Package 'bit64'

May 7, 2026

Title A S3 Class for Vectors of 64bit Integers

Version 4.8.0

Depends R (>= 3.5.0)

Description Package 'bit64' provides serializable S3 atomic 64bit (signed) integers.
These are useful for handling database keys and exact counting in +-2^63.
WARNING: do not use them as replacement for 32bit integers, integer64 are not supported for subscripting by R-core and they have different semantics when combined with double, e.g. integer64 + double => integer64.
Class integer64 can be used in vectors, matrices, arrays and data.frames.
Methods are available for coercion from and to logicals, integers, doubles, characters and factors as well as many elementwise and summary functions.
Many fast algorithmic operations such as 'match' and 'order' support interactive data exploration and manipulation and optionally leverage caching.

License GPL-2 | GPL-3

LazyLoad yes

ByteCompile yes

URL https://github.com/r-lib/bit64, https://bit64.r-lib.org

Encoding UTF-8

Imports bit (>= 4.0.0), graphics, methods, stats, utils

Suggests patrick (>= 0.3.0), testthat (>= 3.3.0), withr

Config/testthat/edition 3

Config/Needs/development patrick, testthat

Config/Needs/website tidyverse/tidytemplate

RoxygenNote 7.3.3

NeedsCompilation yes

Author Michael Chirico [aut, cre],
Jens Oehlschlägel [aut],
Leonardo Silvestri [ctb],
Ofek Shilon [ctb],
Christian Ullerich [ctb]

Maintainer Michael Chirico <michaelchirico4@gmail.com>

Repository CRAN

Date/Publication 2026-04-21 06:10:02 UTC

Contents

`all.equal.integer64` *Test if two integer64 vectors are all.equal*

Description

A utility to compare integer64 objects 'x' and 'y' testing for 'near equality', see `all.equal()`.

Usage

```
## S3 method for class 'integer64'
all.equal(
  target,
  current,
  tolerance = sqrt(.Machine$double.eps),
  scale = NULL,
  countEQ = FALSE,
  formatFUN = function(err, what) format(err),
  ...,
  check.attributes = TRUE
)
```

Arguments

`target`	a vector of 'integer64' or an object that can be coerced with `as.integer64()`
`current`	a vector of 'integer64' or an object that can be coerced with `as.integer64()`
`tolerance`	numeric > 0. Differences smaller than `tolerance` are not reported. The default value is close to `1.5e-8`.
`scale`	`NULL` or numeric > 0, typically of length 1 or `length(target)`. See Details.
`countEQ`	logical indicating if the `target == current` cases should be counted when computing the mean (absolute or relative) differences. The default, `FALSE` may seem misleading in cases where `target` and `current` only differ in a few places; see the extensive example.
`formatFUN`	a `function()` of two arguments, `err`, the relative, absolute or scaled error, and `what`, a character string indicating the *kind* of error; maybe used, e.g., to format relative and absolute errors differently.
`...`	further arguments are ignored
`check.attributes`	logical indicating if the `attributes()` of `target` and `current` (other than the names) should be compared.

Details

In `all.equal.numeric()` the type `integer` is treated as a proper subset of `double` i.e. does not complain about comparing `integer` with `double`. Following this logic `all.equal.integer64` treats `integer` as a proper subset of `integer64` and does not complain about comparing `integer`

with integer64. double also compares without warning as long as the values are within lim.integer64(), if double are bigger all.equal.integer64 complains about the all.equal.integer64 overflow warning. For further details see all.equal().

Value

Either 'TRUE' ('NULL' for 'attr.all.equal') or a vector of 'mode' '"character"' describing the differences between 'target' and 'current'.

Note

all.equal() only dispatches to this method if the first argument is integer64, calling all.equal() with a non-integer64 first and a integer64 second argument gives undefined behavior!

See Also

all.equal()

Examples

```
all.equal(as.integer64(1:10), as.integer64(0:9))
all.equal(as.integer64(1:10), as.integer(1:10))
all.equal(as.integer64(1:10), as.double(1:10))
all.equal(as.integer64(1), as.double(1e300))
```

---

as.character.integer64

Coerce from integer64

Description

Methods to coerce integer64 to other atomic types. 'as.bitstring' coerces to a human-readable bit representation (strings of zeroes and ones). The methods format(), as.character(), as.double(), as.logical(), as.integer() do what you would expect.

Usage

```
as.bitstring(x, ...)

## S3 method for class 'integer64'
as.double(x, ...)

## S3 method for class 'integer64'
as.numeric(x, ...)

## S3 method for class 'integer64'
as.complex(x, ...)
```

```
## S3 method for class 'integer64'
as.integer(x, ...)

## S3 method for class 'integer64'
as.raw(x, ...)

## S3 method for class 'integer64'
as.logical(x, ...)

## S3 method for class 'integer64'
as.character(x, ...)

## S3 method for class 'integer64'
as.bitstring(x, ...)

## S3 method for class 'integer64'
as.Date(x, origin, ...)

## S3 method for class 'integer64'
as.POSIXct(x, tz = "", origin, ...)

## S3 method for class 'integer64'
as.POSIXlt(x, tz = "", origin, ...)

as.factor(x)

as.ordered(x)

## S3 method for class 'integer64'
as.list(x, ...)
```

Arguments

x	an integer64 vector
..., origin, tz	further arguments to the NextMethod()

Value

as.bitstring returns a string of class 'bitstring'.

The other methods return atomic vectors of the expected types

See Also

as.integer64.character() integer64()

Examples

```
as.character(lim.integer64())
as.bitstring(lim.integer64())
```

```
as.bitstring(as.integer64(c(-2, -1, NA, 0:2)))
```

`as.data.frame.integer64` *integer64: Coercing to data.frame column*

Description

Coercing integer64 vector to data.frame.

Usage

```
## S3 method for class 'integer64'
as.data.frame(x, row.names = NULL, optional = FALSE, ...)
```

Arguments

`x`	an integer64 vector
`row.names, optional, ...`	passed to NextMethod `as.data.frame()` after removing the 'integer64' class attribute

Details

'as.data.frame.integer64' is rather not intended to be called directly, but it is required to allow integer64 as data.frame columns.

Value

a one-column data.frame containing an integer64 vector

Note

This is currently very slow – any ideas for improvement?

See Also

`cbind.integer64()` `integer64()`

Examples

```
as.data.frame(as.integer64(1:12))
data.frame(a=1:12, b=as.integer64(1:12))
```

`as.integer64.character` *Coerce to integer64*

Description

Methods to coerce from other atomic types to integer64.

Usage

```
as.integer64(x, ...)

## S3 method for class '`NULL`'
as.integer64(x, ...)

## S3 method for class 'integer64'
as.integer64(x, ..., keep.names = FALSE)

## S3 method for class 'double'
as.integer64(x, ..., keep.names = FALSE)

## S3 method for class 'complex'
as.integer64(x, ...)

## S3 method for class 'integer'
as.integer64(x, ...)

## S3 method for class 'raw'
as.integer64(x, ...)

## S3 method for class 'logical'
as.integer64(x, ...)

## S3 method for class 'character'
as.integer64(x, ...)

## S3 method for class 'factor'
as.integer64(x, ...)

## S3 method for class 'Date'
as.integer64(x, ...)

## S3 method for class 'POSIXct'
as.integer64(x, ...)

## S3 method for class 'POSIXlt'
as.integer64(x, ...)
```

```
## S3 method for class 'difftime'
as.integer64(x, units = "auto", ...)

## S3 method for class 'bitstring'
as.integer64(x, ...)

NA_integer64_
```

Arguments

x	an atomic vector
..., units	further arguments to the NextMethod()
keep.names	Logical, default FALSE. If TRUE, the input's names are retained.

Format

An object of class integer64 of length 1.

Details

as.integer64.character is realized using C function strtoll which does not support scientific notation. Instead of '1e6' use '1000000'. as.integer64.bitstring evaluates characters '0' and ' ' as zero-bit, all other one byte characters as one-bit, multi-byte characters are not allowed, strings shorter than 64 characters are treated as if they were left-padded with '0', strings longer than 64 bytes are mapped to NA_INTEGER64 and a warning is emitted.

Value

The other methods return atomic vectors of the expected types

See Also

as.character.integer64() integer64()

Examples

```
as.integer64(as.character(lim.integer64()))
as.integer64(
  structure(c("1111111111111111111111111111111111111111111111111111111111111110",
              "1111111111111111111111111111111111111111111111111111111111111111",
              "1000000000000000000000000000000000000000000000000000000000000000",
              "0000000000000000000000000000000000000000000000000000000000000000",
              "0000000000000000000000000000000000000000000000000000000000000001",
              "0000000000000000000000000000000000000000000000000000000000000010"
  ), class = "bitstring")
)
as.integer64(
 structure(c("............................................................ ",
             ".............................................................",
```

```
              ".                                                                    ",
              "",
              ".",
              "10"
  ), class = "bitstring")
)
```

---

benchmark64 *Function for measuring algorithmic performance of high-level and low-level integer64 functions*

Description

Function for measuring algorithmic performance of high-level and low-level integer64 functions

Usage

```
benchmark64(nsmall = 2L^16L, nbig = 2L^25L, timefun = repeat.time)

optimizer64(
  nsmall = 2L^16L,
  nbig = 2L^25L,
  timefun = repeat.time,
  what = c("match", "%in%", "duplicated", "unique", "unipos", "table", "rank",
    "quantile", "factor"),
  uniorder = c("original", "values", "any"),
  taborder = c("values", "counts"),
  plot = TRUE
)
```

Arguments

nsmall	size of smaller vector
nbig	size of larger bigger vector
timefun	a function for timing such as bit::repeat.time() or system.time()
what	a vector of names of high-level functions
uniorder	one of the order parameters that are allowed in unique.integer64() and unipos.integer64()
taborder	one of the order parameters that are allowed in table()
plot	set to FALSE to suppress plotting

Details

benchmark64 compares the following scenarios for the following use cases:

scenario name	explanation
32-bit	applying Base R function to 32-bit integer data

64-bit	applying bit64 function to 64-bit integer data (with no cache)
hashcache	ditto when cache contains `hashmap()`, see `hashcache()`
sortordercache	ditto when cache contains sorting and ordering, see `sortordercache()`
ordercache	ditto when cache contains ordering only, see `ordercache()`
allcache	ditto when cache contains sorting, ordering and hashing

use case name	**explanation**
cache	filling the cache according to scenario
match(s, b)	match small in big vector
s %in% b	small %in% big vector
match(b, s)	match big in small vector
b %in% s	big %in% small vector
match(b, b)	match big in (different) big vector
b %in% b	big %in% (different) big vector
duplicated(b)	duplicated of big vector
unique(b)	unique of big vector
table(b)	table of big vector
sort(b)	sorting of big vector
order(b)	ordering of big vector
rank(b)	ranking of big vector
quantile(b)	quantiles of big vector
summary(b)	summary of of big vector
factor(b)	coercion to factor of big vector
SESSION	exemplary session involving multiple calls (including cache filling costs)

Note that the timings for the cached variants do *not* contain the time costs of building the cache, except for the timing of the exemplary user session, where the cache costs are included in order to evaluate amortization.

