Instruction of *integer_converter.py*

Writer: Hao Jiang

Version: 0.1.0

Date: June 1, 2022

Contents

1. Introduction

The Python file *integer_converter.py* provides some extra functions for integers, particularly for the calculation of integers with large number of digits. With *integer_converter.py*, 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, 0.1.0.

2. Integer Format

In *integer_converter.py*, the format of integers, which can be used for all the following functions, is a *list* or a *tuple*, where the first element is of type *bool* (*True* represents positive integer, *False* represents negative integer) and the following elements of type *int* are the digits of the integer. Alongside with the integer of type *list* or *tuple*, a radix of type *int*, which is greater than or equal to 2, is also required.

For example, 123456 with radix 1000 is

```
a = (True, 456, 123)
print(a)
#------------------------------

(True, 456, 123)
```

-123456 with radix 2 is

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

(False, 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 *list* or *tuple*, the element is with the higher power of radix.

Zero is

```
a = [True, 0]
print(a)
#------------------------------

[True, 0]
```

or

```
a = [False, 0]
print(a)
#------------------------------

[False, 0]
```

3. Check Function and Conversion Functions

3.1. integer_check

```
integer_check(x, radix)
```

The function to check whether *x* of type *list* or *tuple* is a valid format of an integer with *radix*, like

```
rad = 16
a = [False, 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 = [False, 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 = [False, 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 = [False, 15, 3]
b = integer_check(x = a, radix = rad)
print(b)
#------------------------------

False
```

3.2. integer_int_2_tuple

```
integer_int_2_tuple(x, radix, check = True)
```

The function to convert *x* of type *int* to the integer of type *list* or *tuple* with *radix*, like

```
rad = 16
a = -63
b = integer_int_2_tuple(x = a, radix = rad)
print(b)
#-------------------------------

(False, 15, 3)
```

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

3.3. integer_tuple_2_int

```
integer_tuple_2_int(x, radix, check = True)
```

The function to convert *x* of type *list* or *tuple* with *radix* to the variable of *int*, like

```
rad = 16
a = (False, 15, 3)
b = integer_tuple_2_int(x = a, radix = rad)
print(b)
#-------------------------------

-63
```

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

3.4. integer_tuple_change_radix

```
integer_tuple_change_radix(x, origin_radix, new_radix, check =
True)
```

The function to convert *x* of type *list* or *tuple* with *origin_radix* to the integer with *new_radix*, like

```
rad1 = 16
rad2 = 10
a = (False, 15, 3)
b = integer_tuple_change_radix(x = a,
                               origin_radix = rad1,
                               new_radix = rad2)
print(b)
#------------------------------

 (False, 3, 6)
```

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

4. Comparison Functions

4.1. check_less_tuple_tuple

```
check_less_tuple_tuple(x, y, radix, check = True)
```

The function to check whether *x* of type *list* or *tuple* with *radix* is less than *y* of type *list* or *tuple* with *radix*, like

```
rad = 1000
a = 1234
b = integer_int_2_tuple(a, rad)
c = 5678
d = integer_int_2_tuple(c, rad)
e1 = check_less_tuple_tuple(x = b, y = d,
                            radix = rad)
e2 = check_less_tuple_tuple(x = b, y = b,
                            radix = rad)
e3 = check_less_tuple_tuple(x = d, y = b,
                            radix = rad)
print(e1)
print(e2)
print(e3)
#------------------------------

True
False
False
```

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

4.2. check_lesseq_tuple_tuple

```
check_lesseq_tuple_tuple(x, y, radix, check = True)
```

The function to check whether *x* of type *list* or *tuple* with *radix* is less than or equal to *y* of type *list* or *tuple* with *radix*, like

```
rad = 1000
a = 1234
b = integer_int_2_tuple(a, rad)
c = 5678
d = integer_int_2_tuple(c, rad)
e1 = check_lesseq_tuple_tuple(x = b, y = d,
                              radix = rad)
e2 = check_lesseq_tuple_tuple(x = b, y = b,
                              radix = rad)
e3 = check_lesseq_tuple_tuple(x = d, y = b,
                              radix = rad)
print(e1)
print(e2)
print(e3)
#-------------------------------

True
True
False
```

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

4.3. check_more_tuple_tuple

```
check_more_tuple_tuple(x, y, radix, check = True)
```

