Refine was a list, now we are referring to it as a set. By Theorem 1, part (b), this is legitimate.

We have already seen that (a) and (b) hold above. To show that (c) holds, it suffices to observe that by Theorem 1 we have $m_i = \prod_j n_j^{f_{ij}}$. Hence $n_j^{f_{ij}} | m_i$, and since the n_j are pairwise relatively prime, we have $n_j^{f_{ij}} \| m_i$.

Now let us prove that N is maximal. Let R be any set of integers satisfying (a)–(c). Let L_t denote, as above, the list of pairs L after the tth refinement step of the algorithm, and let $S(L_t)$ denote the set of first entries of the pairs in L_t.

We will show by induction on t that $R \| S(L_t)$. This is true for $t = 0$ by assumption. Now assume that $R \| S(L_t)$; we need to show that $R \| S(L_{t+1})$. It suffices to consider what happens when the pair

$$(n_i, e_i), (n_j, e_j)$$

is replaced by

$$(n_i/d, e_i), (d, e_i + e_j), (n_j/d, e_j).$$

Since $R \| S(L_t)$, for any $r \in R$ there exist a, b such that $r^a \| n_i$ and $r^b \| n_j$. Therefore, $r^{\min(a,b)} \| \gcd(n_i, n_j) = d$. From this we see $r^{a - \min(a,b)} \| n_i/d$ and $r^{b - \min(a,b)} \| n_j/d$. Thus $R \| S(L_{t+1})$. Thus we see $R \| S(L) = N$, so N is maximal.

To see that N is unique, assume there is another set $U = \{u_1, u_2, \ldots, u_r\}$ with properties (a)–(d). Then $N \| U$ and $U \| N$. Let $n \in N$. There must be a $u \in U$ with $\gcd(n, u) \neq 1$, for by (a) there exists $h_1, h_2, \ldots, h_r$ such that

$$\prod_{1 \le j \le r} u_j^{h_j} = m = \prod_{1 \le j \le l} n_j^{e_j}.$$

Then there exist $b, c > 0$ such that $n^b \| u$ and $u^c \| n$. Then $b = c = 1$ and $n = u$. Thus $N = U$. □

IV. A Modified Refinement Algorithm

In this section and the next two, we are concerned with bounding the running time of factor refinement.

Consider the factor refinement algorithm with exactly two inputs. We show that there is an easy way to keep track of the pairs (n_i, n_j) with nontrivial gcd. To do this, we revise the algorithm of the previous section so as to keep the pairs in an ordered list such that only elements adjacent in the list can have a nontrivial gcd. Let (n_i, e_i) refer to the ith pair in the ordered list.

ALGORITHM Pair-Refine.

Input. Integers $m_1, m_2 \geq 2$.

initialize $n_1 \leftarrow m_1, n_2 \leftarrow m_2, e_1 \leftarrow e_2 \leftarrow 1$.
while there remains i with both $n_i, n_{i+1} \neq 1$ **do**
begin
$d \leftarrow \gcd(n_i, n_{i+1})$;
replace the pairs (n_i, e_i) and (n_{i+1}, e_{i+1}) with
$(n_i/d, e_i)$ and $(n_{i+1}/d, e_{i+1})$;
insert the pair $(d, e_i + e_{i+1})$ as the new (i + 1)th pair;
end;
output List of pairs $L = \{(n_i, e_i) : n_i \neq 1\}$

The algorithm is still nondeterministic in that the order in which the adjacent pairs are selected is left unspecified. A deterministic algorithm can carry out the refinement without generating pairs with $n_i = 1$, or otherwise marking adjacent pairs with trivial gcd's, by repeatedly refining on (n_i, e_i) and (n_{i+1}, e_{i+1}) until $\gcd(n_i, n_{i+1}) = 1$, and then incrementing i.

In fact, this algorithm amounts to the following: given m_1 and m_2, first compute $d = \gcd(m_1, m_2)$ and recursively refine m_1/d and d. At the end of this refinement, we are left with a list that terminates in a new d', which is $\leq d$. At this point, we recursively refine d' and m_2/d.

We give an example of the deterministic algorithm. Suppose we wish to refine 22891869 and 8164233. We display the list $n_i^{e_i}$ after each refinement step, with the ones left in for clarity. The underlining in each row identifies the numbers that are refined, giving the list in the following row:

$$\underline{22891869\ \ 8164233}$$
$$\underline{143}\ 160083^2\ 51$$
$$\underline{13\ 11^3}\ 14553^2\ 51$$
$$13\ 1\ \underline{11^3\ 14553^2}\ 51$$
$$13\ 1\ 1\ \underline{11^5\ 1323^2}\ 51$$
$$13\ 1\ 1\ 11^5\ 1\ \underline{1323^2\ 51}$$
$$13\ 1\ 1\ 11^5\ 1\ \underline{441^2\ 3^3}\ 17$$
$$13\ 1\ 1\ 11^5\ 1\ \underline{147^2\ 3^5}\ 1\ 17$$
$$13\ 1\ 1\ 11^5\ 1\ \underline{49^2\ 3^7}\ 1\ 1\ 17$$
$$13\ 1\ 1\ 11^5\ 1\ 49^2\ 1\ 3^7\ 1\ 1\ 17$$

Thus we conclude that $22891869 \cdot 8164233 = 3^7 \cdot 49^2 \cdot 11^5 \cdot 13 \cdot 17$.