Value

`benchmark64` returns a matrix with elapsed seconds, different high-level tasks in rows and different scenarios to solve the task in columns. The last row named 'SESSION' contains the elapsed seconds of the exemplary sesssion.

`optimizer64` returns a dimensioned list with one row for each high-level function timed and two columns named after the values of the `nsmall` and `nbig` sample sizes. Each list cell contains a matrix with timings, low-level-methods in rows and three measurements `c("prep", "both", "use")` in columns. If it can be measured separately, `prep` contains the timing of preparatory work such as sorting and hashing, and `use` contains the timing of using the prepared work. If the function timed does both, preparation and use, the timing is in `both`.

Functions

- `benchmark64()`: compares high-level integer64 functions against the integer functions from Base R

- optimizer64(): compares for each high-level integer64 function the Base R integer function with several low-level integer64 functions with and without caching

See Also

integer64()

Examples

```
message("this small example using system.time does not give serious timings\n
this we do this only to run regression tests")
benchmark64(nsmall=2^7, nbig=2^13, timefun=function(expr)system.time(expr, gcFirst=FALSE))
optimizer64(nsmall=2^7, nbig=2^13, timefun=function(expr)system.time(expr, gcFirst=FALSE)
, plot=FALSE
)
## Not run:
message("for real measurement of sufficiently large datasets run this on your machine")
benchmark64()
optimizer64()

## End(Not run)
message("let's look at the performance results on Core i7 Lenovo T410 with 8 GB RAM")
data(benchmark64.data)
print(benchmark64.data)

matplot(log2(benchmark64.data[-1, 1]/benchmark64.data[-1, ])
, pch=c("3", "6", "h", "s", "o", "a")
, xlab="tasks [last=session]"
, ylab="log2(relative speed) [bigger is better]"
)
matplot(t(log2(benchmark64.data[-1, 1]/benchmark64.data[-1, ]))
, type="b", axes=FALSE
, lwd=c(rep(1, 14), 3)
, xlab="context"
, ylab="log2(relative speed) [bigger is better]"
)
axis(1
, labels=c("32-bit", "64-bit", "hash", "sortorder", "order", "hash+sortorder")
, at=1:6
)
axis(2)
data(optimizer64.data)
print(optimizer64.data)
oldpar <- par(no.readonly = TRUE)
par(mfrow=c(2, 1))
par(cex=0.7)
for (i in 1:nrow(optimizer64.data)) {
 for (j in 1:2) {
   tim <- optimizer64.data[[i, j]]
  barplot(t(tim))
  if (rownames(optimizer64.data)[i]=="match")
  title(paste("match", colnames(optimizer64.data)[j], "in", colnames(optimizer64.data)[3-j]))
  else if (rownames(optimizer64.data)[i]=="%in%")
```

```
   title(paste(colnames(optimizer64.data)[j], "%in%", colnames(optimizer64.data)[3-j]))
  else
   title(paste(rownames(optimizer64.data)[i], colnames(optimizer64.data)[j]))
 }
}
par(mfrow=c(1, 1))
```

---

benchmark64.data *Results of performance measurement on a Core i7 Lenovo T410 8 GB RAM under Windows 7 64bit*

Description

These are the results of calling benchmark64()

Usage

```
data(benchmark64.data)
```

Format

The format is:

```
num [1:16, 1:6] 2.55e-05 2.37 2.39 1.28 1.39 ...
- attr(*, "dimnames")=List of 2
..$ : chr [1:16] "cache" "match(s,b)" "s %in% b" "match(b,s)" ...
..$ : chr [1:6] "32-bit" "64-bit" "hashcache" "sortordercache" ...
```

Examples

```
data(benchmark64.data)
print(benchmark64.data)
matplot(log2(benchmark64.data[-1,1]/benchmark64.data[-1,])
, pch=c("3", "6", "h", "s", "o", "a")
, xlab="tasks [last=session]"
, ylab="log2(relative speed) [bigger is better]"
)
matplot(t(log2(benchmark64.data[-1,1]/benchmark64.data[-1,]))
, axes=FALSE
, type="b"
, lwd=c(rep(1, 14), 3)
, xlab="context"
, ylab="log2(relative speed) [bigger is better]"
)
axis(1
, labels=c("32-bit", "64-bit", "hash", "sortorder", "order", "hash+sortorder")
, at=1:6
)
axis(2)
```

bit64S3 *Turning base R functions into S3 generics for bit64*

Description

Turn those base functions S3 generic which are used in bit64

Usage

```
from:to
is.double(x)
match(x, table, ...)
x %in% table
rank(x, ...)
order(...)

## Default S3 method:
is.double(x)

## S3 method for class 'integer64'
is.double(x)

## S3 method for class 'integer64'
mtfrm(x)

## Default S3 method:
match(x, table, ...)

## Default S3 method:
x %in% table

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

## Default S3 method:
order(...)
```

Arguments

x	integer64 vector: the values to be matched, optionally carrying a cache created with hashcache()
table	integer64 vector: the values to be matched against, optionally carrying a cache created with hashcache() or sortordercache()
...	ignored
from	scalar denoting first element of sequence
to	scalar denoting last element of sequence

Details

The following functions are turned into S3 generics in order to dispatch methods for `integer64()`:

- `:`
- `is.double()`
- `match()`
- `%in%`
- `rank()`
- `order()`

Value

`invisible()`

Note

- `is.double()` returns `FALSE` for `integer64`
- `:` currently only dispatches at its first argument, thus `as.integer64(1):9` works but `1:as.integer64(9)` doesn't
- `match()` currently only dispatches at its first argument and expects its second argument also to be integer64, otherwise throws an error. Beware of something like `match(2, as.integer64(0:3))`
- `%in%` currently only dispatches at its first argument and expects its second argument also to be integer64, otherwise throws an error. Beware of something like `2 %in% as.integer64(0:3)`
- `order()` currently only orders a single argument, trying more than one raises an error

See Also

`bit64()`, S3

Examples

```
is.double(as.integer64(1))
as.integer64(1):9
match(as.integer64(2), as.integer64(0:3))
as.integer64(2) %in% as.integer64(0:3)

unique(as.integer64(c(1,1,2)))
rank(as.integer64(c(1,1,2)))


order(as.integer64(c(1,NA,2)))
```

`c.integer64` *Concatenating integer64 vectors*

Description

The ususal functions 'c', 'cbind' and 'rbind'

Usage

```
## S3 method for class 'integer64'
c(..., recursive = FALSE)

## S3 method for class 'integer64'
cbind(..., deparse.level = 1)

## S3 method for class 'integer64'
rbind(..., deparse.level = 1)
```

Arguments

`...`	two or more arguments coerced to 'integer64' and passed to `NextMethod()`
`recursive`	logical. If `recursive = TRUE`, the function recursively descends through lists (and pairlists) combining all their elements into a vector.
`deparse.level`	integer controlling the construction of labels in the case of non-matrix-like arguments

Value

`c()` returns a vector of the appropriate mode. This could be a integer64 vector or a list of objects

`cbind()` and `rbind()` return a matrix, data.frame or list with dimensions

Note

R currently only dispatches generic 'c' to method 'c.integer64' if the first argument is 'integer64'

See Also

`rep.integer64()` `seq.integer64()` `as.data.frame.integer64()` `integer64()`

Examples

```
c(as.integer64(1), 2:6)
cbind(1:6, as.integer64(1:6))
rbind(1:6, as.integer64(1:6))
```

cache *Atomic Caching*

Description

Functions for caching results attached to atomic objects

Usage

```
newcache(x)

jamcache(x)

cache(x)

setcache(x, which, value)

getcache(x, which)

remcache(x)

## S3 method for class 'cache'
print(x, all.names = FALSE, pattern, ...)
```

Arguments

x	an integer64 vector (or a cache object in case of `print.cache`)
which	A character naming the object to be retrieved from the cache or to be stored in the cache
value	An object to be stored in the cache
all.names, pattern	passed to `ls()` when listing the cache content
...	ignored

Details

A cache is an `environment` attached to an atomic object with the `attribute` name 'cache'. It contains at least a reference to the atomic object that carries the cache. This is used when accessing the cache to detect whether the object carrying the cache has been modified meanwhile.

Value

See details

Functions

- `newcache()`: creates a new cache referencing `x`
- `jamcache()`: forces `x` to have a cache
- `cache()`: returns the cache attached to `x` if it is not found to be outdated
- `setcache()`: assigns a value into the cache of `x`
- `getcache()`: gets cache value 'which' from `x`
- `remcache()`: removes the cache from `x`

See Also

`bit::still.identical()` for testing whether to symbols point to the same RAM.

