Package ‘bit’

August 4, 2020

Type Package

Title Classes and Methods for Fast Memory-Efficient Boolean Selections

Version 4.0.4

Date 2020-08-03

Author Jens Oehlschlägel [aut, cre], Brian Ripley [ctb]

Maintainer Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

Depends R (>= 2.9.2)

Suggests testthat (>= 0.11.0), roxygen2, knitr, rmarkdown, microbenchmark, bit64 (>= 4.0.0), ff (>= 4.0.0)

Description Provided are classes for boolean and skewed boolean vectors, fast boolean methods, fast unique and non-unique integer sorting, fast set operations on sorted and unsorted sets of integers, and foundations for ff (range index, compression, chunked processing).

License GPL-2 | GPL-3

LazyLoad yes

ByteCompile yes

Encoding UTF-8

URL https://github.com/truecluster/bit

VignetteBuilder knitr, rmarkdown

RoxygenNote 7.1.0.9000

NeedsCompilation yes

Repository CRAN

Date/Publication 2020-08-04 04:20:08 UTC

R topics documented:

  bit-package .......................................................... 3
  .BITS ...................................................................... 3
  as.bit.NULL ............................................................. 4
<table>
<thead>
<tr>
<th>R topic documented:</th>
</tr>
</thead>
<tbody>
<tr>
<td>as.bitwhich.NULL</td>
</tr>
<tr>
<td>as.booltype.default</td>
</tr>
<tr>
<td>as.character.bit</td>
</tr>
<tr>
<td>as.character.bitwhich</td>
</tr>
<tr>
<td>as.r1.r1</td>
</tr>
<tr>
<td>as.which.which</td>
</tr>
<tr>
<td>bbatch</td>
</tr>
<tr>
<td>bit</td>
</tr>
<tr>
<td>bitsort</td>
</tr>
<tr>
<td>bitwhich</td>
</tr>
<tr>
<td>bitwhich_representation</td>
</tr>
<tr>
<td>bit_in</td>
</tr>
<tr>
<td>bit_rangediff</td>
</tr>
<tr>
<td>bit_setops</td>
</tr>
<tr>
<td>bit_sort</td>
</tr>
<tr>
<td>bit_sort_unique</td>
</tr>
<tr>
<td>bit_unidup</td>
</tr>
<tr>
<td>booltype</td>
</tr>
<tr>
<td>booltypes</td>
</tr>
<tr>
<td>c.booltype</td>
</tr>
<tr>
<td>chunk</td>
</tr>
<tr>
<td>chunks</td>
</tr>
<tr>
<td>clone</td>
</tr>
<tr>
<td>CoercionToStandard</td>
</tr>
<tr>
<td>copy_vector</td>
</tr>
<tr>
<td>countsort</td>
</tr>
<tr>
<td>Extract</td>
</tr>
<tr>
<td>firstNA</td>
</tr>
<tr>
<td>getsetattr</td>
</tr>
<tr>
<td>get_length</td>
</tr>
<tr>
<td>in.bitwhich</td>
</tr>
<tr>
<td>intrle</td>
</tr>
<tr>
<td>is.booltype</td>
</tr>
<tr>
<td>is.na.bit</td>
</tr>
<tr>
<td>length.bit</td>
</tr>
<tr>
<td>maxindex.default</td>
</tr>
<tr>
<td>merge_rev</td>
</tr>
<tr>
<td>Metadata</td>
</tr>
<tr>
<td>physical.default</td>
</tr>
<tr>
<td>print.bit</td>
</tr>
<tr>
<td>print.bitwhich</td>
</tr>
<tr>
<td>quicksort2</td>
</tr>
<tr>
<td>quicksort3</td>
</tr>
<tr>
<td>range_na</td>
</tr>
<tr>
<td>range_nanozero</td>
</tr>
<tr>
<td>range_sortna</td>
</tr>
<tr>
<td>rep.booltype</td>
</tr>
<tr>
<td>repeat.time</td>
</tr>
</tbody>
</table>
Description

Provided are classes for boolean and skewed boolean vectors, fast boolean methods, fast unique and non-unique integer sorting, fast set operations on sorted and unsorted sets of integers, and foundations for ff (range indices, compression, chunked processing).

Details

For details view the vignettes ../doc/bit-usage.pdf and ../doc/bit-performance.pdf

.BITS

Initializing bit masks

Description

Functions to allocate (and de-allocate) bit masks

Usage

.BITS

bit_init()

bit_done()
Format

An object of class integer of length 1.

Details

The C-code operates with bit masks. The memory for these is allocated dynamically. bit_init is called by .First.lib and bit_done is called by .Last.lib. You don’t need to care about these under normal circumstances.

Author(s)

Jens Oehlschlägel

See Also

bit

Examples

bit_done()
bit_init()

| as.bit.NULL | Coercing to bit |

Description

Coercing to bit vector

Usage

```r
## S3 method for class 'NULL'
as.bit(x, ...)

## S3 method for class 'bit'
as.bit(x, ...)

## S3 method for class 'logical'
as.bit(x, ...)

## S3 method for class 'integer'
as.bit(x, ...)

## S3 method for class 'double'
as.bit(x, ...)
```
## S3 method for class 'bitwhich'
\texttt{as.bit(x, \ldots)}

## S3 method for class 'which'
\texttt{as.bit(x, length = attr(x, "maxindex"), \ldots)}

## S3 method for class 'ri'
\texttt{as.bit(x, \ldots)}

\texttt{as.bit(x = \text{NULL}, \ldots)}

### Arguments

\textit{x} 
\hspace{1cm} an object of class \texttt{bit, logical, integer, bitwhich} or an integer from \texttt{as.another} or a boolean \texttt{ff}

\textit{\ldots} 
\hspace{1cm} further arguments

\textit{length} 
\hspace{1cm} the length of the new bit vector

### Details

Coercing to bit is quite fast because we use a double loop that fixes each word in a processor register

### Value

\texttt{is.bit} returns FALSE or TRUE, \texttt{as.bit} returns a vector of class 'bit'

### Methods (by class)

- \texttt{NULL}: method to coerce to \texttt{bit} (zero length) from \texttt{NULL}
- \texttt{bit}: method to coerce to \texttt{bit} from \texttt{bit}
- \texttt{logical}: method to coerce to \texttt{bit} from \texttt{logical}
- \texttt{integer}: method to coerce to \texttt{bit} from \texttt{integer} (0L and NA become FALSE, everthing else becomes TRUE)
- \texttt{double}: method to coerce to \texttt{bit} from \texttt{double} (0 and NA become FALSE, everthing else becomes TRUE)
- \texttt{bitwhich}: method to coerce to \texttt{bit} from \texttt{bitwhich}
- \texttt{which}: method to coerce to \texttt{bit} from \texttt{which}
- \texttt{ri}: method to coerce to \texttt{bit} from \texttt{ri}

### Note

Zero is coerced to FALSE, all other numbers including NA are coerced to TRUE. This differs from the NA-to-FALSE coercion in package \texttt{ff} and may change in the future.

### Author(s)

Jens Oehlschlägel
See Also

CoercionToStandard, as.booltype, as.bit, as.bitwhich, as.which, as.ri, as.hi, as.ff

Examples

as.bit(c(0L,1L,2L,-2L,NA))
as.bit(c(0,1,2,-2,NA))
as.bit(c(FALSE, NA, TRUE))

Description

Functions to coerce to bitwhich

Usage

## S3 method for class '``NULL``'
as.bitwhich(x, ...)

## S3 method for class 'bitwhich'
as.bitwhich(x, ...)

## S3 method for class 'which'
as.bitwhich(x, maxindex = attr(x, "maxindex"), ...)

## S3 method for class 'ri'
as.bitwhich(x, ...)

## S3 method for class 'integer'
as.bitwhich(x, poslength = NULL, ...)

## S3 method for class 'double'
as.bitwhich(x, poslength = NULL, ...)

## S3 method for class 'logical'
as.bitwhich(x, poslength = NULL, ...)

## S3 method for class 'bit'
as.bitwhich(x, range = NULL, poslength = NULL, ...)
as.bitwhich(x = NULL, ...)
Arguments

- `x`: An object of class 'bitwhich', 'integer', 'logical' or 'bit' or an integer vector as resulting from 'which'
- `...`: further arguments
- `maxindex`: the length of the new bitwhich vector
- `poslength`: the number of selected elements
- `range`: an `ri` or an integer vector of length==2 giving a range restriction for chunked processing

Value

a value of class `bitwhich`

Methods (by class)

- `NULL`: method to coerce to `bitwhich` (zero length) from `NULL`
- `bitwhich`: method to coerce to `bitwhich` from `bitwhich`
- `which`: method to coerce to `bitwhich` from `which`
- `ri`: method to coerce to `bitwhich` from `ri`
- `integer`: method to coerce to `bitwhich` from `integer` (0 and NA become FALSE, everything else becomes TRUE)
- `double`: method to coerce to `bitwhich` from `double` (0 and NA become FALSE, everything else becomes TRUE)
- `logical`: method to coerce to `bitwhich` from `logical`
- `bit`: method to coerce to `bitwhich` from `bit`

Author(s)

Jens Oehlschlägel

See Also

`CoercionToStandard`, `as.booltype`, `as.bit`, `as.bitwhich`, `as.which`, `as.ri`, `as.hi`, `as.ff`

Examples

```r
as.bitwhich(c(0L,1L,2L,-2L,NA))
as.bitwhich(c(0,1,2,-2,NA))
as.bitwhich(c(NA,NA,NA))
as.bitwhich(c(FALSE, FALSE, FALSE))
as.bitwhich(c(FALSE, FALSE, TRUE))
as.bitwhich(c(FALSE, TRUE, TRUE))
as.bitwhich(c(TRUE, TRUE, TRUE))
```
Description

Coerce to booltype (generic)

Usage

```r
## Default S3 method:
as.booltype(x, booltype = "logical", ...)
as.booltype(x, booltype, ...)
```

Arguments

- `x`: object to coerce
- `booltype`: target `booltype` given as integer or as character
- `...`: further arguments

Value

`x` coerced to `booltype`

Methods (by class)

- default: default method for `as.booltype`

See Also

- CoercionToStandard, `booltypes`, `booltype`, `is.booltype`

Examples

```r
as.booltype(0:1)
as.booltype(0:1, "logical")
as.booltype(0:1, "bit")
as.booltype(0:1, "bitwhich")
as.booltype(0:1, "which", maxindex=2)
as.booltype(0:1, "ri")
```
as.character.bit  

Coerce bit to character

Description
Coerce bit to character

Usage
## S3 method for class 'bit'
as.character(x, ...)

Arguments
x  
a bit vector

Value
a character vector of zeroes and ones

Examples
as.character(bit(12))

as.character.bitwhich  

Coerce bitwhich to character

Description
Coerce bitwhich to character

Usage
## S3 method for class 'bitwhich'
as.character(x, ...)

Arguments
x  
a bitwhich vector

Value
a character vector of zeroes and ones
Examples

as.character(bitwhich(12))

Description

Coerce to ri

Usage

## S3 method for class 'ri'
as.ri(x, ...)

## Default S3 method:
as.ri(x, ...)

Arguments

x          object to coerce
...

Value

an ri object

Methods (by class)

- ri: method to coerce ri to ri
- default: default method to coerce to ri

Author(s)

Jens Oehlschlägel

See Also

CoercionToStandard, as.booltype, as.bit, as.bitwhich, as.which, as.ri, as.hi, as.ff

Examples

as.ri(c(FALSE, TRUE, FALSE, TRUE))
Coercion to (positive) integer positions

Description

Coercing to something like the result of which `which`.

