FACTORING INTEGERS USING THE WEB AND THE NUMBER FIELD SIEVE

ARJEN K. LENSTRA

1. INTRODUCTION

This note provides background on the www-factoring project, which was started in the fall of 1995. Factoring a positive integer \( n \) means finding two positive integers \( u \) and \( v \) such that the product of \( u \) and \( v \) equals \( n \), and such that both \( u \) and \( v \) are greater than 1. Such \( u \) and \( v \) are called factors (or divisors) of \( n \), and \( n = u \cdot v \) is called a factorization of \( n \). Positive integers that can be factored are called composites. Positive integers greater than 1 that cannot be factored are called primes. For example, \( n = 15 \) can be factored as the product of the primes \( u = 3 \) and \( v = 5 \), and \( n = 105 \) can be factored as the product of the prime \( u = 7 \) and the composite \( v = 15 \). There are efficient methods to distinguish primes from composites that do not require factoring the composites (cf. [9], [20], and Appendix). These methods can be used to establish beyond doubt that a certain number is composite without, however, giving any information about its factors.

Factoring a composite integer is believed to be a hard problem. This is, of course, not the case for all composites—composites with small factors are easy to factor—but, in general, the problem seems to be difficult. Currently the limits of our factoring capabilities lie around 130 decimal digits. Factoring hard integers in that range requires enormous amounts of computing power. One way to get the computing power needed is to distribute the computation over the Internet. The www-factoring effort is intended to be a convenient way to divide the factoring work among volunteers on the Internet.

Factoring on the Internet is not new. The approach described in [12] was first used in 1988 to factor a 100-digit integer; since then, to factor many integers in the 100 to 120 digit range; and most recently (1993-1994), to factor the famous 129-digit RSA-challenge number (cf. [1]).

This note is intended for contributors to the www-factoring effort who want to understand how modern factoring algorithms work. Using simple examples...
we illustrate the basic steps involved in the factoring methods used to obtain
the factorizations just mentioned and explain how these methods can be run
in parallel on a loosely coupled computer network, such as the Internet.

First we describe the two main types of factoring methods: those that work
quickly if one is lucky, and those that are almost guaranteed to work no matter
how unlucky one is. We then examine more closely the latter methods. In
Section 2 we sketch the basic approach of these ‘guaranteed performance’ fac-
toring methods. We show that they consist of two main steps: ‘data collection’,
and ‘data processing’. Section 3 concentrates on the quadratic sieve factoring
algorithm: how the data collection step works, how it can be parallelized over a
network, and how its efficiency can be improved using a simple additional trick.
Section 3 concludes with some data from quadratic sieve factoring efforts.

The algorithm that we will be using in the www-factoring work—the number
field sieve—is sketched, more or less, in Section 4. This sketch offers only a
vague indication of what the algorithm does. It should help contributors to
the www-factoring project understand what their machines are doing, but it
omits most of the mathematics that would be required to understand why the
algorithm makes sense.

In the Appendix we give the expected run times of various factoring algo-
rithms; we show how composites can quickly be identified; and we describe the
relation between factoring and cryptography. The latter is one of the main rea-
sons why people are interested in evaluating the practical difficulty of factoring
integers.

Understanding the material presented in this note requires some willingness
to bear with a few easy examples and a few slightly more complicated formulas.
Some of the descriptions below are oversimplified to the point of being partially
inaccurate—in particular the description of the number field sieve factoring
algorithm is seriously deficient. Nevertheless, we hope that the www-factoring-
helpers find this note useful, and that it inspires them to consult the literature
referred to in the references.

General-purpose and special-purpose factoring algorithms. Integer
factoring algorithms are usually categorized as either general-purpose algo-
rithms (with an expected run time that depends solely on the size of the num-
ber \( n \) being factored) or special-purpose algorithms (with an expected run time
that also depends on properties of the—unknown—factors of \( n \)). When eval-
uating the security of factoring-based cryptosystems (see Appendix), people
employ general-purpose factoring algorithms. The present note focuses on this
category of integer factoring algorithms.

Examples of special-purpose algorithms whose run time depends on the size
of the smallest factor of \( n \) are ‘trial division’, ‘Pollard’s rho-method’, and the
elliptic curve method'. Refer to [9] and [20] for a discussion of these and other special-purpose methods that depend on other properties of the factors of $n$; they are not discussed here.

2. The Morrison-Brillhart Approach

Congruence of squares. The factorizations mentioned above were obtained using the quadratic sieve factoring algorithm, which is Carl Pomerance's variation (1981) of Richard Schroeppel's linear sieve algorithm (1977). These are both general-purpose factoring algorithms, and both are based on the classical 'congruence of squares method'. As an example of this approach, consider the number $n = 143$. By writing $143$ as $144 - 1 = 12^2 - 1^2$ we can easily derive the factorization of $143$, since $12^2 - 1^2 = (12 - 1)(12 + 1) = 11 \cdot 13$, and because $11$ and $13$ are prime. Apparently, writing $n$ as the difference $x^2 - y^2$ of two squares is helpful in factoring $n$, as long as we are not so unlucky that $x - y$ equals $1$.

More generally, for factoring it is useful to find integers $x$ and $y$ such that $x^2 - y^2$ is a multiple of $n$ (we write $x^2 - y^2 \equiv 0 \mod n$, or $x^2 \equiv y^2 \mod n$, and we say that $x^2$ and $y^2$ are congruent modulo $n$). Namely, if $n$ divides $x^2 - y^2$, it also divides $(x - y)(x + y) = x^2 - y^2$. Therefore, the factors of $n$ must be factors of $x - y$, or they must be factors of $x + y$, or some of them must be factors of $x - y$ and some must be factors of $x + y$. In the first case, $n$ would be a factor of $x - y$, which can be checked easily. In the second case, $n$ would be a factor of $x + y$, which can also be checked easily. If neither of those cases hold, then the factors of $n$ must be split, in some way, among $x - y$ and $x + y$. This would give us a way to find factors of $n$ if we had a method to find out which factors $n$ and $x - y$ have in common, and which factors $n$ and $x + y$ have in common. Such a method has been known for more than 2000 years. It is called 'Euclid's algorithm', and it computes the greatest common divisor of two integers very rapidly (cf. [7]). Thus, to find the common factors of $n$ and $x - y$ we compute

$$\text{gcd}(n, x - y),$$

the greatest common divisor of $n$ and $x - y$. Similarly, we can compute $\text{gcd}(n, x + y)$.