Functions that get and set small cache-content automatically when a cache is present: `bit::na.count()`, `bit::nvalid()`, `bit::is.sorted()`, `bit::nunique()` and `bit::nties()`

Setting big caches with a relevant memory footprint requires a conscious decision of the user: `hashcache`, `sortcache`, `ordercache`, `sortordercache`

Functions that use big caches: `match.integer64()`, `%in%.integer64`, `duplicated.integer64()`, `unique.integer64()`, `unipos()`, `table()`, `keypos()`, `tiepos()`, `rank.integer64()`, `prank()`, `qtile()`, `quantile.integer64()`, `median.integer64()`, and `summary.integer64()`

Examples

```
x = as.integer64(sample(c(rep(NA, 9), 1:9), 32, TRUE))
y = x
bit::still.identical(x, y)
y[1] = NA
bit::still.identical(x, y)
mycache = newcache(x)
ls(mycache)
mycache
rm(mycache)
jamcache(x)
cache(x)
x[1] = NA
cache(x)
getcache(x, "abc")
setcache(x, "abc", 1)
getcache(x, "abc")
remcache(x)
cache(x)
```

cumsum.integer64 *Cumulative Sums, Products, Extremes and lagged differences*

Description

Cumulative Sums, Products, Extremes and lagged differences

Usage

```
## S3 method for class 'integer64'
diff(x, lag = 1L, differences = 1L, ...)

## S3 method for class 'integer64'
cummin(x)

## S3 method for class 'integer64'
cummax(x)

## S3 method for class 'integer64'
cumsum(x)

## S3 method for class 'integer64'
cumprod(x)
```

Arguments

x	an atomic vector of class 'integer64'
lag	see diff()
differences	see diff()
...	ignored

Value

cummin(), cummax() , cumsum() and cumprod() return a integer64 vector of the same length as their input

diff() returns a integer64 vector shorter by lag*differences elements

See Also

sum.integer64() integer64()

Examples

```
cumsum(rep(as.integer64(1), 12))
diff(as.integer64(c(0, 1:12)))
cumsum(as.integer64(c(0, 1:12)))
diff(cumsum(as.integer64(c(0, 0, 1:12))), differences=2)
```

duplicated.integer64 *Determine Duplicate Elements of integer64*

Description

duplicated() determines which elements of a vector or data frame are duplicates of elements with smaller subscripts, and returns a logical vector indicating which elements (rows) are duplicates.

Usage

```
## S3 method for class 'integer64'
duplicated(x, incomparables = FALSE, nunique = NULL, method = NULL, ...)
```

Arguments

x	a vector or a data frame or an array or NULL.
incomparables	ignored
nunique	NULL or the number of unique values (including NA). Providing nunique can speed-up matching when x has no cache. Note that a wrong nunique can cause undefined behaviour up to a crash.
method	NULL for automatic method selection or a suitable low-level method, see details
...	ignored

Details

This function automatically chooses from several low-level functions considering the size of x and the availability of a cache.

Suitable methods are

- hashdup (hashing)
- sortorderdup (fast ordering)
- orderdup (memory saving ordering).

Value

duplicated(): a logical vector of the same length as x.

See Also

duplicated(), unique.integer64()

Examples

```
x <- as.integer64(sample(c(rep(NA, 9), 1:9), 32, TRUE))
duplicated(x)

stopifnot(identical(duplicated(x),  duplicated(as.integer(x))))
```

`extract.replace.integer64` *Extract or Replace Parts of an integer64 vector*

Description

Methods to extract and replace parts of an integer64 vector.

Usage

```
## S3 method for class 'integer64'
x[i, j, ..., drop = TRUE]

## S3 replacement method for class 'integer64'
x[...] <- value

## S3 method for class 'integer64'
x[[...]]

## S3 replacement method for class 'integer64'
x[[...]] <- value
```

Arguments

`x`	an atomic vector
`i, j`	indices specifying elements to extract
`...`	further arguments to the `NextMethod()`
`drop`	relevant for matrices and arrays. If TRUE the result is coerced to the lowest possible dimension.
`value`	an atomic vector with values to be assigned

Value

A vector, matrix, array or scalar of class 'integer64'

Note

You should not subscript non-existing elements and not use `NA`s as subscripts. The current implementation returns `9218868437227407266` instead of `NA`.

See Also

`[` `integer64()`

Examples

```
as.integer64(1:12)[1:3]
x <- matrix(as.integer64(1:12), nrow = 3L)
x
x[]
x[, 2:3]
```

`factor` *Factors*

Description

The function factor is used to encode a vector as a factor.

Usage

```
factor(
  x = character(),
  levels,
  labels = levels,
  exclude = NA,
  ordered = is.ordered(x),
  nmax = NA
)

ordered(x = character(), ...)
```

Arguments

`x`	a vector of data, usually taking a small number of distinct values.
`levels`	an optional vector of the unique values (as character strings) that `x` might have taken. The default is the unique set of values taken by `as.character(x)`, sorted into increasing order *of* `x`. Note that this set can be specified as smaller than `sort(unique(x))`.
`labels`	*either* an optional character vector of labels for the levels (in the same order as `levels` after removing those in `exclude`), *or* a character string of length 1. Duplicated values in `labels` can be used to map different values of `x` to the same factor level.
`exclude`	a vector of values to be excluded when forming the set of levels. This may be factor with the same level set as `x` or should be a `character`.
`ordered`	logical flag to determine if the levels should be regarded as ordered (in the order given).
`nmax`	an upper bound on the number of levels.
`...`	(in `ordered(.)`): any of the above, apart from `ordered` itself.

Value

An object of class "factor" or "ordered".

See Also

factor

Examples

```
x <- as.integer64(c(132724613L, -2143220989L, -1L, NA, 1L))
factor(x)
ordered(x)
```

`format.integer64` *Unary operators and functions for integer64 vectors*

Description

Unary operators and functions for integer64 vectors.

Usage

```
## S3 method for class 'integer64'
format(x, justify = "right", ...)

## S3 method for class 'integer64'
sign(x)

## S3 method for class 'integer64'
abs(x)

## S3 method for class 'integer64'
sqrt(x)

## S3 method for class 'integer64'
log(x, base = NULL)

## S3 method for class 'integer64'
log10(x)

## S3 method for class 'integer64'
log2(x)

## S3 method for class 'integer64'
trunc(x, ...)

## S3 method for class 'integer64'
```

```
floor(x)

## S3 method for class 'integer64'
ceiling(x)

## S3 method for class 'integer64'
signif(x, digits = 6L)

## S3 method for class 'integer64'
scale(x, center = TRUE, scale = TRUE)

## S3 method for class 'integer64'
round(x, digits = 0L)

## S3 method for class 'integer64'
is.na(x)

## S3 method for class 'integer64'
is.finite(x)

## S3 method for class 'integer64'
is.infinite(x)

## S3 method for class 'integer64'
is.nan(x)
```

Arguments

x	an atomic vector of class 'integer64'
justify	should it be right-justified (the default), left-justified, centred or left alone.
...	further arguments to the NextMethod()
base	an atomic scalar (we save 50% log-calls by not allowing a vector base)
digits	integer indicating the number of decimal places (round) or significant digits (signif) to be used. Negative values are allowed (see round())
center	see scale()
scale	see scale()

Value

format() returns a character vector

is.na() and ! return a logical vector

sqrt(), log(), log2() and log10() return a double vector

sign(), abs(), floor(), ceiling(), trunc() and round() return a vector of class 'integer64'

signif() is not implemented

See Also

ops64 integer64()

Examples

```
sqrt(as.integer64(1:12))
```

---

hashcache *Big caching of hashing, sorting, ordering*

Description

Functions to create cache that accelerates many operations

Usage

```
hashcache(x, nunique = NULL, ...)

sortcache(x, has.na = NULL, na.last = FALSE)

sortordercache(x, has.na = NULL, stable = NULL, na.last = FALSE)

ordercache(x, has.na = NULL, stable = NULL, optimize = "time", na.last = FALSE)
```

Arguments

x	an atomic vector (note that currently only integer64 is supported)
nunique	giving *correct* number of unique elements can help reducing the size of the hashmap
...	passed to hashmap()
has.na	boolean scalar defining whether the input vector might contain NAs. If we know we don't have NAs, this may speed-up. *Note* that you risk a crash if there are unexpected NAs with has.na=FALSE.
na.last	boolean scalar defining whether NA should be last.
stable	boolean scalar defining whether stable sorting is needed. Allowing non-stable may speed-up.
optimize	by default ramsort optimizes for 'time' which requires more RAM, set to 'memory' to minimize RAM requirements and sacrifice speed.

Details

The result of relative expensive operations hashmap(), bit::ramsort(), bit::ramsortorder(), and bit::ramorder() can be stored in a cache in order to avoid multiple excutions. Unless in very specific situations, the recommended method is hashsortorder only.

Value

x with a cache() that contains the result of the expensive operations, possible together with small derived information (such as nunique.integer64()) and previously cached results.

Note

Note that we consider storing the big results from sorting and/or ordering as a relevant side-effect, and therefore storing them in the cache should require a conscious decision of the user.

See Also

cache() for caching functions and nunique.integer64() for methods benefiting from small caches

Examples

```
x = as.integer64(sample(c(rep(NA, 9), 1:9), 32, TRUE))
sortordercache(x)
```

---

hashmap *Hashing for 64bit integers*

Description

This is an explicit implementation of hash functionality that underlies matching and other functions in R. Explicit means that you can create, store and use hash functionality directly. One advantage is that you can re-use hashmaps, which avoid re-building hashmaps again and again.

Usage

```
hashfun(x, ...)

## S3 method for class 'integer64'
hashfun(x, minfac = 1.41, hashbits = NULL, ...)

hashmap(x, ...)

## S3 method for class 'integer64'
hashmap(x, nunique = NULL, minfac = 1.41, hashbits = NULL, cache = NULL, ...)

hashpos(cache, ...)

## S3 method for class 'cache_integer64'
hashpos(cache, x, nomatch = NA_integer_, ...)

hashrev(cache, ...)
```

```
## S3 method for class 'cache_integer64'
hashrev(cache, x, nomatch = NA_integer_, ...)

hashfin(cache, ...)

## S3 method for class 'cache_integer64'
hashfin(cache, x, ...)

hashrin(cache, ...)

## S3 method for class 'cache_integer64'
hashrin(cache, x, ...)

hashdup(cache, ...)

## S3 method for class 'cache_integer64'
hashdup(cache, ...)

hashuni(cache, ...)

## S3 method for class 'cache_integer64'
hashuni(cache, keep.order = FALSE, ...)

hashupo(cache, ...)

## S3 method for class 'cache_integer64'
hashupo(cache, keep.order = FALSE, ...)

hashtab(cache, ...)

## S3 method for class 'cache_integer64'
hashtab(cache, ...)

hashmaptab(x, ...)

## S3 method for class 'integer64'
hashmaptab(x, nunique = NULL, minfac = 1.5, hashbits = NULL, ...)

hashmapuni(x, ...)

## S3 method for class 'integer64'
hashmapuni(x, nunique = NULL, minfac = 1.5, hashbits = NULL, ...)

hashmapupo(x, ...)

## S3 method for class 'integer64'
hashmapupo(x, nunique = NULL, minfac = 1.5, hashbits = NULL, ...)
```

