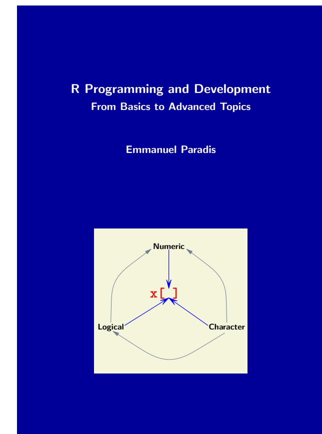
Exercices and Solutions to:

R Programming and Development
From Basics to Advanced Topics



Chapter 2

1. Give examples, relevant to your field, of quantitative and qualitative variables. Search the answer to this question to some of your colleagues who work in different fields and compare their answers with yours.

2. Create a vector with the values 1, 2, and 3. Transform this vector as a factor and store the result in a different object. Compare the characteristics of these two objects and explain the usefulness of the additional attributes attached to the factor.

   The vector and the factor are named `x` and `z`, respectively:

```
> x <- c(1, 2, 3)
> z <- factor(x)
> x
[1] 1 2 3
> z
[1] 1 2 3
Levels: 1 2 3
```

   When printed, it is clear that both objects store the same values, although we can see that `z` has additional information with it. This is clearer when printing the attributes of each object:

```
> attributes(x)
NULL
> attributes(z)
$levels
[1] "1" "2" "3"

$class
[1] "factor"
```

Thus, `x` stores only these three numerical values with no additional information; therefore, R will interpret it has a quantitative data. On the other hand, `z` has additional attributes: the class `"factor"` says that the values must be interpreted as categories, and `"levels"` gives the names (or labels) of the categories.

3. What are the mode and length of the results returned by the functions `mode` and `length`?

   We take as example the vector `x` created in the previous exercise:

```
> lx <- length(x)
> mx <- mode(x)
> mode(lx); length(lx)
[1] "numeric"
[1] 1
> mode(mx); length(mx)
[1] "character"
[1] 1
```

4. Look at the example at the beginning of Section 2.2.2: explain why the parentheses changed the result returned by the function `mode`. (Hint: you may have a look at the next chapter for help.)

   This is because the parentheses *execute* the function, so the command `environment()` returns an object of class `"environment"`, whereas the command `environment` is identical to `print(environment)`.

5. What is the difference between data read from files and data input from the keyboard?

   There is none: all data are stored in the types of objects described in this chapter, whatever the way they were input.

6. Look at the help page of the function `attributes`. Try this function on the objects created in the small examples above. Compare this function with the function `attr`.

   The help page is displayed with `?attributes`. See the previous exercise for the use of this function. `attr()` can access only a single attribute at a time, for example:

```
> attr(z, "levels")
[1] "1" "2" "3"
> attr(z, "levels") <- as.character(1:4)
> z
[1] 1 2 3
Levels: 1 2 3 4
> table(z)
z
1 2 3 4
1 1 1 0
```

7. Explain, as simply as possible, the difference between these two operators in R: `[[` and `$`.

Both operators can extract (or modify if followed by `<-`) an element of a list specified by its name. With the first operator, the name must be quoted but not with the second one:

```
> L <- list(value = pi)
> L[["value"]]
[1] 3.141593
> L$value
[1] 3.141593
```

With the first operator, the name must be given exactly, whereas the second one accepts a partial match:

```
> L[["va"]]
NULL
> L$va
[1] 3.141593
```

However, if there is an ambiguity (i.e., several names starting with "va" in this case), the result is always `NULL`.

If the name includes spaces, the use of the first operator is obviously unchanged, whereas it is necessary to wrap the name within backticks with the second one:

```
> L <- list(pi)
> names(L) <- "value of pi"
> L[["value of pi"]]
[1] 3.141593
> L$value of pi
Error: unexpected symbol in "L$value of"
> L$`value of pi`
[1] 3.141593
```

The `[[` operator has two uses that cannot be done with `$`: the element of the list can be specified by its position (an out-of-range value gives an error):

```
> L[[1]]
[1] 3.141593
> L[[2]]
Error in L[[2]] : subscript out of bounds
```

And, the argument can be an object storing the name or the position of the element:

```
> v <- "value of pi"
> L[[v]]
[1] 3.141593
> i <- 1
```

```
> L[[i]]
[1] 3.141593
```

This last feature makes this operator more practical in programs or scripts than the dollar.

8. Explain, using your knowledge (not only about R), why single characters are not commonly used as data in R.

   In most computing languages, a character string is coded as a set of characters which can be accessed or modified individually. For instance, in C (see Chap. 8) a string can be created with:

```
char x[] = "R Programming and Development";
```

   The individual characters are with `x[0]` (returning `'R'`), `x[1]` (returning `' '`), `x[2]` (returning `'P'`), and so on. Thus, the R way of creating a string with:

```
> x <- "R Programming and Development"
```

   where `x[1]` returns the whole string, or a vector such as:

```
> x <- c("R Programming and Development",
         "From Basics to Advanced Topics")
```

   is not intuitive to computer programmers. However, in statistical data analyses single characters are rarely data themselves. Instead, linguists are more likely to be interested in frequencies of words (or groups of) in texts.

9. Create a matrix, say `X`, with three rows, three columns, and nine values of your choice so that they are all distinct. Execute the command `X[9]` and explain its result. What other command could give the same result?

```
> X <- matrix(1:9, 3, 3)
> X[9]
[1] 9
```

   Matrices in R are stored as vectors, so there are indeed nine elements in `X`. Since the above indexing operation does not use the comma, standard vector indexing is assumed. The same value can be found by using matrix indexing which includes a comma:

```
> X[3, 3]
[1] 9
```

10. Create a list with the command `L <- list(a = NULL)`. Compare the outputs from the commands `L$a` and `L$b`. Do the same with the commands `L[["a"]]` and `L[["b"]]`.

```
> L <- list(a = NULL)
> L$a
```

```
NULL
> L$b
NULL
> L[["a"]]
NULL
> L[["b"]]
NULL
```

There is no difference between both series of outputs.[1] However, when using list indexing, the names (if existing) are used in the output:

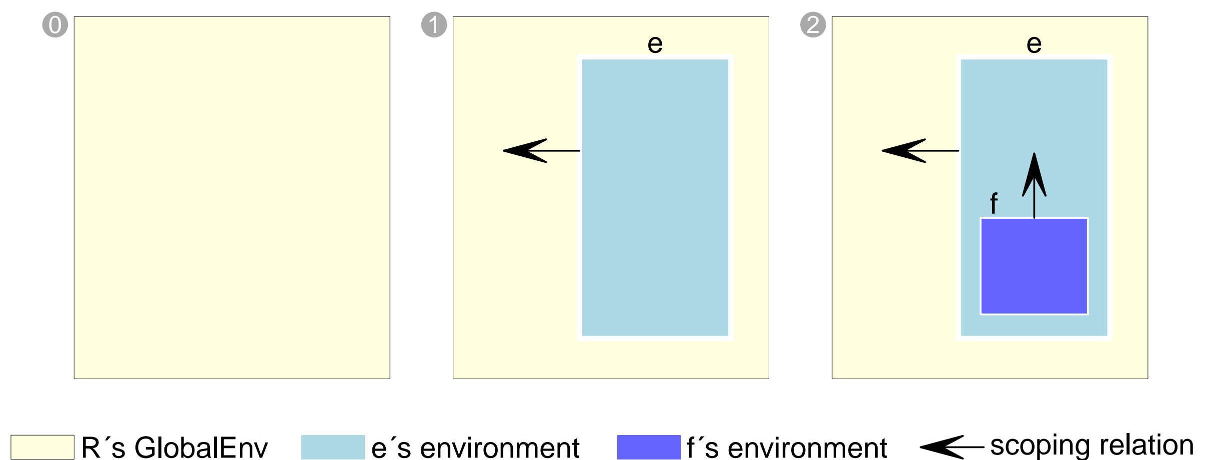
```
> L["a"]
$a
NULL

> L["b"]
$<NA>
NULL
```

Chapter 3

1. Create an environment e in your workspace. Create another environment f which is inside e. Explain, eventually with a picture, the scoping relation of these environments.

```
> e <- new.env()
> assign("f", new.env(), envir = e)
```

The figure below shows the state of these environments initially (0), after the first command (1), and after the second one (2).



[1]This behaviour has varied slightly in previous versions of R.

2. Suppose there is a function with two arguments. How many ways are there to pass these arguments when calling this function? Answer the question by supposing first that there are no default values, then that both arguments have a default value.

   In the first case, the function would be defined like this:

```
> fun1 <- function(x, y) { ...... }
```

   Suppose we want to call `fun1` giving the values 1 and 2 to its arguments, there are five ways to do so:

```
> foo(1, 2)
> foo(1, y = 2)
> foo(x = 1, 2)
> foo(x = 1, y = 2)
> foo(y = 2, x = 1)
```

   In the second case, the function definition would become:

```
> fun2 <- function(x = 1, y = 2) { ...... }
```

   This function can be called without argument so that the default values will be (the same would give an error with `fun1()`):

```
> fun2()
```

   It is possible to modify the value of one argument in different ways:

```
> fun2(10)
> fun2(, 20)
> fun2(x = 10)
> fun2(y = 20)
```

   Note the comma in the second command. Finally, both arguments can be modified in the following ways:

```
> fun2(10, 20)
> fun2(x = 10, 20)
> fun2(10, y = 20)
> fun2(x = 10, y = 20)
> fun2(y = 20, x = 10)
```

   Because there are only two arguments, mixing passing arguments with position and with names is of limited interest.

3. Write a function that "captures" the '...' argument into a list, modifies this list, and returns it. Explain the usefulness of this manipulation.

   If a function has the '...' argument, the command (inside the function) `dots <- list(...)` copies any extra argument in a list with names set from the named arguments (if any). This makes possible to do additional checks on these arguments.

For instance, in a data plot, if the values given to `xlim` are not finite, this will give an error. Here, we write a function that checks this argument and correct the values with limits $[0, 1]$ (say we plot proportions):

```
fun <- function(...)
{
    dots <- list(...)
    i <- which(names(dots) == "xlim")
    if (length(i)) {
        if (any(!is.finite(dots[[i]]))) {
            warning("argument 'xlim' had non-finite value(s): changed
                to [0,1]")
            dots[[i]] <- c(0, 1)
        }
    }
    ## other commands to set up the arguments (x) to plot()
    ....
    do.call(plot, dots)
}
```

In this example, we check for both missing and infinite values since `is.finite(NA)` and `is.finite(NaN)` also return `FALSE`.

4. Explain why the function `foo` at the end of Section 3.3.1 "does nothing".

   Actually, the function evaluates the condition within the `if( )` which is always `FALSE`, so the value 1 is never evaluated, thus the function returns `NULL`.

5. Explain as simply as possible what is happening when executing the code that demonstrates that $2 + 2 = 5$ on page 29.

   The command `` `2` <- 3 `` is identical to the following one:

```
> assign(x = "2", value = 3)
> ls()
[1] "2"
```

   So there is an object named `"2"` in our workspace. To get its contents, the command `` `2` `` is identical to:

```
> get("2")
[1] 3
```

6. Assess the performances of the different R implementations of the factorial function (p. 30). You will also comment on the memory resources required by these functions. (Hint: you may need to use some resources from Chap. 7.)

   After a few tests, we find that the four R functions and R's `factorial` are very fast even with $n = 170$. So we replicate each command one thousand times in order to reach running times that are at least 1 ms. We limit ourselves to the case $n = 170$:

```
> system.time(replicate(1000, fact(170)))
   user  system elapsed
  0.103   0.004   0.108
> system.time(replicate(1000, fact1(170)))
   user  system elapsed
  0.012   0.000   0.012
> system.time(replicate(1000, fact2(170)))
   user  system elapsed
  0.006   0.000   0.006
> system.time(replicate(1000, fact3(170)))
   user  system elapsed
  0.002   0.000   0.002
> system.time(replicate(1000, factorial(170)))
   user  system elapsed
  0.002   0.000   0.001
```

7. The Fibonacci series is defined by: $f_0 = 0$, $f_1 = 1$, and $f_i = f_{i-1} + f_{i-2}$ for $i \geq 2$. Propose R functions to implement Fibonacci series either with or without a recursive function. Compare the performances of both functions. (See the hint of the previous question.)

