
Designs, Codes and Cryptography, 19, 101–128 (2000)

Integer Factoring

ARJEN K. LENSTRA arjen.lenstra@citicorp.com
Citibank, N.A., 1 North Gate Road, Mendham, NJ 07945-3104, USA

Abstract. Using simple examples and informal discussions this article surveys the key ideas and major advances of the last quarter century in integer factorization.

Keywords: Integer factorization, quadratic sieve, number field sieve, elliptic curve method, Morrison–Brillhart Approach

1. Introduction

Factoring a positive integer n means finding 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 $v = 15$. A factorization of a composite number is not necessarily unique: $n = 105$ can also be factored as the product of the prime $u = 5$ and the composite $v = 21$. But the *prime factorization* of a number—writing it as a product of prime numbers—is unique, up to the order of the factors: $n = 3 \cdot 5 \cdot 7$ is *the* prime factorization of $n = 105$, and $n = 5$ is the prime factorization of $n = 5$.

In this article we concentrate on finding just a factorization. The prime factorization can be obtained by further factoring the factors that happen to be composite: both factorizations $n = 7 \cdot 15$ and $n = 5 \cdot 21$ of $n = 105$ can be further refined to the prime factorization $n = 3 \cdot 5 \cdot 7$ of $n = 105$, the first by further factoring 15, the second by factoring 21. There are efficient methods to distinguish primes from composites that do not require factoring the composites (cf. [29], [50], and Section 2). 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. As yet there is no firm mathematical ground on which this assumption can be based. The only evidence that factoring is hard consists of our failure so far to find a fast and practical factoring algorithm. (The polynomial-time factoring algorithms that are based on the use of quantum computers are not considered to be practical and not addressed in this survey.) Interestingly, and to an outsider maybe surprisingly, an entire industry is based on this belief that factoring is hard: the security,

i.e., the unbreakability, of one of the most popular public key cryptosystems relies on the supposed difficulty of factoring (cf. Appendix).

This relation between factoring and cryptography is one of the main reasons why people are interested in evaluating the practical difficulty of the integer factorization problem. Currently the limits of our factoring capabilities lie around 130 decimal digits. Factoring hard integers in that range requires enormous amounts of computing power. A cheap and convenient way to get the computing power needed is to distribute the computation over the Internet. This approach was first used in 1988 to factor a 100-digit integer [32], since then to factor many integers in the 100 to 120 digit range, and in 1994 to factor the famous 129-digit RSA-challenge number (cf. [4]).[1] Most recently, in 1996 a 130-digit number was factored, partially using a World Wide Web interface [13].

This survey is intended for people who want to get an impression 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 we explain how these methods can be run in parallel on a loosely coupled computer network, such as the Internet.

We distinguish 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. The latter are referred to as *general-purpose algorithms* and have an expected run time that depends solely on the size of the number n being factored. The former are called *special-purpose algorithms*; they have an expected run time that also depends on the properties of the—unknown—factors of n. When evaluating the security of factoring-based cryptosystems, people employ general-purpose factoring algorithms. This survey therefore focuses on this category of integer factoring algorithms, after a short description how primes can be efficiently distinguished from composites (Section 2) and of some of the most important special purpose algorithms in Section 3.

In Section 4 we sketch the basic approach of the general purpose algorithms. We show that they consist of two main steps: *data collection*, and *data processing*. Section 5 concentrates on the quadratic sieve factoring algorithm, the algorithm that was considered to be the most practical general purpose method from 1982 to 1994. We describe the data collection step, how it can be improved using some simple additional tricks, and how it can be parallelized over a network. Section 5 concludes with some data from quadratic sieve factoring efforts.

The algorithm that is currently considered to be the most practical method (for sufficiently large numbers)—the number field sieve—is sketched in Section 6. This sketch offers only a vague indication of the algorithm: it omits most of the mathematics required to fully understand the algorithm. In the appendix we describe the relation between factoring and cryptography.

Understanding the material presented in this survey requires some willingness to bear with a few easy examples and a few slightly more complicated formulas and descriptions. 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 this survey provides a useful introduction to factoring that inspires the readers to consult the literature referred to in the references.

2. Preliminaries

Notation. By '$\log_b$' we denote the base b logarithm, and 'ln' denotes the natural logarithm, i.e., $\ln = \log_e$ with $e \approx 2.71828$. The largest integer $<= x$ is denoted by '$[x]$'. The number of primes $\leq x$ is denoted by '$\pi(x)$', the *prime counting function*; due to the *Prime number theorem* [24] we know that $\pi(x) \approx x/\ln(x)$. To give an example of some values of the prime counting function and its growth rate:

$$\{\pi(10^i)\colon\ 1 \leq i \leq 19\}$$
$$= \{4,\ 25,\ 168,\ 1229,\ 9592,\ 78498,\ 664579,\ 57\,61455,\ 508\,47534,\ 4550\,52511,$$
$$41180\,54813,\ 3\,76079\,12018,\ 34\,60655\,36839,\ 320\,49417\,50802,$$
$$2984\,45704\,22669,\ 27923\,83410\,33925,\ 2\,62355\,71576\,54233,$$
$$24\,73995\,42877\,40860,\ 234\,05766\,72763\,44607\};$$

furthermore, $\pi(4185\,29658\,14676\,95669) = 10^{17}$ (cf. [16,27]). Building a table of all primes $\leq 10^{50}$ or of all 256-bit primes, as has often been proposed, is therefore completely infeasible.

Smoothness. We say that a positive integer is *B-smooth* if all its prime factors are $<= B$. An integer is said to be *smooth with respect to S*, where S is some set of integers, if it can be completely factored using the elements of S. We often simply use the term *smooth*, in which case the bound B or the set S is clear from the context.

Smoothness probability. In the algorithms described in this paper we are interested in the probability that randomly selected integers are smooth. Intuitively, the smaller a number is, the higher the probability that it is smooth. For example, there are 39 positive 5-smooth numbers ≤ 143, but there are 29 positive 5-smooth numbers ≤ 72. Therefore, if we randomly pick positive numbers $\leq m$, we get a smoothness probability of $39/143 = 0.27$ for $m = 143$ but a higher probability $29/72 = 0.40$ for $m = 72$. For $m = 1000$ we get $87/1000 = 0.08$, and for $m = 10^6$ we get only $508/10^6 = 0.0005$.

To express how the smoothness probability depends on the smoothness bound and the size of the numbers involved, we introduce

$$L_x[u, v] = \exp(v(\ln x)^u(\ln\ln x)^{1-u}).$$

Let α, β, r, and s be real numbers with $\alpha, \beta > 0$, $0 < r \leq 1$, and $0 < s < r$. It follows from [10,15] that a random positive integer $\leq L_x[r, \alpha]$ is $L_x[s, \beta]$-smooth with probability $L_x[r - s, -\alpha(r - s)/\beta + o(1)]$ for $x \to \infty$. Thus, with $r = 1$ and $s = 1/2$, a random positive integer $\leq n^\alpha$ is $L_n[1/2, \beta]$-smooth with probability $L_n[1/2, -\alpha/(2\beta) + o(1)]$, for $n \to \infty$.

Run times. Throughout this survey, the function L_n is often used in run time estimates. Note that, in such applications, its first argument interpolates between polynomial time ($u = 0$) and exponential time ($u = 1$) in $\ln n$:

$$L_n[0, v] = \exp(v \ln\ln n) = (\ln n)^v \quad \text{and} \quad L_n[1, v] = \exp(v \ln n) = n^v.$$

A run time $L_n[u, v]$ with $u < 1$ and v constant is referred to as *subexponential time*.

All run times involving L_n are for $n \to \infty$. This implies that the $o(1)$'s that might occur in run time expressions go 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 factoring methods presented here 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.

Modular arithmetic. Throughout this paper '$x \equiv y \bmod z$' means that $x - y$ is a multiple of z, for integers x, y, and z with $z > 0$. Similarly, '$x \not\equiv y \bmod z$' means that $x - y$ is not a multiple of z. Thus, $308 \equiv 22 \bmod 143$ because $308 - 22 = 286 = 2 \cdot 143$ is a multiple of 143, and $143 \equiv 11 \bmod 22$ because $143 - 11 = 132 = 6 \cdot 22$ is a multiple of 22; but $4 \not\equiv 1 \bmod 15$ because $4 - 1 = 3$ is not a multiple of 15. By '$x \bmod z$' we mean any integer y such that $x \equiv y \bmod z$; in practical circumstances we often use the *least non-negative remainder*, i.e., we assume that $0 \le y < z$, or the *least absolute remainder*, i.e., we assume that $-z/2 < y \le z/2$. Thus, by $143 \bmod 22$ we mean 11, or 33, or -11, or any integer of the form $11 + k \cdot 22$, for some integer k; the least non-negative remainder and the least absolute remainder of $143 \bmod 22$ are both equal to 11.

