Representing sequencing data in R and Bioconductor

Mark Dunning

Last modified: 22 Jul 2015

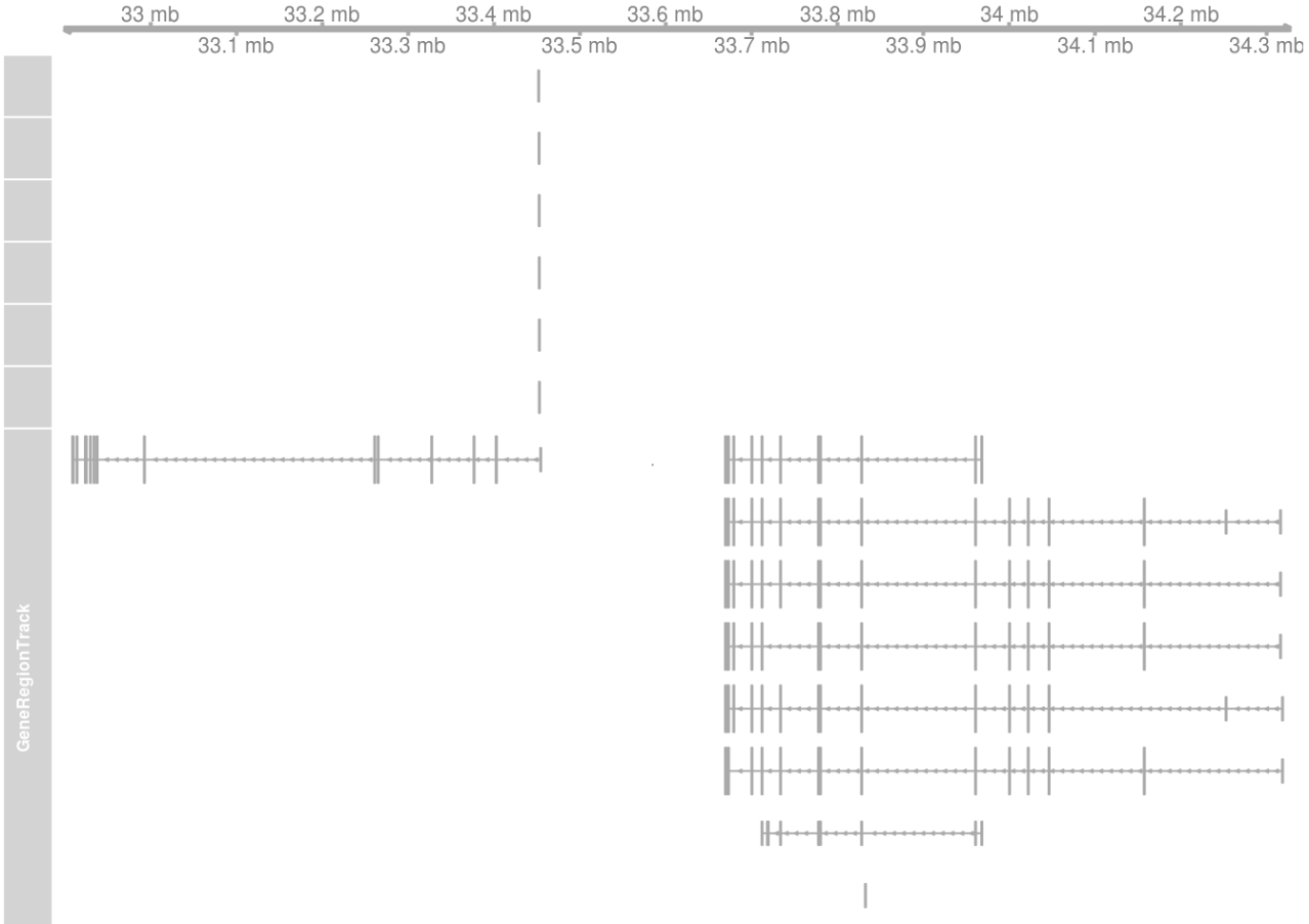
Overview

Aims

- By the end of this session you should be familiar with
- How to create and compare genomic intervals
- How DNA sequences are represented in R
- How to read bam files into R
- Interactions between the packages

Motivation

- We will often want to find information about the genomic region around the reads
    - genes, transcripts, exons
    - genomic sequence



Core data-type 1: Genome Intervals

IRanges

- A Genome is typically represented as linear sequence
- Ranges are an ordered set of consecutive integers defined by a start and end position
  - start $\leq$ end
- Ranges are a common scaffold for many genomic analyses
- Ranges can be associated with genomic information (e.g. gene name) or data derived from analysis (e.g. counts)
- The `IRanges` package in Bioconductor allows us to work with intervals
  - one of the aims of Bioconductor is to encourage core object-types and functions
  - `IRanges` is an example of this

IRanges is crucial for many packages

Just some of the packages that *depend* on IRanges (http://bioconductor.org/packages/release/bioc/html/IRanges.html)

Depends On Me	pd.hc.g110, pd.hg.focus, pd.hg.u133.plus.2, pd.hg.u133a, pd.hg.u133a.2, pd.hg.u133a.tag, pd.hg.u133b, pd.hg.u219, pd.hg.u95a, pd.hg.u95av2, pd.hg.u95b, pd.hg.u95c, pd.hg.u95d, pd.hg.u95e, pd.hg18.60mer.expr, pd.ht.hg.u133.plus.pm, pd.ht.hg.u133a, pd.ht.mg.430a, pd.hta.2.0, pd.hu6800, pd.huex.1.0.st.v2, pd.hugene.1.0.st.v1, pd.hugene.1.1.st.v1, pd.hugene.2.0.st, pd.hugene.2.1.st, pd.maize, pd.mapping250k.nsp, pd.mapping250k.sty, pd.mapping50k.hind240, pd.mapping50k.xba240, pd.margene.1.0.st, pd.margene.1.1.st, pd.medgene.1.0.st, pd.medgene.1.1.st, pd.medicago, pd.mg.u74a, pd.mg.u74av2, pd.mg.u74b, pd.mg.u74bv2, pd.mg.u74c, pd.mg.u74cv2, pd.mirna.1.0, pd.mirna.2.0, pd.mirna.3.0, pd.mirna.4.0, pd.moe430a, pd.moe430b, pd.moex.1.0.st.v1, pd.mogene.1.0.st.v1, pd.mogene.1.1.st.v1, pd.mogene.2.0.st, pd.mogene.2.1.st, pd.mouse430.2, pd.mouse430a.2, pd.mta.1.0, pd.mu11ksuba, pd.mu11ksubb, pd.nugo.hs1a520180, pd.nugo.mm1a520177, pd.ovigene.1.0.st, pd.ovigene.1.1.st, pd.pae.g1a, pd.plasmodium.anopheles, pd.poplar, pd.porcine, pd.porgene.1.0.st, pd.porgene.1.1.st, pd.rabgene.1.0.st, pd.rabgene.1.1.st, pd.rae230a, pd.rae230b, pd.raex.1.0.st.v1, pd.ragene.1.0.st.v1, pd.ragene.1.1.st.v1, pd.ragene.2.0.st, pd.ragene.2.1.st, pd.rat230.2, pd.rcngene.1.0.st, pd.rcngene.1.1.st, pd.rg.u34a, pd.rg.u34b, pd.rg.u34c, pd.rhegene.1.0.st, pd.rhegene.1.1.st, pd.rhesus, pd.rice, pd.rjpgene.1.0.st, pd.rjpgene.1.1.st, pd.rn.u34, pd.rta.1.0, pd.rusgene.1.0.st, pd.rusgene.1.1.st, pd.s.aureus, pd.soybean, pd.soygene.1.0.st, pd.soygene.1.1.st, pd.sugar.cane, pd.tomato, pd.u133.x3p, pd.vitis.vinifera, pd.wheat, pd.x.laevis.2, pd.x.tropicalis, pd.xenopus.laevis, pd.yeast.2, pd.yg.s98, pd.zebgene.1.0.st, pd.zebgene.1.1.st, pd.zebrafish, pepStat, PING, proBAMr, PSICQUIC, R453Plus1Toolbox, RefNet, rfPred, rGADEM, rGREAT, RIPSeeker, rMAT, Rsamtools, scsR, segmentSeq, SGSeq, SNPlocs.Hsapiens.dbSNP.20090506, SNPlocs.Hsapiens.dbSNP.20100427, SNPlocs.Hsapiens.dbSNP.20101109, SNPlocs.Hsapiens.dbSNP.20110815, SNPlocs.Hsapiens.dbSNP.20111119, SNPlocs.Hsapiens.dbSNP.20120608, SNPlocs.Hsapiens.dbSNP141.GRCh38, SNPlocs.Hsapiens.dbSNP142.GRCh37, SomatiCA, TEQC, TitanCNA, triform, triplex, VariantTools, XtraSNPlocs.Hsapiens.dbSNP141.GRCh38, XVector
Imports Me	AllelicImbalance, annmap, ArrayExpressHTS, ballgown, bamsignals, BayesPeak, beadarray, Biostrings, biovizBase, BiSeq, BitSeq, BSgenome, BubbleTree, CAGEr, cgdv17, ChAMP, charm, chipenrich, chipenrich.data, ChIPQC, ChIPseeker, chipseq, ChIPseqR, ChIPsim, ChromHeatMap, cleaver, CNEr, CNVrd2, cobindR, coMET, compEpiTools, conumee, copynumber, CopywriteR, CoverageView, csaw, customProDB, DECIPHER, derfinder, derfinderHelper, derfinderPlot, DiffBind, diffHic, DOQTL, easyRNASeq, EDASeq, facopy, fastseq, flipflop, flowQ, FunciSNP, genomation, GenomicAlignments, GenomicInteractions, GenomicTuples, genoset, ggbio, GGtools, girafe, gmapR, GoogleGenomics, GOTHiC, gQTLstats, gwascat, h5vc, HTSeqGenie, InPAS, intansv, IVAS, M3D, MafDb.ALL.wgs.phase1.release.v3.20101123, MafDb.ALL.wgs.phase3.release.v5a.20130502, MafDb.ESP6500SI.V2.SSA137, MafDb.ExAC.r0.3.sites, MatrixRider, MEDIPS, methVisual, methyAnalysis, methylPipe, MethylSeekR, methylumi, minfi, MinimumDistance, MMDiff, mosaics, motifRG, MotIV, msa, MSnbase, NarrowPeaks, nucleR, oligoClasses, Pbase, pd.081229.hg18.promoter.medip.hx1, pd.2006.07.18.hg18.refseq.promoter, pd.2006.07.18.mm8.refseq.promoter, pd.2006.10.31.rn34.refseq.promoter, pd.atdschip.tiling, pd.charm.hg18.example, pd.feinberg.hg18.me.hx1, pd.feinberg.mm8.me.hx1, pd.mirna.3.1, pdInfoBuilder, phastCons100way.UCSC.hg19, phastCons7way.UCSC.hg38, PICS, PING, plethy, podkat, polyester, prebs, Pviz, qpgraph, QuasR, R3CPET, r3Cseq, Rariant, REDseq, regionReport, Repitools, ReportingTools, rGADEM, rMAT, rnaSeqMap, RnBeads, Rolexa, Rqc, rSFFreader, RSVSim, RTN, rtracklayer, SCAN.UPC, SegArray, segPattern, seqplots, SeqVarTools, ShortRead, skewr, SNPchip, SNPlocs.Hsapiens.dbSNP.20090506, SNPlocs.Hsapiens.dbSNP.20100427, SNPlocs.Hsapiens.dbSNP.20101109, SNPlocs.Hsapiens.dbSNP.20110815, SNPlocs.Hsapiens.dbSNP.20111119, SNPlocs.Hsapiens.dbSNP.20120608, SNPlocs.Hsapiens.dbSNP141.GRCh38, SNPlocs.Hsapiens.dbSNP142.GRCh37, soGGi, SomatiCA, SomaticCancerAlterations, SomaticSignatures, spliceR, SplicingGraphs, SVM2CRM, TFBSTools, tracktables, TransView, triform, TSSi, VanillaICE, VariantAnnotation, VariantFiltering, wavClusteR, waveTiling, XtraSNPlocs.Hsapiens.dbSNP141.GRCh38, XVector
Suggests Me	BaseSpaceR, BiocGenerics, gQTLBase, HilbertVis, HilbertVisGUI, MiRaGE, S4Vectors, STAN, yeastRNASeq

IRanges paper

OPEN ACCESS Freely available online

PLOS | COMPUTATIONAL BIOLOGY

Software for Computing and Annotating Genomic Ranges

Michael Lawrence[1]*, Wolfgang Huber[2,3], Hervé Pagès[4], Patrick Aboyoun[4], Marc Carlson[4], Robert Gentleman[1], Martin T. Morgan[4], Vincent J. Carey[5]

1 Bioinformatics and Computational Biology, Genentech, Inc., South San Francisco, California, United States of America, **2** European Molecular Biology Laboratory Genome Biology Unit, Heidelberg, Germany, **3** The European Bioinformatics Institute, Cambridge, United Kingdom, **4** Computational Biology, Fred Hutchinson Cancer Research Center, Seattle, Washington, United States of America, **5** Channing Division of Network Medicine, Brigham and Women's Hospital, Harvard Medical School, Boston, Massachusetts, United States of America

Abstract

We describe Bioconductor infrastructure for representing and computing on annotated genomic ranges and integrating genomic data with the statistical computing features of R and its extensions. At the core of the infrastructure are three packages: *IRanges*, *GenomicRanges*, and *GenomicFeatures*. These packages provide scalable data structures for representing annotated ranges on the genome, with special support for transcript structures, read alignments and coverage vectors. Computational facilities include efficient algorithms for overlap and nearest neighbor detection, coverage calculation and other range operations. This infrastructure directly supports more than 80 other Bioconductor packages, including those for sequence analysis, differential expression analysis and visualization.

