Fast Modular Arithmetic
Probabilistic Algorithms for Primality Testing
Public Key Cryptography

Jan van Eijck
CWI & ILLC, Amsterdam

Specification and Testing, Week 6, 2014

Abstract

Fast modular arithmetic allows us to generate and recognize large primes efficiently, but we need the concept of a probabilistic algorithm.

Given two large primes p and q, computing their product pq is easy, but finding the two factors from the semi-prime pq is (believed to be) hard. No fast algorithm for this is known.

These facts have important consequences for cryptography. The lecture gives an introduction to fast modular arithmetic, to probabilistic algorithms for primality testing, and to their application in public key cryptography. The lecture also prepares for the lab exercises of this week.

For background, see also [1, Chapter 33], on number-theoretic algorithms.

Module Declaration

```
module Week6

where

import Data.List
import System.Random
```

Factorization

Here is a naive factorization algorithm:

```
factors_naive :: Integer -> [Integer]
factors_naive n = factors' n 2 where
  factors' 1 _ = []
  factors' n m
    | n `mod` m == 0 = m : factors' (n `div` m) m
    | otherwise      =     factors' n (m+1)
```

This gives:

```
*Week6> factors_naive 362880
[2,2,2,2,2,2,2,3,3,3,3,5,7]
*Week6> factors_naive 524287
[524287]
```

Improvement

This can be improved slightly by only trying candidate factors that are primes:

```
factors :: Integer -> [Integer]
factors n = let
   ps = takeWhile (\m -> m^2 <= n) primes
 in factors' n ps where
   factors' 1 _  = []
   factors' n [] = [n]
   factors' n (p:ps)
    | n `mod` p == 0 = p: factors' (n `div` p) (p:ps)
    | otherwise      =    factors' n ps
```

The Sieve . . .

This uses the list of prime numbers, as generated by the Sieve of Eratosthenes:

```
primes = sieve [2..]
sieve (n:ns) = n : sieve
    (filter (\ m -> rem m n /= 0) ns)
```

Comparison

To compare the two versions, try this out with:

```
*Week6> map factors [m8..]

*Week6> map factors_naive [m8..]
```

Here `m8` is the eighth Mersenne prime (see below).

Despite this improvement, it is generally believed that factorization of numbers with large factors is hard.

No efficient algorithm for factorization is known. All existing methods use trial division with large numbers of candidates.

Primality Testing Using Factorisation

The following test is 100 percent reliable but inefficient:

```
isPrime :: Integer -> Bool
isPrime n = factors n == [n]
```

Try this out . . .

```
Prelude Week6> isPrime m8
True
Prelude Week6> isPrime m9
```

Mersenne Primes

Father Marin Mersenne (1588–1647) was a French priest and amateur mathematician. He discovered some large primes by studying numbers of the form $2^p - 1$. Such primes are called Mersenne primes.

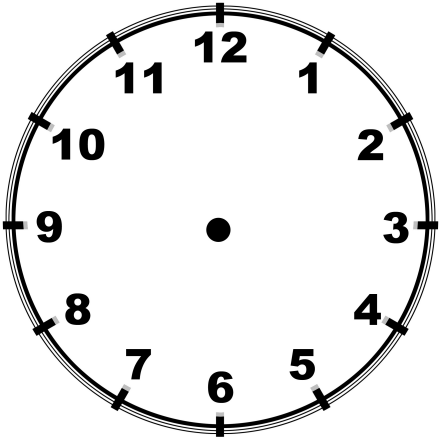
The first 25 Mersenne primes

```
m1  = 2^2-1;     m2  = 2^3-1;      m3  = 2^5-1
m4  = 2^7-1;     m5  = 2^13-1;     m6  = 2^17-1
m7  = 2^19-1;    m8  = 2^31-1;     m9  = 2^61-1
m10 = 2^89-1;    m11 = 2^107-1;    m12 = 2^127-1
m13 = 2^521-1;   m14 = 2^607-1;    m15 = 2^1279-1
m16 = 2^2203-1;  m17 = 2^2281-1;   m18 = 2^3217-1
m19 = 2^4253-1;  m20 = 2^4423-1;   m21 = 2^9689-1
m22 = 2^9941-1;  m23 = 2^11213-1;  m24 = 2^19937-1
m25 = 2^21701-1
```

$2^n - 1$ can only be prime if n is also prime (Exercise 3.36 in The Haskell Road).

