Package 'RcppAlgos'

October 18, 2026

Version 2.10.2

Title High Performance Tools for Combinatorics and Computational Mathematics

Description Provides optimized functions and flexible iterators implemented in C++ for solving problems in combinatorics and computational mathematics. Handles various combinatorial objects including combinations, permutations, integer partitions and compositions, Cartesian products, unordered Cartesian products, and partition of groups. Utilizes the RMatrix class from 'RcppParallel' for thread safety. The combination and permutation functions contain constraint parameters that allow for generation of all results of a vector meeting specific criteria (e.g. finding all combinations such that the sum is between two bounds). Capable of ranking/unranking combinatorial objects efficiently (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 high performance 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'. The primeSieve function is based on the segmented sieve of Eratosthenes implementation by Kim Walisch. It is also efficient for large numbers by using the cache friendly improvements originally developed by Tomás Oliveira. Finally, there is a prime counting function that implements Legendre's formula based on the work of Kim Walisch.

URL https://jwood000.github.io/RcppAlgos/, https://github.com/jwood000/RcppAlgos, https://gmplib.org/, https://github.com/kimwalisch/primesieve, https://libdivide.com, https://github.com/kimwalisch/primecount, https://ridiculousfish.com/, https://sweet.ua.pt/tos/software/prime_sieve.html

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

LinkingTo cpp11

Imports gmp, methods

Suggests testthat, partitions, microbenchmark, knitr, RcppBigIntAlgos, rmarkdown, prettydoc, covr, xml2

Config/Needs/website pkgdown

License GPL (>= 2)

SystemRequirements gmp (>= 4.2.3)

VignetteBuilder knitr

NeedsCompilation yes

Encoding UTF-8

RoxygenNote 7.2.3

Author Joseph Wood [aut, cre], Joseph J. Allaire [ctb] (Original author of RMatrix.h, copied from RcppParallel)

Maintainer Joseph Wood <jwood000@gmail.com>

Repository CRAN

Date/Publication 2026-10-17 23:10:08 UTC

Contents

RcppAlgos-package . . . 3
Cartesian-class . . . 4
Combo-class . . . 5
comboCount . . . 6
comboGeneral . . . 7
comboGrid . . . 14
comboGroups . . . 16
ComboGroups-class . . . 19
comboGroupsCount . . . 20
comboGroupsIter . . . 21
comboGroupsSample . . . 23
comboIter . . . 25
comboRank . . . 30
comboSample . . . 32
Constraints-class . . . 35
divisorsRcpp . . . 36
divisorsSieve . . . 38
eulerPhiSieve . . . 39
expandGrid . . . 41
expandGridCount . . . 43
expandGridIter . . . 44
expandGridSample . . . 45
isPrimeRcpp . . . 47
numDivisorSieve . . . 49

Partitions-class 50
partitionsCount 51
partitionsGeneral 53
partitionsIter 55
partitionsRank 58
partitionsSample 60
primeCount 62
primeFactorize 64
primeFactorizeSieve 65
primeSieve 67
stdThreadMax 69

Index **70**

`RcppAlgos-package` *High Performance 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, nThreads = 32)`
   - Easily explore combinations/permutations/partitions that would otherwise be inaccessible due to time of execution/memory constraints:
     - ```
       c_iter = comboIter(10000, 100)
       bigSamp = gmp::urand.bigz(3, gmp::log2.bigz(comboCount(10000, 100)))
       c_iter[[bigSamp]] ## flexible iterator allows random sampling
       ```
     - ```
       p_iter = partitionsIter(5000, 100, target = 6000)
       p_iter[[1e9]] ## start iterating from index = 1e9
       p_iter@nextIter()
       p_iter@nextNIter(1e3)
       ```
     - `comboGeneral(150, 5, constraintFun = "sum", Parallel = TRUE)`
     - ```
       parallel::mclapply(seq(...), function(x) {
                   temp = permuteGeneral(15, 10, lower = x, upper = y)
                   ## analyze permutations
                   ## output results
           }, mc.cores = detectCores() - 1))
       ```
       ```

```
– partitionsGeneral(0:80, repetition = TRUE)
– permuteSample(rnorm(100), 10, freqs = rep(1:4, 25), n = 15, seed = 123)
– set.seed(123)
  comboGeneral(runif(42, 0, 50), 10,
               constraintFun = "mean",
               comparisonFun = c(">","<"),
               limitConstraints = c(39.876, 42.123))
```

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

`Cartesian-class` *S4-class for Exposing C++ Cartesian Class*

Description

The `Cartesian` class is an S4-class that exposes C++ classes that provide access to iterators and other useful methods.

Slots

`nextIter` Retrieve the **next** lexicographical result

`nextNIter` Pass an integer *n* to retrieve the **next** *n* lexicographical results

`nextRemaining` Retrieve all remaining lexicographical results

`currIter` Returns the current iteration

`startOver` Resets the iterator

`sourceVector` View the source vector

`summary` Returns a list of summary information about the iterator

`front` Retrieve the **first** lexicographical result

`back` Retrieve the **last** lexicographical result

`randomAccess` Random access method. Pass a single value or a vector of valid indices. If a single value is passed, the internal index of the iterator will be updated, however if a vector is passed the internal state will not change. GMP support allows for flexible indexing.

Author(s)

Joseph Wood

See Also

`ComboGroups-class`, `Combo-class`, `Partitions-class`

Examples

```
showClass("Cartesian")
```

`Combo-class` *S4-classes for Exposing C++ Combinatorial Classes*

Description

The `Combo` class family are S4-classes that expose C++ classes that provide access to iterators and other useful methods.

Slots

of `"Combo"` and all classes inheriting from it:

`nextIter` Retrieve the **next** lexicographical result

`nextNIter` Pass an integer *n* to retrieve the **next** *n* lexicographical results

`nextRemaining` Retrieve all remaining lexicographical results

`currIter` Returns the current iteration

`prevIter` Retrieve the **previous** lexicographical result (the **next** *reverse* lexicographical result)

`prevNIter` Pass an integer *n* to retrieve the **previous** *n* lexicographical results (the **next** *n reverse* lexicographical results)

`prevRemaining` Retrieve all remaining *reverse* lexicographical results

`startOver` Resets the iterator

`sourceVector` View the source vector

`summary` Returns a list of summary information about the iterator

`front` Retrieve the **first** lexicographical result

`back` Retrieve the **last** lexicographical result

`randomAccess` Random access method. Pass a single value or a vector of valid indices. If a single value is passed, the internal index of the iterator will be updated, however if a vector is passed the internal state will not change. GMP support allows for flexible indexing.

Author(s)

Joseph Wood

See Also

`Partitions-class`, `Constraints-class`

Examples

```
showClass("Combo")
```

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, ...)
permuteCount(v, m = NULL, ...)

## Default S3 method:
comboCount(v, m = NULL, repetition = FALSE, freqs = NULL, ...)
## Default S3 method:
permuteCount(v, m = NULL, repetition = FALSE, freqs = NULL, ...)

## S3 method for class 'table'
comboCount(v, m = NULL, ...)
## S3 method for class 'table'
permuteCount(v, m = NULL, ...)
```

Arguments

v	Source vector. If v is a positive integer, it will be converted to the sequence 1:v. If v is a negative integer, it will be converted to the sequence v:-1. All atomic types are supported (See is.atomic).
m	Number of elements to choose. If repetition = TRUE or freqs is utilized, m can exceed the length of v. If m = NULL, the length will default to length(v) or sum(freqs).
...	Further arguments passed to methods.
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 representing the total number of combinations/permutations.

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

permuteCount(-5)
permuteCount(5)
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 Combinations and Permutations of a Vector with/without Constraints*

Description

- Generate combinations or permutations of a vector with or without constraints.
- Produce results in parallel using the `Parallel` or `nThreads` arguments. You can also apply each of the five compiled functions given by the argument `constraintFun` in parallel.
- The arguments `lower` and `upper` make it possible to generate combinations/permutations in chunks allowing for parallelization via the parallel-package. This is convenient when you want to apply a custom function to the output in parallel.
- Attack integer partition and general subset sum problems.
- GMP support allows for exploration of combinations/permutations of vectors with many elements.
- The output is in lexicographical order.

Usage

```
comboGeneral(v, m = NULL, ...)
permuteGeneral(v, m = NULL, ...)

## S3 method for class 'numeric'
comboGeneral(v, m = NULL, repetition = FALSE, freqs = NULL,
             lower = NULL, upper = NULL, constraintFun = NULL,
             comparisonFun = NULL, limitConstraints = NULL,
             keepResults = NULL, FUN = NULL, Parallel = FALSE,
             nThreads = NULL, tolerance = NULL, FUN.VALUE = NULL, ...)
```

```
## S3 method for class 'numeric'
permuteGeneral(v, m = NULL, repetition = FALSE, freqs = NULL,
               lower = NULL, upper = NULL, constraintFun = NULL,
               comparisonFun = NULL, limitConstraints = NULL,
               keepResults = NULL, FUN = NULL, Parallel = FALSE,
               nThreads = NULL, tolerance = NULL, FUN.VALUE = NULL, ...)

## S3 method for class 'factor'
comboGeneral(
    v, m = NULL, repetition = FALSE, freqs = NULL, lower = NULL, upper = NULL,
    FUN = NULL, Parallel = FALSE, nThreads = NULL, FUN.VALUE = NULL, ...
)
## S3 method for class 'factor'
permuteGeneral(
    v, m = NULL, repetition = FALSE, freqs = NULL, lower = NULL, upper = NULL,
    FUN = NULL, Parallel = FALSE, nThreads = NULL, FUN.VALUE = NULL, ...
)

## Default S3 method:
comboGeneral(v, m = NULL, repetition = FALSE,
             freqs = NULL, lower = NULL, upper = NULL,
             FUN = NULL, FUN.VALUE = NULL, ...)
## Default S3 method:
permuteGeneral(v, m = NULL, repetition = FALSE,
               freqs = NULL, lower = NULL, upper = NULL,
               FUN = NULL, FUN.VALUE = NULL, ...)

