SØREN WENGEL MOGENSEN

EXERCISES IN R

1
Exercises

The structure of the sections is the same as of the lectures. The first section of exercises can be solved after the first lecture and so forth. Some exercises have hints on the last pages of the document.

1 *Vectors*

Exercise 1.1. This is a fundamental exercise on the creation and use of vectors in R.

(a) Create a vector v with entries 0,0,0,0,0, and 0

(b) Create a vector w with entries 0,0,0,1,1, and 1

(c) Create a vector x with entries 2, 4, 6, 8, and 10.

(d) Create a vector y with entries -1, 1, -1, 1, and -1.

(e) Create a vector z with entries 1, 2, 4, 8, 16, 32, 64.

For vectors $x, y \in \mathbb{R}^n$ we can calculate the inner product $x^t y$.

(f) Calculate the inner product of x and y. Remember that the result should be a single real number.

(g) Evaluate x + z. This gives a result even though the vectors are not of the same length. Explain what R does to obtain a result.

(h) Add the names a, b, ..., g to z and extract the 4th entry using the name.

(i) Discard the last entry of z to obtain a vector of length six.

○

Exercise 1.2. Let's have a closer look at how R interpretes vector operations when the vectors are not of equal length.

(a) Generate a vector $x = (1,2,3)^t$ and a vector $y = (1,2,3,4,5)^t$.

(b) What is the result of `x*y`? How does `R` make sense of the operation?

The way `R` makes sense of operations with vectors of unequal length is perhaps surprising but it's neat when generating certain vectors.

(c) Generate the vector $z_1 = (0,2,0,4,0,6)^t$.

(d) Generate the vector $z_2 = (5,-4,3,-2,1)^t$.

○

Exercise 1.3. Let's have a look at subsetting of vectors. Type in the vector $x = (4,9,1,4,5,7,10,1,4,5)^t$.

(a) Extract the subvector of `x` consisting of the 1st, 3rd, 5th, etc. elements.

(b) Extract the subvector of `x` consisting of all elements for which the previous element is strictly larger than 4 (ignore the first element of `x`).

(c) Extract the subvector of `x` consisting of all elements for which the sum of the previous and the following elements is strictly larger than 10 (ignore the first and last elements of `x`).

○

Exercise 1.4. This exercise is on mixing different data types. This can be a clever way of manipulating data.

(a) Try adding different data types, e.g. `TRUE + 1` or `TRUE + "a"`. What happens in each case? Explain the logic behind.

The *Boolean values* are in `R` denoted `TRUE` and `FALSE`, or `T` and `F`

Hopefully, you found some sort of translation between the Boolean values (`TRUE`/`FALSE`) and integers (or reals). We can exploit this relationship to make our code more readable. Run the following code.

```
set.seed(101)
x <- sample(c(TRUE, FALSE), size = 10, replace = TRUE)
```

(b) What does `sum(x)` return? How does this make sense?

(c) How does multiplication (*) apply to Boolean values and how can this be explained?

(d) Argue that in `R`, `+` is not defined for character strings. What would be a sensible definition?

○

Exercise 1.5. Sometimes it's useful to manipulate *character strings* in R. This exercise introduces the basic functions for doing so.

(a) Construct a vector `x` with entries `"a","b","c","d"`, and `"e"`.

(b) Construct a vector `y` with entries `"a1","b1","c1","d1"`, and `"e1"`.

(c) Construct a vector of length 1 with the entry `"a1b1c1d1e1"`.

After this quick warm-up, let's look at some more complicated examples. Run the following code to generate the vector `w`.

```
set.seed(101)
w <- replicate(10, paste(sample(letters, 6, replace = TRUE),
                         collapse = ""))
```

(d) Use the `substr` function to extract every 3rd and 4th character of each element of `w` and name the resulting vector `r`.

(e) Generate a vector `v` of length 10 in which the first element is the last in `w`, the second element of `v` is the second to last in `w` etc.

(f) In the previous question you *reversed* a vector. Now generate a vector `z` such that each element of `z` is the reverse of the corresponding element in `r`. That is, if the first element in `r` is `"sr"` then the first element in `z` should be `"rs"`.

○

2 *Matrices and arrays*

Exercise 2.1. In this exercise you will work with matrices.
Create two matrices in the following way.

```
M1 <- matrix(1:12, nrow = 3)
M2 <- matrix(1:12, nrow = 3, byrow = TRUE)
```

(a) What does the `byrow` argument do?

