Module 3 (R basics part III)

Contents

HUMAN CAPITAL
NATIONAL COHESION STRATEGY

EUROPEAN UNION
EUROPEAN
SOCIAL FUND

Project co-financed by the European Union under the European Social Fund

1 Attributes

1.1 Introduction

Almost all[1] R objects may be equipped with various **attributes**. Attributes are kind of ***metadata***, i.e. additional information on objects. Setting some attributes may have a significant impact on the way an R object interacts with R functions.

1.2 Getting and setting attributes

The `attr()` function may be used to set an object's attribute. An attribute is a key-value pair. Except for some special attributes (see below), we are free to associate any metadata with an R object.

```
x <- (-5):5
attr(x, "color") <- "green"
attr(x, "which_positive") <- which(x > 0)
attr(x, "favorite_fun") <- exp
```

The above is equivalent to:

```
x <- structure((-5):5, color="green",
   which_positive=which(x > 0), favorite_fun=exp)
```

The `attr()` function may also be used to get an object's attribute.

```
attr(x, "which_positive")
## [1]  7  8  9 10 11
attr(x, "favorite")  # autocompletion
## function (x)  .Primitive("exp")
attr(x, "no_such_attribute")
## NULL
```

Note how an R object equipped with such metadata is printed:

```
x
##  [1] -5 -4 -3 -2 -1  0  1  2  3  4  5
## attr(,"color")
## [1] "green"
## attr(,"which_positive")
## [1]  7  8  9 10 11
## attr(,"favorite_fun")
## function (x)  .Primitive("exp")
str(x)
##  atomic [1:11] -5 -4 -3 -2 -1 0 1 2 3 4 ...
##  - attr(*, "color")= chr "green"
##  - attr(*, "which_positive")= int [1:5] 7 8 9 10 11
##  - attr(*, "favorite_fun")=function (x)
```

What is most important, `x` is still an "ordinary" numeric vector.

```
mode(x)
## [1] "numeric"
x[1]
## [1] -5
mean(x)
## [1] 0
x[attr(x, "which_positive")]
## [1] 1 2 3 4 5
```

In other words, we may do anything with `x` as with any other numeric vector.

To remove an attribute, "set" its value to `NULL`.

[1] All except for `NULL`.

```
attr(x, "favorite_fun") <- NULL
x
##  [1] -5 -4 -3 -2 -1  0  1  2  3  4  5
## attr(,"color")
## [1] "green"
## attr(,"which_positive")
## [1]  7  8  9 10 11
```

On the other hand, to fetch all attributes, use the `attributes()` function.

```
attributes(x)  # returns a (named-see below) list
## $color
## [1] "green"
##
## $which_positive
## [1]  7  8  9 10 11
```

Some R functions set attributes in order to provide their users with additional information on resulting objects:

```
x <- c(1, 2, NA, 4, NA)
na.omit(x)
## [1] 1 2 4
## attr(,"na.action")
## [1] 3 5
## attr(,"class")
## [1] "omit"
```

Here, `na.action` gives the positions at which `NA`s were found. Such an information is not useful for most users – they are free to ignore it.

1.2.1 Special attributes

There are attributes that have a **special meaning**. Their values must meet some strict constraints, see Tab. 1. What is important, special attributes may be accessed via `comment()`, `class()`, etc. functions. These functions may sometimes return a sensible value even if `attr()` has not been called explicitly.

Table 1: Special R attributes

attribute	meaning
`comment`	ignored by `print()`; Value: character vector
`class`	an object's S3 class; Value: character vector
`names`	a vector's elements' names; Value: character vector

Also: `dim`, `dimnames`, `row.names` – see *Compound types*.

1.2.2 The comment attribute

The `comment` attribute is probably the least interesting one.

```
x <- 1:5
comment(x) <- "What a nice object!"
x  # comment printing is suppressed
## [1] 1 2 3 4 5
attr(x, "comment")
## [1] "What a nice object!"
```

```
comment(x)
## [1] "What a nice object!"
comment(x) <- 10
## Error:  attempt to set invalid 'comment' attribute
```

Note the usage of `comment()` and the fact that only character vectors constitute valid comments.

1.2.3 The names attribute

The `names` attribute may be fed with a character vector. It is used to label a vector's elements.

```
x <- seq(0, 1, length.out = 5)
names(x) <- c("1st", "2nd", "3rd", "4th", "5th")
x
##  1st  2nd  3rd  4th  5th
## 0.00 0.25 0.50 0.75 1.00
```

Other example:

```
structure(list(1:10, mean), names = c("vector", "function"))
## $vector
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $`function`
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x3007df8>
## <environment: namespace:base>
```

Note the following:

```
structure(1:4, names = c("a", "b", "c", "d", "e"))
## Error:  'names' attribute [5] must be the same length as the vector [4]
(x <- structure(1:4, names = c("a", "b", "c")))
##    a    b    c <NA>
##    1    2    3    4
names(x)
## [1] "a" "b" "c" NA
x <- structure(1:4, names = c("a", "b", "c", "d"))
unname(x)  # x <- unname(x) is equivalent to attr(x,'names')<-NULL
## [1] 1 2 3 4
names(x)[2] <- "zzz"
x
##   a zzz   c   d
##   1   2   3   4
```

The `c()` and `list()` functions may set the `names` attribute automatically:

```
list(first=1:10, second=100:110)
## $first
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $second
##  [1] 100 101 102 103 104 105 106 107 108 109 110
c("1st"=1, "2nd"=2) # 1st = invalid syntactic name = use quotes
## 1st 2nd
##   1   2
```

The `names` attribute affects not only the `print()` function. The "`[`" and "`[[`" operators also may take it into account.

```
x <- structure(1:4, names = c("a", "b", "c", "d"))
x["a"]
```

```
## a
## 1
x[c("a", "d", "a")]
## a d a
## 1 4 1
```

Moreover:

```
y <- list(a = 1, b = 2)
y["a"]  # subset
## $a
## [1] 1
y[["a"]]  # extract
## [1] 1
```

A simple database:

```
(numberOfParticipants <- c(male=20, female=23))
##   male female
##     20     23
# a new male participant joined the experiment:
numberOfParticipants["male"] <- numberOfParticipants["male"]+1
numberOfParticipants
##   male female
##     21     23
```

Note that `names` are not identifiers: They might be non-unique.

```
x <- c(one = 1, two = 2, one = 3)
x["one"]  # first occurrence returned
## one
##   1
x[names(x) == "one"]
## one one
##   1   3
```

Searching for a given label is $O(n)$, pessimistically.

```
x <- structure(as.list(1:1000000), names=as.character(1:1000000))
microbenchmark(x[[10000]], x[[100000]], x[[1000000]],
   unit="ms") # index - O(1)
## Unit: milliseconds
##        expr      min       lq    median        uq      max neval
##  x[[10000]] 0.000226 0.000235 0.0002940 0.0004185 0.001988   100
##  x[[1e+05]] 0.000225 0.000235 0.0003095 0.0004435 0.021301   100
##  x[[1e+06]] 0.000225 0.000235 0.0002370 0.0003650 0.044168   100
microbenchmark(x[["10000"]], x[["100000"]], x[["1000000"]],
   unit="ms") # name - O(n)
## Unit: milliseconds
##            expr     min      lq  median      uq     max neval
##    x[["10000"]]  0.2093  0.2906  0.2983  0.3214  0.5109   100
##   x[["100000"]]  3.4481  3.6339  3.7777  4.3198  6.9472   100
##  x[["1000000"]] 31.8469 34.6002 35.7525 36.7708 47.5799   100
```

If `x` is a list, then `x$label` is most often equivalent to `x[["label"]]`.

```
x <- list(one = 1, `2nd` = 2)
x$one
## [1] 1
x$three <- 3  # adjust length
str(x)
## List of 3
##  $ one  : num 1
##  $ 2nd  : num 2
```

```
##  $ three: num 3
x$"2nd"  # 2nd - not a syntactic name - use quotes
## [1] 2
```

By the way, "$" implements *partial matching* of labels, but its usage is not recommended.

Exercise: *Write a function* `textHist()` *which takes a numeric vector* `x` *as an argument.*

The function should use the result of a call to `hist(x, plot=FALSE)` *to "plot" a histogram of* `x` *on the console. You may assume that there are no more than 30 observations in each bin. For example, this is the desired result of* `textHist(c(1, 1.2, 1.3, 1.6, 2.1, 2.3, 2.6))`*:*

```
## *** 1.0-1.5
## *   1.5-2.0
## **  2.0-2.5
## *   2.5-3.0
```

1.2.4 The class attribute

The `class` attribute denotes an object's so-called **S3 class**. The S3-style object-oriented programming will be covered in-depth later on. However, the concept is so important that we should at least sketch some basic S3 concepts now.

Firstly, note that `class()` returns a sensible value even though the attribute is not set explicitly.

```
x <- c("a", "b", "c")
class(x)
## [1] "character"
attr(x, "class")
## NULL
```

By default, `class(x) == mode(x)`, except for `typeof(x) == "integer"`.