## S3 method for class 'table'
comboGeneral(
    v, m = NULL, lower = NULL, upper = NULL, constraintFun = NULL,
    comparisonFun = NULL, limitConstraints = NULL, keepResults = NULL,
    FUN = NULL, Parallel = FALSE, nThreads = NULL, tolerance = NULL,
    FUN.VALUE = NULL, ...
)
## S3 method for class 'table'
permuteGeneral(
    v, m = NULL, lower = NULL, upper = NULL, constraintFun = NULL,
    comparisonFun = NULL, limitConstraints = NULL, keepResults = NULL,
    FUN = NULL, Parallel = FALSE, nThreads = NULL, tolerance = NULL,
    FUN.VALUE = NULL, ...
)

## S3 method for class 'list'
comboGeneral(v, m = NULL, repetition = FALSE,
             freqs = NULL, lower = NULL, upper = NULL, ...)
## S3 method for class 'list'
permuteGeneral(v, m = NULL, repetition = FALSE,
               freqs = NULL, lower = NULL, upper = NULL, ...)
```

Arguments

v	Source vector. If v is a positive integer, it will be converted to the sequence 1:v. If v is a negative integer, it will be converted to the sequence v:-1. All atomic types are supported (See is.atomic).
m	Number of elements to choose. If repetition = TRUE or freqs is utilized, m can exceed the length of v. If m = NULL, the length will default to length(v) or sum(freqs).
...	Further arguments passed to methods.
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 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 improve run time if used judiciously as we call the member function reserve of std::vector. See examples below.
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, *e.g* comparisonFun = c(comp1, comp2), and limitConstraints is a vector of two numerical values, *e.g* limitConstraints = c(x1, x2), the combinations/permutations will be filtered in one of the following two ways:

1. When comp1 is one of the 'greater-than' operators (*i.e.* ">=" or ">"), comp2 is one of the 'less-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.

In other words, the first comparison operator is applied to the first limit and the second operator is applied to the second limit.

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 — Logical value indicating whether combinations/permutations should be generated in parallel using $n-1$ threads, where n is the maximum number of threads. The default is FALSE. If nThreads is not NULL, it will be given preference (*e.g.* if user has 8 threads with Parallel = TRUE and nThreads = 4, only 4 threads will be spawned). If your system is single-threaded, the arguments Parallel and nThreads are ignored.

nThreads — Specific number of threads to be used. The default is NULL. See Parallel.

tolerance — A numeric value greater than or equal to zero. This parameter is utilized when a constraint is applied on a numeric vector. The default value is 0 when it can be determined that whole values are being utilized, otherwise it is sqrt(.Machine$double.eps) which is approximately $1.5e-8$. N.B. If the input vector is of type integer, this parameter will be ignored and strict equality will be enforced.

FUN.VALUE — A template for the return value from FUN. See 'Details' of vapply for more information.

Details

For the general case, 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.

There are special cases where more efficient algorithms are dyncamically deployed. These cases center around the subject of integer partitions. See partitionsGeneral for more information.

When there are any negative values in v, constraintFun = "prod", and limitConstraints is provided, 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. Throughout the documentation, this case is referred to as *special constraints*.

Value

- In general, a matrix with m or $m + 1$ columns, depending on the value of keepResults
- If FUN is utilized and FUN.VALUE = NULL, a list is returned
- When both FUN and FUN.VALUE are not NULL, the return is modeled after the return of vapply. See the 'Value' section of vapply.

Note

- Parallel and nThreads will be ignored in the following cases:
  - When the output is constrained, except for most partition cases and special constraints when lower is explicitly supplied (see the last note below).
  - If the class of the vector passed is character, raw, and complex (N.B. Rcpp::CharacterMatrix is not thread safe). Alternatively, you can generate an indexing matrix in parallel.
  - If FUN is utilized.
- If either constraintFun, comparisonFun or limitConstraints is NULL –or– if the class of the vector passed is logical, character, raw, factor, or complex, 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 except when FUN is used.
- 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 up 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.
- For special constraints (see the Details section), multithreading is only available when lower is explicitly supplied. In this case, the search space can be divided among threads beginning at the requested lower bound. When lower is not supplied, RcppAlgos must continue searching until the requested number of results satisfying the constraints is found. Because the locations of these results are not known in advance, this search is performed serially, even when multiple threads are requested.

Author(s)

Joseph Wood

References

- Passing user-supplied C++ functions
- Monotonic Sequence
- Multiset
- Lexicographical Order
- Subset Sum Problem
- Partition (Number Theory)

Examples

```
comboGeneral(4, 3)
permuteGeneral(3)

permuteGeneral(factor(letters[1:3]), repetition = TRUE)

## permutations of the multiset :
## c(1,1,1,2,2,3)
permuteGeneral(table(c(1,1,1,2,2,3)))

## Example with list
comboGeneral(
    v = list(
        p1 = matrix(1:10, ncol = 2),
        p2 = data.frame(a = letters, b = 1:26),
        p3 = as.complex(1:10)
    ),
    m = 2
)

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

## Researcher only needs 10 7-tuples of mySamp
## such that the sum is greater than 7200.
## Generate some random data
set.seed(1009)
mySamp = rnorm(75, 997, 23)
comboGeneral(mySamp, 7, constraintFun = "sum",
             comparisonFun = ">", limitConstraints = 7200, upper = 10)

## Similarly, you can use "lower" to obtain the last rows.
## Generate the last 10 rows
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
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)

#### General subset sum problem
set.seed(516781810)
comboGeneral(runif(100, 0, 42), 5, constraintFun = "mean",
             comparisonFun = "==", limitConstraints = 30,
             tolerance = 0.0000002)


#### Integer Partitions
comboGeneral(0:5, 5, TRUE, constraintFun = "sum",
             comparisonFun = "==", limitConstraints = 5)


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

## Not run:
## Parallel example generating more than 2^31 - 1 combinations.
```

```
library(parallel)
numCores = detectCores() - 1

## 10086780 evenly divides choose(35, 15) and is "small enough" to
## generate quickly in chunks.
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))

## These are called weak integer compositions. Below, we call
## compositionsGeneral which gives the same output except it
## in lexicographical order. See 'Note' above
system.time(compositionsGeneral(seq(0L,100L,10L), 8, TRUE, weak = TRUE))

## End(Not run)
```

---

comboGrid *Unordered Cartesian Product*

Description

Efficient version of `expand.grid` where order does not matter. This is a combinatorial variant where groups of elements are treated as equivalent regardless of order. For example, given: `{1,`

2}, {1, 2}, the unordered Cartesian product is {1, 1}, {1, 2}, {2, 2}. It is loosely equivalent to the following:

- t = expand.grid(lst)
- t = t[do.call(order, t),]
- key = apply(t, 1, function(x) paste0(sort(x), collapse = ""))
- t[!duplicated(key),]

Usage

```
comboGrid(..., repetition = TRUE, return_df = FALSE)
```

Arguments

...	vectors, factors or a list containing these. (See ?expand.grid).
repetition	Logical value indicating whether results should be with or without repetition. The default is TRUE.
return_df	Logical flag to force the output to be a data.frame. The default is FALSE.

Value

When all of the input is of the same type, by default comboGrid produce a matrix (a data.frame otherwise). This can be ignored by setting the argument return_df = TRUE.

Author(s)

Joseph Wood

See Also

expandGrid

Examples

```
## description example
lst = list(1:2, 1:2)

t = expand.grid(lst)
t = t[do.call(order, t), ]
key = apply(t, 1, function(x) paste0(sort(x), collapse = ""))
t[!duplicated(key), ]

## vs using comboGrid. N.B. Output is a matrix
comboGrid(lst)

## Force a data.frame to be returned
comboGrid(lst, return_df = TRUE)

## Input vectors are of different type, so a data.frame is returned
expGridNoOrder = comboGrid(1:5, 3:9, letters[1:5], letters[c(1,4,5,8)])
```

```
head(expGridNoOrder)
tail(expGridNoOrder)

expGridNoOrderNoRep = comboGrid(1:5, 3:9, letters[1:5],
                                letters[c(1,4,5,8)], repetition = FALSE)

head(expGridNoOrderNoRep)
tail(expGridNoOrderNoRep)
```

---

comboGroups *Partition a Vector into Groups*

Description

- Generate partitions of a vector into groups. See Create Combinations in R by Groups on https://stackoverflow.com for a direct use case of when the groups sizes are equal.
- Produce results in parallel using the `Parallel` or `nThreads` arguments.
- GMP support allows for exploration where the number of results is large.
- The output is in lexicographical order by groups.

Usage

```
comboGroups(v, numGroups = NULL, grpSizes = NULL,
            retType = "matrix", lower = NULL, upper = NULL,
            Parallel = FALSE, nThreads = NULL)
```

Arguments

`v`	Source vector. If `v` is a positive integer, it will be converted to the sequence `1:v`. If `v` is a negative integer, it will be converted to the sequence `v:-1`. All atomic types are supported (See `is.atomic`).
`numGroups`	An Integer. The number of groups that the vector will be partitioned into. The default is `NULL`. If provided and `grpSize` is `NULL`, it must divide the length of `v` (if `v` is a vector) or `v` (if `v` is a scalar).
`grpSizes`	A vector of whole numbers representing the size of each group. The default is `NULL`. If provided, the sum of the elements must total the length of `v` (if `v` is a vector) or `v` (if `v` is a scalar).
`retType`	A string, "3Darray" or "matrix", that determines the shape of the output. The default is "matrix". Note, "3Darray" can only be used when the size of each group is uniform. When the size of each group varies, the return output will always be a matrix.
`lower`	The lower bound. Partitions of groups are generated lexicographically, thus utilizing this argument will determine which specific result to start generating from (*e.g.* `comboGroups(8, 2, lower = 30)` is equivalent to `comboGroups(8, 2)[30:comboGroupsCount(8, 2), ]`). This argument along with `upper` is very useful for generating results 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 result (*e.g.* `comboGroups(8, 2, upper = 5)` is equivalent to `comboGroups(8, 2)[1:5, ]`)

`Parallel` Logical value indicating whether results should be generated in parallel using $n-1$ threads, where n is the maximum number of threads. The default is `FALSE`. If `nThreads` is not `NULL`, it will be given preference (*e.g.* if user has 8 threads with `Parallel = TRUE` and `nThreads = 4`, only 4 threads will be spawned). If your system is single-threaded, the arguments `Parallel` and `nThreads` are ignored.

`nThreads` Specific number of threads to be used. The default is `NULL`. See `Parallel`.

Details

Conceptually, this problem can be viewed as generating all permutations of the vector `v` and removing the within group permutations. To illustrate this, let us consider the case of generating partitions of `1:8` into 2 groups each of size 4.

- To begin, generate the permutations of `1:8` and group the first/last four elements of each row.

			Grp1			Grp2		
	C1	C2	C3	C4	C5	C6	C7	C8
R1	| 1	2	3	4 |	| 5	6	7	8 |
R2	| 1	2	3	4 |	| 5	6	8	7 |
R3	| 1	2	3	4 |	| 5	7	6	8 |
R4	| 1	2	3	4 |	| 5	7	8	6 |
R5	| 1	2	3	4 |	| 5	8	6	7 |
R6	| 1	2	3	4 |	| 5	8	7	6 |

- Note that the permutations above are equivalent partitions of 2 groups of size 4 as only the last four elements are permuted. If we look at at the 25^{th} lexicographical permutation, we observe our second distinct partition.

			Grp1			Grp2		
	C1	C2	C3	C4	C5	C6	C7	C8
R24	| 1	2	3	4 |	| 8	7	6	5 |
R25	**| 1**	**2**	**3**	**5 |**	**| 4**	**6**	**7**	**8 |**
R26	| 1	2	3	5 |	| 4	6	8	7 |
R27	| 1	2	3	5 |	| 4	7	6	8 |
R28	| 1	2	3	5 |	| 4	7	8	6 |

- Continuing on, we will reach the $3,457^{th}$ lexicographical permutation, which represents the last result:

			Grp1			Grp2		
	C1	C2	C3	C4	C5	C6	C7	C8
R3454	| 1	6	7	5 |	|8	3	4	2 |
R3455	| 1	6	7	5 |	|8	4	2	3 |
R3456	| 1	6	7	5 |	|8	4	3	2 |

R3457	**| 1**	**6**	**7**	**8 |**	**| 2**	**3**	**4**	**5 |**
R3458	| 1	6	7	8 |	|2	3	5	4 |

- For this small example, the method above will not be that computationally expensive. In fact, there are only 35 total partitions of `1:8` into 2 groups of size 4 out of a possible `factorial(8) = 40320` permutations. However, just doubling the size of the vector will make this approach infeasible as there are over 10 trillion permutations of `1:16`.
- The algorithm in `comboGroups` avoids these duplicate partitions of groups by utilizing an efficient algorithm analogous to the std::next_permutation found in the standard algorithm library in C++.

Value

By default, a matrix is returned with column names corresponding to the associated group. If `retType = "3Darray"`, a named 3D array is returned.

Note

- The maximum number of partitions of groups that can be generated at one time is $2^{31} - 1$. Utilizing `lower` and `upper` makes it possible to generate additional combinations/permutations.
- The length of `grpSizes` must equal `numGroups` if both `grpSize` and `numGroups` are provided.

Author(s)

Joseph Wood

Examples

