Faster deterministic integer factorisation

David Harvey
(joint work with Edgar Costa, NYU)

University of New South Wales

25th October 2011

"The obvious mathematical breakthrough would be the development of an easy way to factor large prime numbers"

— Bill Gates, 1995

Plan for talk

1. The integer factorisation problem
2. Summary of modern factoring algorithms
3. Detour: probabilistic vs deterministic algorithms
4. Strassen's deterministic factoring algorithm
5. Our improvements to Strassen's algorithm
6. Bostan–Gaudry–Schost algorithm (briefly)

The integer factorisation problem

Input: a positive integer N.

Output: complete prime factorisation of N.

Example 1: Input is $N = 91$. Output is 7×13.

Example 2 (RSA-768 challenge): Input is $N =$

1230186684530117755130494958384962720772853569595334792197
3224521517264005072636575187452021997864693899564749427740
6384592519255732630345373154826850791702612214291346167042
921431160222124047927473779408066535141959745985690214341 3.

Output is

$$
\begin{matrix}
33478071698956898786044169848 \\
21269081770479498371376856891 \\
24313889828837938780022876147 \\
11652531743087737814467999489
\end{matrix}
\quad \times \quad
\begin{matrix}
36746043666799590428244633799 \\
62795263227915816434308764267 \\
60322838157396665112792333734 \\
17143396810270092798736308917.
\end{matrix}
$$

The integer factorisation problem

Simplest factoring algorithm is **trial division**:

For each $k = 2, 3, 4, \ldots$, test whether N is divisible by k.

Stop when k reaches $\lfloor \sqrt{N} \rfloor$.

Worst case complexity: $O(N^{1/2})$ divisibility tests.

Size of input is $O(\log N)$ bits.

Complexity is **exponential** in the size of the input.

Only practical for very small N.

Modern factoring algorithms

Three main factoring algorithms used in practice for large N:

- Quadratic sieve (QS)
- Number field sieve (NFS)
- Elliptic curve method (ECM)

I will not discuss algorithms for quantum computers in this talk.

Modern factoring algorithms

The **quadratic sieve** (QS) factors N with heuristic complexity

$$\exp\left((1+o(1))(\log N)^{1/2}(\log\log N)^{1/2}\right)$$

bit operations.

This is halfway between polynomial time

$$\exp\left(C(\log N)^0(\log\log N)^1\right) = (\log N)^C$$

and exponential time

$$\exp\left(C(\log N)^1(\log\log N)^0\right) = N^C.$$

It is subexponential (better than N^C for any $C > 0$) but superpolynomial (worse than $(\log N)^C$ for any $C > 0$).

Modern factoring algorithms

The **number field sieve** (NFS) factors N with heuristic complexity

$$\exp\left(((64/9)^{1/3} + o(1))(\log N)^{1/3}(\log\log N)^{2/3}\right).$$

This is a bit 'closer' to polynomial time than the quadratic sieve.

The record factorisation shown on the first slide was achieved using the NFS.

In practice, for 'hard' composites, QS is the best choice for N up to about 120 decimal digits, and NFS is best for N larger than this.

Modern factoring algorithms

The **elliptic curve method** (ECM) finds the smallest factor p of N with heuristic complexity

$$\exp\left((\sqrt{2}+o(1))(\log p)^{1/2}(\log\log p)^{1/2}\right).$$

Here complexity measures the number of arithmetic operations on integers of size $\log N$.

ECM can find factors of reasonable size even if N if way too large to apply QS/NFS.

Record is a 73-digit factor.

Modern factoring algorithms

Theoretical problem with QS, NFS, ECM:

We can't prove that they work.

Complexity bounds are heuristic, not rigorous.

In practice this is irrelevant — N gets factored anyway. The factorisation, once found, is trivial to verify.

The complexity argument for ECM gets pretty close to a proof. The missing ingredient is a proof that the interval

$$p - 2\sqrt{p} < n < p + 2\sqrt{p}$$

contains sufficiently many smooth integers (numbers whose prime factors are all 'small'). This is a very difficult problem in analytic number theory.

Modern factoring algorithms

What about algorithms with proven complexity bounds?

The **class group relations method** provably factors N with expected running time

$$\exp\left((1+o(1))(\log N)^{1/2}(\log\log N)^{1/2}\right).$$

This is subexponential. It has the same shape as the bound for QS, but the $o(1)$ is larger, making it uncompetitive in practice.

