1986

Methods of Factoring Large Integers

Samuel S. Wagstaff Jr
Purdue University, ssw@cs.purdue.edu

J. W. Smith

Report Number:
86-585

Wagstaff, Samuel S. Jr and Smith, J. W., "Methods of Factoring Large Integers" (1986). *Department of Computer Science Technical Reports*. Paper 504.
<https://docs.lib.purdue.edu/cstech/504>

METHODS OF FACTORING LARGE INTEGERS

Samuel S. Wagstaff, Jr.
and
J. W. Smith
University of Georgia

CSD-TR-585
March 1986

Methods of Factoring Large Integers

Samuel S. Wagstaff, Jr.*
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
and
J. W. Smith*
Department of Computer Science
University of Georgia
Athens, GA 30602

0. Introduction.

Several papers on factoring and primality testing justify these endeavors by quoting a passage from *Disquisitiones Arithmeticae* in which Gauss says that they are important and useful. We agree with Gauss and explain in Section 1 our views on the importance and usefulness of factoring.

Sections 2 and 3 describe two of the fastest general integer factoring algorithms, the continued fraction method and the quadratic sieve method. These are the two algorithms we considered in 1981 when we made our plans to build the first special processor for factoring large integers. In Section 4 we present this processor, which uses the continued fraction algorithm, and mention another processor now being designed to factor numbers via the quadratic sieve algorithm.

Section 6 describes Lenstra's elliptic curve algorithm. As a prelude to his method, in Section 5 we discuss briefly Pollard's $p - 1$ algorithm, which is older but closely related to Lenstra's method.

Section 7 summarizes the running times of the four factoring algorithms. Section 8 describes a special computer for the elliptic curve method.

Good general references for the factoring methods we discuss—and some we omit—are [14], [29] and [38].

* Work partially supported by NSF grants.

1. Why factor large integers?

There are many reasons why mathematicians and computer scientists want to factor integers.

The easiest way to compute arithmetic functions like $\phi(N)$, $\sigma(N)$, $d(N)$, etc., uses the factorization of N . For example, one has

$$\phi(N) = N \prod_{i=1}^k \left(1 - \frac{1}{p_i}\right) \text{ and } d(N) = \prod_{i=1}^k (1 + \alpha_i)$$

if $N = \prod_{i=1}^k p_i^{\alpha_i}$. There are a few recursion formulas like

$$\sum_{\substack{i \in \mathbb{Z} \\ \frac{3i^2+i}{2} \leq n}} (-1)^i \sigma\left(n - \frac{3i^2+i}{2}\right) = 0, \quad (1)$$

where $\sigma(n - n)$, if it appears, has the value n . For large n , this sum has about $2\sqrt{\frac{2n}{3}}$ terms.

Thus, $O(n^{\frac{3}{2}})$ steps are required to compute $\sigma(1), \sigma(2), \dots, \sigma(n)$ using this formula. Since you can factor N completely in $O(N^{\frac{1}{4}})$ steps, even with simple methods, you can compute $\sigma(1), \sigma(2), \dots, \sigma(n)$ in $O(n^{\frac{5}{4}})$ steps by factoring each N and then using the well-known formula for $\sigma(N)$ in terms of the prime factors of N . Not only is this method faster than using (1), it can compute $\sigma(N)$ for selected large N , which (1) cannot do.

Knowledge of factorization increases our understanding of the integers. For example, we may discover algebraic identities from examination of tables of factored numbers. Consider this extract from a table of factorizations of $(3^{3h} + 1)/(3^h + 1) = 3^{2h} - 3^h + 1$.

h	$3^{2h} - 3^h + 1$		
1	7	=	7
3	703	=	$19 \cdot 37$
5	58807	=	$7 \cdot 31 \cdot 271$
7	4780783	=	$7^2 \cdot 43 \cdot 2269$
9	387400807	=	$19441 \cdot 19927$

If we write these factorizations as

h	$3^{2h} - 3^h + 1$
1	$1 \cdot 7$
3	$19 \cdot 37$
5	$217 \cdot 271$
7	$2107 \cdot 2269$
9	$19441 \cdot 19927$

and consider the average and difference of the two factors on each line, we are led to the algebraic factorization

$$3^{2h} - 3^h + 1 = (3^h - 3^{\frac{h+1}{2}} + 1)(3^h + 3^{\frac{h+1}{2}} + 1), \quad (2)$$

which is valid for odd positive integers h . Note that the polynomial $x^{2h} - x^h + 1$ is not the product of two degree h polynomials in $\mathbb{Z}[x]$. The factorization (2) and others like it were discovered by Aurifeuille (see page 383 of [10]) about 100 years ago. We believe that new algebraic factorizations may be discovered by study of tables of factored integers like [5], which updates [8].

Factors also provide counterexamples to false conjectures. Fermat asserted that he thought that all numbers $2^{2^n} + 1$ were prime. Euler disproved this statement by noting that 641 is a proper factor of $2^{2^5} + 1$. When some students learn Euclid's proof that there are infinitely many primes, they think that $1 + \prod_{p \leq x} p$ is always prime. The first counterexample is $1 + 2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 = 59 \cdot 509$.

It is much easier to tell whether a large integer is prime or not than it is to factor it when it is composite. It is a challenging enterprise to factor huge numbers. Factoring algorithms are interesting to analyze. The subject has a long tradition. Many famous mathematicians have developed algorithms for factoring integers.

A few years ago the study of factoring methods suddenly found a new application, in cryptography. Rivest, Shamir and Adleman [30] used Euler's theorem for large composite numbers to realize the trap-door function envisioned by Diffie and Hellman [11].

The RSA system works this way: Choose two large primes p and q . Make them, say, 100 digits each; it is easy to construct random primes of that size. Keep p and q secret, but tell everyone the product $r = pq$. Also, make public a fixed random number e which is between 0 and $\phi(r) - 1$ and relatively prime to $\phi(r)$. Messages can be broken into pieces and encoded as numbers x with $0 \leq x < r$. Someone who wants to send a message to you should encipher such

a number x by sending you $y = (x^e \bmod r)$, that is, $y \equiv x^e \pmod{r}$, $0 \leq y < r$. (Notation: $a \bmod b$ means the least non-negative residue of a modulo b .)

Meanwhile, you should compute a multiplicative inverse d of e modulo $\phi(r) = (p-1)(q-1)$ by the extended Euclidean algorithm (see page 325 of [16]). When you receive y , decipher it by computing $y^d \bmod r$. This is x because $y^d \equiv x^{ed} \equiv x^1 \pmod{r}$ by Euler's theorem, and $0 \leq x < r$.

You are able to recover the plaintext x from the ciphertext y because you know the factorization of r . Anyone else who can factor r can decipher the message, too, of course. However, so far as anyone knows, the only sure way to compute x from y , r and e is to factor r . You should choose p and q about the same size and large enough so that $r = pq$ is too big for your enemies to factor. Several years ago security experts thought that 100-digit numbers r were large enough to be safe. Now (1986), r should be chosen with at least 200 digits. Since the amount of effort needed to encipher and decipher a message is proportional to the number of digits in r , you should not make r too large. The numbers are not so large that fast multiplication techniques help much. Redundant arithmetic [33] is the most suitable computational accelerator for numbers of this size.

Since r and e are public, it is not necessary to send a secret key by courier before using the cipher. Also, the RSA system can be used to "sign" a message in an unforgeable way. Because of these nice properties the system is used by government and business to protect their secrets. They are interested in the difficulty of factoring so that they can know how large to choose p and q .