Summarizing this argument, if $x^2 \equiv y^2 \mod n$, then $n$ divides $(x - y)(x + y)$, and therefore

$$n \text{ divides } \text{gcd}(x - y, n) \cdot \text{gcd}(x + y, n)$$

Since gcd's can be computed rapidly, one can quickly check whether the latter identity leads to a factorization of $n$: if $n$ is composite, not a prime power, and $x$ and $y$ are random integers satisfying $x^2 \equiv y^2 \mod n$, then there is at least a
50% chance that \( \gcd(x - y, n) \) and \( \gcd(x + y, n) \) are indeed non-trivial factors of \( n \).

**Finding congruences of squares.** For practical purposes the above explanation implies that in order to factor \( n \), one need only generate a few random looking pairs \( x, y \) such that \( x^2 \equiv y^2 \mod n \). Note that simply picking some random positive \( v \), computing \( s_v \) as the least positive remainder modulo \( n \) of \( v^2 \), and hoping that \( s_v \) is the square of some integer \( y \) (in which case \( x \) is set equal to \( v \)), is unlikely to work (unless \( v < \sqrt{n} \), but in that case \( x = y \) and \( \gcd(x - y, n) = n \)): there are only \( \sqrt{n} \) squares less than \( n \), so the chance of hitting one of them is only \( 1/\sqrt{n} \), which implies that this ‘factoring algorithm’ cannot be expected to be faster than trial division.

The Morrison-Brillhart approach does something that is similar, but instead of waiting for a single very lucky and unlikely ‘big hit’, it combines the results of several much more likely ‘small hits’: instead of randomly picking \( v \)'s until one is found for which the corresponding \( s_v \equiv v^2 \mod n \) is a perfect square, we collect \( v \)'s for which \( s_v \) satisfies a certain much weaker condition. We then combine the resulting \( v, s_v \) pairs once we have a sufficient number of them. Thus, the factoring process (i.e., the method to obtain solutions to the congruence \( x^2 \equiv y^2 \mod n \)) is split into two main steps: the ‘data collection step’ where \( v, s_v \) pairs satisfying some particular condition are collected, and the ‘data processing step’ where the pairs are combined to find solutions to the congruence. The ‘much weaker condition’ on \( s_v \) can informally be described as ‘it should be easy to fully factor \( s_v \)’. How the pairs \( v, s_v \) can be combined can be seen in the example below.

**An example using random squares.** Even though we already know that \( n = 143 = 11 \cdot 13 \), here is how the Morrison-Brillhart approach works for \( n = 143 \). Since factors 2, 3, and 5 can easily be recognized, the ‘much weaker condition’ for the example \( n = 143 \) will be ‘it should be possible to factor \( s_v \) completely using only 2, 3, and 5’. In general, for larger numbers than 143, more primes will be allowed in the factorization of \( s_v \); this set of primes is usually referred to as the factor base. In the example, the factor base is the set \( \{2, 3, 5\} \). We say that a number is smooth if it can be factored using the elements of the factor base; the definition of smoothness thus depends on the factor base that is used. In the example, a number is smooth if it can be factored using the primes 2, 3, and 5.

To find pairs \( x, y \) such that \( x^2 \equiv y^2 \mod 143 \) using ‘Dixon’s algorithm’ (the most straightforward example of the Morrison-Brillhart approach) we simply randomly pick \( v \)'s and keep those for which \( s_v \) is smooth (i.e., can be factored using 2, 3, and 5), and we wait until we have a few different pairs \( v, s_v \) for which \( s_v \) is smooth.
Let \( v = 17 \) be the first random choice. We find that \( v^2 = 289 = 3 + 2 \cdot 143 \equiv 3 \mod 143 \), so that \( s_{17} = 3 \). Obviously, \( s_{17} = 3 \) is smooth, which implies that we keep \( v = 17 \), for which we have the following relation:

\[
17^2 \equiv 2^0 \cdot 3^1 \cdot 5^0 \mod 143.
\]

Since \((v + 1)^2 = v^2 + 2v + 1\), a convenient next choice is \( v = 18 \): \( 18^2 = 17^2 + 2 \cdot 17 + 1 \equiv 3 + 35 = 38 = 2 \cdot 19 \mod 143 \), and \( s_{18} = 2 \cdot 19 \) is not smooth, so that \( v = 18 \) can be thrown away. Proceeding to 19 we find that \( 19^2 = 18^2 + 2 \cdot 18 + 1 \equiv 38 + 37 = 75 \mod 143 \), and \( s_{19} = 75 \) is smooth, so that we keep \( v = 19 \) and have found our second relation:

\[
19^2 \equiv 2^0 \cdot 3^1 \cdot 5^2 \mod 143.
\]

The next attempt \( 20^2 = 19^2 + 2 \cdot 19 + 1 \equiv 75 + 39 = 114 = 2 \cdot 3 \cdot 19 \mod 143 \) fails again, after which we find the relation

\[
21^2 = 20^2 + 2 \cdot 20 + 1 \equiv 114 + 41 = 155 = 12 + 143 \equiv 12 = 2^2 \cdot 3^1 \cdot 5^0 \mod 143.
\]

Looking at the three relations obtained so far, we observe that the product of the first two, the product of the last two, and the product of the first and the last all lead to a congruence of squares:

\[
(17 \cdot 19)^2 \equiv 2^0 \cdot 3^2 \cdot 5^2 \mod 143,
\]

\[
(19 \cdot 21)^2 \equiv 2^2 \cdot 3^2 \cdot 5^2 \mod 143,
\]

and

\[
(17 \cdot 21)^2 \equiv 2^2 \cdot 3^2 \cdot 5^0 \mod 143.
\]

The first of these leads to \( x = 17 \cdot 19, y = 3 \cdot 5 \) and the factors \( \gcd(323 - 15, 143) = 11 \) and \( \gcd(323 + 15, 143) = 13 \). The second leads to \( x = 19 \cdot 21, y = 2 \cdot 3 \cdot 5 \) and the trivial factors \( \gcd(399 - 30, 143) = 1 \), \( \gcd(399 + 30, 143) = 143 \). The third one gives \( x = 17 \cdot 21, y = 2 \cdot 3 \) and the factors \( \gcd(357 - 6, 143) = 13 \) and \( \gcd(357 + 6, 143) = 11 \).

The relation after the one for \( v = 21 \) would be \( 23^2 \equiv 2^2 \cdot 3^0 \cdot 5^2 \mod 143 \) which is already of the form \( x^2 \equiv y^2 \mod n \). This congruence leads to \( x = 23, y = 10 \) and the non-trivial factors \( \gcd(23 - 10, 143) = 13 \) and \( \gcd(23 + 10, 143) = 11 \). For more challenging numbers than 143 we cannot expect to be so lucky—indeed, after factoring hundreds of numbers in the 70 to 129 digit range, this never happened.

