Package 'partitions'

July 21, 2022

Type Package

Title Additive Partitions of Integers

Version 1.10-7

Depends R (>= 3.6.0)

Maintainer Robin K. S. Hankin <hankin.robin@gmail.com>

Imports gmp, polynom, sets, mathjaxr, Rdpack

Description Additive partitions of integers. Enumerates the partitions, unequal partitions, and restricted partitions of an integer; the three corresponding partition functions are also given. Set partitions and now compositions and riffle shuffles are included.

Suggests testthat

License GPL

URL https://github.com/RobinHankin/partitions

BugReports https://github.com/RobinHankin/partitions/issues

RdMacros mathjaxr,Rdpack

R topics documented:

partitions-package *Integer partitions*

Description

Routines to enumerate all partitions of an integer; includes restricted and unequal partitions.

Details

This package comprises eight functions: `P()`, `Q()`, `R()`, and `S()` give the number of partitions, unequal partitions, restricted partitions, and block partitions of an integer.

Functions `parts()`, `diffparts()`, `restrictedparts()`, and `blockparts()` enumerate these partitions.

Function `conjugate()` gives the conjugate of a partition and function `durfee()` gives the size of the Durfee square.

NB the emphasis in this package is terse, efficient C code. This means that there is a minimum of argument checking. For example, function `conjugate()` assumes that the partition is in standard form (i.e. nonincreasing); supplying a vector in nonstandard form will result in garbage being returned silently. Note that a block partition is not necessarily in standard form.

Author(s)

Robin K. S. Hankin

References

- G. E. Andrews 1998 *The Theory of Partitions*, Cambridge University Press
- M. Abramowitz and I. A. Stegun 1965. *Handbook of Mathematical Functions*, New York: Dover
- G. H. Hardy and E. M. Wright 1985 *An introduction to the theory of numbers*, Clarendon Press: Oxford (fifth edition)
- R. K. S. Hankin 2006. "Additive integer partitions in R". *Journal of Statistical Software*, Volume 16, code snippet 1
- R. K. S. Hankin 2007. "Urn sampling without replacement: enumerative combinatorics in R". *Journal of Statistical Software*, Volume 17, code snippet 1
- R. K. S. Hankin 2007. "Set partitions in R". *Journal of Statistical Software*, Volume 23, code snippet 2

Examples

```
parts(5)
diffparts(9)
restrictedparts(15,10)
P(10,give=TRUE)
Q(10,give=TRUE)
R(5,10)
```

as.matrix.partition *Coerce partitions to matrices and vice versa*

Description

Coercion to and from partitions

Usage

```
## S3 method for class 'partition'
as.matrix(x, ...)
as.partition(x, ...)
```

Arguments

`x`	Object to be coerced
`...`	Further arguments

Author(s)

Robin K. S. Hankin

Examples

```
as.matrix(parts(5))
```

bin *Sundry binary functionality*

Description

Utilities to convert things to binary

Usage

```
tobin(n, len, check=TRUE)
todec(bin)
comptobin(comp, check=TRUE)
bintocomp(bin,  use.C=TRUE, check=TRUE)
```

Arguments

`n`	Integer, to be converted to binary by function `tobin()`
`len`	Length of the binary vector returned by function `tobin()`
`bin`	Binary: a vector of `0`s and `1`s
`comp`	A composition
`check`	Boolean, with default `TRUE` meaning to perform various checks
`use.C`	Boolean, with default `TRUE` meaning to use `C`

Details

These functions are not really intended for the end user; they are used in `nextcomposition()`.

- Function `tobin()` converts integer `n` to a binary string of length `len`
- Function `todec()` converts a binary string to decimal, so `todec(tobin(n,i))==n`, provided `i` is big enough
- Function `comptobin()` converts a composition to binary
- Function `bintocomp()` converts a binary string to a composition

Author(s)

Robin K. S. Hankin

References

Wikipedia contributors, 2020. "Composition (combinatorics) — Wikipedia, The Free Encyclopedia", https://en.wikipedia.org/w/index.php?title=Composition_(combinatorics)&oldid=944285378

Examples

```
tobin(10,5)
todec(tobin(10,5))
comptobin(c(1,1,4))
bintocomp(c(1,1,0,0,1,1,1,1))
```

`conjugate` *Conjugate partitions and Durfee squares*

Description

Given a partition, provide its conjugate or Durfee square

Usage

```
conjugate(x, sorted = TRUE)
durfee(x, sorted = TRUE)
durfee_sorted(x)
```

Arguments

`x`	Either a vector describing a partition or a matrix whose columns are partitions
`sorted`	A logical indicating whether the data is already in standard form. That is to say, are the data within each column sorted in decreasing order?

