Some Parallel Algorithms for Integer Factorisation[*]

Richard P. Brent
Computing Laboratory
University of Oxford
rpb@comlab.ox.ac.uk

3 September 1999

[*]Invited paper presented at Europar'99, Toulouse.

Outline

- Introduction and motivation
- The elliptic curve method (ECM)
- The quadratic sieve (QS and MPQS)
- The number field sieve (NFS)
  - Special (SNFS)
  - General (GNFS)
- History and extrapolations
- Summary and conclusions

Introduction

Any positive integer N has a unique *prime power decomposition*

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

($p_1 < p_2 < \cdots < p_k$ primes, $\alpha_j > 0$). The standard proof gives no hint of an efficient algorithm for computing the prime power decomposition. In order to compute it, we need –

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

Public Key Cryptography

Fortunately or unfortunately, depending on one's point of view, problem 2 is generally believed to be hard. There is no known polynomial-time algorithm for finding a factor of a given composite integer N.

This empirical fact is of great interest because the most popular algorithm for public-key cryptography, the RSA algorithm, would be insecure if a fast integer factorisation algorithm could be implemented.

Today I will survey some of the most successful integer factorisation algorithms, concentrating on the computational aspects, and particularly on parallel/distributed implementations. Due to shortage of time, many topics covered in the Proceedings paper[1] will not be mentioned[2].

[1]*LNCS* Vol. 1685 (1999), 1–22.

[2]e.g. Pollard's $p-1$ and ρ methods, the second phase of ECM, large-prime variants of MPQS and NFS, the discrete log problem, . . .

Integer Factorisation Algorithms

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

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

- Lehman's algorithm, which has worst-case run time $O(N^{1/3})$.
- The Multiple Polynomial Quadratic Sieve (MPQS) algorithm, which under plausible assumptions has expected run time
$$O(\exp(\sqrt{c \ln N \ln\ln N}))\ ,$$
where $c \approx 1$ is a constant.
- The Number Field Sieve (NFS) algorithm, which under plausible assumptions has expected run time
$$O(\exp(c(\ln N)^{1/3}(\ln\ln N)^{2/3}))\ ,$$
where c is a (different) constant.

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

- The trial division algorithm, which has run time $O(f \cdot (\ln N)^2)$.
- Pollard's "rho" algorithm, which under plausible assumptions has expected run time
$$O(f^{1/2} \cdot (\ln N)^2)\ .$$
- Lenstra's Elliptic Curve (ECM) algorithm, which under plausible assumptions has expected run time
$$O(\exp(\sqrt{c \ln f \ln\ln f}) \cdot (\ln N)^2)\ ,$$
where $c \approx 2$ is a constant.

In these examples, the time bounds are for a sequential machine, and the term $(\ln N)^2$ is a generous allowance for the cost of performing arithmetic operations on numbers which are $O(N^2)$.

Quantum factorisation algorithms

In 1994 Shor showed that it is possible to factor in polynomial expected time on a quantum computer. However, despite the best efforts of several research groups, such a computer has not yet been built, and it remains unclear whether it will ever be feasible to build one. Thus, we restrict our attention to algorithms which run on classical (serial or parallel) computers.

Elliptic Curves Over Finite Fields

A curve of the form
$$y^2 = x^3 + ax + b \qquad (1)$$
over some field F is known as an *elliptic curve.* A more general cubic in x and y can be reduced to the form (1), which is known as the Weierstrass normal form, by rational transformations, provided $\mathrm{char}(F) \neq 2$ or 3.

There is a well-known way of defining an Abelian group $(G, +)$ on an elliptic curve over a field. If $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ are points on the curve, then the point $P_3 = (x_3, y_3) = P_1 + P_2$ is defined by –
$$(x_3, y_3) = (\lambda^2 - x_1 - x_2,\ \lambda(x_1 - x_3) - y_1)\ ,$$
where
$$\lambda = \begin{cases} (3x_1^2 + a)/(2y_1) & \text{if } P_1 = P_2 \\ (y_1 - y_2)/(x_1 - x_2) & \text{otherwise.} \end{cases}$$

The zero element in G is the "point at infinity", (∞, ∞). We write it as 0.

Geometric Interpretation

The geometric interpretation of "+" is straightforward: the straight line $P_1 P_2$ intersects the elliptic curve at a third point $P_3' = (x_3, -y_3)$, and P_3 is the reflection of P_3' in the x-axis.

