Instruction of *integer_converter.r* (R)

Writer: Hao Jiang

Version: 0.1.2

Date: June 24, 2022

Contents

1. Introduction

The R file *integer_converter.r* provides some extra functions for integers, particularly for the calculation of integers with large number of digits. With *integer_converter.r*, it is good for study of integers, research of cryptography and simulation. Because it is not designed to speed up the computation, but to sustain as many functions as possible in dealing with integers with large number of digits, it is not appropriate to be installed directly into the applications related to cryptography or something else, which require calculation speed for integers with large number of digits.

- Current version 0.1.2, June 24, 2022. The bug in *integer_pm_seq_index* is fixed; the extended Euclidean algorithm is installed in *integer_mod_inverse*; the modular inverse of b in *integer_pm_seq_complete_root_condition* is calculated by Carmichael's λ, other than Euler's ϕ in previous versions; some more functions are added.
- Version 0.1.1, June 6, 2022. The script of R is also provided; some more functions are added.
- Version 0.1.0, June 1, 2022. This is the initial script of Python.

2. Integer Format

In *integer_converter.r*, the format of integers, which can be used for all the following functions, is a *numeric & vector*, where the first element is -1 or 0 or 1 (1 represents positive integer, -1 represents negative integer, 0 represents zero) and the following elements of type *numeric* are the digits of the integer. Alongside with the integer of type *numeric & vector*, a radix of type *numeric*, which is greater than or equal to 2, is also required.

For example, 123456 with radix 1000 is

```
a = c(1, 456, 123)
print(a)
#------------------------------

1 456 123
```

-123456 with radix 2 is

```
a = c(-1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1)
print(a)
#------------------------------

-1  0  0  0  0  0  0  1  0  0  1  0  0  0  1  1  1  1
```

where the higher index of the element in the *numeric & vector*, the element is with the higher power of radix.

Zero is

```
a = c(0, 0)
print(a)
#------------------------------

0 0
```

3. Check Function, Conversion Functions, Comparison Functions

3.1. integer_check

```
integer_check = function(x, radix)
```

The function to check whether *x* of type *numeric & vector* is a valid format of an integer with *radix*, like

```
rad = 16
a = c(1, 15, 3)
b = integer_check(x = a, radix = rad)
print(b)
#------------------------------

TRUE
```

If these exists one digit in *x* which is greater than or equal to the radix, then, the integer is invalid, like

```
rad = 16
a = c(1, 16, 3)
b = integer_check(x = a, radix = rad)
print(b)
#------------------------------

FALSE
```

If *x* is not zero, and the highest digit is 0, then, the integer is invalid, like

```
rad = 16
a = c(-1, 15, 3, 0)
b = integer_check(x = a, radix = rad)
print(b)
#------------------------------

FALSE
```

If the *radix* is less than or equal to 1, then, the integer is invalid, like

```
rad = 1
a = c(-1, 15, 3)
b = integer_check(x = a, radix = rad)
print(b)
#------------------------------

FALSE
```

3.2. integer_num_2_vec

```
integer_num_2_vec = function(x, radix, check = TRUE)
```

The function to convert *x* of type *numeric* to the integer of type *numeric & vector* with *radix*, like

```
rad = 16
a = -63
b = integer_num_2_vec(x = a, radix = rad)
print(b)
#-------------------------------

-1 15  3
```

where the argument *check* is whether to scan the validity of other arguments.

3.3. integer_vec_2_num

```
integer_vec_2_num = function(x, radix, check = TRUE)
```

The function to convert *x* of type *numeric & vector* with *radix* to the variable of *numeric*, like

```
rad = 16
a = c(-1, 15, 3)
b = integer_vec_2_num(x = a, radix = rad)
print(b)
#-------------------------------

-63
```

where the argument *check* is whether to scan the validity of other arguments.

3.4. Integer_vec_change_radix

```
integer_vec_change_radix = function(x, origin_radix, new_radix,
check = TRUE)
```

The function to convert *x* of type *numeric & vector* with *origin_radix* to the integer with *new_radix*, like

```
rad1 = 16
rad2 = 10
a = c(-1, 15, 3)
b = integer_vec_change_radix(x = a,
                             origin_radix = rad1,
                             new_radix = rad2)
print(b)
#------------------------------

-1  3  6
```

where the argument *check* is whether to scan the validity of other arguments.

