Primality Testing and Integer Factorisation

Richard P. Brent, FAA
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 recent algorithms for primality testing and factorisation, give examples of their use and outline some applications.

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*

$$N = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_k^{\alpha_k} \tag{1.1}$$

($p_1 < p_2 < \cdots < p_k$ rational primes, $\alpha_j > 0$). The prime powers $p_j^{\alpha_j}$ are called *components* of N, and we write $p_j^{\alpha_j} \| N$. To compute the prime power decomposition we need –

1. An algorithm to test if an integer N is prime.
2. An algorithm to find a nontrivial factor f of a composite integer N.

Given these there is a simple recursive algorithm to compute (1.1): if N is prime then stop, otherwise

1. find a nontrivial factor f of N;
2. apply the algorithm recursively to f and N/f;
3. put the pieces (the prime power decompositions of f and N/f) together to obtain the prime power decomposition of N.

Presented at the Australian Academy of Science Annual General Meeting Symposium on *The Role of Mathematics in Science*, Canberra, 20 April 1990. Appeared in the *Proceedings*, 1991, 14–26.

E-mail address: `rpb@cslab.anu.edu.au` rpb120 typeset using TEX

2. Primality Testing

We can test a number N for primality by dividing by all primes up to $N^{1/2}$, but this is too slow unless N is small. Arithmetic operations on numbers in $[0, \ldots, N]$ take time $O((\log N)^2)$ on a serial computer, since the binary or decimal representation of N has $O(\log N)$ digits. We would like an algorithm for testing the primality of N in time

$$O((\log N)^c)$$

for some constant c. Such an algorithm is called a *polynomial time* algorithm because its run time is bounded by a polynomial in the length of the input.

2.1 Use of Fermat's Theorem

Fermat's "little" Theorem states that if p is prime and $a \neq 0 \pmod p$ then

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

We can often verify that a number n is *composite* using Fermat's theorem: it is sufficient to find a positive integer $a < n$ such that

$$a^{n-1} \neq 1 \pmod n$$

Unfortunately, we can never prove *primality* using Fermat's theorem because

$$a^{n-1} = 1 \pmod n \tag{2.1}$$

does *not* imply that n is prime. There even exist composite n such that (2.1) holds for all a relatively prime to n. Such n are called *Carmichael numbers*; they are squarefree composite numbers such that $p - 1 | n - 1$ for every prime factor p of n. Examples are

$$n = 3 \cdot 11 \cdot 17 = 561$$

and

$$n = 7 \cdot 13 \cdot 19 = 1729$$

(the latter being of some historical interest).

2.2 A rigorous primality test

To prove that n is prime it is sufficient to find a *primitive root* of n, *i.e.* an integer a such that

$$a^{n-1} = 1 \pmod n \tag{2.2}$$

and

$$a^j \neq 1 \pmod n \tag{2.3}$$

for $1 \le j < n - 1$. To verify (2.3) it is sufficient to check that

$$a^{(n-1)/p} \neq 1 \pmod n \tag{2.4}$$

for each prime factor p of $n - 1$.

If n is prime, there is usually no difficulty in finding a primitive root of n by trial and error. The difficulty in applying (2.4) lies in factorising $n-1$. It is sometimes possible to get by with partial factorisations of $n \pm 1$ (see Theorem 12 of [10]), but it would be preferable to be able to test primality of n without having to factorise (even partially) numbers close to n.

2.3 A probabilistic polynomial time primality test

If n is an odd prime then

$$n - 1 = 2^k q$$

for some positive integer k and odd integer q. For any integer a, let

$$S(a) = (a^q, a^{2q}, a^{2^2 q}, \ldots, a^{2^k q}) \bmod n.$$

We say that n *passes Test(a)* if $S(a)$ has the form

$$(1, 1, \ldots, 1)$$

or

$$(?, \ldots, ?, -1, 1, \ldots, 1),$$

i.e. if either

$$a^{2^j q} = 1 \pmod{n}$$

for all $j = 0, 1, \ldots, k$, or there is some j in the range $0 \le j < k$ such that

$$a^{2^j q} = -1 \pmod{n}.$$

If n is prime and $1 < a < n$, then n passes Test(a). The converse is usually true, as shown by the following result (essentially due to Rabin: see [19, Sec. 4.5.4]).

