Package ‘bit’

February 10, 2020

Title  A Class for Vectors of 1-Bit Booleans

Version  1.1-15.2

Date  2020-01-14

Depends  R (>= 2.9.2)

Description  True boolean datatype (no NAs),
coercion from and to logicals, integers and integer subscripts;
fast boolean operators and fast summary statistics.
With 'bit' vectors you can store true binary booleans \{FALSE,TRUE\} at the
expense of 1 bit only, on a 32 bit architecture this means factor 32 less
RAM and ~ factor 32 more speed on boolean operations. Due to overhead of
R calls, actual speed gain depends on the size of the vector: expect gains
for vectors of size > 10000 elements. Even for one-time boolean operations
it can pay-off to convert to bit, the pay-off is obvious, when such
components are used more than once.
Reading from and writing to bit is approximately as fast as accessing
standard logicals - mostly due to R's time for memory allocation. The package
allows to work with pre-allocated memory for return values by calling .Call()
directly: when evaluating the speed of C-access with pre-allocated vector
memory, coping from bit to logical requires only 70% of the time for copying
from logical to logical; and copying from logical to bit comes at a
performance penalty of 150%. the package now contains further classes for
representing logical selections: 'bitwhich' for very skewed selections and
'ri' for selecting ranges of values for chunked processing. All three index
classes can be used for subsetting 'ff' objects (ff-2.1-0 and higher).

License  GPL-2

LazyLoad  yes

ByteCompile  yes

Encoding  UTF-8

NeedsCompilation  yes

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

Maintainer  ORPHANED
X-CRAN-Original-Maintainer  Jens Oehlschlägel
<Jens.Oehlschlaegel@truecluster.com>

X-CRAN-Comment  Orphaned on 2020-01-13 as check errors were not corrected.

Repository  CRAN
Date/Publication  2020-02-10 10:06:33 UTC

R topics documented:

- bit-package ......................................................... 3
- as.bit .............................................................. 8
- as.bitwhich ......................................................... 9
- as.logical.bit .................................................... 10
- as.which ............................................................ 11
- bbatch .............................................................. 13
- bitwhich ............................................................ 14
- bit_init ............................................................. 15
- c.bit ................................................................. 16
- chunk .............................................................. 16
- clone ............................................................... 18
- Extract ............................................................. 19
- intrle ............................................................... 21
- is.bit ............................................................... 22
- is.sorted .......................................................... 23
- length.bit ........................................................ 24
- LogicBit ........................................................... 26
- physical ............................................................ 29
- ramsort ............................................................ 30
- regtest.bit ......................................................... 32
- repeat.time ....................................................... 33
- repfromto .......................................................... 34
- ri ................................................................. 35
- rlepack ............................................................ 36
- setattributes ..................................................... 37
- Summary ........................................................... 39
- unattr ............................................................... 42
- vecseq .............................................................. 43

Index  45
Description

Package ‘bit’ provides bitmapped vectors of booleans (no NAs), coercion from and to logicals, integers and integer subscripts; fast boolean operators and fast summary statistics.

With bit vectors you can store true binary booleans \{FALSE,TRUE\} at the expense of 1 bit only, on a 32 bit architecture this means factor 32 less RAM and factor 32 more speed on boolean operations. With this speed gain it even pays-off to convert to bit in order to avoid a single boolean operation on logicals or a single set operation on (longer) integer subscripts, the pay-off is dramatic when such components are used more than once.

Reading from and writing to bit is approximately as fast as accessing standard logicals - mostly due to R’s time for memory allocation. The package allows to work with pre-allocated memory for return values by calling .Call() directly: when evaluating the speed of C-access with pre-allocated vector memory, coping from bit to logical requires only 70\% of the time for copying from logical to logical; and copying from logical to bit comes at a performance penalty of 150\%.

Since bit objects cannot be used as subsripts in R, a second class ‘bitwhich’ allows to store selections as efficiently as possible with standard R types. This is usefull either to represent parts of bit objects or to represent very asymetric selections.

Class ‘ri’ (range index) allows to select ranges of positions for chunked processing: all three classes ‘bit’, ‘bitwhich’ and ‘ri’ can be used for subsetting ‘ff’ objects (ff-2.1.0 and higher).

Usage

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

Arguments

length length of vector in bits
x a bit vector
... further arguments to print

Details

Package: bit
Type: Package
Version: 1.1.0
Date: 2012-06-05
License: GPL-2
LazyLoad: yes
Encoding: latin1

Index:

<table>
<thead>
<tr>
<th>bit function</th>
<th>bitwhich function</th>
<th>ri function</th>
<th>see also</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.BITS</td>
<td></td>
<td></td>
<td></td>
<td>variable holding number of bits on</td>
</tr>
<tr>
<td>bit_init</td>
<td></td>
<td>.First.lib</td>
<td></td>
<td>initially allocate bit-masks (done in</td>
</tr>
<tr>
<td>bit_done</td>
<td></td>
<td>.Last.lib</td>
<td></td>
<td>finally de-allocate bit-masks (done in</td>
</tr>
<tr>
<td>bit</td>
<td>bitwhich</td>
<td>ri</td>
<td>logical</td>
<td>create bit object</td>
</tr>
<tr>
<td>print.bit</td>
<td>print.bitwhich</td>
<td>print.ri</td>
<td>print</td>
<td>print bit vector</td>
</tr>
<tr>
<td>length.bit</td>
<td>length.bitwhich</td>
<td>length.ri</td>
<td>length</td>
<td>get length of bit vector</td>
</tr>
<tr>
<td>length&lt;-.bit</td>
<td>length&lt;-.bitwhich</td>
<td></td>
<td></td>
<td>change length of bit vector</td>
</tr>
<tr>
<td>c.bit</td>
<td>c.bitwhich</td>
<td>is.ri</td>
<td>is.logical</td>
<td>concatenate bit vectors</td>
</tr>
<tr>
<td>is.bit</td>
<td>is.bitwhich</td>
<td>as.bit</td>
<td>as.logical</td>
<td>test for bit class</td>
</tr>
<tr>
<td>as.bit</td>
<td>as.bitwhich</td>
<td>as.bit.ri</td>
<td>as.bitwhich.ri</td>
<td>generically coerce to bit or bitwhich</td>
</tr>
<tr>
<td>as.bitlogical</td>
<td>as.bitwhich.logical</td>
<td></td>
<td></td>
<td>coerce logical to bit vector (FALSE)</td>
</tr>
<tr>
<td>as.bit.integer</td>
<td>as.bitwhich.integer</td>
<td></td>
<td></td>
<td>coerce integer to bit vector (0 =&gt; F)</td>
</tr>
<tr>
<td>as.bit.double</td>
<td>as.bitwhich.double</td>
<td></td>
<td>double</td>
<td>coerce double to bit vector (0 =&gt; F)</td>
</tr>
<tr>
<td>as.double.bit</td>
<td>as.double.bitwhich</td>
<td>as.double.ri</td>
<td>as.double</td>
<td>coerce bit vector to double (0/1)</td>
</tr>
<tr>
<td>as.integer.bit</td>
<td>as.integer.bitwhich</td>
<td>as.integer.ri</td>
<td>as.integer</td>
<td>coerce bit vector to integer (0L/1L)</td>
</tr>
<tr>
<td>as.logical.bit</td>
<td>as.logical.bitwhich</td>
<td>as.logical.ri</td>
<td>as.logical</td>
<td>coerce bit vector to logical (FALSE/TRUE)</td>
</tr>
<tr>
<td>as.bit.bitwhich</td>
<td>as.bitwhich.bitwhich</td>
<td></td>
<td></td>
<td>coerce bit vector to positive integer</td>
</tr>
<tr>
<td>as.bit.bit</td>
<td>as.bitwhich.bit</td>
<td>as.bit.ri</td>
<td>as.bitwhich.ri</td>
<td>coerce bitvector to bit</td>
</tr>
<tr>
<td>as.bit.ri</td>
<td>as.bitwhich.ri</td>
<td></td>
<td></td>
<td>coercion from bit</td>
</tr>
<tr>
<td>as.bit.ff</td>
<td></td>
<td></td>
<td></td>
<td>coercion from range index</td>
</tr>
<tr>
<td>as.ff.bit</td>
<td></td>
<td></td>
<td></td>
<td>coerce ff boolean to bit vector</td>
</tr>
<tr>
<td>as.hi.bit</td>
<td>as.hi.bitwhich</td>
<td>as.hi.ri</td>
<td>as.hi</td>
<td>coercion from hybrid index (requires package ff)</td>
</tr>
<tr>
<td>as.bit.hi</td>
<td>as.bitwhich.hi</td>
<td></td>
<td></td>
<td>coercion to hybrid index (requires package ff)</td>
</tr>
<tr>
<td>[.[bit</td>
<td></td>
<td></td>
<td></td>
<td>get single bit (index checked)</td>
</tr>
<tr>
<td>[[&lt;-.bit</td>
<td></td>
<td></td>
<td></td>
<td>set single bit (index checked)</td>
</tr>
<tr>
<td>[[]&lt;-.bit</td>
<td></td>
<td></td>
<td></td>
<td>get vector of bits (unindexed)</td>
</tr>
<tr>
<td>[[]&lt;-.bit</td>
<td></td>
<td></td>
<td></td>
<td>set vector of bits (unindexed)</td>
</tr>
<tr>
<td>![.bit</td>
<td>![.bitwhich</td>
<td>![</td>
<td>![</td>
<td>boolean NOT on bit</td>
</tr>
<tr>
<td>&amp;.bit</td>
<td>&amp;.bitwhich</td>
<td>&amp;</td>
<td>&amp;</td>
<td>boolean AND on bit</td>
</tr>
<tr>
<td></td>
<td>.bit</td>
<td></td>
<td>.bitwhich</td>
<td></td>
</tr>
<tr>
<td>xor.bit</td>
<td>xor.bitwhich</td>
<td>xor</td>
<td>xor</td>
<td>boolean XOR on bit</td>
</tr>
<tr>
<td>!=.bit</td>
<td>!=.bitwhich</td>
<td>!=</td>
<td>!=</td>
<td>boolean inequality (same as XOR)</td>
</tr>
<tr>
<td>==.bit</td>
<td>==.bitwhich</td>
<td>==</td>
<td>==</td>
<td>boolean equality</td>
</tr>
<tr>
<td>all.bit</td>
<td>all.bitwhich</td>
<td>all.ri</td>
<td>all</td>
<td>aggregate AND</td>
</tr>
<tr>
<td>any.bit</td>
<td>any.bitwhich</td>
<td>any.ri</td>
<td>any</td>
<td>aggregate OR</td>
</tr>
<tr>
<td>min.bit</td>
<td>min.bitwhich</td>
<td>min.ri</td>
<td>min</td>
<td>aggregate MIN (first TRUE position)</td>
</tr>
<tr>
<td>max.bit</td>
<td>max.bitwhich</td>
<td>max.ri</td>
<td>max</td>
<td>aggregate MAX (last TRUE position)</td>
</tr>
<tr>
<td>range.bit</td>
<td>range.bitwhich</td>
<td>range.ri</td>
<td>range</td>
<td>aggregate [MIN,MAX]</td>
</tr>
</tbody>
</table>
bit-package

<table>
<thead>
<tr>
<th>sum.bit</th>
<th>sum.bitwhich</th>
<th>sum.ri</th>
<th>sum</th>
<th>aggregate SUM (count of TRUE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>summary.bit</td>
<td>summary.bitwhich</td>
<td>summary.ri</td>
<td>tabulate</td>
<td>aggregate c(nFALSE, nTRUE, minRange, maxRange)</td>
</tr>
<tr>
<td>regtest.bit</td>
<td>regressiontests for the package</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Value

bit returns a vector of integer sufficiently long to store 'length' bits (but not longer) with an attribute 'n' and class 'bit'

Note

Currently operations on bit objects have some overhead from R-calls. Do expect speed gains for vectors of length ~ 10000 or longer.
Since this package was created for high performance purposes, only positive integer subscripts are allowed: All R-functions behave as expected - i.e. they do not change their arguments and create new return values. If you want to save the time for return value memory allocation, you must use .Call directly (see the dontrun example in sum.bit).