Usage

```r
## S3 method for class 'which'
as.which(x, maxindex = NA_integer_, ...)

## S3 method for class 'NULL'
as.which(x, ...)

## S3 method for class 'numeric'
as.which(x, maxindex = NA_integer_, ...)

## S3 method for class 'integer'
as.which(x, maxindex = NA_integer_, is.unsorted = TRUE, has.dup = TRUE, ...)

## S3 method for class 'logical'
as.which(x, ...)

## S3 method for class 'ri'
as.which(x, ...)

## S3 method for class 'bit'
as.which(x, range = NULL, ...)

## S3 method for class 'bitwhich'
as.which(x, ...)

as.which(x, ...)
```

Arguments

- `x`: an object of classes `bit`, `bitwhich`, `ri` or something on which `which` works
- `maxindex`: the length of the boolean vector which is represented
- `...`: further arguments (passed to `which` for the default method, ignored otherwise)
- `is.unsorted`: a logical scalar indicating whether the data may be unsorted
- `has.dup`: a logical scalar indicating whether the data may have duplicates
- `range`: a `ri` or an integer vector of length==2 giving a range restriction for chunked processing
Details

as.which.bit returns a vector of subscripts with class 'which'

Value

a vector of class 'logical' or 'integer'

Methods (by class)

- `which`: method to coerce to `which` from `which`
- `NULL`: method to coerce to zero length `which` from `NULL`
- `numeric`: method to coerce to `which` from `numeric`
- `integer`: method to coerce to `which` from `integer`
- `logical`: method to coerce to `which` from `logical`
- `ri`: method to coerce to `which` from `ri`
- `bit`: method to coerce to `which` from `bit`
- `bitwhich`: method to coerce to `which` from `bitwhich`

Author(s)

Jens Oehlschlägel

See Also

`CoercionToStandard`, `as.booltype`, `as.bit`, `as.bitwhich`, `as.which`, `as.ri`, `as.hi`, `as.ff`

Examples

```r
r <- ri(5, 20, 100)
x <- as.which(r)
x

stopifnot(identical(x, as.which(as.logical(r))))
stopifnot(identical(x, as.which(as.bitwhich(r))))
stopifnot(identical(x, as.which(as.bit(r))))
```
Description

`bbatch` calculates batch sizes in 1..N so that they have rather balanced sizes than very different sizes.

Usage

`bbatch(N, B)`

Arguments

- `N` total size in 0..integer_max
- `B` desired batch size in 1..integer_max

Details

Tries to have \( rb == 0 \) or \( rb \) as close to \( b \) as possible while guaranteeing that \( rb < b \) && \( (b - rb) \leq \min(nb, b) \)

Value

a list with components

- `b` the batch size
- `nb` the number of batches
- `rb` the size of the rest

Author(s)

Jens Oehlschlägel

See Also

`repmat`, `ffvecapply`

Examples

```
bbatch(100, 24)
```
bit

Create empty bit vector

Description

Bit vectors are a boolean type without NA that requires by factor 32 less RAM than logical. For details on usage see the usage-vignette and for details on performance see performance-vignette.

Usage

\texttt{bit(length = 0L)}

Arguments

\begin{tabular}{ll}
\textbf{length} & length in bits \\
\end{tabular}

Value

\texttt{bit} returns a vector of integer sufficiently long to store 'length' bits.

See Also

booltype, bitwhich, logical

Examples

\begin{verbatim}
bit(12)
!bit(12)
str(bit(128))
\end{verbatim}

bitsort

Low-level sorting: bit sort

Description

In one pass over the vector NAs are handled according to parameter \texttt{na.last} by range_sortna, then, if the vector is unsorted, bit sort is invoked.

Usage

\texttt{bitsort(x, na.last = NA, depth = 1)}

Arguments

\begin{tabular}{ll}
\textbf{x} & an integer vector \\
\textbf{na.last} & NA removes NAs, FALSE puts NAs at the beginning, TRUE puts NAs at the end \\
\textbf{depth} & an integer scalar giving the number of bit-passed before switching to quicksort \\
\end{tabular}
Value

a sorted vector

Examples

```r
g <- c(2L, 0L, 1L, NA, 2L)
bitwhich(g)
bitwhich(g, na.last=TRUE)
bitwhich(g, na.last=FALSE)
```

Description

A bitwhich object represents a boolean filter like a \texttt{bit} object (NAs are not allowed) but uses a sparse representation suitable for very skewed (asymmetric) selections. Three extreme cases are represented with logical values, no length via \texttt{logical()}, all TRUE with \texttt{TRUE} and all FALSE with \texttt{FALSE}. All other selections are represented with positive or negative integers, whatever is shorter. This needs less RAM compared to \texttt{logical} (and often less than \texttt{bit} or \texttt{which}). Logical operations are fast if the selection is asymmetric (only few or almost all selected).

Usage

```r
bitwhich(
  maxindex = 0L,
  x = NULL,
  xempty = FALSE,
  poslength = NULL,
  is.unsorted = TRUE,
  has.dup = TRUE
)
```

Arguments

- `maxindex`: length of the vector
- `x`: Information about which positions are FALSE or TRUE: either \texttt{logical()} or \texttt{TRUE} or \texttt{FALSE} or a integer vector of positive or of negative subscripts.
- `xempty`: what to assume about parameter \texttt{x} if \texttt{x=integer(0)}, typically \texttt{TRUE} or \texttt{FALSE}.
- `poslength`: tuning: \texttt{poslength} is calculated automatically, you can give \texttt{poslength} explicitly, in this case it must be correct and \texttt{x} must be sorted and not have duplicates.
- `is.unsorted`: tuning: \texttt{FALSE} implies that \texttt{x} is already sorted and sorting is skipped
- `has.dup`: tuning: \texttt{FALSE} implies that \texttt{x} has no duplicates
Value

an object of class 'bitwhich' carrying two attributes

maxindex  see above
poslength  see above

See Also

bitwhich_representation, as.bitwhich, bit

Examples

bitwhich()
bitwhich(12)
bitwhich(12, x=TRUE)
bitwhich(12, x=3)
bitwhich(12, x=-3)
bitwhich(12, x=integer())
bitwhich(12, x=integer(), xempty=TRUE)

bitwhich_representation

Diagnose representation of bitwhich

Description

Diagnose representation of bitwhich

Usage

bitwhich_representation(x)

Arguments

x      a bitwhich object

Value

a scalar, one of logical(), FALSE, TRUE, -1 or 1

Examples

bitwhich_representation(bitwhich())
bitwhich_representation(bitwhich(12, FALSE))
bitwhich_representation(bitwhich(12, TRUE))
bitwhich_representation(bitwhich(12, -3))
bitwhich_representation(bitwhich(12, 3))
Description

fast %in% for integers

Usage

bit_in(x, table, retFUN = as.bit)

Arguments

x an integer vector of values to be looked-up
table an integer vector used as lookup-table
retFUN a function that coerces bit and logical vectors

Details

determines the range of the integers and checks if the density justifies use of a bit vector; if yes, maps x or table – whatever is smaller – into a bit vector and searches the other of table or x in the it vector; if no, falls back to %in%

Value

a boolean vector coerced to retFUN

See Also

%in%

Examples

bit_in(1:2, 2:3)
bit_in(1:2, 2:3, retFUN=as.logical)
Description

Fast version of `setdiff(rx[1]:rx[2], y)`.

Usage

```
bit_rangediff(rx, y, revx = FALSE, revy = FALSE)
```

Arguments

- `rx`: range of integers given as `ri` or as a two-element `integer`
- `y`: an integer vector of elements to exclude
- `revx`: FALSE as is, TRUE to reverse the direction and sign of `rx`
- `revy`: FALSE as is, TRUE to reverse the direction and sign of `y`

Details

determines the range of the integers `y` and checks if the density justifies use of a bit vector; if yes, uses a bit vector for the set operation; if no, falls back to a quicksort and `merge_rangediff`

Value

an integer vector

See Also

`bit_setdiff`, `merge_rangediff`

Examples

```
bit_rangediff(c(1L,6L), c(3L,4L))
bit_rangediff(c(6L,1L), c(3L,4L))
bit_rangediff(c(6L,1L), c(3L,4L), revx=TRUE)
bit_rangediff(c(6L,1L), c(3L,4L), revx=TRUE, revy=TRUE)
```
Description

Fast versions of \texttt{union}, \texttt{intersect}, \texttt{setdiff}, symmetric difference and \texttt{setequal} for integers.

Usage

\begin{verbatim}
bit_union(x, y)
bit_intersect(x, y)
bit_setdiff(x, y)
bit_symdiff(x, y)
bit_setequal(x, y)
\end{verbatim}

Arguments

\begin{verbatim}
x an integer vector
y an integer vector
\end{verbatim}

Details

determines the range of the integers and checks if the density justifies use of a bit vector; if yes, uses a bit vector for finding duplicates; if no, falls back to \texttt{union}, \texttt{intersect}, \texttt{setdiff}, \texttt{union(setdiff(x,y),setdiff(y,x))} and \texttt{setequal}

Value

an integer vector

Functions

- \texttt{bit_union}: union
- \texttt{bit_intersect}: intersection
- \texttt{bit_setdiff}: asymmetric difference
- \texttt{bit_symdiff}: symmetric difference
- \texttt{bit_setequal}: equality

See Also

\texttt{bit_in, bit_rangediff}
Examples

- `bit_union(1:2, 2:3)`
- `bit_intersect(1:2, 2:3)`
- `bit_setdiff(1:2, 2:3)`
- `bit_symdiff(1:2, 2:3)`
- `bit_setequal(1:2, 2:3)`
- `bit_setequal(1:2, 2:1)`

Description

Fast sorting of integers

Usage

`bit_sort(x, decreasing = FALSE, na.last = NA, has.dup = TRUE)`

Arguments

- `x` an integer vector
- `decreasing` (currently only FALSE is supported)
- `na.last` NA removes NAs, FALSE puts NAs at the beginning, TRUE puts NAs at the end
- `has.dup` TRUE (the default) assumes that `x` might have duplicates, set to FALSE if duplicates are impossible

Details

determines the range of the integers and checks if the density justifies use of a bit vector; if yes, sorts the first occurrences of each integer in the range using a bit vector, sorts the rest and merges; if no, falls back to quicksort.

Value

a sorted vector

See Also

`sort, ramsort, bit_sort_unique`
**Examples**

```r
bit_sort(c(2L,1L,NA,NA,1L,2L))
bit_sort(c(2L,1L,NA,NA,1L,2L), na.last=FALSE)
bit_sort(c(2L,1L,NA,NA,1L,2L), na.last=TRUE)
```

```r
## Not run:
x <- sample(1e7, replace=TRUE)
system.time(bit_sort(x))
system.time(sort(x))
## End(Not run)
```

---

**bit_sort_unique**

**bit sort unique**

**Description**

fast combination of `sort` and `unique` for integers

**Usage**

```r
bit_sort_unique(
  x,
  decreasing = FALSE,
  na.last = NA,
  has.dup = TRUE,
  range_na = NULL
)
```

**Arguments**

- **x**: an integer vector
- **decreasing**: FALSE (ascending) or TRUE (descending)
- **na.last**: NA removes NAs, FALSE puts NAs at the beginning, TRUE puts NAs at the end
- **has.dup**: TRUE (the default) assumes that `x` might have duplicates, set to FALSE if duplicates are impossible
- **range_na**: NULL calls `range_na`, optionally the result of `range_na` can be given here to avoid calling it again

**Details**

determines the range of the integers and checks if the density justifies use of a bit vector; if yes, creates the result using a bit vector; if no, falls back to `sort(unique())`

**Value**

a sorted unique integer vector
See Also

\texttt{sort, unique, bit_sort, bit_unique}

Examples

\begin{verbatim}
bit_sort_unique(c(2L,1L,NA,NA,1L,2L))
bit_sort_unique(c(2L,1L,NA,NA,1L,2L), na.last=FALSE)
bit_sort_unique(c(2L,1L,NA,NA,1L,2L), na.last=TRUE)
bit_sort_unique(c(2L,1L,NA,NA,1L,2L), decreasing = TRUE)
bit_sort_unique(c(2L,1L,NA,NA,1L,2L), decreasing = TRUE, na.last=FALSE)
bit_sort_unique(c(2L,1L,NA,NA,1L,2L), decreasing = TRUE, na.last=TRUE)
\end{verbatim}

## Not run:
x <- sample(1e7, replace=TRUE)
system.time(bit_sort_unique(x))
system.time(sort(unique(x)))
x <- sample(1e7)
system.time(bit_sort_unique(x))
system.time(sort(x))
## End(Not run)

---

\textbf{bit_unidup} \hspace{1cm} \textit{bit unique and duplicated}

Description

Fast versions of \texttt{unique, duplicated, anyDuplicated} and \text{sum(duplicated(x))} for integers.

