An Overview of the *IRanges* package

Patrick Aboyoun, Michael Lawrence, Hervé Pagès

Edited: February 2018; Compiled: October 29, 2024

Contents

1 Introduction . . . 1

2 *IRanges* objects . . . 2

2.1 Normality . . . 4

2.2 Lists of *IRanges* objects . . . 5

2.3 Vector Extraction . . . 5

2.4 Finding Overlapping Ranges . . . 5

2.5 Counting Overlapping Ranges . . . 6

2.6 Finding Neighboring Ranges . . . 6

2.7 Transforming Ranges . . . 6

2.7.1 Adjusting starts, ends and widths . . . 6

2.7.2 Making ranges disjoint . . . 9

2.7.3 Other transformations . . . 10

2.8 Set Operations . . . 10

3 Vector Views . . . 11

3.1 Creating Views . . . 11

3.2 Aggregating Views . . . 11

4 Lists of Atomic Vectors . . . 12

5 Session Information . . . 15

1 Introduction

When analyzing sequences, we are often interested in particular consecutive subsequences. For example, the a vector could be considered a sequence of lower-case letters, in alphabetical order. We would call the first five letters (*a* to *e*) a consecutive subsequence, while the subsequence containing only the vowels would not be consecutive. It is not uncommon for an analysis task to focus only on the geometry of the regions, while ignoring the underlying sequence values. A list of indices would be a simple way to select a subsequence. However, a sparser representation for consecutive subsequences would be a range, a pairing of a start position and a width, as used when extracting sequences with `window`.

Two central classes are available in Bioconductor for representing ranges: the *IRanges* class defined in the *IRanges* package for representing ranges defined on a single space, and the *GRanges* class defined in the *GenomicRanges* package for representing ranges defined on multiple spaces.

In this vignette, we will focus on *IRanges* objects. We will rely on simple, illustrative example datasets, rather than large, real-world data, so that each data structure and algorithm can be explained in an intuitive, graphical manner. We expect that packages that apply *IRanges* to a particular problem domain will provide vignettes with relevant, realistic examples.

The *IRanges* package is available at bioconductor.org and can be downloaded via `BiocManager::install`:

```
> if (!require("BiocManager"))
+     install.packages("BiocManager")
> BiocManager::install("IRanges")
```

```
> library(IRanges)
```

2 *IRanges* objects

To construct an *IRanges* object, we call the `IRanges` constructor. Ranges are normally specified by passing two out of the three parameters: start, end and width (see `help(IRanges)` for more information).

```
> ir1 <- IRanges(start=1:10, width=10:1)
> ir1

IRanges object with 10 ranges and 0 metadata columns:
           start       end     width
       <integer> <integer> <integer>
   [1]         1        10        10
   [2]         2        10         9
   [3]         3        10         8
   [4]         4        10         7
   [5]         5        10         6
   [6]         6        10         5
   [7]         7        10         4
   [8]         8        10         3
   [9]         9        10         2
  [10]        10        10         1

> ir2 <- IRanges(start=1:10, end=11)
> ir3 <- IRanges(end=11, width=10:1)
> identical(ir1, ir2) && identical(ir1, ir3)

[1] FALSE

> ir <- IRanges(c(1, 8, 14, 15, 19, 34, 40),
+               width=c(12, 6, 6, 15, 6, 2, 7))
> ir

IRanges object with 7 ranges and 0 metadata columns:
          start       end     width
```

```
      <integer> <integer> <integer>
  [1]         1        12        12
  [2]         8        13         6
  [3]        14        19         6
  [4]        15        29        15
  [5]        19        24         6
  [6]        34        35         2
  [7]        40        46         7
```

All of the above calls construct the same *IRanges* object, using different combinations of the `start`, `end` and `width` parameters.

