Usage of the bit package

Dr. Jens Oehlschlägel

2024-09-19

Contents

Boolean data types 1
- Available classes 2
- Available methods 2
- Creating and manipulating 3
- Coercion 5
- Boolean operations 7
- Manipulation methods 8
- Aggregation methods for `booltype` 9

Fast methods for `integer` set operations 10
- Methods using random access to bit vectors 11
- Methods using bit vectors for sorting integers 14
- Methods for sets of sorted integers 15

The bit package provides the following S3 functionality:

- alternatives to the `logical` data type which need less memory with methods that are often faster
  - the `bit` type for boolean data (without `NA`, factor 32 smaller)
  - the `bitwhich` type for very skewed boolean data (without `NA`, potentially even smaller)
- further alternatives for representing Boolean selections
  - the `which` type for positive selections maintaining the original vector length
  - the `ri` range index
- very fast methods for `integer`, particularly
  - methods for *unsorted* integers leveraging `bit` vectors rather than *hash* tables
  - methods for *sorting* integers leveraging $O(N)$ *synthetic* sorting rather than $O(N \log N)$ divide and conquer
  - methods for *sorted* integers leveraging *merging* rather than *hash* tables
- some foundations for package `ff`, particularly
  - `bit` and `bitwhich` vectors and `ri` range indices for filtering and subscripting `ff` objects
  - `rlepack` compressing sets of *sorted* integers for `hi` hybrid indexing
  - methods for *chunking*
- helper methods for avoiding unwanted data copies

Boolean data types

R's `logical` vectors cost 32 bit per element in order to code three states `FALSE`, `TRUE` and `NA`. By contrast `bit` vectors cost only 1 bit per element and support only the two Boolean states `FALSE` and `TRUE`. Internally `bit` are stored as `integer`. `bitwhich` vectors manage very skewed Boolean data as a set of sorted unique integers denoting either *included* or *excluded* positions, whatever costs less memory. Internally `bitwhich` are stored as `integer` *or* as `logical` for the extreme cases *all included* (`TRUE`), *all excluded* (`FALSE`) or *zero length* (`logical()`). Function `bitwhich_representation` allows to distinguish these five cases without the copying-cost of `unclass(bitwhich)`. All three Boolean types `logical`, `bit`, and `bitwhich` are unified in a super-class `booltype` and have `is.booltype(x)==TRUE`. Classes `which` and `ri` can be somewhat considered as a fourth and fifth even more special Boolean types (for skewed data with mostly `FALSE` that represents the sorted positions of the few `TRUE`, and as a consecutive series of the latter), and hence have `booltype 4` and `5` but coerce differently to `integer` and `double` and hence are not `is.booltype`.

Available classes

The `booltypes` `bit` and `bitwhich` behave very much like `logical` with the following exceptions

- length is limited to `.Machine$integer.max`
- currently only `vector` methods are supported, not `matrix` or `array`
- subscripting from them does return `logical` rather than returning their own class (like `ff boolean` vectors do)
- values assigned to them are internally coerced to `logical` before being processed, hence `x[] <- x` is a potentially expensive operation
- they do not support `names` and `character` subscripts
- only scalar `logical` subscripts are allowed, i.e. `FALSE`, `TRUE`
- assigned or coerced `NA` are silently converted to `FALSE`
- increasing the `length` of `bit` has semantics differing from `logical`: new elements are consistently initialized with `FALSE` instead of `NA`
- increasing the `length` of `bitwhich` has semantics differing from `logical` and `bit`: new elements are initialized with the *more frequent value* of the skewed distribution (`FALSE` winning in the equally frequent case).
- aggregation methods `min`, `max` and `range` and `summary` have special meaning, for example `min` corresponds to `which.max`.

Note the following features

- aggregation functions support a `range` argument for *chunked processing*
- sorted positive subscripts marked as class `which` are processed faster than 'just positive subscripts', and unlike the result of the function `which`, class `which` retains the length of the Boolean vector as attribute `maxindex`, but its `length` is the number of positive positions
- `ri` range indices are allowed as subscripts which supports *chunked looping* over in-memory `bit` or `bitwhich` vectors not unlike *chunked looping* over on-disk `ff` objects.
- binary Boolean operations will promote differing data type to the data type implying fewer assumptions, i.e. `logical` wins over `bit`, `bit` wins over `bitwhich` and `which` wins over `bitwhich`.

Note the following warnings

- currently `bit` and `bitwhich` may answer `is.logical` and `is.integer` according to their internal representation. Do not rely on this, it may be subject to change. Do use `booltype`, `is.booltype`, `is.bit`, `is.bitwhich` and `bitwhich_representation` for reasoning about the data type.