Exercise: *Write a function* `theSameClass()`*, which checks if all the elements of a given list have the same* `class` *attribute. If so, return a string with the* `class` *name. Otherwise return* `FALSE`*.*

A function's behavior may change drastically depending on its argument's `class` attribute.

```
print
## function (x, ...)
## UseMethod("print")
## <bytecode: 0x2758508>
## <environment: namespace:base>
mean
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x3402980>
## <environment: namespace:base>
```

Each function that makes a call to `UseMethod()` is called a **generic function**. Each generic function dispatches the control flow to another function, called **method**.

Let `f()` be a generic function. Assume that we are calling it on an object of class `classname`.

1. If there exists a function named `f.classname()`, this is the routine to be evaluated on a given object.
2. Otherwise, `f.default()` will be called.

Compare, for example, `?plot` with `?plot.default`.

```
print.Pretty <- function(x, ...) {
    cat("PRETTY", paste(x, collapse = ", "), ":-) \n")
}

x <- 1:5
x
```

```
## [1] 1 2 3 4 5
class(x) <- "Pretty"
x
## PRETTY 1, 2, 3, 4, 5 :-)
print.default(x)
## [1] 1 2 3 4 5
## attr(,"class")
## [1] "Pretty"
```

Example: A hypothesis test.

```
test <- shapiro.test(rnorm(100))
test
##
##  Shapiro-Wilk normality test
##
## data:  rnorm(100)
## W = 0.9897, p-value = 0.6375
```

What is that?

```
class(test)
## [1] "htest"
```

And more precisely…?

```
typeof(test)
## [1] "list"
```

Thus, an object of class `htest` is an ordinary R list.

```
str(unclass(test))
## List of 4
##  $ statistic: Named num 0.99
##   ..- attr(*, "names")= chr "W"
##  $ p.value  : num 0.637
##  $ method   : chr "Shapiro-Wilk normality test"
##  $ data.name: chr "rnorm(100)"
test$p.value
## [1] 0.6374777
test[["method"]]
## [1] "Shapiro-Wilk normality test"
```

The only reason why `htest` is presented in a non-standard way is due to the overloaded `print()` method.

```
print.htest  # inaccessible directly
## Error:  object 'print.htest' not found
getS3method("print", "htest")  # here it is
```

```
## function (x, digits = 4L, quote = TRUE, prefix = "", ...)
##  ...
```

Exercise: *Write a function* `Range()`*, which for a given numeric vector* `x` *returns a named list with the following components:*

- `x` *– a copy of the input vector,*
- `min` *– the minimum of* `x`*,*
- `max` *– the maximum.*

The return value should have the `class` *attribute set to* `"Range"`.

Then write a method `print.Range()`*, which echoes an object of class* `"Range"` *in a form resembling:*

```
## x   = 1, 3, 4, 5, 2
## min = 1
## max = 5
```

1.3 What attributes are preserved by base R functions?

Indexing operator – `?"["`: *Subsetting (except by an empty index) will drop all attributes except* `names`, `dim` *and* `dimnames`.

```
structure(1:5, names = letters[1:5], attrib = "val")[2]
## b
## 2
```

Binary operators – `?"+"`: *Most attributes are taken from the longer argument.* `Names` *will be copied from the first if it is the same length as the answer, otherwise from the second if that is. If the arguments are the same length, attributes will be copied from both, with those of the first argument taking precedence when the same attribute is present in both arguments.*

```
structure(1:5, a1 = "v1") * structure(1, a2 = "v2")
## [1] 1 2 3 4 5
## attr(,"a1")
## [1] "v1"
```

Vectorized math functions – should generally preserve all the attributes.

```
log(structure(1:10, attrib = "val"))
##  [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595 1.9459101
##  [8] 2.0794415 2.1972246 2.3025851
## attr(,"attrib")
## [1] "val"
```

Aggregation functions – generally drop all the attributes.

```
mean(1:5, class = "xx", names = letters[1:5], attrib = "val")
## [1] 3
```

1.4 Summary

Attributes are a way to associate some *metadata* with R objects. Setting some attributes (like `class` or `names`) may have a significant impact on the way an R object interacts with R functions.

1.5 Bibliography

- R Core Team, *An introduction to R*, 2014, Sec. 3
- R Core Team, *R language definition*, 2014, Sec. 2.2
- R Core Team, *Writing R extensions*, 2014, Sec. 3, 4
- Gagolewski M., *Programowanie w jezyku R*, PWN, 2014 (in Polish), Chap. 7

2 Compound types

2.1 Introduction

Let us examine the 3 most commonly used R compound types:

- Factors
- Matrices (and their generalization, Arrays)
- Data frames

Each of the above are in fact "only slightly extended" versions of base types, like atomic vectors and lists.

2.2 Factors

Factors are vector-like objects storing qualitative data. The predefined set of categories (*levels*) should be relatively small. We may say that factors are *enumerated types*.

```
factor(c("male", "female", "female", "male", "female"))
## [1] male   female female male   female
## Levels: female male
str(factor(c(1, 3, 1, 2, 5, 2)))
##  Factor w/ 4 levels "1","2","3","5": 1 3 1 2 4 2
```

How are factors represented in R?

```
f <- factor(c(1, 3, 1, 2, 5, 2))
class(f)
## [1] "factor"
typeof(f)
## [1] "integer"
```

We see that an object of class `factor` is in fact an integer vector. Moreover:

```
f <- factor(c(1, 3, 1, 2, 5, 2))
unclass(f)
## [1] 1 3 1 2 4 2
## attr(,"levels")
## [1] "1" "2" "3" "5"
```

The `levels` attribute is a character vector of all levels' labels. The numeric values (consecutive natural numbers), on the other hand, represent the categories' identifiers.

```
attr(f, "levels")[as.integer(f)]
## [1] "1" "3" "1" "2" "5" "2"
as.character(f)
## [1] "1" "3" "1" "2" "5" "2"
as.integer(as.character(f))  # not the same as as.integer(f)
## [1] 1 3 1 2 5 2
```

In fact, factors may be created manually:

```
test <- c(1L, 4L, 3L, 2L, 1L, 3L)
levels(test) <- c("one", "two", "three", "four")
class(test) <- "factor"
test
## [1] one   four  three two   one   three
## Levels: one two three four
```

Note that `levels(·)` is the same as `attr(·, "levels")`.

Note that many methods have been overloaded for factors. In particular, R's authors did not want factors to be confused with "standard" vectors:

```
f <- factor(c(1, 3, 1, 2, 5, 2))
is.factor(f)
## [1] TRUE
is.vector(f)
## [1] FALSE
is.atomic(f)
## [1] TRUE
is.integer(f)
## [1] FALSE
is.character(f)
## [1] FALSE
```

We may define any strict linear order on a factor's levels.

```
f <- factor(c("one", "three", "two", "one"),
   levels=c("one", "two", "three"), ordered=TRUE)
sort(f)
## [1] one   one   two   three
## Levels: one < two < three
which(f > "one")
## [1] 2 3
f[f > "two"] # not a lexicographic order here
## [1] three
## Levels: one < two < three
max(f)
## [1] three
## Levels: one < two < three
```

Creating contingency tables:

```
table(factor(c("one", "three", "two", "one")))
##
##   one three   two
##     2     1     1
table(factor(c("male", "female", "male", "male", "female")),
      factor(c("low",  "low",    "high", "high", "high")))
##
##          high low
##   female    1   1
##   male      2   1
```

Dropping unused levels:

```
(f <- factor(c(1, 2, 3, 5, 1), levels = 1:5))
## [1] 1 2 3 5 1
## Levels: 1 2 3 4 5
nlevels(f)  # length(levels(f))
## [1] 5
(f <- droplevels(f))
## [1] 1 2 3 5 1
## Levels: 1 2 3 5
nlevels(f)
## [1] 4
```

By the way, it is easy to change the labels in one call:

```
levels(f) <- c("one", "two", "three", "five")
f
## [1] one   two   three five  one
## Levels: one two three five
```

Such an operation is not easy in case of ordinary character vectors.