Usage

\begin{verbatim}
bit_unique(x, na.rm = NA, range_na = NULL)
bit_duplicated(x, na.rm = NA, range_na = NULL, retFUN = as.bit)
bit_anyDuplicated(x, na.rm = NA, range_na = NULL)
bit_sumDuplicated(x, na.rm = NA, range_na = NULL)
\end{verbatim}

Arguments

\begin{verbatim}
x \hspace{1cm} an integer vector
na.rm \hspace{1cm} NA treats NAs like other integers, TRUE treats all NAs as duplicates, FALSE treats no NAs as duplicates
range_na \hspace{1cm} NULL calls \texttt{range_na}, optionally the result of \texttt{range_na} can be given here to avoid calling it again
retFUN \hspace{1cm} a function that coerces \texttt{bit} and \texttt{logical} vectors
\end{verbatim}
Details
determines the range of the integers and checks if the density justifies use of a bit vector; if yes, uses a bit vector for finding duplicates; if no, falls back to unique, duplicated, anyDuplicated and sum(duplicated(x))

Value
bit_unique returns a vector of unique integers,
bit_duplicated returns a boolean vector coerced to retFUN,
bit_anyDuplicated returns the position of the first duplicate (or zero if no duplicates)
bit_sumDuplicated returns the number of duplicated values (as.integer)

Functions
- bit_unique: extracts unique elements
- bit_duplicated: determines duplicate elements
- bit_anyDuplicated: checks for existence of duplicate elements
- bit_sumDuplicated: counts duplicate elements

See Also
bit_sort_unique

Examples
bit_unique(c(2L,1L,NA,NA,1L,2L))
bit_unique(c(2L,1L,NA,NA,1L,2L), na.rm=FALSE)
bit_unique(c(2L,1L,NA,NA,1L,2L), na.rm=TRUE)

bit_duplicated(c(2L,1L,NA,NA,1L,2L))
bit_duplicated(c(2L,1L,NA,NA,1L,2L), na.rm=FALSE)
bit_duplicated(c(2L,1L,NA,NA,1L,2L), na.rm=TRUE)

bit_anyDuplicated(c(2L,1L,NA,NA,1L,2L))
bit_anyDuplicated(c(2L,1L,NA,NA,1L,2L), na.rm=FALSE)
bit_anyDuplicated(c(2L,1L,NA,NA,1L,2L), na.rm=TRUE)

bit_sumDuplicated(c(2L,1L,NA,NA,1L,2L))
bit_sumDuplicated(c(2L,1L,NA,NA,1L,2L), na.rm=FALSE)
bit_sumDuplicated(c(2L,1L,NA,NA,1L,2L), na.rm=TRUE)
Usage

booltype(x)

Arguments

x an R object

Details

Function booltype returns the boolean type of its argument. There are currently six boolean types, booltypes is an ordered vector with the following ordinal levels

- **nobool** non-boolean types
- **logical** for representing any boolean data including NA
- **bit** for representing dense boolean data
- **bitwhich** for representing sparse (skewed) boolean data
- **which** for representing sparse boolean data with few TRUE
- **ri** range-indexing, for representing sparse boolean data with a single range of TRUE

Value

one scalar element of booltypes in case of ’nobool’ it carries a name attribute with the data type.

Note

do not rely on the internal integer codes of these levels, we might add-in hi later

See Also

booltypes, is.booltype, as.booltype

Examples

unnname(booltypes)
str(booltypes)
sapply(list(double(),integer(),logical(),bit(),bitwhich(),as.which(),ri(1,2,3)), booltype)
**booltypes**

<table>
<thead>
<tr>
<th>booltypes</th>
<th>Boolean types</th>
</tr>
</thead>
</table>

**Description**

The *ordered* factor `booltypes` ranks the boolean types.

**Usage**

`booltypes`

**Format**

An object of class `ordered` (inherits from `factor`) of length 6.

**Details**

There are currently six boolean types, `booltypes` is an `ordered` vector with the following ordinal levels:

- **nobool**: non-boolean types
- **logical**: for representing any boolean data including NA
- **bit**: for representing dense boolean data
- **bitwhich**: for representing sparse (skewed) boolean data
- **which**: for representing sparse boolean data with few TRUE
- **ri**: range-indexing, for representing sparse boolean data with a single range of TRUE

`booltypes` has a *names* attribute such that elements can be selected by name.

**Note**

Do not rely on the internal integer codes of these levels, we might add-in `hi` later.

**See Also**

`booltype`, `is.booltype`, `as.booltype`
c.booltype  

**Concatenating booltype vectors**

**Description**

Creating new boolean vectors by concatenating boolean vectors

**Usage**

```r
## S3 method for class 'booltype'
c(...)  
## S3 method for class 'bit'
c(...)  
## S3 method for class 'bitwhich'
c(...)  
```

**Arguments**

...  

**Value**

a vector with the lowest input booltype (but not lower than logical)

**Author(s)**

Jens Oehlschlägel

**See Also**

`c`, `bit`, `bitwhich`, `which`

**Examples**

```r
c(bit(4), !bit(4))
c(bit(4), !bitwhich(4))
c(bitwhich(4), !bit(4))
c(ri(1,2,4), !bit(4))
c(bit(4), !logical(4))
message("logical in first argument does not dispatch: c(logical(4), bit(4))")
c.booltype(logical(4), !bit(4))
```
chunk

Methods for chunked range index

Description

Calls `chunks` to create a sequence of range indexes along the object which causes the method dispatch.

Usage

```r
chunk(x = NULL, ...)  
```  
```r
## Default S3 method:
chunk(x = NULL, ..., RECORDBYTES = NULL, BATCHBYTES = NULL)
```  

Arguments

- `x` the object along we want chunks
- `...` further arguments passed to `chunks`
- `RECORDBYTES` integer scalar representing the bytes needed to process a single element of the boolean vector (default 4 bytes for logical)
- `BATCHBYTES` integer scalar limiting the number of bytes to be processed in one chunk, default from `getOption("ffbatchbytes")` if not null, otherwise 16777216

Details

`chunk` is generic, the default method is described here, other methods that automatically consider RAM needs are provided with package 'ff', see for example `chunk.ffdf`

Value

returns a named list of `ri` objects representing chunks of subscripts

Methods (by class)

- default: default vector method

available methods

- `chunk.default`, `chunk.ff_vector`, `chunk.ffdf`

Author(s)

Jens Oehlschlägel

See Also

- `chunks`, `ri`, `seq`, `bbatch`
Examples

chunk(complex(1e7))
chunk(raw(1e7))
chunk(raw(1e7), length=3)

chunks(1,10,3)
# no longer do
chunk(1,100,10)
# but for backward compatibility this works
chunk(from=1,to=100,by=10)

---

chunks

Function for chunked range index

Description

creates a sequence of range indexes using a syntax not completely unlike `seq`

Usage

chunks(
  from = NULL,
  to = NULL,
  by = NULL,
  length.out = NULL,
  overlap = 0L,
  method = c("bbatch", "seq"),
  maxindex = NA
)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>from</td>
<td>the starting value of the sequence.</td>
</tr>
<tr>
<td>to</td>
<td>the (maximal) end value of the sequence.</td>
</tr>
<tr>
<td>by</td>
<td>increment of the sequence</td>
</tr>
<tr>
<td>length.out</td>
<td>desired length of the sequence.</td>
</tr>
<tr>
<td>overlap</td>
<td>number of values to overlap (will lower the starting value of the sequence, the first range becomes smaller)</td>
</tr>
<tr>
<td>method</td>
<td>default 'bbatch' will try to balance the chunk size, see bbatch, 'seq' will create chunks like seq</td>
</tr>
<tr>
<td>maxindex</td>
<td>passed to ri</td>
</tr>
</tbody>
</table>
Value
returns a named list of \texttt{ri} objects representing chunks of subscripts

Author(s)
Jens Oehlschlägel

See Also
generic \texttt{chunk, ri, seq, bbatch}

Examples

\begin{verbatim}
chunks(1, 100, by=30)
chunks(1, 100, by=30, method="seq")
## Not run:
require(foreach)
m <- 10000
k <- 1000
n <- m*k
message("Four ways to loop from 1 to n. Slowest foreach to fastest chunk is 1700:1
on a dual core notebook with 3GB RAM\n")
z <- 0L;
print(k*system.time({it <- icount(m); foreach (i = it) %do% { z <- i; NULL }}))
z

z <- 0L
print(system.time({i <- 0L; while (i<n) {i <- i + 1L; z <- i}}))
z

z <- 0L
print(system.time(for (i in 1:n) z <- i))
z

z <- 0L; n <- m*k;
print(system.time(for (ch in chunks(1, n, by=m)){for (i in ch[1]:ch[2])z <- i}))
z
message("Seven ways to calculate sum(1:n).
Slowest foreach to fastest chunk is 61000:1 on a dual core notebook with 3GB RAM\n")
print(k*system.time({it <- icount(m); foreach (i = it, .combine="+") %do% { i }}))
z

z <- 0;
print(k*system.time({it <- icount(m); foreach (i = it) %do% { z <- z + i; NULL }}))
z

z <- 0; print(system.time({i <- 0L;while (i<n) {i <- i + 1L; z <- z + i}})); z

z <- 0; print(system.time(for (i in 1:n) z <- z + i)); z
print(system.time(sum(as.double(1:n))))
\end{verbatim}
z <- 0; n <- m*k
print(system.time(for (ch in chunks(1, n, by=m)){for (i in ch[1]:ch[2])z <- z + i}))

z <- 0; n <- m*k
print(system.time(for (ch in chunks(1, n, by=m)){z <- z+sum(as.double(ch[1]:ch[2])))})

## End(Not run)

close

Cloning ff and ram objects

Description

close physically duplicates objects and can additionally change some features, e.g. length.

Usage

close(x, ...)

## Default S3 method:
close(x, ...)

Arguments

x x

... further arguments to the generic

Details

close is generic. close.default handles ram objects. Further methods are provided in package 'ff'. still.identical returns TRUE if the two atomic arguments still point to the same memory.

Value

an object that is a deep copy of x

Methods (by class)

• default: default method uses R’s C-API ’duplicate()

Author(s)

Jens Oehlschlägel
See Also

clone.ff, copy_vector

Examples

```r
x <- 1:12
y <- x
still.identical(x,y)
y[1] <- y[1]
still.identical(x,y)
y <- clone(x)
still.identical(x,y)
rm(x,y); gc()
```

Coercion from bit, bitwhich, which and ri to logical, integer, double

Description

Coercion from bit is quite fast because we use a double loop that fixes each word in a processor register.

Usage

```r
## S3 method for class 'bit'
as.logical(x, ...)

## S3 method for class 'bit'
as.integer(x, ...)

## S3 method for class 'bit'
as.double(x, ...)

## S3 method for class 'bitwhich'
as.integer(x, ...)

## S3 method for class 'bitwhich'
as.double(x, ...)

## S3 method for class 'bitwhich'
as.logical(x, ...)

## S3 method for class 'ri'
as.logical(x, ...)
```
## S3 method for class 'ri'
as.integer(x, ...)

## S3 method for class 'ri'
as.double(x, ...)

## S3 method for class 'which'
as.logical(x, length = attr(x, "maxindex"), ...)

### Arguments

- **x** an object of class `bit`, `bitwhich` or `ri`
- **...** ignored
- **length** length of the boolean vector (required for `as.logical.which`)

### Value

- `as.logical` returns a vector of `FALSE, TRUE`, `as.integer` and `as.double` return a vector of `0, 1`.

### Author(s)

Jens Oehlschlägel

### See Also

`CoercionToStandard, as.booltype, as.bit, as.bitwhich, as.which, as.ri, as.hi, as.ff`

### Examples

```r
x <- ri(2, 5, 10)
y <- as.logical(x)
y
stopifnot(identical(y, as.logical(as.bit(x))))
stopifnot(identical(y, as.logical(as.bitwhich(x))))

y <- as.integer(x)
y
stopifnot(identical(y, as.integer(as.logical(x))))
stopifnot(identical(y, as.integer(as.bit(x))))
stopifnot(identical(y, as.integer(as.bitwhich(x))))

y <- as.double(x)
y
stopifnot(identical(y, as.double(as.logical(x))))
stopifnot(identical(y, as.double(as.bit(x))))
stopifnot(identical(y, as.double(as.bitwhich(x))))
```
**copy_vector**  

*Copy atomic R vector*

**Description**

Creates a true copy of the underlying C-vector – dropping all attributes – and optionally reverses the direction of the elements.

