Lecture 1

Fast Algorithms for Elementary Functions, π, and γ*

*Six lectures on Algorithms, Trinity term 1999.
 lec01

Summary

- High-precision arithmetic
- Use of Newton's method
- Some algorithms for exp and ln
- The arithmetic-geometric mean
- Faster algorithms for exp and ln
- Gauss-Legendre algorithm for π
- Some other algorithms for π
- Other elementary functions
- Euler's constant γ
- A connection with Ramanujan

High-precision arithmetic

We are concerned with high-precision computations on integers or floating-point approximations to real numbers.

If N is a (large) integer we can represent N using base (or radix) β, say

$$N = \pm \sum_{i=0}^{t-1} d_i \beta^i \ ,$$

where the d_i are "base β digits", i.e.

$$0 \le d_i < \beta \ .$$

The sign can be handled in several ways. We usually assume $N \ge 0$ for simplicity.

On a binary computer with a fixed word length, say w bits, it is convenient to choose the β so that base-β digits can be represented exactly in a single word. Thus, we choose

$$2 \le \beta \le 2^w \ .$$

It is often convenient to choose β significantly smaller than 2^w. For example, if $w = 32$, we might choose $\beta = 2^{31}$ or 2^{14} or 10^4.

Addition

The number of operations required to add two t-digit numbers is $O(t)$. If $\beta = 2$ this is called the "bit complexity" to distinguish it from the "operation complexity" (which is simply 1).

On a machine with fixed wordlength w bits the "bit complexity" and the "word complexity" only differ by a constant factor so we ignore the distinction (though in practice the constant is important).

Since we are considering a serial computer and ignoring constant factors, we usually call the bit complexity "time".

Multiplication

The time required to multiply two t-digit numbers $\sum a_i\beta^i$ and $\sum b_j\beta^j$ by the "schoolboy" method is $O(t^2)$. However, this is not optimal. Most of the computation involves forming convolutions

$$c_k = \sum_{i+j=k} a_i b_j$$

and we can do this by the *Fast Fourier Transform* (FFT) in $O(t \log t)$ operations. In fact, to ensure that the c_k are computed exactly, these "operations" are more than single word operations, and we get another $\log t$ factor. A more complicated algorithm, the *Schönhage-Strassen* algorithm, reduces this to $\log \log t$.

Thus, the conclusion is that the time required to multiply t-digit numbers is

$$M(t) = O(t \log t \log \log t) \ .$$

Optimality

Is the Schönhage-Strassen algorithm optimal ?

The answer depends on the model of computation. On certain models $O(t)$ is attainable. However, for realistic models Schönhage-Strassen is the best known algorithm.

In case the Schönhage-Strassen algorithm is not optimal on your model of computation (or you choose to implement a simpler algorithm), we express the time bounds in terms of the function $M(t)$ rather than explicitly using $M(t) = O(t \log t \log \log t)$.

Assumptions

We need to assume that $M(t)$ satisfies some reasonable smoothness properties. For example, assume that $\exists\, \alpha, \beta \in (0,1)$, for all sufficiently large t,

$$M(\alpha t) \leq \beta M(t) \ .$$

This assumption certainly holds if

$$M(t) \sim ct \log t \log \log t \ .$$

For convenience, we also assume that $M(t) = 0$ for $t < 1$.

Our assumptions ensure that

$$\sum_{k=0}^{\infty} M(\lceil t/2^k \rceil) = O(M(t))$$

which is useful in analysing the complexity of Newton's method.

MP Floating Point

We can represent multiple-precision "floating-point" numbers by pairs (e, N) of (multiple-precision) integers e (the exponent) and N (the scaled fraction).

The pair (e, N) is interpreted as $\beta^e N$ or perhaps as $\beta^{e-t} N$.

Clearly we can perform addition and multiplication on such floating-point numbers, obtaining a t-digit result, i.e. a result with relative error $O(\beta^{-t})$, by performing operations on the exponents and (shifted) fractions, in time $O(M(t))$.

Reciprocals by Newton's method

Newton's iteration for finding a simple zero of a function f is

$$x_{i+1} = x_i - f(x_i)/f'(x_i) \ .$$

Apply this to the function

$$f(x) = a - 1/x \ ,$$

where $a \neq 0$ is constant. Provided the initial approximation x_0 is not too bad, the sequence converges to the (unique) zero of f, i.e. $x = 1/a$. Also, the iteration simplifies to

$$x_{i+1} = x_i + x_i(1 - ax_i) = x_i(2 - ax_i)$$

which *only involves additions/subtractions and multiplications.*

Convergence is quadratic, as we can easily see by taking $\epsilon_i = 1 - ax_i$ and observing that

$$\epsilon_{i+1} = 1 - (1 - \epsilon_i)(1 + \epsilon_i) = \epsilon_i^2 \ .$$

Thus, the number of correct digits approximately *doubles* at each iteration.

Complexity of reciprocals

To obtain a t-digit reciprocal, we need to perform about $\lg(t)$ Newton iterations (here, as usual, lg denotes $\log_2$). If these are performed using t-digit arithmetic then the time required is $T_{rec}(t) = O(M(t) \log t)$.

However, we can improve the bound by observing that Newton's algorithm is *self-correcting*, so we can start with a small number of digits and (almost) double it at each step. It is most efficient to contrive that the last iteration is performed with (slightly more than) t digits.

For example, to get a 100-digit reciprocal, we might use $2, 2, 3, 5, 8, 14, 26, 51, 101$ digits at successive iterations.

By our smoothness assumptions, the total time is only $T_{rec}(t) = O(M(t))$. We have avoided the factor $\log(t)$.

It is also possible to show that $M(t) = O(T_{rec}(t))$, so in a sense multiplication and reciprocation are *equivalent.*

Division

To perform a t-digit "floating point" division, we can use

$$a/b = a \times (1/b)$$

where the reciprocal $1/b$ is computed by Newton's method. Thus, floating-point division also takes time $O(M(t))$.

For integer division, it is easy to obtain the correct quotient and remainder in time $O(M(t))$.

If b is small, then the integer division of a by b takes time $O(t)$.

Square roots

Newton's method can be applied to compute t-digit square root approximations in time $O(M(t))$, using the classical iteration

$$x_{i+1} = \frac{1}{2}\left(x_i + \frac{a}{x_i}\right) \ .$$

In fact, it is slightly more efficient to approximate the *inverse square root* $a^{-1/2}$ by Newton's method (avoiding divisions), then use

$$a^{1/2} = a \times a^{-1/2} \ .$$

Algorithms for $\exp(x)$ and $\ln(x)$

$\exp(x)$ and $\ln(x)$ are inverse functions, i.e.

$$\exp(\ln(x)) = x \ ,$$

at least for real positive x. Thus, if we can compute one then we can compute the other in the same time (up to a small constant factor) using Newton's method.

Whenever I describe an algorithm for $\exp(x)$, one for $\ln(x)$ is implied, and *vice versa*.

To avoid technicalities, we assume that the argument x is real and in some bounded, closed interval $[a, b]$. Also, in the case of $\ln(x)$, we assume $a > 0$.

$\exp(x)$ – Algorithm 1

The most obvious way to approximate $\exp(x)$ is to sum a sufficient number of terms in the power series

$$\exp(x) = \sum_{n=0}^{\infty} \frac{x^n}{n!} \ .$$

Using Stirling's approximation to $n!$, it is not hard to see that we need $\Theta(t/\log t)$ terms to obtain t-digit accuracy. Thus, the time required for $\exp(x)$ is

$$T_{exp}(t) = O\left(M(t)\frac{t}{\log t}\right) \ .$$

Using Newton's method, we deduce that the time required for $\ln(x)$ is

$$T_{ln}(t) = O\left(M(t)\frac{t}{\log t}\right) \ .$$

$\exp(x)$ – Algorithm 2

The power series

$$\exp(x) = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

converges faster for smaller $|x|$. Thus, an obvious idea is to apply the identity

$$\exp(x) = (\exp(x/2))^2$$

k times before summing the power series.

In other words, evaluate $\exp(x/2^k)$ using the power series, then compute

$$\exp(x) = \left(\exp\left(x/2^k\right)\right)^{2^k}$$

by squaring the result k times.

To get t-digit accuracy by this method we need $\Theta(t/k)$ terms in the power series, so the overall time is

$$T_{exp}(t) = O\left(M(t)\left(k + \frac{t}{k}\right)\right) \ .$$

Optimal choice of k

Choosing $k \sim \sqrt{t}$ gives

$$T_{exp}(t) = O(M(t)\sqrt{t})$$

which is better by a factor $\sqrt{t}/\log t$ than the bound obtained from Algorithm 1.

Although there are asymptotically faster algorithms (as we shall see), Algorithm 2 is the fastest algorithm in practice for a wide range of t.

Historical notes

Algorithm 2 was proposed and analysed in my 1976 paper "The complexity of multiple-precision arithmetic" [7].

For the history of other results, e.g. T_{rec}, see the bibliographic notes in Aho, Hopcroft and Ullman [1, Ch. 8].

The AGM

Can we do better than Algorithm 2 ?
Yes, but first we need to describe the *arithmetic-geometric mean* (AGM) iteration.

Let a_0, b_0 be given initial values. For simplicity we assume they are positive reals (though an extension to the complex plane is possible so long as care is taken with the branch of the square root below). The iteration is defined by

$$a_{i+1} = \frac{a_i + b_i}{2}$$

$$b_{i+1} = \sqrt{a_i b_i}$$

Gauss studied the AGM and showed that a_i and b_i converge to a common limit

$$\mathrm{AGM}(a_0, b_0)$$

which can be expressed in terms of a complete elliptic integral.

Complete elliptic integrals

We define the *complete elliptic integral of the first kind* by

$$K(\phi) = \int_0^{\pi/2} \frac{d\theta}{\sqrt{1 - \sin^2\phi \sin^2\theta}} \; .$$

Similarly, the *complete elliptic integral of the second kind* is

$$E(\phi) = \int_0^{\pi/2} \sqrt{1 - \sin^2\phi \sin^2\theta}\, d\theta \; .$$

Gauss showed that

$$\mathrm{AGM}(1, \cos\phi) = \frac{\pi}{2K(\phi)} \; . \tag{1}$$

For a proof, see Borwein and Borwein, "Pi and the AGM" [4, §1.2].

Also, if $c_0 = \sin\phi$ and $c_{i+1} = (a_i - b_i)/2$, then

$$1 - \frac{E(\phi)}{K(\phi)} = \sum_{i=0}^{\infty} 2^{i-1} c_i^2 \; . \tag{2}$$

Thus, both $K(\phi)$ and $E(\phi)$ can be computed via the AGM, given $\cos\phi$.

Remarks

There is no loss of generality in assuming that $a_0 \geq b_0$, since

$$\mathrm{AGM}(a_0, b_0) = \mathrm{AGM}(b_0, a_0) \; .$$

Also, there is no real loss of generality in assuming that $a_0 = 1$, since

$$\lambda\, \mathrm{AGM}(a_0, b_0) = \mathrm{AGM}(\lambda a_0, \lambda b_0) \; .$$

Thus, we can assume $a_0 = 1$ and $b_0 = \cos\phi$, as above.

Notation

Instead of the argument ϕ, the argument

$$k = \sin\phi$$

is often used. k is known as the *modulus* and

$$k' = \cos\phi$$

is known as the *complementary modulus*. We also define $\phi' = \pi/2 - \phi$ so $k' = \sin\phi'$.

Quadratic convergence of the AGM

The reason why the AGM is of computational interest is that it converges quadratically. In fact, if we define ϵ_i by

$$b_i/a_i = 1 - \epsilon_i \; ,$$

then it is easy to verify that

$$\epsilon_{i+1} = \epsilon_i^2/8 + O(\epsilon_i^3) \; .$$

Also, if $b_i/a_i \ll 1$ then

$$b_{i+1}/a_{i+1} = \frac{2\sqrt{b_i/a_i}}{1 + b_i/a_i} \approx 2\sqrt{b_i/a_i} \; ,$$

so after about $\lg(a_0/b_0)$ iterations we get $a_i/b_i = O(1)$. In other words, even if a_0/b_0 is large, it does not take long before quadratic convergence sets in.

Extreme cases

If ϕ is small, then

$$K(\phi) = \int_0^{\pi/2} \frac{d\theta}{\sqrt{1 - \sin^2\phi \sin^2\theta}} = \frac{\pi}{2} + O(\phi^2) \ .$$

Also, less obviously, if $k = \sin\phi$ is small and $\phi' = \pi/2 - \phi$, then

$$K(\phi') = \Big(1 + O(k^2)\Big) \ln\left(\frac{4}{k}\right) \ .$$

See Borwein and Borwein [4, ex. 1.3.4(b)]

$\log(x)$ – Algorithm 3

The last result implies (by exchanging primed and unprimed variables) that, if $a_0 = 1$ and $b_0 = \cos\phi = \epsilon^{1/2}$ is small, then

$$K(\phi) = (1 + O(\epsilon)) \ln\left(\frac{4}{b_0}\right) \ .$$

However, we can compute $K(\phi)$ by the AGM. In other words, if $y = 4/b_0$ is large, then

$$\ln y = \frac{\pi}{2\,\mathrm{AGM}(1, 4/y)} \left(1 + O\left(\frac{1}{y^2}\right)\right) \ .$$

This is fine for computing $\ln y$ provided

1. y is sufficiently large ($y \geq \beta^{t/2}$).
2. We know π to sufficient accuracy.

The first condition is easy to handle. We can take $M \approx \beta^{t/2}$ and compute

$$\ln y = \ln(My) - \ln(M) \ ,$$

using an additional $O(\log t)$ digits to compensate for cancellation.

Approximating π

Regarding the second condition: if we do not already know π to sufficient accuracy, we can compute

$$\frac{\ln(1 + \beta^{-t})}{\pi}$$

by the above method, and use

$$\ln(1 + \beta^{-t}) = \beta^{-t}(1 + O(\beta^{-t}))$$

to find a sufficiently accurate approximation to π. However, we have to work with about $2t$ digits to compensate for cancellation.

This shows that

$$T_\pi(t) = O(M(t) \log t) \ .$$

However, the constant factor implicit in the "O" is large. A better way of finding π, with smaller constant factor, will be discussed soon.

Complexity of ln

Using the results obtained so far, we have

$$T_{ln}(t) = O(M(t) \log t)$$

because the AGM converges quadratically and only $O(\log t)$ iterations are required.

Remark

The AGM is *not* self-correcting. Thus, we can not get rid of the "$\log t$" factor in these bounds by starting with low precision as we did for reciprocals and square roots by Newton's method.

Complexity of exp

Using Newton's method and Algorithm 3 for ln, we obtain

$$T_{exp}(t) = O(M(t) \log t) \ .$$

Historical notes

The $O(M(t)\log t)$ algorithms for log and exp were first published in my 1975 paper "Multiple-precision zero-finding methods and the complexity of elementary function evaluation" [5]. The algorithm for ln is implicit in Beeler, Gosper and Schroeppel's unpublished 1972 MIT Technical Report "HAKMEM" [2].

Different $O(M(t)\log t)$ algorithms for log and exp were published in my 1976 J. ACM paper "Fast multiple-precision evaluation of elementary functions" [6]. These algorithms use incomplete elliptic integrals and Landen transformations. The 1976 paper was actually written before the 1975 paper (J. ACM has a long publication delay).

Legendre's relation

Legendre found a beautiful relation between the four quantities $K(\phi), K(\phi'), E(\phi)$ and $E(\phi')$ where, as usual, $\phi + \phi' = \pi/2$:

$$E(\phi)K(\phi') + E(\phi')K(\phi) - K(\phi)K(\phi') = \frac{\pi}{2}\,.$$

For a proof, see Borwein and Borwein [4, §1.6].

All we need to obtain a fast algorithm for the computation of π is the special case $\phi = \phi' = \pi/4$. Then, abbreviating $K(\pi/4)$ by K and $E(\pi/4)$ by E, Legendre's relation reduces to

$$2EK - K^2 = \frac{\pi}{2}\,. \tag{3}$$

Fast evaluation of π

Suppose we perform the AGM iteration with initial values $a_0 = 1$ and $b_0 = \cos(\pi/4) = 1/\sqrt{2}$, obtaining a good approximation to the limit $a = \mathrm{AGM}(1, 1/\sqrt{2})$. Then, by Gauss's result (1) on the limit of the AGM,

$$2a = \frac{\pi}{K}\,.$$

Also, using the relation (2), we can find E/K from quantities occurring during the AGM computation. Hence, dividing each side of (3) by K^2, we obtain a known quantity

$$\frac{2E}{K} - 1$$

on the left, and the quantity

$$\frac{1}{2\pi}\left(\frac{\pi}{K}\right)^2$$

on the right, where everything is known except for the factor $\frac{1}{2\pi}$. Hence, we can find π !

The Gauss-Legendre algorithm for π

Simplifying the above, we obtain a nice algorithm for the computation of π. In my 1975–76 papers [5, 6] I called it the *Gauss-Legendre algorithm* because, as we have seen, it depends on results of Gauss and Legendre. The algorithm was discovered independently by Salamin (1976) [12].

$A \leftarrow 1$; $B \leftarrow 2^{-1/2}$;
$T \leftarrow 1/4$; $X \leftarrow 1$;
while $A - B > \beta^{-t}$ **do**
  begin
  $Y \leftarrow A$;
  $A \leftarrow (A + B)/2$;
  $B \leftarrow \sqrt{B \times Y}$;
  $T \leftarrow T - X \times (A - Y)^2$;
  $X \leftarrow 2 \times X$;
  end;
return A^2/T {or, better, $(A + B)^2/(4T)$}.

The rate of convergence is illustrated in the following table.

Convergence of the Gauss-Legendre Method

Iteration	$A^2/T - \pi$	$\pi - (A+B)^2/(4T)$
0	8.6'-1	2.3'-1
1	4.6'-2	1.0'-3
2	8.8'-5	7.4'-9
3	3.1'-10	1.8'-19
4	3.7'-21	5.5'-41
5	5.5'-43	2.4'-84
6	1.2'-86	2.3'-171
7	5.8'-174	1.1'-345
8	1.3'-348	1.1'-694
9	6.9'-698	6.1'-1393

From the table we see that

$$(A+B)^2/(4T) < \pi < A^2/T$$

(this can be proved rigorously). Also, convergence is quadratic, as expected. It takes 9 iterations to get 1000 decimals, 19 iterations to get 10^6 decimals, and only 29 iterations to get 10^9 decimals !.

Complexity of π evaluation

From the algorithms above,

$$T_\pi(t) = O(M(t)\log t)\ .$$

It is an open question whether this bound is the best possible.

Comparison with other algorithms

Archimedes suggested bounding π by the areas of regular n-gons inside/outside the unit circle (or half their lengths). For example, starting with regular hexagons, we can obtain a recurrence for doubling n which gives bounds for $n = 12, 24, 48, 96, \ldots$ (see Borwein & Borwein [4, Ch. 11]).

Each iteration involves a square root and a few multiplications/additions, so is comparable to an AGM iteration. However, convergence is only *linear*, not quadratic. To obtain 10^6 digits of π by such a method would take more than 10^6 iterations (compare 19 for the Gauss-Legendre method).

Improvements on Archimedes

Other methods involve the arctan formula

$$\arctan\theta = \theta - \theta^3/3 + \theta^5/5 - \cdots$$

combined with Machin's formula

$$\frac{\pi}{4} = 4\arctan\frac{1}{5} - \arctan\frac{1}{239}$$

or similar formulae. They are good for moderate precision but give only linear convergence, so Gauss-Legendre must win eventually.

There are many other methods, but they nearly all have linear (or worse) convergence. The only known methods competitive (for high precision computations) with the Gauss-Legendre method are similar methods based on the AGM – see Borwein and Borwein [4].

Other elementary functions

By considering the AGM for complex arguments, or by using incomplete elliptic integrals and Landen transformations, we can compute any elementary function sin, cos, tan, sinh etc or its inverse arctan etc, in a compact interval not containing any singularities of the function, in time

$$O(M(t)\log t)\ .$$

It is not known whether this result is best possible.

Euler's constant

Euler's constant $\gamma = -\Gamma'(1)$ is usually defined by

$$\gamma = \lim_{n\to\infty} \left(H_n - \ln(n)\right) \ ,$$

where

$$H_n = \sum_{k=1}^{n} \frac{1}{k} \ .$$

It is not known whether γ is rational or irrational. Hence, there is some interest in computing γ and its regular continued fraction (perhaps more interest than in computing π, since π is known to be transcendental).

The defining limit converges much too slowly to be useful, but it can be accelerated by approximating the truncation error by an Euler-Maclaurin expansion. The difficulty here is how to quickly generate the Bernoulli numbers required for the Euler-Maclaurin expansion.

Use of Bessel functions

The fastest known method for computing γ, due to Brent and McMillan [9], was derived using results on Bessel functions, but it can be regarded as a "smoothing" of the defining limit. Specifically, let

$$U(n) = \sum_{k=0}^{\infty} \left(\frac{n^k}{k!}\right)^2 H_k \ - \ \ln(n) V(n) \ ,$$

where

$$V(n) = \sum_{k=0}^{\infty} \left(\frac{n^k}{k!}\right)^2 \ .$$

Then

$$0 < \frac{U(n)}{V(n)} - \gamma < \pi e^{-4n} \ ,$$

so computation of $U(n)/V(n)$ gives γ to of order n digits. This can be done in time $O(n^2)$, or even faster if we use rational arithmetic and a binary tree for summation. Thus

$$T_\gamma(t) = O(M(t)(\log t)^2) \ .$$

For details see [9].