More elegantly, if a straight line intersects the elliptic curve at three points Q_1, Q_2, Q_3 then

$$Q_1 + Q_2 + Q_3 = 0 .$$

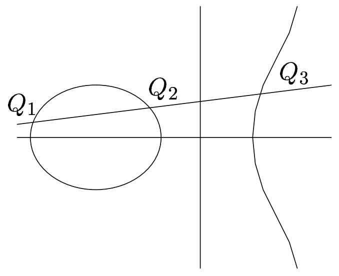


Figure 1: The Group Operation

Brief Description of ECM

The *elliptic curve method* (ECM) for integer factorisation was discovered by H. W. Lenstra, Jr. in 1985. Various practical refinements were suggested by Montgomery, Suyama, and others.

ECM uses groups defined by pseudo-random elliptic curves over GF(p), where $p > 3$ is the prime factor we hope to find. (Fortunately, we don't need to know p in advance.) The group order g for an elliptic curve over GF(p) satisfies

$$|g - p - 1| < 2\sqrt{p} ,$$

and all g satisfying this inequality are possible.

ECM is similar to an earlier method, Pollard's "$p - 1$" method, but the $p - 1$ method has the disadvantage that the group is fixed and the method fails if $p - 1$ has a large prime factor. We can think of ECM as a "randomised" version of the $p - 1$ method. It works if we are lucky enough to hit a group whose order g has no large prime factors. (Jargon – g is "smooth".)

Lenstra's Analysis of ECM

Consider applying ECM to a composite integer N with smallest prime factor p. Making an unproved but plausible assumption regarding the distribution of prime factors of random integers in "short" intervals, Lenstra showed that ECM will find p in an expected number

$$W(p) = \exp\left(\sqrt{(2 + o(1)) \ln p \ln \ln p}\right)$$

of multiplications (mod N), where the "$o(1)$" term tends to zero as $p \to \infty$.

In Lenstra's algorithm the field F is the finite field GF(p) of p elements, where p is a prime factor of N. Since p is not known in advance, computation is performed in the ring Z/NZ of integers modulo N rather than in GF(p). We can regard this as using a redundant group representation.

One Trial of ECM

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

1. Choose a parameter m.
2. Choose x_0, y_0 and a randomly in $[0, N)$. This defines $b = y_0^2 - (x_0^3 + ax_0) \bmod N$. Set $P \leftarrow P_0 = (x_0, y_0)$.
3. For each prime $\le m$ take its maximal power $q \le m$ and set $P \leftarrow qP$ in the group G defined by a and b.

If $P = 0$ then the trial succeeds as a factor of N will have been found during an attempt to compute an inverse mod N. (We expect $P = 0$ if no prime power factors of the group order are larger than m.) Otherwise the trial fails.

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

Optimal Choice of m

Making Lenstra's plausible assumption, one may show that the optimal choice of m is $m = p^{1/\alpha}$, where

$$\alpha \sim \sqrt{\frac{2 \ln p}{\ln \ln p}} \, .$$

It follows that the expected run time is

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

The exponent $2/\alpha$ should be compared with 1 (for trial division) or 1/2 (for Pollard's "rho" method).

A Practical Problem

The optimal choice of m depends on the size of the factor p. Since p is unknown, we have to guess or use some sort of adaptive strategy.

Fortunately, the expected performance of ECM is not very sensitive to the choice of parameters, so the precise strategy does not matter much.

Expected Performance of ECM

In Table 1 we give a small table of $\log_{10} W$ for factors of D decimal digits. The precise figures depend on assumptions about the implementation.

Table 1: Expected work for ECM

digits D	$\log_{10} W$
20	7.35
30	9.57
40	11.49
50	13.22
60	14.80

Note that in the region $D = 50$ to $D = 60$ the expected work W increases by a factor of about 2^6, and by Moore's law we might predict that hardware will improve by this factor in about 9 years, i.e. Moore's law gives about one digit per year.

ECM Example 1

After the factorisation of the ninth Fermat number $F_9 = 2^{2^9} + 1$ in 1990 (we'll say more about this later), $F_{10} = 2^{2^{10}} + 1$ was the "most wanted" number in various lists of composite numbers.