Author(s)

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

See Also

logical in base R and vmode in package 'ff'

Examples

```r
x <- bit(12) # create bit vector
x
length(x) <- 16 # change length
length(x) # get length
x[[2]] # extract single element
x[[2]] <- TRUE # replace single element
x[1:2] # extract parts of bit vector
x[1:2] <- TRUE # replace parts of bit vector
as.which(x)
x <- as.bit.which(3:4, 4) # coerce subscripts to bit
as.logical(x) # coerce bit to logical
y <- as.bit(c(FALSE, TRUE, FALSE, TRUE)) # coerce logical to bit
is.bit(y) # test for bit
!x # boolean NOT
x & y # boolean AND
x | y # boolean OR
xor(x, y) # boolean Exclusive OR
x != y # boolean unequality (same as xor)
x == y # boolean equality
all(x) # aggregate AND
any(x) # aggregate OR
```
\begin{verbatim}
min(x) # aggregate MIN (integer version of ALL)
max(x) # aggregate MAX (integer version of ANY)
range(x) # aggregate [MIN,MAX]
sum(x) # aggregate SUM (count of TRUE)
summary(x) # aggregate count of FALSE and TRUE

## Not run:
message("\nEven for a single boolean operation transforming logical to bit pays off")
n <- 10000000
x <- sample(c(FALSE, TRUE), n, TRUE)
y <- sample(c(FALSE, TRUE), n, TRUE)
system.time(x|y)
system.time({
x <- as.bit(x)
y <- as.bit(y)
})
system.time( z <- x | y )
system.time( as.logical(z) )
message("Even more so if multiple operations are needed :-)"

message("\nEven for a single set operation transforming subscripts to bit pays off\n")
n <- 10000000
x <- sample(n, n/2)
y <- sample(n, n/2)
system.time( union(x,y) )
system.time({
x <- as.bit.which(x, n)
y <- as.bit.which(y, n)
})
system.time( as.which.bit( x | y ) )
message("Even more so if multiple operations are needed :-)"

message("\nSome timings WITH memory allocation")
n <- 2000000
l <- sample(c(FALSE, TRUE), n, TRUE)
# copy logical to logical
system.time(for(i in 1:100){ # 0.0112
  l2 <- l
  l2[1] <- TRUE # force new memory allocation (copy on modify)
  rm(l2)
})/100
# copy logical to bit
system.time(for(i in 1:100){ # 0.0123
  b <- as.bit(l)
  rm(b)
})/100
# copy bit to logical
b <- as.bit(l)
system.time(for(i in 1:100){ # 0.009
  l2 <- as.logical(b)
  rm(l2)
})/100
# copy bit to bit
\end{verbatim}
b <- as.bit(l)

system.time(for(i in 1:100){  # 0.009
  b2 <- b
  b2[1] <- TRUE  # force new memory allocation (copy on modify)
  rm(b2)
})/100

l2 <- l
# replace logical by TRUE
system.time(for(i in 1:100){
  1[] <- TRUE
})/100
# replace bit by TRUE (NOTE that we recycle the assignment
# value on R side == memory allocation and assignment first)
system.time(for(i in 1:100){
  b[] <- TRUE
})/100
# THUS the following is faster
system.time(for(i in 1:100){
  b <- !bit(n)
})/100

# replace logical by logical
system.time(for(i in 1:100){
  1[] <- l2
})/100
# replace bit by logical
system.time(for(i in 1:100){
  b[] <- l2
})/100
# extract logical
system.time(for(i in 1:100){
  l2[]
})/100
# extract bit
system.time(for(i in 1:100){
  b[]
})/100

message("\nSome timings WITHOUT memory allocation (Serge, that's for you")

n <- 2000000L
l <- sample(c(FALSE, TRUE), n, TRUE)
b <- as.bit(l)
# read from logical, write to logical
l2 <- logical(n)
system.time(for(i in 1:100).Call("R_filter_getset", l, l2, PACKAGE="bit")) / 100
# read from bit, write to logical
l2 <- logical(n)
system.time(for(i in 1:100).Call("R_bit_get", b, l2, c(1L, n), PACKAGE="bit")) / 100
# read from logical, write to bit
system.time(for(i in 1:100).Call("R_bit_set", b, l2, c(1L, n), PACKAGE="bit")) / 100
## End(Not run)

### as.bit

**Description**

Coercing to bit vector

**Usage**

```r
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 'bitwhich'
as.bit(x, ...)  
## S3 method for class 'which'
as.bit(x, length, ...)  
## S3 method for class 'ri'
as.bit(x, ...)
```

**Arguments**

- `x` an object of class `bit`, `logical`, `integer`, `bitwhich` or an integer from `as.which` or a boolean `ff`
- `length` the length of the new bit vector
- `...` further arguments

**Details**

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

**Value**

- `is.bit` returns FALSE or TRUE, `as.bit` returns a vector of class 'bit'

**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 ff and may change in the future.
Author(s)
Jens Oehlschlägel

See Also
bit, as.logical

Examples

```r
x <- as.bit(c(FALSE, NA, TRUE))
as.bit(x)
as.bit.which(c(1,3,4), 12)
```
Value

a value of class `bitwhich`

Author(s)