Note that given $x \bmod z$ and $y \bmod z$ it it possible to efficiently compute $(x+y) \bmod z$, $(x-y) \bmod z$, or $(x \cdot y) \bmod z$: simply compute $(x \bmod z) + (y \bmod z)$, $(x \bmod z) - (y \bmod z)$, or $(x \bmod z) \cdot (y \bmod z)$ and if necessary remove multiples of z from the result if least remainders are used. The latter operation can be done using a division with remainder by z. Examples of this so-called *modular arithmetic* (with *modulus* z) can be found throughout the paper.

To be able to divide in modular arithmetic, for instance to compute $(1/x) \bmod z$, we need a little more. An important operation on which many factoring and other algorithms rely is finding the *greatest common divisor* of two non-negative integers, say x and z, i.e., the largest factor that x and z have in common. Of course, the greatest common divisor of x and z ('$\gcd(x, z)$' for short) can be found by computing the prime factorizations of x and z and multiplying all prime factors they have in common. A much faster method to compute $\gcd(x, z)$ is *Euclid's algorithm*, a method that was invented more than 2000 years ago. It is based on the observation that $\gcd(x, 0) = x$, that $\gcd(x, z) = \gcd(z, x \bmod z)$ if $z \neq 0$, and that, if $x \ge z$ and least non-negative remainders are used, the 'new' pair $(z, x \bmod z)$ is substantially 'smaller' than the 'old' pair (x, z). As an example:

$$\gcd(308, 143) = \gcd(143, 22) = \gcd(22, 11) = \gcd(11, 0) = 11,$$

and

$$\gcd(143, 19) = \gcd(19, 10) = \gcd(10, 9) = \gcd(9, 1) = \gcd(1, 0) = 1.$$

If $\gcd(x, z) = 1$ as in the latter example, we say that x and z are coprime, i.e., x and z do not have any factors > 1 in common.

If x and z are coprime, we can compute $(1/x) \bmod z$, using a variant of Euclid's algorithm that is generally referred to as the *extended Euclidean algorithm*. Actually, the extended Euclidean algorithm does more: it computes $\gcd(x, z)$ and, if the latter equals 1, it computes

$(1/x) \bmod z$ as well. The process is illustrated in the following example where we compute $(1/19) \bmod 143$. In the ith line we have $x = 19$, $z = 143$ and two other numbers, r_i and s_i, such that $x \cdot r_i \equiv s_i \bmod z$. Assuming that $0 \leq x < z$ we have $r_1 = 0$, $s_1 = z$, $r_2 = 1$, and $s_2 = x$. The $(i+1)$st line follows from the $(i-1)$st and ith by subtracting the ith as many times as possible from the $(i-1)$st, without making the right hand side of the resulting $(i+1)$st line negative. The process terminates as soon as some $s_i = 0$; if $s_k = 0$ then $s_{k-1} = \gcd(x, z)$, and if s_{k-1} equals 1, then $r_{k-1} \equiv (1/x) \bmod z$:

$$\begin{aligned}
19 \cdot 0 &\equiv 143 \bmod 143 \\
19 \cdot 1 &\equiv 19 \bmod 143 \quad \text{(subtract } [143/9] = 7 \text{ times)} \\
19 \cdot (-7) &\equiv 10 \bmod 143 \quad \text{(subtract } [19/10] = 1 \text{ times)} \\
19 \cdot 8 &\equiv 9 \bmod 143 \quad \text{(subtract } [10/9] = 1 \text{ times)} \\
19 \cdot (-15) &\equiv 1 \bmod 143 \quad \text{(subtract } [9/1] = 9 \text{ times)} \\
19 \cdot 143 &\equiv 0 \bmod 143 \quad \text{(done).}
\end{aligned}$$

Thus, $128 = -15 + 143$ is the least non-negative remainder of $(1/19) \bmod 143$. We say that 128 is the *inverse* of 19 modulo 143. Note that the numbers on the right hand sides in the example also appear in the earlier example where we computed gcd(143, 19). For more background on Euclid's algorithm and the extended Euclidean algorithm see [25].

Compositeness testing. A famous theorem of Fermat (his *little theorem*) says that if n is prime and a is an integer that is not divisible by n, then

$$a^{n-1} \equiv 1 \bmod n.$$

For instance, for $n = 7$ and $a = 2$ we find that

$$2^6 = 64 = 1 + 9 \cdot 7 \equiv 1 \bmod 7.$$

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

$$a^{n-1} \not\equiv 1 \bmod n,$$

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

$$2^{14} = 16384 = 4 + 1092 \cdot 15 \equiv 4 \not\equiv 1 \bmod 15,$$

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 $a^{n-1} \bmod n$, because the latter can be done using a quick method called *repeated square and multiply*. Using this method in the example, we compute

$$2^2 \bmod 15 = 4,$$

$$2^3 \bmod 15 = 2 \cdot (2^2 \bmod 15) \bmod 15 = 2 \cdot 4 = 8,$$

$$2^6 \bmod 15 = (2^3 \bmod 15)^2 \bmod 15 = 8^2 \bmod 15 = 64 = 4 + 4 \cdot 15 \equiv 4 \bmod 15,$$

$$2^7 \bmod 15 = 2 \cdot (2^6 \bmod 15) \bmod 15 \equiv 2 \cdot 4 = 8,$$

and

$$2^{14} \bmod 15 = (2^7 \bmod 15)^2 \bmod 15 = 8^2 \bmod 15 = 64 \equiv 4 \bmod 15.$$

If we use least non-negative remainders, all numbers involved in this computation are $< n^2$. The number of squares and multiplies is bounded by $2 \cdot \log_2(n)$. The pattern of squares and multiplies can be found by looking at the binary representation of the exponent $n - 1$ (cf. [25]).

Thus, we can compute $a^{n-1} \bmod n$ efficiently, which should allow us to easily prove that n is composite if we simplemindedly assume that witnesses are not too rare: simply pick a random a with $1 < a < n$, check that n and a are coprime[2], compute $a^{n-1} \bmod n$ 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 $a^{n-1} \equiv 1 \bmod n$ for all a that are coprime to n. These numbers are called *Carmichael numbers*; the smallest one is 561. It has recently been proved that there are infinitely many Carmichael numbers: there are at least $x^{2/7}$ of them $\leq x$, once x is sufficiently large (cf. [2]). This invalidates the simple compositeness test based on Fermat's little theorem: for a Carmichael number n the test $a^{n-1} \equiv 1 \bmod n$ 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, if we use Selfridge's slight variation of Fermat's little theorem: if n is an odd prime, $n - 1 = 2^t \cdot u$ for integers t and u with u odd, and a is an integer that is not divisible by n, then

$$\text{either } a^u \equiv 1 \bmod n \text{ or } a^{2^i u} \equiv -1 \bmod n \text{ for some } i \text{ with } 0 \leq i < t.$$

For odd composite n it can be proved that a randomly selected integer $a \in \{2, 3, \ldots, n-1\}$ has a chance of at least 75% not to satisfy these conditions and thereby be a witness to n's compositeness (cf. [38,49]); see also [3]. This makes proving compositeness of n in practice an easy matter: apply Selfridge's 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; proving primality is an entirely different subject which will not be discussed in this paper. In [31: 2.5] it is shown how Selfridge's test can also be used to rule out prime powers.

3. Special Purpose Factoring Algorithms

We briefly discuss six of the most important special purpose factoring methods: *trial division*, *Pollard's rho method*, *Pollard's* $p-1$ *method*, the *elliptic curve method*, *Fermat's method*, and *squfof*. None of these methods is currently considered to be applicable to composites that are used in cryptosystems. But for numbers that come from different sources, and that might have small or otherwise 'lucky' factors, any of these methods can be quite useful. Examples are the eighth, tenth, and eleventh Fermat numbers ($F_k = 2^{2^k}+1$ for $k = 8, 10, 11$ cf. [8,7]), and also numbers that have to be factored in the course of the general purpose algorithms described in the next sections.

Throughout this section n denotes the number to be factored. Using the results from Section 2 we may assume that n is composite and not a prime power.

Trial division. The smallest prime factor p of n can in principle be found by trying if n is divisible by 2, 3, 5, 7, 11, 13, 17, ..., i.e., all primes in succession, until p is reached. If we assume that a table of all primes $\leq p$ is available (which can be generated in approximately p steps using for instance the *sieve of Erathostenes*, cf. [25]), this process takes $\pi(p)$ division attempts (so-called 'trial divisions'), where π is the prime counting function from Section 2. Because $\pi(p) \approx p/\ln(p)$, finding the factor p of n in this way takes at least approximately p steps—how many precisely depends on how we count the cost of each trial division. Even for fairly small p, say $p > 10^6$, trial division is already quite inefficient compared to the methods described below.

Since n has at least one factor $\leq \sqrt{n}$, factoring n using trial division takes approximately $\sqrt{n}$ operations, in the worst case. For many composites trial division is therefore infeasible as factoring method. For most numbers it is very effective, however, because most numbers have small factors: 88% of all positive integers have a factor < 100, and almost 92% have a factor < 1000.

