Package 'numbers'

November 26, 2019

Type Package

Title Number-Theoretic Functions

Version 0.7-5

Date 2019-11-26

Author Hans Werner Borchers

Maintainer Hans W. Borchers <hwborchers@googlemail.com>

Depends R (>= 3.1.0)

Suggests gmp (>= 0.5-1)

Description Provides number-theoretic functions for factorization, prime numbers, twin primes, primitive roots, modular logarithm and inverses, extended GCD, Farey series and continuous fractions. Includes Legendre and Jacobi symbols, some divisor functions, Euler's Phi function, etc.

License GPL (>= 3)

Repository CRAN

Repository/R-Forge/Project optimist

Repository/R-Forge/Revision 480

Repository/R-Forge/DateTimeStamp 2019-11-26 18:32:45

Date/Publication 2019-11-26 20:10:02 UTC

NeedsCompilation no

R topics documented:

numbers-package *Number-Theoretic Functions*

Description

Provides number-theoretic functions for factorization, prime numbers, twin primes, primitive roots, modular logarithm and inverses, extended GCD, Farey series and continuous fractions. Includes Legendre and Jacobi symbols, some divisor functions, Euler's Phi function, etc.

Details

The DESCRIPTION file:

Package:	numbers
Type:	Package
Title:	Number-Theoretic Functions
Version:	0.7-5
Date:	2019-11-26
Author:	Hans Werner Borchers
Maintainer:	Hans W. Borchers <hwborchers@googlemail.com>
Depends:	R (>= 3.1.0)
Suggests:	gmp (>= 0.5-1)
Description:	Provides number-theoretic functions for factorization, prime numbers, twin primes, pr
License:	GPL (>= 3)
Repository:	R-Forge
Repository/R-Forge/Project:	optimist
Repository/R-Forge/Revision:	480
Repository/R-Forge/DateTimeStamp:	2019-11-26 18:32:45
Date/Publication:	2019-11-26 18:32:45

Index of help topics:

GCD	GCD and LCM Integer Functions
Primes	Prime Numbers
Sigma	Divisor Functions
agm	Arithmetic-geometric Mean
bell	Bell Numbers
catalan	Catalan Numbers
cf2num	Generalized Continous Fractions
chinese	Chinese Remainder Theorem
collatz	Collatz Sequences
contFrac	Continous Fractions
coprime	Coprimality
div	Integer Division
divisors	List of Divisors
dropletPi	Droplet Algorithm for pi and e

egyptian_complete	Egyptian Fractions - Complete Search
egyptian_methods	Egyptian Fractions - Specialized Methods
eulersPhi	Eulers's Phi Function
extGCD	Extended Euclidean Algorithm
fibonacci	Fibonacci and Lucas Series
hermiteNF	Hermite Normal Form
iNthroot	Integer N-th Root
isIntpower	Powers of Integers
isNatural	Natural Number
isPrime	isPrime Property
isPrimroot	Primitive Root Test
legendre_sym	Legendre and Jacobi Symbol
mersenne	Mersenne Numbers
miller_rabin	Miller-Rabin Test
mod	Modulo Operator
modinv	Modular Inverse and Square Root
modlin	Modular Linear Equation Solver
modlog	Modular (or: Discrete) Logarithm
modpower	Power Function modulo m
moebius	Moebius Function
necklace	Necklace and Bracelet Functions
nextPrime	Next Prime
numbers-package	Number-Theoretic Functions
omega	Number of Prime Factors
ordpn	Order in Faculty
previousPrime	Previous Prime
primeFactors	Prime Factors
primroot	Primitive Root
pythagorean_triples	Pythagorean Triples
quadratic_residues	Quadratic Residues
ratFarey	Farey Approximation
rem	Integer Remainder
twinPrimes	Twin Primes
zeck	Zeckendorf Representation

Although R does not have a true integer data type, integers can be represented exactly up to 2^53-1 . The numbers package attempts to provided basic number-theoretic functions that will work correcty and relatively fast up to this level.

Author(s)

Hans Werner Borchers

Maintainer: Hans W. Borchers <hwborchers@googlemail.com>

References

Hardy, G. H., and E. M. Wright (1980). An Introduction to the Theory of Numbers. 5th Edition, Oxford University Press.

Riesel, H. (1994). Prime Numbers and Computer Methods for Factorization. Second Edition, Birkhaeuser Boston.

Crandall, R., and C. Pomerance (2005). Prime Numbers: A Computational Perspective. Springer Science+Business.

Shoup, V. (2009). A Computational Introduction to Number Theory and Algebra. Second Edition, Cambridge University Press.

Arndt, J. (2010). Matters Computational: Ideas, Algorithms, Source Code. 2011 Edition, Springer-Verlag, Berlin Heidelberg.

Forster, O. (2014). Algorithmische Zahlentheorie. 2. Auflage, Springer Spektrum Wiesbaden.

`agm` *Arithmetic-geometric Mean*

Description

The arithmetic-geometric mean of real or complex numbers.

Usage

```
agm(a, b)
```

Arguments

`a, b`	real or complex numbers.

Details

The arithmetic-geometric mean is defined as the common limit of the two sequences $a_{n+1} = (a_n + b_n)/2$ and $b_{n+1} = \sqrt{(a_n b_n)}$.

Value

Returnes one value, the algebraic-geometric mean.

Note

The calculation of the AGM is continued until the two values of `a` and `b` are identical (in machine accuracy).

References

Borwein, J. M., and P. B. Borwein (1998). Pi and the AGM: A Study in Analytic Number Theory and Computational Complexity. Second, reprinted Edition, A Wiley-interscience publication.

See Also

Arithmetic, geometric, and harmonic mean.

Examples

```
##  Gauss constant
1 / agm(1, sqrt(2))  # 0.834626841674073

##  Calculate the (elliptic) integral 2/pi \int_0^1 dt / sqrt(1 - t^4)
f <- function(t) 1 / sqrt(1-t^4)
2 / pi * integrate(f, 0, 1)$value
1 / agm(1, sqrt(2))

##  Calculate pi with quadratic convergence (modified AGM)
#   See algorithm 2.1 in Borwein and Borwein
y <- sqrt(sqrt(2))
x <- (y+1/y)/2
p <- 2+sqrt(2)
for (i in 1:6){
  cat(format(p, digits=16), "\n")
  p <- p * (1+x) / (1+y)
  s <- sqrt(x)
  y <- (y*s + 1/s) / (1+y)
  x <- (s+1/s)/2
  }

## Not run:
##  Calculate pi with arbitrary precision using the Rmpfr package
require("Rmpfr")
vpa <- function(., d = 32) mpfr(., precBits = 4*d)
# Function to compute \pi to d decimal digits accuracy, based on the
# algebraic-geometric mean, correct digits are doubled in each step.
agm_pi <- function(d) {
    a <- vpa(1, d)
    b <- 1/sqrt(vpa(2, d))
    s <- 1/vpa(4, d)
    p <- 1
    n <- ceiling(log2(d));
    for (k in 1:n) {
        c <- (a+b)/2
        b <- sqrt(a*b)
        s <- s - p * (c-a)^2
        p <- 2 * p
        a <- c
    }
    return(a^2/s)
}
d <- 64
pia <- agm_pi(d)
print(pia, digits = d)
# 3.14159265358979323846264338327950288419716939937510582097494459 2
# 3.14159265358979323846264338327950288419716939937510582097494459 23 exact

## End(Not run)
```

bell *Bell Numbers*

Description

Generate Bell numbers.

