Package 'RcppAlgos'

September 15, 2018

Version 2.2.0

Title High Performance Tools for Combinatorics and Computational Mathematics

Description Provides optimized functions implemented in C++ with 'Rcpp' for solving problems in combinatorics and computational mathematics. There are combination/permutation functions that are both flexible as well as efficient with respect to speed and memory. Constraint parameters allow for generation of all combinations/permutations of a vector meeting specific criteria (e.g. finding all combinations such that the sum is less than a bound). Capable of generating specific combinations/permutations (e.g. retrieve only the nth lexicographical result) which sets up nicely for parallelization as well as random sampling. Gmp support permits exploration where the total number of results is large (e.g. comboSample(10000, 500, n = 4)). Additionally, there are several highly efficient number theoretic functions that are useful for problems common in computational mathematics. Some of these functions make use of the fast integer division library 'libdivide' by <http://ridiculousfish.com>. The primeSieve function is based on the segmented sieve of Eratosthenes implementation by Kim Walisch. It is capable of generating all primes less than a billion in just over 1 second. It can also quickly generate prime numbers over a range (e.g. primeSieve(10^13, 10^13+10^9)). Finally, there is a prime counting function that implements a simple variations of Legendre's formula based on the algorithm by Kim Walisch.

URL https://github.com/jwood000/RcppAlgos, https://gmplib.org/, http://primesieve.org/, https://github.com/kimwalisch/primesieve, https://github.com/kimwalisch/primecount, http://libdivide.com/

BugReports https://github.com/jwood000/RcppAlgos/issues

Imports gmp, Rcpp (>= 0.12.8)

LinkingTo Rcpp

Suggests testthat, numbers (>= 0.6-6), bigIntegerAlgos, microbenchmark

License GPL (>= 2)

LazyData TRUE

SystemRequirements gmp (>= 4.2.3)

NeedsCompilation yes

Author Joseph Wood

Maintainer Joseph Wood <jwood000@gmail.com>

Encoding UTF-8

RoxygenNote 6.0.1

Repository CRAN

Date/Publication 2018-09-14 22:22:23 UTC

R topics documented:

`RcppAlgos-package` *Tools for Combinatorics and Computational Mathematics*

Description

The **RcppAlgos** package attacks age-old problems in combinatorics and computational mathematics.

Goals

1. The main goal is to encourage fresh and creative approaches to foundational problems. The question that most appropriately summarizes `RcppAlgos` is: ***"Can we do better?"***.
2. Provide highly optimized functions that facilitates a broader spectrum of researchable cases. *E.g*
    - Investigating the structure of large numbers over wide ranges:

```
  – primeFactorizeSieve(10^13 - 10^7, 10^13 + 10^7)
  – primeSieve(2^53 - 10^10, 2^53 - 1)
```

- Easily explore combinations/permutations that would otherwise be inaccessible due to time of execution/memory constraints:

```
  – comboGeneral(primeSieve(1000), m = 5, repetition = TRUE,
                 constraintFun = "prod", comparisonFun = "<",
                 limitConstraints = 500000, upper = 10^5)
  – parallel::mclapply(seq(...), function(x) {
                temp = permuteGeneral(15, 10, lower = x, upper = y)
                ## analyze permutations
                ## output results
        }, mc.cores = detectCores() - 1))
  – permuteSample(rnorm(100), 10, freqs = rep(1:4, 25), n = 15, seed = 123)
```

3. *Speed!!!*.... You will find that the functions in RcppAlgos are some of the fastest of their type available in R.

Author(s)

Joseph Wood

---

comboCount *Number of combinations/permutations*

Description

Calculate the number of combinations/permutations of a vector chosen m at a time with or without replacement. Additionally, these functions can calculate the number of combinations/permutations of multisets.

Usage

```
comboCount(v, m = NULL, repetition = FALSE, freqs = NULL)

permuteCount(v, m = NULL, repetition = FALSE, freqs = NULL)
```

Arguments

v	Source vector. If v is an integer, it will be converted to the sequence 1:v.
m	Number of elements to choose. If repetition = TRUE, m can exceed the length of v. The default is NULL.
repetition	Logical value indicating whether combinations/permutations should be with or without repetition. The default is FALSE.
freqs	A vector of frequencies used for producing all combinations/permutations of a multiset of v. Each element of freqs represents how many times each element of the source vector, v, is repeated. It is analogous to the times argument in rep. The default value is NULL.

Value

A numerical value.

Note

When the number of results exceeds $2^{53} - 1$, a number of class `bigz` is returned.

See Also

`comboGeneral`, `permuteGeneral`

Examples

```
## Same interface as the respective "general" functions:
## i.e. comboGeneral & permuteGeneral

comboCount(25, 12)

permuteCount(15, 7, TRUE)

comboCount(25, 12, freqs = rep(2, 25))

## Return object of class 'bigz'
comboCount(250, 15, freqs = rep(2, 250))
```

`comboGeneral` *Generate all Combinations/Permutations of a Vector with/without Constraints*

Description

- Quickly generate all combinations or permutations of a vector, chosen m at a time, with or without constraints using Rcpp.
- Produce results in parallel using the arguments `lower` and `upper` along with the package parallel.
- GMP support allows for exploration of combinations/permutations of vectors with many elements.
- The output is in lexicographical order.