2. CFRAC—The Continued Fraction Factoring Algorithm.

In 1981, the authors decided to build a special computer for factoring large numbers quickly. In this section and the next, we describe the two factoring algorithms we considered for use on the machine. They are CFRAC, the continued fraction method, and QS, the quadratic sieve method. They were the two fastest factoring algorithms known in 1981. At that time, CFRAC was about ten years old and had recently been improved. QS was just being implemented.

Let N be the number we wish to factor. In CFRAC, you compute the beginning of the simple continued fraction expansion of $\sqrt{N}$. Specifically, you compute sequences $\{Q_n\}$, $\{q_n\}$, $\{r_n\}$ and $\{A_n\}$ of integers by the recursion formulas

$$Q_n := Q_{n-2} + q_{n-1}(r_{n-1} - r_{n-2})$$

$$q_n := \left\lfloor \frac{2 \lfloor \sqrt{N} \rfloor - r_{n-1}}{Q_n} \right\rfloor$$

$$r_n := (2 \lfloor \sqrt{N} \rfloor - r_{n-1}) \bmod Q_n$$

$$A_n := (q_n A_{n-1} + A_{n-2}) \bmod N$$

valid for $n \geq 1$. The initial values are

$$Q_{-1} := N, Q_0 := 1, q_0 := \lfloor \sqrt{N} \rfloor, r_{-1} := 0, A_{-1} := 1 \text{ and } A_0 := \lfloor \sqrt{N} \rfloor.$$

We are most interested in the two sequences $\{Q_n\}$ and $\{A_n\}$ because they satisfy

$$A_{n-1}^2 \equiv (-1)^n Q_n \pmod{N} \text{ and } 0 < Q_n < 2\sqrt{N} \quad (n \geq 0).$$

Thus, the continued fraction expansion of $\sqrt{N}$ produces many small quadratic residues modulo N , a fact known to Legendre, who used them to aid in factoring N in a different way from CRAC.

We seek collections of the Q 's whose product, with proper sign, is a square. Suppose $\prod_{n \in S} (-1)^n Q_n = Y^2$. Let $X = \prod_{n \in S} A_n$. Then $X^2 \equiv Y^2 \pmod{N}$, so N divides $(X - Y)(X + Y)$. If some prime factor of N divides $X - Y$ but not $X + Y$, and this seems to happen at least half of the time, then $\text{GCD}(N, X - Y)$ is a proper divisor of N . If $X \equiv \pm Y \pmod{N}$, however, we try again with another collection of Q 's. Legendre used non-trivial congruences $X^2 \equiv Y^2 \pmod{N}$ to factor N . Lehmer and Powers [19] used the continued fraction expansion of $\sqrt{N}$ to construct congruences $X^2 \equiv Y^2 \pmod{N}$ and factor N .

How can we quickly find sets of Q 's whose product is a square? If we had factored the Q 's, then we could try to match their prime factors so that each prime occurs an even number of times in the product. It is easy to obtain the correct sign: just treat -1 as another "prime", which divides Q_n for odd n .

Suppose we have factored some of the Q 's. Let $p_0 = -1$ and $p_1, \dots, p_m$ be all the primes which occur as factors of any of the Q 's which we factored. Write $(-1)^n Q_n = \prod_{i=0}^m p_i^{e_{ni}}$. Then Q_n is represented by the exponent vector $\{e_{ni}\}$. Note that the exponent vector of the product of Q 's is the sum of their exponent vectors. The product of some Q 's is a square if and only if the sum of their exponent vectors has all its components even. If we regard the vectors as elements of an $(m+1)$ -dimensional vector space over $\text{GF}(2)$, then sets of Q 's whose product is a square correspond to linear dependencies among their exponent vectors. We can use Gaussian elimination over $\text{GF}(2)$ to find a basis for the null space of the matrix $[e_{ni}]$. Then each basis vector

yields a different set S , namely, the set S of all n for which the n^{th} component of the vector is 1.

A good strategy for factoring the Q 's is to choose the primes $p_1, p_2, \dots, p_m$ first and use trial division by just these primes. One can prove that if an odd prime p divides any Q_n , then the Legendre symbol $(\frac{N}{p}) = +1$ (or 0, which we ignore because we are trying to factor N). Thus we choose a *factor base* consisting of $p_1 = 2$ and the first $m - 1$ odd primes with $(\frac{N}{p}) = +1$. The outer loop of the CFRAC algorithm generates the sequences $\{Q_n\}$, $\{A_n\}$, etc. The trial division of Q_n by the p_i 's is done in the inner loop, which looks like this:

```

Q := Q_n
i := 1
while (i ≤ m) do
  if (Q mod p_i = 0) then
    Q := Q / p_i
  else
    i := i + 1
  end if
end while
if (Q = 1) then save A_{n-1}, Q_n for later

```

Naturally, when you compute $Q \bmod p_i$, you save the quotient for possible use in the next line. This trial division loop is the computational bottleneck of CFRAC. The divisions are expensive ones because the quotients are large.

Morrison and Brillhart [22] introduced the uniform factor base and the idea of using elimination over $\text{GF}(2)$ to find congruences $X^2 \equiv Y^2 \pmod{N}$.

We have just described the *basic* CFRAC algorithm. It admits of several variations. The *large prime variation* effectively enlarges the factor base at little extra cost. Suppose that the remaining cofactor $Q > 1$ at the end of the trial division instructions displayed above. Then $Q > p_m$ because we have removed all divisors of Q which are $\leq p_m$. If $Q < p_m^2$, then Q must be prime and we have obtained the complete factorization of Q_n . This Q_n may be useful for forming a square provided we can find another Q_i which leaves the same prime Q after trial division by the factor base. Then $Q_n Q_i / Q^2$ has all of its prime factors in the factor base. To perform the large prime variation, change the last line in the code above to

```

if (Q < p_m^2) then save Q, A_{n-1}, Q_n for later

```


Before the elimination step the triples Q, A_{n-1}, Q_n are sorted on Q and those triples whose Q is not repeated in some other triple are discarded. Each triple having $Q = 1$ produces one row of the exponent matrix. Each $Q > 1$ which appears in exactly k triples yields $k - 1$ rows of the matrix.

In the *early abort variation* you pause one or more times during the trial division loop and compare the remaining cofactor Q with a bound. If Q exceeds this bound, then you abandon work on Q_n and proceed to Q_{n+1} . Although this strategy may cause you to miss some Q_n 's which would have factored, you will not lose many of them if you wisely choose the bounds and places to pause, and you will save a lot of trial division. See [26] for good choices of the parameters and for some other variations of CFRAC.

Let $L(N) = \exp(\sqrt{\ln N \ln \ln N})$. Note that as $N \rightarrow \infty$, $L(N)$ grows faster than any power of $\ln N$ and more slowly than N^ϵ for any $\epsilon > 0$. Assume that the numbers Q_n are just as likely as other numbers of the same size to factor completely over the factor base. Then the basic CFRAC algorithm takes $L(N)^{\sqrt{2} + o(1)}$ steps to factor N . The early abort variation reduces this effort to $L(N)^{\sqrt{3/2} + o(1)}$ steps. The large prime variation does not reduce the exponent in the asymptotic running time. However, it roughly halves the running time and this is an important practical consideration. When factoring numbers of about 50 digits, the early abort variation speeds up CFRAC about tenfold. The acceleration is greater for larger numbers. Other variations can reduce the time to $L(N)^{1 + o(1)}$.

Morrison and Brillhart [22] used the large prime variation and tried the early abort variation with one abort. Pomerance [24] analyzed these variations, found optimal choices for the early abort parameters and promoted that variation. Wunderlich [39], [40] has programmed CFRAC on the MPP supercomputer.

