FACTOR REFINEMENT

by

Eric Bach
James Driscoll
Jeffrey Shallit

Computer Sciences Technical Report #883

October 1989

Factor Refinement

Eric Bach[*]
Computer Sciences Dept.
University of Wisconsin
Madison, WI 53706

James Driscoll[‡]
Math. and Comp. Sci.
Dartmouth College
Hanover, NH 03755

Jeffrey Shallit[§]
Math. and Comp. Sci.
Dartmouth College
Hanover, NH 03755

driscoll@cannon.dartmouth.edu

bach@cs.wisc.edu

shallit@dartmouth.edu

Abstract.

Suppose we have obtained a partial factorization of an integer m, say $m = m_1 m_2 \cdots m_j$. Can we efficiently "refine" this factorization of m to a more complete factorization

$$m = \prod_{1 \le i \le k} {n_i}^{e_i},$$

where all the $n_i \ge 2$ are pairwise relatively prime, and $k \ge 2$? A procedure to find such refinements can be used to convert a method for splitting integers into one that produces complete factorizations, to combine independently generated factorizations of a composite number, and to parallelize the generalized Chinese remainder algorithm.

We apply Sleator and Tarjan's formulation of amortized analysis to prove the surprising fact that our factor refinement algorithm takes $O((\log m)^2)$ bit operations, the same as required for a single gcd. This is our main result, and appears to be the first application of amortized techniques to the analysis of a number-theoretic algorithm.

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.

University of Wisconsin, Computer Sciences Technical Report #883

[*] Research supported by NSF grants DCR-8504485 and DCR-8552596.

[‡] Research supported by NSF grant CCR-8809573 and a Walter Burke award.

[§] Research supported by NSF grant CCR-8817400, the Wisconsin Alumni Research Foundation and a Walter Burke award.

I. Introduction.

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

$$m = \prod_{1 \leq i \leq k} {n_i}^{e_i}, \tag{1}$$

where all the $n_i \geq 2$ are pairwise relatively prime, and $k \geq 2$?

The need for factor refinement 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 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 has asked how we can efficiently combine these factorizations. One way is to use factor refinement on the product $m^2 = abcd$, obtaining $m^2 = \prod_{1 \leq i \leq k} {n_i}^{e_i}$; then we can show using our methods that $m = \prod_{1 \leq i \leq k} {n_i}^{e_i/2}$ 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 final 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 $x \equiv x_i \pmod{m_i}$, $1 \leq i \leq r$. Gauss [G3, Art. 32] gave a polynomial-time method that converts pairs of congruences to a single congruence, and so $r - 1$ applications of his method suffice to obtain x. However, suppose we need to solve many such congruences for a single set of moduli $\{m_1, m_2, \ldots, m_r\}$, and we have $O(\log_2 m_1 m_2 \cdots m_r)$ 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 they are not.

Our solution is to use factor refinement to preprocess the moduli $\{m_1, m_2, \ldots, m_r\}$ and convert them to a set of relatively prime numbers $\{{n_1}^{e_1}, {n_2}^{e_2}, \ldots, {n_s}^{e_s}\}$ such that there exist $f_1, f_2, \ldots, f_s$ with

$$\operatorname{lcm}(m_1, m_2, \ldots, m_r) = \prod_{1 \leq i \leq s} {n_i}^{f_i};$$

further, each m_i can be written as $\prod_{1 \leq j \leq s} {n_j}^{g_{ij}}$. 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 $\log_2 m_1 m_2 \cdots m_r$ processors.

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 apply Sleator and Tarjan's technique of amortized analysis [T] to prove the surprising fact that our factor refinement algorithm actually uses $O((\log m)^2)$ bit operations, asymptotically the same time required for a single gcd computation. This appears to be the first time amortized techniques have been used in the analysis of a number-theoretic algorithm.

We also present an interesting and valuable 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. As an application, we give algorithms to produce relatively prime squarefree factorizations and normal bases.

II. Previous work.

Dedekind appears to have been the first to study integer factorization by the repeated use of Euclid's algorithm [D2]. His 1897 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. The uniqueness of the output was proved by Kaltofen in 1985 [K1, K2], in the context of a general unique factorization domain. Kaltofen also gave a characterization of a fully refined factorization that is similar to our Theorem 3.

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 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 [B3] 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 an 1985 algorithm of Lüneburg [L3]. The latter method takes as input two elements a, b from some principal ideal domain, and finds another element r such that $r \mid a$, $\gcd(r, b) = 1$, 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 VII; he did not analyze its complexity.

III. The Factor Refinement Algorithm.

In this section, we describe an algorithm that successively refines a partial factorization $m = m_1 m_2 \ldots m_r$ into relatively prime pieces.

First, however, we 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 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 $O((\lg m/n)(\lg n))$ bit operations. From this, it is not difficult to show that we can compute $d = \gcd(m, n)$ in $O((\lg m/d)(\lg n))$ bit operations, when $m \ge n$.

Now we state our first factor refinement algorithm:

```
ALGORITHM Refine
INPUT: Positive integers $m_1, m_2, \ldots, m_r \ge 2$.
OUTPUT: List of pairs $L_f = \{(n_1, e_1), (n_2, e_2), \ldots, (n_s, e_s)\}$ such that
    (1) $\prod_{1 \le i \le s} {n_i}^{e_i} = m$;
    (2) $\gcd(n_i, n_j) = 1$ for all $i \ne j$;

comment The algorithm maintains a list $L$ of pairs $(n_i, e_i)$ such that $m = \prod_i {n_i}^{e_i}$.
initialize $n_i \leftarrow m_i$, $e_i \leftarrow 1$, for $1 \le i \le r$.
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 $L = \{(n_1, e_1), (n_2, e_2), \ldots, (n_s, e_s)\}$.
```

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 $m_1, m_2, \ldots m_r$ produces as output a list of pairs $L_f = \{(n_1, e_1), (n_2, e_2), \ldots, (n_s, e_s)\}$ such that*

(a) $\prod_{1 \leq i \leq s} {n_i}^{e_i} = m$;

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

(c) Each m_i is a multiplicative combination of the n_j; that is, there exist integers a_{ij}, $1 \leq i \leq r$, $1 \leq j \leq s$ such that $m_i = \prod_j {n_j}^{a_{ij}}$.