Available methods

Basic methods

`is as length length<- [ [<- [[ [[<- rev rep c print`

Boolean operations

`is.na ! | & == != xor`

Aggregation methods

`anyNA any all sum min max range summary`

Creating and manipulating

`bit` and `bitwhich` vectors are created like `logical`, for example zero length vectors

```
logical()
#> logical(0)
bit()
#> bit length=0 occupying only 0 int32
bitwhich()
#> bitwhich: 0/0 occupying only 0 int32 in  representation
```

or vectors of a certain length initialized to `FALSE`

```
logical(3)
#> [1] FALSE FALSE FALSE
bit(3)
#> bit length=3 occupying only 1 int32
#>     1     2     3
#> FALSE FALSE FALSE
bitwhich(3)
#> bitwhich: 0/3 occupying only 1 int32 in FALSE representation
#>     1     2     3
#> FALSE FALSE FALSE
```

`bitwhich` can be created initialized to all elements TRUE with

```
bitwhich(3, TRUE)
#> bitwhich: 3/3 occupying only 1 int32 in TRUE representation
#>    1    2    3
#> TRUE TRUE TRUE
```

or can be created initialized to a few included or excluded elements

```
bitwhich(3, 2)
#> bitwhich: 1/3 occupying only 1 int32 in 1 representation
#>     1     2     3
#> FALSE  TRUE FALSE
bitwhich(3, -2)
#> bitwhich: 2/3 occupying only 1 int32 in -1 representation
#>     1     2     3
#>  TRUE FALSE  TRUE
```

Note that `logical` behaves somewhat inconsistent, when creating it, the default is `FALSE`, when increasing the length, the default is `NA`:

```
l <- logical(3)
length(l) <- 6
l
#> [1] FALSE FALSE FALSE    NA    NA    NA
```

Note that the default in `bit` is always `FALSE`, for creating and increasing the length.

```
b <- bit(3)
length(b) <- 6
b
#> bit length=6 occupying only 1 int32
#>     1     2     3     4     5     6
#> FALSE FALSE FALSE FALSE FALSE FALSE
```

Increasing the length of `bitwhich` initializes new elements to *the majority* of the old elements, hence a `bitwhich` with a few exclusions has majority `TRUE` and will have new elements initialized to `TRUE` (if both, `TRUE` and `FALSE` have equal frequency the default is `FALSE`).

```
w <- bitwhich(3,2)
length(w) <- 6
w
#> bitwhich: 1/6 occupying only 1 int32 in 1 representation
#>     1     2     3     4     5     6
#> FALSE  TRUE FALSE FALSE FALSE FALSE
w <- bitwhich(3,-2)
length(w) <- 6
w
#> bitwhich: 5/6 occupying only 1 int32 in -1 representation
#>     1     2     3     4     5     6
#>  TRUE FALSE  TRUE  TRUE  TRUE  TRUE
```

Vector subscripting non-existing elements returns `NA`

```
l <- logical(3); l[6]
#> [1] NA
b <- bit(3); b[6]
#> [1] NA
#> attr(,"vmode")
#> [1] "boolean"
w <- bitwhich(3); w[6]
#> [1] NA
#> attr(,"vmode")
#> [1] "boolean"
```

while assigned `NA` turn into `FALSE` and assigning to a non-existing element does increase vector length

```
l[6] <- NA; l
#> [1] FALSE FALSE FALSE    NA    NA    NA
b[6] <- NA; b
#> bit length=6 occupying only 1 int32
#>     1     2     3     4     5     6
#> FALSE FALSE FALSE FALSE FALSE FALSE
w[6] <- NA; w
#> bitwhich: 0/6 occupying only 1 int32 in FALSE representation
#>     1     2     3     4     5     6
#> FALSE FALSE FALSE FALSE FALSE FALSE
```

As usual list subscripting is only allowed for existing elements

```
l[[6]]
#> [1] NA
b[[6]]
#> [1] FALSE
#> attr(,"vmode")
#> [1] "boolean"
w[[6]]
#> [1] FALSE
#> attr(,"vmode")
#> [1] "boolean"
```

while assignments to non-existing elements do increase length.

```
l[[9]] <- TRUE
b[[9]] <- TRUE
w[[9]] <- TRUE
#> Warning in `[[<-.bitwhich`(`*tmp*`, 9, value = TRUE): increasing length of
#> bitwhich, which has non-standard semantics
l
#> [1] FALSE FALSE FALSE    NA    NA    NA    NA    NA  TRUE
b
#> bit length=9 occupying only 1 int32
#>     1     2     3     4     5     6     7     8     9
#> FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
w
#> bitwhich: 1/6 occupying only 1 int32 in 1 representation
#>     1     2     3     4     5     6  <NA>  <NA>  <NA>
#> FALSE FALSE FALSE FALSE FALSE FALSE    NA    NA  TRUE
```

Coercion

There are coercion functions between classes `logical`, `bit`, `bitwhich`, `which`, `integer` and `double`. However, only the first three of those represent Boolean vectors, whereas `which` represents subscript positions, and `integer` and `double` are ambiguous in that they can represent both, Booleans or positions.