**Usage**

```r
copy_vector(x, revx = FALSE)
```

**Arguments**

- `x` an R vector
- `revx` default FALSE, set to TRUE to reverse the elements in `x`

**Details**

This can be substantially faster than `duplicate(as.vector(unclass(x)))`

**Value**

copied R vector

**See Also**

`clone`, `still.identical`, `reverse_vector`

**Examples**

```r
x <- factor(letters)
y <- x
z <- copy_vector(x)
still.identical(x, y)
still.identical(x, z)
str(x)
str(y)
str(z)
```
countsort

Low-level sorting: counting sort

Description

In one pass over the vector NAs are handled according to parameter na.last by range_sortna, then, if the vector is unsorted, counting sort is invoked.

Usage

countsort(x, na.last = NA)

Arguments

- **x**: an integer vector
- **na.last**: NA removes NAs, FALSE puts NAs at the beginning, TRUE puts NAs at the end

Value

A sorted vector

Examples

countsort(c(2L, 0L, 1L, NA, 2L))
countsort(c(2L, 0L, 1L, NA, 2L), na.last=TRUE)
countsort(c(2L, 0L, 1L, NA, 2L), na.last=FALSE)

Extract

Extract or replace part of an boolean vector

Description

Operators acting on bit or bitwhich objects to extract or replace parts.

Usage

### S3 method for class 'bit'
x[[i]]

### S3 replacement method for class 'bit'
x[[i]] <- value

### S3 method for class 'bit'
x[i]

### S3 replacement method for class 'bit'
x[i] <- value

## S3 method for class 'bitwhich'
x[[i]]

## S3 replacement method for class 'bitwhich'
x[[i]] <- value

## S3 method for class 'bitwhich'
x[i]

## S3 replacement method for class 'bitwhich'
x[i] <- value

Arguments

  x     a bit or bitwhich object
  i     preferably a positive integer subscript or a ri, see text
  value new logical or integer values

Details

The typical use case for '[]' and '[<-]' is subscripting with positive integers, negative integers are
allowed but slower, as logical subscripts only scalars are allowed. The subscript can be given as a
bitwhich object. Also ri can be used as subscript.

Extracting from bit and bitwhich is faster than from logical if positive subscripts are used.
integer subscripts make sense. Negative subscripts are converted to positive ones, beware the RAM
consumption.

Value

The extractors [[ and ] return a logical scalar or vector. The replacement functions return an object
of class(x).

Author(s)

Jens Oehlschlägel

See Also

bit, Extract

Examples

x <- as.bit(c(FALSE, NA, TRUE))
x[] <- c(FALSE, NA, TRUE)
x[1:2]
x[-3]
x[ri(1,2)]
x[as.bitwhich(c(TRUE,TRUE,FALSE))]
x[[1]]
x[] <- TRUE
x[1:2] <- FALSE
x[[1]] <- TRUE

---

<table>
<thead>
<tr>
<th>firstNA</th>
<th>Position of first NA</th>
</tr>
</thead>
</table>

**Description**

This is substantially faster than `which.max(is.na(x))`

**Usage**

`firstNA(x)`

**Arguments**

- `x`: an R vector

**Value**

a reversed vector

**See Also**

`which.max`, `is.na`, `anyNA`, `anyDuplicated`, `bit_anyDuplicated`

**Examples**

```r
x <- c(FALSE,NA,TRUE)
firstNA(x)
reverse_vector(x)
```

---

```r
## Not run:
x <- 1:1e7
system.time(rev(x))
system.time(reverse_vector(x))
```

```r
## End(Not run)
```
Description

Function `getsetattr` sets a single attribute and function `setattributes` sets a list of attributes.

Usage

```r
getsetattr(x, which, value)
setattr(x, which, value)
setattributes(x, attributes)
```

Arguments

- **x**: an R object
- **which**: name of the attribute
- **value**: value of the attribute, use `NULL` to remove this attribute
- **attributes**: a named list of attribute values

Details

The attributes of `x` are changed in place without copying `x`. Function `setattributes` does only change the named attributes, it does not delete the non-names attributes like `attributes` does.

Value

`invisible()`, we do not return the changed object to remind you of the fact that this function is called for its side-effect of changing its input object.

Functions

- `setattr`
- `setattributes`

Author(s)

Jens Oehlschlögel

References

Writing R extensions – System and foreign language interfaces – Handling R objects in C – Attributes (Version 2.11.1 (2010-06-03 ) R Development)
See Also

\[\texttt{attr unattr}\]

Examples

```r
x <- as.single(runif(10))
attr(x, "Csingle")

f <- function(x)attr(x, "Csingle") <- NULL
g <- function(x)setattr(x, "Csingle", NULL)

f(x)
x
f(x)

## Not run:

# restart R
library(bit)

mysingle <- function(length = 0){
  ret <- double(length)
  setattr(ret, "Csingle", TRUE)
  ret
}

# show that mysinge gives exactly the same result as single
identical(single(10), mysingle(10))

# look at the speedup and memory-savings of mysingle compared to single
system.time(mysingle(1e7))
memory.size(max=TRUE)

system.time(single(1e7))
memory.size(max=TRUE)

# look at the memory limits
# on my win32 machine the first line fails because of not enough RAM, the second works
x <- single(1e8)
x <- mysingle(1e8)

# .g. performance with factors
x <- rep(factor(letters), length.out=1e7)
x[1:10]
# look how fast one can do this
system.time(setattr(x, "levels", rev(letters)))
x[1:10]
# look at the performance loss in time caused by the non-needed copying
system.time(levels(x) <- letters)
x[1:10]
```
# restart R
library(bit)

simplefactor <- function(n){
  factor(rep(1:2, length.out=n))
}

mysimplefactor <- function(n){
  ret <- rep(1:2, length.out=n)
 setattr(ret, "levels", as.character(1:2))
  setattr(ret, "class", "factor")
  ret
}

identical(simplefactor(10), mysimplefactor(10))

system.time(x <- mysimplefactor(1e7))
memory.size(max=TRUE)

system.time(setattr(x, "levels", c("a","b")))
memory.size(max=TRUE)
x[1:4]
memory.size(max=TRUE)
rm(x)
gc()

system.time(x <- simplefactor(1e7))
memory.size(max=TRUE)

system.time(levels(x) <- c("x","y"))
memory.size(max=TRUE)
x[1:4]
memory.size(max=TRUE)
rm(x)
gc()

## End(Not run)

---

### Description

Gets C length of a vector ignoring any length-methods dispatched by classes

### Usage

```r
get_length(x)
```
Arguments

x a vector

Details

Queries the vector length using C-macro LENGTH, this can be substantially faster than length(unclass(x))

Value

integer scalar

Examples

length(bit(12))
get_length(bit(12))

in.bitwhich Check existence of integers in table

Description

If the table is sorted, this can be much faster than %in%

Usage

in.bitwhich(x, table, is.unsorted = NULL)

Arguments

x a vector of integer
table a bitwhich object or a vector of integer
is.unsorted logical telling the function whether the table is (un)sorted. With the default NULL FALSE is assumed for bitwhich tables, otherwise TRUE

Value

logical vector

See Also

%in%

Examples

x <- bitwhich(100)
x[3] <- TRUE
in.bitwhich(c(NA,2,3), x)
**intrle**

*Hybrid Index, C-coded utilities*

**Description**

These C-coded ultilitites speed up index preprocessing considerably.

**Usage**

```r
intrle(x)
intisasc(x, na.method = c("none", "break", "skip")[2])
intisdesc(x, na.method = c("none", "break", "skip")[1])
```

**Arguments**

- `x`: an integer vector
- `na.method`: one of "none", "break", "skip", see details. The strange defaults stem from the initial usage.

**Details**

`intrle` is by factor 50 faster and needs less RAM (2x its input vector) compared to `rle` which needs 9x the RAM of its input vector. This is achieved because we allow the C-code of `intrle` to break when it turns out, that rle-packing will not achieve a compression factor of 3 or better. `intisasc` is a faster version of `is.unsorted`: it checks whether `x` is sorted. `intisdesc` checks for being sorted descending and by default default assumes that the input `x` contains no NAs. `na.method="none"` treats NAs (the smallest integer) like every other integer and hence returns either TRUE or FALSE `na.method="break"` checks for NAs and returns either NA as soon as NA is encountered. `na.method="skip"` checks for NAs and skips over them, hence decides the return value only on the basis of non-NA values.

**Value**

- `intrle` returns an object of class `rle` or NULL, if rle-compression is not efficient (compression factor <3 or length(x)<3).
- `intisasc` returns one of FALSE,NA,TRUE
- `intisdesc` returns one of FALSE,TRUE (if the input contains NAs, the output is undefined)

**Functions**

- `intisasc`: check whether integer vector is ascending
- `intisdesc`: check whether integer vector is descending

**Author(s)**

Jens Oehlschlägel
See Also

hi, rle, is.unsorted, is.sorted

Examples

intrle(sample(1:10))
intrle(diff(1:10))
intisasc(1:10)
intrisasc(10:1)
intrisasc(c(NA, 1:10))
intrisdesc(1:10)
intrisdesc(c(10:1, NA))
intrisdesc(c(10:6, NA, 5:1))
intrisdesc(c(10:6, NA, 5:1), na.method="skip")
intrisdesc(c(10:6, NA, 5:1), na.method="break")

Description

All booltypes including logical except 'nobool' types are considered 'is.booltype'.

Usage

is.booltype(x)

is.bit(x)

is.bitwhich(x)

is.which(x)

is.hi(x)

is.ri(x)

Arguments

x an R object

Value

logical scalar
is.na.bit

**Functions**

- is.bit: tests for **bit**
- is.bitwhich: tests for **bitwhich**
- is.which: tests for **which**
- is.hi: tests for **hi**
- is.ri: tests for **ri**

**See Also**

booltypes, booltype, as.booltype

**Examples**

```r
sapply(list(double(), integer(), logical(), bit(), bitwhich(), as.which(), ri(1, 2, 3)), is.booltype)
```

---

### Description

Test for NA in bit and bitwhich

### Usage

```r
## S3 method for class 'bit'
is.na(x)

## S3 method for class 'bitwhich'
is.na(x)
```

### Arguments

- `x` a **bit** or **bitwhich** vector

### Value

vector of same type with all elements FALSE

### Functions

- is.na.bitwhich: method for **is.na** from **bitwhich**

### See Also

**is.na**
Examples

is.na(bit(6))
is.na(bitwhich(6))

Description

Query the number of bits in a bit vector or change the number of bits in a bit vector.
Query the number of bits in a bitwhich vector or change the number of bits in a bit vector.