(b) Extract the top-left entry of `M1`, the middle row of `M2`, and the last column of `M2`.

(c) Explain why the following command results in an error,

```
M1%*%M2
```

and why the next one doesn't.

```
M1*M2
```

○

Exercise 2.2. The following commands define a matrix, then assign a different content to the second column, and finally attempt a multiplication of the first and third columns of the matrix which results in an error.

```
M <- matrix(1:9, nrow = 3, ncol = 3)
M[,2] <- letters[1:3]
t(M[,1])%*%M[,1]
```

(a) Run the commands and explain why an error occurs.

`R` works with different data types and this is related to the above question.

(b) Try to do something similar with other data types. Determine a hierarchy between them.[1]

(c) What are the options if an `R` user wishes to store information of different data types in a single object?

[1] By *hierarchy*, an ordering of data types is meant such data of a specific type can always be converted to another type higher in the hierarchy, but not necessarily to one lower.

○

Exercise 2.3. In this exercise, you'll work with subsetting of matrices as well as some "nice-to-know" functions for use on matrices.

(a) Familiarize yourself with functions `diag`, `col`, `row`, `upper.tri`, and `lower.tri`.

(b) Construct a 4×4 matrix `M1` which has 1's in the diagonal, 0's above the diagonal, and 2's below the diagonal.

(c) Construct another 4×4 matrix `M2` in which each entry is the product of the column index and the row index.

(d) Name the columns and rows of `M2`. Extract the central part of `M2`, that is, the 2×2 matrix corresponding the rows 2 and 3 and columns 2 and 3 in `M2` using both the column/row names and the column/row indices.

(e) Extract the elements of `M2` with elements strictly larger than 4.

○

Exercise 2.4. This exercise is about arrays in `R`. Arrays are a generalization of matrices to higher dimensions. Generate this three-dimensional array.

```
myarray <- array(seq(24), dim = c(4,3,2))
```

In `R` terminology, we say that some dimensions *vary faster* than others. In a matrix, the row dimension varies faster than the column dimension because (by default) `R` fills a matrix with a vector by first filling the first column, then the second etc.

(a) Print `myarray` to the screen. How is it displayed?

(b) What dimension varies the fastest, and which varies slowest?

For a `matrix` the function `t` gives you the transpose. For an `array`, the analogous operation is carried out by the `aperm` function.

(c) Construct a new `array` using `aperm` that has the same entries as `myarray` but has dimensions $3 \times 4 \times 2$.

○

Exercise 2.5. Table 1.1 is an example of a *contingency table* where each cell is a count of the frequence of a certain combination of values of the variables A, B, and C. Enter the data in an `array` and add appropriate names.

○

		C	
A	B	Yes	No
Yes	Yes	11	99
	No	14	2
No	Yes	45	7
	No	9	8

Table 1.1: Example of a contingency table.

Exercise 2.6. This exercise is about indexing of arrays.

(a) Take a look on the functions `row` and `col`. What do these functions return when you input a matrix?

(b) Write your own function that generalizes both `row` and `col`, and to arrays of any dimension.

○

3 *Loops and lists*

Exercise 3.1. Create a list in the following way.

```
mylist <- list("a")
```

(a) What is the difference between the commands `mylist[[1]]` and `mylist[1]`?

(b) Use a for-loop to iteratively build a list with elements `a`, `b`, `c`, ..., `z`.

(c) Assign names to the elements of the list and extract an element using the assigned name of that element.

○

Exercise 3.2. Write some code that keeps generating variates from a standard normal distribution until a number which is larger than 2 in absolute value is obtained and print the number to the screen. ○

Exercise 3.3. In this exercise we'll use nested for-loops, that is, a loop inside a loop (though one could also solve the exercise without the use of loops).

Create a list of length 10 such that the x'th element of the list is a vector containing the divisors of x. ○

Exercise 3.4. This exercise is an illustration of the fact that `R` by default *drops* dimensions of an matrix (or more generally of an array) if possible. If the programmer is not aware of this behaviour, it might lead to some confusion. Run the following code which throws an error at some point.

```
set.seed(10)
M <- matrix(seq(16), nrow = 4)

for (i in 1:10){
  nr <- sample(1:ncol(M), 1)
  tmp <- M[,1:nr]
  print(i) # print the iteration number
  print(tmp) # print the subsetted matrix
  colSums(tmp) # attempt to find the column sums
}
```

Fix this broken code.