LEMMA 4. *Algorithm* Pair-Refine *has the same input/output behavior as Algorithm* Refine.

Proof. It suffices to show that after every refinement step only adjacent pairs have a nontrivial gcd. This is because when the algorithm terminates, every adjacent pair has at least one component equal to one; hence we may conclude that all the n_j will be relatively prime.

We proceed by induction on the number of refinement steps. Initially $|L| = 2$, and the result is true. Assume the result is true after t refinement steps and we choose to refine (n_i, n_{i+1}). Before the refinement step we have

$$\ldots, n_{i-1}, n_i, n_{i+1}, n_{i+2}, \ldots$$

and after the refinement step we have

$$\ldots, n_{i-1}, n_i/d, d, n_{i+1}/d, n_{i+2}, \ldots,$$

where $d = \gcd(n_i, n_{i+1})$. By induction we have $\gcd(n_r, n_i) = 1$ for $r < i - 1$ and $r > i + 1$. Hence $\gcd(n_r, n_i/d) = 1$ for $r < i - 1$ and $r > i + 1$. Putting $e = n_i/d$, we see $\gcd(n_r, d) = \gcd(n_r, n_i/e) = 1$ for $r < i - 1$ and $r > i + 1$. Also, $\gcd(n_i/d, n_{i+1}/d) = 1$. By symmetry, the same results hold for $\gcd(n_r, n_{i+1}/d)$ for $r \leq i - 1$ and $r > i + 2$. Thus the result is true after $t + 1$ refinement steps as well. □

We now present a proof, due to H. W. Lenstra, Jr., that the deterministic version of Algorithm Pair-Refine discussed above runs in quadratic time. (We had previously given a more complicated proof based on amortized analysis; see [B0].)

In our proof we will use the following convention, which will simplify the analysis enormously: the cost to compute $d = \gcd(m_1, m_2)$, m_1/d, and m_2/d is bounded by $2(\log m_1)(\log m_2)$ bit operations. Provided $m_1, m_2 \neq 1$, this cost differs only by a constant factor from the naive bit complexity model discussed earlier in Section III. However, if $m_1 = 1$ or $m_2 = 1$, this cost is zero, while the naive bit complexity is positive. Thus our assumption might conceivably lead to an inaccurate estimate of the running time of the algorithm if many ones were encountered.

One way to handle this difficulty is simply to rewrite the algorithm slightly to avoid refining against one, whenever a one appears; this can be accomplished as discussed previously. But this requires an extra cost, which is attributed to discarding all ones as they appear. Let us convince ourselves that only $O(\log_2 m)$ ones are ever encountered, if we use the modification just suggested:

Let $W(m)$ denote the maximum number of ones encountered when refining two numbers m_1, m_2 whose product is m. We will prove by

induction that, for $m \geq 2$, $W(m) \leq 6 \log_2 m - 4$. In the refinement algorithm, we first compute $d = \gcd(m_1, m_2)$. Then we refine m_1/d and d, obtaining a result $s_1, s_2, \ldots, s_{k-1}, d'$. Then we refine d' and m_2/d. Thus we see that $W(m) \leq V(m)$, where V is defined as

$$V(m) = \begin{cases} 2, & \text{if } m = 1 \text{ or } m = p, \text{ a prime;} \\ 4 + \max\limits_{\substack{e \mid m \\ e \neq 1, m}} (V(m/e) + V(e)), & \text{otherwise.} \end{cases}$$

It is now easy to see by induction that $V(m) \leq 6 \log_2 m - 4$ for all $m \geq 2$. Hence we are free to assume that we never attempt to further refine an input that contains ones.

Now suppose we perform our factor refinement on input m_1, m_2. Let $m = m_1 m_2$; we will prove by induction on m that the algorithm uses no more than $(\log m)^2$ bit operations (exclusive of the time needed to handle any ones that appear). The base case, $m = 1$, is left to the reader.

Now the time to refine m, say $T(m)$, is bounded by the time to refine $(m_1/d)d = m_1$, the time to refine $d'(m_2/d) \leq m_2$, and the cost of computing $\gcd(m_1, m_2)$ and the quotients $m_1/d, m_2/d$. By our convention, the cost of the gcd and the two divisions is bounded by $2(\log m_1)(\log m_2)$. Thus

$$T(m) \leq T(m_1) + T(m_2) + 2(\log m_1)(\log m_2).$$

Using the induction hypothesis, it follows that

$$T(m) \leq (\log m_1)^2 + (\log m_2)^2 + 2(\log m_1)(\log m_2);$$

hence $T(m) \leq (\log m)^2$. We have proved

THEOREM 5. *Let m_1 and m_2 be integers ≥ 2, and write $m = m_1 m_2$. Algorithm* Pair-Refine *correctly produces a refinement of m_1 and m_2 in $O((\lg m)^2)$ bit operations.*

V. Factor Refinement: The Case of More Than Two Inputs

We now turn to the general case: that of computing a refinement of a product of $k > 2$ numbers $m_1, m_2, \ldots, m_k$. Earlier, in [B0], we had given a complicated argument based on amortized analysis that this problem can also be solved in quadratic time, using our naive arithmetic model. However, H. W. Lenstra, Jr. subsequently found a much shorter proof of this bound, which he has graciously allowed us to present here.