3. QS—The Quadratic Sieve Factoring Algorithm.

The other factoring algorithm which we considered for use on our special computer was the quadratic sieve algorithm. QS resembles CFRAC in that both algorithms produce pairs A, Q with $A^2 \equiv \pm Q \pmod{N}$ and Q factored. The methods differ in the source of these pairs and the method of factoring the Q 's.

Many ideas of QS are quite old. (See [17] and the discussions in [25] and [41].) Pomerance [24] discovered QS independently before he learned of the earlier work. He analyzed QS and predicted correctly that it would be a powerful factoring method. It was first programmed by Gerver [13] who used it to factor a 47-digit cofactor of $3^{225} - 1$.

The original version of QS used the polynomial $Q(X) = (\lfloor \sqrt{N} \rfloor + X)^2 - N$. One has $(A + \lfloor \sqrt{N} \rfloor)^2 \equiv Q(A) \pmod{N}$, $Q(A) < 2\sqrt{N} |A| + A^2$ and $Q(A) > 0$ for $A \neq 0$. The congruence plays the role of $A_{n-1}^2 \equiv (-1)^n Q_n \pmod{N}$ in CFRAC. For small $|A|$, the $Q(A)$ are not many times larger than the Q_n of CFRAC. However, most of them *are* larger, and, if we had to factor them by trial division, QS would be inferior to CFRAC.

In fact, at least for large N , QS beats CFRAC because the numbers $Q(A)$ can be factored by a sieve, which is much faster than trial division. One chooses a factor base, $p_1, p_2, \dots, p_m$, as in CFRAC. As usual, we assume that no small prime divides N . Then the factor base consists of $p_1 = 2$ and the first $m - 1$ odd primes for which the congruence

$$Q(X) \equiv 0 \pmod{p} \quad (3)$$

has two solutions. There are two solutions precisely for the odd primes p satisfying $(\frac{N}{p}) = +1$, as in CFRAC. Let p be an odd prime in the factor base. Let A_1 and A_2 be the two solutions to (3). Then the X for which p divides $Q(X)$ are precisely the X in the union of the two arithmetic progressions $X \equiv A_1 \pmod{p}$ and $X \equiv A_2 \pmod{p}$.

The QS algorithm proceeds this way: Initialize an array Q with the values of the polynomial Q , that is, let $Q[X] := Q(X)$ for $0 \leq X \leq M$, say. Compute the factor base $p_1, \dots, p_m$. Then

```

for i = 1 to m do
  Let  $A_1$  and  $A_2$  be the solutions to  $Q(X) \equiv 0 \pmod{p_i}$ 
  for j = 1 to 2 do
     $X := A_j$ 
    while ( $X \leq M$ ) do
      (*)       $Q[X] := Q[X]/p_i$  (* The division is exact. *)
       $X := X + p_i$ 
    end while
  end for
end for
if ( $-Q[0] \leq p_m$ ) save 0 for later
for  $X = 1$  to  $M$  do
  if ( $Q[X] \leq p_m$ ) save  $X$  for later
end for

```

This program sieves the interval $0 \leq X \leq M$. Of course, you should sieve the interval $-M \leq X \leq -1$, too.

Note that there is no *trial* division in the program. Every time we execute line (*) we know that p_i divides $Q[X]$. In CFRAC nearly all of the divisions have a non-zero remainder.

Note also that we have neglected repeated prime factors. There are three ways the program could deal with this problem. After (*) it could simply check whether p_i divides the new $Q[X]$, as in CFRAC, and divide it out if it does. Another method is to regard the powers of each p_i as "primes", solve $Q(X) \equiv 0 \pmod{p_i^k}$ for appropriate k and sieve by these progressions. The third method is to disregard them completely, as we did in the program above. When you use the second or third method you must not require $Q[X] = 1$ as was done in CFRAC. Save those X for which $Q[X] \leq p_m$ instead. Such small $Q[X]$ certainly factor completely over the factor base, even though the program may not have achieved the complete factorization. With these methods you may occasionally miss a $Q[X]$ which factors completely over the factor base, but it would take much longer to get them all.

Another trick which is always worthwhile is to use logarithms. Compute the logarithm of each prime as you set up the factor base. In each line of the program in which $Q[X]$ appears, replace $Q[X]$, p_i and p_m each by its logarithm and change division to subtraction in line (*). This trick speeds the program because subtraction is simpler than division. The logarithms need not be very accurate. In fact, you may use something like $\left\lfloor 1024 * \log_2 p \right\rfloor$ instead of " $\log p$ " if integer subtraction is faster than floating point subtraction on your machine. Furthermore, except when X is very near 0, $\log Q(X)$ changes so slowly that you can initialize $Q[X]$ to constant values over very long intervals.

When you use logarithms you cannot tell whether p_i divides $Q[X]$ more than once. Therefore, you cannot handle repeated prime factors by the first method above.

Just as in CFRAC, there is a large prime variation of QS: In the penultimate line of the program replace p_m (or $\log p_m$) by an appropriate larger bound.

On the other hand, the early abort strategy would not make sense for QS. A large fraction of the work is spent removing the small primes. It is easier to sieve by the large primes, and they have greater value in factoring $Q(X)$. It would be silly to abort after doing the hard work and before doing the easy work. Furthermore, we could not continue to sieve after an abort because we cannot use a sieve for selected values of $Q(X)$ —only for all values in an interval.

One drawback of the basic QS algorithm just described is that as $|X|$ moves away from 0, $Q(X)$ grows and becomes less likely to have all of its prime divisors in the factor base. So long as $|X|$ remains small compared to $N^{1/4}$, $Q(X)$ is approximately $2\sqrt{N} |X|$. This problem can be overcome by using other polynomials and sieving each one over a shorter interval. The

discriminants of all these quadratic polynomials must be small multiples of N in order for the factor base to remain unchanged. Subject to this discriminant restriction we choose the polynomials so that their values are as small as possible over the sieving interval. To decide how often to change polynomials, we must balance the reward of having small polynomial values (which factor easily) against the penalty of setting up each new polynomial, that is, the cost of finding the roots of the polynomial modulo p for each p in the factor base. Since we can predict about how many polynomial values will factor completely, the length of the sieving interval is determined by the frequency of polynomial changes. The roots of the quadratic polynomial may be computed in probabilistic polynomial time (in $\log p$) by the algorithm of Berlekamp [2] or Lehmer [18], or in $O(p)$ steps by trial and error. A good program might use the former for large p and the latter for small p .

Any variation of QS in which the polynomial is changed several times is called the *multiple-polynomial quadratic sieve*.

Here are some ways of choosing other polynomials: First suppose we have sieved $Q(X)$ (the polynomial discussed earlier) with the large prime variation. Let p be a large prime, that is, $p > p_m$, p divides $Q(A_0)$ for some A_0 and all other prime divisors of $Q(A_0)$ are in the factor base. Define a new polynomial $Q_p(X) = Q(A_0 + pX)$. Then p divides $Q_p(X)$ for every X , and the quotient $Q_p(X)/p \approx 2\sqrt{N} |X|$. When the known factor p is removed from each $Q_p(X)$, the quotient is about as large as $Q(X)$. Although $Q_p(0) = Q(A_0)$ is a number we have already factored, nearly all $Q_p(X)$ for $X \neq 0$ are new to us. We sieve $Q_p(X)$ over some interval, for example, the same one used for $Q(X)$, but we do *not* use the large prime variation. Davis and Holdridge [9] suggested $Q_p(X)$.