Splitting another vector w.r.t. a qualitative variable:

```
height <- c(164, 182, 173, 194, 159)
gender <- c("m", "f", "m", "m", "f")
split(height, gender)
## $f
## [1] 182 159
##
## $m
## [1] 164 173 194
lapply(split(height, gender), mean)  # avg height in each group
## $f
## [1] 170.5
##
```

```
## $m
## [1] 177
```

A somewhat similar operation:

```
height <- c(164, 182, 173, 194, 159)
gender <- c("m", "f", "m", "m", "f")
tapply(height, gender, mean)
##     f     m
## 170.5 177.0
```

"Discretizing" numeric data:

```
(x <- round(rnorm(10), 1))
##  [1] -0.6 -0.2  1.6  0.1  0.1  1.7  0.5 -1.3 -0.7 -0.4
cut(x, c(-Inf, -1, 0, 1, Inf))
##  [1] (-1,0]    (-1,0]    (1, Inf]  (0,1]     (0,1]     (1, Inf]  (0,1]
##  [8] (-Inf,-1] (-1,0]    (-1,0]
## Levels: (-Inf,-1] (-1,0] (0,1] (1, Inf]
cut(x, c(-Inf, -1, 0, 1, Inf),
   labels=c("very_small", "small", "large", "very_large"))
##  [1] small      small      very_large large      large      very_large
##  [7] large      very_small small      small
## Levels: very_small small large very_large
```

Exercise: *Write a function* `chisq.gof.test()` *to calculate the* χ^2 *goodness-of-fit test.*

Parameters:

- *a numeric vector of observations,* `x`*;*
- *a cumulative distribution function* `F`*, i.e. a nondecreasing real function such that* $\lim_{y\to-\infty} \mathtt{F}(y, \ldots) = 0$ *and* $\lim_{y\to\infty} \mathtt{F}(y, \ldots) = 1$*;*
- *an increasingly sorted numeric vector* `g` *of length* k*;*
- *„...” parameter, giving additional arguments to* `F`*;*
- *a positive real value of* `df`*, giving the test's so-called number of degrees of freedom, which defaults to* k*.*

The algorithm:

1. `g` *defines* $k+1$ *disjoint intervals* $P_1, \ldots, P_{k+1}$*,* $P_i = (g_{i-1}, g_i]$*, with convention that* $g_0 = -\infty$ *and* $g_{k+1} = \infty$*,* $i = 1, \ldots, k+1$*.*
2. *Create a vector* $(o_1, \ldots, o_{k+1})$*, where* o_i *denotes the number of observations in* P_i*,* $i = 1, \ldots, k+1$*.*
3. *Create a vector* $(e_1, \ldots, e_{k+1})$*, where* e_i *denotes the expected probability of observing a value in* P_i*,* $i = 1, \ldots, k+1$*. We of course have* $e_i = \mathtt{F}(g_i, \ldots) - \mathtt{F}(g_{i-1}, \ldots)$*.*
4. *Calculate the test statistic* $T = \sum_{i=1}^{k+1} (o_i - ne_i)^2/(ne_i)$*, where* n *is the length of* `x`*.*
5. *Calculate the p-value,* $1 - \mathtt{pchisq}(T, \mathtt{df})$*.*

The function should return an object of class `htest`*, i.e. a list with the following named components:*

- `statistic` *– test statistic,*
- `parameter` *– degrees of freedom,* `df`*,*
- `p.value` *– p-value,*
- `method` *– a string,* `"Chi-square goodness of fit test"`*,*
- `data.name` *– a string,* `deparse(substitute(x))`*.*

An exemplary call – testing for $N(1, 10)$*:*

```
set.seed(123)  # reproducible results
data <- rnorm(100, 1, 10)
chisq.gof.test(data, pnorm, c(-4, 1, 5), 1, 10)
##
##  Chi-square goodness of fit test
##
## data:  data
## T = 1.9688, df = 3, p-value = 0.5789
```

2.3 Matrices and Arrays

Matrices are built upon vectors (atomic ones or lists).

```
x <- 1:6
dim(x) <- c(2, 3)  # or attr(x, 'dim') <- c(2, 3)
x
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
class(x)  # vector + dim attr => implicit matrix class
## [1] "matrix"
is.matrix(x)
## [1] TRUE
is.numeric(x)
## [1] TRUE
```

Note that a matrix's elements are stored in a column-wise order.

Once again: a matrix is nothing more than a vector equipped with the `dim` attribute.

```
x <- 1:6
dim(x) <- c(2, 3)  # or attr(x, 'dim') <- c(2, 3)
x
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
dim(x) <- c(3, 2)
x
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
as.numeric(x)  # drops the dim attribute
## [1] 1 2 3 4 5 6
```

Moreover:

```
x <- matrix(1:6, nrow=2, ncol=3) # == structure(1:6, dim=c(2, 3))
x^2
##      [,1] [,2] [,3]
## [1,]    1    9   25
## [2,]    4   16   36
x*c(-1, 2)
##      [,1] [,2] [,3]
## [1,]   -1   -3   -5
## [2,]    4    8   12
x*x
##      [,1] [,2] [,3]
## [1,]    1    9   25
## [2,]    4   16   36
```

Note that all the above are vectorized *elementwise* ops.

Here is a character matrix:

```
matrix(letters[1:4], ncol=2) # nrow auto-guessed
##      [,1] [,2]
## [1,] "a"  "c"
## [2,] "b"  "d"
```

And some list-based matrices:

```
structure(list(mean, sd, var, median), dim = c(2, 2))
##      [,1] [,2]
## [1,] ?    ?
## [2,] ?    ?
structure(list(1:5, c(1, 5.4)), dim = c(1, 2))
##      [,1]      [,2]
## [1,] Integer,5 Numeric,2
```

Note some printing issues.

An array is a generalization of a matrix:

```
structure(1:12, dim = c(2, 3, 2))
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]    7    9   11
## [2,]    8   10   12
```

See also: `?array`.

The `dimnames` attribute may be set in order to label row/column names (cf. the `names` attrib for atomic vectors). Generally, it is a list of `dim(·)` character vectors, of lengths `dim(·)[1]`, `dim(·)[2]`, ..., respectively.

```
(x <- matrix(1:6, nrow=2,
   dimnames=list(c("r1", "r2"), c("c1", "c2", "c3"))))
##    c1 c2 c3
## r1  1  3  5
## r2  2  4  6
dim(x)
## [1] 2 3
str(dimnames(x))
## List of 2
##  $ : chr [1:2] "r1" "r2"
##  $ : chr [1:3] "c1" "c2" "c3"
```

"Multidimensional" indices are available for "[" if `dim` is set:

```
(x <- matrix(letters[1:6], nrow = 2))
##      [,1] [,2] [,3]
## [1,] "a"  "c"  "e"
## [2,] "b"  "d"  "f"
x[1, 2]  # 1st row, 2nd column
## [1] "c"
x[1, ]  # 1st row
## [1] "a" "c" "e"
x[1:2, 2:3]  # 1st and 2nd row, 2nd and 3rd column [select block]
```

```
##      [,1] [,2]
## [1,] "c"  "e"
## [2,] "d"  "f"
```

Here is a `dimnames`-based subsetting:

```
(x <- matrix(letters[1:6], nrow=2,
   dimnames=list(c("r1", "r2"), c("c1", "c2", "c3"))))
##    c1  c2  c3
## r1 "a" "c" "e"
## r2 "b" "d" "f"
x[c("r2", "r1"), c("c3", "c1")]
##    c3  c1
## r2 "f" "b"
## r1 "e" "a"
```

A matrix can also be subsetted with a 2-column numeric matrix:

```
(x <- matrix(letters[1:6], nrow=2))
##      [,1] [,2] [,3]
## [1,] "a"  "c"  "e"
## [2,] "b"  "d"  "f"
(y <- matrix(c(1, 3,   2, 1,    1, 1), byrow=TRUE, ncol=2))
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    1
## [3,]    1    1
x[y]
## [1] "e" "b" "a"
```

The "*" operator does an element-wise multiplication. Matrix multiplication is available via "%*%":

```
(a <- matrix(1:4, nrow = 2))
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
(b <- matrix(c(1, -1, -1, 1), nrow = 2))
##      [,1] [,2]
## [1,]    1   -1
## [2,]   -1    1
a %*% b  # compare with a*b
##      [,1] [,2]
## [1,]   -2    2
## [2,]   -2    2
```

"Expanding" matrices:

```
(x <- matrix(1:6, nrow = 2))
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
cbind(x, 1:2)
##      [,1] [,2] [,3] [,4]
## [1,]    1    3    5    1
## [2,]    2    4    6    2
rbind(x, 1:3)
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
## [3,]    1    2    3
rbind(1:5, 11:15)
##      [,1] [,2] [,3] [,4] [,5]
```

```
## [1,]    1    2    3    4    5
## [2,]   11   12   13   14   15
```

`simplify2array()` tries to "simplify" a given list:

```
simplify2array(list(1, 2, 3))  # result = atomic vector
## [1] 1 2 3
simplify2array(list(1:2, 3:4, 5:6))  # result = matrix
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
simplify2array(list(1, 2:3))  # cannot simplify
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2 3
```

Thus, `sapply(·)` is a convenient substitute for `simplify2array(lapply(·))`:

```
sapply(list(1:10, 11:20, 21:30), mean)
## [1]  5.5 15.5 25.5
sapply(list(1:10, 11:20, 21:30), range)
##      [,1] [,2] [,3]
## [1,]    1   11   21
## [2,]   10   20   30
```

On the other hand, `apply()` applies a given operation on each row/column:

```
(x <- matrix(1:6, nrow = 2))
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
apply(x, 1, sum)  # each row
## [1]  9 12
apply(x, 2, mean)  # each column
## [1] 1.5 3.5 5.5
```

See also: `?rowSums`, `?colSums`, `?rowMeans`, `?colMeans`.