F_{10} was proved composite in 1952 by Robinson, using Pépin's test on the SWAC. A small factor, 45592577, was found by Selfridge in 1953 (also on the SWAC). Another small factor, 6487031809, was found by Brillhart in 1962 on an IBM 704. Brillhart later found that the cofactor was a 291-digit composite.

Using ECM I found a 40-digit factor $p_{40} =$

$$4659775785220018543264560743076778192897$$

of F_{10} in October, 1995. The 252-digit cofactor c_{291}/p_{40} passed a probabilistic primality test and was soon proved to be prime using the method of Atkin and Morain (based, appropriately, on elliptic curves). Thus, the complete factorisation of F_{10} is

$$F_{10} = 45592577 \cdot 6487031809 \cdot p_{40} \cdot p_{252} \, .$$

ECM Example 2

ECM can routinely find factors p of size up to 30 decimal digits, and it often finds larger factors. The largest factor known to have been found by ECM is the 53-digit factor

$$\begin{aligned} p_{53} &= 53625112691923843508117942\backslash \\ & 311516428173021903300344567 \end{aligned}$$

of $2^{677} - 1$, found by Conrad Curry in September 1998 using a program written by George Woltman and running on 16 Pentiums.

The group order for the lucky trial was

$$\begin{aligned} g &= 2^4 \cdot 3^9 \cdot 3079 \cdot 152077 \cdot 172259 \cdot 1067063 \cdot \\ & 3682177 \cdot 3815423 \cdot 8867563 \cdot 15880351 \end{aligned}$$

We expect only one in $2,400,000$ curves to have such a "smooth" group order.

ECM Factoring Records

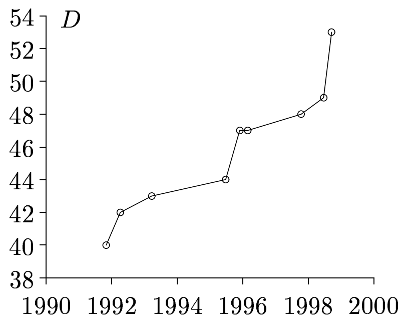


Figure 1: Factors found by ECM versus year

Figure 1 shows the size D (in decimal digits) of the largest factor found by ECM against the year it was done, from 1991 (40D) to 1999 (53D).

Curve Fitting

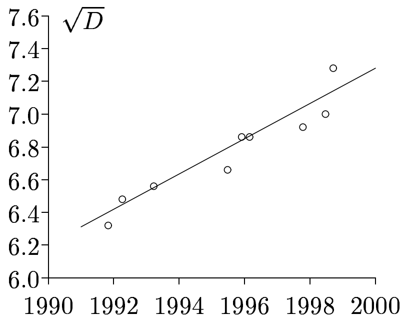


Figure 2: $\sqrt{D}$ versus year Y for ECM

Let D be the number of decimal digits in the largest factor found by ECM up to a given date. From the theoretical time bound for ECM, assuming Moore's law, we expect $\sqrt{D}$ to be roughly a linear function of calendar year (in fact $\sqrt{D \ln D}$ should be linear, but given the other uncertainties we have assumed for simplicity that $\sqrt{\ln D}$ is roughly a constant).

Extrapolation of ECM Records

The straight line shown in the Figure 2 is

$$\sqrt{D} = \frac{Y - 1932.3}{9.3}$$

and extrapolation gives $D = 60$ in the year $Y = 2004$ and $D = 70$ in the year $Y = 2010$.

Quadratic Sieve Algorithms

Quadratic sieve algorithms belong to a large class of algorithms which try to find two integers x and y such that $x \neq \pm y \pmod{N}$ but

$$x^2 = y^2 \pmod{N} . \qquad (2)$$

Once such x and y are found, then GCD $(x - y, N)$ is a nontrivial factor of N. One way to find x and y satisfying (2) is to find a set of *relations* of the form

$$u_i^2 = v_i^2 w_i \pmod{N}, \qquad (3)$$

where the w_i have all their prime factors in a moderately small set of primes (called the *factor base*). Each relation (3) gives a row in a matrix A whose columns correspond to the primes in the factor base.