**Finding the right combinations of relations.** To pick the right relations so that their product yields a solution to the congruence \( x^2 \equiv y^2 \mod n \) we
use ‘linear algebra’. In the above example, consider the triples consisting of
the three exponents of the three primes in the factor base on the right hand
sides of the ‘≡’ sign in the relations: for \( v = 17 \) the triple \((0, 1, 0)\), for \( v = 19 \)
the triple \((0, 1, 2)\), and for \( v = 21 \) the triple \((2, 1, 0)\). Each of these triples is a 3-
dimensional vector, i.e., a row with three entries. Two vectors can be added to
yield another vector of the same length, simply by adding the corresponding
entries of the two vectors. For instance, \((0, 1, 0) + (0, 1, 2) = (0, 2, 2)\). Note
that addition of these two exponent vectors corresponds to multiplying the
right hand sides of the relations for \( v = 17 \) and \( v = 19 \). For our application,
combinations of vectors that add up to vectors with all even entries correspond
to the combinations of relations that we are looking for.

Finding all even combinations of vectors is a common problem in linear
algebra, for which several good algorithms exist: (structured) Gaussian elim-
nation, (blocked) Lanczos, and (blocked) Wiedemann are currently the most
popular choices for our applications (see [3], [8], [16], and [19] and the references
therein). In general, if there are \( m \) relations and \( k \) primes in the factor ba.
se, we have an \( m \times k \)-matrix (i.e., a matrix consisting of \( m \) rows and \( k \) columns, where
the \( m \) rows correspond to the \( m \) different \( k \)-dimensional vectors consisting of
the \( k \)-tuples of exponents in the \( m \) relations). For the example given here, we
would get the matrix

\[
\begin{pmatrix}
0 & 1 & 0 \\
0 & 1 & 2 \\
2 & 1 & 0
\end{pmatrix}.
\]

If the matrix is over-square, i.e., if \( m > k \), it can be shown that there are
at least \( m - k \) all even combinations of the rows (i.e., of the \( k \)-dimensional
vectors) each of which leads to an independent chance to factor \( n \). It follows
that sufficiently many relations will in practice always lead to a factorization;
it also shows that we have been rather lucky in our example by finding so many
all even combinations in a \( 3 \times 3 \)-matrix.

3. Quadratic sieve

Finding relations faster, sieving. Intuitively, the smaller a number is,
the higher its probability that it is smooth. For example, there are 39 positive
smooth numbers \(< 143\), but there are 28 positive smooth numbers \(< 72\). There
fore, if we randomly pick positive numbers \(< B\), we would get a smoothness
probability of \( 39/142 = 0.27 \) for \( B = 143 \) but a higher probability \( 28/71 = 0.39 \)
for \( B = 72 \). For \( B = 1000 \) we get \( 86/999 = 0.09 \), and for \( B = 10^6 \) we get only
\( 507/999999 = 0.0005 \). Thus, the smaller \( |s_v| \) can be made, the higher proba-
bility we should get that it is smooth. Therefore, it would be to our advantage
to find ways of selecting \( v \) such that \( |s_v| \) can be guaranteed to be substantially smaller than \( n \).

For randomly selected \( v \), the number \( s_v \) (the least positive remainder of \( v^2 \) modulo \( n \)) can be expected to have roughly the same size as \( n \). At best we can guarantee that \( |s_v| \) is one bit smaller than \( n \) if we redefine \( s_v \) as the least absolute remainder of \( v^2 \) modulo \( n \), and we include \(-1\) in the factor base.

A better way to find small \( s_v \)'s is by taking \( v \) close to \( \sqrt{n} \). Let \( v(i) = i + [\sqrt{n}] \) for some small integer \( i \) (where \([\sqrt{n}]\) denotes the largest integer that is at most \( \sqrt{n} \)). It follows that \( s_{v(i)} = (i + [\sqrt{n}])^2 - n \) and that \( |s_{v(i)}| \) is of the same order of magnitude as \( 2i\sqrt{n} \), because \( ||[\sqrt{n}]^2 - n| \) is at most \( 2\sqrt{n} \). This implies that \( |s_{v(i)}| \) for small \( i \) has a much higher chance to be smooth than \( s_v \) for a randomly selected \( v \). Note, however, that the smoothness probability decreases if \( i \) gets larger.

Quadratic sieve (QS) combines this better way of choosing \( v = v(i) \) with the following important observation: if some \( p \) divides \( s_{v(i)} \), then \( p \) divides \( s_{v(i+tp)} \) for any integer \( t \). This makes it possible to use a sieve to quickly identify many possibly smooth \( s_{v(i)} \) with \( i \) in some predetermined interval. The sieve is used to record 'hits' by the primes in the factor base in an efficient manner: if a prime \( p \) divides a certain \( s_{v(i)} \), then this is recorded at the \( i \)th location of the sieve, and it is also recorded at all locations \( i + tp \) in the sieve. Thus, for each \( p \), we can quickly step through the sieve, with step-size \( p \), once we know where we have to make the first step. To make the process of 'recording \( p \)' efficient, we simply add \( \log p \) to the relevant locations.

Assuming that all sieve locations are initially zero, the \( i \)th location contains (after the sieving) the sum of the logarithms of those primes that divide \( s_{v(i)} \). Therefore, if the \( i \)th location is close to \( \log |s_{v(i)}| \), we check whether \( |s_{v(i)}| \) is indeed smooth, simply by trial dividing \( |s_{v(i)}| \) with all primes in the factor base. This entire process is called sieving—it is much faster than checking the smoothness of each individual \( |s_{v(i)}| \) by trial dividing with all primes in the factor base.

In the 'multiple polynomial' variation of QS the single polynomial \((X + [\sqrt{n}])^2 - n\) is replaced by a sequence of polynomials that have more or less the same properties as \((X + [\sqrt{n}])^2 - n\), all for the same number \( n \) to be factored. The advantage of multiple polynomials is that for each polynomial the same small \( i \)'s can be used, thereby avoiding the less profitable larger \( i \)'s. A second important advantage is that different processors can work independently of each other on different polynomials.

Another way of increasing the smoothness probability is by extending the factor base (thus relaxing the definition of smoothness). However, this also
implies that more relations have to be found to make the matrix over-square, and that the linear algebra becomes more involved. The optimal factor base size follows from a careful analysis of all these issues. Refer to [18] for examples of such analyses, and to [15] for another informal description of QS.