   The recursive solution is named `fibo1` with the following code:

```
fibo1 <- function(n)
{
    if (n < 2) n else fibo1(n - 1) + fibo1(n - 2)
}
```

   We consider three alternatives without recursions. The first one, `fibo2`, initializes a numeric vector with $n + 1$ values so that f_0 is stored in `x[1]`, and so on, until f_n which is stored in `x[n + 1]`. The cases $n = 0$ and $n = 1$ are handled at the start of the code, then the values of the vector are calculated by iterations along a `for` loop, and the last value is returned:

```
fibo2 <- function(n)
{
    if (n < 2) return(n)
    x <- numeric(n + 1)
    x[2] <- 1
    for (i in 3:(n + 1)) x[i] <- x[i - 1] + x[i - 2]
    x[n + 1]
}
```

   The second alternative is based on the fact that f_n is calculated with only the two previous values of the series, so `fibo3` stores only three values (`x1`, `x2`, `x3`) which are updated successively along the `for` loop:

```
fibo3 <- function(n)
{
    if (n < 2) return(n)
    x1 <- 0
    x2 <- 1
    for (i in 3:(n + 1)) {
        x3 <- x2 + x1
        x1 <- x2
        x2 <- x3
    }
    x3
}
```

Finally, the third alternative considers that the series grows exponentially so that the largest possible value is reached with $n = 1476$ (see Chap. 8 for explanations), so it is possible to create a vector with 1477 values, then pre-calculate all values for $n = 0, \dots, 1476$, and write a function (`fibo4`) that returns the appropriate value with respect to its argument (`n`):

```
.fib.series <- numeric(1477)
.fib.series[2] <- 1
for (i in 3:1477)
    .fib.series[i] <- .fib.series[i - 1] + .fib.series[i - 2]
fibo4 <- function(n) .fib.series[n + 1]
```

When the above code is loaded, the vector `.fib.series` is stored in the user's workspace together with the code of `fibo4`.

We can now compare the running-times of these four functions (see Chap. 7) using the following script:

```
N <- c(10, 20, 31:38, 100, 1000)
TIMES <- matrix(NA_real_, length(N), 4)
rownames(TIMES) <- N
colnames(TIMES) <- paste0("fibo", 1:4)
for (n in N) {
    i <- as.character(n)
    if (n <= 38) TIMES[i, 1] <- system.time(fibo1(n))[3]
    TIMES[i, 2] <- system.time(fibo2(n))[3]
    TIMES[i, 3] <- system.time(fibo3(n))[3]
    TIMES[i, 4] <- system.time(fibo4(n))[3]
}
```

We assess the performance of the recursive version with only $n \leq 38$ for reasons that will become clear soon. The results (in seconds) are given in the table below:

n	fibo1	fibo2	fibo3	fibo4
10	0	0	0	0
20	0.009	0	0	0
31	1.998	0	0	0
32	3.289	0	0	0
33	5.346	0	0	0
34	8.659	0	0	0
35	13.901	0	0	0
36	22.471	0	0	0
37	36.423	0	0	0
38	60.052	0	0	0
100		0	0	0
1000		0	0	0

Clearly, all times reported as zero were actually less than 0.001 sec. Also, the script was run several times before recording the running times so that the first executions of the functions were not significantly slower than the subsequent ones (see `?cmpfun`). The running times of `fibo1` are well approximated by the following linear model (we drop the first value which was found to be $< 10^{-3}$ sec):

```
> summary(lm(log10(TIMES[2:10, 1]) ~ N[2:10]))

Call:
lm(formula = log10(TIMES[2:10, 1]) ~ N[2:10])

Residuals:
      Min        1Q    Median        3Q       Max
-0.008244 -0.005710 -0.002000  0.004708  0.008877

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept) -6.2849850  0.0152917  -411.0   <2e-16
N[2:10]      0.2122866  0.0004596   461.9   <2e-16

Residual standard error: 0.006953 on 7 degrees of freedom
Multiple R-squared:      1, Adjusted R-squared:      1
F-statistic: 2.134e+05 on 1 and 7 DF,  p-value: < 2.2e-16
```

So we can predict the running times t of this function for large values of n with:

$$\log_{10} t \approx 0.21 \times n - 6.28$$

Which gives for $n = 100$ and $n = 1000$:

```
> 10^(0.21 * 100 - 6.28)
[1] 5.248075e+14
> 10^(0.21 * 1000 - 6.28)
```

```
[1] 5.248075e+203
```

As a comparison, the age of the Universe is $\approx 4.7 \times 10^{17}$ sec (one year is approximately 31.5 million seconds).

Bonus. With the concepts discussed in Chapter 8 it is possible to evaluate whether a C code computing the Fibonacci series would alleviate the limitations of the above R recursive version. We write a stand-alone C program to do this and print the running time:

```
#include <stdio.h>
#include <time.h>

double fiboC(double n)
{
    return n < 2 ? n : fiboC(n - 1) + fiboC(n - 2);
}

void main(void)
{
  clock_t t0, t1;
  int i;
  double n, z;
  double N[]={10, 20, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42};

  for (i = 0; i < 14; i++) {
    n = N[i];
    t0 = clock();
    z = fiboC(n);
    t1 = clock();
    printf("%f %.6f\n", n, (t1 - t0)/1e6);
  }
}
```

The variables are defined as double precision numbers in order to be consistent with the R versions. However, it is possible here to store n as a long integer which is simply done by replacing all instances of `double` with `long` in the above program (see Chap. 8 for details). Additionally, when compiling it is possible to specify different optimisation levels which are likely to improve the performance: we test here with -O0 (the default of GCC), -O1 (basic optimisations), -O2 (used when compiling code with R), and -O3 (aggressive optimisations). The results are in the following table:

	double				long			
n	-O0	-O1	-O2	-O3	-O0	-O1	-O2	-O3
10	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0
31	0.014	0.015	0.006	0.007	0.019	0.011	0.004	0.004
32	0.023	0.029	0.012	0.012	0.030	0.018	0.007	0.006
33	0.041	0.045	0.016	0.019	0.044	0.031	0.013	0.010
34	0.062	0.073	0.028	0.026	0.067	0.047	0.018	0.015
35	0.098	0.110	0.055	0.034	0.116	0.075	0.024	0.025
36	0.157	0.170	0.091	0.054	0.192	0.116	0.043	0.042
37	0.262	0.283	0.141	0.101	0.306	0.196	0.066	0.075
38	0.411	0.420	0.173	0.149	0.444	0.340	0.116	0.107
39	0.674	0.635	0.276	0.225	0.626	0.510	0.151	0.284
40	1.114	0.841	0.409	0.473	1.012	0.871	0.273	0.368
41	1.802	1.797	0.674	0.880	1.931	1.392	0.426	0.696
42	3.301	2.630	1.179	1.275	3.201	2.130	0.796	0.700

