Package ‘bit64’

September 22, 2024

Type Package

Title A S3 Class for Vectors of 64bit Integers

Version 4.5.2

Date 2024-09-22

Author Jens Oehlschlägel [aut, cre],
Leonardo Silvestri [ctb],
Ofek Shilon [ctb]

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

Depends R (>= 3.0.1), bit (>= 4.0.0), utils, methods, stats

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

License GPL-2 | GPL-3

LazyLoad yes

ByteCompile yes

URL https://github.com/truecluster/bit64

Encoding UTF-8

Repository CRAN

Repository/R-Forge/Project ff

Repository/R-Forge/Revision 177

Repository/R-Forge/DateTimeStamp 2018-08-17 17:45:18

Date/Publication 2024-09-22 13:50:02 UTC

NeedsCompilation yes

Contents

`bit64-package` *A S3 class for vectors of 64bit integers*

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 logicals, 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 wich 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

```
 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  = strO$vec.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
Version:	0.5.0
Date:	2011-12-12
License:	GPL-2
LazyLoad:	yes
Encoding:	latin1

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 paragdim: no modification of arguments or other side-effects. 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 `integer`s, 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 internaly 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.

argument1	**op**	**argument2**	**->**	**coerced1**	**op**	**coerced2**	**->**	**result**
integer64	+	double	->	integer64	+	integer64	->	integer64
double	+	integer64	->	integer64	+	integer64	->	integer64
integer64	-	double	->	integer64	-	integer64	->	integer64
double	-	integer64	->	integer64	-	integer64	->	integer64
integer64	%/%	double	->	integer64	%/%	integer64	->	integer64
double	%/%	integer64	->	integer64	%/%	integer64	->	integer64
integer64	%%	double	->	integer64	%%	integer64	->	integer64
double	%%	integer64	->	integer64	%%	integer64	->	integer64
integer64	*	double	->	integer64	*	long double	->	integer64
double	*	integer64	->	integer64	*	integer64	->	integer64
integer64	/	double	->	integer64	/	long double	->	double
double	/	integer64	->	integer64	/	long double	->	double
integer64	^	double	->	integer64	/	long double	->	double
double	^	integer64	->	integer64	/	long double	->	double

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 behavior 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 possible that 'integer64' becomes a vmode in package ff.
Further methods for creating integer64 are range which returns the range of the data type if calles without arguments, rep, seq.
For all available methods on integer64 vectors see the index below and the examples.

Index of implemented methods

creating,testing,printing	**see also**	**description**
NA_integer64_	NA_integer_	NA constant
integer64	integer	create zero atomic vector
runif64	runif	create random vector
rep.integer64	rep	
seq.integer64	seq	
is.integer64	is	
	is.integer	inherited from Base R
is.vector.integer64	is.vector	
identical.integer64	identical	
length<-.integer64	length<-	
	length	inherited from Base R
	names<-	inherited from Base R
	names	inherited from Base R
	dim<-	inherited from Base R
	dim	inherited from Base R
	dimnames<-	inherited from Base R
	dimnames	inherited from Base R
	str	inherited from Base R, does not print values correctly
print.integer64	print	
str.integer64	str	

coercing to integer64	**see also**	**description**
as.integer64		generic
as.integer64.bitstring	as.bitstring	
as.integer64.character	character	
as.integer64.double	double	
as.integer64.integer	integer	
as.integer64.integer64	integer64	
as.integer64.logical	logical	
as.integer64.NULL	NULL	

coercing from integer64	**see also**	**description**
as.list.integer64	as.list	generic
as.bitstring	as.bitstring	generic
as.bitstring.integer64		

as.character.integer64	as.character	
as.double.integer64	as.double	
as.integer.integer64	as.integer	
as.logical.integer64	as.logical	

data structures	**see also**	**description**
c.integer64	c	vector concatenate
cbind.integer64	cbind	column bind
rbind.integer64	rbind	row bind
as.data.frame.integer64	as.data.frame	coerce atomic object to data.frame
	data.frame	inherited from Base R since we have coercion

subscripting	**see also**	**description**
[.integer64	[	vector and array extract