○

4 *Functions*

Exercise 4.1. Write an `R` function that takes a single vector as its argument and calculates the difference between the largest and smallest elements of the vector. ○

Exercise 4.2. `R` has many built-in functions to generate random variates from different distributions (such as `runif`, `rnorm`, etc.) with support in $\mathbb{R}$. Write a function that returns n random variates from a mixture distribution of two normal random variables with means μ_1 and μ_2 and standard deviation σ_1 and σ_2 for some value of the mixture parameter p. Use e.g. `hist` to examine the output of your function for various combinations of input values. ○

Exercise 4.3. The family of `apply` functions is very useful. The basic idea is that we have some functions that we wish to apply repeatedly e.g. to each element of a vector or list. Answer the below questions using functions `apply`, `lapply`, `sapply`, etc.

(a) Implement a function that can return the row sums of a `matrix` (not using the `rowSums` function, obviously).

(b) Implement a function that takes a list of square matrices as input and outputs a list of eigenvalues.

○

Exercise 4.4. This exercise is on *vectorization*. Many functions and operations in `R` are *vectorized* in the sense that we can apply them to a vector and then they operate on each entry (or possibly on pairs of entries, in the case of multiple arguments).

(a) Determine if the operators `&` and `&&` are vectorized.

Define the following function.

```
f <- function(n){
  sample(letters, n)
}
```

A vectorized version of `f` should be able to take a vector consisting of m integers as input and output a list of length m, each element being a sample of letters of appropriate size.

(a) Argue that `f` is not vectorized and use the function `Vectorize` to obtain a vectorized version.

○

Exercise 4.5. In R, one can have a function call itself.[2] Somewhat mind-boggling at first, it can actually be useful.

Using recursion, write a *factorial function*, that is, a function that returns $n!$ for n being a natural number or zero. ○

[2] This is called *recursion* in computer science.

5 *Control flow*

Exercise 5.1. In this exercise, you'll be working with `if` and `else` statements.

The following command gives you the number of files in your current working directory.

```
length(list.files())
```

Write a piece of code that prints the names of all the files in your current working directory if there is less than 10, and otherwise prints the number of files. ○

Exercise 5.2. Write an `R` function that takes a single number as its argument and returns `"a"` if the number is strictly positive, and otherwise `"b"`. What does your function do if you input a character string instead of a number? ○

Exercise 5.3. There's a subtle difference between

```
if (a) {0} else {1}
```

and

```
ifelse(a,0,1)
```

that has to do with vectorization (see Exercise 4.4). Figure out which is vectorized and which is not. ○

Exercise 5.4. In this exercise, we'll combine several control-flow constructs. Generate a vector $x = (x_1, x_2, \ldots, x_{100})^t$ of independent standard normal random variates. Using a for-loop, determine the smallest i (if any such i exists) such that $\sum_{j=1}^{i} x_j$ exceeds 10 in absolute value, and then break the loop and return i (if any such i exists). ○

6 *Documentation and help files,* `data.frames`

Exercise 6.1. When doing statistics in R, we often use a `data.frame` to hold the data set. A common task is then to extract parts of the `data.frame` or in other ways reference variables therein.

Clear your work space and generate the following small data set.

```
rm(list = ls()) # this deletes all objects in your work space!
mydata <- data.frame(from = c("A", "B", "A", "A"),
                     to = c("A", "A", "B", "B"),
                     amount = c(10, 12, NA, 13))
```

(a) Use the functions `str` and `complete.cases` to familiarize yourself with data.

(b) Find the subset of data for which the variable `to` is equal to `"B"`.

To extract a variable from the data frame, we can use the following command.

```
mydata$from

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

In this way, you give R both the name of the variable and where to look for it, that is in `mydata`. We can avoid stating the latter if we `attach` the data frame. Understanding exactly what this function does is not that simple and requires some knowledge of the inner workings of R. This exercise only aims at developing our understanding of the consequences of calling this function.

(c) Check if there's in object `from` in your work space. Attach `mydata` using the `attach` function and check again.

(d) Now assign `rep("A", 4)` to `from`. Does this change `mydata$from`?

(e) Remove `from` from your work space. Attach `mydata` again. Now assign `rep("B", 4)` to `mydata$from`. What do you get if you call `from` now?

(f) What can be learned about the use of `attach` from this exercise?

○

Exercise 6.2. Write an R function that takes a data frame as its single argument. The function should count the number of observations that are not complete (i.e. have `NA` entries), and return this number, the row indices of the non-complete observations, and the subset of the data which is complete. ○

Exercise 6.3. Loosely speaking, a `data.frame` is a `list` in the shape of a `matrix`. For instance, when doing subsetting of `data.frame` we can often use the techniques we've learned for matrices and lists. We generate the `data.frame` from Exercise 6.1 again.