```
## return a matrix
comboGroups(8, 2)

## or a 3 dimensional array
temp = comboGroups(8, 2, retType = "3Darray")

## view the first partition
temp[1, , ]

## Example with groups of varying size
comboGroups(8, grpSizes = c(3, 5))

total = comboGroupsCount(11, grpSizes = c(3, 3, 5))

## Start generating from particular index
comboGroups(11, grpSizes = c(3, 3, 5), lower = total - 20)
```

ComboGroups-class *S4-class for Exposing C++ ComboGroups Class*

Description

The ComboGroups class is an S4-class that exposes C++ classes that provide access to iterators and other useful methods.

Slots

`nextIter` Retrieve the **next** lexicographical result

`nextNIter` Pass an integer *n* to retrieve the **next** *n* lexicographical results

`nextRemaining` Retrieve all remaining lexicographical results

`currIter` Returns the current iteration

`startOver` Resets the iterator

`sourceVector` View the source vector

`summary` Returns a list of summary information about the iterator

`front` Retrieve the **first** lexicographical result

`back` Retrieve the **last** lexicographical result

`randomAccess` Random access method. Pass a single value or a vector of valid indices. If a single value is passed, the internal index of the iterator will be updated, however if a vector is passed the internal state will not change. GMP support allows for flexible indexing.

Author(s)

Joseph Wood

See Also

Combo-class, Constraints-class, Partitions-class

Examples

```
showClass("ComboGroups")
```

comboGroupsCount *Number of Partitions of a Vector into Groups*

Description

Calculate the number of partitions of a vector into groups. See the related integer sequences A025035-A025042 at OEIS (E.g. A025036 for Number of partitions of $1, 2, ..., 4n$ into sets of size 4.)

Usage

```
comboGroupsCount(v, numGroups = NULL, grpSizes = NULL)
```

Arguments

v	Source vector. If v is a positive integer, it will be converted to the sequence 1:v. If v is a negative integer, it will be converted to the sequence v:-1. All atomic types are supported (See is.atomic).
numGroups	An Integer. The number of groups that the vector will be partitioned into. The default is NULL. If provided and grpSize is NULL, it must divide the length of v (if v is a vector) or v (if v is a scalar).
grpSizes	A vector of whole numbers representing the size of each group. The default is NULL. If provided, the sum of the elements must total the length of v (if v is a vector) or v (if v is a scalar).

Value

A numerical value representing the total number of partitions of groups.

Note

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

Author(s)

Joseph Wood

References

OEIS Integer Sequence A025036

Examples

```
comboGroupsCount(16, 4)
comboGroupsCount(16, grpSizes = c(1:4, 6))
comboGroupsCount(28, grpSizes = rep(2:5, each = 2))
```

comboGroupsIter *comboGroups Iterator*

Description

- Returns an iterator for iterating over partitions of a vector into groups.
- Supports random access via the `[[` method.
- GMP support allows for exploration of cases where the number of comboGroups is large.
- Use the `next` methods to obtain results in lexicographical order.

Usage

```
comboGroupsIter(v, numGroups = NULL, grpSizes = NULL,
                retType = "matrix", Parallel = FALSE,
                nThreads = NULL)
```

Arguments

`v`	Source vector. If `v` is a positive integer, it will be converted to the sequence `1:v`. If `v` is a negative integer, it will be converted to the sequence `v:-1`. All atomic types are supported (See `is.atomic`).
`numGroups`	An Integer. The number of groups that the vector will be partitioned into. The default is `NULL`. If provided and `grpSize` is `NULL`, it must divide the length of v (if v is a vector) or v (if v is a scalar).
`grpSizes`	A vector of whole numbers representing the size of each group. The default is `NULL`. If provided, the sum of the elements must total the length of v (if v is a vector) or v (if v is a scalar).
`retType`	A string, "3Darray" or "matrix", that determines the shape of the output. The default is "matrix". Note, "3Darray" can only be used when the size of each group is uniform. When the size of each group varies, the return output will always be a matrix.
`Parallel`	Logical value indicating whether results should be generated in parallel using $n-1$ threads, where n is the maximum number of threads. The default is `FALSE`. If `nThreads` is not `NULL`, it will be given preference (*e.g.* if user has 8 threads with `Parallel = TRUE` and `nThreads = 4`, only 4 threads will be spawned). If your system is single-threaded, the arguments `Parallel` and `nThreads` are ignored.
`nThreads`	Specific number of threads to be used. The default is `NULL`. See `Parallel`.

Details

Once you initialize a new iterator, the following methods are available:

`nextIter` Retrieve the **next** lexicographical result

`nextNIter` Pass an integer n to retrieve the **next** n lexicographical results

`nextRemaining` Retrieve all remaining lexicographical results

`currIter` Returns the current iteration

`startOver` Resets the iterator

`sourceVector` View the source vector

`summary` Returns a list of summary information about the iterator

`front` Retrieve the **first** lexicographical result

`back` Retrieve the **last** lexicographical result

`[[` Random access method. Pass a single value or a vector of valid indices. If a single value is passed, the internal index of the iterator will be updated, however if a vector is passed the internal state will not change. GMP support allows for flexible indexing.

Value

- If `nextIter` is called, a named vector is returned if `retType = "matrix"`. If `retType = "3Darray"`, a named matrix is returned.
- Otherwise a named matrix is returned when `retType = "matrix"` and a named 3D array is returned when `retType = "3Darray"`.

Note

- If `nThreads` is utilized, it will only take effect if the number of elements requested is greater than some threshold (determined internally). *E.g*:

```
serial   <- comboGroupsIter(50, 10)
multi    <- comboGroupsIter(50, 10, nThreads = 4)
fetch1e6 <- multi@nextNIter(1e6)  ## much faster than serial@nextNIter(1e6)
fetch1e3 <- multi@nextNIter(1e3)  ## only one thread used... same as serial@nextNIter(1e3)

library(microbenchmark)
microbenchmark(multi@nextNIter(1e6), serial@nextNIter(1e6), times = 20)
microbenchmark(multi@nextNIter(1e3), serial@nextNIter(1e3), times = 20)
```

- The maximum number of comboGroups that can be generated at one time is $2^{31} - 1$.
- If an error occurs while generating results, the iterator's internal state is not guaranteed to remain at its previous position. Call `startOver()` before continuing to use the iterator.

Author(s)

Joseph Wood

See Also

comboGroups

Examples

```
a = comboGroupsIter(12, 3)
a@nextIter()
a@nextNIter(3)
a@front()
all_remaining = a@nextRemaining()
dim(all_remaining)
a@summary()
a@back()
a[[5]]
a@summary()
a[[c(1, 17, 3)]]
a@summary()
```

---

comboGroupsSample *Sample Partitions of a Vector into Groups*

Description

- Generate a specific (lexicographically) or random sample of partitions of groups.
- Produce results in parallel using the `Parallel` or `nThreads` arguments.
- GMP support allows for exploration where the number of results is large.

Usage

```
comboGroupsSample(v, numGroups = NULL, grpSizes = NULL, retType = "matrix",
                  n = NULL, sampleVec = NULL, seed = NULL, Parallel = FALSE,
                  nThreads = NULL, namedSample = FALSE)
```

Arguments

`v`	Source vector. If `v` is a positive integer, it will be converted to the sequence `1:v`. If `v` is a negative integer, it will be converted to the sequence `v:-1`. All atomic types are supported (See `is.atomic`).
`numGroups`	An Integer. The number of groups that the vector will be partitioned into. The default is `NULL`. If provided and `grpSize` is `NULL`, it must divide the length of `v` (if `v` is a vector) or `v` (if `v` is a scalar).
`grpSizes`	A vector of whole numbers representing the size of each group. The default is `NULL`. If provided, the sum of the elements must total the length of `v` (if `v` is a vector) or `v` (if `v` is a scalar).
`retType`	A string, "3Darray" or "matrix", that determines the shape of the output. The default is "matrix". Note, "3Darray" can only be used when the size of each group is uniform. When the size of each group varies, the return output will always be a matrix.
`n`	Number of results to return. The default is `NULL`.

sampleVec	A vector of numbers representing the lexicographical partition of groups to return. Accepts vectors of class bigz as well as vectors of characters
seed	Random seed initialization. The default is NULL. N.B. If the gmp library is needed, this parameter must be set in order to have reproducible results (*E.g* set.seed() has no effect in these cases).
Parallel	Logical value indicating whether results should be generated in parallel. The default is FALSE. If TRUE and nThreads = NULL, the number of threads used is equal to the minimum of one minus the number of threads available on your system and the number of results requested (*e.g.* if user has 16 threads and only needs 5 results, 5 threads will be used (*i.e.* min(16 - 1, 5) = 5)). If nThreads is not NULL, it will be given preference (*e.g.* if user has 8 threads with Parallel = TRUE and nThreads = 4, only 4 threads will be spawned). If your system is single-threaded, the arguments Parallel and nThreads are ignored.
nThreads	Specific number of threads to be used. The default is NULL. See Parallel.
namedSample	Logical flag. If TRUE, rownames corresponding to the lexicographical result, will be added to the returned matrix. The default is FALSE.

Details

These algorithms rely on efficiently generating the n^{th} lexicographical result.

Value

By default, a matrix is returned with column names corresponding to the associated group. If retType = "3Darray", a 3D array is returned.

Author(s)

Joseph Wood

References

Lexicographical order

Examples

```
## generate 10 random partitions of groups of equal size
comboGroupsSample(10, 2, n = 10, seed = 123)

## generate 10 random partitions of groups of varying sizes
comboGroupsSample(10, grpSizes = 1:4, n = 10, seed = 123)

## using sampleVec to generate specific results
comboGroupsSample(15, 5, sampleVec = c(1, 100, 1e3, 1e6))

all.equal(comboGroupsSample(10, 5,
            sampleVec = 1:comboGroupsCount(10, 5)),
          comboGroups(10, 5))

## Examples with enormous number of total results
```

```
num = comboGroupsCount(100, 20)
gmp::log2.bigz(num)
## [1] 325.5498

first = gmp::urand.bigz(n = 1, size = 325, seed = 123)
mySamp = do.call(c, lapply(0:10, function(x) gmp::add.bigz(first, x)))

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

## using the sampling function
cbgSamp = comboGroupsSample(100, 20, sampleVec = mySamp)

## using the standard function
cbgGeneral = comboGroups(100, 20,
                         lower = first,
                         upper = gmp::add.bigz(first, 10))

identical(cbgSamp, cbgGeneral)
## [1] TRUE

## Not run:
## Using Parallel
system.time(comboGroupsSample(1000, 20, n = 80, seed = 10, Parallel = TRUE))

## End(Not run)
```

`comboIter` *Combination and Permutation Iterator*

Description

- Returns an iterator for iterating over combinations or permutations of a vector with or without constraints.
- Supports random access via the `[[` method.
- GMP support allows for exploration of combinations/permutations of vectors with many elements.
- The output is in lexicographical order for the `next` methods and reverse lexicographical order for the `prev` methods.
- Learn more in `vignette("iterators")`.

Usage

```
comboIter(v, m = NULL, ...)
permuteIter(v, m = NULL, ...)