[<-.integer64	[<-	vector and array assign
[[.integer64	[[	scalar extract
[[<-.integer64	[[<-	scalar assign

binary operators	**see also**	**description**
+.integer64	+	returns integer64
-.integer64	-	returns integer64
*.integer64	*	returns integer64
^.integer64	^	returns double
/.integer64	/	returns double
%/%.integer64	%/%	returns integer64
%%.integer64	%%	returns integer64

comparison operators	**see also**	**description**
==.integer64	==	
!=.integer64	!=	
<.integer64	<	
<=.integer64	<=	
>.integer64	>	
>=.integer64	>=	

logical operators	**see also**	**description**
!.integer64	!	
&.integer64	&	
|.integer64	|	
xor.integer64	xor	

math functions	**see also**	**description**
is.na.integer64	is.na	returns logical
format.integer64	format	returns character
abs.integer64	abs	returns integer64
sign.integer64	sign	returns integer64
log.integer64	log	returns double
log10.integer64	log10	returns double

log2.integer64	log2	returns double
sqrt.integer64	sqrt	returns double
ceiling.integer64	ceiling	dummy returning its argument
floor.integer64	floor	dummy returning its argument
trunc.integer64	trunc	dummy returning its argument
round.integer64	round	dummy returning its argument
signif.integer64	signif	dummy returning its argument

cumulative functions	**see also**	**description**
cummin.integer64	cummin	
cummax.integer64	cummax	
cumsum.integer64	cumsum	
cumprod.integer64	cumprod	
diff.integer64	diff	

summary functions	**see also**	**description**
range.integer64	range	
min.integer64	min	
max.integer64	max	
sum.integer64	sum	
mean.integer64	mean	
prod.integer64	prod	
all.integer64	all	
any.integer64	any	

algorithmically complex functions	**see also**	**description (caching)**
match.integer64	match	position of x in table (h//o/so)
%in%.integer64	%in%	is x in table? (h//o/so)
duplicated.integer64	duplicated	is current element duplicate of previous one? (h//o/so)
unique.integer64	unique	(shorter) vector of unique values only (h/s/o/so)
unipos.integer64	unipos	positions corresponding to unique values (h/s/o/so)
tiepos.integer64	tiepos	positions of values that are tied (//o/so)
keypos.integer64	keypos	position of current value in sorted list of unique values (//o/so)
as.factor.integer64	as.factor	convert to (unordered) factor with sorted levels of previous values (/
as.ordered.integer64	as.ordered	convert to ordered factor with sorted levels of previous values (//o/s
table.integer64	table	unique values and their frequencies (h/s/o/so)
sort.integer64	sort	sorted vector (/s/o/so)
order.integer64	order	positions of elements that would create sorted vector (//o/so)
rank.integer64	rank	(average) ranks of non-NAs, NAs kept in place (/s/o/so)
quantile.integer64	quantile	(existing) values at specified percentiles (/s/o/so)
median.integer64	median	(existing) value at percentile 0.5 (/s/o/so)
summary.integer64	summary	(/s/o/so)
all.equal.integer64	all.equal	test if two objects are (nearly) equal (/s/o/so)

helper functions	**see also**	**description**
minusclass	minusclass	removing class attritbute
plusclass	plusclass	inserting class attribute
binattr	binattr	define binary op behaviour

tested I/O functions	**see also**	**description**
	`read.table`	inherited from Base R
	`write.table`	inherited from Base R
	`serialize`	inherited from Base R
	`unserialize`	inherited from Base R
	`save`	inherited from Base R
	`load`	inherited from Base R
	`dput`	inherited from Base R
	`dget`	inherited from Base R

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 `123LL` 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 desireable 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(integer64,integer64))
```

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 `NA`s is currently not supported. Such subscripting currently returns `9218868437227407266` 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.c

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           # 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(I(y)))
d <- data.frame(x=x, y=runif(length(x), 0, 100))
d
d$x

message("Using integer64 with csv files")
fi64 <- tempfile()
write.csv(d, file=fi64, row.names=FALSE)
e <- read.csv(fi64, colClasses=c("integer64", NA))
unlink(fi64)
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
#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) --")
td64 <- 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
t64/td64


