Examination Advanced R Programming

Linköpings Universitet, IDA, Statistik

Course code and name:	732A94 Advanced R Programming
Date:	2018/02/28, 8–12
Teacher:	Krzysztof Bartoszek
Allowed aids:	The extra material is included in the zip file **exam_material.zip**
Grades:	A= $[19-20]$ points B= $[17-19)$ points C= $[12-17)$ points D= $[10-12)$ points E= $[8-10)$ points F= $[0-8)$ points
Instructions:	Write your answers in an R script file named **[your exam account].R** The R code should be complete and readable code, possible to run by copying directly into a script. Comment directly in the code whenever something needs to be explained or discussed. Follow the instructions carefully. There are **THREE** problems (with sub–questions) to solve.

Problem 1 (5p)

a) (2p) Both `library()` and `require()` are used to load packages. Explain the difference between them.
b) (2p) When writing your own package it will nearly always depend on functions from other packages. Is it a good idea to use either `library()` or `require()` to load these packages inside your package? Why or why not? How should your package load the necessary dependent packages?
c) (1p) You need to use two functions that are provided by two different packages but they have the same name, i.e. function `f1()` from package `pkg1` and function `f1()` from package `pkg2`. How do you solve this problem?

Problem 2 (10p)

READ THE WHOLE QUESTION BEFORE STARTING TO IMPLEMENT! Remember that your functions should **ALWAYS** check for correctness of user input!

a) (3p) In this task you should use object oriented programming in S3 or RC to write code that stores and manipulates information on the contents of a refrigerator. The first task is to implement a function called `create_fridge()` that returns a fridge object. The `create_fridge()` function should take two arguments: `fridge_content` and `fridge_volume`. The object shall store information on the fridge's content in the form of a data frame with three variables (columns). The first one is a unique ID of each product, the second the amount of the product and the third the amount of space a unit (i.e. `amount` equalling `1`) of the product occupies. So for example if for some product `amount` is `4` and `unit_space` equals `2`, then the amount of volume this product occupies is `8`. Furthermore, the fridge object has to know how much free space there is in the fridge.

```
## S3 and RC call to create_fridge() function
> dfFC<- data.frame(product_id=c("milk","eggs","juice","cheese","ham"),
amount=c(2,20,4,1,0.5),unit_space=c(1,0.05,1,0.25,0.25))
> fridge_1 <- create_fridge(fridge_content=dfFC,fridge_volume=100)
```

b) (6p) Now implement two functions called `add_to_fridge()` and `take_from_fridge()`. They are, as the names suggest, there to add new contents to the fridge or take products out.

The `add_to_fridge()` function has to have as its argument a data frame with three variables (columns). The first one is a unique ID of each product, the second the amount of the product and the third the amount of space a unit (i.e. amount equalling 1) of the product occupies. When adding a product you should check by its ID if it is already in the fridge. If it is, then a new row should not be created but the old one should be updated. Your function should check in such a situation whether the `unit_space` values agree. **YOU** decide how to react when they do not. In your code write a comment that explains why you choose a particular reaction **(EVEN IF YOU DID NOT MANAGE TO IMPLEMENT IT!)**. Your function should be able to handle new products, that are not present in the fridge.

The `take_from_fridge()` function has to have as its argument a data frame with two variables (columns). The first one is the unique ID of each product and the second the amount of the product. **YOU** decide how to react when all of a given product is taken. In your code write a comment that explains why you choose a particular reaction **(EVEN IF YOU DID NOT MANAGE TO IMPLEMENT IT!)**.

Remember that you should react appropriately if someone attempts to take from the fridge a product that is not there, more than there is of a given product or store more products that all together take up more space than there is in the fridge. It is up to you to decide how to react in such a situation (but of course you cannot store more than there is space and you cannot give out of the fridge what is not there).

Provide some example calls to your code.

```
# S3 and RC call for fridge usage
fridge_1 <- add_to_fridge(fridge_1,data.frame(product_id=c("milk","ham","yogurt"),
amount=c(1,0.5,1),unit_space=c(1,0.25,1)))
fridge_1 <- take_from_fridge(fridge_1,data.frame(product_id=c("milk","eggs"),
amount=c(0.5,3)))

# It is also OK to use the following OO style if using RC
# fridge_1$add_to_fridge(data.frame(product_id=c("milk","ham","yogurt"),
amount=c(1,0.5,1),unit_space=c(1,0.25,1)))
# fridge_1$take_from_fridge(data.frame(product_id=c("milk","eggs"),amount=c(0.5,3)))
```

c) (1p) Implement a plot **OR (NO NEED TO DO BOTH!)** print function for your fridge objects that informs the user of the fridge's content. You are free to choose yourself how to report the content!

```
# Plotting call
plot(fridge_1)
# Printing call
print(fridge_1)
```

Problem 3 (5p)

a) (3p) Often one has a sorted numerical vectors and one wants to add a new number in the correct place. Your task is to write a function, called `add_element_to_sorted_vector()` that takes as its input a sorted in ascending order vector and a number. The output is a sorted in ascending order vector, composed of the input's vector and number. Your implementation has to be done using a loop and you may not use `sort()`, `which()` or similar functions. The function should check for correctness of the input and react appropriately.

```
> v<-c(1:4,6)
> x<-5
> add_element_to_sorted_vector(v,x)
[1] 1 2 3 4 5 6
```

b) (1p) What is the complexity of your solution in terms of the input vector's dimension?
c) (1p) The same can be achieved using the following R code.

```
> v<-c(1:4,6)
> x<-5
> sort(c(x,v))
[1] 1 2 3 4 5 6
```

Implement a unit test that compares your implementation with the above code.