Introduction to R: Basic string and DNA sequence handling

Manuela Benary and Pål O. Westermark

Strings and other data structures

So far, we have concentrated on numerical data. However, can we store characters too? Let's try working with the alphabet of the DNA. In this section we will search for occurences of the hormone response element in the promoter of a gene.

```
nucleobase <- T
print(nucleobase)
```

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

```
nucleobase <- "T"
print(nucleobase)
```

```
## [1] "T"
```

Here in the first example, the variable 'nucleobase' will have the value 'TRUE'. 'T' and 'F' are the short form for boolean 'TRUE' and 'FALSE'. So, as you can see, characters need to be surrounded by quotation marks (""). We can compare different strings with '==', and test whether they are identical.

```
nucleobase == "T"
```

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

```
nucleobase == "A"
```

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

Base complement
Please write a function 'basecomp' to translate nucleotides into their complement.

If you considered all cases, testing your function would give you something like

```
basecomp("A")
```

```
## [1] "T"
```

```
basecomp("T")
```

```
## [1] "A"
```

```
basecomp("P")
```

```
## Warning in basecomp("P"):  You have entered an invalid base
```

But there's an easier way. Let's create a vector of the bases (similar to a vector of numbers)

```
thebases <- c("A", "C", "G", "T")
thebases[2]
```

```
## [1] "C"
```

```
names(thebases)
```

```
## NULL
```

```
names(thebases) <- c("T", "G", "C", "A")
thebases["C"]
```

```
##   C
## "G"
```

Each element in a vector can be accessed either by an index or by giving the element a name. The function 'names' assigns the names for the individual elements of a vector.

Of course, we can store also whole sequences of characters. One example is the sequence of the hormone response element, which can be bound by the activated steroid receptor and thus initiates gene expression (Figure 1). A second example is the TATA-box, which can be found in the core promoter region of 10–20% of human genes. We can combine different strings in a vector as well:

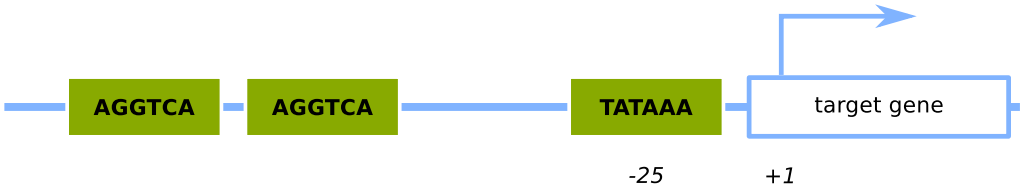

Figure 1: A hypothetical example of an eukaryotic core promoter including the TATA-box and the (dimeric) binding site of a hormone receptor.

```
hre <- "AGGTCA"
print(hre)
```

```
## [1] "AGGTCA"
```

```
dnaseq <- c(hre, "TATAAA")
print(dnaseq)
```

```
## [1] "AGGTCA" "TATAAA"
```

```
print(dnaseq[2])
```

```
## [1] "TATAAA"
```

We can easily split strings using a separating character and we will end up with a list of characters. A list can hold all types of different things. The list is your first encounter with a *data structure*. Lists work like boxes within boxes. If we have a vector of strings, the result for each string will be added as a new element in the list. Each element in the list can be accessed via `[[]]` (mind the difference: for vectors only one bracket is necessary).

```
dnabases <- strsplit(dnaseq, split=NULL)
print(dnabases)
```

```
## [[1]]
## [1] "A" "G" "G" "T" "C" "A"
##
## [[2]]
## [1] "T" "A" "T" "A" "A" "A"
```

```
print(dnabases[[1]])
```

```
## [1] "A" "G" "G" "T" "C" "A"
```

```
dnabases[[2]] <- 11
print(dnabases)
```

```
## [[1]]
## [1] "A" "G" "G" "T" "C" "A"
##
## [[2]]
## [1] 11
```

