Biostrings Lab (BioC2007)

H. Pagès
Gentleman Lab
Fred Hutchinson Cancer Research Center
Seattle, WA

6 August, 2007

Introduction

The Biostrings package: provides the infrastructure for storing and manipulating large nucleotide and amino acid sequences (up to hundreds of millions of letters) in R.

We need R version 2.6 and Biostrings version 2.5.19 for this lab:

```
> library(Biostrings)
> sessionInfo()

R version 2.6.0 Under development (unstable) (2007-08-14 r42506)
x86_64-unknown-linux-gnu

locale:
LC_CTYPE=en_CA.UTF-8;LC_NUMERIC=C;LC_TIME=en_CA.UTF-8;LC_COLLATE=en_CA.UTF-8;LC_MONETARY=en_

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

other attached packages:
[1] Biostrings_2.5.19

loaded via a namespace (and not attached):
[1] rcompgen_0.1-15
```

The BSgenome.* packages: data packages containing the full genome sequence for a given organism. Ex: BSgenome.Celegans.UCSC.ce2, BSgenome.Hsapiens.UCSC.hg18, etc... Use the `available.genomes()` function from the BSgenome package to see the list of available BSgenome.* packages (network access required):

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

```
 [1] "BSgenome.Celegans.UCSC.ce2"
 [2] "BSgenome.Dmelanogaster.BDGP.Release5"
 [3] "BSgenome.Dmelanogaster.FlyBase.r51"
 [4] "BSgenome.Dmelanogaster.UCSC.dm2"
 [5] "BSgenome.Hsapiens.UCSC.hg16"
 [6] "BSgenome.Hsapiens.UCSC.hg17"
 [7] "BSgenome.Hsapiens.UCSC.hg18"
 [8] "BSgenome.Mmusculus.UCSC.mm6"
 [9] "BSgenome.Mmusculus.UCSC.mm7"
[10] "BSgenome.Mmusculus.UCSC.mm8"
[11] "BSgenome.Scerevisiae.UCSC.sacCer1"
```

Then install with `biocLite()`:

```
> source("http://bioconductor.org/biocLite.R")
> biocLite("BSgenome.Dmelanogaster.FlyBase.r51")
```

Note that those BSgenome.* packages are big (from a few Mb to more than 800 Mb for BSgenome.Hsapiens.UCSC.hg18) so this installation over the network can take a long time.

BString objects

The *BString* class is the basic container for big sequences. Unlike standard character vectors in R that can store an arbitrary number of strings, a *BString* object can only contain 1 string. For sequences commonly found in biology, 3 variants of the *BString* class are available:

1. The *DNASstring* class for storing a DNA sequence.
2. The *RNAString* class for storing a RNA sequence.
3. The *AAString* class for storing a sequence of amino acids.

For each of these classes, there is a constructor that allows to create an instance of the class from a character string:

```
> s1 <- BString("hello")
> s2 <- DNAString("GACCCT")
> s3 <- AAString("MARKSLEMSIR")
```

Exercise 1

1. Create some arbitrary *BString*, *DNAString* and *AAString* instances.
2. Apply `nchar` and `alphabet` on them.
3. Apply `alphabetFrequency` and `reverseComplement` on the *DNAString* objects.

4. Extract substrings by using the subsetting operator [and by using the subBString function.

IMPORTANT: The subBString function should always be preferred to the subsetting operator [for substring extraction. More on this later...

BSgenome.* packages

A BSgenome.* package is a data package containing the full genome sequence for a given organism.

Please refer to the introduction of this document for how to display the list of BSgenome.* packages currently available and how to install them over the network.

The name of a BSgenome.* package is made of 4 parts separated by a dot (e.g. BSgenome.Celegans.UCSC.ce2). The 1st part is always BSgenome, the 2nd part is the name of the organism (abbreviated), the 3rd part is the name of the organisation who assembled the genome and the 4th part is the release string or number used by this organisation for this assembly of the genome.