Pollard's rho method. Pollard's rho method [44] is based on a combination of two ideas that are also useful for various other factoring methods. The first idea is the well known *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 p numbers, the probability of picking the same number twice exceeds 50% after $1.177\sqrt{p}$ numbers have been picked. The first duplicate can be expected after $c \cdot \sqrt{p}$ numbers have been selected, for some small constant c. The second idea is the following: if p is some unknown divisor of n and x and y are two integers that are suspected to be identical modulo p, i.e., $x \equiv y \bmod p$, then this can be checked by computing $\gcd(|x-y|, n)$; more importantly, this computation may reveal a factorization of n, unless x and y are also identical modulo n.

These ideas can be combined into a factoring algorithm in the following way. Generate a sequence in $\{0, 1, \ldots, n-1\}$ by randomly selecting x_0 and by defining x_{i+1} as the least non-negative remainder of $x_i^2 + 1 \bmod n$. Since p divides n the least non-negative remainders $x_i \bmod p$ and $x_j \bmod p$ are equal if and only if x_i and x_j are identical modulo p. Since the $x_i \bmod p$ behave more or less as random integers in $\{0, 1, \ldots, p-1\}$ we can expect to factor n by computing $\gcd(|x_i - x_j|, n)$ for $i \neq j$ after about $c\sqrt{p}$ elements of the sequence have been computed.

This suggests that approximately $(c\sqrt{p})^2/2$ pairs x_i, x_j have to be considered. However, this can easily be avoided by only computing $\gcd(|x_i - x_{2i}|, n)$, for $i = 0, 1, \ldots$, i.e., by generating two copies of the sequence, one at the regular speed and one at the double speed, until the sequence 'bites in its own tail' (which explains the 'rho' (ρ) in the name of the method); this can be expected to result in a factorization of n after approximately $2\sqrt{p}$ gcd computations.

As an example, consider $n = 143$ and $x_0 = 2$:

$$x_1 = 2^2 + 1 = 5, x_2 = 5^2 + 1 = 26 : \gcd(|5 - 26|, 143) = 1,$$

$$x_2 = 26, x_4 = (26^2 + 1)^2 + 1 \equiv 15 \bmod 143 : \gcd(|26 - 15|, 143) = 11.$$

With $x_0 = 3$ it goes even faster, but we find a different factor:

$$x_1 = 3^2 + 1 = 10, x_2 = 10^2 + 1 = 101 : \gcd(|10 - 101|, 143) = 13.$$

The most remarkable success of Pollard's rho method so far was the discovery in 1980 by Brent and Pollard of the factorization of the eighth Fermat number (cf. [8]):

$$2^{2^8} + 1 = 1\,23892\,63615\,52897 \cdot p62,$$

where $p62$ denotes a 62-digit prime number.

Pollard's $p-1$ method. Pollard's $p-1$ method [43] follows, very roughly, from Pollard's rho method by replacing the birthday paradox by Fermat's little theorem (cf. Section 2). Let p again be a prime factor of n. For any integer a with $1 < a < p$ we have, according to Fermat's little theorem, that $a^{p-1} \equiv 1 \bmod p$, so that $a^{k(p-1)} \equiv 1^k \equiv 1 \bmod p$ for any integer k. Therefore, for any multiple m of $p-1$ we have that $a^m \equiv 1 \bmod p$, i.e., p divides $a^m - 1$. Thus, computing $\gcd(a^m - 1, n)$ might reveal a factorization of n. Note that it suffices to compute $\gcd((a^m - 1) \bmod n, n)$ (and that p divides $(a^m - 1) \bmod n$ as well, because p divides n).

It remains to find a multiple $m > 1$ of $p - 1$. The idea here is that one simply hopes that $p - 1$ is B-smooth (cf. Section 2) for some relatively small bound B, i.e., that $p - 1$ has only prime factors $\leq B$. This would imply that an m of the form $\prod_{q \leq B} q$, with the product ranging over prime powers q, could be a multiple of $p - 1$. Since $(a^m - 1) \bmod n$ for such m can be computed in time roughly proportional to B, Pollard's $p - 1$ method can be used to discover factors p in time roughly proportional to the largest prime factor in $p - 1$. Evidently, this is only going to be efficient for p for which $p - 1$ is smooth. It explains why some people insist on using primes of the form $2q + 1$ (with q prime) in factoring-based cryptosystems, a precaution that is rendered useless by the elliptic curve method.

As an example, let n again be 143, and let $a = 2$. If we raise a to small successive prime powers and compute the relevant gcd's, we find $p = 13 = 2^2 \cdot 3 + 1$ after processing the prime powers 2^2 and 3:

$$2^4 = 16, \gcd(16 - 1, 143) = 1,$$

$$16^3 = (16^2) \cdot 16 \equiv 113 \cdot 16 \equiv 92 \bmod 143, \gcd(92 - 1, 143) = 13.$$

If, on the other hand, we simply keep raising $a = 2$ to the next prime, we find $p = 11 = 2 \cdot 5 + 1$ after processing the primes 2, 3, and 5:

$$2^2 = 4, \gcd(4 - 1, 143) = 1,$$

$$4^3 = 64, \gcd(64 - 1, 143) = 1,$$

$$64^5 = (64^2)^2 \cdot 64 \equiv 92^2 \cdot 64 \equiv 12 \bmod 143, \gcd(12 - 1, 143) = 11.$$

For variations of Pollard's $p - 1$ method and fast ways to implement it refer to [39].

The elliptic curve method. The major disadvantage of Pollard's $p - 1$ method is that it only works efficiently if the number to be factored happens to have a factor p for which $p - 1$ is B-smooth, for some reasonably small bound B. So, it only works for 'lucky' n. The elliptic curve method [34] can be regarded as a variation of the $p - 1$ method that does not have this disadvantage. It consists of any number of trials, where each trial can be lucky—and factor n—independently of the other trials: a trial is successful if some random number close to some prime factor of n is smooth. Thus, the probability of success of each trial depends only on the size and not on any other fixed properties of the factors of n (cf. Section 2).

A detailed description of the method is beyond the scope of this survey. Roughly speaking, the following happens. During each trial an elliptic curve modulo n is selected at random. For any prime p dividing n, any point a on the curve satisfies an equation that is similar to Fermat's little theorem, with two important differences. In the first place, and this is why the elliptic curve method is so powerful, the exponent $p - 1$ is replaced by some random number $\hat{p}$ close to $p - 1$. Secondly, the exponentiation is not a regular integer exponentiation modulo n: since a is not an integer but a point on a curve, other operations have to be performed on it to 'exponentiate on the curve'. The number of elementary arithmetic operations to be carried out for such an exponentiation is a constant multiple of the number of operations needed for a regular integer exponentiation modulo n with the same exponent.

Just as in Pollard's $p - 1$ method it is the case that if a is exponentiated on the curve to a power that is a multiple of $\hat{p}$, then a factorization of n may be discovered; if $\hat{p}$ is B-smooth, then this can be done in roughly $c(\ln n)^2 B$ elementary arithmetic operations, where c is a small constant. Thus, it suffices to keep trying new curves (thereby getting new $\hat{p}$'s), and to exponentiate the points to large smooth powers, till a $\hat{p}$ divides the smooth power.

From the smoothness probability in Section 2, and assuming that $\hat{p}$ behaves as a random positive integer close to p, it follows that $\hat{p}$ is $L_p[1/2, \sqrt{1/2}]$-smooth with probability $L_p[1/2, -\sqrt{1/2} + o(1)]$, for $p \to \infty$. Therefore, if one runs $L_p[1/2, \sqrt{1/2} + o(1)]$ trials in parallel, spending time proportional to $(\ln n)^2 L_p[1/2, \sqrt{1/2}]$ per trial, one may expect to find p. We find that the heuristic asymptotic expected run time of the elliptic curve method to find the smallest prime factor p of n is

$$(\ln n)^2 L_p[1/2, \sqrt{2} + o(1)],$$

for $p \to \infty$. In the worst case, i.e., $p \approx \sqrt{n}$, this becomes $L_n[1/2, 1 + o(1)]$, for $n \to \infty$ (note that the $(\ln n)^2$ disappears in the $o(1)$). Thus, in the worst case the elliptic curve method can be expected to run in subexponential time. This is substantially faster than any

of the other methods discussed in this section, which all have an exponential-time worst case behavior.

Two remarkable factorizations obtained using the elliptic curve method are those of the tenth and eleventh Fermat numbers, both by Brent[3] [7]. In 1988 he found a 21 and a 22-digit factor of $(2^{2^{11}}+1)/(319489 \cdot 974849)$, thereby completing the factorization of F_{11}:

$$2^{2^{11}}+1 = 319489 \cdot 974849 \cdot 1\,67988\,55634\,17604\,75137 \cdot 35\,60841\,90644\,58339\,20513 \cdot p564,$$

where $p564$ denotes a 564-digit prime; and in 1995 he found a 40-digit factor of $(2^{2^{10}}+1)/(45592577 \cdot 6487031809)$, which completed the factorization of F_{10}:

$$2^{2^{10}}+1 = 455\,92577 \cdot 64870\,31809 \cdot 46597\,75785\,22001\,85432\,64560\,74307\,67781\,92897 \cdot p252,$$

where $p252$ denotes a 252-digit prime. The largest factor found by the elliptic curve method, as of March 1996, has 47 digits (155 bits), and was found by P. L. Montgomery. For a complete description of the elliptic curve method refer to [34] and [29]. For implementation details, refer to [6, 39].