It is a **probabilistic**, **Las Vegas** algorithm: it requires a source of random bits as input. The running time is a random variable with finite expectation. The output is always correct.

Modern factoring algorithms

What about **deterministic** algorithms — algorithms that are not permitted a source of random bits?

Current state of knowledge:

All known deterministic factoring algorithms with rigorously established complexity bounds have **exponential** running time.

(Example: trial division!)

Detour: deterministic vs probabilistic algorithms

Why is there such a huge gap between probabilistic and deterministic algorithms? Why is randomness so powerful?

My favourite example in number theory illustrating the power of randomness is the problem of finding **quadratic nonresidues**.

This is a key ingredient in many algorithms in number theory, for example finding square roots in finite fields.

Input: a prime p.

Output: a quadratic nonresidue modulo p, i.e. an integer b such that $x^2 \equiv b \bmod p$ has no solution.

Example: for $p = 7$, the squares modulo 7 are $\{1, 2, 4\}$. The output could be $b = 3$, or $b = 5$, or $b = 6$.

Detour: deterministic vs probabilistic algorithms

Here is a very simple probabilistic algorithm:

Choose random $2 \leq b \leq p-1$. Test if b is a quadratic nonresidue. If yes, we win. If no, start again with a new b.

Probability of success on any trial is 50%, since half the integers are quadratic residues. So the expected number of trials is 2.

Testing whether b is a quadratic residue can be done in time $O((\log p)^C)$.

(One method: use identity $(b/p) \equiv b^{(p-1)/2} \bmod p$. Another method: use quadratic reciprocity for Jacobi symbols.)

Therefore we can find a quadratic nonresidue with expected running time $O((\log p)^C)$.

Detour: deterministic vs probabilistic algorithms

What is the best known **deterministic** algorithm for finding a quadratic nonresidue?

Essentially, try each $b = 2, 3, 4, \ldots$ until we succeed.

The best we can prove is that the smallest nonresidue is $O(p^{C+\varepsilon})$ where $C = \frac{1}{4\sqrt{e}} \approx 0.152$.

This yields an **exponential time** algorithm!

(Under the extended Riemann hypothesis, we can prove that the smallest nonresidue is $O((\log p)^2)$, so we get a polynomial time algorithm.)

Strassen's algorithm

For the rest of the talk I will discuss only deterministic factoring algorithms with rigorously established complexity bounds.

(None of these are remotely competitive in practice with the heuristic algorithms.)

The best complexity bounds in this case have the general shape

$$O(N^{1/4+o(1)}).$$

I will start with the simplest variant, due to Strassen (1976).

Strassen's algorithm

Tools:

- Fast integer multiplication
- Fast polynomial multiplication over $\mathbf{Z}/N\mathbf{Z}$
- Fast product tree
- Fast multipoint evaluation

Strassen's algorithm

Let $M_{\rm int}(k)$ = cost of multiplying integers with k bits.

Primary school algorithm achieves $M_{\rm int}(k) = O(k^2)$.

Current best result (Fürer 2007), very clever use of FFT:

$$M_{\rm int}(k) = O(k \log k \, 2^{\log^*(k)}),$$

where $\log^*(x)$ is the iterated logarithm:

$$\log^*(x) = \begin{cases} 0 & x < 1, \\ 1 + \log^*(\log x) & x \geq 1. \end{cases}$$

This grows *very* slowly, so $M_{\rm int}(k)$ behaves essentially like $k \log k$.

Strassen's algorithm

Let $M(k)$ = cost of multiplying degree k polynomials over $\mathbf{Z}/N\mathbf{Z}$.

Kronecker substitution converts this to an integer multiplication problem. For example, $(2x^2 + 3x + 5)(7x^2 + 11x + 13)$ becomes

$$200030005 \cdot 700110013 = 140043009400940065,$$

so the product is $14x^4 + 43x^3 + 94x^2 + 94x + 65$.

We need to leave enough 'space' for the coefficients of the product, which are bounded by $N^2 k$.

Thus $M(k) = O(M_{\mathrm{int}}(k \log(N^2 k)))$.

