Parallel Algorithms for Integer Factorisation

Richard P. Brent
Computer Sciences Laboratory
Australian National University
Canberra, ACT 2601

Abstract

The problem of finding the prime factors of large composite numbers has always been of mathematical interest. With the advent of public key cryptosystems it is also of practical importance, because the security of some of these cryptosystems, such as the Rivest-Shamir-Adelman (RSA) system, depends on the difficulty of factoring the public keys.

In recent years the best known integer factorisation algorithms have improved greatly, to the point where it is now easy to factor a 60-decimal digit number, and possible to factor numbers larger than 120 decimal digits, given the availability of enough computing power.

We describe several algorithms, including the *elliptic curve method* (ECM), and the *multiple-polynomial quadratic sieve* (MPQS) algorithm, and discuss their parallel implementation. It turns out that some of the algorithms are very well suited to parallel implementation. Doubling the degree of parallelism (i.e. the amount of hardware devoted to the problem) roughly increases the size of a number which can be factored in a fixed time by 3 decimal digits.

Some recent computational results are mentioned – for example, the complete factorisation of the 617-decimal digit Fermat number $F_{11} = 2^{2^{11}} + 1$ which was accomplished using ECM.

1. Introduction

It has been known since Euclid's time (though first clearly stated and proved by Gauss in 1801) that any natural number N has a unique *prime power decomposition*

$$N = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_k^{\alpha_k}$$

($p_1 < p_2 < \cdots < p_k$ rational primes, $\alpha_j > 0$), and for many purposes we would like an efficient algorithm for computing this decomposition. Note that it is sufficient to have an algorithm for finding a nontrivial factor f of N, because this can be applied recursively to f and N/f to obtain the complete prime power decomposition of N.

Appeared in *Number Theory and Cryptography* (edited by J. H. Loxton), Cambridge University Press, 1990, 26–37.
E-mail address: `rpb@cslab.anu.edu.au` rpb115 typeset using TeX

1.1 Serial algorithms

A *polynomial time* algorithm would run in time $O(\log N)^c$ for some constant c. However, no such algorithm is known. The algorithms described in Sections 5 and 6 run in time $O(N^\epsilon)$ for any positive ϵ, in fact they are conjectured to run in time $O(\exp(c(\log N \log\log N)^{1/2}))$, where c is a certain constant.

Most useful factorisation algorithms fall into one of two classes –

A. The run time depends mainly on the size of N, the number being factored, and is not strongly dependent on the size of the factor found. Examples are –

   Lehman's algorithm [18] which has a rigorous worst-case run time bound $O(N^{1/3})$.

   Shanks's SQUFOF algorithm [38], which has expected run time $O(N^{1/4})$.

   Shanks's Class Group algorithm [34, 35] which has run time $O(N^{1/5+\epsilon})$ on the assumption of the Generalised Riemann Hypothesis.

   The Continued Fraction algorithm [25] and the Multiple Polynomial Quadratic Sieve algorithm [29], which under plausible assumptions have expected run time $O(\exp(c(\log N \log\log N)^{1/2}))$, where c is a constant (depending on details of the algorithm).

B. The run time depends mainly on the size of f, the factor found. (We can assume that $f \le N^{1/2}$.) Examples are –

   The trial division algorithm, which has run time $O(f \cdot (\log N)^2)$.

   The Pollard "rho" algorithm [1, 28] which under plausible assumptions has expected run time $O(f^{1/2} \cdot (\log N)^2)$.

   Lenstra's "Elliptic Curve Method" (ECM) [4, 22] which under plausible assumptions has expected run time $O(\exp(c(\log f \log\log f)^{1/2}) \cdot (\log N)^2)$, where c is a constant.

In these examples, the term $(\log N)^2$ is a generous allowance for the cost of performing arithmetic operations on numbers of size $O(N)$ or $O(N^2)$, and could theoretically be replaced by $(\log N)^{1+\epsilon}$ for any $\epsilon > 0$.