Usage

```
bell(n)
```

Arguments

n	integer, asking for the n-th Bell number.

Details

Bell numbers, commonly denoted as B_n, are defined as the number of partitions of a set of n elements. They can easily be calculated recursively.

Bell numbers also appear as moments of probability distributions, for example B_n is the n-th momentum of the Poisson distribution with mean 1.

Value

A single integer, as long as n<=22.

Examples

```
sapply(0:10, bell)
#      1      1      2      5     15     52    203    877   4140  21147 115975
```

catalan *Catalan Numbers*

Description

Generate Catalan numbers.

Usage

```
catalan(n)
```

Arguments

n	integer, asking for the n-th Catalan number.

Details

Catalan numbers, commonly denoted as C_n, are defined as

$$C_n = \frac{1}{n+1}\binom{2n}{n}$$

and occur regularly in all kinds of enumeration problems.

Value

A single integer, as long as `n<=30`.

Examples

```
C <- numeric(10)
for (i in 1:10) C[i] <- catalan(i)
C[5]                                #=> 42
```

`cf2num` *Generalized Continous Fractions*

Description

Evaluate a generalized continuous fraction as an alternating sum.

Usage

```
cf2num(a, b = 1, a0 = 0, finite = FALSE)
```

Arguments

`a`	numeric vector of length greater than 2.
`b`	numeric vector of length 1 or the same length as a.
`a0`	absolute term, integer part of the continuous fraction.
`finite`	logical; shall Algorithm 1 be applied.

Details

Calculates the numerical value of (simple or generalized) continued fractions of the form

$$a_0 + \frac{b1}{a1+}\frac{b2}{a2+}\frac{b3}{a3+...}$$

by converting it into an alternating sum and then applying the accelleration Algorithm 1 of Cohen et al. (2000).

The argument b is by default set to $b = (1, 1, ...)$, that is the continued fraction is treated in its simple form.

With `finite=TRUE` the accelleration is turned off.

Value

Returns a numerical value, an approximation of the continued fraction.

Note

This function is *not* vectorized.

References

H. Cohen, F. R. Villegas, and Don Zagier (2000). Experimental Mathematics, Vol. 9, No. 1, pp. 3-12. <www.emis.de/journals/EM>

See Also

contFrac

Examples

```
##  Examples from Wolfram Mathworld
print(cf2num(1:25), digits=16)  # 0.6977746579640077, eps()

a = 2*(1:25) + 1; b = 2*(1:25); a0 = 1  # 1/(sqrt(exp(1))-1)
cf2num(a, b, a0)                        # 1.541494082536798

a <- b <- 1:25                          # 1/(exp(1)-1)
cf2num(a, b)                            # 0.5819767068693286

a <- rep(1, 100); b <- 1:100; a0 <- 1   # 1.5251352761609812
cf2num(a, b, a0, finite = FALSE)        # 1.525135276161128
cf2num(a, b, a0, finite = TRUE)         # 1.525135259240266
```

`chinese remainder theorem`

Chinese Remainder Theorem

Description

Executes the Chinese Remainder Theorem (CRT).

Usage

```
chinese(a, m)
```

Arguments

a	sequence of integers, same length as m.
m	sequence of integers, relatively prime to each other.

Details

The Chinese Remainder Theorem says that given integers a_i and natural numbers m_i, relatively prime (i.e., coprime) to each other, there exists a unique solution $x = x_i$ such that the following system of linear modular equations is satisfied:

$$x_i = a_i \, mod \, m_i, \quad 1 \leq i \leq n$$

More generally, a solution exists if the following condition is satisfied:

$$a_i = a_j \, mod \, gcd(m_i, m_j)$$

This version of the CRT is not yet implemented.

Value

Returns th (unique) solution of the system of modular equalities as an integer between `0` and `M=prod(m)`.

See Also

`extGCD`

Examples

```
m <- c(3, 4, 5)
a <- c(2, 3, 1)
chinese(a, m)    #=> 11

# ... would be sufficient
# m <- c(50, 210, 154)
# a <- c(44,  34, 132)
# x = 4444
```

`collatz` *Collatz Sequences*

Description

Generates Collatz sequences with n -> k*n+l for n odd.

Usage

```
collatz(n, k = 3, l = 1, short = FALSE, check = TRUE)
```

Arguments

n	integer to start the Collatz sequence with.
k, l	parameters for computing k*n+l.
short	logical, abbreviate stps with (k*n+l)/2
check	logical, check for nontrivial cycles.

Details

Function n,k,l generates iterative sequences starting with n and calculating the next number as n/2 if n is even and k*n+l if n is odd. It stops automatically when 1 is reached.

The default parameters k=3,l=1 generate the classical Collatz sequence. The Collatz conjecture says that every such sequences will end in the trivial cycle ...,4,2,1. For other parameters this does not necessarily happen.

k and l are not allowed to be both even or both odd – to make k*n+l even for n odd. Option short=TRUE calculates (k*n+l)/2 when n is odd (as k*n+l is even in this case), shortening the sequence a bit.

With option check=TRUE will check for nontrivial cycles, stopping with the first integer that repeats in the sequence. The check is disabled for the default parameters in the light of the Collatz conjecture.

Value

Returns the integer sequence generated from the iterative rule.

Sends out a message if a nontrivial cycle was found (i.e. the sequence is not ending with 1 and end in an infinite cycle). Throws an error if an integer overflow is detected.

Note

The Collatz or 3n+1-conjecture has been experimentally verified for all start numbers n up to 10^20 at least.

References

See the Wikipedia entry on the 'Collatz Conjecture'.

Examples

```
collatz(7)  # n -> 3n+1
## [1]  7 22 11 34 17 52 26 13 40 20 10  5 16  8  4  2  1
collatz(9, short = TRUE)
## [1]  9 14  7 11 17 26 13 20 10  5  8  4  2  1

collatz(7, l = -1)  # n -> 3n-1
## Found a non-trivial cycle for n = 7 !
##     [1]  7 20 10  5 14  7

## Not run:
collatz(5, k = 7, l = 1)  # n -> 7n+1
```

```
## [1]  5 36 18  9 64 32 16  8  4  2  1
collatz(5, k = 7, l = -1)  # n -> 7n-1
## Info: 5 --> 1.26995e+16 too big after 280 steps.
## Error in collatz(5, k = 7, l = -1) :
##     Integer overflow, i.e. greater than 2^53-1

## End(Not run)
```

`contFrac` *Continous Fractions*

Description

Evaluate a continuous fraction or generate one.

Usage

```
contFrac(x, tol = 1e-06)
```

Arguments

`x`	a numeric scalar or vector.
`tol`	tolerance; default `1e-6` to make a nicer appearance for `pi`.

Details

If `x` is a scalar its continuous fraction will be generated up to the accuracy prescribed in `tol`. If it is of length greater 1, the function assumes this is a continuous fraction and computes its value.