Both C versions are faster than any of the R ones, but there are significant differences among them: the long integer versions run slightly faster than the double versions, and the optimisation option can result in code which is 2–3 times faster. However, all these versions scale in the same way with n: linear regressions similar to the above one give coefficients around 0.19 and intercepts around -8, so the predicted running times for large values of n are still astronomical:

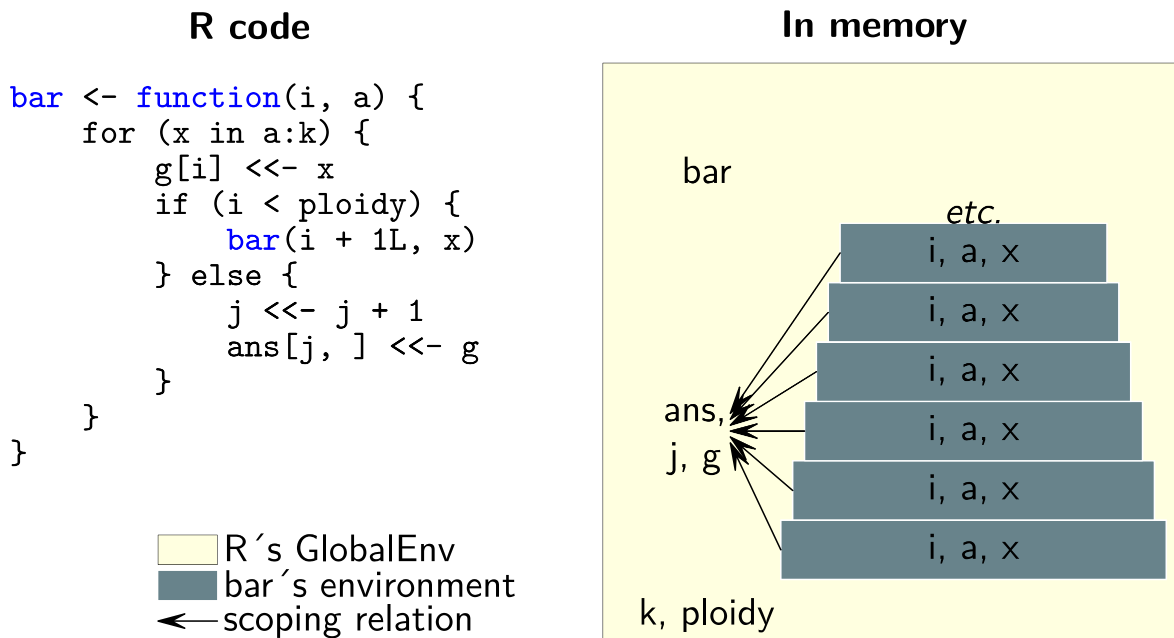
```
> 10^(0.19 * 100 - 8)
[1] 1e+11
> 10^(0.19 * 1000 - 8)
[1] 1e+182
```

8. Write a function solving the example of nested ZIP archives on page 32.

```
recursiveUnzip <- function(file)
{
    res <- unzip(file)
    zips <- grep("\\.zip$", res, ignore.case = TRUE, value = TRUE)
    if (length(zips)) sapply(zips, recursiveUnzip)
}
```

See Section 4.3.2 for a description of `grep()`.

9. Draw a picture similar to Fig. 3.2 applied to the recursive function listing all genotypes (p. 34).



10. Give the logic (or, better, the algorithm) explaining how the recursive function listing all genotypes (p. 34) works.

    Let's consider the simplest case with two chromosomes (*ploidy* = 2) and two alleles ($k = 2$): there are three possible genotypes, 1/1, 1/2, and 2/2, since the order of the alleles is not important (there is only one locus). So if the first allele is 2, the second allele can be only 2 as well, since there is no need to "revisit" allele 1. We generalise to k alleles by setting the first allele to any one between 1 and k while the second allele can be any one but avoiding those already visited for the first one. We further generalise to more than two chromosomes by applying this rule to the pairs of successive chromosomes.

    With a fixed value of ploidy, the algorithm would be straightforward with a series of nested `for` loops, but we want the code to work for any number of chromosomes. Thus, a way to describe the recursive algorithm is to view a genotype as a stack of *ploidy* elements where the ith element varies between the value in the previous position and k. The recursive function calls are stopped when the required number of elements are filled. The starting positions being different for each recursion, they are stored as local variables (`x`).

11. Create a data structure which associates a similarity matrix with a factor. Give a class (with the name of your choice) to this structure, and write a `print` method for it. Use this method, then delete it from your workspace and print the structure. Explain what you observe.

    We start by creating a dissimilarity (or distance) matrix between four values, and a factor that gives their classes (these values are arbitrary and help here to design the

class and its associated methods):

```
> d <- dist(rnorm(4)
> g <- gl(2, 2)
```

We now create the data structure: the most obvious way is to associate the two previous objects in a list with named elements, and we set its class to, say, `"myclass"`:

```
>data <- list(Dis = d, Class = g)
> class(data) <- "myclass"
```

To display this oject, we write a very simple `print` method (of course, it can be improved later):

```
> print.myclass <- function(x, ...)
+   cat("Object of class \"myclass\" with", length(x$class),
+   "observation(s)\n")
> X
Object of class "myclass" with 4 observation(s)
```

We delete this method and print `X` again:

```
> rm(print.myclass)
> X
$Dis
          1         2         3
2 1.2528289
3 1.1022235 2.3550524
4 0.4685073 0.7843216 1.5707308

$class
[1] 1 1 2 2
Levels: 1 2

attr(,"class")
[1] "myclass"
```

R now uses the function `print.default` to print the object.