Theorem 1. *If n is an odd composite number then the number of a in the range $1 < a < n$ for which n passes Test(a) is less than $(n-2)/4$.*

A probabilistic interpretation of Theorem 1 is:

Corollary. *If n is an odd composite and a is chosen randomly from $\{2, 3, \ldots, n-1\}$ then the probability that n passes Test(a) is less than 1/4.*

From the Corollary, we can construct a simple, polynomial time primality test which has a positive (but arbitrarily small) probability of giving the wrong answer. Suppose an error probability of ϵ is acceptable. Choose m such that $4^{-m} \le \epsilon$, and select $a_1, \ldots, a_m$ randomly and independently from $\{2, 3, \ldots, n-1\}$. If n fails Test(a_i) for some i then n is *certainly* composite, but if n passes Test(a_i) for $i = 1, \ldots, m$ then n is *probably* prime (in the sense that the probability that a composite n will erroneously be declared to be prime is less than ϵ).

We conclude that for all practical purposes we can test primality in polynomial time. However, probabilistic algorithms are not to everyone's taste. It would be more satisfying to have a strictly deterministic polynomial time test for primality. Recent developments which are beyond the scope of this paper include:

1. Probabilistic algorithms which always give the correct answer – only the run time depends on chance, and is expected to be polynomial [1, 15].
2. A deterministic algorithm which is "almost" polynomial time – its run time is

$$O((\log n)^{c \log\log\log n})$$

[13, 14, 32]. For practical purposes the exponent $c \log\log\log n$ is essentially a constant, and implementations of the algorithm have proved the primality of numbers of several hundred decimal digits.

3. Public Key Cryptography

Large primes have an interesting practical application – they can be used to construct *public key* cryptosystems (also known as *asymmetric* cryptosystems and *open encryption key* cryptosystems). In this section we outline the use of large primes and primality testing algorithms in the construction of a Rivest-Shamir-Adleman (RSA) cryptosystem [33]. For details and alternative systems we refer to [32, 35]

Public key cryptosystems are based on *trapdoor* functions (also known as *one-way* functions). Let S be a finite set. A trapdoor function is an invertible function

$$f : S \to S$$

such that $f(x)$ is easy to compute but the inverse function $f^{-1}(y)$ is hard to compute.

3.1 Example of a trapdoor function

An example of a trapdoor function of the form used in the RSA cryptosystem is

$$f(x) = x^k \bmod N,$$

where $N = pq$ (a product of two large primes), $S = \{0, 1, \ldots, N-1\}$, $k > 1$, and GCD $(k, \lambda) = 1$, $\lambda =$ LCM $(p-1, q-1) = (p-1)(q-1)/$GCD $(p-1, q-1)$. We assume that k and N are publicly known but p, q and λ are not.

The inverse function is

$$f^{-1}(y) = y^{k'} \bmod N,$$

where $kk' = 1 \pmod{\lambda}$. Clearly it is easy to compute $f^{-1}(y)$ if k' is known. The assumption underlying the RSA cryptosystem is that it is hard to compute $f^{-1}(y)$ without knowing k'. Note that knowledge of p, q or λ makes it easy to compute k'.

3.2 Construction of a trapdoor function

The steps involved in the construction of a good trapdoor function of the form above are:

1. Test sufficiently large random integers using a probabilistic primality test to find distinct large primes p and q such that
   a) $|p - q|$ is large;
   b) $p = -1 \pmod{12}$, $q = -1 \pmod{12}$; and
   c) $p' = (p-1)/2$, $p'' = (p+1)/12$,

$q' = (q-1)/2, \quad q'' = (q+1)/12$ are prime (see below).

2. Compute $N = pq$ and $\lambda = 2p'q'$.
3. Choose a random k relatively prime to λ such that $k-1$ is not a multiple of p' or q'. (For a simpler but less symmetric system, just fix $k = 3$.)
4. Apply the extended Euclidean algorithm [19] to k and λ to find k', λ' such that $0 < k' < \lambda$ and
$$kk' + \lambda\lambda' = 1$$
(If $k = 3$ there is no need to apply the Euclidean algorithm as $k' = (\lambda+1)/3$.)
5. Destroy all evidence of p, q, λ, λ'.
6. Make (k, N) public but keep k' secret.

