Introduction to Strings and Ranges in Bioconductor

Mark Dunning mark.dunning@cruk.cam.ac.uk

Last modified May 20, 2015

Contents

1 Introduction 1

2 Strings 1

2.1 Strings in R 1

2.2 Strings in Bioconductor; Biostrings 2

3 Fastq analysis with ShortRead 4

4 Ranges and GRanges 6

4.1 Ranges 6

4.2 GRanges 8

5 Reading bam alignments 9

5.1 Reading a particular genomic region 10

1 Introduction

The purpose of this practical is to familiarise you with the concept of strings and ranges in Bioconductor. These are fundamental objects upon which many analysis tools for next-generation sequencing data are built-upon.

In order for the example code to run, you must be in the `Practicals` directory.

2 Strings

2.1 Strings in R

Before explaining the specialised string manipulation features available in Bioconductor, we will use some featuers from the R base code. Written long before the age of next-generation sequencing, these are not designed to deal with large volumes of data. Nevertheless, some useful operations are possible.

Here we will define a function to create a random set of DNA sequences with configurable lengths that we will work with in the following exercises. Each string is a separate item in a vector, so the usual subset conventions apply.

Use Case: Use the `randomDNAString` function, defined in the scripts folder, to construct 500 random sequence of A, T, G,, C. By default each sequence will be of length between 10 and 20 [How do you know this?] . Print out the first five strings.

```
source("scripts/randomDNAString.R")
randomDNAString
rand <- randomDNAString()
rand
rand[1:5]
```

Use Case: What is the distribution of the number of characters in each string? Select the strings with exactly 10 characters.

```
nchar(rand)
summary(nchar(rand))
hist(nchar(rand))
rand[nchar(rand)== 10]
```

Use Case: How many occurences of the the string AAA are there? Replace AAA with the string NNN

```
rand[grep("AAAA", rand)]

gsub("AAA", "NNN", rand)
```

Use Case: Subset each string to the first 3 characters. What is the most common 3-letter string?

```
rand.sub <- substr(rand, 1,3)
rand[1:10]
rand.sub[1:10]
sort(table(substr(rand, 1,3)))
```

2.2 Strings in Bioconductor; Biostrings

The *Biostrings* package in Bioconductor provides tools for working with sequences. The packages implements the base string manipulation functions in R, plus much more besides. The functions are written in a memory-efficient manner and the implemented object-types are the foundation upon which more-complex operations are built-upon. The basic object type that we will use is the *DNAStringSet*, which is a collection of the more basic *DNAString*. The subsetting of *DNAStringSet* object operates as you would expect.

Use Case: Create a set of 1000 DNA sequences that vary between length 5 and 100 and use this to make a DNAStringSet instance

```r
library(Biostrings)
rand2 <- randomDNAString(N=1000,minL=5,maxL=100)
myseq <- DNAStringSet(rand2)
myseq
```

You will notice that the object is displayed in a more convenient way to the random string, and the length (width) of each string is displayed alongside it. The width of each string can be accessed by the `width` function. The *DNAStringSet* objects can be subset in the same ways as standard R vectors.

Use Case: Create a summary of the string length distributions. Select all strings with less than 10 characters.

```r
summary(width(myseq))
hist(width(myseq))
myseq[width(myseq)<10]
as.character(myseq[width(myseq)<10])
```

Many standard R functions have also been adapted to work for the *DNAStringSet*. These include `sort`, `head`, `rev` and `tail`.

Use Case: Explore the behaviour of the `subseq`, `sort`, `head`, `tail` and `rev`

```r
subseq(myseq, 1, 3)
sort(myseq)
head(myseq)
tail(myseq)
head(rev(myseq))
```

Another common operation is to be able to count the ocurrences of letters in each string. This could be achieved by use of the `table` function, but a more efficient implementation is provided in `alphabetFrequency`. Calculating the frequency of 2-mer and 3-mers is also possible.