Remember first that coercing `logical` to `integer` (or `double`) gives `0` and `1` and not `integer` subscript positions:

```
l <- c(FALSE, TRUE, FALSE)
i <- as.integer(l)
as.logical(i)
#> [1] FALSE  TRUE FALSE
```

To obtain `integer` subscript positions use `which` or better `as.which` because the latter S3 class remembers the original vector length and hence we can go coerce back to logical

```
l <- c(FALSE, TRUE, FALSE)
w <- as.which(l)
w
#> [1] 2
#> attr(,"maxindex")
#> [1] 3
#> attr(,"class")
#> [1] "booltype" "which"
as.logical(w)
#> [1] FALSE  TRUE FALSE
```

coercing back to `logical` fails using just `which`

```
l <- c(FALSE, TRUE, FALSE)
w <- which(l)
w
#> [1] 2
as.logical(w)  # does not coerce back
#> [1] TRUE
```

Furthermore from class `which` we can deduce that the positions are sorted which can be leveraged for performance. Note that `as.which.integer` is a core method for converting integer positions to class `which` and it enforces sorting

```
i <- c(7,3)
w <- as.which(i, maxindex=12)
w
#> [1] 3 7
#> attr(,"maxindex")
#> [1] 12
#> attr(,"class")
#> [1] "booltype" "which"
```

and `as.integer` gives us back those positions (now sorted)

```
as.integer(w)
#> [1] 3 7
```

You see that the `as.integer` generic is ambiguous as the `integer` data type in giving positions on some, and zeroes and ones on other inputs. The following set of Boolean types can be coerced without loss of information

`logical bit bitwhich which`

The same is true for this set

`logical bit bitwhich integer double`

Furthermore positions are retained in this set

`which integer double`

although the length of the Boolean vector is lost when coercing from `which` to `integer` or `double`, therefore coercing to them is not reversible. Let's first create all six types and compare their sizes:

```
r <- ri(1, 2^16, 2^20) # sample(2^20, replace=TRUE, prob=c(.125,875))
all.as <- list(
  double = as.double
, integer= as.integer
, logical = as.logical
, bit = as.bit
, bitwhich = as.bitwhich
, which = as.which
, ri = function(x)x
)
all.types <- lapply(all.as, function(f)f(r))
sapply(all.types, object.size)
#>   double  integer  logical      bit bitwhich    which       ri
#>  8388656  4194352  4194352   132584   262832   262656      360
```

Now let's create all combinations of coercion:

```
all.comb <- vector('list', length(all.types)^2)
all.id <- rep(NA, length(all.types)^2)
dim(all.comb)      <- dim(all.id)      <-    c(from=length(all.types),  to=length(all.types))
dimnames(all.comb) <- dimnames(all.id) <- list(from= names(all.types) , to= names(all.types))
for (i in seq_along(all.types))
  for (j in seq_along(all.as)){
    # coerce all types to all types (FROM -> TO)
    all.comb[[i,j]] <- all.as[[j]](all.types[[i]])
    # and test whether coercing back to the FROM type gives the orginal object
    all.id[i,j] <- identical(all.as[[i]](all.comb[[i,j]]),  all.types[[i]])
  }
all.id
#>           to
#> from       double integer logical   bit bitwhich which   ri
#>   double     TRUE    TRUE    TRUE  TRUE     TRUE FALSE TRUE
#>   integer    TRUE    TRUE    TRUE  TRUE     TRUE FALSE TRUE
#>   logical    TRUE    TRUE    TRUE  TRUE     TRUE  TRUE TRUE
#>   bit        TRUE    TRUE    TRUE  TRUE     TRUE  TRUE TRUE
#>   bitwhich   TRUE    TRUE    TRUE  TRUE     TRUE  TRUE TRUE
#>   which     FALSE   FALSE    TRUE  TRUE     TRUE  TRUE TRUE
#>   ri        FALSE   FALSE   FALSE FALSE    FALSE FALSE TRUE
```

Do understand the `FALSE` above!

The functions `booltype` and `is.booltype` diagnose the Boolean type as follows

```
data.frame(booltype=sapply(all.types, booltype), is.boolean=sapply(all.types, is.booltype), row.names=na
#>          booltype is.boolean
#> double     nobool      FALSE
#> integer    nobool      FALSE
#> logical   logical       TRUE
#> bit           bit       TRUE
#> bitwhich bitwhich       TRUE
#> which       which       TRUE
#> ri             ri       TRUE
```

Class `which` and `ri` are currently not `is.boolean` (no subscript, assignment and Boolean operators), but since it is even more specialized than `bitwhich` (assuming skew towards `TRUE`), we have ranked it as the most specialized `booltype`.

Boolean operations

The usual Boolean operators

`! | & == != xor is.na`

are implemented and the binary operators work for all combinations of `logical`, `bit` and `bitwhich`.

Technically this is achieved in S3 by giving `bit` and `bitwhich` another class `booltype`. Note that this is not inheritance where `booltype` implements common methods and `bit` and `bitwhich` overrule this with more specific methods. Instead the method dispatch to `booltype` dominates `bit` and `bitwhich` and coordinates more specific methods, this was the only way to realize binary logical operators that combine `bit` and `bitwhich` in S3, because in this case R dispatches to neither `bit` nor `bitwhich`: if both arguments are custom classes R does a non-helpful dispatch to `integer` or `logical`.

Anyhow, if a binary Boolean operator meets two different types, argument and result type is promoted to *less assumptions*, hence `bitwhich` is promoted to `bit` dropping the assumption of *strong skew* and `bit` is promoted to `logical` dropping the assumption that no `NA` are present.