Algorithms in class B are useful for "naturally occurring" numbers N which are quite likely to have small factors. Note that the difficulty of factoring a number N by an algorithm in class B depends on the size of the *second-largest* prime factor p_{k-1} of N rather than on the size of the *largest* prime factor p_k. For randomly chosen N there is a 50 percent chance that $p_{k-1} < N^{0.212}$. Thus, if we have an algorithm in class B which can find factors of size 10^{22} in a reasonable time, there is a 50 percent chance that the algorithm will be able to completely factor a random number N of size about 10^{100}. (See [7] for an example, and [12, 16] for the theory.)

In cryptographic applications [33] the number N to be factored are not random. More likely they have been constructed with the intention of being difficult to factor. For such numbers, algorithms in class A are preferable. However, it is generally worth attempting to find small factors with an algorithm in class B before embarking on a long computation with an algorithm in class A.

1.2 Parallel algorithms

The time bounds mentioned above assume that only one arithmetic operation is performed at a time. A practical way of speeding up computations is by the use of parallelism. We would hope that an algorithm which required time T_1 on a computer with one processor could be implemented to run in time $T_P \sim T_1/P$ on a computer with P independent processors. This is not always the case, since it may be impossible to use all P processors effectively. However, it is often true for integer factorisation algorithms, provided that P is not too large.

The *speedup* of a parallel algorithm is $S = T_1/T_P$ and the *efficiency* is $E = S/P$. We aim for a linear speedup, i.e. $S = \Omega(P)$. If the speedup is linear in the number of processors P, then each processor is being used with efficiency bounded below by a positive constant.

There are several recent surveys of integer factorisation algorithms [8, 9, 13, 24, 29, 32]. In this paper we concentrate on the efficient parallel implementation of the algorithms.

2. Trial division

Trial division is a straightforward factorisation algorithm. We just try potential divisors $d = 2, 3, \ldots$ until one of the following occurs –

1. $d > N^{1/2}$, in which case N is prime; or
2. $d < N$ and $d \mid N$, in which case d is a nontrivial prime divisor of N; or
3. d exceeds some preassigned bound $B < N^{1/2}$, in which case all we can say is that any prime factor p of N satisfies $p > B$.

Naturally, refinements are possible. If N is odd, only odd d need be considered. Multiples of 3 may also be excluded if $N \neq 0 \bmod 3$. Carrying this process to its logical conclusion, we need only consider prime divisors d, but it is necessary to consider how the set of primes less than B is computed and whether there is any overall saving.

The simplest version of trial division takes time $O(p \cdot \log N)$ to find the smallest prime factor p of N. Here the factor "$\log N$" allows for division of the multiple-precision number N by single-precision trial divisors $d \leq p$. The run time might be reduced by a factor of $\log N$ or $\log p$ with various refinements.

The parallel implementation of trial division is extremely straightforward. With P processors we can perform up to P trials in parallel. Thus, provided $P \ll p$, a linear speedup is obtained.

In Sections 3 to 6 we assume that N is composite, since in practice this is easily checked using a probabilistic primality test [16, 32] which runs in time $O(\log N)^3$. It is also convenient to assume that all "small" factors of N have been removed by trial division.

3. The Pollard "rho" algorithm

Pollard's "rho" algorithm [28] uses an iteration of the form

$$x_{i+1} = f(x_i) \bmod N, \quad i \geq 0,$$

where N is the number to be factored, x_0 is a random starting value, and f is a polynomial with integer coefficients. In practice a quadratic polynomial

$$f(x) = x^2 + a$$

is used ($a \neq 0, -2 \bmod N$).

