Primality Testing

and

Integer Factorisation

Richard Brent, FAA

Computer Sciences Lab

Australian National University

The Fundamental Theorem of Arithmetic

A positive integer N has a unique prime power decomposition

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

$$(p_1 < p_2 < \ldots < p_k \ \text{primes}, \ \alpha_j > 0)$$

(Gauss 1801, but probably known to Euclid)

The Computational Problem

To compute the prime power decomposition we need :

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

Recursive Algorithm

If N composite, find nontrivial factor f and recursively apply the algorithm to f and N/f

Fermat's Little Theorem

If p is prime and $a \neq 0 \pmod{p}$

then

$$a^{p-1} = 1 \pmod{p}$$

In modern terminology, the ring of residue classes (mod p) is a field.

The converse of Fermat's Theorem is false as

$$a \neq 0 \pmod{p} \text{ and } a^{p-1} = 1 \pmod{p}$$

does not imply that p is prime.

There even exist composite n such that :

$$a^{n-1} = 1 \pmod{n}$$

for all a relatively prime to n

Such n are called *Carmichael Numbers*

e.g. $n = 7.13.19 = 1729$

Primality Testing

We can test a number n for primality by dividing by all primes up to $\sqrt{n}$, but this is too slow.

We would like a *polynomial time* algorithm, i.e. one with guaranteed running time

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

for some constant c, to decide if n is prime.

Use of Fermat's Theorem

We can usually verify that a number n is composite by finding $a < n$ such that

$$a^{n-1} \neq 1 \pmod{n}$$

We can never prove primality this way

A Rigorous Primality Test

To prove that n is prime it is sufficient to find a such that

$$a^{n-1} = 1 \pmod n$$

and

$$a^{j} \neq 1 \pmod n$$

for $1 < j < n$-1

a is called a *primitive root* (mod n)

To verify the second condition it is sufficient to check that

$$a^{(n-1)/p} \neq 1 \pmod n$$

for all prime factor p of n - 1

Problems

1. Need to factorise n - 1 (may be hard)
2. Need to find primitive root a (usually easy)

Avoiding Factorisation of n - 1

If n is prime and

$$n - 1 = 2^{k}q \ (q \text{ odd})$$

then the sequence

$$(a^{q}, a^{2q}, a^{4q}, \ldots, a^{n-1})$$

has the form

$$(1, 1, 1, \ldots, 1)$$

or

$$(?, ?, \ldots, -1, 1, 1, \ldots, 1)$$

when considered mod n (for any a, $1 < a < n$).

Say that n *passes Test(a)* if the sequence

(a^{q} mod n, ...) has the form expected for prime n

Theorem (Rabin)

If n is an odd composite number then the number

of a in the range $1 < a < n$ for which n passes Test(a)

is less than (n - 2)/4

Probabilistic Interpretation

If n is composite and a is chosen randomly then

the probability that n passes Test(a) is less than 1/4

Probabilistic Primality Testing

Given odd $n > 1$, choose $a_1, \ldots, a_m$ independently and randomly from {2, 3, ... , n - 1}.

If n fails Test(a_i) for some i then

n is certainly composite

but if n passes Test(a_i) for $i = 1, \ldots, m$ then

n is probably prime

Formally, the probability that a composite n will wrongly be declared to be prime is less than 4^{-m}

e.g. $m = 10$ $\quad 4^{-m} < 10^{-6}$

$m = 167$ $\quad 4^{-m} < 10^{-100}$

Conclusion

For all practical purposes we can test primality in polynomial time

At this point the ARC might ask

What *use* are large primes ?

Large primes can be used to construct *public-key* cryptosystems (also known as *asymmetric* crypto-systems and *open encryption key* cryptosystems)

Attempts to avoid large primes or their analogues (such as irreducible polynomials) have generally failed to produce secure cryptosystems or have proved to be impractical

Public Key Cryptosystems

[Figure to be drawn by hand here to illustrate sender, encryption, receiver etc.]

