Package 'fastmatch'

December 20, 2024

Version 1.1-6

Title Fast 'match()' Function

Author Simon Urbanek [aut, cre, cph]
(https://urbanek.org, <https://orcid.org/0000-0003-2297-1732>)

Maintainer Simon Urbanek <simon.urbanek@r-project.org>

Description Package providing a fast match() replacement for cases that require repeated look-ups. It is slightly faster that R's built-in match() function on first match against a table, but extremely fast on any subsequent lookup as it keeps the hash table in memory.

License GPL-2

Depends R (>= 2.3.0)

URL http://www.rforge.net/fastmatch

NeedsCompilation yes

Contents

`coalesce` *Create an index that groups unique values together*

Description

`coalesce` makes sure that a given index vector is coalesced, i.e., identical values are grouped into contiguous blocks. This can be used as a much faster alternative to `sort.list` where the goal is to group identical values, but not necessarily in a pre-defined order. The algorithm is linear in the length of the vector.

Usage

```
coalesce(x)
```

Arguments

x	character, integer or real vector to coalesce

Details

The current implementation takes two passes through the vector. In the first pass it creates a hash table for the values of x counting the occurrences in the process. In the second pass it assigns indices for every element based on the index stored in the hash table.

The order of the groups of unique values is defined by the first occurence of each unique value, hence it is identical to the order of unique.

One common use of coalesce is to allow the use of arbitrary vectors in ctapply via ctapply(x[coalesce(x)] ...).

Value

Integer vector with the resulting permutation. x[coalesce(x)] gives x with contiguous unique values.

Author(s)

Simon Urbanek

See Also

unique, sort.list, ctapply

Examples

```
i = rnorm(2e6)
names(i) = as.integer(rnorm(2e6))
## compare sorting and coalesce
system.time(o <- i[order(names(i))])
system.time(o <- i[coalesce(names(i))])

## more fair comparison taking the coalesce time (and copy) into account
system.time(tapply(i, names(i), sum))
system.time({ o <- i[coalesce(names(i))]; ctapply(o, names(o), sum) })

## in fact, using ctapply() on a dummy vector is faster than table() ...
## believe it or not ... (that that is actually wasteful, since coalesce
## already computed the table internally anyway ...)
ftable <- function(x) {
   t <- ctapply(rep(0L, length(x)), x[coalesce(x)], length)
   t[sort.list(names(t))]
}
system.time(table(names(i)))
system.time(ftable(names(i)))
```

ctapply *Fast tapply() replacement functions*

Description

ctapply is a fast replacement of tapply that assumes contiguous input, i.e. unique values in the index are never speparated by any other values. This avoids an expensive split step since both value and the index chungs can be created on the fly. It also cuts a few corners to allow very efficient copying of values. This makes it many orders of magnitude faster than the classical lapply(split(), ...) implementation.

Usage

```
ctapply(X, INDEX, FUN, ..., MERGE=c)
```

Arguments

X	an atomic object, typically a vector
INDEX	numeric or character vector of the same length as X
FUN	the function to be applied
...	additional arguments to FUN. They are passed as-is, i.e., without replication or recycling
MERGE	function to merge the resulting vector or NULL if the arguments to such a functiona re to be returned instead

Details

Note that ctapply supports either integer, real or character vectors as indices (note that factors are integer vectors and thus supported, but you do not need to convert character vectors). Unlike tapply it does not take a list of factors - if you want to use a cross-product of factors, create the product first, e.g. using paste(i1, i2, i3, sep='\01') or multiplication - whetever method is convenient for the input types.

ctapply requires the INDEX to contiguous. One (slow) way to achieve that is to use sort or order.

Author(s)

Simon Urbanek

See Also

tapply

Examples

```
i = rnorm(4e6)
names(i) = as.integer(rnorm(1e6))
i = i[order(names(i))]
system.time(tapply(i, names(i), sum))
system.time(ctapply(i, names(i), sum))
```

`fmatch` *Fast match() replacement*

Description

`fmatch` is a faster version of the built-in `match`() function. It is slightly faster than the built-in version because it uses more specialized code, but in addition it retains the hash table within the table object such that it can be re-used, dramatically reducing the look-up time especially for large tables.