3.5. integer_less_vec_vec

```
integer_less_vec_vec = function(x, y, radix, check = TRUE)
```

The function to check whether *x* of type *numeric & vector* with *radix* is less than *y* of type *numeric & vector* with *radix*, like

```
rad = 1000
a = 1234
b = integer_num_2_vec(a, rad)
c = 5678
d = integer_num_2_vec(c, rad)
e1 = integer_less_vec_vec(x = b, y = d,
                          radix = rad)
e2 = integer_less_vec_vec(x = b, y = b,
                          radix = rad)
e3 = integer_less_vec_vec(x = d, y = b,
                          radix = rad)
#------------------------------

> print(e1)
TRUE
> print(e2)
FALSE
> print(e3)
FALSE
```

where the argument *check* is whether to scan the validity of other arguments.

3.6. integer_lesseq_vec_vec

```
integer_lesseq_vec_vec = function(x, y, radix, check = TRUE)
```

The function to check whether *x* of type *numeric & vector* with *radix* is less than or equal to *y* of type *numeric & vector* with *radix*, like

```
rad = 1000
a = 1234
b = integer_num_2_vec(a, rad)
c = 5678
d = integer_num_2_vec(c, rad)
e1 = integer_lesseq_vec_vec(x = b, y = d,
                            radix = rad)
e2 = integer_lesseq_vec_vec(x = b, y = b,
                            radix = rad)
e3 = integer_lesseq_vec_vec(x = d, y = b,
                            radix = rad)
#-------------------------------

> print(e1)
TRUE
> print(e2)
TRUE
> print(e3)
FALSE
```

where the argument *check* is whether to scan the validity of other arguments.

3.7. integer_more_vec_vec

```
integer_more_vec_vec = function(x, y, radix, check = TRUE)
```

The function to check whether *x* of type *numeric & vector* with *radix* is more than *y* of type *numeric & vector* with *radix*, like

```
rad = 1000
a = 1234
b = integer_num_2_vec(a, rad)
c = 5678
d = integer_num_2_vec(c, rad)
e1 = integer_more_vec_vec(x = b, y = d,
                          radix = rad)
e2 = integer_more_vec_vec(x = b, y = b,
                          radix = rad)
e3 = integer_more_vec_vec(x = d, y = b,
                          radix = rad)
#-------------------------------

> print(e1)
FALSE
> print(e2)
FALSE
> print(e3)
TRUE
```

where the argument *check* is whether to scan the validity of other arguments.

3.8. integer_moreeq_vec_vec

```
integer_moreeq_vec_vec = function(x, y, radix, check = TRUE)
```

The function to check whether *x* of type *numeric & vector* with *radix* is more than or equal to *y* of type *numeric & vector* with *radix*, like

```
rad = 1000
a = 1234
b = integer_num_2_vec(a, rad)
c = 5678
d = integer_num_2_vec(c, rad)
e1 = integer_moreeq_vec_vec(x = b, y = d,
                            radix = rad)
e2 = integer_moreeq_vec_vec(x = b, y = b,
                            radix = rad)
e3 = integer_moreeq_vec_vec(x = d, y = b,
                            radix = rad)
#------------------------------

> print(e1)
FALSE
> print(e2)
TRUE
> print(e3)
TRUE
```

where the argument *check* is whether to scan the validity of other arguments.

3.9. integer_eq_vec_vec

```
integer_eq_vec_vec = function(x, y, radix, check = TRUE)
```

The function to check whether *x* of type *numeric & vector* with *radix* is equal to *y* of type *numeric & vector* with *radix*, like

```
rad = 1000
a = 1234
b = integer_num_2_vec(a, rad)
c = 5678
d = integer_num_2_vec(c, rad)
e1 = integer_eq_vec_vec(x = b, y = d,
                        radix = rad)
e2 = integer_eq_vec_vec(x = b, y = b,
                        radix = rad)
e3 = integer_eq_vec_vec(x = d, y = b,
                        radix = rad)
#-------------------------------

> print(e1)
FALSE
> print(e2)
TRUE
> print(e3)
FALSE
```

where the argument *check* is whether to scan the validity of other arguments.