Primality of p', p'', q', and q'' is not essential. All that is necessary is that $p \pm 1$ and $q \pm 1$ each have at least one large prime factor. Otherwise it would be easy to factorise N using Pollard's "$p \pm 1$" algorithm (see Section 4.4).

The probability that a randomly chosen integer in $[1, M]$ is prime is $\sim 1/\ln M$. Thus, the expected number of random trials required to find p is conjectured to be $O((\log N)^3)$ (the result would be true if p, p' and p'' were independent). On this assumption, the expected time required to construct the trapdoor function is $O((\log N)^6)$.

3.3 The RSA cryptosystem

Suppose a *sender* A wants to send a message M to a *receiver* B. B will already have chosen a trapdoor function f as described above, and published his *public key* (k, N), so we can assume that both A and any potential adversary knows k and N.

A splits the message M into blocks of $\lfloor \log_2 N \rfloor$ bits (padded on the *right* with zeros for the last block), and treats each block as an integer $x \in \{0, 1, \ldots, N-1\}$. A computes $y = x^k \bmod N$ and transmits y to B. B, who knows the secret key k', computes $x = y^{k'} \bmod N$. An adversary who intercepts the encrypted message should be unable to decrypt it without knowledge of k'.

3.4 Security of the RSA system

A possible cause for concern is that certain x are *fixed points*, *i.e.* $f(x) = x$, and for such x the message is not concealed at all by encryption. However, it may be shown that with our choice of f there are only 9 fixed points, so the probability of encountering one is $O(1/N)$.

There is no known way of cracking the RSA system without essentially factorising N. This is provable for a slight modification of the system (with $k = 2$). Unfortunately, it has not been proved that the factorisation problem is difficult. All we can say is that many mathematicians and computer scientists have studied the problem, and no polynomial time algorithm has been published. (If one were found now, would it be published or kept secret ?)

4. Integer Factorisation Algorithms

There are many algorithms for finding a nontrivial factor f of a composite integer N. The most useful algorithms fall into one of two classes –

A. The run time depends mainly on the size of N, and is not strongly dependent on the size of f. Examples are –

   Lehman's algorithm [21] which has a rigorous worst-case run time bound $O(N^{1/3})$.

   Shanks's SQUFOF algorithm [39], which has expected run time $O(N^{1/4})$.

   Shanks's Class Group algorithm [34, 36] which has run time $O(N^{1/5+\epsilon})$ on the assumption of the Generalised Riemann Hypothesis.

   The Continued Fraction algorithm [26] 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).

   The Number Field Sieve algorithm [22] which under plausible assumptions has expected run time $O(\exp(c(\log N)^{1/3}(\log\log N)^{2/3}))$, where c is a constant, provided N has a suitable form (see Section 4.7).

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

   Lenstra's "Elliptic Curve Method" (ECM) [4, 24] 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 which are $O(N)$ or $O(N^2)$.

Our survey of integer factorisation algorithms is necessarily cursory. For more information the reader is referred to [6, 11, 16, 25, 29, 30].

4.1 Pollard's "rho" algorithm

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

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

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 ($a \ne 0 \pmod N$).

Let p be the smallest prime factor of N, and j the smallest positive index such that $x_{2j} = x_j \pmod p$. 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 $j \le k$.

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 GCD greater than 1 is found.

Occasionally two or more prime factors may be found simultaneously, *i.e.* $f =$ GCD $(x_{2i} - x_i, N)$ is composite ($f = N$ is possible). In this (unlikely) event we have to try again with a different x_0 or a different polynomial f.

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 [9]. In fact

$$F_8 = 1238926361552897 \cdot p_{62},$$

where p_{62} is a 62-digit prime [3].

