

Finite fields

Ray D. Sameshima

2016/09/23 ~

Contents

0	Preface	5
0.1	References	5
0.2	Set theoretical gadgets	5
0.2.1	Numbers	5
0.2.2	Algebraic structures	6
0.3	Haskell	6
1	Basics	9
1.1	Finite field	9
1.1.1	Rings	9
1.1.2	Fields	10
1.1.3	An example of finite rings \mathbb{Z}_n	10
1.1.4	Bézout's lemma	10
1.1.5	Greatest common divisor	11
1.1.6	Extended Euclidean algorithm	12
1.1.7	Coprime	16
1.1.8	Corollary (Inverses in \mathbb{Z}_n)	16
1.1.9	Corollary (Finite field \mathbb{Z}_p)	16
1.1.10	A map from \mathbb{Q} to \mathbb{Z}_p	18
1.1.11	Reconstruction from \mathbb{Z}_p to \mathbb{Q}	20
1.1.12	Chinese remainder theorem	22
1.2	Polynomials and rational functions	24
1.2.1	Notations	24
1.2.2	Polynomials and rational functions	24
1.2.3	As data	25
2	Functional reconstruction	27
2.1	Univariate polynomials	27
2.1.1	Newtons' polynomial representation	27

2.1.2	Towards canonical representations	28
2.2	Univariate rational functions	29
2.2.1	Thiele's interpolation formula	29
2.2.2	Towards canonical representations	30
2.3	Multivariate polynomials	30
2.3.1	Foldings as recursive applications	30
2.4	Multivariate rational functions	30
2.4.1	The canonical normalization	30
2.4.2	An auxiliary t	31

Chapter 0

Preface

0.1 References

1. Scattering amplitudes over finite fields and multivariate functional reconstruction (Tiziano Peraro)
<https://arxiv.org/pdf/1608.01902.pdf>
2. Haskell Language
www.haskell.org
3. http://qiita.com/bra_cat_ket/items/205c19611e21f3d422b7
(Japanese tech support sns)

0.2 Set theoretical gadgets

0.2.1 Numbers

Here is a list of what we assumed that the readers are familiar with:

1. \mathbb{N} (Peano axiom: \emptyset, suc)
2. \mathbb{Z}
3. \mathbb{Q}
4. \mathbb{R} (Dedekind cut)
5. \mathbb{C}

0.2.2 Algebraic structures

1. Monoid: $(\mathbb{N}, +)$, (\mathbb{N}, \times)
2. Group: $(\mathbb{Z}, +)$, (\mathbb{Z}, \times)
3. Ring: \mathbb{Z}
4. Field: \mathbb{Q} , \mathbb{R} (continuous), \mathbb{C} (algebraic closed)

0.3 Haskell

From "A Brief, Incomplete and Mostly Wrong History of Programming Languages":¹

1990 - A committee formed by Simon Peyton-Jones, Paul Hudak, Philip Wadler, Ashton Kutcher, and People for the Ethical Treatment of Animals creates Haskell, a pure, non-strict, functional language. Haskell gets some resistance due to the complexity of using monads to control side effects. Wadler tries to appease critics by explaining that "a monad is a monoid in the category of endofunctors, what's the problem?"



Figure 1: Haskell's logo, the combinations of λ and monad's bind $>>=$.

Haskell language is a standardized purely functional declarative statically typed programming language.

In declarative languages, we describe "what" or "definition" in its codes, however imperative languages, like C/C++, "how" or "procedure".

Functional languages can be seen as 'executable mathematics'; the notation was designed to be as close as possible to the mathematical way of writing.²

¹ <http://james-iry.blogspot.com/2009/05/brief-incomplete-and-mostly-wrong.html>

² Algorithms: A Functional Programming Approach (Fethi A. Rabhi, Guy Lapalme)

Instead of loops, we use (implicit) recursions in functional language.³

```
> sum :: [Int] -> Int
> sum []      = 0
> sum (i:is) = i + sum is
```

³Of course, as a best practice, we should use higher order function rather than explicit recursions.

Chapter 1

Basics

We have assumed living knowledge on (axiomatic, i.e., ZFC) set theory, algebraic structures.

1.1 Finite field

1.1.1 Rings

A ring $(R, +, *)$ is a structured set R with two binary operations

$$(+)\ ::\ R\ \rightarrow\ R\ \rightarrow\ R \quad (1.1)$$

$$(*)\ ::\ R\ \rightarrow\ R\ \rightarrow\ R \quad (1.2)$$

satisfying the following 3 (ring) axioms:

1. $(R, +)$ is an abelian, i.e., commutative group, i.e.,

$$\forall a, b, c \in R, (a + b) + c = a + (b + c) \quad (\text{associativity for } +) \quad (1.3)$$

$$\forall a, b \in R, a + b = b + a \quad (\text{commutativity}) \quad (1.4)$$

$$\exists 0 \in R, \text{ s.t. } \forall a \in R, a + 0 = a \quad (\text{additive identity}) \quad (1.5)$$

$$\forall a \in R, \exists (-a) \in R \text{ s.t. } a + (-a) = 0 \quad (\text{additive inverse}) \quad (1.6)$$