3.10. integer_noteq_vec_vec

```
integer_noteq_vec_vec = function(x, y, radix, check = TRUE)
```

The function to check whether *x* of type *numeric & vector* with *radix* is not equal to *y* of type *numeric & vector* with *radix*, like

```
rad = 1000
a = 1234
b = integer_num_2_vec(a, rad)
c = 5678
d = integer_num_2_vec(c, rad)
e1 = integer_noteq_vec_vec(x = b, y = d,
                           radix = rad)
e2 = integer_noteq_vec_vec(x = b, y = b,
                           radix = rad)
e3 = integer_noteq_vec_vec(x = d, y = b,
                           radix = rad)
#-------------------------------

> print(e1)
TRUE
> print(e2)
FALSE
> print(e3)
TRUE
```

where the argument *check* is whether to scan the validity of other arguments.

3.11. integer_is_even

```
integer_is_even = function(x, radix, check = TRUE)
```

The function to check whether *x* of type *numeric & vector* with *radix* is an even integer, like

```
rad = 1000
a = 123
b = integer_num_2_vec(a, rad)
c = -234
d = integer_num_2_vec(c, rad)
e1 = integer_is_even(x = b, radix = rad)
e2 = integer_is_even(x = d, radix = rad)
#------------------------------

> print(e1)
FALSE
> print(e2)
TRUE
```

where the argument *check* is whether to scan the validity of other arguments.

4. Basic Operation Functions

4.1. integer_inverse

```
integer_inverse = function(x, radix, check = TRUE)
```

The function to return the inverse value of *x* of type *numeric & vector* with *radix*, like

```
rad = 1000
a = 123456
a = integer_num_2_vec(a, rad)
b = integer_inverse(x = a, radix = rad)
print(b)
#------------------------------

-1 456 123
```

where the argument *check* is whether to scan the validity of other arguments.

4.2. integer_absolute

```
integer_absolute = function(x, radix, check = TRUE)
```

The function to return the absolute value of *x* of type *numeric & vector* with *radix*, like

```
rad = 1000
a = -123456
a = integer_num_2_vec(a, rad)
b = integer_absolute(x = a, radix = rad)
print(b)
#------------------------------

1 456 123
```

where the argument *check* is whether to scan the validity of other arguments.

4.3. integer_plus

```
integer_plus = function(x, y, radix, check = TRUE)
```

The function to calculate the sum (x+y) of x and y of type *numeric & vector* with *radix*, like

```
rad = 1000
a = -5678
a = integer_num_2_vec(a, rad)
b = 1234
b = integer_num_2_vec(b, rad)
c = integer_plus(x = a, y = b, radix = rad)
print(c)
#------------------------------

-1 444   4
```

where the argument *check* is whether to scan the validity of other arguments.

4.4. integer_minus

```
integer_minus = function(x, y, radix, check = TRUE)
```

The function to calculate the difference (*x*-*y*) of *x* and *y* of type *numeric & vector* with *radix*, like

```
rad = 1000
a = -5678
a = integer_num_2_vec(a, rad)
b = 1234
b = integer_num_2_vec(b, rad)
c = integer_minus(x = a, y = b, radix = rad)
print(c)
#------------------------------

-1 912   6
```

where the argument *check* is whether to scan the validity of other arguments.

4.5. integer_multiply

```
integer_multiply = function(x, y, radix, check = TRUE)
```

The function to calculate the product (*x*×*y*) of *x* and *y* of type *numeric & vector* with *radix*, like

```
rad = 1000
a = -5678
a = integer_num_2_vec(a, rad)
b = 1234
b = integer_num_2_vec(b, rad)
c = integer_multiply(x = a, y = b, radix = rad)
print(c)
#------------------------------

-1 652   6   7
```

where the argument *check* is whether to scan the validity of other arguments.

4.6. integer_modulo

```
integer_modulo = function(x, y, radix, check = TRUE)
```

The function to calculate the modulo of *x* and *y* of type *numeric & vector* with *radix*, like

```
rad = 1000
a = -5678
a = integer_num_2_vec(a, rad)
b = 1234
b = integer_num_2_vec(b, rad)
c = integer_modulo(x = a, y = b, radix = rad)
print(c)
#------------------------------

$quo
-1  5

$rem
1 492
```

where the argument *check* is whether to scan the validity of other arguments, and the output is the *list*, in which *quo* is the quotient and *rem* is the remainder.