Peter Montgomery [21] found a way to "fit" polynomials to N and the sieve interval length $2M$. (Choose M first.) Consider the polynomial $F(X) = aX^2 + 2bX + c$. If we require that N divide $b^2 - ac$, then

$$aF(X) = (aX + b)^2 - (b^2 - ac) \equiv (aX + b)^2 \pmod{N}, \quad (4)$$

so that $aF(X)$ is a quadratic residue modulo N . The requirement that $|F(X)|$ be small for $-M \leq X \leq M$ lead Montgomery to choose $a \approx \frac{\sqrt{2N}}{M}$, where a is a prime with $(\frac{N}{a}) = +1$, b a solution of the congruence $b^2 \equiv N \pmod{a}$ and $c = \frac{b^2 - N}{a}$. Since there are many primes a near $\frac{\sqrt{2N}}{M}$ with $(\frac{N}{a}) = +1$, we can construct many good polynomials. The maximum value of $|F(X)|$ on the interval $-M \leq X \leq M$ is about $\frac{1}{2\sqrt{2}}$ of the maximum value of $Q(X)$ on the same

interval. This makes the $F(X)$ slightly easier to factor than $Q(X)$.

Because $aF(X)$ rather than $F(X)$ is the quadratic residue modulo N , the prime a must be eliminated. We lose one congruence (4) when we perform this elimination. This loss becomes a serious problem when we change polynomials so frequently that the expected number of factored $F(X)$ per polynomial is 1 or less, as happens when factoring numbers of 90 or more digits.

Pomerance [25] overcame this difficulty by choosing $a = q^2$, where q is a prime near $\left\lceil \frac{\sqrt{2N}}{M} \right\rceil^2$

with $\left(\frac{N}{q}\right) = +1$. If we also insist that $q \equiv 3 \pmod{4}$, then we can solve $b^2 \equiv N \pmod{q^2}$ easily

by setting $b_1 = N^{\frac{q+1}{4}} \pmod{q}$ (so that $b_1^2 \equiv N \pmod{q}$) and $b_2 = (2b_1)^{-1} \left[\frac{N - b_1^2}{q} \right] \pmod{q}$.

Then $b = b_1 + qb_2$. With this idea all arithmetic can be done modulo q rather than modulo q^2 , and this is a considerable savings when q^2 exceeds the word size.

Silverman [32] has implemented all these ideas on a VAX. His program has factored many numbers from [5] up to 69 digits.

The running time for QS is $L(N)^{\sqrt{9/8} + o(1)}$. Variations not described here reduce the exponent to $1 + o(1)$.

4. Special Computers for Factoring Integers.

Our 1981 choice of CFRAC as the algorithm for a special processor for factoring large integers was influenced by the following considerations: CFRAC was a well-established algorithm. It had just become clear that CFRAC would be much accelerated by the early abort strategy. CFRAC uses little memory during the time-consuming trial division, and computer memory was more expensive in 1981 than now. At the heart of the algorithm are the costly division operations; we thought of a nice way to do them in parallel.

In Fall, 1981, QS showed theoretical promise of being an effective factoring algorithm. Gerver was programming it, but it had not yet factored a single number. The multiple-polynomial variation was unknown, at least to us. It was clear that QS was faster than CFRAC for sufficiently large numbers, but it was not clear how large "sufficiently large" was. QS needs a larger factor base than CFRAC to factor N . The memory must hold the sieve interval (the Q array), too. It appeared that QS needed a lot of memory and was not obviously superior to CFRAC for numbers of the size (50 to 75 digits) we hoped to factor. Therefore, we chose CFRAC.

The Extended Precision Operand Computer

We call our computer the Extended Precision Operand Computer or EPOC. It consists of a main processor and two peripheral units. The reason for the machine's name is that the operand size of the main processor is 128 bits, which is "extended" compared to most machines. Also, the operand word size is adjustable in the EPOC architecture. Integers as large as $2^{128} - 1$ can be manipulated in single instructions. Recall that, except for N and the A_n 's, all variables in CFRAC are $< 2\sqrt{N}$. Since $2\sqrt{N} < 2^{128}$ means $N < 2^{254} \approx 10^{77}$, the EPOC can factor numbers as large as 77 digits using mostly single precision arithmetic. Naturally, this large word size makes the CFRAC program for the EPOC simpler than it would be for a machine with a smaller word size. The choice of word size was influenced by the fact that the EPOC would take many months to factor a 77-digit number.

The main processor, which consists of the sequencer, arithmetic-logic unit and the memories in the diagram, generates the four sequences $\{Q_n\}$, $\{q_n\}$, $\{r_n\}$ and $\{A_n\}$. It sends each Q_n to a trial division unit for factoring. That unit, called the "mod squad", has an array of remainder elements which divides one Q by many different primes at once. The elements are loaded with the factor base when the program starts. As each dividend Q is generated, it is broadcast to all elements one bit at a time. The mod squad returns a bit vector which tells which primes divided Q . The main processor examines this vector and makes the early abort decisions. When Q_n factors completely, or with a large prime, the numbers n , A_{n-1} and Q_n must be stored for later use. Since the EPOC has little memory these numbers must be sent to another machine (the "host") for storage. The EPOC does not compute the large primes because it uses the logarithm trick which we described for QS. Instead, the host re-factors Q_n to determine the large prime. The host performs the elimination step, too, when it decides that enough Q 's have been factored. The input/output unit of the EPOC handles the communication with the host. Since there is little i/o during CFRAC for large N , this unit can be relatively slow. See [28] and [34] for more about the EPOC.

The EPOC language is horizontal microcode. Source programs are prepared on the host and loaded into the EPOC by the i/o unit. As an example of EPOC code, here are the instructions which compute

$$Q_n := Q_{n-2} + q_{n-1}(r_{n-1} - r_{n-2}).$$

```

# Compute QNEXT = LILQ*(RNEXT - RNOW) + QNOW
# There are two cases depending on whether RNEXT ≥ RNOW or not.
# RNEXT is already in R6. Get RNOW and compare.
# We also copy QNOW := QNEXT and RNOW := RNEXT
    MAR<AF AF<AD RNOW          # mar := A(RNOW)
    MS-E RB5 ALU-C             # R5 := RNOW
    RA6 MS<RA                  # RNOW := R6 = RNEXT
    MAR<AF AF<AD LILQ          # mar := A(LILQ)
    MS-E RB2 ALU-C             # R2 := LILQ
    MAR<AF AF<AD QNEXT         # mar := A(QNEXT)
    MS-E RB0 ALU-C             # R0 := QNEXT
    MAR<AF AF<AD QNOW          # mar := A(QNOW)
    MS-E RB7 ALU-C             # R7 := QNOW
    RA0 MS<RA                  # QNOW := R0 = QNEXT
    RA6 RB5 R-S-1+C C1 F ALU-E # R6 ≥ R5?
    CJP C.SIGN CLESS           # Jump if R6 < R5
# Case R6 ≥ R5, that is, RNEXT ≥ RNOW
    RA6 RB5 R-S-1+C C1 F ALU-E ALU-C # R5 := R6 - R5
    RB1 RB<RA RA5 _            # R1 := R5
    JS RMUL # (R0,R1) := LILQ*(RNEXT - RNOW)
    RB7 RA1 R+S+C C0 F ALU-E ALU-C _ # R7 := R7 + R1
    JP CDONE                    # Jump over the second case
# Case R6 < R5, that is, RNEXT < RNOW
CLESS: RA6 RB5 S-R-1+C C1 F ALU-E ALU-C # R5 := R6 - R5
    RB1 RB<RA RA5 _            # R1 := R5
    JS RMUL # (R0,R1) := LILQ*(RNOW - RNEXT)
    RB7 RA1 S-R-1+C C1 F ALU-E ALU-C # R7 := R7 - R1
CDONE: MAR<AF AF<AD QNEXT      # mar := A(QNEXT)
    RA7 MS<RA                  # QNEXT := R7

```

The comments explain the code fairly well. The embedded "<" signifies register transfer. JS calls a subroutine. RMUL is the one which multiplies non-negative integers. CJP is a conditional jump. C.SIGN is the condition that the sign bit is 1. MS-E and ALU-E enable the main store and ALU registers, respectively, that is, these control signals tell them to write a 128-bit number out onto a bus. The control signal ALU-C stores a number in an ALU register.