Linear Algebra mod 2

Once enough rows have been generated, we can use sparse Gaussian elimination in GF(2) to find a linear dependency (mod 2) between a set of rows of A. Multiplying the corresponding relations now gives an expression of the form (2). With probability at least 1/2, we have $x \neq \pm y \bmod N$ so a nontrivial factor of N will be found. If not, we need to obtain a different linear dependency and try again.

Exact and Approximate Systems

In the MPQS and NFS factorisation algorithms we have to solve very large, sparse linear systems *exactly* over the finite field GF(2). (More precisely, we have to find dependencies amongst the rows of a large matrix.) For discrete log problems we also get large sparse linear systems, although the field is different.

More familiar to most people is the *approximate* solution of large sparse linear systems over the real or complex fields (using floating-point arithmetic).

Use of Iterative Methods

To avoid problems with "fill in", variants of some familiar "iterative" methods can be used. These methods (based on conjugate gradients or Lanczos) only require matrix-vector multiplications and inner products. Some significant differences are:

- Nonzero vectors can be orthogonal to themselves !
- Many more iterations are required to find the exact solution (actually several exact dependencies) than an approximate solution.
- Preconditioning is useless (although other forms of preprocessing may be useful).
- The matrix is never symmetric.
- Operations over GF(2) can be parallelised using logical operations on words of (typically) 32 or 64 bits.

Sieving

In quadratic sieve algorithms the numbers w_i are the values of one (or more) polynomials with integer coefficients. This makes it easy to find relations by *sieving.* The inner loop of the sieving process has the form

```
while $j < bound$ do
    begin
    $s[j] \leftarrow s[j] + c$;
    $j \leftarrow j + q$;
    end
```

Here *bound* depends on the size of the (single-precision real) sieve array s, q is a small prime or prime power, and c is a single-precision real constant depending on q ($c = \Lambda(q) = \ln p$ if $q = p^e$, p prime).

It is possible to use scaling to avoid floating point additions, which is desirable on a small processor without floating-point hardware.

MPQS

MPQS is a quadratic sieve method which uses several polynomials to improve the efficiency of sieving (an idea of Montgomery). MPQS can, under plausible assumptions, factor a number N in time

$$\Theta(\exp\sqrt{c \ln N \ln\ln N}) ,$$

where $c \sim 1$.

If $p \approx \sqrt{N}$ this is essentially the same bound as for ECM. Thus, MPQS has no theoretical advantage over ECM. (Theoretically, ECM is *almost always* faster than MPQS).

However, in practice MPQS is usually faster than ECM if N is the product of two primes which both exceed $N^{1/3}$. This is because the inner loop of MPQS involves only single-precision (sieving) operations, whereas the corresponding loop of ECM involves multiple-precision operations mod N.

MPQS Examples

MPQS has been used to obtain many impressive factorisations. Arjen Lenstra and Mark Manasse (with many assistants scattered around the world) have factored several numbers larger than 10^{100}. For example, the 116-decimal digit number $(3^{329}+1)/(\text{known small factors})$ was split into a product of 50-digit and 67-digit primes. The final factorisation is

$$\begin{aligned} 3^{329}+1 \quad = \quad & 2^2 \cdot 547 \cdot 16921 \cdot 256057 \cdot 36913801 \cdot \\ & 177140839 \cdot 1534179947851 \cdot \\ & 24677078822840014266652777\backslash \\ & 903676806291837269743524 1 \cdot p_{67} \end{aligned}$$

Such factorisations require many years of CPU time, but a real time of only a month or so because of the number of different processors which are working in parallel.

The Magic Words are $\cdots$

At the time of writing, the largest number factored by MPQS is the 129-digit "RSA Challenge" number RSA129. It was factored in 1994 by Atkins *et al.* RS&A had predicted in *Scientific American* that it would take millions of years to factor RSA129.

The factors of RSA129 allow decryption of a 'secret' message from RS&A. Using the decoding scheme $01 = A, 02 = B, \ldots, 26 = Z$, and 00 a space between words, the decoded message reads

THE MAGIC WORDS ARE SQUEAMISH OSSIFRAGE

It is certainly feasible to factor larger numbers by MPQS, but for numbers of more than about 110 decimal digits GNFS is faster. For example, to factor RSA129 by MPQS required 5000 Mips-years, but to factor the slightly larger number RSA130 by GNFS required only 1000 Mips-years.