2. $(R, *)$ is a monoid, i.e.,

$$\forall a, b, c \in R, (a * b) * c = a * (b * c) \quad (\text{associativity for } *) \quad (1.7)$$

$$\exists 1 \in R, \text{ s.t. } \forall a \in R, a * 1 = a = 1 * a \quad (\text{multiplicative identity}) \quad (1.8)$$

3. Multiplication is distributive w.r.t addition, i.e., $\forall a, b, c \in R$,

$$a * (b + c) = (a * b) + (a * c) \quad (\text{left distributivity}) \quad (1.9)$$

$$(a + b) * c = (a * c) + (b * c) \quad (\text{right distributivity}) \quad (1.10)$$

1.1.2 Fields

A field is a ring $(\mathbb{K}, +, *)$ whose non-zero elements form an abelian group under multiplication, i.e., $\forall r \in \mathbb{K}$,

$$r \neq 0 \Rightarrow \exists r^{-1} \in \mathbb{K} \text{ s.t. } r * r^{-1} = 1 = r^{-1} * r. \quad (1.11)$$

A field \mathbb{K} is a finite field iff the underlying set \mathbb{K} is finite. A field \mathbb{K} is called infinite field iff the underlying set is infinite.

1.1.3 An example of finite rings \mathbb{Z}_n

Let $n(> 0) \in \mathbb{N}$ be a non-zero natural number. Then the quotient set

$$\mathbb{Z}_n := \mathbb{Z}/n\mathbb{Z} \quad (1.12)$$

$$\cong \{0, \dots, (n-1)\} \quad (1.13)$$

with addition, subtraction and multiplication under modulo n is a ring.¹

1.1.4 Bézout's lemma

Consider $a, b \in \mathbb{Z}$ be nonzero integers. Then there exist $x, y \in \mathbb{Z}$ s.t.

$$a * x + b * y = \gcd(a, b), \quad (1.19)$$

where \gcd is the greatest common divisor (function), see §1.1.5. We will prove this statement in §1.1.6.

¹ Here we have taken an equivalence class,

$$0 \leq \forall k \leq (n-1), [k] := \{k + n * z | z \in \mathbb{Z}\} \quad (1.14)$$

with the following operations:

$$[k] + [l] := [k + l] \quad (1.15)$$

$$[k] * [l] := [k * l] \quad (1.16)$$

This is equivalent to take modular n :

$$(k \bmod n) + (l \bmod n) := (k + l \bmod n) \quad (1.17)$$

$$(k \bmod n) * (l \bmod n) := (k * l \bmod n). \quad (1.18)$$

1.1.5 Greatest common divisor

Before the proof, here is an implementation of gcd using Euclidean algorithm with Haskell language:

```
> -- Euclidian algorithm.
> myGCD :: Integral a => a -> a -> a
> myGCD a b
>   | b < 0 = myGCD a (-b)
> myGCD a b
>   | a == b = a
>   | b > a = myGCD b a
>   | b < a = myGCD (a-b) b
```

Example, by hands

Let us consider the gcd of 7 and 13. Since they are primes, the gcd should be 1. First it binds `a` with 7 and `b` with 13, and hit `b > a`.

$$\text{myGCD } 7 \ 13 == \text{myGCD } 13 \ 7 \quad (1.20)$$

Then it hits main line:

$$\text{myGCD } 13 \ 7 == \text{myGCD } (13-7) \ 7 \quad (1.21)$$

In order to go to next step, Haskell evaluate $(13 - 7)$,² and

$$\text{myGCD } (13-7) \ 7 == \text{myGCD } 6 \ 7 \quad (1.22)$$

$$== \text{myGCD } 7 \ 6 \quad (1.23)$$

$$== \text{myGCD } (7-6) \ 6 \quad (1.24)$$

$$== \text{myGCD } 1 \ 6 \quad (1.25)$$

$$== \text{myGCD } 6 \ 1 \quad (1.26)$$

Finally it ends with 1:

$$\text{myGCD } 1 \ 1 == 1 \quad (1.27)$$

² Since Haskell language adopts lazy evaluation, i.e., call by need, not call by name.

