Number Theory with Clojure

Valery Kocubinsky

June 3, 2024

Contents

1 About

This project cover some topics in number theory such as integer factorization, arithmetic functions, congruences, primitive roots.

I wrote this document with Emacs Org Mode. Then I generate markdown file readme.md with Org Mode export to markdown `C-c C-e m m`, and generate pdf file readme.pdf with Org Mode export to pdf `C-c C-e l p`. Github by default renders readme.md in project root if a project has such file. Github markdown looks enough good, even math equation is supported. However for now math equation doesn't render for link text, pdf file doesn't have any issues with math equations.

I use Emacs babel for clojure to produce real output inside the document, so it is live documentation and one can open org file in Emacs. I use `org-latex-preview`, to preview math equation inside Emacs, to display all image for all fragments need to use prefix argument for `org-latex-preview` function, type `C-u C-u C-c C-x C-l` for preview all math equations inside Emacs.

Many applications require integer factorization. Integer factorization is relatively expensive procedure. Most straightforward way to factorize integer n is try to divide n to numbers $2, 3, 4, \ldots, \sqrt{n}$. There is some known optimization of this procedure. For instance exclude from candidates multiple of 2, or multiple of 2 and multiple of 3. It is enough good strategy to factorize one number. But if need to factorize many numbers more fast way to factorize number is precalculate table of first N primes and try to divide number n to primes $p_1, p_2, \ldots, p_n \leq \sqrt{n}$. Table of first N primes can be caculated with Sieve of Eratosthenes. There is a way to improve performance of factorization. With slightly modified Sieve of Eratosthenes procedure we can precalculate least prime divisor table. This table keep least prime divisor for every integer up to N. This table may works as linked list, in fact we know all divisors of any number up to N and to factorize integer we need just iterate over "linked list" to build output data structure. But least prime divisor table require more memory compare to Sieve of Eratosthenes. There is a way to improve performance again, it is another modification of Sieve of Eratosthenes procedure to build full factorization table for all numbers $1, 2, \ldots, N$. But it require much more memory for keep table of full factorization and it require more time for calculate full factorization table, compare to build least prime divisor table.

The purpose of this library/tool is to provide fast way to play with relatively small numbers(millions) and I made a decision to use least prime divisor table as the best compromise between build factorization table time, required memory for factorization table, integer factorization time. So now factorization is relatively cheap, but it require 4 bytes per number or 4MB for 1000000 numbers.

In this document I load number theory packages as:

```
(require '[vk.ntheory.basic :as b])
```

```
(require '[vk.ntheory.primes :as p])
(require '[vk.ntheory.arithmetic-functions :as af])
(require '[vk.ntheory.congruences :as c])
(require '[vk.ntheory.primitive-roots :as pr])
(require '[clojure.math :as math])
```

So below I will use above aliases.

2 Notation

- **N** - Natural numbers or positive integers $1, 2, 3, \dots$
- **C** - Complex numbers
- **Z** - Integers $\cdots - 3, -2, -1, 0, 1, 2, 3, \dots$
- $\mathbf{Z}/m\mathbf{Z}$ - Ring of integers modulo m
- (a, b) - the greatest common divisor of a and b
- $[a, b]$ - the least common multiple of a and b
- $a \mid b$ - a divides b
- $a \nmid b$ - a does not divide b

3 Some basic functions

This section covers namespace `vk.ntheory.basic`. It contains some common functions, which can be used directly or by other namespaces. One can use `vk.ntheory.basic` namespace as:

```
(require '[vk.ntheory.basic :as b])
```

3.1 Check functions

There are set of `check-*` functions which can be helpful to validate user input:

- `check-int`
- `check-int-pos`

- `check-int-non-neg`
- `check-int-non-zero`

All of above accept one argument, check does argument satisfy to expectation, if does return argument, otherwise throws an exception.

```
(b/check-int 5)
```

```
5
```

Some check functions accept more than one argument. Those functions throw an exception if expectation failed and returns vector of input arguments otherwise:

- `check-relatively-prime`
- `check-not-divides`
- `check-at-least-one-non-zero`

```
(b/check-relatively-prime 5 6)
```

```
[5 6]
```

There are also two helper functions `check-true` and `check-predicate` which helps to implement another `check-*` functions.