Arguments

x	an integer64 vector
...	further arguments, passed from generics, ignored in methods
minfac	minimum factor by which the hasmap has more elements compared to the data x, ignored if hashbits is given directly
hashbits	length of hashmap is 2^hashbits
nunique	giving *correct* number of unique elements can help reducing the size of the hashmap
cache	an optional cache() object into which to put the hashmap (by default a new cache is created
nomatch	the value to be returned if an element is not found in the hashmap
keep.order	determines order of results and speed: FALSE (the default) is faster and returns in the (pseudo)random order of the hash function, TRUE returns in the order of first appearance in the original data, but this requires extra work

Details

function	see also	description
hashfun	digest	export of the hash function used in hashmap
hashmap	match()	return hashmap
hashpos	match()	return positions of x in hashmap
hashrev	match()	return positions of hashmap in x
hashfin	%in%.integer64	return logical whether x is in hashmap
hashrin	%in%.integer64	return logical whether hashmap is in x
hashdup	duplicated()	return logical whether hashdat is duplicated using hashmap
hashuni	unique()	return unique values of hashmap
hashmapuni	unique()	return unique values of x
hashupo	unique()	return positions of unique values in hashdat
hashmapupo	unique()	return positions of unique values in x
hashtab	table()	tabulate values of hashdat using hashmap in keep.order=FALSE
hashmaptab	table()	tabulate values of x building hasmap on the fly in keep.order=FALSE

Value

See Details

See Also

match(), runif64()

Examples

```
x <- as.integer64(sample(c(NA, 0:9)))
y <- as.integer64(sample(c(NA, 1:9), 10, TRUE))
hashfun(y)
hx <- hashmap(x)
hy <- hashmap(y)
ls(hy)
hashpos(hy, x)
hashrev(hx, y)
hashfin(hy, x)
hashrin(hx, y)
hashdup(hy)
hashuni(hy)
hashuni(hy, keep.order=TRUE)
hashmapuni(y)
hashupo(hy)
hashupo(hy, keep.order=TRUE)
hashmapupo(y)
hashtab(hy)
hashmaptab(y)

    ## Not run:
    message("explore speed given size of the hasmap in 2^hashbits and size of the data")
    message("more hashbits means more random access and less collisions")
    message("i.e. more data means less random access and more collisions")
    bits <- 24
    b <- seq(-1, 0, 0.1)
    tim <- matrix(NA, length(b), 2, dimnames=list(b, c("bits", "bits+1")))
    for (i in 1:length(b)) {
      n <- as.integer(2^(bits+b[i]))
      x <- as.integer64(sample(n))
      tim[i, 1] <- repeat.time(hashmap(x, hashbits=bits))[3]
      tim[i, 2] <- repeat.time(hashmap(x, hashbits=bits+1))[3]
      print(tim)
      matplot(b, tim)
    }
    message("we conclude that n*sqrt(2) is enough to avoid collisions")

## End(Not run)
```

`identical.integer64` *Identity function for class 'integer64'*

Description

This will discover any deviation between objects containing integer64 vectors.

Usage

```
identical.integer64(
  x,
  y,
  num.eq = FALSE,
  single.NA = FALSE,
  attrib.as.set = TRUE,
  ignore.bytecode = TRUE,
  ignore.environment = FALSE,
  ignore.srcref = TRUE,
  ...
)
```

Arguments

x, y	Atomic vector of class 'integer64'
num.eq, single.NA, attrib.as.set, ignore.bytecode, ignore.environment, ignore.srcref	See identical().
...	Passed on to identical(). Only extptr.as.ref= is available as of R 4.4.1, and then only for versions of R >= 4.2.0.

Details

This is simply a wrapper to identical() with default arguments num.eq = FALSE, single.NA = FALSE.

Value

A single logical value, TRUE or FALSE, never NA and never anything other than a single value.

See Also

==.integer64 identical() integer64()

Examples

```
i64 <- as.double(NA); class(i64) <- "integer64"
identical(i64-1, i64+1)
identical.integer64(i64-1, i64+1)
```

`is.sorted.integer64` *Small cache access methods*

Description

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

Usage

```
## S3 method for class 'integer64'
na.count(x, ...)

## S3 method for class 'integer64'
nvalid(x, ...)

## S3 method for class 'integer64'
is.sorted(x, ...)

## S3 method for class 'integer64'
nunique(x, ...)

## S3 method for class 'integer64'
nties(x, ...)
```

Arguments

`x`	some object
`...`	ignored

Details

All these functions benefit from a `sortcache()`, `ordercache()` or `sortordercache()`. `na.count()`, `nvalid()` and `nunique()` also benefit from a `hashcache()`.

Value

`is.sorted` returns a logical scalar, the other methods return an integer scalar.

Functions

- `na.count(integer64)`: returns the number of `NA`s
- `nvalid(integer64)`: returns the number of valid data points, usually `length()` minus `na.count`.
- `is.sorted(integer64)`: checks for sortedness of `x` (NAs sorted first)
- `nunique(integer64)`: returns the number of unique values
- `nties(integer64)`: returns the number of tied values.

Note

If a cache() exists but the desired value is not cached, then these functions will store their result in the cache. We do not consider this a relevant side-effect, since these small cache results do not have a relevant memory footprint.

See Also

cache() for caching functions and sortordercache() for functions creating big caches

Examples

```
x = as.integer64(sample(c(rep(NA, 9), 1:9), 32, TRUE))
length(x)
bit::na.count(x)
bit::nvalid(x)
bit::nunique(x)
bit::nties(x)
table(x)
x
```

---

keypos *Extract Positions in redundant dimension table*

Description

keypos returns the positions of the (fact table) elements that participate in their sorted unique subset (dimension table)

Usage

```
keypos(x, ...)

## S3 method for class 'integer64'
keypos(x, method = NULL, ...)
```

Arguments

x	a vector or a data frame or an array or NULL.
...	ignored
method	NULL for automatic method selection or a suitable low-level method, see details

Details

NAs are sorted first in the dimension table, see `ramorder.integer64()`.

This function automatically chooses from several low-level functions considering the size of `x` and the availability of a cache.

Suitable methods are

- `sortorderkey` (fast ordering)
- `orderkey` (memory saving ordering).

Value

an integer vector of the same length as `x` containing positions relative to `sort(unique(x), na.last=FALSE)`

See Also

`unique.integer64()` for the unique subset and `match.integer64()` for finding positions in a different vector.

Examples

```
x <- as.integer64(sample(c(rep(NA, 9), 1:9), 32, TRUE))
keypos(x)
```

`match.integer64` *64-bit integer matching*

Description

`match` returns a vector of the positions of (first) matches of its first argument in its second. `%in%` is a more intuitive interface as a binary operator, which returns a logical vector indicating if there is a match or not for its left operand.

Usage

```
## S3 method for class 'integer64'
match(x, table, nomatch = NA_integer_, nunique = NULL, method = NULL, ...)

## S3 method for class 'integer64'
x %in% table, ...
```

Arguments

x	integer64 vector: the values to be matched, optionally carrying a cache created with hashcache()
table	integer64 vector: the values to be matched against, optionally carrying a cache created with hashcache() or sortordercache()
nomatch	the value to be returned in the case when no match is found. Note that it is coerced to integer.
nunique	NULL or the number of unique values of table (including NA). Providing nunique can speed-up matching when table has no cache. Note that a wrong nunique can cause undefined behaviour up to a crash.
method	NULL for automatic method selection or a suitable low-level method, see details
...	ignored

Details

These functions automatically choose from several low-level functions considering the size of x and table and the availability of caches.

Suitable methods for %in%.integer64 are

- hashpos (hash table lookup)
- hashrev (reverse lookup)
- sortorderpos (fast ordering)
- orderpos (memory saving ordering).

Suitable methods for match.integer64 are

- hashfin (hash table lookup)
- hashrin (reverse lookup)
- sortfin (fast sorting)
- orderfin (memory saving ordering).

Value

A vector of the same length as x.

match: An integer vector giving the position in table of the first match if there is a match, otherwise nomatch.

If x[i] is found to equal table[j] then the value returned in the i-th position of the return value is j, for the smallest possible j. If no match is found, the value is nomatch.

%in%: A logical vector, indicating if a match was located for each element of x: thus the values are TRUE or FALSE and never NA.

See Also

match()

Examples

```
x <- as.integer64(c(NA, 0:9), 32)
table <- as.integer64(c(1:9, NA))
match(x, table)
x %in% table

x <- as.integer64(sample(c(rep(NA, 9), 0:9), 32, TRUE))
table <- as.integer64(sample(c(rep(NA, 9), 1:9), 32, TRUE))

## Not run:
    library(bit)
    message("check when reverse hash-lookup beats standard hash-lookup")
    e <- 4:24
    timx <- timy <- matrix(NA, length(e), length(e), dimnames=list(e, e))
    for (iy in seq_along(e))
    for (ix in 1:iy) {
        nx <- 2^e[ix]
        ny <- 2^e[iy]
        x <- as.integer64(sample(ny, nx, FALSE))
        y <- as.integer64(sample(ny, ny, FALSE))
        #hashfun(x, bits=as.integer(5))
        timx[ix, iy] <- repeat.time({
        hx <- hashmap(x)
        py <- hashrev(hx, y)
        })[3]
        timy[ix, iy] <- repeat.time({
        hy <- hashmap(y)
        px <- hashpos(hy, x)
        })[3]
        #identical(px, py)
        print(round(timx[1:iy, 1:iy]/timy[1:iy, 1:iy], 2), na.print="")
    }

    message("explore best low-level method given size of x and table")
    B1 <- 1:27
    B2 <- 1:27
    tim <- array(NA, dim=c(length(B1), length(B2), 5)
 , dimnames=list(B1, B2, c("hashpos", "hashrev", "sortpos1", "sortpos2", "sortpos3")))
    for (i1 in B1)
    for (i2 in B2)
    {
      b1 <- B1[i1]
      b2 <- B1[i2]
      n1 <- 2^b1
      n2 <- 2^b2
      x1 <- as.integer64(c(sample(n2, n1-1, TRUE), NA))
      x2 <- as.integer64(c(sample(n2, n2-1, TRUE), NA))
      tim[i1, i2, 1] <- repeat.time({h <- hashmap(x2);hashpos(h, x1);rm(h)})[3]
      tim[i1, i2, 2] <- repeat.time({h <- hashmap(x1);hashrev(h, x2);rm(h)})[3]
      s <- clone(x2); o <- seq_along(s); ramsortorder(s, o)
      tim[i1, i2, 3] <- repeat.time(sortorderpos(s, o, x1, method=1))[3]
      tim[i1, i2, 4] <- repeat.time(sortorderpos(s, o, x1, method=2))[3]
```

```
    tim[i1, i2, 5] <- repeat.time(sortorderpos(s, o, x1, method=3))[3]
    rm(s, o)
    print(apply(tim, 1:2, function(ti)if(any(is.na(ti)))NA else which.min(ti)))
  }

## End(Not run)
```

---

matrix64 *Working with integer64 arrays and matrices*

Description

These functions and methods facilitate working with integer64 objects stored in matrices. As ever, the primary motivation for having tailor-made functions here is that R's methods often receive input from bit64 and treat the vectors as doubles, leading to unexpected and/or incorrect results.