As another example, consider 15 and 25:

$$\text{myGCD } 15 \ 25 == \text{myGCD } 25 \ 15 \quad (1.28)$$

$$== \text{myGCD } (25-15) \ 15 \quad (1.29)$$

$$== \text{myGCD } 10 \ 15 \quad (1.30)$$

$$== \text{myGCD } 15 \ 10 \quad (1.31)$$

$$== \text{myGCD } (15-10) \ 10 \quad (1.32)$$

$$== \text{myGCD } 5 \ 10 \quad (1.33)$$

$$== \text{myGCD } 10 \ 5 \quad (1.34)$$

$$== \text{myGCD } (10-5) \ 5 \quad (1.35)$$

$$== \text{myGCD } 5 \ 5 \quad (1.36)$$

$$== 5 \quad (1.37)$$

Example, by Haskell

Let us check simple example using Haskell:

```
*Ffield> myGCD 7 13
1
*Ffield> myGCD 7 14
7
*Ffield> myGCD (-15) (20)
5
*Ffield> myGCD (-299) (-13)
13
```

The final result is from

```
*Ffield> 13*23
299
```

1.1.6 Extended Euclidean algorithm

Here we treat the extended Euclidean algorithm.

As intermediate steps, this algorithm makes sequences of integers $\{r_i\}_i$, $\{s_i\}_i$, $\{t_i\}_i$ and quotients $\{q_i\}_i$ as follows. The base case are

$$(r_0, s_0, t_0) := (a, 1, 0) \quad (1.38)$$

$$(r_1, s_1, t_1) := (b, 0, 1) \quad (1.39)$$

and inductively,

$$q_i := \text{quot}(r_{i-2}, r_{i-1}) \quad (1.40)$$

$$r_i := r_{i-2} - q_i * r_{i-1} \quad (1.41)$$

$$s_i := s_{i-2} - q_i * s_{i-1} \quad (1.42)$$

$$t_i := t_{i-2} - q_i * t_{i-1}. \quad (1.43)$$

The termination condition³ is

$$r_k = 0 \quad (1.44)$$

for some $k \in \mathbb{N}$ and

$$\gcd(a, b) = r_{k-1} \quad (1.45)$$

$$x = s_{k-1} \quad (1.46)$$

$$y = t_{k-1}. \quad (1.47)$$

Proof

By definition,

$$\gcd(r_{i-1}, r_i) = \gcd(r_{i-1}, r_{i-2} - q_i * r_{i-1}) \quad (1.48)$$

$$= \gcd(r_{i-1}, r_{i-2}) \quad (1.49)$$

and this implies

$$\gcd(a, b) =: \gcd(r_0, r_1) = \cdots = \gcd(r_{k-1}, 0), \quad (1.50)$$

i.e.,

$$r_{k-1} = \gcd(a, b). \quad (1.51)$$

Next, for $i = 0, 1$ observe

$$a * s_i + b * t_i = r_i. \quad (1.52)$$

Let $i \geq 2$, then

$$r_i = r_{i-2} - q_i * r_{i-1} \quad (1.53)$$

$$= a * s_{i-2} + b * t_{i-2} - q_i * (a * s_{i-1} + b * t_{i-1}) \quad (1.54)$$

$$= a * (s_{i-2} - q_i * s_{i-1}) + b * (t_{i-2} - q_i * t_{i-1}) \quad (1.55)$$

$$=: a * s_i + b * t_i. \quad (1.56)$$

³ This algorithm will terminate eventually, since the sequence $\{r_i\}_i$ is non-negative by definition of q_i , but strictly decreasing. Therefore, $\{r_i\}_i$ will meet 0 in finite step k .

Therefore, inductively we get

$$\gcd(a, b) = r_{k-1} = a * s_{k-1} + b * t_{k-1}. =: a * s + b * t. \quad (1.57)$$

This prove Bézout's lemma.

■

Haskell implementation

Here I use lazy lists for intermediate lists of qs, rs, ss, ts , and pick up (second) last elements for the results.

Here we would like to implement the extended Euclidean algorithm. See the algorithm, examples, and pseudo code at:

https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm

I've asked at Qiita(Japanese) and get some solutions:

http://qiita.com/bra_cat_ket/items/205c19611e21f3d422b7

```
> exGCD' :: Integral n => n -> n -> ([n], [n], [n], [n])
> exGCD' a b = (qs, rs, ss, ts)
>   where
>     qs = zipWith quot rs (tail rs)
>     rs = takeBefore (==0) r'
>     r' = steps a b
>     ss = steps 1 0
>     ts = steps 0 1
>     steps a b = rr
>       where rr@(_:rs) = a:b: zipWith (-) rr (zipWith (*) qs rs)
>
> takeBefore :: (a -> Bool) -> [a] -> [a]
> takeBefore _ [] = []
> takeBefore p (l:ls)
>   | p l      = []
>   | otherwise = l : (takeBefore p ls)
```

Here we have used so called lazy lists, and higher order function⁴. The gcd of a and b should be the last element of second list, and our targets (s, t)

⁴ Naively speaking, the function whose inputs and/or outputs are functions is called a higher order function.

are second last elements of last two lists. The following example is from wikipedia:

```
*Ffield> exGCD' 240 46
([5,4,1,1,2],[240,46,10,6,4,2],[1,0,1,-4,5,-9,23],[0,1,-5,21,-26,47,-120])
*Ffield> gcd 240 46
2
*Ffield> 240*(-9) + 46*(47)
2
```