Recent computations

γ has recently been computed to 10^6 decimals by Thomas Papanikolaou. By computing 470,006 partial quotients of its continued fraction, Papanikolaou has shown that *if* γ is rational, say $\gamma = p/q$, then

$$q > 10^{242080} \ .$$

By computing more partial quotients, it should be possible to improve this to

$$q > 10^{499900} \ .$$

Papanikolaou's computation improved the 1980 result of Brent and McMillan, who computed γ to 30,100 decimals and showed that $q > 10^{15000}$, using 29,200 partial quotients.

A connection with Ramanujan

Ramanujan published several series for γ, e.g. generalisations of Glaisher's

$$\gamma = 1 - \sum_{k=1}^{\infty} \frac{\zeta(2k+1)}{(k+1)(2k+1)} \ ,$$

but these are not suitable for computation of γ.

In his first notebook [3, I,p.98] Ramanujan states that (in modern notation)

$$\sum_{k=1}^{\infty} \frac{(-1)^{k-1}}{nk} \left(\frac{x^k}{k!}\right)^n = \ln x + \gamma + o(1) \qquad (4)$$

as $x \to +\infty$. Here n is a fixed positive integer.

The case $n = 1$ is correct, in fact the error term is just an exponential integral

$$\int_x^{\infty} \frac{e^{-u}}{u}\, du = O\left(\frac{e^{-x}}{x}\right)$$

(this result is due to Euler, and has been used by Sweeney and others to compute γ).

Unfortunately, (4) is <u>false</u> if $n > 2$.

What might have been

The case $n = 1$ (with improved error term) is

$$\sum_{k=1}^{\infty} \frac{(-1)^{k-1} x^k}{k!k} = \ln x + \gamma + O\left(\frac{e^{-x}}{x}\right) ,$$

and the sum on the left side can be written as

$$e^{-x} \sum_{k=0}^{\infty} H_k \frac{x^k}{k!} .$$

This is easy to prove, and was known by Ramanujan (see Berndt [3, I, pp. 46–47]). Thus, Ramanujan knew that

$$\sum_{k=0}^{\infty} H_k \frac{x^k}{k!} \Big/ \sum_{k=0}^{\infty} \frac{x^k}{k!} = \ln x + \gamma + O\left(\frac{e^{-x}}{x}\right) .$$

He could have generalised and made the "better" conjecture

$$\sum_{k=0}^{\infty} H_k \left(\frac{x^k}{k!}\right)^n \Big/ \sum_{k=0}^{\infty} \left(\frac{x^k}{k!}\right)^n = \ln x + \gamma + o(1) \tag{5}$$

instead of his incorrect claim (4).

Comments on the better conjecture

The case $n = 2$ of (5) is what was used (with a sharper error bound) by Brent and McMillan. The function $(x^k/k!)^n$ acts as a smoothing kernel with a peak at $k \approx x - \frac{1}{2}$.

In fact, (5) is correct and the error term $o(1)$ can be improved to

$$O\left(\exp(-c_n x)\right) ,$$

where

$$c_n = \begin{cases} 1 & \text{if } n = 1, \\ 2n \sin^2(\pi/n) & \text{if } n \geq 2. \end{cases}$$

The case $n = 3$ is interesting because

$$\max_{n=1,2,\ldots} c_n = c_3 = 4.5 ,$$

but no one seems to have used $n > 2$ in a serious computation of γ.

Historical notes

Early computations of γ, up to Knuth (1962), used the Euler-Maclaurin formula.

Sweeney (1963) used essentially the (correct) case $n = 1$ of Ramanujan's (4), with the error term replaced by an asymptotic expansion. I used Sweeney's method and continued fractions in 1977 to show that $q > 10^{10000}$.

In 1980, Brent and McMillan[1] used the case $n = 2$ of the "better" conjecture (proved using results on the asymptotic behaviour of the modified Bessel functions $I_0(z)$ and $K_0(z)$).

In a 1994 paper [10] I noted the connection with Ramanujan[2].

[1] Edwin McMillan (1907–1991), a physicist, is better known for his invention of the synchrotron (independently of Veksler) and for the discovery of neptunium and plutonium (1941). He shared the 1951 Nobel prize in chemistry with Seaborg. See *Nature* 353 (1991), 602.

[2] Ramanujan's story is too well known to need a footnote.

References

[1] Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[2] M. Beeler, R. W. Gosper, and R. Schroeppel, *HAKMEM*, MIT AI Lab, 1972.

[3] B. C. Berndt, *Ramanujan's Notebooks*, Parts I–III, Springer-Verlag, New York, 1985–1991.

[4] Jonathan M. and Peter B. Borwein, *Pi and the AGM*, Wiley-Interscience, John Wiley and Sons, New York, 1987.

[5] R. P. Brent, Multiple-precision zero-finding methods and the complexity of elementary function evaluation, in *Analytic Computational Complexity* (edited by J. F. Traub), Academic Press, New York, 1975, 151–176.

[6] R. P. Brent, Fast multiple-precision evaluation of elementary functions, *J. ACM* 23 (1976), 242–251.

[7] R. P. Brent, The complexity of multiple-precision arithmetic, in *The Complexity of Computational Problem Solving*, University of Queensland Press, Brisbane, 1976, 126–165.

[8] R. P. Brent, Computation of the regular continued fraction for Euler's constant, *Mathematics of Computation* 31 (1977), 771–777.

[9] R. P. Brent and E. M. McMillan, Some new algorithms for high-precision computation of Euler's constant, *Mathematics of Computation* 34 (1980), 305–312.

[10] R. P. Brent, Ramanujan and Euler's constant, in *Proceedings of Symposia in Applied Mathematics*, Vol. 48 (edited by W. Gautschi), American Mathematical Society, Providence, Rhode Island, 1994, 541–545.

[11] Donald E. Knuth, *The Art of Computer Programming* Volume 2: *Seminumerical Algorithms* (third edition, 1997), Addison-Wesley, 1997.

[12] E. Salamin, Computation of π using arithmetic-geometric mean, *Mathematics of Computation* 30 (1976), 565–570.

Lecture 2

Communication and computation in some parallel algorithms*

*Six lectures on Algorithms, Trinity term 1999.
 lec02

Summary

- The importance of communication in hardware
  - Area-time bounds for VLSI chips
- Linear algebra
  - Matrix multiplication
  - Solution of linear systems
  - The SVD and eigenvalue problems
- Non-numerical problems
  - Merging
  - Sorting

Area-time bounds for VLSI

It is interesting to consider algorithms implemented in *hardware* before we consider those implemented in *software*.

Specifically, consider the model of VLSI circuits described in Ullman's book *Computational Aspects of VLSI* [13]. Variations on this model were used by Thompson [9], Brent and Kung [5], Brent and Goldschlager [4], etc to obtain lower bounds on the *area* and *time* required for some fundamental computations, e.g. binary multiplication, sorting, the FFT, evaluation of propositional calculus formulae, set equality, context-free language recognition, etc.

Sketch of the model

- A computation is performed in a planar, convex region R of area A.
- Wires of finite width λ are used for communication within R.
- I/O ports of finite size on the boundary of R are used for input and output (communication with the outside world).
- Wires are allowed to overlap, but the degree of overlap is bounded by ν, e.g. $\nu = 2$ is sufficient.
- Each input is read only once.
- Storage of one bit of information takes a fixed area.
- The circuit is synchronous with a fixed cycle time τ.
- Wires can transmit one bit in time τ.

For details, see Ullman [13] or the original papers.

Bisection

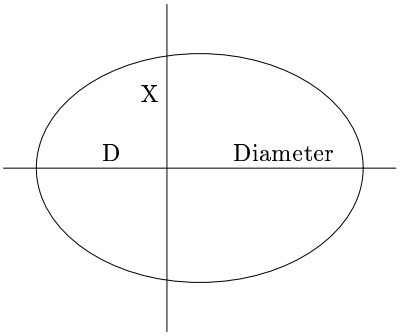


Figure 1: A VLSI chip

The ellipse is the boundary of a VLSI chip. D is the diameter and X is a chord of length L perpendicular to the diameter.

Given a nonempty set V of processing elements (ports and/or gates), X is chosen so that some of the elements of V lie on each side of it. This is called a *bisection* of V.

Information transfer

The (maximal) information transfer across X during a computation of time T is

$$I = WT/\tau \ ,$$

where W is the number of wires which intersect X.

Theorem. If there is a bisection of V with information transfer I then

$$AT^2 = \Omega(I^2) \ .$$

Idea of proof: $A = \Omega(L^2)$ and $L \geq \lambda W/\nu$, so

$$AT^2 = \Omega(W^2T^2) = \Omega(I^2) \ .$$

This theorem is from Brent and Goldschlager [4]. There are similar results (with slightly different definitions) in Brent and Kung [5] and Thompson [9].

Applications

To apply the theorem, we use a "crossing sequence" argument to give a lower bound on I, the information which has to cross X. Typically we obtain

$$I = \Omega(n) \ ,$$

where n is a measure of the size of the problem. Thus, we can conclude that

$$AT^2 = \Omega(n^2) \ .$$

For example, this bound applies to n-bit binary multiplication, evaluation of a propositional calculus formula (optionally in disjunctive normal form), and for testing set equality.

AT bounds

Often we can obtain an independent lower bound on the area, typically

$$A = \Omega(n) \ .$$

Then, multiplying the two bounds and taking a square root, we have

$$AT = \Omega(n^{3/2}) \ .$$

A lower bound on AT seems to be more natural than a bound on AT^2.

For example, the $AT = \Omega(n^{3/2})$ bound applies to binary multiplication, and shows that in a sense

multiplication is harder than addition

because we can perform binary addition in the same model with

$$AT = O(n) \ .$$

Parallel machines

There are many examples of parallel computer architectures (a few alive and well, many extinct). They all involve multiple processors but differ in other important respects –

- Memory may be local or shared (or actually local but apparently shared, or some combination, ...)
- The processors (real or virtual) may have independent instruction streams (MIMD) or a common instruction stream (SIMD). If MIMD, we can program them using a common program (SPMD) or different programs in each processor.
- The processors and memories may be connected in various different ways, which may or may not be visible to the programmer. For example, rings, tori, hypercubes, cliques (crossbars), trees, ...
- There may be a small number of powerful processors (perhaps vector processors) or a larger number of feeble processors.

Machine-independent models

It is because of these differences that machine-independent models such as BSP have been introduced. However, no standard has emerged yet, and programmers still resort to low-level features of machines in order to get higher efficiency (typically for benchmarks such as the *Linpack* benchmark, because they help to sell machines).

Data distribution

Given a parallel machine with p processors and a problem with input data D, we have to distribute D over the processors in some manner. (Of course, with a smart enough compiler, the data distribution might be invisible to the programmer.) The result may end up distributed over the processors and we have to specify how this is to occur[1].

[1] Otherwise problems such as sorting are trivial !

Linear algebra on parallel machines

Linear algebra problems with dense matrices provide nice, regular examples and are (relatively) easy to solve with high efficiency on parallel machines. If only all problems were so well-defined and regular !

Matrix multiplication

Consider, for example, the problem of forming the product C of two $n \times n$ matrices A and B. (The rectangular case is interesting and important, but we consider the square case for simplicity.)

Data distribution

We could distribute A over the processors by *rows, columns*, or *blocks*, and similarly for B and C. Whichever way we partition A, B and C, some communication between processors is necessary (unless everything is done on one processor and the others remain idle).

We assume that the classical, $O(n^3)$ matrix multiplication algorithm is used, and that the time for one multiply and add on a single processor (with data in cache etc) is τ. Thus, it would take time $T_1 \approx n^3\tau$ to solve the problem on a single processor. We hope to solve the problem p times faster on p processors, i.e. we hope that the *speedup*

$$S = \frac{T_1}{T_p}$$

is close to p, or that the *efficiency*

$$E = \frac{S}{p} = \frac{T_1}{pT_p}$$

is close to 1. Here T_p is the time required to solve the problem on p processors.

Communication versus computation

With current technology, communication between processors is typically much slower than communication/computation within a processor. To communicate a message of w words might take time

$$(Gw + H)\tau \ ,$$

where G and H are constants, typically much greater than 1.

We can interpret

- $1/(G\tau)$ as the processor to processor *communication bandwidth*, and
- $H\tau$ as the *startup cost* of a communication.

In practice such a simple linear model is inaccurate, but we use it for lack of anything better.

Block distribution

In order to minimise communication costs, the best way to distribute A and B is by blocks. Suppose that $p = s^2$ is a perfect square and $s|n$, for the sake of simplicity[2].

We partition A, B and C into $s \times s$ block matrices, where each block is of size $n/s \times n/s$.

For example, if $s = 2$, $p = 4$, we partition A as

$$A = \left[\begin{array}{c|c} A_{0,0} & A_{0,1} \\ \hline A_{1,0} & A_{1,1} \end{array}\right] \ ,$$

where each $A_{i,j}$ is a matrix of size $n/2 \times n/2$. (Here and below indices run from zero.)

[2]Such assumptions are usually made by lecturers; only programmers have to consider the difficult cases.

Estimate of efficiency

The processor assigned the block with indices (i, k) has to accumulate the sum

$$C_{i,k} = \sum_j A_{i,j} B_{j,k} \ ,$$

so it needs data from the "block row" i of A and the "block column" k of B.

For each $n/s \times n/s$ matrix product, taking time $(n/s)^3\tau$, the processor needs $2(n/s)^2$ words of data, which can to be transferred in time $2G(n/s)^2 + 2H$. If we neglect H and low order terms, we see that

$$T_p \approx \frac{n^3}{s^2}\left(1 + \frac{Gs}{n}\right) \ ,$$

so

$$E \approx 1/(1 + Gs/n) \ .$$

Hence, E is close to 1 only if

$$n \gg Gs \ .$$

Interpretation

Since G is typically in the range 100 to 1000, the inequality

$$n \gg Gs$$

means that n has to be large relative to $s = \sqrt{p}$.

Thus, we can not make efficient use of a large number of processors unless n is huge.

Gaussian elimination

In practice, it is more likely that we want to solve a linear system

$$Ax = b$$

than multiply two matrices. Consider the method of Gaussian elimination (without pivoting, for the time being). If we partition A into $p = s^2$ blocks as before, a problem of *load balance* rears its head.

As the elimination proceeds, all the "action" moves to the bottom right corner, and more and more processors have nothing to do. This is because A is gradually transformed into the upper triangular matrix U in the matrix factorisation $A = LU$. After the j-th iteration the first j columns are zero below the diagonal (or traditionally are used to store columns of L).

Gaussian elimination

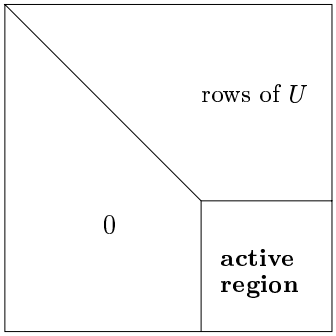


Figure 2: Decomposition $A \to LU$

Solution - scattered decomposition

A solution to the load balance problem is to use a *scattered* or *cyclic* distribution of the data A. Here matrix element $a_{i,j}$ is stored in processor

$$(i \bmod s, j \bmod s)$$

instead of in processor

$$(\lfloor is/n \rfloor, \lfloor js/n \rfloor) \ .$$

For such mappings it is convenient to take indices running from 0, as in C, rather than from 1, because the range of the "mod s" function is $\{0, 1, \ldots, s-1\}$.

Assume a cyclic data distribution. As the computation proceeds, each processor has a matrix of roughly the same "shape" (tending towards upper triangular), and we gain a factor of almost three in efficiency over that obtained using the block decomposition.

Scattered versus block decomposition

We have seen that the scattered decomposition gives better load balance than the block decomposition, at least for Gaussian elimination. (A little thought shows that they are equivalent for matrix multiplication.)

Another advantage of the scattered decomposition is that the location of element $a_{i,j}$ of the matrix A depends only on (i, j) and is *independent* of the dimensions of A. This is a great advantage if we are writing software to operate on matrices of arbitrary shape (not necessarily known at compile time).

Partial pivoting

Except for special classes of matrices, Gaussian elimination is unstable unless we perform pivoting to constrain the size of the multipliers (elements of L).

For example, the use of partial pivoting corresponds the a matrix factorisation

$$PA = LU \ ,$$

where P is a permutation matrix, chosen so that the elements $m_{i,j}$ of L are at most 1 in absolute value.

Communication overhead of pivoting

On a parallel machine, pivoting introduces additional communication overheads. At the k-th step we need to find the best pivot element in the k-th column, and this requires communication between the processors which have access to elements of the column. The volume of communication is small, but the startup costs are significant. To find the maximum of a vector stored in $s = \sqrt{p}$ processors, using a binary tree, and to broadcast the result to the processors, costs us about $H\tau \lg(p)$ just in startup costs, so a term

$$H\tau n \lg(p)$$

will occur in the estimate of T_p.

H may be about 100 (if special communication hardware is provided) or 10^6 or more (if communication is done entirely in software using interrupts), so the startup cost may well dominate for small and moderate n.

Distribution by columns

Instead of distributing both rows and columns in a scattered (cyclic) manner, it is tempting to distribute just the columns of A in this way. More precisely, element $a_{i,j}$ could be stored in processor

$$j \bmod p \ ,$$

where the processors are numbered $0, 1, \ldots, p-1$.

The advantage of this "cyclic by column" distribution is that a single processor has access to a whole column, so no communication is required to find the pivot element in that column. Information about the pivot element and its location still has to be broadcast to the other processors.

Disadvantages of distribution by columns

The distribution by columns has significant disadvantages.

- Communication in the horizontal direction has bandwidth reduced by a factor $s = \sqrt{p}$, from $s/(G\tau)$ to $1/(G\tau)$.
- For matrix multiplication $C \leftarrow AB$, do we distribute B by columns (for consistency) or by rows (for compatibility with the definition of matrix multiplication) ?

Generally, it seems preferable to use a distribution where rows and columns are treated in the same way.

Memory references per flop

On many machines it is impossible to achieve close to peak performance if Gaussian elimination is performed in the obvious way (via saxpys or rank-1 updates). This is because performance is limited by memory accesses rather than by floating-point arithmetic. Saxyps and rank-1 updates have a high ratio of memory references to floating point operations.

Close to peak performance can be obtained for matrix-vector or (better) matrix-matrix multiplication which (if implemented properly) have a lower ratio of memory references to floating-point operations.

Blocking

It is possible to reformulate Gaussian elimination so that most of the floating-point arithmetic is performed in matrix-matrix multiplications (level 3 BLAS). The idea is to introduce a "blocksize" or "bandwidth" parameter ω. Gaussian elimination is performed via saxpys or rank-1 updates in vertical strips of width ω. Once ω pivots have been chosen, a horizontal strip of height ω can be updated. At this point, a matrix-matrix multiplication can be used to update the lower right corner of A. The optimal choice of ω is usually about $\sqrt{n}$.

It is important to note that the introduction of the parameter w is *independent* of the data distribution on a parallel machine. There is certainly no need to distribute A in $\omega \times \omega$ blocks. For more on this and related topics, see my paper "The Linpack benchmark on the AP1000" [3].

Illustration

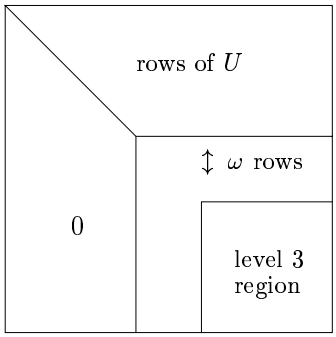


Figure 3: Blocked LU decomposition

The SVD

Suppose $m \geq n$. A singular value decomposition (SVD) of a real $m \times n$ matrix A is its factorisation into the product of three matrices

$$A = U\Sigma V^T ,$$

where U is $m \times n$ with orthonormal columns, Σ is an $n \times n$ non-negative diagonal matrix, and V is an $n \times n$ orthogonal matrix.

The diagonal elements σ_i of Σ are the *singular values* of A. The SVD has many applications (see Golub and Van Loan [7]).

Computing the SVD in parallel

The SVD is usually computed by a two-sided orthogonalisation process, e.g. by two-sided reduction to bidiagonal form followed by the QR algorithm. It is difficult to implement this Golub-Kahan-Reinsch algorithm efficiently on a parallel machine. It is much simpler to use a one-sided orthogonalisation method due to Hestenes.

The idea of Hestenes is to generate an orthogonal matrix V such that AV has orthogonal columns. Normalising the Euclidean length of each non-null column to unity, we get

$$AV = \tilde{U}\Sigma$$

As a null column of $\tilde{U}$ is always associated with a zero diagonal element of Σ, this gives the SVD of A.

Parallel implementation of the Hestenes method

Let $A_1 = A$ and $V_1 = I$. The Hestenes method uses a sequence of plane rotations Q_k chosen to orthogonalise two columns in

$$A_{k+1} = A_k Q_k \ .$$

If the matrix V is required, the plane rotations are accumulated using $V_{k+1} = V_k Q_k$. Under suitable conditions $\lim Q_k = I$, $\lim V_k = V$ and $\lim A_k = AV$. The matrix A_{k+1} differs from A_k only in two columns, say columns i and j, and the new columns are obtained from the old columns by a plane rotation through a certain angle θ, where $|\theta| \leq \pi/4$.

It is desirable for a "sweep" of $n(n-1)/2$ rotations to include all pairs (i, j) with $i < j$.

The chess tournament analogy

On a parallel machine we would like to orthogonalise several pairs of columns simultaneously. This should be possible so long as no column occurs in more than one pair. The problem is similar to that of organising a round-robin tournament between n players.