3.2 Some predicates

Function `divides?` determine does one number divides another.

```
(b/divides? 2 8)
```

```
true
```

Function `relatively-prime?` determine does two numbers relatively prime.

```
(b/relatively-prime? 2 3)
```

```
true
```

3.3 Combinatorics

Function `product` returns cartesian product of n sequences. For instance,

```
(b/product [(range 1 3) (range 1 5)])
```

```
((1 1) (1 2) (1 3) (1 4) (2 1) (2 2) (2 3) (2 4))
```

3.4 Operations in $\mathbf{Z}/m\mathbf{Z}$

Integer a is congruent to integer b modulo $m > 0$ if m divides $a - b$.

$$a \equiv b \mod m$$

There is a function `congruent?` which compare does two integers are congruent modulo m. For instance $19 \equiv 7 \pmod{12}$

```
(b/congruent? 12 19 7)
```

```
true
```

Similar to addition function `+` and multiplication function `*` there are defined addition modulo m `mod-add` and multiplication modulo m `mod-mul` functions. For instance $2 + 4 \equiv 1 \pmod{5}$ in $\mathbf{Z}/m\mathbf{Z}$

```
(b/mod-add 5 2 4)
```

```
1
```

and $2 \cdot 4 \equiv 3 \pmod 5$ in $\mathbf{Z}/m\mathbf{Z}$

```
(b/mod-mul 5 2 4)
```

```
3
```

First argument of these functions is a modulo. The fact that modulo is a first argument allow bind modulo in let expression and then use addition and multiplication modulo m without specify a modulo.

```
(let [mul (partial b/mod-mul 5)
      add (partial b/mod-add 5)]
  ;; ...
  (add 1 (mul 2 4)))
```

```
4
```

There is another helpful function modulo m - exponentiation. It is implementation of a fast exponentiation algorithm described in D.Knuth, The Art of Computer Programming, Volume II. and in book Structure and Interpretation of Computer Programs (SICP) of Harold Abelson and Gerald Jay Sussman. For instance, $101^{900} \equiv 701 \pmod{997}$.

```
(b/mod-pow 997 101 900)
```

```
701
```

3.5 Power function

Clojure has built-in `clojure.math/pow` function, but it return `java.lang.Double`. The library provide integer analog. Also the implementation share fast exponentiation algorithm with `mod-pow` function.

```
(b/pow 2 3)
```

```
8
```

3.6 Order function

Order function $ord_p(n)$ is a greatest power of p divides n. For instance, $2^3|24$, but $2^4 \nmid 24$, so $ord_2(24) = 3$

```
(b/order 2 24)
```

```
3
```

3.7 Sign function

The `sign` function defined as follows:

$$sign(n) = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ 1 & \text{if } x > 0 \end{cases}$$

```
(mapv b/sign [(- 5) 10 0])
```

```
[-1 1 0]
```

3.8 The greatest common divisor

The greatest common divisor of two integers a and b, not both zero, is the largest positive integer d which divides both a and b.

```
(b/gcd 12 18)
```

```
6
```

The greatest common divisors of a and b is denoted by (a, b). Furthermore, for any two integers a and b exists integers s and t such that $as+bt = d$, where d is the greatest common divisor. For example, $6 = 12(-1) + 18(1)$

```
(b/gcd-extended 12 18)
```

```
[6 [-1 1]]
```