Distributed factoring using QS. We have seen that QS consists of two major steps: the sieving step, to collect the relations, and the matrix step, where the relations are combined and the factorization is derived. For numbers in our current range of interest, the sieving step is by far the most time consuming. It is also the step that allows easy parallelization, with hardly any need for the processors to communicate. All a processor needs to stay busy for at least a few weeks is the number to be factored, the size of the factor base, and a unique collection of polynomials to sieve with in order to find relations—the latter can be achieved quite easily by assigning a unique integer to a processor. Given those data, any number of processors can work independently and simultaneously on the sieving step for the factorization of the same number. The resulting relations can be communicated to a central location using electronic mail, say once per day, or each time some pre-set number of relations has been found.

This parallelization approach is completely fault-tolerant. In the first place, the correctness of all relations received at the central location can easily be verified by checking the congruence. Furthermore, no particular relation is important, only the total number of distinct relations received counts. Finally, there is a virtually infinite pool of ‘good’ almost limitless intervals in which to look for polynomials. Thus, no matter how many processors crash or do not use the interval assigned to them for other reasons, and no matter how mailers or malicious contributors mangle the relations, as long as some processors contribute some relations that check out, progress will be made in the sieving step. Since there is no way to guarantee that relations are sent only once, all data have to be kept sorted at the receiving site to be able to weed out the duplicates. Currently there is also no way to prevent contributors from flooding the mailbox at the central collecting site, but so far this has not been a problem in distributed factoring.

All these properties make the sieving step for QS ideal for distribution over a loosely coupled and rather informal network, such as the Internet, without any need to trust anyone involved in the computation. Refer to [12] and [1] for information on how such factoring efforts have been organized in the past.

The matrix step is done at a central location, as soon as the sieving step is complete (i.e., as soon as a sufficient number of relations have been received to make the matrix over-square). For details, refer to [12].

Large primes, partial relations, and cycles. In practice, sieving is not
a very precise process: one often does not sieve with the small primes in the
factor base, or with powers of elements of the factor base; \( \log p \) is rounded to
the nearest integer value; and the base of the logarithm is chosen so that the
values that are accumulated in the \( s(i) \)'s can be represented by single bytes.
The process can tolerate these imperfections because there are plenty of good
polynomials to use for sieving. It is not a problem, therefore, if occasionally a
good location is overlooked as long as the sieve identifies a sufficient number of
possibly smooth numbers as quickly as possible. How many relations we find
per unit of time is more important than how many we might have missed.

As a consequence of the approximations that are made during the sieving,
the condition that \( s(i) \) should be close to \( \log |s_{v(i)}| \) should be interpreted quite
liberally. This, in turn, leads to many \( v(i) \)'s for which \( s_{v(i)} \) is ‘almost’ smooth
(i.e., smooth with the exception of one reasonably small factor that is not in
the factor base). Such ‘almost smooth’ relations are often referred to as ‘partial
relations’ if the non-smooth factor is prime, and ‘double partial relations’ if the
non-smooth factor is the product of two primes. The non-smooth primes are
referred to as the ‘large primes’. The relations for which \( s_{v(i)} \) can be factored
completely over the factor base may be distinguished by calling them ‘full
relations’.

Partial relations will be found at no extra cost during the sieving step, and
double partial relations at little extra cost. But keeping them, and investing
that little extra effort to find them, only makes sense if they can be used in the
factoring process. As an example why partial relations can be useful, consider
the example \( n = 143 \) again. The choice \( v = 18 \) was rejected because \( s_{18} = 2 \cdot 19 \)
is not smooth (with respect to the factor base \( \{2, 3, 5\} \)). After trial dividing
\( s_{18} \) with 2, 3, and 5, it follows immediately that 19 is prime (from the fact that
\( 19 < 5^2 \)), so that \( v = 18 \) leads to a partial relation with large prime 19:

\[
18^2 \equiv 2^1 \cdot 3^0 \cdot 5^0 \cdot 19 \mod 143.
\]

Another choice that was rejected was \( v = 20 \), because \( s_{20} = 2 \cdot 3 \cdot 19 \), which
leads, for the same reason as above, to a partial relation—again with large
prime 19:

\[
20^2 \equiv 2^1 \cdot 3^1 \cdot 5^0 \cdot 19 \mod 143.
\]

These two partial relations have the same large prime, so we can combine them
by multiplying them together, and get the following:

\[
(18 \cdot 20)^2 \equiv 2^2 \cdot 3^1 \cdot 5^0 \cdot 19^2 \mod 143.
\]

Except for the ‘\( 19^2 \)’ on the right hand side, this looks like a full relation.
Because the greatest common divisor of 19 and 143 equals 1 (something that
can easily be checked), we can use a classical method called the ‘extended Euclidean algorithm’ to find an integer \( x \) such that \( x \cdot 19 \equiv 1 \mod 143 \), which can then be used to get rid of the ‘\( 19^2 \)’ (cf. [7]). We find \( x = 128 \):

\[
128 \cdot 19 = 2432 = 1 + 17 \cdot 143 \equiv 1 \mod 143.
\]

If we multiply both sides of the above ‘almost smooth’ relation by \( 128^2 \), we get

\[
(128 \cdot 18 \cdot 20)^2 \equiv 2^2 \cdot 3^1 \cdot 5^0 \cdot (128 \cdot 19)^2 \mod 143.
\]

Because \( 128 \cdot 18 \cdot 20 = 46080 = 34 + 322 \cdot 143 \equiv 34 \mod 143 \), and because \( 128 \cdot 19 \equiv 1 \mod 143 \), we find

\[
34^2 \equiv 2^2 \cdot 3^1 \cdot 5^0 \mod 143,
\]

which is, for factoring purposes, equivalent to a full relation.

Double partials can be used in a slightly more complicated but similar way. Combinations of partial and/or double partial relations in which the large primes disappear (and that are therefore as useful as full relations) are often referred to as ‘cycles’. Note that the cycle that we have found in the example does not provide any useful new information, because it happens to be the relation for \( v = 17 \) multiplied by \( 2^2 \).

How much luck is needed to find two partials with the same large primes, or to find a double partial for which both large primes can be combined with large primes found in other partials or double partials? The answer to this question is related to the ‘Birthday Paradox’. A group of at least 23 (randomly selected) people contains two persons with the same birthday in more than 50% of the cases. More generally: if numbers are picked at random from a set containing \( r \) numbers, the probability of picking the same number twice exceeds 50% after \( 1.177\sqrt{r} \) numbers have been picked. In QS, the set consists of prime numbers larger than any in the factor base, but smaller than a limit which is typically \( 2^{30} \) or so. There are only a few tens of millions of primes in this range, so we expect to be able to find matches between the large primes once we have more than a few thousand partial and double partial relations. This is the reason why the simple trick of using partial relations is so effective.

