SCR week 4: exercises

Exercises part 1

1.1 Write your own Songtext (Part 1)

In this exercise we will program a function to produce a file with the lyrics of the '99 bottles song' in three different versions; based on `lapply()`, `sapply()` and (perhaps?) `replicate()`.

a.

Take a look at the following code:

```
in_stock <- 99
paragraph <- paste0(
  in_stock, " bottles of beer on the wall, ",
  in_stock, " bottles of beer. \n",
  "Take one down, pass it around, ",
  in_stock - 1," bottles of beer on the wall...\n\n"
)
cat(paragraph)
```

```
## 99 bottles of beer on the wall, 99 bottles of beer.
## Take one down, pass it around, 98 bottles of beer on the wall...
```

Use `lapply()`, `sapply()`, to produce the paragraphs for `B = 10` to `B = 2` bottles of beer on the wall. What is the class of the output of `sapply()` what is the class of the output of `lapply()`?

Answer:

```
CreateParagraphsBottlesBeer <- function(bttls) {
  paragraph <- paste0(
    bttls, " bottles of beer on the wall, ",
    bttls, " bottles of beer. \n",
    "Take one down, pass it around, ",
    bttls - 1, " bottles of beer on the wall...\n\n"
  )
  # cat(paragraph)
  return(paragraph)
}
bttls10to2_lapply <- lapply(10:3, CreateParagraphsBottlesBeer)
class(bttls10to2_lapply)
```

```
## [1] "list"
```

```
bttls10to2_sapply <- sapply(10:3, CreateParagraphsBottlesBeer)
class(bttls10to2_sapply) # = atomic (one mode) in a vector structure.
```

```
## [1] "character"
```

b.

Since `replicate()` is a wrapper around the `sapply()` (and thus the `lapply()`) function, would it be "natural" the use the `replicate()` function to write the paragraphs of the N Bottles of Beers song?

Answer:

Nope, as is stated in the helpfile: "`replicate()` is for repeated evaluation of an expression". For each paragraph the expression changes, i.e. the number of bottles changes.

c.

Time the functions `lapply()` and `sapply()` for `B = 10` to `B = 2` "bottles on the wall" with (only) the use of `system.time()`. Can you say which of the two functions is slower?

Answer:

```
system.time(
  bttls10to2_lapply <- lapply(10:3, CreateParagraphsBottlesBeer)
)
```

```
##    user  system elapsed 
##       0       0       0
```

```
system.time(
  bttls10to2_sapply <- sapply(10:3, CreateParagraphsBottlesBeer)
)
```

```
##    user  system elapsed 
##   0.000   0.000   0.001
```

Nope they seem equally fast.

d.

Use the `replicate()` function to measure the running time when the expressions of the previous subquestions where each $B = 1000$ times repeated. Do you know have a more definite answer on which of the two functions is slower?

Answer:

```
B <- 1e3
system.time(
  replicate(B, bttls10to2_lapply <- lapply(10:3, CreateParagraphsBottlesBeer))
)
```

```
##    user  system elapsed 
##   0.043   0.005   0.047
```

```
system.time(
  replicate(B, bttls10to2_sapply <- sapply(10:3, CreateParagraphsBottlesBeer))
)
```

```
##    user  system elapsed
##   0.050   0.003   0.053
```

Jep, `sapply()` is slower, which makes sense since it is a wrapper around `lapply()`

e.

In this case using `simplify2array` or `unlist` would not make any difference for the output of `lapply()` in this exercise. Could you tell which of the two functions is faster?

Answer:

```
system.time(replicate(B, unlist(bttls10to2_lapply)))
```

```
##    user  system elapsed
##   0.002   0.000   0.003
```

```
system.time(replicate(B, simplify2array(bttls10to2_lapply)))
```

```
##    user  system elapsed
##   0.008   0.000   0.008
```

Here, `unlist()` is faster.

f.

Write a function that can put the lyrics of the song into a `.txt` file and can store it to any destination on your computer (given that the destination exists) with use of the `cat()` function. The input parameters (arguments) of your functions are: `B`, the number of bottles your start with; and the address for your "to be created" lyrics file. The function should return: `invisible(NULL)`.

Use (if you like) the following variables / objects inside your function:

```
header_song <- paste0("\n", N = 99, " Bottles of Beers \n\n")
end_song <- c("2 bottles of beer on the wall, 2 bottles of beer\n",
  "Take one down, pass it round 1 bottles of beer\n\n",
  " 1 bottle of beer on the wall, 1 bottle of beer\n",
  "No more bottles of beer\n"
)
```

Last, apply your created function to write your lyrics to the folder `0_data` in your working directory.

Answer:

```
SingTheBottleSong <- function(
  bottle_nr,
  address
) {

  each_bottle_nr <- bottle_nr:3
```

```r
  header_song <- paste0("\n", bottle_nr, " Bottles of Beers \n\n")

  CreateParagraphsBottlesBeer <- function(bttls) {
    paragraph <- paste0(
      bttls, " bottles of beer on the wall, ",
      bttls, " bottles of beer. \n",
      "Take one down, pass it around, ",
      bttls - 1, " bottles of beer on the wall...\n\n"
    )
    # cat(paragraph)
    return(paragraph)
  }

  firstpart_song <- lapply(each_bottle_nr, CreateParagraphsBottlesBeer)
  firstpart_song <- unlist(firstpart_song)
  end_song <- c(
    "2 bottles of beer on the wall, 2 bottles of beer.\n",
    "Take one down, pass it round 1 bottle of beer.\n\n",
    "1 bottle of beer on the wall, 1 bottle of beer.\n",
    "No more bottles of beer.\n"
  )

  lyrics <- c(
    header_song,
    firstpart_song,
    end_song
  )

  cat(lyrics, sep = "", file = address)
  return(invisible(NULL))
}
SingTheBottleSong(
  bottle_nr = 5,
  address = "0_data/my_bottlesong_lyrics.txt"
)
```

1.2 Wine Equality: <*>apply() family and aggregate

If you haven't done so, make sure you have finished the `swirl()` module: `11: vapply and tapply`.

a.