Details

Conjugation is described in Andrews, and (e.g.) Hardy and Wright.

The *conjugate* of a partition may be calculated by taking its Ferrers diagram and considering the partition defined by columns instead of rows. This may be visualised by flipping the Ferrers diagram about the leading diagonal.

Essentially, `conjugate()` carries out R idiom

`rev(cumsum(table(factor(a[a>0],levels=max(a):1))))`

but is faster.

The "Durfee square" of a partition is defined on page 281 of Hardy and Wright. It is the largest square of nodes contained in the partition's Ferrers graph. Function `durfee()` returns the length of the side of the Durfee square, which Andrews denotes $d(\lambda)$. It is equivalent to R idiom

`function(a){sum(a>=1:length(a))}`

but is faster.

Value

Returns either a partition in standard form, or a matrix whose columns are partitions in standard form.

Note

If argument `x` is not non-increasing, you must use the `sorted = FALSE` flag. Otherwise, these functions will not work and will silently return garbage. Caveat emptor! (output from `blockparts()` is not necessarily non-increasing)

Author(s)

Robin K. S. Hankin

Examples

```
parts(5)
conjugate(parts(5))

restrictedparts(6,4)
conjugate(restrictedparts(6,4))

durfee(10:1)

# A partition in nonstandard form --- use `sorted = FALSE`
x <- parts(5)[sample(5),]
durfee(x, sorted = FALSE)
conjugate(x, sorted = FALSE)

# Suppose one wanted partitions of 8 with no part larger than 3:

conjugate(restrictedparts(8,3))

# (restrictedparts(8,3) splits 8 into at most 3 parts;
# so no part of the conjugate partition is larger than 3).
```

`nextpart` *Next partition*

Description

Given a partition, return the "next" one; or determine whether it is the last one.

Usage

```
  nextpart(part, check=TRUE)
islastpart(part)
 firstpart(n)
  nextdiffpart(part, check=TRUE)
islastdiffpart(part)
 firstdiffpart(n)
  nextrestrictedpart(part, check=TRUE)
islastrestrictedpart(part)
 firstrestrictedpart(n, m, include.zero=TRUE)
  nextblockpart(part, f, n=sum(part), include.fewer=FALSE, check=TRUE)
islastblockpart(part, f, n=NULL     , include.fewer=FALSE)
 firstblockpart(      f, n=NULL     , include.fewer=FALSE)
  nextcomposition(comp, restricted, include.zero=TRUE, check=TRUE)
islastcomposition(comp, restricted, include.zero=TRUE)
 firstcomposition(n,    m=NULL    , include.zero=TRUE)
```

Arguments

`part,comp`	A partition or composition
`check`	Boolean, with default `TRUE` meaning to carry out various safety checks; the `next()` functions use `C` calls which might crash the session with some inputs
`f, n, include.fewer, m, include.zero`	Other arguments as per the vectorized version
`restricted`	In function `nextcomposition()` and `islastcomposition()`, Boolean, with `TRUE` meaning to consider compositions of fixed length [eg, to iterate through the columns of `compositions(6,3)`], and `FALSE` meaning to consider compositions of any length [eg to iterate through the columns of `compositions(6)`]

Details

These functions are intended to enumerate partitions one at a time, eliminating the need to store a huge matrix. This is useful for optimization over large domains and makes it possible to investigate larger partitions than is possible with the vectorized codes.

The idea is to use a `first...()` function to generate the first partition, then iterate using a `next...()` function, stopping when the `islast...()` function returns `TRUE`.

An example is given below, in which the "scrabble" problem is solved; note the small size of the sample space. More examples are given in the `tests/aab.R` file.

Note

Functions `nextpart()` and `nextdiffpart()` require a vector of the right length: they require and return a partition padded with zeros. Functions `nextrestrictedpart()` and `nextblockpart()` work with partitions of the specified length. Function `nextcomposition()` truncates any zeros at the end of the composition. This behaviour is inherited from the `C` code.

