Package ‘bit64’

May 8, 2017

Type Package
Title A S3 Class for Vectors of 64bit Integers
Version 0.9-7
Date 2017-05-07
Author Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>
Maintainer Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>
Depends R (>= 3.0.1), bit (>= 1.1-12), utils, methods, stats
Description Package ‘bit64’ provides serializable S3 atomic 64bit (signed) integers. These are useful for handling database keys and exact counting in \( \pm 2^{63} \).

WARNING: do not use them as replacement for 32bit integers, integer64 are not supported for subcripting 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
LazyLoad yes
ByteCompile yes
URL http://ff.r-forge.r-project.org/
Encoding UTF-8
NeedsCompilation yes
Repository CRAN
Date/Publication 2017-05-08 13:21:40 UTC

R topics documented:

  bit64-package .......................................................... 2
  as.character.integer64 .............................................. 16
Description

Package 'bit64' provides fast serializable S3 atomic 64bit (signed) integers that can be used in vectors, matrices, arrays and data.frames. Methods are available for coercion from and to logistics, integers, doubles, characters and factors as well as many elementwise and summary functions.

Version 0.8 With 'integer64' vectors you can store very large integers at the expense of 64 bits, which is by factor 7 better than 'int64' from package 'int64'. Due to the smaller memory footprint, the atomic vector architecture and using only S3 instead of S4 classes, most operations are one to three orders of magnitude faster: Example speedups are 4x for serialization, 250x for adding, 900x for coercion and 2000x for object creation. Also 'integer64' avoids an ongoing (potentially infinite)
penalty for garbage collection observed during existence of 'int64' objects (see code in example section).

**Version 0.9** Package 'bit64' - which extends R with fast 64-bit integers - now has fast (single-threaded) implementations the most important univariate algorithmic operations (those based on hashing and sorting). We now have methods for 'match', 'quantile', 'median' and 'summary'. Regarding data management we also have novel generics 'unipos' (positions of the unique values), 'tiepos' (positions of ties), 'keypos' (positions of foreign keys in a sorted dimension table) and derived methods 'as.factor' and 'as.ordered'. This 64-bit functionality is implemented carefully to be not slower than the respective 32-bit operations in Base R and also to avoid outlying waiting times observed with 'order', 'rank' and 'table' (speedup factors 20/16/200 respective). This increases the dataset size with which we can work truly interactive. The speed is achieved by simple heuristic optimizers in high-level functions choosing the best from multiple low-level algorithms and further taking advantage of a novel caching if activated. In an example R session using a couple of these operations the 64-bit integers performed 22x faster than base 32-bit integers, hash-caching improved this to 24x, sortorder-caching was most efficient with 38x (caching hashing and sorting is not worth it with 32x at duplicated RAM consumption).

**Usage**

```r
integer64(length)
## S3 method for class 'integer64'
is(x)
## S3 replacement method for class 'integer64'
length(x) <- value
## S3 method for class 'integer64'
print(x, quote=FALSE, ...)
## S3 method for class 'integer64'
str(object, vec.len = strSvec.len, give.head = TRUE, give.length = give.head, ...)
```

**Arguments**

- `length` length of vector using `integer`
- `x` an integer64 vector
- `object` an integer64 vector
- `value` an integer64 vector of values to be assigned
- `quote` logical, indicating whether or not strings should be printed with surrounding quotes.
- `vec.len` see `str`
- `give.head` see `str`
- `give.length` see `str`
- `...` further arguments to the `NextMethod`

**Details**

- **Package:** bit64
- **Type:** Package
Value

`integer64` returns a vector of `integer64`, i.e. a vector of `double` decorated with class `integer64`.

Design considerations

64 bit integers are related to big data: we need them to overcome address space limitations. Therefore performance of the 64 bit integer type is critical. In the S language – designed in 1975 – atomic objects were defined to be vectors for a couple of good reasons: simplicity, option for implicit parallelization, good cache locality. In recent years many analytical databases have learnt that lesson: column based data bases provide superior performance for many applications, the result are products such as MonetDB, Sybase IQ, Vertica, Exasol, Ingres Vectorwise. If we introduce 64 bit integers not natively in Base R but as an external package, we should at least strive to make them as 'basic' as possible. Therefore the design choice of bit64 not only differs from `int64`, it is obvious: Like the other atomic types in Base R, we model data type 'integer64' as a contiguous `atomic` vector in memory, and we use the more basic `S3` class system, not `S4`. Like package `int64` we want our 'integer64' to be `serializeable`, therefore we also use an existing data type as the basis. Again the choice is obvious: R has only one 64 bit data type: doubles. By using `doubles`, `integer64` inherits some functionality such as `is.atomic`, `length`, `length<->`, `names`, `names<->`, `dim`, `dim<->`, `dimnames`, `dimnames`.

Our R level functions strictly follow the functional programming paradigm: no modification of arguments or other sideeffects. Before version 0.93 we internally deviated from the strict paradigm in order to boost performance. Our C functions do not create new return values, instead we pass-in the memory to be returned as an argument. This gives us the freedom to apply the C-function to new or old vectors, which helps to avoid unnecessary memory allocation, unnecessary copying and unnecessary garbage collection. Prior to 0.93 within our R functions we also deviated from conventional R programming by not using `attr<->` and `attributes<->` because they always did new memory allocation and copying in older R versions. If we wanted to set attributes of return values that we have freshly created, we instead used functions `setattr` and `setattributes` from package `bit`.

From version 0.93 `setattr` is only used for manipulating `cache` objects, in `ramsort.integer64` and `sort.integer64` and in `as.data.frame.integer64`.

Arithmetic precision and coercion

The fact that we introduce 64 bit long long integers – without introducing 128-bit long doubles – creates some subtle challenges: Unlike 32 bit `integers`, the `integer64` are no longer a proper subset of `double`. If a binary arithmetic operation does involve a double and a integer, it is a no-brainer to return `double` without loss of information. If an `integer64` meets a `double`, it is not trivial what type to return. Switching to `integer64` limits our ability to represent very large numbers, switching to `double` limits our ability to distinguish `x` from `x+1`. Since the latter is the purpose
of introducing 64 bit integers, we usually return integer64 from functions involving integer64, for example in c, cbind and rbind.

Different from Base R, our operators +, -, %/% and %% coerce their arguments to integer64 and always return integer64.

The multiplication operator * coerces its first argument to integer64 but allows its second argument to be also double: the second argument is internally coerced to 'long double' and the result of the multiplication is returned as integer64.

The division / and power ^ operators also coerce their first argument to integer64 and coerce internally their second argument to 'long double', they return as double, like sqrt, log, log2 and log10 do.

Creating and testing S3 class 'integer64'

Our creator function integer64 takes an argument length, creates an atomic double vector of this length, attaches an S3 class attribute 'integer64' to it, and that’s it. We simply rely on S3 method dispatch and interpret those 64bit elements as 'long long int'.

is.double currently returns TRUE for integer64 and might return FALSE in a later release. Consider is.double to have undefined behaviour and do query is.integer64 before querying is.double.

The methods is.integer64 and is.vector both return TRUE for integer64. Note that we did not patch storage.mode and typeof, which both continue returning 'double' Like for 32 bit integer, mode returns 'numeric' and as.double) tries coercing to double). It is likely that 'integer64' becomes a vmode in package ff.

Further methods for creating integer64 are range which returns the range of the data type if called without arguments, rep, seq.

For all available methods on integer64 vectors see the index below and the examples.

Index of implemented methods

<table>
<thead>
<tr>
<th>creating, testing, printing</th>
<th>see also</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NA_integer64</td>
<td>NA_integer_</td>
<td>NA constant</td>
</tr>
</tbody>
</table>
integer64 | integer |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>rep.integer64</td>
<td>rep</td>
</tr>
<tr>
<td>seq.integer64</td>
<td>seq</td>
</tr>
<tr>
<td>is.integer64</td>
<td>is</td>
</tr>
<tr>
<td>is.vector.integer64</td>
<td>is.vector</td>
</tr>
<tr>
<td>identical.integer64</td>
<td>identical</td>
</tr>
<tr>
<td>length&lt;-.integer64</td>
<td>length&lt;-</td>
</tr>
<tr>
<td>names&lt;-</td>
<td>names</td>
</tr>
<tr>
<td>dim&lt;-</td>
<td>dim</td>
</tr>
<tr>
<td>dimnames&lt;-</td>
<td>dimnames</td>
</tr>
<tr>
<td>str</td>
<td>str</td>
</tr>
<tr>
<td>print.integer64</td>
<td>print</td>
</tr>
<tr>
<td>str.integer64</td>
<td>str</td>
</tr>
</tbody>
</table>

### coeering to integer64

- as.integer64
- as.integer64.character
- as.integer64.double
- as.integer64.integer
- as.integer64.integer64
- as.integer64.logical
- as.integer64.NULL