A BSgenome.* package contains a single top level object whose name matches the second part of the package name:

```
> library(BSgenome.Celegans.UCSC.ce2)
> Celegans

C. elegans genome:
- organism: Caenorhabditis elegans
- provider: UCSC
- provider version: ce2
- release date: Mar. 2004
- release name: WormBase v. WS120
- single sequences (DNAString objects, see '?seqnames'):
    chrI    chrII   chrIII  chrIV   chrV    chrX    chrM
- multiple sequences (BStringViews objects, see '?mseqnames'):
    upstream1000  upstream2000  upstream5000
  (use the '$' or '[[' operator to access a given sequence)
```

Displaying this object shows some information about the provenance of this genome plus 2 indexes of sequences: the single sequences and the multiple sequences. The single sequences are *DNAString* objects and the multiple sequences are *BStringViews* objects (will be introduced soon).

Exercise 2

1. Display several chromosomes of C. elegans. Do they contain IUPAC extended letters?

2. Load BSgenome.Dmelanogaster.FlyBase.r51 (fruit fly genome). Do the chromosomes contain IUPAC extended letters?
3. Extract the last 10M bases from fruit fly chromosome 2L. Use the `system.time` function to compare the performance of `subBString` vs the subsetting operator `[`.

BStringViews objects

A *BStringViews* object contains a set of views on the same sequence called "the subject". Each view is defined by its start and end locations: both are integers such that start <= end. We use the `views` function to create a *BStringViews* object for a given set of start and end locations. Here we create a *BStringViews* object with 4 views:

```
> v <- views(DNAString("TAATAATG"), 3:0, 5:8)
> v
```

```
  Views on a 8-letter DNAString subject
Subject: TAATAATG
Views:
    start end width
[1]     3   5     3 [ATA]
[2]     2   6     5 [AATAA]
[3]     1   7     7 [TAATAAT]
[4]     0   8     9 [ TAATAATG]
```

Note that two views can overlap and that a view can be "out of limits" i.e. it can start before the first letter of the subject or/and end after its last letter.

The subject is the DNA sequence `TAATAATG`:

```
> subject(v)
```

```
  8-letter "DNAString" instance
Value: TAATAATG
```

A *BStringViews* object can be subsetted like a standard character vector:

```
> length(v)
```

```
[1] 4
```

```
> v[4:2]
```

```
  Views on a 8-letter DNAString subject
Subject: TAATAATG
Views:
    start end width
[1]     0   8     9 [ TAATAATG]
[2]     1   7     7 [TAATAAT]
[3]     2   6     5 [AATAA]
```

The start/end/width (integer vectors) can be extracted with `start()`, `end()` and `width()`:

```
> start(v)

[1] 3 2 1 0

> end(v)

[1] 5 6 7 8

> width(v)

[1] 3 5 7 9
```

A given view can be extracted as a *BString* (or derived) object with the `[[` operator:

```
> v[[2]]

  5-letter "DNAString" instance
Value: AATAA
```

Note this operator returns an instance of the same class as the subject.

Exercise 3

1. Compare `nchar` vs `width` on `v`.
2. Find a programatical way to remove ”out of limits” views from a *BStringViews* object.

Exercise 4

1. Use the `mask` function to mask the `N`s in fruit fly chromosome 2L. Apply `mask` again to the result (and without the second argument) to see the location of the `N`s. Do we have isolated `N`s or `N`-blocks?
2. Find the location of the `N`s in C. elegans chromosome III and in H. sapiens chromosome 1.
3. What's the longest `A`-block in fruit fly chromosome 2L?

Exercise 5

The objective of this exercise is to ”look at” the introns of a given eukaryote. For this we need:

1. The BSgenome.* package with the full genome sequence of the target organism.

2. Annotations that give us the gene and exon locations of the organism. Note that it's important to make sure that these annotations are targetting exactly the version (build/assembly) of the genome that we are using.