In functions `nextcomposition()` and `firstcomposition()`, argument `include.zero` is ignored if `restricted` is `FALSE`.

I must say that the performance of these functions is terrible; they are much much slower than their vectorized equivalents. The magnitude of the difference is much larger than I expected. Heigh ho. Frankly you would better off working directly in `C`.

Author(s)

Robin K. S. Hankin

See Also

`parts`

Examples

```
# Do the optimization in scrabble vignette, one partition at a time:
# (but with a smaller letter bag)
scrabble <- c(a=9 , b=2 , c=2 , d=4 , e=12 , f=2 , g=3)

f <- function(a){prod(choose(scrabble,a))/choose(sum(scrabble),7)}
bestsofar <- 0
a <- firstblockpart(scrabble,7)
while(!islastpart(a)){
  jj <- f(a)
  if(jj>bestsofar){
    bestsofar <- jj
    bestpart <- a
  }
  a <- nextblockpart(a,scrabble)
}
```

`P` *Number of partitions of an integer*

Description

Given an integer, `P()` returns the number of additive partitions, `Q()` returns the number of unequal partitions, and `R()` returns the number of restricted partitions. Function `S()` returns the number of block partitions.

Usage

```
P(n, give = FALSE)
Q(n, give = FALSE)
R(m, n, include.zero = FALSE)
S(f, n = NULL, include.fewer = FALSE)
```

Arguments

`n`	Integer whose partition number is desired. In function `S()`, the default of `NULL` means to return the number of partitions of any size
`m`	In function `R()`, the order of the decomposition
`give`	Boolean, with default `FALSE` meaning to return just `P(n)` or `Q(n)` and `TRUE` meaning to return `P(1:n)` or `Q(1:n)` (this option takes no extra computation)
`include.zero`	In `restrictedparts()`, Boolean with default `FALSE` meaning to count only partitions of n into *exactly* m parts; and `TRUE` meaning to include partitions of n into *at most* m parts (because parts of zero are included)
`include.fewer`	In function `blockparts()`, Boolean with default `FALSE` meaning to return partitions into *exactly* `n` and `TRUE` meaning to return partitions into *at most* `n`
`f`	In function `S()`, the stack vector

Details

Functions `P()` and `Q()` use Euler's recursion formula. Function `R()` enumerates the partitions using Hindenburg's method (see Andrews) and counts them until the recursion bottoms out.

Function `S()` finds the coefficient of x^n in the generating function $\prod_{i=1}^{L}\sum_{j=0}^{f_i} x^j$, where L is the length of `f`, using the **polynom** package.

All these functions return a double.

Note

Functions `P()` and `Q()` use `unsigned long long` integers, a type which is system-dependent. For me, `P()` works for n equal to or less than 416, and `Q()` works for n less than or equal to 792. YMMV; none of the methods test for overflow, so use with care!

Author(s)

Robin K. S. Hankin; `S()` is due to an anonymous JSS referee

Examples

```
P(10,give=TRUE)
Q(10,give=TRUE)
R(10,20,include.zero=FALSE)
R(10,20,include.zero=TRUE)

S(1:4,5)
```

parts — *Enumerate the partitions of an integer*

Description

Given an integer, return a matrix whose columns enumerate various partitions.

Function `parts()` returns the unrestricted partitions; function `diffparts()` returns the unequal partitions; function `restrictedparts()` returns the restricted partitions; function `blockparts()` returns the partitions subject to specified maxima; and function `compositions()` returns all compositions of the argument.