4.2 The advantages of a group operation

The Pollard rho algorithm takes

$$x_{i+1} = f(x_i) \bmod N$$

where f is a pseudorandom polynomial. Suppose instead that

$$x_{i+1} = x_1 * x_i$$

where "$*$" is an *associative* operator, *i.e.*

$$x * (y * z) = (x * y) * z \ .$$

Then we can compute x_n in $O(\log n)$ steps by the *binary powering* method [19].

4.3 Computation of the identity mod p

Let m be some bound assigned in advance, and let E be the product of all maximal prime powers q^e, $q^e \le m$. If the cyclic group $<x_1>$ has order g whose prime power components are bounded by m, then $g|E$ and

$$x_1^E = I,$$

where I is the group identity.

We may consider a group defined mod p but work mod N, where p is an unknown divisor of N. This amounts to using a redundant representation for the group elements. When we compute the identity I, its representation mod N may allow us to compute p via a GCD computation (compare the Pollard rho algorithm). We give two examples below: Pollard's $p-1$ algorithm and Lenstra's elliptic curve method.

4.4 Pollard's $p-1$ algorithm

Pollard's "$p-1$" algorithm [27] may be regarded as an attempt to generate the identity in the multiplicative group of $F_p = GF(p)$. Here "$*$" is just multiplication mod p, so (by Fermat's theorem) $g|p-1$ and

$$x_1^E = I \iff x_1^E = 1 \pmod{p},$$

so

$$x_1^E = I \Rightarrow p|\text{GCD } (x_1^E - 1, N)$$

The "$p-1$" algorithm is very effective in the fortunate case that $p-1$ has no large 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 $m \geq 1050151$ is sufficient.

In the worst case, when $(p-1)/2$ is prime, the "$p-1$" algorithm is no better than trial division. Since the group has fixed order $p-1$ there is nothing to be done except try a different algorithm. For example, we might try Pollard's "$p+1$" algorithm [25], which uses a group of order $p+1$ (essentially a subgroup of the multiplicative group of $GF(p^2)$). We refer to a combination of Pollard's "$p-1$" and "$p+1$" algorithms as the "$p \pm 1$" algorithm.

4.5 Lenstra's elliptic curve algorithm

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. This is the motivation for H. W. Lenstra's *elliptic curve algorithm* (usually denoted "ECM").

A curve of the form

$$y^2 = x^3 + ax + b \tag{4.1}$$

over some field F is known as an *elliptic curve*. A more general cubic in x and y can be reduced to the form (4.1), 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 –

$$(x_3, y_3) = (\lambda^2 - x_1 - x_2,\ \lambda(x_1 - x_3) - y_1), \tag{4.2}$$

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: the straight line P_1P_2 intersects the elliptic curve at a third point $P_3' = (x_3, -y_3)$, and P_3 is the reflection of P_3' in the x-axis. We refer the reader to [17, 20] for an introduction to the theory of elliptic curves.

In Lenstra's algorithm [24] 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 (4.1) and (4.2). Since p is not known in advance, computation is performed in the *ring* Z/NZ of integers modulo N rather than in F_p. We can regard this as using a redundant representation for elements of F_p.

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 $q \le m$ set $P \leftarrow P^{q^e}$ in the group G defined by a and b, where e is an exponent chosen as in Section 4.3. If $P = I$ then the trial succeeds as a factor of N will have been found during an attempt to compute an inverse mod N. Otherwise the trial fails.

The work involved in a trial is $O(m)$ group operations. There is a tradeoff involved in the choice of m, as a trial with large m is expensive, but a trial with small m is unlikely to succeed.

Given $x \in F_p$, there are at most two values of $y \in F_p$ satisfying (4.1). Thus, allowing for the identity element, we have $g = |G| \le 2p + 1$. A much stronger result, the *Riemann hypothesis for finite fields* [18], is known –

$$|g - p - 1| < 2p^{1/2}. \tag{4.3}$$

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

$$\alpha \sim (2 \ln p / \ln \ln p)^{1/2} \tag{4.4}$$

It follows that the expected run time is

$$T = p^{2/\alpha + o(1/\alpha)}. \tag{4.5}$$

For details, see [4, 24]. From (4.5), 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. The following two examples illustrate the power of ECM.