The function to check whether *x* of type *list* or *tuple* with *radix* is more than *y* of type *list* or *tuple* with *radix*, like

```
rad = 1000
a = 1234
b = integer_int_2_tuple(a, rad)
c = 5678
d = integer_int_2_tuple(c, rad)
e1 = check_more_tuple_tuple(x = b, y = d,
                            radix = rad)
e2 = check_more_tuple_tuple(x = b, y = b,
                            radix = rad)
e3 = check_more_tuple_tuple(x = d, y = b,
                            radix = rad)
print(e1)
print(e2)
print(e3)
#-------------------------------

False
False
True
```

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

4.4. check_moreeq_tuple_tuple

```
check_moreeq_tuple_tuple(x, y, radix, check = True)
```

The function to check whether *x* of type *list* or *tuple* with *radix* is more than or equal to *y* of type *list* or *tuple* with *radix*, like

```
rad = 1000
a = 1234
b = integer_int_2_tuple(a, rad)
c = 5678
d = integer_int_2_tuple(c, rad)
e1 = check_moreeq_tuple_tuple(x = b, y = d,
                              radix = rad)
e2 = check_moreeq_tuple_tuple(x = b, y = b,
                              radix = rad)
e3 = check_moreeq_tuple_tuple(x = d, y = b,
                              radix = rad)
print(e1)
print(e2)
print(e3)
#------------------------------

False
True
True
```

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

4.5. check_eq_tuple_tuple

```
check_eq_tuple_tuple(x, y, radix, check = True)
```

The function to check whether *x* of type *list* or *tuple* with *radix* is equal to *y* of type *list* or *tuple* with *radix*, like

```
rad = 1000
a = 1234
b = integer_int_2_tuple(a, rad)
c = 5678
d = integer_int_2_tuple(c, rad)
e1 = check_eq_tuple_tuple(x = b, y = d,
                          radix = rad)
e2 = check_eq_tuple_tuple(x = b, y = b,
                          radix = rad)
e3 = check_eq_tuple_tuple(x = d, y = b,
                          radix = rad)
print(e1)
print(e2)
print(e3)
#------------------------------

False
True
False
```

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

4.6. check_noteq_tuple_tuple

```
check_noteq_tuple_tuple(x, y, radix, check = True)
```

The function to check whether *x* of type *list* or *tuple* with *radix* is not equal to *y* of type *list* or *tuple* with *radix*, like

```
rad = 1000
a = 1234
b = integer_int_2_tuple(a, rad)
c = 5678
d = integer_int_2_tuple(c, rad)
e1 = check_noteq_tuple_tuple(x = b, y = d,
                             radix = rad)
e2 = check_noteq_tuple_tuple(x = b, y = b,
                             radix = rad)
e3 = check_noteq_tuple_tuple(x = d, y = b,
                             radix = rad)
print(e1)
print(e2)
print(e3)
#------------------------------

True
False
True
```

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

5. Basic Operation Functions

5.1. integer_inverse

```
integer_inverse(x, radix, check = True)
```

The function to return the inverse value of *x* of type *list* or *tuple* with *radix*, like

```
rad = 1000
a = 123456
a = integer_int_2_tuple(a, rad)
b = integer_inverse(x = a, radix = rad)
print(b)
#------------------------------

(False, 456, 123)
```

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

5.2. integer_absolute

```
integer_absolute(x, radix, check = True)
```

The function to return the absolute value of *x* of type *list* or *tuple* with *radix*, like

```
rad = 1000
a = -123456
a = integer_int_2_tuple(a, rad)
b = integer_absolute(x = a, radix = rad)
print(b)
#------------------------------

(True, 456, 123)
```

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

5.3. integer_plus

```
integer_plus(x, y, radix, check = True)
```

The function to calculate the sum (*x*+*y*) of *x* and *y* of type *list* or *tuple* with *radix*, like

```
rad = 1000
a = 1234
a = integer_int_2_tuple(a, rad)
b = 5678
b = integer_int_2_tuple(b, rad)
c = integer_plus(x = a, y = b, radix = rad)
print(c)
#------------------------------

(True, 912, 6)
```

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

5.4. integer_minus

```
integer_minus(x, y, radix, check = True)
```

The function to calculate the difference (*x*-*y*) of *x* and *y* of type *list* or *tuple* with *radix*, like

```
rad = 1000
a = 1234
a = integer_int_2_tuple(a, rad)
b = 5678
b = integer_int_2_tuple(b, rad)
c = integer_minus(x = a, y = b, radix = rad)
print(c)
#------------------------------

(False, 444, 4)
```