Print out `m25` to see that it is a huge beast.

Modular Arithmetic



$$11 + 4 = 15 \equiv 3 \pmod{12}.$$

Operations for modular addition, multiplication

Modular addition:

```
addM :: Integer -> Integer -> Integer -> Integer
addM x y = rem (x+y)
```

Modular multiplication:

```
multM :: Integer -> Integer -> Integer -> Integer
multM x y = rem (x*y)
```

Modular division, modular inverse

Modular division is the same as multiplication by modular inverse. Modular inverses only exist for numbers that are co-prime with their modulus.

```
invM :: Integer -> Integer -> Integer
invM x n = let
    (u,v) = fct_gcd x n
    copr  = x*u + v*n == 1
    i     = if signum u == 1 then u else u + n
 in
    if copr then i else error "no inverse"
```

This uses the extended Euclidean algorithm for GCD.

The Extended Euclidean Algorithm

```
fct_gcd :: Integer -> Integer -> (Integer,Integer)
fct_gcd a b =
  if b == 0
  then (1,0)
  else
     let
       (q,r) = quotRem a b
       (s,t) = fct_gcd b r
     in (t, s - q*t)
```

Modular Exponentiation

```
expM ::  Integer -> Integer -> Integer -> Integer
expM x y = rem (x^y)
```

This is not efficient, for we first compute x^y, and then reduce the result modulo N.

Instead, we should have performed the intermediate computation steps for x^y modulo N.

Modular Exponentiation, Fast Version

One of your exercises for this week's lab session is to implement a function that does modular exponentiation of x^y in polynomial time, by repeatedly squaring modulo N.

E.g., $x^{33} \bmod 5$ can be computed by means of

$$x^{33} \bmod 5 = x^{32} \bmod 5 \times x \bmod 5.$$

$x^{32} \pmod N$ is computed in five steps by means of repeatedly squaring modulo N:

$$x \bmod N \to x^2 \bmod N \to x^4 \bmod N \to \ldots \to x^{32} \bmod N.$$

```
exM :: Integer -> Integer -> Integer -> Integer
exM = expM -- to be replaced by a fast version
```

Fermat's Test for Primality

Pierre de Fermat

Primality testing using a probabilistic algorithm is based on efficient exponentiation modulo.

This uses a theorem by the famous French mathematician Pierre de Fermat (1601–1665).

Fermat's Little Theorem

The following is called the Little Theorem of Fermat, to distinguish it from Fermat's famous Last Theorem[1]

Theorem 1 (Fermat) *If p is prime, then for every $1 \leq a < p$:*

$$a^{p-1} \equiv 1 \pmod{p}.$$

[1] Fermat's Last Theorem says that $x^n + y^n = z^n$ has no non-zero integer solutions for natural numbers $n > 2$. Fermat had stated this without proof. A proof was found by Andrew Wiles in 1995.

Some Examples

To see what Fermat's Little Theorem says, look at some examples: Assume $p = 5$. Let us check 2^4, 3^4 and 4^4.

$$2^4 = 16 \equiv 1 \pmod 5, 3^4 = 81 \equiv 1 \pmod 5, 4^4 = 256 \equiv 1 \pmod 5.$$

Next, use Haskell to check for a larger prime:

```
*Week6> [ a^28 `mod` 29 | a <- [1..28] ]
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
```

It turns out that for every $1 \leq a < p$, the set of remainders modulo p is equal to their product with a modulo p. In other words, multiplying the set $\{1, \ldots, p-1\}$ with a modulo p is simply to permute the set.