Read the file `winequality-red.ssv` into a `data.frame` object and write code that checks the `type` (or `mode`) of each column in the `data.frame`.

Answer:

```r
wine_quality <- read.csv("0_data/winequality-red.ssv", sep = ";", header = TRUE )
apply(wine_quality, 2, typeof)
```

```
##        fixed.acidity     volatile.acidity          citric.acid
##             "double"             "double"             "double"
```

```
##       residual.sugar            chlorides  free.sulfur.dioxide
##             "double"             "double"             "double"
## total.sulfur.dioxide              density                   pH
##             "double"             "double"             "double"
##            sulphates              alcohol              quality
##             "double"             "double"             "double"
```

b.

Categorize the columns `density` and `pH` into two new factors columns with each two categories: higher than the median, and lower or equal to the median.

Answer:

```
wine_quality$dens_fctr <- factor(
  wine_quality$density > median(wine_quality$density),
  labels = c("low", "high")
)
wine_quality$pH_fctr <- factor(
  wine_quality$pH > median(wine_quality$pH),
  labels = c("low", "high")
)
```

c.

Use `tapply()` to create a table for the means of `fixed.acidity` in a cross-table for "low and high density" with "low and high pH".

Answer:

t for table

```
tapply(X = wine_quality$fixed.acidity,
       INDEX = list(Density =  wine_quality$dens_fctr, pH =  wine_quality$pH_fctr),
       FUN = mean
)
```

```
##        pH
## Density      low     high
##    low  8.245079 6.943648
##    high 9.953452 7.858131
```

d.

Use `aggregate()` to find the means for `fixed.acidity` of all combinations of "low and high density" with "low and high pH".

Answer:

```
aggregate(
  x = wine_quality$fixed.acidity,
  by = list(
    Density = wine_quality$dens_fctr,
    pH = wine_quality$pH_fctr
```

```
  ),
  FUN = mean
)
```

```
##   Density   pH        x
## 1     low  low 8.245079
## 2    high  low 9.953452
## 3     low high 6.943648
## 4    high high 7.858131
```

e.

Repeat the sub-exercises **c** and **d**, but use an anonymous function that outputs the mean as well as the standard deviation. Does the output of both `tapply()` and `aggregate()` makes sense?

Answer:

```
aggregate(
  x = wine_quality$fixed.acidity,
  by = list(
    Density = wine_quality$dens_fctr,
    pH = wine_quality$pH_fctr
  ),
  FUN = function(x) {
         m <- mean(x, na.rm = TRUE)
         s <- sd(x, na.rm = TRUE)
         return(c(mean = m, sd = s))
       }
)
```

```
##   Density   pH    x.mean      x.sd
## 1     low  low 8.2450794 1.0290368
## 2    high  low 9.9534517 1.7092179
## 3     low high 6.9436475 0.8825252
## 4    high high 7.8581315 1.0434042
```

```
tapply(X = wine_quality$fixed.acidity,
       INDEX = list(
         Density = wine_quality$dens_fctr,
         pH =  wine_quality$pH_fctr
       ),
       FUN = function(x) {
         m <- mean(x, na.rm = TRUE)
         s <- sd(x, na.rm = TRUE)
         return(c(mean = m, sd = s))
       }
)
```

```
##        pH
## Density low       high
##    low  Numeric,2 Numeric,2
##    high Numeric,2 Numeric,2
```

The output of aggregate seems to be more user-friendly. The `tapply()` output is a matrix with elements of mode list. Each element of the matrix is a list of 1 element / component that contains the two numeric values: the mean and the standard deviation.

Btw, also notice the difference in speed of the code:

```
system.time(replicate(1e3, {
  obj_agg <- aggregate(
    x = wine_quality$fixed.acidity,
    by = list(
      Density = wine_quality$dens_fctr,
      pH = wine_quality$pH_fctr
    ),
    FUN = function(x) {
         m <- mean(x, na.rm = TRUE)
         s <- sd(x, na.rm = TRUE)
         return(c(mean = m, sd = s))
       }
  )
}))
```

```
##    user  system elapsed
##   1.313   0.042   1.358
```

```
system.time(replicate(1e3, {
  obj_tap <- tapply(X = wine_quality$fixed.acidity,
     INDEX = list(
       Density = wine_quality$dens_fctr,
       pH =  wine_quality$pH_fctr
     ),
     FUN = function(x) {
        m <- mean(x, na.rm = TRUE)
        s <- sd(x, na.rm = TRUE)
        return(c(mean = m, sd = s))
     }
  )
}))
```

```
##    user  system elapsed
##   0.193   0.004   0.198
```

Exercises: part 2

2.1 From <*>pply towards for

In the Week 2 exercises you created a list object:

```
set.seed(20181005)
my_list <- replicate(10, list(rnorm(5)))
```

This object replicated the `list(rnorm(5))` command 10 times, and put the results into a list (of length 10). On each of these ten entries we calculated the mean of the 5 random normal draws, this was done by the code:

```
my_list_means <- lapply(my_list, mean)
```

a

Create a similar `my_list` object, but instead of using the `replicate` function, use a `for` loop. Do not forget to first create a list of length 10, using for example:

`my_list <- vector(length = 10, mode = "list")`

Due to the fact that you're sampling from a normal distribution, the values in your list will be different from the list of the Week 2 exercises.