Here, we used a *named argument*, 'split'. This tells us at which characters we want to split our strings. If set to NULL, like above, this means splitting between all characters. As you can see, you can mix numbers and characters in your list and R will not complain. If we want to get the complement of the HR-element, we can use the function 'basecomp' from before and apply this function to each nucleotide in our sequence. One way is using a for-loop again (if you do not feel confident yet, give it another try). However, the better way (faster and efficient) is to use the function 'apply', in particular 'lapply'. 'l' stands for 'list' here. A general overview of apply-functions in R can be found at Neil Saunders Blog.

```
dnacomplist <- lapply(dnabases[[1]], basecomp)
print(dnacomplist)
```

You got back a list, but this is not the easiest way to handle strings. Let's use 'sapply' ('s' stands for 'simplify'), which you already have encountered:

```
dnacompvec <- sapply(dnabases[[1]], basecomp)
print(dnacompvec)
```

Lists and vectors

- Print the third element of 'dnacompvec' and of 'dnacomplist'. Are they the same?
- Use the vector 'thebases' to define the complement of the first element of 'dnabases'. Remember, you can use a vector to access elements of a vector.

In general, DNA sequences are stored as a single strand. If we want to find occurences of the HRE we have to consider the original pattern as well as the reverse complement: in general proteins can bind to any string, sometimes even in any orientation. We already know how to generate the complement, but we need to reverse the sequence to generate the reverse complement.

```
1:5
```

```
## [1] 1 2 3 4 5
```

```
rev(1:5)
```

```
## [1] 5 4 3 2 1
```

```
dnaseqcompt <- dnacompvec[rev(1:length(dnacompvec))]
paste(dnaseqcompt, collapse="")
```

```
## [1] "TGACCT"
```

(There are other ways too to reverse, index, and slice vectors, but we cannot cover all here.)

Reverse complement
Write a function 'revcomplement', which returns the reverse complement of a given DNA sequence.

Testing the function with the HR-element should result in the following.

```
revcomplement(hre)
```

```
## [1] "TGACCT"
```

Finally, we have the building blocks to actually search for occurences of the hormone receptor element. We will use the

promoter of the gene 'Prdx1' as an example here.

Download a gene sequence
Go to `genome.ucsc.edu` and download 3000 bp upstream of the transcription start site of the gene 'Prdx1'. → Genome Browser, select Mammal / Mouse / mm10, type Prdx1. Click blue box with prdx1, upper left → Genomic Sequence. Select only Promoter/Upstream, fill in 3000 bases. One FASTA record per gene. Check All upper case, uncheck mask repeats. Save as/Speichern unter → choose name prdx1.fa and format Text. Choose the same directory as your R working directory, which you obtain with `getwd()`.

We will now read the sequence from the file line by line and store these in a vector. The first line should be the header, which we do not keep. And we will collapse the array of strings into one gigantic string.

```
rawseq <- readLines("prdx1.fa")

prdx1arr <- rawseq[2:length(rawseq)]
prdx1seq <- paste(prdx1arr, collapse="")
```

You can check whether everything went right by counting the number of characters in your string using the function `nchar()`. We are now going to write code to count the occurences of a given string in this DNA sequence. As always in good engineering tradition, we approach the problem by creating small building blocks, which we later combine to achieve our goal. The first building block is a function for picking out substrings:

```
substr(prdx1seq, 1, 2)
```

```
## [1] "TG"
```

Substrings

- Extract the bases from position 4 to 9.
- Using substr and nchar, extract the last 6 bases of the prdx1 gene.

