Package ‘rpartitions’

August 29, 2016

Title Code for integer partitioning
Description Provides algorithms for randomly sampling a feasible set defined
       by a given total and number of elements using integer partitioning.
Version 0.1
Author Ken Locey, Daniel McGlinn
Maintainer Daniel McGlinn <danmcglinn@gmail.com>
Depends R (>= 2.15.1), hash
Suggests testthat (>= 0.2)
NeedsCompilation yes
Repository CRAN
URL https://github.com/klocey/partitions
License MIT
LazyData true
Collate ‘rpartitions.R’ ‘rpartitions-package.r’
Date/Publication 2013-12-11 07:34:59

R topics documented:

  bottom_up ................................................................. 2
  conjugate ............................................................... 2
  divide_and_conquer .................................................... 3
  get_multiplicity ....................................................... 3
  get_rand_int ............................................................ 4
  last ......................................................................... 5
  multiplicity ............................................................. 5
  NrParts ................................................................. 6
  P ........................................................------------- 6
  rand_partitions ........................................................ 7
  rpartitions ............................................................. 8
  top_down ............................................................... 8

Index 10
**bottom_up**

Bottom up method of generating uniform random partitions of $Q$ having $N$ parts.

**Description**

Bottom up method of generating uniform random partitions of $Q$ having $N$ parts.

**Usage**

```plaintext
bottom_up(part, Q, D, rand_int, use_c, use_hash)
```

**Arguments**

- **part**
  a list to hold the partition
- **Q**
  the total sum of the partition
- **D**
  a dictionary for the number of partitions of $Q$ having $N$ or less parts (or $N$ or less as the largest part), i.e. $P(Q + N, N)$.
- **rand_int**
  a number representing a member of the feasible set
- **use_c**
  boolean if TRUE then compiled c code is used
- **use_hash**
  boolean, if TRUE then a hash table is used

**Examples**

```plaintext
bottom_up(c(5, 4), 4, list(), 1, TRUE, FALSE)
```

---

**conjugate**

Find the conjugate of an integer partition Recoded (originally on 24-Apr-2013) from the Sage source code:

http://www.sagenb.org/src/combinat/partition.py

**Description**

Find the conjugate of an integer partition Recoded (originally on 24-Apr-2013) from the Sage source code: http://www.sagenb.org/src/combinat/partition.py

**Usage**

```plaintext
conjugate(partition, use_c = TRUE)
```

**Arguments**

- **partition**
  a vector that represents an integer partition
- **use_c**
  logical, defaults to TRUE, the conjugate is computed in c
**divide_and_conquer**

**Examples**

```
conjugate(c(3,3,1,1), FALSE)
```

---

**Description**

Divide and conquer method of generating uniform random partitions of Q having N parts.

**Usage**

```
divide_and_conquer(part, Q, N, D, rand_int, use_c, use_hash)
```

**Arguments**

- **part**: a list to hold the partition
- **Q**: the total sum of the partition
- **N**: Number of parts to sum over
- **D**: a dictionary for the number of partitions of Q having N or less parts (or N or less as the largest part), i.e. $P(Q, Q + N)$.
- **rand_int**: a number representing a member of the feasible set
- **use_c**: boolean if TRUE then compiled c code is used
- **use_hash**: boolean, if TRUE then a hash table is used

**Examples**

```
divide_and_conquer(c(5, 4), 5, 4, hash(), 2, TRUE, FALSE)
```

---

**get_multiplicity**

Find the number of times a value $k$ occurs in a partition that is being generated at random by the `multiplicity()` function. The resulting multiplicity is then passed back to the `multiplicity()` function along with an updated value of count and an updated dictionary $D$.

**Description**

Find the number of times a value $k$ occurs in a partition that is being generated at random by the `multiplicity()` function. The resulting multiplicity is then passed back to the `multiplicity()` function along with an updated value of count and an updated dictionary $D$. 
Usage

get_multiplicity(Q, k, D, rand_int, count, use_c, use_hash)

Arguments

Q the total sum of the partition
k the size of the largest (and also first) part
D a dictionary for the number of partitions of Q having N or less parts (or N or less as the largest part), i.e. P(Q, Q + N).
rand_int the random integer
count count < rand_int
use_c boolean if TRUE then compiled c code is used
use_hash boolean, if TRUE then a hash table is used

Examples

get_multiplicity(10, 5, hash(), 3, 2, TRUE, FALSE)

get_rand_int

Generate a random integer between two integers

Usage

get_rand_int(min = 0, max = 1)

Arguments

min minimum value
max maximum value

Examples

get_rand_int(min=0, max=10)
last

Returns the last element of a vector

Description

Returns the last element of a vector

Usage

last(x)

Arguments

x a vector

Examples

last(1:10)
last(letters[1:10])

multiplicity

multiplicity method of generating uniform random partitions of Q having N parts.

Description

multiplicity method of generating uniform random partitions of Q having N parts.