Chapter 4

1. In the example on page 45, explain why `L` is a list and not a vector. What is the result of `mode(unlist(L))`?

   In a vector, all elements are of the same (storage) mode.

```
> mode(unlist(L))
[1] "character"
```

All elements are coerced to the mode of the first one.

2. One of your colleagues has a data set arranged in a data frame with 100 rows and 1000 columns. Overall, there is 0.1% of missing data, and your colleague thinks this is not a problem. Explain why this could be a problem bigger than they think.

   It all depends on how missing values are handled. In most model fitting functions (e.g., `lm`), rows with at least one `NA` are removed before analysis. In the present case, there are 100 `NA`'s, and if they are clustered in a few columns (for instance, because these variables were difficult to measure), this could lead to remove a very big proportion of the data frame. In that situation, it might be a better idea to not consider these variables from the analyses.

3. Generate one million random variables following a normal distribution using the default parameters. How many of these variables fulfill the condition $|x| > 4$? Repeat this exercise but setting the variance of the normal distribution to $\sigma^2 = 2$. Could you calculate the numbers expected in both cases? (Hint: use `pnorm()`).

```
> x <- rnorm(1e6)
> sum(abs(x) > 4)
[1] 68
> z <- rnorm(1e6, sd = sqrt(2))
> sum(abs(z) > 4)
[1] 4711
> (1 - pnorm(4)) * 2 * 1e6
[1] 63.34248
> (1 - pnorm(4, sd = sqrt(2))) * 2 * 1e6
[1] 4677.735
```

4. Find the regular expression (regexp) that will match the string 'R' and only this one. Look at the options in `grep` (and other functions) and find the one that bypasses the need to find this regexp.

   `grep("^R$", x)` and `grep("R", x, fixed = TRUE)` do the same operation.

5. List all objects (functions, data, ...) loaded in memory with a name starting with 'lm'.

   With R recently started:

```
> apropos("^lm")
[1] "lm"           "lm.fit"       "lm.influence" "lm.wfit"
```

6. You want to split character strings into single words: find the efficient code to do this operation with `strsplit`.

   If `x` is a character vector with n strings, the command `strsplit(x, "[[:space:]]+")` returns a list with n vectors where the strings in `x` have been cut along the spaces, tabulations, and/or end of lines. We use the code for character classes (Table 4.4) and the plus sign meaning that these spaces must be present once or more (Table 4.3). We note that the brackets are doubled.

7. Explain how `match()` can be used when handling several objects, particularly data frames.

   Two data frames with the same identifiers (subject, species, ...) but in different orders can be merged easily after `match()` finds the correspondences.

8. Explain why logical values used as indices are recycled but not numeric ones.

   It would not make sense to recycle numeric indices because these can be repeated if the user wants to replicate some values, rows, or elements.

9. Explain why `v` is not quoted in the above example (p. 60). What would happen if it were quoted (i.e., `X[["v"]]`)?

   Because the value stored in `v` is evaluated at each iteration, and takes successively the values in `names(X)`.

   The command `X[["v"]]` might be succesful if there is an element named `"v"` in the list `X`.

10. Explain the difference between `X[["2022"]]` and `X[[2022]]` from the examples above.

    The first command extracts the element named `"2022"` from `X` or `NULL` is there is none, whereas, the second one extracts its 2022th element or gives an error if the length of `X` is less than 2022.

11. Compare the performance of logical and numeric indexing for vectors of different sizes (up to 10^8).

    Let's say we want to select the positive values from a vector `x` which can be done by both types of indexing. We use the very simple following code:

```
N <- 10^(5:8)
RES <- NULL
for (n in N) {
    x <- rnorm(n)
    sel <- x > 0
    RES <- rbind(RES, system.time(x[sel]))
    RES <- rbind(RES, system.time(x[which(sel)]))
}
```

Note that the logical operation is not included in the timing evaluation since it is common to both indexing types. We display the results with a data frame:

```
R> data.frame(n = nn, Meth = c("Logi", "Nume"), RES[, 1:3])
       n Meth user.self sys.self elapsed
1 1e+05 Logi     0.001    0.000   0.001
2       Nume     0.001    0.000   0.001
3 1e+06 Logi     0.006    0.000   0.006
4       Nume     0.006    0.000   0.006
5 1e+07 Logi     0.062    0.012   0.074
6       Nume     0.070    0.016   0.086
7 1e+08 Logi     0.625    0.216   0.842
```

```
8        Nume      0.694     0.216    0.911
```

There is a slight advantage to logical indexing but which hardly seems significant in practice. Besides, it could be shown that about two thirds of the running times of the numerical indexing is due to the `which()` function. Therefore, there is no performance argument, but only practical ones, to prefer one type of indexing over the other.

12. Write a program to perform Dawkins's weasel problem.[2] You will use the approximate string distance to evaluate the fitness of the new mutants. In addition to `adist`, you will probably need the following functions: `runif`, `sample`, `substr`, `which.min`, and others introduced in this chapter. Compare your results with an implementation that uses a fitness function based on the Hamming distance.[3]

Briefly, the algorithm starts with an initial random set of 28 characters made of letters and spaces. The set is replicated 100 times. Each of the 2800 characters is mutated with a fixed probability, and the resulting set of 28 characters closest to the target is selected to be the next generation which follows the same process of replication–mutation–selection.

There are a number of ways to implement this algorithm. We choose a matrix (`X`) with 100 rows and 28 columns which simplifies the mutation process since we generate 2800 random uniform values and test which ones are less than the mutation rate (and avoids the use of `substr`).

We define the parameters:

```
TARGET <- "METHINKS IT IS LIKE A WEASEL"
target <- strsplit(TARGET, NULL)[[1]]
mu <- 0.05 # mutation rate
AD <- function(x, y) adist(paste(x, collapse =  ""), TARGET)
n <- length(target)
N <- 100
```

We store separately the target as a single string (`TARGET`) and as a vector of single characters (`target`) since the approximate distance requires the former.

```
X <- matrix(sample(Z, n, replace = TRUE), N, n, byrow = TRUE)
g <- 0
repeat {
    g <- g + 1
    if (g > 1e3) break
    s <- which(runif(N * n) <= mu)
    X[s] <- sample(Z, length(s), replace = TRUE)
    best <- which.min(apply(X, 1, AD))
    cat(g, "\t", paste(X[best, ], collapse = ""), "\n")
    if (identical(X[best, ], target)) break
```

[2] https://en.wikipedia.org/wiki/Weasel_program
[3] http://rosettacode.org/wiki/Evolutionary_algorithm#R

```
    X[] <- matrix(X[best, ], N, n, byrow = TRUE)
}
```

This was replicated 100 times: the algorithm did not converge after 1000 generations in 82 cases. For 18 other cases, the target was reached after a mean of 114 generations (range: 39, 395).