4.7. integer_power

```
integer_power = function(x, y, radix, check = TRUE)
```

The function to calculate the power of base *x* and exponent *y* of type *numeric & vector* with *radix*, like

```
rad = 1000
a = -123
a = integer_num_2_vec(a, rad)
b = 45
b = integer_num_2_vec(b, rad)
c = integer_power(x = a, y = b, radix = rad)
print(c)
#-------------------------------

 -1  43  48 127 966 283 743 242   1 111 866 199
 73 629 823  21 268 212 153 559 918 451 176 587
790 910 285 956 131 185 408 110  11
```

where the argument *check* is whether to scan the validity of other arguments, and *y* should be a non-negative integer.

Specially, $0^0 = 1$.

4.8. integer_factorial

```
integer_factorial = function(x, radix, check = TRUE)
```

The function to calculate the factorial of *x* of type *numeric & vector* with *radix*, like

```
rad = 1000
a = 30
a = integer_num_2_vec(a, rad)
b = integer_factorial(x = a, radix = rad)
print(b)
#-------------------------------

1   0   0 480 308 636  58 191 812 859 252 265
```

where the argument *check* is whether to scan the validity of other arguments.

4.9. integer_permute

```
integer_permute = function(m, n, radix, check = TRUE)
```

The function to calculate (*m* permute *n*), where *m* and *n* are of type *numeric & vector* with *radix*, like

```
rad = 1000
a = 30
a = integer_num_2_vec(a, rad)
b = 3
b = integer_num_2_vec(b, rad)
c = integer_permute(m = a, n = b, radix = rad)
print(c)
#------------------------------

1 360  24
```

where the argument *check* is whether to scan the validity of other arguments.

4.10. integer_choose

```
integer_choose = function(m, n, radix, check = TRUE)
```

The function to calculate (*m* choose *n*), where *m* and *n* are of type *numeric & vector* with *radix*, like

```
rad = 1000
a = 30
a = integer_num_2_vec(a, rad)
b = 3
b = integer_num_2_vec(b, rad)
c = integer_choose(m = a, n = b, radix = rad)
print(c)
#------------------------------

1 60  4
```

where the argument *check* is whether to scan the validity of other arguments.

4.11. integer_mod_inverse

```
integer_mod_inverse = function(x, m, radix, check = TRUE)
```

The function to calculate the modular inverse integer of *x* modulo *m*, where *m* and *n* are of type *numeric & vector* with *radix*, like

```
rad = 1000
a = 15
a = integer_num_2_vec(a, rad)
b = 22
b = integer_num_2_vec(b, rad)
c = integer_mod_inverse(x = a, m = b,
                        radix = rad)
print(c)
#-------------------------------

1  3
```

where the argument *check* is whether to scan the validity of other arguments.

In the above example, it is

$$(15 \times 3)\ mod\ 22 = 1$$

4.12. integer_crt

```
integer_crt = function(r, m, radix, check = TRUE)
```

The function to calculate the number of the Chinese remainder theorem, where *r* is the *list* of remainders, *m* is the *list* of divisors. For example,

```
rad = 1000
r1 = 3
r1 = integer_num_2_vec(r1, rad)
r2 = 4
r2 = integer_num_2_vec(r2, rad)
r3 = 5
r3 = integer_num_2_vec(r3, rad)
r_list = list(r1, r2, r3)
a = 35
a = integer_num_2_vec(a, rad)
b = 22
b = integer_num_2_vec(b, rad)
c = 17
c = integer_num_2_vec(c, rad)
m_list = list(a, b, c)
d = integer_crt(r = r_list, m = m_list,
                radix = rad)
print(d)
#-------------------------------

1  378   11
```

where the argument *check* is whether to scan the validity of other arguments.