3.9 The least common multiple

The least common multiple of two non zero integers a and b is denoted by $[a, b]$, is an smallest positive integer which is multiple of a and b. It defined in code as follows:

$$[a, b] = \frac{|ab|}{(a, b)}$$

```
(b/lcm 12 18)
```

```
36
```

4 Primes and integer factorization

This section cover namespace `vk.ntheory.primes`. It primary designed for integer factorization and get list of primes. One can use `vk.ntheory.primes` namespace as:

```
(require '[vk.ntheory.primes :as p])
```

4.1 Check functions

Addition to `vk.nthery.basic` namespace, namespace `vk.ntheory.primes` provides additional set of `check-*` functions:

- `check-int-pos-max`
- `check-int-non-neg-max`
- `check-int-non-zero-max`

It is similar to `vk.ntheory.basic` check functions, but additionally check that given number does not exceed `max-int` constant. And there are some more check functions:

- `check-prime`
- `check-odd-prime`

4.2 Performance and cache

This library is designed to work with relatively small integers. Library keep in cache least prime divisor table for fast integer factorization. Least prime divisor of an positive integer is least divisor, but not `1`. Cache grows automatically. The strategy of growing is extends cache to the least power of `10` more than required number. For instance, if client asked to factorize number `18`, cache grows to `100`, if client asked to factorize number `343`, cache grows to `1000`. List of primes also cached and recalculated together with least prime divisor table. Recalculation is not incremental, but every recalculation of least prime divisor table make a table which is in `10` times more than previous, and time for previous calculation is `10` times less than for new one. So we can say that recalculation spent almost all time for recalculate latest least prime divisor table.

Internally, least prime divisor table is java array of `int`, so to store least prime divisor table for first `1 000 000` number approximately `4M` memory is required, `4` bytes per number.

There is a limit for max size of least prime divisor table. It is value of `max-int`:

```
p/max-int
```

```
10000000
```

Cache can be reset:

```
(p/cache-reset!)
```

```
{:least-divisor-table , :primes , :upper 0}
```

Least prime divisor table is implementation details, but one can see it:

```
;; load first 10 numbers into cache
(p/int->factors-map 5)
(deref p/cache)
```

```
{:least-divisor-table [0, 1, 2, 3, 2, 5, 2, 7, 2, 3, 2],
 :primes (2 3 5 7),
 :upper 10}
```

For number `n` least prime divisor table contains least prime divisor of number `n` at index `n`. For instance, least prime divisor of number `6` is `2`. If number `n > 1` is a prime, least prime divisor is `n` and conversely. So at index `7` least prime divisor table contains `7`. Index zero is not used, index `1` is a special case and value for index `1` is `1`.

4.3 Primes

`primes` function returns prime numbers which not exceeds given `n`.

```
(p/primes 30)
```

```
(2 3 5 7 11 13 17 19 23 29)
```

4.4 Integer factorization

Integer p is a prime if

- $p > 1$
- p has just two divisors, namely 1 and p.

There is `prime?` predicate:

```
(p/prime? 7)
```

```
true
```

Integer n is a composite number if

- $n > 1$

- n has at least one divisor except 1 and n

There is `composite?` predicate:

```
(p/composite? 12)
```

```
true
```

Integer 1 is not a prime and it is not a composite

```
(p/unit? 1)
```

```
true
```

So all natural numbers can be divided into 3 categories: prime, composite and unit.

Every integer more than 1 can be represented uniquely as a product of primes.

$$n = {p_1}^{a_1} {p_2}^{a_2} \dots {p_k}^{a_k}$$

or we can write it in more compact form:

$$n = \prod_{i=1}^{k} {p_i}^{a_i}$$

or even write as:

$$n = \prod_{p|n} p^a$$

For example, $360 = 2^3 3^2 5^1$.

There are some functions to factorize integers. Each of them accept natural number as an argument and returns factorized value. It have slightly different output, which may be more appropriate to different use cases. For each factorize function there is also inverse function, which accept factorized value and convert it back to integer. It is convenient to consider empty product as 1. All factorization functions below returns corresponded empty data structure for argument 1.