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 k refinement steps as L_k. Thus $L_0 = \{(m_1, 1), (m_2, 1), \ldots, (m_r, 1)\}$. We also write

$$L_k = \{(n_1, e_1)^{(k)}, (n_2, e_2)^{(k)}, \ldots, (n_s, e_s)^{(k)}\},$$

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

We claim that the sequence

$$S_k = \sum_i (e_i^{(k)} - 1) \tag{4}$$

is strictly increasing. To prove this, we show that $S_{k+1} - S_k > 0$. 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_k. 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_{k+1} is $2e_i + 2e_j - 3$, so $S_{k+1} - S_k = e_i + e_j - 1 \geq 1$. If one of m_i/d, m_j/d equals unity (let us say m_i/d), the contribution to S_{k+1} is $e_i + 2e_j - 2$, so $S_{k+1} - S_k = e_j \geq 1$. And finally, if both m_i/d and m_j/d equal unity, the contribution to S_{k+1} is $e_i + e_j - 1$, so $S_{k+1} - S_k = 1$.

Now it is clear that $S_k \leq \log_2 m$, 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 10 of Section VI that we can compute the a_{ij} such that $m_i = \prod_j {n_j}^{a_{ij}}$ using $O((\lg m)^2)$ bit operations.

We observe that the bound of $\log_2 m$ refinement steps is tight, as the algorithm performs r refinement steps on input $m_1 = 2^r$, $m_2 = 2$.

Corollary 2. *The 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

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

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 $p^a \mid m$ but $p^{a+1} \not| m$, then we say p^a divides m exactly, and we write $p^a \mid\mid m$. We now generalize this familiar concept to the case where m is not prime:

Definition.

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

(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 $L_f = \{(5,1),(6,2),(7,1)\}$, 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 $6 \mid\mid 30$ and $6 \mid\mid 42$ 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. We say that N divides M exactly (and write $N \mid\mid M$) if for all $m \in M$ and $n \in N$, there exists an $a \geq 0$ such that $n^a \mid\mid m$.

Example: $\{2,3,5,7\} \mid\mid \{5,6,7\}$.

Note that $\mid\mid$ induces a partial order on sets of integers.

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

Theorem 3. *Let the input to Refine be $M = \{m_1, m_2, \ldots, m_r\}$. Then the output of Refine is the unique set of pairs $L_f = \{(n_1, e_1), \ldots, (n_s, e_s)\}$ such that*

(a) $\prod_{1 \leq i \leq s} {n_i}^{e_i} = m$, and $n_i > 1$.

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

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

(d) N is maximal (for the ordering defined by $\mid\mid$) 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 (c) holds, it suffices to observe that by Theorem 1, we have $m_i = \prod_j {n_j}^{a_j}$. Hence ${n_k}^{a_k} \mid m_i$, and since the n_j are pairwise relatively prime, we have ${n_k}^{a_k} \mid\mid m_i$.

Now let us prove that N is maximal. Let R be any set of integers satisfying (a)-(c). Let L_k denote, as above, the list of pairs L after the k-th refinement step of the algorithm, and let $S(L_k)$ denote the set of first entries of the pairs in L_k. Thus $N = S(L_f)$.

We will show by induction on k that $R \mid\mid S(L_k)$. This is true for $k = 0$ by assumption. Now assume that $R \mid\mid S(L_k)$; we need to show that $R \mid\mid S(L_{k+1})$. 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 $R \mid\mid S(L_k)$, for any $r \in R$ there exist a, b such that $r^a \mid\mid n_i$ and $r^b \mid\mid n_j$. Therefore, $r^{\min(a,b)} \mid\mid \gcd(n_i, n_j) = d$. From this we see $r^{a-\min(a,b)} \mid\mid n_i/d$ and $r^{b-\min(a,b)} \mid\mid n_j/d$. Thus $R \mid\mid S(L_{k+1})$. Thus we see $R \mid\mid S(L_f) = N$, so N is maximal.

To see N is unique, assume there is another set T with properties (a)-(d). Then $N \mid\mid T$ and $T \mid\mid N$. Let $n \in N$. There must be a $t \in T$ with $\gcd(n, t) \neq 1$, for by (a) we have

$$\prod_{1 \leq j \leq k} {t_j}^{f_j} = m = \prod_{1 \leq i \leq s} {n_i}^{e_i}.$$

Then there exist $b, c > 0$ such that $n^b \mid\mid t$ and $t^c \mid\mid n$. Then $b = c = 1$ and $n = t$. Thus $N = T$. ■

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_j, n_k) 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: Positive integers $m_1$, $m_2$.

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 $i + 1$st 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.

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 1; hence we may conclude that all the n_i 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 k 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 $r \le i-1$ and $r > i+2$. Thus the result is true after $k+1$ refinement steps as well. ■

After we discovered the results in Sections V and VI below, H. W. Lenstra showed us the following intuitive argument that refining the factorization $m = m_1 m_2$ can be done in $O((\log m)^2)$ time:

Suppose we refine $m = m_1 m_2$ using the Pair-Refine algorithm above. We first write $m = (m_1/d)d^2(m_2/d)$ and then proceed to completely refine the leftmost pair, $(m_1/d), d$. After this is done, we are left with a new d, say $d' \le d$, and we refine $d', m_2/d$. Thus 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 $d'(m_2/d) \le m_2$, and the cost of a gcd. (We ignore the cost of the divisions, as they can be subsumed in the gcd cost.) Hence we find

$$T(m) \le T(m_1) + T(m_2) + c(\lg m_1)(\lg m_2).$$

If we replace this inequality with the equation,

$$T(m) = T(m_1) + T(m_2) + c(\log m_1)(\log m_2),$$

then we find the solution $T(n) = (c/2)(\log n)^2$.