A game between players i and j corresponds to orthogonalising columns i and j, a round of several games played at the same time corresponds to orthogonalising several pairs of (disjoint) columns, and a tournament where each player plays each other player once corresponds to a sweep in which each pair of columns is orthogonalised. Thus, schemes which are well-known to chess players can be used to give orderings amenable to parallel computation.

It is desirable to minimise the number of parallel steps in a sweep, which corresponds to the number of rounds in the tournament.

Lazy players

On a parallel machine with restricted communication paths there are constraints on the orderings which we can implement efficiently. A useful analogy is a tournament of lazy chess players. After each round the players want to walk only a short distance to the board where they are to play the next round.

Using this analogy, suppose that each chess board corresponds to a processor and each player corresponds to a column of the matrix. A game between two players corresponds to orthogonalisation of the corresponding columns. If the chess boards (processors) are arranged in a linear array with nearest-neighbour communication paths, then the players should have to walk (at most) to an adjacent board between the end of one round and the beginning of the next round, i.e. columns of the matrix should have to be exchanged only between adjacent processors. Several orderings satisfying these conditions are known.

The number of steps in one sweep

Since A has n columns and at most $\lfloor n/2 \rfloor$ pairs can be orthogonalised in parallel, a sweep requires as least $n-1$ parallel steps (n even) or n parallel steps (n odd). The ordering of Brent and Luk [6] attains this minimum, and convergence can be guaranteed.

Data distribution

As described, each processor deals with two columns, so the column-wrapped representation is convenient. However, the block or scattered representations can also be used. The block representation involves less communication between processors than does the scattered representation if the standard orderings are used. However, the two representations are equivalent if different orderings are used.

The scattered representation does not have a load-balancing advantage here, since the matrix does not change shape.

The symmetric eigenvalue problem

There is a close connection between the Hestenes method for finding the SVD of a matrix A and the Jacobi method for finding the eigenvalues of the symmetric matrix $B = A^T A$.

Important differences are that the formulas defining the rotation angle θ involve elements $b_{i,j}$ of B rather than inner products of columns of A, and transformations must be performed on the left and right instead of just on the right (since $(AV)^T(AV) = V^T B V$).

Instead of permuting columns of A, we have to apply the same permutation to both rows and columns of B.

An implementation on a square systolic array of $n/2$ by $n/2$ processors is possible, and can be adapted to other parallel architectures. Again, a blocked data representation is desirable to minimise communication costs.

Parallel merging and sorting

To conclude, we consider a "non-numerical" problem – sorting data into order. It is assumed that each item of data has a key and the keys are totally ordered (an example is lexicographic ordering).

First, consider a simpler problem: merging data held on two processors (the solution of this problem will be useful for sorting).

The merge-exchange problem

Suppose processors P and Q each have n items of data and the data on each processor is already sorted. For example, if $n = 4$, processor P might have (A, B, F, Z) and processor Q might have (C, D, E, G).

The problem is to merge the $2n$ items of data (we assume they are all distinct) into one sorted list, and end with the first half of the list on processor P and the last half on processor Q. In our example, P should end up with (A, B, C, D) and Q with (E, F, G, Z).

We could transfer Q's data to P, merge it with P's data, then send the last half of the merged list back to Q. However, this is inefficient because P does all the work (Q is idle while P is merging), and some data may be transferred unnecessarily. Also, P needs more memory than is required just to store $2n$ items.

A solution – first find the median

Suppose the final sorted list is $(A_1, A_2, \ldots, A_{2n})$. Thus, P ends up with $(A_1, \ldots, A_n)$ and Q ends up with $(A_{n+1}, \ldots, A_{2n})$. If the processors can determine, by a small amount of communication and local computation, the value of the *median* element A_n, then a more efficient solution is possible –

P sends to Q all of its elements which are greater than A_n, and Q sends to P all of its elements which are less than or equal to A_n. Then, all each processor has to do is a local merge.

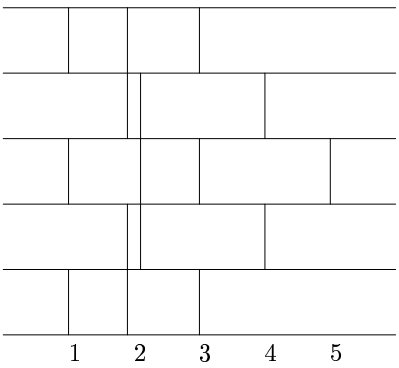
Finding the median can be done by binary search. At each stage, if the candidate median is M say, each processor counts how many of its elements exceed M, and sends the count to the other processor. Each processor now can determine how many elements in the final sorted list will exceed M. If this number is greater than n then M is increased, if less than n then M is decreased, ...

The communication tradeoff

Overall, there are about $\lg(n)$ communication steps to find the median. Thus on average, we trade the time required to communicate $O(n)$ elements for $O(\lg n)$ communication startup times. This is worthwhile if n is sufficiently large.

Sorting networks

A *sorting network* is a sorting circuit built from *comparators*, which are circuit elements which can sort two inputs (for the definition, see Knuth, Vol. 3). For example:



The network shown will sort 6 items in 5 steps using 12 comparators.

A generalisation of sorting networks

In a sorting network, we can replace each comparator by a "generalised comparator" which takes two sorted lists, merges them, and outputs the lower and upper halves of the result (as in the merge-exchange problem considered previously).

It can be shown that, *provided* the input and output lists of the generalised comparators are all of the same size k, and the initial inputs are sorted lists of size k, then the generalised network will sort correctly.

The restriction on input sizes is necessary, as examples show, but can be circumvented by the use of "virtual" elements, so is not a problem in practice (see Tridgell and Brent [10]).

Generic parallel sorting algorithm

We can take any serial algorithm which can be implemented as a sorting network (e.g. Batcher's merge-exchange algorithm, see Knuth [8, Algorithm M]), and convert it into a parallel algorithm which uses the merge-exchange operation.

Practical parallel sorting

Simply extending Batcher's algorithm is inefficient. A practical algorithm could add the steps of pre-balancing, fast internal sorting, and perhaps "almost sorting". For details see Tridgell and Brent [10].

Other parallel sorting algorithms

There are many serial sorting algorithms, and even more parallel algorithms. The main competitors appear to be

- Algorithms based on merge-exchange, as above.
- Algorithms based on sample-sort.
- Algorithms based on radix sort.

A disadvantage of the parallel algorithms based on sample-sort and radix-sort is that they require *all to all* communication, whereas algorithms based on merge-exchange require only processor to processor communication.

Another disadvantage of algorithms based on radix sort is that the keys must have fixed length and the ordering of keys can not be defined by the user.

For more on parallel sorting, see Andrew Tridgell's thesis [12].

References

[1] R. P. Brent, Parallel algorithms for digital signal processing, in *Numerical Linear Algebra, Digital Signal Processing and Parallel Algorithms* Springer-Verlag, 1991, 93–110.

[2] R. P. Brent, Parallel algorithms in linear algebra, *Algorithms and Architectures: Proc. Second NEC Research Symposium* SIAM, Philadelphia, 1993, 54–72.

[3] R. P. Brent, The LINPACK benchmark on the AP 1000, *Proceedings of Frontiers '92* (McLean, Virginia, October 1992), IEEE Press, 1992, 128–135.

[4] R. P. Brent and L. M. Goldschlager, Some area-time tradeoffs for VLSI, *SIAM J. Computing* 11 (1982), 737-747.

[5] R. P. Brent and H. T. Kung, The area-time complexity of binary multiplication, *J. ACM* 28 (1981), 521–534.

[6] R. P. Brent and F. T. Luk, The solution of singular-value and symmetric eigenvalue problems on multiprocessor arrays, *SISSC* 6 (1985), 69–84.

[7] G. H. Golub and C. Van Loan, *Matrix Computations*, second edition, Johns Hopkins Press, Baltimore Maryland, 1989.

[8] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, Addison-Wesley, Menlo Park, 1981.

[9] C. D. Thompson, Area-time complexity for VLSI, *Proc. 11th ACM Symp. on Theory of Computing*, 1979, 81–88.

[10] A. Tridgell and R. P. Brent, A general-purpose parallel sorting algorithm, *International J. of High Speed Computing* 7 (1995), 285–301.

[11] A. Tridgell, R. P. Brent and B. D. McKay, *Parallel Integer Sorting*, Tech. Report TR-CS-97-10, CSL, ANU, May 1997, 32 pp.

[12] A. Tridgell, *Efficient Algorithms for Sorting and Synchronization*, Ph. D. thesis, Australian National University, 1999.

[13] J. D. Ullman, *Computational Aspects of VLSI*, Computer Science Press, Maryland, 1984.

[14] B. B. Zhou, R. P. Brent and A. Tridgell, Efficient implementation of sorting algorithms on asynchronous distributed-memory machines, *Proc. ICPDS'94*, IEEE CS Press, Los Alamitos, California, 1994, 102–106.

Lecture 3

Fast and Numerically Stable Algorithms for Structured Matrices*

*Six lectures on Algorithms, Trinity term 1999.
 lec03

Abstract

We consider the numerical stability/instability of fast algorithms for solving systems of linear equations or linear least squares problems with a low displacement-rank structure. For example, the matrices involved may be Toeplitz or Hankel.

In particular, we consider algorithms which incorporate pivoting without destroying the structure, such as the GKO algorithm, and describe some recent results by Sweet and Brent, Ming Gu, Michael Stewart and others on the stability of these algorithms.

It is interesting to compare these results with the corresponding stability results for algorithms based on the seminormal equations and for the well known algorithms of Schur/Bareiss and Levinson.

Outline

- Structured matrices
  - Displacement structure
  - Cauchy-like matrices
  - Toeplitz-like matrices
  - Toeplitz $\leftrightarrow$ Cauchy
- Partial pivoting algorithms
  - Possible growth of generators
  - Improvements of Gu and Stewart
- Positive definite structured matrices
  - Schur/Bareiss algorithms
  - Comparison with Levinson
  - Generalised Schur algorithm
- Orthogonal factorisation
  - Weak stability
  - The problem of computing Q

Because of shortage of time, I will not consider look-ahead algorithms or iterative algorithms.

Acronyms

BBH	=	Bojanczyk, Brent & de Hoog.
BBHS	=	BBH & Sweet.
GKO	=	Gohberg, Kailath & Olshevsky.

Notation

R is a structured matrix,
T is a Toeplitz or Toeplitz-type matrix,
P is a permutation matrix,
L is lower triangular,
U is upper triangular,
Q is orthogonal.

Error Bounds

In error bounds $O_n(\varepsilon)$ means $O(\varepsilon f(n))$, where $f(n)$ is a polynomial in n.

Stability

Consider algorithms for solving a nonsingular, $n \times n$ linear system $Ax = b$.
There are many definitions of numerical stability in the literature. Our definitions follow those of Bunch [11]. Definition 1 says that the *computed* solution has to be the *exact* solution of a problem which is close to the original problem. This is the classical *backward stability* of Wilkinson.

Definition 1 *An algorithm for solving linear equations is* stable *for a class of matrices $\mathcal{A}$ if for each A in $\mathcal{A}$ and for each b the computed solution $\tilde{x}$ to $Ax = b$ satisfies $\widehat{A}\tilde{x} = \widehat{b}$, where $\widehat{A}$ is close to A and $\widehat{b}$ is close to b.*

Note that the matrix $\widehat{A}$ does not have to be in the class $\mathcal{A}$. For example, $\mathcal{A}$ might be the class of nonsingular Toeplitz matrices, but $\widehat{A}$ need not be a Toeplitz matrix. (If we do require $\widehat{A} \in \mathcal{A}$ we get what Bunch calls *strong stability*.)

Closeness

In Definition 1, "close" means close in a relative sense, using some norm, i.e.

$$\|\widehat{A} - A\|/\|A\| = O(\varepsilon),\ \|\widehat{b} - b\|/\|b\| = O(\varepsilon).$$

Recall our convention that polynomials in n may be omitted from $O(\varepsilon)$ terms.
We are ruling out faster than polynomial growth in n, such as $O(2^n\varepsilon)$ or $O(n^{\frac{\log n}{4}}\varepsilon)$. Perhaps this too strict (consider Gaussian elimination).

The Residual

The condition of Definition 1 is equivalent to saying that the scaled residual $\|A\tilde{x} - b\|/(\|A\| \cdot \|\tilde{x}\|)$ is small.

How Good is the Solution ?

Provided $\kappa\varepsilon$ is sufficiently small, stability implies that

$$\|\tilde{x} - x\|/\|x\| = O(\kappa\varepsilon).$$

Weak Stability

Definition 2 *An algorithm for solving linear equations is* weakly stable *for a class of matrices $\mathcal{A}$ if for each well-conditioned A in $\mathcal{A}$ and for each b the computed solution $\tilde{x}$ to $Ax = b$ is such that $\|\tilde{x} - x\|/\|x\|$ is small.*

In Definition 2, "small" means $O(\varepsilon)$, and "well-conditioned" means that $\kappa(A)$ is bounded by a polynomial in n. It is easy to see that stability implies weak stability.
Define the *residual*

$$r = A\tilde{x} - b.$$

It is well-known that

$$\frac{1}{\kappa}\frac{\|r\|}{\|b\|} \leq \frac{\|\tilde{x} - x\|}{\|x\|} \leq \kappa\frac{\|r\|}{\|b\|}.$$

Thus, for well-conditioned A, $\|\tilde{x} - x\|/\|x\|$ is small *if and only if* $\|r\|/\|b\|$ is small. (This gives an equivalent definition of weak stability.)

Displacement Structure

Structured matrices R satisfy a *Sylvester equation* which has the form

$$\nabla_{\{A_f, A_b\}}(R) \equiv A_f R - R A_b = \Phi\Psi\ ,$$

where A_f and A_b have some simple structure (usually banded, with 3 or fewer full diagonals), Φ and Ψ are $n \times \alpha$ and $\alpha \times n$ respectively, and α is some (small) integer.

The pair of matrices (Φ, Ψ) is called the $\{A_f, A_b\}$-*generator* of R.

α is called the $\{A_f, A_b\}$-*displacement rank* of R. We are interested in cases where α is small (say at most 4).

Example – Cauchy

Particular choices of A_f and A_b lead to definitions of basic classes of matrices. Thus, for a Cauchy matrix

$$C(\mathbf{t},\mathbf{s}) = \left[\frac{1}{t_i - s_j}\right]_{ij} ,$$

we have

$$A_f = D_t = \mathrm{diag}(t_1, t_2, \ldots, t_n) ,$$

$$A_b = D_s = \mathrm{diag}(s_1, s_2, \ldots, s_n)$$

and

$$\Phi^T = \Psi = [1, 1, \ldots, 1] .$$

More general matrices, where Φ and Ψ are any rank-α matrices, are called *Cauchy-type*.

Example – Toeplitz

For a Toeplitz matrix $T = [t_{ij}] = [a_{i-j}]$

$$A_f = Z_1 = \begin{bmatrix} 0 & 0 & \cdots & 0 & 1 \\ 1 & 0 & & & 0 \\ 0 & 1 & & & \vdots \\ \vdots & & \ddots & & \vdots \\ 0 & \cdots & 0 & 1 & 0 \end{bmatrix} ,$$

$$A_b = Z_{-1} = \begin{bmatrix} 0 & 0 & \cdots & 0 & -1 \\ 1 & 0 & & & 0 \\ 0 & 1 & & & \vdots \\ \vdots & & \ddots & & \vdots \\ 0 & \cdots & 0 & 1 & 0 \end{bmatrix} ,$$

$$\Phi = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ a_0 & a_{1-n} + a_1 & \cdots & a_{-1} + a_{n-1} \end{bmatrix}^T$$

and

$$\Psi = \begin{bmatrix} a_{n-1} - a_{-1} & \cdots & a_1 - a_{1-n} & a_0 \\ 0 & \cdots & 0 & 1 \end{bmatrix} .$$

We can generalize to *Toeplitz-type* matrices in the obvious way.

GE and Schur Complements

Let the input matrix, R_1, have the partitioning

$$R_1 = \begin{bmatrix} d_1 & \mathbf{w}_1^T \\ \mathbf{y}_1 & \tilde{R}_1 \end{bmatrix} .$$

The first step of normal Gaussian elimination is to premultiply R_1 by $\begin{bmatrix} 1 & \mathbf{0}^T \\ -\mathbf{y}_1/d_1 & I \end{bmatrix}$, which reduces it to $\begin{bmatrix} d_1 & \mathbf{w}_1^T \\ \mathbf{0} & R_2 \end{bmatrix}$, where

$$R_2 = \tilde{R}_1 - \mathbf{y}_1 \mathbf{w}_1^T / d_1$$

is the *Schur complement* of d_1 in R_1.

At this stage, R_1 has the factorisation

$$R_1 = \begin{bmatrix} 1 & \mathbf{0}^T \\ \mathbf{y}_1/d_1 & I \end{bmatrix} \begin{bmatrix} d_1 & \mathbf{w}_1^T \\ \mathbf{0} & R_2 \end{bmatrix} .$$

One can proceed recursively with the Schur complement R_2, eventually obtaining a factorisation $R_1 = LU$.

Structured Gaussian Elimination – The Idea

The key to *structured* Gaussian elimination is that the displacement structure is preserved under Schur complementation, and that the generators of the Schur complement R_{k+1} can be computed from the generators of R_k in $O(n)$ operations.

More precisely, we have the following theorem from GKO [21].

Theorem 1 *Let a matrix* $R_1 = \begin{bmatrix} d_1 & \mathbf{w}_1^T \\ \mathbf{y}_1 & \tilde{R}_1 \end{bmatrix}$ *satisfy the Sylvester equation*

$$\nabla_{\{A_{f,1},A_{b,1}\}}(R_1) = A_{f,1}R_1 - R_1A_{b,1} = \Phi^{(1)}\Psi^{(1)} ,$$

where $\Phi^{(1)} = [\varphi_1^{(1)T} \quad \varphi_2^{(1)T} \quad \cdots \quad \varphi_n^{(1)T}]^T$, $\Psi^{(1)} = [\psi_1^{(1)} \quad \psi_2^{(1)} \quad \cdots \quad \psi_n^{(1)}]$, $\varphi_i^{(1)} \in \mathbf{C}^{1\times\alpha}$ *and* $\psi_i^{(1)} \in \mathbf{C}^{1\times\alpha}$, $(i = 1, 2, \ldots, n)$. *Then* R_2, *the Schur complement of* d_1 *in* R_1, *satisfies the Sylvester equation*

$$\nabla_{\{A_{f,2},A_{b,2}\}}(R_2) = A_{f,2}R_2 - R_2A_{b,2} = \Phi^{(2)}\Psi^{(2)} ,$$

where $A_{f,2}$ *and* $A_{b,2}$ *are respectively* $A_{f,1}$ *and* $A_{b,1}$ *with their first rows and first columns deleted, and where* $\Phi^{(2)} = [0,\ \varphi_2^{(2)T},\ \varphi_3^{(2)T},\ \cdots,\varphi_n^{(2)T}]^T$ *and* $\Psi^{(2)} = [0,\ \psi_2^{(2)},\ \psi_3^{(2)},\ \cdots,\psi_n^{(2)}]$ *are given by*

$$\Phi^{(2)}_{2:n,:} = \Phi^{(1)}_{2:n,:} - \mathbf{y}_1\varphi_1^{(1)}/d_1 , \qquad (1)$$

$$\Psi^{(2)}_{:,2:n} = \Psi^{(1)}_{:,2:n} - \psi_1^{(1)}\mathbf{w}_1^T/d_1 . \qquad (2)$$

Structured Gaussian elimination

Algorithm 1 *(Structured Gaussian elimination)*

1. *Recover from the generator* $\Phi^{(1)}$, $\Psi^{(1)}$ *the first row and column of* $R_1 = \begin{bmatrix} d_1 & \mathbf{w}_1^T \\ \mathbf{y}_1 & \tilde{R}_1 \end{bmatrix}$.
2. $[1 \quad \mathbf{y}_1^T/d_1]^T$ *and* $[d_1 \quad \mathbf{w}_1^T]$ *are respectively the first column and row of* L_1 *and* U_1 *in the LU factorisation of* R_1.
3. *Compute by equations (1) and (2), the generator* $(\Phi^{(2)}, \Psi^{(2)})$ *for the Schur complement of* d_1 *in* R_1.
4. *Proceed recursively with* $\Phi^{(2)}$ *and* $\Psi^{(2)}$. *Each major step yields* $[1 \quad \mathbf{y}_k^T/d_k]^T$ *and* $[d_k \quad \mathbf{w}_k^T]$, *which are respectively the first column and row of* L_k *and* U_k *in the LU factorisation of* R_k. *Column* k *of* L *and row* k *of* U *are respectively* $[\mathbf{0}_{k-1}^T \quad 1 \quad \mathbf{y}_k^T/d_k]^T$ *and* $[\mathbf{0}_{k-1}^T \quad d_k \quad \mathbf{w}_k^T]$.

Partial Pivoting

Row and/or column interchanges destroy the structure of matrices such as Toeplitz matrices. However, if A_f is diagonal (which is the case for Cauchy and Vandermonde type matrices), then

> *the structure is preserved under row permutations.*

This observation leads to the *GKO-Cauchy* algorithm for fast factorisation of Cauchy-type matrices with partial pivoting, and many recent variations on the theme by Boros, Gohberg, Ming Gu, Heinig, Kailath, Olshevsky, M. Stewart, *et al.*

Toeplitz to Cauchy

Heinig (1994) showed that, if T is a Toeplitz-type matrix, then

$$R = FTD^{-1}F^*$$

is a Cauchy-type matrix, where

$$F = \frac{1}{\sqrt{n}}[e^{2\pi i(k-1)(j-1)/n}]_{1\le k,j\le n}$$

is the Discrete Fourier Transform matrix,

$$D = \mathrm{diag}(1, e^{\pi i/n}, \ldots, e^{\pi i(n-1)/n}),$$

and the generators of T and R are related by unitary transformations (see [38, Thm. 2.2] for the details).