In the above example, it is

$$\begin{cases} 11378\ mod\ 35 = 3 \\ 11378\ mod\ 22 = 4 \\ 11378\ mod\ 17 = 5 \end{cases}$$

5. Prime-Related, Composite-Related Functions

5.1. integer_is_prime

```
integer_is_prime = function(x, radix, check = TRUE)
```

The function to check whether *x* of type *numeric & vector* with *radix* is a prime or composite, like

```
rad = 1000
a = 4641919
a = integer_num_2_vec(a, rad)
b = integer_is_prime(x = a, radix = rad)
print(b)
#------------------------------

TRUE
```

or

```
rad = 1000
a = -4641920
a = integer_num_2_vec(a, rad)
b = integer_is_prime(x = a, radix = rad)
print(b)
#------------------------------

FALSE
```

where the argument *check* is whether to scan the validity of other arguments.

If *x* is -1 or 0 or 1, it will return *NULL*.

5.2. integer_factorization

```
integer_factorization = function(x, radix, check = TRUE)
```

The function to calculate the prime factors of *x* of type *numeric & vector* with *radix*, like

```
rad = 1000
a = -4641920
a = integer_num_2_vec(a, rad)
b = integer_factorization(x = a, radix = rad)
print(b)
#------------------------------

$prime
$prime[[1]]
1  2

$prime[[2]]
1  5

$prime[[3]]
1  253   7


$power
$power[[1]]
1  7

$power[[2]]
1  1

$power[[3]]
1  1
```

where the argument *check* is whether to scan the validity of other arguments, and the output is the *list*, in which *prime* is of the prime factors and *power* is of the corresponding power of the prime factors.

In the above example, it is

$$-4641914 = -2^7 \times 5^1 \times 7253^1$$

5.3. integer_divisors

```
integer_divisors = function(x, radix, check = TRUE)
```

The function to calculate all the divisors of *x* of type *numeric & vector* with *radix*, like

```
rad = 1000
a = (3^2)*7*11
a = integer_num_2_vec(a, rad)
b = integer_divisors(x = a, radix = rad)
print(b)
#-------------------------------

[[1]]
1 1
[[2]]
1 3
[[3]]
1 7
[[4]]
1 9
[[5]]
1 11
[[6]]
1 21
[[7]]
1 33
[[8]]
1 63
[[9]]
1 77
[[10]]
1 99
[[11]]
1 231
[[12]]
1 693
```

where the argument *check* is whether to scan the validity of other arguments.

In the above example, $\{1, 3, 7, 9, 11, 21, 33, 63, 77, 99, 231, 693\}$ is the set of divisors of $3^2 \times 7 \times 11 = 693$.

5.4. integer_Euler_phi

```
integer_Euler_phi = function(x, radix, check = TRUE)
```

The function to calculate the Euler's ϕ of *x* of type *numeric & vector* with *radix*, like

```
rad = 1000
a = (3**2) * 7
a = integer_num_2_vec(a, rad)
b = integer_Euler_phi(x = a, radix = rad)
print(b)
#-------------------------------

1 36
```

where the argument *check* is whether to scan the validity of other arguments.

In the above example, it is

$$\phi(3^2 \times 7 = 63) = [3^{2-1} \times (3-1)] \times [7^{1-1} \times (7-1)] = 36$$

5.5. integer_gcd

```
integer_gcd = function(x, y, radix, check = TRUE)
```

The function to calculate the greatest common divisor of *x* and *y* of type *numeric & vector* with *radix*, like

```
rad = 1000
a = 75
a = integer_num_2_vec(a, rad)
b = -125
b = integer_num_2_vec(b, rad)
c = integer_gcd(x = a, y = b, radix = rad)
print(c)
#-------------------------------

1 25
```

where the argument *check* is whether to scan the validity of other arguments.

5.6. integer_is_relatively_primes

```
integer_is_relatively_primes = function(x, y, radix, check = TRUE)
```

The function to calculate check whether *x* and *y* of type *numeric & vector* with *radix* are relatively primes, like

```
rad = 1000
a = 75
a = integer_num_2_vec(a, rad)
b = -125
b = integer_num_2_vec(b, rad)
c = 77
c = integer_num_2_vec(c, rad)
d = integer_is_relatively_primes(x = a, y = b, radix = rad)
e = integer_is_relatively_primes(x = a, y = c, radix = rad)
#------------------------------

> print(d)
[1] FALSE
> print(e)
[1] TRUE
```

where the argument *check* is whether to scan the validity of other arguments.