On the other hand, an implementation with the Hamming distance has a simpler function:

```
HD <- function(x) sum(x != target)
```

We simply need to replace `AD` by `HD`. The target was reached after a mean of 76 generations (range: 39, 148).

The command `cat(....` shows that the approximate distance gets easily "trapped" with alternative positions of the spaces.

Chapter 5

1. Figure 5.1 could have an additional arrow. Explain why this is obvious.

   An expression can be built with the function `expression()`: this could represented by a second arrow from "KEYBOARD" to "expression".

2. Build the following expression `e <- expression(x <- x + 1)`. Then, run the command `eval(e)`. Do you expect an error? Explain your answer.

   If an object `x` is already present in the user's workspace, there would be no error if the addition operation is valid for this object. Otherwise, an error will occur.

3. Run the command `plot(0, 0, "n")`. Add the annotation $\sqrt{2\pi}$ at the center of the plot.

   ```
   text(0, 0, expression(sqrt(2*pi)))
   ```

   Note that no multiplication sign ($\times$) is printed between 2 and π.

4. Find the second partial derivative of the logarithm of x (i.e., $\partial^2 \ln x / \partial x^2$) using the function `D`.

   The answer is given directly by nesting two calls of `D`:

   ```
   > D(D(expression(log(x)), "x"), "x")
   -(1/x^2)
   ```

   Since `D` returns an expression, a more readable version of this code could be (and it leaves copies of the different expression objects for possible future evaluations):

   ```
   > log.expr <- expression(log(x))
   > deriv1x <- D(log.expr, "x")
   > deriv2x <- D(deriv1x, "x")
   ```

```
> deriv1x
1/x
> deriv2x
-(1/x^2)
```

5. Build the formula `y ~ x1 + x2`. Explain why no addition is made when building this formula.

   Mathematical operators have different meanings when they are used in a formula: here it means that the model describes the response `y` as the additive effect of the predictors `x1` and `x2`.

6. Read a date written in the standard US format '01/31/2022' into an object of class `"Date"`.

```
> as.Date("01/31/2022", "%m/%d/%Y")
[1] "2022-01-31"
```

7. Below are the five top rows of a file:

```
Year   Month   Day
2019    Feb    12
2020    May    15
2021    Nov    14
2021    Dec    14
```

   How would you convert these data into the class `"Date"`?

   We first read the file and build a character vector with the dates:

```
> X <- read.table("dates.txt", header = TRUE)
> dates <- paste(X$Year, X$Month, X$Day, sep = "-")
> dates
[1] "2019-Feb-12" "2020-May-15" "2021-Nov-14" "2021-Dec-14"
```

   We note that the order is not important because we will need to specify the format since the months are in abbreviated form. Besides, if we may need to set the locale:

```
> Sys.setlocale(locale = "en_US.UTF-8")
## <output skipped>
> (dates <- as.Date(dates, "%Y-%b-%d"))
[1] "2019-02-12" "2020-05-15" "2021-11-14" "2021-12-14"
```

   We can now easily calculate, for instance, the date 30 days later each of these:

```
> dates + 30
[1] "2019-03-14" "2020-06-14" "2021-12-14" "2022-01-13"
```

8. You have data including dates marked with either "BCE" or "AD" (the latter could be "CE" in a more recent notation). What special care (and eventually manipulation) you should take when calculating time intervals with these data?

9. Type the following command in R: `1e400 < 1e500`. Explain the results using a figure.

```
> 1e400 < 1e500
[1] FALSE
```

From Fig. 5.3B, we see that all (finite) numbers greater than the largest representable number (displayed next) are represented as "Inf":

```
> .Machine$double.xmax
[1] 1.797693e+308
> 1e400
[1] Inf
> 1e500
[1] Inf
```

10. Explain the following result:

```
> A <- 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 2^52
> B <- 2^52 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1
> A > B
[1] TRUE
```

In Appendix B we can learn that $2^{52} + 0.1$ is not a representable number so the result of this operation is stored as 2^{52}. On the other hand, $2^{52} + 0.6$ (which is not representable too) is stored as $2^{52} + 1$.

The above additions are done successively from left to right. So the calculation of `A` is identical to `0.6 + 2^52`, while in the case of `B` each addition of 0.1 is "lost".

What if instead of adding six times 0.1, we would add five times this same number?

Since $2^{52} + 0.5$ is stored as 2^{52}, `A` and `B` would be identical.

Show (and explain) how parentheses could lead to a more "logical" result.

```
> B <- 2^52 + (0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1)
> A == B
[1] TRUE
```

The parentheses modify the priorities of the operators.

11. The mass of a blue whale is about 30 tons and that of a bacterium is about 1 pg ($= 10^{-12}$ g). Logically a whale and a bacterium are heavier than a whale alone: how can you show this using `log1p`? (Reminder: 1 ton = 1 Mg = 10^6 g.)