Usage

## S3 method for class 'bit'
length(x)

## S3 replacement method for class 'bit'
length(x) <- value

## S3 method for class 'bitwhich'
length(x)

## S3 replacement method for class 'bitwhich'
length(x) <- value

## S3 method for class 'ri'
length(x)

Arguments

x a bit, bitwhich or ri object
value the new number of bits

Details

NOTE that the length does NOT reflect the number of selected (TRUE) bits, it reflects the sum of both, TRUE and FALSE bits. Increasing the length of a bit object will set new bits to FALSE. The behaviour of increasing the length of a bitwhich object is different and depends on the content of the object:

- TRUE – all included, new bits are set to TRUE
- positive integers – some included, new bits are set to FALSE
- negative integers – some excluded, new bits are set to TRUE
- FALSE – all excluded, new bits are set to FALSE

Decreasing the length of bit or bitwhich removes any previous information about the status bits above the new length.
Value

the length A bit vector with the new length

Author(s)

Jens Oehlschlägel

See Also

length, sum, poslength, maxindex

Examples

```
stopifnot(length(ri(1, 1, 32))==32)

x <- as.bit(ri(32, 32, 32))
stopifnot(length(x)==32)
stopifnot(sum(x)==1)
length(x) <- 16
stopifnot(length(x)==16)
stopifnot(sum(x)==0)
length(x) <- 32
stopifnot(length(x)==32)
stopifnot(sum(x)==0)

x <- as.bit(ri(1, 1, 32))
stopifnot(length(x)==32)
stopifnot(sum(x)==1)
length(x) <- 16
stopifnot(length(x)==16)
stopifnot(sum(x)==0)
length(x) <- 32
stopifnot(length(x)==32)
stopifnot(sum(x)==0)

x <- as.bitwhich(bit(32))
stopifnot(length(x)==32)
stopifnot(sum(x)==0)
length(x) <- 16
stopifnot(length(x)==16)
stopifnot(sum(x)==0)
length(x) <- 32
stopifnot(length(x)==32)
stopifnot(sum(x)==0)

x <- as.bitwhich(!bit(32))
stopifnot(length(x)==32)
stopifnot(sum(x)==32)
length(x) <- 16
stopifnot(length(x)==16)
stopifnot(sum(x)==16)
```
length(x) <- 32
stopifnot(length(x)==32)
stopifnot(sum(x)==32)

x <- as.bitwhich(ri(32, 32, 32))
stopifnot(length(x)==32)
length(x) <- 16
stopifnot(length(x)==16)
stopifnot(sum(x)==0)
length(x) <- 32
stopifnot(length(x)==32)
stopifnot(sum(x)==0)

x <- as.bitwhich(ri(2, 32, 32))
stopifnot(length(x)==32)
length(x) <- 16
stopifnot(length(x)==16)
stopifnot(sum(x)==15)
length(x) <- 32
stopifnot(length(x)==32)
stopifnot(sum(x)==31)

x <- as.bitwhich(ri(1, 1, 32))
stopifnot(length(x)==32)
length(x) <- 16
stopifnot(length(x)==16)
stopifnot(sum(x)==1)
length(x) <- 32
stopifnot(length(x)==32)
stopifnot(sum(x)==31)

x <- as.bitwhich(ri(1, 31, 32))
stopifnot(length(x)==32)
stopifnot(sum(x)==31)
message("NOTE the change from 'some excluded' to 'all excluded' here")
length(x) <- 16
stopifnot(length(x)==16)
stopifnot(sum(x)==16)
length(x) <- 32
stopifnot(length(x)==32)
stopifnot(sum(x)==32)

maxindex.default

Get maxindex (length of boolean vector) and poslength (number of 'selected' elements)
**Description**

For `is.booltype` objects the term `length` is ambiguous. For example, the length of `which` corresponds to the sum of `logical`. The generic `maxindex` gives `length(logical)` for all `booltypes`. The generic `poslength` gives the number of positively selected elements, i.e. `sum(logical)` for all `booltypes` (and gives NA if NAs are present).

**Usage**

```r
## Default S3 method:
maxindex(x, ...)

## Default S3 method:
poslength(x, ...)

## S3 method for class 'logical'
maxindex(x, ...)

## S3 method for class 'logical'
poslength(x, ...)

## S3 method for class 'bit'
maxindex(x, ...)

## S3 method for class 'bit'
poslength(x, ...)

## S3 method for class 'bitwhich'
maxindex(x, ...)

## S3 method for class 'bitwhich'
poslength(x, ...)

## S3 method for class 'which'
maxindex(x, ...)

## S3 method for class 'which'
poslength(x, ...)

## S3 method for class 'ri'
maxindex(x, ...)

## S3 method for class 'ri'
poslength(x, ...)
```
Arguments

x  an R object, typically a is.booltype object.
...
  further arguments (ignored)

Value

an integer scalar

Methods (by class)

- default: default method for maxindex
- default: default method for poslength
- logical: maxindex method for class logical
- logical: poslength method for class logical
- bit: maxindex method for class bit
- bit: poslength method for class bit
- bitwhich: maxindex method for class bitwhich
- bitwhich: poslength method for class bitwhich
- which: maxindex method for class which
- which: poslength method for class which
- ri: maxindex method for class ri
- ri: poslength method for class ri

Examples

```r
r <- ri(1,2,12)
i <- as.which(r)
w <- as.bitwhich(r)
b <- as.bit(r)
l <- as.logical(r)
u <- which(l)  # unclassed which

sapply(list(r=r,u=u,i=i,w=w,b=b,l=l), function(x){
  c(length=length(x), sum=sum(x), maxindex=maxindex(x), poslength=poslength(x))
})
```
Description

The `merge_` functions allow unary and binary operations on (ascending) sorted vectors of `link{integer}`.

- `merge_rev(x)` will do in one scan what costs two scans in `-rev(x)`, see also `reverse_vector(x)`.
- Many of these `merge_` can optionally scan their input in reverse order (and switch the sign), which again saves extra scans for calling `merge_rev(x)` first.

Usage

```r
merge_rev(x)
merge_match(x, y, revx = FALSE, revy = FALSE, nomatch = NA_integer_)
merge_in(x, y, revx = FALSE, revy = FALSE)
merge_notin(x, y, revx = FALSE, revy = FALSE)
merge_duplicated(x, revx = FALSE)
merge_anyDuplicated(x, revx = FALSE)
merge_sumDuplicated(x, revx = FALSE)
merge_unique(x, revx = FALSE)
merge_union(
  x,
  y,
  revx = FALSE,
  revy = FALSE,
  method = c("unique", "exact", "all")
)
merge_setdiff(x, y, revx = FALSE, revy = FALSE, method = c("unique", "exact"))
merge_symdiff(x, y, revx = FALSE, revy = FALSE, method = c("unique", "exact"))
merge_intersect(
  x,
  y,
  revx = FALSE,
  revy = FALSE,
  method = c("unique", "exact")
)
```
merge_setequal(x, y, revx = FALSE, revy = FALSE, method = c("unique", "exact"))
merge_rangein(rx, y, revx = FALSE, revy = FALSE)
merge_rangenotin(rx, y, revx = FALSE, revy = FALSE)
merge_rangesect(rx, y, revx = FALSE, revy = FALSE)
merge_rangediff(rx, y, revx = FALSE, revy = FALSE)
merge_first(x, revx = FALSE)
merge_last(x, revx = FALSE)
merge_firstin(rx, y, revx = FALSE, revy = FALSE)
merge_lastin(rx, y, revx = FALSE, revy = FALSE)
merge_firstnotin(rx, y, revx = FALSE, revy = FALSE)
merge_lastnotin(rx, y, revx = FALSE, revy = FALSE)

**Arguments**

- **x**: a sorted set
- **y**: a sorted set
- **revx**: default FALSE, set to TRUE to reverse scan parameter 'x'
- **revy**: default FALSE, set to TRUE to reverse scan parameter 'y'
- **nomatch**: integer value returned for non-matched elements, see `match`
- **method**: one of "unique", "exact" (or "all") which governs how to treat ties, see the function descriptions
- **rx**: range of integers given as `ri` or as a two-element `integer`

**Details**

These are low-level functions and hence do not check whether the set is actually sorted. Note that the 'merge_*' and 'merge_range*' functions have no special treatment for 'NA'. If vectors with 'NA' are sorted ith 'NA' in the first positions ('na.last=FALSE') and arguments 'revx=' or 'revy=' have not been used, then 'NAs' are treated like ordinary integers. 'NA' sorted elsewhere or using 'revx=' or 'revy=' can cause unexpected results (note for example that 'revx=' switches the sign on all integers but 'NAs').

The *binary* 'merge_*' functions have a 'method="exact"' which in both sets treats consecutive occurrences of the same value as if they were different values, more precisely they are handled as if the identity of ties were tuples of ties, `rank(ties)` method="exact" delivers unique output if the input is unique, and in this case works faster than method="unique".
**merge_rev**

**Value**

merge_rev(x) returns \(-rev(x)\) for **integer** and **double** and \(\text{!rev}(x)\) for **logical**

**Functions**

- **merge_match**: returns integer positions of sorted set x in sorted set y, see `match(x,y,...)`
- **merge_in**: returns logical existence of sorted set x in sorted set y, see `x %in% y`
- **merge_notin**: returns logical in-existence of sorted set x in sorted set y, see `!(x %in% y)`
- **merge_duplicated**: returns the duplicated status of a sorted set x, see `duplicated`
- **merge_anyDuplicated**: returns the anyDuplicated status of a sorted set x, see `anyDuplicated`
- **merge_sumDuplicated**: returns the sumDuplicated status of a sorted set x, see `bit_sumDuplicated`
- **merge_unique**: returns unique elements of sorted set x, see `unique`
- **merge_union**: returns union of two sorted sets. Default method='unique' returns a unique sorted set, see `union`; method='exact' returns a sorted set with the maximum number of ties in either input set; method='all' returns a sorted set with the sum of ties in both input sets.
- **merge_setdiff**: returns sorted set x minus sorted set y Default method='unique' returns a unique sorted set, see `setdiff`; method='exact' returns a sorted set with sum(x ties) minus sum(y ties);
- **merge_symdiff**: returns those elements that are in sorted set y xor in sorted set y Default method='unique' returns the sorted unique set complement, see `symdiff`; method='exact' returns a sorted set set complement with abs(sum(x ties) minus sum(y ties));
- **merge_intersect**: returns the intersection of two sorted sets x and y Default method='unique' returns the sorted unique intersect, see `intersect`; method='exact' returns the intersect with the minium number of ties in either set;
- **merge_setequal**: returns TRUE for equal sorted sets and FALSE otherwise Default method='unique' compares the sets after removing ties, see `setequal`; method='exact' compares the sets without removing ties;
- **merge_rangein**: returns logical existence of range rx in sorted set y, see `merge_in`
- **merge_rangenotin**: returns logical in-existence of range rx in sorted set y, see `merge_notin`
- **merge_rangesect**: returns the intersection of range rx and sorted set y, see `merge_intersect`
- **merge_rangediff**: returns range rx minus sorted set y, see `merge_setdiff`
- **merge_first**: quickly returns the first element of a sorted set x (or NA if x is empty), hence x[1] or `merge_rev(x)[1]`
- **merge_last**: quickly returns the last element of a sorted set x, (or NA if x is empty), hence x[n] or `merge_rev(x)[n]`
- **merge_firstin**: quickly returns the first common element of a range rx and a sorted set y, (or NA if the intersection is empty), hence `merge_first(merge_rangesect(rx,y))`
- **merge_lastin**: quickly returns the last common element of a range rx and a sorted set y, (or NA if the intersection is empty), hence `merge_last(merge_rangesect(rx,y))`
- **merge_firstnotin**: quickly returns the first element of a range rx which is not in a sorted set y (or NA if all rx are in y), hence `merge_first(merge_rangediff(rx,y))`
- **merge_lastnotin**: quickly returns the last element of a range rx which is not in a sorted set y (or NA if all rx are in y), hence `merge_last(merge_rangediff(rx,y))`
Note

xx OPTIMIZATION OPPORTUNITY These are low-level functions could be optimized with initial binary search (not findInterval, which coerces to double).

Examples

merge_rev(1:9)
merge_match(1:7, 3:9)
# merge_match(merge_rev(1:7), 3:9)
merge_match(merge_rev(1:7), 3:9, revx=TRUE)
merge_match(merge_rev(1:7), 3:9, revy=TRUE)
merge_match(merge_rev(1:7), merge_rev(3:9))

merge_in(1:7, 3:9)
merge_notin(1:7, 3:9)
merge_anyDuplicated(c(1L,1L,2L,3L))
merge_duplicated(c(1L,1L,2L,3L))
merge_unique(c(1L,1L,2L,3L))

merge_union(c(1L,2L,2L,2L), c(2L,2L,3L))
merge_union(c(1L,2L,2L,2L), c(2L,2L,3L), method="exact")
merge_union(c(1L,2L,2L,2L), c(2L,2L,3L), method="all")

merge_setdiff(c(1L,2L,2L,2L), c(2L,2L,3L))
merge_setdiff(c(1L,2L,2L,2L), c(2L,2L,3L), method="exact")
merge_setdiff(c(1L,2L,2L,2L), c(2L,2L,3L), method="exact")

merge_symdiff(c(1L,2L,2L,2L), c(2L,2L,3L))
merge_symdiff(c(1L,2L,2L,2L), c(2L,2L,3L), method="exact")
merge_symdiff(c(1L,2L,2L,2L), c(2L,2L,3L), method="exact")

merge_intersect(c(1L,2L,2L,2L), c(2L,2L,3L))
merge_intersect(c(1L,2L,2L,2L), c(2L,2L,3L), method="exact")

merge_setequal(c(1L,2L,2L), c(1L,2L))
merge_setequal(c(1L,2L,2L), c(1L,2L,2L))
merge_setequal(c(1L,2L,2L), c(1L,2L), method="exact")
merge_setequal(c(1L,2L,2L), c(1L,2L,2L), method="exact")

Generics related to cache access

Description

These generics are packaged here for methods in packages bit64 and ff.
Usage

is.sorted(x, ...)

is.sorted(x, ...) <- value

na.count(x, ...)

na.count(x, ...) <- value

nvalid(x, ...)

nunique(x, ...)

nunique(x, ...) <- value

nties(x, ...)

nties(x, ...) <- value

Arguments

x           some object
...
value       value assigned on responsibility of the user

Details

see help of the available methods

Value

see help of the available methods

Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

See Also

is.sorted.integer64, na.count.integer64, nvalid.integer64, nunique.integer64, nties.integer64

Examples

methods("na.count")
**Description**

Compatibility functions (to package ff) for getting and setting physical and virtual attributes.