Use Case: Calculate the frequencing of letters in each string and make sure that you understand the dimensions of the `alphabetFrequency` output. Check for any biases in base distribution. Do any sequences not contain any As?

```r
af <- alphabetFrequency(myseq, baseOnly=TRUE)
dim(af)
head(af)
myseq[af[,1] ==0,]
boxplot(af)
boxplot(dinucleotideFrequency(myseq),las=2)
boxplot(trinucleotideFrequency(myseq),horizontal=T,las=2)
```

Use Case: Modify the alphabet frequency table so that it accounts the different string lengths

```
af2 <- alphabetFrequency(myseq, baseOnly=TRUE)/width(myseq)
boxplot(af2)
```

Use Case: Calculate the percentage of Gs and Cs for each string. Find any strings with greater than 75% GC bases.

```
gcCounts <- af2[,"C"] + af2[,"G"]
summary(gcCounts)
myseq[gcCounts > 0.75]
```

Use Case: Reverse, complement, and translate the sequences

```
reverse(myseq)
reverseComplement(myseq)
translate(myseq)
```

After manipulating our set of sequences, we may wish to export them for analysis in some external tool. A common format for sequence representation is *fasta* which usually has the file extension `.fa`.

Use Case: Write your sequences out to a file

```
writeXStringSet(myseq, file="myseq.fa")
```

3 Fastq analysis with ShortRead

We are now going to deal with real, rather than simulated, data. The *ShortRead* packages is one of the many Bioconductor packages that relies on *Biostrings* for its core functionality.

The FASTQ format is now the industry-standard way that sequencers produce output.

```
@SRR031724.1 HWI-EAS299_4_30M2BAAXX:5:1:1513:1024 length=37
GTTTTGTCCAAGTTCTGGTAGCTGAATCCTGGGGCGC
+SRR031724.1 HWI-EAS299_4_30M2BAAXX:5:1:1513:1024 length=37
IIIIIIIIIIIIIIIIIIIIIIIIIIII+HIIII<IE
```

Each read is given a unique identifier that is repeated on the first and third lines, although it can be omitted from the third line. Typically this identifier includes the name of the sequencing machine, lane, tile and coordinates of the read. The second and fourth lines are the nucleotides and quality scores respectively. The quality score is represented as a letter from the ASCII alphabet, with letters higher in the alphabet representing a higher quality and more-confident base-call.

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
```

Reading the fastq file into R allows quality control steps to be performed and the generation of the type of plots that are produced by the fastqc tool [1].

Use Case:

Remind yourself of the format of a fastq file by reading the first few lines of example file that we have provided in the `exampleData` folder. Then, read the file using the *ShortRead* package. How many sequencing reads does the file contain?

```
myfile <- "exampleData/sample.fq"

readLines(myfile,n=10)
library(ShortRead)
fq <- readFastq(myfile)
fq
```

ℹTo read your own fastq file into ShortRead, simply change the value of the `myfile` variable to point to the fastq file that you want to read. If you have trouble typing the path to the file, try the `file.choose` function.

The `fq` object is of type *ShortReadQ* and particular aspects of the object can be accessed by the `sread`, `quality` and `width` functions.

Use Case: Explore the ShortRead object by extracing the names, qualities and sequences of the reads themselves. What length are the reads?

```
sread(fq)
quality(fq)
id(fq)
table(width(fq))
```

Hopefully you will recognise the format that the reads are stored in. By using the Biostrings common infrastrucure, the operations we introduced earlier can be used on sequencing reads. Also, some commonly-used operations have been provided. A typical usage for reading fastq files is for quality assessment using the imported quality scores and seqeunces themselves. The `alphabetByCycle` function is provided as a convenience function to count the number of occurences of each base at each position in the read.

Use Case: Obtain a matrix of the base-frequency at each cycle and compare how it changes along the read by making separate lines for each base.

```
abc <- alphabetByCycle(sread(fq))
dim(abc)
plot(abc[1,],type="n",ylim=range(0,max(abc)))

text(abc[1,],labels="A",col="red")
text(abc[2,],labels="C",col="orange")
text(abc[3,],labels="G",col="green")
```

[1]`http://www.bioinformatics.babraham.ac.uk/projects/fastqc/`

```
text(abc[4,],labels="T",col="blue")
```

ℹBy using `type='n'` we tell the `plot` function to create an empty plot with axes and labels, but no points. We add details to this empty plot using the `text` function.

Use Case: Calculate the GC content of each read using the `gcFunction` function in the `scripts` folder. Find any reads with over 90% GC bases

```
source("scripts/gcFunction.R")



gc <- gcFunction(sread(fq))
hist(gc)

sread(fq)[gc > 0.9]
```

We can also seek to find how many times each sequence occur in the file. Over-represented sequences could be adaptor sequences, or other artefacts.

Use Case: Find the most-commonly ocurring sequences in the file

```
tbl <- tables(fq)
names(tbl)
tbl$top[1:5]
tbl$distribution
```

If we see an unacceptable drop in quality at the end of a read, we may wish to trim the sequences before aligning.

Use Case: Convert the quality scores into a more manageable format and visualise the quality drop-off towards the end of the read.