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))
t32<- system.time(for(i in 1:1000)i32[sample(1e6, 1e3)])
i64 <- as.integer64(i32);
t64 <- system.time(for(i in 1:1000)i64[sample(1e6, 1e3)])
rm(i64); gc()
#I64 <- as.int64(i32);
#T64 <- system.time(for(i in 1:100)I64[sample(1e6, 1e3)])*10
#rm(I64); gc()
#(T64-ts32)/(t64-ts32)
(t64-ts32)/(t32-ts32)

message("-- integer64 assigns 200x/90x faster than int64
(and 50x/160x slower than integer) --")
ts32 <- system.time(for(i in 1:100)sample(1e6, 1e3))
t32 <- system.time(for(i in 1:100)i32[sample(1e6, 1e3)] <- 1:1e3)
i64 <- as.integer64(i32);
i64 <- system.time(for(i in 1:100)i64[sample(1e6, 1e3)] <- 1:1e3)
rm(i64); gc()
#I64 <- as.int64(i32);
#I64 <- system.time(for(i in 1:10)I64[sample(1e6, 1e3)] <- 1:1e3)*10
#rm(I64); gc()
#(T64-ts32)/(t64-ts32)
(t64-ts32)/(t32-ts32)


tdfi32 <- system.time(dfi32 <- data.frame(a=i32, b=i32, c=i32))
```

```
tdfsi32 <- system.time(dfi32[1e6:1,])
fi32 <- tempfile()
tdfwi32 <- system.time(write.csv(dfi32, file=fi32, row.names=FALSE))
tdfri32 <- system.time(read.csv(fi32, colClasses=rep("integer", 3)))
unlink(fi32)
rm(dfi32); gc()

i64 <- as.integer64(i32);
tdfi64 <- system.time(dfi64 <- data.frame(a=i64, b=i64, c=i64))
tdfsi64 <- system.time(dfi64[1e6:1,])
fi64 <- tempfile()
tdfwi64 <- system.time(write.csv(dfi64, file=fi64, row.names=FALSE))
tdfri64 <- system.time(read.csv(fi64, colClasses=rep("integer64", 3)))
unlink(fi64)
rm(i64, dfi64); gc()

#I64 <- as.int64(i32);
#tdfI64 <- system.time(dfI64<-data.frame(a=I64, b=I64, c=I64))
#tdfsI64 <- system.time(dfI64[1e6:1,])
#fI64 <- tempfile()
#tdfwI64 <- system.time(write.csv(dfI64, file=fI64, row.names=FALSE))
#tdfrI64 <- system.time(read.csv(fI64, colClasses=rep("int64", 3)))
#unlink(fI64)
#rm(I64, 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) --")
#tdfsI64/tdfsi64
tdfsi64/tdfsi32
message("-- integer64 csv writes about 2x/0.5x faster than int64
(and about 1.5x/5x slower than integer) --")
#tdfwI64/tdfwi64
tdfwi64/tdfwi32
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 gargbage 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")

## End(Not run)
```

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

Description

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

Usage

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

Arguments

`target`	a vector of 'integer64' or an object that can be coerced with `as.integer64`
`current`	a vector of 'integer64' or an object that can be coerced with `as.integer64`
`tolerance`	numeric ≥ 0. Differences smaller than `tolerance` are not reported. The default value is close to `1.5e-8`.
`scale`	`NULL` or numeric > 0, typically of length 1 or `length(target)`. See 'Details'.

countEQ
: logical indicating if the `target == current` cases should be counted when computing the mean (absolute or relative) differences. The default, `FALSE` may seem misleading in cases where `target` and `current` only differ in a few places; see the extensive example.

formatFUN
: a `function` of two arguments, `err`, the relative, absolute or scaled error, and `what`, a character string indicating the *kind* of error; maybe used, e.g., to format relative and absolute errors differently.

...
: further arguments are ignored

check.attributes
: logical indicating if the `attributes` of `target` and `current` (other than the names) should be compared.

Details

In `all.equal.numeric` the type `integer` is treated as a proper subset of `double` i.e. does not complain about comparing `integer` with `double`. Following this logic `all.equal.integer64` treats `integer` as a proper subset of `integer64` and does not complain about comparing `integer` with `integer64`. `double` also compares without warning as long as the values are within `lim.integer64`, if `double` are bigger `all.equal.integer64` complains about the `all.equal.integer64 overflow warning`. For further details see `all.equal`.

Value

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

Note

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

Author(s)

Leonardo Silvestri (for package nanotime)

See Also

`all.equal`

Examples

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

`as.character.integer64` *Coerce from integer64*

Description

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

Usage

```
 as.bitstring(x, ...)
 ## S3 method for class 'integer64'
as.bitstring(x, ...)
 ## S3 method for class 'bitstring'
print(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)
 ## S3 method for class 'integer64'
as.list(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 class 'bitstring'.
The other methods return atomic vectors of the expected types

Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

See Also

as.integer64.character integer64

Examples

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

---

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

Description

Coercing integer64 vector to data.frame.