1-st factorization representation is ordered sequence of primes:

```
(p/int->factors 360)
```

```
(2 2 2 3 3 5)
```

And reverse function is:

```
(p/factors->int [2 2 2 3 3 5])
```

```
360
```

2-nd factorization representation is ordered sequence of primes grouped into partitions by a prime:

```
(p/int->factors-partitions 360)
```

```
((2 2 2) (3 3) (5))
```

And reverse function is:

```
(p/factors-partitions->int [[2 2 2] [3 3] [5]])
```

```
360
```

3-rd factorization representation is ordered sequence of pairs `[p k]`, where `p` is a prime and `k` is a power of prime:

```
(p/int->factors-count 360)
```

```
([2 3] [3 2] [5 1])
```

And reverse function is:

```
(p/factors-count->int [[2 3] [3 2] [5 1]])
```

```
360
```

4-th factorization representation is very similar to 3-rd, but it is a map instead of sequence of pairs.

```
(p/int->factors-map 360)
```

```
{2 3, 3 2, 5 1}
```

Reverse function is the same as for 3-rd representation:

```
(p/factors-count->int {2 3, 3 2, 5 1})
```

```
360
```

5-th factorization representation is sequence of coprime integers, that's it every pair of factors from the sequence are relatively prime. If we take 4-th factorization and raise every prime to appropriate power we get 5-th factorization. For instance, $2^3 3^2 5^1 = 8 \cdot 9 \cdot 5$

```
(p/int->coprime-factors 360)
```

```
(8 9 5)
```

Reverse function is the same as for 1-st representation

```
(p/factors->int [8 9 5])
```

```
360
```

Implementation of factorization functions use least prime divisor table. Actually least prime divisor table is a kind of linked list, to get ordered list of all divisors of an integer `n`, need to get least prime divisor at index `n`, let it be `p`, `p` is a first element of the list, then divide `n` on `p`, the index of quotient `n/p` is next element of "linked list".

5 Arithmetical functions

This section cover namespace `vk.ntheory.primes`. It contains some well known arithmetical functions and also functions which allow build new arithmetical functions.

```
(require '[vk.ntheory.arithmetic-functions :as af])
```

5.1 Arithmetical function

Arithmetical function is an any function which accept natural number and return complex number $f : \mathbf{N} \to \mathbf{C}$. The library mostly works with functions which also returns integer $f : \mathbf{N} \to \mathbf{Z}$.

5.2 Function equality

Two arithmetical function f and g are equal if $f(n) = g(n)$ for all natual n. There is helper function `f=` which compare two functions on some sequence(sample) of natual numbers. Function `f=` accept two functions and optionally sequence of natural numbers. There is a default for sequence of natural numbers, it is a variable `default-natural-sample`, which is currently `range(1,100)`.

```
(take 10 af/default-natural-sample)
```

```
(1 2 3 4 5 6 7 8 9 10)
```

If we like identify does two function `f` and `g` equals on some sequence of natural number we can for example do next:

```
;; Let we have some f and g
(def f identity)
```

```
(def g (constantly 1))
;; Then we able to check does those functions are equals
(af/f= f g) ;; true
(af/f= f g (range 1 1000)) ;; true
(af/f= f g (filter even? (range 1 100))) ;; true
```

5.3 Pointwise addition

For two functions f an g pointwise addition defined as follows:

$$(f + g)(n) = f(n) + f(n)$$

In Clojure function `f+` returns pointwise addition:

```
(let [f #(* % %)
      g #(* 2 %)]
 ((af/f+ f g) 3))
```

```
15
```

5.4 Pointwise multiplication

For two functions f an g pointwise multiplication defined as follows:

$$(f \cdot g)(n) = f(n) \cdot f(n)$$

In clojure function `f*` returns pointwise multiplication:

```
(let [f #(* % %)
      g #(* 2 %)]
 ((af/f* f g) 3))
```

```
54
```

5.5 Divisors

Some arithmetic functions and Dirichlet convolutions need to iterate over positive divisors on an integer. For get list of all positive divisors of number `n` there is `divisor` function. List of divisors is unordered.

```
(af/divisors 30)
```

```
(1 5 3 15 2 10 6 30)
```

5.6 Additive functions

Additive function is a function for which

$$f(mn) = f(m) + f(n)$$

if m relatively prime to n. If above equality holds for all natural m and n function called completely additive.