Answer:

```
for (i in 1:10) {
  my_list[[i]] <- rnorm(5)
}
```

b

We can time the code from lecture 2 as follows:

```
system.time(replicate(1e4, {
  set.seed(20181005)
  my_list <-  replicate(10, list(rnorm(5)))
}))
```

```
##    user  system elapsed
##   0.500   0.023   0.524
```

Which is of the two code chunks is faster? the code with the `for` loop or the code from the exercise of lecture 2?

Answer:

```
system.time(replicate(1e4, {
  my_list <- vector(length = 10, mode = "list")
  for (i in 1:length(my_list)) {
    my_list[[i]] <- rnorm(5)
  }
  }))
```

```
##    user  system elapsed
##   0.224   0.029   0.254
```

The for loop is faster.

c

We assign the mean of each entry in `my_list` towards the object `my_means` as follows:

```
my_means1 <- lapply(my_list, mean)
```

Create a `for` loop with which we can create the same `my_means2` object. Can you conclude which of the two ways of coding is faster, is it the code with the `lapply` function, or your code with the `for` loop?

Answer:

```
my_means2 <- vector(mode = "list", length = length(my_list))
for (i in 1:length(my_list)) {
  my_means2[[i]] <- mean(my_list[[i]])
}
all.equal(my_means2, my_means1)
```

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

The computing time needed with the `for` loop:

```
system.time(replicate(1e4, {
  my_means2 <- vector(mode = "list", length = length(my_list))
  for (i in 1:length(my_list)) {
    my_means2[[i]] <- mean(my_list[[i]])
  }
}))
```

```
##    user  system elapsed
##   0.251   0.001   0.252
```

The computing time needed with the `lapply` function:

```
system.time(replicate(1e4, {
  my_means1 <- lapply(my_list, mean)
}))
```

```
##    user  system elapsed
##   0.283   0.002   0.285
```

The difference is very hard to detect. The code with `lapply()` function seems to be slightly faster...

d

Quicker code would be to combine to calculate the mean direcly on the five samples generated from the standard normal distribution.

The `replicate()` and `lapply()` together would give us

```
system.time(replicate(1e4, {
  set.seed(20181005)
  my_list <- replicate(10, list(rnorm(5)))
  my_means <- lapply(my_list, mean)
}))
```

```
##    user  system elapsed
##   0.818   0.012   0.833
```

whereas using `lapply()` only, gives us:

```
system.time(replicate(1e4, {
  set.seed(20181005)
  my_means <- lapply(1:10, function(b) {
    mean(rnorm(5))
  })
}))
```

```
##    user  system elapsed
##   0.647   0.020   0.671
```

Can you rewrite the second `R` code chunk into a `for` loop? That is, `system.time()` a replication of the `for loop` for `1e3` times.

Answer:

```
system.time(replicate(1e3, {
  set.seed(20181005)
  my_means <- vector(length = 10, mode = "list")
  for (i in 1:10) {
    my_means[[i]] <- mean(rnorm(5))
  }
}))
```

```
##    user  system elapsed
##   0.051   0.000   0.051
```

It is interestingly to see that the `for` loop seems to the "fastest" here.

When using an anonymous function for the `FUN` argument in `lapply()`, then, the more complex this function becomes, we see that a "correctly applied" explicit loop (such as the for loop) has a speed gain over `lapply()`

2.3 An improvement of the 99 Bottles of Beer song?

Step-by step we are going to improve the `SingTheBottleSong()` function in this exercise (see Exercises part 1).

Remember, the main function inside `SingTheBottleSong()` was:

```
CreateParagraphsBottlesBeer <- function(bttls) {
    paragraph <- paste0(
      bttls, " bottles of beer on the wall, ",
      bttls, " bottles of beer. \n",
      "Take one down, pass it around, ",
      bttls - 1, " bottles of beer on the wall...\n\n"
    )
    # cat(paragraph)
    return(paragraph)
  }
```

for the complete lyrics, take a look at:

http://99-bottles-of-beer.net/lyrics.html

a.

The first improvement of the function `CreateParagraphsBottlesBeer()` is to make sure that `bttls` can be coerced to an integer using:

`bttls <- suppressWarnings(as.integer(bttls))`

Take a look at the help file of the functions `stop()` and `supressWarnings()`.

If bttls is not coerced to an integer, it probably gives `NA`. If this is the case, then use `stop()` to exit the function by leaving an appropriate message. Hint: Look at the help-file or internet for examples on how to use `stop()` in a function.

Answer:

Add the following two lines in the beginning of the body of the `CreateParagraphsBottlesBeer()` function:

```
bttls <- suppressWarnings(as.integer(bttls))
if(is.na(bttls)) stop("bttls should be able to be coerced to integer")
```

b.