Usage

```
comboGeneral(v, m = NULL, repetition = FALSE, freqs = NULL,
             lower = NULL, upper = NULL, constraintFun = NULL,
              comparisonFun = NULL, limitConstraints = NULL,
               keepResults = NULL, FUN = NULL, Parallel = FALSE)

permuteGeneral(v, m = NULL, repetition = FALSE, freqs = NULL,
```

```
lower = NULL, upper = NULL, constraintFun = NULL,
  comparisonFun = NULL, limitConstraints = NULL,
    keepResults = NULL, FUN = NULL, Parallel = FALSE)
```

Arguments

v	Source vector. If v is an integer (including nonpositive integers), it will be converted to the sequence 1:v.
m	Number of elements to choose. If repetition = TRUE, m can exceed the length of v. The default is NULL.
repetition	Logical value indicating whether combinations/permutations should be with or without repetition. The default is FALSE.
freqs	A vector of frequencies used for producing all combinations/permutations of a multiset of v. Each element of freqs represents how many times each element of the source vector, v, is repeated. It is analogous to the times argument in rep. The default value is NULL.
lower	The lower bound. Combinations/permutations are generated lexicographically, thus utilizing this argument will determine which specific combination/permutation to start generating from (e.g. comboGeneral(5, 3, lower = 6) is equivalent to comboGeneral(5, 3)[6:choose(5, 3),]). This argument along with upper is very useful for quickly generating combinations/permutations in chunks allowing for easy parallelization.
upper	The upper bound. Similar to lower, however this parameter allows the user to *stop* generation at a specific combination/permutation (e.g. comboGeneral(5, 3, upper = 5) is equivalent to comboGeneral(5, 3)[1:5,]) If the output is constrained and lower isn't supplied, upper serves as a cap for how many results will be returned that meet the criteria (e.g. setting upper = 100 alone will return the first 100 results that meet the criteria, while setting lower = 1 and upper = 100 will test the first 100 results against the criteria). In addition to the benefits listed for lower, this parameter is useful when the total number of combinations/permutations without constraint is large and you expect/need a small number of combinations/permutations that meet a certain criteria. Using upper can drastically improve run time and avoid unnecessary crashes due to lack of memory. See examples below. N.B. This argument was formerly called rowCap.
constraintFun	Function to be applied to the elements of v that should be passed as a string (*E.g.* constraintFun = "sum"). The possible constraint functions are: "sum", "prod", "mean", "max", & "min". The default is NULL, meaning no function is applied.
comparisonFun	Comparison operator that will be used to compare limitConstraints with the result of constraintFun applied to v. It should be passed as a string or a vector of two strings (*E.g.* comparisonFun = "<=" or comparisonFun = c(">","<")). The possible comparison operators are: "<", ">", "<=", ">=", "==". The default is NULL. When comparisonFun is a vector of two comparison strings, say comparisonFun = c(comp1, comp2), and limitConstraints is a vector of two numerical values, say limitConstraints = c(x1, x2), the combinations/permutations will be filtered in one of the following two ways:

1. When `comp1` is one of the 'less than' operators (i.e. "<=" or "<"), `comp2` is one of the 'greater-than' operators (i.e. ">=" or ">"), and `x1 < x2`, the combinations/permutations that are returned will have a value (after `constraintFun` has been applied) between `x1` and `x2`.
2. When `comp1` and `comp2` are defined as in #1 and `x1 > x2`, the combinations/permutations that are returned will have a value outside the range of `x1` and `x2`. See the examples below.

`limitConstraints`
: This is the value(s) that will be used for comparison. Can be passed as a single value or a vector of two numerical values. The default is `NULL`. See the definition of `comparisonFun` as well as the examples below for more information.

`keepResults`
: A logical flag indicating if the result of `constraintFun` applied to `v` should be displayed; if `TRUE`, an additional column of results will be added to the resulting matrix. The default is `FALSE`. If user is only applying `constraintFun`, `keepResults` will default to `TRUE`.

  E.g. The following are equivalent and will produce a 4^{th} column of row sums:

  - `comboGeneral(5, 3 constraintFun = "sum", keepResults = TRUE)`
  - `comboGeneral(5, 3 constraintFun = "sum")`

`FUN`
: Function to be applied to each combination/permutation. The default is `NULL`.

`Parallel`
: Currently has no effect. Is only here for backwards compatibility. This feature will be available in later versions

Details

Finding all combinations/permutations with constraints is optimized by organizing them in such a way that when `constraintFun` is applied, a *partially* monotonic sequence is produced. Combinations/permutations are added successively, until a particular combination exceeds the given constraint value for a given constraint/comparison function combo. After this point, we can safely skip several combinations knowing that they will exceed the given constraint value.