Fermat's method. In the course of the general purpose factoring methods described below we frequently have to factor numbers n that are suspected to have two relatively large prime factors and for which typically $2^{32} < n < 2^{64}$. If those factors are close to each other, they can easily be found using Fermat's method. Let $n = p_1 \cdot p_2$ with $p_1 < p_2$, both p_1 and p_2 odd, and $p_2 - p_1 = 2d$ for some small d. Then $x = p_1 + d$, $y = d$ satisfy $n = (x-y)(x+y)$, and therefore $n = x^2 - y^2$. The proper x can thus be found by trying $x = [\sqrt{n}]+1, [\sqrt{n}]+2, [\sqrt{n}]+3, \ldots$ in succession until $x^2 - n$ is a perfect square (in which case $y^2 = x^2 - n$). Obviously, this method is efficient only if d is small. For the example $n = 143$ Fermat's method needs only one trial: the first x equals $[\sqrt{143}]+1 = 12$ and $x^2 - n = 12^2 - 143 = 1$ is a perfect square, so that $x = 12$, $y = 1$, and $143 = (12-1)(12+1)$.

Congruence of squares. More generally, in Fermat's method one attempts to solve a *congruence of squares*, i.e., integers x and y such that $x^2 - y^2$ is a *multiple* of 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 is a factor of $x - y$, which can be checked easily. In the second case, n is 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 gives us a way to find factors of n because we have an efficient method to find out which factors n and $x - y$ have in common, and which factors n and $x + y$ have in common: as we have seen in Section 2 we simply compute $\gcd(n, x \pm y)$, the greatest common divisor of n and $x \pm y$. If n is composite, not a prime power, and x and y are random integers satisfying $x^2 \equiv y^2 \bmod n$, then there is at least a 50% chance that $\gcd(x-y, n)$ and $\gcd(x+y, n)$ are non-trivial factors of n.

Fermat's method is surprisingly efficient in the application mentioned above, and often more efficient than Pollard's rho method. The reason is that Pollard's rho method requires

rather intensive arithmetic on numbers modulo n, which is relatively inefficient for such small n that are nevertheless too large to be conveniently handled on most 32-bit processors. Another method that is particularly efficient in this case is the following.

Squfof. Squfof stands for 'square form factorization'. It makes use of binary quadratic forms, a subject that is beyond the scope of this survey. The expected time needed by squfof to factor n is proportional to $n^{1/5}$, on assumption of certain generalized Riemann hypotheses. After a short initialization it only requires arithmetic on numbers that are at most $\sqrt{n}$. This makes the method remarkably efficient for the application mentioned above, when run on 32-bit processors. For a description of squfof refer to [11,52,53].

4. The Morrison–Brillhart Approach

Most factorizations mentioned in the introduction were obtained using the *quadratic sieve* factoring algorithm, Carl Pomerance's variation (1981, cf. [46][4]) 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, on which also Fermat's method is based. There we have seen that to factor n it is useful to find integers x and y such that $x^2 - y^2$ is a multiple of n. Summarizing the argument presented above, if $x^2 \equiv y^2 \bmod n$, then n divides $(x - y)(x + y)$, and therefore

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

Since gcd's can be computed rapidly, one can quickly check whether the latter identity leads to a factorization of n, and if n is composite there is at least a 50% chance that the factorization is non-trivial.

Finding congruences of squares. For practical purposes in order to factor n, one need only generate a few random looking pairs x, y such that $x^2 \equiv y^2 \bmod n$. Note that simply picking some random positive v, computing s_v as the least non-negative 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 $s_v \equiv v^2 \bmod n$ is a perfect square, we collect v's for which s_v satisfies a certain much weaker condition. Once we have a sufficient number of pairs v, s_v, we combine them to solve $x^2 \equiv y^2 \bmod n$. Thus, the factoring process (i.e., the method to obtain solutions to the congruence $x^2 \equiv y^2 \bmod n$) 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', i.e., s_v should be B-smooth for some reasonably small B (cf. Section 2). How the pairs v, s_v can be combined can be seen in the example below.

To find pairs v, s_v such that s_v is smooth Morrison and Brillhart, in their original paper that introduced the Morrison-Brillhart approach, used a technique based on continued fractions. For a description of their method, 'CFRAC', see [42]. It was used, in 1970, to factor the seventh Fermat number:

$$2^{2^7} + 1 = 59\,64958\,91274\,97217 \cdot 57\,04689\,20068\,51290\,54721.$$

A less efficient but conceptually much easier method to find pairs v, s_v such that s_v is smooth is *Dixon's algorithm*: simply randomly pick v's and keep those for which s_v is smooth until we have sufficiently many different pairs v, s_v for which s_v is smooth.

An example using random squares. Even though we already know that $n = 143 = 11 \cdot 13$, here is how Dixon's version of the Morrison–Brillhart approach works for $n = 143$. Since factors 2, 3, and 5 can easily be recognized, we use $B = 5$, i.e., 's_v should be 5-smooth', or 'it should be possible to factor s_v completely using only 2, 3, and 5'. In general, for larger numbers than 143, a larger B will be used, so that more primes will be allowed in the factorization of s_v. This set of primes is usually referred to as the *factor base*; we will be interested in s_v's that are smooth with respect to the factor base. In the example, the factor base is the set $\{2, 3, 5\}$.

Since we use Dixon's algorithm we begin by randomly selecting some integer v; let $v = 17$ be the first random choice. We find that $v^2 = 289 = 3 + 2 \cdot 143 \equiv 3 \bmod 143$, so that $s_{17} = 3$. Obviously, $s_{17} = 3$ is smooth, so that we find the identity

$$17^2 \equiv 2^0 \cdot 3^1 \cdot 5^0 \bmod 143;$$

thus, we keep the pair v, s_v for $v = 17$. Such identities are often referred to as *relations*—relations are the data collected during the data collection step. Since $(v+1)^2 = v^2 + 2v + 1$, a convenient next choice is $v = 18$: $18^2 = 17^2 + 2 \cdot 17 + 1 \equiv 3 + 35 = 38 = 2 \cdot 19 \bmod 143$, and $s_{18} = 2 \cdot 19$ is not smooth, so that $v = 18$ can be thrown away. Proceeding to 19 we find that $19^2 = 18^2 + 2 \cdot 18 + 1 \equiv 38 + 37 = 75 \bmod 143$, and $s_{19} = 75$ is smooth, so that we keep $v = 19$ and have found our second relation:

$$19^2 \equiv 2^0 \cdot 3^1 \cdot 5^2 \bmod 143.$$

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

$$21^2 = 20^2 + 2 * 20 + 1 \equiv 114 + 41 = 155 = 12 + 143 \equiv 12 = 2^2 \cdot 3^1 \cdot 5^0 \bmod 143.$$

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:

$$(17 \cdot 19)^2 \equiv 2^0 \cdot 3^2 \cdot 5^2 \bmod 143,$$

$$(19 \cdot 21)^2 \equiv 2^2 \cdot 3^2 \cdot 5^2 \bmod 143, \text{ and}$$

$$(17 \cdot 21)^2 \equiv 2^2 \cdot 3^2 \cdot 5^0 \bmod 143.$$

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 last 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 first relation after the one for $v = 21$ would be $23^2 \equiv 2^2 \cdot 3^0 \cdot 5^2 \bmod 143$ which is already of the form $x^2 \equiv y^2 \bmod n$. 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 130 digit range, this *never* happened.

Finding the right combinations of relations. Suppose we have a set V of relations as a result of the data collection step. In the data processing step we have to pick a subset W of V so that the relations from W when multiplied together yield a solution to the congruence $x^2 \equiv y^2 \bmod n$. This can be achieved as follows. First observe that for any $W \subset V$ the product of the 'left hand sides' $\prod_{v \in W} v^2$ is a square, since it is a product of squares. The product of the corresponding 'right hand sides', however, is not always a square: for each prime p in the factor base the exponent in the product over W is the sum of the exponents of p in the relations in W, and this sum is not necessarily even. If we identify each relation with the vector of its exponents with respect to all elements of the factor base, the exponents of the factor base elements in the product over W are given by the vector that is the sum of the vectors for the relations in W. Thus, a W for which the product of the right hand sides is also a square can be found by looking for a subset of vectors whose sum is a vector with all even entries.

Finding all even combinations of vectors is a common problem in linear algebra, for which several good algorithms exist: (structured) Gaussian elimination, (blocked) Lanczos, and (blocked) Wiedemann are currently the most popular choices for our applications (see [12,28,41,48] and the references therein). In general, if there are m relations and k primes in the factor base, 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 above, we 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 $m > k$, 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 3×3-matrix.

The data processing step, i.e., finding the right combinations of relations, is often referred to as the *matrix step*.