The `outer()` function applies a given binary operation on each pair of elements in two given vectors:

```
outer(c(TRUE, FALSE, NA), c(TRUE, FALSE, NA), "|")
##      [,1]  [,2] [,3]
## [1,] TRUE  TRUE TRUE
## [2,] TRUE FALSE   NA
## [3,] TRUE    NA   NA
outer(c("a", "b"), 1:3, paste, sep = "")
##      [,1] [,2] [,3]
## [1,] "a1" "a2" "a3"
## [2,] "b1" "b2" "b3"
```

Exercise: *We are given a $n \times m$ matrix $P \geq 0$ such that $\sum_{i=1}^{n}\sum_{j=1}^{m} p_{i,j} = 1$ and two increasingly sorted numeric vectors* `x` *(of size n) and* `y` *(of size m). The triple $(\mathtt{x}, \mathtt{y}, P)$ represents a joint distribution of a pair of R.V.s (X, Y).*

For example: The joint probability distribution describing the probability of obtaining each possible combination of final marks (2=fail, 5=very good) for R and Philosophy courses in some college are as follows:

		P			
		2	3	4	5
R	2	0	0,01	0,1	0,2
	3	0,01	0,05	0,03	0,1
	4	0,1	0,03	0,05	0,01
	5	0,2	0,1	0,01	0

X and Y are independent iff for all i, j it holds $p_{i,j} = (\sum_{k=1}^{n} p_{k,j})(\sum_{l=1}^{m} p_{i,l})$. Write a function `indep()` *to determine if $(\mathtt{x}, \mathtt{y}, P)$ fulfills this criterion.*

Write a function `basicchar()` *which, given $(\mathtt{x}, \mathtt{y}, P)$, returns a named numeric vector representing basic characteristics of (X, Y):*

- *Expected values:* $\mathbb{E}\,X = \sum_{i=1}^{n} x_i \sum_{j=1}^{m} p_{i,j}$, $\mathbb{E}\,Y = \sum_{j=1}^{m} y_j \sum_{i=1}^{n} p_{i,j}$,
- *Variances:* $\operatorname{Var} X = \mathbb{E}\,X^2 - (\mathbb{E}\,X)^2$, *where* $\mathbb{E}\,X^2 = \sum_{i=1}^{n} x_i^2 \sum_{j=1}^{m} p_{i,j}$ *and* $\operatorname{Var} Y = \mathbb{E}\,Y^2 - (\mathbb{E}\,Y)^2$, *where* $\mathbb{E}\,Y^2 = \sum_{j=1}^{m} y_j^2 \sum_{i=1}^{n} p_{i,j}$,
- *Covariance:* $\operatorname{Cov}(X, Y) = \mathbb{E}\,(XY) - \mathbb{E}\,X\,\mathbb{E}\,Y$ *for* $\mathbb{E}\,(XY) = \sum_{i=1}^{n} \sum_{j=1}^{m} x_i y_j p_{i,j}$,
- *Coefficient of correlation:* $\varrho(X, Y) = \operatorname{Cov}(X, Y)/\sqrt{\operatorname{Var} X \operatorname{Var} Y}$.

For more information on matrix operations, read more: `?t`, `?diag`, `?upper.tri`, `?lower.tri`, `?isSymmetric`, `?maxCol`, `?aperm`, `?norm`, `?dist`, `?det`, `?eigen`, `?qr`, `?svd`, `?chol`, `?kappa`, `?solve`, `?lsfit`. See also: the Matrix package (e.g. sparse and band matrices, etc.) and igraph (graphs), relations (binary relations), ...

Exercise: *Write a function* `diagprod()` *to calculate the diagonal product of a given square matrix A.*
In order to have more fun, don't use the `diag()` *function.*

Exercise: *A finite graph is a pair $G = (V, E)$, where $V = \{v_1, \ldots, v_n\}$ (vertex set) and $E \subseteq V \times V$ (edge set, i.e. a binary relation on V).*

Each graph may be represented via a 0-1 square matrix K, where $k_{i,j} = 1$ iff $(v_i, v_j) \in E$, $i, j \in \{1, \ldots, n\}$.

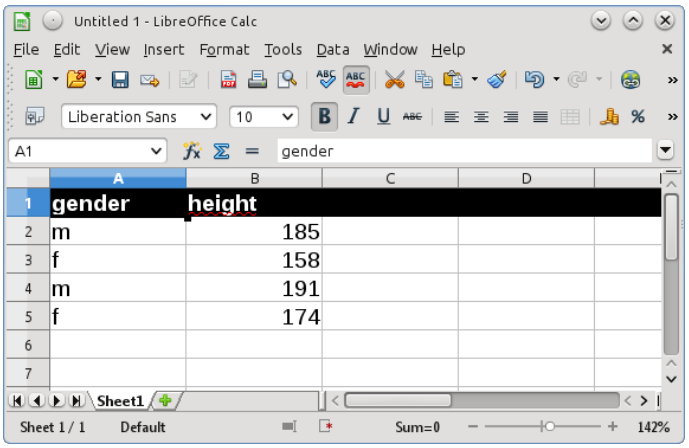
A simple (strict) graph is an undirected (i.e. $(v_i, v_j) \in E \Rightarrow (v_j, v_i) \in E$) graph containing no loops (i.e. $(v_i, v_i) \notin E$).

Write a function `simplegraph()`*, which checks if a 0-1 square matrix represents a simple graph.*

2.4 Data frames

A **data frame** is a list of vectors, each having the same length. In the practice of data analysis it is very often used to represent *tabular data*.

```
data.frame(gender=c("m", "f", "m", "f"),
   height=c(185, 158, 191, 174))
##   gender height
## 1      m    185
## 2      f    158
## 3      m    191
## 4      f    174
```



Some basic type information:

```
df <- data.frame(gender=c("m", "f", "m", "f"),
   height=c(185, 158, 191, 174))
class(df)
## [1] "data.frame"
typeof(df)
## [1] "list"
is.list(df)
## [1] TRUE
is.data.frame(df)
## [1] TRUE
str(unclass(df))
## List of 2
##  $ gender: Factor w/ 2 levels "f","m": 2 1 2 1
##  $ height: num [1:4] 185 158 191 174
##  - attr(*, "row.names")= int [1:4] 1 2 3 4
```

Here is how to create a data frame manually:

```
df <- list(c("m", "f", "m", "f"),
           c(185, 158, 191, 174))
names(df) <- c("gender", "height")
attr(df, "row.names") <- c("1", "2", "3", "4")
class(df) <- "data.frame"
df
##   gender height
## 1      m    185
## 2      f    158
## 3      m    191
## 4      f    174
```

This enables us to create very fancy data frames:

```
(x <- structure(list(
   list(matrix(1:4, ncol=2),
        matrix(11:14, ncol=2)
      )
   ),
   class="data.frame",
   names="c1",
   row.names=c("r1", "r2")
))
##                c1
## r1     1, 2, 3, 4
## r2 11, 12, 13, 14
x[["c1"]][[1]]
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

Let us consider the following data set:

```
(survey <- data.frame(
   gender = c("m", "m", "f", "m", "f", "f"),
   coffee = c(FALSE, TRUE, TRUE, FALSE, TRUE, FALSE),
   time   = c(23, 25, 31, 46, 24, 38),
   weight = c(69, 71, 58, 98, 63, 41)
))
##   gender coffee time weight
## 1      m  FALSE   23     69
## 2      m   TRUE   25     71
## 3      f   TRUE   31     58
```

```
## 4       m  FALSE   46      98
## 5       f   TRUE   24      63
## 6       f  FALSE   38      41
```

Each data frame is a named list: here is how we may *extract* the last column:

```
survey$weight
## [1] 69 71 58 98 63 41
survey[["weight"]]
## [1] 69 71 58 98 63 41
survey[[4]]
## [1] 69 71 58 98 63 41
```

On the other hand, here is how we may *subset* a data frame:

```
survey[c(1, 3)]
##   gender time
## 1       m   23
## 2       m   25
## 3       f   31
## 4       m   46
## 5       f   24
## 6       f   38
```

We can do that even though the `dim` attribute is not set explicitly:

```
attr(survey, "dim")
## NULL
```

The `dim()` function returns a sensible value:

```
dim(survey)
## [1] 6 4
```

Thus, a data frame may sometimes imitate a matrix's behavior. In particular, a two-dimensional indexing operator is available:

```
survey[1, ]  # 1st row
##   gender coffee time weight
## 1       m  FALSE   23     69
survey[, 4]  # == survey[[4]] != survey[4]
## [1] 69 71 58 98 63 41
survey[1:2, 3:4]
##   time weight
## 1   23     69
## 2   25     71
```

In fact, a data frame may be easily converted to a matrix.

