Integer factorization

Daniel J. Bernstein *

djb@cr.yp.to

"The problem of distinguishing prime numbers from composite numbers, and of resolving the latter into their prime factors, is known to be one of the most important and useful in arithmetic," Gauss wrote in his *Disquisitiones Arithmeticae* in 1801. "The dignity of the science itself seems to require that every possible means be explored for the solution of a problem so elegant and so celebrated."

But what exactly *is* the problem?

Do we want to distinguish prime numbers from composite numbers? Or do we want to find all the prime factors of composite numbers? These are quite different problems. Imagine, for example, that someone gives you a 10000-digit composite number. It turns out that you can use "Artjuhov's generalized Fermat test"—the "sprp test"—to quickly write down a reasonably short proof that the number is in fact composite. However, unless you're extremely lucky, you won't be able to find all the prime factors of the number, even with today's state-of-the-art factorization methods.

Do we care whether the answer is accompanied by a proof? Is it good enough to have an answer that's always correct but not accompanied by a proof? Is it good enough to have an answer that has never been observed to be incorrect? Consider, for example, the "Baillie-Pomerance-Selfridge-Wagstaff test": if $n \in 3 + 40\mathbf{Z}$ is a prime number then $2^{(n-1)/2} + 1$ and $x^{(n+1)/2} + 1$ are both zero in the ring $(\mathbf{Z}/n)[x]/(x^2 - 3x + 1)$; nobody has been able to find a composite $n \in 3 + 40\mathbf{Z}$ satisfying the same condition, even though such n's are conjectured to exist. (Similar comments apply to arithmetic progressions other than $3+40\mathbf{Z}$.) If both $2^{(n-1)/2} + 1$ and $x^{(n+1)/2} + 1$ are zero, is it unacceptable to claim that n is prime?

Do we actually want to find *all* the prime factors of the input? Or are we satisfied with *one* prime divisor? Or *any* factorization? More than 70% of all integers n are divisible by 2 or 3 or 5, and are therefore very easy to factor if we're satisfied with *one* prime divisor. On the other hand, some integers n have the form pq where p and q are primes; for these integers n, finding one factor is just as difficult as finding the complete factorization.

Do we want to be able to find the prime factors of every integer n? Or are we satisfied with an algorithm that gives up when n has large prime factors? Some algorithms don't seem to care how large the prime factors are: for example, "the Pollard-Buhler-Lenstra-Pomerance-Adleman number-field sieve" is conjectured to find the prime factors of n using $\exp((64/9 + o(1))^{1/3}(\log n)^{1/3}(\log\log n)^{2/3})$ simple operations. Other algorithms are much faster at finding small primes:

* Date of this document: 2005.12.26.

for example, "Lenstra's elliptic-curve method" is conjectured to find all prime factors $p \leq y$ using $\exp(\sqrt{(2+o(1))(\log y)\log\log y})$ simple operations.

More generally, consider an algorithm that tries to find the prime factors of n, and that has different performance (different speed; different success chance) for different inputs n. Are we interested in the algorithm's performance for *typical* inputs n? Or its *average* performance over all inputs n? Or its performance for *worst-case* inputs n, such as integers chosen by cryptographers to be difficult to factor? Consider, for example, the "Schnorr-Lenstra-Shanks-Pollard-Atkin-Rickert class-group method." This method was originally conjectured to find the prime factors of n using $\exp(\sqrt{(1+o(1))(\log n)\log\log n})$ simple operations, but the conjecture was later modified: the method seems to run into all sorts of trouble when n is divisible by the square of a large prime.

Do we compare algorithms according to their *conjectured* speeds? Or do we compare them according to *proven* bounds? The "Schnorr-Seysen-Lenstra-Lenstra-Pomerance class-group method" is *proven* to find the prime factors of n using at most $\exp(\sqrt{(1+o(1))(\log n)\log\log n})$ simple operations; the number-field sieve is *conjectured* to be much faster, once n is large enough, but we don't even know how to prove that the number-field sieve *works* for every n, let alone that it's fast.