5.7. integer_lcm

```
integer_lcm = function(x, y, radix, check = TRUE)
```

The function to calculate the least common multiple of *x* and *y* of type *numeric & vector* with *radix*, like

```
rad = 1000
a = 75
a = integer_num_2_vec(a, rad)
b = -125
b = integer_num_2_vec(b, rad)
c = integer_lcm(x = a, y = b, radix = rad)
print(c)
#------------------------------

1 375
```

where the argument *check* is whether to scan the validity of other arguments.

5.8. integer_Carmichael_lambda

```
integer_Carmichael_lambda = function(x, radix, check = TRUE)
```

The function to calculate the Carmichael's λ of *x* of type *numeric & vector* with *radix*, like

```
rad = 1000
a = (2**2)*7
a = integer_int_2_tuple(a, rad)
b = integer_Carmichael_lambda(x = a, radix = rad)
print(b)
#------------------------------

1 6
```

where the argument *check* is whether to scan the validity of other arguments.

In the above example, it is

$$\lambda(2^2 \times 7 = 28) = \mathrm{lcm}[\lambda(2^2), \lambda(7)] = \mathrm{lcm}[2, 6] = 6$$

6. Functions of Arithmetic-Modulo Sequence

6.1. integer_am_seq_complete_root_condition

```
integer_am_seq_complete_root_condition = function(c, a, m, radix)
```

The function to check whether the integers *c*, *a*, *m* can construct an arithmetic-modulo sequence (AM sequence) with shift term *a*

$$(a_1, a_2, \dots, a_n, \dots)$$

where $a_n = c(n + a)\ mod\ m$, and *c* is the complete root.

For example,

```
rad = 1000
c0 = 3
c0 = integer_num_2_vec(c0, rad)
a0 = 10
a0 = integer_num_2_vec(a0, rad)
m0 = 25
m0 = integer_num_2_vec(m0, rad)
condition = integer_am_seq_complete_root_condition(c = c0,
                                                   a = a0,
                                                   m = m0,
                                                   radix = rad)
print(condition)
#-------------------------------
$encode
$encode$type
"am_seq"

$encode$c
1 3

$encode$a
1 10

$encode$m
1 25

$decode
$decode$type
"am_seq"

$decode$c
1 17

$decode$a
1 20

$decode$m
1 25
```

where the output is the *list*, in which *encode* is of the parameters of AM sequence conversion and *decode* is of the parameters for the inverse conversion.

6.2. integer_am_seq_convert

```
integer_am_seq_convert = function(x, am_seq_list, radix)
```

The function to convert from *x* to a_x in the arithmetic-modulo sequence (AM sequence) with shift term *a*

$$(a_1, a_2, \dots, a_n, \dots)$$

where $a_n = c(n + a)\ mod\ m$, and *c* is the complete root.

For example, if the argument of *am_seq_list* is the *encode* from *integer_am_seq_complete_root_condition*, then,

```
rad = 1000
c0 = 3
c0 = integer_num_2_vec(c0, rad)
a0 = 10
a0 = integer_num_2_vec(a0, rad)
m0 = 25
m0 = integer_num_2_vec(m0, rad)
condition = integer_am_seq_complete_root_condition(c0,
                                                    a0,
                                                    m0,
                                                    rad)
encode_condition = condition[["encode"]]
decode_condition = condition[["decode"]]

n = 5
n = integer_num_2_vec(n, rad)
an = integer_am_seq_convert(x = n,
                            am_seq_list = encode_condition,
                            radix = rad)
print(an)
#-------------------------------

1 20
```

and if *n* is 20 and argument of *am_seq_list* is the *decode* from *integer_am_seq_complete_root_condition*, then,

```
rad = 1000
c0 = 3
c0 = integer_num_2_vec(c0, rad)
a0 = 10
a0 = integer_num_2_vec(a0, rad)
m0 = 25
m0 = integer_num_2_vec(m0, rad)
condition = integer_am_seq_complete_root_condition(c0,
                                                    a0,
                                                    m0,
                                                    rad)
encode_condition = condition[["encode"]]
decode_condition = condition[["decode"]]

n = 20
n = integer_num_2_vec(n, rad)
an = integer_am_seq_convert(x = n,
                            am_seq_list = decode_condition,
                            radix = rad)
print(an)
#------------------------------

1 5
```