Usage

```
## S3 method for class 'integer64'
matrix(data = NA_integer64_, ...)

## S3 method for class 'integer64'
array(data = NA_integer64_, ...)

## S3 method for class 'integer64'
colSums(x, na.rm = FALSE, dims = 1L)

## S3 method for class 'integer64'
rowSums(x, na.rm = FALSE, dims = 1L)

## S3 method for class 'integer64'
aperm(a, perm, ...)

matrix(data = NA, nrow = 1L, ncol = 1L, byrow = FALSE, dimnames = NULL)

array(data = NA, dim = length(data), dimnames = NULL)

colSums(x, na.rm = FALSE, dims = 1L)

## Default S3 method:
colSums(x, na.rm = FALSE, dims = 1L)

rowSums(x, na.rm = FALSE, dims = 1L)

## Default S3 method:
rowSums(x, na.rm = FALSE, dims = 1L)
```

Arguments

data, nrow, ncol, byrow, dimnames, dim	Arguments for matrix() and array().
...	Passed on to subsequent methods.
x	An array of integer64 numbers.
na.rm, dims	Same interpretation as in colSums().
a, perm	Passed on to aperm().

Details

As of now, the colSums() and rowSums() methods are implemented as wrappers around equivalent apply() approaches, because re-using the default routine (and then applying integer64 to the result) does not work for objects with missing elements. Ideally this would eventually get its own dedicated C routine mimicking that of colSums() for integers; feature requests and PRs welcome.

aperm() is required for apply() to work, in general, otherwise FUN gets applied to a class-stripped version of the input.

Examples

```
A = matrix(as.integer64(1:6), 3)

colSums(A)
rowSums(A)
aperm(A, 2:1)
```

---

ops64 *Binary operators for integer64 vectors*

Description

Binary operators for integer64 vectors.

Usage

```
binattr(e1, e2)

## S3 method for class 'integer64'
e1 + e2

## S3 method for class 'integer64'
e1 - e2

## S3 method for class 'integer64'
e1 %/% e2

## S3 method for class 'integer64'
```

```
e1 %% e2

## S3 method for class 'integer64'
e1 * e2

## S3 method for class 'integer64'
e1 ^ e2

## S3 method for class 'integer64'
e1 / e2

## S3 method for class 'integer64'
e1 == e2

## S3 method for class 'integer64'
e1 != e2

## S3 method for class 'integer64'
e1 < e2

## S3 method for class 'integer64'
e1 <= e2

## S3 method for class 'integer64'
e1 > e2

## S3 method for class 'integer64'
e1 >= e2

## S3 method for class 'integer64'
e1 & e2

## S3 method for class 'integer64'
e1 | e2

## S3 method for class 'integer64'
!x
```

Arguments

e1, e2, x	numeric or complex vectors or objects which can be coerced to such, or other objects for which methods have been written for - especially 'integer64' vectors.

Value

&, |, !, !=, ==, <, <=, >, >= return a logical vector

/ returns a double vector

+, -, *, %/%, %%, ^ return a vector of class 'integer64' or different class depending on the operands

See Also

integer64()

Examples

```
as.integer64(1:12) - 1
options(integer64_semantics="new")
d <- 2.5
i <- as.integer64(5)
d/i  # new 0.5
d*i  # new 13
i*d  # new 13
options(integer64_semantics="old")
d/i  # old: 0.4
d*i  # old: 10
i*d  # old: 13
```

---

optimizer64.data

Results of performance measurement on a Core i7 Lenovo T410 8 GB RAM under Windows 7 64bit

---

Description

These are the results of calling optimizer64()

Usage

```
data(optimizer64.data)
```

Format

The format is:

```
List of 16
 $ : num [1:9, 1:3] 0 0 1.63 0.00114 2.44 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : chr [1:9] "match" "match.64" "hashpos" "hashrev" ...
  .. ..$ : chr [1:3] "prep" "both" "use"
 $ : num [1:10, 1:3] 0 0 0 1.62 0.00114 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : chr [1:10] "%in%" "match.64" "%in%.64" "hashfin" ...
  .. ..$ : chr [1:3] "prep" "both" "use"
 $ : num [1:10, 1:3] 0 0 0.00105 0.00313 0.00313 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : chr [1:10] "duplicated" "duplicated.64" "hashdup" "sortorderdup1" ...
  .. ..$ : chr [1:3] "prep" "both" "use"
 $ : num [1:15, 1:3] 0 0 0 0.00104 0.00104 ...
  ..- attr(*, "dimnames")=List of 2
```

```
  .. ..$ : chr [1:15] "unique" "unique.64" "hashmapuni" "hashuni" ...
  .. ..$ : chr [1:3] "prep" "both" "use"
 $ : num [1:14, 1:3] 0 0 0 0.000992 0.000992 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : chr [1:14] "unique" "unipos.64" "hashmapupo" "hashupo" ...
  .. ..$ : chr [1:3] "prep" "both" "use"
 $ : num [1:13, 1:3] 0 0 0 0 0.000419 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : chr [1:13] "tabulate" "table" "table.64" "hashmaptab" ...
  .. ..$ : chr [1:3] "prep" "both" "use"
 $ : num [1:7, 1:3] 0 0 0 0.00236 0.00714 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : chr [1:7] "rank" "rank.keep" "rank.64" "sortorderrnk" ...
  .. ..$ : chr [1:3] "prep" "both" "use"
 $ : num [1:6, 1:3] 0 0 0.00189 0.00714 0 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : chr [1:6] "quantile" "quantile.64" "sortqtl" "orderqtl" ...
  .. ..$ : chr [1:3] "prep" "both" "use"
 $ : num [1:9, 1:3] 0 0 0.00105 1.17 0 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : chr [1:9] "match" "match.64" "hashpos" "hashrev" ...
  .. ..$ : chr [1:3] "prep" "both" "use"
 $ : num [1:10, 1:3] 0 0 0 0.00104 1.18 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : chr [1:10] "%in%" "match.64" "%in%.64" "hashfin" ...
  .. ..$ : chr [1:3] "prep" "both" "use"
 $ : num [1:10, 1:3] 0 0 1.64 2.48 2.48 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : chr [1:10] "duplicated" "duplicated.64" "hashdup" "sortorderdup1" ...
  .. ..$ : chr [1:3] "prep" "both" "use"
 $ : num [1:15, 1:3] 0 0 0 1.64 1.64 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : chr [1:15] "unique" "unique.64" "hashmapuni" "hashuni" ...
  .. ..$ : chr [1:3] "prep" "both" "use"
 $ : num [1:14, 1:3] 0 0 0 1.62 1.62 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : chr [1:14] "unique" "unipos.64" "hashmapupo" "hashupo" ...
  .. ..$ : chr [1:3] "prep" "both" "use"
 $ : num [1:13, 1:3] 0 0 0 0 0.32 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : chr [1:13] "tabulate" "table" "table.64" "hashmaptab" ...
  .. ..$ : chr [1:3] "prep" "both" "use"
 $ : num [1:7, 1:3] 0 0 0 2.96 10.69 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : chr [1:7] "rank" "rank.keep" "rank.64" "sortorderrnk" ...
  .. ..$ : chr [1:3] "prep" "both" "use"
 $ : num [1:6, 1:3] 0 0 1.62 10.61 0 ...
  ..- attr(*, "dimnames")=List of 2
```

```
  .. ..$ : chr [1:6] "quantile" "quantile.64" "sortqtl" "orderqtl" ...
  .. ..$ : chr [1:3] "prep" "both" "use"
 - attr(*, "dim")= int [1:2] 8 2
 - attr(*, "dimnames")=List of 2
  ..$ : chr [1:8] "match" "%in%" "duplicated" "unique" ...
  ..$ : chr [1:2] "65536" "33554432"
```

Examples

```
data(optimizer64.data)
print(optimizer64.data)
oldpar <- par(no.readonly = TRUE)
par(mfrow=c(2,1))
par(cex=0.7)
for (i in 1:nrow(optimizer64.data)) {
 for (j in 1:2) {
   tim <- optimizer64.data[[i,j]]
  barplot(t(tim))
  if (rownames(optimizer64.data)[i]=="match")
  title(paste("match", colnames(optimizer64.data)[j], "in", colnames(optimizer64.data)[3-j]))
  else if (rownames(optimizer64.data)[i]=="%in%")
   title(paste(colnames(optimizer64.data)[j], "%in%", colnames(optimizer64.data)[3-j]))
  else
   title(paste(rownames(optimizer64.data)[i], colnames(optimizer64.data)[j]))
 }
}
par(mfrow=c(1,1))
```

`prank` *(P)ercent (Rank)s*

Description

Function `prank.integer64` projects the values `[min..max]` via ranks `[1..n]` to `[0..1]`. `qtile.integer64()` is the inverse function of 'prank.integer64' and projects `[0..1]` to `[min..max]`.

Usage

```
prank(x, ...)

## S3 method for class 'integer64'
prank(x, method = NULL, ...)
```

Arguments

`x`	a integer64 vector
`...`	ignored
`method`	NULL for automatic method selection or a suitable low-level method, see details

Details

Function `prank.integer64` is based on `rank.integer64()`.

Value

`prank` returns a numeric vector of the same length as `x`.

See Also

`rank.integer64()` for simple ranks and `qtile()` for the inverse function quantiles.

Examples

```
x <- as.integer64(sample(c(rep(NA, 9), 1:9), 32, TRUE))
prank(x)

x <- x[!is.na(x)]
stopifnot(identical(x,  unname(qtile(x, probs=prank(x)))))
```

`qtile` *(Q)uan(Tile)s*

Description

Function `prank.integer64()` projects the values `[min..max]` via ranks `[1..n]` to `[0..1]`.