For implementation `contfrac` uses the representation of continuous fractions through 2-by-2 matrices, i.e. the recursion formula.

Value

Either a numeric value, or a list with components `cf`, numeric vector representing the continuous fraction $[b_0; b_1, \ldots, b_{n-1}]$; `rat`, the rational number as a vector with (numerator, denumerator); and `prec`, the difference between `x` and the value of the contimuous fraction.

Note

This function is *not* vectorized.

References

Hardy, G. H., and E. M. Wright (1979). An Introduction to the Theory of Numbers. Fifth Edition, Oxford University Press, New York.

See Also

`cf2num`, `ratFarey`

Examples

```
contFrac(pi)
contFrac(c(3, 7, 15, 1))        # rational Approx: 355/113

contFrac(0.555)                 #  0  1  1  4 22
contFrac(c(1, rep(2, 25)))      #  1.414213562373095, sqrt(2)
```

`coprime` *Coprimality*

Description

Determine whether two numbers are coprime, i.e. do not have a common prime divisor.

Usage

```
coprime(n,m)
```

Arguments

`n, m`	integer scalars

Details

Two numbers are coprime iff their greatest common divisor is 1.

Value

Logical, being TRUE if the numbers are coprime.

See Also

GCD

Examples

```
coprime(46368, 75025)  # Fibonacci numbers are relatively prime to each other
coprime(1001, 1334)
```

div *Integer Division*

Description

Integer division.

Usage

```
div(n, m)
```

Arguments

n	numeric vector (preferably of integers)
m	integer vector (positive, zero, or negative)

Details

`div(n,m)` is integer division, that is discards the fractional part, with the same effect as `n %/% m`. It can be defined as `floor(n/m)`.

Value

A numeric (integer) value or vector/matrix.

See Also

mod, rem

Examples

```
div(c(-5:5), 5)
div(c(-5:5), -5)
div(c(1, -1), 0)  #=> Inf -Inf
div(0,c(0, 1))    #=> NaN  0
```

divisors *List of Divisors*

Description

Generates a list of divisors of an integer number.

Usage

```
divisors(n)
```

Arguments

n	integer whose divisors will be generated.

Details

The list of all divisors of an integer n will be calculated and returned in ascending order, including 1 and the number itself. For n>=1000 the list of prime factors of n will be used, for smaller n a total search is applied.

Value

Returns a vector integers.

See Also

primeFactors, Sigma, tau

Examples

```
divisors(1)          # 1
divisors(2)          # 1 2
divisors(2^5)        # 1  2  4  8 16 32
divisors(1000)       # 1  2  4  5  8 10 ... 100 125 200 250 500 1000
divisors(1001)       # 1  7 11 13 77 91 143 1001
```

---

dropletPi *Droplet Algorithm for pi and e*

Description

Generates digits for pi resp. the Euler number e.

Usage

```
dropletPi(n)
dropletE(n)
```

Arguments

n	number of digits after the decimal point; should not exceed 1000 much as otherwise it will be *very* slow.

Details

Based on a formula discovered by S. Rabinowitz and S. Wagon.

The droplet algorithm for pi uses the Euler transform of the alternating Leibniz series and the so-called “radix conversion".

Value

String containing "3.1415926..." resp. "2.718281828..." with n digits after the decimal point (i.e., internal decimal places).

References

Borwein, J., and K. Devlin (2009). The Computer as Crucible: An Introduction to Experimental Mathematics. A K Peters, Ltd.

Arndt, J., and Ch. Haenel (2000). Pi – Algorithmen, Computer, Arithmetik. Springer-Verlag, Berlin Heidelberg.

Examples

```
##  Example
dropletE(20)                    # [1] "2.71828182845904523536"
print(exp(1), digits=20)        # [1]  2.7182818284590450908

dropletPi(20)                   # [1] "3.14159265358979323846"
print(pi, digits=20)            # [1]  3.141592653589793116

## Not run:
E <- dropletE(1000)
table(strsplit(substring(E, 3, 1002), ""))
#    0   1   2   3   4   5   6   7   8   9
#  100  96  97 109 100  85  99  99 103 112

Pi <- dropletPi(1000)
table(strsplit(substring(Pi, 3, 1002), ""))
#   0   1   2   3   4   5   6   7   8   9
#  93 116 103 102  93  97  94  95 101 106
## End(Not run)
```

---

egyptian_complete *Egyptian Fractions - Complete Search*

Description

Generate all Egyptian fractions of length 2 and 3.

Usage

```
egyptian_complete(a, b, show = TRUE)
```

Arguments

a, b	integers, a != 1, a < b and a, b relatively prime.
show	logical; shall solutions found be printed?

Details

For a rational number `0 < a/b < 1`, generates all Egyptian fractions of length 2 and three, that is finds integers `x1, x2, x3` such that

`a/b = 1/x1 + 1/x2`
`a/b = 1/x1 + 1/x2 + 1/x3`.

Value

All solutions found will be printed to the console if `show=TRUE`; returns invisibly the number of solutions found.

References

http://www.ics.uci.edu/~eppstein/numth/egypt/

See Also

egyptian_methods

Examples

```
egyptian_complete(6, 7)         # 1/2 + 1/3 + 1/42
egyptian_complete(8, 11)        # no solution with 2 or 3 fractions

# TODO
# 2/9 = 1/9 + 1/10 + 1/90       # is not recognized, as similar cases,
                                # because 1/n is not considered in m/n.
```

---

egyptian_methods *Egyptian Fractions - Specialized Methods*

Description

Generate Egyptian fractions with specialized methods.

Usage

```
egyptian_methods(a, b)
```

Arguments

`a, b`	integers, a != 1, a < b and a, b relatively prime.

Details

For a rational number `0 < a/b < 1`, generates Egyptian fractions that is finds integers `x1, x2, ..., xk` such that

`a/b = 1/x1 + 1/x2 + ... + 1/xk`

using the following methods:

- 'greedy'
- Fibonacci-Sylvester
- Golomb (same as with Farey sequences)
- continued fractions (not yet implemented)

Value

No return value, all solutions found will be printed to the console.

References

http://www.ics.uci.edu/~eppstein/numth/egypt/

See Also

egyptian_complete

Examples

```
egyptian_methods(8, 11)
# 8/11 = 1/2 +  1/5 + 1/37 + 1/4070  (Fibonacci-Sylvester)
# 8/11 = 1/2 +  1/6 + 1/21 + 1/77    (Golomb-Farey)

# Other solutions
# 8/11 = 1/2 +  1/8 + 1/11 + 1/88
# 8/11 = 1/2 + 1/12 + 1/22 + 1/121
```

---

eulersPhi *Eulers's Phi Function*

Description

Euler's Phi function (aka Euler's 'totient' function).

Usage

```
eulersPhi(n)
```

Arguments

n	Positive integer.

Details

The phi function is defined to be the number of positive integers less than or equal to n that are *coprime* to n, i.e. have no common factors other than 1.

Value

Natural number, the number of coprime integers <= n.

Note

Works well up to 10^9.