Some more details are required to make this argument precise: for example, we must replace the $c(\log m_1)(\log m_2)$ term with $c(\lg m_1)(\lg m_2)$.

Also, this simple and attractive proof does not seem to generalize easily to more than 2 factors. In the next section, we will prove the $O((\log m)^2)$ time bound using amortized analysis.

V. Factor Refinement with 2 Inputs: Amortized Analysis.

Definition. Let c_{div} and c_{gcd} be constants such that the number of bit operations needed to divide a by b for $a \ge b > 1$ is no more than $c_{\text{div}}(\log a - \log b + 1)\log b$, and the number of bit operations required to compute $d = \gcd(a, b)$ for $a \ge b > 1$ is no more than $c_{\text{gcd}}(\log a - \log d + 1)\log b$.

In this section, we use the techniques of amortized analysis to prove a quadratic running time bound for the algorithm Pair-Refine.

The basic idea of the proof is to find a metric for the amount of progress that a single refinement step makes, and then show that the cost of a single refinement step is not out of line with the amount of progress made.

One simple minded metric is the total number of bits that remain to be refined, which is roughly $\sum_{i=1}^{s} \log n_i$. A single refinement step on the pair (n_i, e_i) and (n_{i+1}, e_{i+1}) costs roughly $(\log n_i - \log d)\log n_{i+1}$ (for $n_i \geq n_{i+1}$). Since n_i and n_{i+1} are replaced by n_i/d and n_{i+1}/d, and the new pair $(d,1)$ is added, the number of bits decreases by $\log d$. If d is "big", that is $d^2 \geq n_{i+1}$, the cost of the step is $O(l \log d)$, where $l = \log m_1 + \log m_2$. If all gcd's are big, then the total cost of the refinement would be $O(l \sum_i \log d_i)$ where d_i is the gcd at the ith refinement step. Since the number of bits remaining decreases by $\log d$ at each step, it follows that the $\sum_i \log d_i$ is at most l. Thus, this argument shows that the refinement takes $O(l^2)$ time, provided all the gcd's are big.

If, however, a gcd is small, then the time for the refinement step is not at all in proportion to the progress made, if progress is measured as the number of bits remaining. (Consider a gcd of 1 which makes no progress at all, but can have arbitrarily large cost.) If all the gcd's are small, then the cost of each step could be $O(l^2)$, an unpleasant prospect. However, the result of finding a small gcd is to place a small number between two large numbers. This improves the situation, since now we must twice refine a large number and a small number, whereas previously we would have had to refine two large numbers. Although we have not substantially reduced the *size* of the problem, we have broken up the problem into two more manageable pieces. Thus it seems natural to include in our measure of progress the *difference of the number of bits* of adjacent n_i in the list. If we take this difference so that it is always ≤ 0, then this measure of progress is 0 when all the n_i are the same. When we are done, all $n_i \neq 0$ are adjacent to a 1, and the measure is at a minimum. If a refinement step on (n_i, e_i) and (n_{i+1}, e_{i+1}) has a small gcd, then this new measure of progress will decrease by $O(\log n_i)$ and the cost for the gcd is $O(l \log n_i)$. Thus, if all the gcd's were small, the total cost would be $O(l)$ times the sum of the decreases in the measure. But the measure starts at no more that 0, and ends at no less that $-2l$ (since each element in the list differs in bit size from its two neighbors by at most twice its own bit size). Thus, the total cost of the refinement, provided all gcd's are small, is $O(l^2)$.

We now combine and make precise the preceding intuitive arguments.

Theorem 5. *If $l = \log m_1 + \log m_2$, then Pair-Refine uses $O(l^2)$ bit operations.*

Proof. We measure the progress after k refinement steps by

$$\Phi_k = -c_1(l+1)\sum_{i=1}^{s-1}\left|\log\frac{n_i^{(k)}}{n_{i+1}^{(k)}}\right|,$$

where $2c_1 = c_{\text{gcd}} + 2c_{\text{div}}$. Let the *amortized* cost of the kth refinement step, A_k, be its actual cost (in bit operations) C_k, plus the increase in the "potential function" Φ. The sum of the amortized costs of the refinement steps, $\sum_{k=1}^{f} A_k$, is $\sum_{k=1}^{f}(C_k + (\Phi_k - \Phi_{k-1}))$, which is $\left(\sum_{k=1}^{f} C_k\right) + \Phi_f - \Phi_0$, the total cost plus the net increase in potential. Thus we conclude that the total cost is $\left(\sum_{k=1}^{f} A_k\right) + \Phi_0 - \Phi_f$. The initial potential Φ_0 is at most 0 and the final potential Φ_f is at least $-2c_1(l+1)l$ since every n_i contributes at most $2\log n_i$ to $\sum_{i=1}^{s-1}\left|\log(n_i^{(f)}/n_{i+1}^{(f)})\right|$ and $\sum_{i=1}^{s}\log n_i \leq l$. Thus

$$\sum_{k=1}^{f} C_k \leq \sum_{k=1}^{f} A_k + 2c_1(l+1)l$$

and therefore to show that Pair-Refine has $O(l^2)$ running time, it suffices to show that the sum of the amortized costs is $O(l^2)$.