In our application $k \leq N$, so we get simply

$$M(k) = O(M_{\mathrm{int}}(k \log N)).$$

Strassen's algorithm

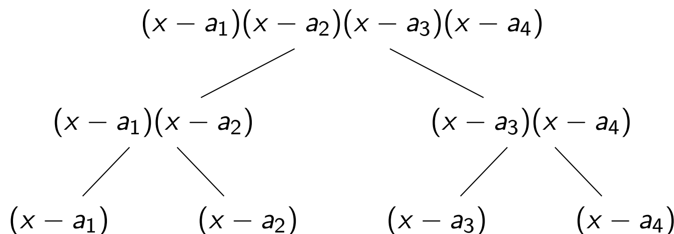
Let $a_1, \ldots, a_k \in \mathbf{Z}/N\mathbf{Z}$.

Consider the problem of computing the coefficients of

$$g(x) = (x - a_1)(x - a_2) \cdots (x - a_k) \in (\mathbf{Z}/N\mathbf{Z})[x].$$

The straightforward algorithm has complexity $O(k^2)$.

Better way: divide and conquer. We get a 'product tree':



Strassen's algorithm

Let $T(k)$ = cost of computing $(x - a_1) \cdots (x - a_k)$.

Ignoring complications from parity of k, we get

$$T(k) = 2T(k/2) + M(k/2),$$

and then we deduce that

$$T(k) = O(M(k) \log k).$$

In other words we do $O(M(k))$ work for each of $O(\log k)$ levels of the tree.

Strassen's algorithm

Let $a_1, \ldots, a_k \in \mathbf{Z}/N\mathbf{Z}$, and let $f \in (\mathbf{Z}/N\mathbf{Z})[x]$ of degree k.

Consider the problem of computing $f(a_1), \ldots, f(a_k)$.

This is a standard **multipoint evaluation** problem.

If we evaluate each $f(a_i)$ separately, we get complexity $O(k^2)$.

Better way: divide and conquer. Let $r = \lfloor k/2 \rfloor$. Let

$$f_1(x) = f(x) \bmod (x - a_1) \cdots (x - a_r),$$
$$f_2(x) = f(x) \bmod (x - a_{r+1}) \cdots (x - a_k).$$

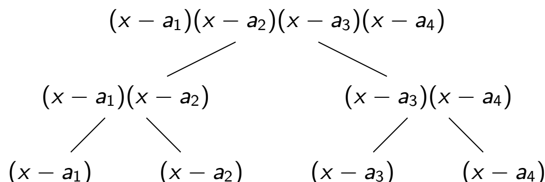
Then

$$f(a_i) = \begin{cases} f_1(a_i) & 1 \le i \le r, \\ f_2(a_i) & r + 1 \le i \le k. \end{cases}$$

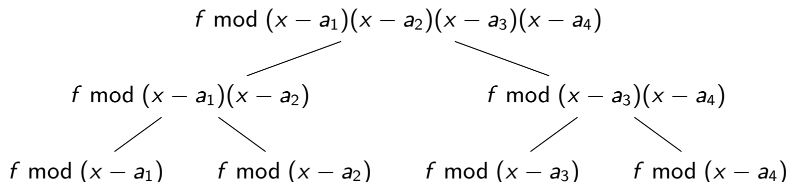
So we have reduced to two subproblems of half the size.

Strassen's algorithm

In other words, we build a product tree from bottom to top:



and then reduce f down the tree from the root to the leaves:



Finally $f(a_i) = f \bmod (x - a_i)$ for each i.

Strassen's algorithm

Let $E(k)$ = cost of evaluating $f(a_1), \ldots, f(a_k)$, given the product tree for $a_1, \ldots, a_k$ as input. Then

$$E(k) = 2E(k/2) + 2D(k/2),$$

where $D(k/2)$ = cost of dividing a polynomial of degree k by a monic polynomial of degree $k/2$, obtaining quotient and remainder.

Using Newton's method for power series inversion (details omitted!), we get

$$D(k) = O(M(k)).$$

Conclusion:

$$E(k) = O(M(k) \log k).$$

Strassen's algorithm

Summary of results so far:

- $M_{\rm int}(k) = O(k \log k \, 2^{\log^*(k)})$ (integer multiplication)
- $M(k) = O(M_{\rm int}(k \log N))$ (polynomial multiplication)
- $T(k) = O(M(k) \log k)$ (product tree)
- $E(k) = O(M(k) \log k)$ (multipoint evaluation)

Strassen's algorithm

Finally I can describe Strassen's algorithm for factoring N.

I will explain only how to find one nontrivial factor.

Getting the complete factorisation within the same complexity bound is not difficult (but slightly fiddly).