```
as.matrix(survey)
##      gender coffee  time weight
## [1,] "m"    "FALSE" "23" "69"
## [2,] "m"    " TRUE" "25" "71"
## [3,] "f"    " TRUE" "31" "58"
## [4,] "m"    "FALSE" "46" "98"
## [5,] "f"    " TRUE" "24" "63"
## [6,] "f"    "FALSE" "38" "41"
```

Note that coercion occurred.

By the way, when a data frame is created, character data are automatically converted to factors:

```
class(survey$gender)
## [1] "factor"
```

See, however, the `stringsAsFactors` argument of `data.frame()` and `stringsAsFactors` option (`?options`).

All row names should be unique:

```
row.names(survey) <- c("Frank", "Avishai", "Ella", "John", "Ella", "Elis")
## Warning:  non-unique value when setting 'row.names':  'Ella'
## Error:  duplicate 'row.names' are not allowed
row.names(survey) <- c("Frank", "Avishai", "Ella", "John", "Nina", "Elis")
```

Thus, row names serve as row identifiers:

```
survey["Ella", ]
##      gender coffee time weight
## Ella      f   TRUE   31     58
```

It is recommended that column `names` are syntactically valid names, see `?make.names`:

```
test <- data.frame(good.name=1:2, "1bad.name"=3:4)
test # auto-fix
##   good.name X1bad.name
## 1         1          3
## 2         2          4
test$good.name
## [1] 1 2
names(test)[2] <- "1bad.name"
test$"1bad.name" # test$1bad.name won't work
## [1] 3 4
```

Filtering:

```
survey[survey$gender == "m" & survey$weight > 70, ]
##         gender coffee time weight
## Avishai      m   TRUE   25     71
## John         m  FALSE   46     98
subset(survey, gender == "m" & weight > 70)
##         gender coffee time weight
## Avishai      m   TRUE   25     71
## John         m  FALSE   46     98
subset(survey, gender == "f", select = -weight)
##      gender coffee time
## Ella      f   TRUE   31
## Nina      f   TRUE   24
## Elis      f  FALSE   38
```

See also: `?with`, `?within`, `?transform`.

"Expanding" data frames:

```
cbind(survey, height = c(184, 159, 173, 162, 195, 178))
##         gender coffee time weight height
## Frank        m  FALSE   23     69    184
## Avishai      m   TRUE   25     71    159
## Ella         f   TRUE   31     58    173
## John         m  FALSE   46     98    162
## Nina         f   TRUE   24     63    195
## Elis         f  FALSE   38     41    178
```

Exercise: *The following table represents some basic physical characteristics of the Hundred Acre Wood dwellers.*

weight [kg]	87	64	62	50	64	83	62	84	66	64
height [cm]	148	162	160	162	170	172	169	162	162	159

Create a data frame with the above data. Add a column with the BMI (Body Mass Index) values. Then, add a column with the BMI categories (a factor variable): underweight (BMI $<$ 18,5), normal (18,5 $\leq$ BMI $<$ 25), overweight (25 $\leq$ BMI $<$ 30), obese (BMI $\geq$ 30).

"Expanding" data frames:

```
rbind(survey, data.frame(gender = "m", coffee = TRUE, time = 41, weight = 98,
    row.names = "Steinar"))
##         gender coffee time weight
## Frank        m  FALSE   23     69
## Avishai      m   TRUE   25     71
## Ella         f   TRUE   31     58
## John         m  FALSE   46     98
## Nina         f   TRUE   24     63
## Elis         f  FALSE   38     41
## Steinar      m   TRUE   41     98
```

Applying various operations:

```
sapply(survey, class)  # class of each column (it's a list...)
##    gender    coffee      time    weight
##  "factor" "logical" "numeric" "numeric"
tapply(survey$time, survey$gender, mean)  # avg time / each gender
##        f        m
## 31.00000 31.33333
```

See also: `?aggregate`, `?by`, `?ave`.

Exercise: *Calculate basic descriptive statistics for some of the variables in the* `nlschools` *dataset (package* MASS*).*

The `order()` function uses a stable sorting algorithm. Thus, it is possible to order rows of a data frame w.r.t. multiple criteria.

```
survey[order(survey$gender, survey$time), ]
##         gender coffee time weight
## Nina         f   TRUE   24     63
## Ella         f   TRUE   31     58
## Elis         f  FALSE   38     41
## Frank        m  FALSE   23     69
## Avishai      m   TRUE   25     71
## John         m  FALSE   46     98
```

Note that within each gender, the observations are sorted w.r.t. time. If the algorithm was not stable, sorting w.r.t. gender could break the established relative order w.r.t. time.

Exercise: *Play with the* `wine` *data set (package* gamair*, call* `data(wine)` *to make it available).*

Here is a list of "hot" data frame processing-related packages:

- data.table
- reshape2
- plyr2 and dplyr
- magrittr

I recommended that you take a look at their features.

2.5 Time series

Time series are objects of class `ts`. Try to find out yourself how are they represented.

Exercise: *Write a function* `movingavg()` *to calculate a* `k`*-moving average of a given time series* `x` *of size* n*,* $k < n$*,* k*-odd. A* `k`*-moving average is a time series* $(w_1, \ldots, w_{n-k+1})$ *such that* $w_i = \sum_{j=1}^{k} x_{i+j-1}/k$*. Test your function on the built-in* `UKgas` *object (see e.g.* `plot(UKgas)`*).*

2.6 Summary

Compound types are extensions of basic types. A matrix is based on an ordinary vector. A factor is "something between" a character and an integer vector. A data frame is a special kind of a list.

2.7 Bibliography

- R Core Team, *An introduction to R*, 2014, Sec. 4, 5, 6.3
- Gagolewski M., *Programowanie w jezyku R*, PWN, 2014 (in Polish), Chap. 8
- Matloff N., *The art of R programming*, No Starch Press, 2011, Chap. 3, 5, 6

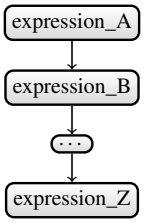
3 Controlling program flow

3.1 Introduction

Up to now, our functions followed the following scheme:

```
f <- function(...) {
    expression_A
    expression_B
    # ...
    expression_Z
}
```

Here is the corresponding program control flow diagram:



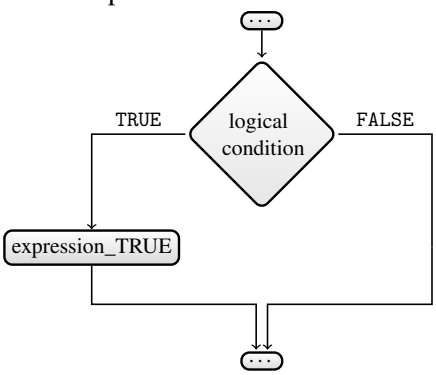
In this case, each expression is evaluated exactly once. As we may sometimes need a different control flow scheme, in this section we will discuss the R control-flow expressions:

- Conditional execution: `if`
- Repetitive execution: `for`, `while`, `repeat`

3.2 Conditional execution

3.2.1 if..else: Syntax

Here is the control flow diagram for the `if` expression:



Syntax:

```
if (logical_condition) expression_TRUE
```

`logical_condition` is a single R expression such that `as.logical(logical_condition)` $\in$ {`TRUE`, `FALSE`} (a single logical value, not `NA`).

Note that `expression_TRUE` will be evaluated if and only if `as.logical(logical_condition)` gives `TRUE`. `expression_TRUE` is a single R expression. If you want to evaluate more expressions conditionally, group them with "`{...}`".

An example:

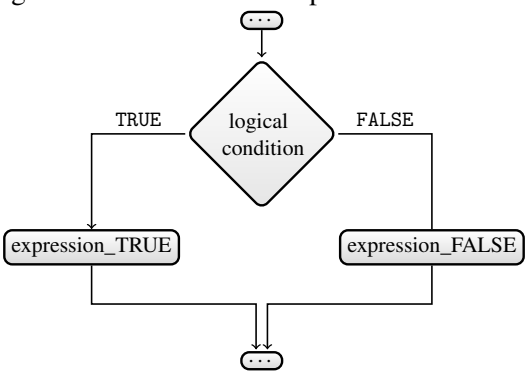
```
sum2 <- function(x) {
   if (!is.numeric(x)) # gives either TRUE or FALSE
      x <- as.numeric(x) # will throw an error if it's impossible
   sum(x) # return value
}

sum2(1:4) # already numeric
## [1] 10
sum2(c("1", "2", "3")) # coercion
## [1] 6
sum2(mean)
## Error:  cannot coerce type 'closure' to vector of type 'double'
```

Other examples:

```
if (TRUE) cat("!")
## !
if (FALSE) cat("!")  # nothing done
if (NA) cat("!")
## Error:  missing value where TRUE/FALSE needed
if (5) cat("!")  # coercion
## !
if (c(TRUE, FALSE)) cat("!")  # warning != error
## Warning:  the condition has length > 1 and only the first element will be used
## !
if (c(FALSE, NA, TRUE)) cat("!")  # warning != error
## Warning:  the condition has length > 1 and only the first element will be used
```

And here is a control flow diagram of the `if..else` expression:



Syntax:

```
if (logical_condition) expression_TRUE else expression_FALSE
```

`expression_TRUE` will be evaluated if and only if `as.logical(logical_condition)` gives `TRUE`. Otherwise, `expression_FALSE` is executed.

Example – nested `if`s:

```
sgn <- function(x) {
   stopifnot(is.numeric(x), length(x) == 1, is.finite(x))
```

```
   if (x > 0)
      cat("positive\n")
   else { # if not positive, then either negative or zero
      if (x < 0)
         cat("negative\n")
      else
         cat("zero\n")
   }
}

sgn(5)
## positive
sgn(0)
## zero
```

The above is of course equivalent to:

```
sgn <- function(x) {
   stopifnot(is.numeric(x), length(x) == 1, is.finite(x))
   if (x > 0)
      cat("positive\n")
   else if (x < 0)
      cat("negative\n")
   else
      cat("zero\n")
}

sgn(5)
## positive
sgn(0)
## zero
```

Note that the R parser in some cases will not interpret the following as we wish to:

```
if (TRUE) print(TRUE)
else print(FALSE)
```

In the R console:

```
> if (TRUE) print(TRUE)
[1] TRUE
> else print(FALSE)
Error: unexpected 'else' in "else"
```

This is because the parser cannot predict whether `else` follows after `if`. Possible fixups:

```
if (TRUE) print(TRUE) else print(FALSE) # one-liner
## [1] TRUE
```

or:

```
{ # as a grouped expression (e.g. within a function)
   if (TRUE) print(TRUE)
   else print(FALSE)
}
## [1] TRUE
```

or:

```
if (TRUE) {
   print(TRUE)
} else { # no newline before `else`
   print(FALSE)
}
## [1] TRUE
```

3.2.2 if..else: Return value

`if..else` is just a syntactic sugar, equivalent to calling some R function.