To define an additive function it is enough to define how to calculate a function on power of primes. If $n = p_1^{a_1} p_2^{a_2} \dots p_k^{a_k}$ then:

$$f(n) = \sum_{i=1}^{k} f({p_i}^{a_i})$$

5.7 Multiplicative functions

Multiplicative function is a function not equal to zero for all n for which

$$f(mn) = f(m)f(n)$$

if m relatively prime to n. If above equality holds for all natural m and n function called completely multiplicative.

To define multiplicative function it is enough to define how to calculate a function on power of primes. If $n = p_1^{a_1} p_2^{a_2} \dots p_k^{a_k}$ then:

$$f(n) = \prod_{i=1}^{k} f({p_i}^{a_i})$$

5.8 Higher order function for define multiplicative and additive functions

As we have seen, to define either multiplicative or additive function it is enough define function on power of a prime. There is helper function `reduce-on-prime-count` which provide a way to define a function on power of a prime. The first parameter of `reduce-on-prime-count` is reduce function which usually `*` for multiplicative function and usually `+` for additive function, but custom reduce function also acceptable.

For instance, we can define function which calculate number of divisors of integer `n`. If $n = p_1^{a_1} p_2^{a_2} \dots p_k^{a_k}$ count of divisors of number `n` can be calculated by formula:

$$d(n) = \prod_{i=1}^{k} (a_i + 1)$$

With helper function it can be defined as

```
(def my-divisors-count
(af/reduce-on-prime-count * (fn [p k] (inc k))))
(my-divisors-count 6)
```

```
4
```

Of course there is predefined function `divisors-count`, but it is an example how to define custom function.

5.9 Some additive functions

5.9.1 Count of distinct primes - ω

Count of distinct primes is a number of distinct primes which divides given n. If $n = p_1^{a_1} p_2^{a_2} \dots p_k^{a_k}$ then $\omega = k$.

```
(af/primes-count-distinct (* 2 2 3))
```

```
2
```

5.9.2 Total count of primes - Ω

Total count of primes is a number of primes and power of primes which divides n. If $n = p_1^{a_1} p_2^{a_2} \dots p_k^{a_k}$ then:

$$\Omega = a_1 + a_2 + \cdots + a_k$$

```
(af/primes-count-total (* 2 2 3))
```

```
3
```

5.10 Some multiplicative functions

5.10.1 Mobius function - μ.

Mobius function μ is defined as follows:

$$\mu(n) = \begin{cases} 1 & \text{if } n = 1 \\ (-1)^k & \text{if } n \text{ product of distinct primes} \\ 0 & \text{otherwise} \end{cases}$$

For example, $\mu(6) = \mu(2 \cdot 3) = 1$

```
(af/mobius 6)
```

```
1
```

5.10.2 Euler totient function - φ

Euler totient function $\varphi(n)$ is a number of positive integers not exceeding n which are relatively prime to n. It can be calculated as follows:

$$\varphi(n) = \prod_{p|n} (p^a - p^{a-1})$$

For example, count of numbers relative prime to 6 are 1 and 5, so $\varphi(6) = 2$

```
(af/totient 6)
```

```
2
```

5.10.3 Unit function - ε

Unit or identity function defined as follows:

$$\varepsilon(n) = \begin{cases} 1, & \text{if } n = 1 \\ 0, & \text{if } n > 1 \end{cases}$$

```
(af/unit 6)
```

```
0
```

The name `unit` was chosen to make it different from `clojure.core/identity` function.