When there are any negative values in `v` and `constraintFun = "prod"`, producing a monotonic set is non-trivial for the general case. As a result, performance will suffer as all combinations/permutations must be tested against the constraint criteria. Additionally, `upper` will not have its normal effectiveness (*i.e.* it will only limit the number of rows after producing all combinations/permutations).

Value

In general, a matrix is returned with each row containing a vector of length m or $m + 1$ depending on the value of `keepResults`. If `m` isn't supplied and `freqs` is given, a matrix is returned with each row containing a vector of length `sum(freqs)`. If `FUN` is utilized, a list is returned.

Note

- If either `constraintFun`, `comparisonFun` or `limitConstraints` is `NULL`, the constraint check will not be carried out. This is equivalent to simply finding all combinations/permutations of v choose m.

- The maximum number of combinations/permutations that can be generated at one time is $2^{31} - 1$. Utilizing `lower` and `upper` makes it possible to generate additional combinations/permutations.
- Factor vectors are accepted. Class and level attributes are preserved.
- Lexicographical ordering isn't guaranteed for permutations if `lower` isn't supplied and the output is constrained.
- If `lower` is supplied and the output is constrained, the combinations/permutations that will be tested will be in the lexicographical range `lower` to `upper` or to the total possible number of results if `upper` is not given. See the second paragraph for the definition of `upper`.
- `FUN` will be ignored if the constraint check is satisfied.

Author(s)

Joseph Wood

References

- Passing user-supplied C++ functions
- Monotonic Sequence
- Multiset
- Lexicographical order

Examples

```
system.time(comboGeneral(17, 8))
system.time(permuteGeneral(13, 6))

system.time(comboGeneral(13,10,repetition = TRUE))
system.time(permuteGeneral(factor(letters[1:9]),6,TRUE))

## permutations of the multiset (with or w/o setting m) :
## c(1,1,1,1,2,2,2,3,3,4,4,4,4,4)
system.time(permuteGeneral(4, freqs = c(4,3,2,5)))

permuteGeneral(4, m = 2, freqs = c(4,3,2,5))

## or combinations
comboGeneral(4, m = 2, freqs = c(4,3,2,5))


#### Examples using "upper" and "lower":

## Generate some random data
set.seed(1009)
mySamp = sort(rnorm(75, 997, 23))

permuteCount(75, 10, freqs = rep(1:3, 25))
# >Big Integer ('bigz') :
# >[1] 4606842576291495952
```

```
## See specific range of permutations
permuteGeneral(75, 10, freqs = rep(1:3, 25),
               lower = 1e12, upper = 1e12 + 10)

## Researcher only needs 1000 7-tuples of mySamp
## such that the sum is greater than 7200.
system.time(comboGeneral(mySamp, 7, FALSE, constraintFun = "sum",
             comparisonFun = ">", limitConstraints = 7200, upper = 1000))

## If you leave "upper" NULL in examples like the above,
## it can take much longer when the total number of
## possible combinations is large (still fast enough
## most of the time). In some cases, it can crash your
## computer as the underlying code allocates enough
## space to account for every combination (e.g. In our
## example, there are choose(75, 7) = 1984829850 rows,
## 7 columns, with each cell occupying 8 bytes. This
## gives a total over 100 GB (i.e. choose(75,7)*7*8/(2^30))

## Similarly, you can use "lower" to obtain the last rows.
## Generate the last 10 rows
system.time(comboGeneral(mySamp, 7, lower = choose(75, 7) - 9))

## Or if you would like to generate a specific chunk,
## use both "lower" and "upper". E.g. Generate one
## million combinations starting with the 900,000,001
## lexicographic combination.
t1 = comboGeneral(mySamp, 7, lower = 9*10^8 + 1,
                        upper = 9*10^8 + 10^6)

## class of the source vector is preserved
class(comboGeneral(5,3)[1,]) == class(1:5)
class(comboGeneral(c(1,2:5),3)[1,]) == class(c(1,2:5))
class(comboGeneral(factor(month.name),3)[1,]) == class(factor(month.name))

## Using keepResults will add a column of results
t2 = permuteGeneral(-3,6,TRUE,constraintFun = "prod",
                         keepResults = TRUE)

t3 = comboGeneral(-3,6,TRUE,constraintFun = "sum",
                        comparisonFun = "==",
                        limitConstraints = -8,
                        keepResults = TRUE)

## Using multiple constraints:

## Get combinations such that the product
## is between 3000 and 4000 inclusive
comboGeneral(5, 7, TRUE, constraintFun = "prod",
             comparisonFun = c(">=","<="),
             limitConstraints = c(3000, 4000),
             keepResults = TRUE)
```

```
## Or, get the combinations such that the
## product is less than or equal to 10 or
## greater than or equal to 40000
comboGeneral(5, 7, TRUE, constraintFun = "prod",
             comparisonFun = c("<=",">="),
             limitConstraints = c(10, 40000),
             keepResults = TRUE)

## Using FUN
comboGeneral(10000, 5, lower = 20, upper = 22,
             FUN = function(x) {
                 which(cummax(x) %% 2 == 1)
             })
# [[1]]
# [1] 1 3
#
# [[2]]
# [1] 1 3 5
#
# [[3]]
# [1] 1 3

## Not run:
## Parallel example generating more than 2^31 - 1 combinations.
library(parallel)
numCores = detectCores() - 1

system.time(mclapply(seq(1, comboCount(35, 15), 10086780), function(x) {
    a = comboGeneral(35, 15, lower = x, upper = x + 10086779)
    ## do something
    x
}, mc.cores = numCores))


## Find 13-tuple combinations of 1:25 such
## that the mean is less than 10
system.time(myComb <- comboGeneral(25, 13, FALSE,
                                   constraintFun = "mean",
                                   comparisonFun = "<",
                                   limitConstraints = 10))

## Alternatively, you must generate all combinations and subsequently
## subset to obtain the combinations that meet the criteria
system.time(myComb2 <- combn(25, 13))
system.time(myCols <- which(colMeans(myComb2) < 10))
system.time(myComb2 <- myComb2[, myCols])

## Any variation is much slower
system.time(myComb2 <- combn(25, 13)[,combn(25, 13, mean) < 10])

## Test equality with myComb above
all.equal(myComb, t(myComb2))
```

```
## Fun example... see stackoverflow:
## https://stackoverflow.com/q/22218640/4408538
system.time(permuteGeneral(seq(0L,100L,10L),8,TRUE,
                                constraintFun = "sum",
                                comparisonFun = "==",
                                limitConstraints = 100, upper = 10^5))

## End(Not run)
```

