Examples

```
>>> from sympy import sieve
>>> sieve._reset() # this line for doctest only
>>> sieve.extend(30)
>>> sieve[10] == 29
True
```

extend_to_no(*i*)

Extend to include the ith prime number.

Parameters

i : integer

Examples

```
>>> from sympy import sieve
>>> sieve._reset() # this line for doctest only
>>> sieve.extend_to_no(9)
>>> sieve._list
array('l', [2, 3, 5, 7, 11, 13, 17, 19, 23])
```

Notes

The list is extended by 50% if it is too short, so it is likely that it will be longer than requested.

mobiusrange(*a*, *b*)

Generate all mobius numbers for the range [a, b).

Parameters

a : integer

First number in range

b : integer

First number outside of range

Examples

```
>>> from sympy import sieve
>>> print([i for i in sieve.mobiusrange(7, 18)])
[-1, 0, 0, 1, -1, 0, -1, 1, 1, 0, -1]
```

primerange(*a*, *b=None*)

Generate all prime numbers in the range [2, a) or [a, b).

Examples

```
>>> from sympy import sieve, prime
```

All primes less than 19:

```
>>> print([i for i in sieve.primerange(19)])
[2, 3, 5, 7, 11, 13, 17]
```

All primes greater than or equal to 7 and less than 19:

```
>>> print([i for i in sieve.primerange(7, 19)])
[7, 11, 13, 17]
```

All primes through the 10th prime

```
>>> list(sieve.primerange(prime(10) + 1))
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

search(*n*)

Return the indices i, j of the primes that bound n.

If n is prime then i == j.

Although n can be an expression, if ceiling cannot convert it to an integer then an n error will be raised.

Examples

```
>>> from sympy import sieve
>>> sieve.search(25)
(9, 10)
>>> sieve.search(23)
(9, 9)
```

totientrange(*a*, *b*)

Generate all totient numbers for the range [a, b).

Examples

```
>>> from sympy import sieve
>>> print([i for i in sieve.totientrange(7, 18)])
[6, 4, 6, 4, 10, 4, 12, 6, 8, 8, 16]
```

Ntheory Functions Reference

sympy.ntheory.generate.**prime**(*nth*)

Return the nth prime, with the primes indexed as prime(1) = 2, prime(2) = 3, etc.... The nth prime is approximately $n \log(n)$.

Logarithmic integral of x is a pretty nice approximation for number of primes $\le x$, i.e. li(x) ~ pi(x) In fact, for the numbers we are concerned about(x<1e11), li(x) - pi(x) < 50000

Also, li(x) > pi(x) can be safely assumed for the numbers which can be evaluated by this function.

Here, we find the least integer m such that li(m) > n using binary search. Now pi(m-1) < li(m-1) <= n,

We find pi(m - 1) using primepi function.

Starting from m, we have to find n - pi(m-1) more primes.

For the inputs this implementation can handle, we will have to test primality for at max about 10**5 numbers, to get our answer.

Examples

```
>>> from sympy import prime
>>> prime(10)
29
>>> prime(1)
2
>>> prime(100000)
1299709
```

See also:

sympy.ntheory.primetest.isprime **(page 1517)**
Test if n is prime

primerange **(page 1481)**
Generate all primes in a given range

primepi **(page 1479)**
Return the number of primes less than or equal to n

References

[R599], [R600], [R601]

sympy.ntheory.generate.**primepi**(*n*)

Represents the prime counting function pi(n) = the number of prime numbers less than or equal to n.

Algorithm Description:

In sieve method, we remove all multiples of prime p except p itself.

Let phi(i,j) be the number of integers 2 <= k <= i which remain after sieving from primes less than or equal to j. Clearly, pi(n) = phi(n, sqrt(n))

If j is not a prime, phi(i,j) = phi(i, j - 1)

if j is a prime, We remove all numbers(except j) whose smallest prime factor is j.

Let $x = j \times a$ be such a number, where $2 \leq a \leq i/j$ Now, after sieving from primes $\leq j - 1$, a must remain (because x, and hence a has no prime factor $\leq j - 1$) Clearly, there are phi(i / j, j - 1) such a which remain on sieving from primes $\leq j - 1$

Now, if a is a prime less than equal to j - 1, $x = j \times a$ has smallest prime factor = a, and has already been removed(by sieving from a). So, we do not need to remove it again. (Note: there will be pi(j - 1) such x)

Thus, number of x, that will be removed are: phi(i / j, j - 1) - phi(j - 1, j - 1) (Note that pi(j - 1) = phi(j - 1, j - 1))

$\Rightarrow$ phi(i,j) = phi(i, j - 1) - phi(i / j, j - 1) + phi(j - 1, j - 1)

So,following recursion is used and implemented as dp:

phi(a, b) = phi(a, b - 1), if b is not a prime phi(a, b) = phi(a, b-1)-phi(a / b, b-1) + phi(b-1, b-1), if b is prime

Clearly a is always of the form floor(n / k), which can take at most $2\sqrt{n}$ values. Two arrays arr1,arr2 are maintained arr1[i] = phi(i, j), arr2[i] = phi(n // i, j)

Finally the answer is arr2[1]

Examples

```
>>> from sympy import primepi, prime, prevprime, isprime
>>> primepi(25)
9
```

So there are 9 primes less than or equal to 25. Is 25 prime?

```
>>> isprime(25)
False
```

It is not. So the first prime less than 25 must be the 9th prime:

```
>>> prevprime(25) == prime(9)
True
```

See also:

sympy.ntheory.primetest.isprime **(page 1517)**
Test if n is prime

primerange **(page 1481)**
Generate all primes in a given range

prime **(page 1479)**
Return the nth prime

sympy.ntheory.generate.**nextprime**(*n*, *ith=1*)

Return the ith prime greater than n.

i must be an integer.

Notes

Potential primes are located at 6*j +/- 1. This property is used during searching.

```
>>> from sympy import nextprime
>>> [(i, nextprime(i)) for i in range(10, 15)]
[(10, 11), (11, 13), (12, 13), (13, 17), (14, 17)]
>>> nextprime(2, ith=2) # the 2nd prime after 2
5
```

See also:

prevprime **(page 1481)**
Return the largest prime smaller than n

primerange **(page 1481)**
Generate all primes in a given range

sympy.ntheory.generate.**prevprime**(*n*)

Return the largest prime smaller than n.

Notes

Potential primes are located at 6*j +/- 1. This property is used during searching.