We now calculate the change in potential and the actual cost of a single refinement step in order to determine the amortized cost of a step. Suppose the $k+1$st step refines $n_i^{(k)}$ and $n_{i+1}^{(k)}$ (neither of which can be 1). Then the change in potential due to the refinement step is

$$\begin{aligned}\Phi_{k+1} - \Phi_k = &- c_1(l+1)\left(\left|\log \frac{dn_{i-1}^{(k)}}{n_i^{(k)}}\right| + \left|\log \frac{n_i^{(k)}}{d^2}\right| + \left|\log \frac{d^2}{n_{i+1}^{(k)}}\right| + \left|\log \frac{n_{i+1}^{(k)}}{dn_{i+2}^{(k)}}\right|\right) \\ &+ c_1(l+1)\left(\left|\log \frac{n_{i-1}^{(k)}}{n_i^{(k)}}\right| + \left|\log \frac{n_i^{(k)}}{n_{i+1}^{(k)}}\right| + \left|\log \frac{n_{i+1}^{(k)}}{n_{i+2}^{(k)}}\right|\right),\end{aligned}$$

where $d = \gcd(n_i^{(k)}, n_{i+1}^{(k)})$, except when $i = 0$ or $i = s-1$, when the terms involving n_0 and n_{s+1} contribute nothing to the change in potential. From the triangle inequality we see

$$\left|\log \frac{a}{b}\right| - \left|\log \frac{da}{b}\right| \le \log d,$$

and hence

$$\Phi_{k+1} - \Phi_k \le c_1(l+1)\left(\left|\log \frac{n_i^{(k)}}{n_{i+1}^{(k)}}\right| - \left|\log \frac{n_i^{(k)}}{d^2}\right| - \left|\log \frac{d^2}{n_{i+1}^{(k)}}\right| + 2\log d\right).$$

The actual cost for the step is the cost of the gcd plus the two divisions. Because neither $n_i^{(k)}$ nor $n_{i+1}^{(k)}$ are 1, the actual cost is (without loss of generality, $n_i^{(k)} \ge n_{i+1}^{(k)}$)

$$\begin{aligned}c_{\mathrm{gcd}}(\log n_i^{(k)} - \log d + 1)\log n_{i+1}^{(k)} + c_{\mathrm{div}}(\log n_i^{(k)} - \log d + 1)\log d \\ + c_{\mathrm{div}}(\log n_{i+1}^{(k)} - \log d + 1)\log d\end{aligned}$$

$$\begin{aligned}&\le (c_{\mathrm{gcd}} + 2c_{\mathrm{div}})(l+1)\log n_{i+1}^{(k)} \\ &= 2c_1(l+1)\log n_{i+1}^{(k)}.\end{aligned}$$

(The time to add the exponents is not considered here, since the sum of the exponents is no more than l and thus all additions are easily accomplished in $O(l^2)$ time. Nor is the time to maintain the list considered, since its length is $O(l)$ and there are $O(l)$ refinement steps. To see that the length of the list and the number of refinement steps are both $O(l)$, observe that for every step with gcd $d > 1$, $\sum_i \log n_i$ is reduced by $\log d$. Also, there are at most l elements of the list that are greater than 1, hence there are at most $l+1$ steps with the gcd equal to 1.)

Next we will show that the amortized time for each refinement step, (i.e. the actual cost plus the increase in potential), is no more than $6c_1(l+1)\log d_k$, where d_k is the gcd at the kth refinement step. There are three cases according to the relative magnitudes of

$d = d_{k+1}$, $n_i^{(k)}$, and $n_{i+1}^{(k)}$. We may assume that $n_i^{(k)} \geq n_{i+1}^{(k)}$, since reversing the list does not affect the potential.

Case I $\left(n_i^{(k)} \geq d^2 \geq n_{i+1}^{(k)}\right)$*:* The change in Φ can be seen to be

$$\begin{aligned}\Phi_{k+1} - \Phi_k &\leq c_1(l+1)\left(\log \frac{n_i^{(k)}}{n_{i+1}^{(k)}} - \log \frac{n_i^{(k)}}{d^2} - \log \frac{d^2}{n_{i+1}^{(k)}} + 2\log d\right) \\ &= 2c_1(l+1)\log d.\end{aligned}$$

Thus the amortized cost, the actual cost plus the change in potential, is at most

$$2c_1(l+1)\log n_{i+1}^{(k)} + 2c_1(l+1)\log d \leq 6c_1(l+1)\log d,$$

since $2\log d \geq \log n_{i+1}^{(k)}$ because $d^2 \geq n_{i+1}^{(k)}$.

Case II $\left(d^2 \geq n_i^{(k)} \geq n_{i+1}^{(k)}\right)$*:* In this case the change in Φ is

$$\begin{aligned}\Phi_{k+1} - \Phi_k &\leq c_1(l+1)\left(\log \frac{n_i^{(k)}}{n_{i+1}^{(k)}} - \log \frac{d^2}{n_i^{(k)}} - \log \frac{d^2}{n_{i+1}^{(k)}} + 2\log d\right) \\ &= c_1(l+1)(2\log n_i^{(k)} - 2\log d) \\ &\leq 2c_1(l+1)\log d,\end{aligned}$$

since $2\log d \geq \log n_i^{(k)}$. As in Case I, the amortized cost is no more than $6c_1(l+1)\log d$ since $d^2 \geq n_{i+1}^{(k)}$.

Case III $\left(n_i^{(k)} \geq n_{i+1}^{(k)} \geq d^2\right)$*:* The change in Φ is

$$\begin{aligned}\Phi_{k+1} - \Phi_k &\leq c_1(l+1)\left(\log \frac{n_i^{(k)}}{n_{i+1}} - \log \frac{n_i^{(k)}}{d^2} - \log \frac{n_{i+1}^{(k)}}{d^2} + 2\log d\right) \\ &= -\,2c_1(l+1)\log n_{i+1}^{(k-1)} + 6c_1(l+1)\log d.\end{aligned}$$

Adding in the actual cost of $2c_1(l+1)\log n_{i+1}^{(k)}$ shows that the amortized cost is at most $6c_1(l+1)\log d$.