Such promotion comes at the price of increased memory requirements, for example the following multiplies the memory requirement by factor 32

```
x <- bit(1e6)
y <- x | c(FALSE, TRUE)
object.size(y) / object.size(x)
#> 31.6 bytes
```

Better than lazily relying on automatic propagation is

```
x <- bit(1e6)
y <- x | as.bit(c(FALSE, TRUE))
object.size(y) / object.size(x)
#> 1 bytes
```

Manipulation methods

Concatenation follows the same promotion rules as Boolean operators. Note that `c` dispatches on the first argument only, hence when concatenating multiple Boolean types the first must not be logical, otherwise we get corrupt results:

```
l <- logical(6)
b <- bit(6)
c(l,b)
#> [1] 0 0 0 0 0 0 0
```

because `c.logical` treats the six bits as a single value. The following expressions work

```
c(b,l)
#>  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
c(l, as.logical(b))
#>  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

and of course the most efficient is

```
c(as.bit(l), b)
#> bit length=12 occupying only 1 int32
#>     1     2     3     4     5     6     7     8     9    10    11    12
#> FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

If you want your code to process any `is.booltype`, you can use `c.booltype` directly

```
c.booltype(l, b)
#>  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

Both, `bit` and `bitwhich` also have replication (`rep`) and reverse (`rev`) methods that work as expected:

```
b <- as.bit(c(FALSE,TRUE))
rev(b)
#> bit length=2 occupying only 1 int32
#>     1     2
#>  TRUE FALSE
rep(b, 3)
#> bit length=6 occupying only 1 int32
#>     1     2     3     4     5     6
#> FALSE  TRUE FALSE  TRUE FALSE  TRUE
rep(b, length.out=6)
#> bit length=6 occupying only 1 int32
#>     1     2     3     4     5     6
#> FALSE  TRUE FALSE  TRUE FALSE  TRUE
```

Aggregation methods for booltype

The usual logical aggregation functions `length`, `all`, `any` and `anyNA` work as expected. Note the exception that `length(which)` does not give the length of the Boolean vector but the length of the vector of positive integers (like the result of function `which`). `sum` gives the number of `TRUE` for all Boolean types. For the `booltype > 1 min` gives the first position of a `TRUE` (i.e. `which.max`), `max` gives the last position of a `TRUE`, `range` gives both, the range at which we find `TRUE`, and finally `summary` gives the the counts of `FALSE` and `TRUE` as well as `min` and `max`. For example

```
l <- c(NA,NA,FALSE,TRUE,TRUE)
b <- as.bit(l)
length(b)
#> [1] 5
anyNA(b)
#> [1] FALSE
any(b)
#> [1] TRUE
all(b)
#> [1] FALSE
sum(b)
#> [1] 2
min(b)
#> [1] 4
max(b)
#> [1] 5
range(b)
#> [1] 4 5
summary(b)
#> FALSE  TRUE  Min.  Max.
#>     3     2     4     5
```

These special interpretations of `min`, `max`, `range` and `summary` can be enforced for type `logical`, `integer`, and `double` by using the `booltype` methods directly as in

```
# minimum after coercion to integer
min(c(FALSE,TRUE))
#> [1] 0
# minimum  position of first TRUE
min.booltype(c(FALSE,TRUE))
#> [1] 2
```

Except for `length` and `anyNA` the aggregation functions support an optional argument `range` which restricts evaluation the specified range of the Boolean vector. This is useful in the context of *chunked processing*. For example analyzing the first 30% of a million Booleans

```
b <- as.bit(sample(c(FALSE, TRUE), 1e6, TRUE))
summary(b,range=c(1,3e5))
#>  FALSE   TRUE   Min.   Max.
#> 149762 150238      4 300000
```

and analyzing all such chunks

```
sapply(chunk(b, by=3e5, method="seq"), function(i)summary(b, range=i))
#>       1:300000 300001:600000 600001:900000 900001:1000000
#> FALSE   149762        149881        149915          50062
#> TRUE    150238        150119        150085         49938
#> Min.         4        300004        600002        900003
#> Max.    300000        600000        900000        999997
```

or better balanced

```
sapply(chunk(b, by=3e5), function(i)summary(b, range=i))
#>       1:250000 250001:500000 500001:750000 750001:1000000
#> FALSE    124804        124979        124859         124978
#> TRUE     125196        125021        125141         125022
#> Min.          4        250001        500004         750001
#> Max.     249995        500000        749999         999997
```

The real use-case for chunking is ff objects, where instead of processing huge objects at once

```
x <- ff(vmode="single", length=length(b))   # create a huge ff vector
x[as.hi(b)] <- runif(sum(b))      # replace some numbers at filtered positions
summary(x[])
#>      Min.   1st Qu.    Median      Mean   3rd Qu.      Max. 
#> 0.0000000 0.0000000 0.0007731 0.2505775 0.5014399 0.9999987
```

we can process the ff vector in chunks