The Special Number Field Sieve (SNFS)

Most of our numerical examples have involved numbers of the form

$$a^e \pm b\ , \tag{4}$$

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

The *special number field sieve* (SNFS) is a relatively new (*c.* 1990) algorithm which does take advantage of the special form (4). In concept it is similar to the quadratic sieve algorithm, but it works over an algebraic number field defined by a, e and b.

The details are rather technical and depend on concepts from algebraic number theory, so we simply give two examples to show the power of the algorithm.

SNFS Example 1

Consider the 155-decimal digit number

$$F_9 = N = 2^{2^9} + 1$$

as a candidate for factoring by SNFS. Note that $8N = m^5 + 8$, where $m = 2^{103}$. We may work in the number field $Q(\alpha)$, where α satisfies

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

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

$$m^5 + 8 = 0 \pmod N,$$

the mapping $\phi : \alpha \mapsto m \bmod N$ is a ring homomorphism from $Z[\alpha]$ to Z/NZ.
The idea is to search for pairs of small coprime integers u and v such that both the algebraic integer $u + \alpha v$ and the (rational) integer $u + mv$ can be factored. The factor base now includes prime ideals and units as well as rational primes.

Example 1 continued

Because

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

each such pair gives a relation analogous to (3). The prime ideal factorisation of $u + \alpha v$ can be obtained from the factorisation of the *norm* $u^5 - 8v^5$ of $u + \alpha v$. Thus, we have to factor simultaneously two integers $u + mv$ and $|u^5 - 8v^5|$. Note that, for moderate u and v, both these integers are much smaller than N, in fact they are $O(N^{1/d})$, where $d = 5$ is the degree of the algebraic number field.

Using these and related ideas, Lenstra *et al* factored F_9 in June 1990, obtaining

$$\begin{aligned} F_9 \;=\; & 2424833 \cdot \\ & 7455602825647884208337395\backslash \\ & 736200454918783366342657 \cdot p_{99}\ , \end{aligned}$$

where p_{99} is an 99-digit prime, and the 7-digit factor was already known (although SNFS was unable to take advantage of this).

Details

The collection of relations took less than two months on a network of several hundred workstations. A sparse system of about 200,000 relations was reduced to a dense matrix with about 72,000 rows. Using Gaussian elimination, dependencies (mod 2) between the rows were found in three hours on a Connection Machine. These dependencies implied equations of the form $x^2 = y^2 \bmod F_9$. The second such equation was nontrivial and gave the desired factorisation of F_9.

SNFS Example 2

The current SNFS record is the 211-digit number $10^{211} - 1$, factored early in 1999 by a collaboration called "The Cabal". In fact, $10^{211} - 1 = 3^2 \cdot p_{93} \cdot p_{118}$, where

$$\begin{aligned} p_{93} \;=\; & 6926245573243896206627 8\backslash \\ & 2322677336711138108482 5\backslash \\ & 8828173973437557050649 2\backslash \\ & 3919318495246367318668 79 \end{aligned}$$

and p_{118} may be found by division.[3]

[3]In the paper in the Proceedings, I forgot the factor 9 of $10^{211} - 1 = 99 \cdots 99$ (!)

Details

The factorisation of $N = 10^{211} - 1$ used two polynomials

$$f(x) = x - 10^{35}$$

and

$$g(x) = 10x^6 - 1$$

with common root $m = 10^{35} \bmod N$. After sieving and reduction a sparse matrix over GF(2) was obtained with about 4.8×10^6 rows and weight (number of nonzero entries) about 2.3×10^8, an average of about 49 nonzeros per row. Montgomery's block Lanczos program took 121 hours on a Cray C90 to find 64 dependencies. Finally, the square root program needed 15.5 hours on one CPU of an SGI Origin 2000, and three dependencies to find the two prime factors.

The General Number Field Sieve (GNFS)

The *general number field sieve* (GNFS or just NFS) is a logical extension of the special number field sieve (SNFS).

When using SNFS to factor an integer N, we require two polynomials $f(x)$ and $g(x)$ with a common root m mod N but no common root over the field of complex numbers.

If N has the special form $a^e \pm b$ then it is usually easy to write down suitable polynomials with small coefficients, as illustrated by the two examples given above.

If N has no special form, but is just some given composite number, we can also find $f(x)$ and $g(x)$, but they no longer have small coefficients.