THEOREM 6. *Let $m_1, m_2, \ldots, m_k$ be integers ≥ 2, and let $m = m_1 m_2 \cdots m_k$. Then we can compute a refinement of $m_1, m_2, \cdots, m_k$ using $O((\lg m)^2)$ bit operations.*

Proof. The algorithm we present is *incremental*: it first produces a refinement of m_1 and m_2, then m_1, m_2, and m_3, etc. For many applications (e.g., combining different integer factorizations as they are found), this "on-line" property is desirable.

To avoid unnecessary complications, we will avoid talking about the exponents attached to each number in our description of the algorithm. The reader can easily verify that the arithmetic on the exponents can be done within the quadratic time bound.

The algorithm is stated inductively: if $k = 2$, we compute a refinement of m_1 and m_2, using the deterministic version of Pair-Refine given in Section IV. Otherwise, we may assume that we have computed a refinement of $m_1, m_2, \ldots, m_k$, and the result is $n_1, n_2, \ldots, n_l$. Now we refine m_{k+1} against each of the n_j in turn, as follows: let $c_1 = m_{k+1}$, and first refine n_1 against c_1 (using our algorithm for two inputs), obtaining the result $d_{1,1}, d_{1,2}, \ldots, d_{1,s_1}, c_2$. Here c_2 is "what is left" of c_1 after the refinement is done; it is c_1 with the factors of n_1 removed. Next, we refine n_2 against c_2, obtaining the result $d_{2,1}, d_{2,2}, \ldots, d_{2,s_2}, c_3$. We continue in this fashion, and at the last step refine n_l against c_l, obtaining $d_{l,1}, d_{l,2}, \ldots, d_{l,s_l}, c_{l+1}$. We now claim that the complete refinement of $m_1, m_2, \ldots, m_k$ is given by the set

$$D = \{d_{i,j} : 1 \leq i \leq l, 1 \leq j \leq s_j\} \cup \{c_{l+1}\}.$$

In order to prove correctness of this algorithm, we only need verify that the members of D are pairwise relatively prime. This is easy to see, since each $d_{i,j}$ is a divisor of n_i, and all the n_i are relatively prime.

We now prove the quadratic time bound. As in the proof of Theorem 5, we will again assume that the cost of computing $d = \gcd(u, v)$ and the quotients u/d and v/d is bounded by $2(\log u)(\log v)$. For this bound to accurately reflect the cost of performing algorithm, we must remove any ones encountered in the refinement process, as before. Using the analysis given for Theorem 5, it easily follows that the number of ones encountered is $O((\log m)^2)$.

We now prove by induction on the number of inputs k that if the algorithm refines $m_1, m_2, \ldots, m_k$, obtaining an output of $n_1, n_2, \ldots, n_l$, then the number of bit operations used is

$$(\log m)^2 - \sum_{1 \leq j \leq l} (\log n_j)^2,$$

where $m = m_1 m_2 \cdots m_k$.

For $k = 2$, this follows simply from reexamining the argument given for Theorem 5. Letting $T(m)$ be the cost to refine $m = m_1 m_2$, we saw that $T(m) \le T(m_1) + T(m_2) + 2(\log m_1)(\log m_2)$. Here the term $T(m_1)$ comes from the cost of refining m_1/d and d, obtaining a set $n_1, n_2, \ldots, n_l = d'$. The term $T(m_2)$ comes from the cost of refining d' and m_2/d, obtaining a set $n'_1, n'_2, \ldots, n'_{l'}$. The complete refined set for m is then

$$\{n_1, n_2, \ldots, n_{l-1}, n'_1, n'_2, \ldots, n'_{l'}\}.$$

By induction, $T(m_1) \le (\log m_1)^2 - \sum_{1 \le j \le l} (\log n_j)^2$, and $T(m_2) \le (\log m_2)^2 - \sum_{1 \le j \le l'} (\log n'_j)^2$. Thus we find

$$\begin{aligned} T(m) &\le T(m_1) + T(m_2) + 2(\log m_1)(\log m_2) \\ &\le \left((\log m_1)^2 - \sum_{1 \le j \le l} (\log n_j)^2\right) + \left((\log m_2)^2 - \sum_{1 \le j \le l'} (\log n'_j)^2\right) \\ &\quad + 2(\log m_1)(\log m_2) \\ &\le (\log m)^2 - \sum_{1 \le j \le l-1} (\log n_j)^2 - \sum_{1 \le j \le l'} (\log n'_j)^2, \end{aligned}$$

which gives us the desired bound.

Now we prove the result for more than two inputs, using induction. Assume it is true for all $k' \le k$; we will prove it is true for $k + 1$. Let $m = m_1 m_2 \cdots m_k m_{k+1}$. Then the cost of producing the refinement $n_1, n_2, \ldots, n_l$ of $m_1, \ldots, m_k$ is (by induction) bounded by $(\log m/m_{k+1})^2 - \sum_{1 \le j < l} (\log n_j)^2$. We now add to this the cost of refining n_1 against c_1, which is

$$(\log n_1 + \log c_1)^2 - (\log c_2)^2 - \sum_{1 \le i \le s_1} (\log d_{1,i})^2.$$

Then we refine n_2 against c_2, which costs

$$(\log n_2 + \log c_2)^2 - (\log c_3)^2 - \sum_{1 \le i \le s_2} (\log d_{2,i})^2.$$