It works, and we have other simpler examples:

```
*Ffield> exGCD' 15 25
([0,1,1,2],[15,25,15,10,5],[1,0,1,-1,2,-5],[0,1,0,1,-1,3])
*Ffield> 15 * 2 + 25*(-1)
5
*Ffield> exGCD' 15 26
([0,1,1,2,1,3],[15,26,15,11,4,3,1],[1,0,1,-1,2,-5,7,-26],[0,1,0,1,-1,3,-4,15])
*Ffield> 15*7 + (-4)*26
1
```

Now what we should do is extract gcd of a and b , and (s, t) from the tuple of lists:

```
> -- a*x + b*y = gcd a b
> exGcd a b = (g, x, y)
>   where
>     (_,r,s,t) = exGCD' a b
>     g = last r
>     x = last . init $ s
>     y = last . init $ t
```

where the underscore $_$ is a special symbol in Haskell that hits every pattern. So, in order to get gcd and (s, t) we don't need quotients list.

```
*Ffield> exGcd 46 240
(2,47,-9)
*Ffield> 46*47 + 240*(-9)
2
*Ffield> gcd 46 240
2
```

1.1.7 Coprime

Let us define a binary relation as follows:

```
coprime :: Integral a => a -> a -> Bool
coprime a b = (gcd a b) == 1
```

1.1.8 Corollary (Inverses in \mathbb{Z}_n)

For a non-zero element

$$a \in \mathbb{Z}_n, \quad (1.58)$$

there is a unique number

$$b \in \mathbb{Z}_n \text{ s.t. } ((a * b) \bmod n) = 1 \quad (1.59)$$

iff a and n are coprime.

Proof

From Bézout's lemma, a and n are coprime iff

$$\exists s, t \in \mathbb{Z}, a * s + n * t = 1. \quad (1.60)$$

Therefore

$$a \text{ and } n \text{ are coprime} \Leftrightarrow \exists s, t \in \mathbb{Z}, a * s + n * t = 1 \quad (1.61)$$

$$\Leftrightarrow \exists s, t' \in \mathbb{Z}, a * s = 1 + n * t'. \quad (1.62)$$

This s , by taking its modulo n is our $b = a^{-1}$:

$$a * s = 1 \bmod n. \quad (1.63)$$

■

1.1.9 Corollary (Finite field \mathbb{Z}_p)

If p is prime, then

$$\mathbb{Z}_p := \{0, \dots, (p-1)\} \quad (1.64)$$

with addition, subtraction and multiplication under modulo n is a field.

Proof

It suffices to show that

$$\forall a \in \mathbb{Z}_p, a \neq 0 \Rightarrow \exists a^{-1} \in \mathbb{K} \text{ s.t. } a * a^{-1} = 1 = a^{-1} * a, \quad (1.65)$$

but since p is prime, and

$$\forall a \in \mathbb{Z}_p, a \neq 0 \Rightarrow \gcd a p == 1 \quad (1.66)$$

so all non-zero element has its inverse in \mathbb{Z}_p .

■

Example and implementation

Let us pick 11 as a prime and consider \mathbb{Z}_{11} :

Example $\mathbb{Z}_{\{11\}}$

```
*Ffield> isField 11
True
*ffield> map (exGcd 11) [0..10]
[(11,1,0),(1,0,1),(1,1,-5),(1,-1,4),(1,-1,3)
,(1,1,-2),(1,-1,2),(1,2,-3),(1,3,-4),(1,-4,5)
,(1,1,-1)
]

*ffield> map (('mod' 11) . \(_,_,x)->x) . exGcd 11 [1..10]
[1,6,4,3,9,2,8,7,5,10]
*ffield> zip [1..10] it
[(1,1),(2,6),(3,4),(4,3),(5,9),(6,2),(7,8),(8,7),(9,5),(10,10)]
```

Let us generalize these flow into a function⁵:

```
> inverses :: Int -> Maybe [(Int, Int)]
> inverses n
```

⁵ From <https://hackage.haskell.org/package/base-4.9.0.0/docs/Data-Maybe.html>:

The Maybe type encapsulates an optional value. A value of type Maybe a either contains a value of type a (represented as Just a), or it is empty (represented as Nothing). Using Maybe is a good way to deal with errors or exceptional cases without resorting to drastic measures such as error.

```

> | isField n = Just lst -- isPrime n
> | otherwise = Nothing
> where
>   lst' = map (('mod' n) . (\(_,_,c)->c) . exGcd n) [1..(n-1)]
>   lst = zip [1..] lst'

```

Now we define `inversep`,⁶ whose 1st input is the base p of our ring(field) and 2nd input is an element in \mathbb{Z}_p .

```

> inversep' :: Int -> Int -> Maybe Int
> inversep 'p a = do
>   l <- inverses p
>   let a' = (a 'mod' p)
>   return (snd $ l !! (a'-1))

```

```

*Ffield> inverses' 11
Just [(1,1),(2,6),(3,4),(4,3),(5,9),(6,2),(7,8),(8,7),(9,5),(10,10)]