Usage

```
qtile(x, probs = seq(0, 1, 0.25), ...)

## S3 method for class 'integer64'
qtile(x, probs = seq(0, 1, 0.25), names = TRUE, method = NULL, ...)

## S3 method for class 'integer64'
quantile(
  x,
  probs = seq(0, 1, 0.25),
  na.rm = FALSE,
  names = TRUE,
  type = 0L,
  ...
)

## S3 method for class 'integer64'
median(x, na.rm = FALSE, ...)

## S3 method for class 'integer64'
```

```
mean(x, na.rm = FALSE, ...)

## S3 method for class 'integer64'
summary(object, ...)
```

Arguments

x	a integer64 vector
probs	numeric vector of probabilities with values in [0, 1] - possibly containing NAs
...	ignored
names	logical; if TRUE, the result has a names attribute. Set to FALSE for speedup with many probs.
method	NULL for automatic method selection or a suitable low-level method, see details
na.rm	logical; if TRUE, any NA and NaN's are removed from x before the quantiles are computed.
type	an integer selecting the quantile algorithm, currently only 0 is supported, see details
object	a integer64 vector

Details

qtile.integer64 is the inverse function of 'prank.integer64' and projects [0..1] to [min..max].

Functions quantile.integer64 with type=0 and median.integer64 are convenience wrappers to qtile.

Function qtile behaves very similar to quantile.default with type=1 in that it only returns existing values, it is mostly symmetric but it is using 'round' rather than 'floor'.

Note that this implies that median.integer64 does not interpolate for even number of values (interpolation would create values that could not be represented as 64-bit integers).

This function automatically chooses from several low-level functions considering the size of x and the availability of a cache.

Suitable methods are

- sortqtl (fast sorting)
- orderqtl (memory saving ordering).

Value

prank returns a numeric vector of the same length as x.

qtile returns a vector with elements from x at the relative positions specified by probs.

See Also

rank.integer64() for simple ranks and quantile() for quantiles.

Examples

```
x <- as.integer64(sample(c(rep(NA, 9), 1:9), 32, TRUE))
qtile(x, probs=seq(0, 1, 0.25))
quantile(x, probs=seq(0, 1, 0.25), na.rm=TRUE)
median(x, na.rm=TRUE)
summary(x)

x <- x[!is.na(x)]
stopifnot(identical(x,  unname(qtile(x, probs=prank(x)))))
```

`ramsort.integer64` *Low-level intger64 methods for in-RAM sorting and ordering*

Description

Fast low-level methods for sorting and ordering. The `..sortorder` methods do sorting and ordering at once, which requires more RAM than ordering but is (almost) as fast as as sorting.

Usage

```
## S3 method for class 'integer64'
shellsort(x, has.na = TRUE, na.last = FALSE, decreasing = FALSE, ...)

## S3 method for class 'integer64'
shellsortorder(x, i, has.na = TRUE, na.last = FALSE, decreasing = FALSE, ...)

## S3 method for class 'integer64'
shellorder(x, i, has.na = TRUE, na.last = FALSE, decreasing = FALSE, ...)

## S3 method for class 'integer64'
mergesort(x, has.na = TRUE, na.last = FALSE, decreasing = FALSE, ...)

## S3 method for class 'integer64'
mergeorder(x, i, has.na = TRUE, na.last = FALSE, decreasing = FALSE, ...)

## S3 method for class 'integer64'
mergesortorder(x, i, has.na = TRUE, na.last = FALSE, decreasing = FALSE, ...)

## S3 method for class 'integer64'
quicksort(
  x,
  has.na = TRUE,
  na.last = FALSE,
  decreasing = FALSE,
  restlevel = floor(1.5 * log2(length(x))),
  ...
)
```

```
## S3 method for class 'integer64'
quicksortorder(
  x,
  i,
  has.na = TRUE,
  na.last = FALSE,
  decreasing = FALSE,
  restlevel = floor(1.5 * log2(length(x))),
  ...
)

## S3 method for class 'integer64'
quickorder(
  x,
  i,
  has.na = TRUE,
  na.last = FALSE,
  decreasing = FALSE,
  restlevel = floor(1.5 * log2(length(x))),
  ...
)

## S3 method for class 'integer64'
radixsort(
  x,
  has.na = TRUE,
  na.last = FALSE,
  decreasing = FALSE,
  radixbits = 8L,
  ...
)

## S3 method for class 'integer64'
radixsortorder(
  x,
  i,
  has.na = TRUE,
  na.last = FALSE,
  decreasing = FALSE,
  radixbits = 8L,
  ...
)

## S3 method for class 'integer64'
radixorder(
  x,
  i,
```

```
  has.na = TRUE,
  na.last = FALSE,
  decreasing = FALSE,
  radixbits = 8L,
  ...
)

## S3 method for class 'integer64'
ramsort(
  x,
  has.na = TRUE,
  na.last = FALSE,
  decreasing = FALSE,
  stable = TRUE,
  optimize = c("time", "memory"),
  VERBOSE = FALSE,
  ...
)

## S3 method for class 'integer64'
ramsortorder(
  x,
  i,
  has.na = TRUE,
  na.last = FALSE,
  decreasing = FALSE,
  stable = TRUE,
  optimize = c("time", "memory"),
  VERBOSE = FALSE,
  ...
)

## S3 method for class 'integer64'
ramorder(
  x,
  i,
  has.na = TRUE,
  na.last = FALSE,
  decreasing = FALSE,
  stable = TRUE,
  optimize = c("time", "memory"),
  VERBOSE = FALSE,
  ...
)
```

Arguments

x	a vector to be sorted by `ramsort.integer64()` and `ramsortorder.integer64()`, i.e. the output of `sort.integer64()`

has.na	boolean scalar defining whether the input vector might contain NAs. If we know we don't have NAs, this may speed-up. *Note* that you risk a crash if there are unexpected NAs with has.na=FALSE
na.last	boolean scalar telling ramsort whether to sort NAs last or first. *Note* that 'boolean' means that there is no third option NA as in sort()
decreasing	boolean scalar telling ramsort whether to sort increasing or decreasing
...	further arguments, passed from generics, ignored in methods
i	integer positions to be modified by ramorder.integer64() and ramsortorder.integer64(), default is 1:n, in this case the output is similar to order.integer64()
restlevel	number of remaining recursionlevels before quicksort switches from recursing to shellsort
radixbits	size of radix in bits
stable	boolean scalar defining whether stable sorting is needed. Allowing non-stable may speed-up.
optimize	by default ramsort optimizes for 'time' which requires more RAM, set to 'memory' to minimize RAM requirements and sacrifice speed
VERBOSE	cat some info about chosen method

Details

See bit::ramsort()

Value

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

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 sort-methods change x, the order-methods change i, and the sortoder-methods change both x and i

See Also

bit::ramsort() for the generic, ramsort.default for the methods provided by package ff, sort.integer64() for the sort interface and sortcache() for caching the work of sorting

Examples

```
x <- as.integer64(sample(c(rep(NA, 9), 1:9), 32, TRUE))
x
message("ramsort example")
s <- bit::clone(x)
bit::ramsort(s)
message("s has been changed in-place - whether or not ramsort uses an in-place algorithm")
s
message("ramorder example")
s <- bit::clone(x)
```

```
o <- seq_along(s)
bit::ramorder(s, o)
message("o has been changed in-place - s remains unchanged")
s
o
s[o]
message("ramsortorder example")
o <- seq_along(s)
bit::ramsortorder(s, o)
message("s and o have both been changed in-place - this is much faster")
s
o
```

`rank.integer64` *Sample Ranks from integer64*

Description

Returns the sample ranks of the values in a vector. Ties (i.e., equal values) are averaged and missing values propagated.

Usage

```
## S3 method for class 'integer64'
rank(x, method = NULL, ...)
```

Arguments

`x`	a integer64 vector
`method`	NULL for automatic method selection or a suitable low-level method, see details
`...`	ignored

Details

This function automatically chooses from several low-level functions considering the size of `x` and the availability of a cache. Suitable methods are

- `sortorderrnk()` (fast ordering)
- `orderrnk()` (memory saving ordering).

Value

A numeric vector of the same length as `x`.

See Also

`order.integer64()`, `rank()` and `prank()` for percent rank.

Examples

```
x <- as.integer64(sample(c(rep(NA, 9), 1:9), 32, TRUE))
rank.integer64(x)

stopifnot(identical(rank.integer64(x),  rank(as.integer(x)
, na.last="keep", ties.method = "average")))
```

`rep.integer64` *Replicate elements of integer64 vectors*

Description

Replicate elements of integer64 vectors

Arguments

x	a vector of 'integer64' to be replicated
...	further arguments passed to `NextMethod()`

Value

`rep()` returns a integer64 vector

See Also

`c.integer64()` `rep.integer64()` `as.data.frame.integer64()` `integer64()`

Examples

```
rep(as.integer64(1:2), 6)
rep(as.integer64(1:2), c(6, 6))
rep(as.integer64(1:2), length.out=6)
```

`runif64` *integer64: random numbers*

Description

Create uniform random 64-bit integers within defined range

Usage

```
runif64(
  n,
  min = lim.integer64()[1L],
  max = lim.integer64()[2L],
  replace = TRUE
)
```

Arguments

n	length of return vector
min	lower inclusive bound for random numbers
max	upper inclusive bound for random numbers
replace	set to FALSE for sampleing from a finite pool, see sample()

Details

For each random integer we call R's internal C interface unif_rand() twice. Each call is mapped to 2^32 unsigned integers. The two 32-bit patterns are concatenated to form the new integer64. This process is repeated until the result is not a NA_INTEGER64_.

Value

a integer64 vector

See Also

runif(), hashfun()

Examples

```
runif64(12)
runif64(12, -16, 16)
runif64(12, 0, as.integer64(2^60)-1)  # not 2^60-1 !
var(runif(1e4))
var(as.double(runif64(1e4, 0, 2^40))/2^40)  # ~ = 1/12 = .08333

table(sample(16, replace=FALSE))
table(runif64(16, 1, 16, replace=FALSE))
table(sample(16, replace=TRUE))
table(runif64(16, 1, 16, replace=TRUE))
```

seq.integer64 *Generating sequence of integer64 values*

Description

Generating sequence of integer64 values

Usage

```
## S3 method for class 'integer64'
seq(
  from = NULL,
  to = NULL,
  by = NULL,
  length.out = NULL,
  along.with = NULL,
  ...
)
```

Arguments

from	integer64 scalar (in order to dispatch the integer64 method of seq())
to	scalar
by	scalar
length.out	scalar
along.with	scalar
...	ignored

Details

seq.integer64 coerces its arguments from, to, and by to integer64. Consistency with seq() is typically maintained, though results may differ when mixing integer64 and double inputs, for the same reason that any arithmetic with these mixed types can be ambiguous. Whereas seq(1L, 10L, length.out=8L) can back up to double storage to give an exact result, this not possible for generic inputs seq(i64, dbl, length.out=n).