5.10.4 Constant one function - 1

Constant one function $\mathbf{1}(n)$ defiened as follows:

$$\mathbf{1}(n) = 1$$

```
(af/one 6)
```

```
1
```

5.10.5 Divisors count - σ_0

Function divisors count is a number of positive divisors which divides given number n.

$$\sigma_0(n) = \sum_{d|n} 1$$

Function $\sigma_0(n)$ is often denoted as $d(n)$. For example, number 6 has 4 divisors, namely $1, 2, 3, 6$, so $d(6) = 4$.

```
(af/divisors-count 6)
```

```
4
```

5.10.6 Divisors sum - σ_1

Function divisors sum is sum of positive divisors which divides given number n

$$\sigma_1(n) = \sum_{d|n} d$$

Function σ_1 is often denoted as σ. For instance, $\sigma(6) = 1+2+3+6 = 12$

```
(af/divisors-sum 6)
```

```
12
```

5.10.7 Divisors square sum

Function divisors square sum defined as follows:

$$\sigma_2(n) = \sum_{d|n} d^2$$

For instance, $\sigma_2(6) = 1^2 + 2^2 + 3^2 + 6^2 = 50$

```
(af/divisors-square-sum 6)
```

```
50
```

5.10.8 Divisors higher order function - σ_x

In general σ_x function is a sum of x-th powers divisors of given n

$$\sigma_x(n) = \sum_{d|n} d^x$$

If $x \neq 0$, σ_x can be calculated as follows:

$$\sigma_x(n) = \prod_{i=1}^{k} \frac{p_i^{x(a_i+1)}}{p_i^x - 1}$$

and if $x = 0$ as follows:

$$\sigma_0(n) = \prod_{i=1}^{k} (a_i + 1)$$

There is higher order function `divisors-sum-x` which accept `x` and return appropriate function.

For example we can define divisors cube sum as follows:

```
(def my-divisors-cube-sum (af/divisors-sum-x 3))
```

5.10.9 Liouville function - λ

Liouville function λ defined as $\lambda(1) = 1$ and if $n = p_1^{a_1} p_2^{a_2} \dots p_k^{a_k}$ $\lambda(n) = (-1)^{a_1+a_2+\dots+a_k}$ or with early defined Ω function we can write definition of λ as follows:

$$\lambda(n) = (-1)^{\Omega}$$

```
(af/liouville (* 2 3))
```

```
1
```

Liouville function is completely multiplicative.

5.11 Some other arithmetic functions

5.11.1 Mangoldt function - Λ

Mangoldt function Λ defined as follows:

$$\Lambda(n) = \begin{cases} \log p, & \text{if } n = p^k \text{ for some prime } p \text{ and some } k \geq 1 \\ 0, & \text{otherwise} \end{cases}$$

For example $\Lambda(8) = \log 2$, $\Lambda(6) = 0$

```
(af/mangoldt 2)
```

```
0.6931471805599453
```

5.11.2 Chebyshev functions θ and ψ

If $x > 0$ Chebyshev θ function is defined as follows:

$$\theta(x) = \sum_{p \leq x} \log p$$

```
(af/chebyshev-theta 2)
```

```
0.6931471805599453
```

If $x > 0$ Chebyshev ψ function is defined as follows:

$$\psi = \sum_{n \leq x} \Lambda(n)$$

```
(af/chebyshev-psi 2)
```

```
0.6931471805599453
```