Let's try this out with Haskell:

```
Prelude Week6> [n `mod` 13 | n <- [1..12] ]
[1,2,3,4,5,6,7,8,9,10,11,12]
Prelude Week6> [2*n `mod` 13 | n <- [1..12] ]
[2,4,6,8,10,12,1,3,5,7,9,11]
Prelude Week6> [3*n `mod` 13 | n <- [1..12] ]
[3,6,9,12,2,5,8,11,1,4,7,10]
Prelude Week6> [4*n `mod` 13 | n <- [1..12] ]
[4,8,12,3,7,11,2,6,10,1,5,9]
Prelude Week6> [5*n `mod` 13 | n <- [1..12] ]
[5,10,2,7,12,4,9,1,6,11,3,8]
Prelude Week6> [6*n `mod` 13 | n <- [1..12] ]
[6,12,5,11,4,10,3,9,2,8,1,7]
Prelude Week6> [7*n `mod` 13 | n <- [1..12] ]
[7,1,8,2,9,3,10,4,11,5,12,6]
Prelude Week6> [8*n `mod` 13 | n <- [1..12] ]
[8,3,11,6,1,9,4,12,7,2,10,5]
Prelude Week6> [9*n `mod` 13 | n <- [1..12] ]
[9,5,1,10,6,2,11,7,3,12,8,4]
```

Take the case with $p = 13$. Multiplying the numbers in the set

$$\{1, \ldots, 12\}$$

and the numbers in the set

$$\{5n \bmod 13 \mid n \in \{1, \ldots, 12\}\}$$

we get the same outcome:

```
*Week6> product [1..12]
479001600
*Week6> product (map (\ n -> 5*n `mod` 13) [1..12])
479001600
```

Dividing both sides by 12!, this gives $5^{12} \equiv 1 \bmod 13$.

Fermat's Little Theorem: Proof

Now for the general case. We have to show that if the numbers in the set

$$S = \{1, \ldots, p-1\}$$

get multiplied by a modulo p, the resulting numbers are distinct and $\neq 0$. So let $i \neq j \in S$, and consider ai and aj. Suppose $ai \equiv aj \pmod{p}$. Then $i \equiv j \pmod{p}$ and contradiction with $i \neq j$. So ai and aj are distinct modulo p. If $ai \equiv 0$ then, since a and p are relatively prime, we can divide by a and get $i \equiv 0$, and contradiction. So the resulting numbers are $\neq 0$. This means the result of multiplying the numbers in S with a modulo p is a permutation of S.

This gives

$$S = \{1, \ldots, p-1\} = \{a \times 1 \bmod p, \ldots, a \times p - 1 \bmod p\}.$$

Multiplying the numbers left and right gives:

$$(p-1)! = a^{p-1} \times (p-1)! \pmod{p}.$$

We can divide both sides by $(p-1)!$ because $(p-1)!$ and p are relatively prime. This gives $a^{p-1} \equiv 1 \pmod{p}$.

Fermat's Algorithm for Primality Testing

Fermat's Test

1. Pick a with $1 < a < N$ at random.
2. Compute $a^{N-1} \pmod{N}$ using fast exponentiation.
3. If the outcome is 1, output "Probably Prime",
   otherwise output "Composite".

If N is indeed prime then $a^{N-1} \equiv 1 \pmod{N}$, and the test works fine.

But if N is composite, it may still happen that $a^{N-1} \equiv 1 \pmod{N}$, for Fermat's Little Theorem does not specify what happens for composite numbers ...

Implementation

Fermat's algorithm yields the following primality test:

```
prime_test_F :: Integer -> IO Bool
prime_test_F n = do
   a <- randomRIO (1, n-1) :: IO Integer
   return (exM a (n-1) n == 1)
```

Improving the Fermat Algorithm

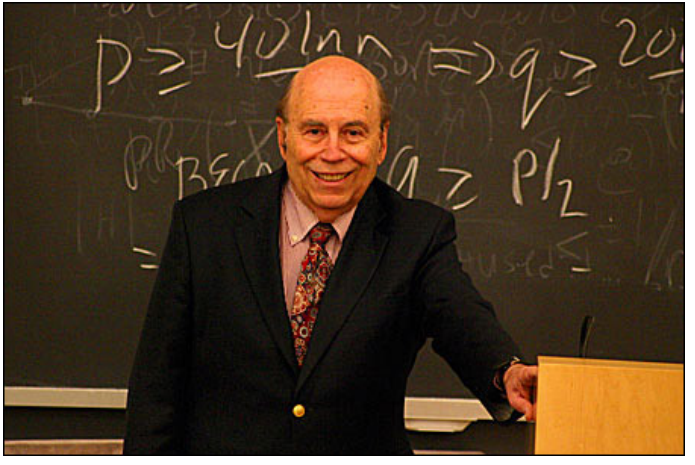
A better prime test results if we try out more candidates for a:

```
primeF :: Int -> Integer -> IO Bool
primeF _ 2 = return True
primeF 0 _ = return True
primeF k n = do
--   a <- randomRIO (1, n-1) :: IO Integer
   a <- randomRIO (1, n-2) :: IO Integer
   if (exM a (n-1) n /= 1)
      then return False
      else primeF (k-1) n
```

There remains a possibility of getting false positives . . .