See Also

primeFactors, Sigma

Examples

```
eulersPhi(9973)  == 9973 - 1                      # for prime numbers
eulersPhi(3^10)  == 3^9 * (3 - 1)                 # for prime powers
eulersPhi(12*35) == eulersPhi(12) * eulersPhi(35)  # TRUE if coprime

## Not run:
x <- 1:100; y <- sapply(x, eulersPhi)
plot(1:100, y, type="l", col="blue",
               xlab="n", ylab="phi(n)", main="Euler's totient function")
points(1:100, y, col="blue", pch=20)
grid()
## End(Not run)
```

---

extGCD *Extended Euclidean Algorithm*

Description

The extended Euclidean algorithm computes the greatest common divisor and solves Bezout's identity.

Usage

```
extGCD(a, b)
```

Arguments

a, b	integer scalars

Details

The extended Euclidean algorithm not only computes the greatest common divisor d of a and b, but also two numbers n and m such that $d = na + mb$.

This algorithm provides an easy approach to computing the modular inverse.

Value

a numeric vector of length three, `c(d,n,m)`, where `d` is the greatest common divisor of `a` and `b`, and `n` and `m` are integers such that `d = n*a + m*b`.

Note

There is also a shorter, more elegant recursive version for the extended Euclidean algorithm. For R the procedure suggested by Blankinship appeared more appropriate.

References

Blankinship, W. A. “A New Version of the Euclidean Algorithm." Amer. Math. Monthly 70, 742-745, 1963.

See Also

`GCD`

Examples

```
extGCD(12, 10)
extGCD(46368, 75025)  # Fibonacci numbers are relatively prime to each other
```

`fibonacci` *Fibonacci and Lucas Series*

Description

Generates single Fibonacci numbers or a Fibonacci sequence; or generates a Lucas series based on the Fibonacci series.

Usage

```
fibonacci(n, sequence = FALSE)
lucas(n)
```

Arguments

`n`	an integer.
`sequence`	logical; default: FALSE.

Details

Generates the n-th Fibonacci number, or the whole Fibonacci sequence from the first to the n-th number; starts with (1, 1, 2, 3, ...). Generates only single Lucas numbers. The Lucas series can be extenden to the left and starts as (... -4, 3, -1, 2, 1, 3, 4, ...).

The recursive version is too slow for values n>=30. Therefore, an iterative approach is used. For numbers n > 78 Fibonacci numbers cannot be represented exactly in R as integers (>2^53-1).

Value

A single integer, or a vector of integers.

Examples

```
fibonacci(0)                            # 0
fibonacci(2)                            # 1
fibonacci(2, sequence = TRUE)           # 1 1
fibonacci(78)                           # 8944394323791464 < 9*10^15

lucas(0)                                # 2
lucas(2)                                # 3
lucas(76)                               # 7639424778862807

# Golden ratio
F <- fibonacci(25, sequence = TRUE)     # ... 46368 75025
f25 <- F[25]/F[24]                      # 1.618034
phi <- (sqrt(5) + 1)/2
abs(f25 - phi)                          # 2.080072e-10

# Fibonacci numbers w/o iteration
  fibo <- function(n) {
    phi <- (sqrt(5) + 1)/2
    fib <- (phi^n - (1-phi)^n) / (2*phi - 1)
    round(fib)
  }
fibo(30:33)                             # 832040 1346269 2178309 3524578

for (i in -8:8) cat(lucas(i), " ")
# 47  -29  18  -11  7  -4  3  -1  2  1  3  4  7  11  18  29  47

# Lucas numbers w/o iteration
  luca <- function(n) {
    phi <- (sqrt(5) + 1)/2
    luc <- phi^n + (1-phi)^n
    round(luc)
  }
luca(0:10)
# [1]   2   1   3   4   7  11  18  29  47  76 123

# Lucas primes
#   for (j in 0:76) {
#     l <- lucas(j)
```

```
#     if (isPrime(l)) cat(j, "\t", l, "\n")
#   }
# 0   2
# 2   3
# ...
# 71  688846502588399
```

GCD, LCM *GCD and LCM Integer Functions*

Description

Greatest common divisor and least common multiple

Usage

```
GCD(n, m)
LCM(n, m)

mGCD(x)
mLCM(x)
```

Arguments

n, m	integer scalars.
x	a vector of integers.

Details

Computation based on the Euclidean algorithm without using the extended version.

mGCD (the multiple GCD) computes the greatest common divisor for all numbers in the integer vector x together.

Value

A numeric (integer) value.

Note

The following relation is always true:

n * m = GCD(n,m) * LCM(n,m)

See Also

extGCD, coprime

Examples

```
GCD(12, 10)
GCD(46368, 75025)  # Fibonacci numbers are relatively prime to each other

LCM(12, 10)
LCM(46368, 75025)  # = 46368 * 75025

mGCD(c(2, 3, 5, 7) * 11)
mGCD(c(2*3, 3*5, 5*7))
mLCM(c(2, 3, 5, 7) * 11)
mLCM(c(2*3, 3*5, 5*7))
```

Hermite normal form *Hermite Normal Form*

Description

Hermite normal form over integers (in column-reduced form).

Usage

```
hermiteNF(A)
```

Arguments

A	integer matrix.

Details

An mxn-matrix of rank r with integer entries is said to be in Hermite normal form if:

(i) the first r columns are nonzero, the other columns are all zero;
(ii) The first r diagonal elements are nonzero and d[i-1] divides d[i] for i = 2,...,r .
(iii) All entries to the left of nonzero diagonal elements are non-negative
and strictly less than the corresponding diagonal entry.

The lower-triangular Hermite normal form of A is obtained by the following three types of column operations:

(i) exchange two columns
(ii) multiply a column by -1
(iii) Add an integral multiple of a column to another column

U is the unitary matrix such that AU = H, generated by these operations.

Value

List with two matrices, the Hermite normal form H and the unitary matrix U.

Note

Another normal form often used in this context is the Smith normal form.

References

Cohen, H. (1993). A Course in Computational Algebraic Number Theory. Graduate Texts in Mathematics, Vol. 138, Springer-Verlag, Berlin, New York.

See Also

chinese

Examples

```
n <- 4; m <- 5
A = matrix(c(
 9,  6,  0, -8,  0,
-5, -8,  0,  0,  0,
 0,  0,  0,  4,  0,
 0,  0,  0, -5,  0), n, m, byrow = TRUE)

Hnf <- hermiteNF(A); Hnf
# $H =  1    0    0    0    0
#       1    2    0    0    0
#      28   36   84    0    0
#     -35  -45 -105    0    0
# $U = 11   14   32    0    0
#      -7   -9  -20    0    0
#       0    0    0    1    0
#       7    9   21    0    0
#       0    0    0    0    1

r <- 3                  # r = rank(H)
H <- Hnf$H; U <- Hnf$U
all(H == A %*% U)       #=> TRUE

##  Example: Compute integer solution of A x = b
#   H = A * U, thus H * U^-1 * x = b, or H * y = b
b <- as.matrix(c(-11, -21, 16, -20))

y <- numeric(m)
y[1] <- b[1] / H[1, 1]
for (i in 2:r)
    y[i] <- (b[i] - sum(H[i, 1:(i-1)] * y[1:(i-1)])) / H[i, i]
# special solution:
xs <- U %*% y         #  1 2 0 4 0

# and the general solution is xs + U * c(0, 0, 0, a, b), or
# in other words the basis are the m-r vectors c(0,...,0, 1, ...).
# If the special solution is not integer, there are no integer solutions.
```

iNthroot *Integer N-th Root*

Description

Determine the integer n-th root of .