7. Functions of Type-1 Geometric-Modulo Sequence

7.1. integer_t1gm_seq_primitive_root_condition

```
integer_t1gm_seq_primitive_root_condition = function(c, a, p,
p_power, radix, double = FALSE)
```

The function to check whether the integers *c*, *a*, *m* can construct a type-1 geometric-modulo sequence (T1GM sequence) with coefficient *c*

$$(g_1, g_2, \dots, g_n, \dots)$$

where $g_n = ca^n \; mod \; m$, and

$$m = \begin{cases} p^{\text{p\_power}}, & \text{if double is False} \\ 2p^{\text{p\_power}}, & \text{if double is True} \end{cases}, \qquad p \text{ is a prime}$$

and *a* is the primitive root.

For example,

```
rad = 1000
c0 = 1
c0 = integer_num_2_vec(c0, rad)
a0 = 2
a0 = integer_num_2_vec(a0, rad)
p0 = 11
p0 = integer_num_2_vec(p0, rad)
p_power0 = 3
p_power0 = integer_num_2_vec(p_power0, rad)
condition = integer_t1gm_seq_primitive_root_condition(c = c0,
                                                      a = a0,
                                                      p = p0,
                                                      p_power =
p_power0,
                                                      radix = rad,
                                                      double =
FALSE)
print(condition)
#-------------------------------

$type
"t1gm_seq_primitive_root"

$c
1 1

$a
1 2

$m
1 331   1

$m_prime
$m_prime[[1]]
1 11


$m_power
$m_power[[1]]
1 3


$phi_m
1 210   1

$phi_phi_m
1 440
```

which reveals 2 is the primitive root of the T1GM sequence modulo $11^3 = 1331$.

7.2. integer_t1gm_seq_convert

```
integer_t1gm_seq_convert = function(x, t1pm_seq_pr_list, radix)
```

The function to convert from *x* to g_x in the type-1 geometric-modulo sequence (T1GM sequence) with coefficient *c*

$$(g_1, g_2, \dots, g_n, \dots)$$

where $g_n = ca^n \; mod \; m$, and

$$m = \begin{cases} p^{\text{p\_power}}, & \text{if double is False} \\ 2p^{\text{p\_power}}, & \text{if double is True} \end{cases}, \qquad p \text{ is a prime}$$

and *a* is the primitive root.

For example,

```
rad = 1000
c0 = 1
c0 = integer_num_2_vec(c0, rad)
a0 = 5
a0 = integer_num_2_vec(a0, rad)
p0 = 23
p0 = integer_num_2_vec(p0, rad)
p_power0 = 1
p_power0 = integer_num_2_vec(p_power0, rad)
condition = integer_t1gm_seq_primitive_root_condition(c0,
                                                       a0,
                                                       p0,
                                                       p_power0,
                                                       rad)

n = 12
n = integer_num_2_vec(n, rad)
gn = integer_t1gm_seq_convert(x = n,
                              t1pm_seq_pr_list = condition,
                              radix = rad)
print(gn)
#-------------------------------

1 18
```

where 5 is a primitive root of T1GM sequence modulo 23, and

$$5^{12} \; mod \; 23 = 18$$

7.3. integer_t1gm_seq_index

```
integer_t1gm_seq_index = function(x, t1pm_seq_pr_list, radix)
```

The function to find the index of *x* in the type-1 geometric-modulo sequence (T1GM sequence) with coefficient *c*

$$(g_1, g_2, \dots, g_n, \dots)$$

where $g_n = ca^n \; mod \; m$, and

$$m = \begin{cases} p^{\text{p\_power}}, & \text{if double is False} \\ 2p^{\text{p\_power}}, & \text{if double is True} \end{cases}, \qquad p \text{ is a prime}$$

and a is the primitive root.