We continue in this fashion, finally refining n_l against c_l, which costs

$$(\log n_l + \log c_l)^2 - (\log c_{l+1})^2 - \sum_{1 \le i \le s_l} (\log d_{2,l})^2.$$

Thus the total cost is bounded by

$$
\begin{aligned}
&(\log m/m_{k+1})^2 - \sum_{1\le j\le l} (\log n_j)^2 + \sum_{1\le j\le l} (\log n_j + \log c_j)^2 \\
&\quad - \sum_{2\le j\le l} (\log c_j)^2 - \sum_{d\in D} (\log d)^2 \\
&\quad \le (\log m - \log c_1)^2 + (\log c_1)^2 + \sum_{1\le j\le l} 2(\log n_j)(\log c_j) \\
&\qquad - \sum_{d\in D} (\log d)^2 \\
&\quad \le (\log m)^2 - 2(\log m)(\log c_1) + 2(\log c_1)\left(\left(\sum_{1\le j\le l} \log n_j\right) + \log c_1\right) \\
&\qquad - \sum_{d\in D} (\log d)^2 \\
&\quad \le (\log m)^2 - 2(\log m)(\log c_1) + 2(\log c_1)(\log m) - \sum_{d\in D} (\log d)^2 \\
&\quad \le (\log m)^2 - \sum_{d\in D} (\log d)^2,
\end{aligned}
$$

where we have used the fact that $c_1 \ge c_2 \ge \cdots \ge c_l$, and $c_1 n_1 n_2 \cdots n_l \le m$. This completes the proof. □

An interesting question, which we had not been able to resolve, was the following: does the quadratic time bound for refining $k > 2$ numbers hold independently of the order chosen to refine pairs with nontrivial gcd? H. W. Lenstra, Jr. (personal communication) has recently answered this positively, assuming one never refines numbers that are already "known" to be relatively prime.

Finally, we turn to the problem of expressing each m_i as a product of powers of the n_j in quadratic time.

THEOREM 7. *Given a gcd-free basis* $\{n_1, n_2, \ldots, n_l\}$ *for* $\{m_1, m_2, \ldots, m_k\}$, *a set of integers* ≥ 2, *we can express each* $m_i = \prod_j n_j^{f_{ij}}$ *using* $O((\lg m)^2)$ *bit operations, where* $m = m_1 m_2 \ldots m_k$.

Proof. For each m_i, we do the following: for each n_j, we trial divide m_i by n_j. If $n_j | m_i$, we set $m_i' = m_i/n_j$, and continue trial dividing m_i' by n_j, etc., until eventually a nonzero remainder is found.

The cost of the divisions for each pair (m_i, n_j) is therefore $(f_{ij} + 1)(\lg m_i)(\lg n_j)$, where $m_i = \prod_j n_j^{f_{ij}}$. Hence the total cost is bounded by

$$\begin{aligned}
&\sum_{i,j} (f_{ij} + 1)(\lg m_i)(\lg n_j) \\
&\quad \le \sum_{i,j} (f_{ij} + 1)(1 + \log_2 m_i)(1 + \log_2 n_j) \\
&\quad \le 4\sum_{i,j} f_{ij}(\log_2 m_i)(\log_2 n_j) + 4\sum_{i,j} (\log_2 m_i)(\log_2 n_j) \\
&\quad = 4\sum_{i,j} (\log_2 m_i)\left(\log_2 n_j^{f_{ij}}\right) + 4\sum_{i,j} (\log_2 m_i)(\log_2 n_j) \\
&\quad = 4\sum_{i} (\log_2 m_i) \sum_{j} \log_2 n_j^{f_{ij}} + 4\sum_{j} (\log_2 n_j) \sum_{i} \log_2 m_i \\
&\quad = 4\sum_{i} (\log_2 m_i)^2 + 4(\log_2 m) \sum_{j} (\log_2 n_j) \\
&\quad \le 8(\log_2 m)^2.
\end{aligned}$$

This completes the proof. □

VI. Results on Polynomials

Many algorithmic results about the integers also hold for polynomials over a finite field. In this section we indicate how factor refinement generalizes to this setting; our main result is a quadratic running time bound analogous to Theorem 6. We also show how factor refinement can be used to simply solve some problems in polynomial algebra.

Let k be a finite field. We assume that addition and subtraction of elements in k takes $O(\log|k|)$ bit operations and that multiplication and inversion (of a nonzero element) takes $O((\log|k|)^2)$ bit operations. These assumptions will hold if $k = GF(p)$ or k is implemented via an irreducible monic polynomial with coefficients in $GF(p)$, and the arithmetic is done by classical methods.

Let $k[X]$ denote the ring of polynomials in one variable with coefficients in k. This ring is a unique factorization domain: the units of this ring are nonzero constants, and prime elements are irreducible polynomials [W1]. Since we are primarily concerned with divisibility, we will henceforth assume that all polynomials are monic, unless otherwise stated.

Let a and b be polynomials, with $\deg a \ge \deg b > 0$. We will assume that the cost to compute $d = \gcd(a, b)$, a/d, and b/d is $O((\deg a)(\deg b))$ operations in k. This assumption is true if one uses the Euclidean

algorithm to find the greatest common divisor and if one does polynomial arithmetic by classical methods [K5].