```
sapply(chunk(x, by=3e5), function(i)summary(x[i]))
#>             1:250000 250001:500000 500001:750000 750001:1000000
#> Min.    0.000000000  0.0000000000   0.000000000     0.00000000
#> 1st Qu. 0.000000000  0.0000000000   0.000000000     0.00000000
#> Median  0.001411806  0.0002212567   0.001182959     0.00016363
#> Mean    0.250731496  0.2505470859   0.250505759     0.25052570
#> 3rd Qu. 0.503259346  0.5015245974   0.500096723     0.50087063
#> Max.    0.999998331  0.9999987483   0.999981821     0.99999267
```

and even can process a bit-filtered ff vector in chunks

```
sapply(chunk(x, by=3e5), function(i)summary(x[as.hi(b, range=i)]))
#>              1:250000 250001:500000 500001:750000 750001:1000000
#> Min.    5.359529e-06  1.186575e-05  1.779874e-05   3.266078e-05
#> 1st Qu. 2.486259e-01  2.521560e-01  2.524429e-01   2.506243e-01
#> Median  5.025245e-01  5.014488e-01  4.994555e-01   5.007975e-01
#> Mean    5.006779e-01  5.010100e-01  5.004470e-01   5.009632e-01
#> 3rd Qu. 7.503261e-01  7.496561e-01  7.500216e-01   7.515188e-01
#> Max.    9.999983e-01  9.999987e-01  9.999818e-01   9.999927e-01
```

Fast methods for `integer` set operations

R implements *set methods* using *hash tables*, namely `match`, `%in%`, `unique`, `duplicated`, `union`, `intersect`, `setdiff`, `setequal`. Hashing is a powerful method, but it is costly in terms of random access and memory consumption. For integers package 'bit' implements faster set methods using *bit vectors* (which can be an order of magnitude faster and saves up to factor 64 on temporary memory) and *merging* (which is about two orders of magnitude faster, needs no temporary memory but requires sorted input). While many set methods return *unique* sets, the `merge_*` methods can optionally preserve ties, the `range_*` methods below allow to specify one of the sets as a *range*.

R hashing	bit vectors	merging	range merging	rlepack
`match`		`merge_match`		
`%in%`	`bit_in`	`merge_in`	`merge_rangein`	
`!(%in%)`	`!bit_in`	`merge_notin`	`merge_rangenotin`	
`duplicated`	`bit_duplicated`			
`unique`	`bit_unique`	`merge_unique`		`unique(rlepack)`
`union`	`bit_union`	`merge_union`		
`intersect`	`bit_intersect`	`merge_intersect`	`merge_rangesect`	
`setdiff`	`bit_setdiff`	`merge_setdiff`	`merge_rangediff`	
`(a:b)[-i]`	`bit_rangediff`	`merge_rangediff`	`merge_rangediff`	
	`bit_symdiff`	`merge_symdiff`		
`setequal`	`bit_setequal`	`merge_setequal`		
`anyDuplicated`	`bit_anyDuplicated`			`anyDuplicated(rlepack)`
`sum(duplicated)`	`bit_sumDuplicated`			

Furthermore there are very fast methods for sorting integers (unique or keeping ties), reversals that simultaneously change the sign to preserve ascending order and methods for finding min or max in sorted vectors or within ranges of sorted vectors.

R	bit vectors	merging	range merging
`sort(unique)`	`bit_sort_unique`		
`sort`	`bit_sort`		
`rev`	`rev(bit)`	`reverse_vector`	
`-rev`		`copy_vector(revx=TRUE)`	
`min`	`min(bit)`	`merge_first`	`merge_firstin`
`max`	`max(bit)`	`merge_last`	`merge_lastin`
	`min(!bit)`		`merge_firstnotin`
	`max(!bit)`		`merge_lastnotin`

Methods using random access to bit vectors

Set operations using hash tables incur costs for populating and querying the hash table: this is random access cost to a relative large table. In order to avoid *hash collisions* hash tables need more elements than those being hashed, typically `2*N` for `N` elements. That is hashing `N` integer elements using a int32 hash function costs *random access* to `64*N` bits of memory. If the `N` elements are within a range of `N` values, it is much (by factor 64) cheaper to register them in a bit-vector of `N` bits. The `bit_*` functions first determine the range of the values (and count of `NA`), and then use an appropriately sized bit vector. Like the original R functions the `bit_*` functions keep the original order of values (although some implementations could be faster by delivering an undefined order). If `N` is small relative to the range of the values the `bit_*` fall back to the standard R functions using hash tables. Where the `bit_*` functions return Boolean vectors they do so by default as `bit` vectors, but but you can give a different coercion function as argument `retFUN`.

Bit vectors cannot communicate positions, hence cannot replace `match`, however they can replace the `%in%` operator:

```r
set.seed(1); n <- 9
x <- sample(n, replace=TRUE); x
#> [1] 9 4 7 1 2 7 2 3 1
y <- sample(n, replace=TRUE); y
#> [1] 5 5 6 7 9 5 5 9 9
x %in% y
```

