Contents

Chapter 1

Functions

1.1 prime – primality test , prime generation

1.1.1 trialDivision – trial division test

trialDivision(`n`: *integer*, `bound`: *integer/float*=0) → *True/False*

Trial division primality test for an odd natural number.

`bound` is a search bound of primes. If it returns 1 under the condition that `bound` is given and less than the square root of `n`, it only means there is no prime factor less than `bound`.

1.1.2 spsp – strong pseudo-prime test

**spsp(`n`: *integer*, `base`: *integer*, `s`: *integer*=None, `t`: *integer*=None)
→ *True/False***

Strong Pseudo-Prime test on base `base`.

`s` and `t` are the numbers such that $n - 1 = 2^{\mathtt{s}}\mathtt{t}$ and `t` is odd.

1.1.3 smallSpsp – strong pseudo-prime test for small number

**smallSpsp(`n`: *integer*, `s`: *integer*=None, `t`: *integer*=None)
→ *True/False***

Strong Pseudo-Prime test for integer `n` less than 10^{12}.

4 spsp tests are sufficient to determine whether an integer less than 10^{12} is prime or not.

s and t are the numbers such that $n - 1 = 2^{\mathtt{s}}\mathtt{t}$ and t is odd.

1.1.4 miller – Miller's primality test

miller(n: *integer*) → *True/False*

Miller's primality test.

This test is valid under GRH. See **config**.

1.1.5 millerRabin – Miller-Rabin primality test

millerRabin(n: *integer*, times: *integer*=20) → *True/False*

Miller's primality test.

The difference from **miller** is that the Miller-Rabin method uses fast but probabilistic algorithm. On the other hand, **miller** employs deterministic algorithm valid under GRH.
times (default to 20) is the number of repetition. The error probability is at most $4^{-\mathtt{t}imes}$.

1.1.6 lpsp – Lucas test

lpsp(n: *integer*, a: *integer*, b: *integer*) → *True/False*

Lucas Pseudo-Prime test.

Return True if n is a Lucas pseudo-prime of parameters a, b, i.e. with respect to $x^2 - \mathtt{a}x + \mathtt{b}$.

1.1.7 fpsp – Frobenius test

fpsp(n: *integer*, a: *integer*, b: *integer*) → *True/False*

Frobenius Pseudo-Prime test.

Return True if n is a Frobenius pseudo-prime of parameters a, b, i.e. with respect to $x^2 - \mathtt{a}x + \mathtt{b}$.

1.1.8 by_primitive_root – Lehmer's test

by_primitive_root(n: ***integer***, divisors: ***sequence***) → ***True/False***

Lehmer's primality test [2].

Return True iff `n` is prime.
The method proves the primality of `n` by existence of a primitive root.
`divisors` is a sequence (list, tuple, etc.) of prime divisors of $n-1$.

1.1.9 full_euler – Brillhart & Selfridge's test

full_euler(n: ***integer***, divisors: ***sequence***) → ***True/False***

Brillhart & Selfridge's primality test [1].

Return True iff `n` is prime.
The method proves the primality of `n` by the equality $\varphi(n) = n-1$, where φ denotes the Euler totient (see **euler**). It requires a sequence of all prime divisors of $n-1$.
`divisors` is a sequence (list, tuple, etc.) of prime divisors of $n-1$.

1.1.10 apr – Jacobi sum test

apr(n: ***integer***) → ***True/False***

APR (Adleman-Pomerance-Rumery) primality test or the Jacobi sum test.

Assuming `n` has no prime factors less than 32. Assuming `n` is spsp (strong pseudo-prime) for several bases.

1.1.11 primeq – primality test automatically

primeq(n: ***integer***) → ***True/False***

A convenient function for primality test.

It uses one of **trialDivision**, **smallSpsp** or **apr** depending on the size of `n`.

1.1.12 prime – n-th prime number

prime(n: ***integer***) → ***integer***

Return the `n`-th prime number.

1.1.13 nextPrime – generate next prime

nextPrime(`n`: *integer*) → *integer*

Return the smallest prime bigger than the given integer `n`.

1.1.14 randPrime – generate random prime

randPrime(`n`: *integer*) → *integer*

Return a random `n`-digits prime.

1.1.15 generator – generate primes

generator((None)) → *generator*

Generate primes from 2 to ∞ (as generator).

1.1.16 generator_eratosthenes – generate primes using Eratosthenes sieve

generator_eratosthenes(`n`: *integer*) → *generator*

Generate primes up to `n` using Eratosthenes sieve.

1.1.17 primonial – product of primes

primonial(`p`: *integer*) → *integer*

Return the product

$$\prod_{q \in \mathbb{P}_{\leq \mathtt{p}}} q = 2 \cdot 3 \cdot 5 \cdots \mathtt{p} \ .$$

1.1.18 properDivisors – proper divisors

properDivisors(`n`: *integer*) → *list*

Return proper divisors of `n` (all divisors of `n` excluding 1 and `n`).

It is only useful for a product of small primes. Use **proper_divisors** in a more general case.
The output is the list of all proper divisors.

1.1.19 primitive_root – primitive root

primitive_root(`p`: *integer*) → *integer*

Return a primitive root of `p`.

`p` must be an odd prime.

1.1.20 Lucas_chain – Lucas sequence

**Lucas_chain(`n`: *integer*, `f`: *function*, `g`: *function*, `x_0`: *integer*, `x_1`: *integer*)
→ (*integer*, *integer*)**

Return the value of (x_n, x_{n+1}) for the sequnce $\{x_i\}$ defined as:

$$\begin{aligned} x_{2i} &= \mathtt{f}(x_i) \\ x_{2i+1} &= \mathtt{g}(x_i, x_{i+1}) \ , \end{aligned}$$

where the initial values `x_0`, `x_1`.

`f` is the function which can be input as 1-ary integer. `g` is the function which can be input as 2-ary integer.

Examples

```
>>> prime.primeq(131)
True
>>> prime.primeq(133)
False
>>> g = prime.generator()
>>> g.next()
2
```

```
>>> g.next()
3
>>> prime.prime(10)
29
>>> prime.nextPrime(100)
101
>>> prime.primitive_root(23)
5
```

Bibliography

[1] J. Brillhart and J. L. Selfridge. Some factorizations of $2^n \pm 1$ and related results. *Math. Comp.*, Vol. 21, pp. 87–96, 1967.

[2] D. H. Lehmer. Tests for primality by the converse of Fermat's theorem. *Bull. Amer. Math. Soc.*, Vol. 33, pp. 327–340, 1927.