Thus, the sum of the amortized costs is at most $\sum_{k=1}^{f} 6c_1(l+1)\log d_k$. But $\sum_{k=1}^{f} \log d_k \leq l$ since initially $\sum_{i=1}^{s} \log n_i = l$, the kth refinement step reduces $\sum_{i=1}^{s} \log n_i$ by $\log d_k$ (since $(\log n_i + \log n_{i+1}) - (\log(n_i/d) + \log d + \log(n_{i+1}/d)) = \log d$), and $\sum_{i=1}^{s} \log n_i \geq 0$. Thus, if the amortized cost for the kth step is $6c_1(l+1)\log d_k$, then the sum of the amortized costs is $\leq 6c_1 l(l+1)$ and hence the total cost of the algorithm is $\leq 8c_1 l(l+1) = O(l^2)$. ■

VI. Factor Refinement: Amortized Analysis of the General Case.

Now we consider the general case of computing a refinement of a product of i numbers. The refinement will be computed by taking a refinement of $m_1 m_2 \cdots m_j$ and a new m_{j+1} as input and producing a refinement of $m_1 m_2 \cdots m_{j+1}$. We call this step an *augmenting step*. The augmenting step will be carried out as follows:

```
ALGORITHM Augment-Refinement
INPUT: $m_{j+1}$,  $L_j = (n_1, e_1), \ldots, (n_s, e_s)$,
    the refinement of $m_1, m_2, \ldots, m_j$
OUTPUT: The refinement of $m_1, m_2, \ldots m_{j+1}$

initialize $(m, e) \leftarrow (m_{j+1}, 1)$, $L_{j+1} \leftarrow$ empty list
while $L_j$ not empty and $m \neq 1$ do
    begin
    if First($L_j$) has first component unequal to 1 then
        begin
        $L' \leftarrow$ Pair-Refine($(m, e)$, First($L_j$))
        $L_{j+1} \leftarrow$ Concat($L_{j+1}$, Rest($L'$))
        $(m, e) \leftarrow$ First($L'$)
        end
    $L_j \leftarrow$ Rest($L_j$)
    end;
$L_{j+1} \leftarrow$ Concat($L_{j+1}$, Rest($L_j$), $(m, e)$)
output List of pairs $(n_i, e_i) \in L_{j+1}$ with $n_i \neq 1$
```

Lemma 6. *Algorithm Augment-Refinement, given a refinement of $m_1 m_2 \cdots m_j$ and m_{j+1} correctly computes the refinement of $m_1 m_2 \cdots m_j m_{j+1}$.*

Proof. Adding $(m_{j+1}, 1)$ to the existing refinement produces a refinement of

$$m_1 m_2 \cdots m_j m_{j+1};$$

however, not all the pairs are relatively prime. Pairwise refining $(m_{j+1}, 1)$ and some other pair (n_i, e_i) will produce a list of pairs that are relatively prime within this list, but they are not all necessarily relatively prime with the remaining pairs. Observe that if we pairwise refine $(m_{j+1}, 1)$ and (n_i, e_i), then only the first element of the resulting list will not divide n_i. Since n_i is relatively prime to all of the other pairs of the original refinement of $m_1 m_2 \cdots m_j$, it follows that all but that first element from the pairwise refinement of $(m_{j+1}, 1)$ and (n_i, e_i) are relatively prime to $n_1, n_2, \ldots, n_s$. Thus only that first pair needs to be refined with the remaining pairs of the original refinement. Continuing this process of pairwise refining the first element of the last pairwise refinement with one of the remaining elements gives a refinement with all of the n_i relatively prime. ■

As in the case of refining a pair, there is an $O(l^2)$ bound for the general case. The main obstacle to the proof is the possibility of repeatedly pairwise refining elements of greatly differing magnitude. Refining $n_1 n_2$, where $n_1 \gg n_2$ can require $\Omega((\log n_1)^2)$ bit operations (consider the case of 2^r and 2). However, we will see that the cost of refining such a pair in an amortized sense is $O((\log n_1)(\log n_2))$, for $n_1, n_2 \neq 1$.

Lemma 7. *Exclusive of the refinement steps on the first and last pairs of the list, the algorithm Pair-Refine on input* (m_1, e_1) *and* (m_2, e_2) *uses* $O((\log m_1)(\log m_2))$ *bit operations.*

Proof. Choose

$$\Phi_k = -c_1 \min\{1 + \log m_1, 1 + \log m_2\} \sum_{i=2}^{s-2} \left| \log \frac{n_i^{(k)}}{n_{i+1}^{(k)}} \right| .$$

Since all refinement steps that do not involve the first or the last pair will have $n_i < \min\{m_1, m_2\}$, the argument from Theorem 5 shows us that the amortized cost of a refinement step that doesn't involve the first or last pair will be at most

$$6c_1 \min\{1 + \log m_1, 1 + \log m_2\} \log d_i,$$

where d_i is the gcd computed for the refinement step. But $\sum_i \log d_i$ is at most $\min\{\log m_1, \log m_2\}$ since $\sum_{i=2}^{s-1} \log n_i$ is at most $\min\{\log m_1, \log m_2\}$ after the initial refinement of m_1 and m_2 (unless m_1 and m_2 are relatively prime, when the total cost is 0), each refinement step on pairs other than the first or last reduces $\sum_{i=2}^{s-1} \log n_i$ by $\log d_i$ (the gcd at that step), each refinement step on the first or the last pair leaves $\sum_{i=2}^{s-1} \log n_i$ unaffected, and when the refinement is complete $\sum_{i=2}^{s-1} \log n_i \geq 0$. Thus the sum of the amortized costs of the refinement steps that do not involve the first or last pair is $O(\min\{1 + \log m_1, 1 + \log m_2\}^2)$.