How much parallelism do we allow in our algorithms? One number-field-sieve variant uses $L^{1.18563\ldots+o(1)}$ seconds on a machine costing $L^{0.79042\ldots+o(1)}$ dollars for $L^{0.79042\ldots+o(1)}$ tiny parallel CPUs carrying out a total of $L^{1.97605\ldots+o(1)}$ simple operations; here $L = \exp((\log n)^{1/3}(\log\log n)^{2/3})$. Another variant uses $L^{1.75457\ldots+o(1)}$ seconds on a serial machine costing $L^{1.00573\ldots+o(1)}$ dollars for $L^{1.00573\ldots+o(1)}$ bytes of memory and a serial CPU carrying out $L^{2.01147\ldots+o(1)}$ simple operations. Another variant uses $L^{1.90188\ldots+o(1)}$ seconds on a machine costing $L^{0.95094\ldots+o(1)}$ dollars for $L^{0.95094\ldots+o(1)}$ bytes of memory and a serial CPU carrying out $L^{1.90188\ldots+o(1)}$ simple operations. The first variant is designed to minimize price-performance ratio; the second is designed to minimize price-performance ratio for serial computations; the third is designed to minimize the number of simple operations.

Do we want to find the prime factors of just a single integer n? Or do we want to solve the same problem for many integers $n_1, n_2, \ldots$? Or do we want to solve the same problem for *as many of* $n_1, n_2, \ldots$ *as possible*? One might guess that the fastest way to handle many inputs is to handle each input separately. But the "Sieve of Eratosthenes" finds small factors of many consecutive integers $n_1, n_2, \ldots$ in much less time than handling each integer separately. Furthermore, recent factorization methods have removed the words "consecutive" and "small."

Contents of this course. Even a year-long course can't possibly cover all the interesting factorization methods in the literature. I'm going to focus on "congruence-combination" factorization methods, specifically the number-field sieve, which holds the speed records for real-world factorizations of worst-case inputs such as RSA moduli. Here's how this fits into the spectrum of problems considered above:

- I don't merely want to know that the input n is composite; I want to know its prime factors.
- I'm much less concerned with proving the primality of the factors than with finding the factors in the first place.
- I want an algorithm that works well for every n—in particular, I don't want to give up on n's with large prime factors.
- I want to factor n as quickly as possible. I'm willing to sacrifice proven bounds on performance in favor of reasonable conjectures.
- I'm interested in parallelism to the extent that I have parallel computers.
- I might be interested in speedups from factoring many integers at once, but my primary concern is the already quite difficult problem of factoring a single large integer.

A secondary factorization problem of a quite different flavor turns out to be a critical subroutine in "congruence-combination" algorithms. What these algorithms do is

- write down many "congruences" related to the integer n being factored;
- search for "fully factored" congruences; and then
- combine the fully factored congruences into a "congruence of squares" used to factor n.

The secondary factorization problem is the middle step, the search for "fully factored" congruences. This step involves many inputs, not just one; it involves inputs typically much smaller than n; inputs with large prime factors can be, and are, thrown away; the problem is to find small factors as quickly as possible. I'll present the state of the art in small-factors algorithms, including the elliptic-curve method and a newer method relying on the Schönhage-Strassen FFT-based algorithm for multiplying billion-digit integers. I'll also discuss other tools that show up in the number-field sieve: for example, optimizing the initial choice of congruences requires understanding the distribution of smooth elements of a product of number fields.

There are several books presenting integer-factorization algorithms: Knuth's *Art of computer programming*, Section 4.5.4; Cohen's *Course in computational algebraic number theory*, Chapter 10; Riesel's *Prime numbers and computer methods for factorization*; and, newest and generally most comprehensive, the Crandall-Pomerance book *Prime numbers: a computational perspective*. All of these books also cover the problems of recognizing prime numbers and proving primality. The Crandall-Pomerance book is highly recommended.

Student project. The student project attached to this course is to actually use computers to factor a bunch of integers! The project will take the secondary perspective described above: there are many integers to factor; integers with large prime factors—integers that aren't "smooth"—are thrown away; the problem is to find small factors as quickly as possible. We're going to write real programs, see which programs are fastest, and see which programs are most successful at finding factors.

There are several methods in the literature for finding small factors of one integer:

- Trial division.
- Pollard's ρ method.
- Pollard's fast-factorials method. Skip this one; ρ is faster.
- Pollard's $p-1$ method.
- Williams' $p+1$ method.
- Other "cyclotomic" methods. Skip these; $p-1$ and $p+1$ are faster.
- Lenstra's elliptic-curve method.
- The Lenstra-Pila-Pomerance hyperelliptic-curve method. Skip this one; the elliptic-curve method is faster.
- Early aborts. This is an important modification to all of the above methods, trading success probability for speed.

Students can find pointers to the literature in section 2 of my paper "How to find small factors of integers," `http://cr.yp.to/papers.html#sf`.

There are faster methods of finding small factors of many integers. The state of the art is explained in my paper "How to find smooth parts of integers," `http://cr.yp.to/papers.html#smoothparts`.