Value

An integer64 vector with the generated sequence

See Also

c.integer64() rep.integer64() as.data.frame.integer64() integer64()

Examples

```
seq(as.integer64(1), 12, 2)
seq(as.integer64(1), by=2, length.out=6)

# truncation rules
seq(as.integer64(1), 10, by=1.5)
seq(as.integer64(1), 10, length.out=5)
```

`sort.integer64` *High-level intger64 methods for sorting and ordering*

Description

Fast high-level methods for sorting and ordering. These are wrappers to `ramsort.integer64()` and friends and do not modify their arguments.

Usage

```
## S3 method for class 'integer64'
sort(
  x,
  decreasing = FALSE,
  has.na = TRUE,
  na.last = TRUE,
  stable = TRUE,
  optimize = c("time", "memory"),
  VERBOSE = FALSE,
  ...
)

## S3 method for class 'integer64'
order(
  ...,
  na.last = TRUE,
  decreasing = FALSE,
  has.na = TRUE,
  stable = TRUE,
  optimize = c("time", "memory"),
  VERBOSE = FALSE
)
```

Arguments

`x`	a vector to be sorted by `ramsort.integer64()` and `ramsortorder.integer64()`, i.e. the output of `sort.integer64()`
`decreasing`	boolean scalar telling ramsort whether to sort increasing or decreasing

has.na	boolean scalar defining whether the input vector might contain NAs. If we know we don't have NAs, this may speed-up. *Note* that you risk a crash if there are unexpected NAs with has.na=FALSE
na.last	boolean scalar telling ramsort whether to sort NAs last or first. *Note* that 'boolean' means that there is no third option NA as in sort()
stable	boolean scalar defining whether stable sorting is needed. Allowing non-stable may speed-up.
optimize	by default ramsort optimizes for 'time' which requires more RAM, set to 'memory' to minimize RAM requirements and sacrifice speed
VERBOSE	cat some info about chosen method
...	further arguments, passed from generics, ignored in methods

Details

see sort() and order()

Value

sort returns the sorted vector and vector returns the order positions.

See Also

sort(), sortcache()

Examples

```
x <- as.integer64(sample(c(rep(NA, 9), 1:9), 32, TRUE))
x
sort(x)
message(
  "the following has default optimize='time' which is faster ",
  "but requires more RAM, this calls 'ramorder'"
)
order(x)
message("slower with less RAM, this calls 'ramsortorder'")
order(x, optimize="memory")
```

---

sortnut *Searching and other uses of sorting for 64bit integers*

Description

This is roughly an implementation of hash functionality but based on sorting instead on a hashmap. Since sorting is more informative than hashing we can do some more interesting things.

Usage

```
sortnut(sorted, ...)

## S3 method for class 'integer64'
sortnut(sorted, ...)

ordernut(table, order, ...)

## S3 method for class 'integer64'
ordernut(table, order, ...)

sortfin(sorted, x, ...)

## S3 method for class 'integer64'
sortfin(sorted, x, method = NULL, ...)

orderfin(table, order, x, ...)

## S3 method for class 'integer64'
orderfin(table, order, x, method = NULL, ...)

orderpos(table, order, x, ...)

## S3 method for class 'integer64'
orderpos(table, order, x, nomatch = NA, method = NULL, ...)

sortorderpos(sorted, order, x, ...)

## S3 method for class 'integer64'
sortorderpos(sorted, order, x, nomatch = NA, method = NULL, ...)

orderdup(table, order, ...)

## S3 method for class 'integer64'
orderdup(table, order, method = NULL, ...)

sortorderdup(sorted, order, ...)

## S3 method for class 'integer64'
sortorderdup(sorted, order, method = NULL, ...)

sortuni(sorted, nunique, ...)

## S3 method for class 'integer64'
sortuni(sorted, nunique, ...)

orderuni(table, order, nunique, ...)
```

```
## S3 method for class 'integer64'
orderuni(table, order, nunique, keep.order = FALSE, ...)

sortorderuni(table, sorted, order, nunique, ...)

## S3 method for class 'integer64'
sortorderuni(table, sorted, order, nunique, ...)

orderupo(table, order, nunique, ...)

## S3 method for class 'integer64'
orderupo(table, order, nunique, keep.order = FALSE, ...)

sortorderupo(sorted, order, nunique, keep.order = FALSE, ...)

## S3 method for class 'integer64'
sortorderupo(sorted, order, nunique, keep.order = FALSE, ...)

ordertie(table, order, nties, ...)

## S3 method for class 'integer64'
ordertie(table, order, nties, ...)

sortordertie(sorted, order, nties, ...)

## S3 method for class 'integer64'
sortordertie(sorted, order, nties, ...)

sorttab(sorted, nunique, ...)

## S3 method for class 'integer64'
sorttab(sorted, nunique, ...)

ordertab(table, order, nunique, ...)

## S3 method for class 'integer64'
ordertab(table, order, nunique, denormalize = FALSE, keep.order = FALSE, ...)

sortordertab(sorted, order, ...)

## S3 method for class 'integer64'
sortordertab(sorted, order, denormalize = FALSE, ...)

orderkey(table, order, na.skip.num = 0L, ...)

## S3 method for class 'integer64'
orderkey(table, order, na.skip.num = 0L, ...)
```

```
sortorderkey(sorted, order, na.skip.num = 0L, ...)

## S3 method for class 'integer64'
sortorderkey(sorted, order, na.skip.num = 0L, ...)

orderrnk(table, order, na.count, ...)

## S3 method for class 'integer64'
orderrnk(table, order, na.count, ...)

sortorderrnk(sorted, order, na.count, ...)

## S3 method for class 'integer64'
sortorderrnk(sorted, order, na.count, ...)

sortqtl(sorted, na.count, probs, ...)

## S3 method for class 'integer64'
sortqtl(sorted, na.count, probs, ...)

orderqtl(table, order, na.count, probs, ...)

## S3 method for class 'integer64'
orderqtl(table, order, na.count, probs, ...)
```

Arguments

sorted	a sorted integer64 vector
...	further arguments, passed from generics, ignored in methods
table	the original data with original order under the sorted vector
order	an integer order vector that turns 'table' into 'sorted'
x	an integer64 vector
method	see Details
nomatch	the value to be returned if an element is not found in the hashmap
nunique	number of unique elements, usually we get this from cache or call sortnut or ordernut
keep.order	determines order of results and speed: FALSE (the default) is faster and returns in sorted order, TRUE returns in the order of first appearance in the original data, but this requires extra work
nties	number of tied values, usually we get this from cache or call sortnut or ordernut
denormalize	FALSE returns counts of unique values, TRUE returns each value with its counts
na.skip.num	0 or the number of NAs. With 0, NAs are coded with 1L, with the number of NAs, these are coded with NA
na.count	the number of NAs, needed for this low-level function algorithm
probs	vector of probabilities in [0..1] for which we seek quantiles

Details

sortfun	orderfun	sortorderfun	see also	description
sortnut	ordernut			return number of tied and of unique values
sortfin	orderfin		%in%.integer64	return logical whether x is in table
	orderpos	sortorderpos	match()	return positions of x in table
	orderdup	sortorderdup	duplicated()	return logical whether values are duplicated
sortuni	orderuni	sortorderuni	unique()	return unique values (=dimensiontable)
	orderupo	sortorderupo	unique()	return positions of unique values
	ordertie	sortordertie		return positions of tied values
	orderkey	sortorderkey		positions of values in vector of unique values (match in dimens
sorttab	ordertab	sortordertab	table()	tabulate frequency of values
	orderrnk	sortorderrnk		rank averaging ties
sortqtl	orderqtl			return quantiles given probabilities

The functions sortfin, orderfin, orderpos and sortorderpos each offer three algorithms for finding x in table.

With method=1L each value of x is searched independently using *binary search*, this is fastest for small tables.

With method=2L the values of x are first sorted and then searched using *doubly exponential search*, this is the best all-around method.

With method=3L the values of x are first sorted and then searched using simple merging, this is the fastest method if table is huge and x has similar size and distribution of values.

With method=NULL the functions use a heuristic to determine the fastest algorithm.

The functions orderdup and sortorderdup each offer two algorithms for setting the truth values in the return vector.

With method=1L the return values are set directly which causes random write access on a possibly large return vector.

With method=2L the return values are first set in a smaller bit-vector – random access limited to a smaller memory region – and finally written sequentially to the logical output vector.

With method=NULL the functions use a heuristic to determine the fastest algorithm.

Value

see details

See Also

match()

Examples

```
message("check the code of 'optimizer64' for examples:")
print(optimizer64)
```

`sum.integer64` *Summary functions for integer64 vectors*

Description

Summary functions for integer64 vectors. Function 'range' without arguments returns the smallest and largest value of the 'integer64' class.

Usage

```
## S3 method for class 'integer64'
any(..., na.rm = FALSE)

## S3 method for class 'integer64'
all(..., na.rm = FALSE)

## S3 method for class 'integer64'
sum(..., na.rm = FALSE)

## S3 method for class 'integer64'
prod(..., na.rm = FALSE)

## S3 method for class 'integer64'
min(..., na.rm = FALSE)

## S3 method for class 'integer64'
max(..., na.rm = FALSE)

## S3 method for class 'integer64'
range(..., na.rm = FALSE, finite = FALSE)

lim.integer64()
```

Arguments

`...`	atomic vectors of class 'integer64'
`na.rm`	logical scalar indicating whether to ignore NAs
`finite`	logical scalar indicating whether to ignore NAs (just for compatibility with `range.default()`)

Details

The numerical summary methods always return `integer64`. Wherever integer methods would return `Inf` (or its negation), here the extreme 64-bit integer `9223372036854775807` is returned. See `min()` for more details about the behavior.

`lim.integer64` returns these limits in proper order `-9223372036854775807, +9223372036854775807` and without a `warning()`.