## S3 method for class 'numeric'
comboIter(v, m = NULL, repetition = FALSE, freqs = NULL,
```

```
           constraintFun = NULL, comparisonFun = NULL,
           limitConstraints = NULL, keepResults = NULL,
           FUN = NULL, Parallel = FALSE, nThreads = NULL,
           tolerance = NULL, FUN.VALUE = NULL, ...)

## S3 method for class 'numeric'
permuteIter(v, m = NULL, repetition = FALSE, freqs = NULL,
            constraintFun = NULL, comparisonFun = NULL,
            limitConstraints = NULL, keepResults = NULL,
            FUN = NULL, Parallel = FALSE, nThreads = NULL,
            tolerance = NULL, FUN.VALUE = NULL, ...)

## S3 method for class 'factor'
comboIter(
    v, m = NULL, repetition = FALSE, freqs = NULL, FUN = NULL,
    Parallel = FALSE, nThreads = NULL, FUN.VALUE = NULL, ...
)
## S3 method for class 'factor'
permuteIter(
    v, m = NULL, repetition = FALSE, freqs = NULL, FUN = NULL,
    Parallel = FALSE, nThreads = NULL, FUN.VALUE = NULL, ...
)

## Default S3 method:
comboIter(
    v, m = NULL, repetition = FALSE, freqs = NULL,
    FUN = NULL, FUN.VALUE = NULL, ...
)
## Default S3 method:
permuteIter(
    v, m = NULL, repetition = FALSE, freqs = NULL,
    FUN = NULL, FUN.VALUE = NULL, ...
)

## S3 method for class 'table'
comboIter(
    v, m = NULL, constraintFun = NULL, comparisonFun = NULL,
    limitConstraints = NULL, keepResults = NULL, FUN = NULL,
    Parallel = FALSE, nThreads = NULL, tolerance = NULL, FUN.VALUE = NULL, ...
)
## S3 method for class 'table'
permuteIter(
    v, m = NULL, constraintFun = NULL, comparisonFun = NULL,
    limitConstraints = NULL, keepResults = NULL, FUN = NULL,
    Parallel = FALSE, nThreads = NULL, tolerance = NULL, FUN.VALUE = NULL, ...
)

## S3 method for class 'list'
```

```
comboIter(v, m = NULL, repetition = FALSE, freqs = NULL, ...)
## S3 method for class 'list'
permuteIter(v, m = NULL, repetition = FALSE, freqs = NULL, ...)
```

Arguments

v	Source vector. If v is a positive integer, it will be converted to the sequence 1:v. If v is a negative integer, it will be converted to the sequence v:-1. All atomic types are supported (See is.atomic).
m	Number of elements to choose. If repetition = TRUE or freqs is utilized, m can exceed the length of v. If m = NULL, the length will default to length(v) or sum(freqs).
...	Further arguments passed to methods.
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.
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, *e.g* comparisonFun = c(comp1, comp2), and limitConstraints is a vector of two numerical values, *e.g* limitConstraints = c(x1, x2), the combinations/permutations will be filtered in one of the following two ways: 1. When comp1 is one of the 'greater-than' operators (*i.e.* ">=" or ">"), comp2 is one of the 'less-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. In other words, the first comparison operator is applied to the first limit and the second operator is applied to the second limit.
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.

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

Parallel — Logical value indicating whether combinations/permutations should be generated in parallel using $n-1$ threads, where n is the maximum number of threads. The default is FALSE. If nThreads is not NULL, it will be given preference (*e.g.* if user has 8 threads with Parallel = TRUE and nThreads = 4, only 4 threads will be spawned). If your system is single-threaded, the arguments Parallel and nThreads are ignored.

nThreads — Specific number of threads to be used. The default is NULL. See Parallel.

tolerance — A numeric value greater than or equal to zero. This parameter is utilized when a constraint is applied on a numeric vector. The default value is 0 when it can be determined that whole values are being utilized, otherwise it is sqrt(.Machine$double.eps) which is approximately $1.5e-8$. N.B. If the input vector is of type integer, this parameter will be ignored and strict equality will be enforced.

FUN.VALUE — A template for the return value from FUN. See 'Details' of vapply for more information.

Details

Once you initialize a new iterator, the following methods are available via @ (*e.g.* a@nextIter()) or $ (*e.g.* a$nextIter()). The preferred practice is to use @ as it is much more efficient (See examples below). Also note that not all of the methods below are available in all cases. See Combo-class, Constraints-class, and Partitions-class:

nextIter Retrieve the **next** lexicographical result

nextNIter Pass an integer *n* to retrieve the **next** *n* lexicographical results

nextRemaining Retrieve all remaining lexicographical results

currIter Returns the current iteration

prevIter Retrieve the **previous** lexicographical result (the **next** *reverse* lexicographical result)

prevNIter Pass an integer *n* to retrieve the **previous** *n* lexicographical results (the **next** *n reverse* lexicographical results)

prevRemaining Retrieve all remaining *reverse* lexicographical results

startOver Resets the iterator

sourceVector View the source vector

summary Returns a list of summary information about the iterator

front Retrieve the **first** lexicographical result

back Retrieve the **last** lexicographical result

[[Random access method. Pass a single value or a vector of valid indices. If a single value is passed, the internal index of the iterator will be updated, however if a vector is passed the internal state will not change. GMP support allows for flexible indexing.

Value

- If `nextIter` or `prevIter` is called, a vector is returned
- Otherwise, a matrix with m or $m + 1$ columns, depending on the value of `keepResults`
- If `FUN` is utilized, `FUN.VALUE = NULL`, and either `nextIter` or `prevIter` is called, the result will be determined by `FUN`, otherwise a list is returned.
- When both `FUN` and `FUN.VALUE` are not `NULL`, the return is modeled after the return of `vapply`. See the 'Value' section of `vapply`.

Note

- `Parallel` and `nThreads` will be ignored in the following cases:
  - When the output is constrained (except for most partitions cases).
  - For special constraints (see the Details section of `comboGeneral`), `nThreads` is ignored and generation is performed serially. Unlike an explicitly supplied `lower` in the general interface, an iterator's current position does not define a range of candidates that can be safely divided among threads.
  - If the class of the vector passed is `character`, `raw`, and `complex` (N.B. `Rcpp::CharacterMatrix` is not thread safe). Alternatively, you can generate an indexing matrix in parallel.
  - If `FUN` is utilized.
- If either `constraintFun`, `comparisonFun` or `limitConstraints` is `NULL` –or– if the class of the vector passed is `logical`, `character`, `raw`, `factor`, or `complex`, 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 $2^{31} - 1$.
- Factor vectors are accepted. Class and level attributes are preserved except when `FUN` is used.
- Lexicographical ordering isn't guaranteed for permutations if the output is constrained.
- `FUN` will be ignored if the constraint check is satisfied.
- If an error occurs while generating results, the iterator's internal state is not guaranteed to remain at its previous position. Call `startOver()` before continuing to use the iterator.

Author(s)

Joseph Wood

References

- Lexicographical Order
- Reverse Lexicographical Order

See Also

`comboGeneral`, `permuteGeneral`

Examples

```
## Typical usage
a = permuteIter(unique(state.region))
a@nextIter()
a@nextNIter(3)
a@front()
a@nextRemaining()
a@prevIter()
a@prevNIter(15)
a@summary()
a@back()
a@prevRemaining()
a[[5]]
a@summary()
a[[c(1, 17, 3)]]
a@summary()

## See examples for comboGeneral where lower and upper are used
set.seed(1009)
mySamp = sort(rnorm(75, 997, 23))

b = comboIter(mySamp, 7,
              constraintFun = "sum",
              comparisonFun = ">",
              limitConstraints = 7200)
b@nextIter()
b@nextNIter(3)
b@summary()
b@currIter()

## Not run:
## We don't have random access or previous methods
b@back()
#> Error: no slot of name "back" for this object of class "Constraints"
b@prevIter()
#> Error: no slot of name "prevIter" for this object of class "Constraints"

## End(Not run)
```

---

comboRank *Rank Combinations and Permutations*

Description

- Generate the rank (lexicographically) of combinations/permutations. These functions are the complement to `comboSample` and `permuteSample`. See the examples below.
- GMP support allows for exploration of combinations/permutations of vectors with many elements.

Usage

```
comboRank(..., v, repetition = FALSE, freqs = NULL, nThreads = NULL)
permuteRank(..., v, repetition = FALSE, freqs = NULL, nThreads = NULL)
```

Arguments

...	vectors or matrices to be ranked.
v	Source vector. If v is a positive integer, it will be converted to the sequence 1:v. If v is a negative integer, it will be converted to the sequence v:-1. All atomic types are supported (See is.atomic).
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.
nThreads	Specific number of threads to be used. The default is NULL.

Details

These algorithms rely on efficiently ranking the n^{th} lexicographical combination/permutation.

Value

A vector of class integer, numeric, or bigz determined by the total number of combinations/permutations

Note

v must be supplied.

Author(s)

Joseph Wood

References

Lexicographical order ranking/unranking

See Also

comboSample, permuteSample

Examples

```
mySamp = comboSample(30, 8, TRUE, n = 5, seed = 10, namedSample = TRUE)
myRank = comboRank(mySamp, v = 30, repetition = TRUE)
all.equal(as.integer(rownames(mySamp)), myRank)
```

comboSample *Sample Combinations and Permutations*

Description

- Generate a specific (lexicographically) or random sample of combinations/permutations.
- Produce results in parallel using the `Parallel` or `nThreads` arguments.
- GMP support allows for exploration of combinations/permutations of vectors with many elements.

Usage

```
comboSample(v, m = NULL, ...)
permuteSample(v, m = NULL, ...)

## S3 method for class 'numeric'
comboSample(v, m = NULL, repetition = FALSE, freqs = NULL, n = NULL,
            sampleVec = NULL, seed = NULL, FUN = NULL, Parallel = FALSE,
            nThreads = NULL, namedSample = FALSE, FUN.VALUE = NULL, ...)

## S3 method for class 'numeric'
permuteSample(v, m = NULL, repetition = FALSE, freqs = NULL, n = NULL,
              sampleVec = NULL, seed = NULL, FUN = NULL, Parallel = FALSE,
              nThreads = NULL, namedSample = FALSE, FUN.VALUE = NULL, ...)