### coeering from integer64

- as.bitstring
- as.character
- as.double
- as.integer
- as.logical

### data structures

- c.integer64 | c |
- cbind.integer64 | cbind |
- rbind.integer64 | rbind |
- as.data.frame.integer64 | as.data.frame |

### subscripting

- [.integer64 | [ |
- [<-.integer64 | <-
- [[.integer64 | []
- [[<-.integer64 | [[]<-

create zero atomic vector

inherited from Base R

inherited from Base R

inherited from Base R

inherited from Base R

inherited from Base R

inherited from Base R, does not print values correctly

coerce atomic object to data.frame

coerce atomic object to data.frame

vector concatenate

column bind

row bind

inherited from Base R since we have coercion

vector and array extract

vector and array assign

scalar extract

scalar assign

generic

generic
<table>
<thead>
<tr>
<th>Operators</th>
<th>See Also</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary operators</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>.integer64</td>
<td>+</td>
</tr>
<tr>
<td>-</td>
<td>.integer64</td>
<td>-</td>
</tr>
<tr>
<td>*</td>
<td>.integer64</td>
<td>*</td>
</tr>
<tr>
<td>^</td>
<td>.integer64</td>
<td>^</td>
</tr>
<tr>
<td>/.</td>
<td>.integer64</td>
<td>/</td>
</tr>
<tr>
<td>%.</td>
<td>.integer64</td>
<td>%</td>
</tr>
<tr>
<td>%%.</td>
<td>.integer64</td>
<td>%%.</td>
</tr>
<tr>
<td>Comparison operators</td>
<td></td>
<td></td>
</tr>
<tr>
<td>==</td>
<td>.integer64</td>
<td>==</td>
</tr>
<tr>
<td>!=</td>
<td>.integer64</td>
<td>!=</td>
</tr>
<tr>
<td>&lt;</td>
<td>.integer64</td>
<td>&lt;</td>
</tr>
<tr>
<td>&lt;=</td>
<td>.integer64</td>
<td>&lt;=</td>
</tr>
<tr>
<td>&gt;</td>
<td>.integer64</td>
<td>&gt;</td>
</tr>
<tr>
<td>&gt;=</td>
<td>.integer64</td>
<td>&gt;=</td>
</tr>
<tr>
<td>Logical operators</td>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>.integer64</td>
<td>!</td>
</tr>
<tr>
<td>&amp;</td>
<td>.integer64</td>
<td>&amp;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.integer64</td>
</tr>
<tr>
<td>xor</td>
<td>.integer64</td>
<td>xor</td>
</tr>
<tr>
<td>Math functions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>is.na</td>
<td>.integer64</td>
<td>is.na</td>
</tr>
<tr>
<td>format</td>
<td>.integer64</td>
<td>format</td>
</tr>
<tr>
<td>abs</td>
<td>.integer64</td>
<td>abs</td>
</tr>
<tr>
<td>sign</td>
<td>.integer64</td>
<td>sign</td>
</tr>
<tr>
<td>log</td>
<td>.integer64</td>
<td>log</td>
</tr>
<tr>
<td>log10</td>
<td>.integer64</td>
<td>log10</td>
</tr>
<tr>
<td>log2</td>
<td>.integer64</td>
<td>log2</td>
</tr>
<tr>
<td>sqrt</td>
<td>.integer64</td>
<td>sqrt</td>
</tr>
<tr>
<td>ceiling</td>
<td>.integer64</td>
<td>ceiling</td>
</tr>
<tr>
<td>floor</td>
<td>.integer64</td>
<td>floor</td>
</tr>
<tr>
<td>trunc</td>
<td>.integer64</td>
<td>trunc</td>
</tr>
<tr>
<td>round</td>
<td>.integer64</td>
<td>round</td>
</tr>
<tr>
<td>signif</td>
<td>.integer64</td>
<td>signif</td>
</tr>
<tr>
<td>Cumulative functions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cummin</td>
<td>.integer64</td>
<td>cummin</td>
</tr>
<tr>
<td>cummax</td>
<td>.integer64</td>
<td>cummax</td>
</tr>
<tr>
<td>cumin</td>
<td>.integer64</td>
<td>cumin</td>
</tr>
<tr>
<td>cumprod</td>
<td>.integer64</td>
<td>cumprod</td>
</tr>
<tr>
<td>cumsum</td>
<td>.integer64</td>
<td>cumsum</td>
</tr>
<tr>
<td>diff</td>
<td>.integer64</td>
<td>diff</td>
</tr>
<tr>
<td>Summary functions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>range</td>
<td>.integer64</td>
<td>range</td>
</tr>
</tbody>
</table>
### bit64-package

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>min.integer64</td>
<td>min</td>
</tr>
<tr>
<td>max.integer64</td>
<td>max</td>
</tr>
<tr>
<td>sum.integer64</td>
<td>sum</td>
</tr>
<tr>
<td>mean.integer64</td>
<td>mean</td>
</tr>
<tr>
<td>prod.integer64</td>
<td>prod</td>
</tr>
<tr>
<td>all.integer64</td>
<td>all</td>
</tr>
<tr>
<td>any.integer64</td>
<td>any</td>
</tr>
</tbody>
</table>

**Algorithmically complex functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>match.integer64</td>
<td>match position of x in table (h/o/so)</td>
</tr>
<tr>
<td>%in%.integer64</td>
<td>%in% is x in table? (h/o/so)</td>
</tr>
<tr>
<td>duplicated.integer64</td>
<td>duplicated is current element duplicate of previous one? (h/o/so)</td>
</tr>
<tr>
<td>unique.integer64</td>
<td>unique (shorter) vector of unique values only (h/s/o/so)</td>
</tr>
<tr>
<td>unipos.integer64</td>
<td>unipos positions corresponding to unique values (h/s/o/so)</td>
</tr>
<tr>
<td>tiepos.integer64</td>
<td>tiepos positions of values that are tied (h/o/so)</td>
</tr>
<tr>
<td>keypos.integer64</td>
<td>keypos position of current value in sorted list of unique values (h/o/so)</td>
</tr>
<tr>
<td>as.factor.integer64</td>
<td>as.factor convert to (unordered) factor with sorted levels of previous values (h/o/so)</td>
</tr>
<tr>
<td>as.ordered.integer64</td>
<td>as.ordered convert to ordered factor with sorted levels of previous values (h/o/so)</td>
</tr>
<tr>
<td>table.integer64</td>
<td>table unique values and their frequencies (h/s/o/so)</td>
</tr>
<tr>
<td>sort.integer64</td>
<td>sort sorted vector (h/s/o/so)</td>
</tr>
<tr>
<td>order.integer64</td>
<td>order positions of elements that would create sorted vector (h/o/so)</td>
</tr>
<tr>
<td>rank.integer64</td>
<td>rank (average) ranks of non-NAs, NAs kept in place (h/s/o/so)</td>
</tr>
<tr>
<td>quantile.integer64</td>
<td>quantile (existing) values at specified percentiles (h/s/o/so)</td>
</tr>
<tr>
<td>median.integer64</td>
<td>median (existing) value at percentile 0.5 (h/s/o/so)</td>
</tr>
<tr>
<td>summary.integer64</td>
<td>summary (h/s/o/so)</td>
</tr>
</tbody>
</table>

**Helper functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>minusclass</td>
<td>minusclass removing class attribute</td>
</tr>
<tr>
<td>plusclass</td>
<td>plusclass inserting class attribute</td>
</tr>
<tr>
<td>binattr</td>
<td>binattr define binary op behaviour</td>
</tr>
</tbody>
</table>

**Tested I/O functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>read.table</td>
<td>inherited from Base R</td>
</tr>
<tr>
<td>write.table</td>
<td>inherited from Base R</td>
</tr>
<tr>
<td>serialize</td>
<td>inherited from Base R</td>
</tr>
<tr>
<td>unserialize</td>
<td>inherited from Base R</td>
</tr>
<tr>
<td>save</td>
<td>inherited from Base R</td>
</tr>
<tr>
<td>load</td>
<td>inherited from Base R</td>
</tr>
<tr>
<td>dput</td>
<td>inherited from Base R</td>
</tr>
<tr>
<td>dget</td>
<td>inherited from Base R</td>
</tr>
</tbody>
</table>

### Limitations inherited from implementing 64 bit integers via an external package

- **vector size** of atomic vectors is still limited to `.Machine$integer.max`. However, external memory extending packages such as `ff` or `bigmemory` can extend their address space now.
with integer64. Having 64 bit integers also help with those not so obvious address issues that arise once we exchange data with SQL databases and datawarehouses, which use big integers as surrogate keys, e.g. on indexed primary key columns. This puts R into a relatively strong position compared to certain commercial statistical softwares, which sell database connectivity but neither have the range of 64 bit integers, nor have integers at all, nor have a single numeric data type in their macro-glue-language.