The transformation $T \leftrightarrow R$ is perfectly stable because F and D are unitary.

Note that F and R are (in general) complex even if T is real.

GKO-Toeplitz

As pointed out by Heinig (1994) and exploited by GKO (1995), it is possible to convert the generators of T to the generators of R in $O(n \log n)$ operations via FFTs. R can then be factorised as $R = P^T LU$ using GKO-Cauchy. Thus, from the factorisation

$$T = F^* P^T LUFD \ ,$$

a linear system involving T can be solved in $O(n^2)$ (complex) operations.

Other structured matrices, such as Toeplitz-plus-Hankel, Vandermonde, Chebyshev-Vandermonde, etc, can be converted to Cauchy-type matrices in a similar way.

Error Analysis

Because GKO-Cauchy (and GKO-Toeplitz) involve partial pivoting, we might guess that their stability would be similar to that of Gaussian elimination with partial pivoting.

The Catch

Unfortunately, there is a flaw in the above reasoning. During GKO-Cauchy the *generators* have to be transformed, and the partial pivoting does not ensure that the transformed generators are small.

Sweet & Brent (1995) show that significant generator growth can occur if all the elements of $\Phi\Psi$ are small compared to those of $|\Phi||\Psi|$. This can not happen for ordinary Cauchy matrices because $\Phi^{(k)}$ and $\Psi^{(k)}$ have only one column and one row respectively. However, it can happen for higher displacement-rank Cauchy-type matrices, even if the original matrix is well-conditioned.

The Toeplitz Case

In the Toeplitz case there is an extra constraint on the selection of Φ and Ψ, but it is still possible to give examples where the normalised solution error grows like κ^2 and the normalised residual grows like κ, where κ is the condition number of the Toeplitz matrix. Thus, the GKO-Toeplitz algorithm is (at best) weakly stable.

It is easy to think of modified algorithms which avoid the examples given by Sweet & Brent, but it is difficult to prove that they are stable in all cases. Stability depends on the worst case, which may be rare and hard to find by random sampling.

Gu and Stewart's Improvements

The problem with the original GKO algorithm is growth in the generators. Ming Gu suggested exploiting the fact that the generators are not unique.

Recall the *Sylvester equation*

$$\nabla_{\{A_f, A_b\}}(R) = A_f R - R A_b = \Phi\Psi \ ,$$

where the generators Φ and Ψ are $n \times \alpha$ and $\alpha \times n$ respectively. Clearly we can replace Φ by ΦM and Ψ by $M^{-1}\Psi$, where M is any invertible $\alpha \times \alpha$ matrix, because this does not change the product $\Phi\Psi$. Similarly at later stages of the GKO algorithm.

Ming Gu (1995) proposes taking M to orthogonalize the columns of Φ (that is, at each stage we do an orthogonal factorisation of the generators). Michael Stewart (1997) proposes a (cheaper) LU factorisation of the generators. In both cases, clever pivoting schemes give error bounds analogous to those for Gaussian elimination with partial pivoting.

Gu and Stewart's Error Bounds

The error bounds obtained by Ming Gu and Michael Stewart involve an exponentially growing factor K^n where K depends on the ratio of the largest to smallest modulus elements in the Cauchy matrix

$$\left[\frac{1}{t_i - s_j}\right]_{ij}.$$

Although this is unsatisfactory, it is similar to the factor 2^{n-1} in the error bound for Gaussian elimination with partial pivoting.

Michael Stewart (1997) gives some interesting numerical results which indicate that his scheme works well, but more numerical experience is necessary before a definite conclusion can be reached.

In practice, we can use an $O(n^2)$ algorithm such as Michael Stewart's, check the residual, and resort to iterative refinement or a stable $O(n^3)$ algorithm in the (rare) cases that it is necessary.

Positive Definite Structured Matrices

An important class of algorithms, typified by the algorithm of Bareiss (1969), find an LU factorisation of a Toeplitz matrix T, and (in the symmetric case) are related to the classical algorithm of Schur for the continued fraction representation of a holomorphic function in the unit disk.

It is interesting to consider the numerical properties of these algorithms and compare with the numerical properties of the Levinson[1] algorithm (which essentially finds an LU factorisation of T^{-1}).

[1] Discovered independently by Kolmogorov and Wiener in 1941.

Bareiss – Positive Definite Case

BBHS (1995) have shown that the numerical properties of the Bareiss algorithm are similar to those of Gaussian elimination (*without* pivoting). Thus, the algorithm is stable for positive definite symmetric T.

The Levinson algorithm can be shown to be weakly stable for bounded n, and numerical results by Varah, BBHS and others suggest that this is all that we can expect. Thus, the Bareiss algorithm is (generally) better numerically than the Levinson algorithm.

Cybenko showed that if certain quantities called "reflection coefficients" are positive then the Levinson-Durbin algorithm for solving the Yule-Walker equations (a positive-definite system with special right-hand side) is stable. However, "random" positive-definite Toeplitz matrices do not usually satisfy Cybenko's condition.

The Generalised Schur Algorithm

The Schur algorithm can be generalised to factor a large variety of structured matrices – see Kailath and Chun (1994) or Kailath and Sayed (1995). For example, the generalised Schur algorithm applies to block Toeplitz matrices, Toeplitz block matrices, and to matrices of the form $T^T T$ where T is rectangular Toeplitz.

It is natural to ask if the stability results of BBHS (which are for the classical Schur/Bareiss algorithm) extend to the generalised Schur algorithm. This was considered by M. Stewart and Van Dooren (1997), and also (in more generality) by Chandrasekharan and Sayed (1998).

The conclusion is that the generalised Schur algorithm is stable for positive definite matrices, *provided* that the hyperbolic transformations in the algorithm are implemented correctly. In contrast, BBHS showed that stability of the classical Schur/Bareiss algorithm is not so dependent on details of the implementation.

Fast Orthogonal Factorisation

In an attempt to achieve stability without pivoting, and to solve $m \times n$ least squares problems, it is natural to consider algorithms for computing an orthogonal factorisation

$$T = QU$$

of T. The first such $O(n^2)$ algorithm[2] was introduced by Sweet (1982–84). Unfortunately, Sweet's algorithm is unstable.

Other $O(n^2)$ algorithms for computing the matrices Q and U or U^{-1} were given by BBH (1986), Chun *et al* (1987), Cybenko (1987), and Qiao (1988), but none of them has been shown to be stable, and in several cases examples show that they are unstable.

[2] For simplicity the time bounds assume $m = O(n)$.

The Problem – Q

Unlike the classical $O(n^3)$ Givens or Householder algorithms, the $O(n^2)$ algorithms do not form Q in a numerically stable manner as a product of matrices which are (close to) orthogonal.

For example, the algorithms of Bojanczyk, Brent and de Hoog (1986) and Chun *et al* (1987) depend on Cholesky downdating, and numerical experiments show that they do not give a Q which is close to orthogonal.

The generalised Schur algorithm, applied to T^TT, computes the upper triangular matrix U but not the orthogonal matrix Q.

The Saving Grace – U and Semi-Normal Equations

It can be shown (BBH, 1995) that, provided the Cholesky downdates are implemented in a certain way (analogous to the condition for the stability of the generalised Schur algorithm) the BBH algorithm computes U in a weakly stable manner. In fact, the computed upper triangular matrix $\tilde{U}$ is about as good as can be obtained by performing a Cholesky factorisation of T^TT, so

$$\|T^TT - \tilde{U}^T\tilde{U}\| / \|T^TT\| = O_m(\varepsilon) \; .$$

Thus, by solving

$$\tilde{U}^T\tilde{U}x = T^Tb$$

(the so-called *semi-normal* equations) we have a *weakly stable* algorithm for the solution of general Toeplitz systems $Tx = b$ in $O(n^2)$ operations. The solution can be improved by iterative refinement if desired.

Note that the computation of Q is avoided.

Computing Q Stably

It is difficult to give a satisfactory $O(n^2)$ algorithm for the computation of Q in the factorisation

$$T = QU$$

Chandrasekharan and Sayed get close – they give a stable algorithm to compute the factorisation

$$T = LQU$$

where L is lower triangular, provided that T is square. Their algorithm can be used to solve linear equations but not for the least squares problem. Also, because their algorithm involves embedding the $n \times n$ matrix T in a $2n \times 2n$ matrix

$$\begin{bmatrix} T^TT & T^T \\ T & 0 \end{bmatrix} ,$$

the constant factors in the operation count are large: $59n^2 + O(n \log n)$, compared to $8n^2 + O(n \log n)$ for BBH and seminormal equations.

Some Open Questions

- How do the GKO and similar algorithms using partial pivoting compare with the "lookahead" algorithms of Chan and Hansen [13], Freund and Zha [19], Gutknecht [25], and others ?
- Is there a stable (not just weakly stable) fast algorithm for the (rectangular) structured least squares problem ?
- What can be said about the stability (or instability) of the "superfast" algorithms whose running time is

$$O\left(n(\log n)^2\right) \ ?$$

  For these algorithms see Ammar and Gragg [1], Brent, Gustavson and Yun [9].
- What are the best generalisations to block-structured problems, e.g. block Toeplitz with $\sqrt{n} \times \sqrt{n}$ blocks ?

References

[1] G. S. Ammar and W. B. Gragg, Superfast solution of real positive definite Toeplitz systems, *SIAM J. Matrix Anal. Appl.* 9 (1988), 61–76.

[2] E. H. Bareiss, Numerical solution of linear equations with Toeplitz and vector Toeplitz matrices, *Numer. Math.* 13 (1969), 404–424.

[3] A. W. Bojanczyk, R. P. Brent, P. Van Dooren and F. R. de Hoog, A note on downdating the Cholesky factorization, *SISSC* 8 (1987), 210–220.

[4] A. W. Bojanczyk, R. P. Brent and F. R. de Hoog, QR factorization of Toeplitz matrices, *Numer. Math.* 49 (1986), 81–94.

[5] A. W. Bojanczyk, R. P. Brent and F. R. de Hoog, Stability analysis of a general Toeplitz systems solver, *Numerical Algorithms* 10 (1995), 225–244.

[6] A. W. Bojanczyk, R. P. Brent, F. R. de Hoog and D. R. Sweet, On the stability of the Bareiss and related Toeplitz factorization algorithms, *SIAM J. Matrix Anal. Appl.* 16 (1995), 40–57.

[7] T. Boros, T. Kailath and V. Olshevsky, Fast algorithms for solving Cauchy linear systems, preprint, 1995.

[8] R. P. Brent, Stability of fast algorithms for structured linear systems, in *Fast Reliable Algorithms for Matrices with Structure* (A. H. Sayed and T. Kailath, eds.), SIAM, Philadelphia, 1999, to appear.

[9] R. P. Brent, F. G. Gustavson and D. Y. Y. Yun, Fast solution of Toeplitz systems of equations and computation of Padé approximants, *J. Algorithms* 1 (1980), 259–295.

[10] J. R. Bunch, Stability of methods for solving Toeplitz systems of equations, *SISSC* 6 (1985), 349–364.

[11] J. R. Bunch, The weak and strong stability of algorithms in numerical linear algebra, *Linear Alg. Appl.* 88/89 (1987), 49–66.

[12] J. R. Bunch, Matrix properties of the Levinson and Schur algorithms, *J. Numerical Linear Algebra with Applications* 1 (1992), 183–198.

[13] T. F. Chan and P. C. Hansen, A look-ahead Levinson algorithm for general Toeplitz systems, *IEEE Trans. Signal Process.* 40 (1992), 1079–1090.

[14] S. Chandrasekaran and A. H. Sayed, Stabilizing the generalized Schur algorithm, *SIAM J. Matrix Anal. Appl.* 17 (1996), 950–983.

[15] S. Chandrasekaran and A. H. Sayed, A fast stable solver for nonsymmetric Toeplitz and quasi-Toeplitz systems of linear equations, *SIAM J. Matrix Anal. Appl.* 19 (1998), 107–139.

[16] J. Chun, T. Kailath and H. Lev-Ari, Fast parallel algorithms for QR and triangular factorization, *SISSC* 8 (1987), 899–913.

[17] G. Cybenko, The numerical stability of the Levinson-Durbin algorithm for Toeplitz systems of equations, *SISSC* 1 (1980), 303–319.

[18] J. Durbin, The fitting of time-series models, *Rev. Int. Stat. Inst.* 28 (1959), 229–249.

[19] R. W. Freund and H. Zha, Formally biorthogonal polynomials and a look-ahead Levinson algorithm for general Toeplitz systems, *Linear Algebra Appl.* 188/189 (1993), 255–303.

[20] I. Gohberg (editor), *I. Schur Methods in Operator Theory and Signal Processing* (Operator Theory: Advances and Applications, Volume 18), Birkhäuser Verlag, Basel, 1986.

[21] I. Gohberg, T. Kailath and V. Olshevsky, Gaussian elimination with partial pivoting for matrices with displacement structure, *Math. Comp.* 64 (1995), 1557–1576.

[22] G. H. Golub and C. Van Loan, *Matrix Computations*, second edition, Johns Hopkins Press, Baltimore, Maryland, 1989.

[23] Ming Gu, Stable and efficient algorithms for structured systems of linear equations, *SIMAX* 19 (1998), 279–306.

[24] Ming Gu, *New fast algorithms for structured least squares problems*, Tech. Report LBL-37878, Lawrence Berkeley Laboratory, Nov. 1995.

[25] M. H. Gutknecht, Stable row recurrences for the Padé table and generically superfast lookahead solvers for non-Hermitian Toeplitz systems, *Linear Algebra Appl.* 188/189 (1993), 351–422.

[26] P. C. Hansen and H. Gesmar, Fast orthogonal decomposition of rank deficient Toeplitz matrices, *Numerical Algorithms* 4 (1993), 151–166.

[27] G. Heinig, Inversion of generalized Cauchy matrices and other classes of structured matrices, *Linear Algebra for Signal Processing, IMA Volumes in Mathematics and its Applications, Vol. 69,* Springer, 1994, 95–114.

[28] T. Kailath and J. Chun, Generalized displacement structure for block-Toeplitz, Toeplitz-block, and Toeplitz-derived matrices, *SIAM J. Matrix Anal. Appl.* 15 (1994), 114–128.

[29] T. Kailath and A. H. Sayed, Displacement structure: theory and applications, *SIAM Review* 37 (1995), 297–386.

[30] A. N. Kolmogorov, Interpolation and extrapolation of stationary random sequences, *Izvestia Akad. Nauk SSSR* 5 (1941), 3–11 (in Russian). German summary, *ibid* 11–14.

[31] N. Levinson, The Wiener RMS (Root-Mean-Square) error criterion in filter design and prediction, *J. Math. Phys.* 25 (1947), 261–278.

[32] F. T. Luk and S. Qiao, A fast but unstable orthogonal triangularization technique for Toeplitz matrices, *Linear Algebra Appl.* 88/89 (1987), 495–506.

[33] S. Qiao, Hybrid algorithm for fast Toeplitz orthogonalization, *Numer. Math.* 53 (1988), 351–366.

[34] I. Schur, Über Potenzreihen, die im Innern des Einheitskreises beschränkt sind, *J. fur die Reine und Angewandte Mathematik* 147 (1917), 205–232. English translation in [20], 31–59.

[35] M. Stewart, Stable pivoting for the fast factorization of Cauchy-like matrices, preprint, Jan. 13, 1997.

[36] M. Stewart and P. Van Dooren, Stability issues in the factorization of structured matrices, *SIAM J. Matrix Anal. Appl.* 18 (1997), 104–118.

[37] D. R. Sweet, Fast Toeplitz orthogonalization, *Numer. Math.* 43 (1984), 1–21.

[38] D. R. Sweet and R. P. Brent, Error analysis of a fast partial pivoting method for structured matrices, *Proceedings SPIE, Volume 2563, Advanced Signal Processing Algorithms* SPIE, Bellingham, Washington, 1995, 266–280.

[39] G. Szegö, *Orthogonal Polynomials*, AMS Colloquium publ. XXIII, AMS, Providence, Rhode Island, 1939.

[40] N. Wiener, *Extrapolation, Interpolation and Smoothing of Stationary Time Series, with Engineering Applications*, Technology Press and Wiley, New York, 1949 (originally published in 1941 as a Technical Report).

Lecture 4

Uses of Randomness in Computation*

*Six lectures on Algorithms, Trinity term 1999.
 lec04

Summary

Random number generators are widely used in practical algorithms. Examples include simulation, number theory (primality testing and integer factorization), fault tolerance, routing, cryptography, optimization by simulated annealing, and perfect hashing.

Complexity theory usually considers the worst-case behaviour of deterministic algorithms, but it can also consider average-case behaviour if it is assumed that the input data is drawn randomly from a given distribution. Rabin popularised the idea of "probabilistic" algorithms, where randomness is incorporated into the algorithm instead of being assumed in the input data. Yao showed that there is a close connection between the complexity of probabilistic algorithms and the average-case complexity of deterministic algorithms.

In this lecture I give examples of the uses of randomness in computation, discuss the contributions of Rabin, Yao and others, and mention some open questions.

Checking out Charon

Charon, a future spacecraft, is somewhere near Pluto, but because of its distance and low-power transmitter, communication with it is very slow. We want to check that a critical program in the Charon's memory is correct, and has not been corrupted by a passing cosmic ray. How can we do this without transmitting the whole program to or from Charon ?

Here is one way[1]. The program we want to check (say N_1) and the correct program on Earth (say N_2) can be regarded as multiple-precision integers. Choose a random odd number p in the interval $(10^9, 2 \times 10^9)$. Transmit p to Charon and ask it to compute

$$r_1 \leftarrow N_1 \bmod p$$

and send it back to Earth. Only a few bits (no more than 64 for p and r_1) need be transmitted between Earth and Charon, so we can afford to use good error correction/detection.

[1] Rabin's "Library of Congress on Mars" problem.

On Earth · · ·

On Earth we compute $r_2 \leftarrow N_2 \bmod p$, and check if $r_1 = r_2$. There are two possibilities:

- $r_1 \neq r_2$. We conclude that $N_1 \neq N_2$. Charon's program has been corrupted ! If there are only a small number of errors, they can be localised by binary search using $O(\log \log N_1)$ small messages.
- $r_1 = r_2$. We conclude that Charon's program is *probably* correct. More precisely, if Charon's program is *not* correct there is only a probability of less than 10^{-9} that $r_1 = r_2$, i.e. that we have a "false positive". If this probability is too large for the quality-assurance team to accept, just repeat the process (say) ten times with different random odd numbers (preferably prime) $p_1, p_2, \ldots, p_{10}$. If $N_1 \neq N_2$, there is a probability of less than
$$10^{-90}$$
that we get $r_1 = r_2$ ten times in a row. This should be good enough.

The Structure

Our procedure has the following form. We ask a question with a yes/no answer. The precise question depends on a random number. If the answer is "no", we can assume that it is correct. If the answer is "yes", there is a small probability of error, but we can reduce this probability to a negligible level by repeating the procedure a few times with *independent* random numbers.

We call such a procedure a *probabilistic* algorithm; other common names are *randomised* algorithm and *Monte Carlo* algorithm.

Disclaimer

It would be much better to build error correcting hardware into Charon, and not depend on checking from Earth.

Testing Primality

Here is another example[2] with the same structure. We want an algorithm to determine if a given odd positive integer n is prime. Write n as $2^k q + 1$, where q is odd and $k > 0$.

Algorithm P

1. Choose a random integer x in $(1, n)$.
2. Compute $y = x^q \bmod n$. This can be done with $O(\log q)$ operations mod n, using the binary representation of q.
3. If $y = 1$ then return "yes".
4. For $j = 1, 2, \ldots, k$ do

   if $y = n - 1$ then return "yes"

   else if $y = 1$ then return "no"

   else $y \leftarrow y^2 \bmod n$.
5. Return "no".

[2] Due to M. O. Rabin, with improvements by G. L. Miller. See Knuth, Vol. 2, §4.5.4.

Fermat's Little Theorem

To understand the mathematical basis for Algorithm P, recall Fermat's little Theorem: if n is prime and $0 < x < n$, then

$$x^{n-1} = 1 \bmod n.$$

Thus, if $x^{n-1} \neq 1 \bmod n$, we can definitely say that n is composite.

Unfortunately, the converse of Fermat's little theorem is false: if $x^{n-1} = 1 \bmod n$ we can not be sure that n is prime. There are examples (called *Carmichael numbers*) of composite n for which x^{n-1} is always 1 mod n when $\mathrm{GCD}(x, n) = 1$. The smallest example is

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

Another example is[3]

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

[3] Hardy's taxi number, $1729 = 12^3 + 1^3 = 10^3 + 9^3$.

An Extension

A slight extension of Fermat's little Theorem is useful, because its converse is *usually* true.

If $n = 2^k q + 1$ is an odd prime, then either $x^q = 1 \bmod n$, or the sequence

$$\left(x^{2^j q} \bmod n\right)_{j=0,1,\ldots,k}$$

ends with 1, and the value just preceding the first appearance of 1 must be $n - 1$.