```
if (TRUE) print(TRUE) else print(FALSE)
## [1] TRUE
"if"(TRUE, print(TRUE), print(FALSE))
## [1] TRUE
```

Each R function returns some value. What does `if..else` result in? It turns out that the return value of the `if..else` expression is determined by either `expression_TRUE` or `expression_FALSE`.

```
x <- if (runif(1) > 0.5) "head" else "tail"
print(x)
## [1] "tail"
```

Moreover,

```
if (logical_condition) expression_TRUE
```

is equivalent to

```
if (logical_condition) expression_TRUE else invisible(NULL)
```

An example:

```
sgn <- function(x) {
   stopifnot(is.numeric(x), length(x) == 1)
   if (x > 0) "positive"
   else if (x < 0) "negative"
   else "zero"
   # here, retval of if == retval of the function
}

unlist(lapply(c(1, 0, -2, 4), sgn))
## [1] "positive" "zero"     "negative" "positive"
```

3.2.3 Specifying the logical condition

Recall that `&` and `|` are element-wise vector operations denoting logical conjunction and alternative, respectively. The `&&` and `||` operators may be applied only to vectors of length one. They only evaluate their second argument if necessary, e.g. `TRUE || whatever`, `FALSE && whatever`.

```
FALSE || {cat("!"); TRUE}
## !
## [1] TRUE
TRUE  || {cat("!"); TRUE}
## [1] TRUE
```

Also, when specifying the `logical_condition`, `all()` and `any()` are your friends. For example, `stopifnot(cond)` is equivalent to:

```
if (any(is.na(cond)) || !all(cond)) stop("error message")
```

3.2.4 return()

`return()` may be used only within a function. It stops the function's evaluation immediately and returns a given value. A note to C/C++ programmers: `return()` is a function; parentheses are required.

An example: quick sort (Hoare, 1960). It is a **recursive** sorting algorithm.

- A sequence of length 1 is already sorted.
- In order to sort a sequence `x` of length > 1, we:
  - Pick a *pivot* element v from `x`.

– Return a sequence consisting of
sorted elements $< v$, elements $= v$, **sorted** elements $> v$.

By the way, `options("expressions")` determines the maximal number of nested fun calls. Exceeding this limit results in `Error: evaluation nested too deeply: infinite recursion`.

```
qs <- function(x) {
   stopifnot(is.atomic(x), is.vector(x))
   if (length(x) <= 1) # already sorted
      return(x)
   pivot <- sample(x, 1) # random element of x
   c(qs(x[x<pivot]), x[x==pivot], qs(x[x>pivot]))
}

qs(c(5, 1, 4, 2, 3))
## [1] 1 2 3 4 5
qs(c("a", "e", "d", "c", "b"))
## [1] "a" "b" "c" "d" "e"
```

3.2.5 ifelse()

The `ifelse()` function is a vectorized version of `if..else`. We can use it as follows:

```
ifelse(test, values_TRUE, values_FALSE)
```

Some examples:

```
x <- c(5, 3, 1, 2, 4)
ifelse(x < 3, -x, x^2)
## [1] 25  9 -1 -2 16
ifelse(x > 3, NA, x)
## [1] NA  3  1  2 NA
```

Missing values in `test` give missing values in the result:

```
x <- c(-1, 0, NA, 1)
ifelse(x < 0, -x, x)  # NAs handled correctly
## [1]  1  0 NA  1
```

Possible pitfalls:

```
x <- c(-1, 0, 1, 2)
ifelse(x >= 0, sqrt(x), NA)  # sqrt is evaluated on -1 anyway
## Warning:  NaNs produced
## [1]       NA 0.000000 1.000000 1.414214
```

3.3 Repetitive execution

R loop expressions allow for repetitive execution of expressions, possibly on different input data. There are three loop expressions in R:

- `while`
- `repeat`
- `for`

Each R loop returns `invisible(NULL)`.

3.3.1 while

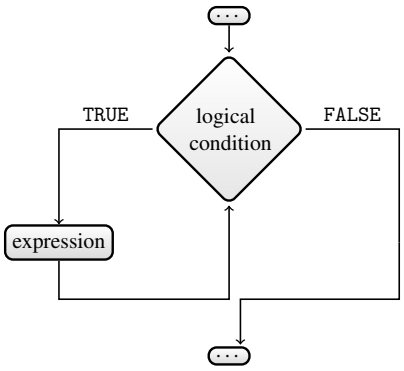
The `while` loop – syntax:

```
while (logical_condition) expression
```

The `while` loop evaluates an `expression` as long as `as.logical(logical_condition)` is `TRUE`. In

order to avoid an infinite loop, the `logical_condition` should e.g. depend on a variable that is modified by the `expression`.

Here is the corresponding program control flow diagram:



For example, this is a possible implementation of `sum()`:

```
sum_while <- function(x) {
   x <- as.numeric(x) # coercion not possible => error
   result <- 0
   i <- 1
   n <- length(x)
   while (i <= n) { # logical_condition depends on i
      result <- result + x[i]
      i <- i + 1 # i changes here
   }
   result # return value
}

sum_while(1:5)
## [1] 15
sum_while(c(1, 2, NA, 4, 5))
## [1] NA
```

3.3.2 break and next

The `break` expression immediately breaks out an (innermost) R loop.

```
i <- 0
while (TRUE) { # infinite loop?
   j <- 0
   while (TRUE) {
      j <- j+1
      if (j > i) break
      print(c(i, j))
   }
   i <- i+1
   if (i > 3) break
}
## [1] 1 1
## [1] 2 1
## [1] 2 2
## [1] 3 1
## [1] 3 2
## [1] 3 3
```

The `next` expression immediately advances to the next loop iteration.

```
i <- 0
while (i < 6) {
   i <- i+1
   if (i %% 2 == 0) next
   print(i)
}
## [1] 1
## [1] 3
## [1] 5
```

Of course the two above examples could be rewritten in a much simpler way.

Exercise: *Let $f : [a, b] \to \mathbb{R}$ be a continuous function. It may be shown that if $f(a)f(b) < 0$, then there exists $c \in [a, b]$ such that $f(c) = 0$.*

Write a function `bisection()` *with the following parameters:*

- *a real function* `f()`,
- *a numeric value* `a`,
- *a numeric value* `b` $>$ `a`,
- *a positive real* `eps` *(defaults to 10^{-16}),*
- *a positive integer* `maxiter` *(defaults to 100).*

Exemplary call: `bisection(function(x) x^2-1, -0.5, 7.81)`.

The function looks for the root of `f()` *by using the bisection algorithm. For $i = 1, 2, \ldots,$* `maxiter`*:*

1. *Let $x_i := (a + b)/2$.*
2. *If $|f(x_i)| <$* `eps`*, then you're done.*
3. *If $f(a)(x_i) \geq 0$, then let $a := x_i$, or $b := x_i$ otherwise.*

If the method doesn't converge, generate a warning.

The function should return the following named list (cf. `?uniroot`*):*

1. `root` – *approximate root's location,*
2. `f.root` – *value of* `f()` *at the above point,*
3. `iter` – *number of iterations considered,*
4. `estim.prec` – *approximation error (half of the length of the remaining $[a, b]$).*

3.3.3 repeat

`repeat` – syntax:

```
repeat expression
```

is equivalent to

```
while (TRUE) expression
```

Thus, we need an explicit call to e.g. `break` or `return()` in the `expression`.

3.3.4 for

`for` – syntax:

```
for (name in vector) expression
```

This is roughly equivalent to:

1. `name <- vector[[1]]; expression`
2. `name <- vector[[2]]; expression`
3. ...
4. `name <- vector[[length(vector)]]; expression`

Here is our 2nd implementation of `sum()`:

```
sum_for1 <- function(x) {
   x <- as.numeric(x) # coercion not possible => error
   result <- 0
   for (elem in x)
      result <- result+elem
   result # return value
}

sum_for1(1:5)
## [1] 15
sum_for1(c(1, 2, NA, 4, 5))
## [1] NA
```

And our 3rd implementation of `sum()`:

```
sum_for2 <- function(x) {
   x <- as.numeric(x) # coercion not possible => error
   result <- 0
   for (i in seq_along(x))
      result <- result+x[i]
   result # return value
}

sum_for2(1:5)
## [1] 15
```

See `?seq_along`. Why would `1:length(x)` be wrong here? Try calling `sum_for2(numeric(0))`.

Exercise: *We are given a lower triangular $n \times n$ matrix A. Each element $a_{i,j}$, $i \geq j$ is a positive integer representing the number of bonbons that Alicia can gather. Her "sweet hunt" starts at $a_{1,1}$. From each element $a_{i,j}$ she may either go down (to $a_{i+1,j}$) or down-right (to $a_{i+1,j+1}$, of course if it's possible). Your task is to determine the maximal number of bonbons that can be gathered in this game.*

Solution 1: a greedy algorithm.
This gives a sub-optimal result = 21 ($4+7+4+6$).