As shown in [12] and [13], cycles are indeed found in practice, and they speed up the factoring process considerably. Using partial relations makes the sieving step approximately 2.5 times faster, and using double partial relations as well saves another factor 2 to 2.5. There is a price to be paid for this acceleration: more data have to be sent to the central site; more disk space is needed to store the data; and the matrix problem gets a bit harder (either due to higher
density of the rows of the matrix, or to larger matrices). The time saved in the sieving step, however, certainly justifies incurring these inconveniences. For a discussion of these issues see [1] and [4].

**Some illustrative QS data.** To give an impression of factor base sizes, the amount of data collected, the influence of large primes, and practical run times of the sieving and matrix steps, some data for the QS-factorizations of a 116-digit, a 120-digit, and a 129-digit number (from [13], [4], and [1], respectively) are presented in Table 1. The sieving step for the 116-digit factorization was done entirely on the Internet using the software from [12]. For the 120-digit number it was carried out on 5 different Local Area Networks (using 3 different implementations of the sieving step), and on the 16384 processor MasPar MP-1 massively parallel computer at Bellcore. Sieving for the 129-digit was mostly done on the Internet using an updated version of the software from [12], with several sites using their own independently written sieving software; about 14% of the sieving was done on several MasPars. The matrix step for all numbers was done on Bellcore's MasPar.

The amount of data is shown in gigabytes of disk space needed to store the data in uncompressed format. The timing for the sieving step is given in units of MY, or 'mips-years'. By definition 1 MY is one year on a VAX 11/780, a relatively ancient machine that can hardly be compared to current workstations. The timings were derived by assigning a reasonable 'mips-rating' to the average workstation that was used; see [4] and [1] for details. Although this measure is not very accurate, it gives a reasonable indication of the growth rate of the sieving time for QS, as long as workstations are rated in a consistent manner.

The numbers of fulls, partials, double partials, and cycles are given in the Table as they were at the end of the sieving step. Note that in all cases the number of fulls plus the number of cycles is larger than the size of the factor base, with a considerable difference for the two Internet factorizations. This 'overshoot' is often large because the number of cycles grows very rapidly toward the end of the sieving step; since the 'cease and desist' message is only sent out to the Internet-workers when the sum is large enough, and since it takes a while before all client-processes are terminated, the final relations received at the central site cause a large overshoot.

The timing for the matrix step is given in hours on the MasPar. By using a better algorithm, the matrix timings can now be improved considerably: the matrix for the 129-digit number can be processed in less than 10 hours on the MasPar, or in about 9 days on a Sparc 10 workstation (cf. [16] and [3]). More about the expected run time of QS is given in the Appendix.
Distributed factoring using NFS. A distributed approach similar to that sketched above was used in 1990 to factor the ninth Fermat number $2^{512} + 1$ (cf. [11]) by means of another factoring algorithm, the number field sieve (NFS). The original version of NFS could only be used to factor numbers of a very special form, such as $2^{512} + 1$. The version of NFS that we will be using in the www-factoring project can handle arbitrary numbers.

NFS consists of the same two major steps as QS. Although the sieving step of NFS is entirely different from that of QS, it can be distributed over a network in almost the same way—except for the way the inputs are handled. In the sieving step of QS it takes the average workstation a considerable amount of time, say a few weeks, to process a single input. Furthermore, for each number to be factored, there are millions of good inputs that are all more or less equally productive, and that lead to distinct relations.

In NFS, each input can be completely processed in a matter of minutes, which means that a new batch of inputs has to be communicated to a contributing processor as soon as its current batch has been processed. Furthermore, there are far fewer inputs available than in QS. This implies that when the results from an input are not received within a reasonable time, the input has to be redistributed, which may possibly lead to even more duplicated results than we already had to deal with in QS.

Relations in NFS. The number field sieve is substantially more complicated than the methods sketched so far. Here we explain, vaguely, the part of the work the client-processors are involved in (i.e., what the relations in NFS look like, and how they can be found). How the relations are combined at the central site to derive the factorization is beyond the scope of this note; it can be found in [10].

<table>
<thead>
<tr>
<th></th>
<th>116-digit</th>
<th>120-digit</th>
<th>129-digit</th>
</tr>
</thead>
<tbody>
<tr>
<td>size factor base</td>
<td>120000</td>
<td>245810</td>
<td>524339</td>
</tr>
<tr>
<td>large prime bound</td>
<td>$10^8$</td>
<td>$2^{30}$</td>
<td>$2^{30}$</td>
</tr>
<tr>
<td>fulls</td>
<td>25361</td>
<td>48665</td>
<td>112011</td>
</tr>
<tr>
<td>partials</td>
<td>284750</td>
<td>884323</td>
<td>143137</td>
</tr>
<tr>
<td>double partials</td>
<td>953242</td>
<td>4172512</td>
<td>6881138</td>
</tr>
<tr>
<td>cycles</td>
<td>117420</td>
<td>203557</td>
<td>457455</td>
</tr>
<tr>
<td>amount of data</td>
<td>0.25 GB</td>
<td>1.1 GB</td>
<td>2 GB</td>
</tr>
<tr>
<td>timing sieving step</td>
<td>400 MY</td>
<td>825 MY</td>
<td>5000 MY</td>
</tr>
<tr>
<td>timing matrix step</td>
<td>0.5 hrs</td>
<td>4 hrs</td>
<td>45 hrs</td>
</tr>
</tbody>
</table>

4. Number field sieve
Suppose we have two polynomials \( f_1 \) and \( f_2 \) with integer coefficients. There is no need to restrict ourselves to only two polynomials, but that is the most straightforward case. The polynomials \( f_1 \) and \( f_2 \) must both be irreducible, and they must have a common root modulo \( n \) (i.e., an integer \( m \) such that both \( f_1(m) \) and \( f_2(m) \) are divisible by \( n \)). How such polynomials are found in general is not relevant here. The presentation in [10] is mostly restricted to the case where \( m \) is an integer close to \( n^{1/(d+1)} \) for some small integer \( d \) (such as 4 or 5); the polynomials can then be chosen as \( f_1(X) = X - m \) and \( f_2(X) = \sum_{i=0}^{d} c_i X^i \), where \( n = \sum_{i=0}^{d} c_i m^i \) with \(-m/2 \leq c_i \leq m/2\) a base \( m \) representation of \( n \). The NFS-sieving program we will be running in the www-factoring effort will most likely be based on this simple choice.

For the factorization of \( 2^{512} + 1 \) for instance, we chose \( n = 8 \cdot (2^{512} + 1) = 2^{515} + 8 \), and took \( d = 5 \), \( m = 2^{103} \), \( f_1(X) = X - 2^{103} \), and \( f_2(X) = X^5 + 8 \). In this case, \( f_1(2^{103}) = 0 \) and \( f_2(2^{103}) = 2^{515} + 8 = n \), so that both \( f_1(m) \) and \( f_2(m) \) are divisible by \( n \).