```

The function `inverses` returns a list of nonzero number with their inverses if p is prime. However, this is not efficient, and we refactor it as follows:

```

> inversep :: Int -> Int -> Maybe Int
> inversep p a = let (_,x,y) = exGcd p a in
>   if isPrime p then Just (y 'mod' p)
>   else Nothing

```

```

map (inversep' 10007) [1..10006]
(1.74 secs, 771,586,416 bytes)

```

1.1.10 A map from \mathbb{Q} to \mathbb{Z}_p

Let p be a prime. Now we have a map

$$- \text{ mod } p : \mathbb{Z} \rightarrow \mathbb{Z}_p; a \mapsto (a \text{ mod } p), \quad (1.67)$$

and a natural inclusion (or a forgetful map)⁷

$$i : \mathbb{Z}_p \hookrightarrow \mathbb{Z}. \quad (1.69)$$

⁶ Here we have used do-notation, a syntactic sugar for use with monadic expressions. From <https://wiki.haskell.org/Monad>:

Monads in Haskell can be thought of as composable computation descriptions.

⁷ By introducing this forgetful map, we can use

$$\times : (\mathbb{Z}, \mathbb{Z}) \rightarrow \mathbb{Z} \quad (1.68)$$

Then we can define a map⁸

$$- \text{ mod } p : \mathbb{Q} \rightarrow \mathbb{Z}_p \quad (1.70)$$

by

$$q = \frac{a}{b} \mapsto (q \text{ mod } p) := ((a \times i(b^{-1} \text{ mod } p)) \text{ mod } p). \quad (1.71)$$

Example and implementation

An easy implementation is the followings:

A map from \mathbb{Q} to \mathbb{Z}_p .

```
> modp :: Ratio Int -> Int -> Int
> q 'modp' p = (a * (bi 'mod' p)) 'mod' p
>   where
>     (a,b) = (numerator q, denominator q)
>     bi = fromJust $ inversep p b
```

Let us consider a rational number $\frac{3}{7}$ on a finite field \mathbb{Z}_{11} :

Example: on \mathbb{Z}_{11}

Consider $(3 \% 7)$.

```
*Ffield Data.Ratio> let q = 3 % 7
*Ffield Data.Ratio> 3 'mod' 11
3
*Ffield Data.Ratio> 7 'mod' 11
7
*Ffield Data.Ratio> inverses 11
Just [(1,1),(2,6),(3,4),(4,3),(5,9),(6,2),(7,8),(8,7),(9,5),(10,10)]
*Ffield Data.Ratio> 7*8 == 11*5+1
True
```

```
on Z_{11}, (7^{-1} 'mod' 11) is equal to (8 'mod' 11) and
(3%7) |-> (3 * (7^{-1} 'mod' 11) 'mod' 11)
          == (3*8 'mod' 11)
          == 2 'mod 11
```

of normal product on \mathbb{Z} .

⁸ This is an example of operator overloadings.

```
*Ffield Data.Ratio> q 'modp' 11
2
```

1.1.11 Reconstruction from \mathbb{Z}_p to \mathbb{Q}

Consider a rational number q and its image $a \in \mathbb{Z}_p$.

$$a := q \mod p \quad (1.72)$$

The extended Euclidean algorithm can be used for guessing a rational number q from $a := q \mod p$.

At each step, the extended Euclidean algorithm satisfies eq.(1.52).

$$a * s_i + p * t_i = r_i \quad (1.73)$$

Therefore

$$r_i = a * s_i \mod p \Leftrightarrow \frac{r_i}{s_i} \mod p = a. \quad (1.74)$$

Hence $\frac{r_i}{s_i}$ is a possible guess for q . We take

$$r_i^2, s_i^2 < p \quad (1.75)$$

as the termination condition for this reconstruction.

Haskell implementation

Let us try to reconstruct from the image $(\frac{1}{3} \mod p)$ of some prime p . Here we have chosen three primes

```
Reconstruction Z_p -> Q
*Ffield> let q = (1%3)
*Ffield> take 3 $ dropWhile (<100) primes
[101,103,107]
```

The images are basically given by the first elements (s_0 's) of second lists:

```
*Ffield> q 'modp' 101
34
*Ffield> let try x = exGCD' (q 'modp' x) x
*Ffield> try 101
([0,2,1,33],[34,101,34,33,1],[1,0,1,-2,3,-101],[0,1,0,1,-1,34])
```

```

*Ffield> try 103
([0,1,2,34],[69,103,69,34,1],[1,0,1,-1,3,-103],[0,1,0,1,-2,69])
*Ffield> try 107
([0,2,1,35],[36,107,36,35,1],[1,0,1,-2,3,-107],[0,1,0,1,-1,36])

```

Look at the first hit of termination condition eq.(1.75), $r_4 = 1$ and $s_4 = 3$. They give us the same guess $\frac{1}{3}$, and that the reconstructed number.