3. The Biostrings package.

We do this analysis on the fruit fly genome: we use genome r5.1 from FlyBase (the BSgenome.Dmelanogaster.FlyBase.r51 package) and a package of annotations for this genome, the ann.Dmelanogaster.FlyBase.r51 package, made from the GFF files provided by FlyBase:

```
> library(ann.Dmelanogaster.FlyBase.r51)
```

The annotations in this package are organized as follow: `CHR_SHORTNAMES` gives the list of chromosomes for which annotations are provided:

```
> CHR_SHORTNAMES
```

```
[1] "2L" "2R" "3L" "3R" "4"  "X"  "M"
```

and for each chromosome, 2 data frames are provided: 1 for the genes and 1 for the exons:

```
> colnames(getAnnGenes("2L"))
```

```
 [1] "seqname"              "source"               "start"
 [4] "end"                  "strand"               "attrib"
 [7] "Name"                 "Alias"                "cyto_range"
[10] "putative_ortholog_of" "Dbxref"
```

```
> colnames(getAnnExons("2L"))
```

```
[1] "seqname" "source"  "start"   "end"     "strand"  "Name"    "Parent"
[8] "Dbxref"
```

For our analysis, we only need the `"seqname"`, `"start"`, `"end"` and `"strand"`.

```
> getAnnGenes("2L")[1:5, c("seqname", "start", "end", "strand")]
```

```
            seqname start   end strand
FBgn0031208      2L  7529  9491      +
FBgn0002121      2L  9836 21372      -
FBgn0031209      2L 21919 23888      -
FBgn0051973      2L 25402 59242      -
FBgn0067779      2L 67044 71390      +
```

```
> getAnnExons("2L")[1:5, c("seqname", "start", "end", "strand")]
```

```
          seqname start   end strand
CG11023:1      2L  7529  8116      +
CG11023:2      2L  8229  8589      +
CG11023:3      2L  8668  9491      +
CG2671:9       2L  9836 11344      -
CG2671:8       2L 11410 11518      -
```

The row names are gene ids for the former and exon ids for the later.

Finally, the data object `EXON2GENE` (named character vector) provides the mappping between exon ids and gene ids (many-to-one relationship):

```
> data(EXON2GENE)
> EXON2GENE["CG11023:1"]

    CG11023:1
"FBgn0031208"
```

To get all the exons belonging to a given gene:

```
> names(EXON2GENE)[EXON2GENE == "FBgn0031208"]

[1] "CG11023:1" "CG11023:2" "CG11023:3"
```

1. Extract the start and end position of the exons belonging to gene FBgn0025803 (chromosome 3R).
2. Load BSgenome.Dmelanogaster.FlyBase.r51 and create a *BStringViews* object where the subject is chromosome 3R and the views are the exon locations found previously.
3. Apply the `mask` function on this *BStringViews* object. Is this new *BStringViews* object representing the introns sequences for gene FBgn0025803? What needs to be adjusted?
4. Are we discovering something?

The `matchPattern` function

This function finds all matches of a given pattern in a sequence. Using `matchPattern` on a *BString*, *DNAString* or *AAString* object is much faster than working with regular expressions, be it in R (on a standard character string) or in another language like Python.

```
> library(BSgenome.Hsapiens.UCSC.hg18)
> chr1 <- Hsapiens$chr1
> system.time(m <- matchPattern("AAAAAAAAAATT", chr1))

   user  system elapsed
  0.384   0.000   0.386
```

```
> m

  Views on a 247249719-letter DNAString subject
Subject: TAACCCTAACCCTAACCCTAACCCTAACCCTAAC...NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
Views:
            start       end width
   [1]      34037     34048    12 [AAAAAAAAAATT]
   [2]     154549    154560    12 [AAAAAAAAAATT]
   [3]     403863    403874    12 [AAAAAAAAAATT]
   [4]     538110    538121    12 [AAAAAAAAAATT]
   [5]     538654    538665    12 [AAAAAAAAAATT]
   [6]     690385    690396    12 [AAAAAAAAAATT]
   [7]     729676    729687    12 [AAAAAAAAAATT]
   [8]     832484    832495    12 [AAAAAAAAAATT]
   [9]    1027883   1027894    12 [AAAAAAAAAATT]
   ...        ...       ...   ... ...
[5126] 245825662 245825673    12 [AAAAAAAAAATT]
[5127] 245946178 245946189    12 [AAAAAAAAAATT]
[5128] 246110416 246110427    12 [AAAAAAAAAATT]
[5129] 246114017 246114028    12 [AAAAAAAAAATT]
[5130] 246248087 246248098    12 [AAAAAAAAAATT]
[5131] 246282785 246282796    12 [AAAAAAAAAATT]
[5132] 246460105 246460116    12 [AAAAAAAAAATT]
[5133] 247068709 247068720    12 [AAAAAAAAAATT]
[5134] 247177225 247177236    12 [AAAAAAAAAATT]
```