The run time of Dixon's algorithm. As an example we show part of the run time analysis of Dixon's algorithm. Let $\beta > 0$. Assuming that the s_v behave as random numbers $\leq n$, it follows from the smoothness probabilities in Section 2 that s_v is $L_n[1/2, \beta]$-smooth with probability $L_n[1/2, -1/(2\beta) + o(1)]$. A single smooth s_v can therefore be

expected to be found after considering $L_n[1/2, 1/(2\beta) + o(1)]$ different v's. The number of smooth s_v's that are needed to make the matrix of exponents over-square is, roughly, $\pi(L_n[1/2, \beta]) \approx L_n[1/2, \beta]/\ln(L_n[1/2, \beta])$ (cf. Section 2), which can conveniently be written as $L_n[1/2, \beta + o(1)]$. It follows that a total of $L_n[1/2, \beta + 1/(2\beta) + o(1)]$ different v's have to be considered.

If we use trial division to check the smoothness of each s_v (at a cost of $L_n[1/2, \beta + o(1)]$ per s_v), the data collection step for Dixon's algorithm requires $L_n[1/2, 2\beta + 1/(2\beta) + o(1)]$ elementary operations. Using traditional matrix techniques, the right combinations of vectors can be found in $L_n[1/2, \beta + o(1)]^3 = L_n[1/2, 3\beta + o(1)]$ operations. Combining these run times, we find that Dixon's algorithm requires $L_n[1/2, \max(2\beta + 1/(2\beta), 3\beta) + o(1)]$ operations, which becomes $L_n[1/2, 2 + o(1)]$ for the optimal choice $\beta = 1/2$. With this approach the data collection takes more time than the matrix step.

If we use the elliptic curve method to check the s_v's for smoothness, each s_v costs only time $L_n[1/2, o(1)]$, so that the data collection step requires $L_n[1/2, \beta + 1/(2\beta) + o(1)]$ steps. Combined with the matrix step this yields $L_n[1/2, \max(\beta + 1/(2\beta), 3\beta) + o(1)] = L_n[1/2, 3/2 + o(1)]$ steps for the optimal choice $\beta = 1/2$. In this case the data collection and matrix steps take the same amount of time, asymptotically. But note that the data collection could have been done faster for $\beta = \sqrt{1/2}$, and that the matrix step forces us to use a β that is suboptimal for the data collection step. If we use the fact, however, that at most $\log_2(n)$ of the $L_n[1/2, \beta + o(1)]$ entries per exponent-vector can be non-zero and the fact that the Lanczos and Wiedemann methods referred to above process an $m \times m$ matrix with w non-zero entries in time proportional to mw, we get a combined time $L_n[1/2, \max(\beta + 1/(2\beta), 2\beta) + o(1)]$. This becomes $L_n[1/2, \sqrt{2} + o(1)]$ for the optimal choice $\beta = \sqrt{1/2}$; data collection and data processing again take the same amount of time, asymptotically.

Thus, with the elliptic curve method for trial division and a matrix step that takes advantage of the sparsity of the matrix, the asymptotic expected run time of Dixon's algorithm is $L_n[1/2, \sqrt{2} + o(1)]$, for $n \to \infty$. This expected run time can rigorously be proved and is not based on any unproved heuristics.

5. Quadratic Sieve

Finding relations faster, sieving. The smaller $|s_v|$ can be made, the higher probability 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 non-negative 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. 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 of $v = v(i)$ 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 $(i + tp)$th location of the sieve, for all integers t such that $i + tp$ is in the interval. 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 $\log_b p$ to the relevant locations, for some appropriately chosen base b.

Assuming that all sieve locations are initially zero, the ith location contains (after the sieving) the sum of the logarithms of those primes that divide $s_{v(i)}$. Therefore, if the ith 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[5].

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. This variation is due to P. L. Montgomery (extending an idea of Davis and Holdridge (cf. [14])) and described in [29,54].

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 an analysis of all these issues, as shown below and in the run time analysis of Dixon's algorithm. Refer to [37] for another informal description of QS.

The run time of Quadratic Sieve. Assuming that s_v behaves as a random integer close to $\sqrt{n}$, it is $L_n[1/2, \beta]$-smooth with probability $L_n[1/2, -1/(4\beta) + o(1)]$, which implies that $L_n[1/2, \beta + 1/(4\beta) + o(1)]$ different s_v's have to be considered. Using the elliptic curve method as smoothness test and taking advantage of the sparsity of the matrix (both as in the analysis of Dixon's algorithm), we find that QS has heuristic asymptotic expected run time $L_n[1/2, \max(\beta + 1/(4\beta), 2\beta) + o(1)] = L_n[1/2, 1 + o(1)]$ for the optimal choice $\beta = 1/2$.

If we use sieving to check $L_n[1/2, \beta + 1/(4\beta) + o(1)]$ consecutive s_v's for $L_n[1/2, \beta]$-smoothness we get the following. Sieving for one prime p takes time $L_n[1/2, \beta + 1/(4\beta) + o(1)]/p$. Sieving over 'all' primes therefore takes time $L_n[1/2, \beta + 1/(4\beta) + o(1)] \cdot \sum 1/p$, where the sum ranges over the first $\pi(L_n[1/2, \beta]) = L_n[1/2, \beta + o(1)]$ primes. The sum $\sum 1/p$ disappears in the $o(1)$, so that the complete sieving step takes time $L_n[1/2, \beta + 1/(4\beta) + o(1)]$. The remainder of the analysis remains the same, and we conclude that QS with sieving has the same heuristic asymptotic expected run time $L_n[1/2, 1 + o(1)]$ that we got for QS with elliptic curve smoothness testing. Note that both the sieving and the elliptic curve overhead disappear in the $o(1)$. In practice, however, sieving is much faster than elliptic curve smoothness testing.

Surprisingly, QS is not the only factoring algorithm with this subexponential expected run time: several other methods were proposed, some radically different from QS, that all have the same heuristic asymptotic expected run time as QS. Even the elliptic curve method 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 $L_n[1/2, 1 + o(1)]$ expected run time can be proved rigorously was published in [35]. As a consequence of this remarkable coincidence there was a growing suspicion that $L_n[1/2, 1 + o(1)]$ would be the best we would ever be able to do for factoring. The $L_n[1/2, 1 + o(1)]$-spell was eventually broken by the number field sieve (cf. Section 6).

Large primes, partial relations, and cycles. In practice, sieving is not a precise process: one often does not sieve with the small primes in the factor base, or with powers of elements of the factor base; $\log_b p$ is rounded to the nearest integer value; and the base b 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 that can be used 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 $\log_b |s_{v(i)}|$ 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 the double partials, 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:

$$18^2 \equiv 2^1 \cdot 3^0 \cdot 5^0 \cdot 19 \bmod 143.$$

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:

$$20^2 \equiv 2^1 \cdot 3^1 \cdot 5^0 \cdot 19 \bmod 143.$$

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

$$(18 \cdot 20)^2 \equiv 2^2 \cdot 3^1 \cdot 5^0 \cdot 19^2 \bmod 143.$$

Except for the '19^2' on the right hand side, this looks like a full relation. In Section 2 we have seen that $128 \equiv (1/19) \bmod 143$. Therefore, if we multiply both sides of the above 'almost smooth' relation by 128^2, we get

$$(128 \cdot 18 \cdot 20)^2 \equiv 2^2 \cdot 3^1 \cdot 5^0 \cdot (128 \cdot 19)^2 \equiv 2^2 \cdot 3^1 \cdot 5^0 \bmod 143,$$

which is, for factoring purposes, equivalent to the full relation

$$34^2 \equiv 2^2 \cdot 3^1 \cdot 5^0 \bmod 143$$

because $128 \cdot 18 \cdot 20 \equiv 34 \bmod 143$. Note that $(1/19) \bmod 143$ exists because 19 and 143 are coprime (cf. Section 2). If n and some large prime are not coprime, then that large prime must be a factor of n.

Double partials can be used in a slightly more complicated but similar way; it requires the factorization of the composite non-smooth factors of the $s_{v(i)}$'s, which can be done using the methods that were mentioned at the end of Section 3. 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 (cf. Section 3): 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. As shown in [33] the distribution of the large primes that we find in QS is not homogeneous, but strongly favors the relatively small large primes. This further increases the number of matches.

As illustrated in [32] and [33], 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 collected; 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 [4] and [17].

QS with large primes still runs in asymptotic expected time $L_n[1/2, 1 + o(1)]$; i.e., all savings disappear in the $o(1)$.

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 remove 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 [32] and [4] 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 [32].

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-factorization of a 116-digit, a 120-digit, and a 129-digit number (from [33], [17], and [4], respectively) are presented in Table 1. The sieving step for the 116-digit factorization was done entirely on the Internet using the software from [32]. For the 120-digit number it was carried out on 5 different Local Area Networks and on the 16384 processor MasPar MP-1 massively parallel computer at Bellcore, using in total four different implementations of the sieving step. Sieving for the 129-digit number was mostly done on the Internet using an updated version of the software from [32], 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 [17] and [4] 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.

Table 1.