From the above observations we can make a simple "guess" function:

```

> guess :: (Int, Int)      -- (q 'modp' p, p)
>      -> (Ratio Int, Int)
> guess (a, p) = let (_,rs,ss,_) = exGCD' a p in
>   (select rs ss p, p)
>   where
>     select :: Integral t => [t] -> [t] -> t -> Ratio t
>     select [] _ _ = 0%1
>     select (r:rs) (s:ss) p
>       | s /= 0 && r^2 <= p && s^2 <= p = (r%s)
>       | otherwise = select rs ss p

```

We have put a list of big primes as follows.

```

> -- Hard code of big primes.
> bigPrimes :: [Int]
> bigPrimes = dropWhile (< 897473) $ takeWhile (<978948) primes

```

We choose 3 times match as the termination condition.

```

> matches3 :: Eq a => [a] -> a
> matches3 (a:bb@(b:c:cs))
>   | a == b && b == c = a
>   | otherwise       = matches3 bb

```

Finally, we can check our gadgets.

What we know is a list of $(q \text{ 'modp' } p)$ and prime p .

```

*Ffield> let q = 10%19
*Ffield> let knownData = zip (map (modp q) bigPrimes) bigPrimes
*Ffield> matches3 $ map (fst . guess) knownData
10 % 19

```

The following is the function we need, its input is the list of tuple which first element is the image in \mathbb{Z}_p and second element is that prime p .

```
> reconstruct :: [(Int,Int)] -> Ratio Int
> reconstruct aps = matches3 $ map (fst . guess) aps
```

Here is a naive test:

```
> let qs = [1 % 3, 10 % 19, 41 % 17, 30 % 311, 311 % 32, 869 % 232, 778 % 123, 331 % 739]
> let modmap q = zip (map (modp q) bigPrimes) bigPrimes
> let longList = map modmap qs
> map reconstruct longList
[1 % 3, 10 % 19, 41 % 17, 30 % 311, 311 % 32, 869 % 232, 778 % 123, 331 % 739]
> it == qs
True
```

1.1.12 Chinese remainder theorem

From wikipedia⁹

There are certain things whose number is unknown. If we count them by threes, we have two left over; by fives, we have three left over; and by sevens, two are left over. How many things are there?

Here is a solution with Haskell:

```
> let lst = [n|n<-[0..], n `mod` 3 == 2, n `mod` 5 == 3, n `mod` 7 == 2]
> head lst
23
```

or more explicitly,

```
> let clst = [n|n<-[0.. (3*5*7)], mod n 3 == 2, mod n 5 == 3, mod n 7 == 2]
> clst
[23]
```

The statement for binary case is the following. Let $n_1, n_2 \in \mathbb{Z}$ be coprime, then for arbitrary $a_1, a_2 \in \mathbb{Z}$, the following a system of equations

$$x = a_1 \pmod{n_1} \tag{1.76}$$

$$x = a_2 \pmod{n_2} \tag{1.77}$$

have a unique solution modular $n_1 * n_2$.

⁹ https://en.wikipedia.org/wiki/Chinese_remainder_theorem

Proof

(existence) With §1.1.6, there are $m_1, m_2 \in \mathbb{Z}$ s.t.

$$n_1 * m_1 + n_2 * m_2 = 1. \quad (1.78)$$

Now we have

$$n_1 * m_1 = 1 \pmod{n_2} \quad (1.79)$$

$$n_2 * m_2 = 1 \pmod{n_1} \quad (1.80)$$

that is

$$m_1 = n_1^{-1} \pmod{n_2} \quad (1.81)$$

$$m_2 = n_2^{-1} \pmod{n_1}. \quad (1.82)$$

Then

$$a := a_1 * n_2 * m_2 + a_2 * n_1 * m_1 \pmod{(n_1 * n_2)} \quad (1.83)$$

is a solution.

(uniqueness) If a' is also a solution, then

$$a - a' = 0 \pmod{n_1} \quad (1.84)$$

$$a - a' = 0 \pmod{n_2}. \quad (1.85)$$

Since n_1 and n_2 are coprime, i.e., no common divisors, this difference is divisible by $n_1 * n_2$, and

$$a - a' = 0 \pmod{(n_1 * n_2)}. \quad (1.86)$$

Therefore, the solution is unique modular $n_1 * n_2$.