Jens Oehlschlägel

See Also

`bitwhich`, `as.bit`

Examples

```r
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

Coercing from bit to logical, integer, which.

### Usage

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

\[ x \] an object of class \texttt{bit, bitwhich} or \texttt{ri}  

\[ \ldots \] ignored

Details

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

Value

\texttt{as.logical} returns a vector of \texttt{FALSE, TRUE}, \texttt{as.integer} and \texttt{as.double} return a vector of \texttt{0, 1}.

Author(s)

Jens Oehlschl"agel

See Also

\texttt{as.bit, as.which, as.bitwhich, as.ff, as.hi}

Examples

\begin{verbatim}
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))))
\end{verbatim}

as.which  
\textit{Coercion to (positive) integer positions}

Description

Coercing to something like the result of which \texttt{which}
Usage

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

Arguments

x an object of classes bit, bitwhich, ri or something on which which works
range a ri or an integer vector of length==2 giving a range restriction for chunked processing
... further arguments (passed to which for the default method, ignored otherwise)

Details

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

Value

a vector of class 'logical' or 'integer'

Author(s)

Jens Oehlschlägel

See Also

as.bit, as.logical, as.integer, as.which, as.bitwhich, as.ff, as.hi

Examples

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))))
**bbatch**  

**Balanced Batch sizes**

**Description**

bbatch calculates batch sizes so that they have rather balanced sizes than very different sizes.

**Usage**

```r
bbatch(N, B)
```

**Arguments**

- `N` total size
- `B` desired batch size

**Details**

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

**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**

`repfromto`, `ffvecapply`

**Examples**

```r
bbatch(100, 24)
```
bitwhich

A class for vectors representing asymmetric selections

Description

A bitwhich object like the result of `which` and `as.which` does represent integer subscript positions, but bitwhich objects represent some subscripts rather with negative integers, if this needs less space. The extreme cases of selecting all/none subscripts are represented by TRUE/FALSE. This needs less RAM compared to `logical` (and often less than `as.which`). Logical operations are fast if the selection is asymmetric (only few or almost all selected).

Usage

`bitwhich(maxindex, poslength = NULL, x = NULL)`

Arguments

- `maxindex`  the length of the vector (sum of all TRUEs and FALSEs)
- `poslength` Only use if `x` is not NULL: the sum of all TRUEs
- `x`  Default NULL or FALSE or unique negative integers or unique positive integers or TRUE

Details

class 'bitwhich' represents a boolean selection in one of the following ways

- FALSE to select nothing
- TRUE to select everything
- unique positive integers to select those
- unique negative integers to exclude those

Value

An object of class 'bitwhich' carrying two attributes

- `maxindex` see above
- `poslength` see above

Author(s)

Jens Oehlschlägel

See Also

`as.bitwhich, as.which, bit`
Examples

bitwhich(12, x=c(1,3), poslength=2)
bitwhich(12, x=-c(1,3), poslength=10)

Description

Functions to allocate (and de-allocate) bit masks

Usage

bit_init()
bit_done()

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.

Value

NULL

Author(s)

Jens Oehlschlägel

See Also

bit

Examples

bit_done()
bit_init()
c.bit

Concatenating bit and bitwhich vectors

Description
Creating new bit by concatenating bit vectors

Usage

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

Arguments

... bit objects

Value
An object of class 'bit'

Author(s)
Jens Oehlschlägel

See Also
c.bit, bitwhich

Examples

```r
c(bit(4), bit(4))```

chunk

Chunked range index

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

Usage

```r
chunk(...)```

## Default S3 method:
chunk(from = NULL, to = NULL, by = NULL, length.out = NULL, along.with = NULL, overlap = 0L, method = c("bbatch", "seq"), maxindex = NA, ...)```
Arguments

from  the starting value of the sequence.
to   the (maximal) end value of the sequence.
by   increment of the sequence
length.out  desired length of the sequence.
along.with  take the length from the length of this argument.
overlap  number of values to overlap (will lower the starting value of the sequence, the
         first range becomes smaller
method  default 'bbatch' will try to balance the chunk size, see bbatch, 'seq' will create
        chunks like seq
maxindex  passed to ri
...  ignored

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

chunk.default returns a list of ri objects representing chunks of subscripts

available methods

chunk.default, chunk.bit, chunk.ff_vector, chunk.ffdf

Author(s)

Jens Oehlschlägel

See Also

ri, seq, bbatch

Examples

chunk(1, 100, by=30)
chunk(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 chunk(1, n, by=m)){for (i in ch[1]:ch[2])z <- i}))

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 <- 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))))

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

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

## End(Not run)

---

clone

Cloning ff and ram objects

**Description**

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

**Usage**

clone(x, ...)
## S3 method for class 'list'
clone(x, ...)
## Default S3 method:
```
clone(x, ...)  
still.identical(x, y)
```

**Arguments**

- `x`  
- `y`  
- `...` further arguments to the generic

**Details**

`clone` is generic. `clone.default` currently only handles atomics. `clone.list` recursively clones list elements. `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`

**Author(s)**

Jens Oehlschlägel

**See Also**

`clone.ff`

**Examples**

```
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()
```

---

**Extract**

*Extract or replace part of an bit vector*

**Description**

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

```r
## 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
```

Arguments

- `x`: a bit object
- `i`: positive integer subscript
- `value`: new logical or integer values

Details

Since this package was created for high performance purposes, only positive integer subscripts make sense. Negative subscripts are converted to positive ones, beware the RAM consumption. Further subscript classes allowed for `[` and `[<-` are range indices `ri` and `bitwhich`. The `[` and `[<-` methods don’t check whether the subscripts are positive integers in the allowed range.

Value

The extractors `[]` and `[` return a logical scalar or vector. The replacement functions return a bit object.

Author(s)

Jens Oehlschlägel

See Also

- `bit`, `Extract`

Examples