We can test the statement "a whale and a bacterium are heavier than a whale alone" after converting both masses into the same unit (gram):

```
> whale <- 30 # unit: Mg (1 Mg = 1 ton)
> bacterium <- 1 # unit: pg
> x <- whale * 1e6
> y <- bacterium * 1e-12
> x < x + y
[1] FALSE
```

This clearly doesn't work. Now let w denote the mass of a blue whale in picograms. We want to show that $w + 1 > w$. Let us divide both sides of the inequality by w (since $w > 0$):

$$\frac{w+1}{w} > \frac{w}{w}$$
$$1 + \frac{1}{w} > 1$$

We now log-transform both sides (reminder: $\forall x, y > 0 : x > y \Leftrightarrow \ln x > \ln y$) to have (since $\ln 1 = 0$):

$$\ln\left(1 + \frac{1}{w}\right) > 0$$

We use $w = 3 \times 10^{19}$ pg:

```
> log1p(1/3e19) > 0
[1] TRUE
```

The number $\frac{1}{3} \times 10^{19}$ is in fact a denormal number (see Appendix A):

```
> 1/3e19 + 1 > 1
[1] FALSE
```

12. Try the following command:
    `(1.2 - 0.8) * 1e16 == 0.4 * 1e16 - 1`
    Explain the results.

```
> (1.2 - 0.8) * 1e16 == 0.4 * 1e16 - 1
[1] TRUE
```

We start with the well-known result (since 0.4 is not a representable number):

```
> 1.2 - 0.8 == 0.4
[1] FALSE
```

The difference between these two quantities is of the order of 10^{-16}:

```
> 1.2 - 0.8 - 0.4
[1] -1.110223e-16
```

So multiplying both of them by 10^{16} gives two numbers which differ by one. A way to show this is to print all digits:

```
> sprintf("%.0f", c((1.2 - 0.8), 0.4) * 1e16)
[1] "3999999999999999" "4000000000000000"
```

13. Compare these two commands and explain the results:
`sqrt(2)^2 == 2`
`sqrt(2^2) == 2`

These commands compute $(\sqrt{2})^2$ and $\sqrt{2^2}$, respectively, which are mathematically identical. The first command first computes $\sqrt{2} \approx 1.414214$ which is an irrational number, and cannot be represented as a 64-bit floating point number. So, it is difficult to predict whether its power of two will result in the initial value. On the other hand the second command first computes $2^2 = 4$ which is a representable number so that its square root can be computed exactly.

```
> sqrt(2)^2 == 2
[1] FALSE
> sqrt(2^2) == 2
[1] TRUE
```

Chapter 6

1. Type the three following commands respecting the (lack of) spaces:

```
x<-rnorm(10)
x<-1
x<+1
```

Explain the results and comment on the good practice of writing R code.

The commands and their outputs in R are:

```
> x<-rnorm(10)
> x<-1
> x<+1
[1] FALSE
```

The second command is interpreted by R to actually mean `x <- 1`:

```
> x<-1
> x
[1] 1
```

On the other hand, the third command is interpreted as `x < +1`. To avoid these ambiguities, it is recommended to add spaces around the assignment operator:

```
> x <- rnorm(10)
> x
 [1]  1.2308499  0.5221689  2.5397653 -2.5392730 -0.2276048
 [6]  1.1502043 -0.8737982  0.3076795 -1.1755966  0.6307133
> x < -1
[1] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE
> x < +1
[1] FALSE  TRUE FALSE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE
```

2. Matching parentheses (or brackets, or braces) are common problems when writing R code. How to avoid these problems?

   Use an editor or an IDE (integrated development environment): most of them highlight matching text delimiters.

3. Suppose we want to execute the sum of a vector step-by-step. We first run `debug(sum)`, then `sum(rnorm(1000))`. Explain what you observe.

   Nothing happens, apart from the expected result. `sum` is a primitive function of R, and `debug` does not work on primitive functions.

4. Suppose you wrote a script with 50 lines of commands; you then try to run your script with `source()` and an error occurred. What is the best strategy to solve this problem?

   Unless the error is easily identified (which is not the general case), it is best to open the script, execute its commands step-by-step, and check if the outputs are as expected.

5. Explain why the functions `mean`, `var`, `median`, `quantile`, `max`, `min`, `range`, and `prod` may lead to errors when missing values are likely to be present in the data. Propose two ways to handle this problem.

   These functions have the option `na.rm = FALSE`, so their output is `NA` if there is at least one missing observation which may be problematic in subsequent analyses.

   One way to handle this problem is to systematically switch the above option to `TRUE` (with some possible caveats that depend on the details of the data and of the study).

   The second way is to add commands that will handle data with missing values and remove them (or replace them by imputation or anoother method).

6. Give some examples of data checking you could use in your code to avoid errors.

   Checking the length (if the input is a vector), the number of rows (if a data frame), the mode of the vectors, the presence of `NA`, ...

7. You are writing a method associated to a generic function (see Sect. 3.5): is it useful to check the class of the input data?

   In the majority of cases, a method is not called directly but through the corresponding generic, so checking the class is superfluous. However, it is still possible to by-pass

this calling mechanism, which might be useful for expert users. It is thus left to the author of the functions to decide on the usefulness to add a check of the class of the data.

8. A common situation to handle when programming a general purpose function is that no data in a vector (or a data frame) meet some requirement(s). How would you test for this and return a message to the users?

   The test can be done in the usual way, for instance, if the input should have only positive values:

```
if (any(x <= 0)) ....
```

   The action to take in this situation depends a lot on the context. If the function is meant to be used in scripts together with other functions, it might not be the best idea to return an error because this would stop the execution of the script. Instead, returning `NULL` (or another conventional value such as `NaN`) with a warning message can give the possibility to continue the computations.

9. What attribute(s) would you test to check the data type input to a function?

   Typically, the length and the mode.

10. Why is it useful to reset the global option `options(error = NULL)` after finishing to debug a function?

    Otherwise a simple typing error when entering a command will open the browser.

Chapter 7

1. Write down a list of the data analysis methods that you use commonly (e.g., correlation, ANOVA, PCA, and so on). Try to associate each of these methods to the curve of type I, II, or III displayed on Fig. 7.1A.

2. Suppose we have a sample of n observations from a standard uniform distribution $\mathcal{U}(0, 1)$. The largest value of the sample is a random variable with variance given by the two mathematically identical expressions:

$$\frac{n}{n+2} - \left(\frac{n}{n+1}\right)^2 = \frac{n}{(n+2)(n+1)^2}.$$

   Which one should be preferred to code in a computer program?