Accessing the starts, ends and widths is supported via the `start`, `end` and `width` getters:

```
> start(ir)
[1]  1  8 14 15 19 34 40
```

```
> end(ir)
[1] 12 13 19 29 24 35 46
```

```
> width(ir)
[1] 12  6  6 15  6  2  7
```

Subsetting an *IRanges* object is supported, by numeric and logical indices:

```
> ir[1:4]
IRanges object with 4 ranges and 0 metadata columns:
          start       end     width
      <integer> <integer> <integer>
  [1]         1        12        12
  [2]         8        13         6
  [3]        14        19         6
  [4]        15        29        15
```

```
> ir[start(ir) <= 15]
IRanges object with 4 ranges and 0 metadata columns:
          start       end     width
      <integer> <integer> <integer>
  [1]         1        12        12
  [2]         8        13         6
  [3]        14        19         6
  [4]        15        29        15
```

In order to illustrate range operations, we'll create a function to plot ranges.

```
> plotRanges <- function(x, xlim=x, main=deparse(substitute(x)),
+                        col="black", sep=0.5, ...)
+ {
+   height <- 1
+   if (is(xlim, "IntegerRanges"))
```

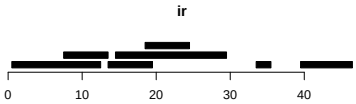

Figure 1: Plot of original ranges.

```
+     xlim <- c(min(start(xlim)), max(end(xlim)))
+   bins <- disjointBins(IRanges(start(x), end(x) + 1))
+   plot.new()
+   plot.window(xlim, c(0, max(bins)*(height + sep)))
+   ybottom <- bins * (sep + height) - height
+   rect(start(x)-0.5, ybottom, end(x)+0.5, ybottom + height, col=col, ...)
+   title(main)
+   axis(1)
+ }
```

```
> plotRanges(ir)
```

2.1 Normality

Sometimes, it is necessary to formally represent a subsequence, where no elements are repeated and order is preserved. Also, it is occasionally useful to think of an *IRanges* object as a set of integers, where no elements are repeated and order does not matter.

The *NormalIRanges* class formally represents a set of integers. By definition an *IRanges* object is said to be *normal* when its ranges are: (a) not empty (i.e. they have a non-null width); (b) not overlapping; (c) ordered from left to right; (d) not even adjacent (i.e. there must be a non empty gap between 2 consecutive ranges).

There are three main advantages of using a *normal IRanges* object: (1) it guarantees a subsequence encoding or set of integers, (2) it is compact in terms of the number of ranges, and (3) it uniquely identifies its information, which simplifies comparisons.

The `reduce` function reduces any *IRanges* object to a *NormalIRanges* by merging redundant ranges.

```
> reduce(ir)

IRanges object with 3 ranges and 0 metadata columns:
          start       end     width
      <integer> <integer> <integer>
  [1]         1        29        29
  [2]        34        35         2
  [3]        40        46         7

> plotRanges(reduce(ir))
```

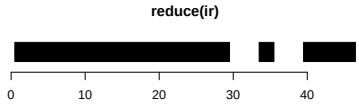

Figure 2: Plot of reduced ranges.

2.2 Lists of *IRanges* objects

It is common to manipulate collections of *IRanges* objects during an analysis. Thus, the *IRanges* package defines some specific classes for working with multiple *IRanges* objects.

The *IRangesList* class asserts that each element is an *IRanges* object and provides convenience methods, such as `start`, `end` and `width` accessors that return *IntegerList* objects, aligning with the *IRangesList* object. Note that *IntegerList* objects will be covered later in more details in the "Lists of Atomic Vectors" section of this document.

To explicitly construct an *IRangesList*, use the `IRangesList` function.

```
> rl <- IRangesList(ir, rev(ir))
```

```
> start(rl)

IntegerList of length 2
[[1]] 1 8 14 15 19 34 40
[[2]] 40 34 19 15 14 8 1
```