Proof: If $y^2 = 1 \bmod n$ then $n | (y-1)(y+1)$. Since n is prime, $n | (y-1)$ or $n | (y+1)$. Thus $y = \pm 1 \bmod n$. □

The extension gives a *necessary* (but not sufficient) condition for primality of n. Algorithm P just checks if this condition is satisfied for a random choice of x, and returns "yes" if it is.

Reliability of Algorithm P

Algorithm P can not give false negatives (unless we make an arithmetic mistake), but it can give false positives (i.e. "yes" when n is composite). However, the probability of a false positive is less than 1/4. (Usually much less – see Knuth, ex. 4.5.4.22.) Thus, if we repeat the algorithm 10 times there is less than 1 in 10^6 chance of a false positive, and if we repeat 100 times the results should satisfy anyone but a member of the PRG.

Algorithm P works even if the input is a Carmichael number.

Use of Randomness

In both our examples randomness was introduced into the *algorithm*.

We did not make any assumption about the distribution of inputs.

Summary of Algorithm P

Given any $\varepsilon > 0$, we can check primality of a number n in

$$O((\log n)^3 \log(1/\varepsilon))$$

bit-operations[4], provided we are willing to accept a probability of error of at most ε.

By way of comparison, the best known *deterministic* algorithm takes

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

bit-operations, and is much more complicated. If we assume the *Generalised Riemann Hypothesis*, the exponent can be reduced to 5. (But who believes in GRH with as much certainty as Algorithm P gives us ?)

[4] We can factor n deterministically in $O(\log n)$ *arithmetic* operations, but this result is useless because the operations are on numbers as large as 2^n. Thus, it is more realistic to consider bit-operations.

Error-Free Algorithms

The probabilistic algorithms considered so far (*Monte Carlo* algorithms) can give the wrong answer with a small probability. There is another class of probabilistic algorithms (*Las Vegas* algorithms) for which the answer is always correct; only the runtime is random[5]. An interesting example is H. W. Lenstra's *elliptic curve method* (ECM) for integer factorisation. To avoid trivial cases, suppose we want to find a prime factor $p > 3$ of an odd composite integer N.

To motivate ECM, consider an earlier algorithm, Pollard's "$p-1$" method. This works if $p-1$ is "smooth", i.e. has only small prime factors. $p-1$ is important because it is the order of the multiplicative group G of the field F_p. The problem is that G is fixed.

[5] In practical cases the expected runtime is finite. It is possible that the algorithm does not terminate, but with probability zero.

Lenstra's Idea

Lenstra had the idea of using a group $G(a,b)$ which depends on parameters (a,b). By randomly selecting a and b, we get a large set of different groups, and some of these should have smooth order.

The group $G(a,b)$ is the group of points on the *elliptic curve*

$$y^2 = x^3 + ax + b \bmod p,$$

and by a famous theorem[6] the order of $G(a,b)$ is an integer in the interval

$$(p - 1 - 2\sqrt{p},\ \ p - 1 + 2\sqrt{p})$$

The distribution in this interval is not uniform, but it is "close enough" to uniform for our purposes.

[6] The "Riemann hypothesis for finite fields". $G(a,b)$ is known as the "Mordell-Weil" group. The result on its order follows from a theorem of Hasse (1934), later generalised by A. Weil and Deligne.

Runtime of ECM

Under plausible assumptions ECM has expected run time

$$T = O\left(\exp(\sqrt{c \ln p \ln \ln p})(\ln N)^2\right),$$

where $c \simeq 2$.

Note that T depends mainly on the size of p, the factor found, and not very strongly on N. In practice the run time is close to an exponentially distributed random variable with mean and variance about T.

Examples of the use of ECM to factor large numbers will be given in Lecture 6.

Diffie-Hellman key exchange

Suppose Bob and Alice want to send messages to each other using ordinary (not public-key) cryptography. They need to agree on a key K to use for encrypting/decrypting their messages. This may be difficult if they are communicating by phone or email and someone (Eve) is eavesdropping. Diffie and Hellman suggested a nice solution.

First, Bob and Alice agree on a large prime p and an element g which is a primitive root mod p. Preferably $q = (p-1)/2$ should be prime.

Bob/Alice can find suitable p and q using Algorithm P, and then find g by a randomized algorithm. Testing that g is a primitive root is made easy because the factorisation of $p-1$ is known.

Bob and Alice can make p and g public. It does not matter if Eve knows them.

Diffie-Hellman continued

The Diffie-Hellman algorithm for generating a key K known to Bob and Alice, but not to Eve, is as follows.

1. Alice chooses a random $x \in \{2, \ldots, p-2\}$, computes $X = g^x \bmod p$, and sends X to Bob.
2. Bob chooses a random $y \in \{2, \ldots, p-2\}$, computes $Y = g^y \bmod p$, and sends Y to Alice.
3. Alice computes $K = Y^x \bmod p$.
4. Bob computes $K = X^y \bmod p$.

Now both Alice and Bob know $K = g^{xy} \bmod p$, so it can be used as a key or transformed into a key in some agreed manner.

Eve may know p, g, X and Y. However, she does not know x or y. Although it has not been proved, it seems that she can not compute $K = X^y \bmod p = Y^x \bmod p$ without effectively finding x or y, and this requires solving a discrete logarithm problem (hard ?).

Minimal Perfect Hashing

Hashing is a common technique used to map words into a small set of integers (which may then be used as indices to address a table). Thus, the computation $r_1 \leftarrow N_1 \bmod p$ used in our "Charon" example can be considered as a hash function.

Formally, consider a set

$$W = \{w_0, w_1, \ldots, w_{m-1}\}$$

of m words w_j, each of which is a finite string of symbols over a finite alphabet Σ. A *hash function* is a function

$$h : W \to I,$$

where $I = \{0, 1, \ldots, k-1\}$ and k is a fixed integer (the table size).

Collisions

A *collision* occurs if two words w_1 and w_2 map to the same address, i.e. if $h(w_1) = h(w_2)$. There are various techniques for handling collisions. However, these complicate the algorithms and introduce inefficiencies. In applications where W is fixed (e.g. the reserved words in a compiler), it is worth trying to avoid collisions.

Perfection

If there are no collisions, the hash function is called *perfect*.

Minimal Perfection

For a perfect hash function, we must have $k \geq m$. If $k = m$ the hash function is *minimal*.

Problem

Given a set W, how can we compute a minimal perfect hash function ?

A Slow Algorithm

We could try "random" hash functions h. However, there are m^m possible functions and only $m!$ are perfect. Thus, the probability of h begin perfect is

$$\frac{m!}{m^m} \sim \frac{\sqrt{2\pi m}}{e^m}$$

and it will take us on average the reciprocal of this, i.e.

$$\frac{e^m}{\sqrt{2\pi m}}$$

trials to find a perfect hash function. This is too slow unless m is very small.

The CHM Algorithm

Czech, Havas and Majewski (CHM) give a probabilistic algorithm which runs in expected time $O(m)$ (ignoring the effect of finite word-length). Their algorithm uses some properties of *random graphs*.

Take $n = 3m$, and let

$$V = \{1, 2, \ldots, n\}.$$

CHM take two independent pseudo-random functions

$$h_1 : W \to V, \quad h_2 : W \to V,$$

and let

$$E = \{(h_1(w), h_2(w)) \mid w \in W\}.$$

We can think of $G = (V, E)$ as a random graph with n vertices V and (at most) m edges E.

Acyclicity

If G has less than m edges or G has cycles, CHM reject the choice of h_1, h_2 and try again. Eventually they get a graph G with m edges and no cycles.

Because $n = 3m$, the expected number of trials is a constant – about $\sqrt{3}$, or more generally

$$\sqrt{\frac{n}{n - 2m}},$$

for large m and $n > 2m$.

The Perfect Hash Function

Once an acceptable G has been found, it is easy to compute (and store in a table) a function

$$g : V \to 0, 1, \ldots, m-1$$

such that

$$h(w) = g(h_1(w)) + g(h_2(w)) \bmod m$$

is the desired minimal perfect hash function. We can even get

$$h(w_j) = j$$

for $j = 0, 1, \ldots, m-1$. All this requires is a depth-first search of G.

Implementation

CHM report that on a Sun SPARCstation 2 they can generate a minimal perfect hash function for a set of $m = 2^{19}$ words in 33 seconds. Earlier algorithms required time which (at least in the worst case) was an exponentially increasing function of m, so could only handle very small m.

A small example

Suppose we want to construct a perfect hash function for the days of the week (Sunday, Monday, ..., Saturday). Since the days are uniquely identified by their first two characters, we can combine these to form an integer (e.g. $K = 26c_1 + c_2$). We have $k = m = 7$, $n = 3m = 21$. The random functions h_i might be of the form

$$h_i(K) = ((a_i K \bmod p_i) + b_i) \bmod n$$

where the constants a_i, b_i are chosen at random. The constants p_i are introduced so $h_i(K_1) \neq h_i(K_2)$ is possible even if $K_1 = K_2 \bmod n$.

Another way, which replaces arithmetic by table lookups, is to take

$$h_i(c_1 c_2 \cdots) = T_{i,1}(c_1) + T_{i,2}(c_2) + \cdots \bmod n\ ,$$

where the $T_{i,j}$ are randomly generated tables indexed by characters.

Definition of h_i by tables

Suppose we randomly construct the following tables (entries omitted are "don't cares").

c	$T_{1,1}$	$T_{2,1}$	c	$T_{1,2}$	$T_{2,2}$
F	4	3	a	19	0
M	5	3	e	1	1
S	0	18	h	1	7
T	1	0	o	5	4
W	3	1	r	5	4
			u	17	0

Then the h_i are given in columns 2–3 of the following table.

w	h_1	h_2	$g(h_1) + g(h_2) = h$
Sunday	17	18	$-2 + 2 = 0$
Monday	10	7	$-3 + 4 = 1$
Tuesday	18	0	$2 + 0 = 2$
Wednesday	4	2	$3 + 0 = 3$
Thursday	2	7	$0 + 4 = 4$
Friday	9	7	$1 + 4 = 5$
Saturday	19	18	$4 + 2 = 6$

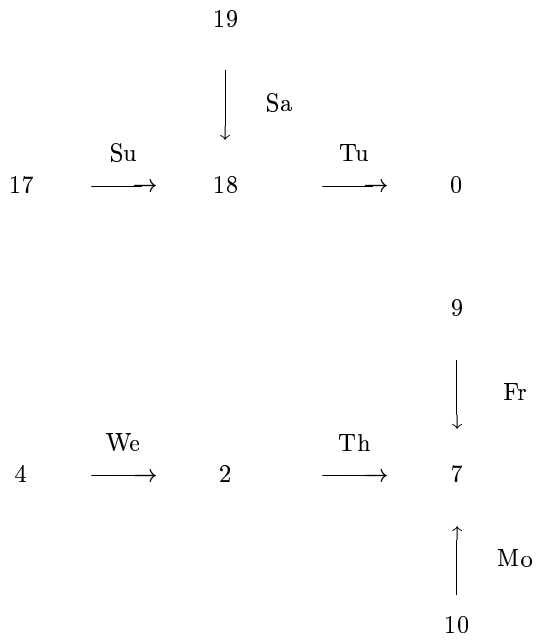
Determining G

G has two nontrivial connected components, containing vertices $\{2, 4, 7, 9, 10\}$ and $\{0, 17, 18, 19\}$. In each component we can assign one $g(w)$ value arbitrarily (say 0), then the other values are determined.

For example, we might assign $g(0) = g(2) = 0$. The constraint $g(2) + g(7) = 4$ implies $g(7) = 4$, then $g(9) + g(7) = 5$ implies $g(9) = 1$, similarly $g(10) = -3$, $g(4) = 3$, etc. Thus g can be defined by the table, where "?" means "don't care":

v	$g(v)$	v	$g(v)$	v	$g(v)$
0	0	1	?	2	0
3	?	4	3	5	?
6	?	7	4	8	?
9	1	10	-3	11	?
12	?	13	?	14	?
15	?	16	?	17	-2
18	2	19	4	20	?

The Graph G



Permutation Routing

A network G is a connected, undirected graph with N vertices $0, 1, \ldots, N-1$.

The permutation routing problem on G is: given a permutation π of the vertices, and a message (called a *packet*) on each vertex, route packet j from vertex j to vertex $\pi(j)$. It is assumed that at most one packet can traverse each edge in unit time, and that we want to minimise the time for the routing.

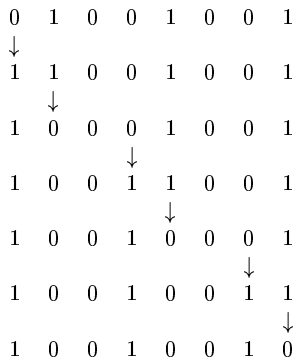
In practice we only want to consider *oblivious* algorithms, where the route taken by packet j depends only on $(j, \pi(j))$.

For simplicity, assume that the G is a d-dimensional hypercube, so $N = 2^d$. Similar results apply to other networks.

Example: Leading Bit Routing

A simple algorithm for routing packets on a hypercube chooses which edge to send a packet along by comparing the current address and the destination address and finding the highest order bit position in which these addresses differ.

For example, consider the bit-reversal permutation 01001001 → 10010010. Each "↓" corresponds to traversal of an edge in the hypercube.



Borodin and Hopcroft's bound

The following result says that there are no "uniformly good" deterministic algorithms for oblivious permutation routing:

Theorem: For any deterministic, oblivious permutation routing algorithm, there is a permutation π for which the routing takes $\Omega(\sqrt{N/d^3})$ steps.

Example: For the leading-bit routing algorithm, take π to be the bit-reversal permutation, i.e.

$$\pi(b_0 b_1 \ldots b_{d-1}) = b_{d-1} \ldots b_1 b_0 \ .$$

Suppose d is even. Then at least $2^{d/2}$ packets are routed through vertex 0. To prove this, consider the routing of

$$xx \ldots xx00 \ldots 00 \ ,$$

where there are at least $d/2$ trailing zeros.

Valiant and Brebner's algorithm

We can do much better with a probabilistic algorithm. Valiant suggested:

1. Choose a random mapping σ (not necessarily a permutation).
2. Route message j from vertex j to vertex $\sigma(j)$ using the leading bit algorithm (for $0 \le j < N$).
3. Route message j from vertex $\sigma(j)$ to vertex $\pi(j)$.

This seems crazy[7], but it works ! Valiant and Brebner prove:

Theorem: With probability greater than $1 - 1/N$, every packet reaches its destination in at most $14d$ steps.

Corollary: The expected number of steps to route all packets is less than $15d$.

[7] I don't know of any manufacturer who has been persuaded to implement it. Probably it would be hard to sell.

Pseudo-deterministic Algorithms

Some probabilistic algorithms use many independent random numbers, and because of the "law of large numbers" their performance is very predictable. One example is the *multiple-polynomial quadratic sieve* (MPQS) algorithm for integer factorisation.

Suppose we want to factor a large composite number N (not a perfect power). The key idea of MPQS is to generate a sufficiently large number of congruences of the form

$$y^2 = p_1^{\alpha_1} \cdots p_k^{\alpha_k} \bmod N,$$

where $p_1, \ldots, p_k$ are small primes in a precomputed "factor base", and y is close to $\sqrt{N}$. Many y are tried, and the "successful" ones are found efficiently by a sieving process.

Making some plausible assumptions, the expected run time of MPQS is

$$T = O(\exp(\sqrt{\ln N \ln\ln N})) \ .$$

MPQS Example

MPQS is currently the best general-purpose algorithm for factoring moderately large numbers N whose factors are in the range $N^{1/3}$ to $N^{1/2}$. For example, A. K. Lenstra and M. S. Manasse found

$$\begin{aligned} 3^{329} + 1 \quad = \quad & 2^2 \cdot 547 \cdot 16921 \cdot 256057 \cdot \\ & 36913801 \cdot 177140839 \cdot \\ & 1534179947851 \cdot p_{50} \cdot p_{67} \ , \end{aligned}$$

where the penultimate factor p_{50} is a 50-digit prime $24677078822840014266652779036768062918372697435241$, and the largest factor p_{67} is a 67-digit prime.

The computation used a network of workstations for "sieving", then a supercomputer for the solution of a very large sparse linear system.

A "random" 129-digit number (RSA129) was factored in a similar way in 1994 to win a $100 prize offered by Rivest, Shamir and Adleman in 1977. The current record is RSA140, found by a different method (Lecture 6).

Complexity Theory of Probabilistic Algorithms

Do probabilistic algorithms have an advantage over deterministic algorithms ? If we allow a small probability of error, the answer is **yes**, as we saw for the Charon example. If no error is allowed, the answer is (probably) **no**.

A. C. Yao considered probabilistic algorithms (modelled as decision trees) for testing properties P of undirected graphs (given by their adjacency matrices) on n vertices. He also considered deterministic algorithms which assume a given distribution of inputs (i.e. a distribution over the set of graphs with n vertices).

Definitions

Yao defines

randomized complexity $F_R(P)$ as an

infimum (over all possible algorithms) of a
maximum (over all graphs
with n vertices) of the
expected runtime.

and

distributional complexity $F_D(P)$ as a

supremum (over input distributions) of a
minimum (over all possible
deterministic algorithms) of the
average runtime.

Informally, $F_R(P)$ is how long the best probabilistic algorithm takes for testing P; and $F_D(P)$ is the average runtime we can always guarantee with a good deterministic algorithm, provided the distribution of inputs is known.

Yao's Result

Yao (1977) claims that $F_D(P) = F_R(P)$ follows from the minimax theorem of John von Neumann (1928). The minimax theorem is familiar from the theory of two-person zero-sum games.

Conclusion

Yao's result should not discourage the use of probabilistic algorithms – we have already given several examples where they out-perform known deterministic algorithms, and there are many similar examples.

Yao's computational model is very restrictive. Because n is fixed, table lookup is permitted, and the maximum complexity of any problem is $O(n^2)$.

Adleman and Gill's result

Less restrictive models have been considered by Adleman and Gill. Without going into details of the definitions, they prove:

Theorem: If a Boolean function has a randomised, polynomial-sized circuit family, then it has a deterministic, polynomial-sized circuit family.

There are two problems with this result:

- The deterministic circuit may be larger (by a factor of about n, the number of variables) than the original circuit.
- The transformation is not "uniform" – it can not be computed in polynomial time by a Turing machine. The proof of the theorem is by a counting argument applied to a matrix with 2^n rows, so it is not constructive in a practical sense.

The Class RP

We can formalise the notion of a probabilistic algorithm and define a class RP of languages L such that $x \in L$ is *accepted* by a probabilistic algorithm in polynomial time with probability $p \geq 1/2$ say[8], but $x \notin L$ is never accepted. Clearly

$$P \subseteq RP \subseteq NP,$$

where P and NP are the well-known classes of problems which are accepted in polynomial time by deterministic and nondeterministic (respectively) algorithms.

It is plausible that

$$P \subset RP \subset NP,$$

but this would imply that $P \neq NP$, so it is a difficult question.

[8] Any fixed value in $(0, 1)$ can be used in the definition.

Perfect Parties

$\cdots$ or, the answer is 42, what is the question ?

Because people at parties tend to cluster in groups of five, we consider a party to be *imperfect* if there are five people who are mutual acquaintances, or five who are mutual strangers. A *perfect* party is one which is not imperfect.

Clearly, many people are interested in the size of the largest perfect party.

B. McKay (ANU) and S. Radziszowski (Rochester) have performed a probabilistic computation which shows that, with high probability, the largest perfect party has 42 people.

Generalisation – Ramsey Numbers

$R(s,t)$ is the smallest n such that each graph on n or more vertices has a clique of size s *or* an independent set of size t.

Examples: $R(3,3) = 6$, $R(4,4) = 18$, $R(4,5) = 25$, and $43 \leq R(5,5) \leq 49$.

Perfect party organisers would like to know $R(5,5) - 1$ $(= 42\ ?)$

The Computation

A $(5,5,n)$-graph is a graph with n vertices, no clique of size 5, and no independent set of size 5. There are 328 known $(5,5,42)$-graphs, not counting complements as different. McKay *et al* generated 5812 $(5,5,42)$-graphs using simulated annealing, starting at random graphs. All 5812 turned out to be known.

If there were any more $(5,5,42)$-graphs, and if the simulated annealing process is about equally likely to find any $(5,5,42)$-graph[9], then another such graph would have been found with probability greater than

$$0.99999998$$

Thus, there is convincing evidence that all $(5,5,42)$-graphs are known. None of these graphs can be extended to $(5,5,43)$-graphs. Thus, it is very unlikely that such a graph exists, and it is very likely that

$$R(5,5) - 1 = 42$$

[9] There is no obvious way to prove this.

A Rigorous Proof ?

A rigorous proof that $R(5,5) - 1 = 42$ would take thousands of years of computer time[10], so the probabilistic argument is the best that is feasible at present, unless we can get time on *Deep Thought*.

[10] Based on the fact that it took seven years of Sparcstation time to show that $R(4,5) = 25$.

Omissions

We did not have time to mention applications of randomness to algorithms for:

- sorting and selection,
- computer security,
- public-key cryptography,
- computational geometry,
- load-balancing,
- collision avoidance,
- online algorithms,
- optimisation,
- numerical integration,
- graphics and virtual reality,
- quantum computing
- avoiding degeneracy, and many other problems.

Another Omission

We did not discuss algorithms for generating pseudo-random numbers – that would require another talk.

> Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.
>
> *John von Neumann, 1951*

Conclusion

- Probabilistic algorithms are useful.
- They are often simpler and use less space than deterministic algorithms.
- They can also be faster, if we are willing to live with a minute probability of error.