For the factorization of RSA-130 (a 130-digit number that is the first number we will try to factor in the www-factoring project), we have the following:

\[
\begin{align*}
n &= 18070 82088 68740 48059 51656 16440 59055 66278 10251 67694 01349 17012 70214 50056 66254 02440 48387 34112 75908 12303 37178 18879 66563 18201 32148 80557, \\
d &= 5, \\
m &= 125 74411 16841 80059 80468, \\
f_1(X) &= X - m, \\
f_2(X) &= 5 48302 48738 05200 X^5 + 9 82261 17482 86102 X^4 \\
&\quad - 13 92499 89128 76685 X^3 + 16 75252 58877 84989 X^2 \\
&\quad + 3 59900 74855 08738 X - 46 69930 53931 05995.
\end{align*}
\]

These polynomials were found by Scott Huddleston.

For \( j = 1, 2 \) and integers \( a, b \), let

\[
N_j(a,b) = f_j(a/b)b^{\text{degree}(f_j)}.
\]

Note that \( N_j(a,b) \) is an integer too. Furthermore, for \( j = 1, 2 \), let there be some factor base consisting of primes up to some bound (depending on \( f_j \)) that may occur in the factorization of \( N_j(a,b) \) for \( a \) and \( b \) that have greatest common divisor equal to 1 (i.e., are ‘coprime’). Smoothness of \( N_j(a,b) \) will always refer to smoothness with respect to the \( j \)th factor base, and \( a \) and \( b \) will always be assumed to be coprime integers with \( b > 0 \). A relation is given by a pair \( a, b \) for which both \( N_1(a,b) \) and \( N_2(a,b) \) are smooth.
The following is an indication why this is considered to be a relation (i.e., something that can be combined with other relations to solve the congruence $x^2 \equiv y^2 \mod n$). If $\alpha_j$ denotes a root of $f_j$, then the prime factorization of $N_j(a, b)$ gives information about something that is called the ‘prime ideal factorization of $a - \alpha_j b$’. Since the $f_j$ have a common root $m$ modulo $n$, the roots $\alpha_1$ and $\alpha_2$ are ‘the same’ when taken mod $n$. This implies that these prime ideal factorizations of $a - \alpha_1 b$ and $a - \alpha_2 b$ give rise to an identity modulo $n$ between two products, where, loosely speaking, all $\alpha_1$’s and $\alpha_2$’s have been replaced by $m$. A sufficient number of congruences of products can be combined into a solution to $x^2 \equiv y^2 \mod n$ using linear algebra—the number we need is roughly the sum of the sizes of the two factor bases.

This picture of how many relations are needed is thoroughly confused by the use of large primes, which can occur both in $N_1(a, b)$ and in $N_2(a, b)$. The experiments with large primes in NFS described in [2] suggest that, unlike QS, the number of cycles that can be built from the partial relations suddenly grows extremely rapidly. If such a cycle explosion occurs, the sieving step is most likely complete, but when this will happen is hard to predict.

**Finding relations in NFS.** Since the smooth values that we are looking for are, as in QS, values of polynomials evaluated at certain points, they can again be found using a sieve: if $p$ divides $N_j(a, b)$ then $p$ also divides $N_j(a+tp, b+wp)$ for any integers $t$ and $w$. The earliest NFS implementations used the following simple sieving strategy: fix $b$; use a sieve to find $a$'s for which both $N_1(a, b)$ and $N_2(a, b)$ might be smooth; and inspect those $N_1(a, b)$ and $N_2(a, b)$ more closely (using trial division). Repeat this for different $b$'s until a sufficient number of relations have been collected. This approach can be distributed over many processors by assigning different ranges of $b$'s to different processors; it was used in [11] and is called ‘line-by-line sieving’. Since smaller $b$'s are better than larger ones the pool of ‘good’ inputs (the $b$'s) eventually dries out, a problem that does not exist in QS.

As shown in [17] the following is more efficient. Fix some reasonably large $q$ that can in principle occur in the factorization of, say, $N_2(a, b)$. Again use a sieve to locate pairs $a, b$ for which $N_1(a, b)$ is smooth and $N_2(a, b)$ factors using only primes $< q$ from the second factor base, but restrict the search to pairs $a, b$ for which $N_2(a, b)$ is divisible by $q$. Repeat this for different $q$'s until a sufficient number of relations have been collected—actually this step should be carried out for all pairs $q, r_q$ where $r_q$ ranges over all roots of $f_2$ modulo $q$, a detail that we will not elaborate upon here. Because of the restriction on the pairs $a, b$, fewer pairs have to be considered per $q$. This makes it possible and necessary to replace the line-by-line sieving by ‘lattice-sieving’. This is a way of quickly identifying, for each $p$, the proper sieve locations in a plane instead...
of on a line. Just as the 1-dimensional line-by-line sieve makes use, for each $p$, of the shortest 1-dimensional vector ($p$), the 2-dimensional lattice sieve makes use, for each $p$, of two 2-dimensional vectors that form a reduced basis for the appropriate lattice of determinant $p$ in the $(a, b)$-plane. Again, the phrase ‘for each $p$’ is oversimplified and should read ‘for each $p$, $r_p$ pair’, where $r_p$ is a root of $f_j$ modulo $p$ (with $p < q$ if $j = 2$).

Lattice-sieving is possible because a substantial part of the $(a, b)$-plane can be made to fit in memory, and it is necessary because this entire process has to be repeated for many $q$'s. The latter implies that we cannot afford the time to look at all $b$-lines for all relevant $p$ for all these $q$'s, i.e., that line-by-line sieving is too slow. The details of lattice sieving are rather messy (though not as bad as some of the rest of NFS) and can be found in [6].

**Description of what a client-processor does.** Lattice sieving can be distributed over many processors by assigning different ranges of $q$'s to different processors. In the www-factoring effort, inputs consist of intervals; processing an interval consists of finding all relevant $q$'s in that interval and doing the lattice sieving for each of them to find the corresponding relations. Ranges of $q$'s are distributed by one or more data distribution sites. The ranges assigned to each contributor depend on the resources available to that contributor: the types of processors, the available computing time on the processors, and the available amount of memory per contributing processor.

Before any $q$ can be sieved, however, the factor bases have to be initialized. This is a fairly compute-intensive job that should preferably be done only once per number to be factored and per client-site. Alternatively, depending on the available bandwidth, the factor bases can be downloaded from one of the data distribution sites. Once they have been computed or downloaded and the files have been moved and/or copied to the right places, the sieving can begin.