2.3 Vector Extraction

As the elements of an *IRanges* object encode consecutive subsequences, they may be used directly in sequence extraction. Note that when a *normal IRanges* is given as the index, the result is a subsequence, as no elements are repeated or reordered. If the sequence is a *Vector* subclass (i.e. not an ordinary *vector*), the canonical `[` function accepts an *IRanges* object.

```
> set.seed(0)
> lambda <- c(rep(0.001, 4500), seq(0.001, 10, length=500),
+             seq(10, 0.001, length=500))
> xVector <- rpois(1e7, lambda)
> yVector <- rpois(1e7, lambda[c(251:length(lambda), 1:250)])
> xRle <- Rle(xVector)
> yRle <- Rle(yVector)
> irextract <- IRanges(start=c(4501, 4901) , width=100)
> xRle[irextract]

integer-Rle of length 200 with 159 runs
  Lengths: 12  1  1  1  2  1  1  1  1  2 ...  1  1  1  1  1  1  1  1  1
  Values :  0  1  0  2  0  1  0  1  0  1 ...  9 12  6  5 10  9  6  9 12
```

2.4 Finding Overlapping Ranges

The function `findOverlaps` detects overlaps between two *IRanges* objects.

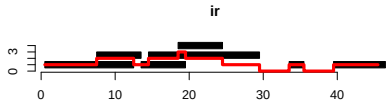

Figure 3: Plot of ranges with accumulated coverage.

```
> ol <- findOverlaps(ir, reduce(ir))
> as.matrix(ol)
     queryHits subjectHits
[1,]         1           1
[2,]         2           1
[3,]         3           1
[4,]         4           1
[5,]         5           1
[6,]         6           2
[7,]         7           3
```

2.5 Counting Overlapping Ranges

The function `coverage` counts the number of ranges over each position.

```
> cov <- coverage(ir)
> plotRanges(ir)
> cov <- as.vector(cov)
> mat <- cbind(seq_along(cov)-0.5, cov)
> d <- diff(cov) != 0
> mat <- rbind(cbind(mat[d,1]+1, mat[d,2]), mat)
> mat <- mat[order(mat[,1]),]
> lines(mat, col="red", lwd=4)
> axis(2)
```

2.6 Finding Neighboring Ranges

The `nearest` function finds the nearest neighbor ranges (overlapping is zero distance). The `precede` and `follow` functions find the non-overlapping nearest neighbors on a specific side.

2.7 Transforming Ranges

Utilities are available for transforming an *IRanges* object in a variety of ways. Some transformations, like `reduce` introduced above, can be dramatic, while others are simple per-range adjustments of the starts, ends or widths.

2.7.1 Adjusting starts, ends and widths

Perhaps the simplest transformation is to adjust the start values by a scalar offset, as performed by the `shift` function. Below, we shift all ranges forward 10 positions.

```
> shift(ir, 10)

IRanges object with 7 ranges and 0 metadata columns:
          start       end     width
      <integer> <integer> <integer>
  [1]        11        22        12
  [2]        18        23         6
  [3]        24        29         6
  [4]        25        39        15
  [5]        29        34         6
  [6]        44        45         2
  [7]        50        56         7
```

There are several other ways to transform ranges. These include `narrow`, `resize`, `flank`, `reflect`, `restrict`, and `threebands`. For example `narrow` supports the adjustment of start, end and width values, which should be relative to each range. These adjustments are vectorized over the ranges. As its name suggests, the ranges can only be narrowed.

```
> narrow(ir, start=1:5, width=2)

IRanges object with 7 ranges and 0 metadata columns:
          start       end     width
      <integer> <integer> <integer>
  [1]         1         2         2
  [2]         9        10         2
  [3]        16        17         2
  [4]        18        19         2
  [5]        23        24         2
  [6]        34        35         2
  [7]        41        42         2
```

The `restrict` function ensures every range falls within a set of bounds. Ranges are contracted as necessary, and the ranges that fall completely outside of but not adjacent to the bounds are dropped, by default.

```
> restrict(ir, start=2, end=3)

IRanges object with 1 range and 0 metadata columns:
          start       end     width
      <integer> <integer> <integer>
  [1]         2         3         2
```