Usage

```
iNthroot(p, n)
```

Arguments

p	any positive number.
n	a natural number.

Details

Calculates the highest natural number below the n-th root of p in a more integer based way than simply `floor(p^{1/n})`.

Value

An integer.

Examples

```
iNthroot(0.5, 6)    # 0
iNthroot(1, 6)      # 1
iNthroot(5^6, 6)    # 5
iNthroot(5^6-1, 6)  # 4
## Not run:
# Define a function that tests whether
isNthpower <- function(p, n) {
    q <- iNthroot(p, n)
    if (q^n == p) { return(TRUE)
    } else { return(FALSE) }
  }

## End(Not run)
```

`isIntpower` *Powers of Integers*

Description

Determine whether `p` is the power of an integer.

Usage

```
isIntpower(p)
```

Arguments

`p`	any integer number.

Details

Determines whether `p` is the power of an integer and returns a tupel `(n,m)` such that `p=n^m` where `m` is as small as possible. E.g., if `p` is prime it returns `c(p,1)`.

Value

A 2-vector of integers.

Examples

```
isIntpower(1)    #  1  1
isIntpower(15)   # 15  1
isIntpower(17)   # 17  1
isIntpower(64)   #  8  2
isIntpower(36)   #  6  2
isIntpower(100)  # 10  2
## Not run:
  for (p in 5^7:7^5) {
      pp <- isIntpower(p)
      if (pp[2] != 1) cat(p, ":\t", pp, "\n")
  }
## End(Not run)
```

isNatural *Natural Number*

Description

Natural number type.

Usage

```
isNatural(n)
```

Arguments

n	any numeric number.

Details

Returns TRUE for natural (or: whole) numbers between 1 and 2^53-1.

Value

Boolean

Examples

```
IsNatural <- Vectorize(isNatural)
IsNatural(c(-1, 0, 1, 5.1, 10, 2^53-1, 2^53, Inf, NA))
```

isPrime *isPrime Property*

Description

Vectorized version, returning for a vector or matrix of positive integers a vector of the same size containing 1 for the elements that are prime and 0 otherwise.

Usage

```
isPrime(x)
```

Arguments

x	vector or matrix of nonnegative integers

Details

Given an array of positive integers returns an array of the same size of 0 and 1, where the i indicates a prime number in the same position.

Value

array of elements 0, 1 with 1 indicating prime numbers

See Also

primeFactors,Primes

Examples

```
x <- matrix(1:10, nrow=10, ncol=10, byrow=TRUE)
x * isPrime(x)

# Find first prime number octett:
octett <- c(0, 2, 6, 8, 30, 32, 36, 38) - 19
while (TRUE) {
    octett <- octett + 210
    if (all(isPrime(octett))) {
        cat(octett, "\n", sep="  ")
        break
    }
}
```

---

isPrimroot *Primitive Root Test*

Description

Determine whether g generates the multiplicative group modulo p.

Usage

```
isPrimroot(g, p)
```

Arguments

g	integer greater 2 (and smaller than p).
p	prime number.

Details

Test is done by determining the order of g modulo p.

Value

Returns TRUE or FALSE.

Examples

```
isPrimroot(2, 7)
isPrimroot(2, 71)
isPrimroot(7, 71)
```

`legendre_sym` *Legendre and Jacobi Symbol*

Description

Legendre and Jacobi Symbol for quadratic residues.

Usage

```
legendre_sym(a, p)

jacobi_sym(a, n)
```

Arguments

`a, n`	integers.
`p`	prime number.

Details

The Legendre Symbol `(a/p)`, where `p` must be a prime number, denotes whether `a` is a quadratic residue modulo `p` or not.

The Jacobi symbol `(a/p)` is the product of `(a/p)` of all prime factors `p` on `n`.

Value

Returns 0, 1, or -1 if `p` divides `a`, `a` is a quadratic residue, or if not.

See Also

`quadratic_residues`

Examples

```
Lsym <- Vectorize(legendre_sym, 'a')

# all quadratic residues of p = 17
qr17 <- which(Lsym(1:16, 17) == 1)      #  1  2  4  8  9 13 15 16
sort(unique((1:16)^2 %% 17))            #  the same

## Not run:
# how about large numbers?
p <- 1198112137                         #  isPrime(p) TRUE
```

```
x <- 4652356
a <- mod(x^2, p)                        #  520595831
legendre_sym(a, p)                      #  1
legendre_sym(a+1, p)                    # -1

## End(Not run)

jacobi_sym(11, 12)                      # -1
```

`mersenne` *Mersenne Numbers*

Description

Determines whether p is a Mersenne number, that is such that $2^p - 1$ is prime.

Usage

```
mersenne(p)
```

Arguments

p	prime number, not very large.

Details

Applies the Lucas-Lehmer test on p. Because intermediate numbers will soon get very large, uses 'gmp' from the beginning.

Value

Returns TRUE or FALSE, indicating whether p is a Mersenne number or not.

References

http://mathworld.wolfram.com/Lucas-LehmerTest.html

Examples

```
mersenne(2)

## Not run:
  P <- Primes(32)
  M <- c()
  for (p in P)
      if (mersenne(p)) M <- c(M, p)
  # Next Mersenne numpers with primes are 521 and 607 (below 1200)
  M                       # 2   3   5    7    13      17  19  31  61  89  107
  gmp::as.bigz(2)^M - 1   # 3   7  31  127  8191  131071  ...
## End(Not run)
```

`miller_rabin` *Miller-Rabin Test*

Description

Probabilistic Miller-Rabin primality test.

Usage

```
miller_rabin(n)
```

Arguments

`n`	natural number.

Details

The Miller-Rabin test is an efficient probabilistic primality test based on strong pseudoprimes. This implementation uses the first seven prime numbers (if necessary) as test cases. It is thus exact for all numbers `n` < `341550071728321`.

Value

Returns TRUE or FALSE.

Note

`miller_rabin()` will only work if package `gmp` has been loaded by the user separately.

References

http://mathworld.wolfram.com/Rabin-MillerStrongPseudoprimeTest.html

See Also

`isPrime`

Examples

```
miller_rabin(2)

## Not run:
  miller_rabin(4294967297)  #=> FALSE
  miller_rabin(4294967311)  #=> TRUE

  # Rabin-Miller 10 times faster than nextPrime()
  N <- n <- 2^32 + 1
  system.time(while (!miller_rabin(n)) n <- n + 1)  # 0.003
  system.time(p <- nextPrime(N))                    # 0.029
```

```
  N <- c(2047, 1373653, 25326001, 3215031751, 2152302898747,
          3474749660383, 341550071728321)
  for (n in N) {
      p <- nextPrime(n)
      T <- system.time(r <- miller_rabin(p))
      cat(n, p, r, T[3], "\n")}
## End(Not run)
```

---

mod *Modulo Operator*

Description

Modulo operator.

Usage

```
mod(n, m)
```

Arguments

n	numeric vector (preferably of integers)
m	integer vector (positive, zero, or negative)

Details

`mod(n,m)` is the modulo operator and returns $n \, mod \, m$. `mod(n,0)` is `n`, and the result always has the same sign as `m`.