For example,

```
rad = 1000
c0 = 1
c0 = integer_num_2_vec(c0, rad)
a0 = 5
a0 = integer_num_2_vec(a0, rad)
p0 = 23
p0 = integer_num_2_vec(p0, rad)
p_power0 = 1
p_power0 = integer_num_2_vec(p_power0, rad)
condition = integer_t1gm_seq_primitive_root_condition(c0,
                                                       a0,
                                                       p0,
                                                       p_power0,
                                                       rad)

g = 18
g = integer_num_2_vec(g, rad)
n = integer_t1gm_seq_index(x = g,
                           t1pm_seq_pr_list = condition,
                           radix = rad)
print(n)
#-------------------------------

1 12
```

8. Functions of Power-Modulo Sequence

8.1. integer_pm_seq_complete_root_condition

```
integer_pm_seq_complete_root_condition = function(c, b, p_list,
radix, inverse = TRUE)
```

The function to check whether the integers *c*, *b*, *m* can construct a power-modulo sequence (T1GM sequence) with coefficient *c*

$$(p_1, p_2, \dots, p_n, \dots)$$

where $p_n = cn^b \; mod \; m$, and

$$m = \prod q_i \text{, where } q_i \text{ are the primes from p\_vec}$$

and b is the complete root.

For example,

```
rad = 1000
c0 = 1
c0 = integer_num_2_vec(c0, rad)
b0 = 5
b0 = integer_num_2_vec(b0, rad)
p0 = 37
p0 = integer_num_2_vec(p0, rad)
p1 = 23
p1 = integer_num_2_vec(p1, rad)
p_list = list(p0, p1)
condition = integer_pm_seq_complete_root_condition(c = c0,
                                                     b = b0,
                                                     p_list =
p_list,
                                                     radix = rad,
                                                     inverse = TRUE)
print(condition)
#------------------------------

$type
"pm_seq_complete_root"

$c
1 1

$b
1 5

$m
1 851

$m_prime
$m_prime[[1]]
1 23

$m_prime[[2]]
1 37


$phi_m
1 792

$phi_phi_m
1 240

$mod_inverse_c
1 1

$mod_inverse_b
1 317
```

8.2. integer_pm_seq_convert

```
integer_pm_seq_convert = function(x, pm_seq_cr_list, radix)
```

The function to convert from x to p_x in the power-modulo sequence (PM sequence) with coefficient c

$$(p_1, p_2, \dots, p_n, \dots)$$

where $p_n = cn^b \; mod \; m$, and

$$m = \prod q_i \text{ , where } q_i \text{ are the primes from p\_vec}$$

and b is the complete root.

For example,

```
rad = 1000
c0 = 1
c0 = integer_num_2_vec(c0, rad)
b0 = 5
b0 = integer_num_2_vec(b0, rad)
p0 = 37
p0 = integer_num_2_vec(p0, rad)
p1 = 23
p1 = integer_num_2_vec(p1, rad)
p_list = list(p0, p1)
condition = integer_pm_seq_complete_root_condition(c = c0,
                                                    b = b0,
                                                    p_list =
p_list,
                                                    radix = rad,
                                                    inverse = TRUE)

n = 18
n = integer_num_2_vec(n, rad)
pn = integer_pm_seq_convert(x = n,
                            pm_seq_cr_list = condition,
                            radix = rad)
print(pn)
#-------------------------------

1 348
```

where 5 is a complete root of PM sequence modulo $23 \times 37 = 851$, and

$$18^5 \; mod \; 851 = 345$$

8.3. integer_pm_seq_index

```
integer_pm_seq_index = function(x, pm_seq_cr_list, radix)
```

The function to find the index of *x* in the power-modulo sequence (PM sequence) with coefficient c

$$(p_1, p_2, \ldots, p_n, \ldots)$$

where $p_n = cn^b \; mod \; m$, and

$$m = \prod q_i \text{ , where } q_i \text{ are the primes from p\_vec}$$

and *b* is the complete root.

For example,

```
rad = 1000
c0 = 1
c0 = integer_num_2_vec(c0, rad)
b0 = 5
b0 = integer_num_2_vec(b0, rad)
p0 = 37
p0 = integer_num_2_vec(p0, rad)
p1 = 23
p1 = integer_num_2_vec(p1, rad)
p_list = list(p0, p1)
condition = integer_pm_seq_complete_root_condition(c = c0,
                                                    b = b0,
                                                    p_list =
p_list,
                                                    radix = rad,
                                                    inverse = TRUE)

p = 348
p = integer_num_2_vec(p, rad)
n = integer_pm_seq_index(x = p,
                         pm_seq_cr_list = condition,
                         radix = rad)
print(n)
#-------------------------------

1 18
```