Usage

```
parts(n)
diffparts(n)
restrictedparts(n, m, include.zero=TRUE, decreasing=TRUE)
blockparts(f, n=NULL, include.fewer=FALSE)
compositions(n, m=NULL, include.zero=TRUE)
multiset(v,n=length(v))
mset(v)
multinomial(v)
allbinom(n,k)
```

Arguments

`n`	Integer to be partitioned. In function `blockparts()`, the default of `NULL` means to return all partitions of any size
`m`	In functions `restrictedparts()` and `compositions()`, the order of the partition
`include.zero`	In functions `restrictedparts()` and `compositions()`, Boolean with default `FALSE` meaning to include only partitions of n into *exactly* m parts; and `TRUE` meaning to include partitions of n into *at most* m parts (because zero parts are included)
`include.fewer`	In function `blockparts()`, Boolean with default `FALSE` meaning to return vectors whose sum is *exactly* `n` and `TRUE` meaning to return partitions whose sum is *at most* `n`
`decreasing`	In `restrictedparts()`, Boolean with default `TRUE` meaning to return partitions whose parts are in decreasing order and `FALSE` meaning to return partitions in lexicographical order, as appearing in Hindenburg's algorithm. Note that setting to `decreasing` to `FALSE` has the effect of making `conjugate()` return garbage
`f`	In function `blockparts()`, a vector of strictly positive integers that gives the maximal number of blocks; see details
`v`	In function `multiset()`, an integer vector representing a multiset. Argument `n` is the size of the sample to be taken
`k`	In function `allbinom()`, the size of the set to be chosen; arguments match those of `choose()`

Details

- Function `parts()` uses the algorithm in Andrews. Function `diffparts()` uses a very similar algorithm that I have not seen elsewhere. These functions behave strangely if given an argument of zero.
- Function `restrictedparts()` uses the algorithm in Andrews, originally due to Hindenburg. For partitions into at most m parts, the same Hindenburg's algorithm is used but with a start vector of `c(rep(0,m-1),n)`.

  Functions `parts()` and `restrictedparts()` overlap in functionality. Note, however, that they can return identical partitions but in a different order: `parts(6)` and `restrictedparts(6,6)` for example.

  If $m > n$, the partitions are padded with zeros.
- Function `blockparts()` enumerates the compositions of an integer subject to a maximum criterion: given vector $y = (y_1, \ldots, y_n)$ all sets of $a = (a_1, \ldots, a_n)$ satisfying $\sum_{i=1}^{p} a_i = n$ subject to $0 \leq a_i \leq y_i$ for all i are given in lexicographical order. If argument `y` includes zero elements, these are treated consistently (ie a position with zero capacity).

  If `n` takes its default value of `NULL`, then the restriction $\sum_{i=1}^{p} a_i = n$ is relaxed (so that the numbers may sum to anything). Note that these solutions are not necessarily in standard form, so functions `durfee()` and `conjugate()` may fail.
- With a single argument, `compositions(n)` returns all 2^{n-1} ways of partitioning an integer; thus `4+1+1` is distinct from `1+4+1` or `1+1+4`.

  With two arguments, `compositions(n,m)` returns all nonnegative solutions to $x_1 + \cdots + x_m = n$.

  This function is different from all the others in the package in that it is written in R; it is not clear that C would be any faster.
- Function `multiset()` returns all ways of ordering a multiset (`mset()` is a low-level helper function).
- Function `multinomial(v)` returns all ways of partitioning a set into *distinguishable* boxes of capacities `v[1], v[2],...,v[n]`. The number of columns is given by the multinomial coefficient $\binom{\sum v_i}{v_1\, v_2 \ldots v_n}$.
- Function `allbinom(n,k)` is provided for convenience; it enumerates the ways of choosing k objects from `n`.

Note

These vectorized functions return a matrix whose columns are the partitions. If this matrix is too large, consider enumerating the partitions individually using the functionality documented in `nextpart.Rd`.

One commonly encountered idiom is `blockparts(rep(n,n),n)`, which is equivalent to `compositions(n,n)` [Sloane's `A001700`].

If you have a *minimum* number of balls in each block, a construction like

```
x <- c(1, 1, 2, 1)
y <- c(2, 3, 4, 5)
sweep(blockparts(y - x, 7 - sum(x)), 1, x, "+")

##: [1,] 2 1 2 1 1 2 1 1 1
##: [2,] 2 3 1 2 1 1 2 1 1
##: [3,] 2 2 3 3 4 2 2 3 2
##: [4,] 1 1 1 1 1 2 2 2 3
```

can be helpful (that is, subtract off the minimum number of balls and add them back again at the end).

```
    blockparts(c(4,3,3,2),5)  # Knuth's example, pre-fascicle 3a, p16
    multiset(c(1,2,2,3))      # also Knuth
```

Author(s)