The `threebands` function extends `narrow` so that the remaining left and right regions adjacent to the narrowed region are also returned in separate *IRanges* objects.

```
> threebands(ir, start=1:5, width=2)

$left
IRanges object with 7 ranges and 0 metadata columns:
          start       end     width
      <integer> <integer> <integer>
  [1]         1         0         0
  [2]         8         8         1
  [3]        14        15         2
```

```
  [4]        15        17         3
  [5]        19        22         4
  [6]        34        33         0
  [7]        40        40         1

$middle
IRanges object with 7 ranges and 0 metadata columns:
          start       end     width
      <integer> <integer> <integer>
  [1]         1         2         2
  [2]         9        10         2
  [3]        16        17         2
  [4]        18        19         2
  [5]        23        24         2
  [6]        34        35         2
  [7]        41        42         2

$right
IRanges object with 7 ranges and 0 metadata columns:
          start       end     width
      <integer> <integer> <integer>
  [1]         3        12        10
  [2]        11        13         3
  [3]        18        19         2
  [4]        20        29        10
  [5]        25        24         0
  [6]        36        35         0
  [7]        43        46         4
```

The arithmetic operators `+`, `-` and `*` change both the start and the end/width by symmetrically expanding or contracting each range. Adding or subtracting a numeric (integer) vector to an *IRanges* causes each range to be expanded or contracted on each side by the corresponding value in the numeric vector.

```
> ir + seq_len(length(ir))

IRanges object with 7 ranges and 0 metadata columns:
          start       end     width
      <integer> <integer> <integer>
  [1]         0        13        14
  [2]         6        15        10
  [3]        11        22        12
  [4]        11        33        23
  [5]        14        29        16
  [6]        28        41        14
  [7]        33        53        21
```

The `*` operator symmetrically magnifies an *IRanges* object by a factor, where positive contracts (zooms in) and negative expands (zooms out).

```
> ir * -2 # double the width

IRanges object with 7 ranges and 0 metadata columns:
```

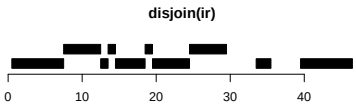

Figure 4: Plot of disjoined ranges.

```
          start       end     width
      <integer> <integer> <integer>
  [1]        -5        18        24
  [2]         5        16        12
  [3]        11        22        12
  [4]         7        36        30
  [5]        16        27        12
  [6]        33        36         4
  [7]        36        49        14
```

WARNING: The semantic of these arithmetic operators might be revisited at some point. Please restrict their use to the context of interactive visualization (where they arguably provide some convenience) but avoid to use them programmatically.

2.7.2 Making ranges disjoint

A more complex type of operation is making a set of ranges disjoint, *i.e.* non-overlapping. For example, `threebands` returns a disjoint set of three ranges for each input range.

The `disjoin` function makes an *IRanges* object disjoint by fragmenting it into the widest ranges where the set of overlapping ranges is the same.

```
> disjoin(ir)

IRanges object with 10 ranges and 0 metadata columns:
           start       end     width
       <integer> <integer> <integer>
   [1]         1         7         7
   [2]         8        12         5
   [3]        13        13         1
   [4]        14        14         1
   [5]        15        18         4
   [6]        19        19         1
   [7]        20        24         5
   [8]        25        29         5
   [9]        34        35         2
  [10]        40        46         7

> plotRanges(disjoin(ir))
```

A variant of `disjoin` is `disjointBins`, which divides the ranges into bins, such that the ranges in each bin are disjoint. The return value is an integer vector of the bins.

```
> disjointBins(ir)

[1] 1 2 1 2 3 1 1
```

2.7.3 Other transformations

Other transformations include `reflect` and `flank`. The former "flips" each range within a set of common reference bounds.

```
> reflect(ir, IRanges(start(ir), width=width(ir)*2))

IRanges object with 7 ranges and 0 metadata columns:
          start       end     width
      <integer> <integer> <integer>
  [1]        13        24        12
  [2]        14        19         6
  [3]        20        25         6
  [4]        30        44        15
  [5]        25        30         6
  [6]        36        37         2
  [7]        47        53         7
```