Also, regular expressions can miss some matches:

```
> length(gregexpr("ATA", "ATATA"))

[1] 1
```

`matchPattern` returns all matches:

```
> length(matchPattern("ATA", "ATATA"))

[1] 2
```

Exercise 6

1. Find all the matches of an arbitrary nucleotide sequence in fruit fly chromosome 2L.
2. In fact, if we don't take any special action, we only get the hits in the plus strand of the chromosome. Find the matches in the minus strand too. IMPORTANT: You should always avoid to `reverseComplement` an entire chromosome sequence (this is very inefficient).

Exercise 7

The objective of this exercise is to check that, for a given probe of a given chip, the hit location found by `matchPattern` is in agrement with the location reported in the annotations available for this chip. We use Affymetrix human chip `hgu95av2` for this analysis.

1. Use `hgu95av2` to find the chromosome location of the gene targetted by probeset `1138_at`.
2. Use `hgu95av2probe` to find all the probe sequences for this probeset id (put the result in a character vector called `probes`).
3. Use `matchPattern` to find the corresponding hit for each probe in the corresponding chromosome (use a `sapply(probes, function(probe) ...)` construction to put all the results in a single list, the returned list will have one element per probe in `probes`). Do we have a hit for every probe? Compare with `hgu95av2CHRLOC[["1138_at"]]`. Do you see something wrong?
4. Try again with `mismatch=1`. Do we have a hit for every probe?
5. In fact, according to `hgu95av2probe`, probeset `1138_at` should hit a gene in the antistrand. Does this look correct?

Exercise 8

IUPAC extended letters can be used to express ambiguities in the pattern or in the subject of the search. This is controlled via the `fixed` argument of the `matchPattern` function. If `fixed` is `TRUE` (the default), all letters in the pattern and the subject are interpreted litterally. If `fixed` is `FALSE`, IUPAC extended letters in the pattern and in the subject are interpreted as ambiguities e.g. `M` will match `A` or `C` and `N` will match any letter (the `IUPAC_CODE_MAP` named character vector gives the mapping between IUPAC letters and possible nucleotides they stand for). The most common use of this feature is to introduce wildcards in the pattern by replacing some of its letters with `N`s.

1. Search pattern `GAACTTTGCCACTC` in fruit fly chromosome 2L.
2. Repeat but this time allow the 2nd `T` in the pattern (6th letter) to match anything. Anything wrong?
3. You can use the `mask` function or `fixed="subject"` to work around the previous problem. Try and compare.

Exercise 9

The objective of this exercise is to simulate a *PCR experiment* on the fruit fly genome. A *PCR experiment* uses primer pairs specifically designed to target the genome of a given organism to "amplify" certain regions of this genome. Each primer pair is made of a forward and a reverse primer. Both primers are short nucleotide sequences with 2 parts: a gene-specific part (the "probe") and a "tag" used for the amplification processus. We'll call the forward probe and the reverse probe the probe parts of the forward and reverse primer respectively. For each primer pair, the forward and reverse probes are designed to hit the same chromosome at nearby locations. The region between the 2 hits is called an amplicons: it is the region that gets "amplified" by the PCR processus.

