Fast Modular Arithmetic and Public Key Cryptography

Back to primes

- Fast modular arithmetic → generate and recognize large primes efficiently
- We need a *probabilistic algorithm* for primality testing
- Computing the product of two large primes is easy.
- Finding the two large prime factors from a semiprime is (still considered) hard.
- Important application in public key cryptography.

Integer factorization

- Naive: iterate over [1,..,n]
- Better: iterate over {p | prime p, p in [1,...,n]}
  - Compare the approaches (mers 8 returns the 8th Mersenne prime)
    ```
    *Lecture6> map factorsNaive [mers 8..]
    *Lecture6> map factors [mers 8..]
    ```
- No polynomial time algorithm (yet)
- Currently: trial division with a large set of candidates
  - semiprimes (product of two primes) are hardest instances
- Reliable, but inefficient → can we improve?
  - Yes, for primality testing
  - Not (yet), for integer factorization

Mersenne primes

Marin Mersenne - 1600, French, priest and mathematician

- Studied 2^p-1 numbers
- If 2^p-1 is prime, p is prime
  - Assume p is not prime → p = ab
  - Take $x = 2^b-1$, $y = 1+2^b+2^{2b}+...+2^{(a-1)b}$
  - $x*y=2^by-y=2^{ab}-1$ → $2^{ab}-1$ is composite

Modular Arithmetic

- Int data ranges from -2^{63} to $2^{63}-1$
  - Stored modulo 2^{64}
  - 2’s complement: for negative numbers, put a 1 at the sign position, flip all the bits, and add 1

```
*Lecture6> maxBound :: Int
9223372036854775807
*Lecture6> 2^63-1
9223372036854775807
*Lecture6> minBound :: Int
-9223372036854775808
*Lecture6> -2^63
-9223372036854775808

*Lecture6> 2^64-1
18446744073709551615
*Lecture6> 2^64-1 :: Int
-1
```

Modular operations

- Addition → `addM x y = rem (x+y)`
- Multiplication → `multM x y = rem (x*y)`
  - `(x mod a) * (y mod a) = (x*y mod a)`
- Inverse
  - Every “equivalence” class [a] for Z/n except [0] has an inverse, [b]
    - [a]*[b]=1 → [a*b]=1 → a*b=1+k*n → a*b - k*n = 1 → (a,k), **(a,n)**, (b,k), (b,n) coprimes
  - Q: How about Z (integer set)?
- Division
  - Multiplication by modular inverse → x has an modular inverse if it is coprime with the modulus
    - gcd(x,n) = 1 iff there are u,v such that ux + vn = 1
    - We use the extended Euclidean algorithm to find u,v

Extended Euclidean Algorithm

- Standard: only keeps remainders → gcd (q*b+r,b) = gcd (b,r)
- Extended: keep also quotients → fctGcd(q*b+r,b) = fctGcd (b,r)

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

We now can write coprime as:

```
coprime n m = fGcd n m == 1
```

or

```
coprime' n m = let (x,y) = fctGcd n m
               in x*n + y*m == 1
```

Modular Exponentiation x^y mod n

- Primality testing using a probabilistic algorithm is based on efficient modulo exponentiation.
- Naive: `expM x y =  rem (x^y)`
- Faster: repeatedly squaring modulo n: $[x] \rightarrow [x^2] \rightarrow [x^4] \rightarrow \ldots$
  - One of your lab exercises is to implement exM, the faster version of expM
  - Based on the idea x^2 mod n = (x mod n) * (x mod n)

Fermat's primality test

- Pierre de Fermat: 1600, French, mathematician, not a priest, but a lawyer
  - Little Theorem of Fermat: if p is prime, then for any integer a with $1 \le a < p$, $a^{p-1} \equiv 1 \bmod p$
  - Proof is based on the fact that the list [[a],[2a]...[(p-1)a]] (whose product is $a^{p-1}(p-1)! \bmod p$) is a permutation of [[1],...,[p-1]] → and this is true because a and p are coprimes
    - $(p-1)! = a^{p-1}(p-1)! \bmod p \rightarrow 1 = a^{p-1} \bmod n$, because (p-1)! and p are coprimes

- Fermat's algorithm for primality testing
  - Pick $1 < a < n$
  - Compute $a^{n-1} \bmod n$ using fast exponentiation (exM)
  - If outcome is 1 → output "Probably prime", else → output "Composite"
  - Trying more candidates increases the accuracy of your test, however increases the runtime

```
primeTestsF :: Int -> Integer -> IO Bool
primeTestsF k n = do
 as <- sequence $ fmap (\_-> randomRIO (2,n-1)) [1..k]
 return (all (\ a -> exM a (n-1) n == 1) as)
```