```
qual <- as(quality(fq), "matrix")
dim(qual)
plot(colMeans(qual), type="b")
abline(h=26)
```

For illustration purposes, we will choose a minimum cut-off of 26.

Use Case: Trim the sequences to ensure a mean quality of 26 and construct a new `fastq` file from the trimmed sequences and qualities.

```
minQual <- 26
trimL <- min(which(colMeans(qual) < minQual))

trimmed.seq <- subseq(sread(fq),1,trimL)
trimmed.seq
quals <- subseq(quality(fq)@quality,1,trimL)
```

```
names(trimmed.seq) <- id(fq)
writeXStringSet(trimmed.seq, file="trimmed.fq", format="fastq",qualities=quals)
```

4 Ranges and GRanges

4.1 Ranges

IRanges are another fundamental concept in Bioconductor used to represent consecutive integers. We will illustrate the usage of the package using simple illustrative examples before introducing genomic data later on.

Use Case: Construct an *IRanges* object with start positions between 50 and 150 and varying lengths between 1 and 25

```
library(IRanges)
starts <- floor(runif(n=50,min=50,max=150))
ends <- starts + floor(runif(n=50,min=1,max=25))
query <- IRanges(start = starts, end=ends)
query <- sort(query)
```

A number of *Intra-range* operations are implemented that act on each range independantly. You may find this function from UC Riverside HT-sequencing guide by Thomas Girke [2] useful in understanding the ranges objects.

```
source("scripts//plotRanges.R")
plotRanges
```

Use Case: Experiment with the basic operations `shift` and `resize`.

```
shift(query,5)
plotRanges(shift(query,5))
plotRanges(resize(query,4))
```

On the other hand, *Inter-range* methods operate on a whole collection of ranges. Examples include `reduce`, `gaps` and `coverage`.

```
plotRanges(reduce(query))
```

Coverage (or sometimes depth) is the operation for calculating how many ranges overlap each position. In the *IRanges* implementation of this calculation, the output is a Run-length encoding (`RLE`).

Use Case: Calculate the coverage of the example IRanges object. Make sure you understand the run-length encoding output.

[2] http://manuals.bioinformatics.ucr.edu/home/ht-seq

```
coverage(query)
plot(coverage(query))
```

Comparisons of *IRanges* objects are crucial for many downstream analyses of sequencing data. e.g. counting reads within particular genes. Many such comparisons are supported by the *IRanges* package. Typically, two sets of ranges are required; a `query` and a `subject`. However, the nature of the output depends on the function.

We will illustrate these functions by constructing windows that start at even intervals, but have varying length

Use Case: Define a series of windows that start at regular intervals between 30 and 160, but with varying lengths

```
start2 <- seq(30,160,by=10)
subject <- IRanges(start = start2, width=sample(1:10,length(start2),replace=T))

par(mfrow=c(3,1))
plotRanges(query,xlim=c(0,150))
plotRanges(subject,xlim=c(0,150))
```

`precede` and `follow` are two ways of comparing Ranges objects For each range in `x`, `precede` returns the index of the interval in `subject` that is directly preceded by the `query` range. Overlapping ranges are excluded. `NA` is returned when there are no qualifying ranges in subject.

The opposite of `precede`, `follow` returns the index of the range in `subject` that a `query` range in `x` directly follows. Overlapping ranges are excluded. `NA` is returned when there are no qualifying ranges in subject.

Use Case: Use `precede`, `follow` and `nearest` on the `query` and `subject` objects that we have created.

```
prec <- precede(query, subject)
foll <- follow(query, subject)
query[1:10]
subject[prec[1:10]]
subject[foll[1:10]]
nearest(query,subject)
```

The degree of overlap between `query` and `subject` ranges can be investigated in a number of ways. The `findOverlaps` function is the simplest way of finding which intervals in one set overlap another. The returned object stored the indices of the query and subject ranges that overlap. The `countOverlaps` is a convenient way of counting the number of ranges in the query overlap with the subject. If we are interested in the actual ranges where an overlap occurs (or does not occur), we can use the `intersect` and `setdiff` functions.

Use Case: Investigate the degree of overlap between the `query` and `subject` ranges.