Citation: Lawrence M, Huber W, Pagès H, Aboyoun P, Carlson M, et al. (2013) Software for Computing and Annotating Genomic Ranges. PLoS Comput Biol 9(8): e1003118. doi:10.1371/journal.pcbi.1003118

Editor: Andreas Prlic, University of California, San Diego, United States of America

Received January 28, 2013; **Accepted** May 7, 2013; **Published** August 8, 2013

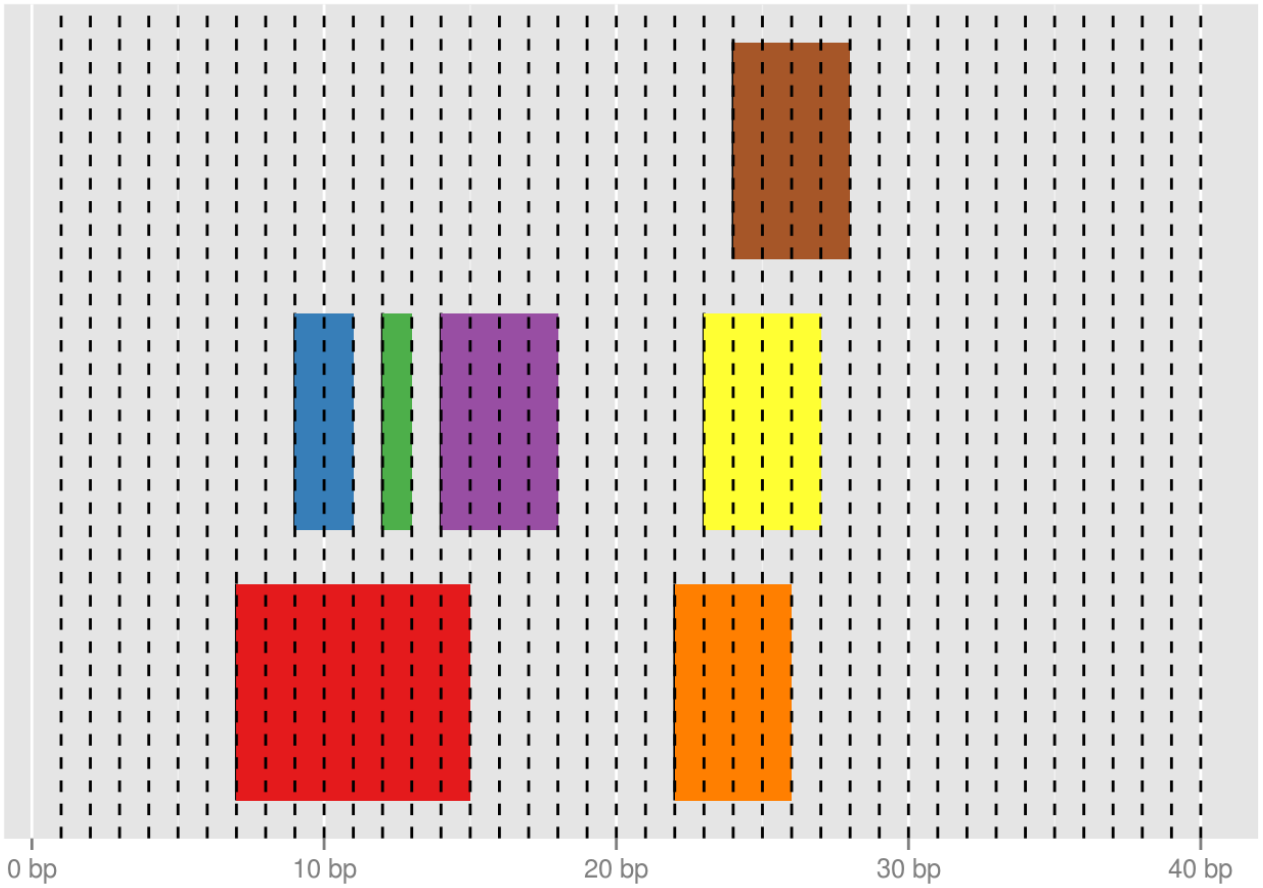
Funding: This work was funded by the National Institutes of Health, National Human Genome Research Group through grants P41 HG004059 and U41 HG004059 and (for VJC) by National Heart, Lung and Blood Institute grants R01 HL086601, R01 HL093076 and R01 HL094635. The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

Competing Interests: The authors have declared that no competing interests exist.

* E-mail: michafla@gene.com

Example

Suppose we want to capture information on the following intervals



Creating the object

- The `IRanges` function from the `IRanges` package is used to *construct* a new object
  - think `data.frame`, `vector` or `matrix`
  - it's structure is quite unlike anything we've seen before

```
library(IRanges)
ir <- IRanges(
start = c(7,9,12,14,22:24),
end=c(15,11,13,18,26,27,28))
str(ir)
```

```
## Formal class 'IRanges' [package "IRanges"] with 6 slots
##   ..@ start          : int [1:7] 7 9 12 14 22 23 24
##   ..@ width          : int [1:7] 9 3 2 5 5 5 5
##   ..@ NAMES          : NULL
##   ..@ elementType    : chr "integer"
##   ..@ elementMetadata: NULL
##   ..@ metadata       : list()
```

Display the object

- Typing the name of the object will print a summary of the object to the screen
  - useful compared to display methods for data frames, which print the whole object

- the square brackets `[]` should give a hint about how to access the data…

```
ir
```

```
## IRanges of length 7
##     start end width
## [1]     7  15     9
## [2]     9  11     3
## [3]    12  13     2
## [4]    14  18     5
## [5]    22  26     5
## [6]    23  27     5
## [7]    24  28     5
```

Ranges as vectors

- `IRanges` can be treated as if they were *vectors*
  - no new rules to learn
    - if we can subset vectors, we can subset ranges
  - vector operations are efficient
  - Remember, square brackets `[` to subset
  - Inside the brackets, put a numeric vector to specify the `indices` that you want values for
    - e.g. get the first two intervals in the object using the `:` shortcut

```
ir[1:2]
```

```
## IRanges of length 2
##     start end width
## [1]     7  15     9
## [2]     9  11     3
```

```
ir[c(2,4,6)]
```

```
## IRanges of length 3
##     start end width
## [1]     9  11     3
## [2]    14  18     5
## [3]    23  27     5
```

Accessing the object

- If we want to extract the properties of the object, the package authors have provided some useful functions
  - we call these *accessor* functions
  - We don't need to know the details of how the objects and implemented to access the data
  - the authors are free to change the implementation at any time
    - we shouldn't notice the difference

- the result is a vector with the same length as the number of intervals

```
start(ir)
```

```
## [1]  7  9 12 14 22 23 24
```

```
end(ir)
```

```
## [1] 15 11 13 18 26 27 28
```

```
width(ir)
```

```
## [1] 9 3 2 5 5 5 5
```

More-complex subsetting

- Recall that '*logical*' vectors can be used in subsetting
  - i.e. `TRUE` or `FALSE`
- Such a vector can be derived using a comparison operator
  - `<`, `>`, `==`

```
width(ir) == 5
```

```
## [1] FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
```

```
ir[width(ir)==5]
```

```
## IRanges of length 4
##     start end width
## [1]    14  18     5
## [2]    22  26     5
## [3]    23  27     5
## [4]    24  28     5
```

More-complex subsetting

```
start(ir) > 10
```

```
## [1] FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

```
end(ir) < 27
```

```
## [1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE
```

```
ir[start(ir) > 10]
```

```
## IRanges of length 5
##     start end width
## [1]    12  13     2
## [2]    14  18     5
## [3]    22  26     5
## [4]    23  27     5
## [5]    24  28     5
```

More-complex subsetting

- Multiple logical vectors can be combined using `&` (and), `|` (or)
  - eg intervals that start after 10, **and** before 27

```
ir[end(ir) < 27]
```

```
## IRanges of length 5
##     start end width
## [1]     7  15     9
## [2]     9  11     3
## [3]    12  13     2
## [4]    14  18     5
## [5]    22  26     5
```

```
ir[start(ir) > 10 & end(ir) < 27]
```

```
## IRanges of length 3
##     start end width
## [1]    12  13     2
## [2]    14  18     5
## [3]    22  26     5
```

Manipulating Ranges

Lots of common use-cases are implemented

Table 1. Summary of the Ranges API.

Category	Function	Description
Accessors	`start, end, width`	Get or set the starts, ends and widths
	`names`	Get or set the names
	`elementMetadata, metadata`	Get or set metadata on elements or object
	`length`	Number of ranges in the vector
	`range`	Range formed from min(start) and max(end)
Ordering	`<, <=, >, >=, ==, !=`	Compare ranges, ordering by start then width
	`sort, order, rank`	Sort by the ordering defined above
	`duplicated`	Find ranges with multiple instances
	`unique`	Find unique instances, removing duplicates
Arithmetic	`r+x, r-x, r * x`	Shrink or expand ranges `r` by number `x`
	`shift`	Move the ranges by specified amount
	`resize`	Change width, anchoring on start, end or mid
	`distance`	Separation between ranges (closest endpoints)
	`restrict`	Clamp ranges to within some start and end
	`flank`	Generate adjacent regions on start or end
Set operations	`reduce`	Merge overlapping and adjacent ranges
	`intersect, union, setdiff`	Set operations on reduced ranges
	`pintersect, punion, psetdiff`	Parallel set operations, on each `x[ i] , y[ i]`
	`gaps, pgap`	Find regions not covered by reduced ranges
	`disjoin`	Ranges formed from union of endpoints
Overlaps	`findOverlaps`	Find all overlaps for each `x` in `y`
	`countOverlaps`	Count overlaps of each `x` range in `y`
	`nearest`	Find nearest neighbors (closest endpoints)
	`precede, follow`	Find nearest `y` that `x` precedes or follows
	`x %in% y`	Find ranges in `x` that overlap range in `y`
Coverage	`coverage`	Count ranges covering each position
Extraction	`r[ i]`	Get or set by logical or numeric index
	`r[[ i]]`	Get integer sequence from `start[ i]` to `end[ i]`
	`subsetByOverlaps`	Subset `x` for those that overlap in `y`
	`head, tail, rev, rep`	Conventional R semantics
Split, combine	`split`	Split ranges by a factor into a *RangesList*
	`c`	Concatenate two or more range objects

Shifting

e.g. sliding windows

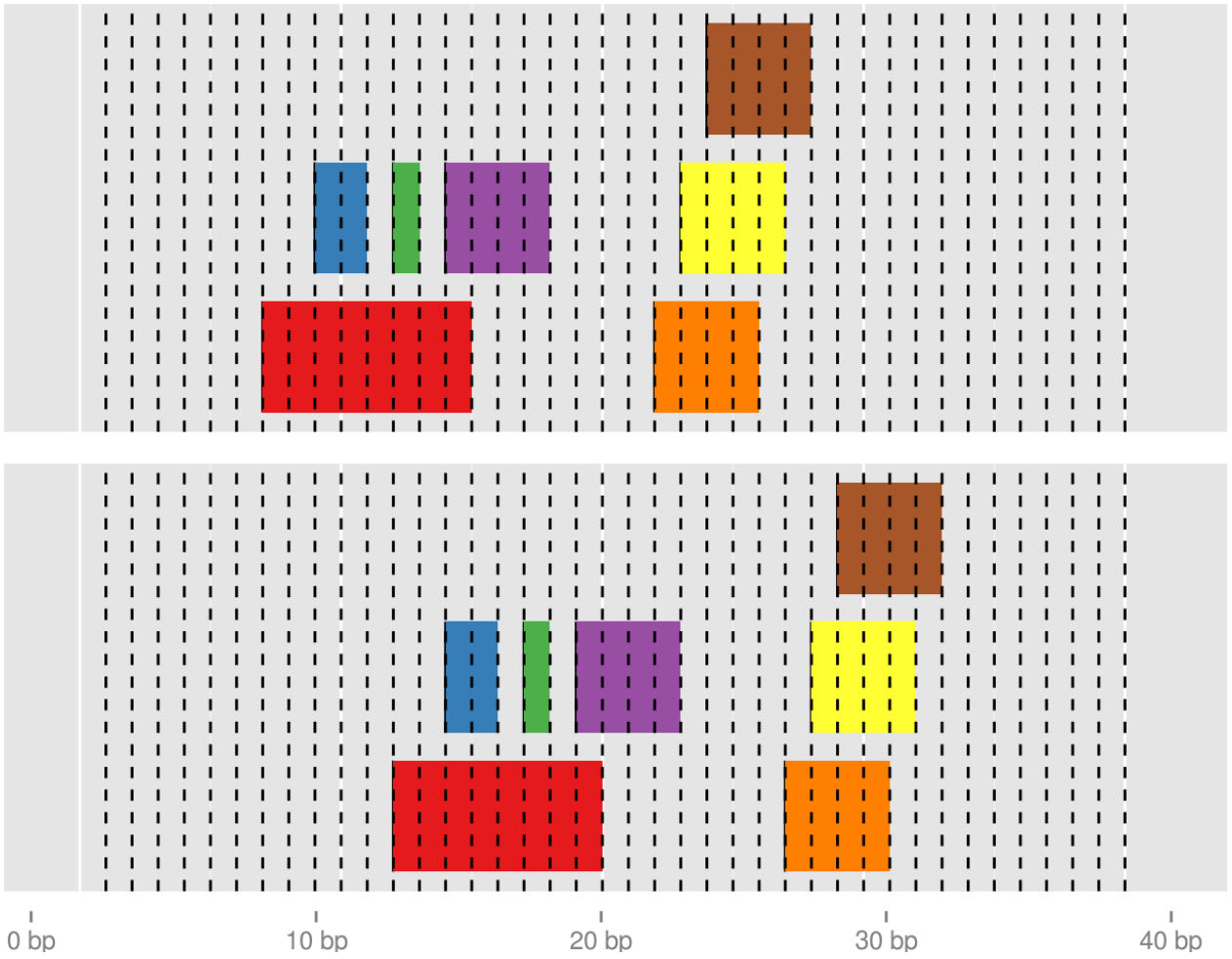
```
ir
```