Although `fmatch` can be used separately, in general it is also safe to use: `match <- fmatch` since it is a drop-in replacement. Any cases not directly handled by `fmatch` are passed to `match` with a warning.

`fmatch.hash` is identical to `fmatch` but it returns the table object with the hash table attached instead of the result, so it can be used to create a table object in cases where direct modification is not possible.

`%fin%` is a version of the built-in `%in%` function that uses `fmatch` instead of `match`().

Usage

```
fmatch(x, table, nomatch = NA_integer_, incomparables = NULL)
fmatch.hash(x, table, nomatch = NA_integer_, incomparables = NULL)
x %fin% table
```

Arguments

`x`	values to be matched
`table`	values to be matched against
`nomatch`	the value to be returned in the case when no match is found. It is coerced to `integer`.
`incomparables`	a vector of values that cannot be matched. Any value other than `NULL` will result in a fall-back to `match` without any speed gains.

Details

See `match` for the purpose and details of the `match` function. `fmatch` is a drop-in replacement for the `match` function with the focus on performance. `incomparables` are not supported by `fmatch` and will be passed down to `match`.

The first match against a table results in a hash table to be computed from the table. This table is then attached as the `".match.hash"` attribute of the table so that it can be re-used on subsequent calls to `fmatch` with the same table.

The hashing algorithm used is the same as the `match` function in R, but it is re-implemented in a slightly different way to improve its performance at the cost of supporting only a subset of types (integer, real and character). For any other types `fmatch` falls back to `match` (with a warning).

Value

`fmatch`: A vector of the same length as `x` - see `match` for details.

`fmatch.hash`: `table`, possibly coerced to match the type of `x`, with the hash table attached.

`%fin%`: A logical vector the same length as `x` - see `%in%` for details.

Note

`fmatch` modifies the `table` by attaching an attribute to it. It is expected that the values will not change unless that attribute is dropped. Under normal circumstances this should not have any effect from user's point of view, but there is a theoretical chance of the cache being out of sync with the table in case the table is modified directly (e.g. by some C code) without removing attributes.

In cases where the `table` object cannot be modified (or such modification would not survive) `fmatch.hash` can be used to build the hash table and return `table` object including the hash table. In that case no lookup is done and `x` is only used to determine the type into which `table` needs to be coerced.

Also `fmatch` does not convert to a common encoding so strings with different representation in two encodings don't match.

Author(s)

Simon Urbanek

See Also

`match`

Examples

```
# some random speed comparison examples:
# first use integer matching
x = as.integer(rnorm(1e6) * 1000000)
s = 1:100
# the first call to fmatch is comparable to match
system.time(fmatch(s,x))
# but the subsequent calls take no time!
system.time(fmatch(s,x))
system.time(fmatch(-50:50,x))
system.time(fmatch(-5000:5000,x))
# here is the speed of match for comparison
system.time(base::match(s, x))
# the results should be identical
identical(base::match(s, x), fmatch(s, x))

# next, match a factor against the table
# this will require both x and the factor
# to be cast to strings
s = factor(c("1","1","2","foo","3",NA))
# because the casting will have to allocate a string
# cache in R, we run a dummy conversion to take
# that out of the equation
dummy = as.character(x)
# now we can run the speed tests
system.time(fmatch(s, x))
system.time(fmatch(s, x))
# the cache is still valid for string matches as well
system.time(fmatch(c("foo","bar","1","2"),x))
# now back to match
system.time(base::match(s, x))
identical(base::match(s, x), fmatch(s, x))

# finally, some reals to match
```

```
y = rnorm(1e6)
s = c(y[sample(length(y), 100)], 123.567, NA, NaN)
system.time(fmatch(s, y))
system.time(fmatch(s, y))
system.time(fmatch(s, y))
system.time(base::match(s, y))
identical(base::match(s, y), fmatch(s, y))

# this used to fail before 0.1-2 since nomatch was ignored
identical(base::match(4L, 1:3, nomatch=0), fmatch(4L, 1:3, nomatch=0))
```

Index