```
>>> from sympy import prevprime
>>> [(i, prevprime(i)) for i in range(10, 15)]
[(10, 7), (11, 7), (12, 11), (13, 11), (14, 13)]
```

See also:

nextprime **(page 1480)**
Return the ith prime greater than n

primerange **(page 1481)**
Generates all primes in a given range

sympy.ntheory.generate.**primerange**(*a*, *b=None*)

Generate a list of all prime numbers in the range [2, a), or [a, b).

If the range exists in the default sieve, the values will be returned from there; otherwise values will be returned but will not modify the sieve.

Examples

```
>>> from sympy import primerange, prime
```

All primes less than 19:

```
>>> list(primerange(19))
[2, 3, 5, 7, 11, 13, 17]
```

All primes greater than or equal to 7 and less than 19:

```
>>> list(primerange(7, 19))
[7, 11, 13, 17]
```

All primes through the 10th prime

```
>>> list(primerange(prime(10) + 1))
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

The Sieve method, primerange, is generally faster but it will occupy more memory as the sieve stores values. The default instance of Sieve, named sieve, can be used:

```
>>> from sympy import sieve
>>> list(sieve.primerange(1, 30))
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

Notes

Some famous conjectures about the occurrence of primes in a given range are [1]:

- **Twin primes: though often not, the following will give 2 primes**

  an infinite number of times:
  primerange(6*n - 1, 6*n + 2)

- **Legendre's: the following always yields at least one prime**
  primerange(n**2, (n+1)**2+1)

- **Bertrand's (proven): there is always a prime in the range**
  primerange(n, 2*n)

- **Brocard's: there are at least four primes in the range**
  primerange(prime(n)**2, prime(n+1)**2)

The average gap between primes is log(n) [2]; the gap between primes can be arbitrarily large since sequences of composite numbers are arbitrarily large, e.g. the numbers in the sequence n! + 2, n! + 3 … n! + n are all composite.

See also:

prime **(page 1479)**
Return the nth prime

nextprime **(page 1480)**
Return the ith prime greater than n

prevprime **(page 1481)**
Return the largest prime smaller than n

randprime **(page 1483)**
Returns a random prime in a given range

primorial **(page 1483)**
Returns the product of primes based on condition

Sieve.primerange **(page 1477)**
return range from already computed primes or extend the sieve to contain the requested range.

References

[R602], [R603]

sympy.ntheory.generate.**randprime**(*a*, *b*)

Return a random prime number in the range [a, b).

Bertrand's postulate assures that randprime(a, 2*a) will always succeed for a > 1.

Examples

```
>>> from sympy import randprime, isprime
>>> randprime(1, 30)
13
>>> isprime(randprime(1, 30))
True
```

See also:

primerange **(page 1481)**
Generate all primes in a given range

References

[R604]

sympy.ntheory.generate.**primorial**(*n*, *nth=True*)

Returns the product of the first n primes (default) or the primes less than or equal to n (when `nth=False`).

Examples

```
>>> from sympy.ntheory.generate import primorial, primerange
>>> from sympy import factorint, Mul, primefactors, sqrt
>>> primorial(4) # the first 4 primes are 2, 3, 5, 7
210
>>> primorial(4, nth=False) # primes <= 4 are 2 and 3
6
>>> primorial(1)
2
>>> primorial(1, nth=False)
1
>>> primorial(sqrt(101), nth=False)
210
```

One can argue that the primes are infinite since if you take a set of primes and multiply them together (e.g. the primorial) and then add or subtract 1, the result cannot be divided by any of the original factors, hence either 1 or more new primes must divide this product of primes.

In this case, the number itself is a new prime:

```
>>> factorint(primorial(4) + 1)
{211: 1}
```

In this case two new primes are the factors:

```
>>> factorint(primorial(4) - 1)
{11: 1, 19: 1}
```

Here, some primes smaller and larger than the primes multiplied together are obtained:

```
>>> p = list(primerange(10, 20))
>>> sorted(set(primefactors(Mul(*p) + 1)).difference(set(p)))
[2, 5, 31, 149]
```

See also:

***primerange* (page 1481)**
Generate all primes in a given range

sympy.ntheory.generate.**cycle_length**(*f, x0, nmax=None, values=False*)

For a given iterated sequence, return a generator that gives the length of the iterated cycle (lambda) and the length of terms before the cycle begins (mu); if `values` is True then the terms of the sequence will be returned instead. The sequence is started with value `x0`.

Note: more than the first lambda + mu terms may be returned and this is the cost of cycle detection with Brent's method; there are, however, generally less terms calculated than would have been calculated if the proper ending point were determined, e.g. by using Floyd's method.

```
>>> from sympy.ntheory.generate import cycle_length
```

This will yield successive values of i <– func(i):

```
>>> def iter(func, i):
...     while 1:
...         ii = func(i)
...         yield ii
...         i = ii
...
```

A function is defined:

```
>>> func = lambda i: (i**2 + 1) % 51
```

and given a seed of 4 and the mu and lambda terms calculated:

```
>>> next(cycle_length(func, 4))
(6, 2)
```

We can see what is meant by looking at the output:

```
>>> n = cycle_length(func, 4, values=True)
>>> list(ni for ni in n)
[17, 35, 2, 5, 26, 14, 44, 50, 2, 5, 26, 14]
```

There are 6 repeating values after the first 2.

If a sequence is suspected of being longer than you might wish, nmax can be used to exit early (and mu will be returned as None):

```
>>> next(cycle_length(func, 4, nmax = 4))
(4, None)
>>> [ni for ni in cycle_length(func, 4, nmax = 4, values=True)]
[17, 35, 2, 5]
```

Code modified from:
https://en.wikipedia.org/wiki/Cycle_detection.

sympy.ntheory.generate.**composite**(*nth*)

Return the nth composite number, with the composite numbers indexed as composite(1) = 4, composite(2) = 6, etc....

Examples

```
>>> from sympy import composite
>>> composite(36)
52
>>> composite(1)
4
>>> composite(17737)
20000
```

See also:

sympy.ntheory.primetest.isprime **(page 1517)**
Test if n is prime

primerange **(page 1481)**
Generate all primes in a given range

primepi **(page 1479)**
Return the number of primes less than or equal to n

prime **(page 1479)**
Return the nth prime

compositepi **(page 1486)**
Return the number of positive composite numbers less than or equal to n

sympy.ntheory.generate.**compositepi**(*n*)

Return the number of positive composite numbers less than or equal to n. The first positive composite is 4, i.e. compositepi(4) = 1.

Examples

```
>>> from sympy import compositepi
>>> compositepi(25)
15
>>> compositepi(1000)
831
```

See also:

sympy.ntheory.primetest.isprime **(page 1517)**
Test if n is prime

primerange **(page 1481)**
Generate all primes in a given range

prime **(page 1479)**
Return the nth prime

primepi **(page 1479)**
Return the number of primes less than or equal to n

composite **(page 1485)**
Return the nth composite number

sympy.ntheory.factor_.**smoothness**(*n*)

Return the B-smooth and B-power smooth values of n.

The smoothness of n is the largest prime factor of n; the power- smoothness is the largest divisor raised to its multiplicity.

Examples

```
>>> from sympy.ntheory.factor_ import smoothness
>>> smoothness(2**7*3**2)
(3, 128)
>>> smoothness(2**4*13)
(13, 16)
>>> smoothness(2)
(2, 2)
```

See also:

factorint (page 1493), *smoothness_p* (page 1487)

sympy.ntheory.factor_.**smoothness_p**(*n*, *m=-1*, *power=0*, *visual=None*)

Return a list of [m, (p, (M, sm(p + m), psm(p + m)))...] where:

1. p**M is the base-p divisor of n
2. sm(p + m) is the smoothness of p + m (m = -1 by default)
3. psm(p + m) is the power smoothness of p + m

The list is sorted according to smoothness (default) or by power smoothness if power=1.

The smoothness of the numbers to the left (m = -1) or right (m = 1) of a factor govern the results that are obtained from the p +/- 1 type factoring methods.