Let $d = \lceil N^{1/4} \rceil$.

If N is composite, it must have at least one nontrivial factor $\leq d^2$.

In particular, $\gcd((d^2)!, N) > 1$.

Strassen's algorithm

Consider the polynomial

$$f(x) = (x+1)(x+2)\cdots(x+d) \in (\mathbf{Z}/N\mathbf{Z})[x].$$

We can compute $f(x)$ in time $O(M(d)\log d)$ (product tree).

Let

$$\begin{aligned} b_0 &= f(0) = 1 \cdot 2 \cdot \cdots \cdot d, \\ b_1 &= f(d) = (d+1)(d+2)\cdots(2d), \\ &\cdots \\ b_{d-1} &= f((d-1)d) = ((d-1)d+1)\cdots(d^2), \end{aligned}$$

We can compute all the $b_j \pmod N$ in time $O(M(d)\log d)$ (multipoint evaluation), and we have

$$(d^2)! = b_0 b_1 \cdots b_{d-1} \pmod N.$$

Strassen's algorithm

Now consider

$$t = \gcd(N, b_0 \cdots b_{d-1} \bmod N).$$

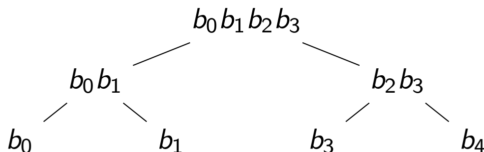
If $t = 1$, then N must be prime.

If $1 < t < N$, then we have recovered a nontrivial factor of N.

There still remains the possibility that $t = N$. This could occur for example if N is a product of three primes near $N^{1/3}$; then all these primes will occur among the b_j.

Strassen's algorithm

In this case, we build a product tree for the b_j:



Taking the GCD of each node with N, working from the root to a leaf, we will find some b_i such that $\gcd(b_i, N) > 1$.

If $1 < \gcd(b_i, N) < N$, we have found a nontrivial factor.

Strassen's algorithm

It's still possible that $\gcd(b_i, N) = N$. This could occur if the prime factors of N were all very close together.

But now recall

$$b_i = f(id) = (id + 1)(id + 2) \cdots (id + d).$$

We can build yet *another* product tree for this expression, and apply the same GCD trick.

Now the prime factors have nowhere left to hide, and we are guaranteed to find one.

Strassen's algorithm

Complexity analysis:

Computing the b_i's dominates the algorithm, with complexity

$$\begin{aligned} O(M(d)\log d) &= O(M(N^{1/4})\log N) \\ &= O(M_{\mathrm{int}}(N^{1/4}\log N)\log N). \end{aligned}$$

The contribution from everything else is negligible ($O(d)$ multiplications in $\mathbf{Z}/N\mathbf{Z}$, and $O(\log d)$ GCDs).

Conclusion: one can deterministically factor N in time

$$O(M_{\mathrm{int}}(N^{1/4}\log N)\log N).$$

Improvements to Strassen's algorithm

Now I will discuss our improvements to Strassen's algorithm.

Key observation: if N is odd and composite, then it must have an **odd** factor $\leq \sqrt{N}$.

It is easy to remove factors of 2 from N, so let us assume N is odd.

Improvements to Strassen's algorithm

Choose $d = \lceil N^{1/4}/\sqrt{2} \rceil$ and consider

$$f(x) = (x+1)(x+3)(x+5)\cdots(x+2d-1).$$

This has degree d, and

$$f(0)f(2d)f(4d)\cdots f((d-1)2d) = \prod_{\substack{1 \le j < 2d^2 \\ j \text{ odd}}} j.$$

The right hand side includes all odd integers $\le N^{1/2}$.

We can run an algorithm almost exactly the same as Strassen's.

The only difference in the complexity analysis is that d is reduced by a factor of $\sqrt{2}$, so we have gained a constant factor of $\sqrt{2}$.

Improvements to Strassen's algorithm

We can push this further.

Suppose that N is not divisible by 2 or 3.

Let $d = \lceil N^{1/4}/\sqrt{12} \rceil$ and consider

$$f(x) = (x+1)(x+5)(x+7)(x+11)\cdots(x+6d-5)(x+6d-1).$$

This has degree $2d$. Evaluate at $2d$ points and multiply:

$$f(0)f(6d)\cdots f((2d-1)6d) = \prod_{\substack{1 \le j < 12d^2 \\ (j,6)=1}} j.$$

Working through the complexity analysis shows that we have saved a factor of $\sqrt{3}$ over Strassen's algorithm.