We now consider the amortized cost of refinement steps on the first and last pairs, because these may increase the potential even though their actual cost is not counted here. Let $n_i^{(k)}$ be the ith list element after k refinement steps. Then the increase in potential due to refining the first pair at the kth step is seen to be

$$\begin{aligned}\Phi_{k+1} - \Phi_k = & - c_1 \min\{1 + \log m_1, 1 + \log m_2\} \left(\left| \log \frac{d^2}{n_2^{(k)}} \right| + \left| \log \frac{n_2^{(k)}}{d n_3^{(k)}} \right| \right) \\ & + c_1 \min\{1 + \log m_1, 1 + \log m_2\} \left(\left| \log \frac{n_2^{(k)}}{n_3^{(k)}} \right| \right),\end{aligned}$$

where $d = \gcd(n_1^{(k)}, n_2^{(k)})$. An easy calculation shows that this sum is at most $c_1 \min\{1 + \log m_1, 1 + \log m_2\} \log d$. Since the sum of the logs of the gcd's from refinement steps on the first two list elements is at most $\max\{\log m_1, \log m_2\}$, the total amortized cost of refinement steps on the first two list elements is no more than $c_1 \min\{1 + \log m_1, 1 + \log m_2 +$

$1\}\max\{\log m_1, \log m_2\} = O((\log m_1)(\log m_2))$. The same is true for the amortized cost of refinement steps on the last two list elements.

Adding the amortized costs of the steps not involving the first and last pairs to the amortized costs of the steps involving the first and last pairs, we find that the total amortized cost of the refinement is

$$O(\min\{1+\log m_1, 1+\log m_2\}^2 + \max\{\log m_1, \log m_2\}\min\{1+\log m_1, 1+\log m_2\}$$

$$= O((\log m_1)(\log m_2)).$$

The initial potential is 0 and the final potential is $-O(\min\{\log m_1, \log m_2\}^2)$. Thus the total actual cost of all operations not involving the first and last pairs is $O((\log m_1)(\log m_2))$. ■

We now extend the above analysis to include the refinement steps on the first and last pairs, showing that the amortized cost of a single pair refinement in a sequence of such refinements is $O((\log m_1)(\log m_2))$. In this case we choose a somewhat different potential function given by

$$\Phi' = c_2 \sum_i (\log n_i)^2,$$

where c_2 is a constant to be chosen later.

Lemma 8. *There exists a constant c_2 such that for the potential function Φ' the amortized cost of a pair refinement of (m_1, e_1) and (m_2, e_2), for $m_1, m_2 \neq 1$, is $O((\log m_1)(\log m_2))$.*

Proof. From the previous lemma, the actual cost of the pair refinement is $O((\log m_1)(\log m_2))$ plus the actual cost for operations involving the first and last pairs.

The initial refinement step involving both m_1 and m_2 has actual cost $O((\log m_1)(\log m_2))$. We now consider the actual cost of subsequent refinement steps on n_1 (respectively, n_s).

Let d_k be the gcd for the kth (for $k > 1$) refinement step involving n_1. We have $n_1 \le m_1/d_{k-1}$ and $n_2 \le d_{k-1}$. Thus, the actual cost of the kth refinement step involving n_1 is

$$O((\log \frac{m_1}{d_{k-1}})(\log d_{k-1})) = O((\log m_1)(\log d_{k-1}) - (\log d_{k-1})^2).$$

And thus the total cost is

$$\begin{aligned} O(\sum_{k>1}(\log m_1)(\log d_{k-1}) - \sum_{k>1}(\log d_{k-1})^2) &= O(\log m_1 \sum_{k>1} \log d_{k-1} - \sum_{k>1}(\log d_{k-1})^2) \\ &= O((\log m_1)(\log \frac{m_1}{n_1^{(f)}}) - \sum_{k>1}(\log d_{k-1})^2) \\ &= O((\log m_1)^2 - (\log m_1)(\log n_1^{(f)}) - \sum_{k>1}(\log d_{k-1})^2) \\ &= O((\log m_1)^2 - (\log n_1^{(f)})^2 - \sum_{k>1}(\log d_{k-1})^2), \end{aligned}$$

where $n_1^{(f)}$ is the final value of n_1. The second line follows because $n_1^{(f)} \prod_k d_k = m_1$.

We now calculate the amortized costs both for those refinement that do not involve the first or last element and for those that do.

For those steps that do not involve the first or last element the actual cost has been seen to be $O((\log m_2)(\log m_2))$. Each of these operations decreases Φ' since

$$(\log n_i)^2 + (\log n_{i+1})^2 \geq (\log \frac{n_i}{d})^2 + (\log d)^2 + (\log \frac{n_{i+1}}{d})^2.$$

Thus the amortized cost of these operations is $O((\log m_1)(\log m_2))$ and hence it suffices to show that the amortized cost of the refinement steps on n_1 and n_s is $O((\log m_1)(\log m_2))$.

For each refinement step involving the first pair, the change in Φ' is due to the addition of the new d_i in the list and the decrease to n_1. Thus the net change to Φ' from refinement steps involving n_1 is

$$\begin{aligned} &c_2\left(\sum_{i>1}(\log d_i)^2 - (\log m_1)^2 + (\log n_1^{(f)})^2\right) \\ \leq &c_2\left(\sum_{i>1}(\log d_{i-1})^2 - (\log m_1)^2 + (\log n_1^{(f)})^2\right) \end{aligned}$$

since $d_{i-1} \geq d_i$. If we choose c_2 to be the asymptotic constant from the actual cost computed above, we see that the amortized cost for the operations involving n_1 other than the initial refinement step is ≤ 0. The same is true for n_s.