The EPOC has been built and is factoring large numbers now. The first number it factored was the 54-digit composite divisor of $7^{135} - 1$, namely,

$$\begin{aligned} &140087490271311776457565042682692598890328116470462211 \\ &= 175561876921802311 \cdot 797937984757981841530188510084780901 \end{aligned}$$

Recently, the EPOC factored this 62-digit divisor of $3^{204} + 1$:

$$\begin{aligned} &11935719952868823358794699404988004449487008573277354764852481 \\ &= 53958784909638588059956801 \cdot 221200680720606092053543425481471681 \end{aligned}$$

As of this writing, this number is the second largest number ever factored by CFRAC. (The largest is another 62-digit number just factored by McCurdy and Wunderlich on the MPP.) The numbers are interesting because they come from [5].

The EPOC takes about 1 hour, 1 day and 1 month to factor numbers of 50, 60 and 70 digits, respectively.

We considered several possible enhancements which would speed up the EPOC so that it could factor a 100-digit number in a few months. Some improvements were a faster technology (ECL instead of TTL) and having several main processors share a large, fast VLSI mod squad. However, the advent of the multiple-polynomial quadratic sieve and the decrease in the price of computer memory made these improvements to the EPOC unnecessary.

Currently, Pomerance, Smith and Tuler [27] are designing a highly pipelined machine for the QS algorithm. Its main processor will set up each polynomial and pass it to the first of a chain of sieving units. Each of these units will sieve the polynomial over a different part of the interval. It is expected that this processor will be able to factor a 100-digit number in about a month at less cost than the enhancements we considered for the EPOC.

See [12] and [31] for information about two other special computers for number theory, especially for factoring large integers.

5. Pollard's $p - 1$ Factoring Algorithm.

In 1985, H. W. Lenstra, Jr. [20] invented a new, powerful factoring algorithm which is called the *elliptic curve method* (ECM) because it uses elliptic curves. It is related to Pollard's " $p - 1$ " factoring method [23] which is about ten years old and which we describe briefly.

Fermat's "Little Theorem" says that if p is an odd prime, then $2^{p-1} \equiv 1 \pmod{p}$. Consequently, $2^{m(p-1)} \equiv 1 \pmod{p}$ for any integer m . If $p - 1$ divides $i!$, say, $i! = m(p - 1)$, then $2^{i!} \equiv 1 \pmod{p}$. Now suppose we want to factor N and that p is an unknown odd prime divisor

of N . If $p - 1$ divides $i!$, then p divides $\text{GCD}(N, 2^{i!} - 1)$. Although $2^{i!} - 1$ grows too swiftly to compute in full, we can compute it modulo N , and this is good enough. Here is Pollard's $p - 1$ algorithm:

```
Choose  $k$ , the amount of effort to expend trying to factor  $N$ .
Choose  $s$ , the number of steps between GCD's.
 $t := 2$ 
for  $i = 2$  to  $k$  do
     $t := t^i \bmod N$  (* Now  $t = 2^{i!} \bmod N$  *)
    if  $(i \bmod s = 0)$  do
         $p := \text{GCD}(N, t - 1)$ 
        if  $(p > 1)$  print  $p$  and then stop
    end if
end for
print "No factor found"
```

The algorithm discovers the prime factor p of N as soon as i is large enough so that all prime factors of $p - 1$ appear in $i!$, with appropriate multiplicity. Usually the multiplicity is not an issue and the algorithm stops with factor p at the next GCD after i reaches the largest prime factor of $p - 1$.

The algorithm is not guaranteed to factor N . It will fail, for example, when for each prime p dividing N , $p - 1$ has a prime factor greater than k . In practice, however, it factors many numbers. It has discovered hundreds of factors in [5]. Typical values for k and s are 10^5 to 10^6 and 10^3 to 10^4 , respectively.

Even when Pollard's $p - 1$ algorithm prints p and stops, p need not be prime. It may happen that N has two prime factors p, q such that the largest prime divisors of $p - 1$ and $q - 1$ both lie between two consecutive multiples of s . Then p and q are discovered together at the next GCD, which is the product pq . One remedy for this coincidence is to repeat the algorithm with a smaller s . Saving t at each multiple of s until i reaches the next multiple of s will reduce the work. When the composite GCD is noticed (especially if it is N), back up to the previous multiple of s and repeat the work, but taking the GCD at each i (or each prime i). If both p and q are discovered at the same i , the only recourse is to repeat the whole algorithm with the initial value of t changed from 2 to some larger prime.

Since small primes cannot appear in $p - 1$ with the enormous multiplicity that they do in $i!$, it is more sensible to replace $i!$ by the product of the primes up to i , with small primes given a modest multiplicity.

There is a two-step variation of Pollard's $p - 1$ method which discovers prime factors p of N when $p - 1$ has one medium-sized prime factor and all its other prime factors are small. See [21] for details. Using the two-step variation of $p - 1$, Baillie has found a large number of factors of numbers from [5]. His most spectacular example is the divisor

$$p = 174463386657191516033932614401$$

of $2^{740} + 1$. He succeeded using the $p - 1$ method because

$$p - 1 = 2^8 \cdot 5^2 \cdot 17 \cdot 37 \cdot 1627 \cdot 5387 \cdot 68111 \cdot 152081 \cdot 477361.$$

There is also a $p + 1$ algorithm (see [37]) which discovers factors p of N when all prime factors of $p + 1$ are small. It uses Lucas sequences in place of powers of 2.

Note that the calculations of the $p - 1$ method are done mostly in the multiplicative group of the integers modulo N and that the $p - 1$ algorithm stops just when $t - 1$ first drops out of this group, that is, when $t - 1$ first becomes not relatively prime to N .

6. ECM—The Elliptic Curve Method.

Lenstra's ECM is similar to Pollard's $p - 1$ method: Just replace the multiplicative group of the integers modulo N by an elliptic curve modulo N . We will define *elliptic curve* in a moment. For the following brief description of Lenstra's algorithm it suffices to know that an elliptic curve is an abelian group under an addition operation.

```
Choose  $k$ , the amount of effort to expend per
      elliptic curve trying to factor  $N$ .
Repeat the following until  $N$  is factored or until you give up.
  Choose a random elliptic curve  $E$ .
  Choose a random point  $P_1$  on  $E$ .
   $P := P_1$ 
  for  $i = 2$  to  $k$  do
     $P := i \cdot P$  in  $E \pmod{N}$     (* Now  $P = i! \cdot P_1$  in  $E \pmod{N}$  *)
  end for
end repeat
```

The notation $i \cdot P$, where i is a positive integer and P is an element of the group E , means $P + P + \dots + P$, where there are i terms in the sum and "+" is the group operation in E . This repeated addition may be performed by the "binary method" (see page 441 of [16]). For

example, to compute $P := 13 \cdot P$, you would use these steps:

$$\begin{aligned} Q &:= P + P \\ Q &:= Q + P & (* = 3 \cdot P *) \\ Q &:= Q + Q & (* = 6 \cdot P *) \\ Q &:= Q + Q & (* = 12 \cdot P *) \\ P &:= Q + P & (* = 13 \cdot P *) \end{aligned}$$

In each step after the first line, you compute either $Q + Q$ or $Q + P$. The sequence of operations is determined by the bit pattern of the binary representation of i . It is easy to do this on a binary computer.

During the calculations for each group addition, you must compute a multiplicative inverse modulo N . If the number X to be inverted modulo N is *not* relatively prime to N , then there is good news and there is bad news.