One of your tasks for this week will be to test this function.

An Addition to the Fermat Test: Miller and Rabin



Gary Miller – Michael Rabin

Unfortunately, there are numbers that can fool the Fermat test, so called Carmichael numbers. You will encounter Carmichael numbers in the lab exercises.

Miller and Rabin [3, 4] propose to add a further test.

Basis of the Miller and Rabin Test

Lemma 2 *If n is a prime, then any number x with the property that $x^2 = 1 \bmod n$ has to satisfy $x = 1 \bmod n$ or $x = -1 \bmod n$.*

Proof. To see why this has to hold, notice that from $x^2 = 1 \bmod n$ it follows that $x^2 - 1 = (x-1)(x+1) = 0 \bmod n$. This means that n divides $(x-1)(x+1)$. Since n is prime, it follows from this (by a lemma called Euclid's Lemma) that n has to divide either $x - 1$ or $x + 1$. In the former case we have $x - 1 = 0 \bmod n$, i.e., $x = 1 \bmod n$. Int the latter case we have $x = -1 \bmod n$. □

Note that $-1 = n - 1 \bmod n$.

Euclid's Lemma

Lemma 3 (Euclid's Lemma) *If p is prime and $p \mid ab$ then either $p \mid a$ or $p \mid b$.*

Proof. Observe that $p \nmid a$ implies that p and a are co-prime.

So suppose $p \mid ab$ and $p \nmid a$. Then, by Euclid's extended GCD algorithm, there are $x, y \in \mathbb{Z}$ with $xa + yp = \gcd(a, p) = 1$. Multiplying both sides by b gives $xab + ypb = b$. Since it is given that $p \mid ab$ we get that $p \mid b$. $\square$

Write n as $2^r \times s$

The following function factors out powers of 2:

```
decomp :: Integer -> (Integer,Integer)
decomp n = decomp' (0,n) where
  decomp' = until (odd.snd) (\ (m,n) -> (m+1,div n 2))
```

This gives:

```
*Week6> decomp 53248
(12,13)
*Week6> 2^12 * 13
53248
```

The Miller-Rabin Primality Test

```
primeMR :: Int -> Integer -> IO Bool
primeMR _ 2 = return True
primeMR 0 _ = return True
primeMR k n = let
    (r,s) = decomp (n-1)
    f = \ x -> takeWhile (/= 1)
        (map (\ j -> exM x (2^j*s) n)  [0..r])
  in
   do
    a <- randomRIO (1, n-1) :: IO Integer
    if exM a (n-1) n /= 1
      then return False
      else
        if exM a s n /= 1 && last (f a) /= (n-1)
          then return False
          else primeMR (k-1) n
```