■

Generalization

Given $a \in \mathbb{Z}_n$ of pairwise coprime numbers

$$n := n_1 * \cdots * n_k, \quad (1.87)$$

a system of equations

$$a_i = a \pmod{n_i}_{i=1}^k \quad (1.88)$$

have a unique solution

$$a = \sum_i m_i a_i \pmod n, \quad (1.89)$$

where

$$m_i = \left(\frac{n_i}{n} \pmod{n_i} \right) \frac{n}{n_i} \Big|_{i=1}^k. \quad (1.90)$$

1.2 Polynomials and rational functions

1.2.1 Notations

Let $n \in \mathbb{N}$ be positive. We use multi-index notation:

$$\alpha = (\alpha_1, \dots, \alpha_n) \in \mathbb{N}^n. \quad (1.91)$$

A monomial is defined as

$$z^\alpha := \prod_i z_i^{\alpha_i}. \quad (1.92)$$

The total degree of this monomial is given by

$$|\alpha| := \sum_i \alpha_i. \quad (1.93)$$

1.2.2 Polynomials and rational functions

Let \mathbb{K} be a field. Consider a map

$$f : \mathbb{F}^n \rightarrow \mathbb{F}; z \mapsto f(z) := \sum_{\alpha} c_{\alpha} z^{\alpha}, \quad (1.94)$$

where

$$c_{\alpha} \in \mathbb{F}. \quad (1.95)$$

We call the value $f(z)$ at the dummy $z \in \mathbb{F}^n$ a polynomial:

$$f(z) := \sum_{\alpha} c_{\alpha} z^{\alpha}. \quad (1.96)$$

We denote

$$\mathbb{F}[z] := \left\{ \sum_{\alpha} c_{\alpha} z^{\alpha} \right\} \quad (1.97)$$

as the ring of all polynomial functions in the variable z with \mathbb{F} -coefficients.

Similarly, a rational function can be expressed as a ratio of two polynomials $p(z), q(z) \in \mathbb{F}[z]$:

$$\frac{p(z)}{q(z)} = \frac{\sum_{\alpha} n_{\alpha} z^{\alpha}}{\sum_{\beta} d_{\beta} z^{\beta}}. \quad (1.98)$$

We denote

$$\mathbb{F}(z) := \left\{ \frac{\sum_{\alpha} n_{\alpha} z^{\alpha}}{\sum_{\beta} d_{\beta} z^{\beta}} \right\} \quad (1.99)$$

as the field of rational functions in the variable z with \mathbb{F} -coefficients. Similar to fractional numbers, there are several equivalent representation of a rational function, even if we simplify with gcd. However there still is an overall constant ambiguity. To have a unique representation, usually we put the lowest degree of term of the denominator to be 1.

1.2.3 As data

We can identify a polynomial

$$\sum_{\alpha} c_{\alpha} z^{\alpha} \quad (1.100)$$

as a set of coefficients

$$\{c_{\alpha}\}_{\alpha}. \quad (1.101)$$

Similarly, for a rational function, we can identify

$$\frac{\sum_{\alpha} n_{\alpha} z^{\alpha}}{\sum_{\beta} d_{\beta} z^{\beta}} \quad (1.102)$$

as an ordered pair of coefficients

$$(\{n_{\alpha}\}_{\alpha}, \{d_{\beta}\}_{\beta}). \quad (1.103)$$

However, there still is an overall factor ambiguity even after gcd simplifications.

Chapter 2

Functional reconstruction

The goal of a functional reconstruction algorithm is to identify the monomials appearing in their definition and the corresponding coefficients.

2.1 Univariate polynomials

2.1.1 Newtons' polynomial representation

Consider a univariate polynomial $f(z)$. Given a sequence of values $y_n|_{n \in \mathbb{N}}$, we evaluate the polynomial form $f(z)$ sequentially:

$$f_0(z) = a_0 \quad (2.1)$$

$$f_1(z) = a_0 + (z - y_0)a_1 \quad (2.2)$$

$$\vdots$$

$$f_r(z) = a_0 + (z - y_0)(a_1 + (z - y_1)(\cdots + (z - y_{r-1})a_r)) \quad (2.3)$$

$$= f_{r-1}(z) + (z - y_0)(z - y_1) \cdots (z - y_{r-1})a_r, \quad (2.4)$$

where

$$a_0 = f(y_0) \quad (2.5)$$

$$a_1 = \frac{f(y_1) - a_0}{y_1 - y_0} \quad (2.6)$$

$$\vdots$$

$$a_r = \left(\left((f(y_r) - a_0) \frac{1}{y_r - y_0} - a_1 \right) \frac{1}{y_r - y_1} - \cdots - a_{r-1} \right) \frac{1}{y_r - y_{r-1}} \quad (2.7)$$

When we have already known the total degree of $f(z)$, say R , then we can terminate this sequential trial:

$$f(z) = f_R(z) \quad (2.8)$$

$$= \sum_{r=0}^R a_r \prod_{i=0}^{r-1} (z - y_i). \quad (2.9)$$

In practice, a consecutive zero on the sequence a_r can be taken as the termination condition for this algorithm.¹

2.1.2 Towards canonical representations

Once we get the Newton's representation

$$\sum_{r=0}^R a_r \prod_{i=0}^{r-1} (z - y_i) = a_0 + (z - y_0)(a_1 + (z - y_1)(\cdots + (z - y_{R-1})a_R)) \quad (2.10)$$

as the reconstructed polynomial, it is convenient to convert it into the canonical form:

$$\sum_{r=0}^R c_r z^r. \quad (2.11)$$

This conversion only requires addition and multiplication of univariate polynomials. These operations are reasonably cheap, especially on \mathbb{Z}_p .