The bad news is that you cannot continue your calculation of $i \cdot P$ because X has no inverse modulo N .

The good news is that $\text{GCD}(X, N)$ is a factor of N , almost certainly a prime factor, so that you have no further need to compute $i \cdot P$.

We now give a brief introduction to elliptic curves. See [1], [15] and [36] for more details. Let A and B be rational numbers such that $4A^3 + 27B^2 \neq 0$. The *elliptic curve* E with parameters A, B is the set of all pairs (x, y) of rational numbers satisfying $y^2 = x^3 + Ax + B$, together with a special point ∞ . This formula is called the *Weierstrass form* of the curve. The group operation "+" is defined on E as follows:

First of all, ∞ is the group identity. If $P \in E$, then $P + \infty = \infty + P = P$.

The inverse of a point $P = (x, y) \neq \infty$ is $-P = (x, -y)$. This means that $(x, y) + (x, -y) = \infty$.

The graph of the cubic equation defining E is a curve in the plane. To add two points of E which are on this graph, draw the straight line through them. (If the two points are identical, draw the tangent line to the graph there.) If the points are not inverses, then the line will intersect the graph in exactly one more point. (A point of tangency counts as two points.) The third intersection point must have rational coordinates. The sum of the two given points is the inverse of the third point. This implies that the sum of the three intersection points of the line with the graph is ∞ .

With a little elementary algebra you can express the addition on E in formulas. Here are the gory details for computing $(x_3, y_3) := (x_1, y_1) + (x_2, y_2)$:

If $x_1 = x_2$ and $y_1 = -y_2$, then the sum is ∞ , which is not represented as a pair.

If $x_1 = x_2$ and $y_1 \neq -y_2$, then $y_1 = y_2 \neq 0$ and we define $s = \frac{3x_1^2 + A}{2y_1}$ (= the slope of the tangent line). If $x_1 \neq x_2$, define $s = \frac{y_1 - y_2}{x_1 - x_2}$ (= the slope of the secant line). In either of these cases, let $x_3 = s^2 - x_1 - x_2$ and $y_3 = sx_1 - y_1 - sx_3$.

It is clear from these formulas that if $(x_1, y_1) \neq (x_2, -y_2)$ and these points have rational coordinates, then (x_3, y_3) has rational coordinates. With additional elementary algebra, you can show that E is an abelian group under "+".

Note that the parameter B is not used in the formulas for addition and that A is used only in one place, where we add a point to itself.

In ECM, we must choose a random elliptic curve E and a random point P_1 on it. One way to do this is to choose a random $P_1 = (x, y)$ with integer coordinates and a random integer A and then let $B = y^2 - x^3 - Ax$. Since B is never used in the algorithm, the last calculation may be omitted, except by purists.

Now we reduce E modulo an integer N : Let A, B be integers such that $\text{GCD}(N, 4A^3 + 27B^2) = 1$. The *elliptic curve* E_N with parameters A, B is the set of all pairs (x, y) of integers modulo N satisfying $y^2 = x^3 + Ax + B$, together with ∞ . Define a binary operation "+" on E_N by the formulas which define "+" on E , except that the arithmetic is performed modulo N . (When "performed modulo N ", the condition "if $y_1 = -y_2$ " becomes "if N divides $y_1 + y_2$ ", etc.) The " $E \pmod{N}$ " in the ECM program means E_N . When N is a prime number, E_N is a finite group of rank 1 or 2.

However, when N is composite, E_N is not a group under "+". But, if N has no small prime divisors, you may calculate for a long time in E_N before you discover that "+" is not defined for all pairs of points. It fails when the denominator of a slope s is not relatively prime to N . In the algorithm for ECM this failure usually occurs when the sum is ∞ in E_p for some p dividing N and we don't notice this event modulo N . We may have $x_1 \equiv x_2 \pmod{p}$ but $x_1 \not\equiv x_2 \pmod{N}$, for example. Working in E_p , we would decide that the sum was ∞ . But, computing in E_N , as we must because p is unknown to us, we would try to calculate the slope of a secant line and discover that $X = x_1 - x_2$ is not invertible modulo N . This event terminates the algorithm with the factor $\text{GCD}(X, N)$ of N . Probably, $\text{GCD}(X, N) = p$.

We now indicate why it is not just dumb luck that this event happens. To show that with high probability it eventually occurs and to estimate the waiting time for it, we need some facts about elliptic curves. The estimate of the running time for ECM will be finished in Section 7.

Let a_p be the order of the finite group E_p and let $a_p(P)$ be the order of the point $P \in E_p$. Then $a_p(P)$ divides a_p . In ECM, as soon as i is large enough for $i!$ to be a multiple of $a_p(P_1)$ we have $i! \cdot P_1 = \infty$ in E_p . Suppose i is the smallest integer for which $a_p(P_1)$ divides $i!$. Then it is very likely that i is prime and that it is the largest prime factor of $a_p(P_1)$. Since $\frac{a_p}{a_p(P_1)}$ is not likely to be large, i is probably also the largest prime factor of a_p . (Compare this situation with that of Pollard's $p-1$ algorithm, which usually stops when i is the largest prime factor of $p-1$.) Therefore, at some step during the computation of $P := i \cdot P$, we must produce ∞ in E_p , but not ∞ in E_N . Since the temporary variable Q is always $j \cdot P$ for various $j < i$, ∞ must appear for the first time in the last step, $P := Q + P$. (This is always the last step when i is odd; i is odd here because it is a prime > 2 .) This means that Q and P are inverses in E_p , so that their x -coordinates are congruent modulo p . On the other hand, the x -coordinates are probably not congruent modulo N and we will attempt to invert their difference modulo N . The GCD which we compute on the way to the inverse will be p .

We have just seen that the running time of ECM for one elliptic curve E depends on the size of the largest prime factor of a_p . What can we say about a_p ? For each fixed x in $0 \leq x < p$, the number of y for which $y^2 \equiv x^3 + Ax + B$ is $1 + (\frac{x^3 + Ax + B}{p})$, where $(\frac{X}{p})$ is the Legendre symbol. Hence,

$$a_p = 1 + \sum_{x=0}^{p-1} (1 + (\frac{x^3 + Ax + B}{p})) = 1 + p + \sum_{x=0}^{p-1} (\frac{x^3 + Ax + B}{p}).$$

(The first 1 is for ∞ .) Since $(\frac{X}{p}) = 0$ for $X = 0$, $= +1$ for $\frac{p-1}{2}$ values of X and $= -1$ for $\frac{p-1}{2}$ values of X , we would expect $a_p \approx 1 + p$. In fact, Hasse (see page 187 of [36]) proved the "Riemann Hypothesis for finite fields," which says that

$$p + 1 - 2\sqrt{p} < a_p < p + 1 + 2\sqrt{p}. \quad (5)$$

Birch [3] has shown that if p is fixed and A and B vary, then a_p has a reasonable distribution in its interval (5) of possible values. Write $p + 1 - a_p = 2\sqrt{p} \cos \theta_p$ with $0 < \theta_p < \pi$. Then θ_p has approximately a $\sin^2$ distribution for large p .

Experiments [21] support the hypothesis that a_p is a "random" number in the interval (5) (as A and B vary) and that its largest prime factor behaves like the largest prime factor of a typical random number in the same interval. This hypothesis is the basis for estimating the running time of ECM.