```
>>> from sympy.ntheory.factor_ import smoothness_p, factorint
>>> smoothness_p(10431, m=1)
(1, [(3, (2, 2, 4)), (19, (1, 5, 5)), (61, (1, 31, 31))])
>>> smoothness_p(10431)
(-1, [(3, (2, 2, 2)), (19, (1, 3, 9)), (61, (1, 5, 5))])
>>> smoothness_p(10431, power=1)
(-1, [(3, (2, 2, 2)), (61, (1, 5, 5)), (19, (1, 3, 9))])
```

If visual=True then an annotated string will be returned:

```
>>> print(smoothness_p(21477639576571, visual=1))
p**i=4410317**1 has p-1 B=1787, B-pow=1787
p**i=4869863**1 has p-1 B=2434931, B-pow=2434931
```

This string can also be generated directly from a factorization dictionary and vice versa:

```
>>> factorint(17*9)
{3: 2, 17: 1}
>>> smoothness_p(_)
'p**i=3**2 has p-1 B=2, B-pow=2\np**i=17**1 has p-1 B=2, B-pow=16'
>>> smoothness_p(_)
{3: 2, 17: 1}
```

The table of the output logic is:

	Visual		
Input	True	False	other
dict	str	tuple	str
str	str	tuple	dict
tuple	str	tuple	str
n	str	tuple	tuple
mul	str	tuple	tuple

See also:

factorint (page 1493), *smoothness* (page 1486)

sympy.ntheory.factor_.**trailing**(*n*)

Count the number of trailing zero digits in the binary representation of n, i.e. determine the largest power of 2 that divides n.

Examples

```
>>> from sympy import trailing
>>> trailing(128)
7
>>> trailing(63)
0
```

sympy.ntheory.factor_.**multiplicity**(*p*, *n*)

Find the greatest integer m such that p**m divides n.

Examples

```
>>> from sympy import multiplicity, Rational
>>> [multiplicity(5, n) for n in [8, 5, 25, 125, 250]]
[0, 1, 2, 3, 3]
>>> multiplicity(3, Rational(1, 9))
-2
```

Note: when checking for the multiplicity of a number in a large factorial it is most efficient to send it as an unevaluated factorial or to call `multiplicity_in_factorial` directly:

```
>>> from sympy.ntheory import multiplicity_in_factorial
>>> from sympy import factorial
>>> p = factorial(25)
>>> n = 2**100
>>> nfac = factorial(n, evaluate=False)
>>> multiplicity(p, nfac)
52818775009509558395695966887
>>> _ == multiplicity_in_factorial(p, n)
True
```

`sympy.ntheory.factor_.`**`perfect_power`**(*n*, *candidates=None*, *big=True*, *factor=True*)

Return `(b, e)` such that `n == b**e` if `n` is a unique perfect power with `e > 1`, else `False` (e.g. 1 is not a perfect power). A ValueError is raised if `n` is not Rational.

By default, the base is recursively decomposed and the exponents collected so the largest possible `e` is sought. If `big=False` then the smallest possible `e` (thus prime) will be chosen.

If `factor=True` then simultaneous factorization of `n` is attempted since finding a factor indicates the only possible root for `n`. This is True by default since only a few small factors will be tested in the course of searching for the perfect power.

The use of `candidates` is primarily for internal use; if provided, False will be returned if `n` cannot be written as a power with one of the candidates as an exponent and factoring (beyond testing for a factor of 2) will not be attempted.

Examples

```
>>> from sympy import perfect_power, Rational
>>> perfect_power(16)
(2, 4)
>>> perfect_power(16, big=False)
(4, 2)
```

Negative numbers can only have odd perfect powers:

```
>>> perfect_power(-4)
False
>>> perfect_power(-8)
(-2, 3)
```

Rationals are also recognized:

```
>>> perfect_power(Rational(1, 2)**3)
(1/2, 3)
>>> perfect_power(Rational(-3, 2)**3)
(-3/2, 3)
```

Notes

To know whether an integer is a perfect power of 2 use

```
>>> is2pow = lambda n: bool(n and not n & (n - 1))
>>> [(i, is2pow(i)) for i in range(5)]
[(0, False), (1, True), (2, True), (3, False), (4, True)]
```

It is not necessary to provide `candidates`. When provided it will be assumed that they are ints. The first one that is larger than the computed maximum possible exponent will signal failure for the routine.

```
>>> perfect_power(3**8, [9])
False
```

(continues on next page)

(continued from previous page)

```
>>> perfect_power(3**8, [2, 4, 8])
(3, 8)
>>> perfect_power(3**8, [4, 8], big=False)
(9, 4)
```

See also:

sympy.core.power.integer_nthroot (page 1008), *sympy.ntheory.primetest.is_square* (page 1514)

sympy.ntheory.factor_.**pollard_rho**(*n*, *s=2*, *a=1*, *retries=5*, *seed=1234*, *max_steps=None*, *F=None*)

Use Pollard's rho method to try to extract a nontrivial factor of n. The returned factor may be a composite number. If no factor is found, None is returned.

The algorithm generates pseudo-random values of x with a generator function, replacing x with F(x). If F is not supplied then the function x**2 + a is used. The first value supplied to F(x) is s. Upon failure (if retries is > 0) a new a and s will be supplied; the a will be ignored if F was supplied.

The sequence of numbers generated by such functions generally have a a lead-up to some number and then loop around back to that number and begin to repeat the sequence, e.g. 1, 2, 3, 4, 5, 3, 4, 5 – this leader and loop look a bit like the Greek letter rho, and thus the name, 'rho'.

For a given function, very different leader-loop values can be obtained so it is a good idea to allow for retries:

```
>>> from sympy.ntheory.generate import cycle_length
>>> n = 16843009
>>> F = lambda x:(2048*pow(x, 2, n) + 32767) % n
>>> for s in range(5):
...     print('loop length = %4i; leader length = %3i' % next(cycle_
↪length(F, s)))
...
loop length = 2489; leader length =  42
loop length =   78; leader length = 120
loop length = 1482; leader length =  99
loop length = 1482; leader length = 285
loop length = 1482; leader length = 100
```

Here is an explicit example where there is a two element leadup to a sequence of 3 numbers (11, 14, 4) that then repeat:

```
>>> x=2
>>> for i in range(9):
...     x=(x**2+12)%17
...     print(x)
...
16
13
11
14
4
```

(continues on next page)

(continued from previous page)

```
11
14
4
11
>>> next(cycle_length(lambda x: (x**2+12)%17, 2))
(3, 2)
>>> list(cycle_length(lambda x: (x**2+12)%17, 2, values=True))
[16, 13, 11, 14, 4]
```

Instead of checking the differences of all generated values for a gcd with n, only the kth and 2*kth numbers are checked, e.g. 1st and 2nd, 2nd and 4th, 3rd and 6th until it has been detected that the loop has been traversed. Loops may be many thousands of steps long before rho finds a factor or reports failure. If `max_steps` is specified, the iteration is cancelled with a failure after the specified number of steps.

Examples

```
>>> from sympy import pollard_rho
>>> n=16843009
>>> F=lambda x:(2048*pow(x,2,n) + 32767) % n
>>> pollard_rho(n, F=F)
257
```

Use the default setting with a bad value of a and no retries:

```
>>> pollard_rho(n, a=n-2, retries=0)
```

If retries is > 0 then perhaps the problem will correct itself when new values are generated for a:

```
>>> pollard_rho(n, a=n-2, retries=1)
257
```

References

[R605]

`sympy.ntheory.factor_.pollard_pm1`(*n*, *B=10*, *a=2*, *retries=0*, *seed=1234*)

Use Pollard's p-1 method to try to extract a nontrivial factor of `n`. Either a divisor (perhaps composite) or `None` is returned.

The value of `a` is the base that is used in the test gcd(a**M - 1, n). The default is 2. If `retries` > 0 then if no factor is found after the first attempt, a new `a` will be generated randomly (using the `seed`) and the process repeated.

Note: the value of M is lcm(1..B) = reduce(ilcm, range(2, B + 1)).

A search is made for factors next to even numbers having a power smoothness less than `B`. Choosing a larger B increases the likelihood of finding a larger factor but takes longer. Whether a factor of n is found or not depends on `a` and the power smoothness of the even number just less than the factor p (hence the name p - 1).

Although some discussion of what constitutes a good a some descriptions are hard to interpret. At the modular.math site referenced below it is stated that if gcd(a**M - 1, n) = N then a**M % q**r is 1 for every prime power divisor of N. But consider the following:

```
>>> from sympy.ntheory.factor_ import smoothness_p, pollard_pm1
>>> n=257*1009
>>> smoothness_p(n)
(-1, [(257, (1, 2, 256)), (1009, (1, 7, 16))])
```

So we should (and can) find a root with B=16:

```
>>> pollard_pm1(n, B=16, a=3)
1009
```

If we attempt to increase B to 256 we find that it does not work:

```
>>> pollard_pm1(n, B=256)
>>>
```

But if the value of a is changed we find that only multiples of 257 work, e.g.:

```
>>> pollard_pm1(n, B=256, a=257)
1009
```

Checking different a values shows that all the ones that did not work had a gcd value not equal to n but equal to one of the factors:

```
>>> from sympy import ilcm, igcd, factorint, Pow
>>> M = 1
>>> for i in range(2, 256):
...     M = ilcm(M, i)
...
>>> set([igcd(pow(a, M, n) - 1, n) for a in range(2, 256) if
...      igcd(pow(a, M, n) - 1, n) != n])
{1009}
```

But does aM % d for every divisor of n give 1?

```
>>> aM = pow(255, M, n)
>>> [(d, aM%Pow(*d.args)) for d in factorint(n, visual=True).args]
[(257**1, 1), (1009**1, 1)]
```

No, only one of them. So perhaps the principle is that a root will be found for a given value of B provided that:

1) the power smoothness of the p - 1 value next to the root does not exceed B

2) a**M % p != 1 for any of the divisors of n.

By trying more than one a it is possible that one of them will yield a factor.

Examples

With the default smoothness bound, this number cannot be cracked:

```
>>> from sympy.ntheory import pollard_pm1
>>> pollard_pm1(21477639576571)
```

Increasing the smoothness bound helps:

```
>>> pollard_pm1(21477639576571, B=2000)
4410317
```

Looking at the smoothness of the factors of this number we find:

```
>>> from sympy.ntheory.factor_ import smoothness_p, factorint
>>> print(smoothness_p(21477639576571, visual=1))
p**i=4410317**1 has p-1 B=1787, B-pow=1787
p**i=4869863**1 has p-1 B=2434931, B-pow=2434931
```

The B and B-pow are the same for the p - 1 factorizations of the divisors because those factorizations had a very large prime factor:

```
>>> factorint(4410317 - 1)
{2: 2, 617: 1, 1787: 1}
>>> factorint(4869863-1)
{2: 1, 2434931: 1}
```

Note that until B reaches the B-pow value of 1787, the number is not cracked;

```
>>> pollard_pm1(21477639576571, B=1786)
>>> pollard_pm1(21477639576571, B=1787)
4410317
```

The B value has to do with the factors of the number next to the divisor, not the divisors themselves. A worst case scenario is that the number next to the factor p has a large prime divisisor or is a perfect power. If these conditions apply then the power-smoothness will be about p/2 or p. The more realistic is that there will be a large prime factor next to p requiring a B value on the order of p/2. Although primes may have been searched for up to this level, the p/2 is a factor of p - 1, something that we do not know. The modular.math reference below states that 15% of numbers in the range of 10**15 to 15**15 + 10**4 are 10**6 power smooth so a B of 10**6 will fail 85% of the time in that range. From 10**8 to 10**8 + 10**3 the percentages are nearly reversed...but in that range the simple trial division is quite fast.

References

[R606], [R607], [R608]

`sympy.ntheory.factor_.`**`factorint`**(*n*, *limit=None*, *use_trial=True*, *use_rho=True*, *use_pm1=True*, *use_ecm=True*, *verbose=False*, *visual=None*, *multiple=False*)

Given a positive integer `n`, `factorint(n)` returns a dict containing the prime factors of `n` as keys and their respective multiplicities as values. For example:

```
>>> from sympy.ntheory import factorint
>>> factorint(2000)    # 2000 = (2**4) * (5**3)
{2: 4, 5: 3}
>>> factorint(65537)   # This number is prime
{65537: 1}
```

For input less than 2, factorint behaves as follows:

- `factorint(1)` returns the empty factorization, `{}`
- `factorint(0)` returns `{0:1}`
- `factorint(-n)` adds `-1:1` to the factors and then factors `n`

Partial Factorization:

If `limit` (> 3) is specified, the search is stopped after performing trial division up to (and including) the limit (or taking a corresponding number of rho/p-1 steps). This is useful if one has a large number and only is interested in finding small factors (if any). Note that setting a limit does not prevent larger factors from being found early; it simply means that the largest factor may be composite. Since checking for perfect power is relatively cheap, it is done regardless of the limit setting.

This number, for example, has two small factors and a huge semi-prime factor that cannot be reduced easily:

```
>>> from sympy.ntheory import isprime
>>> a = 1407633717262338957430697921446883
>>> f = factorint(a, limit=10000)
>>> f == {991: 1, int(202916782076162456022877024859): 1, 7: 1}
True
>>> isprime(max(f))
False
```

This number has a small factor and a residual perfect power whose base is greater than the limit:

```
>>> factorint(3*101**7, limit=5)
{3: 1, 101: 7}
```

List of Factors:

If `multiple` is set to `True` then a list containing the prime factors including multiplicities is returned.

```
>>> factorint(24, multiple=True)
[2, 2, 2, 3]
```

Visual Factorization:

If `visual` is set to `True`, then it will return a visual factorization of the integer. For example:

```
>>> from sympy import pprint
>>> pprint(factorint(4200, visual=True))
 3  1  2  1
2 *3 *5 *7
```

Note that this is achieved by using the evaluate=False flag in Mul and Pow. If you do other manipulations with an expression where evaluate=False, it may evaluate. Therefore, you should use the visual option only for visualization, and use the normal dictionary returned by visual=False if you want to perform operations on the factors.

You can easily switch between the two forms by sending them back to factorint:

```
>>> from sympy import Mul
>>> regular = factorint(1764); regular
{2: 2, 3: 2, 7: 2}
>>> pprint(factorint(regular))
 2  2  2
2 *3 *7
```

```
>>> visual = factorint(1764, visual=True); pprint(visual)
 2  2  2
2 *3 *7
>>> print(factorint(visual))
{2: 2, 3: 2, 7: 2}
```

If you want to send a number to be factored in a partially factored form you can do so with a dictionary or unevaluated expression:

```
>>> factorint(factorint({4: 2, 12: 3})) # twice to toggle to dict form
{2: 10, 3: 3}
>>> factorint(Mul(4, 12, evaluate=False))
{2: 4, 3: 1}
```

The table of the output logic is:

Input	True	False	other
dict	mul	dict	mul
n	mul	dict	dict
mul	mul	dict	dict

Notes

Algorithm:

The function switches between multiple algorithms. Trial division quickly finds small factors (of the order 1-5 digits), and finds all large factors if given enough time. The Pollard rho and p-1 algorithms are used to find large factors ahead of time; they will often find factors of the order of 10 digits within a few seconds:

```
>>> factors = factorint(12345678910111213141516)
>>> for base, exp in sorted(factors.items()):
...     print('%s %s' % (base, exp))
...
2 2
2507191691 1
1231026625769 1
```

Any of these methods can optionally be disabled with the following boolean parameters:

- `use_trial`: Toggle use of trial division
- `use_rho`: Toggle use of Pollard's rho method
- `use_pm1`: Toggle use of Pollard's p-1 method

`factorint` also periodically checks if the remaining part is a prime number or a perfect power, and in those cases stops.

For unevaluated factorial, it uses Legendre's formula(theorem).

If `verbose` is set to `True`, detailed progress is printed.

See also:

smoothness (page 1486), *smoothness_p* (page 1487), *divisors* (page 1497)

`sympy.ntheory.factor_.`**`factorrat`**(*rat*, *limit=None*, *use_trial=True*, *use_rho=True*, *use_pm1=True*, *verbose=False*, *visual=None*, *multiple=False*)

Given a Rational `r`, `factorrat(r)` returns a dict containing the prime factors of `r` as keys and their respective multiplicities as values. For example:

```
>>> from sympy import factorrat, S
>>> factorrat(S(8)/9)    # 8/9 = (2**3) * (3**-2)
{2: 3, 3: -2}
>>> factorrat(S(-1)/987)    # -1/789 = -1 * (3**-1) * (7**-1) * (47**-1)
{-1: 1, 3: -1, 7: -1, 47: -1}
```

Please see the docstring for `factorint` for detailed explanations and examples of the following keywords:

- `limit`: Integer limit up to which trial division is done
- `use_trial`: Toggle use of trial division
- `use_rho`: Toggle use of Pollard's rho method
- `use_pm1`: Toggle use of Pollard's p-1 method
- `verbose`: Toggle detailed printing of progress
- `multiple`: Toggle returning a list of factors or dict
- `visual`: Toggle product form of output

`sympy.ntheory.factor_.`**`primefactors`**(*n*, *limit=None*, *verbose=False*)

Return a sorted list of n's prime factors, ignoring multiplicity and any composite factor that remains if the limit was set too low for complete factorization. Unlike factorint(), primefactors() does not return -1 or 0.

Examples

```
>>> from sympy.ntheory import primefactors, factorint, isprime
>>> primefactors(6)
[2, 3]
>>> primefactors(-5)
[5]
```

```
>>> sorted(factorint(123456).items())
[(2, 6), (3, 1), (643, 1)]
>>> primefactors(123456)
[2, 3, 643]
```

```
>>> sorted(factorint(10000000001, limit=200).items())
[(101, 1), (99009901, 1)]
>>> isprime(99009901)
False
>>> primefactors(10000000001, limit=300)
[101]
```

See also:

divisors (page 1497)

sympy.ntheory.factor_.**divisors**(*n*, *generator=False*, *proper=False*)

Return all divisors of n sorted from 1..n by default. If generator is `True` an unordered generator is returned.

The number of divisors of n can be quite large if there are many prime factors (counting repeated factors). If only the number of factors is desired use divisor_count(n).

Examples

```
>>> from sympy import divisors, divisor_count
>>> divisors(24)
[1, 2, 3, 4, 6, 8, 12, 24]
>>> divisor_count(24)
8
```

```
>>> list(divisors(120, generator=True))
[1, 2, 4, 8, 3, 6, 12, 24, 5, 10, 20, 40, 15, 30, 60, 120]
```

Notes

This is a slightly modified version of Tim Peters referenced at: https://stackoverflow.com/questions/1010381/python-factorization

See also:

primefactors (page 1496), *factorint* (page 1493), *divisor_count* (page 1498)

sympy.ntheory.factor_.**proper_divisors**(*n*, *generator=False*)

Return all divisors of n except n, sorted by default. If generator is True an unordered generator is returned.

Examples

```
>>> from sympy import proper_divisors, proper_divisor_count
>>> proper_divisors(24)
[1, 2, 3, 4, 6, 8, 12]
>>> proper_divisor_count(24)
7
>>> list(proper_divisors(120, generator=True))
[1, 2, 4, 8, 3, 6, 12, 24, 5, 10, 20, 40, 15, 30, 60]
```

See also:

factorint (page 1493), *divisors* (page 1497), *proper_divisor_count* (page 1498)

sympy.ntheory.factor_.**divisor_count**(*n*, *modulus=1*, *proper=False*)

Return the number of divisors of n. If modulus is not 1 then only those that are divisible by modulus are counted. If proper is True then the divisor of n will not be counted.

Examples

```
>>> from sympy import divisor_count
>>> divisor_count(6)
4
>>> divisor_count(6, 2)
2
>>> divisor_count(6, proper=True)
3
```

See also:

factorint (page 1493), *divisors* (page 1497), *totient* (page 1501), *proper_divisor_count* (page 1498)

sympy.ntheory.factor_.**proper_divisor_count**(*n*, *modulus=1*)

Return the number of proper divisors of n.

Examples

```
>>> from sympy import proper_divisor_count
>>> proper_divisor_count(6)
3
>>> proper_divisor_count(6, modulus=2)
1
```

See also:

divisors (page 1497), *proper_divisors* (page 1498), *divisor_count* (page 1498)

`sympy.ntheory.factor_.`**`udivisors`**`(`*n*, *generator=False*`)`

Return all unitary divisors of n sorted from 1..n by default. If generator is `True` an unordered generator is returned.

The number of unitary divisors of n can be quite large if there are many prime factors. If only the number of unitary divisors is desired use udivisor_count(n).

Examples

```
>>> from sympy.ntheory.factor_ import udivisors, udivisor_count
>>> udivisors(15)
[1, 3, 5, 15]
>>> udivisor_count(15)
4
```

```
>>> sorted(udivisors(120, generator=True))
[1, 3, 5, 8, 15, 24, 40, 120]
```

See also:

primefactors (page 1496), *factorint* (page 1493), *divisors* (page 1497), *divisor_count* (page 1498), *udivisor_count* (page 1499)

References

[R609], [R610]

`sympy.ntheory.factor_.`**`udivisor_count`**`(`*n*`)`

Return the number of unitary divisors of `n`.

Parameters

n : integer

Examples

```
>>> from sympy.ntheory.factor_ import udivisor_count
>>> udivisor_count(120)
8
```

See also:

factorint (page 1493), *divisors* (page 1497), *udivisors* (page 1499), *divisor_count* (page 1498), *totient* (page 1501)

References

[R611]

`sympy.ntheory.factor_.`**`antidivisors`**`(`*n*, *generator=False*`)`

Return all antidivisors of n sorted from 1..n by default.

Antidivisors [R612] of n are numbers that do not divide n by the largest possible margin. If generator is True an unordered generator is returned.

Examples