Note that `CreateParagraphsBottlesBeer()` does not make sense for `bttls = 2` or `bttls = 1`. Use an

`if(cond) cons1.expr  else if cons2.expr else alt.expr`

expression to create the three needed different paragraphs inside the updated function.

Answer:

replace the `paragraph <- paste0(___)` inside the body of the function with the following lines of code:

```r
if(bttls >= 3) {
  paragraph <- paste0(
    bttls, " bottles of beer on the wall, ",
    bttls, " bottles of beer. \n",
    "Take one down, pass it around, ",
    bttls - 1, " bottles of beer on the wall...\n\n"
  )
} else if (bttls==2) {
  paragraph <- paste0(
    bttls, " bottles of beer on the wall, ",
    bttls, " bottles of beer. \n",
    "Take one down, pass it around, ",
    bttls - 1, " bottle of beer on the wall...\n\n"
  )
} else (bttls == 1) {
  paragraph <- paste0(
    bttls, " bottle of beer on the wall, ",
    bttls, " bottle of beer. \n",
    "Take one down, pass it around, ",
    "no more bottles of beer on the wall.\n\n"
  )
}
```

c.

Also, use `stop()` with an appropriate message to exit the function when `bttls` has a value `< 1`.

Answer:

Adding

`if(bttls < 1) stop("bttls should be >= 1")`

inside the function gives us the following updated function:

```r
CreateParagraphsBottlesBeer <- function(bttls) {

  bttls <- suppressWarnings(as.integer(bttls))
  if (is.na(bttls)) stop("bttls should be able to be coerced to integer")
  if (bttls < 1) stop("bttls should be >= 1")

  if (bttls >= 3) {
    paragraph <- paste0(
      bttls, " bottles of beer on the wall, ",
      bttls, " bottles of beer. \n",
      "Take one down, pass it around, ",
      bttls - 1, " bottles of beer on the wall...\n\n"
    )
  } else if (bttls == 2) {
    paragraph <- paste0(
      bttls, " bottles of beer on the wall, ",
      bttls, " bottles of beer. \n",
      "Take one down, pass it around, ",
      bttls - 1, " bottle of beer on the wall...\n\n"
    )
```

```r
  } else {
    paragraph <- paste0(
      bttls, " bottle of beer on the wall, ",
      bttls, " bottle of beer. \n",
      "Take one down, pass it around, ",
      "no more bottles of beer on the wall.\n\n"
    )
  }
  return(paragraph)
}
```

The function of the full song now becomes:

```r
SingTheBottleSong <- function(
  bottle_nr,
  address
) {

  each_bottle_nr <- bottle_nr:1 # notice the change here!!

  header_song <- paste0("\n", bottle_nr, " Bottles of Beers \n\n")

  body_song <- unlist(lapply(each_bottle_nr, CreateParagraphsBottlesBeer))

  lyrics <- c(
    header_song,
    body_song
  )

  cat(lyrics, sep = "", file = address)
  return(invisible(NULL))
}
SingTheBottleSong(
  bottle_nr = 5,
  address = "0_data/my_bottlesong_lyrics.txt"
)
```

2.4 Functions and `if else` statements | About your SCR grade

a.

Look at the help-file of the function `round`, can you spot in the help-file where it is written that a number of 8.5 would be rounded to an 8, instead of a 9? What does it say?

Answer:

See the text "Note that for rounding off a 5, the IEC 60559 standard is expected to be used, go to the even digit."

b.

Complete the function `MakeRound` on the underscores `___`, such that it rounds the number 8.5 to a 9.

```
MakeRound <- function(x) {
  tmp <- trunc(x)
  if(___) {
    out <- ceiling(x)
  } else {
    out <- floor(x)
  }
  return(out)
}
MakeRound(8.5)
```

Answer:

```
MakeRound <- function(x) {
  tmp <- trunc(x)
  if (x - tmp >= 0.5) {
    out <- ceiling(x)
  } else {
    out <- floor(x)
  }
  return(out)
}
MakeRound(8.5)
```

```
## [1] 9
```

c.

Final grades for the courses in the Statistical Science programme are rounded to whole and half numbers except the grade 5.5 (e.g. .., 4.5, 5, 6, 6.5, 7, ..). Modify the `MakeRound` function such that it rounds towards whole and half numbers. Show that your modified function rounds 8.25 towards a 8.5 and not towards `round(8.25*2)/2`. The 5.49, however, should round towards 5, and a 5.5 towards 6.

You may want to use the following pogramming structure inside the body of your function:

```
if(x >= 5 && x < 5.5) {
  ---
} else if(x >= 5.5 &  x < 5.75) {
  ---
} else if(___) {
  ---
} else {
  ---
}
```

Answer:

```
MakeRound <- function(x) {
  tmp <- trunc(2*x)
  if (x >= 5 && x < 5.5) {
    out <- 5
  } else if (x >= 5.5 &  x < 5.75) {
```

```
    out <- 6
  } else if (2*x - tmp >= 0.5) {
    out <- ceiling(2*x)/2
  } else {
    out <- floor(2*x)/2
  }
  return(out)
}
MakeRound(5.49)
```

```
## [1] 5
```

```
MakeRound(5.50)
```

```
## [1] 6
```

```
MakeRound(8.25)
```

```
## [1] 8.5
```

d.