5.12 Dirichlet convolution

For two arithmetic functions f and g Dirichlet convolution is a new arithmetic function defined as follows:

$$(f * g)(n) = \sum_{d|n} f(d)g(\frac{n}{d})$$

Dirichlet convolution is associative:

$$(f * g) * h = f * (g * h)$$

Commutative:

$$f * g = g * f$$

Has identify:

$$f * \varepsilon = \varepsilon * f = f$$

For every f, which $f(1) \neq 0$ exists inverse function f^{-1} such that $f * f^{-1} = \varepsilon$. This inverse function called Dirichlet inverse and can by calculated recursively by formula:

$$f^{-1}(n) = \begin{cases} \frac{1}{f(1)} & \text{if } n = 1 \\ \frac{-1}{f(1)} \sum_{\substack{d|n \\ d<n}} f(\frac{n}{d}) f^{-1}(d) & \text{if } n > 1 \end{cases}$$

In clojure, function `d*` calculate Dirichlet convolution:

```
(af/d* af/one af/one)
```

Function `inverse` calculate Dirichlet inverse:

```
(af/inverse af/one)
```

Dirichlet convolution is associative so clojure function `d*` support more than two function as an argument.

```
(af/d* af/mobius af/one af/mobius af/one)
```

```
#function[vk.ntheory.arithmetic-functions/d*/fn--6199]
```

Functions $\mu(n)$ and $1(n)$ are inverse of each other, we can easy check this

```
(af/f= (af/inverse af/one) af/mobius)
```

```
true
```

and conversely

```
(af/f= (af/inverse af/mobius) af/one)
```

```
true
```

Function `inverse` defined as recursive function, it may execute slow. But inverse of completely multiplicative function $f(n)$ is $f(n) \cdot \mu(n)$(usual,pointwise multiplication), for instance inverse of identity function, let's denote it as $N(n)$, is $N(n) \cdot \mu(n)$

```
(af/f=
 (af/d*
    (af/f* identity af/mobius)
    identity
 )
 af/unit)
```

```
true
```

6 Congruences

This section cover namespaces `vk.ntheory.congruence`. It contains functions for solve any congruence with brute force approach and also contains solutions for specific congruences such as linear congruence and system of linear congruences.

```
(require '[vk.ntheory.congruence :as c])
```

6.1 Brute force solution

If we have a congruence

$$f(x) \equiv 0 \pmod{m}$$

we can solve it by try all m residue classes modulo m.There is `solve` function for this. It accept two aruments, first argument is a some function of one argument and second arument is a modulo. Let's for example solve congruence $x^2 \equiv 1 \pmod{8}$

```
(let [f (fn [x] (dec (* x x)))]
  (c/solve f 8)
  )
```

```
(1 3 5 7)
```

6.2 Linear congruence

Let consider linear congruence

$$ax \equiv b \pmod{m}$$

There is function `solve-linear` for solve linear congruence. It accepts 3 arguments `a`, `b` and `m`. Let's solve congruence $6x \equiv 3 \pmod{15}$

```
(c/solve-linear 6 3 15)
```

```
#{3 8 13}
```

6.3 System of linear congruences

Let consider system of linear congruences:

$$\begin{aligned}
x &\equiv c_1 \pmod{m_1} \\
x &\equiv c_2 \pmod{m_2} \\
&\vdots \\
x &\equiv c_n \pmod{m_n}
\end{aligned}$$

There is a function `solve-remaindes` for solve such system. It accepts a sequence of pairs $([c_1, m_1], [c_2, m_2], \dots, [c_n, m_n])$ and returns pair $[r, M]$, where M is the least common multiple of $m_1, m_2, \dots, m_n$, and r is residue to modulo M.