For our *computer-simulated PCR experiment*, we'll use the small library of primer pairs stored in the `plate1_probes.txt` file. This library can be loaded into a data frame with:

```
> probes <- read.table("plate1_probes.txt", stringsAsFactors = FALSE)
> dim(probes)
```

```
[1] 95  2
```

```
> probes[1:5, ]
```

```
                              Fprobe                 Rprobe
plate1_A02 GAGACCACCGCCGCAACTGAAG GAGACCACCGCCGCAACTGAAG
plate1_A03   AGCTCCGAGTTCCTGCAATA   CGTTGTTCACAAATATGCGG
plate1_A04   ACCCAGCAAAAAGAAGAGCA   AGCAGTTCGGACCTCTTGAA
plate1_A05   AGTTAAGGGCCTGTGGTGTG   TACTGTTGTCGCGATTCAGC
plate1_A06   AGACGGTGGTGAACACATGA   CTTTTCGAGAGTGGAGTCCG
```

We've put one primer pair per row in this data frame. The row names of the data frame contain the primer pair ids. Also note that, as suggested by the col names, we've only put the probe parts for each primer pair (the "tag" part has been removed, it can be ignored for the purpose of this exercise).

The first goal of this exercise is to write a function that, given a probe pair and a subject (the PCR target), returns the amplicons associated with this probe pair.

In order to acquire some expertise, we'll try to find the amplicon(s) associated with the following arbitrary probe pair:

```
> Fprobe0 <- DNAString("AGCTCCGAGTT")
> Rprobe0 <- DNAString("CGTTGTTCACA")
```

We choose short probes on purpose so the risk to have several hits in several chromosome is high. This is a good way to learn how to deal in a systematical way with the multi hit problem!

1. Find all `Fprobe0` and `Rprobe0` hits in chromosome 3R. Remember that nothing prevents a probe to hit the strand that it was not designed for! Put the start positions of all the "plus hits" (i.e. the hits of `Fprobe0` and `Rprobe0` in the plus strand) in the `Phits` object (integer vector). Put the end positions of all the "minus hits" in the `Mhits` object (integer vector).

2. Ideally `Phits` and `Mhits` should be of length 1 but it is not always the case. However the fact that we get several ”plus hits” and/or ”minus hits” doesn’t necessarily mean that our original primer pair has a bad design. In fact, some of these hits don’t prevent the PCR experiment to work properly: this is the case for any ”plus hit” that is followed (from left to right) by another ”plus hit” with no ”minus hit” between. And similarly, it is the case for a ”minus hit” that is preceded by another ”minus hit” with no ”plus hit” between. All these hits can be ignored. Write a function that takes 2 arguments, `Phits` and `Mhits`, that removes all the above hits. The function will return a data frame with 2 columns, `Phits` and `Mhits`. Note that using a data frame now makes sense because after this removal, `Phits` and `Mhits` will always have the same length.

3. Use the previous function to ”clean” `Phits` and `Mhits`. Compare your result with the ”theoretical amplicons” returned by `matchProbePair(Fprobe0, Rprobe0, Dmelanogaster[["3R"]])`. From now we will use the `matchProbePair` function to achive this. Are all these ”theoretical amplicons” realistic (PCR amplification is supposed to work for regions up to 5k-10k bases). Would you still consider the `Fprobe0`/`Rprobe0` pair good for amplifying the region it was designed for?

4. Another problem that can prevent the PCR amplification from working properly is when a given primer pair produces more that 1 valid amplicon (e.g. <= 20k) over the whole genome. What about `Fprobe0`/`Rprobe0`?

5. Write a function that takes our probes library (the `probes` data frame) as input and produces a data frame similar to `probes` but with 3 additional columns `chr`, `start` and `end` that contain the chromosome name/start/end of the amplicon found for each primer pair or NAs if 0 or more than 1 amplicon were found. Having NAs in a row means that the design of the primer pair is bad.