1. In the application described in Section 5, we needed many factorisations of numbers of the form $p^n - 1$, where p and n are prime. For example, the factorisation

$$4089568263561830388113662969166474269 \cdot p_{65}$$

of the 101-digit number $(467^{41} - 1)/(466 \cdot 1022869)$ was found by ECM.

2. We recently [5] 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.

4.6 Quadratic sieve algorithms

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

$$x^2 = y^2 \pmod N \tag{4.6}$$

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

$$u_i^2 = v_i^2 w_i \pmod N, \tag{4.7}$$

where the w_i have all their prime factors in a moderately small set of primes (called the *factor base*). Each relation (4.7) 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 sparse Gaussian elimination in F_2 [40] 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 (4.6).

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 [12, 23, 30, 31, 37]. 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 $N^{1/3}$.

MPQS has been used to obtain many spectacular factorisations [10, 31, 37]. Lenstra and Manasse [23] (with many assistants scattered around the world) have factorised several numbers larger than 10^{100}, the largest (at the time of writing) having 107 decimal digits. For example, a recent factorisation 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.

4.7 The number field sieve (NFS)

Our numerical examples have all involved numbers of the form

$$a^e \pm b, \tag{4.8}$$

for small a and b, although the ECM and MPQS factorisation algorithms do not take advantage of this special form.

The *number field sieve* (NFS) is a new algorithm which does take advantage of the special form (4.8). In concept it is similar to the quadratic sieve algorithm, but it works over an algebraic number field defined by a, e and b. We refer the interested reader to Lenstra *et al* [22] for details, and merely give an example from [22] to show the power of the algorithm. For an introduction to the relevant concepts of algebraic number theory, see [38].

Consider the 138-decimal digit number

$$n = 2^{457}+1$$

as a candidate for factoring by NFS. Note that $8n = m^5 + 8$, where $m = 2^{92}$. We work in the number field $Q(\alpha)$, where α satisfies

$$\alpha^5 + 8 = 0,$$

and in the ring of integers of $Q(\alpha)$. Because

$$m^5 + 8 = 0 \pmod{n},$$

the mapping $\phi : \alpha \mapsto m \bmod n$ is a ring homomorphism from $Z[\alpha]$ to Z/nZ.

The idea is to search for pairs of small coprime integers u and v such that both the algebraic integer $u+\alpha v$ and the (rational) integer $u+mv$ can be factorised. (The factor base now includes prime ideals and units as well as rational primes.) Because

$$\phi(u+\alpha v) = (u+mv) \pmod{n},$$

each such pair gives a relation analogous to (4.7).

The prime ideal factorisation of $u + \alpha v$ corresponds to the factorisation of the norm $u^5 - 8v^5$. Thus, we have to factor simultaneously two integers $u + mv$ and $|u^5 - 8v^5|$. Note that, for moderate u and v, both these integers are much smaller than n, in fact they are $O(n^{1/d})$, where $d = 5$ is the degree of the algebraic number field. (The optimal choice of d is discussed in [22].)

Using these ideas, Lenstra *et al* [22] recently factorised n, obtaining

$$n = 3 \cdot p_{49} \cdot p_{89},$$

where

$$p_{49} = 6885357560205319573060633896800918448254904729193$$

is a 49-digit prime and p_{89} is an 89-digit prime. The whole computation took about nine weeks on a network of several hundred CVAX processors. Because the NFS algorithm took advantage of the special form of c_{138}, the reader should not assume that a 138-digit number intended for use in a public key cryptosystem could be factorised in a comparable time.

It is interesting to note that extrapolation of the times required by NFS for several factorisations in [22] suggests that a number of 155 decimal digits such as $F_9 = 2^{2^9} + 1$ could be completely factorised by NFS in about six months if the data-handling problems and difficulties of the sparse Gaussian elimination phase could be overcome. We would not be surprised if F_9 were completely factored by the time this paper appears in print. (Postscript: it has been.)

5. A Mathematical Application of Factorisation

In Section 2 we described a practical application of primality testing algorithms. In this section we describe a mathematical application of primality testing and integer factorisation algorithms. The application is of historical interest but not (as far as we know) of any practical value.