Integers whose largest prime factor is small are called *smooth*. Pollard's $p - 1$ method succeeds when $p - 1$ is smooth for some prime divisor p of N . Lenstra's ECM succeeds when a_p is smooth for some prime divisor p of N and for some elliptic curve E which we happen to choose. Of course, $p - 1$ is a possible value for a_p . In a sense, Lenstra freed the $p - 1$ method from its dependency on the smoothness of a single number. ECM chooses many "random" numbers near p and hopes that at least one of them will be smooth. These numbers are the a_p for various E_p , where p is the fixed, but unknown, prime divisor of N .

ECM has a two-step variation which succeeds when some a_p has one medium-sized prime factor q and $\frac{a_p}{q}$ is smooth. See [4] or [21] for details. Here is a naive version: The first step is the ECM algorithm above. Choose a limit K for the second step to be approximately $10k$. Between the `end for` and the `end repeat` in the ECM program shown above insert these instructions:

```
P2 := P1 + P1 in EN
q := the next prime number after k
P := q · P in EN
for i = 1 to K do
    P := P2 + P in EN
end for
```

There are various algebraic devices to speed the arithmetic modulo N needed to compute one group operation in E_N . You can use a different parameterization (instead of the Weierstrass form) to avoid a few multiplications modulo N . You can save another one or two multiplications by requiring that A , B or P_1 has a special form, such as $P_1 = (0, 1)$. If you use homogeneous coordinates (x, y, z) for E_N , the arithmetic for a group operation has no inversion modulo N . This trick avoids doing a GCD, which is expensive on most machines, for every single group operation. Instead, you compute $\text{GCD}(z, N)$ whenever you like, just as GCD's are computed only every s steps in Pollard's $p - 1$ algorithm. See [4], [7] and [21] for details.

Suyama [35] has found a way to choose E and P_1 to guarantee that a_p is always a multiple of 12. This helps make a_p smooth.

7. The Time Complexity of Some Factoring Algorithms.

The running time for the Pollard $p - 1$ algorithm to find the factor p of N is proportional to the largest prime factor of $p - 1$. If a prime q is much larger than a prime p , then it is likely that the largest prime factor of $q - 1$ will exceed the largest prime factor of $p - 1$. Hence, the Pollard $p - 1$ algorithm tends to find small prime factors before it finds large ones. Occasionally, it will discover an unusually large one such as Baillie's 30-digit prime we mentioned earlier.

Recall that we defined $L(N) = \exp(\sqrt{\ln N \ln \ln N})$. As we said, there is a very reasonable heuristic argument which concludes that CFRAC and QS take $L(N)^{\alpha + o(1)}$ steps to factor N , where α is a constant between 1 and $\sqrt{2}$. These algorithms do not run faster when N has a small prime divisor or slower when it doesn't.

In contrast, the running time for ECM, like that of the $p - 1$ method, depends on the size of the divisor p it finds. Under reasonable assumptions, one can show that ECM takes about $\exp(\sqrt{2 \ln p \ln \ln p})^{1 + o(1)}$ steps to discover p . Since there is always a p with $2 \ln p \leq \ln N$, the time for ECM is always less than $L(N)^{1 + o(1)}$. Therefore, ECM in the worst case is as fast as CFRAC and QS are in most cases. However, the $o(1)$ goes to 0 more slowly for ECM than for QS. QS should be used in cases when it is likely that N is the product of two approximately equal primes, as for RSA keys. ECM is an excellent algorithm for finding small prime factors of large numbers. In recent months, Montgomery and Silverman have used it to discover hundreds of relatively small factors of numbers in [5]. Hiromi Suyama has factored some large numbers with ECM running on a microcomputer. Dubner has just implemented the algorithm on his special processor with an accelerator for arithmetic with large integers [12]; he is beginning to factor numbers from [5] with it.

Here are more remarks about how the running times are estimated. An integer N is α -smooth, where $0 < \alpha < 1$, if all of its prime factors are $< N^\alpha$. We will need the basic fact [6] that the probability that N is α -smooth is approximately $\alpha^{-\frac{1}{\alpha}}$ when N is large.

Now suppose p is a prime divisor of N and let k be the amount of effort spent per elliptic curve in ECM. Write $\alpha = \frac{\log k}{\log p}$, so that $k = p^\alpha$. ECM will discover the factor p provided that all prime factors of a_p are $\leq k$. Since a_p is near p by (5), one curve will find p if a_p is α' -smooth for $\alpha' = \frac{\log k}{\log a_p}$, which is very near α . The probability that p is discovered by any one curve is thus approximately $\alpha^{-\frac{1}{\alpha}}$ by the basic fact about α -smooth numbers. Therefore, the reciprocal $\alpha^{\frac{1}{\alpha}}$ of this probability is an estimate of the number of different curves we need to try

in order for one of them to find p .

Let us estimate $\alpha^{-\frac{1}{\alpha}}$. We assume that p is large enough so that $\ln \ln p$ is not much less than $\ln \ln N$. Define β and γ by $k = L(N)^\beta$ and $p = N^\gamma$. Then $0 < \gamma < \frac{1}{2}$ and $\frac{1}{\alpha} = \frac{\gamma}{\beta} \sqrt{\frac{\ln N}{\ln \ln N}}$. The expected number of curves needed to find p is

$$\alpha^{-\frac{1}{\alpha}} = \left(\frac{1}{\alpha}\right)^{\frac{1}{\alpha}} = L(N)^{\frac{\gamma}{2\beta} + o(1)}.$$

The arithmetic needed to compute $P := i \cdot P$ in E_N takes polynomial time (in $\ln N$), which is $L(N)^{o(1)}$ steps. The total work per curve is $kL(N)^{o(1)} = L(N)^{\beta + o(1)}$ steps. Therefore, the total time needed for ECM to find p is about $L(N)^{\frac{\gamma}{2\beta} + \beta + o(1)}$. The choice of β which minimizes $\frac{\gamma}{2\beta} + \beta$ is $\beta = \sqrt{\frac{\gamma}{2}}$. This β makes the total run time be $L(N)^{\sqrt{2}\gamma + o(1)} = \exp(\sqrt{2} \ln p \ln \ln p + o(1))$, as claimed earlier. But we don't know γ because we are trying to discover p . So how can we choose $\beta = \sqrt{\frac{\gamma}{2}}$? The solution to this problem is easy: let k increase from one curve to the next. Let $k = 1$ for the first curve, $k = 2$ for the second curve, etc. We expect to find p about when k reaches $\exp(\sqrt{1/2} \ln p \ln \ln p)$ because $\beta = \sqrt{\frac{\gamma}{2}}$ there. The total work done is about

$$\frac{1}{2}(\exp(\sqrt{1/2} \ln p \ln \ln p))^2 = \exp(\sqrt{2} \ln p \ln \ln p + o(1)).$$

We now have the tools to estimate the running time of CFRAC. Assume that the Q_n 's factor into primes like typical numbers of the same size, which is about $\sqrt{N}$. Then all prime factors of Q_n are in the factor base if and only if Q_n is α -smooth, where $\alpha = \frac{\log p_m}{\log Q_n}$. Writing $p_m = L(N)^\beta$ and approximating Q_n by $\sqrt{N}$, we find $\alpha = 2\beta \sqrt{\frac{\ln \ln N}{\ln N}}$. The basic CFRAC, with no aborts, makes $m = L(N)^{\beta + o(1)}$ trial divisions per Q , and about m factored Q 's are needed to factor N . The probability that one Q can be factored is $\alpha^{\frac{1}{\alpha}}$, so we need to try about $m \alpha^{-\frac{1}{\alpha}}$ of them. The total number of trial divisions required is about $m^2 \alpha^{-\frac{1}{\alpha}} = L(N)^{2\beta + \frac{1}{4\beta} + o(1)}$. The choice $\beta = \frac{1}{\sqrt{8}}$ minimizes the exponent and yields a running time of $L(N)^{\sqrt{2} + o(1)}$. One estimates the running time for QS in the same way. See Pomerance [24] for details.