Some Open Problems

- Give good lower bounds for the complexity of probabilistic algorithms (with and without error) for interesting problems.
- Show how to generate independent random samples from interesting structures (e.g. finite groups defined by relations, various classes of graphs, ...) to provide a foundation for probabilistic algorithms on these structures.
- Consider the effect of using pseudo-random numbers instead of genuinely random numbers.
- Extend Yao's results to a more realistic model of computation.
- Give a uniform variant of the Adleman-Gill theorem.
- Show that $P \neq RP$ (hard).

References

[1] L. M. Adleman, On distinguishing prime numbers from composite numbers (extended abstract), *Proc. IEEE Symp. Found. Comp. Sci.* 21 (1980), 387–406.

[2] L. M. Adleman and M. A. Huang, Recognizing primes in random polynomial time, *Proc. Nineteenth Annual ACM Symposium on the Theory of Computing*, ACM, New York, 1987, 462–469.

[3] S. L. Anderson, Random number generators on vector supercomputers and other advanced architectures, *SIAM Review* 32 (1990), 221-251.

[4] P. van Emde Boas, Machine models, computational complexity and number theory, in [33], 7–42.

[5] B. Bollobás, *Random Graphs*, Academic Press, New York, 1985.

[6] A. Borodin and J. E. Hopcroft, Routing, merging, and sorting on parallel models of computation, *J. Computer and System Sciences* 30 (1985), 130–145.

[7] R. P. Brent, Parallel algorithms for integer factorisation, in *Number Theory and Cryptography* (edited by J. H. Loxton), Cambridge University Press, 1990.

[8] R. P. Brent, Vector and parallel algorithms for integer factorisation, *Proc. Third Australian Supercomputer Conference*, Melbourne, 1990.

[9] R. P. Brent, Factorization of the tenth Fermat number, *Math. Comp.* 68 (1999), 429–451.

[10] G. Buffon, Essai d'arithmétique morale, *Supplément à l'Histoire Naturelle* 4, 1777.

[11] T. R. Caron and R. D. Silverman, Parallel implementation of the quadratic sieve, *J. Supercomputing* 1 (1988), 273–290.

[12] Z. J. Czech, G. Havas and B. S. Majewski, An optimal algorithm for generating minimal perfect hash functions, *Information Processing Letters* 43 (1992), 257–264.

[13] P. Erdös and J. Spencer, *The Probabilistic Method in Combinatorics*, Academic Press, New York, 1974.

[14] P. Erdös and A. Renyi, On random graphs, I, *Publicationes Mathematicae* 6 (1959), 290–297.

[15] R. W. Floyd and R. L. Rivest, Expected time bounds for selection, *Comm. ACM* 18 (1975), 165–172.

[16] R. Freivalds, Fast probabilistic algorithms, in *Mathematical Foundations of Computer Science* (Lecture Notes in Computer Science, 74), Springer-Verlag, Berlin, 1979.

[17] J. Gill, Computational complexity of probabilistic Turing machines, *SIAM J. Computing* 6 (1977), 675–695.

[18] S. Goldwasser and J. Kilian, Almost all primes can be quickly certified, *Proc. 18th Annual ACM Symposium on Theory of Computing*, 1986, 316–329.

[19] R. L. Graham, B. L. Rothschild and J. H. Spencer, *Ramsey Theory*, John Wiley, New York, 1980.

[20] R. M. Karp, The probabilistic analysis of some combinatorial search algorithms, in [54], 1–19.

[21] R. M. Karp, An introduction to randomized algorithms, *Discrete Applied Mathematics* 34 (1991), 165–201.

[22] R. M. Karp and M. O. Rabin, Efficient randomized pattern-matching algorithms, *IBM J. Research and Development* 31 (1987), 249–260.

[23] R. M. Karp and V. Ramachandran, Parallel algorithms for shared memory machines, in [27], 869–941.

[24] D. E. Knuth, *The Art of Computer Programming*, Vol. 2, 2nd edition, Addison-Wesley, Menlo Park, 1981, §4.5.4.

[25] D. E. Knuth, *The Art of Computer Programming*, Vol. 3, Addison-Wesley, Menlo Park, 1973.

[26] K. de Leeuw, E. F. Moore, C. E. Shannon and N. Shapiro, Computability by probabilistic machines, in *Automata Studies* (C. E. Shannon and J. McCarthy, eds.), Princeton Univ. Press, Princeton, NJ, 1955, 183–212.

[27] J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, Elsevier, Amsterdam, 1990.

[28] A. K. Lenstra and H. W. Lenstra (editors), *The Development of the Number Field Sieve*, Lecture Notes in Mathematics 1554, Springer-Verlag, Berlin, 1993.

[29] A. K. Lenstra, H. W. Lenstra, Jr., M. S. Manasse and J. M. Pollard The factorization of the ninth Fermat number, *Mathematics of Computation* 61 (1993), 319–349.

[30] A. K. Lenstra and H. W. Lenstra, Jr., Algorithms in number theory, in [27], 675–715.

[31] H. W. Lenstra, Jr., Primality testing, in [33], 55–77.

[32] H. W. Lenstra, Jr., Factoring integers with elliptic curves, *Annals of Math.* (2) 126 (1987), 649–673.

[33] H. W. Lenstra, Jr. and R. Tijdeman (editors), *Computational Methods in Number Theory*, I Math. Centre Tracts 154, Amsterdam, 1982.

[34] B. D. McKay and S. P. Radziszowski, A new upper bound for the Ramsey number $R(5,5)$, *Australasian J. Combinatorics* 5 (1991), 13–20.

[35] B. D. McKay and S. P. Radziszowski, Linear programming in some Ramsey problems, *J. Combinatorial Theory, Ser. B* 61 (1994), 125–132.

[36] G. L. Miller, Riemann's hypothesis and tests for primality, *J. Comp. System Sci.* 13 (1976), 300–317.

[37] L. Monier, Evaluation and comparison of two efficient probabilistic primality testing algorithms, *Theoret. Comput. Sci.* 12 (1980), 97–108.

[38] R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, 1995.

[39] K. Mulmuley, *Computational Geometry: An Introduction Through Randomized Algorithms*, Prentice-Hall, New York, 1993.

[40] J. von Neumann, Zur Theorie der Gesellschaftsspiele, *Math. Annalen* 100 (1928), 295–320. Reprinted in *John von Neumann Collected Works* (A. H. Taub, editor), Pergamon Press, New York, 1963, Vol. 6, 1–26.

[41] J. von Neumann and O. Morgenstern, *Theory of Games and Economic Behavior*, Princeton Univ. Press, Princeton, NJ, 1953.

[42] C. Pomerance, J. W. Smith and R. Tuler, A pipeline architecture for factoring large integers with the quadratic sieve algorithm, *SIAM J. on Computing* 17 (1988), 387–403.

[43] V. Pratt, Every prime has a succinct certificate, *SIAM J. Computing* 4 (1975), 214–220.

[44] M. O. Rabin, Probabilistic automata, *Information and Control* 6 (1963), 230–245.

[45] M. O. Rabin, Probabilistic algorithms, in [54], 21–39.

[46] M. O. Rabin, Complexity of computations (1976 Turing Award Lecture), *Comm. ACM* 20 (1977), 625–633. *Corrigendum ibid* 21 (1978), 231.

[47] M. O. Rabin, Probabilistic algorithms for testing primality, *J. Number Theory* 12 (1980), 128–138.

[48] M. O. Rabin and Shallit, Randomized algorithms in number theory, *Comm. Pure Appl. Math.* 39 (1986).

[49] R. L. Rivest, A. Shamir and L. Adleman, A method for obtaining digital dignatures and public-key cryptosystems, *Comm. ACM* 21 (1978), 120–126.

[50] A. Shamir, Factoring numbers in $O(\log n)$ arithmetic steps, *Information Processing Letters* 8 (1979), 28–31.

[51] Peter W. Shor, Polynomial time algorithms for factorization and discrete logarithms on a quantum computer, *SIAM J. Computing* 26, 5 (Oct 1997).

[52] R. D. Silverman, The multiple polynomial quadratic sieve, *Mathematics of Computation* 48 (1987), 329–339.

[53] R. Solovay and V. Strassen, Fast Monte Carlo test for primality, *SIAM J. on Computing* 6 (1977), 84–85; *erratum* 7 (1978), 118.

[54] J. F. Traub (editor), *Algorithms and Complexity*, Academic Press, New York, 1976.

[55] L. G. Valiant and G. J. Brebner, Universal schemes for parallel communication, *Proc. 13th Annual ACM Symposium on the Theory of Computing*, ACM, New York, 1981, 263–277.

[56] A. C. Yao, Probabilistic computations: towards a unified measure of complexity, *Proc. 18th Annual Symposion on Foundations of Computer Science*, IEEE, New York, 1977, 222–227.

Lecture 5

Revisiting the Binary Euclidean Algorithm*

*Six lectures on Algorithms, Trinity term 1999.
 lec05

Summary

The binary Euclidean algorithm is a variant of the classical Euclidean algorithm. It avoids divisions and multiplications, except by powers of two, so is potentially faster than the classical algorithm on a binary machine. In this lecture I describe the classical and binary algorithms, and compare their worst case and average case behaviour. In particular, I correct some small but significant errors in the literature, discuss some recent results of Brigitte Vallée, and describe a numerical computation which verifies Vallée's conjecture to 44 decimal places.

Outline

- The classical Euclidean algorithm
  - The algorithm
  - Worst case
  - Continued fractions
  - Continuous model of Gauss
  - Results of Kuz'min, Lévy, et al
- The binary Euclidean algorithm
  - The algorithm
  - Worst case
  - Continuous model
  - Conjectured/empirical results
  - An error in the literature
  - Useful operators
  - Recent results of Vallée
  - Confirmation of a conjecture
  - Open problems

Notation

$\lg(x)$ denotes $\log_2(x)$.

$\mathrm{Val}_2(u)$ denotes the dyadic valuation of the positive integer u, i.e. the greatest integer j such that $2^j \mid u$.

The Classical Euclidean Algorithm

The Euclidean algorithm finds the greatest common divisor (GCD) of two positive integers m and n. It is one of the best known of all algorithms. Knuth (§1.1) gives the algorithm as:

Algorithm E

E1. $r \leftarrow m \bmod n$

E2. If $r = 0$ terminate with n as the result.

E3. Set $m \leftarrow n$, $n \leftarrow r$, and go to E1

Of course, Euclid did not describe the algorithm in this way. In fact, it is not quite clear what Euclid intended at step E1. For a translation of Euclid's description, see Knuth, §4.5.2.

The algorithm was probably known about 200 years before Euclid. Nevertheless, we shall call Algorithm E the "classical Euclidean algorithm" or just the "classical algorithm".

One-line Version

while $n \neq 0$ do $\binom{m}{n} \leftarrow \binom{n}{m \bmod n}$; return m.

Relation to Continued Fractions

Assume $m \geq n$. The first execution of step E1 gives

$$m = q \times n + r$$

where q is the quotient and r is the remainder on division of m by n. By definition,

$$0 \leq r < n.$$

Define $n_0 = m$, $n_1 = n$, $n_2 = r = m \bmod n$. Suppose step E3 is executed k times. Then

$$n_j = q_j \times n_{j+1} + n_{j+2}$$

holds for $j = 0, 1, \ldots, k$ and $n_{k+2} = 0$. We can write this as

$$\frac{n_j}{n_{j+1}} = q_j + 1 \Big/ \frac{n_{j+1}}{n_{j+2}}$$

so, in the usual notation for continued fractions,

$$\frac{m}{n} = q_0 + 1/q_1 + 1/q_2 + \cdots + 1/q_k.$$

The Worst Case

We have seen that there is an intimate connection between the classical Euclidean algorithm for computing $\mathrm{GCD}(m, n)$ and the continued fraction expansion of the rational number m/n.

The worst case for the classical algorithm occurs when all the quotients q_j are 1. This happens when the inputs are consecutive Fibonacci numbers. (These are defined by $F_0 = 0$, $F_1 = 1$, $F_{j+2} = F_{j+1} + F_j$ for $j \geq 0$.)

For example,

$$(m, n) = (F_6, F_5) = (8, 5) \to (5, 3) \to (3, 2) \to (2, 1).$$

Since $F_k \sim \rho^k/\sqrt{5}$, where $\rho = \frac{\sqrt{5}+1}{2} \simeq 1.618$, the worst case number of iterations of the classical algorithm is

$$\log_\rho N + O(1),$$

where $N = \max(m, n)$.

The Continuous Model of Gauss

To investigate the *average* behaviour of the classical algorithm, we can restrict attention to the case $0 < m < n$ (so $q_0 = 0$). We assume that $n = N$ is large and that m is equally likely to take the values $\{1, 2, \ldots, N-1\}$. Thus, m/n will be approximately uniformly distributed in $(0, 1)$, and the sequence of quotients $q_1, q_2, \ldots$ will "look like" the quotients in the continued fraction expansion of a uniformly distributed random number.

Gauss's Recurrence

Suppose $x_0 \in (0, 1)$, and x_0 has a continued fraction expansion

$$x_0 = 1/q_1 + 1/q_2 + \ldots + 1/(q_k + x_{k+1}),$$

where $q_1, \ldots, q_k$ are positive integers and $x_{k+1} \in (0, 1)$.

We can express the probability distribution function $F_{k+1}(x)$ of x_{k+1} in terms of the distribution function $F_k(x)$ of x_k. Using the fact that $1/x_k = q_k + x_{k+1}$, we see that

$$\begin{aligned} F_{k+1}(x) &= \Pr(0 \le x_{k+1} \le x) \\ &= \sum_{q \ge 1} \Pr(q \le 1/x_k \le q + x) \\ &= \sum_{q \ge 1} \Pr\left(\frac{1}{q+x} \le x_k \le \frac{1}{q}\right) \\ &= \sum_{q \ge 1} \left(F_k\left(\frac{1}{q}\right) - F_k\left(\frac{1}{q+x}\right)\right) \end{aligned}$$

The Limiting Distribution

To investigate average case behaviour in the continuous model, we assume $F_0(x) = x$ for $x \in (0, 1)$ (the uniform distribution of m/n), and consider $F_k(x)$ as $k \to \infty$.

If we *assume* that a limit distribution $F(x) = \lim_{k \to \infty} F_k(x)$ exists, then $F(x)$ satisfies the functional equation

$$F(x) = \sum_{q \ge 1} \left(F\left(\frac{1}{q}\right) - F\left(\frac{1}{q+x}\right)\right).$$

Gauss noticed the simple solution

$$F(x) = \lg(1 + x).$$

However, Gauss was not able to prove convergence. It was eventually proved by Kuz'min (1928). Sharper results were proved by Lévy (1929), Wirsing (1974), and Babenko (1978).

Babenko's Theorem

The definitive result, due to Babenko (1978), is

$$F_k(x) = \lg(1 + x) + \sum_{j \ge 2} \lambda_j^k \Psi_j(x),$$

where $|\lambda_2| > |\lambda_3| \ge |\lambda_4| \ge \cdots$,

$$-\lambda_2 = \lambda = 0.3036630028\cdots$$

is called *Wirsing's constant*, and the $\Psi_j(x)$ are certain analytic functions (see Knuth, §4.5.3 for more details).

The expected number of iterations is

$$\sim \frac{12 \ln 2}{\pi^2} \ln N \sim 0.8428 \ln N \sim 0.5842 \lg N$$

which can be compared with

$$\sim \log_\rho N \sim 2.0781 \ln N \sim 1.4404 \lg N$$

for the worst case.

Remainder of the Lecture

In the time remaining, I will describe what progress has been made towards a similar analysis of the *binary* Euclidean algorithm.

The Binary Euclidean Algorithm

The idea of the *binary* Euclidean algorithm is to avoid the "division" operation $r \leftarrow m \bmod n$ of the classical algorithm, but retain $O(\log N)$ worst (and average) case.

We assume that the algorithm is implemented on a binary computer so division by a power of two is easy. In particular, we assume that the "shift right until odd" operation

$$u \leftarrow u/2^{\mathrm{Val}_2(u)}$$

or equivalently

$$\text{while even}(u) \text{ do } u \leftarrow u/2$$

can be performed in constant time (although time $O(\mathrm{Val}_2(u))$ would be sufficient).

Definition of the Binary Algorithm

There are several almost equivalent ways to define the algorithm. For simplicity, let us assume that u and v are *odd* positive integers. Following is a simplified version of the algorithm given in Knuth, §4.5.2.

Algorithm B

B1. $t \leftarrow |u - v|$;
if $t = 0$ terminate with result u

B2. $t \leftarrow t/2^{\mathrm{Val}_2(t)}$

B3. if $u \geq v$ then $u \leftarrow t$ else $v \leftarrow t$;
go to B1.

History

The binary Euclidean algorithm is attributed to Silver and Terzian (unpublished, 1962) and Stein (1967). However, it seems to go back almost as far as the classical Euclidean algorithm. Knuth (§4.5.2) quotes a translation of a first-century AD Chinese text *Chiu Chang Suan Shu* on how to reduce a fraction to lowest terms:

> If halving is possible, take half.
>
> Otherwise write down the denominator and the numerator, and subtract the smaller from the greater.
>
> Repeat until both numbers are equal.
>
> Simplify with this common value.

This looks very much like Algorithm B !

Another Formulation

It will be useful to rewrite Algorithm B in the following equivalent form (using pseudo-Pascal):

Algorithm V { Assume $u \le v$ }

while $u \neq v$ **do**
 begin
 while $u < v$ **do**
  begin
  $j \leftarrow \mathrm{Val}_2(v-u)$;
  $v \leftarrow (v-u)/2^j$;
  end;
 $u \leftrightarrow v$;
 end;
return u.

Continued Fractions

Vallée [15] shows a connection between Algorithm V and continued fractions of a certain form:

$$\frac{u}{v} = 1/a_1 + 2^{k_1}/a_2 + 2^{k_2}/\ldots/a_r + 2^{k_r} ,$$

where a_j is odd, $k_j > 0$, and $0 < a_j < 2^{k_j}$.

Example

Consider $u = 9$, $v = 55$. The inner loop of Algorithm V finds

$$\begin{aligned} 55 &= 9 + 2 \times 23 \\ &= 9 + 2 \times (9 + 2 \times 7) \\ &= 3 \times 9 + 4 \times 7 \end{aligned}$$

and on the next iteration

$$9 = 7 + 2 \times 1$$

so

$$\frac{55}{9} = 3 + 4 \times \frac{7}{9} ,$$

$$\frac{9}{7} = 1 + \frac{2}{7} ,$$

and finally

$$\frac{9}{55} = \frac{1}{3 + \frac{4}{1 + \frac{2}{3+4}}}$$

which we write as

$$\frac{9}{55} = 1/3 + 4/1 + 2/3 + 4 .$$

Vallée's Results – More Details

Algorithm V has two nested loops. The outer loop exchanges u and v. Between two exchanges, the inner loop performs a sequence of subtractions and shifts which can be written as

$$\begin{aligned} v &\leftarrow u + 2^{b_1} v_1; \\ v_1 &\leftarrow u + 2^{b_2} v_2; \\ &\cdots \\ v_{m-1} &\leftarrow u + 2^{b_m} v_m \end{aligned}$$

with $v_m < u$.

If $x_0 = u/v$ at the beginning of an inner loop, the effect of the inner loop followed by an exchange is the rational $x_1 = v_m/u$ defined by

$$x_0 = \frac{1}{a + 2^k x_1} ,$$

where a is an odd integer given by

$$a = 1 + 2^{b_1} + 2^{b_1+b_2} + \cdots + 2^{b_1+\cdots b_{m-1}} ,$$

and the exponent k is given by

$$k = b_1 + \cdots b_m .$$

Thus, the rational u/v, for $1 \le u < v$, has a unique *binary continued fraction expansion* of the form

$$\frac{u}{v} = \cfrac{1}{a_1 + \cfrac{2^{k_1}}{a_2 + \cfrac{2^{k_2}}{\ddots \; + \cfrac{2^{k_{r-1}}}{a_r + 2^{k_r}}}}}$$

Vallée studies three parameters related to this continued fraction

1. The height or the depth (i.e. the number of exchanges) r.
2. The total number of operations necessary to obtain the expansion; if $p(a)$ denotes the number of "1"s in the binary expansion of the integer a, it is equal to $p(a_1) + p(a_2) + \cdots + p(a_r)$.
3. The sum of exponents of 2 in the numerators of the binary continued fraction, $k_1 + \cdots + k_r$.

Vallée's Theorems

Vallée's main results give the average values of the three parameters above: the average values are asymptotically $A_i \log N$ for certain constants A_1, A_2, A_3 related to the spectral properties of an operator $\mathcal{V}_2$ (to be defined later).

The Worst Case

At step B1, u and v are odd, so t is even. Thus, step B2 always reduces t by at least a factor of two. Using this fact, it is easy to show that step B3 is executed at most

$$\lfloor \lg(u+v) \rfloor$$

times (Knuth, exercise 4.5.2.37). Thus, if $N = \max(u, v)$, step B3 is executed at most

$$\lg(N) + O(1)$$

times.

Remark

Even if step B2 is replaced by single-bit shifts

$$\text{while even}(t) \text{ do } t \leftarrow t/2$$

the overall worst case is still $O(\log N)$.

Extended Binary Algorithm

It is possible to give an *extended* binary GCD algorithm which computes multipliers α and β such that

$$\alpha u + \beta v = \text{GCD}(u, v)$$

(Bojanczyk and Brent [2], 1987).