Robin K. S. Hankin

References

- G. E. Andrews. “The theory of partitions”, Cambridge University Press, 1998
- R. K. S. Hankin 2006. “Additive integer partitions in R”. *Journal of Statistical Software*, Volume 16, code snippet 1
- R. K. S. Hankin 2007. “Urn sampling without replacement: enumerative combinatorics in R”. *Journal of Statistical Software*, Volume 17, code snippet 1
- R. K. S. Hankin 2007. “Set partitions in R”. *Journal of Statistical Software*, Volume 23, code snippet 2
- N. J. A. Sloane, 2008, The On-Line Encyclopedia of Integer Sequences. Sequence A001700
- D. Knuth, 2004. The art of computer programming, pre-fascicle 2B “Generating all permutations”

See Also

`nextpart`

Examples

```
parts(7)

##: [1,] 7 6 5 5 4 4 4 3 3 3 3 2 2 2 1
##: [2,] 0 1 2 1 3 2 1 3 2 2 1 2 2 1 1
##: [3,] 0 0 0 1 0 1 1 1 2 1 1 2 1 1 1
##: [4,] 0 0 0 0 0 0 1 0 0 1 1 1 1 1 1
##: [5,] 0 0 0 0 0 0 0 0 0 0 1 0 1 1 1
##: [6,] 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
##: [7,] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
P(7)
##: [1] 15
diffparts(9)

##: [1,] 9 8 7 6 6 5 5 4
##: [2,] 0 1 2 3 2 4 3 3
##: [3,] 0 0 0 0 1 0 1 2
Q(9)
##: [1] 8
restrictedparts(9, 4)

##: [1,] 9 8 7 6 5 7 6 5 4 5 4 3 6 5 4 4 3 3
##: [2,] 0 1 2 3 4 1 2 3 4 2 3 3 1 2 3 2 3 2
##: [3,] 0 0 0 0 0 1 1 1 1 2 2 3 1 1 1 2 2 2
##: [4,] 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 2
```

```
R(4, 9, include.zero = TRUE)
##: [1] 18
blockparts(1:4, 5)

##: [1,] 1 1 0 1 1 0 1 0 1 1 0 1 0 0 1 0 1 0 0 1 0 0
##: [2,] 2 1 2 2 1 2 0 1 2 1 2 0 1 0 1 2 0 1 0 0 1 0
##: [3,] 2 3 3 1 2 2 3 3 0 1 1 2 2 3 0 0 1 1 2 0 0 1
##: [4,] 0 0 0 1 1 1 1 1 2 2 2 2 2 2 3 3 3 3 3 4 4 4
S(1:4, 5)
##: [1] 22
compositions(5, 3)

##: [1,] 5 4 3 2 1 0 4 3 2 1 0 3 2 1 0 2 1 0 1 0 0
##: [2,] 0 1 2 3 4 5 0 1 2 3 4 0 1 2 3 0 1 2 0 1 0
##: [3,] 0 0 0 0 0 0 1 1 1 1 1 2 2 2 2 3 3 3 4 4 5
S(rep(5, 3), 5)
##: [1] 21
setparts(4)

##: [1,] 1 1 1 1 2 1 1 1 1 1 1 2 2 2 1
##: [2,] 1 1 1 2 1 2 1 2 2 1 2 1 1 3 2
##: [3,] 1 2 1 1 1 2 2 1 3 2 1 3 1 1 3
##: [4,] 1 1 2 1 1 1 2 2 1 3 3 1 3 1 4
setparts(c(1, 2, 2))

##: [1,] 1 1 1 1 1 1 1 1 1 1 1 1 3 3 3
##: [2,] 2 2 3 1 1 1 2 2 3 2 2 3 1 1 1
##: [3,] 3 2 2 3 2 2 1 1 1 3 2 2 2 1 2
##: [4,] 2 3 2 2 3 2 3 2 2 1 1 1 2 2 1
##: [5,] 1 1 1 2 2 3 2 3 2 2 3 2 1 2 2
multinomial(c(a = 1, b = 2, c = 1))

##: a 1 1 1 2 2 3 4 3 4 2 3 4
##: b 2 2 3 1 1 1 1 1 1 3 2 2
##: b 3 4 4 3 4 2 2 4 3 4 4 3
##: c 4 3 2 4 3 4 3 2 2 1 1 1
```

---

perms *Enumerate the permutations of a vector*

Description

Given an integer `n`, return a matrix whose columns enumerate various permutations of `1:n`.