We now indicate how to modify our analysis of integer factor refinement so as to derive analogous results for polynomials.

First, we note that Algorithm Refine works as stated for polynomials in $k[X]$, and Theorem 1 is true as well; here all references to "integers" must be changed to "polynomials." Suitably modified, the proof of Theorem 1 shows that if the inputs $m_1, \ldots, m_r$ are all monic and of positive degree, then the algorithm terminates after at most d refinement steps, where d is the sum of the degrees of the inputs. This shows that Corollary 2 holds in $k[X]$. Finally, an analog of Theorem 3 can be proved for $k[X]$, provided we require that all polynomials be monic.

Algorithm Pair-Refine also works for polynomials; the analog to Theorem 5 is the following.

THEOREM 8. *Let m_1 and m_2 be monic polynomials in $k[X]$, with positive degrees d_1 and d_2, respectively. Let $d = d_1 + d_2$. The* Algorithm Pair-Refine *uses $O(d^2)$ operations in k.*

Proof (Sketch). This is proved with the same calculation as Theorem 5, with the following modification: everywhere that the logarithm of a number appears, it should be replaced by the degree of a polynomial. □

The similarity of this result to Theorem 5 may be strengthened by a notational convention that we now describe. If f is a monic polynomial in $k[X]$, we denote its "nominal length" in bits by $\lg f$. More precisely, this is

$$\lg f = \begin{cases} 1, & \text{if } f = 1; \\ (\deg f)(\log |k|), & \text{otherwise.} \end{cases}$$

With this convention, the ordinary algorithms for polynomials in $k[X]$ have the same bit complexity as corresponding algorithms for the integers, and we have the following result:

COROLLARY 9. *Let m_1 and m_2 be monic polynomials in $k[X]$, of positive degree. If $m = m_1 m_2$, then* Algorithm Pair-Refine *uses $O((\lg m)^2)$ bit operations.*

By similarly replacing logarithms by degrees, one can also prove the following analog of Theorem 6.

THEOREM 10. *Let $m_1, \ldots, m_l$ be monic polynomials in $k[X]$, of positive degree. A refinement of $m = m_1 m_2 \cdots m_l$ may be calculated using $O((\deg m)^2)$ operations in k. Hence this can be done with $O((\lg m)^2)$ bit operations.*

Proof. Similar to that of Theorem 6; left to the reader. □

It will be observed that the factor refinement process can be applied to structures other than integers and polynomials over a finite field. A suitable abstract setting for factor refinement is given by the so-called "Gaussian semigroups" of Jacobson [J1]. These include, for example, the multiplicative semigroup of any unique factorization domain, and the multiplicative semigroup of ideals in a Dedekind domain. If a Gaussian semigroup has the analog of a logarithm or degree function, then one can prove results similar to Theorems 8 and 10. We leave the details of this development to the reader.

We now use factor refinement to solve two problems: squarefree decomposition in $k[X]$, and construction of normal bases for finite fields. Both of these problems have known polynomial time algorithms; in both cases, however, factor refinement leads to simple algorithms that are easy to analyze.

Many algorithms to factor polynomials in $k[X]$ (such as Berlekamp's [B5]) will not work unless the input polynomial has at least two distinct factors. The simplest way to guarantee this is to partially factor the input into squarefree polynomials; then nothing presented to the factorization algorithm will be a power of an irreducible polynomial, unless it is irreducible itself. The theorem below shows that if k is a prime field, this preprocessing may be done in quadratic time. We note that no known polynomial factorization algorithm runs in quadratic time, although there are some that approach this performance [B3, G4, S1].

THEOREM 11. *Let $k = GF(p)$ denote the finite field of p elements, where p is prime. Let $f \in k[X]$ be a monic polynomial, of degree $d \geq 2$. Then we can produce a relatively prime factorization $f = f_1^{e_1} \cdots f_r^{e_r}$, in which each f_i is squarefree, with $O(d^2)$ operations in k. Hence this can be done using $O((\lg f)^2)$ bit operations.*

Proof. The idea is to repeatedly apply factor refinement to $f/\gcd(f, f')$ and $\gcd(f, f')$, rewriting factors of the form $h(X)^p$ as $h(X)^p$ when they appear.

First, let $f = g_1^{e_1} \cdots g_r^{e_r} h^p$, where all factors appearing are pairwise relatively prime, not necessarily irreducible, and $e_1 < e_2 < \cdots < e_r$, with no e_i divisible by p. Then $f' = gh^p$, and $f/\gcd(f, f')$ is an associate of the squarefree polynomial $g_1 \cdots g_r$. Applying factor refinement to the inputs $f/\gcd(f, f')$ and $\gcd(f, f')$, one finds the pairs $\{(g_1, e_1), \ldots, (g_r, e_r), (h^p, 1)\}$. (This can be proved using Theorem 3, but it is easier just to exhibit a sequence of refinement steps with this result.)

If $h = 1$, we have the required factorization. Otherwise, note that for $h \in GF(p)[X]$, we have $h(X)^p = h(X^p)$; applying this as much as needed,

we obtain an expression m^{p^e} for h^p, where m is not a pth power. We now apply the algorithm recursively to m. The result of this, together with the factors of g computed earlier, gives a squarefree decomposition. To obtain a relatively prime decomposition, we apply factor refinement to all the factors thus found.