Let p be the smallest prime factor of N, and j the smallest positive index such that $x_{2j} = x_j \bmod p$. Making some plausible assumptions, it is easy to show that the expected value of j is $E(j) = O(p^{1/2})$. The argument is related to the well-known "birthday" paradox – the probability that $x_0, x_1, \ldots, x_k$ are all distinct mod p is approximately

$$(1 - 1/p) \cdot (1 - 2/p) \cdots (1 - k/p) \sim \exp(-k^2/(2p)),$$

and if $x_0, x_1, \ldots, x_k$ are not all distinct mod p then $j \leq k$.

In practice we do not know p in advance, but we can detect x_j by taking greatest common divisors. We simply compute $GCD(x_{2i} - x_i, N)$ for $i = 1, 2, \ldots$ and stop when a nontrivial GCD (necessarily a factor of N) is found.

Various refinements are possible. Because GCDs are more expensive than multiplications (mod N), it is preferable to avoid the computation of most of the GCDs by accumulating the product $\prod(x_{2i} - x_i) \bmod N$. Also, the choice of subscripts $2i$ and i here is not optimal [1].

The "rho" algorithm is an improvement over trial division in that it has (conjectured) expected run time $O(p^{1/2}(\log N)^2)$ to find a prime factor p of N. A disadvantage is that the run time is now only a (conjectured) expected value, not a rigorous bound.

An example of the success of a variation on the Pollard "rho" algorithm is the complete factorisation of the Fermat number $F_8 = 2^{2^8} + 1$ by Brent and Pollard [7].

Unfortunately, parallel implementation of the "rho" algorithm does not give linear speedup. Because the degree of x_i, regarded as a polynomial in x_0, is 2^i, it does not seem possible to use parallelism to speed up the computation of the sequence (x_i) by a significant amount. A plausible use of parallelism is to try several different pseudo-random sequences (generated by different polynomials f). If we have P processors and use P different sequences in parallel, the probability that the first k values in each sequence are distinct mod p is approximately

$$\exp(-k^2 P/(2p)),$$

so the speedup is $O(P^{1/2})$ and the efficiency is only $O(P^{-1/2})$.

4. The Pollard "p - 1" algorithm

Pollard's "$p - 1$" algorithm [24, 27] is based on Fermat's theorem

$$a^{p-1} = 1 \bmod p$$

for $0 < a < p$, p prime. Suppose that p is a prime factor of N and that E is a multiple of $p-1$. Then, from Fermat's theorem,

$$p \mid GCD(a^E - 1, N)$$

and $GCD(a^E - 1, N)$ gives us a factor (possibly trivial) of N.

Since p is not known in advance, the algorithm supposes that all prime power factors of $p-1$ are bounded above by some arbitrarily chosen number m. Then, taking E as a product of prime powers q^e, $q^e \le m$, we obtain a multiple of $p-1$. If $p-1$ has a prime factor greater than m, then the algorithm will generally fail to give a nontrivial factor of N.

Because E is very large, it is not actually computed. Instead, $a^E \bmod N$ is computed via a sequence of computations

$$a \leftarrow a^{q^e} \bmod N.$$

The work involved in such an exponentiation is $O(\log(q^e))$ multiplications mod N, so the total work involved is $O(m)$ multiplications mod N, and the time required is $O(m \cdot (\log N)^2)$.

The "$p-1$" algorithm is very effective in the fortunate case that $p-1$ has all "small" prime factors. For example, Baillie found the factor

$$p_{25} = 1155685395246619182673033$$

of the Mersenne number $M_{257} = 2^{257} - 1$ (claimed to be prime by Mersenne) using the "$p-1$" algorithm. In this case

$$p_{25} - 1 = 2^3 \cdot 3^2 \cdot 19^2 \cdot 47 \cdot 67 \cdot 257 \cdot 439 \cdot 119173 \cdot 1050151,$$

and $m \ge 1050151$ is sufficient.