---

comboSample *Sample combinations/permutations*

Description

- Generate a specific (lexicographically) or random sample of combinations/permutations.
- GMP support allows for exploration of combinations/permutations of vectors with many elements.

Usage

```
comboSample(v, m = NULL, repetition = FALSE, freqs = NULL, n = NULL,
             sampleVec = NULL, seed = NULL, FUN = NULL, Parallel = FALSE)

permuteSample(v, m = NULL, repetition = FALSE, freqs = NULL, n = NULL,
               sampleVec = NULL, seed = NULL, FUN = NULL, Parallel = FALSE)
```

Arguments

v	Source vector. If v is an integer, it will be converted to the sequence 1:v.
m	Number of elements to choose. If repetition = TRUE, m can exceed the length of v. The default is NULL.
repetition	Logical value indicating whether combinations/permutations should be with or without repetition. The default is FALSE.
freqs	A vector of frequencies used for producing all combinations/permutations of a multiset of v. Each element of freqs represents how many times each element of the source vector, v, is repeated. It is analogous to the times argument in rep. The default value is NULL.
n	Number of combinations/permutations to return. The default is NULL.
sampleVec	A vector of numbers representing the lexicographical combination/permutations to return. Accepts vectors of class bigz as well as vectors of characters
seed	Random seed initialization. The default is NULL.
FUN	Function to be applied to each combination/permutation. The default is NULL.
Parallel	Currently has no effect. Is only here for backwards compatibility. This feature will be available in later versions

Details

These algorithms rely on efficiently generating the n^{th} lexicographical combination/permutation (sometimes called the rank).

Value

In general, a matrix is returned with each row containing a vector of length m. If `m` isn't supplied and `freqs` is given, a matrix is returned with each row containing a vector of length `sum(freqs)`. If `FUN` is utilized, a list is returned.

Note

- `n` and `sampleVec` cannot both be `NULL`.
- Factor vectors are accepted. Class and level attributes are preserved.

Author(s)

Joseph Wood

References

Lexicographical order

Examples

```
set.seed(11)
comboSample(30, 15, TRUE, n = 10)

set.seed(2)
permuteSample(15, 10, freqs = sample(1:2, 15, TRUE),
              sampleVec = c(1, 10^2, 10^5, 10^8, 10^11))

all.equal(comboSample(10, 5,
            sampleVec = 1:comboCount(10, 5)),
         comboGeneral(10, 5))

## Examples with enormous number of total permutations
num = permuteCount(10000, 20)
gmp::log2.bigz(num)
## [1] 265.7268

first = gmp::urand.bigz(n = 1, size = 265, seed = 123)
mySamp = do.call(c, lapply(0:10, function(x) gmp::add.bigz(first, x)))

class(mySamp)
## [1] "bigz"

## using permuteSample
pSamp = permuteSample(10000, 20, sampleVec = mySamp)

## using permuteGeneral
```

```
pGeneral = permuteGeneral(10000, 20,
                          lower = first,
                          upper = gmp::add.bigz(first, 10))

identical(pSamp, pGeneral)
## [1] TRUE
```

`divisorsRcpp` *Vectorized Factorization (Complete)*

Description

Rcpp implementation that quickly generates the complete factorization for many numbers.

Usage

```
divisorsRcpp(v = 100L, namedList = FALSE)
```

Arguments

`v`	Vector of integers or numeric values. Non-integral values will be coerced to whole numbers.
`namedList`	Logical flag. If `TRUE` and the `length(v) > 1`, a named list is returned. The default is `FALSE`.