	116-digit	120-digit	129-digit
size factor base	120000	245810	524339
large prime bound	10^8	2^{30}	2^{30}
fulls	25361	48665	112011
partials	284750	884323	1431337
double partials	953242	4172512	6881138
cycles	117420	203557	457455
amount of data	0.25 GB	1.1 GB	2 GB
timing sieving step	400 MY	825 MY	5000 MY
timing matrix step	0.5 hrs	4 hrs	45 hrs

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 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 (see [12,41], and Table 2 below).

From April 2, 1994, until April 10, 1996, the QS-factorization of the 129-digit number, the 'RSA-challenge number' (cf. [21]), was the largest factorization published that was found using a general purpose factoring method:

$$\begin{aligned} \mathrm{RSA}-129 &= 1143\,81625\,75788\,88676\,69235\,77997\,61466\,12010\,21829\,67212 \\ &\quad 42362\,56256\,18429\,35706\,93524\,57338\,97830\,59712\,35639\,58705 \\ &\quad 05898\,90751\,47599\,29002\,68795\,43541 \\ &= 3490\,52951\,08476\,50949\,14784\,96199\,03898\,13341\,77646\,38493 \\ &\quad 38784\,39908\,20577 \\ &\cdot\ 32769\,13299\,32667\,09549\,96198\,81908\,34461\,41317\,76429\,67992 \\ &\quad 94253\,97982\,88533. \end{aligned}$$

6. Number Field Sieve

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* (SNFS) which can only be applied to numbers of a special

form (similar to the form required by Pollard's original method). In 1990 SNFS was used to factor the ninth Fermat number $2^{2^9}+1$ (cf. [31]):

$$\begin{aligned} 2^{2^9}+1 \;=\; & 24\,24833 \cdot \\ & 7455\,60282\,56478\,84208\,33739\,57362\,00454\,91878\,33663\,42657 \cdot p99, \end{aligned}$$

where $p99$ denotes a 99-digit prime. The 'special form' restrictions were later removed, which resulted in the *general number field sieve*. Currently one often simply uses NFS to refer to the general algorithm. On April 10, 1996, NFS was used to factor the following 130-digit number, thereby breaking the 129-digit record set by QS of the largest published factorization found using a general purpose factoring method.

$$\begin{aligned} \mathrm{RSA}-130 \;=\; & 18070\,82088\,68740\,48059\,51656\,16440\,59055\,66278\,10251\,67694 \\ & 01349\,17012\,70214\,50056\,66254\,02440\,48387\,34112\,75908\,12303 \\ & 37178\,18879\,66563\,18201\,32148\,80557 \\ = \; & 39685\,99945\,95974\,54290\,16112\,61628\,83786\,06757\,64491\,12810 \\ & 06483\,25551\,57243 \\ \cdot \; & 45534\,49864\,67359\,72188\,40368\,68972\,74408\,86435\,63012\,63205 \\ & 06960\,09990\,44599. \end{aligned}$$

More importantly, the NFS-factorization of RSA-130 required much less time than the QS-factorization of RSA-129. Details can be found below.

NFS is considerably more complicated than the methods sketched so far. In this section we explain what relations in NFS look like, why they can be found much faster than QS-relations, and how we distributed the relation collection over the World-Wide-Web. How the relations are combined to derive the factorization is beyond the scope of this survey; it can be found in [30], along with further background on NFS. For additional information, NFS implementations and factorizations, see [9, 13, 18, 19, 23].

SNFS has heuristic asymptotic expected run time $L_n[1/3, (32/9)^{1/3}+o(1)] \approx L_n[1/3, 1.526+o(1)]$, for $n \to \infty$. The general method, NFS, runs in heuristic asymptotic expected time $L_n[1/3, (64/9)^{1/3}+o(1)] \approx L_n[1/3, 1.923+o(1)]$, for $n \to \infty$.

To put the progress from 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 unpublished) polynomial time factoring algorithm would run in time $(\ln n)^c = L_n[0, c]$, 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.

Relations in the number field sieve. Let f_1 and f_2 be two distinct polynomials with integer coefficients. There is no need to restrict ourselves to only two polynomials (cf. [20]), 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 [30] 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 $-m/2 \le c_i \le m/2$ is a base m representation of n.

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. Note that the coefficients of f_2 are quite small.

For the factorization of $n = \text{RSA}-130$ we used $d = 5, m = 125\,74411\,16841\,80059\,80468$, $f_1(X) = X - m$, and

$$\begin{aligned} f_2(X) \;=\; & 5748\,30224\,87384\,05200X^5 + 9882\,26191\,74822\,86102X^4 \\ & - 13392\,49938\,91281\,76685X^3 + 16875\,25245\,88776\,84989X^2 \\ & + 3759\,90017\,48552\,08738X - 46769\,93055\,39319\,05995. \end{aligned}$$

We have that $f_1(m) = 0$ and $f_2(m) = n$, so that f_1 and f_2 have the root m in common modulo n. Note that the coefficients of f_1 and f_2 are of roughly the same order of magnitude. These polynomials for RSA-130 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 a bound depending on f_j) that may occur in the factorization of $N_j(a, b)$ for coprime a and b. Smoothness of $N_j(a, b)$ will always refer to smoothness with respect to the jth 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 $x^2 \equiv y^2 \bmod n$). Let α_j denote a root of f_j. The prime factorization of $N_j(a, b)$ corresponds, roughly speaking, to the 'prime ideal factorization' of $a - \alpha_j b$ in the algebraic number field $\mathbf{Q}(\alpha_j)$. Since f_1 and f_2 have a common root m modulo n, the algebraic numbers $a - \alpha_1 b$ and $a - \alpha_2 b$ are 'the same' when taken mod n: let φ_j denote the homomorphism from $\mathbf{Z}[\alpha_j]$ to $\mathbf{Z}/n\mathbf{Z}$ that maps α_j to m modulo n, then $\varphi_1(a - \alpha_1 b) \equiv \varphi_2(a - \alpha_2 b) \bmod n$.

Assume that the number of relations we have is more than the sum of the sizes of the two factor bases. This implies that we can determine, by means of the usual matrix step, independent subsets S of the set of relations such that $\prod_{(a,b)\in S} N_j(a, b)$ is a square (in $\mathbf{Z}$), both for $j = 1$ and for $j = 2$. For the j with degree$(f_j) > 1$ this does not imply that the corresponding $\gamma_j(S) = \prod_{(a,b)\in S}(a - \alpha_j b)$ is a square in $\mathbf{Z}[\alpha_j]$ (for the j with degree$(f_j) = 1$ it does). But if we include in the matrix some additional information (so-called *quadratic signatures*) for each $N_j(a, b)$ with degree$(f_j) > 1$, then we may safely assume that all $\gamma_j(S)$ are squares in $\mathbf{Z}[\alpha_j]$ (cf. [1]). Note that $\varphi_1(\gamma_1(S)) \equiv \varphi_2(\gamma_2(S)) \bmod n$.

Because the factorization of the norms of the $\gamma_j(S)$'s is known (from the factorizations of the $N_j(a, b)$ with $(a, b) \in S$), the squareroot $\beta_j(S)$ of $\gamma_j(S)$ in $\mathbf{Z}[\alpha_j]$ can be computed:

trivially if degree$(f_j) = 1$, using the method described in [40] otherwise. The resulting squareroots satisfy $(\varphi_1(\beta_1(S))^2 \equiv (\varphi_2(\beta_1(S))^2 \bmod n$, which is the desired congruence of the form $x^2 \equiv y^2 \bmod n$. Note that each S leads to an independent chance to factor n.

If, for the j with degree$(f_j) > 1$, generators for the prime ideals (and units) in $\mathbf{Z}[\alpha_j]$ can be found, the squareroot can be computed faster by applying φ_j to each of those generators (if degree$(f_j) = 1$ the squareroot computation is trivial, as mentioned above). In general (in the general NFS) such generators cannot be found if degree$(f_j) > 1$, but in SNFS it might be possible because the f_j's of degree > 1 have small coefficients (it was used, for instance, for the ninth Fermat number).

Thus, after the sieving step, NFS requires a matrix step to determine several subsets S, followed by a squareroot step for each S until a lucky one that factors n is encountered. The 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 [18] 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.

Why NFS is faster than QS. A heuristic analysis of the asymptotic expected run time of NFS goes along the same lines as the analyses of the run times of Dixon's algorithm and QS. Instead of giving this analysis, we give the following informal explanation why 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.

Consider the 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' depend 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.

If the c_i's are small, as in SNFS, $N_2(a, b)$ is even more likely to be smooth, which explains why SNFS is so much faster than the general NFS.

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 [31]

and is called *classical* or *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 [45] the following is more efficient. Fix some reasonable 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. Because of the restriction on the pairs a, b, fewer pairs have to be considered per q, namely only those pairs that belong to a sublattice L_q of determinant q of the (a, b)-plane. For this reason Pollard called this way of sieving *lattice sieving*.