- **literals** such as 123L would require changes to Base R, up to then we need to write (and call) as.integer64(123L) or as.integer64(123) or as.integer64('123'). Only the latter allows to specify numbers beyond Base R’s numeric data types and therefore is the recommended way to use – using only one way may facilitate migrating code to literals at a later stage.

**Limitations inherited from Base R, Core team, can you change this?**

- **identical** with default parameters does not distinguish all bit-patterns of doubles. For testing purposes we provide a wrapper identical.integer64 that will distinguish all bit-patterns. It would be desirable to have a single call of identical handle both, double and integer64.
- the **colon** operator : officially does not dispatches S3 methods, however, we have made it generic

  ```
  from <- lim.integer64()[1]
  to <- from+99
  from:to
  ```

  As a limitation remains: it will only dispatch at its first argument from but not at its second to.
- **is.double** does not dispatches S3 methods, However, we have made it generic and it will return FALSE on integer64.
- **c** only dispatches c.integer64 if the first argument is integer64 and it does not recursively dispatch the proper method when called with argument recursive=TRUE Therefore

  ```
  c(list(integer64Linteger64Linteger64))
  ```

does not work and for now you can only call

  ```
  c.integer64(list(x,x))
  ```

- **generic binary operators** fail to dispatch *any* user-defined S3 method if the two arguments have two different S3 classes. For example we have two classes bit and bitwhich sparsely representing boolean vectors and we have methods &.bit and &.bitwhich. For an expression involving both as in bit & bitwhich, none of the two methods is dispatched. Instead a standard method is dispatched, which neither handles bit nor bitwhich. Although it lacks symmetry, the better choice would be to dispatch simply the method of the class of the first argument in case of class conflict. This choice would allow authors of extension packages providing coherent behaviour at least within their contributed classes. But as long as none of the package authors methods is dispatched, he cannot handle the conflicting classes at all.
- **unlist** is not generic and if it were, we would face similar problems as with c()
vector with argument mode='integer64' cannot work without adjustment of Base R

as.vector with argument mode='integer64' cannot work without adjustment of Base R

is.vector does not dispatch its method is.vector.integer64

mode<- drops the class 'integer64' which is returned from as.integer64. Also it does not remove an existing class 'integer64' when assigning mode 'integer'.

storage.mode<- does not support external data types such as as.integer64

matrix does drop the 'integer64' class attribute.

array does drop the 'integer64' class attribute. In current R versions (1.15.1) this can be circumvented by activating the function as.vector.integer64 further down this file. However, the CRAN maintainer has requested to remove as.vector.integer64, even at the price of breaking previously working functionality of the package.

str does not print the values of integer64 correctly

further limitations

subscripting non-existing elements and subscripting with NAs is currently not supported. Such subscripting currently returns 921886843727407266 instead of NA (the NA value of the underlying double code). Following the full R behaviour here would either destroy performance or require extensive C-coding.

Note

integer64 are useful for handling database keys and exact counting in +-2^63. 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. Do understand that integer64 can only be useful over double if we do not coerce it to double.

While
integer + double -> double + double -> double
or
1L + 0.5 -> 1.5
for additive operations we coerce to integer64
integer64 + double -> integer64 + integer64 -> integer64
hence
as.integer64(1) + 0.5 -> 1LL + 0LL -> 1LL

see section "Arithmetic precision and coercion" above

Author(s)

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

See Also

integer in base R
Examples

message("Using integer64 in vector")
x <- integer64(8)  # create 64 bit vector
x
is.atomic(x)  # TRUE
is.integer64(x)  # TRUE
is.numeric(x)  # TRUE
is.integer(x)  # FALSE - debatable
is.double(x)  # FALSE - might change
x[1:2] <- 1:2  # assigned value is recycled as usual
x[1:6]  # subscripting as usual
length(x) <- 13  # changing length as usual
x
rep(x, 2)  # replicate as usual
seq(as.integer64(1), 10)  # seq.integer64 is dispatched on first given argument
seq(to=as.integer64(10), 1)  # seq.integer64 is dispatched on first given argument
seq.integer64(along.with=x)  # or call seq.integer64 directly
# c.integer64 is dispatched only if *first* argument is integer64 ...
x <- c(x, runif(length(x), max=100))
# ... and coerces everything to integer64 - including double
x
names(x) <- letters  # use names as usual
x

message("Using integer64 in array - note that 'matrix' currently does not work")
message("as.vector.integer64 removed as requested by the CRAN maintainer")
message("as consequence 'array' also does not work anymore")

message("we still can create a matrix or array by assigning 'dim'")
y <- rep(as.integer64(NA), 12)
dim(y) <- c(3, 4)
dimnames(y) <- list(letters[1:3], LETTERS[1:4])
y["a",] <- 1:2  # assigning as usual
y
y[1:2, -4]  # subscripting as usual
# cbind.integer64 dispatched on any argument and coerces everything to integer64
cbind(E=1:3, F=runif(3, 0, 100), G=c("-1","0","1"), y)

message("Using integer64 in data.frame")
str(as.data.frame(x))
str(as.data.frame(y))
str(data.frame(y))
str(data.frame(y))
d <- data.frame(x=x, y=runif(length(x), 0, 100))
d
d$x

message("Using integer64 with csv files")
f164 <- tempfile()
write.csv(d, file=f164, row.names=FALSE)
e <- read.csv(f164, colClasses=c("integer64", NA))
unlink(f164)
str(e)
identical.integer64(d$x, e$x)

message("Serializing and unserializing integer64")
dput(d, fi64)
e <- dget(fi64)
identical.integer64(d$x, e$x)
e <- d[,]
save(e, file=fi64)
rm(e)
load(file=fi64)
identical.integer64(d, e)

### A couple of unit tests follow hidden in a dontshow() directive ###

## Not run:
message("== Differences between integer64 and int64 ==")
require(bit64)
require(int64)

message("--- integer64 is atomic ---")
is.atomic(integer64())
#is.atomic(int64())
str(integer64(3))
#str(int64(3))

message("--- The following performance numbers are measured under RWin64 ---")
message("--- under RWin32 the advantage of integer64 over int64 is smaller ---")

message("--- integer64 needs 7x/5x less RAM than int64 under 64/32 bit OS
(and twice the RAM of integer as it should be) ---")
#as.vector(object.size(int64(1e6))/object.size(integer64(1e6)))
as.vector(object.size(integer64(1e6))/object.size(integer(1e6)))

message("--- integer64 creates 2000x/1300x faster than int64 under 64/32 bit OS
(and 3x the time of integer) ---")
t32 <- system.time(integer(1e8))
t64 <- system.time(integer64(1e8))
#T64 <- system.time(int64(1e7))×10  # using 1e8 as above stalls our R on an i7 8 GB RAM Thinkpad
#T64/t64
t64/t32

i32 <- sample(1e6)
d64 <- as.double(i32)