Per completed pair $q$, $r_q$, the resulting relations are concatenated to an ASCII file whose name depends on the starting point of the interval being sieved and on the $q$ that is currently being processed. Thus, the name of this file changes when the process moves from one $q$ to the next. File names beginning with a ‘$W$’ indicate that a process is still working on that file, and file names beginning with an ‘$F$’ indicate that the process working on that range has finished. File names beginning with an ‘$X$’ indicate that something unexpected happened—these files should be read and appropriate action should be taken. Unfortunately, due to the limited length of file names in some operating systems, the relationship between the remainder of the file names and the information encoded in them is not very transparent. A utility to interpret the

---

2For a small minority of $q$'s only a few $b$'s have to be considered, in which case line-by-line sieving is the preferred strategy.
The files whose names begin with a 'W' or an 'F' both consist of two types of lines: comment lines containing a '#' character, and the other lines containing the relations. The comment lines contain information that can be used at the central site to log off the q's that have been completed, to keep statistics, etc. The relation lines consist of $j_1$, $j_2$, $a$, $b$, and all primes larger than some fixed predetermined bound in the factorizations of $N_{j_1}(a, b)$ and $N_{j_2}(a, b)$.

All F-files and all W-files whose process got interrupted should be sent, by electronic mail perhaps, to the central receiving location. There the q's are logged off, and the relations are sorted and merged with the data that were received earlier. Once every few days, the data will be processed to keep track of the progress. As mentioned above, the number of cycles tends to grow very quickly at some as yet unpredictable point. For that reason it is hard to estimate when the sieving step will be completed—the 'cease and desist' message for the contributing processors will probably come quite unexpectedly.

Once the sieving step is complete, a non-trivial amount of computing has to be carried out at the central site (or at any other location where enough computing power is available). It may take several days, or even weeks, before the resulting factorization can be announced.

**Why NFS is faster than QS.** As shown in the Appendix, we expect the run time of NFS to grow much more slowly than the run time of QS as the numbers to be factored get larger. This can, informally, be explained as follows. Consider the straightforward choice $f_1(X) = X - m$ and $f_2(X) = \sum_{i=0}^{d} c_i X^i$, with $m$ close to $n^{1/(d+1)}$. The probability that both $N_1(a, b) = a - bm$ and $N_2(a, b) = \sum_{i=0}^{d} c_i a^i b^{d-i}$ are smooth depends on the sizes of $a$, $b$, $m$, and the $c_i$'s.

By their choice, $m$ and the $c_i$'s are all of the order $n^{1/(d+1)}$. The sizes of $a$ and $b$ depend on how many $N_1(a, b)$ and $N_2(a, b)$ have to be considered so that we can expect enough of them to be smooth. But 'enough' and 'smooth' depends on the sizes of the factor bases: as in QS, a larger factor base requires more relations, but at the same time relaxes the definition of smoothness. From an analysis of all relevant smoothness probabilities it follows that if $d$ is of the order $(\log n / \log \log n)^{1/3}$, then it may be expected that the largest $a$'s and $b$'s needed will be such that $a^d$ and $b^d$ are of the same order of magnitude as $m$ and the $c_i$'s, i.e., $n^{1/(d+1)}$. This implies that $N_1(a, b)$ and $N_2(a, b)$ are at worst of order $n^{2/d}$. Now note that $2/d \to 0$ for $n \to \infty$ due to the choice of $d$, so that asymptotically the numbers that have to be smooth in NFS are much smaller than the numbers of order roughly $\sqrt{n}$ that have to be smooth in QS.

**Acknowledgments.** Acknowledgments are due to Bruce Dodson, Stuart
Haber, Paul Leyland, and Sue Lowe for their help with this paper.

REFERENCES


**APPENDIX**

**Run times.** All run times presented here are expected run times to factor $n$ for $n \rightarrow \infty$. This implies that the $o(1)$ in each expression below goes to zero. In practice, however, the $o(1)$’s are not zero. Therefore we cannot encourage the practice of evaluating the run time expression for any of the methods below for a particular $n$ with $o(1) = 0$, and to advertise the resulting number as the ‘number of cycles’ necessary to factor $n$ using that method. The expressions are useful, however, to get an indication of the growth rate of the run time—they can be used (with $o(1) = 0$) for limited range extrapolations to predict the expected run time for $m$ given the run time of $n$, if $|\log m - \log n|$ is not too large.

Let

$$L_n[u, v] = \exp(v(\log n)^u(\log \log n)^{1-u}),$$

where ‘log’ is the natural logarithm. The expected run time of Dixon’s random squares algorithm is $L_n[1/2, 2 + o(1)]$ if ordinary trial division is used to check the smoothness of the $s_y$’s. This expected run time can rigorously be proved, unlike the expected run times presented below, which are all, with one exception, based on unproved heuristic assumptions.

The heuristic expected run time of QS is $L_n[1/2, 1 + o(1)]$. Surprisingly, QS is not the only factoring algorithm with this run time: several other methods were proposed, some radically different from QS, that all have the same heuristic expected run time as QS. Even the elliptic curve method (one of the special purpose methods mentioned above) has the same worst-case heuristic expected run time (where the worst case for the elliptic curve method is the
case where the smallest factor of \( n \) is of order \( \sqrt{n} \). An algorithm for which the \( L_{1/2, 1 + o(1)} \) expected run time can be proved rigorously was published in [14]. As a consequence of this remarkable coincidence there was a growing suspicion that \( L_{1/2, 1 + o(1)} \) would be the best we would ever be able to do for factoring.

The \( L_{1/2, 1 + o(1)} \)-spell was eventually broken by the number field sieve. The number field sieve is based on an idea of John Pollard to rapidly factor numbers of the special form \( x^3 + k \), for small \( k \). This idea first evolved in the 'special number field sieve' which has heuristic expected run time \( L_{1/3, (32/9)^{1/3} + o(1)} \approx L_{1/3, 1.526 + o(1)} \), but which can only be applied to numbers of a special form (similar to the form required by Pollard's original method). The 'special form' restrictions were later removed, and the resulting 'general' number field sieve runs in heuristic expected time \( L_{1/3, (64/9)^{1/3} + o(1)} \approx L_{1/3, 1.923 + o(1)} \). Refer to [10] for details.

To put the progress of QS to NFS in perspective, note that trial division runs in exponential time \( n^{1/2} = L_{n[1, 1/2]} \) in the worst case, and that an (as yet unknown) polynomial time factoring algorithm would run in time \( (\log n)^c = L_{n[0, c]} \), for some constant \( c \). Thus, QS and the other algorithms with expected run time \( L_{n[1/2, v]} \) (with \( v \) constant) are, if we only consider the first argument \( u \) of \( L_{n[u, v]} \), halfway between exponential and polynomial time. In this metric, NFS represents a substantial step in the direction of polynomial time algorithms.