Usage

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

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

```
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

```
NA_integer64_
as.integer64(x, ...)
## 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 'bitstring'
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'. `as.integer64.bitstring` evaluates characters '0' anbd ' ' as zero-bit, all other one byte characters as one-bit, multi-byte characters are not allowed, strings shorter than 64 characters are treated as if they were left-padded with '0', strings longer than 64 bytes are mapped to `NA_INTEGER64` and a warning is emitted.

Value

The other methods return atomic vectors of the expected types

Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

See Also

`as.character.integer64` `integer64`

Examples

```
as.integer64(as.character(lim.integer64()))
as.integer64(
  structure(c("1111111111111111111111111111111111111111111111111111111111111110",
              "1111111111111111111111111111111111111111111111111111111111111111",
              "1000000000000000000000000000000000000000000000000000000000000000",
              "0000000000000000000000000000000000000000000000000000000000000000",
              "0000000000000000000000000000000000000000000000000000000000000001",
              "0000000000000000000000000000000000000000000000000000000000000010"
  ), class = "bitstring")
)
as.integer64(
 structure(c("............................................................... ",
             "................................................................",
             ".                                                               ",
             "",
             ".",
             "10"
  ), class = "bitstring")
)
```

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

Description

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

```
benchmark64(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 sesssion.
`optimizer64` returns a dimensioned list with one row for each high-level function timed and two columns named after the values of the `nsmall` and `nbig` sample sizes. Each list cell contains a matrix with timings, low-level-methods in rows and three measurements `c("prep","both","use")` in columns. If it can be measured separately, `prep` contains the timing of preparatory work such as sorting and hashing, and `use` contains the timing of using the prepared work. If the function timed does both, preparation and use, the timing is in `both`.

Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

See Also

`integer64`

Examples

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

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

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

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

Description

These are the results of calling `benchmark64`

Usage

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

Format

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

Examples

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

---

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

Description

Turn those base functions S3 generic which are used in bit64

Usage

```
from:to
 #--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 gernerics 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 ususal functions 'c', 'cbind' and 'rbind'

Usage

```
## 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

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

Arguments

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

Details

A cache is an link{environment} attached to an atomic object with the link{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 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

Value

see details

Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

See Also

still.identical for testing whether to symbols point to the same RAM.
Functions that get and set small cache-content automatically when a cache is present: 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

```
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
cache(x)
getcache(x, "abc")
setcache(x, "abc", 1)
```

```
getcache(x, "abc")
remcache(x)
cache(x)
```

---

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

Description

Cumulative Sums, Products, Extremes and lagged differences

Usage

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

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

---

duplicated.integer64 *Determine Duplicate Elements of integer64*

Description

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

Usage

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

Arguments

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

Details

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

Suitable methods are `hashdup` (hashing), `sortorderdup` (fast ordering) 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

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

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

---

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

Description

Methods to extract and replace parts of an integer64 vector.

Usage

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

Arguments

x	an atomic vector
i	indices specifying elements to extract
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

Examples

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

---

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

Description

Unary operators and functions for integer64 vectors.

Usage

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

```
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 `NA`s. If we know we don't have NAs, this may speed-up. *Note* that you risk a crash if there are unexpected `NA`s 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.integer64 for methods benefiting from small caches