Function `perms()` returns all permutations in lexicographic order; function `plainperms()` returns all permutations by repeatedly exchanging adjacent pairs.

Usage

```
perms(n)
plainperms(n)
```

Arguments

n	Integer argument; permutations of `1:n` returned

Note

Comments in the C code; algorithm lifted from ‘fasc2b.pdf’.

Author(s)

D. E. Knuth; C and R transliteration by Robin K. S. Hankin

References

- D. E. Knuth 2004. “The art of computer programming, pre-fascicle 2B. A draft of section 7.2.1.2: Generating all permutations”. https://www-cs-faculty.stanford.edu/~knuth/taocp.html

See Also

parts

Examples

```
perms(4)
summary(perms(5))

# Knuth's Figure 18:
matplot(t(apply(plainperms(4),2,order)),
        type='l', lty=1, lwd=5, asp=1,
        frame=FALSE, axes=FALSE, ylab="", col=gray((1:5)/5))
```

---

print.partition — *Print methods for partition objects and equivalence objects*

Description

A print method for partition objects, summary partition objects, and equivalence classes. Includes various configurable options

Usage

```
## S3 method for class 'partition'
print(x, mat = getOption("matrixlike"), h = getOption("horiz"), ...)
## S3 method for class 'summary.partition'
print(x, ...)
## S3 method for class 'equivalence'
print(x, sep = getOption("separator"), ...)
```

Arguments

x	Object to be printed: an object of class either partition or summary.partition
mat	Boolean, with TRUE meaning to print like a matrix, and any other value meaning to print without column names (which usually results in more compact appearance)

`h`	Boolean governing the orientation of the printed matrix, with `TRUE` meaning to print with the rows being the partitions and any other value (the default) meaning to print the transpose
`sep`	Character vector, with special value of `NULL` interpreted as a comma; see examples section
`...`	Further arguments provided for compatibility

Author(s)

Robin K. S. Hankin

Examples

```
print(parts(5))

summary(parts(7))

listParts(3)
options(separator="")
listParts(5)
```

`riffle` *Riffle shuffles*

Description

Enumeration of riffle shuffles

Usage

```
genrif(v)
riffle(p,q=p)
```

Arguments

`p,q,v`	In function `riffle()`, integers `p,q` specify the length of the two increasing sequences. In function `genrif()`, the elements of `v` specify the lengths of all the increasing sequences: there are `sum(v)` cards in the pack

Details

A *riffle shuffle* is a permutation of integers $1, 2, \ldots, n$ containing one or two rising sequences.

A *generalized riffle shuffle*, or r*-riffle shuffle*, contains at most r rising sequences. This is not implemented in the package (earlier versions included a buggy version; the difficulty is ensuring that sequences do not appear more than once).

- `riffle(p,q)` returns all riffle shuffles with rising sequences of `1:p` and `(p+1):q`
- `genrif(v)` returns all riffle shuffles with rising sequences having lengths the entries of `v`, the deck being numbered consecutively

Value

Returns a matrix of class `partition` with columns being riffle shuffles

Note

When we say "contains r rising sequences" we generally mean "contains *at most* r rising sequences"

Author(s)