```
rm(list = ls()) # this deletes all objects in your work space!
mydata <- data.frame(from = c("A", "B", "A", "A"),
                     to = c("A", "A", "B", "B"),
                     amount = c(10, 12, NA, 13))
```

(a) Extract the 1st and 2nd variables using both `matrix` and `list` subsetting.

At other times, it's important that we actually consider a `data.frame` as a `list` as illustrated below.

(b) Apply the `typeof` function to each of the variables, using both `apply` and `sapply` functions. Which one returns the correct result, and why does one of the ways not work properly?

○

Exercise 6.4. Write a function that can take a contingency table (an array with names) and output a data frame that holds the same information. Then write an inverse function. Or find functions that can do this. ○

7 *Simulation*

Exercise 7.1. Take a look at the simulation in F7-8sim.R. Let's generalize this simulation by introducing a new parameter, m, the number of dice rolled. This parameter is hard-coded to be 5 in F7-8sim.R. Write some code to be able to make the analogous simulation for other values of m (the dice should be rolled until all m show the same number of eyes). ○

Exercise 7.2. In this exercise, we will consider the Monty Hall problem. We will not be concerned with mathematical arguments but instead approximate a certain probability using simulation.

The Monty Hall problem is quite famous. Look online for both academic and non-academic discussions of this problem.

Imagine a game show where the contestant is placed in front of three doors. Behind one door is a car, behind each of the other doors a goat. The contestant will rather win a car than a goat. The contestant picks a door in two steps: first the contestant picks some door, A. The game master knows where the car is placed and then opens a door behind which a goat is standing, but not door A. Note that this is always possible. Then the game master offers the contestant a choice: to either change to the other closed door, or stay with the original pick. The contestant wins whatever is behind the door he/she is standing at after this choice.

We will consider two strategies. Using strategy α the contestant always stays at the door he/she originally chose. Using strategy β, the contestant always changes door, when one door has been opened.

Simulate the probability of winning a car using strategies α and β. ○

Exercise 7.3. Let's simulate how a contageous disease spreads through an office building. Say, we have n_0 employees, located in m_0 rooms, at time 0. At each time point, 1,2,... each employee moves randomly to another room inside the building.[3] At time 0, only a single employee has the disease. If a healthy employee is in the same room as any sick employee there's a probability p that the disease will be transmitted to the healthy employee. We assume that no employee ever leaves the building. Let N be the first time point where all employees are infected. Simulate the distribution of N given input parameters n_0, m_0, and p. ○

[3] For simplicity, assume that the distribution of the next room does not depend on the current room and is identical for all employees.

8 Programming style

Exercise 8.1. Find your solution to Exercise 7.3. Restructure and comment your code to make it easily readable. ○

9 S3 Objects

Exercise 9.1. Find your solution from Exercise 7.3. Write a function that returns an S3 object of `class` `contagionSim` which is the result of a single simulation including information at all time points. Write a `plot.contagionSim` function (a `plot` method for the new class). Consider what would be a nice way of visualizing such a simulation. ○

10 *Environments*

Exercise 10.1. Run and read the following code. Explain how this illustrates that (loosely speaking) the parent environment of a function is determined by where it was defined, not where it's called. Figure out what the new assignment operator in the last line does.