Use your last `MakeRound` function to create a modified version of the lecture's `CalculateGrade` function such that it rounds the final grade towards a whole or half number. Here's the

```
CalculateGrade <- function(A, E1, E2) {
  A_comp <- A * (1 / 3)
  E_comp <- mean(c(max(E1, E2), E2)) * (2 / 3)
  grade <- A_comp + E_comp
  return(grade = grade)
}
```

Answer:

```
CalculateGrade <- function(A, E1, E2) {
  A_comp <- A * (1 / 3)
  E_comp <- mean(c(max(E1, E2), E2)) * (2 / 3)

  MakeRound <- function(x) {
    tmp <- trunc(2*x)
    if (x >= 5 && x < 5.5) {
      out <- 5
    } else if (x >= 5.5 &  x < 5.75) {
      out <- 6
    } else if (2*x - tmp >= 0.5) {
      out <- ceiling(2*x)/2
    } else {
      out <- floor(2*x)/2
    }
    return(out)
```

```
  }

  grade <- MakeRound(x = A_comp + E_comp)
  return(grade = grade)
}
```

2.5 SingTheBottleSong2() based on for

a

Rewrite `SingTheBottleSong()` into the function `SingTheBottleSong2()` that uses the `for` loop instead of the implicit `lapply()`.

Answer:

```
SingTheBottleSong2 <- function(
  bottle_nr,
  address
) {

  header_song <- paste0("\n", bottle_nr, " Bottles of Beers \n\n")

  body_song <- vector(length(bottle_nr), mode = "list")

  each_bottle_nr <- bottle_nr:1
  for (bttls in each_bottle_nr) {
    body_song[[bottle_nr - bttls + 1]] <- CreateParagraphsBottlesBeer(bttls)
  }
  body_song <- unlist(body_song)

  lyrics <- c(
    header_song,
    body_song
  )

  cat(lyrics, sep = "", file = address)
  return(invisible(NULL))
}
```

b

Use `system.time()` to measure the time of both functions. Can you choose a number s.t. a distinction in time evaluation become visible?

Note that for the model answers we could not easily find such a distinction. The timing is quite instable. Around 1e5 bottles of beer the `lapply()` based function seems to be faster.

Answer:

```
lyrics_addr <- "0_data/my_bottlesong_lyrics.txt"
if (file.exists(lyrics_addr)) file.remove(lyrics_addr)
```

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

```
system.time(
  SingTheBottleSong(
    bottle_nr = 1e5,
    address = lyrics_addr
  )
)
```

```
##    user  system elapsed
##   1.413   0.303   1.745
```

```
file.remove(lyrics_addr)
```

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

```
system.time(
  SingTheBottleSong2(
    bottle_nr = 1e5,
    address = lyrics_addr
  )
)
```

```
##    user  system elapsed
##   1.479   0.299   1.807
```

```
file.remove(lyrics_addr)
```

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

Now, it seemes that the using the implicit loop with `lapply` and `unlist()`

When using a pre-definied function as `FUN` argument in `lapply()`, most of the times you have a speed gain over `for` or any other types of explicit loops in `R`.

2.6 Bubble sort

Bubble sort (<– click on Bubble sort) is a sorting algorithm that works by stepping through a vector to be sorted, comparing each pair of adjacent elements and swapping them if they are in the wrong order. The walk through the vector is repeated until no swaps are needed. The algrithm sorts a vector in ascending order.

Write a function `BubbleSort()` that implements the Bubble sort algorithm, and show that it can sort a vector (e.g. `sample(50)`) from its smallest value to its highest value.

Tip: it is common to see the following ERROR when swapping values, at least once:

```
x[2] <- x[3]
x[3] <- x[2]
```

Check for yourself why the above swap does not work.

Here's a function template to help you get started!

```r
BubbleSort <- function(x, na.rm = TRUE) {
    # Description of function..
    #
    # Args:
    #   arg1: Description.
    #   arg2: Description + default
    # Returns:
    #   x sorted from smallest to highest values
    #
    # perfrom some checks here...

    n <- length(x)

    repeat {
        swaps <- 0
        for (i in 1:(n - 1)) {

            ......


        }
        if (!swaps)
            break
    }
    return(x)
}
```

Answer:

```r
BubbleSort <- function(x, na.rm = TRUE) {
    # Description of function..
    #
    # Args:
    #   arg1: Description.
    #   arg2: Description + default
    # Returns:
    #   x sorted from smallest to highest values
    #
    # perfrom some checks here...

    n <- length(x)

    repeat {
        swaps <- 0
        for (i in 1:(n - 1)) {

          ###
          # solution part:
          if (x[i] > x[i + 1]) {
            x[c(i, i + 1)] <- x[c(i + 1, i)]
            swaps <- T
          }
          ###
```

```
        }
        if (!swaps)
            break
    }
    return(x)
}

BubbleSort(sample(50))
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
## [24] 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
## [47] 47 48 49 50
```

2.7 Coding a `for` loop into `while` or `repeat`

Take again (a quick) look at the following `for` loop example:

```
grades <- trunc(runif(100, 10, 100))/10 # uniform grades
n <- length(grades)
grd_rnd <- numeric(n)
grd_tbl <- numeric(10)
names(grd_tbl) <- 1:10

for(i in 1:n) {
  grade <- grades[i]
  tmp <- trunc(grade)
  if(grade - tmp >= 0.5) {
    grd_rnd[i] <- ceiling(grade)
  } else {
    grd_rnd[i] <- floor(grade)
  }
  grd_tbl[grd_rnd[i]] <- grd_tbl[grd_rnd[i]] + 1
}
grd_tbl
```

Recode this example into a `while` or `repeat` loop.