A more extreme example: using the "$p-1$" algorithm we found the factor

$$p_{32} = 49858990580788843054012690078841$$

of the Mersenne number $M_{977} = 2^{977} - 1$. Here

$$p_{32} - 1 = 2^3 \cdot 5 \cdot 13 \cdot 19 \cdot 977 \cdot 1231 \cdot 4643 \cdot 74941 \cdot 1045397 \cdot 11535449,$$

but because we used a two-phase algorithm $m \ge 1045397$ was sufficient (see Section 5.4 for the idea of the second phase).

The examples just given are not typical. If we are unlucky, $(p-1)/2$ may be prime, so the worst case time bound for the "$p-1$" algorithm is no better than for trial division.

Parallel implementation of the "$p-1$" algorithm is difficult, because the inner loop seems inherently serial. At best, parallelism can speed up the multiple precision operations by a small factor (depending on $\log N$ but not on p).

In the next section we show that it is possible to overcome the main handicaps of the "$p-1$" algorithm, and obtain an algorithm which is easy to implement in parallel and does not depend on a lucky factorisation of $p-1$.

5. Lenstra's elliptic curve algorithm

Pollard's "$p-1$" algorithm may be regarded as an attempt to generate the identity in the multiplicative group of $F_p = GF(p)$. The motivation for H. W. Lenstra's elliptic curve algorithm (usually denoted "ECM") is as follows – if we can choose a "random" group G with order g close to p, we may be able to perform a computation similar to that involved in Pollard's "$p-1$" algorithm, working in G rather than in F_p. If all prime factors of g are less than the bound m then we find a factor of N. Otherwise, repeat with a different G (and hence, usually, a different g) until a factor is found.

A curve of the form

$$y^2 = x^3 + ax + b \tag{5.1}$$

over some field F is known as an *elliptic curve.* A more general cubic in x and y can be reduced to the form (5.1), which is known as the Weierstrass normal form, by rational transformations.

There is a well-known way of defining an Abelian group $(G, *)$ on an elliptic curve over a field. Formally, if $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ are points on the curve, then the point $P_3 = (x_3, y_3) = P_1 * P_2$ is defined by –

$$(x_3, y_3) = (\lambda^2 - x_1 - x_2,\ \lambda(x_1 - x_3) - y_1), \tag{5.2}$$

where

$$\lambda = \begin{cases} (3x_1^2 + a)/(2y_1) & \text{if } P_1 = P_2 \\ (y_1 - y_2)/(x_1 - x_2) & \text{otherwise.} \end{cases}$$

The identity element I in G is the "point at infinity".

The geometric interpretation is straightforward. We refer the reader to [14, 17] for an introduction to the theory of elliptic curves.

In Lenstra's algorithm [22] the field F is the finite field F_p of p elements, where p is a prime factor of N. The multiplicative group of F_p, used in Pollard's "$p-1$" algorithm, is replaced by the group G defined by (5.1) and (5.2). Since p is not known in advance, computation is performed in the *ring* of integers modulo N rather than in F_p. We can regard this as using a redundant representation for elements of F_p.

5.1 Computing inverses mod N

In order to implement (5.2) we need to compute inverses mod N. Suppose that x is given and we want to compute z such that $xz = 1 \bmod N$. This is easily done via the extended Euclidean algorithm applied to x and N, which gives u and v such that

$$ux + vN = GCD(x, N).$$

If $GCD(x, N) = 1$ then $ux = 1 \bmod N$, so $z = u$. If $GCD(x, N) > 1$ then $GCD(x, N)$ is a nontrivial factor of N, so we stop. It is curious that Lenstra's algorithm finds a factor of N precisely when an inverse computation breaks down (formally, when the identity element of G arises in a nontrivial way).

The cost of an extended GCD computation is about the same as that of 10 to 12 multiplications mod N (see [4, 19]).