For general q, lattice sieving makes it possible and necessary to use *sieving by vectors*, another term introduced by Pollard. 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), sieving by vectors makes use, for each p, of two 2-dimensional vectors that form a reduced basis for the appropriate sublattice of determinant p of L_q. 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 $j = 2$).

Sieving by vectors is possible because a substantial part of L_q can be made to fit in memory. 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 in each L_q is too slow.[6] The details of sieving by vectors are rather messy (though not as bad as some of the rest of NFS) and can be found in [23]; see also [5].

Different q's may lead to duplicate a, b pairs, in particular when large primes are used. This implies that duplicates have to be removed from the resulting relations, even in an implementation where it can be guaranteed that each relevant q is processed only once.

Distributed factoring using NFS. 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 exhaust 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.

The first distributed NFS implementation (cf. [31]) was based on the approach of [32] and on classical sieving. It assigns disjoint ranges of b's to different processors. A single b can be processed in a matter of minutes on a workstation, so each processor needs a range of at least a few thousand b's to stay busy for a week. Larger b's are less productive than smaller ones, with b's on the order of a few million becoming almost worthless. This implies that only a fairly limited number of ranges can be distributed, and that a range should be redistributed when its results are not received within a reasonable amount of time. This leads to even more duplicated results than we have to deal with in QS, but duplicates can again easily be removed by keeping the relations sorted.

A more recent distributed NFS implementation (cf. [13]) is based on use of the World Wide Web and on lattice sieving. Because processing a single q takes at most a few minutes, disjoint ranges of q's are assigned to different processors, just as the b's were distributed in classical sieving. The size of the range assigned to each contributor depends on the resources available to that contributor: the types of processors and the amount of available memory and computing time per processor. An advantage compared to classical sieving is that the pool of 'good' q's is relatively large (cf. [5, 13]), so that lattice sieving tasks can be distributed quite liberally. Nevertheless, some q's are 'better' than others. It is therefore still a good idea to keep track of the dates the q's have been distributed, and to redistribute q's whose results are not received within a reasonable amount of time. Note that there are now three reasons why duplicates may be found: because they are intrinsic to lattice sieving with large primes, because any q might be processed more than once, and because relations from any q may be received or submitted more than once.

In [13] we describe the convenient Web-interface that takes care of most of the interactions with the contributors. Compared to the approach from [32] this interface makes it much easier to contribute to future distributed factoring efforts: a few mouse clicks is all that is needed. It should therefore not be difficult to perform the sieving step for numbers that are considerable larger than the one reported in [13]. Once the sieving step is complete, a non-trivial amount of computing has to be carried out at a location where enough computing power is available. With the current state of technology, this may take considerably more (real) time than the sieving step.

Some illustrative NFS data. In Table 2 we present some data for the general NFS-factorizations of a 116-digit and a 119-digit number (both from [18]), and of a 130-digit number (from [13]). For all three numbers we used two polynomials, with degree$(f_1) = 1$ and degree$(f_2) = 5$. The 116-digit number was the first number sieved using the implementation described in [23], with very conservative (and suboptimal) choices for the factor base sizes. The same implementation was later used for the sieving of the 119-digit number, with a much better choice for the factor base sizes. For the 130-digit number, the implementation of [23] was extended to allow more liberal use of the large primes that define the lattices (the q's), as described in [13].

The 'partials' refer to the relations with one or more large primes: in the implementations used relations can in principle have almost any number of large primes, though the majority has at most 5.

For the matrix step a variety of different algorithms and implementations was used, as shown in the table; 'Gauss' refers to *structured Gaussian elimination* (cf. [28, 48]), and 'Lanczos' refers to P. L. Montgomery's *blocked Lanczos method* (cf. [12, 41]). Note that for the two applications of 'Lanczos' the matrix is much larger than simply the sum of the factor base sizes. This is due to the use of large primes and the fact that they are only partially removed from the matrix during the cycle construction in an attempt to minimize the run time for Lanczos; for details see [18, 13]. For the 116-digit number all large primes were removed. All squareroot computations were carried out at the Centrum voor Wiskunde en Informatica (CWI) in Amsterdam, using P. L. Montgomery's implementation of his own method (cf. [40]). The squareroot timings in the table give the time per dependency.

Table 2.

	116-digit	119-digit	130-digit
size first factor base	100001	100001	250001
size second factor base	400001	360001	750001
large prime bound	2^{30}	2^{30}	10^9
fulls	61849	38741	48400
partials	45876382	35763524	56467272
cycles	2605463	472426	2844859
amount of data	3 GB	2.2 GB	3.5 GB
timing sieving step	220 MY	250 MY	550 MY
matrix size	$\approx 500100^2$	$\approx 1475000^2$	$\approx 3505000^2$
matrix algorithm	Gauss	Lanczos	Lanczos
running on	MasPar MP-1	MasPar MP-1	CRAY C-90
at	Bellcore	Bellcore	CWI
timing matrix step	114 hrs	60 hrs	67.5 hrs
timing squareroot step	60 hrs	20 hrs	49.5 hrs

Recent results. In 1998–1999 P. L. Montgomery and B. Murphy developed a new method to select the polynomials f_1 and f_2. Using their method the 140-digit number RSA-140 was factored on February 2, 1999, and sieving for the 155-digit (and 512-bit) number RSA-155 was completed on July 14, 1999. At the time of writing the matrix step was still in progress. Also, on April 8, 1999, a new SNFS record was set with the factorization of the 211-digit number $(10^{211} - 1)/9$. For details on these factorizations consult the web pages at www.cwi.nl.

Acknowledgments

This paper was written while the author was employed by Bellcore. The Isaac Newton Institute for Mathematical Sciences is gratefully acknowledged for its hospitality. Acknowledgments are due to Bruce Dodson, Matt Fante, Stuart Haber, Paul Leyland, and Sue Lowe for their help with this paper.

Appendix

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. [51]) works as follows. Each party generates two sufficiently large primes p and q, selects integers e and d such that $e \cdot d \equiv 1 \bmod (p-1)(q-1)$, 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 in Section 2); the density of primes is sufficiently high ($\pi(x) \approx x/\ln x$, cf. Section 2); 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.

Let the message m be a bit string shorter than n. To encrypt m using the public key (n, e) one computes $E(m) = m^e \bmod n$, which is equal to m because of Fermat's little theorem and the Chinese remainder theorem (cf. [25]). The modular exponentiations can be done efficiently using the repeated square and multiply method, as shown in Section 2. Since d can be found given e and the factors of n, factoring n suffices to break RSA. Conversely, it is believed that in general, without knowledge of d, factoring n is necessary to be able to decrypt RSA-encrypted messages.

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 $S(m) = m^d \bmod n$ 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 $S(m)^e \bmod n$ equals the original message m.

Notes

1. The 116-digit factorization of a BlackNet PGP key described in [22] used the same software as [4] but was distributed on a much smaller scale than the other efforts.
2. This can be done using Euclid's algorithm, as explained before. Note that if $\gcd(a, n) \neq 1$ we have found a factor > 1 of n (since $1 < a < n$), so that n is composite.
3. 'Some people have all the luck' (cf. [47]).
4. A similar idea can also be found in [26].
5. In CFRAC v's such that s_v is small are generated using continued fractions. If a_i/b_i is the ith continued fraction convergent to $\sqrt{n}$, then $r(a_i) = a_i^2 - nb_i^2$ satisfies $|r(a_i)| < 2\sqrt{n}$. Thus, with $v = a_i$ we have that $|s_v| = |r(a_i)|$ is bounded by $2\sqrt{n}$. Even though this is smaller than the $|s_{v(i)}|$'s that are generated in QS, CFRAC is less efficient than QS because the smoothness of the $|s_v|$'s in CFRAC cannot be detected using a sieve, but has to be checked 'individually' per $|s_v|$ using trial division or elliptic curves.
6. Pollard refers to line-by-line sieving in L_q as *sieving by rows*. For a small minority of q's only a few b's have to be considered, in which case line-by-line (or row) sieving *is* the preferred strategy.

References

1. L. M. Adleman, Factoring numbers using singular integers, *Proc. 23rd Annual ACM Symp. on Theory of Computing (STOC)*, New Orleans, (May 6–8, 1991) pp. 64–71.
2. W. R. Alford, A. Granville, and C. Pomerance, There are infinitely many Carmichael numbers, *Ann. of Math*, Vol. 140 (1994) pp. 703–722.
3. W. R. Alford, A. Granville, and C. Pomerance, On the difficulty of finding reliable witnesses, ANTS'94, Lecture Notes in Comput. Sci., 877 (1994) pp. 1–16.
4. D. Atkins, M. Graff, A. K. Lenstra, and P. C. Leyland, The magic words are squeamish ossifrage, *Advances in Cryptology*, Asiacrypt'94, Lecture Notes in Comput. Sci., 917 (1995) pp. 265–277.
5. D. J. Bernstein, The multiple-lattice number field sieve, Chapter 3 of Ph.D. thesis; ftp://koobera.math.uic.edu/pub/papers/mlnfs.dvi.
6. W. Bosma and A. K. Lenstra, An implementation of the elliptic curve integer factorization method, *Computational Algebra and Number Theory* (W. Bosma and A. van der Poorten, eds.), Kluwer Academic Publishers, Dordrecht, Boston, London (1995) pp. 119–136.
7. R. P. Brent, Factorization of the tenth and eleventh Fermat Numbers, manuscript (1996).