```r
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
```
intrle

Description

These C-coded utilities speed up index preprocessing considerably.

Usage

intrle(x)
intisasc(x)
intisdesc(x)

Arguments

x

an integer vector

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 and returns NA x contains NAs.
intisdesc checks for being sorted descending and assumes that the input x contains no NAs (is used after intisasc and does not check for NAs).

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)

Author(s)

Jens Oehlschlägel

See Also

hi, rle, is.unsorted, is.sorted

Examples

intrle(sample(1:100))
intrle(diff(1:100))
intisasc(1:100)
intrle(100:1)
Testing for bit, bitwhich and ri selection classes

Description

Test whether an object inherits from 'ri', 'bit' or 'bitwhich'

Usage

is.ri(x)
is.bit(x)
is.bitwhich(x)

Arguments

x an R object of unknown type

Value

TRUE or FALSE

Author(s)

Jens Oehlenschlägel

See Also

is.logical, bit, bitwhich

Examples

is.ri(TRUE)
is.ri(ri(1,4,12))
is.bit(TRUE)
is.bitwhich(TRUE)
is.bit(as.bit(TRUE))
is.bitwhich(as.bitwhich(TRUE))
is.sorted

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
... ignored

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")
**length.bit**

**Getting and setting length of bit, bitwhich and ri objects**

**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**

```r
## S3 method for class 'ri'
length(x)
## S3 method for class 'bit'
length(x)
## S3 method for class 'bitwhich'
length(x)
## S3 replacement method for class 'bit'
length(x) <- value
## S3 replacement method for class 'bitwhich'
length(x) <- value
```

**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
**length.bit**

**Author(s)**

Jens Oehlschlägel

**See Also**

`length`, `sum`, `poslength`, `maxindex`

**Examples**

```r
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) == 1)
length(x) <- 32
stopifnot(length(x) == 32)
stopifnot(sum(x) == 1)

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))
```
LogicBit

Boolean operators and functions for class bit

Description

Boolean 'negation', 'and', 'or' and 'exclusive or'.

Usage

## S3 method for class 'bit'

```r
x <- as.bitwhich(ri(2, 32, 32))
stopifnot(length(x)==32)
stopifnot(sum(x)==31)
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)
stopifnot(sum(x)==1)
length(x) <- 16
stopifnot(length(x)==16)
stopifnot(sum(x)==1)
length(x) <- 32
stopifnot(length(x)==32)
stopifnot(sum(x)==1)

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)
```
Arguments

- **x**
  - a bit vector (or one logical vector in binary operators)

- **y**
  - a bit vector or an logical vector

- **e1**
  - a bit vector or an logical vector

- **e2**
  - a bit vector or an logical vector

Details

Binary operators and function `xor` can combine 'bit' objects and 'logical' vectors. They do not recycle, thus the lengths of objects must match. Boolean operations on bit vectors are extremely fast because they are implemented using C’s bitwise operators. If one argument is 'logical' it is converted to 'bit'.

Binary operators and function `xor` can combine 'bitwhich' objects and other vectors. They do not recycle, thus the lengths of objects must match. Boolean operations on bitwhich vectors are fast if the distribution of TRUE and FALSE is very asymmetric. If one argument is not 'bitwhich' it is converted to 'bitwhich'.
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 'bit' (or 'bitwhich')

Author(s)

Jens Oehlschlägel

See Also

`bit`, `Logic`

Examples

```r
x <- as.bit(c(FALSE, FALSE, FALSE, NA, NA, NA, TRUE, TRUE, TRUE))
yl <- c(FALSE, NA, TRUE, FALSE, NA, TRUE, FALSE, NA, TRUE)
y <- as.bit(yl)
!x
x & y
x | y
xor(x, y)
x != y
x == y
x & yl
x | yl
xor(x, yl)
x != yl
x == yl
```

```r
x <- as.bitwhich(c(FALSE, FALSE, FALSE, NA, NA, NA, TRUE, TRUE, TRUE))
yl <- c(FALSE, NA, TRUE, FALSE, NA, TRUE, FALSE, NA, TRUE)
y <- as.bitwhich(yl)
!x
x & y
x | y
xor(x, y)
x != y
x == y
x & yl
x | yl
xor(x, yl)
x != yl
x == yl
```
Physical and virtual attributes

Description

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

Usage