Examples

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

---

hashmap *Hashing for 64bit integers*

Description

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

Usage

```
hashfun(x, ...)
## S3 method for class 'integer64'
hashfun(x, minfac=1.41, hashbits=NULL, ...)
hashmap(x, ...)
## S3 method for class 'integer64'
hashmap(x, nunique=NULL, minfac=1.41, hashbits=NULL, cache=NULL, ...)
hashpos(cache, ...)
## S3 method for class 'cache_integer64'
hashpos(cache, x, nomatch = NA_integer_, ...)
hashrev(cache, ...)
## S3 method for class 'cache_integer64'
hashrev(cache, x, nomatch = NA_integer_, ...)
hashfin(cache, ...)
## S3 method for class 'cache_integer64'
hashfin(cache, x, ...)
hashrin(cache, ...)
## S3 method for class 'cache_integer64'
hashrin(cache, x, ...)
hashdup(cache, ...)
## S3 method for class 'cache_integer64'
hashdup(cache, ...)
hashuni(cache, ...)
## S3 method for class 'cache_integer64'
hashuni(cache, keep.order=FALSE, ...)
hashmapuni(x, ...)
## S3 method for class 'integer64'
hashmapuni(x, nunique=NULL, minfac=1.5, hashbits=NULL, ...)
hashupo(cache, ...)
```

```
## S3 method for class 'cache_integer64'
hashupo(cache, keep.order=FALSE, ...)
hashmapupo(x, ...)
## S3 method for class 'integer64'
hashmapupo(x, nunique=NULL, minfac=1.5, hashbits=NULL, ...)
hashtab(cache, ...)
## S3 method for class 'cache_integer64'
hashtab(cache, ...)
hashmaptab(x, ...)
## S3 method for class 'integer64'
hashmaptab(x, nunique=NULL, minfac=1.5, hashbits=NULL, ...)
```

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 `x`, ignored if `hashbits` is given directly
`hashbits`	length of hashmap is `2^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

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

Value

see details

Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

See Also

match, runif64

Examples

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

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[hashupo(hy, keep.order=FALSE)], hashuni(hy, keep.order=FALSE)))
stopifnot(identical(y[hashupo(hy, keep.order=TRUE)], hashuni(hy, keep.order=TRUE)))
stopifnot(identical(hashpos(hy, hashuni(hy, keep.order=TRUE)), hashupo(hy, keep.order=TRUE)))
stopifnot(identical(hashpos(hy, hashuni(hy, keep.order=FALSE)), hashupo(hy, keep.order=FALSE)))
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(hashupo(hy, keep.order=TRUE), hashmapupo(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")
bits <- 24
b <- seq(-1, 0, 0.1)
tim <- matrix(NA, length(b), 2, dimnames=list(b, c("bits","bits+1")))
    for (i in 1:length(b)){
  n <- as.integer(2^(bits+b[i]))
  x <- as.integer64(sample(n))
  tim[i,1] <- repeat.time(hashmap(x, hashbits=bits))[3]
  tim[i,2] <- repeat.time(hashmap(x, hashbits=bits+1))[3]
  print(tim)
      matplot(b, tim)
}
message("we conclude that n*sqrt(2) is enough to avoid collisions")

## End(Not run)
```

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

Description

This will discover any deviation between objects containing integer64 vectors.

Usage

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

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`
`ignore.environment`	see `identical`
`ignore.srcref`	see `identical`
`extptr.as.ref`	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

`==.integer64` `identical` `integer64`

Examples

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

---

is.sorted.integer64 *Small cache access methods*

Description

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

Usage

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

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 `NA`s
`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.Oehlschlaegel@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)
 nties(x)
 table.integer64(x)
 x
```

---

keypos *Extract Positions in redundant dimension table*

Description

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

Usage

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

Arguments

x	a vector or a data frame or an array or NULL.
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 x containing positions relativ to sort(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

```
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))))
```

---

match.integer64 *64-bit integer matching*

Description

match returns a vector of the positions of (first) matches of its first argument in its second.

%in% is a more intuitive interface as a binary operator, which returns a logical vector indicating if there is a match or not for its left operand.

Usage

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

Arguments

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

Details

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

Suitable methods for %in%.integer64 are hashpos (hash table lookup), hashrev (reverse lookup), sortorderpos (fast ordering) 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 table[j] then the value returned in the i-th position of the return value is j, for the smallest possible j. If no match is found, the value is nomatch.