Answer:

```
grades <- trunc(runif(100, 10, 100))/10 # uniform grades
n <- length(grades)
grd_rnd <- numeric(n)
grd_tbl <- numeric(10)
names(grd_tbl) <- 1:10
i <- 1
while (i <= n) {
  grade <- grades[i]
  tmp <- trunc(grade)
  if (grade - tmp >= 0.5) {
    grd_rnd[i] <- ceiling(grade)
  } else {
    grd_rnd[i] <- floor(grade)
```

```
  }
  grd_tbl[grd_rnd[i]] <- grd_tbl[grd_rnd[i]] + 1
  # print(i)
  i <- i + 1
}
```

Selfstudy Exercises

3.1 Vectorization: All roads lead to Rome... some are more efficient.

a.

Rewrite the following code into code which uses a `[*]pply()` function, and into a vectorized solution.

```
a <- numeric(5)
for (i in 1:5) a[i] <- i + 3
rm(i)
a
```

```
## [1] 4 5 6 7 8
```

Answer:

```
a <- sapply(1:5, function(i) i ) + 3
a
```

```
## [1] 4 5 6 7 8
```

```
a <- 1:5 + 3
a
```

```
## [1] 4 5 6 7 8
```

b.

The following code creates 2 vectors of length 100, with i.i.d. random values from a standard normal distribution (`x`) and from a normal distribution with `mean = 2`, and `sd = 3`.

inefficient code:

```
x <- y <- NULL
for (i in 1:100) {
    x[i] <- rnorm(1)
    y[i] <- rnorm(1, 2, 3)
}
```

Although you probably feel that it is not a wise decision to use a loop for this coding problem, try to recode this `for (i in 1:100) {}` without a growing object, And delete i at the end with rm() as in the lecture notes.

Answer:

```
x <- y <- numeric(100)
for (i in 1:100) {
    x[i] <- rnorm(1)
    y[i] <- rnorm(1, 2, 3)
}
rm(i)
```

c.

Rewrite the code of exercise b twice by using one of the `[*]apply` functions (which can be tricky!) AND using a vectorized solution.

Answer using `sapply`:

```
x <- sapply(1:100, function(elem) rnorm(1))
y <- sapply(1:100, function(elem) rnorm(1))*3 + 2
```

Answer using a vectorized solution:

```
x <- rnorm(100)
y <- rnorm(100)*3 + 2
```

d

Take a look at the help file of the vectorized `pmin` function. We can use it to find the parallel minima of the elements of the objects `x` and `y`. See the following code:

```
head(pmin(x,y))
```

```
## [1]  0.03586826  0.05496134 -1.28159602 -0.18110238 -1.14483591 -1.45179128
```

Now try to rewrite this code example into your most cumbersome code possible by using the `for` loop and `if() else {}` control statements.

Answer (that will lower your grade if you would use it at your exam or assignments):

```
pminoutresult <- NULL # ugly long name
for (i in 1:100) {
  tmpx <- x[i] # unnecessary storage of object
  tmpy <- y[i] # unnecessary storage of object
  if ( tmpx < tmpy) {
    pminoutresult[i] <- tmpx # unnecessary assigning
  } else {
    pminoutresult[i] <- tmpy # unnecessary assigning
  }
  rm(list = c("tmpx", "tmpy")) # nice, but would not have been needed
}
rm(i) # not necessary if nowhere else used
all(pminoutresult == pmin(x,y))
```

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

A better example using the for loop that would not have lost you points:

```
pmout <- numeric(100)
for (i in 1:100) {
  pmout[i] <- if (x[i] < y[i]) {
    x[i]
```

```r
  } else {
    y[i]
  }
}
rm(i) # not necessary if nowhere else used
all(pmout == pmin(x,y))
```

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

Without the if-else statement:

```r
pmout <- numeric(100)
for (i in 1:length(pmout)) {
  pmout[i] <- c(x[i], y[i])[(x[i] > y[i]) + 1]
}
rm(i) # not necessary if nowhere else used
all(pmout == pmin(x,y))
```

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

e.

We want to create the matrix with ij'th element equal to 3 times the ith element of 'x' minus 2 times the square of the jth element of 'y'

inefficient code:

```r
system.time(replicate(1e2, {
  rslt1e <- matrix(0, 100, 100)
  for (i in 1:100) {
    for (j in 1:100) {
        rslt1e[i, j] <- 3 * x[i] - 2 * y[j]^2
    }
  }
}))
```

```
##    user  system elapsed
##   0.206   0.002   0.209
```

Carefully study what happens in the above code. Could you rewrite it using one (or even two) `sapply` implicit loop(s)? If yes which of the pieces of code is faster? Use the function `system.time` (and possibly `replicate`) to compare the codes on speed (check the help-file and internet, for example: http://stats.stackexchange.com/questions/3235/timing-functions-in-r).