The `flank` returns ranges of a specified width that flank, to the left (default) or right, each input range. One use case of this is forming promoter regions for a set of genes.

```
> flank(ir, width=seq_len(length(ir)))

IRanges object with 7 ranges and 0 metadata columns:
          start       end     width
      <integer> <integer> <integer>
  [1]         0         0         1
  [2]         6         7         2
  [3]        11        13         3
  [4]        11        14         4
  [5]        14        18         5
  [6]        28        33         6
  [7]        33        39         7
```

2.8 Set Operations

Sometimes, it is useful to consider an *IRanges* object as a set of integers, although there is always an implicit ordering. This is formalized by *NormalIRanges*, above, and we now present versions of the traditional mathematical set operations *complement*, *union*, *intersect*, and *difference* for *IRanges* objects. There are two variants for each operation. The first treats each *IRanges* object as a set and returns a *normal* value, while the other has a "parallel" semantic like `pmin`/`pmax` and performs the operation for each range pairing separately.

The *complement* operation is implemented by the `gaps` and `pgap` functions. By default, `gaps` will return the ranges that fall between the ranges in the (normalized) input. It is possible to specify a set of bounds, so that flanking ranges are included.

```
> gaps(ir, start=1, end=50)
```

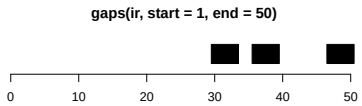

Figure 5: Plot of gaps from ranges.

```
IRanges object with 3 ranges and 0 metadata columns:
          start       end     width
      <integer> <integer> <integer>
  [1]        30        33         4
  [2]        36        39         4
  [3]        47        50         4
```

```
> plotRanges(gaps(ir, start=1, end=50), c(1,50))
```

`pgap` considers each parallel pairing between two *IRanges* objects and finds the range, if any, between them. Note that the function name is singular, suggesting that only one range is returned per range in the input.

The remaining operations, *union*, *intersect* and *difference* are implemented by the `[p]union`, `[p]intersect` and `[p]setdiff` functions, respectively. These are relatively self-explanatory.

3 Vector Views

The *IRanges* package provides the virtual *Views* class, which stores a vector-like object, referred to as the "subject", together with an *IRanges* object defining ranges on the subject. Each range is said to represent a *view* onto the subject.

Here, we will demonstrate the *RleViews* class, where the subject is of class *Rle*. Other *Views* implementations exist, such as *XStringViews* in the *Biostrings* package.

3.1 Creating Views

There are two basic constructors for creating views: the `Views` function based on indicators and the `slice` based on numeric boundaries.

```
> xViews <- Views(xRle, xRle >= 1)
> xViews <- slice(xRle, 1)
> xRleList <- RleList(xRle, 2L * rev(xRle))
> xViewsList <- slice(xRleList, 1)
```

Note that *RleList* objects will be covered later in more details in the "Lists of Atomic Vectors" section of this document.

3.2 Aggregating Views

While `sapply` can be used to loop over each window, the native functions `viewMaxs`, `viewMins`, `viewSums`, and `viewMeans` provide fast looping to calculate their respective statistical summaries.

```
> head(viewSums(xViews))

[1] 1 1 1 1 1 2

> viewSums(xViewsList)

IntegerList of length 2
[[1]] 1 1 1 1 1 2 1 1 2 3 1 6 1 3 4 ... 12 6 37 10 8 11 6 4 5 1 1 5 1 1
[[2]] 2 2 10 2 2 10 8 12 22 16 20 74 12 ... 2 12 2 6 4 2 2 4 2 2 2 2 2

> head(viewMaxs(xViews))

[1] 1 1 1 1 1 2

> viewMaxs(xViewsList)

IntegerList of length 2
[[1]] 1 1 1 1 1 2 1 1 1 2 1 2 1 2 3 1 ... 3 5 2 5 6 2 8 3 2 2 1 1 2 1 1
[[2]] 2 2 4 2 2 4 4 6 16 4 12 10 4 10 6 ... 4 2 4 2 4 2 2 2 4 2 2 2 2 2
```