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

Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

See Also

match

Examples

```
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)))

## 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(ti)if(any(is.na(ti)))NA else which.min(ti)))
}

## 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" $: num [1:10, 1:3] 0 0 0 1.62 0.00114 ... ..- attr(*, "dimnames")=List of 2 .. ..$: chr [1:10] "%in%" "match.64" "%in%.64" "hashfin" ... .$: chr [1:3] "prep" "both" "use" $: num [1:10, 1:3] 0 0 0.00105 0.00313 0.00313 ... ..- attr(*, "dimnames")=List of 2 .. ..$: chr [1:10] "duplicated" "duplicated.64" "hashdup" "sortorderdup1" ... .$: chr [1:3] "prep" "both" "use" $: num [1:15, 1:3] 0 0 0 0.00104 0.00104 ... ..- attr(*, "dimnames")=List of 2 .. ..$: chr [1:15] "unique" "unique.64" "hashmapuni" "hashuni" ... .$: chr [1:3] "prep" "both" "use" $: num [1:14, 1:3] 0 0 0 0.000992 0.000992 ... ..- attr(*, "dimnames")=List of 2 .. ..$: chr [1:14] "unique" "unipos.64" "hashmapupo" "hashupo" ... .$: chr [1:3] "prep" "both" "use" $: num [1:13, 1:3] 0 0 0 0 0.000419 ... ..- attr(*, "dimnames")=List of 2 .. ..$: chr [1:13] "tabulate" "table" "table.64" "hashmaptab" ... .$: chr [1:3] "prep" "both" "use" $: num [1:7, 1:3] 0 0 0 0.00236 0.00714 ... ..- attr(*, "dimnames")=List of 2 .. ..$: chr [1:7] "rank" "rank.keep" "rank.64" "sortorderrnk" ... .$: chr [1:3] "prep" "both" "use" $: num [1:6, 1:3] 0 0 0.00189 0.00714 0 ... ..- attr(*, "dimnames")=List of 2 .. ..$: chr [1:6] "quantile" "quantile.64" "sortqtl" "orderqtl" ... .$: chr [1:3] "prep" "both" "use" $: num [1:9, 1:3] 0 0 0.00105 1.17 0 ... ..- attr(*, "dimnames")=List of 2 .. ..$: chr [1:9] "match" "match.64" "hashpos" "hashrev" ... .$: chr [1:3] "prep" "both" "use" $: num [1:10, 1:3] 0 0 0 0.00104 1.18 ... ..- attr(*, "dimnames")=List of 2 .. ..$: chr [1:10] "%in%" "match.64" "%in%.64" "hashfin" ... .$: chr [1:3] "prep" "both" "use" $: num [1:10, 1:3] 0 0 1.64 2.48 2.48 ... ..- attr(*, "dimnames")=List of 2 .. ..$: chr [1:10] "duplicated" "duplicated.64" "hashdup" "sortorderdup1" ... .$: chr [1:3] "prep" "both" "use" $: num [1:15, 1:3] 0 0 0 1.64 1.64 ... ..- attr(*, "dimnames")=List of 2 .. ..$: chr [1:15] "unique" "unique.64" "hashmapuni" "hashuni" ... .$: chr [1:3] "prep" "both" "use" $: num [1:14, 1:3] 0 0 0 1.62 1.62 ... ..- attr(*, "dimnames")=List of 2 .. ..$: chr [1:14] "unique" "unipos.64" "hashmapupo" "hashupo" ... .$: chr [1:3] "prep" "both" "use" $: num [1:13, 1:3] 0 0 0 0 0.32 ... ..- attr(*, "dimnames")=List of 2 .. ..$: chr [1:13] "tabulate" "table" "table.64" "hashmaptab" ... .$: chr [1:3] "prep" "both" "use" $: num [1:7, 1:3] 0 0 0 2.96 10.69 ... ..- attr(*, "dimnames")=List of 2 .. ..$: chr [1:7] "rank" "rank.keep" "rank.64" "sortorderrnk" ... .$: chr [1:3] "prep" "both" "use" $: num [1:6, 1:3] 0 0 1.62 10.61 0 ... ..- attr(*, "dimnames")=List of 2 .. ..$: chr [1:6] "quantile" "quantile.64" "sortqtl" "orderqtl" ... .$: chr [1:3] "prep" "both" "use" - attr(*, "dim")= int [1:2] 8 2 - attr(*, "dimnames")=List of 2 ..$: chr [1:8] "match" "%in%" "duplicated" "unique" ... ..$: chr [1:2] "65536" "33554432"

Examples

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

---

prank *(P)ercent (Rank)s*

Description

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

Usage

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

Arguments

x	a integer64 vector
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.Oehlschlaegel@truecluster.com>

See Also

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

Examples

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

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

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

Description

Function `prank.integer64` projects the values [min..max] via ranks [1..n] to [0..1]. `qtile.ineger64` is the inverse function of 'prank.integer64' and projects [0..1] to [min..max].

Usage

