Examination Advanced R Programming

Linköpings Universitet, IDA, Statistik

Course code and name:	732A94 Advanced R Programming
Date:	2023/03/01, 8–12
Teacher:	Krzysztof Bartoszek phone 013–281 885
Allowed aids:	The extra material is included in the zip file **exam_help_material_732A94.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 (4p)

a) (2p) Discuss the pros and cons of cloud storage of data, third party computational services, e.t.c, i.e., working in the cloud.
b) (2p) Discuss the pros and cons of a normalized database. Who should be responsible for the normalization. What are the pros and cons each choice of responsible party?

Problem 2 (10p)

READ THE WHOLE QUESTION BEFORE STARTING TO IMPLEMENT! Remember that your functions should **ALWAYS** check for correctness of user input! For each subquestion please provide **EXAMPLE CALLS!**

a) (2p) In this task you should use object oriented programming in S3 or RC to write code that manages a compressed file archive. The archive is able to store multiple files, and a user can add or remove files from the archive. Different parts of each file can be compressed with different algorithms. The archive manager has to store information on the files it contains and with what algorithm which parts of each file is compressed. Your goal is to first initialize the archive. Depending on your chosen OO system you can do it through a constructor (RC) or by implementing a function `create_archive()` (S3). The constructing function should not take any arguments. The archive object should contain for each file the information describing it, in particular a unique id of it, a text name (you may have the same one e.g. `"a"` for every file), its size (a natural number), and a structure that stores which parts of the file are compressed by what algorithms. The possible algorithms have to be predefined in the object and cannot be changed. A part of a file is a pair of numbers, the start of the given chunk and its end. Neither of these numbers can be greater than the size of the file. The first chunk has to start at 1, the last at the size of the file. Together the chunks have to cover the whole of the file size. The archive has to store how many files are currently in it. You may assume some maximum number of files that can be stored and a maximum size of a file.

```
## example call to create an archive object
my_archive <- create_archive() # S3
my_archive <- archive$new() # RC
```

b) (4p) Now implement a function called `add_file_to_archive()` that allows one to add a new file to the archive. The function should have two parameters: the text name of the file, and its size. The id is to be automatically generated. You can randomly assign different compressing algorithms to different chunks of the file. You cannot assign a single compressing algorithm to the whole file

```
## S3 and RC example calls
my_archive <-add_file_to_archive(my_archive,"file.txt",100)
## if using RC you may also call in this way
my_archive$add_file_to_archive("file.txt",100)
```

c) (3p) Now implement a function called `remove_file()` that removes a file with a given id (not file name as these can be repeated). The function should take one parameter, the id of the file. If a file with the given id is not present, your code should react accordingly.

```
## S3 and RC example call
my_archive<-(my_archive,"file_id_1")
## if using RC you may also call in this way
my_archive$remove_file("file_id_1")
```

d) (1p) Implement a function that displays the state of the archive. You are free to choose yourself how to report the state! This function has to also work directly with `print()`.

```
# calls to show state of archive
my_archive; print(my_archive)
```

Problem 3 (6p)

a) (3p) Please implement a function that takes as its input the adjacency matrix of an unweighted, undirected graph and returns the maximum degree of the graph. Your function should be general and work for any legitimate adjacency matrix.
Do not forget to check for correctness of input. Please provide **EXAMPLE CALLS** to your function.

For a given graph an adjacency matrix is a binary matrix that has entry (i, j) equalling 1 if and only if nodes i and j are connected by an edge. Otherwise the entry is 0. Notice that this adjacency matrix has to be symmetric, i.e., if entry (i, j), then so has to entry (j, i). Each node has it *degree*, i.e., the number of nodes that it is connected to. The maximum degree of the graph is the maximum over all the degrees of the nodes. Consider the example graph in Fig. 1. It has adjacency matrix

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix}$$

and degrees of nodes 1–6: 3, 4, 2, 3, 3, 3. The maximum degree is 4.

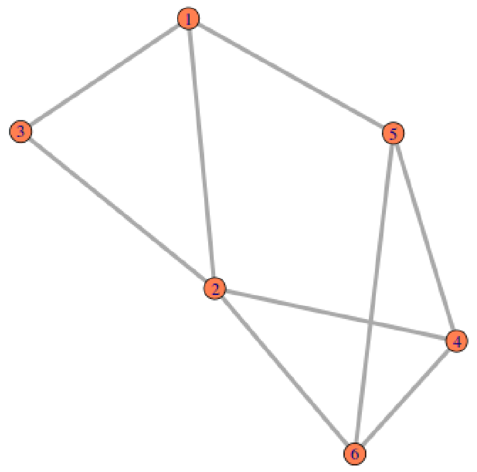

Figure 1: Graph for Problem 3.

b) (1p) What is the (pessimistic) computational complexity of your implementation in terms of nodes of the graph?
c) (2p) The **igraph** package contains a function, `igraph::degree()` that takes as input an `igraph` object and returns the degree of every node of the graph. You can create an `igraph` object from an adjacency matrix using `igraph::graph_from_adjacency_matrix()`. Set the argument `mode="undirected"`. Implement a unit test, based on that `igraph::degree()`, that tests your implementation.