Answer using two sapply loops:

```r
system.time(replicate(1e2, {
  rslt1e <- sapply(y, function(y_elem) {
    sapply(x, function(x_elem) {
      3 * x_elem - 2 * y_elem^2
    })
  })
}))
```

```
##    user  system elapsed
##   0.928   0.010   0.945
```

Answer using one sapply loop:

```
system.time(replicate(1e3, {
  rslt1e <- sapply(y, function(y_elem) {
      3*x - 2*y_elem ^ 2
  })
}))
```

sapply + vectorization calculation

```
##    user  system elapsed
##   0.416   0.109   0.527
```

f.

Check the help file of the function `outer()`. Run the examples and get comfortable with the functions `example(outer)`. If the examples do not make you feel comfortable on what the function does, explore the world wide web for more examples.

When comfortable with `outer`, rewrite the code problem 1e. into a vectorized solution by using the function `outer`

Answer:

```
rslt1e <- outer(x, y, function(x_elem, y_elem) {
  3*x_elem - 2*y_elem ^ 2
})
```

g.

Rewrite the following code into a vectorized solution that uses only the `set.seed(42)` and the function `sample`.

```
set.seed(42)
a1 <- replicate(12, sample(c(0,1), 1))
```

Answer:

```
set.seed(42)
a2 <- sample(c(0,1), 12, replace = TRUE)
all.equal(a1,a2)
```

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

h.

Check the helpfile of `ifelse()`. Could you rewrite the lines that start from `obtained_ects` in the following code chunk into only one line of code using the `ifelse()` function?

Code to be rewritten:

```r
set.seed(20181005)
ideal_grades <- rnorm(42, mean = 7, sd = 1.5)
obtained_ects <- character(length(ideal_grades))
for (student in 1:length(ideal_grades)) {
  if (ideal_grades[student] > 5.5)  {
    obtained_ects[student] <- "yes"
  } else {
    obtained_ects[student] <- "no"
  }
}
```

Answer:

```r
set.seed(20181005)
ideal_grades <- rnorm(42, 7, sd = 1.5)
obtained_ects <- ifelse(ideal_grades > 5.5, "yes", "no")
```

i.

Take a look at the source of `ifelse()` function. Although it is vectorized, it still uses an `if` statements, i.e.

```r
if (any(test[ok]))
        ans[test & ok] <- rep(yes, length.out = length(ans))[test & ok]
```

Instead of using the `ifelse()` function, or `if` statments, only `[` brackets and filtering to create `obtained_ects`. Given that you have `ideal_grades` already in your workspace, you should be able to do so in two lines.

Answer:

```r
obtained_ects <- rep(FALSE, length(ideal_grades))
obtained_ects[ideal_grades > 5.5] <- TRUE
```

j.

Could yo create a function that can replace `SingTheBottlesSong()` and `SingTheBottlesSong2()`, and does not use any implicit or explicit loops.

Hint: Check the live coding and the paste function

Answer:

The following lines of code should bring you towards an answers:

```r
  bttls <- suppressWarnings(as.integer(bttls))
  if(is.na(bttls)) stop("bttls should be able to be coerced to integer")
  if(bttls < 1) stop("bttls should be >= 1")

  if(bttls > 2) {
    paragraphs_plural <- paste0(
      bttls:3, " bottles of beer on the wall, ",
      bttls:3, " bottles of beer. \n",
```

```
      "Take one down, pass it around, ",
      (bttls - 1):2, " bottles of beer on the wall...\n\n"
    )
  } else {
    paragraphs_plural <- NULL
  }
  if(bttls > 1) {
    paragraph_2bttls <- paste0(
      2, " bottles of beer on the wall, ",
      2, " bottles of beer. \n",
      "Take one down, pass it around, ",
      1, " bottle of beer on the wall...\n\n"
    )
  }
  paragraph_last <- paste0(
    1, " bottle of beer on the wall, ",
    1, " bottle of beer. \n",
    "Take one down, pass it around, ",
    "no more bottles of beer on the wall...\n\n"
  )
  body_song <- c(paragraphs_plural, paragraph_2bttls, paragraph_last)
```

如果个数不一样，会循环利用

3.3 A Real (Non-Statistical) Programming Assignment

In this exercise we will make a `data.frame` containing functions from the package `stats`

a

Make a character vector called `functs` that contains all the objects from the package `stats` using the code `ls(envir = as.environment("package:stats"))`. Take a careful look at what each individual part does (run each part separately). Try to think/guess why this works.

Answer:

```
functs <- ls(envir = as.environment("package:stats"))
```

b

Loop over the vector `functs` and use the function `get` to check for each of the objects from package `stats` if it is of the class `function` or not. Save only the function objects.

Answer:

which 不需要

```
index.to.keep <- which(sapply(functs, function(i) class(get(i))) == "function")
functs <- functs[index.to.keep]
```

get: 通过名字来获得

c

We are now going to create a look-up table. We are going to make a data.frame that one can use to see which functions use a particular argument, and which arguments a particular function uses. We are going to create an entry for *each* function, and denote with `TRUE` or `FALSE` whether an argument is used in that particular function or not.

Extract from the functions, whose names you've collected in `functs`, the arguments. Create an empty `data.frame` called `stats.fncts` and add an entry for each function in the `stats` package: the first column will contain the name, the following columns (as many columns as there are (unique) arguments in the `stats` package) will contain a `TRUE` or `FALSE` depending on whether the argument is used by the function or not.