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

5.5. integer_multiply

```
integer_multiply(x, y, radix, check = True)
```

The function to calculate the product (*x*×*y*) of *x* and *y* of type *list* or *tuple* with *radix*, like

```
rad = 1000
a = 1234
a = integer_int_2_tuple(a, rad)
b = -5678
b = integer_int_2_tuple(b, rad)
c = integer_multiply(x = a, y = b, radix = rad)
print(c)
#------------------------------

(False, 652, 6, 7)
```

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

5.6. integer_modulo

```
integer_modulo(x, y, radix, check = True)
```

The function to calculate the modulo of *x* and *y* of type *list* or *tuple* with *radix*, like

```
rad = 1000
a = 1234
a = integer_int_2_tuple(a, rad)
b = -5678
b = integer_int_2_tuple(b, rad)
c = integer_modulo(x = b, y = a, radix = rad)
print(c)
#------------------------------

{'quo': (False, 5), 'rem': (True, 492)}
```

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

5.7. integer_power

```
integer_power(x, y, radix, check = True)
```

The function to calculate the power of base *x* and exponent *y* of type *list* or *tuple* with *radix*, like

```
rad = 1000
a = -123
a = integer_int_2_tuple(a, rad)
b = 45
b = integer_int_2_tuple(b, rad)
c = integer_power(x = a, y = b, radix = rad)
print(c)
#------------------------------

(False, 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, $0^0 = 1$.

6. Other Operation Functions

6.1. integer_factorial

```
integer_factorial(x, radix, check = True)
```

The function to calculate the factorial of *x* of type *list* or *tuple* with *radix*, like

```
rad = 1000
a = 30
a = integer_int_2_tuple(a, rad)
b = integer_factorial(x = a, radix = rad)
print(b)
#------------------------------

(True, 0, 0, 480, 308, 636, 58, 191, 812, 859, 252, 265)
```

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

6.2. integer_permute

```
integer_permute(m, n, radix, check = True)
```

The function to calculate (*m* permute *n*), where *m* and *n* are of type *list* or *tuple* with *radix*, like

```
rad = 1000
a = 30
a = integer_int_2_tuple(a, rad)
b = 3
b = integer_int_2_tuple(b, rad)
c = integer_permute(m = a, n = b, radix = rad)
print(c)
#------------------------------

(True, 360, 24)
```

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

6.3. integer_choose

```
integer_choose(m, n, radix, check = True)
```

The function to calculate (*m* choose *n*), where *m* and *n* are of type *list* or *tuple* with *radix*, like

```
rad = 1000
a = 30
a = integer_int_2_tuple(a, rad)
b = 3
b = integer_int_2_tuple(b, rad)
c = integer_choose(m = a, n = b, radix = rad)
print(c)
#-------------------------------

(True, 60, 4)
```

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

6.4. integer_modular_inverse

```
integer_modular_inverse(x, m, radix, check = True)
```

The function to calculate the modular inverse integer of *x* modulo *m*, where *m* and *n* are of type *list* or *tuple* with *radix*, like

```
rad = 1000
a = 15
a = integer_int_2_tuple(a, rad)
b = 22
b = integer_int_2_tuple(b, rad)
c = integer_modular_inverse(x = a,
                            m = b,
                            radix = rad)
print(c)
#-------------------------------

(True, 3)
```

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

In the above example, it is

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

6.5. integer_crt

```
integer_crt(r, m, radix, check = True)
```

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

```
rad = 1000
r1 = 3
r1 = integer_int_2_tuple(r1, rad)
r2 = 4
r2 = integer_int_2_tuple(r2, rad)
r3 = 5
r3 = integer_int_2_tuple(r3, rad)
r_bundle = [r1, r2, r3]
a = 35
a = integer_int_2_tuple(a, rad)
b = 22
b = integer_int_2_tuple(b, rad)
c = 17
c = integer_int_2_tuple(c, rad)
m_bundle = [a, b, c]
d = integer_crt(r = r_bundle, m = m_bundle,
                radix = rad)
print(d)
#------------------------------

(True, 378, 11)
```

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

In the above example, it is

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

7. Prime-Related, Composite-Related Functions

7.1. integer_is_prime

```
integer_is_prime(x, radix, check = True)
```

The function to check whether *x* of type *list* or *tuple* with *radix* is a prime or composite, like

```
rad = 1000
a = 4641919
a = integer_int_2_tuple(a, rad)
b = integer_is_prime(x = a, radix = rad)
print(b)
#------------------------------

True
```

or

```
rad = 1000
a = -4641920
a = integer_int_2_tuple(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 *None*.