```
4
7 5
3 4 6
9 6 2 3
```

Solution 2: a dynamic programming algorithm (optimal). Use an auxiliary lower triangular $n \times n$ matrix B. $b_{i,j} > 0$ gives info on the number of sweets which may be gathered by taking the optimal subpath starting from $a_{i,j}$. We have $b_{n,j} = a_{n,j}$ and $b_{i,j} = a_{i,j} + \max\{b_{i+1,j}, b_{i+1,j+1}\}$, $i < n$.

Exercise: *Write your own implementation of the „%*%" operator.*
Given a $n \times p$ matrix `A` *and a $p \times m$ matrix* `B`*, return a $n \times m$ matrix C such that: $c_{i,j} = \sum_{k=1}^{p} a_{i,k}\, b_{k,j}$ for $i = 1, \ldots, n$ and $j = 1, \ldots, m$.*

Our implementation of `lapply()`:

```
lapply_for <- function(x, f, ...) {
    stopifnot(is.vector(x))
    f <- match.fun(f)  # see ?match.fun
    result <- vector("list", length(x))  # preallocate
    for (i in seq_along(x)) result[[i]] <- f(x[[i]], ...)
    result
}

lapply_for(list(1:5, 2:6), "*", 2)
## [[1]]
```

```
## [1]  2  4  6  8 10
##
## [[2]]
## [1]  4  6  8 10 12
```

Exercise: *Write a function* `split1()` *which splits a given numeric vector* `x` *by using the following scheme (the result should be stored as a list consisting of numeric vectors).*

Let $[a, b)$, $a, b \in \mathbb{Z}$ *be the smallest interval such that* $(\forall i)\ x_i \in [a, b)$. *The* j*-th output list's element,* $j = 1, \ldots, b - a$, *consists of all the values* x_i *such that* $x_i \in [a + j - 1, a + j)$.

Implement at least 2 different algorithms.

3.4 Summary

R loops are **very slow**:

```
x <- runif(100000)
microbenchmark::microbenchmark(unit="relative",
   sum_while(x), sum_for1(x), sum_for2(x), sum(x))
## Unit: relative
##          expr  min  lq median  uq max neval
##  sum_while(x) 1029 789    768 734 627   100
##   sum_for1(x)  243 186    183 160 122   100
##   sum_for2(x)  422 328    322 285 266   100
##        sum(x)    1   1      1   1   1   100
```

We should rather rely on vectorized R functions. But it does not mean that we must always avoid loops at any cost – some tasks cannot be implemented without them.

3.5 Bibliography

- R Core Team, *An introduction to R*, 2014, Sec. 9
- R Core Team, *Writing R extensions*, 2014, Sec. 3, 4
- Gagolewski M., *Programowanie w jezyku R*, PWN, 2014 (in Polish), Chap. 6

4 Run-time measurement and estimation

4.1 Introduction

What is a **high-quality code**? It surely must do exactly what it is supposed to (e.g. it's compliant with a technical specification). Moreover it must be:

- **readable** – the reader is able to understand the coder's intentions,
- **testable** – organized in such a way that unit testing is possible,
- **maintainable** – corrections and enhancements are relatively easy to make,
- **portable** – gives the same results on different platforms.

High-quality code is also **economic**: is is characterized with high speed and low memory consumption.

4.2 Run-time measurement

Let us play with the following implementations of the same algorithm:

```
ex1 <- sum

ex2 <- function(x) {
    res <- 0
```

```
    for (e in x) res <- res + e
    res
}
```

There are a few ways to measure the run-time of a code block in R. `system.time()` "starts" the timer, evaluates a chunk of code and returns the elapsed time (in secs).

```
x <- runif(10000000)
system.time(ex1(x))
##    user  system elapsed
##   0.016   0.000   0.016
system.time(ex2(x))
##    user  system elapsed
##   4.695   0.002   4.729
```

- `user` – execution time of user instructions of the calling process
- `system` – execution time of system calls on behalf of the calling process
- `elapsed` – total time elapsed

`system.time()` should rather be used in case of time-consuming tasks. It is because the timer's resolution is not very high.

`rbenchmark::benchmark()` goes a lever higher. We are able to inspect several code chunks at a time and instruct the function to consider a number of replications of our experiment.

```
x <- runif(1000)
rbenchmark::benchmark(ex1(x), ex2(x), replications = 100)
##     test replications elapsed relative user.self sys.self user.child
## 1 ex1(x)          100   0.001        1     0.001        0          0
## 2 ex2(x)          100   0.053       53     0.052        0          0
##   sys.child
## 1         0
## 2         0
```

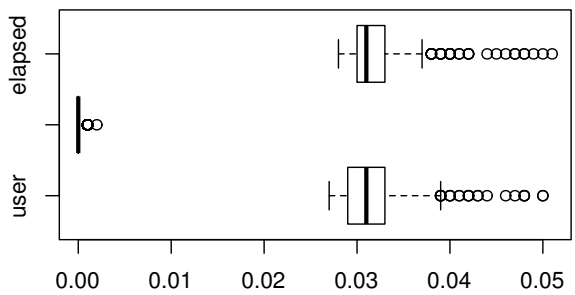
Still, it is a simple wrapper around `system.time()`.

By the way, the run-time distribution is most often right-skewed:

- `elapsed` time is affected by other background processes (e.g. an MP3 player),
- `user` time is affected by i.a. R's **garbage collector**.

This is because e.g. the garbage collector, which gets rid of no-longer-used R objects, is run occasionally (The process is automatic, we have no control over it). Read more: `?gc`, `?Memory`.

```
x <- runif(100000)
res <- replicate(1000, system.time(ex2(x)))
boxplot(list(user=res[1,], system=res[2,], elapsed=res[3,]),
   horizontal=TRUE)
```



On the other hand, `microbenchmark::microbenchmark()` relies on a much more precise time measuring method.

```
x <- runif(100)
microbenchmark::microbenchmark(ex1(x), ex2(x), times=100)
## Unit: nanoseconds
##    expr   min      lq  median      uq   max neval
##  ex1(x)   476   948.5  1348.5  2105.5  5132   100
##  ex2(x) 30103 56020.0 59565.0 66998.0 78648   100
```

```
microbenchmark::microbenchmark(1, {1}, (1), times=100000)
## Unit: nanoseconds
##       expr min  lq median  uq   max neval
##          1   8  20     25  26 10389 1e+05
##  {     1 }  88 106    118 131 36900 1e+05
##        (1)  88  95    113 118  7979 1e+05
```

Here, perhaps `min` and `median` are the most interesting.

Exercise: *Write a chunk of code that performs the following experiment. A single iteration consists in:*

1. *Generate a realization of* $(X_1, \ldots, X_{25})$ *i.i.d.* $N(\mu, \sigma)$, *with* $\mu = 0, \sigma = 1$.
2. *Calculate* $[l(\mathbf{x}), u(\mathbf{x})]$ *– a 95% confidence interval for* μ *(assume a normal model with* μ, σ *– unknown).*
3. *Return 1 if* $\mu = 0 \in [l(\mathbf{x}), u(\mathbf{x})]$ *or 0 otherwise.*

Replicate the above 100000 *times. You'll get a realization of* $(Y_1, \ldots, Y_{100000})$ *i.i.d.* $\mathrm{Bern}(p)$. *Estimate* p *by calculating* $\bar{\mathbf{y}}$ *(we have* $p = 0.95$*, at least theoretically).*

Hint: Use `t.test()`.

```
experiment <- function(m, n) {
   res <- replicate(m, {
      x <- rnorm(n, 0, 1)
      conf.int <- t.test(x, conf.level=0.95)$conf.int
      0 <= conf.int[2] && 0 >= conf.int[1]
   })
   mean(res)
}
set.seed(123) # gives reproducible results
system.time({
   p <- experiment(100000, 25)
})
##    user  system elapsed
##  25.484   0.008  25.694
print(p)
## [1] 0.95137
```

For some reasons we are dissatisfied with the current implementation's run-time. Let us use **code profiling** to detect the most time consuming code fragments.

```
{ # this chunk must be evaluated as a whole
   set.seed(123)
   Rprof(filename="Rprof.out")
   experiment(100000, 25)
   Rprof(NULL) # that's it
}
```

The `Rprof.out` file has been created. Call:

```
summaryRprof("Rprof.out")$by.total
```

to summarize the result.