message("--- the following timings are rather conservative since timings
of integer64 include garbage collection -- due to looped calls")
message("--- integer64 coerces 900x/100x faster than int64
under 64/32 bit OS (and 2x the time of coercing to integer) --")
t32 <- system.time(for(i in 1:1000)as.integer(d64))
t64 <- system.time(for(i in 1:1000)as.integer64(d64))
#T64 <- system.time(as.int64(d64))×1000
```r
#T64/t64
t64/t32
td64 <- system.time(for(i in 1:1000)as.double(i32))
t64 <- system.time(for(i in 1:1000)as.integer64(i32))
#T64 <- system.time(for(i in 1:10)as.int64(i32))*100
#T64/t64
t64/td64

message("-- integer64 serializes 4x/0.8x faster than int64
under 64/32 bit OS (and less than 2x/6x the time of integer or double) --")
t32 <- system.time(for(i in 1:10)serialize(i32, NULL))
td64 <- system.time(for(i in 1:10)serialize(d64, NULL))
i64 <- as.integer64(i32);
t64 <- system.time(for(i in 1:10)serialize(i64, NULL))
rm(i64); gc()
#i64 <- as.int64(i32);
#T64 <- system.time(for(i in 1:10)serialize(I64, NULL))
#rm(I64); gc()
#T64/t64
t64/t32
t64/td64

message("-- integer64 adds 250x/60x faster than int64
under 64/32 bit OS (and less than 6x the time of integer or double) --")
td64 <- system.time(for(i in 1:100)d64+d64)
t32 <- system.time(for(i in 1:100)i32+i32)
i64 <- as.integer64(i32);
t64 <- system.time(for(i in 1:100)i64+i64)
rm(i64); gc()
#i64 <- as.int64(i32);
#T64 <- system.time(for(i in 1:10)I64+I64)*10
#rm(I64); gc()
#T64/t64
t64/t32
t64/td64

message("-- integer64 sums 3x/0.2x faster than int64
(and at about 5x/60X the time of integer and double) --")
td64 <- system.time(for(i in 1:100)sum(d64))
t32 <- system.time(for(i in 1:100)sum(i32))
i64 <- as.integer64(i32);
t64 <- system.time(for(i in 1:100)sum(i64))
rm(i64); gc()
#i64 <- as.int64(i32);
#T64 <- system.time(for(i in 1:100)sum(I64))
#rm(I64); gc()
#T64/t64
t64/t32
t64/td64

message("-- integer64 diffs 5x/0.85x faster than integer and double
(int64 version 1.0 does not support diff) --")
```
```r
t664 <- system.time(for(i in 1:10) diff(d64, lag=2L, differences=2L))
t32 <- system.time(for(i in 1:10) diff(i32, lag=2L, differences=2L))
i64 <- as.integer64(i32);
t64 <- system.time(for(i in 1:10) diff(i64, lag=2L, differences=2L))
rm(i64); gc()
t64/t32

message("-- integer64 subscripts 1000x/340x faster than int64
(and at the same speed / 10x slower as integer) --")
ts32 <- system.time(for(i in 1:1000) sample(1e6, 1e3))
t64 <- system.time(for(i in 1:1000) i64[sample(1e6, 1e3)])
rn(i64); gc()

message("-- integer64 assigns 200x/90x faster than int64
(and 50x/160x slower than integer) --")
ts32 <- system.time(for(i in 1:1000) sample(1e6, 1e3))
t64 <- system.time(for(i in 1:1000) i64[sample(1e6, 1e3)] <- 1:1e3)
rm(i64); gc()

message("-- integer64 subscripts 1000x/340x faster than int64
(and at the same speed / 10x slower as integer) --")
```
```
#i64 <- as.int64(i32);
tdfi64 <- system.time(dfi64<-data.frame(a=i64, b=i64, c=i64))
tdfs164 <- system.time(dfi64[1e6:1])
#f164 <- tempfile()
tdfw164 <- system.time(write.csv(dfi64, file=F164, row.names=FALSE))
tdfri64 <- system.time(read.csv(F164, colClasses=rep("int64", 3)))
#unlink(F164)
#rm(F164, dfi64); gc()

message("-- integer64 coerces 40x/6x faster to data.frame than int64
(and factor 1/9 slower than integer) --")
tdfi64/tdfi64
tdfi64/tdfi32
message("-- integer64 subscripts from data.frame 20x/2.5x faster than int64
(and 3x/13x slower than integer) --")
tdfs164/tdfsi64
tdfs164/tdfsi32
message("-- integer64 csv writes about 2x/0.5x faster than int64
(and about 1.5x/5x slower than integer) --")
tdfw164/tdfw164
tdfw164/tdfw132
message("-- integer64 csv reads about 3x/1.5 faster than int64
(and about 2x slower than integer) --")
tdfri64/tdfri64
tdfri64/tdfri32

rm(i32, d64); gc()

message("-- investigating the impact on garbage collection: --")
message("-- the fragmented structure of int64 messes up R's RAM --")
message("-- and slows down R's garbage collection just by existing --")

td32 <- double(21)
td32[1] <- system.time(d64 <- double(1e7))[3]
for (i in 2:11) td32[i] <- system.time(gc(), gcFirst=FALSE)[3]
rm(d64)
for (i in 12:21) td32[i] <- system.time(gc(), gcFirst=FALSE)[3]

t64 <- double(21)
t64[1] <- system.time(i64 <- integer64(1e7))[3]
for (i in 2:11) t64[i] <- system.time(gc(), gcFirst=FALSE)[3]
rm(i64)
for (i in 12:21) t64[i] <- system.time(gc(), gcFirst=FALSE)[3]

#T64 <- double(21)
#T64[1] <- system.time(I64 <- int64(1e7))[3]
#for (i in 2:11) T64[i] <- system.time(gc(), gcFirst=FALSE)[3]
#rm(I64)
#for (i in 12:21) T64[i] <- system.time(gc(), gcFirst=FALSE)[3]

#matplot(1:21, cbind(td32, t64, T64), pch=c("d", "i", "I"), log="y")
matplot(1:21, cbind(td32, t64), pch=c("d", "i"), log="y")
```
## 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

```r
as.bitstring(x, ...)  # S3 method for class 'integer64'
as.character(x, ...)  # S3 method for class 'integer64'
as.double(x, keep.names = FALSE, ...)  # S3 method for class 'integer64'
as.integer(x, ...)  # S3 method for class 'integer64'
as.logical(x, ...)  # S3 method for class 'integer64'
as.factor(x)  # S3 method for class 'integer64'
as.ordered(x)
```

### Arguments

- `x` an integer64 vector
- `keep.names` FALSE, set to TRUE to keep a names vector
- `...` further arguments to the `NextMethod`

### Value

- `as.bitstring` returns a string of.
- The other methods return atomic vectors of the expected types

### Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>
as.data.frame.integer64

Description

Coercing integer64 vector to data.frame.

Usage

```r
## S3 method for class 'integer64'
as.data.frame(x, ...)
```

Arguments

- `x` an integer64 vector
- `...` 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?

Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

See Also

cbind.integer64 integer64
Examples

```r
as.data.frame.integer64(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**

```r
NA_integer64_
as.integer64(x, ...)
```

```r
## S3 method for class 'integer64'
as.integer64(x, ...)
## S3 method for class 'NULL'
as.integer64(x, ...)
## S3 method for class 'character'
as.integer64(x, ...)
## S3 method for class 'double'
as.integer64(x, keep.names = FALSE, ...)
## S3 method for class 'integer'
as.integer64(x, ...)
## S3 method for class 'logical'
as.integer64(x, ...)
## S3 method for class 'factor'
as.integer64(x, ...)
```

**Arguments**

- `x`: an atomic vector
- `keep.names`: FALSE, set to TRUE to keep a names vector
- `...`: further arguments to the NextMethod

**Details**

`as.integer64.character` is realized using C function `strtoll` which does not support scientific notation. Instead of '1e6' use '1000000'.

**Value**

The other methods return atomic vectors of the expected types.
benchmark64

Author(s)
Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

See Also
as.character.integer64 integer64

Examples

as.integer64(as.character(lim.integer64()))

Description

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.

Usage

benchmarck64(nsmall = 2^16, nbig = 2^25, timefun = repeat.time)
optimizer64(nsmall = 2^16, nbig = 2^25, timefun = repeat.time,
, what = c("match", "%in%", "duplicated", "unique", "unipos", "table", "rank", "quantile")
, 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 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.integer64
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 | dito when cache contains hashmap, see hashcache
sortordercache | dito when cache contains sorting and ordering, see sortordercache
ordercache | dito when cache contains ordering only, see ordercache
allcache | dito 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
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 session.

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.

**Author(s)**

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>
See Also

integer64

Examples

message("this small example using system.time does not give serious timings\nthis 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[-1,1]),
  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[-1,1])),
  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]))}
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)
Turning base R functions into S3 generics for bit64

Description

Turn those base functions S3 generic which are used in bit64

Usage

```r
from:to
#--as-cran complains about \method{:}{default}(from, to)
#--as-cran complains about \method{:}{integer64}(from, to)
is.double(x)
  ## Default S3 method:
is.double(x)
  ## S3 method for class 'integer64'
is.double(x)
match(x, table, ...)
  ## Default S3 method:
match(x, table, ...)
x %in% table
  ## Default S3 method:
x %in% table
rank(x, ...)
  ## Default S3 method:
rank(x, ...)
order(...)
  ## 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`
- `from`: scalar denoting first element of sequence
- `to`: scalar denoting last element of sequence
- `...`: ignored

Details

The following functions are turned into S3 generics in order to dispatch methods for `integer64`:
\code{|link{:}}
\code{|link{is.double}}
\code{|link{match}}
\code{|link{%in%}}
\code{|link{rank}}
\code{|link{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

**Author(s)**

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

**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 usual functions ‘c’, ‘cbind’ and ‘rbind’

**Usage**

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

**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.

**Value**

- `c` returns a integer64 vector of the total length of the input
- `cbind` and `rbind` return a integer64 matrix

**Note**

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

**Author(s)**

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

**See Also**

`rep.integer64`, `seq.integer64`, `as.data.frame.integer64`, `integer64`

**Examples**

```r
c(as.integer64(1), 2:6)  
cbind(1:6, as.integer(1:6))  
rbind(1:6, as.integer(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)
getchache(x, which)
remcache(x)
## S3 method for class 'cache'
print(x, all.names = FALSE, pattern, ...)
still.identical(x, y)

Arguments

x an integer64 vector (or a cache object in case of print.cache)
y an integer64 vector
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 passed to ls when listing the cache content
pattern passed to ls when listing the cache content
... ignored

Details

A cache is an environment attached to an atomic object with the attrib 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.
Function still.identical(x,y) checks whether the objects x and y
Function newcache(x) creates a new cache referencing x
Function jamcache(x) forces x to have a cache
Function cache(x) returns the cache attached to x if it is not found to be outdated
Function setcache(x, which, value) assigns a value into the cache of x
Function getcache(x, which) gets cache value 'which' from x
Function remcache removes the cache from x
**cumsum.integer64**

**Value**

see details

**Author(s)**

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

**See Also**

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

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

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

**Examples**

```r
x <- as.integer64(sample(c(rep(NA, 9), 1:9), 32, TRUE))
y <- x
still.identical(x, y)
y[1] <- NA
still.identical(x, y)
mycache <- newcache(x)
ls(mycache)
mycache
rm(mycache)
jamcache(x)
cache(x)
x[1] <- NA
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

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

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

Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

See Also

`sum.integer64` `integer64`

Examples

```r
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**

```r
## 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) and `orderdup` (memory saving ordering).