Example result (using only two functions):

```
data.frame(
  fnames = c("anova", "aov"),
  contrasts = c(F, T),
  data = c(F, T),
  formula = c(F, T),
  object = c(T, F),
  projections = c(F, T),
  qr = c(F, T)
)
```

```
##   fnames contrasts  data formula object projections    qr
## 1  anova     FALSE FALSE   FALSE   TRUE       FALSE FALSE
## 2    aov      TRUE  TRUE    TRUE  FALSE        TRUE  TRUE
```

Hint: as always, divide the task up in several smaller pieces. What do we need to do this task? We need all the functions from the `stats` *package, we already have those in* `functs`*! We need to extract from each function its arguments (go to the slides to see how to do this!). We don't want a look-up table with duplicate entries, so we need a way to get all the* `unique`*(!) entries!*

Answer:

```
# First get all the formals of all the functions, then use unique to get only the unique ones, then sor
stat.args <- sort(unique(names(unlist(lapply(functs, formals)))))

#First create the entire sized data.frame, to avoid growing an object:
stat.mat <- matrix(F, ncol = length(stat.args), nrow = length(functs))
#using a for loop check for each function, which arguments in the entire list of arguments, match argum
for (i in 1:length(functs)) {
  stat.mat[i, ] <- stat.args %in% names(formals(functs[i]))
}

#collect the function names and the other information into a single data.frame
stat.fncts <- data.frame(
  fnames = functs,
  stat.mat
)

#give nice names
colnames(stat.fncts) <- c("fnames", stat.args)
#str(stat.fncts)
```

d

There are functions that have the exact same arguments. How big is the largest group of functions that share the exact same arguments?

Hint: work out this exercise first on the first 10 rows of the datafram stat.fncts to see how to make the code for the big data set. Use a (statistical) programmer's strategy. . .

Answer:

```
# crude solution that takes ages..: just check for every column, how many columns are exactly the same!
similarity <- numeric(nrow(stat.mat))
for (j in 1:nrow(stat.mat)) { # use -1 to exclude fname
  for (k in 1:nrow(stat.mat)) {
    if (all(stat.mat[j,] == stat.mat[k,])) {
      similarity[j] <- similarity[j] + 1
    }
  }
}
max(similarity)
```

```
## [1] 41
```

```
#A more refined solution:
to01 <- apply(X = stat.mat, MARGIN = 1, FUN = function(x){
  x <- as.numeric(x)
  paste(x, collapse = "")
})
```

每行，把10拼接

得到频数最高的字符串

哪些是一样的字符串？

```
indx <- names(which.max(table(to01))) == to01
sum(indx)# the number of functions with the same arguments
```

```
## [1] 41
```

```
grp_max <- stat.fncts[indx, ]
grp_max$fnames
```

```
##  [1] aggregate     ansari.test   ar.burg       ar.yw         as.hclust
##  [6] as.stepfun    as.ts         bartlett.test biplot        cor.test
## [11] cycle         deltat        density       diffinv       end
## [16] fligner.test  formula       frequency     ftable        getCall
## [21] kernapply     kruskal.test  lag           loadings      model.tables
## [26] monthplot     mood.test     naprint       ppr           prcomp
## [31] princomp      quantile      reorder       screeplot     start
## [36] t.test        terms         time          var.test      wilcox.test
## [41] window
## 437 Levels: acf acf2AR add.scope add1 addmargins ... xtabs
```

3.4 Gradient Descent

There are various methods of estimating the parameters of a statistical model e.g. least squares or maximum likelihood estimation. In maximum likelihood estimation a maximum of a real function (the likelihood function) needs to be found by means of optimization. Various optimization methods exist for finding the minimum or maximum of a real function.

Gradient descent is an iterative algrotihm that can be used to find a local minimum of a function. If a function f(x) is defined and differentiable in a point x_0, then $f(x)$ decreases fastest if one goes from x_0 in the direction of the negative gradient of f at x_0, denote - $f'(x_0)$. It follows that, if $x_1 = x_0 - \alpha f'(x_0)$ for small enough α, then $f(x_0) \geq f(x_1)$. A minimum of $f(x)$ can be found by starting with x_0 and considering the sequence $x_0, x_1, x_2, \ldots$ such that

$$x_{n+1} = x_n - \alpha f(x_n).$$

Write a function that implements the gradient descent algorithm for the polynomial

$$f(x) = x^2 - 3x + 2$$

using `repeat` and $\alpha = 0.01$. Use an auxiliary function for evaluating the first derivative $f'(x)$. The `while` loop stops when the absolute difference between x_{n+1} and x_n is smaller then `1/1e4`. Demonstrate the algorithm using the start values $x_0 = -2$ and $x_0 = 3$. Report the number of iterations and the x-value for which $f(x)$ is minimal.

Answer:

```
dvtv <- function(x){
  return(2*x - 3)
}

gradD <- function(x.init, alpha = 0.01, conv = 1/1e4){
  # x0 = , alpha = 0.01, conv = 1/1e4
  x <- x.init; it <- 0
  repeat {
    x <- x.init - alpha*dvtv(x.init)
    it <- it + 1
    #cat("cost: ", x^2 - 3*x + 2, " it:", it, "\n")
    if ( abs(x.init - x) < conv) break
    x.init <- x
  }
  list(x = x, it = it)
}

gradD(x.init = +3)
```

```
## $x
## [1] 1.504834
##
## $it
## [1] 284
```

```
gradD(x.init = -2)
```

```
## $x
## [1] 1.495172
##
## $it
## [1] 326
```