Complexity of Factoring Integers

Important for public key cryptography.

Recall RSA: $n = pq$ hard to factor. Choose e with $\gcd(e, (p-1)(q-1)) = 1$, Via extended Euclid, find d with $ed \equiv 1 \pmod{(p-1)(q-1)}$. Discard p and q. Public key is n, e. Private key is d. Encipher m as $c = m^e \bmod n$. Decipher c as $m = c^d \bmod n$.

1. RSA problem: Given n, e, c, find m.

2. Compute d: Given n, e, find d.

3. Factor n: Given n, find p and q.

Clearly, $3 \to 2 \to 1$.

In fact, $3 \equiv 2$. It is not known whether $3 \equiv 1$. All three problems seem hard, although Shor showed that one can solve 3 quickly on a quantum computer.

The factoring problem:

Input: n, a composite positive integer.

Output: m, an integer in $1 < m < n$ with $m \mid n$.

Example: Input $n = 45$, Output $m = 15$.

OR

Input: n, a composite positive integer.

Output: A list of all the prime factors of n.

Example: Input $n = 45$, Output $n = 3 \cdot 3 \cdot 5$.

The primality problem:

Input: n, a positive integer.

Output: Either "n is prime" OR "n is not prime (composite)".

Theorem. If n is composite, then it has a prime factor $p \leq \sqrt{n}$.

Proof. If n is composite, then it at least two prime factors. Let p and q be two of them. Assume $p \leq q$. Then $p^2 \leq pq \leq n$, so $p \leq \sqrt{n}$.

This theorem suggests the Trial Division algorithm to factor n or test n for primality.

Let $p_0 = 2$, $p_1 = 3$, $p_2 = 5$, $p_3 = 7$, ..., be the primes.

For $i = 0$, 1, 2, ..., if p_i divides n, declare n is composite with factor p_i; else if $p_i > \sqrt{n}$, then declare n is prime.

Example: Factor $n = 903$. $p_1 = 3$ divides n, so n is composite. If you want all factors of 903, continue Trial Division with the new $n = 903/3 = 301$. One finds that $p_3 = 7$ divides 301. Continue Trial Division with $n = 301/7 = 43$. Since $7 > \sqrt{43}$, 43 is prime. Finally, $903 = 3 \cdot 7 \cdot 43$.

Trial Division factors n (and also tests whether n is prime) in $O(n^{1/2})$ steps.

This algorithm is deterministic (no random choices) and rigorous (no unproved hypotheses).

But it runs in exponential time in $\log n$, to wit, $\exp(0.5 \ln n) = n^{1/2}$.

A century ago, this is all one could do. A few shortcuts were known, but the time was still $O(n^{1/2})$ (exponential).

All known deterministic (no random choices), rigorous (no unproved hypotheses) factoring algorithms have exponential time. The fastest known ones are due to

Shanks (1971)

Pollard (1974)

Strassen (1976)

All three take $O(n^{1/4})$ steps to factor n.

All three algorithms use complicated higher math, such as the class group of a quadratic field.

Known deterministic (no random choices), NONrigorous (has an unproved hypothesis) factoring algorithms may be faster.

There is a variation of Shanks (1971) that assumes the (widely-believed) Extended Riemann Hypothesis that factors n in $O(n^{1/5})$ steps, still exponential time.

The fastest practical factoring algorithms, CFRAC (1970), QS (1980) and NFS (1990), are deterministic and NONrigorous. They take

$$O(e^{\sqrt{(\ln n)(\ln \ln n)}})$$

steps to factor n, which is subexponential but not polynomial time. The unproved hypothesis they assume is that the probability that certain auxiliary numbers used in the algorithm are as likely to have no large prime factors as random numbers of the same size.

Quadratic Sieve Method

Theorem. If $n = pq$ is the product of two distinct odd primes, and if $x^2 \equiv y^2 \pmod{n}$, but $x \not\equiv \pm y \pmod{n}$, then $\gcd(x + y, n) = p$ or q.

Proof: We are given that n divides $(x+y)(x-y)$ but not $(x+y)$ or $(x-y)$. Hence, one of p, q must divide $(x + y)$ and the other must divide $(x - y)$.

In fact, if n has more than two prime factors and the congruence conditions of the theorem hold, then $\gcd(x+y, n)$ and $\gcd(x-y, n)$ will be proper factors of n even if they are not prime. The conditions fail to lead to a proper factor of n only in case n is a power of a prime.

The quadratic sieve algorithm tries to factor n simply by finding x and y with $x^2 \equiv y^2 \pmod{n}$, ignoring the conditions $x \not\equiv \pm y \pmod{n}$. (It just hopes for the best. Usually, it finds several such pairs x, y. Each pair succeeds in factoring n with probability at least 1/2.)

Definition. An integer k is a *square* if there exists an integer x so that $k = x^2$.

The quadratic sieve method tries to factor n by finding two congruent squares modulo n.