Let's solve system:

$$
\begin{aligned}
x &\equiv 2 \pmod{7} \\
x &\equiv 5 \pmod{9} \\
x &\equiv 11 \pmod{15}
\end{aligned}
$$

```
(c/solve-remainders [[2 7][5 9][11 15]])
```

```
[86 315]
```

So the answer is $86 \pmod{315}$

6.4 Coprime moduli case

When we have system of linear conguences:

$$
\begin{aligned}
x &\equiv c_1 \pmod{m_1} \\
x &\equiv c_2 \pmod{m_2} \\
&\vdots \\
x &\equiv c_n \pmod{m_n}
\end{aligned}
$$

and any pair of moduli relatively prime, i.e. $(m_i, m_j) = 1$ if $i \neq j$, the the system has one solution modulo the product $m_1 m_2 \dots m_n$. This statement called Chinese Remainder Theorem. We can solve such system with `solve-remainder` function, but there is another function `solve-coprime-remainders`. It accepts a sequence of pairs $([c_1, m_1], [c_2, m_2], \dots, [c_n, m_n])$ and returns pair $[r, M]$, where M is the product $m_1 m_2 \dots m_n$, and r is residue to modulo M.

Let's solve system:

$$
\begin{aligned}
x &\equiv 6 \pmod{17} \\
x &\equiv 4 \pmod{11} \\
x &\equiv -3 \pmod{8}
\end{aligned}
$$

```
(c/solve-remainders [[6 17][4 11][-3 8]])
```

```
[125 1496]
```

So the answer is 125 (mod 1496)

7 Primitive roots

This section cover namespace `vk.ntheory.primitive-roots`.

It contains functions for test modulo for primitive roots, find primitive roots. If modulo has a primitive root there are functions for find index, test for n-th power residue, solve n-th power residue congruence, get all n-th power residues. Also there is a function for find reduced residues by any modulo.

```
(require '[vk.ntheory.primitive-roots :as pr])
```

7.1 Reduced residues

Function `reduced-residues` return reduced residues modulo m.

```
(pr/reduced-residues 12)
```

```
(7 11 1 5)
```

7.2 Order

Order n of integer a coprime to m modulo m is least positive integer for which $a^n \equiv 1 \pmod{m}$.

```
(pr/order 2 11)
```

```
1
```

7.3 Primitive root

If order a modulo m is equal to $\varphi(m)$, a called primitive root. Primitive roots exists only for moduli $1, 2, 4, p^a, 2p^a$, where p odd prime. To find any primitive root there is fucntion `find-primitive-root`.

```
(pr/find-primitive-root 11)
```

```
2
```

There is a predicate `primitive-root?` to check does given number a is a primitive root modulo m

```
(pr/primitive-root? 2 11)
```

```
true
```

To find all primitive roots there is function `primitive-roots`.

```
(pr/primitive-roots 11)
```

```
(7 6 2 8)
```

7.4 Index

If m has a primitive root g and $(a, m) = 1$ there is an unique integer k, $0 \leq k \leq \varphi(m) - 1$ such that

$$a \equiv g^k \pmod m$$

Number k is called an index of a to the base g

$$k = ind_g a$$

There is clojure function `index [m g a]` to calculate index.

```
(pr/index 11 2 5)
```

```
4
```

Often it doesn't matter which primitive root used as base. There is another arity of function `index [m a]`, where g is value get by `find-primitive-root` function, which is 2 for modulo 11.

```
(pr/index 11 5)
```

```
4
```

7.5 n-th power residues

If $(a, m) = 1$ and congruence $x^n \equiv a \pmod{m}$ has a solution, number a is called n-th power residue mod m. Otherwise number a is called n-th power non residue mod m.

To check does given number power residue there is a function `power-residue?`. For instance 4 is 2 power residue mod 11

```
(pr/power-residue? 11 2 4)
```

```
true
```

If need to get solutions of equation $x^n \equiv a \pmod{m}$ there is another function ~solve-power-residue$. For instance for get a solution of equation $x^2 \equiv 4 \pmod{m}$

```
(pr/solve-power-residue 11 2 4)
```

```
#{2 9}
```

If need to get all n-th power residues modulo m there is function `power-residues`. For instance for get all a fro which $x^2 \equiv a \pmod{m}$

```
(pr/power-residues 11 2)
```

```
(1 4 5 9 3)
```