Improvements to Strassen's algorithm

What happens in general?

Suppose we include all primes $\leq B$.

Let

$$Q = \prod_{\substack{p \leq B \\ p \text{ prime}}} p, \qquad R = \phi(Q) = \prod_{\substack{p \leq B \\ p \text{ prime}}} (p-1),$$

$$h(x) = \prod_{\substack{i=1 \\ (i,Q)=1}}^{Q-1} (x+i).$$

For example, if $B = 5$, then $Q = 2 \cdot 3 \cdot 5 = 30$, $R = 1 \cdot 2 \cdot 4 = 8$, and

$$h(x) = (x+1)(x+7)(x+11)(x+13)(x+17)(x+19)(x+23)(x+29).$$

Note that $\deg h = R$.

Improvements to Strassen's algorithm

Now assume N is not divisible by any primes $\leq B$.

Put $d = \lceil N^{1/4}/\sqrt{QR} \rceil$, and

$$f(x) = h(x)h(x+Q)h(x+2Q)\cdots h(x+(d-1)Q).$$

Then $\deg f = Rd$. Evaluate at Rd points and multiply:

$$f(0)f(Qd)\cdots f((Rd-1)Qd) = \prod_{\substack{1 \leq j < QRd^2 \\ (j,Q)=1}} j.$$

Savings over Strassen's algorithm is

$$\sqrt{Q/R} = \prod_{\substack{p \leq B \\ p \text{ prime}}} \sqrt{\frac{p}{p-1}}.$$

Improvements to Strassen's algorithm

For example:

B	speedup $= \sqrt{Q/R}$
2	1.414
3	1.732
5	1.936
10	2.092
100	2.883
1000	3.514
10000	4.053
100000	4.523

Improvements to Strassen's algorithm

In fact, by Merten's theorem, for large B we have

$$\sqrt{Q/R} = \prod_{\substack{p \leq B \\ p \text{ prime}}} \sqrt{\frac{p}{p-1}} \approx e^{\gamma/2}\sqrt{\log B}.$$

where $\gamma = 0.5772...$ is Euler's constant.

So the improvement over Strassen's algorithm does diverge to infinity as $B \to \infty$!

Improvements to Strassen's algorithm

Unfortunately we are not free to choose B as large as we like.

One obvious restriction is that we need enough time to compute $h(x)$, which has degree R. So we need R no larger than $N^{1/4}$.

For example, $R \sim N^{1/8}$ is quite safe.

But R grows quite rapidly with B. In fact

$$\log R = \sum_{p \leq B} \log(p-1) \sim \sum_{p \leq B} \log p \sim B$$

by the Prime Number Theorem.

So we can only take B as large as $O(\log N)$.

Improvements to Strassen's algorithm

Conclusion: by choosing B a suitable multiple of $\log N$, we can speed up Strassen's algorithm by a factor of

$$\sqrt{\log \log N}.$$

The Bostan–Gaudry–Schost algorithm

Recall that Strassen's algorithm has complexity

$$O(M_{\mathrm{int}}(N^{1/4}\log N)\log N).$$

Bostan–Gaudry–Schost (2007) show how to improve this to

$$O(M_{\mathrm{int}}(N^{1/4}\log N)),$$

i.e. they save a factor of $\log N$.

The Bostan–Gaudry–Schost algorithm

I will not explain the Bostan–Gaudry–Schost algorithm.

The basic idea is that they use the same polynomial $f(x) = (x+1)\cdots(x+d)$, but they take advantage of the structure of this polynomial and that the evaluation points lie in an arithmetic progression.

Their approach actually saves a logarithmic factor in many other contexts.

This can make a huge difference in practice. In my Ph.D. thesis, I worked on algorithms for counting points on hyperelliptic curves over finite fields. For a curve over $\mathbf{F}_p$ with $p \sim 10^{16}$, this can mean a factor of 50 savings in running time!

The Bostan–Gaudry–Schost algorithm

With a little effort, our improvements apply also to the Bostan–Gaudry–Schost algorithm:

Theorem 1: An integer $N > 1$ can be deterministically factored in

$$O(M_{\mathrm{int}}(N^{1/4} \log N / \sqrt{\log \log N}))$$

bit operations.

This is the best known bound for deterministic integer factorisation that I am aware of.

Thank you!