**Value**

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

**Author(s)**

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

**See Also**

duplicated, unique.integer64

**Examples**

```r
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

```r
## S3 method for class 'integer64'
_x_ ...
## 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
- `value` an atomic vector with values to be assigned
- `...` further arguments to the `NextMethod`

Value

A vector or scalar of class 'integer64'

Note

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

Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

See Also

- `[ integer64 `
format.integer64

Examples

```r
as.integer64(1:12)[1:3]
x <- as.integer64(1:12)
dim(x) <- c(3,4)
x
x[1]
x[,2:3]
```

Description

Unary operators and functions for integer64 vectors.

Usage

```r
## S3 method for class 'integer64'
format(x, justify="right", ...)
## S3 method for class 'integer64'
is.na(x)
## S3 method for class 'integer64'
is.nan(x)
## S3 method for class 'integer64'
is.finite(x)
## S3 method for class 'integer64'
is.infinite(x)
## S3 method for class 'integer64'
!x
## 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)
## S3 method for class 'integer64'
log2(x)
## S3 method for class 'integer64'
log10(x)
## S3 method for class 'integer64'
floor(x)
## S3 method for class 'integer64'
ceiling(x)
## S3 method for class 'integer64'
trunc(x, ...)
```
## S3 method for class 'integer64'
round(x, digits=0)

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

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

### Arguments

- **x**: an atomic vector of class 'integer64'
- **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`)
- **justify**: should it be right-justified (the default), left-justified, centred or left alone.
- **center**: see `scale`
- **scale**: see `scale`
- **...**: further arguments to the `NextMethod`

### 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

### Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

### See Also

- `xor.integer64`
- `integer64`

### Examples

```r
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)
sortordercache(x, has.na = NULL, stable = NULL)
ordercache(x, has.na = NULL, stable = NULL, optimize = "time")

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
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
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
... passed to hashmap

Details

The result of relative expensive operations hashmap, ramsort, ramsortorder and 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.

Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>
See Also

cache for caching functions and nunique for methods benefitting from small caches

Examples

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

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

```r
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, ...)
hashmapuni(x, ...)
## S3 method for class 'integer64'
hashmapuni(x, nunique=NULL, minfac=1.5, hashbits=NULL, ...)
hashupo(cache, ...)
```
## Arguments

**x**  
an integer64 vector

**hashmap**  
an object of class 'hashmap' i.e. here 'cache_integer64'

**minfac**  
minimum factor by which the hasmap has more elements compared to the data

**hashbits**  
length of hashmap is $2^{\text{hashbits}}$

**cache**  
an optional cache object into which to put the hashmap (by default a new cache is created)

**nunique**  
giving correct number of unique elements can help reducing the size of the hashmap

**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

...  
further arguments, passed from generics, ignored in methods

## Details

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

Author(s)
Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

See Also
match

Examples
```r
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)
hy <- hashmap(y)
hashpos(hy, x)
hashrev(hx, y)
hashfin(hy, x)
hashrin(hx, y)
hashdup(hy)
hashuni(hy)
hashuni(hy, keep.order=TRUE)
hashmapuni(y)
hashup0(hy)
hashup0(hy, keep.order=TRUE)
hashmapup0(y)
hashtab(hy)
hashmaptab(y)

stopifnot(identical(match(as.integer(x),as.integer(y)),hashpos(hy, x)))
stopifnot(identical(match(as.integer(x),as.integer(y)),hashrev(hx, y)))
stopifnot(identical(as.integer(x) %in% as.integer(y), hashfin(hy, x)))
stopifnot(identical(as.integer(x) %in% as.integer(y), hashrin(hx, y)))
stopifnot(identical(duplicated(as.integer(y)), hashdup(hy)))

stopifnot(identical(as.integer64(unique(as.integer(y))), hashuni(hy, keep.order=TRUE)))
stopifnot(identical(sort(hashuni(hy, keep.order=FALSE)), sort(hashuni(hy, keep.order=TRUE))))
stopifnot(identical(y[hashup0(hy, keep.order=FALSE)], hashuni(hy, keep.order=FALSE)))
stopifnot(identical(y[hashup0(hy, keep.order=TRUE)], hashuni(hy, keep.order=TRUE)))
stopifnot(identical(hashpos(hy, hashuni(hy, keep.order=FALSE)), hashup0(hy, keep.order=FALSE)))
stopifnot(identical(hashpos(hy, hashuni(hy, keep.order=TRUE)), hashup0(hy, keep.order=TRUE)))
stopifnot(identical(hashuni(hy, keep.order=FALSE), hashtab(hy)$values))
stopifnot(identical(as.vector(table(as.integer(y), useNA="ifany")),
                    hashtab(hy)$counts[order.integer64(hashtab(hy)$values)]))

stopifnot(identical(hashuni(hy, keep.order=TRUE), hashmapuni(y)))
stopifnot(identical(hashup0(hy, keep.order=TRUE), hashmapup0(y)))
```
stopifnot(identical(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\")
b <- 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)
```

**Arguments**

- `x`  
  atomic vector of class 'integer64'
- `y`  
  atomic vector of class 'integer64'
- `num.eq`  
  see `identical`
- `single.NA`  
  see `identical`
- `attrib.as.set`  
  see `identical`
- `ignore.bytecode`  
  see `identical`

**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.

Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

See Also

\[\text{is.sorted.integer64} \]

Examples

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

Description

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

Usage

```r
## S3 method for class 'integer64'
is.sorted(x, ...)
## S3 method for class 'integer64'
na.count(x, ...)
## S3 method for class 'integer64'
nvalid(x, ...)
## S3 method for class 'integer64'
nunique(x, ...)
## S3 method for class 'integer64'
nties(x, ...)
```

Arguments

- `x` some object
- `...` ignored
keypos

Details

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

is.sorted checks for sortedness of x (NAs sorted first)

na.count returns the number of NAs

nvalid returns the number of valid data points, usually length minus na.count.
nunique returns the number of unique values
nties returns the number of tied values.

Value

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

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.

Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegal@truecluster.com>

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)
na.count(x)
nvalid(x)
nunique(x)
tties(x)
table.integer64(x)
x

<table>
<thead>
<tr>
<th>keypos</th>
<th>Extract Positions in redundant dimension table</th>
</tr>
</thead>
</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.  
method NULL for automatic method selection or a suitable low-level method, see details  
... ignored

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) and `orderkey` (memory saving ordering).

Value

an integer vector of the same length as codex containing positions relativ to codesort(unique(x), na.last=FALSE)

Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

See Also

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

Examples

```r
x <- as.integer64(sample(c(rep(NA, 9), 1:9), 32, TRUE))  
keypos(x)  
stopifnot(identical(keypos(x), match.integer64(x, sort(unique(x), na.last=FALSE))))
```
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

```r
## 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%` are `hashpos` (hash table lookup), `hashrev` (reverse lookup), `sortorderpos` (fast ordering) and `orderpos` (memory saving ordering). Suitable methods for `match.integer64` are `hashfin` (hash table lookup), `hashrin` (reverse lookup), `sortfin` (fast sorting) and `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 \( \text{table}[j] \), 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 \text{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.

**Author(s)**

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

**See Also**

match

**Examples**

```r
x <- as.integer64(c(NA, 0:9), 32)
table <- as.integer64(c(1:9, NA))
match.integer64(x, table)
"%in%.integer64"(x, 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))
stopifnot(identical(match.integer64(x, table), match(as.integer(x), as.integer(table))))
stopifnot(identical("%in%.integer64"(x, table), as.integer(x) & in% as.integer(table)))
```

---

```r
# Not run:
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(t)if(any(is.na(t)))NA else which.min(t)))
}

# End(Not run)

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 ... ...$ : 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 ... ...$ : chr [1:10] "duplicated.64" "hashdup" "sortorderdup1" ... ...$ : chr [1:3] "prep" "both" "use" $ : num [1:15, 1:3] 0 0 0.00104 0.00104 ... ...$ : chr [1:3] "unique.64" "hashmapuni" "hashuni" ... ...$ : chr [1:3] "prep" "both" "use" $ : num [1:14, 1:3] 0 0 0.000992 0.000992 ... ...$ : chr [1:14] "unique" "unipos.64" "hashmapupo" "hashupo" ... ...$ : chr [1:3] "prep" "both" "use" $ : num [1:13, 1:3] 0 0 0 0.000419 ... ...$ : chr [1:7] "rank" "rank.keep" "rank.64" "sortorderrnk"
prank

(P)ercent (Rank)s

Description

Function `prank` 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].