```
rm(list = ls())
f <- function(x){
  x + y
}
y <- 1
f(2)

## [1] 3

sapply(seq(3), f)

## [1] 2 3 4

sapply(seq(3), function(x) {y <- 2; f(x)})

## [1] 2 3 4

sapply(seq(3), function(x) {y <<- 2; f(x)})

## [1] 3 4 5
```

○

Exercise 10.2. Run and read the following code. Explain what's happening. ○

```
rm(list = ls()); myenv <- new.env(); is.environment(myenv)

## [1] TRUE

y <- 1; myenv$y <- 2

f <- function(x) x + y
f(1); environment(f)

## [1] 2
## <environment: R_GlobalEnv>

environment(f) <- myenv
environment(f); f(1)

## <environment: 0x000001791738ed00>
## [1] 3
```

11 *knitr and Shiny*

Take a look at e.g. http://kbroman.org/knitr_knutshell/ if you need some help for getting started with `knitr`.

Exercise 11.1. Compile a .pdf from the knitrExample.Rnw document using `knitr` and work with the questions in the document. You'll need to install the `knitr` package and change some options in RStudio. ○

Exercise 11.2. Take some exercise that you've solved, write a solution in a .Rnw document and compile a .pdf. ○

Exercise 11.3. In RStudio, open a new .Rmd document and compile it to a .html file. ○

Exercise 11.4. Take some exercise that you've solved, write a solution in a .Rmd document and compile a .html file. ○

Exercise 11.5. Take a look at the Shiny app in the files ui.R and server.R and run the app. This can be done by putting the two files in your working directory and use `runApp` in the Shiny package. One can also use app.R to have every in a single file, and again use `runApp`.

Try to understand how the app works. Extend the app by allowing the generalization in Exercise 7.1. ○

Exercise 11.6. Run the following code which adds a `diamonds` data set to your work space.

```
library(ggplot2)
data("diamonds")
```

Visualize the data set using a Shiny app in which the user can change various parameters. ○

12 *Puzzles*

Exercise 12.1. In how many ways can you place eight queens on a chess board such that no queen can take any other queen in one move? Note that the purpose of the exercise is to have R make the bulk of the work, not for you to look for a pen-and-paper solution. Note that we assume that the queens are interchangeable, i.e. we do not distinguish between the queens. ○

This exercise requires a little knowledge of chess rules. If need be, find the rules of chess somewhere online.

13 Hints

Hints to 1.1. Take a look at functions `rep` and `seq`. Recall that many operations are vectorized in `R` and also that `R` repeats vectors if they're too short to make sense of. ○

Hints to 1.3. Use logical subsetting. That is, find a vector such that you can determine if each element of that vector fits the criterion. ○

Hints to 1.5. Recall that `R` has the built-in vector `letters` which can save you some typing. Look into functions `paste` and `rev`. ○

Hints to 2.2. To determine a hierarchy, try experimenting by putting data of two data types into e.g. a matrix (or just a vector). Data of one of the types will be turned into the other type. This latter type is higher in the hierarchy. If you don't see any options for storing multiple data types in a single `R object`, you'll learn about some in subsequent sections. ○

Hints to 2.3. Note that `upper.tri` and `lower.tri` return matrices of Boolean values which can be used e.g. for changing only some entries of a matrix. The final submatrix can be obtained by `M2[A]` for an appropriate matrix A of Boolean values. ○

Hints to 3.1. The build-in vector `letters` is useful for this exercise. ○

Hints to 3.2. This can be done using a `while` loop and the `rnorm` function. ○

Hints to 3.3. Use a double-loop. Let the outer loop iterate over `x` in 1,2,...,10, and let the inner loop iterate over the numbers 1,2,...,`x`. Use `%%` to find out if some number in 1,2,...,`x` is a divisor of `x`. ○

Hints to 3.4. Take a look at the `drop` argument of the `"["` function (run e.g. `?"["` in the console). ○

Hints to 4.1. The functions `max` and `min` can be useful for solving this exercise. ○

Hints to 4.3. The function `eigen` can be used to find the eigen values of a square `matrix`. ○

Hints to 4.4. To argue that `f` is not vectorized look at what happens when you input vectors of Boolean values with length strictly larger than 1. ○

Hints to 4.5. The main idea of the recursion is that $n! = n \cdot (n-1)!$ for $n > 0$. Note that you'll probably also need the `if () {} else {}` construct. ○

Hints to 6.2. The function `complete.cases` is useful for this exercise.○

Hints to 12.1. The eight queens can be placed on the board in $64 \cdot 63 \cdot \ldots \cdot 57$ different ways and that's probably to many to have your computer check. To reduce this number, note that if eight queens are placed such that no one can take another one in a single move we could replace these queens with rooks and in this case the rooks would not be able to take each other in one move either. This means that we only have to search among those arrangements where the eight rooks can't take each other. There are 8! such arrangements. ○