   At first sight, the rhs expression contains five operations while the lhs one contains six. Because of the priority of multiplication and division over addition and subtraction, the lhs expression will first involve two ratios which tend to $\frac{n}{n} = 1$ when n is large, so that the final result will be zero. The rhs expression, on the other hand, requires to first compute the denominator which is $n^3 + 4n^2 + 5n + 2$. Although the difference is minute, it is apparent for large values of n:

```
> n <- c(2:10, 10^(2:12))
> cbind(n, Left = n/(n+2) - (n/(n+1))^2, Right = n/((n+2)*(n+1)^2))
          n         Left        Right
 [1,] 2e+00 5.555556e-02 5.555556e-02
 [2,] 3e+00 3.750000e-02 3.750000e-02
 [3,] 4e+00 2.666667e-02 2.666667e-02
 [4,] 5e+00 1.984127e-02 1.984127e-02
 [5,] 6e+00 1.530612e-02 1.530612e-02
 [6,] 7e+00 1.215278e-02 1.215278e-02
 [7,] 8e+00 9.876543e-03 9.876543e-03
 [8,] 9e+00 8.181818e-03 8.181818e-03
 [9,] 1e+01 6.887052e-03 6.887052e-03
[10,] 1e+02 9.610746e-05 9.610746e-05
[11,] 1e+03 9.960110e-07 9.960110e-07
[12,] 1e+04 9.996001e-09 9.996001e-09
[13,] 1e+05 9.999590e-11 9.999600e-11
[14,] 1e+06 1.000089e-12 9.999960e-13
[15,] 1e+07 9.880985e-15 9.999996e-15
[16,] 1e+08 2.220446e-16 1.000000e-16
[17,] 1e+09 0.000000e+00 1.000000e-18
[18,] 1e+10 0.000000e+00 1.000000e-20
[19,] 1e+11 0.000000e+00 1.000000e-22
[20,] 1e+12 0.000000e+00 1.000000e-24
```

3. The k-means method is an unsupervised classification method that finds structure (grouping) from numerical, continuous data (see `?kmeans`). Assess the running time of this method with 10^6 observations from two normal distributions with means -2 and 2 (with equal sample size in each group). Repeat the same analysis but with all observations drawn from a normal distribution with mean zero.

```
> X <- rnorm(1e6, c(-2, 2))
> Y <- rnorm(1e6)
> system.time(kmeans(X, 2))
   user  system elapsed
  0.078   0.012   0.091
> system.time(kmeans(Y, 2))
   user  system elapsed
  0.180   0.008   0.188
```

This is an illustration that running times do not depend only on data size or on the implementation, but also on the internal (statistical) structure of the data that the algorithm aims to find. This suspicion is confirmed by the fact that if we increase the number of groups, both data sets need longer running times:

```
> system.time(kmeans(X, 3))
   user  system elapsed
```

```
  0.409   0.008   0.418
> system.time(kmeans(Y, 3))
   user  system elapsed
  0.279   0.012   0.292
```

4. Repeat the comparison between the functions `princomp` and `prcomp` using `Rprof()` but this time setting $n = 10^4$ and $p = 1000$. Comment on the differences with the results above.

5. Explain why it is common that the code of functions include the command `n <- length(x)`. Give other examples of similar commands.

6. You need to run simulations that are expected to take several days. The output of each simulation replicate will then be analysed with a short R script that you have downloaded from Internet. After looking at the code of the script, you realise that the code can be easily improved to make it faster. What is your decision?

7. Build a matrix (say `X`) with random values of your choice with n rows and p columns. Evaluate the running times for several values of n and p. Represent the results graphically.

8. Do performance profiling of the command `x <- rnorm(1)`. Explain the observed results. Repeat this analysis with larger values passed to `rnorm`. Which option of `Rprof` you may need to adjust and why?

9. Perform the memory profiling of the command `x <- rnorm(1e7)`. Are the results as expected?

10. Find the code of the function `g` used at the end of this chapter.

```
g <- function(x)
{
    class(x) <- NULL
    p <- length(x)
    res <- numeric(p)
    for (i in 1:p) res[i] <- sum(x[[i]])
    res
}
```

This version results in even faster running times than reported in the book (the code for `f()` is given in the book):

```
> system.time(of <- f(DF))
   user  system elapsed
  0.153   0.000   0.153
> system.time(og <- g(DF))
   user  system elapsed
  0.015   0.000   0.015
```

Chapter 8

1. What is the largest amount of memory usable (i.e., addressable) by a 32-bit CPU?
2. Explain why the size of a pointer is eight bytes on a 64-bit system.
3. How much memory is needed to store one million numerical values in R? What is the gain in terms of memory space if these values are binary?
4. Suppose you need to store numerical values between 0 and 9 in R: what is the most economical way to store them and compare with the standard numeric vectors.
5. Explain why indices in C start at 0 whereas they start at 1 in R.
6. Write the C code which is illustrated in Figure 8.1.
7. Write a C function doing the sum of the values of a vector similar to `sum_C` but using the `.Call` interface. Compare the performance of this version with the one called with `.C` (under different data sizes). Explain the observed differences (if any).
8. Write C code to find the indices of names (or labels) with possibly duplicated names (see p. 116).
9. In the above exercise, what is the value of `i` if the string is not found? (See p. 116.)
10. Write a C program, to be called from R, to do the sum of an indefinite number of vectors.
11. Matrices in C can be coded with an array of pointers, such as `**x`, and manipulated with two sets of indices so that the first element is accessed with `x[0][0]`, and the last one with `x[n - 1][p - 1]`, where `n` and `p` are the numbers of rows and columns, respectively. Explain the difference with the actual system used in R and its C interfaces (see Table 8.4), and why one system is more efficient than the other.

Chapter 9

1. Write down a list of the data analysis methods that you use commonly (e.g., correlation, ANOVA, PCA, and so on; see Exercises in Chap. 7). Try to write down whether these methods can be parallelised, and if yes sketch the approach which seems appropriate to you.
2. A colleague is performing a data analysis running in parallel on their computer while listening to a podcast and checking emails. What advice would you give to them?
3. Try to implement a parallel version of the sum of a vector with `mclapply()`. Compare the performance with `sum()`. Were the results predictable?
4. Do you think a parallel version of the factorial is a good idea?
5. Which model depicted on Fig. 9.1 seems the most appropriate to run a bootstrap in parallel? Same question for Monte Carlo simulations?

6. You need to analyse many data sets with the same method. Do you think this is a good idea to run them in parallel?
7. What is the probability that two files have the same name with the procedure explained in the previous section? What is the required condition to be sure that this does not happen? What if we had added the option `replace = TRUE` in the call to `sample()`?

January 5, 2024