Examination Advanced R Programming

Linköpings Universitet, IDA, Statistik

Course code and name:	732A94 Advanced R Programming
Date:	2019/10/24, 8–12
Teacher:	Krzysztof Bartoszek
Jour:	Héctor Rodriguez Déniz
Allowed aids:	The extra material is included in the zip file **exam_material.zip**
Grades:	A= $[18-20]$ points B= $[16-18)$ points C= $[14-16)$ points D= $[12-14)$ points E= $[10-12)$ points F= $[0-10)$ 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) (3p) Briefly discuss, in a few sentences, some good coding practices like writing style, commenting, variable, function naming e.t.c.
b) (2p) What information should be provided in a function's documentation?

Problem 2 (10p)

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

a) (4p) In this task you should use object oriented programming in S3 or RC to write code that simulates a futuristic refrigerator. The future intelligent fridge is directly connected through a system of pipes with grocery warehouses. Hence, food can be provided on demand and replenished immediately. However, each fridge has also inbuilt the permissible products and optimal diet for each user, so that by chance one cannot eat too much of a given product nor eat a not recommended combination of products. After a user identifies themselves and requests a given product from the fridge, the fridge first checks whether they are permitted to consume this. If verification is positive, then the given product is dispensed, otherwise the user is informed that they are not allowed to consume the product but are welcome to make another choice. Your goal is to first construct an initial fridge. The `create_fridge()` function should take one argument, `number_users`, the number of users in the household. The fridge object should contain for each user a data frame/list/matrix (your choice) with permissible food items. This object should store information for each food item. Information that needs to be remembered on a food item (for each fridge user) is:

1. food item id (positive integer, has to be unique, **control for this**),
2. permissible number of this food item that can be eaten each day
3. ids of food items that cannot be eaten together with this food item (if it helps you may assume that the number of such items is lesser than some number).

Initially each user may have the same permissible food items. Populate this object in some way of your choice when creating the fridge object. Furthermore, for each user you need to create a data structure that holds how much of each food item did the user take in a day (our simulation does not run longer than a day).

```
## S3 and RC call to build_diet_list() function
modern_fridge <- create_fridge(number_users=2)
```

b) (2p) From time to time diet recommendations change. Now implement a function called `update_food_list()` that takes as parameters

1. `user_id`: for whom to update the permissible food list,
2. `food_id`: what food item is concerned,
3. `update_max`: the new maximum daily amount, does not need to be passed when the function is called—then no update,
4. `update_conflicting_food`: update on the conflicting foods, a vector of integers, a positive value indicates that the food item with the given id is to be added, a negative that removed. You decide how to handle the situation that a food item to be removed is not on the original list, or a food item to be added is not in the list of food items. This vector does not need to be passed when the function is called—no update in such a case.

Provide some example calls to your code.

```
user_id<-1
food_id<-1
update_max<-2
update_conflicting_food<-c(-2,3)

## S3 and RC call
modern_fridge <-update_food_list(modern_fridge,user_id,food_id,
update_max,update_conflicting_food)

## if using RC you may also call in this way
modern_fridge$update_food_list(user_id,food_id,new_max,conflicting_food)
```

c) (3p) Now implement a function called `request_food()` that takes as parameters

1. `user_id`: id of user requesting food item,
2. `food_ids`: vector of requested food ids.

Your function should check whether there are no conflicts inside the `food_ids` vector and also if the daily amount is not exceeded. Decide how you handle the situation that a requested food item is not on the list of food items. If the requested food may be dispensed, then the object storing the amounts eaten during the day must be updated. If the requested food may not be dispensed, then the user should be informed that the food request is denied. Provide some example calls to your code.

```
user_id<-1
vfood_ids<-1:3
## S3 and RC call
modern_fridge <-request_food(modern_fridge,user_id,food_ids)

## if using RC you may also call in this way
modern_fridge$request_food(user_id,vfood_ids)
```

d) (1p) Implement a plot **OR (NO NEED TO DO BOTH!)** print function that informs how much of each food item a user is permitted to still consume during the day. You are free to choose yourself how to report the content!

```
# Plotting and printing calls
plot(modern_fridge); print(modern_fridge)
```

Problem 3 (5p)

a) (2p) In probability, statistics and combinatorics a key value to calculate is the factorial of a natural number, $f : \mathbb{N} \to \mathbb{N}$ defined as

$$f(n) = n! = 1 \cdot 2 \cdot 3 \ldots n$$

for $n > 0$ and $f(0) := 1$.

Write your own function that takes as its input an integer and returns the value of the factorial. Do not forget that your function should check for correctness of input and react appropriately.

b) (1p) What is the complexity of your solution in terms of the number of required multiplication operations?

c) (2p) The factorial $f(n) = n!$ can be approximated, especially for large n, with Stirling's formula

$$n! \sim \sqrt{2\pi n} \left(\frac{n}{e}\right)^n.$$

Implement Stirling's formula and implement a unit test that compares your implementation of $n!$ with your Stirling's formula approximation. As Stirling's formula is a approximation you cannot expect exact equality. Hence, you should check if the two functions agree within some tolerance. You may choose the tolerance level yourself but also can use the theoretical result

$$\sqrt{2\pi} n^{n+0.5} e^{-n} \leq n! \leq e n^{n+0.5} e^{-n}.$$

You can obtain π as `pi` and e as `exp(1)`.