**Testing for compositeness.** A famous theorem of Fermat (his so-called 'little theorem') says that if \( n \) is prime and \( a \) is an integer that is not divisible by \( n \), then

\[
a^{n-1} \equiv 1 \mod n.
\]

For instance, for \( n = 7 \) and \( a = 2 \) we find that

\[
2^6 = 64 = 1 + 9 \cdot 7 \equiv 1 \mod 7.
\]

This does not prove that 7 is prime, it is merely an example of Fermat's little theorem for \( n = 7 \) and \( a = 2 \). Note, however, that if we have two integers \( n > 1 \) and \( a \) such that \( n \) and \( a \) do not have any factor in common, and such that

\[
a^{n-1} \not\equiv 1 \mod n,
\]

then \( n \) cannot be a prime number because that would contradict Fermat's little theorem. Therefore, Fermat's little theorem can be used to prove that a number is composite. An \( a \) that can be used in this way to prove the compositeness of
n is often called a witness to the compositeness of n. For instance, for n = 15 and a = 2 we find that

\[ 2^{14} = 16384 = 4 + 1092 \cdot 15 \equiv 4 \not\equiv 1 \mod 15, \]

so that 2 is a witness to the compositeness of 15.

This is certainly not the fastest way to prove that 15 is composite—indeed, it is much faster to note that 15 = 3 \cdot 5. But for general n, finding a factor of \( n \) is much harder than computing \( a^{n-1} \mod n \), because the latter can be done using a quick method called repeated square and multiply. Using this method in the example, we would compute

\[
\begin{align*}
2^2 \mod 15 &= 4, \\
2^3 \mod 15 &= 2 \cdot (2^2 \mod 15) \mod 15 = 2 \cdot 4 = 8, \\
2^6 \mod 15 &= (2^3 \mod 15)^2 \mod 15 = 8^2 \mod 15 = 64 - 4 + 4 \cdot 15 \equiv 4 \mod 15, \\
2^7 \mod 15 &= 2 \cdot (2^6 \mod 15) \mod 15 \equiv 2 \cdot 4 = 8,
\end{align*}
\]

and

\[ 2^{14} \mod 15 = (2^7 \mod 15)^2 \mod 15 = 8^2 \mod 15 = 64 \equiv 4 \mod 15. \]

For general \( n \) all numbers involved in this computation are \(< n^2\), and the number of squares and multiplies is bounded by \( 2 \cdot \log_2(n) \) (where \( \log_2 \) denotes the base 2 logarithm). The pattern of squares and multiplies can be found by looking at the binary representation of the exponent \( n - 1 \) (cf. [7]).

Thus, we can compute \( a^{n-1} \mod n \) efficiently, which should allow us to easily prove that \( n \) is composite: simply pick a random \( a \) with \( 1 < a < n \), check that \( n \) and \( a \) are coprime, compute \( a^{n-1} \mod n \) if they are, and hope that the outcome is not equal to 1. Unfortunately, this process does not work for all composite \( n \): there are composite numbers for which \( a^{n-1} \equiv 1 \mod n \) for all \( a \) that are coprime to \( n \). These numbers are called 'Carmichael numbers'. It has recently been proved that there are infinitely many of them, which invalidates this simple compositeness test based on Fermat's little theorem: for a Carmichael number \( n \) the test \( a^{n-1} \equiv 1 \mod n \) never fails, if \( n \) and \( a \) are coprime, and therefore never proves the compositeness of \( n \).

Fortunately, there is an easy fix to this problem. For a slight variation of the above test it can be proved that a randomly selected integer in \( \{2, 3, \ldots, n-1\} \), for an odd composite integer \( n \), has a chance of at least 75% to be a witness to \( n \)'s compositeness. This makes proving compositeness of \( n \) in practice an easy
matter: apply the altered version of the test for randomly picked $a$'s, until an $a$ is found that is a witness to the compositeness of $n$. If no witness can be found after some reasonable number of attempts, the compositeness test fails, and $n$ is declared to be probably prime. The chance that a composite number is declared to be probably prime after $k$ trials is less than $1/4^k$. Note that a probably prime number is only a number for which we failed to prove the compositeness—this does not imply that its primality has been proved. Refer to [7] and [9] for this 'improved' version of Fermat's little theorem. In [11: 2.5] it is shown how this method can also be used to rule out prime powers.

**Factoring and public-key cryptography.** In public-key cryptography each party has two keys: a public key and a corresponding secret key. Anyone can encrypt a message using the public key of the intended recipient, but only parties that know the secret key can decrypt the encrypted message. One way to make such a seemingly impossible system work is based on the supposed difficulty of factoring. The 'RSA-system' (named after the inventors Ron Rivest, Adi Shamir, and Len Adleman, cf. [21]) works as follows. Each party generates two sufficiently large primes $p$ and $q$, selects integers $e$ and $d$ such that $e \cdot d \equiv 1 \mod (p - 1)(q - 1)$, and computes the product $n = p \cdot q$; the public key consists of the pair $(n, e)$, the secret key consists of the integer $d$. This computation can be carried out efficiently: randomly picked numbers can easily be checked for primality using probabilistic primality tests (as shown above); the density of primes is sufficiently high due to the 'Prime number theorem'; $d$ can be derived from $e$, $p$, and $q$, using the extended Euclidean algorithm (if $e$ and $(p - 1)(q - 1)$ are coprime); and multiplication is easy. It follows that $d$ can be found if the factors of $n$ are known; conversely, it is believed that factoring $n$ is necessary to be able to decrypt RSA-encrypted messages.

Let the message $m$ be a bit string shorter than $n$. To encrypt $m$ using the public key $(n, e)$ one computes $E(m) = m^e \mod n$. To decrypt the encrypted message $E(m)$ the recipient computes $E(m)^d \mod n$, which is equal to $m$ because of Fermat's little theorem and the Chinese remainder theorem (cf. [7]). The modular exponentiations can be done efficiently using the 'repeated square and multiply' method, as shown above.

RSA can also be used as a signature scheme: the owner of secret key $d$, whose public key is $(n, e)$, is the only one who can compute the signature $S(m) = m^d \mod n$ for some message $m$, but everyone can check that $S(m)$ is the signature on $m$ of the owner of the secret key corresponding to $(n, e)$, by verifying that $S(m)^e \mod n$ equals the original message $m$.

ROOM MCC-1C317B, BELLcore, 445 SOUTH STREET, MORRISTOWN, NJ 07960-6438, U.S.A.
E-mail address: lenstra@bellcore.com