Details

Highly efficient algorithm that builds on `primeFactorize` to quickly generate the complete factorization of many numbers.

Value

- Returns an unnamed vector if `length(v) == 1` regardless of the value of `namedList`. If $v < 2^{31}$, the class of the returned vector will be integer, otherwise the class will be numeric.
- If `length(v) > 1`, a named/unnamed list of vectors will be returned. If `max(bound1, bound2)` $< 2^{31}$, the class of each vector will be integer, otherwise the class will be numeric.

Note

The maximum value for each element in v is $2^{53} - 1$.

Author(s)

Joseph Wood

References

- Divisor
- 53-bit significand precision

See Also

divisors, primeFactorize

Examples

```
## Get the complete factorization of a single number
divisorsRcpp(10^8)

## Or get the complete factorization of many numbers
set.seed(29)
myVec <- sample(-1000000:1000000, 1000)
system.time(myFacs <- divisorsRcpp(myVec))

## Return named list
myFacsWithNames <- divisorsRcpp(myVec, namedList = TRUE)
```

---

divisorsSieve *Generate Complete Factorization for Numbers in a Range*

Description

Rcpp sieve implementation that quickly generates the complete factorization of all numbers between bound1 and bound2 (if supplied) or all numbers up to bound1.

Usage

```
divisorsSieve(bound1 = 100L, bound2 = NULL, namedList = FALSE)
```

Arguments

bound1	Positive integer or numeric value.
bound2	Positive integer or numeric value.
namedList	Logical flag. If TRUE, a named list is returned. The default is FALSE.

Details

This function is useful when many complete factorizations are needed. Instead of generating the complete factorization on the fly, one can reference the indices/names of the generated list.

This algorithm benefits greatly from the fast integer division library 'libdivide'. The following is from http://libdivide.com/:

- "*libdivide allows you to replace expensive integer divides with comparatively cheap multiplication and bitshifts. Compilers usually do this, but only when the divisor is known at compile time. libdivide allows you to take advantage of it at runtime. The result is that integer division can become faster - a lot faster.*"

Value

Returns a named/unnamed list of integer vectors if `max(bound1, bound2)` $< 2^{31}$, or a list of numeric vectors otherwise.

Note

The maximum value for either of the bounds is $2^{53} - 1$.

Author(s)

Joseph Wood

References

- Divisor
- ridiculousfish (author of libdivide)
- github.com/ridiculousfish/libdivide
- 53-bit significand precision

See Also

`divisorsRcpp`, `divisors`, `primeFactorizeSieve`

Examples

```
## Generate some random data
set.seed(33550336)
mySamp <- sample(10^5, 5*10^4)

## Quickly generate complete factorizations up
## to 10^5 (max element from mySamp)
system.time(allFacs <- divisorsSieve(10^5))

## Use generated complete factorization for further
## analysis by accessing the index of allFacs
for (s in mySamp) {
    myFac <- allFacs[[s]]
    ## Continue algorithm
}

## Generating complete factorizations over
## a range is efficient as well
system.time(divisorsSieve(10^12, 10^12 + 10^5))

## Set 'namedList' to TRUE to return a named list
divisorsSieve(27, 30, namedList = TRUE)
```

`eulerPhiSieve` *Apply Euler's Phi Function to Every Element in a Range*

Description

Rcpp sieve implementation that quickly generates the number of coprime elements for every number between `bound1` and `bound2` (if supplied) or all numbers up to `bound1`. This is equivalent to applying Euler's phi function (often written as $\phi(x)$) to every number in a given range.

Usage

```
eulerPhiSieve(bound1 = 100L, bound2 = NULL, namedVector = FALSE)
```

Arguments

`bound1`	Positive integer or numeric value.
`bound2`	Positive integer or numeric value.
`namedVector`	Logical flag. If `TRUE`, a named vector is returned. The default is `FALSE`.

Details

For the simple case (i.e. when `bound2 = NULL`), this algorithm first generates all primes up to n via the sieve of Eratosthenes. We use these primes to sieve over the sequence `1:n`, dividing each value by p, creating a temporary value that will be subtracted from the original value at each index (i.e. equivalent to multiply each index by $(1 - 1/p)$ but more efficient as we don't have to deal with floating point numbers). The case when `is.null(bound2) = FALSE` is more complicated but the basic ideas still hold.

This function is very useful when you need to calculate Euler's phi function for many numbers in a range as performing this calculation on the fly can be computationally expensive.

This algorithm benefits greatly from the fast integer division library 'libdivide'. The following is from http://libdivide.com/:

- "*libdivide allows you to replace expensive integer divides with comparatively cheap multiplication and bitshifts. Compilers usually do this, but only when the divisor is known at compile time. libdivide allows you to take advantage of it at runtime. The result is that integer division can become faster - a lot faster.*"

Value

Returns a named/unnamed integer vector if `max(bound1, bound2)` $< 2^{31}$, or a numeric vector otherwise.