Let $\sigma(N)$ be the sum of the positive divisors of an integer N. For example, $\sigma(28) = 1+2+4+7+14+28 = 56$. An integer N is called *perfect* if $\sigma(N) = 2N$. It is known that an *even* number is perfect if and only if it has the form $N = 2^{k-1}(2^k - 1)$, where $2^k - 1$ is a (Mersenne) prime. For example, $6 = 2 \cdot 3$ and $28 = 4 \cdot 7$ have this form and are perfect. No odd perfect number has ever been found, nor has it been proved that none exists.

The function $\sigma(N)$ is *multiplicative*: it satisfies

$$\sigma(mn) = \sigma(m)\sigma(n)$$

for all relatively prime, positive m, n. Also, it is easy to see that

$$\sigma(p^\alpha) = 1 + p + p^2 + \cdots + p^\alpha = \frac{p^{\alpha+1} - 1}{p - 1}$$

for prime p and $\alpha > 0$. Thus, if N has the prime power decomposition (1.1) we can easily evaluate

$$\sigma(N) = \prod_{j=1}^{k} \sigma(p_j^{\alpha_j}).$$

There is an algorithm which will find an odd perfect number less than a given bound B (if one exists), or prove that there is none, in much less time than would be required to check each odd $N < B$ explicitly. To outline the algorithm, suppose that $N < B$ is an odd perfect number with prime power decomposition (1.1). Using

$$2 = \frac{\sigma(N)}{N} < \prod_{j=1}^{k} \frac{1}{1 - 1/p_j} \ ,$$

it is easy to show that N must be divisible by a reasonably small prime $p = o(\log B)$, which gives a finite number of possible components p^α. Now

$$p^\alpha \| N \Rightarrow \sigma(p^\alpha) | 2N,$$

so factorisation of $\sigma(p^\alpha)$ gives a number of primes which divide N. Proceeding in this way we deduce that more and more primes divide N, until eventually a contradiction to $N < B$ is obtained, or (optimistically) we converge to a finite set of primes which do divide an odd perfect number. In practice the latter alternative never seems to occur, and we obtain a tree of factorisations which prove that there is no odd perfect number less than B.

The difficulty in applying the procedure outlined is in factorising $\sigma(p^\alpha)$ for large components p^α. Using various tricks [7, 8], it is possible to restrict attention to components $p^\alpha < B^{2/5}$ (approximately). In this way it has recently been shown that

Theorem 2 [8]. *There is no odd perfect number less than* 10^{300}.

The proof required many factorisations, which were accomplished using the ECM and MPQS algorithms. No doubt the upper bound 10^{300} could be increased by using the NFS algorithm.

6. Conclusion

We have sketched some algorithms for primality testing and integer factorisation. The algorithms draw on results in elementary number theory, algebraic number theory and probability theory. As well as their inherent interest and applicability to other areas of mathematics, advances in computing technology have given them practical applications in the area of secure communications.

Despite much progress in the development of efficient algorithms, our knowledge of the complexity of factorisation (and hence the security of the RSA cryptosystem) is inadequate. We need to find a polynomial time factorisation algorithm or else prove that one does not exist! If one does exist, then the RSA cryptosystem is insecure and we need to find a provably secure replacement for it.

Acknowledgement

Some of the numerical results quoted above were obtained with the assistance of the Australian National University Supercomputer Facility and the Dutch Working Group on the Use of Supercomputers.