4 Lists of Atomic Vectors

In addition to the range-based objects described in the previous sections, the *IRanges* package provides containers for storing lists of atomic vectors such as *integer* or *Rle* objects. The *IntegerList* and *RleList* classes represent lists of *integer* vectors and *Rle* objects respectively. They are subclasses of the *AtomicList* virtual class which is itself a subclass of the *List* virtual class defined in the *S4Vectors* package.

```
> showClass("RleList")

Virtual Class "RleList" [package "IRanges"]

Slots:

Name:        elementType   elementMetadata          metadata
Class:         character DataFrame_OR_NULL              list

Extends:
Class "AtomicList", directly
Class "List", by class "AtomicList", distance 2
Class "Vector", by class "AtomicList", distance 3
Class "list_OR_List", by class "AtomicList", distance 3
Class "Annotated", by class "AtomicList", distance 4
Class "vector_OR_Vector", by class "AtomicList", distance 4

Known Subclasses: "SimpleRleList", "RleViews", "CompressedRleList"
```

As the class definition above shows, the *RleList* class is virtual with subclasses *SimpleRleList* and *CompressedRleList*. A *SimpleRleList* class uses an ordinary *R* list to store the underlying elements and the *CompressedRleList* class stores the elements in an unlisted form and keeps track of where the element breaks are. The former "simple list" class is useful when the *Rle* elements are long and the latter "compressed list" class is useful when the list is long and/or sparse (i.e. a number of the list elements have length 0).

In fact, all of the atomic vector types (logical, integer, numeric, complex, character, raw, and factor) have similar list classes that derive from the *List* virtual class. For example, there is an *IntegerList* virtual class with subclasses *SimpleIntegerList* and *CompressedIntegerList*.

Each of the list classes for atomic sequences, be they stored as vectors or *Rle* objects, have a constructor function with a name of the appropriate list virtual class, such as *IntegerList*, and an optional argument `compress` that takes an argument to specify whether or not to create the simple list object type or the compressed list object type. The default is to create the compressed list object type.

```
> args(IntegerList)

function (..., compress = TRUE)
NULL

> cIntList1 <- IntegerList(x=xVector, y=yVector)
> cIntList1

IntegerList of length 2
[["x"]] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ... 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[["y"]] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ... 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

> sIntList2 <- IntegerList(x=xVector, y=yVector, compress=FALSE)
> sIntList2

IntegerList of length 2
[["x"]] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ... 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[["y"]] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ... 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

> ## sparse integer list
> xExploded <- lapply(xVector[1:5000], function(x) seq_len(x))
> cIntList2 <- IntegerList(xExploded)
> sIntList2 <- IntegerList(xExploded, compress=FALSE)
> object.size(cIntList2)

33208 bytes

> object.size(sIntList2)

294016 bytes
```

The `length` function returns the number of elements in a *Vector*-derived object and, for a *List*-derived object like “simple list" or “compressed list", the `lengths` function returns an integer vector containing the lengths of each of the elements:

```
> length(cIntList2)

[1] 5000

> Rle(lengths(cIntList2))

integer-Rle of length 5000 with 427 runs
  Lengths:  780    1  208    1 1599    1 ...    1    1    1    1    1
  Values :    0    1    0    1    0    1 ...   10    9    6    9   12
```

Just as with ordinary *R list* objects, *List*-derived object support the `[[` for element extraction, `c` for concatenating, and `lapply`/`sapply` for looping. When looping over sparse lists, the "compressed list" classes can be much faster during computations since only the non-empty elements are looped over during the `lapply`/`sapply` computation and all the empty elements are assigned the appropriate value based on their status.

```
> system.time(sapply(xExploded, mean))

   user  system elapsed
  0.021   0.000   0.021

> system.time(sapply(sIntList2, mean))

   user  system elapsed
  0.022   0.000   0.022

> system.time(sapply(cIntList2, mean))

   user  system elapsed
  0.023   0.000   0.023

> identical(sapply(xExploded, mean), sapply(sIntList2, mean))

[1] TRUE

> identical(sapply(xExploded, mean), sapply(cIntList2, mean))

[1] TRUE
```