```
## IRanges of length 7
##     start end width
## [1]     7  15     9
## [2]     9  11     3
## [3]    12  13     2
## [4]    14  18     5
## [5]    22  26     5
## [6]    23  27     5
## [7]    24  28     5
```

```
shift(ir, 5)
```

```
## IRanges of length 7
##     start end width
## [1]    12  20     9
## [2]    14  16     3
## [3]    17  18     2
## [4]    19  23     5
## [5]    27  31     5
## [6]    28  32     5
## [7]    29  33     5
```

Shifting



Shifting

Size of shift doesn't need to be constant

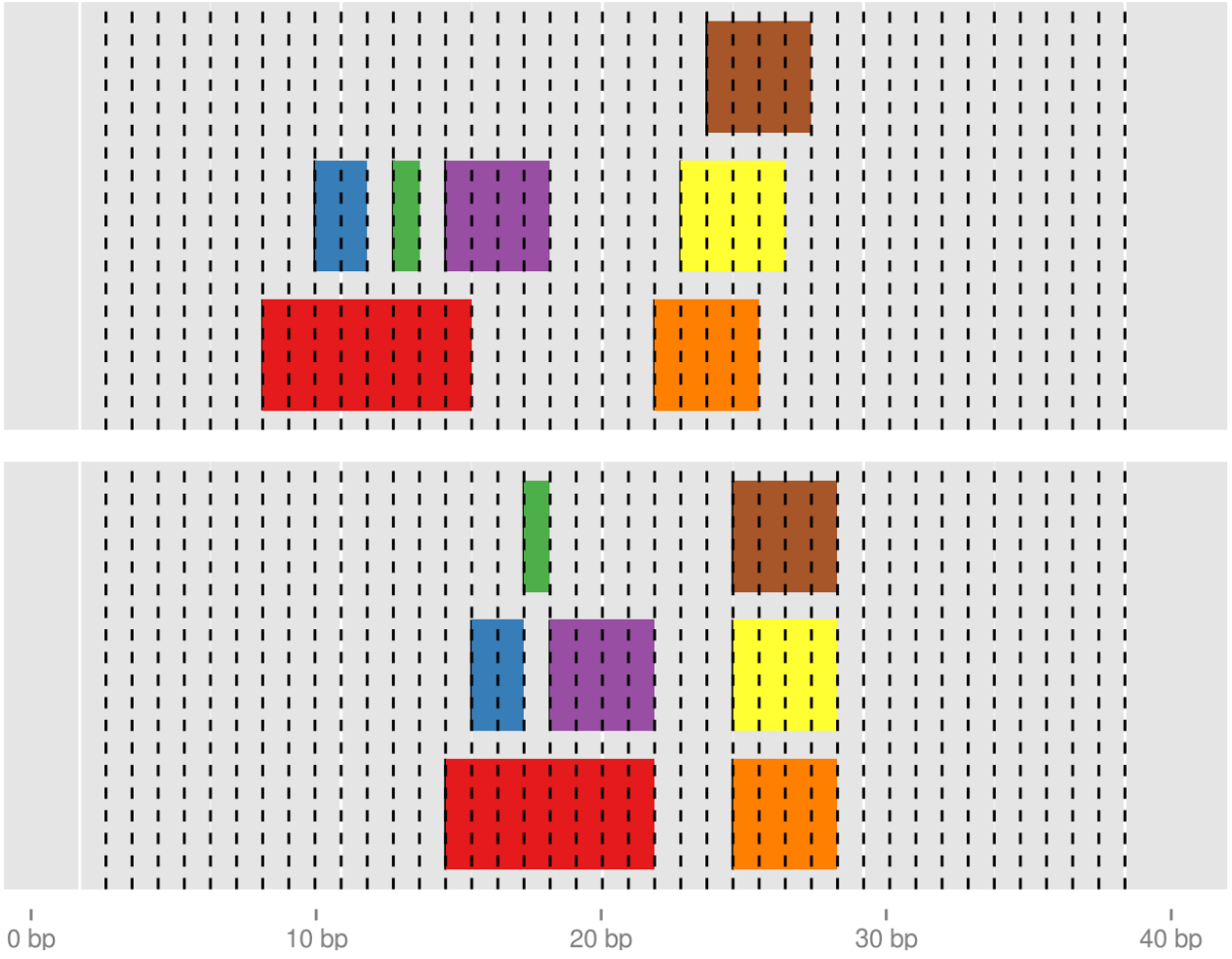
```
ir
```

```
## IRanges of length 7
##     start end width
## [1]     7  15     9
## [2]     9  11     3
## [3]    12  13     2
## [4]    14  18     5
## [5]    22  26     5
## [6]    23  27     5
## [7]    24  28     5
```

```
shift(ir, 7:1)
```

```
## IRanges of length 7
##     start end width
## [1]    14  22     9
## [2]    15  17     3
## [3]    17  18     2
## [4]    18  22     5
## [5]    25  29     5
## [6]    25  29     5
## [7]    25  29     5
```

Shifting



Resize

e.g. trimming reads

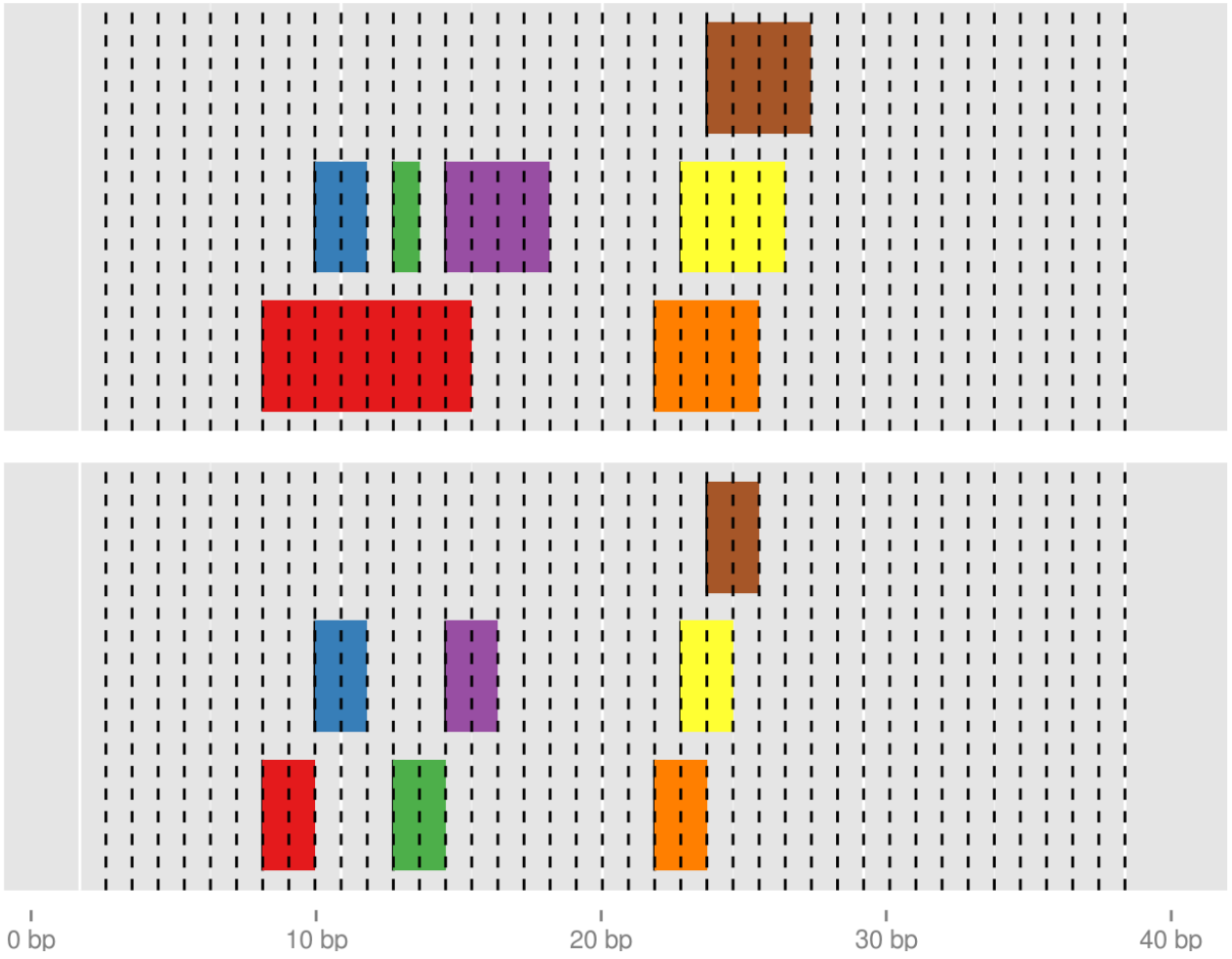
```
ir
```

```
## IRanges of length 7
##     start end width
## [1]     7  15     9
## [2]     9  11     3
## [3]    12  13     2
## [4]    14  18     5
## [5]    22  26     5
## [6]    23  27     5
## [7]    24  28     5
```

```
resize(ir,3)
```

```
## IRanges of length 7
##     start end width
## [1]     7   9     3
## [2]     9  11     3
## [3]    12  14     3
## [4]    14  16     3
## [5]    22  24     3
## [6]    23  25     3
## [7]    24  26     3
```

Resize



Coverage

- Often we want to know how much sequencing we have at particular positions
  - i.e. depth of coverage

`coverage` returns a *Run Length Encoding* - an efficient representation of repeated values

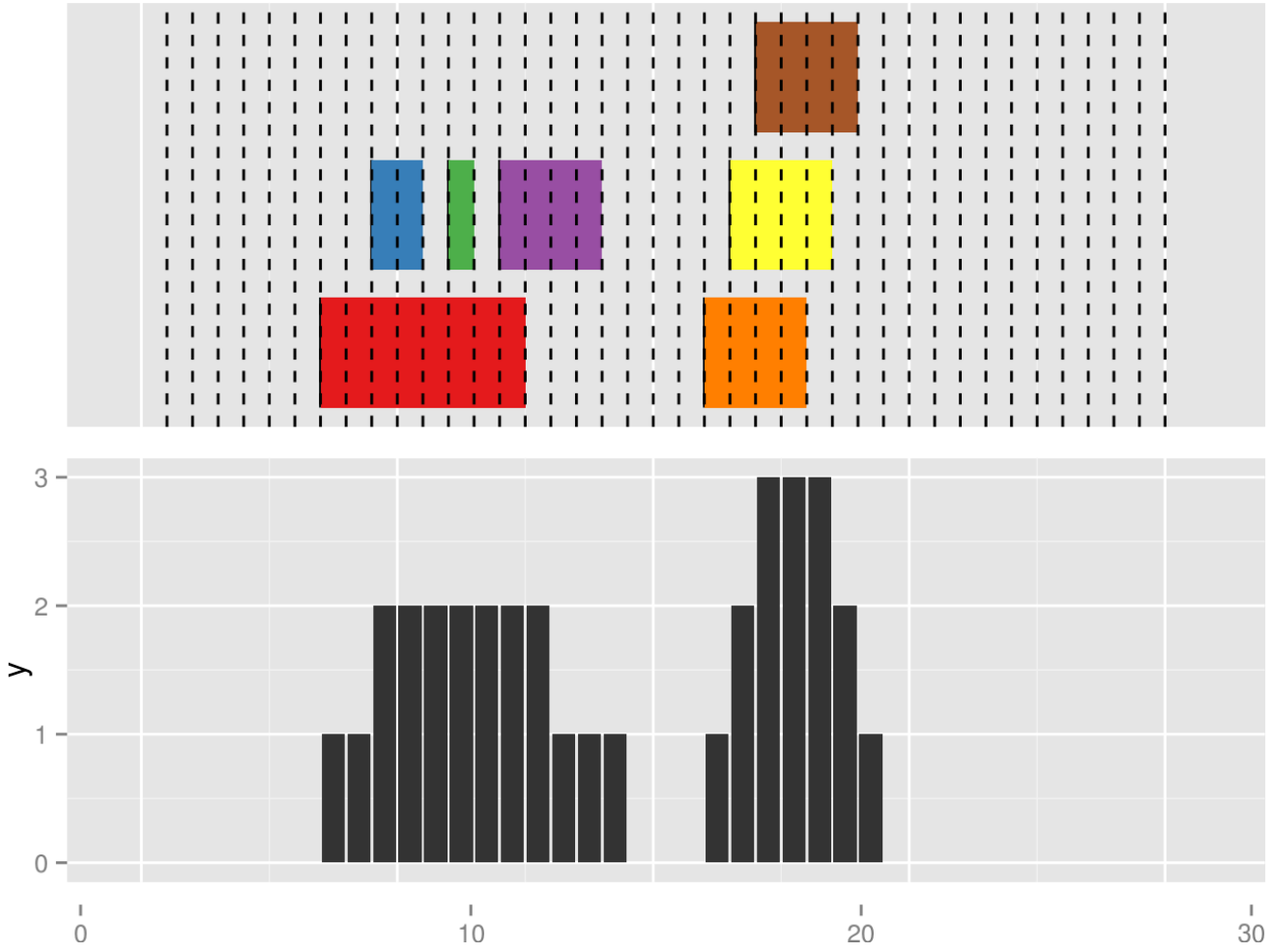
```
cvg <- coverage(ir)
cvg
```