Note

The maximum allowed value is $2^{53} - 1$.

Author(s)

Joseph Wood

References

- Euler's totient function
- ridiculousfish (author of libdivide)
- github.com/ridiculousfish/libdivide
- 53-bit significand precision

See Also

eulersPhi

Examples

```
## Generate some random data
set.seed(496)
mySamp <- sample(10^6, 5*10^5)

## Quickly generate number of coprime elements for many numbers
system.time(myPhis <- eulerPhiSieve(10^6))

## Now use result in algorithm
for (s in mySamp) {
    sPhi <- myPhis[s]
    ## Continue algorithm
}

## See https://projecteuler.net
system.time(which.max((1:10^6)/eulerPhiSieve(10^6)))

## Generating number of coprime elements
## for every number in a range is no problem
system.time(myPhiRange <- eulerPhiSieve(10^13, 10^13 + 10^6))

## Returning a named vector
eulerPhiSieve(10, 20, namedVector = TRUE)
eulerPhiSieve(10, namedVector = TRUE)
```

---

isPrimeRcpp *Vectorized Primality Test*

Description

Vectorized Rcpp implementation of the Miller-Rabin primality test. Based on the "mp_prime_p" function from the "factorize.c" source file found in the gmp library: http://gmplib.org.

Usage

```
isPrimeRcpp(v = 100L, namedVector = FALSE)
```

Arguments

v	Vector of integers or numeric values.
namedVector	Logical flag. If TRUE, a named vector is returned. The default is FALSE.

Details

The Miller-Rabin primality test is a probabilistic algorithm that makes heavy use of modular exponentiation. At the heart of modular exponentiation is the ability to accurately obtain the remainder of the product of two numbers $\pmod p$.

With the gmp library, producing accurate calculations for problems like this is trivial because of the nature of the multiple precision data type. However, standard C++ does not afford this luxury and simply relying on a strict translation would have limited this algorithm to numbers less than $\sqrt{2^{63}-1}$ (N.B. We are taking advantage of the signed 64-bit fixed width integer from the stdint library in C++. If we were confined to base R, the limit would have been $\sqrt{2^{53}-1}$). RcppAlgos::isPrimeRcpp gets around this limitation with a divide and conquer approach taking advantage of properties of arithmetic.

The problem we are trying to solve can be summarized as follows:

$$(x_1 * x_2) \pmod p$$

Now, we rewrite x_2 as $x_2 = y_1 + y_2 + \ldots + y_n$, so that we obtain:

$$(x_1 * y_1) \pmod p + (x_1 * y_2) \pmod p + \ldots + (x_1 * y_n) \pmod p$$

Where each product $(x_1 * y_j)$ for $j <= n$ is smaller than the original $x_1 * x_2$. With this approach, we are now capable of handling much larger numbers. Many details have been omitted for clarity.

For a more in depth examination of this topic see Accurate Modular Arithmetic with Double Precision.

Value

Returns a named/unnamed logical vector. If an index is TRUE, the number at that index is prime, otherwise the number is composite.

Note

The maximum value for each element in v is $2^{53} - 1$.

References

- THE MILLER-RABIN TEST
  - Conrad, Keith. "THE MILLER-RABIN TEST." http://www.math.uconn.edu/~kconrad/blurbs/ugradnumthy/miller
- 53-bit significand precision

See Also

primeFactorize, isprime, isPrime

Examples

```
## check the primality of a single number
isPrimeRcpp(100)

## check the primality of every number in a vector
isPrimeRcpp(1:100)

set.seed(42)
mySamp <- sample(10^13, 100)

## return named vector for easy identification
isPrimeRcpp(mySamp, namedVector = TRUE)
```

---

numDivisorSieve *Apply Divisor Function to Every Element in a Range*

Description

Rcpp sieve implementation that quickly generates the number of divisors for every number between bound1 and bound2 (if supplied) or all numbers up to bound1. This is equivalent to applying the divisor function (often written as $\sigma(x)$) to every number in a given range.

Usage

```
numDivisorSieve(bound1 = 100L, bound2 = NULL, namedVector = FALSE)
```

Arguments

bound1	Positive integer or numeric value.
bound2	Positive integer or numeric value.
namedVector	Logical flag. If TRUE, a named vector is returned. The default is FALSE.

Details

Simple and efficient sieve that quickly calculates the number of divisors for every number in a given range. This function is very useful when you need to calculate the number of divisors for many numbers.

This algorithm benefits greatly from the fast integer division library 'libdivide'. The following is from http://libdivide.com/:

- "*libdivide allows you to replace expensive integer divides with comparatively cheap multiplication and bitshifts. Compilers usually do this, but only when the divisor is known at compile time. libdivide allows you to take advantage of it at runtime. The result is that integer division can become faster - a lot faster.*"

Value

Returns a named/unnamed integer vector

Note

The maximum allowed value is $2^{53} - 1$.