For the analysis, we observe that everything up to the recursive step can be done with $O(d^2)$ field operations. Since $\deg m \leq d/2$, the total number of operations to find a squarefree decomposition will be at most a constant times

$$d^2 + \frac{d^2}{4} + \frac{d^2}{4^2} + \cdots = O(d^2).$$

This bound holds for the final refinement step as well, so the total number of field operations is $O(d^2)$. The bit complexity bound follows easily from this. □

The bit complexity bound also applies to fields k that do not have prime order, if these fields are presented along with suitable information about their Galois groups. It suffices to be able to compute the Frobenius automorphism $x \mapsto x^p$ and its inverse in $O((\log|k|)^2)$ bit operations. This will be true, for example, if one knows matrices for these automorphisms, or k is defined using a quadratic or cyclotomic polynomial. In general, however, such matrices must be computed, and this takes $O((n + \log p)(\log|k|)^2)$ bit operations, when $|k| = p^n$.

Theorem 11 can also be proved using results of Yun [Y], who gave a squarefree decomposition algorithm for polynomials over a field of characteristic zero. As stated, Yun's algorithm does not work in characteristic p, but it may be easily modified to do so. The necessary modification results from the following observation. Write $f = \hat{f}g^p$, where no irreducible polynomial divides $\hat{f}$ to a power p or higher. Then in characteristic p, Yun's algorithm computes the squarefree decomposition of $\hat{f}$.

Yun showed that his algorithm uses $O(M(d)\log d)$ field operations, where $M(d)$ denotes the time required to multiply two polynomials of degree d. With our assumptions, this is $O(d^2 \log d)$; however, a more precise analysis shows that $O(d^2)$ field operations suffice, and this is also true for the extension of his algorithm to $GF(p)[X]$.

For other polynomial time algorithms, see [D1, G1, K5, M].

We now give another application of polynomial factor refinement. If k is a finite field of p^n elements, then by a *normal basis* of k over $GF(p)$ we mean a set $\{b_0, \ldots, b_{n-1}\} \subset k$ that forms a basis for k as a $GF(p)$-vector space, for which $b_i^p = b_{i+1}$ (with subscripts taken modulo n). It is a standard result of field theory that such a basis exists; it can be computed

in deterministic polynomial time [L2, L4]. (We are unaware of any bounds in the literature more precise than this, although both methods cited can be shown to have the same complexity as ours.)

One standard construction of a normal basis for a finite field [J2, p. 61] requires one to factor polynomials over $GF(p)$, and this cannot be done in deterministic polynomial time by any known method when p is large. Here we show that if this construction is modified to replace complete factorization by relatively prime factorization, then a polynomial time algorithm also results.

THEOREM 12. *Let k be a finite field of p^n elements, where p is prime. Then a normal basis for k may be computed using $O((n^2 + \log p)(\log|k|)^2)$ bit operations.*

Proof. Here is the construction. We consider k as a $GF(p)[X]$-module, where the operation of polynomials on field elements is determined by $(\sum c_i X^i)a = \sum c_i a^{p^i}$. Then, as $GF(p)[X]$-modules, we have

$$k \cong GF(p)[X]/(X^n - 1).$$

Linear algebra now guarantees that for some $b \in k$,

$$b, b^p, b^{p^2}, \ldots, b^{p^{n-1}}$$

are linearly independent; taking $b_i = b^{p^i}$ gives the desired normal basis.

To find b, let $k = GF(p)(\alpha)$, where α satisfies some irreducible monic polynomial equation of degree n. For each $i = 0, \ldots, n-1$, find the monic polynomial $f_i \in GF(p)[X]$ of least degree for which $f_i(X)\alpha^i = 0$. Now, apply factor refinement to the list of polynomials $f_0, \ldots, f_{n-1}$. This will give pairwise relatively prime polynomials $g_1, \ldots, g_s$ for which

$$f_i = \prod_{1 \le j \le s} g_j^{e_{ij}}, \qquad i = 0, \ldots, n-1.$$

For $j = 1, \ldots, s$, find an index i for which e_{ij} is maximized, let

$$h_j = f_i / g_j^{e_{ij}},$$

and take $\beta_j = \hat{h}_j(X)\alpha^i$. Then take

$$b = \sum_{j=1}^{s} \beta_j.$$

The running time analysis is straightforward. We note only that it is more efficient to use a matrix for the Frobenius map $x \mapsto x^p$ than to repeatedly compute pth powers. □

We make some comments on randomized algorithms for this problem. If we define $\varphi(f)$ for a monic polynomial $f \in GF(p)[X]$ to be the number of elements in $GF(p)[X]/(f)^*$, then a randomly chosen element of $GF(p^n)$ will generate a normal basis with probability

$$\varphi(X^n - 1)/p^n;$$

see [O]. The running time of our deterministic algorithm does not appear to be improved if a randomized method is used to factor $X^n - 1$; the bottleneck is in the computation of the annihilating polynomials f_i.

However, Artin's normal basis construction [A2], ostensibly for infinite fields, is a good randomized algorithm for large p. For completeness we describe it here. Let $k = GF(p)(\alpha)$, where α is a root of the irreducible monic polynomial f of degree n. Let

$$g(X) = \frac{f(X)}{(X - \alpha)f'(X)}.$$

Choose $t \in GF(p)$ at random, and let $b = g(t)$. Then if $p > 2n(n-1)$, the conjugates of b are linearly independent with probability at least $\frac{1}{2}$. The entire computation, including computation of a pth power matrix, uses $O((n + \log p)(\log|k|)^2)$ bit operations. For other randomized normal basis algorithms, see [F, G3].