```
## integer-Rle of length 28 with 10 runs
##   Lengths: 6 2 7 3 3 1 1 3 1 1
##   Values : 0 1 2 1 0 1 2 3 2 1
```

```
as.vector(cvg)
```

```
##  [1] 0 0 0 0 0 0 1 1 2 2 2 2 2 2 2 1 1 1 0 0 0 1 2 3 3 3 2 1
```

Coverage Results



Overlapping

e.g. counting - The terminology of overlapping defines a *query* and a *subject*

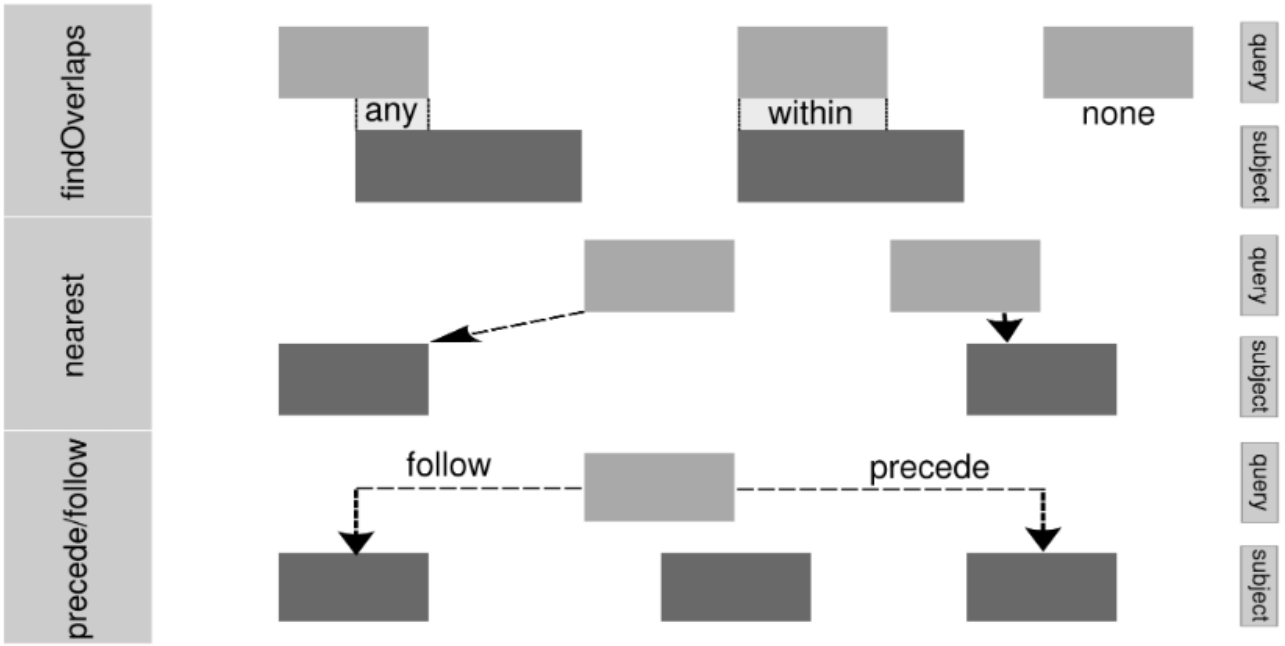


Figure 3. Illustration of overlap (top) and adjacency (bottom) relationships. The *any* mode detects hits with partial or complete overlap, while *within* requires that the query range represents a subregion of the subject range.
doi:10.1371/journal.pcbi.1003118.g003

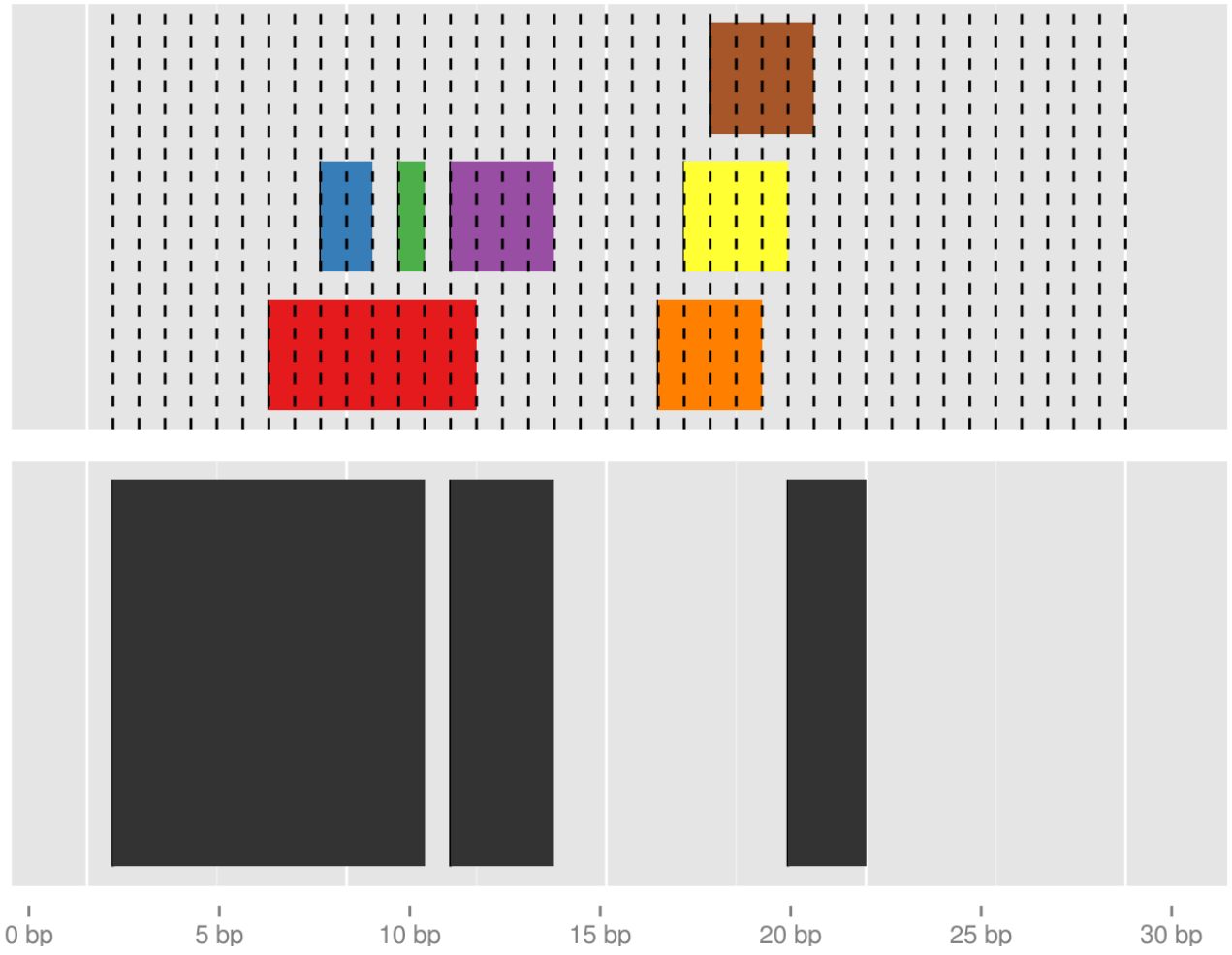
Overlaps

- lets start be defining a new set of ranges

```
ir3 <- IRanges(start = c(1, 14, 27), end = c(13,
    18, 30))
ir3
```

```
## IRanges of length 3
##     start end width
## [1]     1  13    13
## [2]    14  18     5
## [3]    27  30     4
```

Overlaps



Overlaps

- The `findOverlaps` function is used for overlap
  - the output isn't immediately obvious
  - length of output is the number of *hits*
    - each hit is defined by a subject and query index
  - require accessor functions to get the data; `queryHits` and `subjectHits`

```
query <- ir
subject <- ir3
ov <- findOverlaps(query, subject)
ov
```

```
## Hits object with 7 hits and 0 metadata columns:
##       queryHits subjectHits
##       <integer>   <integer>
##   [1]         1           1
##   [2]         1           2
##   [3]         2           1
##   [4]         3           1
##   [5]         4           2
##   [6]         6           3
##   [7]         7           3
##   -------
##   queryLength: 7
##   subjectLength: 3
```

queryHits

- `queryHits` returns *indices* from the ***query***
  - each query may overlap with many in the subject

```
queryHits(ov)
```

```
## [1] 1 1 2 3 4 6 7
```

- `subjectHits` returns *indices* from the ***subject***
  - each subject range may overlap with many in the query

```
subjectHits(ov)
```

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

- e.g. 1 from the query overlaps with 1 from the subject

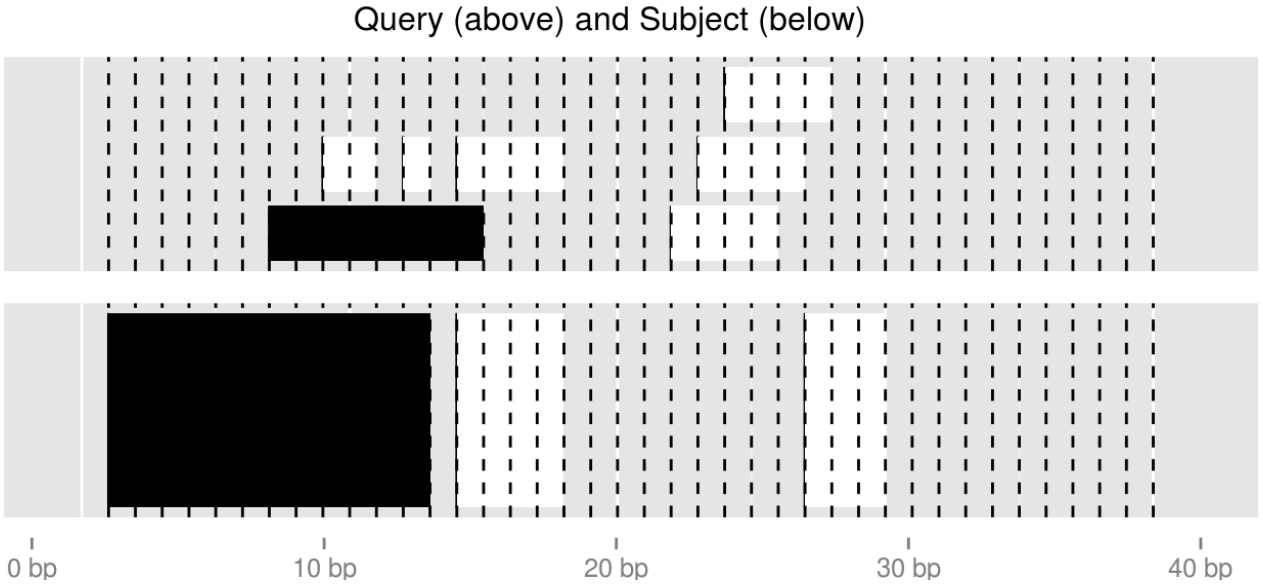
Overlap example - First hit

```
query[queryHits(ov)[1]]
```

```
## IRanges of length 1
##     start end width
## [1]     7  15     9
```

```
subject[subjectHits(ov)[1]]
```

```
## IRanges of length 1
##     start end width
## [1]     1  13    13
```



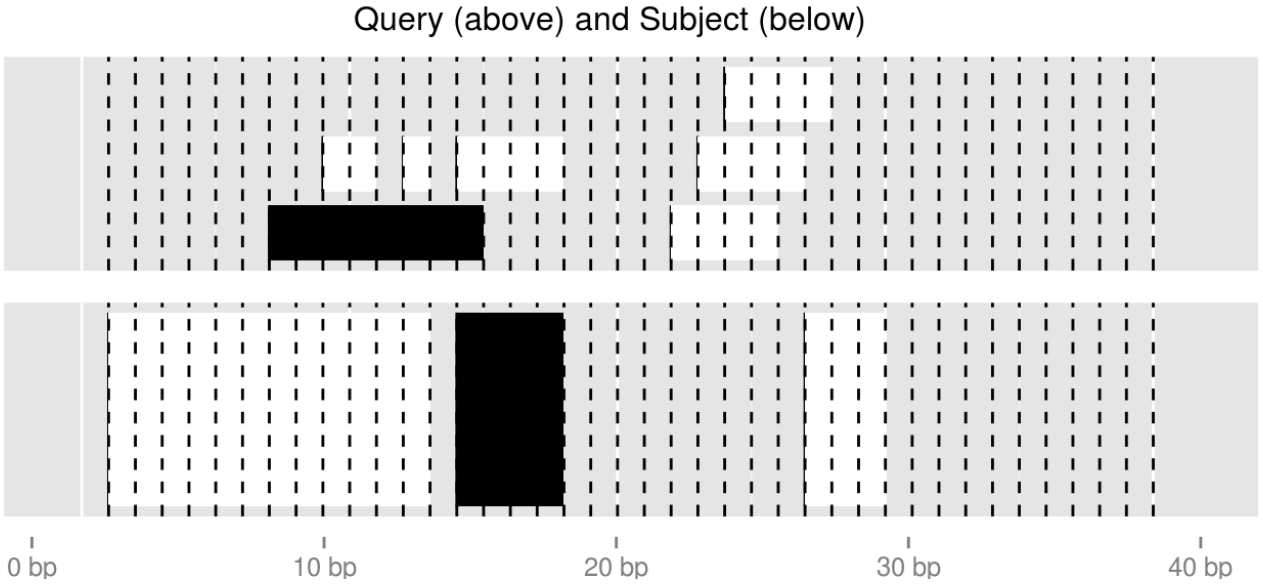
Overlap example - second hit