```
olaps <- findOverlaps(subject, query)
queryHits(olaps)
countOverlaps(query, subject)
countOverlaps(subject, query)


plotRanges(intersect(query,subject),xlim=c(0,200))
plotRanges(setdiff(query,subject),xlim=c(0,200))
```

4.2 GRanges

GRanges objects are an extension of IRanges that can incorporate Genomic information such as chromosome names (commonly called seqnames) and strand information.

Use Case: Construct a GRanges object from the query and subject data in the previous exercise, allocating each range to sequences **A**, **B** or **C**. Explore the relationship between the objects as in the previous exercises.

```
library(GenomicRanges)
gr <- GRanges(seqnames=sample(LETTERS[1:3],length(query),replace=T), ranges=query)
gr <- sort(gr)

gr2 <- GRanges(seqnames=sample(LETTERS[1:3],length(subject),replace=T), ranges=subject)
gr2 <- sort(gr2)
shift(gr,5)
reduce(gr)
gaps(gr)
gaps(gr2)
coverage(gr)
coverage(gr2)
```

ℹIn later practicals, we will see how can perform overlaps of genomic features using this same infrastructure.

5 Reading bam alignments

An obvious application of the *IRanges* and *GRanges* infrastructure is to facilitate the storage and querying of genomic alignments. The *Rsamtools* is one such package that allows aligned reads to be imported into R. However, we will use functionality from the *GenomicRanges* package for simplicity. In your own time, feel free to check out the scanBam function in *Rsamtools*.

An example bam file is provided in the exampleData folder.

Use Case: Read the example bam file into R as a `GRanges` object. How many reads are in the file?

```
mybam <- "exampleData/NA19914.chr22.bam"
bam <- readGAlignments(file=mybam)
bam
```

The essential information from the bam file has been imported. Finer grain control over what data are read can be achieved by using the `param` argument. The object returned by `readGAlignments` acts like a data frame, with columns that can be assessed by a number of convenience functions `strand`, `width`, `cigar`, `start`, `end`

Use Case: Use accessor (`strand`, `width`, `cigar`, `start`, `end`) functions to explore the `bam` object.

```
table(strand(bam))
table(width(bam))
range(start(bam))
head(sort(table(cigar(bam)), decreasing=TRUE))
```

Several range operations will work on the GappedAlignment object, such as `coverage`.

Use Case: Create a coverage vector and identify regions with more than 1000 reads covering

```
cov <- coverage(bam)
HighCov <- ranges(slice(cov, 1000))[["22"]]
HighCov
```

An efficient way of creating a subset of reads based on genomic location is to use the overlapping functionality that we saw previously.

Use Case: Restrict the alignments to just those in the high-coverage (say ¿ 1000 reads) regions

```
HighCovGR <- GRanges("22", IRanges(start = start(HighCov), end=end(HighCov)))
bam.sub <- bam[bam %over% HighCovGR,]
bam.sub
length(bam.sub)
```

Use Case: Use `findOverlaps` and `countOverlaps` to find out how many reads align to these high-coverage regions.

```
findOverlaps(HighCovGR,bam)
countOverlaps(HighCovGR,bam)
```

5.1 Reading a particular genomic region

If we were only really interested in a relatively small region of a chromosome, then another option would be to read the whole bam file, especially if the file can run to several gigabytes in size. As the bam file is indexed, we can easily jump to the location we are interested in. This is achieved by using the `ScanBamParam` function with the GRanges object for the region we want to read. See the help page on `ScanBamParam` for more details about how to customise how to read a bam file.

Use Case: Read a portion of the bam file corresponding to the high-coverage regions rather the whole file. Verify that the same reads as the previous exercise are returned.

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

The previous two exercises have restricted the analysis to reads that fall inside any of the high-coverage regions. However, we might want to look at each region separately.

Use Case: Classify the reads according to which high-coverage region they belong to. Verify that the correct number of overlapping reads are returned.

```
names(HighCovGR) <- paste("Region", 1:length(HighCov),sep="")
regionLst <- split(HighCovGR, names(HighCovGR))

splitRegions <- lapply(regionLst, function(x) bam[bam %over% x])

lapply(splitRegions,length)
```