```
>>> from sympy.ntheory.factor_ import antidivisors
>>> antidivisors(24)
[7, 16]
```

```
>>> sorted(antidivisors(128, generator=True))
[3, 5, 15, 17, 51, 85]
```

See also:

primefactors (page 1496), *factorint* (page 1493), *divisors* (page 1497), *divisor_count* (page 1498), *antidivisor_count* (page 1500)

References

[R612]

`sympy.ntheory.factor_.`**`antidivisor_count`**`(`*n*`)`

Return the number of antidivisors [R613] of n.

Parameters

n : integer

Examples

```
>>> from sympy.ntheory.factor_ import antidivisor_count
>>> antidivisor_count(13)
4
>>> antidivisor_count(27)
5
```

See also:

factorint (page 1493), *divisors* (page 1497), *antidivisors* (page 1500), *divisor_count* (page 1498), *totient* (page 1501)

References

[R613]

class sympy.ntheory.factor_.**totient**(*n*)

Calculate the Euler totient function phi(n)

`totient(n)` or $\phi(n)$ is the number of positive integers $\leq$ n that are relatively prime to n.

Parameters

n : integer

Examples

```
>>> from sympy.ntheory import totient
>>> totient(1)
1
>>> totient(25)
20
>>> totient(45) == totient(5)*totient(9)
True
```

See also:

divisor_count (page 1498)

References

[R614], [R615]

class sympy.ntheory.factor_.**reduced_totient**(*n*)

Calculate the Carmichael reduced totient function lambda(n)

`reduced_totient(n)` or $\lambda(n)$ is the smallest m > 0 such that $k^m \equiv 1 \mod n$ for all k relatively prime to n.

Examples

```
>>> from sympy.ntheory import reduced_totient
>>> reduced_totient(1)
1
>>> reduced_totient(8)
2
>>> reduced_totient(30)
4
```

See also:

totient (page 1501)

References

[R616], [R617]

class sympy.ntheory.factor_.**divisor_sigma**(*n*, *k=1*)

Calculate the divisor function $\sigma_k(n)$ for positive integer n

`divisor_sigma(n, k)` is equal to `sum([x**k for x in divisors(n)])`

If n's prime factorization is:

$$n = \prod_{i=1}^{\omega} p_i^{m_i},$$

then

$$\sigma_k(n) = \prod_{i=1}^{\omega}(1 + p_i^k + p_i^{2k} + \cdots + p_i^{m_i k}).$$

Parameters

n : integer

k : integer, optional

power of divisors in the sum

for k = 0, 1: `divisor_sigma(n, 0)` is equal to `divisor_count(n)` `divisor_sigma(n, 1)` is equal to `sum(divisors(n))`

Default for k is 1.

Examples

```
>>> from sympy.ntheory import divisor_sigma
>>> divisor_sigma(18, 0)
6
>>> divisor_sigma(39, 1)
56
>>> divisor_sigma(12, 2)
210
>>> divisor_sigma(37)
38
```

See also:

divisor_count (page 1498), *totient* (page 1501), *divisors* (page 1497), *factorint* (page 1493)

References

[R618]

class sympy.ntheory.factor_.**udivisor_sigma**(*n*, *k=1*)

Calculate the unitary divisor function $\sigma_k^*(n)$ for positive integer n

`udivisor_sigma(n, k)` is equal to `sum([x**k for x in udivisors(n)])`

If n's prime factorization is:

$$n = \prod_{i=1}^\omega p_i^{m_i},$$

then

$$\sigma_k^*(n) = \prod_{i=1}^\omega (1+p_i^{m_ik}).$$

Parameters

k : power of divisors in the sum

for k = 0, 1: `udivisor_sigma(n, 0)` is equal to `udivisor_count(n)` `udivisor_sigma(n, 1)` is equal to `sum(divisors(n))`

Default for k is 1.

Examples

```
>>> from sympy.ntheory.factor_ import udivisor_sigma
>>> udivisor_sigma(18, 0)
4
>>> udivisor_sigma(74, 1)
114
>>> udivisor_sigma(36, 3)
47450
>>> udivisor_sigma(111)
152
```

See also:

divisor_count (page 1498), *totient* (page 1501), *divisors* (page 1497), *udivisors* (page 1499), *udivisor_count* (page 1499), *divisor_sigma* (page 1502), *factorint* (page 1493)

References

[R619]

`sympy.ntheory.factor_.core(n, t=2)`

Calculate core(n, t) = $core_t(n)$ of a positive integer n

`core_2(n)` is equal to the squarefree part of n

If n's prime factorization is:

$$n = \prod_{i=1}^{\omega} p_i^{m_i},$$

then

$$core_t(n) = \prod_{i=1}^{\omega} p_i^{m_i \mod t}.$$

Parameters

n : integer

t : integer

core(n, t) calculates the t-th power free part of n

`core(n, 2)` is the squarefree part of n `core(n, 3)` is the cubefree part of n

Default for t is 2.

Examples

```
>>> from sympy.ntheory.factor_ import core
>>> core(24, 2)
6
>>> core(9424, 3)
1178
>>> core(379238)
379238
>>> core(15**11, 10)
15
```

See also:

factorint (page 1493), *sympy.solvers.diophantine.diophantine.square_factor* (page 730)

References

[R620]

`sympy.ntheory.factor_.digits(n, b=10, digits=None)`

Return a list of the digits of `n` in base `b`. The first element in the list is `b` (or `-b` if `n` is negative).

Parameters

n: integer

The number whose digits are returned.

b: integer

The base in which digits are computed.

digits: integer (or None for all digits)

The number of digits to be returned (padded with zeros, if necessary).

Examples

```
>>> from sympy.ntheory.digits import digits
>>> digits(35)
[10, 3, 5]
```

If the number is negative, the negative sign will be placed on the base (which is the first element in the returned list):

```
>>> digits(-35)
[-10, 3, 5]
```

Bases other than 10 (and greater than 1) can be selected with `b`:

```
>>> digits(27, b=2)
[2, 1, 1, 0, 1, 1]
```

Use the `digits` keyword if a certain number of digits is desired:

```
>>> digits(35, digits=4)
[10, 0, 0, 3, 5]
```

`class sympy.ntheory.factor_.primenu(n)`

Calculate the number of distinct prime factors for a positive integer n.

If n's prime factorization is:

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

then `primenu(n)` or $\nu(n)$ is:

$$\nu(n) = k.$$

Examples

```
>>> from sympy.ntheory.factor_ import primenu
>>> primenu(1)
0
>>> primenu(30)
3
```

See also:

factorint (page 1493)

References

[R621]

class sympy.ntheory.factor_.**primeomega**(*n*)

Calculate the number of prime factors counting multiplicities for a positive integer n.

If n's prime factorization is:

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

then primeomega(n) or $\Omega(n)$ is:

$$\Omega(n) = \sum_{i=1}^{k} m_i.$$

Examples

```
>>> from sympy.ntheory.factor_ import primeomega
>>> primeomega(1)
0
>>> primeomega(20)
3
```

See also:

factorint (page 1493)

References

[R622]

sympy.ntheory.factor_.**mersenne_prime_exponent**(*nth*)

Returns the exponent i for the nth Mersenne prime (which has the form $2^i - 1$).

Examples