**Usage**

```r
## Default S3 method:
physical(x)

## Default S3 replacement method:
physical(x) <- value

## Default S3 method:
virtual(x)

## Default S3 replacement method:
virtual(x) <- value

## S3 method for class 'physical'
print(x, ...)

## S3 method for class 'virtual'
print(x, ...)

physical(x)
physical(x) <- value
virtual(x)
virtual(x) <- value
```

**Arguments**

- `x` a ff or ram object
- `value` a list with named elements
- `...` further arguments

**Details**

`ff` objects have physical and virtual attributes, which have different copying semantics: physical attributes are shared between copies of `ff` objects while virtual attributes might differ between copies. `as.ram` will retain some physical and virtual attributes in the ram clone, such that `as.ff` can restore an `ff` object with the same attributes.
Value

physical and virtual returns a list with named elements

Author(s)

Jens Oehlschlägel

See Also

physical.ff, physical.ffdf

Examples

physical(bit(12))
virtual(bit(12))

print.bit

Print method for bit

Description

Print method for bit

Usage

```r
## S3 method for class 'bit'
print(x, ...)
```

Arguments

- `x`: a bit vector
- `...`: passed to print

Value

a character vector showing first and last elements of the bit vector

Examples

```r
print(bit(120))
```
### print.bitwhich

*Print method for bitwhich*

**Description**

Print method for bitwhich

**Usage**

```r
## S3 method for class 'bitwhich'
print(x, ...)
```

**Arguments**

- `x`: a `bitwhich` object
- `...`: ignored

---

### quicksort2

*Low-level sorting: binary quicksort*

**Description**

In one pass over the vector NAs are handled according to parameter `na.last` by `range_sortna`, then, if the vector is unsorted, binary quicksort is invoked.

**Usage**

```r
quicksort2(x, na.last = NA)
```

**Arguments**

- `x`: an integer vector
- `na.last`: `NA` removes NAs, `FALSE` puts NAs at the beginning, `TRUE` puts NAs at the end

**Value**

a sorted vector

**Examples**

```r
quicksort2(c(2L, 0L, 1L, NA, 2L))
quicksort2(c(2L, 0L, 1L, NA, 2L), na.last=TRUE)
quicksort2(c(2L, 0L, 1L, NA, 2L), na.last=FALSE)
```
quicksort3  

Low-level sorting: threeway quicksort

Description

In one pass over the vector NAs are handled according to parameter na.last by range_sortna, then, if the vector is unsorted, threeway quicksort is invoked.

Usage

quicksort3(x, na.last = NA)

Arguments

x an integer vector
na.last NA removes NAs, FALSE puts NAs at the beginning, TRUE puts NAs at the end

Value

a sorted vector

Examples

countsort(c(2L,0L,1L,NA,2L))
countsort(c(2L,0L,1L,NA,2L), na.last=TRUE)
countsort(c(2L,0L,1L,NA,2L), na.last=FALSE)

range_na  

Get range and number of NAs

Description

Get range and number of NAs

Usage

range_na(x)

Arguments

x an integer vector
Value

an integer vector with three elements

1  min integer
2  max integer
3  number of NAs

See Also

range_nanozero and range_sortna

Examples

range_nanozero(c(0L,1L,2L,NA))

---

range_nanozero

Remove zeros and get range and number of NAs

Description

Remove zeros and get range and number of NAs

Usage

range_nanozero(x)

Arguments

x  an integer vector

Value

an integer vector without zeros and with an attribute range_na with three elements

1  min integer
2  max integer
3  number of NAs

See Also

range_na and range_sortna

Examples

range_nanozero(c(0L,1L,2L,NA))
range_sortna

Prepare for sorting and get range, number of NAs and unsortedness

Description

In one pass over the vector NAs are treated according to parameter na.last exactly like sort does, the range, number of NAs and unsortedness is determined.

Usage

range_sortna(x, decreasing = FALSE, na.last = NA)

Arguments

x an integer vector
decreasing (currently only FALSE is supported)
na.last NA removes NAs, FALSE puts NAs at the beginning, TRUE puts NAs at the end

Value

an integer vector with NAs are treated and an attribute range_na with four elements

1 min integer
2 max integer
3 number of NAs
3 0 for sorted vector and 1 for is.unsorted

See Also

range_na and range_nanozero

Examples

range_sortna(c(0L,1L,NA,2L))
range_sortna(c(2L,NA,1L,0L))
range_sortna(c(0L,1L,NA,2L), na.last=TRUE)
range_sortna(c(2L,NA,1L,0L), na.last=TRUE)
range_sortna(c(0L,1L,NA,2L), na.last=FALSE)
range_sortna(c(2L,NA,1L,0L), na.last=FALSE)
Description

Creating new bit or bitwhich by recycling such vectors

Usage

## S3 method for class 'bit'
rep(x, times = 1L, length.out = NA, ...)

## S3 method for class 'bitwhich'
rep(x, times = 1L, length.out = NA, ...)

Arguments

- **x**: bit or bitwhich object
- **times**: number of replications
- **length.out**: final length of replicated vector (dominates times)
- **...**: not used

Value

An object of class 'bit' or 'bitwhich'

Author(s)

Jens Oehlschlägel

See Also

rep.bit, bitwhich

Examples

rep(as.bit(c(FALSE,TRUE)), 2)
rep(as.bit(c(FALSE,TRUE)), length.out=7)
rep(as.bitwhich(c(FALSE,TRUE)), 2)
rep(as.bitwhich(c(FALSE,TRUE)), length.out=1)
repeat.time

Adaptive timer

Description

Repeats timing expr until minSec is reached

Usage

repeat.time(expr, gcFirst = TRUE, minSec = 0.5, envir = parent.frame())

Arguments

expr

Valid expression to be timed.

gcFirst

Logical - should a garbage collection be performed immediately before the timing? Default is TRUE.

minSec

number of seconds to repeat at least

eenvi

the environment in which to evaluate expr (by default the calling frame)

Value

A object of class "proc_time": see proc.time for details.

Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

See Also

system.time

Examples

system.time(1+1)
repeat.time(1+1)

system.time(sort(runif(1e6)))
repeat.time(sort(runif(1e6)))
Description

`repfromto` virtually recycles object `x` and cuts out positions from `from` to `to`.

Usage

```r
repfromto(x, from, to)
repfromto(x, from, to) <- value
```

Arguments

- `x`: an object from which to recycle
- `from`: first position to return
- `to`: last position to return
- `value`: value to assign

Details

`repfromto` is a generalization of `rep`, where `rep(x, n) == repfromto(x, 1, n)`. You can see this as an R-side (vector) solution of the `mod_iterate` macro in arithmetic.c.

Value

A vector of length `from - to + 1`.

Author(s)

Jens Oehlschlägel

See Also

`rep`, `ffvecapply`

Examples

```r
message("a simple example")
repfromto(0:9, 11, 20)
```
Reversing bit and bitwhich vectors

Description

Creating new bit or bitwhich by reversing such vectors

Usage

```r
## S3 method for class 'bit'
rev(x)

## S3 method for class 'bitwhich'
rev(x)
```

Arguments

- `x`: bit or bitwhich object

Value

An object of class 'bit' or 'bitwhich'

Author(s)

Jens Oehlschlägel

See Also

`rev`, `bit`, `bitwhich`

Examples

```r
rev(as.bit(c(FALSE, TRUE)))
rev(as.bitwhich(c(FALSE, TRUE)))
```
Reverse atomic vector

| reverse_vector       | Reverse atomic vector |

**Description**

Returns a reversed copy – with attributes retained.

**Usage**

reverse_vector(x)

**Arguments**

x  an R vector

**Details**

This is substantially faster than rev

**Value**

a reversed vector

**See Also**

rev, copy_vector

**Examples**

```r
x <- factor(letters)
rev(x)
reverse_vector(x)
## Not run:
x <- 1:1e7
system.time(rev(x))
system.time(reverse_vector(x))
## End(Not run)
```
### Description

A range index can be used to extract or replace a continuous ascending part of the data.

### Usage

```r
ri(from, to = NULL, maxindex = NA)
```

```r
## S3 method for class 'ri'
print(x, ...)
```

### Arguments

- `from`: first position
- `to`: last position
- `maxindex`: the maximal length of the object-to-be-subscripted (if known)
- `x`: an object of class ‘ri’
- `...`: further arguments

### Value

A two element integer vector with class ‘ri’

### Author(s)

Jens Oehlschlägel

### See Also

`as.hi`

### Examples

```r
bit(12)[ri(1,6)]
```
rlepack

Hybrid Index, rle-pack utilities

Description
Basic utilities for rle packing and unpacking and apropriate methods for rev and unique.

Usage
rlepack(x, ...)

## S3 method for class 'integer'
rlepack(x, pack = TRUE, ...)

rleunpack(x)

## S3 method for class 'rlepack'
rleunpack(x)

## S3 method for class 'rlepack'
rev(x)

## S3 method for class 'rlepack'
unique(x, incomparables = FALSE, ...)

## S3 method for class 'rlepack'
anyDuplicated(x, incomparables = FALSE, ...)

Arguments
x in 'rlepack' an integer vector, in the other functions an object of class 'rlepack'
...
just to keep R CMD CHECK quiet (not used)
pack
incomparables

Value
A list with components

first the first element of the packed sequence
dat either an object of class rle or the complete input vector x if rle-packing is not efficient
last the last element of the packed sequence

Author(s)
Jens Oehlschlägel
See Also

`hi, intrle, rle, rev, unique`

Examples

```r
x <- rlepack(rep(0L, 10))
```

---

**Description**

These are generic stubs for low-level sorting and ordering methods implemented in packages `bit64` and `ff`. The `.sortorder` methods do sorting and ordering at once, which requires more RAM than ordering but is (almost) as fast as as sorting.

**Usage**

```r
ramsort(x, ...)
ramorder(x, i, ...)
ramsortorder(x, i, ...)
mergesort(x, ...)
mergeorder(x, i, ...)
mergesortorder(x, i, ...)
quicksort(x, ...)
quickorder(x, i, ...)
quicksortorder(x, i, ...)
shellsort(x, ...)
shellorder(x, i, ...)
shellsortorder(x, i, ...)
radixsort(x, ...)
```
radixorder(x, i, ...)  
radixsortorder(x, i, ...)  
keysort(x, ...)  
keyorder(x, i, ...)  
keysortorder(x, i, ...)

Arguments

x a vector to be sorted by ramsort and ramsortorder, i.e. the output of sort
... further arguments to the sorting methods
i integer positions to be modified by ramorder and ramsortorder, default is 1:n, in this case the output is similar to order

Details

The sort generics do sort their argument 'x', some methods need temporary RAM of the same size as 'x'. The order generics do order their argument 'i' leaving 'x' as it was, some methods need temporary RAM of the same size as 'i'. The sortorder generics do sort their argument 'x' and order their argument 'i', this way of ordering is much faster at the price of requiring temporary RAM for both, 'x' and 'i', if the method requires temporary RAM. The ram generics are high-level functions containing an optimizer that chooses the 'best' algorithms given some context.