References

1. L. M. Adleman and M. A. Huang, "Recognizing primes in random polynomial time", *Proc. Nineteenth Annual ACM Symposium on the Theory of Computing*, ACM, New York, 1987, 462-469.
2. R. P. Brent, "An improved Monte Carlo factorization algorithm", *BIT* 20 (1980), 176-184.
3. R. P. Brent, "Succinct proofs of primality for the factors of some Fermat numbers", *Mathematics of Computation* 38 (1982), 253-255.
4. R. P. Brent, "Some integer factorization algorithms using elliptic curves", *Australian Computer Science Communications* 8 (1986), 149-163.
5. R. P. Brent, "Factorization of the eleventh Fermat number (preliminary report)", *AMS Abstracts* 10 (1989), 89T-11-73.
6. R. P. Brent, "Parallel algorithms for integer factorisation", in *Number Theory and Cryptography* (edited by J. H. Loxton), Cambridge University Press, 1990.
7. R. P. Brent and G. L. Cohen, "A new lower bound for odd perfect numbers", *Mathematics of Computation* 53 (1989), 431-437 and S7-S24.
8. R. P. Brent, G. L. Cohen and H. J. J. te Riele, *Improved techniques for lower bounds for odd perfect numbers*, Report CMA-R50-89, Centre for Mathematical Analysis, Australian National University, October 1989, 198 *pp.*
9. R. P. Brent and J. M. Pollard, "Factorization of the eighth Fermat number", *Mathematics of Computation* 36 (1981), 627-630.
10. 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.
11. D. A. Buell, "Factoring: algorithms, computations, and computers", *J. Supercomputing* 1 (1987), 191-216.
12. T. R. Caron and R. D. Silverman, "Parallel implementation of the quadratic sieve", *J. Supercomputing* 1 (1988), 273-290.
13. H. Cohen and H. W. Lenstra, Jr., "Primality testing and Jacobi sums", *Mathematics of Computation* 42 (1984), 297-330.
14. J. D. Dixon, "Factorization and primality tests", *Amer. Math. Monthly* 91 (1984), 333-352.
15. S. Goldwasser and J. Kilian, "Almost all primes can be certified quickly", *Proc. Eighteenth Annual ACM Symposium on the Theory of Computing*, ACM, New York, 1986, 316-329.
16. R. K. Guy, "How to factor a number", *Congressus Numerantum XVI*, Proc. Fifth Manitoba Conference on Numerical Mathematics, Winnipeg, 1976, 49-89.
17. K. F. Ireland and M. Rosen, *A Classical Introduction to Modern Number Theory*, Springer-Verlag, 1982, Ch. 18.
18. J-R. Joly, "Equations et variétés algébriques sur un corps fini", *L'Enseignement Mathématique* 19 (1973), 1-117.

19. D. E. Knuth, *The Art of Computer Programming*, Vol. 2, Addison Wesley, 2nd edition, 1982.
20. S. Lang, *Elliptic Curves – Diophantine Analysis*, Springer-Verlag, 1978.
21. R. S. Lehman, "Factoring large integers", *Mathematics of Computation* 28 (1974), 637-646.
22. A. K. Lenstra, H. W. Lenstra, Jr., M. S. Manasse and J. M. Pollard, *The number field sieve*, preprint, December 1989.
23. A. K. Lenstra and M. S. Manasse, *Factoring by electronic mail*, preprint, 10 June 1989.
24. H. W. Lenstra, Jr., "Factoring integers with elliptic curves", *Ann. of Math.* (2) 126 (1987), 649-673.
25. P. L. Montgomery, "Speeding the Pollard and elliptic curve methods of factorization", *Mathematics of Computation* 48 (1987), 243-264.
26. M. A. Morrison and J. Brillhart, "A method of factorization and the factorization of F_7", *Mathematics of Computation* 29 (1975), 183-205.
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", *Belgian J. Comp. Appl. Math.* 27(1989), 267-278.
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. J. Seberry and J. Pieprzyk, *Cryptography: An Introduction to Computer Security*, Prentice Hall, Sydney, 1989.
36. D. Shanks, "Class number, a theory of factorization, and genera", *Proc. Symp. Pure Math.* 20, American Math. Soc., 1971, 415-440.
37. R. D. Silverman, "The multiple polynomial quadratic sieve", *Mathematics of Computation* 48 (1987), 329-339.
38. I. N. Stewart and D. O. Tall, *Algebraic Number Theory*, second edition, Chapman and Hall, 1987.
39. M. Voorhoeve, "Factorization", in *Studieweek Getaltheorie en Computers* (edited by J. van de Lune), Math. Centrum, Amsterdam, 1980, 61-68.
40. D. Wiedemann, "Solving sparse linear equations over finite fields", *IEEE Trans. Inform. Theory* 32 (1986), 54-62.