## S3 method for class 'factor'
comboSample(
    v, m = NULL, repetition = FALSE, freqs = NULL, n = NULL,
    sampleVec = NULL, seed = NULL, FUN = NULL, Parallel = FALSE,
    nThreads = NULL, namedSample = FALSE, FUN.VALUE = NULL, ...
)
## S3 method for class 'factor'
permuteSample(
    v, m = NULL, repetition = FALSE, freqs = NULL, n = NULL,
    sampleVec = NULL, seed = NULL, FUN = NULL, Parallel = FALSE,
    nThreads = NULL, namedSample = FALSE, FUN.VALUE = NULL, ...
)

## Default S3 method:
comboSample(
    v, m = NULL, repetition = FALSE, freqs = NULL, n = NULL, sampleVec = NULL,
    seed = NULL, FUN = NULL, namedSample = FALSE, FUN.VALUE = NULL, ...
)
## Default S3 method:
permuteSample(
    v, m = NULL, repetition = FALSE, freqs = NULL, n = NULL, sampleVec = NULL,
```

```
    seed = NULL, FUN = NULL, namedSample = FALSE, FUN.VALUE = NULL, ...
)

## S3 method for class 'table'
comboSample(
    v, m = NULL, n = NULL, sampleVec = NULL, seed = NULL, FUN = NULL,
   Parallel = FALSE, nThreads = NULL, namedSample = FALSE, FUN.VALUE = NULL, ...
)
## S3 method for class 'table'
permuteSample(
    v, m = NULL, n = NULL, sampleVec = NULL, seed = NULL, FUN = NULL,
   Parallel = FALSE, nThreads = NULL, namedSample = FALSE, FUN.VALUE = NULL, ...
)

## S3 method for class 'list'
comboSample(
    v, m = NULL, repetition = FALSE, freqs = NULL, n = NULL,
    sampleVec = NULL, seed = NULL, namedSample = FALSE, ...
)
## S3 method for class 'list'
permuteSample(
    v, m = NULL, repetition = FALSE, freqs = NULL, n = NULL,
    sampleVec = NULL, seed = NULL, namedSample = FALSE, ...
)
```

Arguments

v	Source vector. If v is a positive integer, it will be converted to the sequence 1:v. If v is a negative integer, it will be converted to the sequence v:-1. All atomic types are supported (See is.atomic).
m	Number of elements to choose. If repetition = TRUE or freqs is utilized, m can exceed the length of v. If m = NULL, the length will default to length(v) or sum(freqs).
...	Further arguments passed to methods.
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 indices representing the lexicographical combination/permutations to return. Accepts whole numbers as well as vectors of class bigz as well as vectors of characters
seed	Random seed initialization. The default is NULL. N.B. If the gmp library is needed, this parameter must be set in order to have reproducible results (*E.g* set.seed() has no effect in these cases).

FUN	Function to be applied to each combination/permutation. The default is NULL.
Parallel	Logical value indicating whether combinations/permutations should be generated in parallel. The default is FALSE. If TRUE and nThreads = NULL, the number of threads used is equal to the minimum of one minus the number of threads available on your system and the number of results requested (*e.g.* if user has 16 threads and only needs 5 results, 5 threads will be used (*i.e.* min(16 - 1, 5) = 5)). If nThreads is not NULL, it will be given preference (*e.g.* if user has 8 threads with Parallel = TRUE and nThreads = 4, only 4 threads will be spawned). If your system is single-threaded, the arguments Parallel and nThreads are ignored.
nThreads	Specific number of threads to be used. The default is NULL. See Parallel.
namedSample	Logical flag. If TRUE, rownames corresponding to the lexicographical combination/permutation, will be added to the returned matrix. The default is FALSE.
FUN.VALUE	A template for the return value from FUN. See 'Details' of vapply for more information.

Details

These algorithms rely on efficiently generating the n^{th} lexicographical combination/permutation. This is the process of unranking.

Value

- In general, a matrix with m or $m + 1$ columns, depending on the value of keepResults
- If FUN is utilized and FUN.VALUE = NULL, a list is returned
- When both FUN and FUN.VALUE are not NULL, the return is modeled after the return of vapply. See the 'Value' section of vapply.

Note

- Parallel and nThreads will be ignored in the following cases:
  - If the class of the vector passed is character (N.B. Rcpp::CharacterMatrix is not thread safe). Alternatively, you can generate an indexing matrix in parallel.
  - If FUN is utilized.
- n and sampleVec cannot both be NULL.
- Factor vectors are accepted. Class and level attributes are preserved except when FUN is used.

Author(s)

Joseph Wood

References

Lexicographical order

See Also

comboRank, permuteRank

Examples

```
## generate 10 random combinations
comboSample(30, 8, TRUE, n = 5, seed = 10)

## Using sampleVec to generate specific permutations
fqs   = c(1,2,2,1,2,2,1,2,1,2,2,1,2,1,1)
s_idx = c(1, 10^2, 10^5, 10^8, 10^11)

permuteSample(15, 10, freqs = fqs, sampleVec = s_idx)

## Same example using 'table' method
permuteSample(table(rep(1:15, times = fqs)), 10, sampleVec = s_idx)

## Generate each result one by one...
## Same, but not as efficient as generating iteratively
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)

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)

## 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)

## Using nThreads
permPar = permuteSample(10000, 50, n = 8, seed = 10, nThreads = 2)

## Using FUN
permuteSample(10000, 50, n = 4, seed = 10, FUN = sd)

## Not run:
## Using Parallel
permuteSample(10000, 50, n = 80, seed = 10, Parallel = TRUE)

## End(Not run)
```

`Constraints-class` *S4-class for Exposing C++ Constraints Class*

Description

The `Constraints` class is an S4-class that exposes C++ classes that provide access to iterators and other useful methods.

Slots

`nextIter` Retrieve the **next** lexicographical result

`nextNIter` Pass an integer *n* to retrieve the **next** *n* lexicographical results

`nextRemaining` Retrieve all remaining lexicographical results

`currIter` Returns the current iteration

`startOver` Resets the iterator

`sourceVector` View the source vector

`summary` Returns a list of summary information about the iterator

Author(s)

Joseph Wood

See Also

`Combo-class`, `Partitions-class`

Examples

```
showClass("Constraints")
```

`divisorsRcpp` *Vectorized Factorization (Complete)*

Description

Function for generating the complete factorization for a vector of numbers.

Usage

```
divisorsRcpp(v, namedList = FALSE, nThreads = NULL)
```

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`.
`nThreads`	Specific number of threads to be used. The default is `NULL`.

Details

Efficient algorithm that builds on `primeFactorize` to 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

`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)

## Using nThreads
system.time(divisorsRcpp(myVec, nThreads = 2))
```

`divisorsSieve` *Generate Complete Factorization for Numbers in a Range*

Description

Sieve that generates the complete factorization of all numbers between `bound1` and `bound2` (if supplied) or all numbers up to `bound1`.

Usage

```
divisorsSieve(bound1, bound2 = NULL, namedList = FALSE, nThreads = NULL)
```

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`.
`nThreads`	Specific number of threads to be used. The default is `NULL`.

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 https://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, primeFactorizeSieve

Examples

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

## 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))

## Use nThreads for improved efficiency
system.time(divisorsSieve(10^12, 10^12 + 10^5, nThreads = 2))

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

## Using nThreads
system.time(divisorsSieve(1e5, 2e5, nThreads = 2))
```

---

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

Description

Sieve that 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, bound2 = NULL, namedVector = FALSE, nThreads = NULL)
```

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.
nThreads	Specific number of threads to be used. The default is NULL.

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 https://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

Examples

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

## 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)

## Using nThreads
system.time(eulerPhiSieve(1e5, 2e5, nThreads = 2))
```

---

expandGrid *Cartesian Product*

Description

Generate the Cartesian Product of the input vectors. It is very similar to `expand.grid` however there are some notable differences:

- Produces lexicographic ordered output consistent with other functions in `RcppAlgos`. Compared to `expand.grid` where the first column varies the fastest, `expandGrid` varies the first column the slowest.
- When all of the input is of the same type, by default `expandGrid` produce a `matrix` (a `data.frame` otherwise). This can be ignored by setting the argument `return_df = TRUE`.
- No attributes are added nor are strings converted to factors. In `expand.grid` we would achieve this by setting `KEEP.OUT.ATTRS = FALSE` and `stringsAsFactors = FALSE`.
- If it is possible to return a matrix, we can utilize the argument `nThreads` in order to produce results in parallel for maximal efficiency.

Usage

```
expandGrid(..., lower = NULL, upper = NULL,
           nThreads = NULL, return_df = FALSE)
```

Arguments

...	vectors, factors or a list containing these. (See ?expand.grid).
lower	The lower bound. Cartesian products are generated lexicographically, thus utilizing this argument will determine which specific product to start generating from (*e.g.* expandGrid(1:5, 3:11, lower = 6) is equivalent to expandGrid(1:5, 3:11)[6:expandGridCount(1:5, 3:11),]). This argument along with upper is very useful for generating products 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 product (*e.g.* expandGrid(1:5, 3:11, upper = 5) is equivalent to expandGrid(1:5, 3:11)[1:5,])
nThreads	Specific number of threads to be used. The default is NULL.
return_df	Logical flag to force the output to be a data.frame. The default is FALSE.

Value

When all of the input is of the same type, by default expandGrid produces a matrix (a data.frame otherwise). This can be ignored by setting the argument return_df = TRUE.

Author(s)

Joseph Wood

See Also

comboGrid

Examples

```
## description example
lst = list(1:2, 1:2)

## Using base R
t = expand.grid(lst)

## vs using expandGrid. N.B. Output is a matrix
expandGrid(lst)

## Force a data.frame to be returned
expandGrid(lst, return_df = TRUE)

lst = Map(function(x, y) x:y, 8:14, 15:21)

## Use multiple threads for greater efficiency
system.time(expandGrid(lst, nThreads = 2))
```

expandGridCount *Count of the Cartesian Product*

Description

Calculate the number of Cartesian products of the input vectors.

Usage

```
expandGridCount(...)
```

Arguments

...	vectors, factors or a list containing these. (See ?expand.grid).

Value

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

Author(s)

Joseph Wood

See Also

expandGrid

Examples

```
## description example
lst = list(1:2, 1:2)

## Using base R
t = expand.grid(lst)
nrow(t)

## vs calling expandGridCount directly
expandGridCount(lst)

## Call it just like you would if you were generating the results
expandGridCount(1:5, 3:9, letters[1:5], letters[c(1,4,5,8)])

## Same as
nrow(expand.grid(1:5, 3:9, letters[1:5], letters[c(1,4,5,8)]))

lst = Map(function(x, y) x:y, 8:33, 15:40)

## Return object of class 'bigz'
expandGridCount(lst)
```

`expandGridIter` *expandGrid Iterator*

Description

- Returns an iterator for iterating over the Cartesian product of the input vectors.
- Supports random access via the `[[` method.
- GMP support allows for exploration of cases where the number of products is large.
- Use the `next` methods to obtain results in lexicographical order.

Usage

```
expandGridIter(..., nThreads = NULL, return_df = FALSE)
```

Arguments

`...`	vectors, factors or a list containing these. (See `?expand.grid`).
`nThreads`	Specific number of threads to be used. The default is `NULL`.
`return_df`	Logical flag to force the output to be a `data.frame`. The default is `FALSE`.

Details

Once you initialize a new iterator, the following methods are available:

`nextIter` Retrieve the **next** lexicographical result

`nextNIter` Pass an integer *n* to retrieve the **next** *n* lexicographical results

`nextRemaining` Retrieve all remaining lexicographical results

`currIter` Returns the current iteration

`startOver` Resets the iterator

`sourceVector` View the source input

`summary` Returns a list of summary information about the iterator

`front` Retrieve the **first** lexicographical result

`back` Retrieve the **last** lexicographical result

`[[` Random access method. Pass a single value or a vector of valid indices. If a single value is passed, the internal index of the iterator will be updated, however if a vector is passed the internal state will not change. GMP support allows for flexible indexing.

Value

- If `nextIter` is called, a named vector is returned if a `matrix` can be returned in the general case. Otherwise, a `data.frame` is returned.
- When `nextNIter` and `nextRemaining` are called, a named `matrix` is returned when all of the input is of the same type and `return_df = FALSE`. Otherwise, a `data.frame` is returned.

Note

- If `nThreads` is utilized, it will only take effect if the number of elements requested is greater than some threshold (determined internally). *E.g*:

```
serial   <- expandGridIter(Map(\(x, y) x:y, 1:10, 11:20))
multi    <- expandGridIter(Map(\(x, y) x:y, 1:10, 11:20), nThreads = 4)
fetch1e6 <- multi@nextNIter(1e6)  ## much faster than serial@nextNIter(1e6)
fetch1e3 <- multi@nextNIter(1e3)  ## only one thread used... same as serial@nextNIter(1e3)

library(microbenchmark)
microbenchmark(multi@nextNIter(1e6), serial@nextNIter(1e6), times = 20)
microbenchmark(multi@nextNIter(1e3), serial@nextNIter(1e3), times = 20)
```

- The maximum number of expandGrid that can be generated at one time is $2^{31} - 1$.
- If an error occurs while generating results, the iterator's internal state is not guaranteed to remain at its previous position. Call `startOver()` before continuing to use the iterator.

Author(s)

Joseph Wood

See Also

expandGrid

Examples

```
a = expandGridIter(factor(state.abb), euro, islands)
a@nextIter()
a@nextNIter(3)
a@front()
all_remaining = a@nextRemaining()
dim(all_remaining)
a@summary()
a@back()
a[[5]]
a@summary()
a[[c(1, 17, 3)]]
a@summary()
```

---

expandGridSample *Sample the Cartesian Product*

Description

- Generate a specific (lexicographically) or random sample of the Cartesian product of the input vectors.
- Produce results in parallel using the `nThreads` arguments.
- GMP support allows for exploration where the number of results is large.

Usage

```
expandGridSample(
    ..., n = NULL, sampleVec = NULL, seed = NULL,
    nThreads = NULL, namedSample = FALSE, return_df = FALSE
)
```

Arguments

...	vectors, factors or a list containing these. (See ?expand.grid).
n	Number of results to return. The default is NULL.
sampleVec	A vector of numbers representing the lexicographical partition of groups to return. Accepts vectors of class bigz as well as vectors of characters
seed	Random seed initialization. The default is NULL. N.B. If the gmp library is needed, this parameter must be set in order to have reproducible results (*E.g* set.seed() has no effect in these cases).
nThreads	Specific number of threads to be used. The default is NULL.
namedSample	Logical flag. If TRUE, rownames corresponding to the lexicographical result, will be added to the returned matrix. The default is FALSE.
return_df	Logical flag to force the output to be a data.frame. The default is FALSE.

Details

These algorithms rely on efficiently generating the n^{th} lexicographical result.

Value

When all of the input is of the same type, by default expandGridSample produces a matrix (a data.frame otherwise). This can be ignored by setting the argument return_df = TRUE.

Author(s)

Joseph Wood

References

Lexicographical order

Examples

```
## input vectors
lst = list(factor(state.abb), euro, islands)

## generate 10 random products
expandGridSample(lst, n = 10, seed = 123)

## using sampleVec to generate specific results
expandGridSample(lst, sampleVec = c(1, 100, 1e3))
```

```
all.equal(expandGridSample(lst, sampleVec = 1:expandGridCount(lst)),
         expandGrid(lst))

## Examples with enormous number of total results
big_lst = Map(function(x, y) x:y, 8:33, 15:40)
num = expandGridCount(big_lst)
gmp::log2.bigz(num)
## [1] 78

first = gmp::urand.bigz(n = 1, size = 78, seed = 123)
mySamp = do.call(c, lapply(0:10, function(x) gmp::add.bigz(first, x)))

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

## using the sampling function
cartSamp = expandGridSample(big_lst, sampleVec = mySamp)

## using the standard function
cartGeneral = expandGrid(big_lst,
                         lower = first,
                         upper = gmp::add.bigz(first, 10))

identical(cartSamp, cartGeneral)
## [1] TRUE
```

`isPrimeRcpp` *Vectorized Primality Test*

Description

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: https://gmplib.org.

Usage

```
isPrimeRcpp(v, namedVector = FALSE, nThreads = NULL)
```

Arguments

v	Vector of integers or numeric values.
namedVector	Logical flag. If TRUE, a named vector is returned. The default is FALSE.
nThreads	Specific number of threads to be used. The default is NULL.

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 $(\mod p)$.

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 $x_2 = y_1 + y_2 + \cdots + y_n$, so that we obtain:

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

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." https://www.math.uconn.edu/~kconrad/blurbs/ugradnumthy/mille
- 53-bit significand precision

See Also

`primeFactorize`, `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, 10)

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

## Using nThreads
system.time(isPrimeRcpp(mySamp, nThreads = 2))
```

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

Description

Sieve that 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, bound2 = NULL, namedVector = FALSE, nThreads = NULL)
```

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`.
`nThreads`	Specific number of threads to be used. The default is `NULL`.

Details

Simple and efficient sieve that 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 https://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)

## 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)

## Using nThreads
system.time(numDivisorSieve(1e5, 2e5, nThreads = 2))
```

Partitions-class *S4-class for Exposing C++ Partitions Class*

Description

The `Partitions` class is an S4-class that exposes C++ classes that provide access to iterators and other useful methods.

Slots

`nextIter` Retrieve the **next** lexicographical result

`nextNIter` Pass an integer *n* to retrieve the **next** *n* lexicographical results

`nextRemaining` Retrieve all remaining lexicographical results

`currIter` Returns the current iteration

`startOver` Resets the iterator

`sourceVector` View the source vector

`summary` Returns a list of summary information about the iterator

`front` Retrieve the **first** lexicographical result

`back` Retrieve the **last** lexicographical result

`randomAccess` Random access method. Pass a single value or a vector of valid indices. If a single value is passed, the internal index of the iterator will be updated, however if a vector is passed the internal state will not change. GMP support allows for flexible indexing.

Author(s)

Joseph Wood

See Also

Combo-class, Constraints-class

Examples

```
showClass("Partitions")
```

`partitionsCount` *Number of Partitions/Compositions*

Description

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

Usage

```
partitionsCount(v, m = NULL, ...)
compositionsCount(v, m = NULL, ...)

## Default S3 method:
partitionsCount(v, m = NULL, repetition = FALSE,
                freqs = NULL, target = NULL, ...)
## Default S3 method:
compositionsCount(v, m = NULL, repetition = FALSE,
```

```
                    freqs = NULL, target = NULL, weak = FALSE, ...)

## S3 method for class 'table'
partitionsCount(v, m = NULL, target = NULL, ...)
## S3 method for class 'table'
compositionsCount(v, m = NULL, target = NULL, weak = FALSE, ...)
```

Arguments

v	Source vector. If v is a positive integer, it will be converted to the sequence 1:v. If v is a negative integer, it will be converted to the sequence v:-1. Only integer and numeric vectors are accepted.
m	Width of the partition. If m = NULL, the length will be determined by the partitioning case (*e.g.* When we are generating distinct partitions of n, the width will be equal to the smallest m such that sum(1:m) >= n).
...	Further arguments passed to methods.
repetition	Logical value indicating whether partitions/compositions should be with or without repetition. The default is FALSE.
freqs	A vector of frequencies used for producing all partitions 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.
target	Number to be partitioned. If NULL, max(v) will be used.
weak	(Compositions only) Logical flag indicating whether to allow terms of the sequence to be zero.

Value

A numerical value representing the total number of partitions/compositions.

Note

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

See Also

partitionsGeneral, compositionsGeneral

Examples

```
## Same interface as partitionsGeneral
partitionsCount(25, 5)
compositionsCount(25, 5, TRUE)
partitionsCount(15, 7, TRUE)
partitionsCount(25, 5, freqs = rep(2, 25))

## Return object of class 'bigz'
partitionsCount(2500, 15, TRUE)
compositionsCount(2500, 15, TRUE)
```

partitionsGeneral *Generate Partitions/Compositions*

Description

The algorithms in RcppAlgos go beyond the traditional integer partition algorithms and can tackle a wide variety of cases.

- Efficient algorithms for partitioning numbers under various constraints:
  - Standard (with repetition)
  - Distinct
  - Restricted
  - Where each part has a specific multiplicity (i.e. when using freqs for multisets).
  - Arbitrary target and source vector (*e.g.* partitionsGeneral(sample(1000, 20), 10, TRUE, target = 5000))
- Produce results in parallel using the nThreads arguments.
- Alternatively, the arguments lower and upper make it possible to generate partitions/compositions in chunks allowing for parallelization via the parallel package.
- GMP support allows for exploration of cases where the number of partitions/compositions is large.
- The output is in lexicographical order.

Usage

```
partitionsGeneral(v, m = NULL, ...)
compositionsGeneral(v, m = NULL, ...)

## Default S3 method:
partitionsGeneral(
    v, m = NULL, repetition = FALSE, freqs = NULL, target = NULL,
    lower = NULL, upper = NULL, nThreads = NULL, tolerance = NULL, ...
)
## Default S3 method:
compositionsGeneral(
    v, m = NULL, repetition = FALSE, freqs = NULL, target = NULL, weak = FALSE,
    lower = NULL, upper = NULL, nThreads = NULL, tolerance = NULL, ...
)

## S3 method for class 'table'
partitionsGeneral(
    v, m = NULL, target = NULL, lower = NULL,
    upper = NULL, nThreads = NULL, tolerance = NULL, ...
)
## S3 method for class 'table'
compositionsGeneral(
```

```
    v, m = NULL, target = NULL, weak = FALSE, lower = NULL,
    upper = NULL, nThreads = NULL, tolerance = NULL, ...
)
```

Arguments

v	Source vector. If v is a positive integer, it will be converted to the sequence 1:v. If v is a negative integer, it will be converted to the sequence v:-1. Only integer and numeric vectors are accepted.
m	Width of the partition. If m = NULL, the length will be determined by the partitioning case (*e.g.* When we are generating distinct partitions of n, the width will be equal to the smallest m such that sum(1:m) >= n).
...	Further arguments passed to methods.
repetition	Logical value indicating whether partitions/compositions should be with or without repetition. The default is FALSE.
freqs	A vector of frequencies used for producing all partitions 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. Partitions/compositions are generated lexicographically, thus utilizing this argument will determine which specific partition to start generating from (*e.g.* partitionsGeneral(15, 3, lower = 6) is equivalent to partitionsGeneral(15, 3)[6:partitionsCount(15, 3),]). This argument along with upper is very useful for generating partitions/compositions 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 partition (*e.g.* partitionsGeneral(15, 3, upper = 5) is equivalent to partitionsGeneral(15, 3)[1:5,])
target	Number to be partitioned. If NULL, max(v) will be used.
weak	(Compositions only) Logical flag indicating whether to allow terms of the sequence to be zero.
nThreads	Specific number of threads to be used. The default is NULL.
tolerance	A numeric value greater than or equal to zero. This parameter is utilized when a constraint is applied on a numeric vector. The default value is 0 when it can be determined that whole values are being utilized, otherwise it is sqrt(.Machine$double.eps) which is approximately $1.5e-8$. N.B. If the input vector is of type integer, this parameter will be ignored and strict equality will be enforced.