Unlike ordinary *R list* objects, *AtomicList* objects support the `Ops` (e.g. `+`, `==`, `&`), `Math` (e.g. `log`, `sqrt`), `Math2` (e.g. `round`, `signif`), `Summary` (e.g. `min`, `max`, `sum`), and `Complex` (e.g. `Re`, `Im`) group generics.

```
> xRleList > 0

RleList of length 2
[[1]]
logical-Rle of length 10000000 with 197127 runs
  Lengths:   780     1   208     1  1599 ...     1    91     1   927
  Values : FALSE  TRUE FALSE  TRUE FALSE ...  TRUE FALSE  TRUE FALSE

[[2]]
logical-Rle of length 10000000 with 197127 runs
  Lengths:   927     1    91     1     5 ...     1   208     1   780
  Values : FALSE  TRUE FALSE  TRUE FALSE ...  TRUE FALSE  TRUE FALSE

> yRleList <- RleList(yRle, 2L * rev(yRle))
> xRleList + yRleList

RleList of length 2
[[1]]
integer-Rle of length 10000000 with 1957707 runs
  Lengths: 780   1 208   1  13   1 413 ...   5   1  91   1 507   1 419
  Values :   0   1   0   1   0   1   0 ...   0   1   0   1   0   1   0

[[2]]
integer-Rle of length 10000000 with 1957707 runs
  Lengths: 419   1 507   1  91   1   5 ... 413   1  13   1 208   1 780
```

```
  Values :   0   2   0   2   0   2   0 ...   0   2   0   2   0   2   0

> sum(xRleList > 0 | yRleList > 0)

[1] 2105185 2105185
```

Since these atomic lists inherit from *List*, they can also use the looping function `endoapply` to perform endomorphisms.

```
> safe.max <- function(x) { if(length(x)) max(x) else integer(0) }
> endoapply(sIntList2, safe.max)

IntegerList of length 5000
[[1]] integer(0)
[[2]] integer(0)
[[3]] integer(0)
[[4]] integer(0)
[[5]] integer(0)
[[6]] integer(0)
[[7]] integer(0)
[[8]] integer(0)
[[9]] integer(0)
[[10]] integer(0)
...
<4990 more elements>

> endoapply(cIntList2, safe.max)

IntegerList of length 5000
[[1]] integer(0)
[[2]] integer(0)
[[3]] integer(0)
[[4]] integer(0)
[[5]] integer(0)
[[6]] integer(0)
[[7]] integer(0)
[[8]] integer(0)
[[9]] integer(0)
[[10]] integer(0)
...
<4990 more elements>

> endoapply(sIntList2, safe.max)[[1]]

integer(0)
```

5 Session Information

Here is the output of `sessionInfo()` on the system on which this document was compiled:

```
R version 4.4.1 (2024-06-14)
Platform: x86_64-pc-linux-gnu
Running under: Ubuntu 24.04.1 LTS
```

```
Matrix products: default
BLAS:   /usr/lib/x86_64-linux-gnu/openblas-pthread/libblas.so.3
LAPACK: /usr/lib/x86_64-linux-gnu/openblas-pthread/libopenblasp-r0.3.26.so;  LAPACK version 3.12.0

locale:
 [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
 [3] LC_TIME=en_US.UTF-8        LC_COLLATE=C
 [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
 [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
 [9] LC_ADDRESS=C               LC_TELEPHONE=C
[11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C

time zone: Etc/UTC
tzcode source: system (glibc)

attached base packages:
[1] stats4    stats     graphics  grDevices utils     datasets
[7] methods   base

other attached packages:
[1] IRanges_2.41.0      S4Vectors_0.43.2    BiocGenerics_0.53.0

loaded via a namespace (and not attached):
 [1] digest_0.6.37       fastmap_1.2.0       xfun_0.48
 [4] maketools_1.3.1     knitr_1.48          htmltools_0.5.8.1
 [7] rmarkdown_2.28      buildtools_1.0.0    cli_3.6.3
[10] compiler_4.4.1      sys_3.4.3           tools_4.4.1
[13] evaluate_1.0.1      yaml_2.3.10         BiocManager_1.30.25
[16] rlang_1.1.4         BiocStyle_2.33.1
```