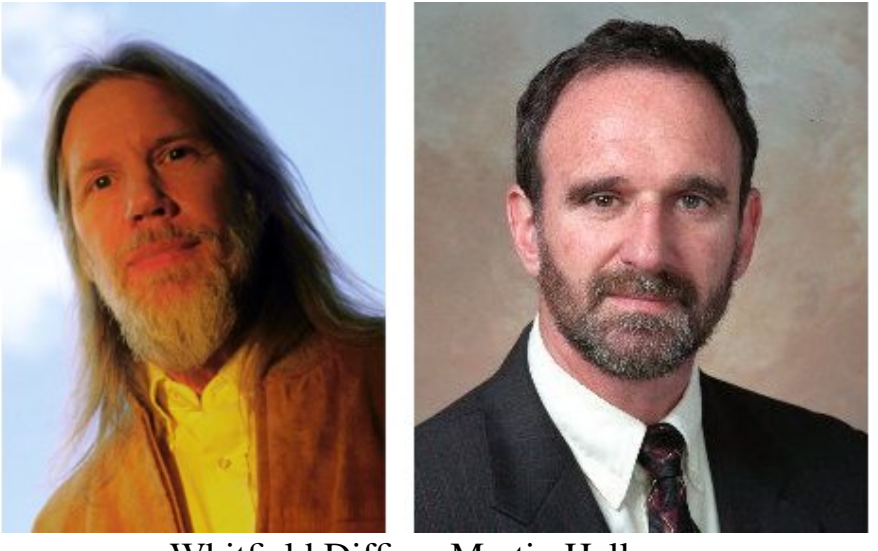
Testing the Primality Tests

For testing our primality test algorithms the list of prime numbers generated by Eratosthenes' sieve is useless, for the algorithms all correctly classify the primes as primes. Where they can go wrong is on classifying composite numbers; these can slip through the Fermat test, and also through the Rabin/Miller test, although the probability of this can be made arbitrarily small.

One of your exercises for this week will be to write a function for generating the composite numbers, for testing purposes.

Application to Cryptography

The idea to base public key cryptography on the fact that finding large primes, multiplying them, and exponentiation modulo a prime are easy while factoring numbers that are multiples of large primes, and finding x from $x^a \mod p$ (taking the discrete logarithm) are hard was put forward by Diffie and Hellman in [2].



Whitfield Diffie – Martin Hellman

Diffie-Hellman Key Exchange Protocol

Key Exchange Over Insecure Channel

1. Alice and Bob agree on a large prime p and a base $g < p$ such that g and $p - 1$ are co-prime.
2. Alice picks a secret a and sends $g^a \bmod p = A$ to Bob.
3. Bob picks a secret b and sends $g^b \bmod p = B$ to Alice.
4. Alice calculates $k = B^a \bmod p$.
5. Bob calculates $k = A^b \bmod p$.
6. They now have a shared key k. This is because $k = (g^a)^b = (g^b)^a \bmod p$.

Use of a Shared Secret Key for Secure Communication

Let p be the prime that Alice and Bob have agreed on, and let k be their shared key. Then message m is encoded as

$$m \times k \bmod p.$$

```
encodeDH :: Integer -> Integer -> Integer -> Integer
encodeDH p k m = m*k `mod` p
```

Such messages can be decoded by both Alice and Bob.

Alice knows p, k, g^b and a. She decodes cipher c with

$$c \times (g^b)^{(p-1)-a} \bmod p.$$

Bob knows p, k, g^a and b. He decodes cipher c with

$$c \times (g^a)^{(p-1)-b} \bmod p.$$

What is behind this is again Fermat's Little Theorem

We have:

$$(g^a)^{(p-1)-b} = g^{a((p-1)-b)} = g^{a(p-1)} \times g^{-ab} = (g^{p-1})^a \times g^{-ab}$$
$$\overset{Fermat}{=} 1^a \times g^{-ab} = g^{-ab} \bmod p,$$

and therefore:

$$c \times (g^a)^{(p-1)-b} = (m \times g^{ab}) \times g^{-ab} = m \times (g^{ab} \times g^{-ab}) = m \bmod p.$$

This gives:

```
decodeDH :: Integer -> Integer -> Integer
         -> Integer -> Integer -> Integer
decodeDH p k ga b c = let
    gab' = exM ga ((p-1)-b) p
  in
    rem (c*gab') p
```

Symmetric Key Cyphers

This is called a symmetric key cypher.

Alice and Bob use the same key to encrypt, and a small variation on this key, together with the private information they have, to decrypt.

Symmetric Key Cyphers Using Fast Exponentiation Modulo

```
encode :: Integer -> Integer -> Integer -> Integer
encode p k m = let
   p' = p-1
   e  = head [ x | x <- [k..], gcd x p' == 1 ]
 in
   exM m e p

decode :: Integer -> Integer -> Integer -> Integer
decode p k m = let
   p' = p-1
   e  = head [ x | x <- [k..], gcd x p' == 1 ]
   d  = invM e p'
 in
   exM m d p
```

Efficient With Key, Hard Without

Finding appropriate e and d for coding and decoding when p and k are known is fast.