Value

A matrix is returned with each row containing a vector of length m.

Note

- nThreads will be ignored in the following cases (i.e. Generating the n^{th} partition in these cases are currently unavailable):

  - With standard multisets. If zero is the only element with a non-trivial multiplicity, multi-threading is possible (*e.g.* `partitionsGeneral(0:100, freqs = c(100, rep(1, 100)), nThreads = 4)`).
  - If the source vector is not isomorphic to `1:length(v)` (*e.g.* `v = c(1, 4, 6, 7, 8)`).
- The maximum number of partitions/compositions that can be generated at one time is $2^{31} - 1$. Utilizing `lower` and `upper` makes it possible to generate additional partitions/compositions.

Author(s)

Joseph Wood

References

- Lexicographical Order
- Subset Sum Problem
- Partition (number theory)
- Composition (combinatorics))

Examples

```
partitionsGeneral(1)
partitionsGeneral(-1:0, 1)
partitionsGeneral(-1:0, 1, target = -1)
partitionsGeneral(20, 5)
partitionsGeneral(20, 5, repetition = TRUE)
partitionsGeneral(20, 5, freqs = rep(1:4, 5))
partitionsGeneral(20, 5, TRUE, target = 80)
partitionsGeneral(0:10, repetition = TRUE)
partitionsGeneral(seq(2L, 500L, 23L), 5, target = 1804)

compositionsGeneral(0:10, 5, repetition = TRUE)

set.seed(111)
partitionsGeneral(sample(1000, 20), 5, TRUE, target = 2500)

system.time(one_thread  <- partitionsGeneral(80, 10, TRUE))
system.time(two_threads <- partitionsGeneral(80, 10, TRUE, nThreads = 2))
identical(one_thread, two_threads)
```

`partitionsIter` *Partition/Composition Iterator*

Description

- Returns an iterator for iterating over partitions/compositions of a numbers.
- Supports random access via the `[[` method.
- GMP support allows for exploration of cases where the number of partitions/compositions is large.
- Use the `next` methods to obtain results in lexicographical order.

Usage

```
partitionsIter(v, m = NULL, ...)
compositionsIter(v, m = NULL, ...)

## Default S3 method:
partitionsIter(v, m = NULL, repetition = FALSE,
               freqs = NULL, target = NULL,
               nThreads = NULL, tolerance = NULL, ...)

## Default S3 method:
compositionsIter(v, m = NULL, repetition = FALSE, freqs = NULL,
                 target = NULL, weak = FALSE, nThreads = NULL,
                 tolerance = NULL, ...)

## S3 method for class 'table'
partitionsIter(
    v, m = NULL, target = NULL, nThreads = NULL, tolerance = NULL, ...
)
## S3 method for class 'table'
compositionsIter(
   v, m = NULL, target = NULL, weak = FALSE, nThreads = NULL, tolerance = NULL, ...
)
```

Arguments

v	Source vector. If v is a positive integer, it will be converted to the sequence 1:v. If v is a negative integer, it will be converted to the sequence v:-1. Only integer and numeric vectors are accepted.
m	Width of the partition. If m = NULL, the length will be determined by the partitioning case (*e.g.* When we are generating distinct partitions of n, the width will be equal to the smallest m such that sum(1:m) >= n).
...	Further arguments passed to methods.
repetition	Logical value indicating whether partitions/compositions should be with or without repetition. The default is FALSE.
freqs	A vector of frequencies used for producing all partitions 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.
target	Number to be partitioned. If NULL, max(v) will be used.
weak	(Compositions only) Logical flag indicating whether to allow terms of the sequence to be zero.
nThreads	Specific number of threads to be used. The default is NULL.
tolerance	A numeric value greater than or equal to zero. This parameter is utilized when a constraint is applied on a numeric vector. The default value is 0 when it can be determined that whole values are being utilized, otherwise it is sqrt(.Machine$double.eps) which is approximately $1.5e-8$. N.B. If the input vector is of type integer, this parameter will be ignored and strict equality will be enforced.

Details

Once you initialize a new iterator, the following methods are available:

`nextIter` Retrieve the **next** lexicographical result

`nextNIter` Pass an integer *n* to retrieve the **next** *n* lexicographical results

`nextRemaining` Retrieve all remaining lexicographical results

`currIter` Returns the current iteration

`startOver` Resets the iterator

`sourceVector` View the source vector

`summary` Returns a list of summary information about the iterator

`front` Retrieve the **first** lexicographical result

`back` Retrieve the **last** lexicographical result

`[[` Random access method. Pass a single value or a vector of valid indices. If a single value is passed, the internal index of the iterator will be updated, however if a vector is passed the internal state will not change. GMP support allows for flexible indexing.

Value

- If `nextIter` is called, a vector is returned
- Otherwise, a matrix with m columns

Note

- If `nThreads` is utilized, it will only take effect if the number of elements requested is greater than some threshold (determined internally). *E.g*:

```
serial   <- partitionsIter(1000, 10)
multi    <- partitionsIter(1000, 10, nThreads = 4)
fetch1e6 <- multi@nextNIter(1e6)  ## much faster than serial@nextNIter(1e6)
fetch1e3 <- multi@nextNIter(1e3)  ## only one thread used... same as serial@nextNIter(1e3)

library(microbenchmark)
microbenchmark(multi@nextNIter(1e6), serial@nextNIter(1e6))
microbenchmark(multi@nextNIter(1e3), serial@nextNIter(1e3))
```

- `nThreads` will be ignored in the following cases (i.e. Generating the n^{th} partition in these cases are currently unavailable):
  - With standard multisets. If zero is the only element with a non-trivial multiplicity, multi-threading is possible.
  - If the source vector is not isomorphic to `1:length(v)`
- The maximum number of partitions/compositions that can be generated at one time is $2^{31} - 1$.
- If an error occurs while generating results, the iterator's internal state is not guaranteed to remain at its previous position. Call `startOver()` before continuing to use the iterator.

Author(s)

Joseph Wood

References

- Lexicographical Order
- Subset Sum Problem
- Partition (number theory)
- Composition (combinatorics))

See Also

partitionsGeneral, compositionsGeneral

Examples

```
a = partitionsIter(0:10, repetition = TRUE)
a@nextIter()
a@nextNIter(3)
a@front()
a@nextRemaining()
a@summary()
a@back()
a[[5]]
a@summary()
a[[c(1, 17, 3)]]
a@summary()

## Multisets... no random access
b = partitionsIter(40, 5, freqs = rep(1:4, 10), target = 80)
b@nextIter()
b@nextNIter(10)
b@summary()
b@nextIter()
b@currIter()
```

---

partitionsRank *Rank Partitions/Compositions*

Description

- Generate the rank (lexicographically) of partitions/compositions. These functions are the complement to partitions/compositionsSample. See the examples below.
- GMP support allows for exploration of partitions/compositions of vectors with many elements.

Usage

```
partitionsRank(..., v, repetition = FALSE, freqs = NULL,
               target = NULL, nThreads = NULL)

compositionsRank(..., v, repetition = FALSE, freqs = NULL,
                 target = NULL, weak = FALSE, nThreads = NULL)
```

Arguments

...	vectors or matrices to be ranked.
v	Source vector. If v is a positive integer, it will be converted to the sequence 1:v. If v is a negative integer, it will be converted to the sequence v:-1. All atomic types are supported (See is.atomic).
repetition	Logical value indicating whether partitions/compositions should be with or without repetition. The default is FALSE.
freqs	A vector of frequencies used for producing all partitions 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.
target	Number to be partitioned. If NULL, max(v) will be used.
weak	(Compositions only) Logical flag indicating whether to allow terms of the sequence to be zero.
nThreads	Specific number of threads to be used. The default is NULL.

Details

These algorithms rely on efficiently ranking the n^{th} lexicographical partition.

Value

A vector of class integer, numeric, or bigz determined by the total number of partitions/compositions

Note

v must be supplied.

Author(s)

Joseph Wood

References

Lexicographical order ranking/unranking

See Also

partitionsSample, compositionsSample

Examples

```
mySamp = partitionsSample(30, 8, TRUE, n = 5, seed = 10, namedSample = TRUE)
myRank = partitionsRank(mySamp, v = 30, repetition = TRUE)
all.equal(as.integer(rownames(mySamp)), myRank)
```

partitionsSample *Sample Partitions/Compositions*

Description

- Generate a specific (lexicographically) or random sample of partitions/compositions of a number.
- Produce results in parallel using the `Parallel` or `nThreads` arguments.
- GMP support allows for exploration of cases where the number of partitions/compositions is large.

Usage

```
partitionsSample(v, m = NULL, ...)
compositionsSample(v, m = NULL, ...)

## Default S3 method:
partitionsSample(
    v, m = NULL, repetition = FALSE, freqs = NULL, target = NULL,
    n = NULL, sampleVec = NULL, seed = NULL,
    nThreads = NULL, namedSample = FALSE, ...
)
## Default S3 method:
compositionsSample(
    v, m = NULL, repetition = FALSE, freqs = NULL, target = NULL,
    weak = FALSE, n = NULL, sampleVec = NULL, seed = NULL,
    nThreads = NULL, namedSample = FALSE, ...
)

## S3 method for class 'table'
partitionsSample(
    v, m = NULL, target = NULL, n = NULL,
    sampleVec = NULL, seed = NULL, nThreads = NULL, namedSample = FALSE, ...
)
## S3 method for class 'table'
compositionsSample(
    v, m = NULL, target = NULL, weak = FALSE, n = NULL,
    sampleVec = NULL, seed = NULL, nThreads = NULL, namedSample = FALSE, ...
)
```

Arguments

v	Source vector. If `v` is a positive integer, it will be converted to the sequence `1:v`. If `v` is a negative integer, it will be converted to the sequence `v:-1`. Only integer and numeric vectors are accepted.

m	Width of the partition. If m = NULL, the length will be determined by the partitioning case (*e.g.* When we are generating distinct partitions of n, the width will be equal to the smallest m such that sum(1:m) >= n).
...	Further arguments passed to methods.
repetition	Logical value indicating whether partitions/compositions should be with or without repetition. The default is FALSE.
freqs	A vector of frequencies used for producing all partitions 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.
target	Number to be partitioned. If NULL, max(v) will be used.
weak	(Compositions only) Logical flag indicating whether to allow terms of the sequence to be zero.
n	Number of partitions/compositions to return. The default is NULL.
sampleVec	A vector of numbers representing the lexicographical partitions/compositions to return. Accepts vectors of class bigz as well as vectors of characters
seed	Random seed initialization. The default is NULL. N.B. If the gmp library is needed, this parameter must be set in order to have reproducible results (*E.g* set.seed() has no effect in these cases).
nThreads	Specific number of threads to be used. The default is NULL.
namedSample	Logical flag. If TRUE, rownames corresponding to the lexicographical partition, will be added to the returned matrix. The default is FALSE.

Details

These algorithms rely on efficiently generating the n^{th} lexicographical partition. This is the process of unranking.