```
query[queryHits(ov)[2]]
```

```
## IRanges of length 1
##     start end width
## [1]     7  15     9
```

```
subject[subjectHits(ov)[2]]
```

```
## IRanges of length 1
##     start end width
## [1]    14  18     5
```



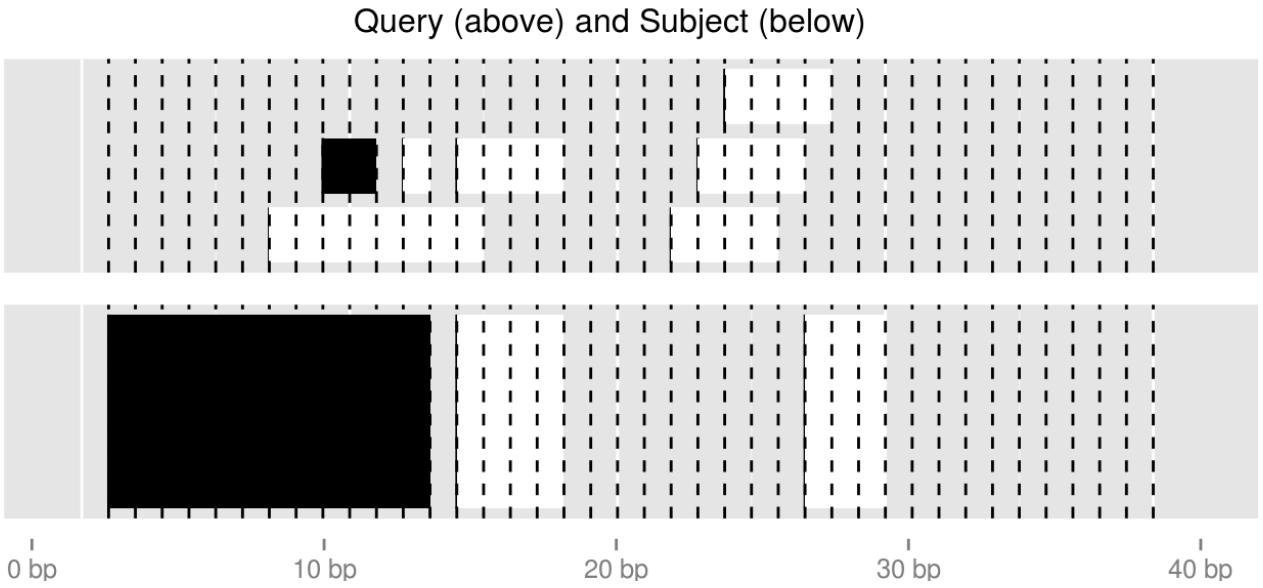
Overlap example - Third hit

```
query[queryHits(ov)[3]]
```

```
## IRanges of length 1
##     start end width
## [1]     9  11     3
```

```
subject[subjectHits(ov)[3]]
```

```
## IRanges of length 1
##     start end width
## [1]     1  13    13
```



Counting

- If we just wanted to count the number of overlaps for each range, we can use `countOverlaps`
  - result is a vector with length the number of intervals in query
  - e.g. interval 1 in the query overlaps with 2 intervals in the subject

```
countOverlaps(query,subject)
```

```
## [1] 2 1 1 1 0 1 1
```

- Order of arguments is important

```
countOverlaps(subject,query)
```

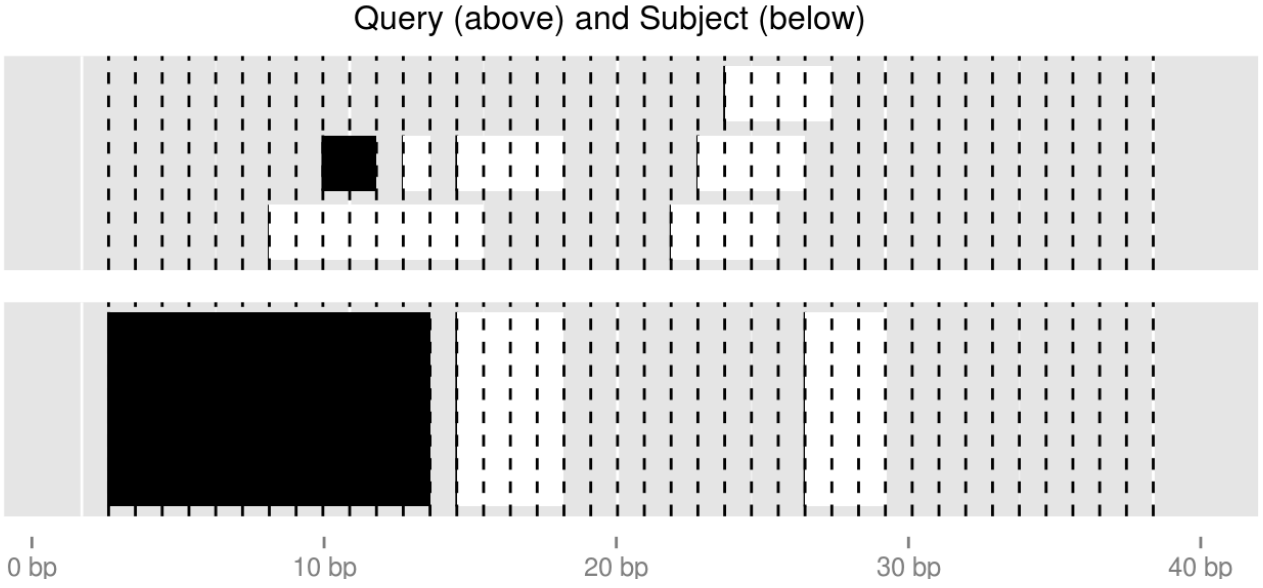
```
## [1] 3 2 2
```

Modify overlap criteria

- There are various ways of defining an overlap
- We can be more stringent by stating that all positions need to be in common

```
findOverlaps(query,subject,type="within")
```

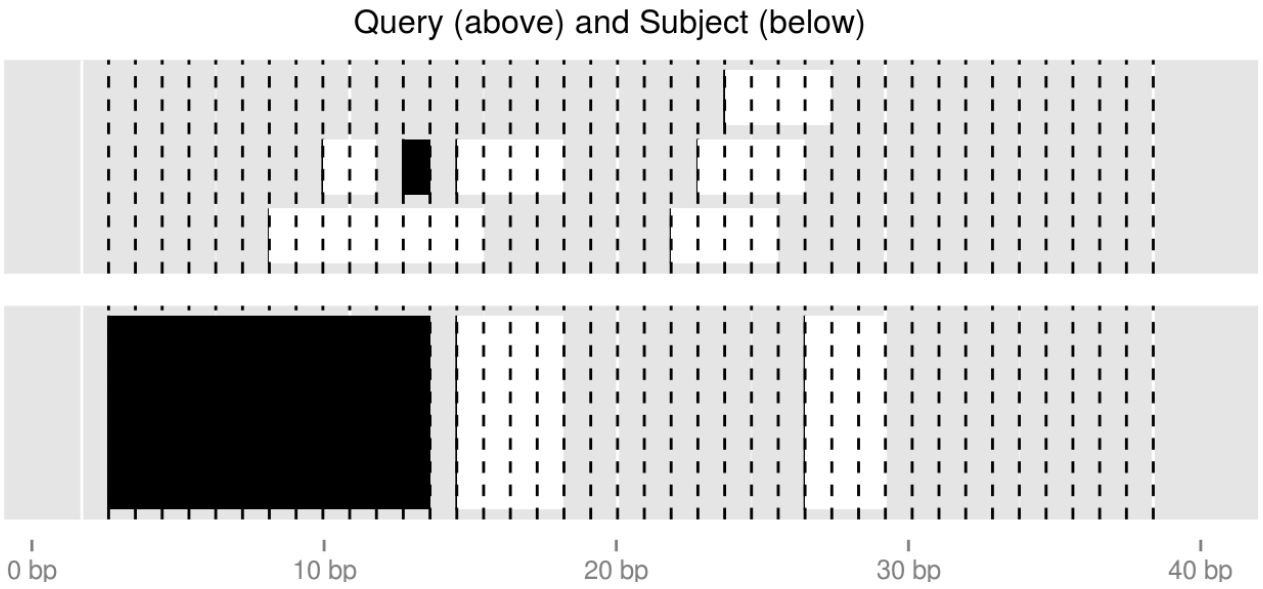
```
## Hits object with 3 hits and 0 metadata columns:
##       queryHits subjectHits
##       <integer>   <integer>
##   [1]         2           1
##   [2]         3           1
##   [3]         4           2
##   -------
##   queryLength: 7
##   subjectLength: 3
```



More stringent overlap

```
findOverlaps(query,subject,type="within")
```

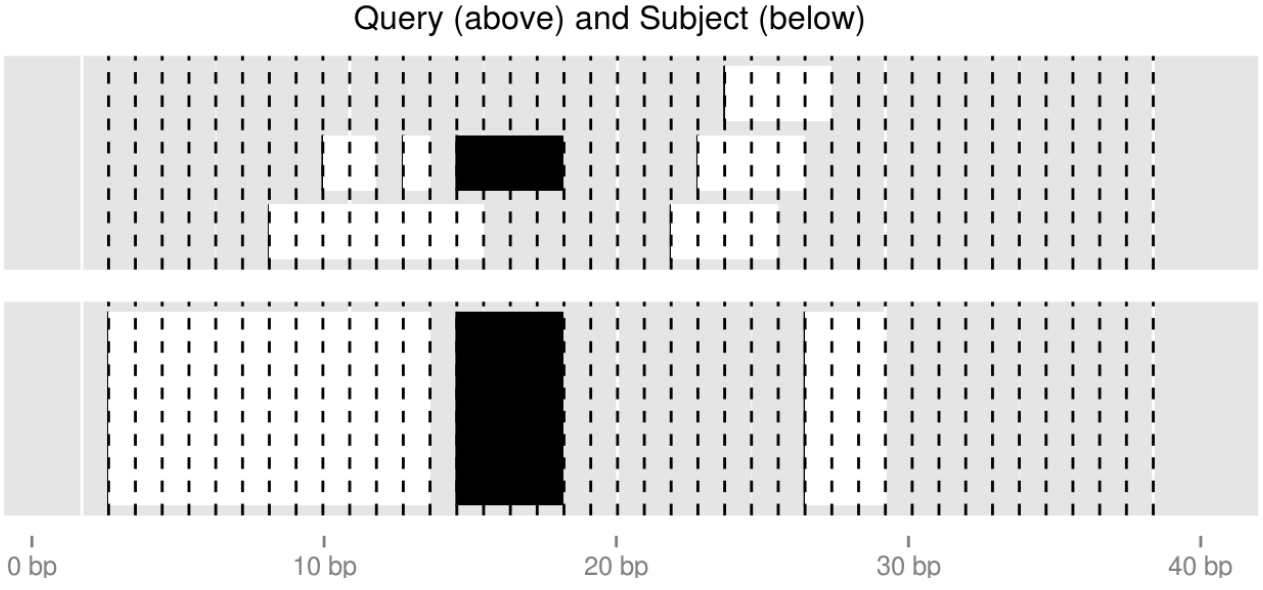
```
## Hits object with 3 hits and 0 metadata columns:
##       queryHits subjectHits
##       <integer>   <integer>
##   [1]         2           1
##   [2]         3           1
##   [3]         4           2
##   -------
##   queryLength: 7
##   subjectLength: 3
```



More strigent overlap

```
findOverlaps(query,subject,type="within")
```

```
## Hits object with 3 hits and 0 metadata columns:
##       queryHits subjectHits
##       <integer>   <integer>
##   [1]         2           1
##   [2]         3           1
##   [3]         4           2
##   -------
##   queryLength: 7
##   subjectLength: 3
```

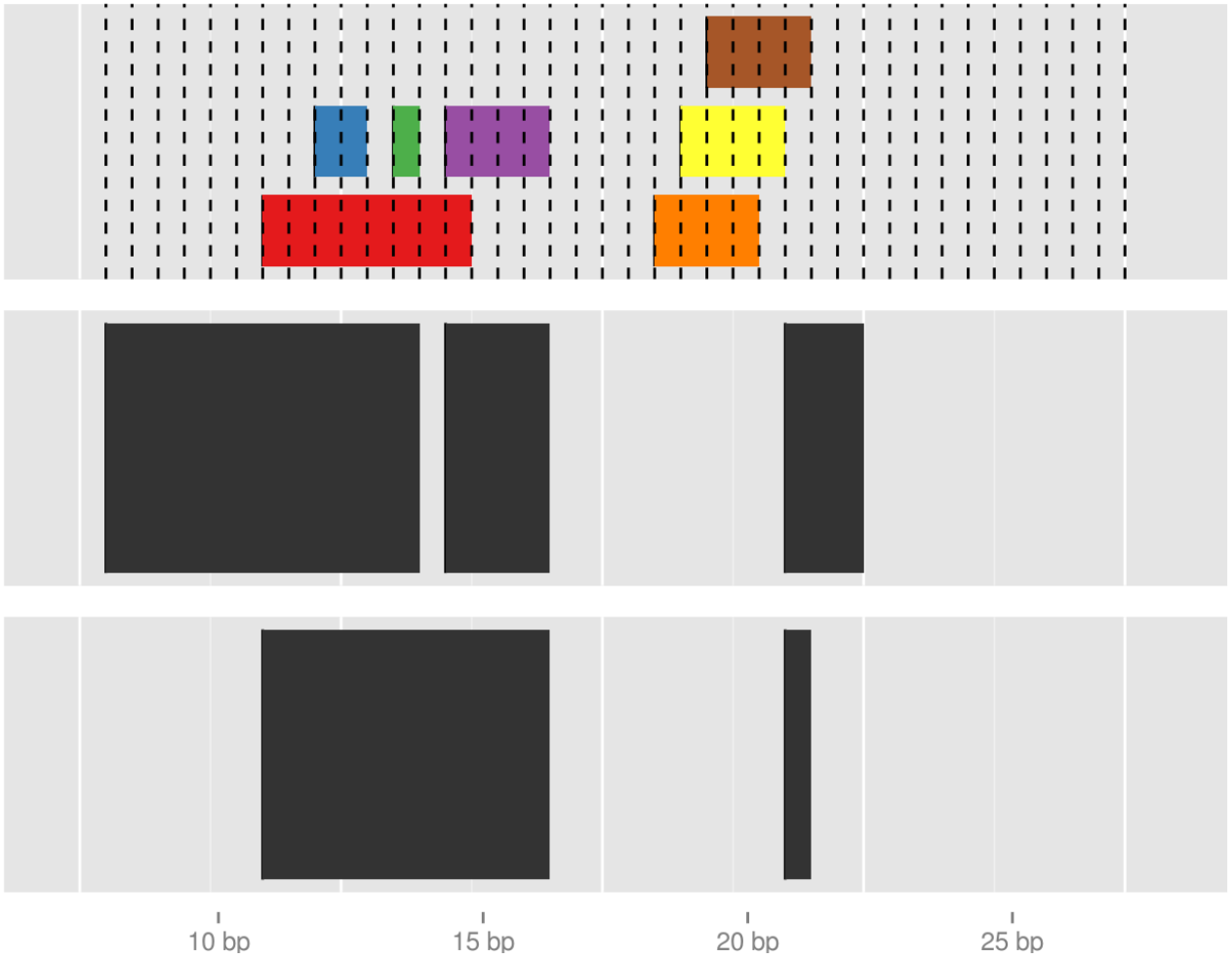


Intersection

- Rather than counting, we might want to know which positions are in common