Value

a numeric (integer) value or vector/matrix

Note

The following relation is fulfilled (for `m != 0`):

`mod(n,m) = n -m * floor(n/m)`

See Also

rem, div

Examples

```
mod(c(-5:5), 5)
mod(c(-5:5), -5)
mod(0, 1)          #=> 0
mod(1, 0)          #=> 1
```

`modinv, modsqrt` *Modular Inverse and Square Root*

Description

Computes the modular inverse of `n` modulo `m`.

Usage

```
modinv(n, m)

modsqrt(a, p)
```

Arguments

`n, m`	integer scalars.
`a, p`	integer modulo p, p a prime.

Details

The modular inverse of `n` modulo `m` is the unique natural number `0 < n0 < m` such that `n * n0 = 1 mod m`. It is a simple application of the extended GCD algorithm.

The modular square root of `a` modulo a prime `p` is a number `x` such that `x^2 = a mod p`. If `x` is a solution, then `p-x` is also a solution module `p`. The function will always return the smaller value.

`modsqrt` implements the Tonelli-Shanks algorithm which also works for square roots modulo prime powers. The general case is NP-hard.

Value

A natural number smaller `m`, if `n` and `m` are coprime, else `NA`. `modsqrt` will return 0 if there is no solution.

See Also

`extGCD`

Examples

```
modinv(5, 1001)  #=> 801, as 5*801 = 4005 = 1 mod 1001

Modinv <- Vectorize(modinv, "n")
((1:10)*Modinv(1:10, 11)) %% 11     #=> 1 1 1 1 1 1 1 1 1 1

modsqrt( 8, 23)  # 10 because 10^2 = 100 = 8 mod 23
modsqrt(10, 17)  #  0 because 10 is not a quadratic residue mod 17
```

modlin *Modular Linear Equation Solver*

Description

Solves the modular equation a x = b mod n.

Usage

```
modlin(a, b, n)
```

Arguments

a, b, n	integer scalars

Details

Solves the modular equation a x = b mod n. This eqation is solvable if and only if gcd(a,n)|b. The function uses the extended greatest common divisor approach.

Value

Returns a vector of integer solutions.

See Also

extGCD

Examples

```
modlin(14, 30, 100)            # 95 45
modlin(3, 4, 5)                # 3
modlin(3, 5, 6)                # []
modlin(3, 6, 9)                # 2 5 8
```

modlog *Modular (or: Discrete) Logarithm*

Description

Realizes the modular (or discrete) logarithm modulo a prime number p, that is determines the unique exponent n such that $g^n = x \bmod p$, g a primitive root.

Usage

```
modlog(g, x, p)
```

Arguments

g	a primitive root mod p.
x	an integer.
p	prime number.

Details

The method is in principle a complete search, cut short by "Shank's trick", the giantstep-babystep approach, see Forster (1996, pp. 65f). g has to be a primitive root modulo p, otherwise exponentiation is not bijective.

Value

Returns an integer.

References

Forster, O. (1996). Algorithmische Zahlentheorie. Friedr. Vieweg u. Sohn Verlagsgesellschaft mbH, Wiesbaden.

See Also

primroot

Examples

```
modlog(11, 998, 1009)  # 505 , i.e., 11^505 = 998 mod 1009
```

---

modpower *Power Function modulo m*

Description

Calculates powers and orders modulo m.

Usage

```
modpower(n, k, m)
modorder(n, m)
```

Arguments

n, k, m	Natural numbers, m >= 1.

Details

modpower calculates n to the power of k modulo m.
Uses modular exponentiation, as described in the Wikipedia article.

modorder calculates the order of n in the multiplicative group module m. n and m must be coprime.
Uses brute force, trick to use binary expansion and square is not more efficient in an R implementation.

Value

Natural number.

Note

This function is *not* vectorized.

See Also

primroot

Examples

```
modpower(2, 100, 7)  #=> 2
modpower(3, 100, 7)  #=> 4
modorder(7, 17)      #=> 16, i.e. 7 is a primitive root mod 17

##  Gauss' table of primitive roots modulo prime numbers < 100
proots <- c(2,  2,  3,  2,  2,  6,  5, 10, 10, 10, 2,  2, 10, 17,  5,  5,
            6, 28, 10, 10, 26, 10, 10,  5, 12, 62, 5, 29, 11, 50, 30, 10)
P <- Primes(100)
for (i in seq(along=P)) {
    cat(P[i], "\t", modorder(proots[i], P[i]), proots[i], "\t", "\n")
}

## Not run:
##  Lehmann's primality test
lehmann_test <- function(n, ntry = 25) {
    if (!is.numeric(n) || ceiling(n) != floor(n) || n < 0)
        stop("Argument 'n' must be a natural number")
    if (n >= 9e7)
        stop("Argument 'n' should be smaller than 9e7.")

    if (n < 2)                      return(FALSE)
    else if (n == 2)                return(TRUE)
    else if (n > 2 && n %% 2 == 0)  return(FALSE)

    k <- floor(ntry)
    if (k < 1) k <- 1
    if (k > n-2) a <- 2:(n-1)
    else         a <- sample(2:(n-1), k, replace = FALSE)

    for (i in 1:length(a)) {
```

```
        m <- modpower(a[i], (n-1)/2, n)
        if (m != 1 && m != n-1) return(FALSE)
    }
    return(TRUE)
}

##  Examples
for (i in seq(1001, 1011, by = 2))
    if (lehmann_test(i)) cat(i, "\n")
# 1009
system.time(lehmann_test(27644437, 50))    # TRUE
#    user  system elapsed
#   0.086   0.151   0.235

## End(Not run)
```

`moebius` *Moebius Function*

Description

The classical Moebius and Mertens functions in number theory.

Usage

```
moebius(n)
mertens(n)
```

Arguments

`n`	Positive integer.

Details

`moebius(n)` is `+1` if `n` is a square-free positive integer with an even number of prime factors, or `+1` if there are an odd of prime factors. It is `0` if `n` is not square-free.

`mertens(n)` is the aggregating summary function, that sums up all values of `moebius` from `1` to `n`.

Value

For `moebius`, `0`, `1` or `-1`, depending on the prime decomposition of `n`.

For `mertens` the values will very slowly grow.

Note

Works well up to `10^9`, but will become very slow for the Mertens function.

See Also

primeFactors, eulersPhi

Examples

```
sapply(1:16, moebius)
sapply(1:16, mertens)

## Not run:
x <- 1:50; y <- sapply(x, moebius)
plot(c(1, 50), c(-3, 3), type="n")
grid()
points(1:50, y, pch=18, col="blue")

x <- 1:100; y <- sapply(x, mertens)
plot(c(1, 100), c(-5, 3), type="n")
grid()
lines(1:100, y, col="red", type="s")
## End(Not run)
```

---

necklace *Necklace and Bracelet Functions*

Description

Necklace and bracelet problems in combinatorics.

Usage

```
necklace(k, n)

bracelet(k, n)
```

Arguments

k	The size of the set or alphabet to choose from.
n	the length of the necklace or bracelet.

