Number Theory with Clojure

Valery Kocubinsky

March 28, 2024

Contents

1 About

This project cover some topics in number theory such as integer factorization, arithmetic functions, congruences, primitive roots. Here defined set of well known arithmetic functions and one can define custom arithmetic function. One can solve linear congruence or system of linear congruences including case when moduli relatively prime.

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 show readme.md if a project has such file. Github markdown looks enough good, even math equation is supported. However for now math equation doesn't rendered for link text. I also generate readme.pdf , 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 `C-u C-u C-c C-x C-l` for preview all math equations inside Emacs.

Many application 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 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 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 caclulated 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 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, it is another modification of Sieve of Eratosthenes procedure to build full factorization for all numberS $1, 2, \ldots, N$. 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 a best compromise between build table time, required memory for table, integer factorization time. So now factorization is cheap, but it require 4 bytes per number or 4MB for 1000000.

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.congruence :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, positive integers $1, 2, 3, \ldots$
- **C** - Complex numbers
- **Z** - Integers $\cdots -3, -2, -1, 0, 1, 2, 3, \ldots$
- $\mathbf{Z}/m\mathbf{Z}$ - 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

3 Some basic functions `vk.ntheory.basic`

This section cover namespace `vk.ntheory.basic`. It contains some common functions, which can be used directly or by other namespaces.

```
(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 throw an exception.

There are also two helper function `check` and `check-not` which helps to implement another `check-*` function for a predicate.

3.2 Some predicates

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

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

```
true
```

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

Similar to addition function `+` and multiplication function `*` there defined addition modulo m `m+` and multiplication modulo m `m*` functions. First argument of these functions is a modulo.

For instance $2 + 4 \equiv 1 \pmod 5$ in $\mathbf{Z}/m\mathbf{Z}$

```
(b/m+ 5 2 4)
```

```
1
```

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

```
(b/m* 5 2 4)
```

```
3
```

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

```
(let [m5* (partial b/m* 5)
      m5+ (partial b/m+ 5)]
  ;; ...
  (m5+ 1 (m5* 2 4)))
```

```
4
```

There is another helpful function modulo m - exponentiation. It is a fast binary exponentiation algorithm described in D.Knuth, The Art of Computer Programming, Volume II.

For instance, $101^{900} \equiv 701 \pmod{997}$

```
(b/m** 997 101 900)
```

```
701
```

3.4 Power function

Clojure has built-in `clojure.math/pow` function, but it return `java.lang.Double`. The library provide integer analog.

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

```
8
```

3.5 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
```

3.6 Sign function

The `sign` function

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

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

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

3.7 The greatest common divisor

The greatest common divisor of two integer a and b is an positive integer d which divides a and b , and any other common divisor a and b divides d.

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

```
6
```

The greatest common divisors of a and b is denoted by (a, b). For convenience $(0, 0) = 0$.

Furthermore, if 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]
```

3.8 The least common multiple

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

$$[a, b] = \begin{cases} \frac{|ab|}{(a,b)} & \text{if } a \neq 0 \text{ and } b \neq 0 \\ 0 & \text{if } a = 0 \text{ or } b = 0 \end{cases}$$

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

```
36
```

4 Primes and Integer Factorization `vk.ntheory.primes`

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

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

4.1 Performance and cache

This library is designed to work with realtive 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
```

```
1000000
```

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`.

4.2 Primes

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

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

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

4.3 Integer factorization

Integer p is a prime if

- $p > 1$
- has only divisors 1 and p.

There is `prime?` function:

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

```
true
```

Integer n is a composite number if

- $n > 1$
- has at least one proper divisor, i.e. divisor except 1 and p

There is `composite?` function:

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

```
true
```

Integer 1 is not a prime and is not a composite

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

```
true
```

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

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:

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

If we accept that empty product is 1 we can say that every natural number can be represent uniquely as a product of primes. 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.

1-st factorization representation is ordered sequence of primes:

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

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

And converse function is:

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

```
360
```

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

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

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

And converse 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 converse 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}
```

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

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

```
360
```

Implementation of factorization use least prime divisor table. Actually least prime divisor table is a kind of linked list, to get next least prime divisor of an integer `n` need just divide `n` on least prime divisor `p`, and quotient `n/p` is an index of next least prime divisor of integer `n/p` and therefore divisor `n`.

4.4 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 exceeds `max-int` constant. And there are some more check functions:

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

5 Arithmetical functions `vk.ntheory.arithmetic-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))
```

```
class clojure.lang.Compiler$CompilerException
```

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 2 3 6 5 10 15 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:

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

```
(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

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

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.ar-func/d*/fn--10570]
```

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 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 Conguences `vk.ntheory.congruence`

This section cover namespaces `vk.ntheory.congruence`. It contains functions for solve linear congruence and system of linear congruences.

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

In progress ...

7 Primitive Roots `vk.ntheory.primitive-roots`

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

In progress...

8 Quadratic residies ~vk.ntheory.quadratic-residues

```
(require '[vk.ntheory.quadratic-residues :as qr])
```

In progress...