5.2 One trial of Lenstra's algorithm

A *trial* is the computation involving one random group G. The steps involved are –

1. Choose x_0, y_0 and a randomly in $[0, N)$. This defines $b = y_0^2 - (x_0^3 + ax_0) \bmod N$. Set $P \leftarrow P_0 = (x_0, y_0)$.
2. For prime $q = 2, \ldots, m$ set $P \leftarrow P^{q^e}$ in the group G defined by a and b, where e is an exponent chosen as in Pollard's "$p - 1$" algorithm, If $P = I$ then a factor of N will have been found during an attempt to compute an inverse mod N.

The work involved is $O(m)$ group operations. Note that several trials can be performed in parallel.

5.3 The choice of m

Given $x \in F_p$, there are at most two values of $y \in F_p$ satisfying (5.1). Thus, allowing for the identity element, we have $g = |G| \leq 2p + 1$. Although this would be sufficient for an approximate analysis of ECM, a much stronger result, the *Riemann hypothesis for finite fields* [15], is known –

$$|g - p - 1| < 2p^{1/2}. \tag{5.3}$$

Making the (incorrect, but close enough) assumption that g behaves like a random integer distributed uniformly in $(p - 2p^{1/2}, p + 2p^{1/2})$, we may show that the optimal choice of m is $m = p^{1/\alpha}$, where

$$\alpha \sim (2 \ln p / \ln \ln p)^{1/2} \tag{5.4}$$

The expected run time is

$$T = p^{2/\alpha + o(1/\alpha)} \tag{5.5}$$

For details, see [4, 22]. From (5.5), we see that the exponent $2/\alpha$ should be compared with 1 (for trial division) or 1/2 (for Pollard's "rho" method). For $10^{10} < p < 10^{30}$, we have $\alpha \in (3.2, 5.0)$. Because of the overheads involved with ECM, a simpler algorithm such as Pollard's "rho" is preferable for finding factors of size up to about 10^{10} (see Figure 1 in [4]), but for larger factors the asymptotic advantage of ECM becomes apparent.

5.4 A second phase

Both the Pollard "$p - 1$" and Lenstra elliptic curve algorithms can be speeded up by the addition of a second phase. The idea of the second phase is to find a factor in the case that the first phase terminates with a group element $P \neq I$, such that $|\langle P \rangle|$ is reasonably small (say $O(m^2)$). (Here $\langle P \rangle$ is the cyclic group generated by P.) There are several possible implementations of the second phase. One of the simplest uses a pseudorandom walk in $\langle P\rangle$. By the birthday paradox argument, there is a good chance that two points in the random walk will coincide after $O(|\langle P\rangle|)^{1/2}$ steps, and when this occurs a nontrivial factor of N can usually be found. Details may be found in [4, 24].

The use of a second phase provides a significant speedup in practice, but does not change the asymptotic time bound (5.5). Similar comments apply to other implementation details, such as ways of avoiding most divisions and speeding up group operations [11, 23, 24], ways of choosing good initial points [24, 37], and ways of using preconditioned polynomial evaluation [4, 26, 40].

5.5 Parallel implementation of ECM

So long as the expected number of trials is much larger than the number P of processors available, linear speedup is possible by performing P trials in parallel. In fact, if T_1 is the expected run time on one processor, then the expected run time on a parallel machine with P processors is

$$T_P = T_1/P + O(T_1^{1/2+\epsilon}) \tag{5.6}$$

The bound (5.6) applies on single-instruction multiple-data (SIMD) machine if we use the Montgomery-Chudnovsky form [11, 24]

$$by^2 = x^3 + ax^2 + x$$

instead of the Weierstrass normal form (5.1) in order to avoid divisions.

In practice, it may be difficult to perform P trials in parallel because of storage limitations. The second phase requires much more storage than the first phase. Fortunately, there are several possibilities for making use of parallelism during the second phase of each trial. Our implementation performs the first phase of P trials in parallel, but the second phase of each trial sequentially, using P processors to speed up the evaluation of the polynomials

$$\prod_{i,j}(x_i - \overline{x}_j)$$

which constitute most of the work during the second phase.