Value

These functions return the number of NAs found or assumed during sorting

Index of implemented methods

generic  ff  bit64
ramsort  ramsort.default  ramsort.integer64
shellsort  shellsort.default  shellsort.integer64
quicksort  quicksort.integer64
mergesort  mergesort.default  mergesort.integer64
radixsort  radixsort.default  radixsort.integer64
keysort  keysort.default

generic  ff  bit64
ramorder  ramorder.default  ramorder.integer64
shellorder  shellorder.default  shellorder.integer64
quickorder  quickorder.integer64
mergeorder  mergeorder.default  mergeorder.integer64
radixorder  radixorder.default  radixorder.integer64
keyorder  keyorder.default
Note

Note that these methods purposely violate the functional programming paradigm: they are called for the side-effect of changing some of their arguments. The rationale behind this is that sorting is very RAM-intensive and in certain situations we might not want to allocate additional memory if not necessary to do so. The sort-methods change x, the order-methods change i, and the sortorder-methods change both x and i. You as the user are responsible to create copies of the input data 'x' and 'i' if you need non-modified versions.

Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

See Also

sort and order in base R, bitsort for faster integer sorting

Description

Test for C-level identity of two atomic vectors

Usage

still.identical(x, y)

Arguments

x an atomic vector
y an atomic vector

Value

logical scalar
Examples

x <- 1:2
y <- x
z <- copy_vector(x)
still.identical(y, x)
still.identical(z, x)

---

str.bit  
Str method for bit

Description

To actually view the internal structure use str(unclass(bit))

Usage

## S3 method for class 'bit'
str(
  object,
  vec.len = strO$vec.len,
  give.head = TRUE,
  give.length = give.head,
  ...
)

Arguments

object  
any R object about which you want to have some information.

vec.len  
numeric (>= 0) indicating how many ‘first few’ elements are displayed of each vector. The number is multiplied by different factors (from .5 to 3) depending on the kind of vector. Defaults to the vec.len component of option “str” (see options) which defaults to 4.

give.head  
logical; if TRUE (default), give (possibly abbreviated) mode/class and length (as <type>[1:...]).

give.length  
logical; if TRUE (default), indicate length (as [1:...]).

...  
potential further arguments (required for Method/Generic reasons).

Value

invisible

Examples

str(bit(120))
Description

To actually view the internal structure use `str(unclass(bitwhich))`

Usage

```r
## S3 method for class 'bitwhich'
str(
  object,
  vec.len = strO$vec.len,
  give.head = TRUE,
  give.length = give.head,
  ...
)
```

Arguments

- `object` any R object about which you want to have some information.
- `vec.len` numeric (>= 0) indicating how many ‘first few’ elements are displayed of each vector. The number is multiplied by different factors (from .5 to 3) depending on the kind of vector. Defaults to the `vec.len` component of option "str" (see `options`) which defaults to 4.
- `give.head` logical; if TRUE (default), give (possibly abbreviated) mode/class and length (as `<type>[1:...]`).
- `give.length` logical; if TRUE (default), indicate length (as `[1:...]`).
- `...` potential further arguments (required for Method/Generic reasons).

Value

`invisible`

Examples

```r
str(bitwhich(120))
```
Summaries

**Summaries of boolean vectors**

**Description**

Fast aggregation functions for `booltype` vectors. namely `bit`, `all`, `any`, `anyNA`, `min`, `max`, `range`, `sum` and `summary`. Now all boolean summaries (except for `anyNA` because the generic does not allow it) have an optional range argument to restrict the range of evaluation. Note that the boolean summaries have meaning and return values differing from logical aggregation functions: they treat `NA` as `FALSE`, `min`, `max` and `range` give the minimum and maximum positions of `TRUE`, `summary` returns counts of `FALSE`, `TRUE` and the range. Note that you can force the boolean interpretation by calling the `booltype` method explicitly on any `booltype` input, e.g. `min.booltype()`, see the examples.

**Usage**

```r
## S3 method for class 'bit'
all(x, range = NULL, ...)

## S3 method for class 'bit'
any(x, range = NULL, ...)

## S3 method for class 'bit'
anyNA(x, recursive = FALSE)

## S3 method for class 'bit'
sum(x, range = NULL, ...)

## S3 method for class 'bit'
min(x, range = NULL, ...)

## S3 method for class 'bit'
max(x, range = NULL, ...)

## S3 method for class 'bit'
range(x, range = NULL, ...)

## S3 method for class 'bit'
summary(object, range = NULL, ...)

## S3 method for class 'bitwhich'
all(x, range = NULL, ...)

## S3 method for class 'bitwhich'
any(x, range = NULL, ...)

## S3 method for class 'bitwhich'
all(x, range = NULL, ...)
```

anyNA(x, ...)

## S3 method for class 'booltype'
sum(x, range = NULL, ...)

## S3 method for class 'booltype'
min(x, range = NULL, ...)

## S3 method for class 'booltype'
max(x, range = NULL, ...)

## S3 method for class 'booltype'
range(x, range = NULL, ...)

## S3 method for class 'booltype'
summary(object, range = NULL, ...)

## S3 method for class 'ri'
all(x, range = NULL, ...)

## S3 method for class 'ri'
any(x, range = NULL, ...)

## S3 method for class 'ri'
anyNA(x, recursive = FALSE)

## S3 method for class 'ri'
sum(x, ...)

## S3 method for class 'ri'
min(x, ...)

## S3 method for class 'ri'
max(x, ...)

## S3 method for class 'ri'
range(x, ...)

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

Arguments

x an object of class bit or bitwhich
range a ri or an integer vector of length==2 giving a range restriction for chunked processing
... formally required but not used
recursive formally required but not used
Symmetric set complement

Description

Symmetric set complement
Usage

symdiff(x, y)

Arguments

x  a vector
y  a vector

Value

union(setdiff(x,y),setdiff(y,x))

Note

that symdiff(x,y) is not identical as symdiff(y,x) without applying sort to the result

See Also

merge_symdiff and xor

Examples

symdiff(c(1L,2L,2L), c(2L,3L))
symdiff(c(2L,3L), c(1L,2L,2L))

Description

Returns object with attributes removed

Usage

unattr(x)

Arguments

x  any R object

Details

attribute removal copies the object as usual

Value

a similar object with attributes removed
vecseq

Author(s)
Jens Oehlschlägel

See Also
attributes, setattributes, unclass

Examples

bit(2)[]
unattr(bit(2)[])

vecseq Vectorized Sequences

Description
vecseq returns concatenated multiple sequences

Usage
vecseq(x, y = NULL, concat = TRUE, eval = TRUE)

Arguments
x vector of sequence start points
y vector of sequence end points (if is.null(y) then x are taken as endpoints, all starting at 1)
concat vector of sequence end points (if is.null(y) then x are taken as endpoints, all starting at 1)
eval vector of sequence end points (if is.null(y) then x are taken as endpoints, all starting at 1)

Details
This is a generalization of sequence in that you can choose sequence starts other than 1 and also have options to no concat and/or return a call instead of the evaluated sequence.

Value
if concat==FALSE and eval==FALSE a list with n calls that generate sequences
if concat==FALSE and eval==TRUE a list with n sequences
if concat==TRUE and eval==FALSE a single call generating the concatenated sequences
if concat==TRUE and eval==TRUE an integer vector of concatenatated sequences
Author(s)
Angelo Canty, Jens Oehlschlägel

See Also:
:: seq, sequence

Examples

```r
sequence(c(3,4))
vecseq(c(3,4))
vecseq(c(1,11), c(5, 15))
vecseq(c(1,11), c(5, 15), concat=FALSE, eval=FALSE)
vecseq(c(1,11), c(5, 15), concat=FALSE, eval=TRUE)
vecseq(c(1,11), c(5, 15), concat=TRUE, eval=FALSE)
vecseq(c(1,11), c(5, 15), concat=TRUE, eval=TRUE)
```

Description

Boolean NEGATION `!`, AND `&`, OR `|` and EXCLUSIVE OR `xor`, see Logic.

Usage

```r
## Default S3 method:
xor(x, y)

## S3 method for class 'logical'
xor(x, y)

## S3 method for class 'bit'
!x

## S3 method for class 'bit'
e1 & e2

## S3 method for class 'bit'
e1 | e2

## S3 method for class 'bit'
e1 == e2
```

```
e1 != e2

## S3 method for class 'bit'
xor(x, y)

## S3 method for class 'bitwhich'
!x

## S3 method for class 'bitwhich'
e1 & e2

## S3 method for class 'bitwhich'
e1 | e2

## S3 method for class 'bitwhich'
e1 == e2

## S3 method for class 'bitwhich'
e1 != e2

## S3 method for class 'bitwhich'
xor(x, y)

## S3 method for class 'booltype'
e1 & e2

## S3 method for class 'booltype'
e1 | e2

## S3 method for class 'booltype'
e1 == e2

## S3 method for class 'booltype'
e1 != e2

## S3 method for class 'booltype'
xor(x, y)

xor(x, y)

Arguments

x a is.booltype vector

y a is.booltype vector

e1 a is.booltype vector

e2 a is.booltype vector
Details

The binary operators and function `xor` can now combine any `is.booltype` vectors. They now re-cycle if vectors have different length. If the two arguments have different `booltypes` the return value corresponds to the lower `booltype` of the two.

Boolean operations on `bit` vectors are extremely fast because they are implemented using C’s bitwise operators. Boolean operations on or `bitwhich` vectors are even faster, if they represent very skewed selections.

The `xor` function has been made generic and `xor.default` has been implemented much faster than R’s standard `xor`. This was possible because actually boolean function `xor` and comparison operator `!=` do the same (even with NAs), and `!=` is much faster than the multiple calls in `(x | y) & !(x & y)`

Value

An object of class `booltype` or `logical`

Methods (by class)

- default: default method for `xor`
- logical: `logical` method for `xor`
- bit: `bit` method for `!`
- bit: `bit` method for `&`
- bit: `bit` method for `|`
- bit: `bit` method for `==`
- bit: `bit` method for `!=`
- bit: `bit` method for `xor`
- bitwhich: `bitwhich` method for `!`
- bitwhich: `bitwhich` method for `&`
- bitwhich: `bitwhich` method for `|`
- bitwhich: `bitwhich` method for `==`
- bitwhich: `bitwhich` method for `!=`
- bitwhich: `bitwhich` method for `xor`
- booltype: `booltype` method for `&`
- booltype: `booltype` method for `|`
- booltype: `booltype` method for `==`
- booltype: `booltype` method for `!=`
- booltype: `booltype` method for `xor`

Author(s)

Jens Oehlschlägel
See Also

booltypes, Logic

Examples

```r
x <- c(FALSE, FALSE, FALSE, NA, NA, NA, TRUE, TRUE, TRUE)
y <- c(FALSE, NA, TRUE, FALSE, NA, TRUE, FALSE, NA, TRUE)