Finally, we discuss how factor refinement solves a problem discussed by Lüneburg [L4].

THEOREM 13. *Let k be a finite field, and let $a, c \in k[X]$. Then we can find a polynomial r such that $r|a$, $\gcd(r, c) = 1$, and each prime divisor of a/r divides c, using $O(\lg(ac))$ bit operations.*

Proof. Apply factor refinement to the pair (a, c). We obtain $a = \prod g_i^{e_i}$, $c = \prod g_i^{e'_i}$, where the g_i's are pairwise relatively prime. Then take

$$r = \prod_{e'_i = 0} g_i^{e_i}. \quad \square$$

We note that Lüneburg's algorithm for this problem does not run in quadratic time, as can be seen by considering inputs of the form $a = f^d$, $c = fg^{d-1}$, where f and g are distinct irreducible polynomials. Evidently, Theorem 13 holds for integers as well.

ACKNOWLEDGMENTS

A previous version of this paper appeared in the proceedings of the 1990 "ACM-SIAM Symposium on Discrete Algorithms" [B0]. We thank E. Kaltofen for bringing the references

[G2, K1, K2, K3, K4, W2, Y] to our attention. H. W. Lenstra, Jr. generously shared with us his proofs and applications relating to factor refinement. We also thank one of the referees, who provided a different proof of the quadratic time bound in Theorem 6 that did not depend on our original amortization argument. Gudmund Frandsen read an earlier draft of this paper and made several useful suggestions. A manuscript of D. Bernstein [B6] analyzed several of the algorithms in this paper in a different way. We are grateful to him for his suggestions. We are also grateful to G. Ge, who pointed out some errors in an earlier version of this paper. Part of this work was done while the third author was at Dartmouth College and later a visiting professor at the University of Wisconsin, Madison.

REFERENCES

[A1] D. ANGLUIN, "Lecture Notes on the Complexity of Some Problems in Number Theory," Yale University, Department of Computer Science, Technical Report 243, August 1982.

[A2] E. ARTIN, "Galois Theory," 2nd ed., Univ. of Notre Dame Press, South Bend, 1966.

[B0] E. BACH, J. DRISCOLL, AND J. SHALLIT, Factor refinement, *in* "Proceedings, 1st Annual ACM-SIAM Symposium on Discrete Algorithms, 1990," pp. 201–211. A longer version of this article appeared as Computer Sciences Technical Report #883, University of Wisconsin, Madison, October 1989.

[B1] E. BACH, G. MILLER, AND J. SHALLIT, Sums of divisors, perfect numbers, and factoring, *in* "Proceedings, 16th ACM Symp. Theor. Comput., 1984," pp. 183–190; revised version appeared in *SIAM J. Comput.* **15** (1986), 1143–1154.

[B2] E. BACH, Linear algebra modulo N, in preparation.

[B3] M. BEN-OR, Probabilistic algorithms in finite fields, *in* "Proceedings, 22nd Annu. Symp. Found. Comput. Sci., 1981," pp. 394–398.

[B4] M. BEN-OR, D. KOZEN, AND J. REIF, The complexity of elementary algebra and geometry, *in* "Proceedings, 16th ACM Symp. Theor. Comput., 1984," 457–464; revised version appeared in *J. Comput. System Sci.* **32** (1986), 251–264.

[B5] E. R. BERLEKAMP, Factoring polynomials over large finite fields, *Math. Comp.* **24** (1970), 713–735.

[B6] D. BERNSTEIN, The coprime base algorithm, unpublished manuscript, March 23, 1992.

[C1] G. E. COLLINS, Computing time analyses for some arithmetic and algebraic algorithms, *in* "Proceedings, 1968 Summer Institute on Symbolic Mathematical Computations, IBM Federal Systems Center, 1968," pp. 195–231; also appeared as Computer Sciences Technical Report #36, University of Wisconsin, July 1968.

[C2] G. E. COLLINS, Quantifier elimination for real closed fields by cylindrical algebraic decomposition—Preliminary report, *SIGSAM Bull.* **8** (1974), 80–90.

[C3] G. E. COLLINS, Quantifier elimination for real closed fields by cylindrical algebraic decomposition, *in* "Proceedings, 2nd GI Conf.," Lecture Notes in Computer Science, Vol. 33, pp. 134–83, Springer-Verlag, New York/Berlin, 1975.

[D1] J. DAVENPORT, "On the Integration of Algebraic Functions," Lecture Notes in Computer Science, Vol. 102, Springer-Verlag, New York/Berlin, 1981.

[D2] R. DEDEKIND, Über Zerlegungen von Zahlen durch ihre grössten gemeinsamen Teiler, in "Festschrift der Technischen Hochschule zu Braunschweig bei Gelegenheit der 69. Versammlung Deutscher Naturforscher und Ärzte," 1897, pp. 1–40; reprinted in "Gesammelte Mathematische Werke," Vol. II, pp. 104–147, Vieweg, Braunschweig, 1932.