6. Quadratic sieve algorithms

Quadratic sieve algorithms belong to a wide class of algorithms which try to find two integers x and y such that

$$x^2 = y^2 \bmod N \tag{6.1}$$

Once such x and y are found, there is a good chance that $GCD(x-y, N)$ is a nontrivial factor of N.

One way to find x and y is to find a set of relations of the form

$$u_i^2 = v_i^2 w_i \bmod N, \tag{6.2}$$

where the w_i have all their prime factors in a moderately small set of primes (called the *factor base*). Each relation (6.1) gives a row in matrix M whose columns correspond to the primes in the factor base. Once enough rows have been generated, we can use Gaussian elimination in F_2 [39] to find a linear dependency (mod 2) between a set of rows of M. Multiplying the corresponding relations now gives a relation of the form (6.1).

In quadratic sieve algorithms the numbers w_i are the values of one (or more) polynomials with integer coefficients. This makes it easy to factorise the w_i by *sieving*. For details of the process, we refer to the recent papers [10, 20, 29, 30, 31, 36]. The conclusion is that the best quadratic sieve algorithms (such as the *multiple polynomial quadratic sieve algorithm* MPQS [29]) can, under plausible assumptions, factor a number N in time $O(\exp(c(\log N \log\log N)^{1/2}))$, where $c \sim 1$. The constants involved are such that MPQS is usually faster than ECM if N is the product of two primes which both exceed $N^{1/3}$. (The exponent "1/3" is empirical, based on experience with $N < 10^{100}$.)

Although MPQS is a probabilistic algorithm, depending on the factorisation of the numbers w_i over the factor base, it is much more predictable than ECM. This is because MPQS depends on obtaining a large number of factorisations of w_i, so the law of large numbers applies and we can predict with confidence how much work will be required, as a function of N. ECM, on the other hand, depends on *one* unlikely event occurring, so the run time behaves like an exponentially distributed random variable whose expectation is a function of p, the factor eventually found. In practice, we know N but not p in advance.

The reader may be surprised that algorithms as different as MPQS and ECM have similar expected time bounds. However, this is not really so surprising. MPQS requires $O(B)$ factorisations of numbers w_i of size $O(N^{1/2+\epsilon})$ over the factor base of size B, and the work per trial is small (because of the sieving process). On the other hand, ECM requires only one number (the order of the group G) to factor completely over primes not exceeding m, but the work per trial is $O(m)$. Use of "partial relations", i.e. incompletely factored w_i, in MPQS is analogous to the second phase of ECM.

6.1 Parallel implementation of MPQS

Like ECM, MPQS is ideally suited to parallel implementation. Different processors may use different polynomials, or sieve over different intervals with the same polynomial. Thus, there is a linear speedup so long as the number of processors is not much larger than the size of the factor base. The process requires very little communication between processors. Each processor can generate relations and forward them to some central collection point. This has been demonstrated most clearly by A. K. Lenstra and M. S. Manasse [20] who distribute their program and collect relations via electronic mail. The processors are scattered around the world – anyone with access to electronic mail and a C compiler can volunteer to contribute. (The final stage – Gaussian elimination to combine the relations – is not so easily distributed. However, in practice it is only a small fraction of the computation.)

7. Some recent computational results

In the process of proving the non-existence of an odd perfect number less than 10^{300} [5, 6], we needed many factorisations of numbers of the form $p^n - 1$, where p and n are prime. For example, the factorisation

$$c_{101} = (467^{41} - 1)/(466 \cdot 1022869) = 4089568263561830388113662969166474269 \cdot p_{65}$$

was found by ECM.

We recently [2] completed the factorisation of the 617-decimal digit Fermat number $F_{11} = 2^{2^{11}} + 1$. In fact

$$F_{11} = 319489 \cdot 974849 \cdot 167988556341760475137 \cdot 3560841906445833920513 \cdot p_{564}$$

where the 21-digit and 22-digit prime factors were found using ECM, and p_{564} is a 564-decimal digit prime. The factorisation required about 360 million multiplications mod N, which took less than 2 hours on a Fujitsu VP 100 vector processor.