Examples

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

```r
prank(x, ...)  
```

## S3 method for class 'integer64'

```r
prank(x, method = NULL, ...)
```

Arguments

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

Details

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

Value

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

Author(s)

Jens Oehlschlägel <Jens.Oehlschlaege@truecluster.com>

See Also

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

Examples

```r
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)uantity*(Tile)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

```r
# S3 method for class 'integer64'
qtile(x, probs = seq(0, 1, 0.25), ...)
```

Arguments

- **x**: a integer64 vector
- **object**: a integer64 vector
- **probs**: numeric vector of probabilities with values in [0,1] - possibly containing NAs
- **names**: logical; if TRUE, the result has a names attribute. Set to FALSE for speedup with many probs.
- **type**: an integer selecting the quantile algorithm, currently only 0 is supported, see details
- **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.
- **...**: ignored

Details

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) and `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.
ramsort.integer64

Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

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 integer64 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'
shullorder(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'
mergoorder(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'
Arguments

- **x**: a vector to be sorted by `ramsort` and `ramsortorder`, i.e. the output of `sort`
- **i**: integer positions to be modified by `ramorder` and `ramsortorder`, default is `1:n`, in this case the output is similar to `order`
- **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
- **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
- **restlevel**: number of remaining recursion levels before `quicksort` switches from recursing to `shellsort`
- **radixbits**: size of radix in bits
- **VERBOSE**: cat some info about chosen method
- **...**: further arguments, passed from generics, ignored in methods

Details

See `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 sortorder-methods change both x and i

Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

See Also

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

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

```r
## 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) and `orderrnk` (memory saving ordering).

Value

A numeric vector of the same length as x.

Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

See Also

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

Examples

```r
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")))
```

Description

Replicate elements of integer64 vectors

Usage

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

Arguments

- **x**: a vector of 'integer64' to be replicated
- **...**: further arguments passed to `NextMethod`
seq.integer64

Value

rep returns a integer64 vector

Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

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)

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 does coerce its arguments 'from', 'to' and 'by' to integer64. If not provided, the argument 'by' is automatically determined as +1 or -1, but the size of 'by' is not calculated as in seq (because this might result in a non-integer value).

Value

an integer64 vector with the generated sequence
sort.integer64

Note
In base R: currently is not generic and does not dispatch, see section "Limitations inherited from Base R" in integer64

Author(s)
Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

See Also
c.integer64 rep.integer64 as.data.frame.integer64 integer64

Examples

# colon not activated: as.integer64(1:12)
seq(as.integer64(1), 12, 2)
seq(as.integer64(1), by=2, length.out=6)

sort.integer64 High-level integer64 methods for sorting and ordering

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

Usage

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

```r
## 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 and ramsortorder, i.e. the output of sort
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
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.

Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

See Also

`sort`, `sortcache`

Examples

```r
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.integer64(x)
message("slower with less RAM, this calls 'ramsortorder'")
order.integer64(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

```r
sortnut(sorted, ...) 
ordenut(table, order, ...) 
sortfin(sorted, x, ...) 
orderfin(table, order, x, ...) 
orderpos(table, order, x, ...) 
sortorderpos(sorted, order, x, ...) 
```
orderdup(table, order, ...)  
sortorderdup(sorted, order, ...)  
orderrun(table, order, nunique, ...)  
sortorderrun(table, sorted, order, nunique, ...)  
orderrup(table, order, nunique, ...)  
sortorderrup(sorted, order, nunique, keep.order = FALSE, ...)  
ordertie(table, order, nties, ...)  
sortordertie(sorted, order, nties, ...)  
sorttab(sorted, nunique, ...)  
ordertab(table, order, nunique, ...)  
sortordertab(sorted, order, nunique, ...)  
orderkkey(table, order, na.skip.num = 0L, ...)  
sortorderkkey(sorted, order, na.skip.num = 0L, ...)  
orderrnk(table, order, na.count, ...)  
sortorderrnk(sorted, order, na.count, ...)  
## S3 method for class 'integer64'  
sortnut(sorted, ...)  
## S3 method for class 'integer64'  
ordernut(table, order, ...)  
## S3 method for class 'integer64'  
sortfin(sorted, x, method=NULL, ...)  
## S3 method for class 'integer64'  
orderfin(table, order, x, method=NULL, ...)  
## S3 method for class 'integer64'  
orderpos(table, order, x, nomatch=NA, method=NULL, ...)  
## S3 method for class 'integer64'  
sortorderpos(sorted, order, x, nomatch=NA, method=NULL, ...)  
## S3 method for class 'integer64'  
orderdup(table, order, method=NULL, ...)  
## S3 method for class 'integer64'  
sortorderdup(sorted, order, method=NULL, ...)  
## S3 method for class 'integer64'  
sortuni(sorted, nunique, ...)  
## S3 method for class 'integer64'  
orderrun(table, order, nunique, keep.order=FALSE, ...)  
## S3 method for class 'integer64'  
sortorderrun(table, sorted, order, nunique, ...)  
## S3 method for class 'integer64'  
orderrup(table, order, nunique, keep.order = FALSE, ...)  
## S3 method for class 'integer64'  
sortorderrup(sorted, order, nunique, keep.order = FALSE, ...)  
## S3 method for class 'integer64'  
ordertie(table, order, nties, ...)  
## S3 method for class 'integer64'  
sortordertie(sorted, order, nties, ...)  
## S3 method for class 'integer64'  
sorttab(sorted, nunique, ...)  
## S3 method for class 'integer64'
## S3 method for class 'integer64'
ordertab(table, order, nunique, denormalize=FALSE, keep.order=FALSE, ...)

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

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

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

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

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

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

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

### Arguments

- x: an **integer64** vector
- sorted: a sorted **integer64** vector
- table: the original data with original order under the sorted vector
- order: an **integer** order vector that turns 'table' into 'sorted'
- nunique: number of unique elements, usually we get this from cache or call sortnut or ordernut
- 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
- 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 sorted order, TRUE returns in the order of first appearance in the original data, but this requires extra work
- probs: vector of probabilities in [0..1] for which we seek quantiles
- 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, the latter needed for **as.factor.integer64**
- na.count: the number of NAs, needed for this low-level function algorithm
- method: see details
- ...: further arguments, passed from generics, ignored in methods

### Details

<table>
<thead>
<tr>
<th>sortfun</th>
<th>orderfun</th>
<th>sortorderfun</th>
<th>see also</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sortnut</td>
<td>ordernut</td>
<td>sortordertab</td>
<td>%in%.integer64</td>
<td>return number of tied and of unique values</td>
</tr>
<tr>
<td>sortfin</td>
<td>orderfin</td>
<td>sortorderkey</td>
<td>match</td>
<td>return logical whether x is in table</td>
</tr>
<tr>
<td></td>
<td>orderpos</td>
<td>sortorderpos</td>
<td>duplicated</td>
<td>return positions of x in table</td>
</tr>
<tr>
<td></td>
<td>orderdup</td>
<td>sortorderdup</td>
<td></td>
<td>return logical whether values are duplicated</td>
</tr>
</tbody>
</table>
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 dimensiontable)
sorttab  ordertab  sortordertab  table  tabulate frequency of values
sortrnk  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 allround 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

Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

See Also

`match`

Examples

```r
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

```r
## S3 method for class 'integer64'
all(..., na.rm = FALSE)
## S3 method for class 'integer64'
any(..., 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)
lim.integer64()
## S3 method for class 'integer64'
sum(..., na.rm = FALSE)
## S3 method for class 'integer64'
prod(..., na.rm = FALSE)
```

Arguments

- `...`: atomic vectors of class 'integer64'
- `na.rm`: logical scalar indicating whether to ignore NAs

Details

The numerical summary methods always return `integer64`. Therefore, the methods for `min`, `max` and `range` do not return `+Inf`, `-Inf` on empty arguments, but `+9223372036854775807`, `-9223372036854775807` (in this sequence). The same is true if only NAs are submitted with argument `na.rm=TRUE`. `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

Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>
See Also

`mean.integer64`, `cumsum.integer64`, `integer64`

Examples

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

description

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

Usage