```r
physical(x)
virtual(x)
physical(x) <- value
virtual(x) <- value
## Default S3 method:
physical(x)
## Default S3 method:
virtual(x)
## Default S3 replacement method:
physical(x) <- value
## Default S3 replacement method:
virtual(x) <- value
## S3 method for class 'physical'
print(x, ...)
## S3 method for class 'virtual'
print(x, ...)
```

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))

---

ramsort Generics for in-RAM sorting and ordering

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

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
- i: integer positions to be modified by ramosordert and ramsortorder, default is 1:n, in this case the output is similar to order
- ...: further arguments to the sorting methods
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

<table>
<thead>
<tr>
<th>generic</th>
<th>ff</th>
<th>bit64</th>
</tr>
</thead>
<tbody>
<tr>
<td>ramsort</td>
<td>ramsort.default</td>
<td>ramsort.integer64</td>
</tr>
<tr>
<td>shellsort</td>
<td>shellsort.default</td>
<td>shellsort.integer64</td>
</tr>
<tr>
<td>quicksort</td>
<td>quicksort.default</td>
<td>quicksort.integer64</td>
</tr>
<tr>
<td>mergesort</td>
<td>mergesort.default</td>
<td>mergesort.integer64</td>
</tr>
<tr>
<td>radixsort</td>
<td>radixsort.default</td>
<td>radixsort.integer64</td>
</tr>
<tr>
<td>keysort</td>
<td>keysort.default</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>generic</th>
<th>ff</th>
<th>bit64</th>
</tr>
</thead>
<tbody>
<tr>
<td>ramorder</td>
<td>ramorder.default</td>
<td>ramorder.integer64</td>
</tr>
<tr>
<td>shellorder</td>
<td>shellorder.default</td>
<td>shellorder.integer64</td>
</tr>
<tr>
<td>quickorder</td>
<td>quickorder.default</td>
<td>quickorder.integer64</td>
</tr>
<tr>
<td>mergeorder</td>
<td>mergeorder.default</td>
<td>mergeorder.integer64</td>
</tr>
<tr>
<td>radixorder</td>
<td>radixorder.default</td>
<td>radixorder.integer64</td>
</tr>
<tr>
<td>keyorder</td>
<td>keyorder.default</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>generic</th>
<th>ff</th>
<th>bit64</th>
</tr>
</thead>
<tbody>
<tr>
<td>ramsortorder</td>
<td>ramsortorder.default</td>
<td>ramsortorder.integer64</td>
</tr>
<tr>
<td>shellsortorder</td>
<td>shellsortorder.default</td>
<td>shellsortorder.integer64</td>
</tr>
<tr>
<td>quicksortorder</td>
<td>quicksortorder.default</td>
<td>quicksortorder.integer64</td>
</tr>
<tr>
<td>mergesortorder</td>
<td>mergesortorder.default</td>
<td>mergesortorder.integer64</td>
</tr>
<tr>
<td>radixsortorder</td>
<td>radixsortorder.default</td>
<td>radixsortorder.integer64</td>
</tr>
<tr>
<td>keysortorder</td>
<td>keysortorder.default</td>
<td></td>
</tr>
</tbody>
</table>

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

Details

random data of random length are generated and correctness of package functions tested on these

Value

a vector of class 'logical' or 'integer'

Author(s)

Jens Oehlschlägel

See Also

bit, as.bit, as.logical, as.integer, which
repeat.time

Examples
if (regtest.bit()){
   message("regtest.bit is OK")
} else{
   message("regtest.bit failed")
}

## Not run:
regtest.bit(10000)

## End(Not run)

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 R 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
envir      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 \( \text{from} \) to \( \text{to} \).

Usage

\[
\text{repfromto}(x, \text{from}, \text{to})
\]
\[
\text{repfromto}(x, \text{from}, \text{to}) \leftarrow \text{value}
\]

Arguments

- \( x \): an object from which to recycle
- \( \text{from} \): first position to return
- \( \text{to} \): last position to return
- \( \text{value} \): value to assign

Details

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

Value

a vector of length \( \text{from} \) to \( \text{to} + 1 \)

Author(s)

Jens Oehlschlägel

See Also

\texttt{rep}, \texttt{ffvecapply}

Examples

\[
\text{message("a simple example")}
\]
\[
\text{repfromto}(0:9, 11, 20)
\]
ri

Range index

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

Usage
ri(from, to = NULL, maxindex=NA)
## S3 method for class 'ri'
print(x, ...)

Arguments
from first position
to last posistion
x an object of class 'ri'
maxindex the maximal length of the object-to-be-subscripted (if known)
... further arguments

Value
A two element integer vector with class 'ri'

Author(s)
Jens Oehlschlägel

See Also
as.hi.ri

Examples
bit(12)[ri(1,6)]
rlepack

Hybrid Index, rle-pack utilities

Description

Basic utilities for rle packing and unpacking and appropriate methods for `rev` and `unique`.

Usage

```r
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`
- `pack` FALSE to suppress packing
- `incomparables` just to keep R CMD CHECK quiet (not used)
- `...` just to keep R CMD CHECK quiet (not used)

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**

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

**Usage**

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

**Arguments**

- `x`
- `which`
- `value`
- `attributes`

**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.

**Author(s)**

Jens Oehlschlägel

**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**

`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
g(x)
x

## Not run:

# restart R
library(bit)

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

# show that mysingle 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){
```

setattributes
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)


## End(Not run)

### Summary

#### Summaries of bit vectors

**Description**

Fast aggregation functions for bit vectors.

**Usage**

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

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

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

```r
## S3 method for class 'bit'
```
max(x, range = NULL, ...)  
## S3 method for class 'bit'
range(x, range = NULL, ...)  
## S3 method for class 'bit'
sum(x, range = NULL, ...)  
## S3 method for class 'bit'
summary(object, range = NULL, ...)  
## S3 method for class 'bitwhich'
all(x, ...)  
## S3 method for class 'bitwhich'
any(x, ...)  
## S3 method for class 'bitwhich'
min(x, ...)  
## S3 method for class 'bitwhich'
max(x, ...)  
## S3 method for class 'bitwhich'
range(x, ...)  
## S3 method for class 'bitwhich'
sum(x, ...)  
## S3 method for class 'bitwhich'
summary(object, ...)  
## S3 method for class 'ri'
all(x, ...)  
## S3 method for class 'ri'
any(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'
sum(x, ...)  
## S3 method for class 'ri'
summary(object, ...)  

Arguments

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

Details

Bit summaries are quite fast because we use a double loop that fixes each word in a processor register. Furthermore we break out of looping as soon as possible.
Summary

Value

as expected

Author(s)

Jens Oehlschlägel

See Also

bit, all, any, min, max, range, sum, summary

Examples

x <- as.bit(c(TRUE, TRUE))
all(x)
any(x)
min(x)
max(x)
range(x)
sum(x)
summary(x)

x <- as.bitwhich(c(TRUE, TRUE))
all(x)
any(x)
min(x)
max(x)
range(x)
sum(x)
summary(x)

## Not run:

n <- .Machine$integer.max
x <- !bit(n)
N <- 1000000L # batchsize
B <- n %% N # number of batches
R <- n %/% N # rest

message("Batched sum (52.5 sec on Centrino duo)")
system.time({
    s <- 0L
    for (b in 1:B){
        s <- s + sum(x[((b-1L)*N+1L):(b*N)])
    }
    if (R)
        s <- s + sum(x[(n-R+1L):n])
})

message("Batched sum saving repeated memory allocation for the return vector
(44.4 sec on Centrino duo)")
system.time({
    s <- 0L
})
1 <- logical(N)
for (b in 1:B){
  .Call("R_bit_extract", x, length(x), ((b-1L)*N+1L):(b*N), l, PACKAGE = "bit")
  s <- s + sum(l)
}
if (R)
  s <- s + sum(x[(n-R+1L):n])
}
message("C-coded sum (3.1 sec on Centrino duo)")
system.time(sum(x))

## End(Not run)

---

**unattr**

**Attribute removal**

**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

**Author(s)**

Jens Oehlschlägel

**See Also**

attributes, setattributes, unclass

**Examples**