```
#> [1]  TRUE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE
bit_in(x,y)
#> bit length=9 occupying only 1 int32
#>     1     2     3     4     5     6     7     8     9
#>  TRUE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE
bit_in(x,y, retFUN=as.logical)
#> [1]  TRUE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE
```

The `bit_in` function combines a bit vector optimization with reverse look-up, i.e. if the range of `x` is smaller than the range of `table`, we build the bit vector on `x` instead of the `table`.

The `bit_duplicated` function can handle `NA` in three different ways

```
x <- c(NA,NA,1L,1L,2L,3L)
duplicated(x)
#> [1] FALSE  TRUE FALSE  TRUE FALSE FALSE
bit_duplicated(x, retFUN=as.logical)
#> [1] FALSE  TRUE FALSE  TRUE FALSE FALSE
bit_duplicated(x, na.rm=NA, retFUN=as.logical)
#> [1] FALSE  TRUE FALSE  TRUE FALSE FALSE

duplicated(x, incomparables = NA)
#> [1] FALSE FALSE FALSE  TRUE FALSE FALSE
bit_duplicated(x, na.rm=FALSE, retFUN=as.logical)
#> [1] FALSE FALSE FALSE  TRUE FALSE FALSE

bit_duplicated(x, na.rm=TRUE, retFUN=as.logical)
#> [1]  TRUE  TRUE FALSE  TRUE FALSE FALSE
```

The `bit_unique` function can also handle `NA` in three different ways

```
x <- c(NA,NA,1L,1L,2L,3L)
unique(x)
#> [1] NA  1  2  3
bit_unique(x)
#> [1] NA  1  2  3

unique(x, incomparables = NA)
#> [1] NA NA  1  2  3
bit_unique(x, na.rm=FALSE)
#> [1] NA NA  1  2  3

bit_unique(x, na.rm=TRUE)
#> [1] 1 2 3
```

The `bit_union` function build a bit vector spanning the united range of both input sets and filters all unites duplicates:

```
x <- c(NA,NA,1L,1L,3L)
y <- c(NA,NA,2L,2L,3L)
union(x,y)
#> [1] NA  1  3  2
bit_union(x,y)
#> [1] NA  1  3  2
```

The `bit_intersect` function builds a bit vector spanning only the intersected range of both input sets and filters all elements outside and the duplicates inside:

```
x <- c(0L,NA,NA,1L,1L,3L)
y <- c(NA,NA,2L,2L,3L,4L)
intersect(x,y)
#> [1] NA  3
bit_intersect(x,y)
#> [1] NA  3
```

The `bit_setdiff` function builds a bit vector spanning the range of the first input set, marks elements of the second set within this range as tabooed, and then outputs the remaining elements of the first set unless they are duplicates:

```
x <- c(0L,NA,NA,1L,1L,3L)
y <- c(NA,NA,2L,2L,3L,4L)
setdiff(x,y)
#> [1] 0 1
bit_setdiff(x,y)
#> [1] 0 1
```

The `bit_symdiff` function implements symmetric set difference. It builds two bit vectors spanning the full range and then outputs those elements of both sets that are marked at exactly one of the bit vectors.

```
x <- c(0L,NA,NA,1L,1L,3L)
y <- c(NA,NA,2L,2L,3L,4L)
union(setdiff(x,y),setdiff(y,x))
#> [1] 0 1 2 4
bit_symdiff(x,y)
#> [1] 0 1 2 4
```

The `bit_setequal` function terminates early if the ranges of the two sets (or the presence of `NA`) differ. Otherwise it builds two bit vectors spanning the identical range; finally it checks the two vectors for being equal with early termination if two unequal integers are found.

```
x <- c(0L,NA,NA,1L,1L,3L)
y <- c(NA,NA,2L,2L,3L,4L)
setequal(y,x)
#> [1] FALSE
bit_setequal(x,y)
#> [1] FALSE
```

The `bit_rangediff` function works like `bit_setdiff` with two differences: the first set is specified as a *range* of integers, and it has two arguments `revx` and `revy` which allow to reverse *order* and *sign* of the two sets *before* the set-diff operation is done. The order of the range is significant, e.g. `c(1L,7L)` is different from `c(7L,1L)`, while the order of the second set has no influence:

```
bit_rangediff(c(1L,7L), (3:5))
#> [1] 1 2 6 7
bit_rangediff(c(7L,1L), (3:5))
#> [1] 7 6 2 1
bit_rangediff(c(1L,7L), -(3:5), revy=TRUE)
#> [1] 1 2 6 7
bit_rangediff(c(1L,7L), -(3:5), revx=TRUE)
#> [1] -7 -6 -2 -1
```

If the range and the second set don't overlap, for example due to different signs, the full range is returned:

```
bit_rangediff(c(1L,7L), (1:7))
#> integer(0)
bit_rangediff(c(1L,7L), -(1:7))
#> [1] 1 2 3 4 5 6 7
bit_rangediff(c(1L,7L), (1:7), revy=TRUE)
#> [1] 1 2 3 4 5 6 7
```

Note that `bit_rangediff` provides faster negative subscripting from a range of integers than the usual phrase `(1:n)[-i]`:

```
(1:9)[-7]
#> [1] 1 2 3 4 5 6 8 9
bit_rangediff(c(1L,9L), -7L, revy=TRUE)
#> [1] 1 2 3 4 5 6 8 9
```

Functions `bit_anyDuplicated` is a faster version of `anyDuplicated`

```
x <- c(NA,NA,1L,1L,2L,3L)
    any(duplicated(x))  # full hash work, returns FALSE or TRUE
#> [1] TRUE
     anyDuplicated(x)   # early termination of hash work, returns 0 or position of first duplicate
#> [1] 2
any(bit_duplicated(x))  # full bit work, returns FALSE or TRUE
#> [1] TRUE
 bit_anyDuplicated(x)   # early termination of bit work, returns 0 or position of first duplicate
#> [1] 2
```

For the meaning of the `na.rm` parameter see `bit_duplicated`. Function `bit_sumDuplicated` is a faster version of `sum(bit_duplicated)`

```
x <- c(NA,NA,1L,1L,2L,3L)
    sum(duplicated(x))  # full hash work, returns FALSE or TRUE
#> [1] 2
sum(bit_duplicated(x))  # full bit work, returns FALSE or TRUE
#> [1] 2
 bit_sumDuplicated(x)   # early termination of bit work, returns 0 or position of first duplicated
#> [1] 2
```

Methods using bit vectors for sorting integers

A bit vector cannot replace a hash table for all possible kind of tasks (when it comes to counting values or to their positions), but on the other hand a bit vector allows something impossible with a hash table: *sorting* keys (without payload). Sorting a large subset of unique integers in [1,N] using a bit vector has been described in *"Programming pearls – cracking the oyster"* by Jon Bentley (where 'uniqueness' is not an input condition but an output feature). This is easily generalized to sorting a large subset of integers in a range [min,max]. A first scan over the data determines the range of the keys, arranges `NA`s according to the usual `na.last=` argument and checks for presortedness (see `range_sortna`), then the range is projected to a bit vector of size `max-min+1`, all keys are registered causing random access and a final scan over the bit vector sequentially writes out the sorted keys. This is a *synthetical sort* because unlike a *comparison sort* the elements are not moved towards their ordered positions, instead they are *synthesized* from the bit-representation.

For `N` consecutive permuted integers this is by an order of magnitude faster than *quicksort*. For a *density*

$d = N/(max - min) \geq 1$ this sort beats quicksort, but for $d << 1$ the bit vector becomes too large relative to N and hence quicksort is faster. bit_sort_unique implements a hybrid algorithm automatically choosing the faster of both, hence for integers the following gives identical results

```
x <- sample(9, 9, TRUE)
unique(sort(x))
#> [1] 1 2 3 4 5 6 9
sort(unique(x))
#> [1] 1 2 3 4 5 6 9
bit_sort_unique(x)
#> [1] 1 2 3 4 5 6 9
```

What if duplicates shall be kept?

- A: sort all non-duplicated occurrences using a bit vector
- B: sort the rest
- merge A and B

The crucial question is how to sort the rest? Recursively using a bit-vector can again be faster than quicksort, however it is non-trivial to determine an optimal recursion-depth before falling back to quicksort. Hence a safe bet is using the bit vector only once and sort the rest via quicksort, let's call that bitsort. Again, using bitsort is fast only at medium density in the data range. For low density quicksort is faster. For high density (duplicates) another synthetic sort is faster: *counting sort*. bit_sort implements a *sandwhich sort* algorithm which given density uses bitsort between two complementing algorithms:

```
low density    medium density    high density
quicksort   << bitsort        << countsort
```

```
x <- sample(9, 9, TRUE)
sort(x)
#> [1] 4 4 6 7 7 8 9 9 9
bit_sort(x)
#> [1] 4 4 6 7 7 8 9 9 9
```

Both, bit_sort_unique and bit_sort can sort *decreasing*, however, currently this requires an extra pass over the data in bit_sort and in the quicksort fallback of bit_sort_unique. So far, bit_sort does not leverage *radix sort* for very large N.

Methods for sets of sorted integers

Efficient handling of sorted sets is backbone of class bitwhich. The merge_* and merge_range* integer functions expect sorted input (non-decreasing sets or increasing ranges) and return sorted sets (some return Boolean vectors or scalars). Exploiting the sortedness makes them even faster than the bit_ functions. Many of them have revx or revy arguments, which reverse the scanning direction of an input vector *and* the interpreted *sign* of its elements, hence we can change signs of input vectors in an order-preserving way and without any extra pass over the data. By default these functions return *unique* sets which have each element not more than once. However, the *binary* merge_* functions have a method="exact" which in both sets treats consecutive occurrences of the same value as different. With method="exact" for example merge_setdiff behaves as if counting the values in the first set and subtraction the respective counts of the second set (and capping the lower end at zero). Assuming positive integers and equal tabulate(, nbins=) with method="exact" the following is identical:

merging then counting	counting then combining
tabulate(merge_union(x,y))	pmax(tabulate(x) ,tabulate(y))
tabulate(merge_intersect(x,y))	pmin(tabulate(x) ,tabulate(y))

merging then counting	counting then combining
`tabulate(merge_setdiff(x,y))`	`pmax(tabulate(x) -tabulate(y), 0)`
`tabulate(merge_symdiff(x,y))`	`abs(tabulate(x) -tabulate(y))`
`tabulate(merge_setequal(x,y))`	`all(tabulate(x)==tabulate(y))`

Note further that `method="exact"` delivers *unique* output if the input is unique, and in this case works faster than `method="unique"`.

Note further, that the `merge_*` and `merge_range*` functions have no special treatment for `NA`. If vectors with `NA` are sorted ith `NA` in the first positions (`na.last=FALSE`) and arguments `revx=` or `revy=` have not been used, then `NA`s are treated like ordinary integers. `NA` sorted elsewhere or using `revx=` or `revy=` can cause unexpected results (note for example that `revx=` switches the sign on all integers but `NA`s).

The unary `merge_unique` function transform a sorted set into a unique sorted set:

```
x = sample(12)
bit_sort(x)
#>  [1]  1  2  3  4  5  6  7  8  9 10 11 12
merge_unique(bit_sort(x))
#>  [1]  1  2  3  4  5  6  7  8  9 10 11 12
bit_sort_unique(x)
#>  [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

For binary functions let's start with *set equality*:

```
x = as.integer(c(3,4,4,5))
y = as.integer(c(3,4,5))
setequal(x,y)
#> [1] TRUE
merge_setequal(x,y)
#> [1] TRUE
merge_setequal(x,y, method="exact")
#> [1] FALSE
```

For *set complement* there is also a `merge_range*` function:

```
x = as.integer(c(0,1,2,2,3,3,3))
y = as.integer(c(1,2,3))
setdiff(x,y)
#> [1] 0
merge_setdiff(x,y)
#> [1] 0
merge_setdiff(x,y, method="exact")
#> [1] 0 2 3 3
merge_rangediff(c(0L,4L),y)
#> [1] 0 4
merge_rangediff(c(0L,4L),c(-3L,-2L)) # y has no effect due to different sign
#> [1] 0 1 2 3 4
merge_rangediff(c(0L,4L),c(-3L,-2L), revy=TRUE)
#> [1] 0 1 4
merge_rangediff(c(0L,4L),c(-3L,-2L), revx=TRUE)
#> [1] -4 -1  0
```

`merge_symdiff` for *symmetric set complement* is used similar (without a `merge_range*` function), as is `merge_intersect`, where the latter is accompanied by a `merge_range*` function:

```
x = -2:1
y = -1:2
setdiff(x,y)
#> [1] -2
union(setdiff(x,y),setdiff(y,x))
#> [1] -2  2
merge_symdiff(x,y)
#> [1] -2  2
merge_intersect(x,y)
#> [1] -1  0  1
merge_rangesect(c(-2L,1L),y)
#> [1] -1  0  1
```

The `merge_union` function has a third method `all` which behaves like `c` but keeps the output sorted

```
x = as.integer(c(1,2,2,3,3,3))
y = 2:4
union(x,y)
#> [1] 1 2 3 4
merge_union(x,y, method="unique")
#> [1] 1 2 3 4
merge_union(x,y, method="exact")
#> [1] 1 2 2 3 3 3 4
merge_union(x,y, method="all")
#> [1] 1 2 2 2 3 3 3 3 4
sort(c(x,y))
#> [1] 1 2 2 2 3 3 3 3 4
c(x,y)
#> [1] 1 2 2 3 3 3 2 3 4
```

Unlike the `bit_*` functions the `merge_*` functions have a `merge_match` function:

```
x = 2:4
y = as.integer(c(0,1,2,2,3,3,3))
match(x,y)
#> [1]  3  5 NA
merge_match(x,y)
#> [1]  3  5 NA
```

and unlike R's `%in%` operator the following functions are directly implemented, not on top of merge, and hence save extra passes over the data:

```
x %in% y
#> [1]  TRUE  TRUE FALSE
merge_in(x,y)
#> [1]  TRUE  TRUE FALSE
merge_notin(x,y)
#> [1] FALSE FALSE  TRUE
```

The range versions extract logical vectors from `y`, but only for the range in `rx`.

```
x <- c(2L,4L)
merge_rangein(x,y)
```

```
#> [1]  TRUE  TRUE FALSE
merge_rangenotin(x,y)
#> [1] FALSE FALSE  TRUE
```

Compare this to `merge_rangesect` above. `merge_rangein` is useful in the context of chunked processing, see the `any`, `all`, `sum` and `[` methods of class `bitwhich`.

The functions `merge_first` and `merge_last` give first and last element of a sorted set. By default that is *min* and *max*, however these functions also have an `revx` argument. There are also functions that deliver the first resp. last elements *of* a certain range that are *in* or *not in* a certain set.

```
x <- bit_sort(sample(1000,10))
merge_first(x)
#> [1] 104
merge_last(x)
#> [1] 983
merge_firstnotin(c(300L,600L), x)
#> [1] 300
merge_firstin(c(300L,600L), x)
#> [1] 326
merge_lastin(c(300L,600L), x)
#> [1] 499
merge_lastnotin(c(300L,600L), x)
#> [1] 600
```