Author(s)

Joseph Wood

References

- Divisor function
- ridiculousfish (author of libdivide)
- github.com/ridiculousfish/libdivide
- 53-bit significand precision

Examples

```
## Generate some random data
set.seed(8128)
mySamp <- sample(10^6, 5*10^5)

## Quickly generate number of divisors for
## every number less than a million
system.time(mySigmas <- numDivisorSieve(10^6))

## Now use result in algorithm
for (s in mySamp) {
    sSig <- mySigmas[s]
    ## Continue algorithm
}

## Generating number of divisors for every
## number in a range is no problem
system.time(sigmaRange <- numDivisorSieve(10^13, 10^13 + 10^6))

## Returning a named vector
numDivisorSieve(10, 20, namedVector = TRUE)
numDivisorSieve(10, namedVector = TRUE)
```

`primeCount` *Prime Counting Function* $\pi(x)$

Description

Rcpp implementation of Legendre's formula for counting prime numbers. It is based on the the algorithm developed by Kim Walisch found here: kimwalisch/primecount.

Usage

```
primeCount(n = 100L)
```

Arguments

`n`	Positive number

Details

Legendre's Formula for counting the number of primes less than n makes use of the inclusion-exclusion principle to avoid explicitly counting every prime up to n. It is given by:

$$\pi(x) = \pi(\sqrt{x}) + \Phi(x, \sqrt{x}) - 1$$

Where $\Phi(x, a)$ is the number of positive integers less than or equal to x that are relatively prime to the first a primes (i.e. not divisible by any of the first a primes). It is given by the recurrence relation (p_a is the ath prime (e.g. $p_4 = 7$)):

$$\Phi(x, a) = \Phi(x, a - 1) + \Phi(x/p_a, a - 1)$$

This algorithm implements five modifications developed by Kim Walisch for calculating $\Phi(x, a)$ efficiently.

1. Cache results of $\Phi(x, a)$
2. Calculate $\Phi(x, a)$ using $\Phi(x, a) = (x/pp) * \phi(pp) + \Phi(xmodpp, a)$ if $a <= 6$
   - $pp = 2 * 3 * ... *$ `prime[a]`
   - $\phi(pp) = (2 - 1) * (3 - 1) * ... *$ (`prime[a]` -1) (i.e. Euler's totient function)
3. Calculate $\Phi(x, a)$ using $\pi(x)$ lookup table
4. Calculate all $\Phi(x, a) = 1$ upfront
5. Stop recursion at 6 if $\sqrt{x} >= 13$ or $\pi(\sqrt{x})$ instead of 1

Value

Whole number representing the number of prime numbers less than or equal to n.

Note

The maximum value of n is $2^{53} - 1$

Author(s)

Joseph Wood

References

- Computing $\pi(x)$: the combinatorial method
  - Tomás Oliveira e Silva, Computing pi(x): the combinatorial method, Revista do DETUA, vol. 4, no. 6, March 2006, p. 761. http://sweet.ua.pt/tos/bib/5.4.pdf
- 53-bit significand precision

See Also

primeSieve

Examples

```
## Get the number of primes less than a billion
primeCount(10^9)
```

---

primeFactorize *Vectorized Prime Factorization*

Description

Rcpp implementation of Pollard's rho algorithm that quickly generates the prime factorization for many numbers. The algorithm is based on the "factorize.c" source file from the gmp library found here http://gmplib.org.

Usage

```
primeFactorize(v = 100L, namedList = FALSE)
```

Arguments

v	Vector of integers or numeric values. Non-integral values will be cured to whole numbers.
namedList	Logical flag. If TRUE and the length(v) > 1, a named list is returned. The default is FALSE.

Details

As noted in the Description section above, this algorithm is based on the "factorize.c" source code from the gmp library. Much of the code in RcppAlgos::primeFactorize is a straightforward translation from multiple precision C data types to standard C++ data types. A crucial part of the algorithm's efficiency is based on quickly determining primality, which is easily computed with gmp. However, with standard C++, this is quite challenging. Much of the research for RcppAlgos::primeFactorize was focused on developing an algorithm that could accurately and efficiently compute primality.

For more details, see the documentation for `isPrimeRcpp`.

Value

- Returns an unnamed vector if `length(v) == 1` regardless of the value of `namedList`. If $v < 2^{31}$, the class of the returned vector will be integer, otherwise the class will be numeric.
- If `length(v) > 1`, a named/unnamed list of vectors will be returned. If `max(bound1, bound2)` $< 2^{31}$, the class of each vector will be integer, otherwise the class will be numeric.

Note

The maximum value for each element in v is $2^{53} - 1$.

Author(s)

Joseph Wood

References

- Pollard's rho algorithm
- Miller-Rabin primality test
- Accurate Modular Arithmetic with Double Precision
- 53-bit significand precision

See Also

`primeFactorizeSieve`, `factorize`, `primeFactors`

Examples

```
## Get the prime factorization of a single number
primeFactorize(10^8)

## Or get the prime factorization of many numbers
set.seed(29)
myVec <- sample(-1000000:1000000, 1000)
system.time(pFacs <- primeFactorize(myVec))

## Return named list
pFacsWithNames <- primeFactorize(myVec, namedList = TRUE)
```

`primeFactorizeSieve` *Generate Prime Factorization for Numbers in a Range*

Description

Efficiently generates the prime factorization of all numbers between `bound1` and `bound2` (if supplied) or all numbers up to `bound1` using Rcpp.