The "Base m" Method

Suppose that $g(x)$ has degree $d > 1$ and $f(x)$ is linear. d is chosen empirically, but it is known from theoretical considerations that the optimum value is

$$d \sim \left(\frac{3\ln N}{\ln\ln N}\right)^{1/3}.$$

We choose $m = \lfloor N^{1/(d+1)} \rfloor$ and write

$$N = \sum_{j=0}^{d} a_j m^j$$

where the a_j are "base m digits". Then, defining

$$f(x) = x - m, \quad g(x) = \sum_{j=0}^{d} a_j x^j ,$$

it is clear that $f(x)$ and $g(x)$ have a common root m mod N. This method of polynomial selection is called the "base m" method.

Other Ingredients of GNFS

Having found two appropriate polynomials, we can proceed as in SNFS, but many difficulties arise because of the large coefficients of $g(x)$. The details are the subject of several theses.

Suffice it to say that the difficulties can be overcome and the method works!

Due to the constant factors involved, GNFS is slower than MPQS for numbers of less than about 110 decimal digits, but faster than MPQS for sufficiently large numbers, as anticipated from the theoretical run times.

Difficulties Overcome

Some of the difficulties which had to be overcome to turn GNFS into a practical algorithm are:

- Polynomial selection. The "base m" method is not very good. Peter Montgomery and Brian Murphy have shown how a very considerable improvement (by a factor of more than ten) can be obtained.

- Linear algebra. After sieving a very large, sparse linear system over GF(2) is obtained, and we want to find dependencies amongst the rows. It is not practical to do this by Gaussian elimination because the "fill in" is too large. Montgomery showed that the Lanczos method could be adapted for this purpose. (This is nontrivial because a nonzero vector x over GF(2) can be orthogonal to itself, i.e. $x^T x = 0$.) His program works with blocks of size 64.

Difficulties continued

- Square roots. The final stage of GNFS involves finding the square root of a (very large) product of algebraic numbers. Once again, Montgomery found a way to do this.
- An idea of Adleman, using quadratic characters, is essential to ensure that the desired square root exists with high probability.

Scalability of GNFS

At present, the main obstacle to a fully parallel and scalable implementation of GNFS is the linear algebra. Montgomery's block Lanczos program runs on a single processor and requires enough memory to store the sparse matrix. It is possible to distribute the block Lanczos solution over several processors of a parallel machine, but the communication to computation ratio is high. There is a tradeoff here – by increasing the time spent on sieving we can reduce the size and weight of the resulting matrix.

If special hardware is built for sieving, as recently proposed by Shamir, the linear algebra will become relatively more important. The argument is similar to Amdahl's law: no matter how fast sieving is done, we can not avoid the linear algebra.

RSA140

At the time of writing the paper for the Proceedings, the largest number factored by GNFS was the 140-digit RSA Challenge number RSA140. It was split into the product of two 70-digit primes in February, 1999, by a team coordinated from CWI, Amsterdam. The amount of computer time required to find the factors was about 2000 Mips-years.

The two polynomials used were

$$f(x) = x - 3443565780924253695177907$$

and

$$\begin{aligned} g(x) \quad = \quad & +439682082840x^5 \\ & +390315678538960x^4 \\ & -7387325293892994572x^3 \\ & -19027153243742988714824x^2 \\ & -63441025694464617913930613x \\ & +318553917071474350392223507494\ . \end{aligned}$$

Polynomial Selection

The polynomial $g(x)$ was chosen (by the method of Murphy and Montgomery) to have a good combination of two properties: being unusually small over the sieving region, and having unusually many roots modulo small primes and small prime powers. The polynomial used had a yield about eight times that of a randomly chosen polynomial (so the polynomial selection sped up the factorisation by a factor of eight).

The polynomial selection took 2000 CPU-hours on four 250 MHz SGI Origin 2000 processors. This is about 60 Mips-years, or 3% of the total factorisation time. It might have been better to spend a larger fraction of the time on polynomial selection – this is an interesting tradeoff.

Sieving

Sieving was done on about 125 SGI and Sun workstations running at 175 MHz on average, and on about 60 PCs running at 300 MHz on average. The total amount of CPU time spent on sieving was 8.9 CPU-years (about 1900 Mips-years).