Using the MPQS algorithm and their worldwide distributed network [20], Lenstra and Manasse (with many assistants, including the present author) have factorised several numbers larger than 10^{100}, the largest (at the time of writing) having 106 decimal digits. For example, the most recently completed was the 103-decimal digit number

$$(2^{361} + 1)/(3 \cdot 174763) = 6874301617534827509350575768454356245025403 \cdot p_{61}$$

Such factorisations require many years of CPU time, but an "elapsed time" of only a month or so because of the number of different processors which are working in parallel, using machine cycles which would otherwise be idle.

Lenstra and Manasse [21] recently announced the factorisation of the 122-decimal digit number $c_{122} = (7^{149} + 1)/(2^3 \cdot 10133)$, in fact

$$c_{122} = 4733843335518992927911065093183780611982900857392850 1623 \cdot p_{66}$$

This impressive factorisation was obtained using an unpublished algorithm, the *Number Field Sieve* (NFS) due to J. M. Pollard, A. K. Lenstra and H. W. Lenstra, Jr. (Unlike ECM or MPQS, the NFS algorithm took advantage of the special form of c_{122}, so it is not clear whether a 122-digit number intended for use in a public key cryptosystem could be factorised in a comparable time.) Since the NFS algorithm uses similar ideas to the MPQS algorithm, it should be possible to implement it equally well on a parallel machine.

Remark

We take this opportunity to announce the availability of an integer factorisation program written in Turbo Pascal for the IBM PC [3].

References

1. R. P. Brent, "An improved Monte Carlo factorization algorithm", *BIT* 20 (1980), 176-184.
2. R. P. Brent, "Factorization of the eleventh Fermat number (preliminary report)", *AMS Abstracts* 10 (1989), 89T-11-73.
3. R. P. Brent, *Factor: an integer factorization program for the IBM PC*, Report TR-CS-89-23, Computer Sciences Laboratory, Australian National University, Oct. 1989. Available from the author.
4. R. P. Brent, "Some integer factorization algorithms using elliptic curves", *Australian Computer Science Communications* 8 (1986), 149-163.
5. R. P. Brent and G. L. Cohen, "A new lower bound for odd perfect numbers", *Mathematics of Computation*, July 1989.
6. R. P. Brent, G. L. Cohen and H. J. J. te Riele, *Improved techniques for lower bounds for odd perfect numbers*, to appear as a Technical Report, Computer Sciences Laboratory, Australian National University, August 1989.
7. R. P. Brent and J. M. Pollard, "Factorization of the eighth Fermat number", *Mathematics of Computation* 36 (1981), 627-630.
8. J. Brillhart, D. H. Lehmer, J. L. Selfridge, B. 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*, American Mathematical Society, Providence, Rhode Island, second edition, 1985.
9. D. A. Buell, "Factoring: algorithms, computations, and computers", *J. Supercomputing* 1 (1987), 191-216.
10. T. R. Caron and R. D. Silverman, "Parallel implementation of the quadratic sieve", *J. Supercomputing* 1 (1988), 273-290.
11. D. V. Chudnovsky and G. V. Chudnovsky, *Sequences of numbers generated by addition in formal groups and new primality and factorization tests*, Dept. of Mathematics, Columbia University, July 1985.
12. K. Dickman, "On the frequency of numbers containing prime factors of a certain relative magnitude", *Ark. Mat., Astronomi och Fysik*, 22A, 10 (1930), 1-14.
13. R. K. Guy, "How to factor a number", *Congressus Numerantum XVI*, Proc. Fifth Manitoba Conference on Numerical Mathematics, Winnipeg, 1976, 49-89.
14. K. F. Ireland and M. Rosen, *A Classical Introduction to Modern Number Theory*, Springer-Verlag, 1982, Ch. 18.
15. J-R. Joly, "Equations et variétés algébriques sur un corps fini", *L'Enseignement Mathématique* 19 (1973), 1-117.
16. D. E. Knuth, *The Art of Computer Programming*, Vol. 2, Addison Wesley, 2nd edition, 1982.
17. S. Lang, *Elliptic Curves – Diophantine Analysis*, Springer-Verlag, 1978.
18. R. S. Lehman, "Factoring large integers", *Mathematics of Computation* 28 (1974), 637-646.
19. D. H. Lehmer, "Euclid's algorithm for large numbers", *Amer. Math. Monthly* 45 (1938), 227-233.
20. A. K. Lenstra and M. S. Manasse, *Factoring by electronic mail*, preprint, 10 June 1989.
21. A. K. Lenstra and M. S. Manasse, personal communication, 28 August 1989.
22. H. W. Lenstra, Jr., "Factoring integers with elliptic curves", *Ann. of Math.* (2) 126 (1987), 649-673.