Usage

```
primeFactorizeSieve(bound1 = 100L, bound2 = NULL, namedList = FALSE)
```

Arguments

`bound1`	Positive integer or numeric value.
`bound2`	Positive integer or numeric value.
`namedList`	Logical flag. If `TRUE`, a named list is returned. The default is `FALSE`.

Details

This function is useful when many prime factorizations are needed. Instead of generating the prime factorization on the fly, one can reference the indices/names of the generated list.

This algorithm benefits greatly from the fast integer division library 'libdivide'. The following is from http://libdivide.com/:

- "*libdivide allows you to replace expensive integer divides with comparatively cheap multiplication and bitshifts. Compilers usually do this, but only when the divisor is known at compile time. libdivide allows you to take advantage of it at runtime. The result is that integer division can become faster - a lot faster.*"

Value

Returns a named/unnamed list of integer vectors if `max(bound1, bound2)` $< 2^{31}$, or a list of numeric vectors otherwise.

Note

The maximum value for either of the bounds is $2^{53} - 1$.

Author(s)

Joseph Wood

References

- Prime Factor
- ridiculousfish (author of libdivide)
- github.com/ridiculousfish/libdivide
- 53-bit significand precision

See Also

primeFactorize, divisorsSieve, factorize, primeFactors

Examples

```
## Generate some random data
set.seed(28)
mySamp <- sample(10^5, 5*10^4)

## Quickly generate prime factorizations up
## to 10^5 (max element from mySamp)
system.time(allPFacs <- primeFactorizeSieve(10^5))

## Use generated prime factorization for further
## analysis by accessing the index of allPFacs
for (s in mySamp) {
    pFac <- allPFacs[[s]]
    ## Continue algorithm
}

## Generating prime factorizations over
## a range is efficient as well
system.time(primeFactorizeSieve(10^12, 10^12 + 10^5))

## Set 'namedList' to TRUE to return a named list
primeFactorizeSieve(27, 30, namedList = TRUE)
```

---

primeSieve *Generate Prime Integers*

Description

Rcpp implementation of the segmented sieve of Eratosthenes with wheel factorization. Generates all prime numbers between bound1 and bound2 (if supplied) or all primes up to bound1. This algorithm is capable of generating primes up to ten billion (i.e. 10^{10}) in just over 15 seconds.

The fundamental concepts of this algorithm are based off of the implementation by Kim Walisch found here: kimwalisch/primesieve.

Usage

```
primeSieve(bound1 = 100L, bound2 = NULL)
```

Arguments

bound1	Positive integer or numeric value.
bound2	Positive integer or numeric value.

Details

At the heart of this algorithm is the traditional sieve of Eratosthenes (i.e. given a prime p, mark all multiples of p as composite), however instead of sieving the entire interval, we only consider small sub-intervals. The benefits of this method are two fold:

1. Reduction of the space complexity from $O(n)$, for the traditional sieve, to $O(\sqrt{n}))$
2. Reduction of cache misses

The latter is of particular importance as cache memory is much more efficient and closer in proximity to the CPU than main memory. Reducing the size of the sieving interval allows for more effective utilization of the cache, which greatly impacts the overall efficiency.

Another optimization over the traditional sieve is the utilization of wheel factorization. With the traditional sieve of Eratosthenes, you typically check every odd index of your logical vector and if the value is true, you have found a prime. With wheel factorization using the first four primes (i.e. 2, 3, 5, and 7) to construct your wheel (i.e. 210 wheel), you only have to check indices of your logical vector that are coprime to 210 (i.e. the product of the first four primes). As an example, with $n = 10000$ and a 210 wheel, you only have to check 2285 indices vs. 5000 with the classical implementation.

Value

Returns an integer vector if `max(bound1, bound2)` $< 2^{31}$, or a numeric vector otherwise.

Note

- It does not matter which bound is larger as the resulting primes will be between `min(bound1, bound2)` and `max(bound1, bound2)` if `bound2` is provided.
- The maximum value for either of the bounds is $2^{53} - 1$.

Author(s)

Joseph Wood

References

- primesieve (Fast C/C++ prime number generator)
- Sieve of Eratosthenes
- Wheel factorization
- 53-bit significand precision

See Also

`Primes`

Examples

```
## Primes up to a thousand
primeSieve(1000)

## Primes between 42 and 1729
primeSieve(42, 1729)

## Equivalent to
primeSieve(1729, 42)

## Primes up to one hundred million in no time
system.time(primeSieve(10^8))

options(scipen = 50)
## Quickly generate large primes over interval
system.time(myPs <- primeSieve(10^13+10^6, 10^13))
## Object created is small
object.size(myPs)
```

Index