```
>>> from sympy.ntheory.factor_ import mersenne_prime_exponent
>>> mersenne_prime_exponent(1)
2
>>> mersenne_prime_exponent(20)
4423
```

`sympy.ntheory.factor_.is_perfect(n)`

Returns True if n is a perfect number, else False.

A perfect number is equal to the sum of its positive, proper divisors.

Examples

```
>>> from sympy.ntheory.factor_ import is_perfect, divisors, divisor_sigma
>>> is_perfect(20)
False
>>> is_perfect(6)
True
>>> 6 == divisor_sigma(6) - 6 == sum(divisors(6)[:-1])
True
```

References

[R623], [R624]

`sympy.ntheory.factor_.is_mersenne_prime(n)`

Returns True if n is a Mersenne prime, else False.

A Mersenne prime is a prime number having the form $2^i - 1$.

Examples

```
>>> from sympy.ntheory.factor_ import is_mersenne_prime
>>> is_mersenne_prime(6)
False
>>> is_mersenne_prime(127)
True
```

References

[R625]

sympy.ntheory.factor_.**abundance**(*n*)

Returns the difference between the sum of the positive proper divisors of a number and the number.

Examples

```
>>> from sympy.ntheory import abundance, is_perfect, is_abundant
>>> abundance(6)
0
>>> is_perfect(6)
True
>>> abundance(10)
-2
>>> is_abundant(10)
False
```

sympy.ntheory.factor_.**is_abundant**(*n*)

Returns True if n is an abundant number, else False.

A abundant number is smaller than the sum of its positive proper divisors.

Examples

```
>>> from sympy.ntheory.factor_ import is_abundant
>>> is_abundant(20)
True
>>> is_abundant(15)
False
```

References

[R626]

sympy.ntheory.factor_.**is_deficient**(*n*)

Returns True if n is a deficient number, else False.

A deficient number is greater than the sum of its positive proper divisors.

Examples

```
>>> from sympy.ntheory.factor_ import is_deficient
>>> is_deficient(20)
False
>>> is_deficient(15)
True
```

References

[R627]

sympy.ntheory.factor_.**is_amicable**(*m*, *n*)

Returns True if the numbers m and n are "amicable", else False.

Amicable numbers are two different numbers so related that the sum of the proper divisors of each is equal to that of the other.

Examples

```
>>> from sympy.ntheory.factor_ import is_amicable, divisor_sigma
>>> is_amicable(220, 284)
True
>>> divisor_sigma(220) == divisor_sigma(284)
True
```

References

[R628]

sympy.ntheory.modular.**symmetric_residue**(*a*, *m*)

Return the residual mod m such that it is within half of the modulus.

```
>>> from sympy.ntheory.modular import symmetric_residue
>>> symmetric_residue(1, 6)
1
>>> symmetric_residue(4, 6)
-2
```

sympy.ntheory.modular.**crt**(*m*, *v*, *symmetric=False*, *check=True*)

Chinese Remainder Theorem.

The moduli in m are assumed to be pairwise coprime. The output is then an integer f, such that f = v_i mod m_i for each pair out of v and m. If `symmetric` is False a positive integer will be returned, else |f| will be less than or equal to the LCM of the moduli, and thus f may be negative.

If the moduli are not co-prime the correct result will be returned if/when the test of the result is found to be incorrect. This result will be None if there is no solution.

The keyword `check` can be set to False if it is known that the moduli are coprime.

Examples

As an example consider a set of residues `U = [49, 76, 65]` and a set of moduli `M = [99, 97, 95]`. Then we have:

```
>>> from sympy.ntheory.modular import crt

>>> crt([99, 97, 95], [49, 76, 65])
(639985, 912285)
```

This is the correct result because:

```
>>> [639985 % m for m in [99, 97, 95]]
[49, 76, 65]
```

If the moduli are not co-prime, you may receive an incorrect result if you use `check=False`:

```
>>> crt([12, 6, 17], [3, 4, 2], check=False)
(954, 1224)
>>> [954 % m for m in [12, 6, 17]]
[6, 0, 2]
>>> crt([12, 6, 17], [3, 4, 2]) is None
True
>>> crt([3, 6], [2, 5])
(5, 6)
```

Note: the order of gf_crt's arguments is reversed relative to crt, and that solve_congruence takes residue, modulus pairs.

Programmer's note: rather than checking that all pairs of moduli share no GCD (an O(n**2) test) and rather than factoring all moduli and seeing that there is no factor in common, a check that the result gives the indicated residuals is performed – an O(n) operation.

See also:

solve_congruence (page 1511)

***sympy.polys.galoistools.gf_crt* (page 2602)**
low level crt routine used by this routine

`sympy.ntheory.modular.`**`crt1`**`(`*m*`)`

First part of Chinese Remainder Theorem, for multiple application.

Examples

```
>>> from sympy.ntheory.modular import crt1
>>> crt1([18, 42, 6])
(4536, [252, 108, 756], [0, 2, 0])
```

`sympy.ntheory.modular.`**`crt2`**`(`*m, v, mm, e, s, symmetric=False*`)`

Second part of Chinese Remainder Theorem, for multiple application.

Examples

```
>>> from sympy.ntheory.modular import crt1, crt2
>>> mm, e, s = crt1([18, 42, 6])
>>> crt2([18, 42, 6], [0, 0, 0], mm, e, s)
(0, 4536)
```

sympy.ntheory.modular.**solve_congruence**(**remainder_modulus_pairs*, ***hint*)

Compute the integer `n` that has the residual `ai` when it is divided by `mi` where the `ai` and `mi` are given as pairs to this function: ((a1, m1), (a2, m2), …). If there is no solution, return None. Otherwise return `n` and its modulus.

The `mi` values need not be co-prime. If it is known that the moduli are not co-prime then the hint `check` can be set to False (default=True) and the check for a quicker solution via crt() (valid when the moduli are co-prime) will be skipped.

If the hint `symmetric` is True (default is False), the value of `n` will be within 1/2 of the modulus, possibly negative.

Examples

```
>>> from sympy.ntheory.modular import solve_congruence
```

What number is 2 mod 3, 3 mod 5 and 2 mod 7?

```
>>> solve_congruence((2, 3), (3, 5), (2, 7))
(23, 105)
>>> [23 % m for m in [3, 5, 7]]
[2, 3, 2]
```

If you prefer to work with all remainder in one list and all moduli in another, send the arguments like this:

```
>>> solve_congruence(*zip((2, 3, 2), (3, 5, 7)))
(23, 105)
```

The moduli need not be co-prime; in this case there may or may not be a solution:

```
>>> solve_congruence((2, 3), (4, 6)) is None
True
```

```
>>> solve_congruence((2, 3), (5, 6))
(5, 6)
```

The symmetric flag will make the result be within 1/2 of the modulus:

```
>>> solve_congruence((2, 3), (5, 6), symmetric=True)
(-1, 6)
```

See also:

`crt` **(page 1509)**
high level routine implementing the Chinese Remainder Theorem

`sympy.ntheory.multinomial.`**`binomial_coefficients`**`(`*n*`)`

Return a dictionary containing pairs $(k1, k2) : C_k n$ where $C_k n$ are binomial coefficients and $n = k1 + k2$.

Examples