B publishes his *public key* (k, N) but keeps his *secret key* k' private

A encrypts a message *M* using (k, N) and sends the encrypted message *C* to *B*

B uses his secret key k' (and N) to retrieve the original message *M*

Trapdoor or One-Way Functions

Let S be a (large) finite set. A *trapdoor* function is an invertible function

$$f: S \to S$$

such that $f(x)$ is *easy*, but $f^{-1}(y)$ is *hard* to compute

Example

$N = p.q$ (a product of two large primes)

$S = \{s \mid 0 < s < N, \text{GCD}(s, N) = 1\}$

$\lambda = \text{LCM}(p - 1, q - 1)$

$k > 1, \quad \text{GCD}(k, \lambda) = 1$

$f(x) = x^k \pmod N$

$f^{-1}(y) = y^{k'} \pmod N$

where

$kk' = 1 \pmod \lambda$

Assumption

Hard to compute k' unless p (or q) is known

Construction of a Trapdoor Function

1. Test sufficiently large random integers using a probabilistic primality test to find primes p', q' such that $p = 2p' + 1$ and $q = 2q' + 1$ are prime

2. Check that $p + 1$ and $q + 1$ each have at least one large prime factor (else go back to step 1)

3. Compute $N = p.q$ and $\lambda = 2p'q'$

4. Choose random k relatively prime to λ (or just choose $k = 3$)

5. Apply the Extended Euclidean algorithm to k and λ to find k', λ' such that $0 < k' < \lambda$ and

$$kk' + \lambda\lambda' = 1$$

6. Destroy all evidence of p, q, λ, λ'

7. Make (k, N) public but keep k' secret

Encryption

The sender splits the message M into blocks of $\lfloor \log_2 N \rfloor$ bits (left-justified), treats each block as integer x in {0, ..., N - 1}, and raises it to the power k (mod N)

$$y = x^k \pmod N$$

The receiver computes

$$x = y^{k'} \pmod N$$

There is an extremely small chance that this fails because GCD (y, N) > 1, i.e. y is divisible by p or q (easy to ensure that this never happens)

Security

There is no known way of *cracking* the system without essentially factorising N. (A Theorem if k = 2)

Note that a knowlege of λ easily gives a factorisation of N, and *vice versa*

Conclusion

Primality testing, integer factorisation,

elementary number theory, elliptic curves and algebraic

numbers turn out to be *useful* in practical applications

as well as *interesting* in their own right

Integer Factorisation Algorithms

There are many algorithms for finding a nontrivial factor f of a composite integer N

Class A

Runtime depends on the size of N but is more or less independent of f

Examples	Runtime
Lehman's Algorithm	$O(N^{1/3})$
Shanks's SQUFOF	$O(N^{1/4})$
Shanks's Class Group Algorithm	$O(N^{1/5+\varepsilon})$
Continued Fraction or MPQS	$O(\exp(c(\log(N)\log\log(N))^{1/2}))$

Class B

Runtime depends mainly on the size of f

Examples	Runtime
Trial division	$O(f.(\log N)^2)$
Pollard *Rho*	$O(f^{1/2}(\log N)^2)$
ECM	$O(\exp(c(\log(f)\log\log(f))^{1/2}).(\log N)^2)$

Pollard's Rho Algorithm

f is a pseudo-random polynomial. In practice we usually take

$$f(x) = x^2 + c \quad (c \neq 0, -2)$$

x_0 is a random starting value.

Compute the sequence (x_0, x_1, ...) where

$$x_{i+1} = f(x_i) \pmod N$$

until

$$\mathrm{GCD}(x_{2i} - x_i, N) > 1$$

If p is the smallest prime factor of N, then probably

$$\mathrm{GCD}(x_{2i} - x_i, N) = p$$

Heuristic Analysis of Expected Runtime

The probability that $x_0, x_1, \ldots, x_k$ are all *distinct (mod p)* is roughly

$$P = (1 - 1/p)(1 - 2/p)\ldots(1 - k/p)$$

(compare birthday paradox *with p = 365)*
so

$$\ln P \sim -k^2/(2p)$$

and the expected number of f evaluations is $O(p^{1/2})$

Each iteration involves operations on numbers of order N^2, so time $O((\log N)^2)$ (we can avoid most of the GCDs)

Thus the expected runtime is $O(p^{1/2}.(\log N)^2)$

Example

$$F_8 = 2^{256} + 1 = 1238926361552897.p_{62}$$

[Brent and Pollard, 1980]

I am now entirely persuaded to employ the method,

a handy trick, on gigantic composite numbers

The Advantage of a Group Operation

The Pollard rho algorithm takes

$$x_{i+1} = f(x_i)$$

Suppose instead that

$$x_{i+1} = x_1 * x_i$$

where * is an *associative* operator, i.e.

$$x * (y * z) = (x * y) * z$$

Then we can compute x_n in $O(\log n)$ steps by the *binary powering* method,

e.g. $x_2 = x_1 * x_1$

$x_4 = x_2 * x_2$

$x_8 = x_4 * x_4$

$x_9 = x_1 * x_8$

Computation of the Identity (mod *p*)

Let m be the product of all prime powers less than some bound B. If the cyclic group <x_1> has order g which is sufficiently *smooth*, then g is a divisor of m and

$$x_1^m = I \quad \text{(the identity)}$$

Why is this useful ?