```r
table.integer64(...
  , return = c("table","data.frame","list")
  , order = c("values","counts")
  , nunique = NULL
  , method = NULL
  , dnn = list.names(...), deparse.level = 1
)
```

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.)

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.

order By default results are created sorted by "values", or by "counts"

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

return choose the return format, see details

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’.
Details

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) and 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.

Arguments exclude, useNA, are not supported, i.e. NAs are always tabulated, and, different from table they are sorted first if order="values".

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.

Note that because of the existence of as.factor.integer64 the standard table function – within its limits – can also be used for integer64, and especially for combining integer64 input with other data types.
See Also
table for more info on the standard version coping with Base R’s data types, tabulate which can faster tabulate integers 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.integer64(x)
table.integer64(x, order="counts")
table.integer64(x, y)
table.integer64(x, y, return="data.frame")

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

message("via as.factor.integer64 we can also use 'table' for integer64")
table(x)
table(x, exclude=NULL)
table(x, z, exclude=NULL)

---
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

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>a vector or a data frame or an array or NULL.</td>
</tr>
<tr>
<td>nties</td>
<td>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.</td>
</tr>
<tr>
<td>method</td>
<td>NULL for automatic method selection or a suitable low-level method, see details</td>
</tr>
<tr>
<td>...</td>
<td>ignored</td>
</tr>
</tbody>
</table>
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) and `ordertie` (memory saving ordering).

Value

an integer vector of positions

Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

See Also

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

Examples

```r
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)))]))
```

---

Description

`unipos` returns the positions of those elements returned by `unique`.

Usage

```r
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
- `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

... 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 `hashmapupo` (simultaneously creating and using a hashmap), `hashupo` (first creating a hashmap then using it), `sortorderupo` (fast ordering) and `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

Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

See Also

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

Examples

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

stopifnot(identical(unipos(x), 1:length(x)[!duplicated(x)]))
stopifnot(identical(unipos(x), match.integer64(unique(x), x)))
stopifnot(identical(unipos(x, order="values"), match.integer64(unique(x, order="values"), x)))
stopifnot(identical(unique(x), x[unipos(x)]))
stopifnot(identical(unique(x, order="values"), x[unipos(x, order="values")]))
```
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

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) and `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).

Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

See Also

`unique` for the generic, `unipos` which gives the indices of the unique elements and `table.integer64` which gives frequencies of the unique elements.

Examples

```r
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))))
```
Description

Binary operators for integer64 vectors.

Usage