7.2. integer_factorization

```
integer_factorization(x, radix, check = True)
```

The function to calculate the prime factors of *x* of type *list* or *tuple* with *radix*, like

```
rad = 1000
a = -4641920
a = integer_int_2_tuple(a, rad)
b = integer_factorization(x = a, radix = rad)
print(b)
#------------------------------

{'prime': ((True, 2), (True, 5), (True, 253, 7)), 'power': ((True,
7), (True, 1), (True, 1))}
```

where the argument *check* is whether to scan the validity of other arguments, and the output is the dictionary, 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

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

7.3. integer_gcd

```
integer_gcd(x, y, radix, check = True)
```

The function to calculate the greatest common divisor of *x* and *y* of type *list* or *tuple* with *radix*, like

```
rad = 1000
a = 75
a = integer_int_2_tuple(a, rad)
b = -125
b = integer_int_2_tuple(b, rad)
c = integer_gcd(x = a, y = b, radix = rad)
print(c)
#-----------------------------

(True, 25)
```

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

7.4. integer_lcm

```
integer_lcm(x, y, radix, check = True)
```

The function to calculate the least common multiple of *x* and *y* of type *list* or *tuple* with *radix*, like

```
rad = 1000
a = 75
a = integer_int_2_tuple(a, rad)
b = -125
b = integer_int_2_tuple(b, rad)
c = integer_lcm(x = a, y = b, radix = rad)
print(c)
#------------------------------

(True, 375)
```

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

8. Functions of Arithmetic-Modulo Sequence

8.1. integer_am_seq_complete_root_condition

```
integer_am_seq_complete_root_condition(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_int_2_tuple(c0, rad)
a0 = 10
a0 = integer_int_2_tuple(a0, rad)
m0 = 25
m0 = integer_int_2_tuple(m0, rad)
condition = integer_am_seq_complete_root_condition(c = c0,
                                                   a = a0,
                                                   m = m0,
                                                   radix = rad)
print(condition)
#------------------------------

{'encode': {'type': 'am_seq', 'c': (True, 3), 'a': (True, 10),
'm': (True, 25)}, 'decode': {'type': 'am_seq', 'c': (True, 17),
'a': (True, 20), 'm': (True, 25)}}
```

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

8.2. integer_am_seq_convert

```
integer_am_seq_convert(x, am_seq_dict, 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_dict* is the *encode* from *integer_am_seq_complete_root_condition*, then,

```
rad = 1000
c0 = 3
c0 = integer_int_2_tuple(c0, rad)
a0 = 10
a0 = integer_int_2_tuple(a0, rad)
m0 = 25
m0 = integer_int_2_tuple(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_int_2_tuple(n, rad)
an = integer_am_seq_convert(x = n,
                            am_seq_dict = encode_condition,
                            radix = rad)
print(an)
#------------------------------

(True, 20)
```

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

```
rad = 1000
c0 = 3
c0 = integer_int_2_tuple(c0, rad)
a0 = 10
a0 = integer_int_2_tuple(a0, rad)
m0 = 25
m0 = integer_int_2_tuple(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_int_2_tuple(n, rad)
an = integer_am_seq_convert(x = n,
                            am_seq_dict = decode_condition,
                            radix = rad)
print(an)
#------------------------------

(True, 5)
```

9. Functions of Type-1 Geometric-Modulo Sequence

9.1. integer_t1gm_seq_primitive_root_condition

```
integer_t1gm_seq_primitive_root_condition(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

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

and *a* is the primitive root.