8. The Double EPOC—A Machine for ECM.

The NSF proposal which supported the EPOC project specified that two EPOC's would be built. The ALU boards (which hold the arithmetic-logic units and the data memories) for the second EPOC were wired recently. But before we fabricated the second sequencer and i/o boards we realized how powerful ECM was and decided to convert the EPOC into an ECM machine. Therefore, we are not building the sequencer or i/o boards for the second EPOC. Instead, we are connecting all the ALU boards for both EPOC's together to form a 256-bit ALU. The sequencer and i/o units from the first EPOC need only trivial changes to accommodate the larger word size. We call the new machine the Double EPOC or DEPOC. It has relatively slow i/o and little memory, but very wide operands. The small memory is quite adequate for ECM. The DEPOC architecture permits a very efficient coding of the binary GCD algorithm (see page 321 of [16]). It can invert a number modulo N in no more time than it takes to multiply two integers modulo N . We are programming ECM on it now and expect it to factor numbers from [5] soon.

References

1. Eric Bach, "Lenstra's algorithm for factoring with elliptic curves, Exposé," Computer Sciences Department, University of Wisconsin, Madison, February, 1985.
2. E. R. Berlekamp, "Factoring polynomials over large finite fields," *Math. Comp.*, 24 (1970), 713-735.
3. B. J. Birch, "How the number of points of an elliptic curve over a fixed prime field varies," *J. London Math. Soc.*, 43 (1968), 57-60.
4. Richard P. Brent, "Some integer factorization algorithms using elliptic curves," Research Report CMA-R32-85, Australian National University, September, 1985.
5. John Brillhart, D. H. Lehmer, J. L. Selfridge, Bryant Tuckerman and S. S. Wagstaff, Jr., *Factorizations of $b^n \pm 1$, $b = 2, 3, 5, 6, 7, 10, 11, 12$ up to high powers*. Amer. Math. Soc., Providence (1983).
6. E. R. Canfield, P. Erdős and C. Pomerance, "On a problem of Oppenheim concerning 'Factorisatio Numerorum'," *J. Number Theory*, 17 (1983), 1-28.
7. D. V. Chudnovsky and G. V. Chudnovsky, "Sequences of numbers generated by addition in formal groups and new primality and factorization tests," Research Report RC 11262 (#50739), IBM Research Center, Yorktown Heights, July, 1985.
8. A. J. C. Cunningham and H. J. Woodall, *Factorisation of $y^n \pm 1$, $y = 2, 3, 5, 6, 7, 10, 11$, 12 Up to High Powers (n)*, Hodgson, London (1925).
9. J. A. Davis and D. B. Holdridge, "Factorization using the quadratic sieve algorithm," in *Advances in Cryptology, Proceedings of Crypto 83*, David Chaum, ed., Plenum Press, New

York (1984), 103-113.

10. L. E. Dickson, *History of the Theory of Numbers, vol. I, Divisibility and Primality*. Carnegie Institute of Washington (1919). Reprinted by Chelsea, New York (1971).
11. W. Diffie and M. Hellman, "New directions in cryptography," *IEEE Trans. on Info. Theory*, IT-22(6) (1976), 644-654.
12. Harvey Dubner and Robert Dubner, "The development of a powerful, low-cost computer for number theory applications," preprint, July, 1985.
13. J. L. Gerver, "Factoring large numbers with a quadratic sieve," *Math. Comp.*, 41 (1983), 287-294.
14. Richard K. Guy, "How to factor a number," *Congressus Numerantium*, 16, Proc. Fifth Manitoba Conf. on Numerical Math., Winnipeg (1976), 49-89.
15. Jean-René Joly, "Equations et variétés algébriques sur un corps fini," *l'Enseignement Math.* (2), 19 (1973), 1-117.
16. Donald E. Knuth, *The Art of Computer Programming, vol. 2, Seminumerical Algorithms*, 2nd ed. Addison-Wesley (1981).
17. M. Kraitchik, *Recherches sur la théorie des nombres*, Tome II, Gauthier-Villars, Paris (1929).
18. D. H. Lehmer, "Computer technology applied to the theory of numbers," in *MAA Studies in Math.* 6, W. J. LeVeque, ed. (1969), 117-151.
19. D. H. Lehmer and R. E. Powers, "On factoring large numbers," *Bull. Amer. Math. Soc.*, 37 (1931), 770-776.
20. H. W. Lenstra, Jr., "Elliptic curve factorization," announcement of February 14, 1985.
21. Peter L. Montgomery, "Speeding the Pollard methods of factorization," preprint, December, 1985.
22. Michael A. Morrison and John Brillhart, A method of factoring and the factorization of F_7 . *Math. Comp.*, 29 (1975) 183-205.
23. J. M. Pollard, "Theorems on factorization and primality testing," *Proc. Cambridge Philos. Soc.*, 76 (1974), 521-528.
24. Carl Pomerance, Analysis and comparison of some integer factoring algorithms. *Computational methods in Number Theory*, 89-139, H. W. Lenstra and R. Tijdeman, eds., Math. Centrum, Amsterdam (1983).
25. Carl Pomerance, "The quadratic sieve factoring algorithm," to appear in *Proc. Eurocrypt 84*, Paris, 1984.
26. Carl Pomerance and S. S. Wagstaff, Jr., Implementation of the continued fraction factoring algorithm. *Congressus Numerantium*, 37 (1983), 99-118.
27. Carl Pomerance, J. W. Smith and Randy Tuler, "A pipe-line architecture for factoring large integers with the quadratic sieve algorithm," preprint, December, 1985.
28. Carl Pomerance, J. W. Smith and S. S. Wagstaff, Jr., "New ideas for factoring large integers," in *Advances in Cryptology, Proceedings of Crypto 83*, David Chaum, ed.,

Plenum Press, New York (1984), 81-85.

29. Hans Riesel, *Prime Numbers and Computer Methods for Factorization*, Birkhäuser, Boston (1985).
30. R. L. Rivest, A. Shamir and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Comm. ACM*, 21(2) (1978), 120-126.
31. W. G. Rudd, Duncan A. Buell and Donald M. Chiarulli, "A high performance factoring machine," *Proceedings of the Eleventh International Symposium on Computer Architecture*, 1984.
32. Robert D. Silverman, "The multiple polynomial quadratic sieve," preprint, August, 1985.
33. Gustavus J. Simmons and Michael J. Norris, "How to cipher faster using redundant number systems," Sandia National Laboratories Report SAND80-1886 (1980).
34. J. W. Smith and S. S. Wagstaff, Jr., An extended precision operand computer. *Proceedings of the Twenty-First Southeast Region ACM Conference*, (1983), 209-216.
35. Hiromi Suyama, "Informal preliminary report (8)," 25 October 1985.
36. John T. Tate, "The arithmetic of elliptic curves," *Invent. Math.*, 23 (1974), 179-206.
37. H. C. Williams, "A $p + 1$ method of factoring," *Math. Comp.*, 39 (1982), 225-234.
38. H. C. Williams, "An overview of factoring," in *Advances in Cryptology, Proceedings of Crypto 83*, David Chaum, ed., Plenum Press, New York (1984), 71-80.
39. Marvin C. Wunderlich, "Factoring numbers on the Massively Parallel Computer," in *Advances in Cryptology, Proceedings of Crypto 83*, David Chaum, ed., Plenum Press, New York (1984), 87-102.
40. Marvin C. Wunderlich, "Implementing the continued fraction factoring algorithm on parallel machines," *Math. Comp.*, 44 (1985), 251-260.
41. John Brillhart, "Fermat's factoring method and its variants," *Congressus Numerantium*, 32 (1981), 29-48.