Usage

multiplicity(part, Q, D, rand_int, use_c, use_hash)

Arguments

part a vector to hold the partition
Q the total sum of the partition
D a dictionary for the number of partitions of Q having N or less parts (or N or less as the largest part), i.e. $P(Q, Q + N)$.
rand_int random integer
use_c boolean if TRUE then compiled c code is used
use_hash boolean, if TRUE then a hash table is used

Examples

multiplicity(c(5, 4), 4, hash(), 1, TRUE, FALSE)
**NrParts**

Find the number of partitions for a given total \( Q \) and number of parts \( N \).

**Description**

This function was recoded and modified from GAP source code: www.gap-system.org. Modifications for speed were based on the proposition that the number of partitions of \( Q \) having \( N \) parts is equal to the number of partitions of \( Q \) having \( N \) parts is equal to the number of partitions of \( Q - N \), if \( N > Q/2 \) (for odd \( Q \)) or if \( N \geq Q/2 \) (for even \( Q \)).

**Usage**

\[
\text{NrParts}(Q, N = \text{NULL}, \text{use}_c = \text{TRUE})
\]

**Arguments**

- **Q**
  - Total sum

- **N**
  - Number of items to sum across, if not specified than all possible values are considered

- **use_c**
  - logical, defaults to TRUE, the number of partitions is computed in c

**Examples**

\[
\text{NrParts}(100)
\]

\[
\text{NrParts}(100, 10)
\]

**P**

Number of partitions of \( Q \) with \( k \) or less parts.

**Description**

This function was derived using the following theorem and proposition. The number of partitions of \( Q \) with \( k \) or less parts equals the number of partitions of \( Q \) with \( k \) or less as the largest part (see Bona 2006). This is a mathematical symmetry, i.e. congruency. Additionally, the number of partitions of \( Q \) with \( k \) or less parts equals the number of partitions of \( Q+k \) with \( k \) as the largest part when \( k>0 \), i.e. \( P(Q + k, k) \). We do not have a source for this proposition, but it can be shown when enumerating the entire feasible set or using the Sage computing environment.

**Usage**

\[
P(D, Q, k, \text{use}_c, \text{use}_\text{hash})
\]
**Arguments**

- `D`: lookup table for numbers of partitions of `Q` having `k` or less parts (or `k` or less as the largest part), i.e. \( P(Q, Q + k) \)
- `Q`: total (i.e., sum across all `k` or `n` parts)
- `k`: the number of parts and also the size of the largest part (congruency)
- `use_c`: boolean, if TRUE the number of partitions is computed in `c`
- `use_hash`: boolean, if TRUE then a hash table is used instead of R’s native list to store the information

**Value**

- a two element list, the first element is `D` the lookup table and the second element is the number of partitions for the specified `Q` and `k` value.

**References**


**Examples**

```r
p(list(), 100, 10, FALSE, FALSE)
```

**Description**

Generate uniform random partitions of `Q` having `N` parts.

**Usage**

```r
rand_partitions(Q, N, sample_size, method = "best",
               D = hash(), zeros = FALSE, use_c = TRUE,
               use_hash = FALSE)
```

**Arguments**

- `Q`: Total sum across parts
- `N`: Number of parts to sum over
- `sample_size`: number of random partitions to generate
- `method`: method to use for generating the partition, options include: 'bottom_up', 'top_down', 'divide_and_conquer', 'multiplicity', and 'best'. Defaults to 'best'
- `D`: a dictionary for the number of partitions of `Q` having `N` or less parts (or `N` or less as the largest part), i.e. \( P(Q, Q + N) \). Defaults to a blank dictionary.
top_down

zeros boolean if True partitions can have zero values, if False partitions have only positive values, defaults to False
use_c boolean if TRUE then compiled c code is used, defaults to TRUE
use_hash boolean, if TRUE then a hash table is used, defaults to FALSE

Value
A matrix where each column is a random partition

Note
method 'best' attempts to use the values of Q and N to infer what the fastest method to compute the partition.
if zeros are allowed, then we must ask whether Q >= N, if not, then the total Q is partitioned among a greater number of parts than there are, say, individuals. In which case, some parts must be zero.
A random partition would then be any random partition of Q with zeros appended at the end. But, if Q >= N, then Q is partitioned among less number of parts than there are individuals. In which case, a random partition would be any random partition of Q having N or less parts.

Examples
rand_partitions(100, 10, 5)

rpartitions
top_down

Description
rpartitions

Top down method of generating uniform random partitions of Q having N parts.

Usage
top_down(part, Q, D, rand_int, use_c, use_hash)
Arguments

- **part**: a list to hold the partition
- **Q**: the total sum of the partition
- **D**: a dictionary for the number of partitions of Q having N or less parts (or N or less as the largest part), i.e. \( P(Q + N, N) \).
- **rand_int**: a number representing a member of the feasible set
- **use_c**: boolean if TRUE then compiled c code is used
- **use_hash**: boolean, if TRUE then a hash table is used

Examples

```r
top_down(c(5, 4), 4, hash(), 1, TRUE, FALSE)
```
Index

bottom_up, 2
conjugate, 2
divide_and_conquer, 3
get_multiplicity, 3
get_rand_int, 4
last, 5
multiplicity, 5
NrParts, 6
P, 6
rand_partitions, 7
rpartitions, 8
rpartitions-package (rpartitions), 8
top_down, 8