Thus, the amortized cost for a single pair refinement is $O((\log m_1)(\log m_2))$ for the first refinement step, $O((\log m_1)(\log m_2))$ for the refinement steps not involving n_1 or n_s, and ≤ 0 for the remaining steps involving n_1 and n_s. The total amortized cost is $O((\log m_1)(\log m_2))$. ■

Theorem 9. *Calculating a refinement of $m = m_1 m_2 \cdots m_l$ by repeated augmentation uses $O((\log m)^2)$ bit operations.*

Proof. First we show that augmenting a refinement of $m_1 m_2 \cdots m_i$ to a refinement of $m_1 m_2 \cdots m_i m_{i+1}$ has an amortized cost, relative to Φ', of

$$O((\sum_{j<i+1} (\log m_{i+1})(\log m_j)) + (\log m_{i+1})^2).$$

Before the augmentation step let $n_1^{e_1} n_2^{e_2} \cdots n_s^{e_s}$ be the refinement of $m_1 m_2 \cdots m_i$. Note that $\prod_{j=1}^{s} n_j \leq \prod_{j=1}^{i} m_j$. In the augmentation step, m_{i+1} will be pairwise refined with n_1, then what remains of m_{i+1} will be pairwise refined with n_2, and so on. From Lemma 8 the amortized cost of this augmentation will thus be

$$O((\sum_{j=1}^{s} \log m_{i+1} \log n_j) + (\log m_{i+1})^2) = O((\log m_{i+1} \sum_{j=1}^{i} \log m_j) + (\log m_{i+1})^2).$$

(The $(\log m_{i+1})^2$ comes from the change in Φ' due to the addition of m_{i+1}.)

Thus the total amortized cost of the $l-1$ augmentations is

$$\begin{aligned}
&O(\sum_{j=2}^{l}(\log m_j \sum_{k=1}^{j-1} \log m_k) + \sum_{j=1}^{l}(\log m_j)^2)\\
&= O(\sum_{j\neq k}(\log m_j)(\log m_k) + \sum_{j=1}^{l}(\log m_j)^2)\\
&= O((\sum_{j=1}^{l}\log m_j)^2)\\
&= O((\log m)^2).
\end{aligned}$$

Since the initial potential is 0 and the final potential is > 0, the total actual cost is bounded above by the sum of the amortized costs. Thus refining by repeated augmentation uses $O((\log m)^2)$ bit operations. ■

The final result of this section shows that once a refinement of $m = m_1 m_2 \cdots m_i$ is computed, we can express each m_i as a product of the n_j in quadratic time.

Theorem 10.

Given the result of factor refinement, we can express each $m_i = \prod_j {n_j}^{a_{ij}}$ using $O((\lg m)^2)$ bit operations.

Proof.

Using factor refinement, we have expressed $m = \prod_i {n_i}^{e_i}$. Now write out the n_i's in a list L, each with multiplicity e_i. For each m_j, we do the following: trial divide by n_i; if a zero-remainder is encountered, we replace m_j with m_j/n_i, remove n_i from L, and continue to trial divide by n_{i+1}. Since $\prod_j m_j = \prod_i n_i$, all the n_i's are used up when we are done. The total time is

$$\sum_j \sum_i (\lg m_j)(\lg n_i) = O((\lg m)^2).$$

■

VII. 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 9. 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 with an irreducible monic polynomial with coefficients in $GF(p)$, and 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].

Let a and b be monic polynomials, with $\deg a \geq \deg b > 0$. We assume that $g = \gcd(a, b)$ – which we also assume to be monic – can be computed with $O(\deg(a/g) \deg b)$ operations in k, and that if $b \mid a$, then a/b may be computed with with $O(\deg(a/b) \deg b)$ operations. These assumptions are true if one uses the Euclidean algorithm to find the greatest common divisor, and 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 the algorithm Refine works as stated for polynomials in $k[X]$, and Theorem 1 is true in this case; here all references to "integers" must be changed to "polynomials." The proof of Theorem 1 will show 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.

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

Theorem 11. *Let m_1 and m_2 be monic polynomials in $k[X]$, of degrees d_1 and d_2, with $d_1, d_2 > 0$. Let $d = d_1 + d_2$. Then Pair-Refine uses $O(d^2)$ operations in k.*

Proof (sketch). This is proved with the same calculation as Theorem 5, except for the following modification: everywhere the logarithm of a number appears, it should be replaced by the degree of a polynomial. For example, the potential function is defined as follows. Let $f_1^{(k)}, \ldots, f_s^{(k)}$ be the polynomials occurring in the list of pairs after k refinement steps. Then the potential function at this stage of the algorithm is

$$\Phi_k = -c_1 d \sum_{i=1}^{s-1} |\deg f_i^{(k)} - \deg f_{i+1}^{(k)}|.$$

■

The similarity of this result to Theorem 5 may be strengthed 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 $\log f$. More precisely, this is

$$\log f = (\deg f)(\log |k|).$$

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 12. *Let m_1 and m_2 be monic polynomials in $k[X]$, of positive degree. If $l = \log m_1 + \log m_2$, then Pair-Refine uses $O(l^2)$ bit operations.*

By similarly replacing logarithms by degrees in the appropriate potential functions, one can prove polynomial analogs of Lemmas 6-8. These lemmas then imply the following result:

Theorem 13. *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 by repeated augmentation using $O((\sum \deg m_i)^2)$ operations in k. Hence this can be done with $O((\log m)^2)$ bit operations.*

Proof. Similar to that of Theorem 9; left to the reader. ■

We now use factor refinement to solve two problems: squarefree decomposition, 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 [B4]) 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 this preprocessing may be done essentially in quadratic time. We note that no known polynomial factorization algorithm runs in quadratic time, although there are some that approach this bound asymptotically [B2,S].