Robin K. S. Hankin

See Also

`parts`

Examples

```
riffle(3, 4)

##: [1,] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
##: [2,] 2 2 2 2 2 4 4 4 4 4 4 4 4 4 4 1 1 1 1 1 1 1 1 1 1 5 5 5 5 5 5 5 5 5 5
##: [3,] 3 4 4 4 4 2 2 2 2 5 5 5 5 5 5 2 2 2 2 5 5 5 5 5 5 1 1 1 1 1 1 6 6 6 6
##: [4,] 4 3 5 5 5 3 5 5 5 2 2 2 6 6 6 3 5 5 5 2 2 2 6 6 6 2 2 2 6 6 6 1 1 1 7
##: [5,] 5 5 3 6 6 5 3 6 6 3 6 6 2 2 7 5 3 6 6 3 6 6 2 2 7 3 6 6 2 2 7 2 2 7 1
##: [6,] 6 6 6 3 7 6 6 3 7 6 3 7 3 7 2 6 6 3 7 6 3 7 3 7 2 6 3 7 3 7 2 3 7 2 2
##: [7,] 7 7 7 7 3 7 7 7 3 7 7 3 7 3 3 7 7 7 3 7 7 3 7 3 3 7 7 3 7 3 3 7 3 3 3
genrif(1:3)

##: [1,] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 4 4 4 4 4 4 4
##: [2,] 2 2 2 2 4 4 4 4 4 4 1 1 1 1 3 3 3 3 4 4 4 4 4 4 4 4 4 4 4 4 1 1 1 1 1 1 2
##: [3,] 3 4 4 4 2 2 2 5 5 5 3 4 4 4 1 4 4 4 1 1 1 3 3 3 5 5 5 5 5 5 2 2 2 5 5 5 1
##: [4,] 4 3 5 5 3 5 5 2 2 6 4 3 5 5 4 1 5 5 3 5 5 1 5 5 1 1 3 3 6 6 3 5 5 2 2 6 3
##: [5,] 5 5 3 6 5 3 6 3 6 2 5 5 3 6 5 5 1 6 5 3 6 5 1 6 3 6 1 6 1 3 5 3 6 3 6 2 5
##: [6,] 6 6 6 3 6 6 3 6 3 3 6 6 6 3 6 6 6 1 6 6 3 6 6 1 6 3 6 1 3 1 6 6 3 6 3 3 6

##: [1,] 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
##: [2,] 2 2 2 2 2 2 2 2 2 2 2 5 5 5 5 5 5 5 5 5 5 5 5
##: [3,] 1 1 3 3 3 5 5 5 5 5 5 1 1 1 2 2 2 2 2 2 6 6 6
##: [4,] 5 5 1 5 5 1 1 3 3 6 6 2 2 6 1 1 3 3 6 6 1 2 2
##: [5,] 3 6 5 1 6 3 6 1 6 1 3 3 6 2 3 6 1 6 1 3 2 1 3
##: [6,] 6 3 6 6 1 6 3 6 1 3 1 6 3 3 6 3 6 1 3 1 3 3 1
```

`setparts` *Set partitions*

Description

Enumeration of set partitions

Usage

```
setparts(x)
listParts(x,do.set=FALSE)
vec_to_set(vec)
vec_to_eq(vec)
```

Arguments

`x`	If a vector of length 1, the size of the set to be partitioned. If a vector of length greater than 1, return all equivalence relations with equivalence classes with sizes of the elements of `x`. If a matrix, return all equivalence classes with sizes of the columns of `x`
`do.set`	Boolean, with `TRUE` meaning to return the set partitions in terms of sets (as per **sets** package) and default `FALSE` meaning to present the result in terms of equivalence classes
`vec`	An integer vector representing a set partition

Details

A *partition* of a set $S = \{1, \ldots, n\}$ is a family of sets $T_1, \ldots, T_k$ satisfying

- $i \neq j \longrightarrow T_i \cap T_j = \emptyset$
- $\cup_{i=1}^{k} T_k = S$
- $T_i \neq \emptyset$ for $i = 1, \ldots, k$

The induced *equivalence relation* has $i \sim j$ if and only if i and j belong to the same partition. Equivalence classes of $S = \{1, \ldots, n\}$ may be listed using `listParts()`. Thus

There are exactly fifteen ways to partition a set of four elements:

$$\begin{array}{l}
(1234) \\
(123)(4), (124)(3), (134)(2), (234)(1) \\
(12)(34), (13)(24), (14)(23) \\
(12)(3)(4), (13)(2)(4), (23)(1)(4), (24)(1)(3), (34)(1)(2) \\
(1)(2)(3)(4)
\end{array}$$

Note that $(12)(3)(4)$ is the same partition as, for example, $(3)(4)(21)$ as the equivalence relation is the same.

Consider partitions of a set S of five elements (named $1, 2, 3, 4, 5$) with sizes 2,2,1. These may be enumerated as follows:

```
> u <- c(2,2,1)
> setparts(u)

[1,] 1 1 1 1 1 1 1 1 1 1 1 1 3 3 3
[2,] 2 2 3 1 1 1 2 2 3 2 2 3 1 1 1
[3,] 3 2 2 3 2 2 1 1 1 3 2 2 2 1 2
[4,] 2 3 2 2 3 2 3 2 2 1 1 1 2 2 1
[5,] 1 1 1 2 2 3 2 3 2 2 3 2 1 2 2
```

See how each column has two 1s, two 2s and one 3. This is because the first and second classes have size two, and the third has size one.