```
qtile(x, probs=seq(0, 1, 0.25), ...)
## S3 method for class 'integer64'
qtile(x, probs = seq(0, 1, 0.25), names = TRUE, method = NULL, ...)
## S3 method for class 'integer64'
quantile(x, probs = seq(0, 1, 0.25), na.rm = FALSE, names = TRUE, type=0L, ...)
## S3 method for class 'integer64'
median(x, na.rm = FALSE, ...)
 ## S3 method for class 'integer64'
mean(x, na.rm = FALSE, ...)
## S3 method for class 'integer64'
summary(object, ...)
 ## mean(x, na.rm = FALSE, ...)
 ## or
 ## mean(x, na.rm = FALSE)
```

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 symetric 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.

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 intger64 methods for in-RAM sorting and ordering*

Description

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

Usage

```
## S3 method for class 'integer64'
shellsort(x, has.na=TRUE, na.last=FALSE, decreasing=FALSE, ...)
## S3 method for class 'integer64'
shellsortorder(x, i, has.na=TRUE, na.last=FALSE, decreasing=FALSE, ...)
## S3 method for class 'integer64'
shellorder(x, i, has.na=TRUE, na.last=FALSE, decreasing=FALSE, ...)
## S3 method for class 'integer64'
mergesort(x, has.na=TRUE, na.last=FALSE, decreasing=FALSE, ...)
## S3 method for class 'integer64'
mergeorder(x, i, has.na=TRUE, na.last=FALSE, decreasing=FALSE, ...)
## S3 method for class 'integer64'
mergesortorder(x, i, has.na=TRUE, na.last=FALSE, decreasing=FALSE, ...)
## S3 method for class 'integer64'
quicksort(x, has.na=TRUE, na.last=FALSE, decreasing=FALSE
, restlevel=floor(1.5*log2(length(x))), ...)
## S3 method for class 'integer64'
quicksortorder(x, i, has.na=TRUE, na.last=FALSE, decreasing=FALSE
, restlevel=floor(1.5*log2(length(x))), ...)
## S3 method for class 'integer64'
quickorder(x, i, has.na=TRUE, na.last=FALSE, decreasing=FALSE
, restlevel=floor(1.5*log2(length(x))), ...)
## S3 method for class 'integer64'
radixsort(x, has.na=TRUE, na.last=FALSE, decreasing=FALSE, radixbits=8L, ...)
## S3 method for class 'integer64'
radixsortorder(x, i, has.na=TRUE, na.last=FALSE, decreasing=FALSE, radixbits=8L, ...)
## S3 method for class 'integer64'
radixorder(x, i, has.na=TRUE, na.last=FALSE, decreasing=FALSE, radixbits=8L, ...)
## S3 method for class 'integer64'
ramsort(x, has.na = TRUE, na.last=FALSE, decreasing = FALSE, stable = TRUE
, optimize = c("time", "memory"), VERBOSE = FALSE, ...)
## S3 method for class 'integer64'
ramsortorder(x, i, has.na = TRUE, na.last=FALSE, decreasing = FALSE, stable = TRUE
, optimize = c("time", "memory"), VERBOSE = FALSE, ...)
## S3 method for class 'integer64'
ramorder(x, i, has.na = TRUE, na.last=FALSE, decreasing = FALSE, stable = TRUE
, optimize = c("time", "memory"), VERBOSE = FALSE, ...)
```

Arguments

`x`	a vector to be sorted by `ramsort.integer64` and `ramsortorder.integer64`, i.e. the output of `sort.integer64`
`i`	integer positions to be modified by `ramorder.integer64` and `ramsortorder.integer64`, default is 1:n, in this case the output is similar to `order.integer64`
`has.na`	boolean scalar defining whether the input vector might contain `NA`s. If we know we don't have NAs, this may speed-up. *Note* that you risk a crash if there are unexpected `NA`s with `has.na=FALSE`
`na.last`	boolean scalar telling ramsort whether to sort `NA`s 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 recursionlevels 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 `NA`s found or assumed during sorting

Note

Note that these methods purposely violate the functional programming paradigm: they are called for the side-effect of changing some of their arguments. The `sort`-methods change `x`, the `order`-methods change `i`, and the `sortoder`-methods change both `x` and `i`

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

```
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

```
## 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

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

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

---

rep.integer64 *Replicate elements of integer64 vectors*

Description

Replicate elements of integer64 vectors

Usage

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

Arguments

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

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)
```

`runif64` *integer64: random numbers*

Description

Create uniform random 64-bit integers within defined range

Usage

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

Arguments

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

Details

For each random integer we call R's internal C interface `unif_rand()` twice. Each call is mapped to 2^32 unsigned integers. The two 32-bit patterns are concatenated to form the new integer64. This process is repeated until the result is not a `NA_INTEGER64`.