```r
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 concatenated sequences

Author(s)

Angelo Canty, Jens Oehlschlägel

See Also

::, seq, sequence
Examples

\begin{verbatim}
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)
\end{verbatim}
Index

*!.bit (LogicBit), 26
*!.bitwhich (LogicBit), 26
*!=.bit (LogicBit), 26
*!=.bitwhich (LogicBit), 26
*Topic **IO**
   *bbatch*, 13
   *clone*, 18
   *intrle*, 21
   *physical*, 29
   *reppfromto*, 34
   *rlepack*, 36
*Topic **arith**
   *rmonsort*, 30
*Topic **attributes**
   *setattributes*, 37
*Topic **attribute**
   *physical*, 29
   *unattr*, 42
*Topic **classes**
   *as.bit*, 8
   *as.bitwhich*, 9
   *as.logical.bit*, 10
   *as.which*, 11
   *bit-package*, 3
   *bit_init*, 15
   *bitwhich*, 14
   *c.bit*, 16
   *Extract*, 19
   *is.bit*, 22
   *length.bit*, 24
   *LogicBit*, 26
   *regtest.bit*, 32
   *ri*, 35
   *Summary*, 39
*Topic **manip**
   *rmonsort*, 30
   *vecseq*, 43
*Topic **methods**
   *is.sorted*, 23
*Topic **package**
   *bit-package*, 3
*Topic **univar**
   *rmonsort*, 30
*Topic **utilities**
   *repeat.time*, 33
   *.BITS (bit_init), 15
   .*Call*, 5
   .*First.lib*, 4, 15
   .*Last.lib*, 4, 15
   ::=, 43
   ==, 4
   ==.bit, 4

physical, 29
reppfromto, 34
rlepack, 36
*Topic **environment**
is.sorted, 23
*Topic **logic**
as.bit, 8
as.bitwhich, 9
as.logical.bit, 10
as.which, 11
bit-package, 3
bit_init, 15
bitwhich, 14
c.bit, 16
Extract, 19
is.bit, 22
length.bit, 24
LogicBit, 26
regtest.bit, 32
ri, 35
Summary, 39
*Topic **manip**
rmonsort, 30
vecseq, 43
*Topic **methods**
is.sorted, 23
*Topic **package**
bit-package, 3
*Topic **univar**
rmonsort, 30
*Topic **utilities**
repeat.time, 33
.BITS (bit_init), 15
.Call, 5
.First.lib, 4, 15
.Last.lib, 4, 15
::, 43
==, 4
==.bit, 4

45
==.bit (LogicBit), 26
==.bitwhich, 4
==.bitwhich (LogicBit), 26
[.4
[.bit, 4
[.bit (Extract), 19
[<-.4
[<-.bit, 4
[<-.bit (Extract), 19
[:4
[[.bit, 4
[[.bit (Extract), 19
[[<-.4
[[<-.bit, 4
[[<-.bit (Extract), 19
&.4
&.bit, 4
&.bit (LogicBit), 26
&.bitwhich, 4
&.bitwhich (LogicBit), 26

all, 4, 41
all.bit, 4
all.bit (Summary), 39
all.bitwhich, 4
all.bitwhich (Summary), 39
all.ri, 4
all.ri (Summary), 39
any, 4, 41
any.bit, 4
any.bit (Summary), 39
any.bitwhich, 4
any.bitwhich (Summary), 39
any.ri, 4
any.ri (Summary), 39
anyDuplicated.rlepack (rlepack), 36
as.bit, 4, 8, 10–12, 32
as.bit, 4
as.bit.bitwhich, 4
as.bit.double, 4
as.bit.ff, 4
as.bit.hl, 4
as.bit.integer, 4
as.bit.logical, 4
as.bit.ri, 4
as.bit.which, 4
as.bitwhich, 4, 9, 11, 12, 14
as.bitwhich.bit, 4
as.bitwhich.bitwhich, 4
as.bitwhich.double, 4
as.bitwhich.hi, 4
as.bitwhich.integer, 4
as.bitwhich.logical, 4
as.bitwhich.ri, 4
as.bitwhich.which, 4
as.double, 4, 11
as.double.bit, 4
as.double.bit (as.logical.bit), 10
as.double.bitwhich, 4
as.double.bitwhich (as.logical.bit), 10
as.double.ri, 4
as.double.ri (as.logical.bit), 10
as.ff, 4, 11, 12, 29
as.ff.bit, 4
as.hl, 4, 11, 12
as.hl.bit, 4
as.hl.bitwhich, 4
as.hl.ri, 4, 35
as.integer, 4, 11, 12, 32
as.integer.bit, 4
as.integer.bit (as.logical.bit), 10
as.integer.bitwhich, 4
as.integer.bitwhich (as.logical.bit), 10
as.integer.ri, 4
as.integer.ri (as.logical.bit), 10
as.logical, 4, 9, 11, 12, 32
as.logical.bit, 4, 10
as.logical.bitwhich, 4
as.logical.bitwhich (as.logical.bit), 10
as.logical.ri, 4
as.logical.ri (as.logical.bit), 10
as.ram, 29
as.which, 4, 8, 11, 11, 12, 14
as.which.bit, 4
as.which.bitwhich, 4
as.which.ri, 4
attr, 37
attributes, 37, 42

bbatch, 13, 17
bit, 4, 8, 9, 11, 12, 14–16, 20, 22, 24, 28, 32, 41
bit (bit-package), 3
bit-package, 3
bit_done, 4
bit_done (bit_init), 15
bit_init, 4, 15
bitwhich, 4, 8, 10–12, 14, 16, 20, 22, 24
INDEX