```
intersect(ir,ir3)
```

```
## IRanges of length 2
##     start end width
## [1]     7  18    12
## [2]    27  28     2
```

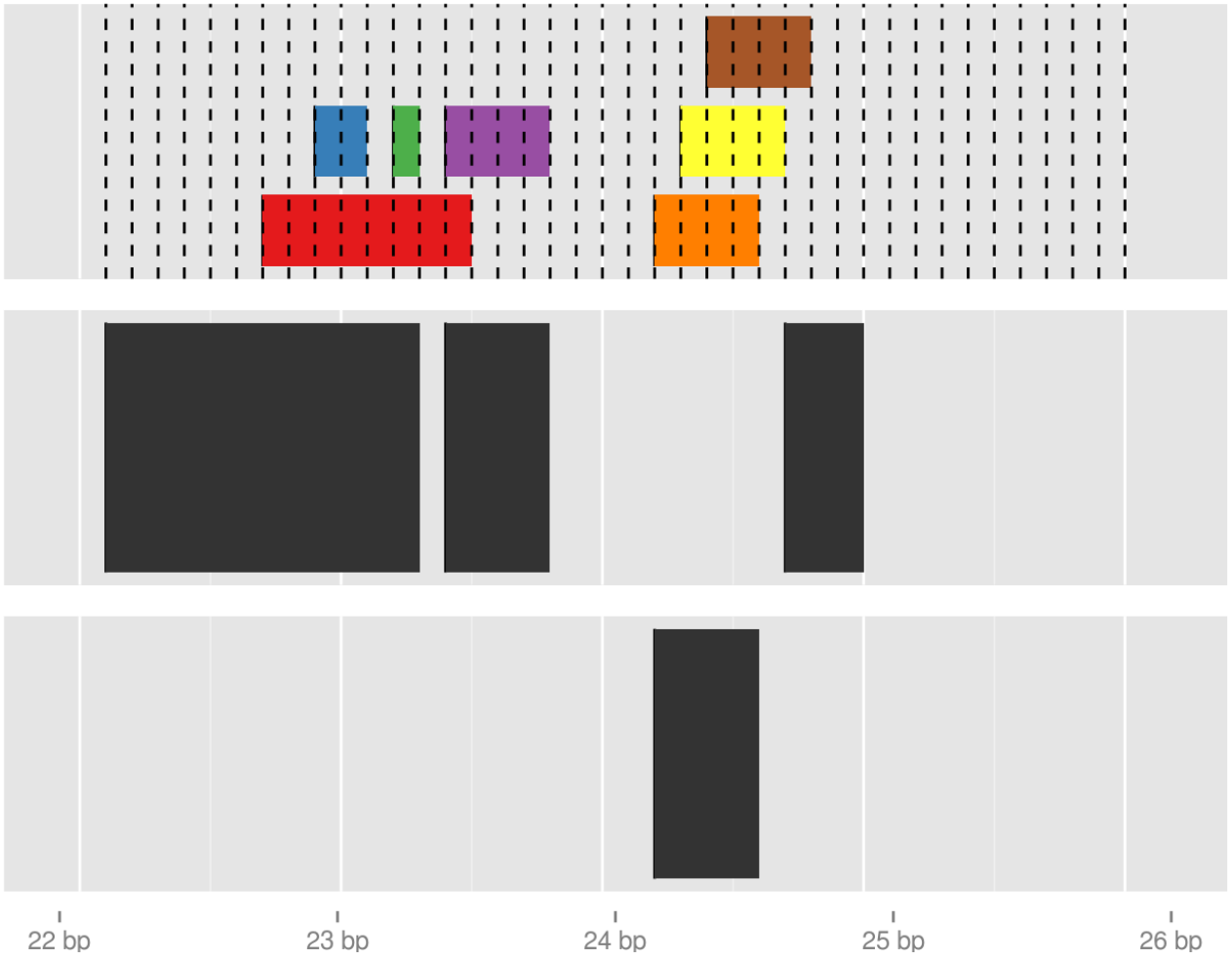


Subtraction

- Or which positions are missing

```
setdiff(ir,ir3)
```

```
## IRanges of length 1
##     start end width
## [1]    22  26     5
```



Core data-type 1: DNA sequences

Biostrings

The Biostrings package is specifically-designed for biological sequences

- It introduces a new object type, the `DNAStringSet` for storing sequences
- We can create an object of this type by using the `DNAStringSet` function
- Typing the name of your new object prints a summary to the screen

```
library(Biostrings)
myseq <- DNAStringSet(randomStrings)
myseq
```

```
##   A DNAStringSet instance of length 100
##       width seq
##   [1]    15 GTGGCTGTTCTTACA
##   [2]    12 ATAGTCATGTCA
##   [3]    11 GCAACAGTAAA
##   [4]    17 CCTCCGGTCTTCTTGCG
##   [5]    11 TGTCAATGAAC
##   ...   ... ...
##  [96]    20 AGGGCTCTGCAATCAAATTT
##  [97]    10 GTCACTTGAG
##  [98]    11 CCGCAGGGACA
##  [99]    18 GACGTGGGGAGCATCCTT
## [100]    18 CGCTGCAACCGCGCCTAC
```

Object structure

- The definition of the object is not for the faint-hearted

```
str(myseq)
```

```
## Formal class 'DNAStringSet' [package "Biostrings"] with 5 slots
##   ..@ pool           :Formal class 'SharedRaw_Pool' [package "XVector"] with
 2 slots
##   .. .. ..@ xp_list                   :List of 1
##   .. .. .. ..$ :<externalptr>
##   .. .. ..@ .link_to_cached_object_list:List of 1
##   .. .. .. ..$ :<environment: 0x6cbdbf0>
##   ..@ ranges         :Formal class 'GroupedIRanges' [package "XVector"] with
 7 slots
##   .. .. ..@ group          : int [1:100] 1 1 1 1 1 1 1 1 1 1 ...
##   .. .. ..@ start          : int [1:100] 1 16 28 39 56 67 86 99 112 124 ...
##   .. .. ..@ width          : int [1:100] 15 12 11 17 11 19 13 13 12 11 ...
##   .. .. ..@ NAMES          : NULL
##   .. .. ..@ elementType    : chr "integer"
##   .. .. ..@ elementMetadata: NULL
##   .. .. ..@ metadata       : list()
##   ..@ elementType    : chr "DNAString"
##   ..@ elementMetadata: NULL
##   ..@ metadata       : list()
```

Biostrings operations

- However, we can treat a `Biostrings` object like a standard vector

```
myseq[1:5]
```

```
##   A DNAStringSet instance of length 5
##     width seq
## [1]    15 GTGGCTGTTCTTACA
## [2]    12 ATAGTCATGTCA
## [3]    11 GCAACAGTAAA
## [4]    17 CCTCCGGTCTTCTTGCG
## [5]    11 TGTCAATGAAC
```

Accessor functions

- If we want to do a calculation on the width and sequences themselves, we can extract them with `width` and `as.character`
  - the result is a vector

```
width(myseq)
```

```
##   [1] 15 12 11 17 11 19 13 13 12 11 13 14 20 17 16 17 16 18 13 11 13 17 11
##  [24] 18 17 14 18 14 11 13 13 17 20 18 10 20 20 19 20 13 14 18 11 10 18 14
##  [47] 19 15 14 19 15 10 20 16 17 18 19 17 11 18 17 15 12 16 12 11 11 18 10
##  [70] 10 17 13 15 11 11 10 20 18 19 18 14 18 13 15 15 13 17 19 15 15 12 20
##  [93] 13 18 16 20 10 11 18 18
```

```
head(as.character(myseq))
```

```
## [1] "GTGGCTGTTCTTACA"     "ATAGTCATGTCA"        "GCAACAGTAAA"
## [4] "CCTCCGGTCTTCTTGCG"   "TGTCAATGAAC"         "TACTACAGTCCAAGGGCTT"
```

Accessor functions

What does this do?

```
myseq[width(myseq)>19]
```

```
##   A DNAStringSet instance of length 9
##     width seq
## [1]    20 TTTTGGTAATATTCTCCACA
## [2]    20 GTCTCCAGCCCTCCGGCTCT
## [3]    20 TCTTTGGCAACTAGGGCATT
## [4]    20 AAGATGGATCTTATCAACTA
## [5]    20 AGCGGGGACGTCGAGCCTAA
## [6]    20 AGCGTTGGAGAACTTGCAGG
## [7]    20 TTTCAAAAAGGGATTCAGTG
## [8]    20 CTCGTTGAACAACTGCAGTA
## [9]    20 AGGGCTCTGCAATCAAATTT
```

More advanced subsetting

```
myseq[subseq(myseq,1,3) == "TTC"]
```

```
##   A DNAStringSet instance of length 2
##     width seq
## [1]    17 TTCATCAATTCGAGGAC
## [2]    18 TTCAGATATCCGAGGTTG
```

We can also use the `matchPattern` function + see practical for details

Other useful operations

Some useful string operation functions are provided

```
af <- alphabetFrequency(myseq, baseOnly=TRUE)
head(af)
```

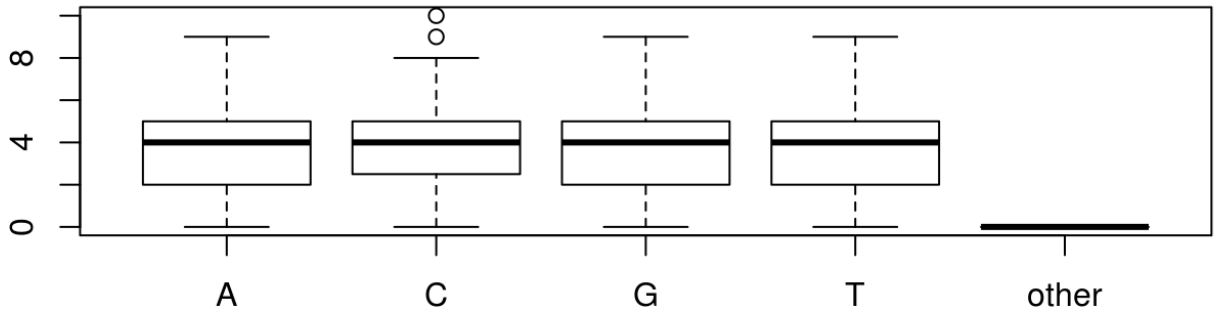
```
##      A C G T other
## [1,] 2 3 4 6     0
## [2,] 4 2 2 4     0
## [3,] 6 2 2 1     0
## [4,] 0 7 4 6     0
## [5,] 4 2 2 3     0
## [6,] 5 5 4 5     0
```

Letter frequencies

```
myseq[af[,1] ==0,]
```

```
##   A DNAStringSet instance of length 5
##     width seq
## [1]    17 CCTCCGGTCTTCTTGCG
## [2]    11 CGTCTTTGCTT
## [3]    12 TCTCTGGTCTGG
## [4]    10 TGTTTTGCCT
## [5]    10 TCTGTTGCTC
```

```
boxplot(af)
```



More-specialised features

```
reverse(myseq)
```

```
##   A DNAStringSet instance of length 100
##       width seq
##   [1]    15 ACATTCTTGTCGGTG
##   [2]    12 ACTGTACTGATA
##   [3]    11 AAATGACAACG
##   [4]    17 GCGTTCTTCTGGCCTCC
##   [5]    11 CAAGTAACTGT
##   ...   ... ...
##  [96]    20 TTTAAACTAACGTCTCGGGA
##  [97]    10 GAGTTCACTG
##  [98]    11 ACAGGGACGCC
##  [99]    18 TTCCTACGAGGGGTGCAG
## [100]    18 CATCCGCGCCAACGTCGC
```