Suppose one were allowed to keep the modulus p and the lower bound for the exponent k hidden in the above coding function. Then p, k is the key to both coding and decoding.

If p, k are both known, coding and decoding are both easy (see above). If p, k are unknown, both coding and decoding are hard.

```
cipher :: Integer -> Integer
cipher = encode secret bound

decipher :: Integer -> Integer
decipher = decode secret bound
```

Look at this

```
*Week6> cipher 123
599196956189954215039167563975032860018984879479201629372399324282200943813364288668253006965436156327542906832599325957585106515246102899061628169372657254294820399394781440125853348050039843242822408715915367029577935707699695635779045758153995268595765059370929872334822799196956189954215039167563975032860018984879479201629372399324282200943813364288668253006965436156327542906832599325957585106515246102899061628169372657254294820399394781440125853348050039843242822408715915367029577935707699695635779045758153995268595765059370929872334822798723348227
123
```

A Problem

There is a problem, however. In order to construct the function `cipher`, the secret prime and the bound for the exponent have to be available, so we cannot make the full recipe for constructing the encoding function publicly available.

This can be solved by a variation on the Diffie-Hellman key exchange protocol.

In asymmetric public key cryptography, it turns out we can do much better than this. We can generate a key and make it publicly available, and from that key anyone can construct an encoding function. Decoding, given only this public key information, is generally thought to be hard.

RSA = Rivest, Shamir, Adleman

Len Adleman – Adi Shamir – Ron Rivest

RSA public key cryptography [5]; generation of a public key

```
rsa_public :: Integer -> Integer -> (Integer,Integer)
rsa_public p q = let
   n   = p * q
   phi = (p-1)*(q-1)
   e   = head [ x | x <- [3..], gcd x phi == 1 ]
 in
   (e,p*q)
```

Here p and q are supposed to be large primes with the same bitlength.

RSA public key cryptography; generation of a private key

Let p, q be large primes. Let $n = pq$.

If (e, n) is the public key pair, where n is the semi-prime given by $n = pq$, and d is the inverse of e modulo $(p-1)(q-1)$, then (d, n) is the private key.

It is believed that the private key is hard to compute from the public key (in the sense that nobody knows how to do this, and it is suspected that it is impossible to find an efficient algorithm for this).

```
rsa_private ::  Integer -> Integer
                -> (Integer,Integer)
rsa_private p q = let
   n = p * q
   phi = (p-1)*(q-1)
   e = head [ x | x <- [3..], gcd x phi == 1 ]
   d = invM e phi
  in
   (d,p*q)
```

RSA encoding using a generated key pair

```
rsa_encode :: (Integer,Integer) -> Integer -> Integer
rsa_encode (e,n) =  \ m -> exM m e n
```

RSA decoding using a private key

Just use the private key to encode again.

```
rsa_decode = rsa_encode
```

This gives:

$$(m^e)^{e^{-1}} \bmod n = m.$$

A genuine trapdoor function


```
trapdoor :: (Integer,Integer) -> Integer -> Integer
trapdoor = rsa_encode
```

Your Tasks for This Week

- Implement fast modular exponentiation. This is one of the key ingredients to primality testing and public key cryptography.
- Test your function for fast modular exponentiation.
- Write a function for generating the composite numbers
- Use this to test Fermat's Primality Algorithm
- Test Fermat's Primality Algorithm for Carmichael numbers
- Use the Miller-Rabin primality check to find large Mersenne primes. Next, check on internet whether the numbers that you found are genuine Mersenne primes
- **Bonus** Use the Miller-Rabin primality check to find large pairs of primes and demonstrate how these are used in RSA cryptography.

References

[1] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. Introduction to Algorithms. MIT Press, 1997.

[2] W. Diffie and M. Hellman. New directions in cryptography. IEEE Transactions on Information Theory, 22(6):644–654, 1976.

[3] Gary L. Miller. Riemann's hypothesis and tests for primality. Journal of Computer and System Sciences, 13(3):300–317, 1976.

[4] Michael O. Rabin. Probabilistic algorithm for testing primality. Journal of Number Theory, 12(1):128–138, 1980.

[5] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM, 21(2):120–126, February 1978.