8. R. P. Brent and J. M. Pollard, Factorization of the eighth Fermat number, *Math. Comp.*, Vol. 36 (1981) pp. 627–630.
9. J. Buchmann, J. Loho, and J. Zayer, An implementation of the general number field sieve, Advances in Cryptology, Crypto '93, Lecture Notes in Comput. Sci, 773 (1994) pp. 159–165.
10. E. R. Canfield, P. Erdös, and C. Pomerance, On a problem of Oppenheim concerning "Factorisatio Numerorum," *J. Number Theory*, Vol. 17 (1983) pp. 1–28.
11. H. Cohen, A course in computational number theory, *Graduate Texts in Mathematics*, Vol. 138, Springer-Verlag, Berlin (1993).
12. S. Contini and A. K. Lenstra, Implementations of blocked Lanczos and Wiedemann algorithms, manuscript.
13. J. Cowie, B. Dodson, R. M. Elkenbracht–Huizing, A. K. Lenstra, P. L. Montgomery, and J. Zayer, A World Wide Number Field Sieve factoring record: on to 512 bits, Advances in Cryptography, Asiacrypt '96, Lecture Notes in Computer Science, 1163 (1996) pp. 382–394.
14. J. A. Davis and D. B. Holdridge, Factorization using the quadratic sieve algorithm, Tech. Report SAND 83-1346, Sandia National Laboratories, Albuquerque, NM (1983).
15. N. G. de Bruijn, On the number of positive integers $\leq x$ and free of prime factors $> y$, II, *Indag. Math.*, Vol. 38 (1966) pp. 239–247.
16. M. Deleglise and J. Rivat, Computing $\pi(x)$: the Meissel, Lehmer, Lagarias, Miller, Odlyzko method, *Math. Comp.*, Vol. 65 (1996) pp. 235–245.
17. T. Denny, B. Dodson, A. K. Lenstra, and M. S. Manasse, On the factorization of RSA-120, Advances in Cryptology, Crypto '93, Lecture Notes in Comput. Sci., 773 (1994) pp. 166–174.
18. B. Dodson and A. K. Lenstra, NFS with four large primes: an explosive experiment, Advances in Cryptology, Crypto '95, Lecture Notes in Comput. Sci., 963 (1995) pp. 372–385.
19. R. M. Elkenbracht–Huizing, An implementation of the number field sieve, Technical Report NM-R9511, Centrum voor Wiskunde en Informatica, Amsterdam, 1995; to appear in *Experimental Mathematics*.
20. R. M. Elkenbracht–Huizing, A multiple polynomial general number field sieve, *Preproceedings ANTS II* (H. Cohen, ed.), Université de Bordeaux (1996) pp. 101–116.
21. M. Gardner, Mathematical games, A new kind of cipher that would take millions of years to break, *Scientific American* (August 1977) pp. 120–124.
22. J. Gillogly, A. K. Lenstra, P. C. Leyland, and A. Muffett, An unnoticed factoring attack on a PGP key, presented at Crypto '95 rump session.
23. R. Golliver, A. K. Lenstra, and K. McCurley, Lattice sieving and trial division, ANTS'94, Lecture Notes in Comput. Sci., 877 (1994) pp. 18–27.
24. G. H. Hardy and W. M. Wright, *An Introduction to the Theory of Numbers*, 5th ed., Oxford University Press, Oxford (1979).
25. D. E. Knuth, Art of computer programming, volume 2, *Seminumerical Algorithms*, 2nd ed., Addison-Wesley, Reading, Massachusetts (1981).
26. M. Kraïtchik, *Theorie de Nombres, II*, Gauthiers-Villars, Paris (1926) pp. 195–208.
27. J. C. Lagarias, V. S. Miller, and A. M. Odlyzko, Computing $\pi(x)$: The Meissel–Lehmer Method, *Math. Comp.*, Vol. 44 (1985) pp. 537–560.
28. B. A. LaMacchia and A. M. Odlyzko, Solving large sparse linear systems over finite fields, Advances in Cryptology, Crypto'90, Lecture Notes in Comput. Sci., 537 (1991) pp. 109–133.
29. A. K. Lenstra and H. W. Lenstra, Jr., Algorithms in number theory, Chapter 12 in *Handbook of Theoretical Computer Science, Volume A, Algorithms and Complexity* (J. van Leeuwen, ed.), Elsevier, Amsterdam (1990).
30. A. K. Lenstra and H. W. Lenstra, Jr., The development of the number field sieve, *Lecture Notes in Math.*, Springer-Verlag, Berlin, 1554 (1993).
31. A. K. Lenstra, H. W. Lenstra, Jr., M. S. Manasse, and J. M. Pollard, The factorization of the ninth Fermat number, *Math. Comp.*, Vol. 61 (1993) pp. 319–349.
32. A. K. Lenstra and M. S. Manasse, Factoring by electronic mail, Advances in Cryptology, Eurocrypt '89, Lecture Notes in Comput. Sci., 434 (1990) pp. 355–371.
33. A. K. Lenstra and M. S. Manasse, Factoring with two large primes, Advances in Cryptology, Eurocrypt '90, Lecture Notes in Comput. Sci., 473 (1990) pp. 72–82; *Math. Comp.*, Vol. 63 (1994) pp. 785–798.
34. H. W. Lenstra, Jr., Factoring integers with elliptic curves, *Ann. of Math*, Vol. 126 (1987) pp. 649–673.
35. H. W. Lenstra, Jr., and C. Pomerance, A rigorous time bound for factoring integers, *J. Amer. Math. Soc.*, Vol. 5 (1992) pp. 483–516.

36. H. W. Lenstra, Jr. and R. Tijdeman (eds.), Computational methods in number theory, *Math. Centre Tracts*, Vol. 154/155, Mathematisch Centrum, Amsterdam (1983).
37. P. C. Leyland, Multiple polynomial quadratic sieve, sans math, ftp://ftp.ox.ac.uk/pub/math/rsa129/mpqs_sans_math.Z (1994).
38. L. Monier, Evaluation and comparison of two efficient probabilistic primality testing algorithms, *Theor. Comp. Science*, Vol. 11 (1980) pp. 97–108.
39. P. L. Montgomery, Speeding the Pollard and elliptic curve methods of factorization, *Math. Comp.*, Vol. 48 (1987) pp. 243–264.
40. P. L. Montgomery, Square roots of products of algebraic numbers, *Proceedings of Symposia in Applied Mathematics* (Walter Gautschi, ed.), Mathematics of Computation 1943–1993, Vancouver (1993).
41. P. L. Montgomery, A block Lanczos algorithm for finding dependencies over GF(2), Advances in Cryptology, Eurocrypt'95, Lecture Notes in Comput. Sci., 921 (1995) pp. 106–120.
42. M. A. Morrison and J. Brillhart, A method of factoring and the factorization of F_7, *Math. Comp.*, Vol. 29 (1975) pp. 183–205.
43. J. M. Pollard, Theorems on factorization and primality testing, *Proc. Cambridge Philos. Soc.*, Vol. 76 (1974) pp. 521–528.
44. J. M. Pollard, A Monte Carlo method for factorization, *BIT*, Vol. 15 (1975) pp. 331–334.
45. J. M. Pollard, The lattice sieve, *Lecture Notes in Math.*, Springer-Verlag, Berlin, 1554 (1993) pp. 43–49.
46. C. Pomerance, Analysis and comparison of some integer factoring algorithms, Computational methods in number theory, *Math. Centre Tracts*, Vol. 154/155, Mathematisch Centrum, Amsterdam (1983) pp. 89–139.
47. C. Pomerance, Private communication (March 1996).
48. C. Pomerance and J. W. Smith, Reduction of huge, sparse matrices over finite fields via created catastrophes, *Experiment. Math.*, Vol. 1 (1992) pp. 89–94.
49. M. O. Rabin, Probabilistic algorithms for primality testing, *J. Number Theory*, Vol. 12 (1980) pp. 128–138.
50. H. Riesel, Prime numbers and computer methods for factorization. *Progr. Math.*, Vol. 57, Birkhäuser, Boston (1985).
51. R. L. Rivest, A. Shamir and L. Adleman, A method for obtaining digital signatures and public-key cryptosystems, *Comm. ACM*, Vol. 21 (1978) pp. 120–126.
52. R. Schoof, Quadratic fields and factorization, Computational methods in number theory, *Math. Centre Tracts*, Vol. 154/155, Mathematisch Centrum, Amsterdam (1983) pp. 235–286.
53. D. Shanks, Class number, a theory of factorization, and genera, *Proc. Symp. Pure Math.*, Vol. XX, AMS (1971) pp. 415–440.
54. R. D. Silverman, The multiple polynomial quadratic sieve, *Math. Comp.*, Vol. 84 (1987) pp. 327–339.