Fooling the Fermat test

- Carmichael numbers → see lab exercises
  - $a^{p-1} \equiv 1 \bmod p$ is also true for some composite p numbers
- Miller and Rabin proposed a further test
  - Miller proposed a deterministic test whose correctness depends on an unproven hypothesis
  - Rabin modified it to a probabilistic algorithm
  - If n is prime → any x with $x^2 = 1 \bmod n$ has to satisfy $x = 1 \bmod n$ or $x = -1 \bmod n$ (Euclid's Lemma)
  - Factoring out powers of 2

```
decomp :: Integer -> (Integer,Integer)
decomp n0 = decomp' (0,n0) where
  decomp' = until (odd.snd) (\ (m,n) -> (m+1,div n 2))
```

- Find a non-trivial square root of 1 mod n
  - let $q = 2^r$, $n-1 = q*s$ and pick $1<a<n$ in $a^s \bmod n, a^{2s} \bmod n, \ldots ,a^{q*s} \bmod n$ → if the list does not end in 1, a^{n-1} fails the Fermat test → n is composite
  - $a^{pow(2,u)*s}$ is a nontrivial square root for 1 mod n if $a^{pow(2,u)*s} \neq n-1$ and $a^{pow(2,u)*s} \neq 1$ and $a^{pow(2,u+1)*s} = 1$

Miller-Rabin Compositeness Test

- If the mrComposite x n test succeeds for some x, 2≤x<n, then n is composite

```
mrComposite :: Integer -> Integer -> Bool
mrComposite x n = let
    (r,s) = decomp (n-1)
    fs    = takeWhile (/= 1)
       (map (\ j -> exM x (2^j*s) n)  [0..r])
  in
    exM x s n /= 1 && last fs /= (n-1)      -- if (x^s mod n) == 1 → (x^s mod n)^pow(2,r) == 1
                                            -- if last fs == (n-1) → x^(pow(2,r)*s) does not fail Fermat
```

The Miller-Rabin Primality Test

- K is the number of candidates to consider

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

- Testing the Tests → lists of prime numbers are useless, we need (interesting) composite numbers → see lab exercises

Application to cryptography

(Diffie and Hellman 1976): public key cryptography

- Easy to find large primes, multiply them, and compute exponentiation modulo a prime
- Hard to factor numbers that are multiples of large primes, and find x from x^a mod p

Exchange protocol

- Alice and Bob agree on a large prime p and a base g<p such that g and p−1 are coprime.
- Alice sends a secret a by sending A = g^a mod p
- Bob sends a secret a by sending B = g^b mod p
- Alice calculates the key k = B^a mod p = $(g^b)^a$ mod p
- Bob calculates the key k = A^b mod p = $(g^a)^b$ mod p
- To encode a message m, Alice and Bob calculate m * k mod p

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

- Alice decodes a cipher c with $c*(g^b)^{(p-1)-a}$ mod p = $(m*g^{ab})(g^{p-1})^b g^{-ab}$ via Fermat $m*1^b$ mod p = m mod p

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

Symmetric key ciphers using fast exponentiation

- If p and k are known, encoding and decoding are easy (computationally)

```
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
```

Why can the public key be public?

Euler's totient function: counts the positive integers $k < n$ that are coprime with n.

- Special case for semiprimes (product of two primes p, q) = $(p-1)(q-1)$
- Let n be a semiprime pq, and let e be coprime with totient(n)=$(p-1)(q-1)$
- x^e is a bijection on $\{0,...,n-1\}$
- let $d = e^{-1}$ mod totient(n), then $(x^e)^d=x \bmod n$, since $ed = 1 \bmod (p-1)(q-1)$ and we can apply Fermat's little theorem for both $(p-1)$ and $(q-1)$
- e and d can be used for encryption and decryption
  - From e and n it is hard to compute d → d is kept a secret

Asymmetric public key cryptography

- Generate a public key and make it public
- From it, anyone can construct an encoding function
- Decoding is considered hard
- Len Adleman, Adi Shami, Ron Rivest
- Select an integer e that is coprime with the totient of semiprime n=pq
  - p,q are large primes of the same bitlength
  - Let $d = e^{-1} \bmod (p-1)(q-1)$
- (e,n) is the public key, (d,n) is the private key

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

rsaDecode :: (Integer,Integer) -> Integer -> Integer
rsaDecode = rsaEncode                                     -- (m^e)^pow(e,-1) mod n = m mod n
```

Bibliography

- Number theoretic algorithms: (Cormen, Leiserson, and Rivest 1997), Chapter 33, and (Dasgupta, Papadimitriou, and Vazirani 2008), Chapter 1.
- Fast primality testing (Miller 1976),(Rabin 1980).
- Cryptographic key exchange (Diffie and Hellman 1976).
- RSA public key cryptography (R. Rivest, Shamir, and Adleman 1978).