```
reverseComplement(myseq)
```

```
##   A DNAStringSet instance of length 100
##       width seq
##   [1]    15 TGTAAGAACAGCCAC
##   [2]    12 TGACATGACTAT
##   [3]    11 TTTACTGTTGC
##   [4]    17 CGCAAGAAGACCGGAGG
##   [5]    11 GTTCATTGACA
##   ...   ... ...
##  [96]    20 AAATTTGATTGCAGAGCCCT
##  [97]    10 CTCAAGTGAC
##  [98]    11 TGTCCCTGCGG
##  [99]    18 AAGGATGCTCCCCACGTC
## [100]    18 GTAGGCGCGGTTGCAGCG
```

```
translate(myseq)
```

```
##   A AAStringSet instance of length 100
##       width seq
##   [1]     5 VAVLT
##   [2]     4 IVMS
##   [3]     3 ATV
##   [4]     5 PPVFL
##   [5]     3 CQ*
##   ...   ... ...
##  [96]     6 RALQSN
##  [97]     3 VT*
##  [98]     3 PQG
##  [99]     6 DVGSIL
## [100]     6 RCNRAY
```

Fastq recap

Recall that sequence reads are represented in text format

```
readLines(path.to.my.fastq ,n=10)
```

It should be possible to represent these as `Biostrings` objects

The `ShortRead` package

One of the first NGS packages in Bioconductor

- Has convenient functions for reading fastq files and performing quality assessment
  - In practice, we would use other tools for processing fastq files
  - e.g. fastqc for quality assessment

```
library(ShortRead)
fq <- readFastq(path.to.my.fastq)
fq
```

Practical application - Representing the genome

The genome as a string - `BSgenome`

```
library(BSgenome)
head(available.genomes())
```

```
## [1] "BSgenome.Alyrata.JGI.v1"
## [2] "BSgenome.Amellifera.BeeBase.assembly4"
## [3] "BSgenome.Amellifera.UCSC.apiMel2"
## [4] "BSgenome.Amellifera.UCSC.apiMel2.masked"
## [5] "BSgenome.Athaliana.TAIR.04232008"
## [6] "BSgenome.Athaliana.TAIR.TAIR9"
```

Various versions of the human genome

```
ag <- available.genomes()
ag[grep("Hsapiens",ag)]
```

```
## [1] "BSgenome.Hsapiens.NCBI.GRCh38"
## [2] "BSgenome.Hsapiens.UCSC.hg17"
## [3] "BSgenome.Hsapiens.UCSC.hg17.masked"
## [4] "BSgenome.Hsapiens.UCSC.hg18"
## [5] "BSgenome.Hsapiens.UCSC.hg18.masked"
## [6] "BSgenome.Hsapiens.UCSC.hg19"
## [7] "BSgenome.Hsapiens.UCSC.hg19.masked"
## [8] "BSgenome.Hsapiens.UCSC.hg38"
## [9] "BSgenome.Hsapiens.UCSC.hg38.masked"
```

The latest human genome

```
library(BSgenome.Hsapiens.UCSC.hg19)
hg19 <- BSgenome.Hsapiens.UCSC.hg19::Hsapiens
hg19
```

```
## Human genome:
## # organism: Homo sapiens (Human)
## # provider: UCSC
## # provider version: hg19
## # release date: Feb. 2009
## # release name: Genome Reference Consortium GRCh37
## # 93 sequences:
## #   chr1                chr2                chr3
## #   chr4                chr5                chr6
## #   chr7                chr8                chr9
## #   chr10               chr11               chr12
## #   chr13               chr14               chr15
## #   ...                 ...                 ...
## #   chrUn_gl000235      chrUn_gl000236      chrUn_gl000237
## #   chrUn_gl000238      chrUn_gl000239      chrUn_gl000240
## #   chrUn_gl000241      chrUn_gl000242      chrUn_gl000243
## #   chrUn_gl000244      chrUn_gl000245      chrUn_gl000246
## #   chrUn_gl000247      chrUn_gl000248      chrUn_gl000249
## # (use 'seqnames()' to see all the sequence names, use the '$' or '[['
## # operator to access a given sequence)
```

Chromosome-level sequence

- The genome package can be accessed at a chromosome level
- The `names` of the object are chromosome names
  - can use list accessing method `[[]]` to get chromsome sequence
  - result is a `DNAString`
    - which we have various tools for dealing with

```
head(names(hg19))
```

```
## [1] "chr1" "chr2" "chr3" "chr4" "chr5" "chr6"
```

```
chrX <- hg19[["chrX"]]
chrX
```

```
##   155270560-letter "DNAString" instance
## seq: NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN...NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
```

```
alphabetFrequency(chrX,baseOnly=TRUE)
```

```
##        A        C        G        T    other
## 45648952 29813353 29865831 45772424  4170000
```

Retrieving sequences

- Of course, we might just want the sequence of a particular region (e.g. gene)
- we can use `getSeq` to do this

```
tp53 <- getSeq(hg19, "chr17", 7577851, 7590863)
tp53
```

```
##   13013-letter "DNAString" instance
## seq: TTGTATTTTTCAGTAGAGACGGGGTTTCACCGTT...GTCTTGAGCACATGGGAGGGGAAAACCCCAATC
```

```
as.character(tp53[1:10])
```

```
## [1] "TTGTATTTTT"
```

```
alphabetFrequency(tp53,baseOnly=TRUE)
```

```
##     A     C     G     T other
##  3102  3375  3025  3511     0
```

```
subseq(tp53, 1000,1010)
```

```
##   11-letter "DNAString" instance
## seq: TATAGGTGTGC
```

Timings

Don't need to load the whole genome into memory, so reading a particular sequence is ***fast***

```
system.time(tp53 <- getSeq(hg19, "chr17", 7577851, 7598063))
```

```
##    user  system elapsed
##   0.115   0.000   0.115
```

Manipulating sequences

We can now use `Biostrings` operations to manipulate the sequence

```
translate(subseq(tp53, 1000,1010))
```

```
## Warning in .Call2("DNAStringSet_translate", x, skip_code,
## dna_codes[codon_alphabet], : last 2 bases were ignored
```

```
##   3-letter "AAString" instance
## seq: YRC
```

```
reverseComplement(subseq(tp53, 1000,2000))
```

```
##   1001-letter "DNAString" instance
## seq: CCTATGGAAACTGTGAGTGGATCCATTGGAAGGG...AAAATTAGCCAGGCATGGTGGTGCACACCTATA
```

Introducing GRanges

- `GRanges` are a special kind of `IRanges` object used to manipulate genomic intervals in an efficient manner
- We can define a 'chromosome' for each range
  - referred to as seqnames
- we have the option to define a strand
- need to supply a ranges object, as we saw before

```
library(GenomicRanges)
gr <- GRanges(c("A","A","A","B","B","B","B"), ranges=ir)
gr
```

```
## GRanges object with 7 ranges and 0 metadata columns:
##       seqnames    ranges strand
##          <Rle> <IRanges>  <Rle>
##   [1]        A  [ 7, 15]      *
##   [2]        A  [ 9, 11]      *
##   [3]        A  [12, 13]      *
##   [4]        B  [14, 18]      *
##   [5]        B  [22, 26]      *
##   [6]        B  [23, 27]      *
##   [7]        B  [24, 28]      *
##   -------
##   seqinfo: 2 sequences from an unspecified genome; no seqlengths
```

Representing a gene

- Creating an object to represent a particular gene is easy if we know its coordinates
  - we will look at represening the full gene structure tomorrow
    - e.g. exons, introns etc

```
mygene <- GRanges("chr17", ranges=IRanges(7577851, 7598063))
myseq <- getSeq(hg19, mygene)
myseq
```

```
##   A DNAStringSet instance of length 1
##     width seq
## [1] 20213 TTGTATTTTTCAGTAGAGACGGGGTTTCACC...CTACTTGGGAGGCTGAGGTGGGAGGATCGCT
```

```
tp53
```

```
##   20213-letter "DNAString" instance
## seq: TTGTATTTTTCAGTAGAGACGGGGTTTCACCGTT...AGCTACTTGGGAGGCTGAGGTGGGAGGATCGCT
```

Intermission

Work through section 1 of the practical

- Examples of creating `IRanges` and `GRanges` objects
- Accessing genome packages
- Manipulating genome sequences

Practical application - Manipulating Aligned Reads

Dealing with aligned reads

We will assume that the sequencing reads have been aligned and that we are interested in processing

the alignments.

- `Rsamtools` provides an interface for doing this.
- However, we will use the `readGAlignments` tool in `GenomicAlignments` which extracts the essential information from the bam file.
    - don't even attempt to try to understand the data structure!

```
library(GenomicAlignments)
bam <- readGAlignments(mybam,use.names = TRUE)
str(bam)
```

```
## Formal class 'GAlignments' [package "GenomicAlignments"] with 8 slots
##   ..@ NAMES          : chr [1:175346] "SRR031715.1138209" "SRR031714.776678"
 "SRR031715.3258011" "SRR031715.4791418" ...
##   ..@ seqnames       :Formal class 'Rle' [package "S4Vectors"] with 4 slots
##   .. .. ..@ values         : Factor w/ 8 levels "chr2L","chr2R",..: 5
##   .. .. ..@ lengths        : int 175346
##   .. .. ..@ elementMetadata: NULL
##   .. .. ..@ metadata       : list()
##   ..@ start          : int [1:175346] 169 184 187 193 326 943 944 946 946 95
7 ...
##   ..@ cigar          : chr [1:175346] "37M" "37M" "37M" "37M" ...
##   ..@ strand         :Formal class 'Rle' [package "S4Vectors"] with 4 slots
##   .. .. ..@ values         : Factor w/ 3 levels "+","-","*": 1 2 1 2 1 2 1 2
 1 2 ...
##   .. .. ..@ lengths        : int [1:37319] 1 2 1 1 3 2 3 10 3 1 ...
##   .. .. ..@ elementMetadata: NULL
##   .. .. ..@ metadata       : list()
##   ..@ elementMetadata:Formal class 'DataFrame' [package "S4Vectors"] with 6
slots
##   .. .. ..@ rownames       : NULL
##   .. .. ..@ nrows          : int 175346
##   .. .. ..@ listData       : Named list()
##   .. .. ..@ elementType    : chr "ANY"
##   .. .. ..@ elementMetadata: NULL
##   .. .. ..@ metadata       : list()
##   ..@ seqinfo        :Formal class 'Seqinfo' [package "GenomeInfoDb"] with 4
 slots
##   .. .. ..@ seqnames   : chr [1:8] "chr2L" "chr2R" "chr3L" "chr3R" ...
##   .. .. ..@ seqlengths : int [1:8] 23011544 21146708 24543557 27905053 13518
57 19517 22422827 347038
##   .. .. ..@ is_circular: logi [1:8] NA NA NA NA NA NA ...
##   .. .. ..@ genome     : chr [1:8] NA NA NA NA ...
##   ..@ metadata       : list()
```

Representation of aligned reads

The result looks a lot like a GRanges object. In fact, a lot of the same operations can be used

```
bam
```

```
## GAlignments object with 175346 alignments and 0 metadata columns:
##                     seqnames strand       cigar    qwidth
##                        <Rle>  <Rle> <character> <integer>
##   SRR031715.1138209     chr4      +         37M        37
##    SRR031714.776678     chr4      -         37M        37
##   SRR031715.3258011     chr4      -         37M        37
##   SRR031715.4791418     chr4      +         37M        37
##   SRR031715.1138209     chr4      -         37M        37
##                 ...      ...    ...         ...       ...
##   SRR031714.1650928     chr4      +         37M        37
##   SRR031714.1650928     chr4      -         37M        37
##   SRR031714.5192891     chr4      +         37M        37
##   SRR031715.2351056     chr4      +         37M        37
##    SRR031714.864195     chr4      +         37M        37
##                       start       end     width     njunc
##                   <integer> <integer> <integer> <integer>
##   SRR031715.1138209       169       205        37         0
##    SRR031714.776678       184       220        37         0
##   SRR031715.3258011       187       223        37         0
##   SRR031715.4791418       193       229        37         0
##   SRR031715.1138209       326       362        37         0
##                 ...       ...       ...       ...       ...
##   SRR031714.1650928   1349708   1349744        37         0
##   SRR031714.1650928   1349838   1349874        37         0
##   SRR031714.5192891   1351640   1351676        37         0
##   SRR031715.2351056   1351640   1351676        37         0
##    SRR031714.864195   1351760   1351796        37         0
##   -------
##   seqinfo: 8 sequences from an unspecified genome
```

Accessing particular reads

- Yet again, we can treat the object as a vector

```
length(bam)
```

```
## [1] 175346
```

```
bam[1:5]
```

```
## GAlignments object with 5 alignments and 0 metadata columns:
##                      seqnames strand       cigar    qwidth
##                         <Rle>  <Rle> <character> <integer>
##   SRR031715.1138209     chr4      +         37M        37
##    SRR031714.776678     chr4      -         37M        37
##   SRR031715.3258011     chr4      -         37M        37
##   SRR031715.4791418     chr4      +         37M        37
##   SRR031715.1138209     chr4      -         37M        37
##                         start       end     width     njunc
##                      <integer> <integer> <integer> <integer>
##   SRR031715.1138209       169       205        37         0
##    SRR031714.776678       184       220        37         0
##   SRR031715.3258011       187       223        37         0
##   SRR031715.4791418       193       229        37         0
##   SRR031715.1138209       326       362        37         0
##   -------
##   seqinfo: 8 sequences from an unspecified genome
```