For example,

```
rad = 1000
c0 = 1
c0 = integer_int_2_tuple(c0, rad)
a0 = 2
a0 = integer_int_2_tuple(a0, rad)
p0 = 11
p0 = integer_int_2_tuple(p0, rad)
p_power0 = 3
p_power0 = integer_int_2_tuple(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': (True, 1), 'a': (True,
2), 'm': (True, 331, 1), 'm_prime': ((True, 11),), 'm_power':
((True, 3),), 'phi_m': (True, 210, 1)}
```

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

9.2. integer_t1gm_seq_convert

```
integer_t1gm_seq_convert(x, t1pm_seq_pr_dict, 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_int_2_tuple(c0, rad)
a0 = 5
a0 = integer_int_2_tuple(a0, rad)
p0 = 23
p0 = integer_int_2_tuple(p0, rad)
p_power0 = 1
p_power0 = integer_int_2_tuple(p_power0, rad)
condition = integer_t1gm_seq_primitive_root_condition(c0,
                                                      a0,
                                                      p0,
                                                      p_power0,
                                                      rad)

n = 12
n = integer_int_2_tuple(n, rad)
gn = integer_t1gm_seq_convert(x = n,
                              t1pm_seq_pr_dict = condition,
                              radix = rad)
print(gn)
#-------------------------------

 (True, 18)
```

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

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

9.3. integer_t1gm_seq_index

```
integer_t1gm_seq_index(x, t1pm_seq_pr_dict, 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_int_2_tuple(c0, rad)
a0 = 5
a0 = integer_int_2_tuple(a0, rad)
p0 = 23
p0 = integer_int_2_tuple(p0, rad)
p_power0 = 1
p_power0 = integer_int_2_tuple(p_power0, rad)
condition = integer_t1gm_seq_primitive_root_condition(c0,
                                                       a0,
                                                       p0,
                                                       p_power0,
                                                       rad)

g = 18
g = integer_int_2_tuple(g, rad)
n = integer_t1gm_seq_index(x = g,
                           t1pm_seq_pr_dict = condition,
                           radix = rad)
print(n)
#-------------------------------

(True, 12)
```

10. Functions of Power-Modulo Sequence

10.1. integer_pm_seq_complete_root_condition

```
integer_pm_seq_complete_root_condition(c, b, p_vec, 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_int_2_tuple(c0, rad)
b0 = 5
b0 = integer_int_2_tuple(b0, rad)
p0 = 37
p0 = integer_int_2_tuple(p0, rad)
p1 = 23
p1 = integer_int_2_tuple(p1, rad)
p_bundle = [p0, p1]
condition = integer_pm_seq_complete_root_condition(c = c0,
                                                   b = b0,
                                                   p_vec =
p_bundle,
                                                   radix = rad,
                                                   inverse = True)
print(condition)
#------------------------------

{'type': 'pm_seq_complete_root', 'c': (True, 1), 'b': (True, 5),
'm': (True, 851), 'm_prime': [(True, 23), (True, 37)], 'phi_m':
(True, 792), 'mod_inverse_c': (True, 1), 'mod_inverse_b': (True,
317)}
```

10.2. integer_pm_seq_convert

```
integer_pm_seq_convert(x, pm_seq_cr_dict, 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_int_2_tuple(c0, rad)
b0 = 5
b0 = integer_int_2_tuple(b0, rad)
p0 = 37
p0 = integer_int_2_tuple(p0, rad)
p1 = 23
p1 = integer_int_2_tuple(p1, rad)
p_bundle = [p0, p1]
condition = integer_pm_seq_complete_root_condition(c = c0,
                                                    b = b0,
                                                    p_vec =
p_bundle,
                                                    radix = rad,
                                                    inverse = True)

n = 18
n = integer_int_2_tuple(n, rad)
pn = integer_pm_seq_convert(x = n,
                            pm_seq_cr_dict = condition,
                            radix = rad)
print(pn)
#-------------------------------

(True, 348)
```

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

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

10.3. integer_pm_seq_index

```
integer_pm_seq_index(x, pm_seq_cr_dict, radix)
```

The function to find the index of *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_int_2_tuple(c0, rad)
b0 = 5
b0 = integer_int_2_tuple(b0, rad)
p0 = 37
p0 = integer_int_2_tuple(p0, rad)
p1 = 23
p1 = integer_int_2_tuple(p1, rad)
p_bundle = [p0, p1]
condition = integer_pm_seq_complete_root_condition(c = c0,
                                                    b = b0,
                                                    p_vec =
p_bundle,
                                                    radix = rad,
                                                    inverse = True)

p = 348
p = integer_int_2_tuple(p, rad)
n = integer_pm_seq_index(x = p,
                         pm_seq_cr_dict = condition,
                         radix = rad)
print(n)
#-------------------------------

 (True, 18)
```