Value

a integer64 vector

Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

See Also

`runif`, `hashfun`

Examples

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

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

`seq.integer64` *integer64: Sequence Generation*

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

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 intger64 methods for sorting and ordering*

Description

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

Usage

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

Arguments

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

```
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 hasmap. Since sorting is more informative than hashingwe can do some more interesting things.

Usage

```
sortnut(sorted, ...)
ordernut(table, order, ...)
sortfin(sorted, x, ...)
orderfin(table, order, x, ...)
orderpos(table, order, x, ...)
sortorderpos(sorted, order, x, ...)
orderdup(table, order, ...)
sortorderdup(sorted, order, ...)
sortuni(sorted, nunique, ...)
orderuni(table, order, nunique, ...)
sortorderuni(table, sorted, order, nunique, ...)
orderupo(table, order, nunique, ...)
```

```
sortorderupo(sorted, order, nunique, keep.order = FALSE, ...)
ordertie(table, order, nties, ...)
sortordertie(sorted, order, nties, ...)
sorttab(sorted, nunique, ...)
ordertab(table, order, nunique, ...)
sortordertab(sorted, order, ...)
orderkey(table, order, na.skip.num = 0L, ...)
sortorderkey(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'
orderuni(table, order, nunique, keep.order=FALSE, ...)
## S3 method for class 'integer64'
sortorderuni(table, sorted, order, nunique, ...)
## S3 method for class 'integer64'
orderupo(table, order, nunique, keep.order=FALSE, ...)
## S3 method for class 'integer64'
sortorderupo(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'
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 `NA`s. With 0, `NA`s are coded with 1L, with the number of `NA`s, these are coded with `NA`, the latter needed for `as.factor.integer64`
`na.count`	the number of `NA`s, needed for this low-level function algorithm
`method`	see details
`...`	further arguments, passed from generics, ignored in methods

Details

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

	orderrnk	sortorderrnk	rank averaging ties
sortqtl	orderqtl		return quantiles given probabilities

The functions `sortfin`, `orderfin`, `orderpos` and `sortorderpos` each offer three algorithms for finding `x` in `table`.
With `method=1L` each value of `x` is searched independently using *binary search*, this is fastest for small `tables`.
With `method=2L` the values of `x` are first sorted and then searched using *doubly exponential search*, this is the best 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

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

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

Description

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

Usage

```
## S3 method for class 'integer64'
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, finite = 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
`finite`	logical scalar indicating whether to ignore NAs (just for compatibility with `range.default`)

Details

The numerical summary methods always return `integer64`. Therefor 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 `NA`s 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

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

---

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

Description

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

Usage

```
table.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 rewquired 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

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

```
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)))]))
```

`unipos` *Extract Positions of Unique Elements*

Description

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

Usage

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

Arguments

`x`	a vector or a data frame or an array or `NULL`.
`incomparables`	ignored
`order`	The order in which positions of unique values will be returned, see details
`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

```
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")]))
```

`unique.integer64` *Extract Unique Elements from integer64*

Description

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

Usage

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

Arguments

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

Details

This function automatically chooses from several low-level functions considering the size of x and the availability of a cache. Suitable methods are hashmapuni (simultaneously creating and using a hashmap) , hashuni (first creating a hashmap then using it) , sortuni (fast sorting for sorted order only) , sortorderuni (fast ordering for original order only) 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

```
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))))
```

xor.integer64 *Binary operators for integer64 vectors*

Description

Binary operators for integer64 vectors.

Usage

```
## 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'
e1 %% e2
binattr(e1,e2) # for internal use only
```

Arguments

e1	an atomic vector of class 'integer64'
e2	an atomic vector of class '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
options(integer64_semantics="new")
d <- 2.5
i <- as.integer64(5)
d/i  # new 0.5
d*i  # new 13
i*d  # new 13
options(integer64_semantics="old")
d/i  # old: 0.4
d*i  # old: 10
i*d  # old: 13
```

Index