Theorem 14. *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 $O((\log f)^2)$ 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 get 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 $|k| = p^n$.

In positive characteristic, the running time bound of Theorem 14 appears to be new. Musser [M] gave a squarefree decomposition algorithm for characteristic p, but he did not analyze it. (For other polynomial time algorithms, see [D1], [G1], [K5].) In retrospect, though, a quadratic time bound follows easily from the work of Yun [Y], who gave a squarefree decomposition algorithm for polynomials over a field of characteristic 0.

As stated, Yun's algorithm does not work in characteristic p, but it may be easily modified so as 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 our extension of his algorithm to $GF(p)[X]$.

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, L3]. (We are unaware of any bounds in the literature more precise than this, although both methods cited have can be shown to have the same complexity as ours.)

The standard construction of a normal basis for a finite field [J, 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 so as to replace complete factorization by relatively prime factorization, then a polynomial time algorithm also results.

Theorem 15. *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

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

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]. In general, this is exponentially small. 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 $1/2$. The entire computation, including computation of a pth power matrix, uses $O((n + \log p)(\log |k|)^2)$ bit operations.

Finally, we discuss how factor refinement solves a problem discussed by Lüneburg [L3].

Theorem 16. *Let k be a finite field, and let $a, c \in k[X]$. Let $l = \log a + \log c$. Then we can find a polynomial r such that $r \mid a$, $\gcd(r, c) = 1$, and each prime divisor of a/r divides c, with $O(l^2)$ bit operations.*

Before giving the proof 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.

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

$$r = \prod_{e_i'=0} g_i^{e_i}.$$

■

Evidently, Theorem 16 holds for integers as well.

VIII. Acknowledgments.

We would like to thank E. Kaltofen for bringing the references [G2], [K1], [K2], [K3], [K4], [W2], and [Y] to our attention.

H. W. Lenstra generously shared with us some proofs and applications relating to factor refinement.

Part of this work was done while the third author was 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., South Bend, Univ. of Notre Dame Press, 1966.

[B1] E. Bach, G. Miller, and J. Shallit, Sums of divisors, perfect numbers, and factoring, *Proc. 16th ACM Symp. Theor. Comput.* (1984), 183-190. (Revised version appeared in *SIAM J. Comput.* **15** (1986), 1143-1154.)

[B2] M. Ben-Or, Probabilistic algorithms in finite fields, *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science* (1981), 394-398.

[B3] M. Ben-Or, D. Kozen, and J. Reif, The complexity of elementary algebra and geometry, *Proc. 16th ACM Symp. Theor. Comput.* (1984), 457-464. (Revised version appeared in *J. Comput. Sys. Sci.* **32** (1986), 251-264.)

[B4] E.R. Berlekamp, Factoring polynomials over large finite fields, *Math. Comp.* **24** (1970), 713-735.

[C1] G. E. Collins, Computing time analyses for some arithmetic and algebraic algorithms, *Proc. 1968 Summer Institute on Symbolic Mathematical Computations*, IBM Federal Systems Center, 1968, 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, *Proc. 2nd GI Conf.*, Lecture Notes in Computer Science #33 (1975), 134-183.

[D1] J. Davenport, On the integration of algebraic functions, Lecture Notes in Computer Science #102, Springer-Verlag, 1981.

[D2] R. Dedekind, Über Zerlegungen von Zahlen durch ihre größten gemeinsamen Teiler, *Gesammelte mathematische Werke*, Vol. II, F. Vieweg & Sohn, Braunschweig, 1932, 104-147.

[E] H. Epstein, Using basis computation to determine pseudo-multiplicative independence, *Proc. 1976 ACM Symp. Symb. Alg. Comput.,* (1976), 229-237.

[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] C. F. Gauss, Disquisitiones Arithmeticae, Springer-Verlag, 1986.

[J] N. Jacobson, Lectures in Abstract Algebra, Vol. III: Theory of Fields and Galois Theory, Van Nostrand, 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, *Proc. EUROCAL '85*, Lecture Notes in Computer Science #204 (1985), 4-17.

[K3] E. Kaltofen, M. S. Krishnamoorthy, and B. D. Saunders, Fast parallel algorithms for similarity of matrices, *SYMSAC '86: Proc. 1986 ACM Symp. Symb. Alg. Comp.* (1986), 65-70.

[K4] E. Kaltofen, M. S. Krishnamoorthy, and B. D. Saunders, Parallel algorithms for matrix normal forms, preprint.

[K5] D.E. Knuth, The Art of Computer Programming, Vol. 2: Seminumerical Algorithms, Addison-Wesley, 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, manuscript (1989).

[L3] H. Lüneburg, On a little but useful algorithm, *Proc. AAEEC-3*, Lecture Notes in Computer Science, #229 (1985), 296-301.

[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, Mathematisch Centrum, Amsterdam, 1982, pp. 89-139.

[S] V. Shoup, On the deterministic complexity of factoring polynomials over finite fields, *Info. Proc. Lett.*, to appear. (Also appeared as: Computer Sciences Technical Report #782, University of Wisconsin, July 1988.)

[T] R. E. Tarjan, Amortized computational complexity, *SIAM J. Appl. Discrete Meth.* **6** (1985), 306-318.

[W1] B. L. van der Waerden, Algebra, Vol. I, 7th ed., New York, Ungar, 1970.

[W2] P. S. Wang, The EEZ-GCD algorithm, *SIGSAM Bulletin* **14** (2) (1980), 50-60.

[Y] D. Y. Y. Yun, Fast algorithm for rational function integration, *Info. Processing 77*, North-Holland, 493-498.