x|y
x|as.bit(y)
x|as.bitwhich(y)
x|as.which(y)
x|ri(1,1,9)
```
Index

!.bit (xor.default), 78
!.bitwhich (xor.default), 78
!=.bit (xor.default), 78
!=.bitwhich (xor.default), 78
!=.booltype (xor.default), 78

* IO
  bbatch, 13
  clone, 30
  intrle, 41
  physical.default, 54
  repfromto, 62
  rlepack, 66

* arith
  Sorting, 67

* attributes
  getsetattr, 37

* attribute
  physical.default, 54
  unattr, 76

* classes
  .BITS, 3
  as.bit.NULL, 4
  as.bitwhich.NULL, 6
  as.ri.ri, 10
  as.which.which, 11
  bit, 14
  c.booltype, 26
  CoercionToStandard, 31
  Extract, 34
  length.bit, 44
  rep.booltype, 60
  rev.booltype, 63
  ri, 65
  Summaries, 72
  xor.default, 78

* datasets
  .BITS, 3
  booltypes, 25

* data
  bbatch, 13
  chunk, 27
  chunks, 28
  clone, 30
  intrle, 41
  physical.default, 54
  repfromto, 62
  rlepack, 66

* environment
  Metadata, 52

* logic
  .BITS, 3
  as.bit.NULL, 4
  as.bitwhich.NULL, 6
  as.ri.ri, 10
  as.which.which, 11
  bit, 14
  c.booltype, 26
  CoercionToStandard, 31
  Extract, 34
  length.bit, 44
  rep.booltype, 60
  rev.booltype, 63
  ri, 65
  Summaries, 72
  xor.default, 78

* manip
  Sorting, 67
  vecseq, 77

* methods
  Metadata, 52

* univar
  Sorting, 67

* utilities
  repeat.time, 61

.BITS, 3
.First.lib, 4
.Last.lib, 4
:. 78
==, 80
==.bit (xor.default), 78
==.bitwhich (xor.default), 78
==.booltype (xor.default), 78
[.bit (Extract), 34
[.bitwhich (Extract), 34
[-.bit (Extract), 34
[--.bit (Extract), 34
[[-.bitwhich (Extract), 34
[[-.bit (Extract), 34
[[-.bitwhich (Extract), 34
[[.bit (Extract), 34
[[.bitwhich (Extract), 34
[[<-.bit (Extract), 34
[[<-.bitwhich (Extract), 34
&.bit (xor.default), 78
&.bitwhich (xor.default), 78
&.booltype (xor.default), 78
%in%, 17, 40, 51
all, 72
all.bit (Summaries), 72
all.bitwhich (Summaries), 72
all.booltype (Summaries), 72
all.ri (Summaries), 72
all.which (Summaries), 72
any, 72
any.bit (Summaries), 72
any.bitwhich (Summaries), 72
any.booltype (Summaries), 72
any.ri (Summaries), 72
any.which (Summaries), 72
anyDuplicated, 22, 23, 36, 51
anyDuplicated.rlepack (rlepack), 66
anyNA, 36, 72
anyNA.bit (Summaries), 72
anyNA.bitwhich (Summaries), 72
anyNA.booltype (Summaries), 72
anyNA.ri (Summaries), 72
anyNA.which (Summaries), 72
as.bit, 6, 7, 10, 12, 32
as.bit(as.bit.NULL), 4
as.bit.NULL, 4
as.bitwhich, 6, 7, 10, 12, 16, 32
as.bitwhich(as.bitwhich.NULL), 6
as.bitwhich.NULL, 6
as.booltype, 6, 7, 10, 12, 24, 25, 32, 43
as.booltype(as.booltype.default), 8
as.booltype.default, 8
as.character.bit, 9
as.character.bitwhich, 9
as.double, 32
as.double.bit (CoercionToStandard), 31
as.double.bitwhich
(CoercionToStandard), 31
as.double.ri (CoercionToStandard), 31
as.ff, 6, 7, 10, 12, 32, 54
as.hi, 6, 7, 10, 12, 32, 65
as.integer, 32
as.integer.bit (CoercionToStandard), 31
as.integer.bitwhich
(CoercionToStandard), 31
as.integer.ri (CoercionToStandard), 31
as.logical, 32
as.logical.bit (CoercionToStandard), 31
as.logical.bitwhich
(CoercionToStandard), 31
as.logical.ri (CoercionToStandard), 31
as.logical.which (CoercionToStandard), 31
as.ram, 54
as.ri, 6, 7, 10, 12, 32
as.ri (as.ri.ri), 10
as.ri.ri, 10
as.which, 5–7, 10, 12, 32
as.which(as.ri.ri.ri), 11
as.which.ri.ri.ri, 11
attr, 38
attributes, 37, 77
bbatch, 13, 27–29
bit, 4, 5, 7, 9, 11, 12, 14, 15–17, 22, 24–26,
32, 34, 35, 43, 44, 48, 60, 63, 72, 75, 80
bit-package, 3
bit_anyDuplicated, 36
bit_anyDuplicated (bit_unidup), 22
bit_done (.BITS), 3
bit_duplicated (bit_unidup), 22
bit_in, 17, 19
bit_init (.BITS), 3
bit_intersect (bit_setops), 19
bit_rangediff, 18, 19
bit_setdiff, 18
bit_setdiff (bit_setops), 19
bit_setequal (bit_setops), 19
bit_setops, 19
bit_sort, 20, 22
bit_sort_unique, 20, 21, 23
bit_sumDuplicated, 31
bit_sumDuplicated (bit_unidup), 22  
bit_symdiff (bit_setops), 19  
bit_unidup, 22  
bit_union (bit_setops), 19  
bit_unique, 22  
bit_unique (bit_unidup), 22  
bisort, 14, 69  
bitwhich, 5, 7, 9, 11, 12, 14, 15, 16, 24–26, 32, 34, 35, 40, 43, 44, 48, 56, 60, 63, 75, 80  
bitwhich_representation, 16, 16  
booltype, 8, 14, 23, 25, 26, 43, 47, 72, 80  
booltypes, 8, 24, 25, 42, 43, 72, 80, 81  
c, 26  
c.bit (c.booltype), 26  
c.bitwhich (c.booltype), 26  
c.booltype, 26  
chunk, 27, 29  
chunk_ff_vector, 27  
chunk_ffdf, 27  
chunks, 27, 28  
clone, 30, 33  
CoercionToStandard, 6–8, 10, 12, 31, 32  
copy_vector, 31, 33, 64  
countsort, 34  
double, 5, 7, 51  
duplicated, 22, 23, 51  

Extract, 34, 35  

ff, 5  
ffvecapply, 13, 62  
firstNA, 36  
get_length, 39  
getsetattr, 37  

hi, 24, 25, 42, 43, 67  

identical, 76  
in.bitwhich, 40  
integer, 5, 7, 12, 18, 50, 51  
intersect, 19, 51  
intisasc (intrle), 41  
intisdesc (intrle), 41  
intrle, 41, 67  
invisible, 70, 71  
is.bit (is.booltype), 42  

is.bitwhich (is.booltype), 42  
is.booltype, 8, 24, 25, 42, 47, 48, 79, 80  
is.hi (is.booltype), 42  
is.na, 36, 43  
is.na.bit, 43  
is.na.bitwhich (is.na.bit), 43  
is.ri (is.booltype), 42  
is.sorted, 42  
is.sorted (Metadata), 52  
is.sorted.integer64, 53  
is.sorted<- (Metadata), 52  
is.unsorted, 41, 42, 59  
is.which (is.booltype), 42  

keyorder (Sorting), 67  
keyorder.default, 68  
keysort (Sorting), 67  
keysort.default, 68  
keysortorder (Sorting), 67  

length, 45, 47, 75  
length.bit, 44  
length.bitwhich (length.bit), 44  
length.ri (length.bit), 44  
length<- (length.bit), 44  
length<=.bitwhich (length.bit), 44  
levels, 24, 25  
Logic, 78, 81  

logical, 5, 7, 12, 14, 15, 17, 22, 24–26, 35, 42, 47, 48, 51, 80  

match, 50, 51  
max, 72  
max.bit (Summaries), 72  
max.bitwhich (Summaries), 72  
max.booltype (Summaries), 72  
max.ri (Summaries), 72  
max.which (Summaries), 72  
maxindex, 45  
maxindex (maxindex.default), 46  
maxindex.default, 46  
merge_anyDuplicated (merge_rev), 49  
merge_duplicated (merge_rev), 49  
merge_first (merge_rev), 49  
merge_firstin (merge_rev), 49  
merge_firstnotin (merge_rev), 49  
merge_in, 51  
merge_in (merge_rev), 49  
merge_intersect, 51
merge_intersect (merge_rev), 49
merge_last (merge_rev), 49
merge_lastin (merge_rev), 49
merge_lastnotin (merge_rev), 49
merge_match (merge_rev), 49
merge_notin, 51
merge_notin (merge_rev), 49
merge_rangediff, 18
merge_rangediff (merge_rev), 49
merge_rangein (merge_rev), 49
merge_rangenotin (merge_rev), 49
merge_rangesect (merge_rev), 49
merge_rev, 49
merge_setdiff, 51
merge_setdiff (merge_rev), 49
merge_setequal (merge_rev), 49
merge_sumDuplicated (merge_rev), 49
merge_symdiff, 76
merge_symdiff (merge_rev), 49
merge_union (merge_rev), 49
merge_unique (merge_rev), 49
mergeorder (Sorting), 67
mergeorder.default, 68
mergeorder.integer64, 68
mergesort (Sorting), 67
mergesort.default, 68
mergesort.integer64, 68
mergesortorder (Sorting), 67
mergesortorder.integer64, 69
Metadata, 52
min, 72
min.bit (Summaries), 72
min.bitwhich (Summaries), 72
min.booltype (Summaries), 72
min.ri (Summaries), 72
min.which (Summaries), 72
na.count (Metadata), 52
na.count.integer64, 53
na.count<= (Metadata), 52
names, 25
nties (Metadata), 52
nties.integer64, 53
nties<= (Metadata), 52
NULL, 5, 7, 12
numeric, 12
nunique (Metadata), 52
nunique.integer64, 53
nunique<= (Metadata), 52
nvalid (Metadata), 52
nvalid.integer64, 53
options, 70, 71
order, 68, 69
ordered, 24, 25
physical (physical.default), 54
physical.default, 54
physical.df, 55
physical.ffdf, 55
physical<= (physical.default), 54
PhysVirt (physical.default), 54
poslength, 45
poslength (maxindex.default), 46
print.bit, 55
print.bitwhich, 56
print.physical (physical.default), 54
print.ri (ri), 65
print.virtual (physical.default), 54
proc.time, 61
quickorder (Sorting), 67
quickorder.integer64, 68
quicksort (Sorting), 67
quicksort.integer64, 68
quicksort2, 56
quicksort3, 57
quicksortorder (Sorting), 67
quicksortorder.integer64, 69
radixorder (Sorting), 67
radixorder.default, 68
radixorder.integer64, 68
radixsort (Sorting), 67
radixsort.default, 68
radixsort.integer64, 68
radixsortorder (Sorting), 67
radixsortorder.integer64, 69
ramorder, 68
ramorder (Sorting), 67
ramorder.default, 68
ramorder.integer64, 68
ramsort, 20, 68
ramsort (Sorting), 67
ramsort.default, 68
ramsort.integer64, 68
ramsortorder, 68
ramsortorder (Sorting), 67
INDEX

ramsortorder.integer64, 69
range, 59, 72
range.bit (Summaries), 72
range.bitwhich (Summaries), 72
range.booltype (Summaries), 72
range.ri (Summaries), 72
range.which (Summaries), 72
range_na, 21, 22, 57, 58, 59
range.nanozero, 21, 22, 57, 58
range_sortna, 14, 34, 56–58, 59
rep, 60, 62
rep.bit (rep.booltype), 60
rep.bitwhich (rep.booltype), 60
rep.booltype, 60
repeat.time, 61
repfromto, 13, 62
repfromto<- (repfromto), 62
rev, 49, 51, 63, 64, 66, 67
rev.bit (rev.booltype), 63
rev.bitwhich (rev.booltype), 63
rev.booltype, 63
rev.rlepack (rlepack), 66
reverse_vector, 33, 49, 64
ri, 5, 7, 10–12, 18, 24, 25, 27–29, 32, 35, 43, 44, 48, 50, 65, 74
rle, 41, 42, 66, 67
rlepack, 66
rleunpack (rlepack), 66

seq, 27–29, 78
sequence, 77, 78
setattr (getsetattr), 37
setattributes, 77
setattributes (getsetattr), 37
setdiff, 19, 51
setequal, 19, 51
shellorder (Sorting), 67
shellorder.default, 68
shellorder.integer64, 68
shellsort (Sorting), 67
shellsort.default, 68
shellsort.integer64, 68
shellsortorder (Sorting), 67
shellsortorder.integer64, 69
sort, 20–22, 59, 68, 69, 76
Sorting, 67
still.identical, 33, 69
str.bit, 70
str.bitwhich, 71

sum, 45, 72
sum.bit (Summaries), 72
sum.bitwhich (Summaries), 72
sum.booltype (Summaries), 72
sum.ri (Summaries), 72
sum.which (Summaries), 72
Summaries, 72
summary, 72
summary.bit (Summaries), 72
summary.bitwhich (Summaries), 72
summary.booltype (Summaries), 72
summary.ri (Summaries), 72
summary.which (Summaries), 72
symdiff, 51, 75
system.time, 61

unattr, 38, 76
unclass, 77
union, 19, 51
unique, 21–23, 51, 66, 67
unique.rlepack (rlepack), 66

vecseq, 77
virtual (physical.default), 54
virtual<- (physical.default), 54

which, 5, 7, 11, 12, 15, 24–26, 43, 47, 48
which.max, 36

xor, 51, 76, 80
xor (xor.default), 78
xor.default, 78