```
bam[sample(1:length(bam),5)]
```

```
## GAlignments object with 5 alignments and 0 metadata columns:
##                      seqnames strand       cigar    qwidth
##                         <Rle>  <Rle> <character> <integer>
##   SRR031715.1975286     chr4      -         37M        37
##    SRR031714.468992     chr4      +         37M        37
##    SRR031714.900356     chr4      -         37M        37
##   SRR031714.4661995     chr4      +         37M        37
##   SRR031715.1931250     chr4      +         37M        37
##                         start       end     width     njunc
##                      <integer> <integer> <integer> <integer>
##   SRR031715.1975286    927885    927921        37         0
##    SRR031714.468992    570072    570108        37         0
##    SRR031714.900356     87328     87364        37         0
##   SRR031714.4661995    692546    692582        37         0
##   SRR031715.1931250   1213334   1213370        37         0
##   -------
##   seqinfo: 8 sequences from an unspecified genome
```

Querying alignments

- As usual, there are a variety of accessor functions to get data from the object

```
table(strand(bam))
```

```
##
##     +     -     *
## 84871 90475     0
```

```
summary(width(bam))
```

```
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
##    37.00    37.00    37.00    58.72    37.00 19350.00
```

```
range(start(bam))
```

```
## [1]     169 1351760
```

```
head(cigar(bam))
```

```
## [1] "37M" "37M" "37M" "37M" "37M" "37M"
```

Overlap aligned reads with GRanges

- A `GAlignments` object can be used in `findOverlaps`

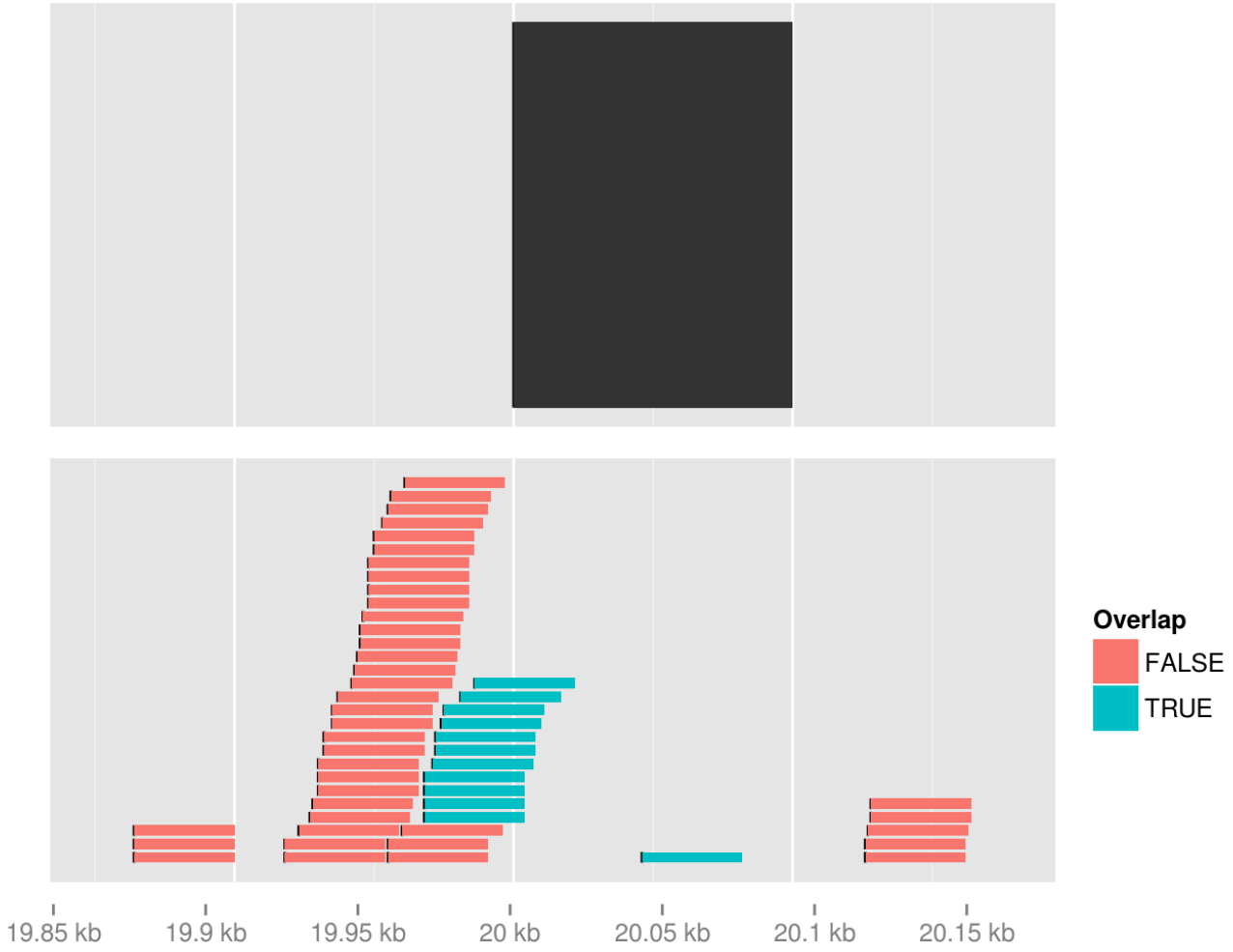
```
gr <- GRanges("chr4", IRanges(start = 20000, end = 20100))
gr
```

```
## GRanges object with 1 range and 0 metadata columns:
##       seqnames         ranges strand
##          <Rle>      <IRanges>  <Rle>
##   [1]     chr4 [20000, 20100]      *
##   -------
##   seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

```
findOverlaps(gr,bam)
```

```
## Hits object with 12 hits and 0 metadata columns:
##        queryHits subjectHits
##        <integer>   <integer>
##    [1]         1        6699
##    [2]         1        6700
##    [3]         1        6701
##    [4]         1        6702
##    [5]         1        6703
##    ...       ...         ...
##    [8]         1        6706
##    [9]         1        6707
##   [10]         1        6708
##   [11]         1        6709
##   [12]         1        6710
##   -------
##   queryLength: 1
##   subjectLength: 175346
```

Identifying the reads



A shortcut

```
bam.sub <- bam[bam %over% gr]
bam.sub
```

```
## GAlignments object with 12 alignments and 0 metadata columns:
##                     seqnames strand       cigar    qwidth
##                        <Rle>  <Rle> <character> <integer>
##   SRR031714.4092638     chr4      -         37M        37
##   SRR031714.4275537     chr4      -         37M        37
##   SRR031715.1315719     chr4      -         37M        37
##   SRR031715.1502533     chr4      -         37M        37
##    SRR031714.336402     chr4      -         37M        37
##                 ...      ...    ...         ...       ...
##   SRR031715.3358559     chr4      +         37M        37
##   SRR031715.4831822     chr4      +         37M        37
##   SRR031715.4459351     chr4      +         37M        37
##   SRR031715.2716654     chr4      -         37M        37
##   SRR031715.1552693     chr4      +         37M        37
##                         start       end     width     njunc
##                     <integer> <integer> <integer> <integer>
##   SRR031714.4092638     19968     20004        37         0
##   SRR031714.4275537     19968     20004        37         0
##   SRR031715.1315719     19968     20004        37         0
##   SRR031715.1502533     19968     20004        37         0
##    SRR031714.336402     19971     20007        37         0
##                 ...       ...       ...       ...       ...
##   SRR031715.3358559     19974     20010        37         0
##   SRR031715.4831822     19975     20011        37         0
##   SRR031715.4459351     19981     20017        37         0
##   SRR031715.2716654     19986     20022        37         0
##   SRR031715.1552693     20046     20082        37         0
##   -------
##   seqinfo: 8 sequences from an unspecified genome
```

Chromosome naming conventions

- Regrettably, people can't seem to agree on how to name chromosomes
  - e.g. chr1 vs 1 etc
- We have to make sure to use the same convention if attempted to overlap

```
gr <- GRanges("4", IRanges(start = 20000, end = 20100))
gr
```

```
## GRanges object with 1 range and 0 metadata columns:
##       seqnames         ranges strand
##          <Rle>      <IRanges>  <Rle>
##   [1]        4 [20000, 20100]      *
##   -------
##   seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

```
findOverlaps(gr,bam)
```

```
## Warning in .Seqinfo.mergexy(x, y): The 2 combined objects have no sequence l
evels in common. (Use
##   suppressWarnings() to suppress this warning.)
```

```
## Hits object with 0 hits and 0 metadata columns:
##    queryHits subjectHits
##    <integer>   <integer>
##   -------
##   queryLength: 1
##   subjectLength: 175346
```

Solution

```
gr
```

```
## GRanges object with 1 range and 0 metadata columns:
##       seqnames         ranges strand
##          <Rle>      <IRanges>  <Rle>
##   [1]        4 [20000, 20100]      *
##   -------
##   seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

```
gr <- renameSeqlevels(gr, c("4"="chr4"))
gr
```

```
## GRanges object with 1 range and 0 metadata columns:
##       seqnames         ranges strand
##          <Rle>      <IRanges>  <Rle>
##   [1]     chr4 [20000, 20100]      *
##   -------
##   seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

Finer-control over reading

- `readGAlignments` uses the `Rsamtools` interface, which allows more control over how we import data
  - the ' `ScanBamParam` (!) function allows the user to customise what fields from the bam file are imported
    - recall yesterday's discussion about bam file contents

```
?ScanBamParam
```

Example: adding mapping quality, base quality and flag

```
bam.extra <- readGAlignments(file=mybam,param=ScanBamParam(what=c("mapq","qual"
,"flag")))
bam.extra[1:5]
```

```
## GAlignments object with 5 alignments and 3 metadata columns:
##       seqnames strand       cigar    qwidth     start
##          <Rle>  <Rle> <character> <integer> <integer>
##   [1]     chr4      +         37M        37       169
##   [2]     chr4      -         37M        37       184
##   [3]     chr4      -         37M        37       187
##   [4]     chr4      +         37M        37       193
##   [5]     chr4      -         37M        37       326
##             end     width     njunc |      mapq
##       <integer> <integer> <integer> | <integer>
##   [1]       205        37         0 |       255
##   [2]       220        37         0 |       255
##   [3]       223        37         0 |       255
##   [4]       229        37         0 |       255
##   [5]       362        37         0 |       255
##                                     qual      flag
##                              <PhredQuality> <integer>
##   [1] IIIIIIIIIIIIIIIIIIIIIIIII8IIIIIIIGII        99
##   [2] IIIIIIIIEIIIIIIIIIIIIIIIIIIIIIIIIII       153
##   [3] II6II7IIIIIIIIIIIIIIIIIIIIIIIIIIII        89
##   [4] IIIIIIIIIIIIIIIIIIIIIIIIIIIIIFII3I       137
##   [5] ++I+4-05>*I2GF/II6IIIIIIIIIIIIIII<I       147
##   -------
##   seqinfo: 8 sequences from an unspecified genome
```

```
table(mcols(bam.extra)$flag)
```

```
##
##    65    73    81    83    89    97    99   113   129   137
##    29  4891 14737 23037  7769 14781 22791    34    29  4576
##   145   147   153   161   163   177
## 14781 22791  7292 14737 23037    34
```

Example: Dealing with PCR duplicates

```
dupReads <- readGAlignments(file=mybam,param=ScanBamParam(scanBamFlag(isDuplica
te = TRUE)))
nodupReads <- readGAlignments(file=mybam,param=ScanBamParam(scanBamFlag(isDupli
cate = FALSE)))
allreads <- readGAlignments(file=mybam,param=ScanBamParam(scanBamFlag(isDuplica
te = NA)))
length(dupReads)
```

```
## [1] 0
```

```
length(nodupReads)
```

```
## [1] 175346
```

```
length(allreads)
```

```
## [1] 175346
```

```
length(allreads) - length(dupReads)
```

```
## [1] 175346
```

Reading a particular region

- Only possible if the bam file has an accompanying *bai* index file

```
bam.sub2 <-
  readGAlignments(file=mybam,param=ScanBamParam(which=gr),use.names = TRUE)
length(bam.sub2)
```

```
## [1] 14
```

```
bam.sub2
```

```
## GAlignments object with 14 alignments and 0 metadata columns:
##                     seqnames strand       cigar    qwidth
##                        <Rle>  <Rle> <character> <integer>
##   SRR031714.4100693     chr4      +  31M7704N6M        37
##   SRR031715.5248298     chr4      +  29M7704N8M        37
##   SRR031714.4092638     chr4      -         37M        37
##   SRR031714.4275537     chr4      -         37M        37
##   SRR031715.1315719     chr4      -         37M        37
##                 ...      ...    ...         ...       ...
##   SRR031715.3358559     chr4      +         37M        37
##   SRR031715.4831822     chr4      +         37M        37
##   SRR031715.4459351     chr4      +         37M        37
##   SRR031715.2716654     chr4      -         37M        37
##   SRR031715.1552693     chr4      +         37M        37
##                       start       end     width     njunc
##                   <integer> <integer> <integer> <integer>
##   SRR031714.4100693     13660     21400      7741         1
##   SRR031715.5248298     13662     21402      7741         1
##   SRR031714.4092638     19968     20004        37         0
##   SRR031714.4275537     19968     20004        37         0
##   SRR031715.1315719     19968     20004        37         0
##                 ...       ...       ...       ...       ...
##   SRR031715.3358559     19974     20010        37         0
##   SRR031715.4831822     19975     20011        37         0
##   SRR031715.4459351     19981     20017        37         0
##   SRR031715.2716654     19986     20022        37         0
##   SRR031715.1552693     20046     20082        37         0
##   -------
##   seqinfo: 8 sequences from an unspecified genome
```

Recap

- Ranges can be used to represent continuous regions
- GRanges are special ranges with extra biological context
- GRanges can be manipulated, compared, overlapped with each other
- Aligned reads can be represented by Ranges
- Genome and sequencing reads can be represented efficiently by Biostrings

Now, work through Section 2 of the practical