¹ We have not proved, but higher power will be dominant when we take sufficiently big input, so we terminate this sequence when we get a consecutive zero in a_r .

2.2 Univariate rational functions

2.2.1 Thiele's interpolation formula

Consider a univariate rational function $f(z)$. Given a sequence of values $y_n|_{n \in \mathbb{N}}$, we evaluate the polynomial form $f(z)$ as a continued fraction:

$$f_0(z) = a_0 \quad (2.12)$$

$$f_1(z) = a_0 + \frac{(z - y_0)}{a_1} \quad (2.13)$$

$$\vdots$$

$$f_r(z) = a_0 + \frac{(z - y_0)}{a_1 + \frac{z - y_1}{a_2 + \frac{z - y_3}{\dots + \frac{z - y_{r-1}}{a_r}}}}, \quad (2.14)$$

where

$$a_0 = f(y_0) \quad (2.15)$$

$$a_1 = \frac{y_1 - y_0}{f(y_1) - a_0} \quad (2.16)$$

$$\vdots$$

$$a_r = \left(\left((f(y_r) - a_0)^{-1} (y_r - y_0) - a_1 \right)^{-1} \frac{1}{y_r - y_1} - \dots - a_{r-1} \right)^{-1} (y_r - y_{r-1}) \quad (2.17)$$

Termination condition(s)

We choose our termination conditions as several agreements among new reconstructed function:²

$$f_{n-1}(z) \neq f_n(z) = f_{n+1}(z) = f_{n+2}(z) = \dots \quad (2.19)$$

² Note that, this does not simply mean

$$a_n = a_{n+1} = a_{n+2} = \dots = 0. \quad (2.18)$$

2.2.2 Towards canonical representations

In order to get a unique representation of canonical form

$$\frac{\sum_{\alpha} n_{\alpha} z^{\alpha}}{\sum_{\beta} d_{\beta} z^{\beta}} \quad (2.20)$$

we put

$$d_{\min} r' = 1 \quad (2.21)$$

as a normalization, instead of d_0 .

2.3 Multivariate polynomials

2.3.1 Foldings as recursive applications

Consider an arbitrary multivariate polynomial

$$f(z_1, \dots, z_n) \in \mathbb{F}[z_1, \dots, z_n]. \quad (2.22)$$

First, fix all the variable but 1st and apply the univariate Newton's reconstruction:

$$f(z_1, z_2, \dots, z_n) = \sum_{r=0}^R a_r(z_2, \dots, z_n) \prod_{i=0}^{r-1} (z_1 - y_i) \quad (2.23)$$

Recursively, pick up one "coefficient" and apply the univariate Newton's reconstruction on z_2 :

$$a_r(z_2, \dots, z_n) = \sum_{s=0}^S b_s(z_3, \dots, z_n) \prod_{j=0}^{s-1} (z_2 - x_j) \quad (2.24)$$

The terminate condition should be the univariate case.

2.4 Multivariate rational functions

2.4.1 The canonical normalization

Our target is a pair of coefficients $(\{n_{\alpha}\}_{\alpha}, \{d_{\beta}\}_{\beta})$ in

$$\frac{\sum_{\alpha} n_{\alpha} z^{\alpha}}{\sum_{\beta} d_{\beta} z^{\beta}} \quad (2.25)$$

A canonical choice is

$$d_0 = d_{(0, \dots, 0)} = 1. \quad (2.26)$$

Accidentally we might face $d_0 = 0$, but we can shift our function and make

$$d'_0 = d_s \neq 0. \quad (2.27)$$

2.4.2 An auxiliary t

Introducing an auxiliary variable t , let us define

$$h(t, z) := f(tz_1, \dots, tz_n), \quad (2.28)$$

and reconstruct $h(t, z)$ as a univariate rational function of t :

$$h(t, z) = \frac{\sum_{r=0}^R p_r(z) t^r}{1 + \sum_{r'=1}^{R'} q_{r'}(z) t^{r'}} \quad (2.29)$$

where

$$p_r(z) = \sum_{|\alpha|=r} n_\alpha z^\alpha \quad (2.30)$$

$$q_{r'}(z) = \sum_{|\beta|=r'} n_\beta z^\beta \quad (2.31)$$

are homogeneous polynomials.

Thus, what we shall do is the (homogeneous) polynomial reconstructions of $p_r(z)|_{0 \leq r \leq R}$, $q_{r'}(z)|_{1 \leq r' \leq R'}$.

A simplification

Since our new targets are homogeneous polynomials, we can consider, say,

$$p_r(1, z_2, \dots, z_n) \quad (2.32)$$

instead of $p_r(z_1, z_2, \dots, z_n)$, reconstruct it using multivariate Newton's method, and homogenize with z_1 .