How can one recognize a square?

Multiple choice question:

Which of these numbers is a square?

a. 21
b. 23
c. 25
d. 27
e. 29

Which of these numbers is a square?

a. 431641
b. 431643
c. 431645
d. 431647
e. 431649

This is harder.

Suppose I give you the prime factorizations of the numbers.

Which of these numbers is a square?

a. $431641 = 7^2 \cdot 23 \cdot 383$
b. $431643 = 3 \cdot 143881$
c. $431645 = 5 \cdot 131 \cdot 659$
d. $431647 = 17 \cdot 25391$
e. $431649 = 3^4 \cdot 73^2$

Theorem. If $n = \prod_{i=1}^{k} p_i^{e_i}$ is the prime factorization of n into the product of powers of distinct primes, then n is square if and only if all exponents e_i are even numbers.

The quadratic sieve factoring algorithm finds congruences $x^2 \equiv y^2 \pmod{n}$ as follows.

Generate many “relations” $j^2 \equiv m \pmod{n}$, where m is small and therefore easy to factor. Factor the numbers m and match their prime factors to form a product of some ms in which each prime occurs as a factor an even number of times, so it is a square. Let y^2 be the product of these ms. Let x be the product of the js in the relations used to make y^2. Then x^2 is the product of the j^2s, which is congruent to the the product of the ms. This product is y^2 by the choice of relations.

Example. Let us factor $n = 1649$. Note that $\sqrt{n} \approx 40.6$, so the numbers $41^2 \bmod n$, $42^2 \bmod n$, ..., will be fairly small compared to n. We have

$$41^2 = 1681 \equiv 32 = 2^5 \pmod{1649},$$

$$42^2 = 1764 \equiv 115 = 5 \cdot 23 \pmod{1649},$$

$$43^2 = 1849 \equiv 200 = 2^3 \cdot 5^2 \pmod{1649}.$$

Now $32 \cdot 200 = 2^8 \cdot 5^2 = 80^2$ is a square. Therefore,

$$(41 \cdot 43)^2 \equiv 80^2 \pmod{1649}.$$

Note that $41 \cdot 43 = 1763 \equiv 114 \pmod{1649}$ and that $114 \not\equiv \pm 80 \pmod{1649}$. We get the factors of 1649 from $\gcd(114 - 80, 1649) = 17$ and $\gcd(114 + 80, 1649) = 97$, so
$1649 = 17 \cdot 97$.

In a real application of the quadratic sieve there may be millions of relations $j^2 \equiv m \pmod{n}$ with m factored. How can we efficiently match the prime factors of the ms to make each prime occur an even number of times?

Answer: Use linear algebra over the field $\mathbf{F}_2$ with 2 elements.

Let p_1, p_2, ..., p_b be all of the prime numbers that occur as factors of any of the ms.

If $m = \prod_{i=1}^{b} p_i^{e_i}$, where each exponent $e_i \geq 0$, associate m to the vector

$$v(m) = (e_1, e_2, \ldots, e_b).$$

Multiplying ms corresponds to adding their associated vectors. If $S \subseteq \{1, 2, \ldots, r\}$, where r is the total number of relations, then $\prod_{i \in S} m_i$ is a square if and only if $\sum_{i \in S} v(m_i)$ has all even coordinates.

Reduce the exponent vectors $v(m)$ modulo 2 and think of them as vectors in the b-dimensional vector space $\mathbf{F}_2^b$ over $\mathbf{F}_2 = \{0,1\}$. A vector $v(m)$ has all even coordinates iff it is the 0 vector in $\mathbf{F}_2^b$.

Linear combinations of distinct vectors $v(m)$ correspond to subset sums. Finding a nonempty subset of integers whose product is a square is reduced to finding a linear dependency among the vectors $v(m)$.

We know from linear algebra that if we have more vectors than the dimension b of the vector space ($r > b$, i.e. more relations than primes), then there will be linear dependencies among the vectors.

Also from linear algebra we have efficient algorithms, such as matrix reduction, for finding linear dependencies. Row reduction over $\mathbf{F}_2$ is especially efficient because adding (or subtracting) two rows is the same as finding their exclusive-or.

The analysis of the quadratic sieve algorithm shows that its time complexity to factor n is about

$$e^{\sqrt{(\ln n)(\ln\ln n)}}$$

bit operations.

To understand what this means, consider

$$e^{\sqrt{(\ln n)(\ln\ln n)}} \le e^{\sqrt{(\ln n)(\ln n)}} = e^{\ln n} = n$$

and

$$e^{\sqrt{(\ln n)(\ln\ln n)}} \ge e^{\sqrt{(\ln\ln n)(\ln\ln n)}} = e^{\ln\ln n} = \ln n.$$

Thus, $e^{\sqrt{(\ln n)(\ln\ln n)}} \le n^{\varepsilon}$ for any $\varepsilon > 0$ and $e^{\sqrt{(\ln n)(\ln\ln n)}} \ge (\ln n)^c$ for any constant $c > 0$. That is, the time complexity is subexponential but not polynomial time.