```
##                  total.time total.pct self.time self.pct
## "experiment"          23.42    100.00      0.00     0.00
```

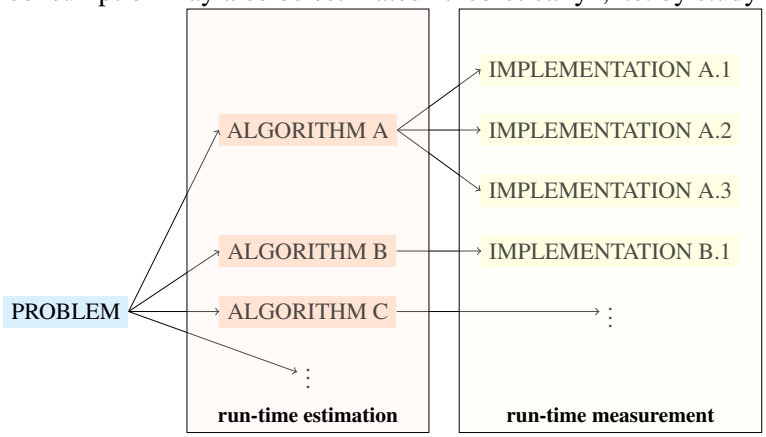
```
## "replicate"          23.42    100.00      0.00      0.00
## "sapply"             23.42    100.00      0.00      0.00
## "lapply"             23.40     99.91      0.32      1.37
## ...
## "t.test.default"     19.08     81.47      3.56     15.20
## "deparse"             5.54     23.65      1.72      7.34
## "match.arg"           4.60     19.64      0.50      2.13
## "var"                 3.88     16.57      0.50      2.13
## ...
## "stopifnot"           2.28      9.74      1.08      4.61
## ...
## "mean"                1.88      8.03      1.26      5.38
## ...
## "pt"                  0.88      3.76      0.36      1.54
## "qt"                  0.72      3.07      0.26      1.11
```

Exercise: *Reimplement the above code having in mind that:*

$$[l(\mathbf{x}), u(\mathbf{x})] = \left[\bar{\mathbf{x}} - \mathtt{qt}(0.975, n-1)\frac{\mathtt{sd}(\mathbf{x})}{\sqrt{n}}, \bar{\mathbf{x}} + \mathtt{qt}(0.975, n-1)\frac{\mathtt{sd}(\mathbf{x})}{\sqrt{n}}\right].$$

4.3 Run-time estimation

Run-time and memory consumption may also be estimated "theoretically", i.e. by studying the source code.



For a given input data set of size n units, we may estimate the number of *steps / instructions*, $f(n)$ required to compute the output. It turns out that often $f(n)$ may be nicely correlated with the real run-time. Here, a "step" may be e.g. a scalar arithmetic/logical/comparison operation, assignment, vector cell access, etc. By the way, a similar analysis may be performed w.r.t. the memory usage.

Usually, we estimate time complexity in the **asymptotic sense**, i.e. f is approximated with some simple function for arbitrary large n.

Big-O notation: $f(n) = O(g(n))$ as $n \to \infty$ iff

$$(\exists C)\ (\exists n_0)\ f(n) \le Cg(n) \quad \text{for all } n > n_0.$$

Big-omega notation: $f(n) = \Omega(g(n))$ as $n \to \infty$ iff

$$(\exists C)\ (\exists n_0)\ f(n) \ge Cg(n) \quad \text{for all } n > n_0.$$

Big-theta notation: $f(n) = \Theta(g(n))$ as $n \to \infty$ iff

$$f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n))$$

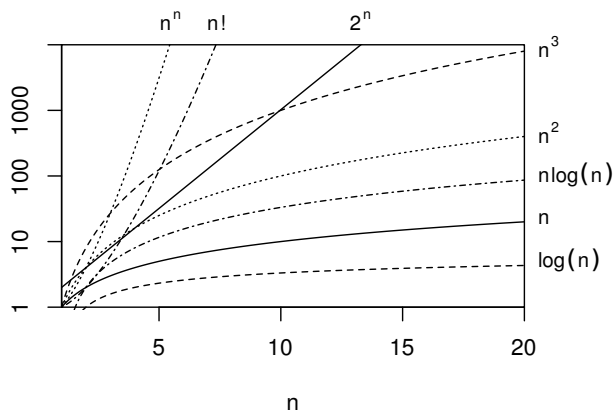


Figure 1: Functions that appear often while measuring an algorithm's time complexity

See Fig. 1 for some typical complexities.

Exercise: *You have a stream of data coming in of size n, but you don't know what n is ahead of time. (Let us suppose that you read elements one by one with a function* `nextval()`*, until you get a value of 0, e.g.* `nextval <- function() sample(0:1000, 1)`*). Calculate the sample variance in $O(1)$ space.*

Exercise: *You have a stream of data coming in of size n, but you don't know what n is ahead of time. Write an algorithm that will take a random sample of $k \leq n$ elements in $O(k)$ space for some k.*

By the way, the Big-O notation is often used quite frivolously: many people talk about an algorithm's time complexity of $O(g(n))$ having really in mind $\Theta(g(n))$, Big-O and Big-omega are most often used in the complexity theory w.r.t. problems, not algorithms.

Time complexity gives us an intuition on the **maximal data set size** we may handle in a given time unit. "Some problem is $O(g(n))$" – we can construct an algorithm to solve it with $\Theta(g(n))$ time complexity. For example, An algorithm with $O(n^2)$ time is also $O(n^3)$, $O(n^2)$, $O(2^n)$, etc. "Some problem is $\Omega(g(n))$" – we can prove that at least $g(n)$ time is needed to obtain a solution.

Does an algorithm of smaller complexity always run faster (when properly implemented) than a one of greater complexity? The answer is no. This is because:

- These notations are asymptotic (valid for large n).
- The C constant may be arbitrarily large.

For example: insertion sort is $O(n^2)$, and merge sort is $O(n \log n)$. However, often insertion $\text{sort}_{\text{time}} <$ merge $\text{sort}_{\text{time}}$ for, say, $n \leq 20$. Also such a rough estimation does not take e.g. memory management issues into account (like cache hits/misses, function call preparation time, etc.).

Here are the time complexities for performing operations on basic data structures:

	index	insert	delete	search
vector	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
linked list	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
balanced binary tree	$\Theta(n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

What about an unbalanced binary search tree? Here, insert/delete/search has:

- average time complexity: $\Theta(\log n)$
- worst-case time complexity: $\Theta(n)$

By the way, it is difficult to estimate the average time complexity: for example, what kind of distribution of data should be assumed for the results to be representative?

Here is another data structure, called a hash map:

	index	insert	delete	search
hash map	—	avg. $\Theta(1)$	avg. $\Theta(1)$	avg. $\Theta(1)$

Sorting a numeric sequence:

- insertion sort: $\Theta(n^2)$,
- merge sort: $\Theta(n \log n)$,
- quick sort: worst-case $\Theta(n^2)$, average $\Theta(n \log n)$.

On the other hand, finding a median can be done in $O(n)$. However, it takes $O(1)$ time if an input sequence is sorted. Searching for a value in a vector is basically $O(n)$, but if an input sequence is sorted, then we get $O(\log n)$ with a binary search algorithm. Thus, pre-processing data may sometimes be helpful.

Exercise: *Implement the following algorithms:*

- *insertion sort*
- *selection sort*
- *bubble sort*
- *merge sort*
- *binary search*

Exercise: *Find out (or guess) the time and memory complexity of the following R operators/functions:* `!`, `+` *(for atomic vectors),* `%*%` *(for matrices),* `eigen()`, `svd()`, `var()`, `which()`, `factor()`, `split()`, `sort()`, `rle()`, `unique()`, `match()`, `table()`, `rev()`, `quantile()`, `findInterval()`, `nchar()`, `paste()`.

Exercise: *Measure the run-time of the above functions for data sets of different sizes, e.g. 100, 1000, and 10000.*

Exercise: *Implement the above yourself (using R's control flow expressions). In which cases you can come up with more than one algorithm? Do the algorithms differ in time or memory complexity?*

What is more, some problems are **hard** – we do not know polynomial-time algorithms to solve them.

For example, in the Traveling Salesman Problem we are given a set of cities and the distances between them. What is the shortest route that visits each city exactly once and returns to the origin city? The Held–Karp algorithm has time complexity of $O(n^2 2^n)$. The only sensible solution must use some heuristics – we do not look for the optimal solution. A good one is often satisfiable.

Exercise: *Write some algorithm to find the best approximate solution to the Traveling Salesman Problem.*

4.4 Summary

A high-quality code uses as few resources as possible. Code benchmarking and profiling can help you to optimize your functions. An algorithm's complexity gives information on a function's *scalability*: e.g. how many more resources are required if size of an input dataset increases twice.

4.5 Bibliography

- R Core Team, *Writing R extensions*, 2014, Sec. 3
- Gagolewski M., *Programowanie w jezyku R*, PWN, 2014 (in Polish), Chap. 6, 7, 8
- Matloff N., *The art of R programming*, No Starch Press, 2011, Chap. 14
- Cormen T.H., Leiserson C.E., Rivest R.L., Stein C., *Introduction to Algorithms*, MIT Press and McGraw-Hill, 2009
- Knuth D.E., *The art of computer programming*, Vols. 1 and 3, Addison-Wesley, 1997

HUMAN CAPITAL NATIONAL COHESION STRATEGY
EUROPEAN UNION EUROPEAN SOCIAL FUND
Project co-financed by the European Union under the European Social Fund