The first partition, `x=c(1,2,3,2,1)`, is read "class 1 contains elements 1 and 5 (because the first and fifth element of `x` is 1); class 2 contains elements 2 and 4 (because the second and fourth element of `x` is 2); and class 3 contains element 3 (because the third element of `x` is 3)". Formally, class `i` has elements `which(x==u[i])`.

You can change the print method by setting, eg, `option(separator="")`.

Functions `vec_to_set()` and `vec_to_eq()` are low-level helper functions. These take an integer vector, typically a column of a matrix produced by `setparts()` and return their set representation.

Value

Returns a matrix each of whose columns show a set partition; an object of class `"partition"`. Type `?print.partition` to see how to change the options for printing.

Note

The **clue** package by Kurt Hornik contains functionality for partitions (specifically `cl_meet()` and `cl_join()`) which might be useful. Option `do.set` invokes functionality from the **sets** package by Meyer et al.

Note carefully that `setparts(c(2,1,1))` does *not* enumerate the ways of placing four numbered balls in three boxes of capacities 2,1,1. This is because there are two boxes of capacity 1, and swapping the balls between these boxes gives the same set partition (because sets are unordered). To do this, use `multinomial(c(a=2,b=1,c=1))`. See the `setparts` vignette for more details.

Author(s)

Luke G. West (C++) and Robin K. S. Hankin (R); `listParts()` provided by Diana Tichy

References

- R. K. S. Hankin 2006. *Additive integer partitions in* R. Journal of Statistical Software, Code Snippets 16(1)
- R. K. S. Hankin 2007. "Set partitions in R". *Journal of Statistical Software*, Volume 23, code snippet 2
- Kurt Hornik (2017). *clue: Cluster ensembles*. R package version 0.3-53. `https://CRAN.R-project.org/package=clue`
- Kurt Hornik (2005). *A CLUE for Cluster Ensembles*. Journal of Statistical Software 14/12. doi:10.18637/jss.v014.i12

See Also

`parts`, `print.partition`

Examples

```
setparts(4)                # all partitions of a set of 4 elements

setparts(c(3,3,2))         # all partitions of a set of 8 elements
                           # into sets of sizes 3,3,2.


listParts(c(2,2,1))        # all 15 ways of defining subsets of
                           # {1,2,3,4,5} with sizes 2,2,1
```

```
jj <- restrictedparts(5,3)
setparts(jj)              # partitions of a set of 5 elements into
                          # at most 3 sets

listParts(jj)             # The induced equivalence classes




jj <- restrictedparts(6,3,TRUE)
setparts(jj)              # partitions of a set of 6 elements into
ncol(setparts(jj))        # _exactly_ 3 sets; cf StirlingS2[6,3]==90


setparts(conjugate(jj))   # partitions of a set of 5 elements into
                          # sets not exceeding 3 elements


setparts(diffparts(5))    # partitions of a set of 5 elements into
                          # sets of different sizes
```

`summary.partition` *Provides a summary of a partition*

Description

Provides a summary of an object of class `partition`: usually the first and last few partitions (columns)

Usage

```
## S3 method for class 'partition'
summary(object, ...)
```

Arguments

`object`	Partition
`...`	Further arguments; see details section below

Details

The ellipsis arguments are used to pass how many columns at the start and the end of the matrix are selected; this defaults to 10.

The function is designed to behave as expected: if there is an argument named “n”, then this is used. If there is no such argument, the first one is used.

Value

A summary object is a list, comprising three elements:

`shortened`	Boolean, with `TRUE` meaning that the middle section of the matrix is omitted, and `FALSE` meaning that the entire matrix is returned because `n` is too big
`n`	Number of columns to return at the start and the end of the matrix
`out`	Matrix returned: just the first and last `n` columns (if `shortened` is `TRUE`), or the whole matrix if not

Author(s)

Robin K. S. Hankin

Examples

```
summary(parts(7))

summary(parts(11),3)
```

Index