```
>>> from sympy.ntheory import binomial_coefficients
>>> binomial_coefficients(9)
{(0, 9): 1, (1, 8): 9, (2, 7): 36, (3, 6): 84,
 (4, 5): 126, (5, 4): 126, (6, 3): 84, (7, 2): 36, (8, 1): 9, (9, 0): 1}
```

See also:

binomial_coefficients_list (page 1512), *multinomial_coefficients* (page 1512)

`sympy.ntheory.multinomial.`**`binomial_coefficients_list`**`(`*n*`)`

Return a list of binomial coefficients as rows of the Pascal's triangle.

Examples

```
>>> from sympy.ntheory import binomial_coefficients_list
>>> binomial_coefficients_list(9)
[1, 9, 36, 84, 126, 126, 84, 36, 9, 1]
```

See also:

binomial_coefficients (page 1511), *multinomial_coefficients* (page 1512)

`sympy.ntheory.multinomial.`**`multinomial_coefficients`**`(`*m, n*`)`

Return a dictionary containing pairs `{(k1,k2,..,km) : C_kn}` where `C_kn` are multinomial coefficients such that `n=k1+k2+..+km`.

Examples

```
>>> from sympy.ntheory import multinomial_coefficients
>>> multinomial_coefficients(2, 5) # indirect doctest
{(0, 5): 1, (1, 4): 5, (2, 3): 10, (3, 2): 10, (4, 1): 5, (5, 0): 1}
```

Notes

The algorithm is based on the following result:

$$\binom{n}{k_1, \ldots, k_m} = \frac{k_1 + 1}{n - k_1} \sum_{i=2}^{m} \binom{n}{k_1 + 1, \ldots, k_i - 1, \ldots}$$

Code contributed to Sage by Yann Laigle-Chapuy, copied with permission of the author.

See also:

binomial_coefficients_list (page 1512), *binomial_coefficients* (page 1511) sympy.ntheory.multinomial.**multinomial_coefficients_iterator**(*m*, *n*, *_tuple=<class 'tuple'>*)

multinomial coefficient iterator

This routine has been optimized for m large with respect to n by taking advantage of the fact that when the monomial tuples t are stripped of zeros, their coefficient is the same as that of the monomial tuples from `multinomial_coefficients(n, n)`. Therefore, the latter coefficients are precomputed to save memory and time.

```
>>> from sympy.ntheory.multinomial import multinomial_coefficients
>>> m53, m33 = multinomial_coefficients(5,3), multinomial_coefficients(3,
↪3)
>>> m53[(0,0,0,1,2)] == m53[(0,0,1,0,2)] == m53[(1,0,2,0,0)] == m33[(0,1,
↪2)]
True
```

Examples

```
>>> from sympy.ntheory.multinomial import multinomial_coefficients_
↪iterator
>>> it = multinomial_coefficients_iterator(20,3)
>>> next(it)
((3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0), 1)
```

sympy.ntheory.partitions_.**npartitions**(*n*, *verbose=False*)

Calculate the partition function P(n), i.e. the number of ways that n can be written as a sum of positive integers.

P(n) is computed using the Hardy-Ramanujan-Rademacher formula [R629].

The correctness of this implementation has been tested through 10^{10}.

Examples

```
>>> from sympy.ntheory import npartitions
>>> npartitions(25)
1958
```

References

[R629]

sympy.ntheory.primetest.**is_euler_pseudoprime**(*n*, *b*)

Returns True if n is prime or an Euler pseudoprime to base b, else False.

Euler Pseudoprime : In arithmetic, an odd composite integer n is called an euler pseudoprime to base a, if a and n are coprime and satisfy the modular arithmetic congruence relation :

a ^ (n-1)/2 = + 1(mod n) or a ^ (n-1)/2 = - 1(mod n)

(where mod refers to the modulo operation).

Examples

```
>>> from sympy.ntheory.primetest import is_euler_pseudoprime
>>> is_euler_pseudoprime(2, 5)
True
```

References

[R630]

`sympy.ntheory.primetest.`**`is_square`**`(`*n*, *prep=True*`)`

Return True if n == a * a for some integer a, else False. If n is suspected of *not* being a square then this is a quick method of confirming that it is not.

Examples

```
>>> from sympy.ntheory.primetest import is_square
>>> is_square(25)
True
>>> is_square(2)
False
```

See also:

sympy.core.power.integer_nthroot (page 1008)

References

[R631]

`sympy.ntheory.primetest.`**`mr`**`(`*n*, *bases*`)`

Perform a Miller-Rabin strong pseudoprime test on n using a given list of bases/witnesses.

Examples

```
>>> from sympy.ntheory.primetest import mr
>>> mr(1373651, [2, 3])
False
>>> mr(479001599, [31, 73])
True
```

References

A list of thresholds and the bases they require are here: https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test#Deterministic_variants

[R632]

sympy.ntheory.primetest.**is_lucas_prp**(*n*)

Standard Lucas compositeness test with Selfridge parameters. Returns False if n is definitely composite, and True if n is a Lucas probable prime.

This is typically used in combination with the Miller-Rabin test.

Examples

```
>>> from sympy.ntheory.primetest import isprime, is_lucas_prp
>>> for i in range(10000):
...     if is_lucas_prp(i) and not isprime(i):
...        print(i)
323
377
1159
1829
3827
5459
5777
9071
9179
```

References

- "Lucas Pseudoprimes", Baillie and Wagstaff, 1980. http://mpqs.free.fr/LucasPseudoprimes.pdf
- OEIS A217120: Lucas Pseudoprimes https://oeis.org/A217120
- https://en.wikipedia.org/wiki/Lucas_pseudoprime

sympy.ntheory.primetest.**is_strong_lucas_prp**(*n*)

Strong Lucas compositeness test with Selfridge parameters. Returns False if n is definitely composite, and True if n is a strong Lucas probable prime.

This is often used in combination with the Miller-Rabin test, and in particular, when combined with M-R base 2 creates the strong BPSW test.

Examples

```
>>> from sympy.ntheory.primetest import isprime, is_strong_lucas_prp
>>> for i in range(20000):
...     if is_strong_lucas_prp(i) and not isprime(i):
...        print(i)
5459
5777
10877
16109
18971
```

References

- "Lucas Pseudoprimes", Baillie and Wagstaff, 1980. http://mpqs.free.fr/LucasPseudoprimes.pdf
- OEIS A217255: Strong Lucas Pseudoprimes https://oeis.org/A217255
- https://en.wikipedia.org/wiki/Lucas_pseudoprime
- https://en.wikipedia.org/wiki/Baillie-PSW_primality_test

sympy.ntheory.primetest.**is_extra_strong_lucas_prp**(*n*)

Extra Strong Lucas compositeness test. Returns False if n is definitely composite, and True if n is a "extra strong" Lucas probable prime.

The parameters are selected using P = 3, Q = 1, then incrementing P until (D|n) == -1. The test itself is as defined in Grantham 2000, from the Mo and Jones preprint. The parameter selection and test are the same as used in OEIS A217719, Perl's Math::Prime::Util, and the Lucas pseudoprime page on Wikipedia.

With these parameters, there are no counterexamples below 2^64 nor any known above that range. It is 20-50% faster than the strong test.

Because of the different parameters selected, there is no relationship between the strong Lucas pseudoprimes and extra strong Lucas pseudoprimes. In particular, one is not a subset of the other.

Examples

```
>>> from sympy.ntheory.primetest import isprime, is_extra_strong_lucas_
↪prp
>>> for i in range(20000):
...     if is_extra_strong_lucas_prp(i) and not isprime(i):
...        print(i)
989
3239
5777
10877
```