The group is defined mod p but we work mod N since p is an unknown divisor of N. This can be considered as using a *redundant* representation for group elements.

When we compute I its representation mod N may allow us to compute p via a GCD computation.

Example 1 - Pollard's *p* - 1 Algorithm

Here * is just multiplication (mod p) so $g \mid p - 1$ and

$$x_1^m = I \quad \text{means} \quad x_1^m = 1 \pmod p$$

so

$$p \mid \mathrm{GCD}(x_1^m - 1, N)$$

The worst case

$p - 1 = 2.prime$ is possible, and in this case we need $B \geq p/2$, so there are of order p group operations.

However, the worst case does not always occur - we may be lucky.

Lucky example

$$p = 1155685395246619182673033 \mid M_{257} = 2^{257} - 1$$

$$p - 1 = 2^3.3^2.19^2.47.67.257.439.119173.1050151$$

[Baillie]

Example 2 - Lenstra's Elliptic Curve Method (ECM)

ECM is an improvement over the Pollard p - 1 algorithm because different groups can be selected until we find one whose order is sufficiently smooth (i.e. has no large prime factors)

Geometry of Elliptic Curves

An elliptic curve is defined by a cubic polynomial in two variables. By rational transformations it can be reduced to the *Weierstrass normal form*

$$y^2 = x^3 + ax + b$$

An Abelian group (G, *) can be defined as shown -

[Equations to be inserted by hand here]

Algebraic Definition of *

If $P_i = (x_i, y_i)$ for $i = 1, 2, 3$

and $P_3 = P_1 * P_2$

then $x_3 = \lambda^2 - x_1 - x_2$

$y_3 = \lambda(x_1 - x_3) - y_1$

where
$$\lambda = \begin{cases} (3x_1^2 + a)/(2y_1) & \text{if } P_1 = P_2 \\ (y_1 - y_2)/(x_1 - x_2) & \text{otherwise} \end{cases}$$

Instead of considering operations in R we may consider operations in a finite field, e.g. F_p

Then $p + 1 - 2p^{1/2} < g < p + 1 + 2p^{1/2}$

Since p is unknown we work mod N and detect p as a nontrivial GCD when attempting to compute an inverse (consider $x_1 = x_2$ (mod p) in the definition of λ)

Some Factors found by ECM

$c_{101} = (467^{41} - 1)/(466.1022869)$

$= 4089568263561830388113662969166474269.p_{65}$

[Brent, Cohen and te Riele - 10^{300} opn proof]

$F_{11} = 2^{2048} + 1$

$= 319489.974849.$
$167988556341760475137.$
$3560841906445833920513.p_{564}$ [Brent, 1988]

The Idea of the Quadratic Sieve Method

If we can generate a *nontrivial* relation

$$x^2 = y^2 \pmod N$$

then provided $x \neq \pm y \pmod N$ the computation of

$$\mathrm{GCD}(x - y, N)$$

gives a nontrivial factor of N

How to find *x*, *y*

Several algorithms generate *relations* of the form

$$u^2 = v^2 w \pmod N$$

where w is in a small set of primes (the *factor base*).

Once enough such relations have been found, Gaussian elimination in F_2 finds a subset of relations whose product has only even exponents.

Example of Factorisation by MPQS

$c_{103} = (2^{361} + 1)/(3.174763) =$

$6874301617534827509350575768454356245025403.p$

[Lenstra, Manasse *et al*, 1989]

Corollary

The composite number N used in the RSA

cryptosystem should have more than 100 decimal digits

The Number Field Sieve (NFS)

Our numerical examples have all involved numbers of the form

$$N = a^n \pm b$$

for *small* a and b, although the factorisation algorithms did not take advantage of this special form.

The Number Field Sieve *does* take advantage of such a special form. It is similar to the Quadratic Sieve algorithm but works over an algebraic number field defined by a, n, and b (impractical unless a and b are small).

Its conjectured runtime is

$$O(\exp(c(\log N)^{1/3}(\log \log N)^{2/3}))$$

which is asymptotically better than the

$$O(\exp(c(\log N)^{1/2}(\log \log N)^{1/2}))$$

for algorithms such as MPQS (though the constants c may differ).

Example

Using $Q((-8)^{1/5})$, the *138-digit* number $(2^{457} + 1)/3$ was split into 49-digit and 89-digit factors,

$$p_{49} = 688535...729193$$

[Lenstra, Lenstra, Manasse and Pollard, 1989]