Value

A matrix is returned with each row containing a vector of length m.

Note

- partitionsSample is not available for the following cases:
  - With standard multisets. If zero is the only element with a non-trivial multiplicity, sampling is allowed (*e.g.* partitionsSample(0:100, freqs = c(100, rep(1, 100)), n = 2))
  - If the source vector is not isomorphic to 1:length(v) (*e.g.* v = c(1, 4, 6, 7, 8)).
- n and sampleVec cannot both be NULL.

Author(s)

Joseph Wood

References

- Lexicographical order
- Partition (Number Theory)

Examples

```
partitionsSample(100, 10, n = 5)
partitionsSample(100, 10, seed = 42, n = 5, target = 200)

## retrieve specific results (lexicographically)
partitionsCount(100, 10, TRUE, target = 500)
## [1] 175591757896
partitionsSample(100, 10, TRUE, target = 500,
                 sampleVec = c(1, 1000, 175591757896))
```

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

Description

Prime counting function for counting the prime numbers less than an integer, n, using Legendre's formula. It is based on the the algorithm developed by Kim Walisch found here: kimwalisch/primecount.

Usage

```
primeCount(n, nThreads = NULL)
```

Arguments

`n`	Positive number
`nThreads`	Specific number of threads to be used. The default is `NULL`.

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. https://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)

## Using nThreads
system.time(primeCount(10^10, nThreads = 2))
```

`primeFactorize` *Vectorized Prime Factorization*

Description

Implementation of Pollard's rho algorithm for generating the prime factorization. The algorithm is based on the "factorize.c" source file from the gmp library found here https://gmplib.org.

Usage

```
primeFactorize(v, namedList = FALSE, nThreads = NULL)
```

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`.
`nThreads`	Specific number of threads to be used. The default is `NULL`.

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

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)

## Using nThreads
system.time(primeFactorize(myVec, nThreads = 2))
```

---

primeFactorizeSieve *Generate Prime Factorization for Numbers in a Range*

Description

Generates the prime factorization of all numbers between bound1 and bound2 (if supplied) or all numbers up to bound1.

Usage

```
primeFactorizeSieve(bound1, bound2 = NULL, namedList = FALSE, nThreads = NULL)
```

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.
nThreads	Specific number of threads to be used. The default is NULL.

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 https://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`

Examples

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

## 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)

## Using nThreads
system.time(primeFactorizeSieve(1e4, 5e4, nThreads = 2))
```

`primeSieve` *Generate Prime Numbers*

Description

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`. See this stackoverflow post for an analysis on prime number generation efficiency in R: Generate a list of primes up to a certain number

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

Usage

```
primeSieve(bound1, bound2 = NULL, nThreads = NULL)
```

Arguments

`bound1`	Positive integer or numeric value.
`bound2`	Positive integer or numeric value.
`nThreads`	Specific number of threads to be used. The default is `NULL`.

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

Examples

```
## Primes up to a thousand
primeSieve(100)

## Primes between 42 and 17
primeSieve(42, 17)

## Equivalent to
primeSieve(17, 42)

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

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

## Using nThreads
system.time(primeSieve(1e7, nThreads = 2))
```

stdThreadMax *Max Number of Concurrent Threads*

Description

Wrapper of std::thread::hardware_concurrency(). As stated by cppreference, the returned value should be considered only a hint.

Usage

```
stdThreadMax()
```

Value

An integer representing the number of concurrent threads supported by the user implementation. If the value cannot be determined, 1L is returned.

See Also

detectCores

Examples

```
stdThreadMax()
```

Index

∗ **cartesian**
Cartesian-class, 4
comboGrid, 14
expandGrid, 41
expandGridCount, 43
expandGridIter, 44
expandGridSample, 45
∗ **classes**
Cartesian-class, 4
Combo-class, 5
ComboGroups-class, 19
Constraints-class, 35
Partitions-class, 50
∗ **combinations**
Combo-class, 5
comboCount, 6
comboGeneral, 7
comboIter, 25
comboRank, 30
comboSample, 32
Constraints-class, 35
∗ **combinatorics**
Cartesian-class, 4
Combo-class, 5
comboCount, 6
comboGeneral, 7
comboGrid, 14
comboGroups, 16
ComboGroups-class, 19
comboGroupsCount, 20
comboGroupsIter, 21
comboGroupsSample, 23
comboIter, 25
comboRank, 30
comboSample, 32
Constraints-class, 35
expandGrid, 41
expandGridCount, 43
expandGridIter, 44
expandGridSample, 45
Partitions-class, 50
partitionsCount, 51
partitionsGeneral, 53
partitionsIter, 55
partitionsRank, 58
partitionsSample, 60
∗ **comboGrid**
comboGrid, 14
∗ **comboGroups**
comboGroups, 16
ComboGroups-class, 19
comboGroupsCount, 20
comboGroupsIter, 21
comboGroupsSample, 23
∗ **divisor**
divisorsRcpp, 36
divisorsSieve, 38
numDivisorSieve, 49
primeFactorize, 64
primeFactorizeSieve, 65
∗ **eratosthenes**
primeSieve, 67
∗ **euler**
eulerPhiSieve, 39
∗ **expand.grid**
Cartesian-class, 4
comboGrid, 14
expandGrid, 41
expandGridCount, 43
expandGridIter, 44
expandGridSample, 45
∗ **expandGrid**
Cartesian-class, 4
comboGrid, 14
expandGrid, 41
expandGridCount, 43
expandGridIter, 44
expandGridSample, 45

∗ **expand**
Cartesian-class, 4
comboGrid, 14
expandGrid, 41
expandGridCount, 43
expandGridIter, 44
expandGridSample, 45
∗ **factorization**
divisorsRcpp, 36
divisorsSieve, 38
primeFactorize, 64
primeFactorizeSieve, 65
∗ **factor**
divisorsRcpp, 36
divisorsSieve, 38
numDivisorSieve, 49
primeFactorize, 64
primeFactorizeSieve, 65
∗ **integer**
ComboGroups-class, 19
comboGroupsIter, 21
Partitions-class, 50
partitionsCount, 51
partitionsGeneral, 53
partitionsIter, 55
partitionsRank, 58
partitionsSample, 60
∗ **iterator**
Cartesian-class, 4
Combo-class, 5
ComboGroups-class, 19
comboGroupsIter, 21
comboIter, 25
Constraints-class, 35
expandGridIter, 44
Partitions-class, 50
partitionsIter, 55
∗ **miller rabin**
isPrimeRcpp, 47
∗ **number theory**
divisorsRcpp, 36
divisorsSieve, 38
eulerPhiSieve, 39
isPrimeRcpp, 47
numDivisorSieve, 49
primeCount, 62
primeFactorize, 64
primeFactorizeSieve, 65
primeSieve, 67
∗ **package**
RcppAlgos-package, 3
∗ **partitions**
comboGroups, 16
ComboGroups-class, 19
comboGroupsCount, 20
comboGroupsSample, 23
Partitions-class, 50
partitionsCount, 51
partitionsGeneral, 53
partitionsIter, 55
partitionsRank, 58
partitionsSample, 60
∗ **permutations**
Combo-class, 5
comboCount, 6
comboGeneral, 7
comboIter, 25
comboRank, 30
comboSample, 32
Constraints-class, 35
∗ **phi**
eulerPhiSieve, 39
∗ **pi**
primeCount, 62
∗ **primecount**
primeCount, 62
∗ **prime**
divisorsRcpp, 36
divisorsSieve, 38
eulerPhiSieve, 39
isPrimeRcpp, 47
primeCount, 62
primeFactorize, 64
primeFactorizeSieve, 65
primeSieve, 67
∗ **product**
Cartesian-class, 4
expandGrid, 41
expandGridCount, 43
expandGridIter, 44
expandGridSample, 45
∗ **random**
comboGroupsSample, 23
comboRank, 30
comboSample, 32
expandGridSample, 45

partitionsRank, 58
partitionsSample, 60
∗ **rank**
comboGroupsSample, 23
comboRank, 30
comboSample, 32
expandGridSample, 45
partitionsRank, 58
partitionsSample, 60
∗ **sample**
comboGroupsSample, 23
comboRank, 30
comboSample, 32
expandGridSample, 45
partitionsRank, 58
partitionsSample, 60
∗ **sieve**
divisorsSieve, 38
primeFactorizeSieve, 65
primeSieve, 67
∗ **tau**
numDivisorSieve, 49
∗ **totient**
eulerPhiSieve, 39
∗ **unrank**
comboGroupsSample, 23
comboRank, 30
comboSample, 32
expandGridSample, 45
partitionsRank, 58
partitionsSample, 60
$,Cartesian-method *(*Cartesian-class*)*, 4
$,Combo-method *(*Combo-class*)*, 5
$,ComboApply-method *(*Combo-class*)*, 5
$,ComboGroups-method *(*ComboGroups-class*)*, 19
$,ComboRes-method *(*Combo-class*)*, 5
$,Constraints-method *(*Constraints-class*)*, 35
$,Partitions-method *(*Partitions-class*)*, 50

Cartesian-class, 4
Combo-class, 5
ComboApply-class *(*Combo-class*)*, 5
comboCount, 6
comboGeneral, *7*, 7, *29*
comboGrid, 14, *42*
comboGroups, 16, *22*
ComboGroups-class, 19
comboGroupsCount, 20
comboGroupsIter, 21
comboGroupsSample, 23
comboIter, 25
comboRank, 30, *34*
ComboRes-class *(*Combo-class*)*, 5
comboSample, *31*, 32
compositionsCount *(*partitionsCount*)*, 51
compositionsGeneral, *52*, *58*
compositionsGeneral *(*partitionsGeneral*)*, 53
compositionsIter *(*partitionsIter*)*, 55
compositionsRank *(*partitionsRank*)*, 58
compositionsSample, *59*
compositionsSample *(*partitionsSample*)*, 60
Constraints-class, 35

detectCores, *69*
divisorsRcpp, 36, *39*
divisorsSieve, 38, *66*

eulerPhiSieve, 39
expand.grid, *41*
expandGrid, *15*, 41, *43*, *45*
expandGridCount, 43
expandGridIter, 44
expandGridSample, 45

factorize, *65, 66*

is.atomic, *6*, *9*, *16*, *20, 21*, *23*, *27*, *31*, *33*, *59*
isprime, *48*
isPrimeRcpp, 47, *64*

numDivisorSieve, 49

Partitions-class, 50
partitionsCount, 51
partitionsGeneral, *10*, *52*, 53, *58*
partitionsIter, 55
partitionsRank, 58
partitionsSample, *59*, 60
permuteCount *(*comboCount*)*, 6
permuteGeneral, *7*, *29*
permuteGeneral *(*comboGeneral*)*, 7
permuteIter *(*comboIter*)*, 25
permuteRank, *34*
permuteRank *(*comboRank*)*, 30

permuteSample, *31*
permuteSample *(*comboSample*)*, 32
primeCount, 62
primeFactorize, *37*, *48*, 64, *66*
primeFactorizeSieve, *39*, *65*, 65
primeSieve, *63*, 67

RcppAlgos *(*RcppAlgos-package*)*, 3
RcppAlgos-package, 3
rep, *6*, *9*, *27*, *31*, *33*, *52*, *54*, *56*, *59*, *61*

stdThreadMax, 69

vapply, *10, 11*, *28, 29*, *34*