The $\ln \ln n$ in the time complexity comes from the probability of an auxiliary integer m near $\sqrt{n}$ having only small prime factors.

The integer factoring problem is interesting because its complexity does not appear to follow the naive intuition that all problems are either in $\mathcal{P}$ or are $\mathcal{NP}$-hard.

There is a striking difference between what can be achieved in factoring with and without randomness.

The fastest known deterministic rigorous factoring algorithms have exponential time $O(n^{1/4})$.

If one abandons rigor, then deterministic NON-rigorous factoring algorithms are known with subexponential, but not polynomial, run times.

The same run times can be achieved if one keeps rigor but drops determinism and allows random choices, as we will see on the next slide.

Next we consider NONdeterministic (probabilistic) factoring algorithms, that is, those that choose random numbers.

Lenstra's (1985) Elliptic Curve Method is probabilistic because it begins by choosing a random elliptic curve and a random point on it. It is also nonrigorous because it assumes that there is a number with only small prime factors in a short interval. It takes $O(e^{\sqrt{(\ln n)(\ln\ln n)}})$ steps to factor n.

Dixon (1981) invented a rigorous probabilistic algorithm for factoring n with subexponential time $O(e^{\sqrt{(\ln n)(\ln\ln n)}})$. It begins by choosing some random numbers. Then it does a calculation that requires the time just mentioned. With a probability near 1, it will factor n.

The model of computation matters in the analysis of the factoring problem.

Suppose that a single arithmetic operation ($+$, $-$, $\times$, $\div$) is one "step" and each memory location may hold any integer.

Shamir (1979) proved that in this model one can factor n in polynomial time. It performs arithmetic on integers as big as $n!$ (in one step per operation), so is not practical.

Can you prove a lower bound on the complexity of factoring?

Can you even prove that there are infinitely many composite n so that every (deterministic) (rigorous) factoring algorithm runs for at least $\log^2 n$ steps on a computer with fixed-length words and operations on numbers with this fixed length?

Complexity of Primality Testing

The primality problem:

Input: n, a positive integer.

Output: Either "n is prime" OR "n is not prime (composite)".

In 2002, Agrawal, Kayal and Saxena proved that primality testing can be done rigorously in deterministic polynomial time $\mathcal{P}$, but that is not the whole story.

The fastest known rigorous deterministic primality test for n takes $O(\log^6 n)$ steps, too slow for practical use.

For numbers n of special form (like $n = 2^p - 1$ or $n = 2^{2^k} + 1$), rigorous deterministic primality tests with time $O(\log^2 n)$ have been known for a century. The largest known primes have the form $2^p - 1$.

Pratt (1975) proved that every prime has a *succinct certificate* of its primeness. The certificate for n has length $O(\log n)$ and can be verified in $O(\log^2 n)$ time. However, it is as hard to find the certificate for n as it is to factor a number of the size of n. (This proved Primality $\in \mathcal{NP}$.)

Probabilistic Prime Tests

There are several Monte Carlo primality tests (1980) (which always run in polynomial time but might give the wrong answer) that take $O(\log^2 n)$ steps, practical to thousands of decimal digits. Most of these always say that a prime input number is prime, but might incorrectly say that a composite is prime (with tiny probability). (This proved Primality $\in \mathbf{BPP}$.)

There are also several Las Vegas primality tests (1990) (which usually run in polynomial time, always give the right answer, but might run longer than polynomial time for some random choices) that usually take $O(\log^2 n)$ steps, practical to thousands of decimal digits.

Remarks on Discrete Logarithms

Case 1: Given p, g, b, solve

$$g^x \equiv b \pmod{p} \qquad (1)$$

for x.

Case 2: Given g and b in a group G, find an integer x with

$$x \cdot g = b \qquad (2)$$

In Case 1, usually p is prime and g is a generator (primitive root) modulo p.

In Case 2, the group g has addition ($+$) for its operation and $x \cdot g$ means g added to itself (in G) x times. G is typically an elliptic curve or a class group.

Most good methods of factoring integers seem to convert easily to solve Case 1 of discrete logarithms. This means that solving (1) is just about as hard as factoring an integer as large as p. In particular, one can solve (1) in subexponential (but not polynomial) time $O(e^{\sqrt{(\ln p)(\ln\ln p)}})$.

In Case 2, the fast techniques for factoring integers do not seem to apply. If n is the size of the group G, then the fastest known methods to solve (2) take $O(n^{1/4})$ steps, exponential time. These methods are due to Shanks, Pollard and others. The methods may involve kangaroos.

If the parameters of an elliptic curve group are chosen poorly, then the discrete log problem (2) may be converted into an easier problem of type (1) by the MOV attack.