23. P. L. Montgomery, "Modular multiplication without trial division", *Mathematics of Computation* 44 (1985), 519-521.
24. P. L. Montgomery, "Speeding the Pollard and elliptic curve methods of factorization", *Mathematics of Computation* 48 (1987), 243-264.
25. M. A. Morrison and J. Brillhart, "A method of factorization and the factorization of F_7", *Mathematics of Computation* 29 (1975), 183-205.
26. M. Paterson and L. Stockmeyer, "On the number of nonscalar multiplications necessary to evaluate polynomials", *SIAM J. on Computing* 2 (1973), 60-66.
27. J. M. Pollard, "Theorems in factorization and primality testing", *Proc. Cambridge Philos. Soc.* 76 (1974), 521-528.
28. J. M. Pollard, "A Monte Carlo method for factorization", *BIT* 15 (1975), 331-334.
29. C. Pomerance, "Analysis and comparison of some integer factoring algorithms", in *Computational Methods in Number Theory* (edited by H. W. Lenstra, Jr. and R. Tijdeman), Math. Centrum Tract 154, Amsterdam, 1982, 89-139.
30. C. Pomerance, J. W. Smith and R. Tuler, "A pipeline architecture for factoring large integers with the quadratic sieve algorithm", *SIAM J. on Computing* 17 (1988), 387-403.
31. H. J. J. te Riele, W. Lioen and D. Winter, *Factoring with the quadratic sieve on large vector computers*, Report NM-R8805, Centre for Mathematics and Computer Science, Amsterdam, 1988.
32. H. Riesel, *Prime Numbers and Computer Methods for Factorization*, Birkhäuser, Boston, 1985.
33. R. L. Rivest, A. Shamir and L. Adelman, "A method for obtaining digital dignatures and public-key cryptosystems", *Communications of the ACM* 21 (1978), 120-126.
34. R. J. Schoof, "Quadratic fields and factorization", in *Studieweek Getaltheorie en Computers* (edited by J. van de Lune), Math. Centrum, Amsterdam, 1980, 165-206.
35. D. Shanks, "Class number, a theory of factorization, and genera", *Proc. Symp. Pure Math.* 20, American Math. Soc., 1971, 415-440.
36. R. D. Silverman, "The multiple polynomial quadratic sieve", *Mathematics of Computation* 48 (1987), 329-339.
37. H. Suyama, *Informal preliminary report* (8), personal communication, October 1985.
38. M. Voorhoeve, "Factorization", in *Studieweek Getaltheorie en Computers* (edited by J. van de Lune), Math. Centrum, Amsterdam, 1980, 61-68.
39. D. Wiedemann, "Solving sparse linear equations over finite fields", *IEEE Trans. Inform. Theory* 32 (1986), 54-62.
40. S. Winograd, "Evaluating polynomials using rational auxiliary functions", *IBM Technical Disclosure Bulletin* 13 (1970), 1133-1135.