47

c, 4, 16

c.bit, 4, 16

c.bitwhich, 4

c.bitwhich(c.bit), 16

chunk, 16

chunk.bit, 17

chunk.ff_vector, 17

chunk.ffdf, 17

clone, 18

clone.ff, 19

double, 4

Extract, 19, 20

ff, 4, 8

ffvecapply, 13, 34

globalenv, 4

hi, 21, 36

integer, 4, 8

intisasc (intrle), 21

intisdesc (intrle), 21

intrle, 21, 36

is.bit, 4, 22

is.bitwhich, 4

is.bitwhich(is.bit), 22

is.logical, 4, 22

is.ri, 4

is.ri(is.bit), 22

is.sorted, 21, 23

is.sorted.integer64, 23

is.sorted<- (is.sorted), 23

is.unsorted, 21

keyorder (ramsort), 30

keyorder.default, 31

keysort (ramsort), 30

keysort.default, 31

keysortorder (ramsort), 30

length, 4, 25

length.bit, 4, 24

length.bitwhich, 4

length.bitwhich(length.bit), 24

length.ri, 4

length.ri(length.bit), 24

length<-, 4

length<-.bit, 4

length<-.bitwhich, 4

length<-.bitwhich(length.bit), 24

length<-.bit(length.bit), 24

Logic, 28

logical, 4, 5, 8, 14

LogicBit, 26

max, 4, 41

max.bit, 4

max.bit(Summary), 39

max.bitwhich, 4

max.bitwhich(Summary), 39

max.ri, 4

max.ri(Summary), 39

maxindex, 25

mergeorder (ramsort), 30

mergeorder.default, 31

mergeorder.integer64, 31

mergesort (ramsort), 30

mergesort.default, 31

mergesort.integer64, 31

mergesortorder (ramsort), 30

mergesortorder.integer64, 31

min, 4, 41

min.bit, 4

min.bit(Summary), 39

min.bitwhich, 4

min.bitwhich(Summary), 39

min.ri, 4

min.ri(Summary), 39

na.count(is.sorted), 23

na.count.integer64, 23

na.count<- (is.sorted), 23

nties(is.sorted), 23

nties.integer64, 23

nties<- (is.sorted), 23

nunique(is.sorted), 23

nunique.integer64, 23

nunique<- (is.sorted), 23

nvalid(is.sorted), 23

nvalid.integer64, 23

order, 30, 32

physical, 29

physical.ff, 30

physical.ffdf, 30
physical<- (physical), 29
poslength, 25
print, 4
print.bit, 4
print.bit (bit-package), 3
print.bitwhich, 4
print.bitwhich (bitwhich), 14
print.physical (physical), 29
print.ri, 4
print.ri (ri), 35
print.virtual (physical), 29
proc.time, 33
quickorder (ramsort), 30
quickorder.integer64, 31
quicksort (ramsort), 30
quicksort.integer64, 31
quicksortorder (ramsort), 30
quicksortorder.integer64, 31
radixorder (ramsort), 30
radixorder.default, 31
radixorder.integer64, 31
radixsort (ramsort), 30
radixsort.default, 31
radixsort.integer64, 31
radixsortorder (ramsort), 30
radixsortorder.integer64, 31
ramorder, 30
ramorder (ramsort), 30
ramorder.default, 31
ramorder.integer64, 31
ramsort, 30, 30
ramsort.default, 31
ramsort.integer64, 31
ramsortorder, 30
ramsortorder (ramsort), 30
ramsortorder.integer64, 31
range, 4, 41
range.bit, 4
range.bit (Summary), 39
range.bitwhich, 4
range.bitwhich (Summary), 39
range.ri, 4
range.ri (Summary), 39
regtest.bit, 5, 32
rep: 34
repeat.time, 33
repmatto<- (repmatto), 34
rev, 36
rev.rlepack (rlepack), 36
ri, 4, 9, 11, 12, 17, 20, 24, 35, 40
rle, 21, 36
rlepack, 36
rleunpack (rlepack), 36
seq, 17, 43
sequence, 43
setattr (setattributes), 37
setattributes, 37, 42
shellorder (ramsort), 30
shellorder.default, 31
shellorder.integer64, 31
shellsort (ramsort), 30
shellsort.default, 31
shellsort.integer64, 31
shellsortorder (ramsort), 30
shellsortorder.integer64, 31
sort, 30, 32
still.identical (clone), 18
sum, 5, 25, 41
sum.bit, 5
sum.bit (Summary), 39
sum.bitwhich, 5
sum.bitwhich (Summary), 39
sum.ri, 5
sum.ri (Summary), 39
Summary, 39
summary, 41
summary.bit, 5
summary.bit (Summary), 39
summary.bitwhich, 5
summary.bitwhich (Summary), 39
summary.ri, 5
summary.ri (Summary), 39
system.time, 33

tabulate, 5
unattr, 37, 42
unclass, 42
unique, 36
unique.rlepack (rlepack), 36
UseMethod, 4
vecseq, 43
virtual (physical), 29
virtual<-(physical), 29
vmode, 5

which, 11, 12, 14, 32

xor, 4, 28
xor (LogicBit), 26
xor.bit, 4
xor.bitwhich, 4