Details

A necklace is a closed string of length n over a set of size k (numbers, characters, clors, etc.), where all rotations are taken as equivalent. A bracelet is a necklace where strings may also be equivalent under reflections.

Polya's enumeration theorem can be utilized to enumerate all necklaces or bracelets. The final calculation involves Euler's Phi or totient function, in this package implemented as eulersPhi.

Value

Returns the number of necklaces resp. bracelets.

References

https://en.wikipedia.org/wiki/Necklace_(combinatorics)

Examples

```
necklace(2, 5)
necklace(3, 6)

bracelet(2, 5)
bracelet(3, 6)
```

`nextPrime` *Next Prime*

Description

Find the next prime above `n`.

Usage

```
nextPrime(n)
```

Arguments

`n`	natural number.

Details

`nextPrime` finds the next prime number greater than `n`, while `previousPrime` finds the next prime number below `n`. In general the next prime will occur in the interval `[n+1,n+log(n)]`.

In double precision arithmetic integers are represented exactly only up to 2^53 - 1, therefore this is the maximal allowed value.

Value

Integer.

See Also

`Primes`, `isPrime`

Examples

```
p <- nextPrime(1e+6)  # 1000003
isPrime(p)            # TRUE
```

omega *Number of Prime Factors*

Description

Sum of all exponents of prime factors in the prime decomposition.

Usage

```
omega(n)
Omega(n)
```

Arguments

n	Positive integer.

Details

Compute the number of prime factors of n resp. the sum of their exponents in the prime decomposition.

`(-1)^Omega(n)` is the Liouville function.

Value

Natural number.

Note

Works well up to `10^9`.

See Also

Sigma

Examples

```
omega(2*3*5*7*11*13*17*19)  #=> 8
Omega(2 * 3^2 * 5^3 * 7^4)  #=> 10
```

ordpn *Order in Faculty*

Description

Calculates the order of a prime number p in n!, i.e. the highest exponent e such that p^e|n!.

Usage

```
ordpn(p, n)
```

Arguments

p	prime number.
n	natural number.

Details

Applies the well-known formula adding terms floor(n/p^k).

Value

Returns the exponent e.

Examples

```
ordpn(2, 100)         #=> 97
ordpn(7, 100)         #=> 16
ordpn(101, 100)       #=>  0
ordpn(997, 1000)      #=>  1
```

previousPrime *Previous Prime*

Description

Find the next prime below n.

Usage

```
previousPrime(n)
```

Arguments

n	natural number.

Details

previousPrime finds the next prime number smaller than n, while nextPrime finds the next prime number below n. In general the previousn prime will occur in the interval [n-1,n-log(n)].

In double precision arithmetic integers are represented exactly only up to 2^53 - 1, therefore this is the maximal allowed value.

Value

Integer.

See Also

Primes, isPrime

Examples

```
p <- previousPrime(1e+6)  # 999983
isPrime(p)                # TRUE
```

---

primeFactors *Prime Factors*

Description

primeFactors computes a vector containing the prime factors of n. radical returns the product of those unique prime factors.

Usage

```
primeFactors(n)
radical(n)
```

Arguments

n	nonnegative integer

Details

Computes the prime factors of n in ascending order, each one as often as its multiplicity requires, such that n == prod(primeFactors(n)).

radical() is used in the abc-conjecture:

abc-triple: 1 <= a < b, a, b coprime, c = a + b

for every e > 0 there are only finitely many abc-triples with

c > radical(a*b*c)^(1+e)

Value

Vector containing the prime factors of n, resp. the product of unique prime factors.

See Also

divisors, gmp::factorize

Examples

```
  primeFactors(1002001)         # 7  7  11  11  13  13
  primeFactors(65537)           # is prime
  # Euler's calculation
  primeFactors(2^32 + 1)        # 641  6700417

  radical(1002001)              # 1001

## Not run:
  for (i in 1:99) {
    for (j in (i+1):100) {
      if (coprime(i, j)) {
        k = i + j
        r = radical(i*j*k)
        q = log(k) / log(r)  # 'quality' of the triple
        if (q > 1)
          cat(q, ":\t", i, ",", j, ",", k, "\n")
        }
      }
    }
## End(Not run)
```

Primes *Prime Numbers*

Description

Eratosthenes resp. Atkin sieve methods to generate a list of prime numbers less or equal n, resp. between n1 and n2.

Usage

```
Primes(n1, n2 = NULL)

atkin_sieve(n)
```

Arguments

n, n1, n2	natural numbers with n1 <= n2.

Details

The list of prime numbers up to n is generated using the "sieve of Eratosthenes". This approach is reasonably fast, but may require a lot of main memory when n is large.

Primes computes first all primes up to sqrt(n2) and then applies a refined sieve on the numbers from n1 to n2, thereby drastically reducing the need for storing long arrays of numbers.

The sieve of Atkins is a modified version of the ancient prime number sieve of Eratosthenes. It applies a modulo-sixty arithmetic and requires less memory, but in R is not faster because of a double for-loop.

In double precision arithmetic integers are represented exactly only up to 2^53 - 1, therefore this is the maximal allowed value.

Value

vector of integers representing prime numbers

References

A. Atkin and D. Bernstein (2004), Prime sieves using quadratic forms. Mathematics of Computation, Vol. 73, pp. 1023-1030.

See Also

isPrime, gmp::factorize, pracma::expint1

Examples

```
Primes(1000)
Primes(1949, 2019)

atkin_sieve(1000)

## Not run:
##  Appendix:  Logarithmic Integrals and Prime Numbers (C.F.Gauss, 1846)

library('gsl')
# 'European' form of the logarithmic integral
Li <- function(x) expint_Ei(log(x)) - expint_Ei(log(2))

# No. of primes and logarithmic integral for 10^i, i=1..12
i <- 1:12;  N <- 10^i
# piN <- numeric(12)
# for (i in 1:12) piN[i] <- length(primes(10^i))
piN <- c(4, 25, 168, 1229, 9592, 78498, 664579,
         5761455, 50847534, 455052511, 4118054813, 37607912018)
cbind(i, piN, round(Li(N)), round((Li(N)-piN)/piN, 6))

#  i     pi(10^i)      Li(10^i)  rel.err
# ---------------------------------------
#  1            4             5  0.280109
#  2           25            29  0.163239
```

```
#  3          168          177  0.050979
#  4         1229         1245  0.013094
#  5         9592         9629  0.003833
#  6        78498        78627  0.001637
#  7       664579       664917  0.000509
#  8      5761455      5762208  0.000131
#  9     50847534     50849234  0.000033
# 10    455052511    455055614  0.000007
# 11   4118054813   4118066400  0.000003
# 12  37607912018  37607950280  0.000001
# ------------------------------------
## End(Not run)
```

`primroot` *Primitive Root*

Description

Find the smallest primitive root modulo m, or find all primitive roots.

Usage

```
primroot(m, all = FALSE)
```

Arguments

`m`	A prime integer.
`all`	boolean; shall all primitive roots module p be found.

Details

For every prime number m there exists a natural number n that generates the field F_m, i.e. $n, n^2, ..., n^{m-1} mod(m)$ are all different.

The computation here is all brute force. As most primitive roots are relatively small, so it is still reasonable fast.