```r
## S3 method for class 'integer64'
e1 & e2
## S3 method for class 'integer64'
e1 | e2
## S3 method for class 'integer64'
xor(x, y)
## 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'
binattr(e1, e2) # for internal use only
```

Arguments

- `e1` an atomic vector of class 'integer64'
- `e2` an atomic vector of class 'integer64'
xor.integer64

x  an atomic vector of class 'integer64'
y  an atomic vector of class 'integer64'

Value

&, |, xor, !=, ==, <, <=, >, >= return a logical vector
^ and / return a double vector
+, -, *, %, %% return a vector of class 'integer64'

Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

See Also

format.integer64 integer64

Examples

as.integer64(1:12) - 1
Index

!.integer64 (format.integer64), 31
!=.integer64 (xor.integer64), 64
*Topic category
table.integer64, 58
*Topic classes
as.character.integer64, 16
as.data.frame.integer64, 17
as.integer64.character, 18
bit64-package, 2
c.integer64, 25
cumsum.integer64, 27
extract.replace.integer64, 30
format.integer64, 31
identical.integer64, 37
rep.integer64, 50
seq.integer64, 51
sum.integer64, 57
xor.integer64, 64
*Topic datasets
benchmark64.data, 22
optimizer64.data, 43
*Topic environment
cache, 26
hashcache, 33
is.sorted.integer64, 38
*Topic logic
duplicated.integer64, 29
match.integer64, 41
unipos, 61
unique.integer64, 62
*Topic manip
as.character.integer64, 16
as.data.frame.integer64, 17
as.integer64.character, 18
bit64-package, 2
c.integer64, 25
cumsum.integer64, 27
duplicated.integer64, 29
extract.replace.integer64, 30
format.integer64, 31
hashmap, 34
identical.integer64, 37
keypos, 39
match.integer64, 41
ramsort.integer64, 47
rep.integer64, 50
seq.integer64, 51
sort.integer64, 52
sortnut, 53
sum.integer64, 57
tiepos, 60
unipos, 61
unique.integer64, 62
xor.integer64, 64
*Topic methods
bit64S3, 23
is.sorted.integer64, 38
*Topic misc
benchmark64, 19
*Topic package
bit64-package, 2
*Topic programming
hashmap, 34
ramsort.integer64, 47
sort.integer64, 52
sortnut, 53
*Topic univar
keypos, 39
prank, 44
qtile, 45
rank.integer64, 49
tiepos, 60
*, 5, 7, 65
*.integer64, 7
*.integer64 (xor.integer64), 64
+, 5, 7, 65
+.integer64, 7
+.integer64 (xor.integer64), 64
\).
\[\text{\textbf{Machine}. } 8\]

\[/. , 7, 65\]

\[. \text{integer64}, 7\]

\[. \text{integer64} (\text{xor.integer64}), 64\]

\[\text{abs}, 7, 32\]

\[\text{abs.integer64}, 7\]

\[\text{abs.integer64} (\text{format.integer64}), 31\]

\[\text{all}, 8, 57\]

\[\text{all.integer64}, 8\]

\[\text{all.integer64} (\text{sum.integer64}), 57\]

\[\text{any}, 8, 57\]

\[\text{any.integer64}, 8\]

\[\text{any.integer64} (\text{sum.integer64}), 57\]

\[\text{array}, 10\]

\[\text{as.bitstring}, 6\]

\[\text{as.bitstring} (\text{as.character.integer64}), 16\]

\[\text{as.bitstring.integer64}, 6\]

\[\text{as.character}, 6, 16\]

\[\text{as.character.integer64}, 6, 16, 19\]

\[\text{as.data.frame}, 6, 17\]

\[\text{as.data.frame.integer64}, 4, 6, 17, 25, 51, 52\]

\[\text{as.double}, 5, 6, 16\]

\[\text{as.double.integer64}, 6\]

\[\text{as.double.integer64} (\text{as.character.integer64}), 16\]

\[\text{as.factor}, 8\]

\[\text{as.factor.integer64}, 8, 27, 55, 59\]

\[\text{as.factor.integer64} (\text{as.character.integer64}), 16\]

\[\text{as.integer}, 6, 16\]

\[\text{as.integer.integer64}, 6\]

\[\text{as.integer.integer64} (\text{as.character.integer64}), 16\]

\[\text{as.integer64}, 6\]

\[\text{as.integer64} (\text{as.integer64}), 64\]

\[\text{as.integer64} (\text{as.integer64} \text{character}), 18\]

\[\text{as.integer64.character}, 6, 17, 18\]

\[\text{as.integer64.double}, 6\]

\[\text{as.integer64.factor}, 59\]

\[\text{as.integer64.integer}, 6\]
as.integer64, integer64, 6
as.integer64.logical, 6
as.integer64:NULL, 6
as.logical, 6, 16
as.logical.integer64, 6
as.logical.integer64
  (as.character.integer64), 16
as.ordered, 8
as.ordered.integer64, 8, 27
as.ordered.integer64
  (as.character.integer64), 16
as.vector, 10
atomic, 4

benchmark64, 19, 22
benchmark64.data, 22
binattr, 8
binattr (xor.integer64), 64
bit, 4, 9
bit64, 24
bit64 (bit64-package), 2
bit64-package, 2
bit64S3, 23
bitwhich, 9
c, 5, 6, 9, 25
c.integer64, 6, 9, 25, 51, 52
cache, 4, 26, 33–35, 39
cbind, 5, 6, 25
cbind.integer64, 6, 17
cbind.integer64 (c.integer64), 25
ceiling, 7, 32
ceiling.integer64, 7
ceiling.integer64 (format.integer64), 31
character, 6
cummax, 7, 28
cummax.integer64, 7
cummax.integer64 (cumsum.integer64), 27
cummin, 7, 28
cummin.integer64, 7
cummin.integer64 (cumsum.integer64), 27
cumprod, 7, 28
cumprod.integer64, 7
cumprod.integer64 (cumsum.integer64), 27
cumsum, 7, 28
cumsum.integer64, 7, 27, 58
data.frame, 6, 59
dget, 8
diff, 7, 28
diff.integer64, 7
diff.integer64 (cumsum.integer64), 27
digest, 35
dim, 4, 6
dim<-, 6
dimnames, 4, 6
dimnames<-, 6
double, 4–6, 9
dput, 8
duplicated, 8, 29, 35, 55
duplicated.integer64, 8, 27, 29
extract.replace.integer64, 30
ff, 5, 8, 49
floor, 7, 32
floor.integer64, 7
floor.integer64 (format.integer64), 31
format, 7, 16, 32
format.integer64, 7, 31, 65
getcache (cache), 26
hashcache, 20, 23, 27, 33, 39, 41
hashdup, 29
hashdup (hashmap), 34
hashfin, 41
hashfin (hashmap), 34
hashfun (hashmap), 34
hashmap, 20, 33, 34
hashmaptab, 59
hashmaptab (hashmap), 34
hashmapuni, 63
hashmapuni (hashmap), 34
hashmapupo, 62
hashmapupo (hashmap), 34
hashpos, 41
hashpos (hashmap), 34
hashrev, 41
hashrev (hashmap), 34
hashrin, 41
hashrin (hashmap), 34
hashstab, 59
hashstab (hashmap), 34
hashuni, 63
hashuni (hashmap), 34
hashupo, 62
hashupo (hashmap), 34
identical, 6, 9, 37, 38
identical.integer64, 6, 9, 37
inherits, 4
integer, 3–6, 10, 55, 60
integer64, 17, 19, 21, 23–25, 28, 30, 32, 38, 51, 52, 55, 58, 59, 65
integer64 (bit64-package), 2
invisible, 24
is, 6
is.atomic, 4
is.double, 5, 9, 24
is.double (bit64S3), 23
is.finite.integer64 (format.integer64), 31
is.infinite.integer64 (format.integer64), 31
is.integer, 6
is.integer.integer64 (bit64-package), 2
is.integer64, 5, 6
is.integer64 (bit64-package), 2
is.na, 7, 32
is.na.integer64, 7
is.na.integer64 (format.integer64), 31
is.nan.integer64 (format.integer64), 31
is.sorted, 27
is.sorted.integer64, 38
is.vector, 5, 6, 10
is.vector.integer64, 6, 10
is.vector.integer64 (bit64-package), 2
jamcache (cache), 26
keypos, 8, 27, 39
keypos.integer64, 8
length, 4, 6, 39
length<-, 6
length<-.integer64, 6
length<-.integer64 (bit64-package), 2
levels, 59
lim.integer64 (sum.integer64), 57
list, 59
load, 8
log, 5, 7, 32
log.integer64, 7
log.integer64 (format.integer64), 31
log10, 5, 7, 32
log10.integer64, 7
log10.integer64 (format.integer64), 31
log2, 5, 7, 32
log2.integer64, 7
log2.integer64 (format.integer64), 31
logical, 6
ls, 26
match, 8, 24, 35, 36, 42, 55, 56
match (bit64S3), 23
match.integer64, 8, 27, 40, 41, 62
matrix, 10
max, 8, 57
max.integer64, 8
max.integer64 (sum.integer64), 57
mean, 8
mean.integer64, 8, 58
mean.integer64 (qtile), 45
median, 8
median.integer64, 8, 27
median.integer64 (qtile), 45
mergeorder.integer64 (ramsort.integer64), 47
mergesort.integer64 (ramsort.integer64), 47
mergesortorder.integer64 (ramsort.integer64), 47
min, 8, 57
min.integer64, 8
min.integer64 (sum.integer64), 57
minusclass, 8
mode, 5
mode<-, 10
na.count, 27
na.count.integer64
(is.sorted.integer64), 38
NA_integer64_ (as.integer64.character), 18
NA_integer_, 5
names, 4, 6
names<-, 6
newcache (cache), 26
NextMethod, 3, 16, 18, 25, 30, 32, 50
nties, 27
nties.integer64 (is.sorted.integer64), 38
NULL, 6
nunique, 27, 34
nunique.integer64, 33
nunique.integer64 (is.sorted.integer64), 38
nvalid, 27
nvalid.integer64 (is.sorted.integer64), 38

optimizer64, 43
optimizer64 (benchmark64), 19
optimizer64.data, 43
order, 8, 24, 48, 53
order (bit64S3), 23
order.integer64, 8, 50
order.integer64 (sort.integer64), 52
ordercache, 20, 27, 39
ordercache (hashcache), 33
orderdup, 29
orderdup (sortnut), 53
orderfin, 41
orderfin (sortnut), 53
orderkey, 40
orderkey (sortnut), 53
ordernut (sortnut), 53
orderpos, 41
orderpos (sortnut), 53
orderqtl, 46
orderqtl (sortnut), 53
orderrnk, 50
orderrnk (sortnut), 53
ordertab, 59
ordertab (sortnut), 53
ordertie, 61
ordertie (sortnut), 53
orderuni, 63
orderuni (sortnut), 53
orderup, 62
orderup (sortnut), 53

plusclass, 8
prank, 27, 44, 50
prank.integer64, 45
print, 6
print.cache (cache), 26
print.integer64, 6
print.integer64 (bit64-package), 2
prod, 8, 57
prod.integer64, 8
prod.integer64 (sum.integer64), 57
qtile, 27, 45, 45
quantile, 8, 47
quantile.integer64, 8, 27
quantile.integer64 (qtile), 45
quickorder.integer64 (ramsort.integer64), 47
quicksort.integer64 (ramsort.integer64), 47
quicksortorder.integer64 (ramsort.integer64), 47
radixorder.integer64 (ramsort.integer64), 47
radixsort.integer64 (ramsort.integer64), 47
radixsortorder.integer64 (ramsort.integer64), 47
ramorder, 33, 48
ramorder.integer64, 40
ramorder.integer64 (ramsort.integer64), 47
ramsort, 33, 48, 49, 52
ramsort.default, 49
ramsort.integer64, 4, 47
ramsortorder, 33, 48, 52
ramsortorder.integer64 (ramsort.integer64), 47
range, 5, 7, 57
range.integer64, 7
range.integer64 (sum.integer64), 57
rank, 8, 50
rank (bit64S3), 23
rank.integer64, 8, 24, 45, 47, 49, 61
rbind, 5, 6, 25
rbind.integer64, 6
rbind.integer64 (c.integer64), 25
read.table, 8
remcache (cache), 26
rep, 5, 6, 51
rep.integer64, 6, 25, 50, 51, 52
repeat.time, 19
round, 7, 32
round.integer64, 7
round.integer64 (format.integer64), 31
s3, 4, 24
s4, 4
save, 8
scale, 32
scale.integer64 (format.integer64), 31
seq, 5, 6, 51
seq.integer64, 6, 25, 51
serialize, 4, 8
setattr, 4
setattributes, 4
setcache (cache), 26
shelloder.integer64
(ramsort.integer64), 47
shellsort.integer64
(ramsort.integer64), 47
shellsortorder.integer64
(ramsort.integer64), 47
sign, 7, 32
sign.integer64, 7
sign.integer64 (format.integer64), 31
signif, 7, 32
signif.integer64, 7
signif.integer64 (format.integer64), 31
sort, 8, 48, 52, 53
sort.integer64, 4, 8, 49, 52
sortcache, 27, 39, 49, 53
sortcache (hashcache), 33
sortfin, 41
sortfin (sortnut), 53
sortnut, 53
sortordercache, 20, 23, 27, 39, 41
sortordercache (hashcache), 33
sortorderdup, 29
sortorderdup (sortnut), 53
sortorderkey, 40
sortorderkey (sortnut), 53
sortorderpos, 41
sortorderpos (sortnut), 53
sortorderrnnk, 50
sortorderrnnk (sortnut), 53
sortordertab, 59
sortordertab (sortnut), 53
sortordertie, 61
sortordertie (sortnut), 53
sortorderuni, 63
sortorderuni (sortnut), 53
sortorderup, 62
sortorderup (sortnut), 53
sortqtl, 46
sortqtl (sortnut), 53
sorttab (sortnut), 53
sortuni, 63
sortuni (sortnut), 53
sqrt, 5, 7, 32
sqrt.integer64, 7
sqrt.integer64 (format.integer64), 31
still.identical (cache), 26
storage.mode, 5
storage.mode<-, 10
str, 3, 6, 10
str.integer64, 6
str.integer64 (bit64-package), 2
sum, 8, 57
sum.integer64, 8, 28, 57
summary, 8
summary.integer64, 8, 27
summary.integer64 (qtile), 45
system.time, 19
table, 8, 35, 56, 59, 60, 62, 63
table.integer64, 8, 19, 27, 58, 63
tabulate, 60
tiepos, 8, 27, 60
tiepos.integer64, 8
trunc, 7, 32
trunc.integer64, 7
trunc.integer64 (format.integer64), 31
typeof, 5
unipos، 8, 27, 61, 63
unipos.integer64, 8, 19, 60, 61
unique, 8, 35, 56, 61–63
unique.integer64, 8, 19, 27, 29, 40, 60, 62, 63
unlist, 9
unserialize, 8
vector, 10
vmode, 5
warning, 57
write.table, 8
xor, 7, 65
xor.integer64, 7, 32, 64
xtabs, 59