If we want to look for occurrences of the motif AGGTCA, we need the motif (should be stored as 'hre') and its reverse complement.

```
comp.hre <- revcomplement(hre)
comp.hre
```

```
## [1] "TGACCT"
```

```
len <- nchar(hre)
```

Next, we want to extract all (overlapping) substrings of the length of the motif ('len') in our sequence, which would be a reoccuring use of the function 'substr':

- substr(prdx1seq, 1, len)
- substr(prdx1seq, 2, len + 1)
- ...
- substr(prdx1seq, nchar(prdx1seq) - (len - 1), nchar(prdx1seq))

This looks similar to the for-loop (or even easier an apply-function) we have used before. So let's do this systematically: We need the starting index and the ending index for the substring. We need to be real careful at the end of the gene sequence to make sure that the substring is as long as the motif.

```
leftlims <- 1:(nchar(prdx1seq) - (len - 1))
rightlims <- len:nchar(prdx1seq)
```

`substr()` is a function which has two arguments, therefore we cannot use the simple 'apply'. However, we can use 'mapply' to apply a function to each element of multiple arguments (see Figure 2]).

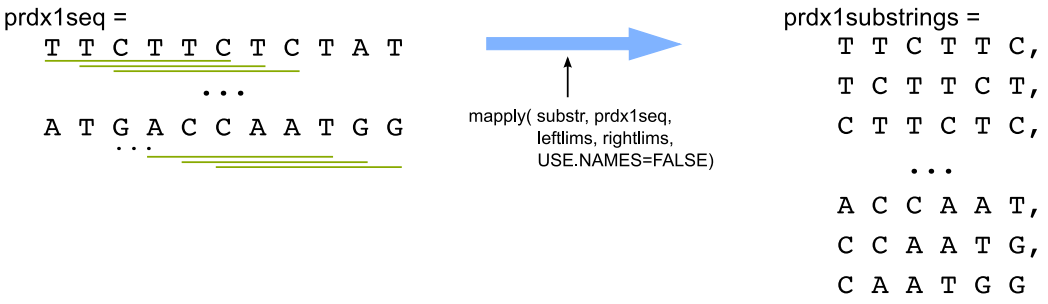

Figure 2: Dissecting a large sequence into a vector of overlapping fragments using the function 'mapply'.

```
prdx1substrings <- mapply(substr, prdx1seq, leftlims, rightlims,
                          USE.NAMES=FALSE)
head(prdx1substrings)
```

```
## [1] "TGATGA" "GATGAC" "ATGACC" "TGACCT" "GACCTG" "ACCTGA"
```

We combined 'substr' and 'mapply' to split our sequence into a vector of overlapping fragments. The next step is a check whether a fragment is a hit. This check should be written as a function as it is to be reused multiple times.

```
hitcounter <- function(fragment, themotif) {
  if (fragment==themotif)
    return(1)
  else
    return(0)
}
```

The if-else-statement can be written in a shorter way using the function 'ifelse'. This function condenses the return statements into one function.

Finding occurences of HRE

- Test your function 'hitcounter' with fragment number 1 and 4.
- Try writing a shorter version of 'hitcounter' using 'ifelse' (if you are not sure what to do, use the help pages of R/RStudio).
- Extend your function 'hitcounter' to search for the reverse complement as well. Test your function again with fragment number 1 and 4.
- *Bonus:* What happens if you give your function 'hitcounter' your complete vector instead of one element?

So, we created our own building block that returns a 1 whenever a fragment matches a given motif. If we now apply this function to all our fragments, we obtain a vector with mostly 0's and a few 1's, as seen in the last exercise. If we sum these numbers up ...

```
scores <- hitcounter(prdx1substrings, hre)
nhits <- sum(scores)
nhits
```

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

... we get the total number of hits in our sequence. As we may want to use that in other sequences or with other motifs as well, we will wrap everything up in a function called 'counthits'

```
counthits <- function(sequence, motif) {
  compmotif <- revcomplement(motif)
  len <- nchar(motif)
  leftlim  <- 1:(nchar(sequence) - (len - 1))
  rightlim  <- len:nchar(sequence)

  frags  <- mapply(substr, sequence, leftlim, rightlim, USE.NAMES=FALSE)
  scores <- ifelse(frags==motif|frags==compmotif, 1, 0)
  return(sum(scores))
}
```