Systolic Binary Algorithm

For hardware implementation, there is a systolic array variant of the binary GCD algorithm (Brent and Kung [5], 1985). This takes time $O(\log N)$ using $O(\log N)$ 1-bit processors and nearest-neighbour communication. The overall bit-complexity is $O(\log N)^2$.

A Heuristic Continuous Model

To analyse the expected behaviour of Algorithm B, we can follow what Gauss did for the classical algorithm. This was first attempted in my 1976 paper [3] and there is a summary in Knuth (Vol. 2, *third* edition, §4.5.2).

Assume that the initial inputs u_0, v_0 to Algorithm B are uniformly and independently distributed in (0, N), apart from the restriction that they are odd. Let (u_n, v_n) be the value of (u, v) after n iterations of step B3.

Let

$$x_n = \frac{\min(u_n, v_n)}{\max(u_n, v_n)}$$

and let $F_n(x)$ be the probability distribution function of x_n (in the limit as $N \to \infty$). Thus $F_0(x) = x$ for $x \in [0, 1]$.

We assume that $\mathrm{Val}_2(t)$ takes the value k with probability 2^{-k} at step B2. (Vallée does not make this assumption – we will discuss her rigorous analysis later.)

The Recurrence for F_n

Consider the effect of steps B2 and B3. We can assume that $u > v$ so $t = u - v$. If $\mathrm{Val}_2(t) = k$ then $X = v/u$ is transformed to

$$\begin{aligned} X' &= \min\left(\frac{u-v}{2^k v}, \frac{2^k v}{u-v}\right) \\ &= \min\left(\frac{1-X}{2^k X}, \frac{2^k X}{1-X}\right) . \end{aligned}$$

It follows that $X' < x$ iff

$$X < \frac{1}{1+2^k/x} \text{ or } X > \frac{1}{1+2^k x} .$$

Thus, the recurrence for $G_n(x) = 1 - F_n(x)$ is

$$G_{n+1}(x) = \sum_{k\geq 1} 2^{-k}\left(G_n\left(\frac{1}{1+2^k/x}\right) - G_n\left(\frac{1}{1+2^k x}\right)\right) ,$$

and $G_0(x) = 1 - x$ for $x \in [0, 1]$.

The Recurrence for f_n

Differentiating the recurrence for G_n we obtain (formally) a recurrence for the probability density $f_n(x) = F_n'(x) = -G_n'(x)$:

$$\begin{aligned} f_{n+1}(x) &= \sum_{k\geq 1}\left(\frac{1}{x+2^k}\right)^2 f_n\left(\frac{x}{x+2^k}\right) \\ &+ \sum_{k\geq 1}\left(\frac{1}{1+2^k x}\right)^2 f_n\left(\frac{1}{1+2^k x}\right) . \end{aligned}$$

Operator Notation

The recurrence for f_n may be written as

$$f_{n+1} = \mathcal{B}_2 f_n,$$

where the operator $\mathcal{B}_2$ is the case $s = 2$ of a more general operator $\mathcal{B}_s$ which will be defined later.

Conjectured and Empirical Results

In my 1976 paper [3] I gave numerical and analytic evidence that $F_n(x)$ converges to a limiting distribution $F(x)$ as $n \to \infty$, and that $f_n(x)$ converges to the corresponding probability density $f(x) = F'(x)$ (note that $f = \mathcal{B}_2 f$ so f is a "fixed point" of the operator $\mathcal{B}_2$). Assuming the existence of F, it is shown in [3] that the expected number of iterations of Algorithm B is $\sim K \lg N$ as $N \to \infty$, where $K = 0.705\ldots$ is a constant defined by

$$K = \ln 2 / E_\infty ,$$

and

$$E_\infty = \ln 2 + \int_0^1 \left(\sum_{k=2}^{\infty}\left(\frac{1-2^{-k}}{1+(2^k-1)x}\right) - \frac{1}{2(1+x)}\right) F(x)\,dx .$$

A Simplification

We can simplify the expression for K to obtain

$$K = 2/b \ ,$$

where

$$b = 2 - \int_0^1 \lg(1-x) f(x)\ dx \ .$$

Using integration by parts we obtain an equivalent expression

$$b = 2 + \frac{1}{\ln 2} \int_0^1 \frac{1-F(x)}{1-x}\ dx \ .$$

For the proofs, see Knuth, third edition, §4.5.2.

An Error in the Literature

In (Brent, 1976) I claimed that, for all $n \geq 0$ and $x \in (0, 1]$,

$$F_n(x) = \alpha_n(x) \lg(x) + \beta_n(x) \ ,$$

where $\alpha_n(x)$ and $\beta_n(x)$ are analytic and regular in the disk $|x| < 1$. However, *this is incorrect*, even in the case $n = 1$.

The error appeared to go unnoticed until 1997, when Don Knuth was revising Volume 2 in preparation for publication of the third edition. Knuth computed the constant K using recurrences for the analytic functions $\alpha_n(x)$ and $\beta_n(x)$, and I computed K directly using the defining integral and recurrences for $F_n(x)$. Our computations disagreed in the 14th decimal place ! Knuth found

$$K = 0.70597\ 12461\ 01945\ 99986\cdots$$

but I found

$$K = 0.70597\ 12461\ 01916\ 39152\cdots$$

Some Detective Work

After a flurry of emails we tracked down the error. It was found independently, and at the same time (within 24 hours), by Flajolet and Vallée.

The source of the error is illustrated by Lemma 3.1 of my 1976 paper [3], which is wrong (and corrected in the solution to ex. 4.5.2.29 of Knuth, third edition).

The *Mellin transform* of a function $g(x)$ is defined by

$$g^*(s) = \int_0^\infty g(x) x^{s-1} dx \ .$$

If $f(x) = \sum_{k\geq 1} 2^{-k} g(2^k x)$ then the Mellin transform of f is

$$f^*(s) = \sum_{k\geq 1} 2^{-k(s+1)} g^*(s) = \frac{g^*(s)}{2^{s+1}-1} \ .$$

Under suitable conditions we can apply the Mellin inversion formula to obtain

$$f(x) = \frac{1}{2\pi i} \int_{c-i\infty}^{c+i\infty} f^*(s) x^{-s} ds \ .$$

Applying these results to $g(x) = 1/(1+x)$, whose Mellin transform is $g^*(s) = \pi / \sin \pi s$ when $0 < \mathcal{R}s < 1$, we find

$$f(x) = \sum_{k\geq 1} \frac{2^{-k}}{1+2^k x}$$

as a sum of residues of

$$\left(\frac{\pi}{\sin \pi s}\right) \frac{x^{-s}}{2^{s+1}-1}$$

for $\mathcal{R}s \leq 0$. This gives

$$f(x) = 1 + x \lg x + \frac{x}{2} + xP(\lg x) - \frac{2}{1}x^2 + \frac{4}{3}x^3 - \cdots \ ,$$

where

$$P(t) = \frac{2\pi}{\ln 2} \sum_{n=1}^{\infty} \frac{\sin 2n\pi t}{\sinh(2n\pi^2/\ln 2)} \ .$$

The "Wobbles" Caused by $P(t)$

$P(t)$ is a very small periodic function:

$$|P(t)| < 7.8 \times 10^{-12}$$

for real t. In [3, Lemma 3.1], the term $xP(\lg x)$ is omitted.

Essentially, we only considered poles on the real axis and ignored those at $s = -1 \pm 2\pi i n/\ln 2$, $n = 1, 2, \ldots$.

Because the residues at these poles are tiny (thanks to the sinh term in the denominator) numerical computations performed using single-precision floating-point arithmetic did not reveal the error.

An Analogy

Ramanujan made a similar error when he gave a formula for $\pi(x)$ (the number of primes $\le x$) which essentially ignored the residues of $x^s\zeta'(s)/\zeta(s)$ arising from zeros of $\zeta(s)$ off the real axis.

It is easier to work with

$$f(x) = \sum_{n=1}^{\infty} \frac{1}{n}\pi(x^{1/n})$$

than with $\pi(x)$. From $f(x)$ we can find $\pi(x)$ by Möbius inversion:

$$\pi(x) = \sum_{n=1}^{\infty} \frac{\mu(n)}{n} f(x^{1/n}) \ .$$

Riemann's formula

Riemann's explicit formula[1] for $f(x)$ is

$$f(x) = \mathrm{li}x - \sum_{\rho} \mathrm{li}x^{\rho} + \int_x^{\infty} \frac{dt}{(t^2-1)t\ln t} - \ln 2 \ .$$

The sum is over all the *complex* zeros ρ of the Riemann zeta function (summed in order of increasing $|\rho|$), and li$x$ is the logarithmic integral.

Ramanujan's error was essentially to ignore the sum over ρ.

[1] Stated by Riemann in 1859, and proved by Von Mangoldt in 1885.

Some Useful Operators

Operators $\mathcal{B}_s, \mathcal{U}_s, \widetilde{\mathcal{U}}_s, \mathcal{V}_s$, useful in the analysis of the binary Euclidean algorithm, are defined on suitable function spaces by

$$\mathcal{U}_s[f](x) = \sum_{k\ge 1} \left(\frac{1}{1+2^k x}\right)^s f\left(\frac{1}{1+2^k x}\right),$$

$$\widetilde{\mathcal{U}}_s[f](x) = \left(\frac{1}{x}\right)^s \mathcal{U}_s[f]\left(\frac{1}{x}\right),$$

$$\mathcal{B}_s = \mathcal{U}_s + \widetilde{\mathcal{U}}_s,$$

$$\mathcal{V}_s[f](x) = \sum_{k\ge 1} \sum_{\substack{a \text{ odd},\\ 0<a<2^k}} \left(\frac{1}{a+2^k x}\right)^s f\left(\frac{1}{a+2^k x}\right) \ .$$

In these definitions s is a complex variable, and the operators are called Ruelle operators [12]. They are linear operators acting on certain function spaces.

The case $s = 2$ is of particular interest. $\mathcal{B}_2$ encodes the effect of one iteration of the inner "while" loop of Algorithm V, and $\mathcal{V}_2$ encodes the effect of one iteration of the outer "while" loop.

Relations between Operators

$\mathcal{B}_2$ (denoted T) was introduced in my 1976 paper [3], and generalised to $\mathcal{B}_s$ by Vallée. $\mathcal{V}_s$ was introduced by Vallée. We shall call $\mathcal{B}_2$ the *binary Euclidean operator* and $\mathcal{V}_s$ *Vallée's operator*. Not surprisingly, the operators are related, as the following Lemma and Theorem show.

Lemma 1

$$\mathcal{V}_s = \mathcal{V}_s\widetilde{\mathcal{U}}_s + \mathcal{U}_s.$$

The following Theorem (which follows from the Lemma) gives a simple relationship between $\mathcal{B}_s$, $\mathcal{V}_s$ and $\mathcal{U}_s$.

Theorem 1

$$(\mathcal{V}_s - \mathcal{I})\mathcal{U}_s = \mathcal{V}_s(\mathcal{B}_s - \mathcal{I}).$$

Algorithmic interpretation

Algorithm V gives an interpretation of Lemma 1 in the case $s = 2$. If the input density of $x = u/v$ is $f(x)$ then execution of the inner "while" loop followed by the exchange of u and v transforms this density to $\mathcal{V}_2[f](x)$. However, by considering the first iteration of this loop (followed by the exchange if the loop terminates) we see that the transformed density is given by

$$\mathcal{V}_2\widetilde{\mathcal{U}}_2[f](x) + \mathcal{U}_2[f](x),$$

where the first term arises if $u < v$ without an exchange, and the second arises if an exchange occurs.

A Conjecture of Vallée

Let $\lambda = f(1)$, where f is the limiting probability density (conjectured to exist) as above. Vallée (see Knuth, third edition, §4.5.2(61)) conjectured that

$$\frac{\lambda}{b} = \frac{2\ln 2}{\pi^2},$$

or equivalently that

$$K = \frac{4\ln 2}{\pi^2\lambda}. \qquad (1)$$

Vallée proved the conjecture under the assumption that the operator $\mathcal{B}_s$ satisfies a certain spectral condition. We have verified the conjectures numerically to 44 decimal places.

Recent Results of Vallée

Using her operator $\mathcal{V}_s$, Vallée recently *proved* that

$$K = \frac{2\ln 2}{\pi^2 g(1)} \sum_{\substack{a \text{ odd},\\ a>0}} 2^{-\lfloor \lg a \rfloor} G\left(\frac{1}{a}\right)$$

where g is a nonzero fixed point of $\mathcal{V}_2$ (i.e. $g = \mathcal{V}_2 g \neq 0$) and $G(x) = \int_0^x g(t)\,dt$. This is yet another expression for K (the only one which has been proved).

Warning: G here is not the same as $G(x) = 1 - F(x)$! Unfortunately Knuth and Vallée use incompatible notation.

Because $\mathcal{V}_s$ can be proved to have nice spectral properties, the existence and uniqueness (up to scaling) of g can be proved rigorously.

Fixed Points of some Operators

It follows immediately from Theorem 1 that, if

$$g = \mathcal{U}_2 f,$$

then

$$(\mathcal{V}_2 - \mathcal{I})g = \mathcal{V}_2(\mathcal{B}_2 - \mathcal{I})f.$$

Thus, if f is a fixed point of the operator $\mathcal{B}_2$, then g is a fixed point of the operator $\mathcal{V}_2$. From the recent result of Vallée [15, Prop. 4] we know that $\mathcal{V}_2$, acting on a certain Hardy space $\mathcal{H}^2(\mathcal{D})$, has a unique positive dominant simple eigenvalue 1, so g must be (a constant multiple of) the corresponding eigenfunction (provided $g \in \mathcal{H}^2(\mathcal{D})$). Also, from the definitions of $\mathcal{B}_2$ and $\mathcal{V}_2$, we have

$$\lambda = f(1) = 2g(1) = 2\sum_{k\ge1}\left(\frac{1}{1+2^k}\right)^2 f\left(\frac{1}{1+2^k}\right),$$

which is useful for proving the consistency of two of the expressions for K given above.

Numerical Results

Using an improvement of the "discretization method" of [3], and the MP package with the equivalent of more than 50 decimal places (50D) working precision, we computed the limiting probability density f, then K, $\lambda = f(1)$, and $K\lambda$. The results were

$$\begin{aligned} K &= 0.7059712461\,0191639152\,9314135852\,8817666677 \\ \lambda &= 0.3979226811\,8831664407\,6707161142\,6549823098 \\ K\lambda &= 0.2809219710\,9073150563\,5754397987\,9880385315 \end{aligned}$$

These are believed to be correctly rounded values.

One of Vallée's conjectures is that

$$K\lambda = 4\ln 2/\pi^2\ .$$

The computed value of $K\lambda$ agrees with $4\ln 2/\pi^2$ to 40 decimals (in fact to 44 decimals).

Conclusion and Open Problems

Since Vallée's recent work [14, 15], analysis of the average behaviour of the binary Euclidean algorithm has a rigorous foundation. However, some interesting open questions remain.

For example, does the binary Euclidean operator $\mathcal{B}_2$ have a unique positive dominant simple eigenvalue 1? Vallée [15, Prop. 4] has proved the corresponding result for her operator $\mathcal{V}_2$.

In order to estimate the speed of convergence of f_n to f (assuming f exists), we need more information on the spectrum of $\mathcal{B}_2$. What can be proved ? Preliminary numerical results indicate that the sub-dominant eigenvalue(s) are a complex conjugate pair:

$$\lambda_2 = \overline{\lambda}_3 = 0.1735 \pm 0.0884i\ ,$$

with $|\lambda_2| = |\lambda_3| = 0.1948$ to 4D.

Acknowledgements

Thanks to Don Knuth for encouraging me to correct and extend my 1976 results for the third edition of *Seminumerical Algorithms*, to Brigitte Vallée for sharing her conjectures and results with me, and to Philippe Flajolet for his notes on Mellin transforms.

References

[1] B. C. Berndt, *Ramanujan's Notebooks*, Parts I-V, Springer-Verlag, New York, 1985, ...

[2] Adam W. Bojanczyk and Richard P. Brent, A systolic algorithm for extended GCD computation, *Comput. Math. Applic.* 14 (1987), 233–238.

[3] Richard P. Brent, Analysis of the Binary Euclidean Algorithm, *New Directions and Recent Results in Algorithms and Complexity*, (J. F. Traub, editor), Academic Press, New York, 1976, 321–355.

[4] Richard P. Brent and H. T. Kung, Systolic VLSI arrays for linear-time GCD computation, in *VLSI 83* (F. Anceau and E. J. Aas, editors), North-Holland, Amsterdam, 1983, 145–154.

[5] Richard P. Brent and H. T. Kung, A systolic VLSI array for integer GCD computation, in *ARITH-7, Proc. Seventh Symposium on Computer Arithmetic* (K. Hwang, editor), IEEE/CS Press, 1985.

[6] Richard P. Brent, *Further analysis of the Binary Euclidean algorithm*, in preparation.

[7] Hervé Daudé, Philippe Flajolet and Brigitte Vallée, An analysis of the Gaussian Algorithm for Lattice Reduction, *Proc. ANTS'94, Lecture Notes in Computer Science*, Vol. 877, Springer-Verlag, 1994, 144–158. Extended version in *Combinatorics, Probability and Computing* **6** (1997), 397–433.

[8] Philippe Flajolet and Brigitte Vallée, Continued Fraction Algorithms, Functional Operators and Structure Constants, *Theoretical Computer Science* **194** (1998), 1–34.

[9] Carl F. Gauss, Brief an Laplace vom 30 Jan. 1812, *Carl Friedrich Gauss Werke*, Bd. $\mathbf{X}_1$, Gottingen, 371–374.

[10] G. H. Hardy, *Ramanujan: Twelve Lectures on Subjects Suggested by his Life and Work*, Cambridge University Press, Cambridge, 1940.

[11] Donald E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms* (third edition). Addison-Wesley, Menlo Park, 1997.

[12] David Ruelle, *Thermodynamic formalism*, Addison Wesley, Menlo Park, 1978.

[13] Brigitte Vallée, Opérateurs de Ruelle–Mayer généralisés et analyse des algorithmes de Gauss et d'Euclide, *Acta Arithmetica* **81** (1997), 101–144.

[14] Brigitte Vallée, The complete analysis of the Binary Euclidean Algorithm, *Proc. ANTS'98*, *Lecture Notes in Computer Science*, Vol. 1423, Springer-Verlag, 1998, 77–94.

[15] Brigitte Vallée. Dynamics of the Binary Euclidean Algorithm: functional analysis and operators, manuscript, Feb. 1998 (to appear in *Algorithmica*).
`http://www.info.unicaen.fr/~brigitte/Publications.bin-gcd.ps`

Lecture 6

Integer Factorisation, Elliptic Curves and Fermat Numbers*

*Six lectures on Algorithms, Trinity term 1999.
 lec06

Abstract

We outline the integer factorisation algorithms ECM, MPQS and NFS, and then compare their expected performance on "typical" or "random" large integers. Finally, we illustrate some of the conclusions by giving a brief historical summary of attempts to factor Fermat numbers.

Outline

Part 1: ECM, MPQS and NFS

- Notation and definitions
- Elliptic curves over finite fields
- The elliptic curve method (ECM)
- The quadratic sieve (QS and MPQS)
- The number field sieve (NFS)
  - Special (SNFS)
  - General (GNFS)

Part 2: Comparison Theorems

- Comparison of ECM and MPQS
- Comparison of ECM and GNFS

Part 3: History

- Attempts to factor Fermat numbers

Notation

n and N always denote positive integers.

p_n denotes a prime number with n decimal digits, e.g. $p_3 = 163$. Similarly, c_n denotes a composite number with n decimal digits, e.g. $c_4 = 1729$.

log or ln denotes the natural logarithm, lg or $\log_2$ denotes the logarithm to base 2.

Almost Always and Almost Never

If $P(n)$ is a predicate, we say that $P(n)$ holds *almost always* if

$$\lim_{N\to\infty} \frac{|\{n \le N : P(n)\}|}{N} = 1$$

and we say that $P(n)$ holds *almost never* if

$$\lim_{N\to\infty} \frac{|\{n \le N : P(n)\}|}{N} = 0 \,.$$

Example (Erdős–Kac): For any $\varepsilon > 0$, n almost always has between $(1-\varepsilon)\log\log n$ and $(1+\varepsilon)\log\log n$ prime factors.

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_1P_2 intersects the elliptic curve at a third point $P_3' = (x_3, -y_3)$, and P_3 is the reflection of P_3' in the x-axis.

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

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

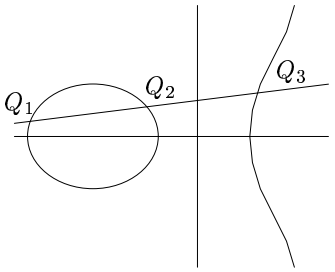


Figure 1: The Group Operation

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 $\mathrm{GF}(p)$, where $p > 3$ is the prime factor we hope to find. (Fortunately, we don't need to know p in advance.) By a theorem of Hasse (1934), the group order g for an elliptic curve over $\mathrm{GF}(p)$ satisfies

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

By a result of Deuring, 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.

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

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

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 $\mathrm{GF}(p)$ 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 $\mathrm{GF}(p)$. 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 B.
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 $\leq B$ take its maximal power $q \leq B$ 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. Otherwise the trial fails.

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

Optimal Choice of B

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

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

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 B 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.

The Second Phase

Both the Pollard "$p - 1$" and Lenstra elliptic curve algorithms can be speeded up by the addition of a second phase. The idea of the second phase is to find a factor in the case that the first phase terminates with a group element $P \neq 0$, such that $|\langle P \rangle|$ is reasonably small (say $O(B^2)$). Here $\langle P \rangle$ is the cyclic group generated by P.