Value

`all()` and `any()` return a logical scalar

`range()` returns a integer64 vector with two elements

`min()`, `max()`, `sum()` and `prod()` return a integer64 scalar

See Also

`mean.integer64()` `cumsum.integer64()` `integer64()`

Examples

```
lim.integer64()
range(as.integer64(1:12))
```

`table` *Cross Tabulation and Table Creation for integer64*

Description

`table.integer64` uses the cross-classifying integer64 vectors to build a contingency table of the counts at each combination of vector values.

Usage

```
table(
  ...,
  exclude = if (useNA == "no") c(NA, NaN),
  useNA = c("no", "ifany", "always"),
  dnn = list.names(...),
  deparse.level = 1L
)

## S3 method for class 'integer64'
table(
  ...,
  return = c("table", "data.frame", "list"),
  order = c("values", "counts"),
  nunique = NULL,
  method = NULL,
  exclude = if (useNA == "no") c(NA, NaN),
  useNA = c("no", "ifany", "always"),
  dnn = list.names(...),
  deparse.level = 1L
)
```

Arguments

...	one or more objects which can be interpreted as factors (including character strings), or a list (or data frame) whose components can be so interpreted. (For as.table and as.data.frame, arguments passed to specific methods.)
exclude	levels to remove for all factors in ....
useNA	whether to include NA values in the table.
dnn	the names to be given to the dimensions in the result (the *dimnames names*).
deparse.level	controls how the default dnn is constructed. See Details.
return	choose the return format, see details
order	By default results are created sorted by "values", or by "counts"
nunique	NULL or the number of unique values of table (including NA). Providing nunique can speed-up matching when table has no cache. Note that a wrong nunique can cause undefined behaviour up to a crash.
method	NULL for automatic method selection or a suitable low-level method, see details

Details

If at least one argument of ... is integer64 and the remaining arguments of ... are integer64 or integer the table.integer64 method is used. Only this method supports the arguments return, order, nunique, and method.

This function automatically chooses from several low-level functions considering the size of x and the availability of a cache.

Suitable methods are

- hashmaptab (simultaneously creating and using a hashmap)
- hashtab (first creating a hashmap then using it)
- sortordertab (fast ordering)
- ordertab (memory saving ordering).

If the argument dnn is not supplied, the internal function list.names is called to compute the 'dimname names'. If the arguments in ... are named, those names are used. For the remaining arguments, deparse.level = 0 gives an empty name, deparse.level = 1 uses the supplied argument if it is a symbol, and deparse.level = 2 will deparse the argument.

Value

By default (with return="table") table() returns a *contingency table*, an object of class "table", an array of integer values. Note that unlike S the result is always an array, a 1D array if one factor is given. Note also that for multidimensional arrays this is a *dense* return structure which can dramatically increase RAM requirements (for large arrays with high mutual information, i.e. many possible input combinations of which only few occur) and that table() is limited to 2^31 possible combinations (e.g. two input vectors with 46340 unique values only). Finally note that the tabulated values or value-combinations are represented as dimnames and that the implied conversion of values to strings can cause *severe* performance problems since each string needs to be integrated into R's global string cache.

You can use the other `return=` options to cope with these problems, the potential combination limit is increased from `2^31` to `2^63` with these options, RAM is only required for observed combinations and string conversion is avoided.

With `return="data.frame"` you get a *dense* representation as a `data.frame()` (like that resulting from `as.data.frame(table(...))`) where only observed combinations are listed (each as a data.frame row) with the corresponding frequency counts (the latter as component named by `responseName`). This is the inverse of `xtabs()`.

With `return="list"` you also get a *dense* representation as a simple `list()` with components

- `values` a integer64 vector of the technically tabulated values, for 1D this is the tabulated values themselves, for kD these are the values representing the potential combinations of input values
- `counts` the frequency counts
- `dims` only for kD: a list with the vectors of the unique values of the input dimensions

Note

Note that by using `as.integer64.factor()` we can also input factors into `table.integer64` – only the `levels()` get lost.

See Also

`table()` for more info on the standard version coping with Base R's data types, `tabulate()` which can faster tabulate `integer`s with a limited range `[1L .. nL not too big]`, `unique.integer64()` for the unique values without counting them and `unipos.integer64()` for the positions of the unique values.

Examples

```
message("pure integer64 examples")
x <- as.integer64(sample(c(rep(NA, 9), 1:9), 32, TRUE))
y <- as.integer64(sample(c(rep(NA, 9), 1:9), 32, TRUE))
z <- sample(c(rep(NA, 9), letters), 32, TRUE)
table(x)
table(x, order="counts")
table(x, y)
table(x, y, return="data.frame")

message("via as.integer64.factor we can use 'table.integer64' also for factors")
table(x, as.integer64(as.factor(z)))
```

`tiepos` *Extract Positions of Tied Elements*

Description

`tiepos` returns the positions of those elements that participate in ties.

Usage

```
tiepos(x, ...)

## S3 method for class 'integer64'
tiepos(x, nties = NULL, method = NULL, ...)
```

Arguments

x	a vector or a data frame or an array or NULL.
...	ignored
nties	NULL or the number of tied values (including NA). Providing nties can speed-up when x has no cache. Note that a wrong nties can cause undefined behaviour up to a crash.
method	NULL for automatic method selection or a suitable low-level method, see details

Details

This function automatically chooses from several low-level functions considering the size of x and the availability of a cache.

Suitable methods are

- sortordertie (fast ordering)
- ordertie (memory saving ordering).

Value

an integer vector of positions

See Also

rank.integer64() for possibly tied ranks and unipos.integer64() for positions of unique values.

Examples

```
x <- as.integer64(sample(c(rep(NA, 9), 1:9), 32, TRUE))
tiepos(x)

stopifnot(identical(tiepos(x),  (1:length(x))[duplicated(x) | rev(duplicated(rev(x)))]))
```

union *Set Operations*

Description

Performs set union, intersection, (asymmetric!) difference, equality and membership on two vectors. As soon as an integer64 vector is involved, the operations are performed using integer64 semantics. Otherwise the base package functions are called.

Usage

```
union(x, y)

intersect(x, y)

setequal(x, y)

setdiff(x, y)

is.element(el, set)
```

Arguments

x, y, el, set	vectors (of the same mode) containing a sequence of items (conceptually) with no duplicated values.

Value

For union, a vector of a common mode or class.

For intersect, a vector of a common mode or class, or NULL if x or y is NULL.

For setdiff, a vector of the same mode or class as x.

A logical scalar for setequal and a logical of the same length as x for is.element.

See Also

base::union

Examples

```
x <- as.integer64(1:5)
y <- c(1L, 3L, 5L, 7L)
union(x, y)
intersect(x, y)
setdiff(x, y)
setequal(x, y)
is.element(x, y)
```

unipos *Extract Positions of Unique Elements*

Description

unipos returns the positions of those elements returned by unique().

Usage

```
unipos(x, incomparables = FALSE, order = c("original", "values", "any"), ...)

## S3 method for class 'integer64'
unipos(
  x,
  incomparables = FALSE,
  order = c("original", "values", "any"),
  nunique = NULL,
  method = NULL,
  ...
)
```

Arguments

x	a vector or a data frame or an array or NULL.
incomparables	ignored
order	The order in which positions of unique values will be returned, see details
...	ignored
nunique	NULL or the number of unique values (including NA). Providing nunique can speed-up when x has no cache. Note that a wrong nunique can cause undefined behaviour up to a crash.
method	NULL for automatic method selection or a suitable low-level method, see details

Details

This function automatically chooses from several low-level functions considering the size of x and the availability of a cache.

Suitable methods are

- hashmapupo (simultaneously creating and using a hashmap)
- hashupo (first creating a hashmap then using it)
- sortorderupo (fast ordering)
- orderupo (memory saving ordering).

The default `order="original"` collects unique values in the order of the first appearance in `x` like in `unique()`, this costs extra processing. `order="values"` collects unique values in sorted order like in `table()`, this costs extra processing with the hash methods but comes for free. `order="any"` collects unique values in undefined order, possibly faster. For hash methods this will be a quasi random order, for sort methods this will be sorted order.

Value

an integer vector of positions

See Also

`unique.integer64()` for unique values and `match.integer64()` for general matching.

Examples

```
x <- as.integer64(sample(c(rep(NA, 9), 1:9), 32, TRUE))
unipos(x)
unipos(x, order="values")
```

---

unique.integer64 *Extract Unique Elements from integer64*

Description

`unique` returns a vector like `x` but with duplicate elements/rows removed.

Usage

```
## S3 method for class 'integer64'
unique(
  x,
  incomparables = FALSE,
  order = c("original", "values", "any"),
  nunique = NULL,
  method = NULL,
  ...
)
```

Arguments

`x`	a vector or a data frame or an array or `NULL`.
`incomparables`	ignored
`order`	The order in which unique values will be returned, see details
`nunique`	NULL or the number of unique values (including NA). Providing `nunique` can speed-up matching when `x` has no cache. Note that a wrong 'nunique" can cause undefined behaviour up to a crash.

method	NULL for automatic method selection or a suitable low-level method, see details
...	ignored

Details

This function automatically chooses from several low-level functions considering the size of x and the availability of a cache.

Suitable methods are

- hashmapuni (simultaneously creating and using a hashmap)
- hashuni (first creating a hashmap then using it)
- sortuni (fast sorting for sorted order only)
- sortorderuni (fast ordering for original order only)
- orderuni (memory saving ordering).

The default order="original" returns unique values in the order of the first appearance in x like in unique(), this costs extra processing. order="values" returns unique values in sorted order like in table(), this costs extra processing with the hash methods but comes for free. order="any" returns unique values in undefined order, possibly faster. For hash methods this will be a quasi random order, for sort methods this will be sorted order.

Value

For a vector, an object of the same type of x, but with only one copy of each duplicated element. No attributes are copied (so the result has no names).

See Also

unique() for the generic, unipos() which gives the indices of the unique elements and table() which gives frequencies of the unique elements.

Examples

```
x <- as.integer64(sample(c(rep(NA, 9), 1:9), 32, TRUE))
unique(x)
unique(x, order="values")

stopifnot(identical(unique(x),  x[!duplicated(x)]))
stopifnot(identical(unique(x),  as.integer64(unique(as.integer(x)))))
stopifnot(identical(unique(x, order="values")
,  as.integer64(sort(unique(as.integer(x)), na.last=FALSE))))
```

Index