[D3] A. DIAZ, E. KALTOFEN, K. SCHMITZ, AND T. VALENTE, DSC: A system for distributed symbolic computation, *in* "ISSAC '90: Proceedings, International Symposium on Sym-

bolic and Algebraic Computation (Stephen M. Watt, Ed.), pp. 323–332, ACM Press, New York, 1990.

[E] H. EPSTEIN, Using basis computation to determine pseudo-multiplicative independence, *in* "Proceedings, 1976 ACM Symp. Symb. Alg. Comput., 1976," pp. 229–237.

[F] G. S. FRANDSEN, "Probabilistic Construction of Normal Basis (note)," Technical Report DAIMI PB-361, Computer Science Department, Aarhus University, Denmark, August 1991.

[G1] J. VON ZUR GATHEN, Parallel algorithms for algebraic problems, *SIAM J. Comput.* **13** (1984), 802–824.

[G2] J. VON ZUR GATHEN, Representations and parallel computations for rational functions, *SIAM J. Comput.* **15** (1986), 432–452.

[G3] J. VON ZUR GATHEN AND M. GIESBRECHT, Constructing normal bases in finite fields, *J. Symbolic Comput.* **10** (1990), 547–570.

[G4] J. VON ZUR GATHEN AND V. SHOUP, Computing Frobenius maps and factoring polynomials, *in* "Proceedings, 24th Annu. ACM Symp. Theor. Comput., 1992," pp. 97–105.

[G5] C. F. GAUSS, "Disquisitiones Arithmeticae," Springer-Verlag, New York/Berlin, 1986.

[J1] N. JACOBSON, "Lectures in Abstract Algebra, Vol. I: Basic Concepts," Van Nostrand, New York, 1951.

[J2] N. JACOBSON, "Lectures in Abstract Algebra, Vol. III: Theory of Fields and Galois Theory," Van Nostrand, New York, 1964.

[K1] E. KALTOFEN, "Sparse Hensel lifting," Rensselaer Polytechnic Institute, Department of Computer Science, Technical Report 85-12, 1985.

[K2] E. KALTOFEN, Sparse Hensel lifting, *in* "Proceedings, EUROCAL '85," Lecture Notes in Computer Science, Vol. 204, pp 4–17, Springer-Verlag, New York/Berlin, 1985.

[K3] E. KALTOFEN, M. S. KRISHNAMOORTHY, AND B. D. SAUNDERS, Fast parallel algorithms for similarity of matrices, *in* "SYMSAC '86: Proceedings, 1986 ACM Symp. Symb. Alg. Comp. 1986," pp. 65–70.

[K4] E. KALTOFEN, M. S. KRISHNAMOORTHY, AND B. D. SAUNDERS, Parallel algorithms for matrix normal forms, *Linear Algebra Appl.* **136** (1990), 189–208.

[K5] D. E. KNUTH, "The Art of Computer Programming, Vol. 2: Seminumerical Algorithms," Addison-Wesley, Reading, MA, 1981.

[L1] H. W. LENSTRA, JR., Factoring integers with elliptic curves, *Ann. Math.* **126** (1987), 649–673.

[L2] H. W. LENSTRA, JR., Finding isomorphisms between finite fields, *Math. Comp.* **56** (1991), 329–347.

[L3] H. W. LENSTRA, JR., Algorithms in algebraic number theory, *Bull. Amer. Math. Soc.* **26** (1992), 211–244.

[L4] H. LÜNEBURG, On a little but useful algorithm, *in* "Proceedings, AAEEC-3," Lecture Notes in Computer Science, Vol. 229, pp. 296–301, Springer-Verlag, New York/Berlin, 1985.

[M] D. R. MUSSER, "Algorithms for Polynomial Factorization," Ph.D. thesis, University of Wisconsin, 1971; also appeared as Computer Sciences Technical Report #134, University of Wisconsin, September 1971.

[O] O. ORE, Contributions to the theory of finite fields, *Trans. Amer. Math. Soc.* **36** (1934), 243–274.

[P] C. POMERANCE, Analysis and comparison of some integer factoring algorithms, *in* "Computational Methods in Number Theory," Math. Centre Tracts 154/155, pp. 89–139, Mathematisch Centrum, Amsterdam, 1982.

[S0] A. SCHÖNHAGE, Schnelle Berechnung von Kettenbruchentwicklungen, *Acta Inform.* **1** (1971), 139–144.

[S1] V. SHOUP, On the deterministic complexity of factoring polynomials over finite fields, *Inform. Process. Lett.* **33** (1990), 261–267.

[S2] T. J. SMEDLEY, Detecting algebraic dependencies between unnested radicals, *in* "ISSAC '90: Proceedings of the International Symposium on Symbolic and Algebraic Computation (Stephen M. Watt, Ed.), pp. 292–293, ACM Press, New York, 1990.

[S3] T. J. STIELTJES, Sur la théorie des nombres, *Ann. Fac. Sci. Toulouse* **4** (1890), 1–103; reprinted in "Oeuvres Complètes," Vol. II, pp. 279–377.

[W1] B. L. VAN DER WAERDEN, "Algebra," Vol. I, 7th ed., Ungar, New York, 1970.

[W2] P. S. WANG, The EEZ-GCD algorithm, *SIGSAM Bull.* **14**, No. 2 (1980), 50–60.

[Y] D. Y. Y. YUN, Fast algorithm for rational function integration, *Inform. Process. 77*, North-Holland, (1977), 493–498.