There are several possible implementations of the second phase. One of the simplest uses a pseudorandom walk in $\langle P \rangle$. By the birthday paradox argument, there is a good chance that two points in the random walk will coincide after $O(|\langle P \rangle|^{1/2})$ steps, and when this occurs a nontrivial factor of N can usually be found.

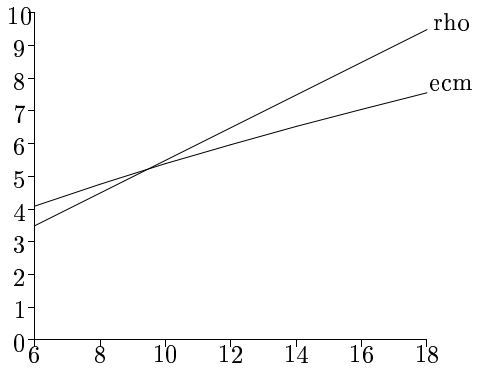
Expected Performance of ECM

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

Comparison with Pollard "rho"



$\log_{10} W$ versus decimal digits in factor

Because of the overheads involved with ECM, a simpler algorithm such as Pollard's "rho" is preferable for finding factors of up to about ten decimal digits, but for larger factors the advantage of ECM becomes apparent.

ECM Example

ECM can routinely find factors p of size about 30 decimal digits. The largest factor known to have been found by ECM is the 53-digit factor

$$p_{53} = 53625112691923843508117942\backslash 31151642817302190330034456 7$$

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

$$g = 2^4 \cdot 3^9 \cdot 3079 \cdot 152077 \cdot 172259 \cdot 1067063 \cdot 3682177 \cdot 3815423 \cdot 8867563 \cdot 15880351$$

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

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 M 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 M. 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.

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 ($c = \Lambda(q) = \log p$ 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

$$\Theta(\exp(c(\ln N \ln\ln N)^{1/2})) ,$$

where $c \sim 1$. The constants involved are such that MPQS is usually faster than ECM if N is the product of two primes which both exceed $N^{1/3}$. This is because the inner loop of MPQS involves only single-precision operations.

P-MPQS and PP-MPQS

In the "one large prime" (P-MPQS) variation w_i is allowed to have one prime factor exceeding B (but not too much larger than B). This is analogous to the second phase of ECM and gives a similar performance improvement.

In the "two large prime" (PP-MPQS) variation w_i can have two prime factors exceeding B – this gives a further performance improvement at the expense of higher storage requirements.

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 $(3^{329}+1)$/(known small factors) was split into a product of 50-digit and 67-digit primes. The final factorisation is

$$\begin{aligned} 3^{329}+1 \;=\; & 2^2 \cdot 547 \cdot 16921 \cdot 256057 \cdot 36913801 \cdot \\ & 177140839 \cdot 1534179947851 \cdot \\ & 24677078822840014266652 77\backslash \\ & 90367680629183726974352 41 \cdot p_{67} \end{aligned}$$

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 ···

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

$$\begin{aligned} F_9 \;=\; & 2424833 \cdot \\ & 74556028256478842083373 9\backslash \\ & 57362004549187833663426 57 \cdot p_{99}\ , \end{aligned}$$

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

$$\begin{aligned} p_{93} \;=\; & 692624557324389620662 78\backslash \\ & 232267733671113810848 25\backslash \\ & 882817397343755705064 92\backslash \\ & 391931849524636731866879 \end{aligned}$$

and p_{118} may be found by division.

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 $m = \lfloor N^{1/d} \rfloor$ and write

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

where the a_j are "base m digits" and $a_d = 1$. 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.

Some 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. Brian Murphy has shown how a very considerable improvement (by a factor of more than ten for number of 140 digits) 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 width 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. In principle it should be possible to distribute the block Lanczos solution over several processors of a parallel machine, but the communication to computation ratio will be 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 pioneered by Lehmer and recently proposed (in more modern form) 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 largest number factored by GNFS is 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

$$f(x) = x - 3443565780924253695177900 7$$

and

$$\begin{aligned} g(x) \;=\; & +439682082840x^5 \\ & +390315678538960x^4 \\ & -7387325293892994572x^3 \\ & -1902715324374298871 4824x^2 \\ & -6344102569446461791 3930613x \\ & +3185539170714743503 92223507494 . \end{aligned}$$

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 effect of the second property alone makes $g(x)$ as effective at generating relations as a polynomial chosen at random for an integer of 121 decimal digits. In effect judicious polynomial selection removed at least 19 digits from RSA140, making it much easier to factor.

The polynomial selection took 2000 CPU-hours on four 250 MHz SGI Origin 2000 processors. This is about 200 Mips-years, or 10% 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, an attempt to factor the 512-bit number RSA155 is well underway. I am willing to bet £100 that it will be factored before the year 2000.

Summary of Part 1

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.

Despite much progress in the development of efficient algorithms, our knowledge of the complexity of factorisation is inadequate. We would like to find a polynomial time factorisation algorithm or else prove that one does not exist. 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.

Predictions

From the predicted run time for GNFS, we would expect RSA155 to take 6.5 times as long as RSA140. On the other hand, 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 almost 4 decimal digits per year improvement in the capabilities of GNFS, without any algorithmic improvements. A similar argument applies 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 thirty 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.

Part 2: Comparison of Algorithms

We now compare the expected behaviour of ECM, MPQS and GNFS on large, "random" or "typical" integers (*not* integers chosen by a cryptographer).

If N is a (large) integer with prime factors $p_1 \geq p_2 \geq \ldots$, we *assume* that the expected time to factor N by these three methods is $T_{ECM}(N), T_{MPQS}(N), T_{GNFS}(N)$ respectively, where

$$\log T_{ECM} = \sqrt{(2+o(1)) \log p_2 \log\log p_2}$$

$$\log T_{MPQS} = \sqrt{(1+o(1)) \log N \log\log N}$$

$$\log T_{GNFS} = \sqrt[3]{(c+o(1)) \log N (\log\log N)^2}$$

Here c is some positive constant, and the $o(1)$ terms are as $p_2 \to \infty$ or $N \to \infty$.

ECM and MPQS

Theorem

$T_{MPQS}(N) > e^{\sqrt{\log N}} T_{ECM}(N)$ almost always.

Idea of Proof

$$\left(\frac{\log T_{MPQS}}{\log T_{ECM}}\right)^2 \geq \frac{\log N}{(2+o(1)) \log p_2}$$

but from the known distribution of $\log p_2 / \log N$ this is at least $1+\varepsilon$ with probability at least $1 - O(\varepsilon^2)$. Thus, the Theorem holds if $e^{\sqrt{\log N}}$ is replaced by any $f(N)$ satisfying

$$\log f(N) = o\left(\sqrt{\log N \log\log N}\right) .$$

Corollary

For all $\varepsilon > 0$, $T_{ECM} < \varepsilon T_{MPQS}$ holds almost always.

ECM and GNFS

Theorem

For all $\varepsilon > 0$, $T_{GNFS} < \varepsilon T_{ECM}$ holds almost always.

However, *this is not the full story*, because ECM can find small factors quickly, and after dividing them out GNFS can finish the factorisation more quickly than if ECM had not been used.

Let $T_{ECM}^{(\lambda)}(N)$ be the expected time for ECM to find at least λk prime factors of N, where k is the total number of prime factors of N. (It does not matter how we count multiple factors.)

The "two thirds" Theorem

Let K be any positive constant, and $\lambda \in [0, 1]$.

If $\lambda < 2/3$ then $T_{ECM}^{(\lambda)} < KT_{GNFS}$ almost always;
if $\lambda > 2/3$ then $T_{ECM}^{(\lambda)} > KT_{GNFS}$ almost always.

Thus, it is better to use a combination of ECM and GNFS than either alone, and with a sensible strategy we expect to find about two thirds of the prime factors by ECM and the remaining one third by GNFS.

Part 3: Some History of Fermat Numbers

For a nonnegative integer n, the n-th *Fermat number* is $F_n = 2^{2^n} + 1$. It is known that F_n is prime for $0 \le n \le 4$, and composite for $5 \le n \le 23$. Also, for $n \ge 2$, the factors of F_n are of the form

$$k2^{n+2} + 1 \ .$$

In 1732 Euler found that $641 = 5 \cdot 2^7 + 1$ is a factor of F_5, thus disproving Fermat's belief that all F_n are prime[1]. Euler apparently used trial division by primes of the form $64k + 1$.

No Fermat primes larger than F_4 are known, and a probabilistic argument makes it plausible that only a finite number of F_n (perhaps only $F_0, \ldots, F_4$) are prime. It is known that F_n is composite for $5 \le n \le 23$.

[1] "Back of envelope" proof: working mod 641, $5.2^7 = -1 \Rightarrow 5^4.2^{28} = 1$, but $5^4 = -2^4$, so $2^{32} = -1$.

Factorisation of Fermat Numbers

The complete factorisation of the Fermat numbers $F_6, F_7, \ldots$ has been a challenge since Euler's time. Because the F_n grow rapidly in size, a method which factors F_n may be inadequate for F_{n+1}.

F_6

In 1880, Landry factored $F_6 = 274177 \cdot p_{14}$. Landry's method was never published in full, but Williams has attempted to reconstruct it.

Hand Computations

In the period 1877–1970, several small factors of F_n for various $n \ge 9$ were found by taking advantage of the special form of these factors. For example, in 1903 Western found the factor $p_7 = 2424833 = 37 \cdot 2^{16} + 1$ of F_9.

Significant further progress was only possible with the development of the digital computer and more efficient algorithms.

$\mathbf{F}_7$

In 1970, Morrison and Brillhart factored

$$F_7 = 59649589127497217 \cdot p_{22}$$

by the continued fraction method. This method has now been superseded by MPQS which, perhaps surprisingly, has never been the first to factor a Fermat number.

$\mathbf{F}_8$

In 1980, Brent and Pollard factored

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

by a modification of Pollard's "rho" method. The "rho" method is now largely superseded by ECM.

Nowadays, F_7 and F_8 are "easy" to factor by ECM or MPQS.

$\mathbf{F}_9$

Logically, the next step after the factorisation of F_8 was the factorisation of F_9. It was known that

$$F_9 = 2424833 \cdot c_{148}$$

The 148-digit composite number resisted attack by methods such as Pollard rho, Pollard $p \pm 1$, and the elliptic curve method (ECM), which would have found "small" factors. It was too large to factor by the continued fraction method or even by MPQS.

The difficulty was finally overcome by the invention of the (special) number field sieve (SNFS), based on a new idea of Pollard.
In 1990, Lenstra, Lenstra, Manasse and Pollard, with the assistance of many collaborators and approximately 700 workstations scattered around the world completely factored F_9 by SNFS. As we already mentioned, the factorisation is

$$F_9 \quad = \quad 2424833 \cdot \ p_{49} \cdot p_{99} \ .$$

$\mathbf{F}_{10}$

After the factorisation of F_9 in 1990, F_{10} 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

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

$\mathbf{F}_{11}$

F_{11} was completely factored in 1988, *before* the factorisation of F_9 and F_{10}. In fact,

$$\begin{aligned} F_{11} \quad = \quad & 319489 \cdot 974849 \cdot \\ & 167988556341760475137 \cdot \\ & 3560841906445833920513 \cdot p_{564} \end{aligned}$$

The two 6-digit factors were found by Cunningham in 1899, and I found the remaining factors in May 1988, using ECM on a Fujitsu VP100. The 564-digit factor passed a probabilistic primality test, and a rigorous proof of primality was provided by Morain.

The reason why F_{11} could be completely factored before F_9 and F_{10} is that the difficulty of completely factoring numbers by ECM is determined mainly by the size of the *second-largest* prime factor of the number.

The second-largest prime factor of F_{11} has 22 digits and is much easier to find by ECM than the 40-digit factor of F_{10} or the 49-digit factor of F_9.

Summary, $\mathbf{F}_5, \ldots, \mathbf{F}_{11}$

A brief summary of the history of factorisation of $F_5, \ldots, F_{11}$ is given in the Table.

Table 2: Complete factorisation of F_n, $n = 5, \ldots, 11$

n	Factorisation	Date	Comments
5	$p_3 \cdot p_7$	1732	Euler
6	$p_6 \cdot p_{14}$	1880	Landry
7	$p_{17} \cdot p_{22}$	1970	Morrison and Brillhart
8	$p_{16} \cdot p_{62}$	1980	Brent and Pollard (p_{16}, p_{62})
		1980	Williams (primality of p_{62})
9	$p_7 \cdot p_{49} \cdot p_{99}$	1903	Western (p_7)
		1990	Lenstra *et al* (p_{49}, p_{99})
10	$p_8 \cdot p_{10} \cdot p_{40} \cdot p_{252}$	1953	Selfridge (p_8)
		1962	Brillhart (p_{10})
		1995	Brent (p_{40}, p_{252})
11	$p_6 \cdot p'_6 \cdot p_{21} \cdot p_{22} \cdot p_{564}$	1899	Cunningham (p_6, p'_6)
		1988	Brent (p_{21}, p_{22}, p_{564})
		1988	Morain (primality of p_{564})

$\mathbf{F}_{12}$

The smallest Fermat number which is not yet completely factored is F_{12}. It is known that

$$\begin{aligned} F_{12} &= 114689 \cdot 26017793 \cdot \\ &\quad 63766529 \cdot 190274191361 \cdot \\ &\quad 1256132134125569 \cdot c_{1187}\ , \end{aligned}$$

where the 16-digit factor was found by Baillie in 1986, using the Pollard $p-1$ method (and rediscovered in 1988 using ECM).

F_{12} has at least seven prime factors, spoiling a "conjecture" based on the observation that F_n has exactly $n-6$ prime factors for $8 \le n \le 11$.

$\mathbf{F}_{13}$

It is known that

$$\begin{aligned} F_{13} &= 2710954639361 \cdot \\ &\quad 2663848877152141313 \cdot \\ &\quad 3603109844542291969 \cdot \\ &\quad 319546020820551643220672513 \cdot c_{2391}\ , \end{aligned}$$

where the 13-digit factor was found by Hallyburton and Brillhart (1975), and the two 19-digit factors were found by Crandall (1991).

I found the 27-digit factor in June 1995, using ECM on an IBM PC equipped with a Dubner Cruncher board.

$\mathbf{F}_{14}$

$F_{14} = c_{4933}$ is composite, but no nontrivial factors are known. The smallest prime factor probably has at least 30 decimal digits.

$\mathbf{F}_{15}$

$$F_{15} = 1214251009 \cdot 2327042503868417 \cdot c_{9840},$$

where the 13- and 16-digit prime factors were found by Kraitchik (1925) and Gostin (1987). In July, 1997, Brent, Crandall, Dilcher & Van Halewyn found a 33-digit factor

$$p_{33} = 168768817029516972383024127016961$$

using ECM. The quotient is c_{9808}.

$\mathbf{F}_{16}$

$$\begin{aligned} F_{16} &= 825753601 \cdot \\ &\quad 188981757975021318420037633 \cdot c_{19694} \end{aligned}$$

where the 9-digit factor was found by Selfridge (1953), and the 27-digit factor was found in December 1996 by Brent, Crandall & Dilcher using ECM.

$\mathbf{F}_{17}$

$$F_{17} = 31065037602817 \cdot c\,.$$

$\mathbf{F}_{18}$

$$F_{18} = 18631489 \cdot 81274690703860512587777 \cdot c\,,$$

where the 23-digit factor was found by McIntosh and Tardif in April 1999, using ECM.

$\mathbf{F}_{19}, \ldots, \mathbf{F}_{24}$

$F_{19}, F_{20}, F_{21}, F_{23}$ are composite and some small factors are known.

F_{22} is composite but no factors are known.

The status of F_{24} is unknown.

References

[1] A. O. L. Atkin and F. Morain, Elliptic curves and primality proving, *Math. Comp.* **61** (1993), 29–68. Programs available from `ftp://ftp.inria.fr/INRIA/ecpp.V3.4.1.tar.Z` .

[2] D. Atkins, M. Graff, A. K. Lenstra and P. C. Leyland, The magic words are squeamish ossifrage, *Advances in Cryptology: Proc. Asiacrypt'94*, *LNCS* **917**, Springer-Verlag, Berlin, 1995, 263–277.

[3] H. Boender and H. J. J. te Riele, Factoring integers with large prime variations of the quadratic sieve, *Experimental Mathematics* **5** (1996), 257–273. Also `ftp://ftp.cwi.nl/pub/CWIreports/NW/NM-R9513.ps.Z` .

[4] R. P. Brent, *Large factors found by ECM,* Oxford University Computing Laboratory, May 1999. `ftp://ftp.comlab.ox.ac.uk/pub/Documents/techpapers/Richard.Brent/champs.txt` .

[5] R. P. Brent, Factorization of the tenth Fermat number, *Math. Comp.* **68** (1999), 429-451. Preliminary version: `ftp://ftp.comlab.ox.ac.uk:/pub/Documents/techpapers/Richard.Brent/rpb161tr.dvi.gz` .

[6] R. P. Brent, Some parallel algorithms for integer factorisation, *Proc. Fifth International Euro-Par Conference* (Toulouse, France, 1-3 Sept 1999), to appear. `ftp://ftp.comlab.ox.ac.uk:/pub/Documents/techpapers/Richard.Brent/rpb193.ps.gz` .

[7] R. P. Brent, R. E. Crandall, K. Dilcher and C. Van Halewyn, Three new factors of Fermat numbers, *Math. Comp.*, to appear. `ftp://ftp.comlab.ox.ac.uk:/pub/Documents/techpapers/Richard.Brent/rpb175.dvi.gz` .

[8] S. Cavallar, B. Dodson, A. K. Lenstra, P. Leyland, W. Lioen, P. L. Montgomery, B. Murphy, H. te Riele and P. Zimmermann, *Factorization of RSA-140 using the number field sieve*, announced 4 February 1999. `ftp://ftp.cwi.nl/pub/herman/NFSrecords/RSA-140` .

[9] M. Elkenbracht-Huizing, An implementation of the number field sieve, *Experimental Mathematics*, **5** (1996), 231–253.

[10] L. Euler, Observationes de theoremate quodam Fermatiano aliisque ad numeros primos spectantibus, *Comm. Acad. Sci. Petropol.* **6**, ad annos 1732–33 (1738), 103–107; also *Leonhardi Euleri Opera Omnia*, Ser. I, vol. II, Teubner, Leipzig, 1915, 1–5.

[11] L. Euler, Theoremata circa divisores numerorum, *Novi. Comm. Acad. Sci. Petropol.* **1**, ad annos 1747–48 (1750), 20–48.

[12] P. de Fermat, *Oeuvres de Fermat,* vol. II: *Correspondance,* P. Tannery and C. Henry (editors), Gauthier-Villars, Paris, 1894.

[13] A. K. Lenstra and H. W. Lenstra, Jr. (editors), The development of the number field sieve, *Lecture Notes in Mathematics* **1554**, Springer-Verlag, Berlin, 1993.

[14] A. K. Lenstra, H. W. Lenstra, Jr., M. S. Manasse, and J. M. Pollard, The factorization of the ninth Fermat number, *Math. Comp.* **61** (1993), 319–349.

[15] A. K. Lenstra and M. S. Manasse, Factoring by electronic mail, *Proc. Eurocrypt '89*, *LNCS* **434**, Springer-Verlag, Berlin, 1990, 355–371.

[16] A. K. Lenstra and M. S. Manasse, Factoring with two large primes, *Math. Comp.* **63** (1994), 785–798.

[17] H. W. Lenstra, Jr., Factoring integers with elliptic curves, *Annals of Mathematics* (2) **126** (1987), 649–673.

[18] P. L. Montgomery, Square roots of products of algebraic numbers, *Mathematics of Computation 1943 – 1993, Proc. Symp. Appl. Math.* **48** (1994), 567–571.

[19] P. L. Montgomery, A block Lanczos algorithm for finding dependencies over $GF(2)$, *Advances in Cryptology: Proc. Eurocrypt'95*, *LNCS* **921**, Springer-Verlag, Berlin, 1995, 106–120.

[20] A. M. Odlyzko, The future of integer factorization, *CryptoBytes* **1**, 2 (1995), 5–12. Available from `http://www.rsa.com/rsalabs/pubs/cryptobytes` .

[21] R. L. Rivest, A. Shamir and L. Adleman, A method for obtaining digital signatures and public-key cryptosystems, *Comm. ACM* **21** (1978), 120–126.

[22] RSA Laboratories, Information on the RSA challenge, `http://www.rsa.com/rsalabs/html/challenges.html` .

[23] R. S. Schaller, Moore's law: past, present and future, *IEEE Spectrum* **34**, 6 (June 1997), 52–59.

[24] J. L. Selfridge, Factors of Fermat numbers, *MTAC* **7** (1953), 274–275.

[25] A. Shamir, *Factoring large numbers with the TWINKLE device* (extended abstract), preprint, 1999. Announced at Eurocrypt'99.

[26] P. W. Shor, Polynomial time algorithms for prime factorization and discrete logarithms on a quantum computer, *SIAM J. Computing* **26** (1997), 1484–1509.

[27] I. N. Stewart and D. O. Tall, *Algebraic Number Theory*, second edition, Chapman and Hall, 1987.

[28] A. M. Vershik, The asymptotic distribution of factorizations of natural numbers into prime divisors, *Dokl. Akad. Nauk SSSR* **289** (1986), 269–272; English transl. in *Soviet Math. Dokl.* **34** (1987), 57–61.

[29] H. C. Williams, How was F_6 factored?, *Math. Comp.* **61** (1993), 463–474.