The Linear Algebra

The resulting matrix had about 4.7×10^6 rows and weight about 1.5×10^8 (about 32 nonzeros per row). Using Montgomery's block Lanczos program, it took almost 100 CPU-hours and 810 MB of memory on a Cray C916 to find 64 dependencies among the rows of this matrix. Calendar time for this was five days.

RSA155

At the time of writing the paper for the Proceedings, an attempt to factor the 512-bit number RSA155 was underway, and I predicted it would be factored before the year 2000.

The factorisation of RSA155 was completed on 22 August 1999 ! It took about 8000 Mips-years (about four times as much as RSA140). The sparse matrix had about 6.7×10^6 rows and columns, and 4.2×10^8 nonzeros.

For more details, see `http://www.loria.fr/~zimmerma/records/RSA155` .

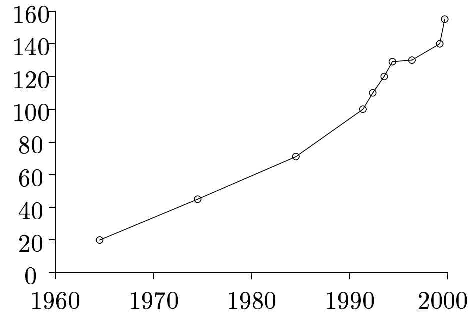
Summary – RSA140 and RSA155

In Table 2 we summarise the RSA140 and RSA155 factorisations.

Table 2: RSA140 and RSA155 factorisations

	RSA140	RSA155
Total mips-years	2000	8000
Improvement due to polynomial selection	8	14
Matrix rows	4.7×10^6	6.7×10^6
Total nonzeros	1.5×10^8	4.2×10^8
Nonzeros per row	32	62
Matrix solution time (on Cray C916)	100 hours	224 hours

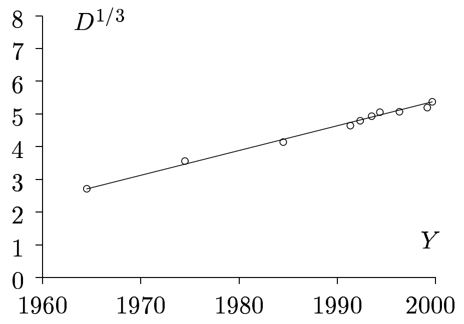
Historical Factoring Records



Size of "general" number factored versus year

The graph shows the size (in decimal digits) of the largest "general" number factored against the year it was done, from 1964 (20D) to 1999 (155D) (historical data from `www.rsa.com`).

Curve Fitting and Extrapolation



From the theoretical time bound for GNFS, assuming Moore's law, we expect $D^{1/3}$ to be roughly a linear function of time (we have assumed that $(\ln D)^{2/3}$ is roughly a constant). The graph shows $D^{1/3}$ versus year Y.

The straight line is

$$D^{1/3} = \frac{Y - 1928.6}{13.24}$$

and extrapolation, for what it is worth, gives $D = 309$ (i.e. 1024 bits) in the year $Y = 2018$.

Predictions

Moore's law predicts that circuit densities will double every 18 months or so. Thus, as long as Moore's law continues to apply and results in correspondingly more powerful parallel computers, we expect to get 3–4 decimal digits per year improvement in the capabilities of GNFS, without any algorithmic improvements. (The extrapolation from historical figures is more optimistic: it predicts 6–7 decimal digits per year in the near future.)

Similar arguments apply to ECM, for which we expect slightly more than 1 decimal digit per year in the size of factor found.

(When) Is RSA Doomed ?

512-bit RSA keys are clearly insecure. 1024-bit RSA keys should remain secure for at least fifteen years, barring the unexpected (but unpredictable) discovery of a completely new algorithm which is better than GNFS, or the development of a practical quantum computer.

Summary and Conclusions

I have sketched some algorithms for integer factorisation. The most important are ECM, MPQS and GNFS. The algorithms draw on results in elementary number theory, algebraic number theory and probability theory. As well as their inherent interest and applicability to other areas of mathematics, advances in public key cryptography have lent them practical importance.

Until a polynomial time algorithm is found or a quantum computer capable of running Shor's algorithm is built, large factorisations will remain an interesting challenge.

The best current algorithm (NFS) has an "embarrassingly parallel" phase (sieving) followed by a "communication intensive" phase (linear algebra), which makes implementation on a single parallel machine nontrivial.