One trick is to factorize $m - 1$ and test only for those prime factors. In R this is not more efficient as factorization also takes some time.

Value

A natural number if `m` is prime, else `NA`.

Note

This function is *not* vectorized.

References

Arndt, J. (2010). Matters Computational: Ideas, Algorithms, Source Code. Springer-Verlag, Berlin Heidelberg Dordrecht.

See Also

modpower, modorder

Examples

```
P <- Primes(100)
R <- c()
for (p in P) {
    R <- c(R, primroot(p))
}
cbind(P, R)  # 7 is the biggest prime root here (for p=71)
```

`pythagorean_triples` *Pythagorean Triples*

Description

Generates all primitive Pythagorean triples (a, b, c) of integers such that $a^2 + b^2 = c^2$, where a, b, c are coprime (have no common divisor) and $c_1 \leq c \leq c_2$.

Usage

```
pythagorean_triples(c1, c2)
```

Arguments

`c1, c2`	lower and upper limit of the hypothenuses `c`.

Details

If (a, b, c) is a primitive Pythagorean triple, there are integers m, n with $1 \leq n < m$ such that

$$a = m^2 - n^2, b = 2mn, c = m^2 + n^2$$

with $gcd(m, n) = 1$ and $m - n$ being odd.

Value

Returns a matrix, one row for each Pythagorean triple, of the form `(m n a b c)`.

References

http://mathworld.wolfram.com/PythagoreanTriple.html

Examples

```
pythagorean_triples(100, 200)
##       [,1] [,2] [,3] [,4] [,5]
##  [1,]   10    1   99   20  101
##  [2,]   10    3   91   60  109
##  [3,]    8    7   15  112  113
##  [4,]   11    2  117   44  125
##  [5,]   11    4  105   88  137
##  [6,]    9    8   17  144  145
##  [7,]   12    1  143   24  145
##  [8,]   10    7   51  140  149
##  [9,]   11    6   85  132  157
## [10,]   12    5  119  120  169
## [11,]   13    2  165   52  173
## [12,]   10    9   19  180  181
## [13,]   11    8   57  176  185
## [14,]   13    4  153  104  185
## [15,]   12    7   95  168  193
## [16,]   14    1  195   28  197
```

---

quadratic_residues *Quadratic Residues*

Description

List all quadratic residues of an integer.

Usage

```
quadratic_residues(n)
```

Arguments

n	integer.

Details

Squares all numbers between 0 and `n/2` and generate a unique list of all these numbers modulo `n`.

Value

Vector of integers.

See Also

`legendre_sym`

Examples

```
quadratic_residues(17)
```

ratFarey *Farey Approximation*

Description

Rational approximation of real numbers through Farey fractions.

Usage

```
ratFarey(x, n, upper = TRUE)
```

Arguments

x	real number.
n	integer, highest allowed denominator in a rational approximation.
upper	logical; shall the Farey fraction be grater than x.

Details

Rational approximation of real numbers through Farey fractions, i.e. find for x the nearest fraction in the Farey series of rational numbers with denominator not larger than n.

Value

Returns a vector with two natural numbers, nominator and denominator.

References

Hardy, G. H., and E. M. Wright (1979). An Introduction to the Theory of Numbers. Fifth Edition, Oxford University Press, New York.

See Also

contFrac

Examples

```
ratFarey(pi, 100)                       # 22/7    0.0013
ratFarey(pi, 100, upper = FALSE)        # 311/99  0.0002
ratFarey(-pi, 100)                      # -22/7
ratFarey(pi - 3, 70)                    # pi ~= 3 + (3/8)^2
ratFarey(pi, 1000)                      # 355/113
ratFarey(pi, 10000, upper = FALSE)      # id.
ratFarey(pi, 1e5, upper = FALSE)        # 312689/99532 - pi ~= 3e-11

ratFarey(4/5, 5)                        # 4/5
ratFarey(4/5, 4)                        # 1/1
ratFarey(4/5, 4, upper = FALSE)         # 3/4
```

`rem` *Integer Remainder*

Description

Integer remainder function.

Usage

```
rem(n, m)
```

Arguments

`n`	numeric vector (preferably of integers)
`m`	must be a scalar integer (positive, zero, or negative)

Details

`rem(n,m)` is the same modulo operator and returns $n \, mod \, m$. `mod(n,0)` is `NaN`, and the result always has the same sign as `n` (for `n != m` and `m != 0`).

Value

a numeric (integer) value or vector/matrix

See Also

`mod`, `div`

Examples

```
rem(c(-5:5), 5)
rem(c(-5:5), -5)
rem(0, 1)         #=> 0
rem(1, 1)         #=> 0  (always for n == m)
rem(1, 0)         #   NA  (should be NaN)
rem(0, 0)         #=> NaN
```

Sigma *Divisor Functions*

Description

Sum of powers of all divisors of a natural number.

Usage

```
Sigma(n, k = 1, proper = FALSE)

tau(n)
```

Arguments

n	Positive integer.
k	Numeric scalar, the exponent to be used.
proper	Logical; if TRUE, n will *not* be considered as a divisor of itself; default: FALSE.

Details

Total sum of all integer divisors of n to the power of k, including 1 and n.

For k=0 this is the number of divisors, for k=1 it is the sum of all divisors of n.

tau is Ramanujan's *tau* function, here computed using Sigma(.,5) and Sigma(.,11).

A number is called *refactorable*, if tau(n) divides n, for example n=12 or n=18.

Value

Natural number, the number or sum of all divisors.

Note

Works well up to 10^9.

References

http://en.wikipedia.org/wiki/Divisor_function

http://en.wikipedia.org/wiki/Tau-function

See Also

primeFactors, divisors

Examples

```
sapply(1:16, Sigma, k = 0)
sapply(1:16, Sigma, k = 1)
sapply(1:16, Sigma, proper = TRUE)
```

twinPrimes *Twin Primes*

Description

Generate a list of twin primes between n1 and n2.

Usage

```
twinPrimes(n1, n2)
```

Arguments

n1, n2	natural numbers with n1 <= n2.

Details

twinPrimes uses Primes and uses diff to find all twin primes in the given interval.

In double precision arithmetic integers are represented exactly only up to 2^53 - 1, therefore this is the maximal allowed value.

Value

Returnes a nx2-matrix, where nis the number of twin primes found, and each twin tuple fills one row.

See Also

Primes

Examples

```
twinPrimes(1e6+1, 1e6+1001)
```

zeck *Zeckendorf Representation*

Description

Generates the Zeckendorf representation of an integer as a sum of Fibonacci numbers.

Usage

```
zeck(n)
```

Arguments

n	integer.

Details

According to Zeckendorfs theorem from 1972, each integer can be uniquely represented as a sum of Fibonacci numbers such that no two of these are consecutive in the Fibonacci sequence.

The computation is simply the greedy algorithm of finding the highest Fibonacci number below n, subtracting it and iterating.

Value

List with components fibs the Fibonacci numbers that add sum up to n, and inds their indices in the Fibonacci sequence.

Examples

```
zeck(  10)  #=> 2 + 8 = 10
zeck( 100)  #=> 3 + 8 + 89 = 100
zeck(1000)  #=> 13 + 987 = 1000
```

Index