LINKÖPINGS UNIVERSITET
Department of information and computer science
Statistics and Machine learning
Leif Jonsson
Examiner: Mattias Villani

Exam

732A94 Advanced Programming in R - Exam solutions

Time:	14:15-18:00, 2016-10-19
Material:	The extra material is included in the zip-file **exam_material.zip**.
Grades:	A = 19-20 points.
	B = 17-18 points.
	C = 12-16 points.
	D = 10-11 points.
	E = 8-9 points.
	F = 0-7 points.

Instructions

Write your code in an R script file named **Main.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.

Problem 1 (6 p)

a) Implement a function you call `clipped_normal()` with three arguments, `limit`, `mean` and `sd`, the arguments should have the following default values, 200, 5 and 2. The functions should draw normally distributed random numbers and return these in a vector. The function should keep drawing random numbers while there are less than `limit` number of values in the resulting vector that is bigger than two times the given `sd`. The function should return a vector containing the drawn random covariates.

```
set.seed(4711)
given_sd <- 2;
given_mean <- 5;
clipped <- clipped_normal(200,given_mean,given_sd);
sum(clipped>(2*given_sd))

[1] 200

length(clipped)

[1] 294
```

Suggested solution:

```
function(limit = 200, mean=5, sd=2) {
  draw_cnt <- 0
  res <- c()
  while(draw_cnt < limit) {
    val <- rnorm(1,mean=mean,sd=sd)
    res <- c(res,val)
    if(val>2*sd) {
      draw_cnt <- draw_cnt + 1
    }
  }
  res
}
```

---

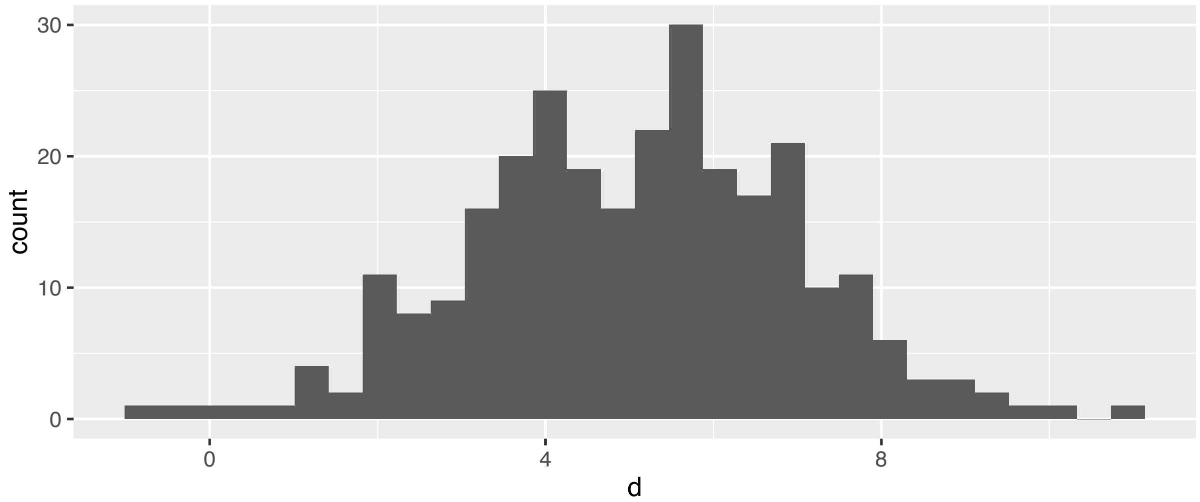
b) What is the complexity of this algorithm with regard to `limit`. Assume that drawing a random number is a constant operation.

Suggested solution:
The complexity is linear, $O(limit)$, in the input size.

---

c) Visualize the draws from your function using a 200 limit (with the default values for the other inputs) as histogram using `ggplot2`.

Suggested solution:



Problem 2 (7 p)

a) Use object oriented programming in S3 or RC to implement a random number generator. You should be able to create new objects of at least two different types of random generators. There should be a function **draw_rnd** which returns a random covariate according to the class of the given object. You are allowed to use the built in R functions for drawing random covariates. In the below examples the classes are created using S3, if you use RC it will look different, but you should still implement the **draw_rnd** function below but in that case they will take your RC objects instead of S3 objects as input.

```
bb <- list(shape1=2,shape2=3)
class(bb) <- "beta"

draw_rnd(bb)

[1] 0.239254

nn <- list(mu=2,sigma=3)
class(nn) <- "norm"

draw_rnd(nn)

[1] 9.50571
```

Suggested solution:

```
draw_rnd <-  function (x, ...) UseMethod("draw_rnd")
draw_rnd.beta <-  function (x, ...)
{
    rbeta(1, x$shape1, x$shape2)
}
draw_rnd.norm <-  function (x, ...)
{
    rnorm(1, mean = x$mu, sd = x$sigma)
}
```

```
RCRnd <- setRefClass("rnd",
  fields = list(),
  methods = list()
)

print.rnd <- function(x, ...){
  print(paste0("I am a random generator of type: ", class(x)[1]))
}

norm <- setRefClass("norm",
                    contains = "rnd",
                    fields = list(mu = "numeric",
                                  sigma = "numeric"),
                    methods = list(
                      draw_rnd = function() {
                          rnorm(1, mean=mu, sd=sigma)
```

```
                             }
))

no <- norm$new(mu=2,sigma=3)
no$draw_rnd()
print(no)

beta <- setRefClass("beta",
                             contains = "rnd",
                             fields = list(shape1 = "numeric",
                                           shape2 = "numeric"),
                             methods = list(
                              draw_rnd = function() {
                                rbeta(1,shape1, shape2)
                              }
))

be <- beta$new(shape1=2,shape2=3)
be$draw_rnd()
print(be)

draw_rnd <- function(x,...) {
  x$draw_rnd()
}

draw_rnd(be)
```

---

b) In R, inheritance hierarchies are specified by the vector of class names attached to each object. Implement a print function for objects inheriting from class "rnd" which prints a string informing that the object is of class random generator and the specific type of random generator. Implement your print function so that **no new code** needs to be added if new random number generators are added.

```
bb <- list(shape1=2,shape2=3)
class(bb) <- c("beta","rnd")
draw_rnd(bb)

[1] 0.579226

print(bb)

[1] "I am a random generator of type: beta"
```

Suggested solution:

```
print.rnd <-  function (x, ...)
{
    print(paste0("I am a random generator of type: ", class(x)[1]))
}
```

---

c) Describe some typical characteristics and differences between object oriented programming and functional programming. Describe some advantages of each approach.

Suggested solution:

In object oriented programming **objects** belong to a **class**, and the behaviour of **methods** depends on the class of the object. Objects have a **class hierarchy**, where children **inherits** behaviour from its parent class. Typically an object oriented programming style deals with objects with internal **states** which are **modified**, while in functional programming a given function always returns the **same result given the same inputs**.

Object oriented programming can lead to a more **natural modelling of domain objects than funtional programming**. Object oriented programming encurages **code reuse** through inheritance and composition. A functional programming style is often simpler to reason about since there is no internal state that is modified, this often makes functional programs **easier to parallelize**.

Problem 3 (7 p)

a) It is World War III. The machines are trying to take over the world. You are fighting on the human side. Implement a function you call `sekrit_msg()` with one argument, `msg` which is a starting secret message. The function should return a list containing two named elements which are functions, the first should be called "change" and the second "tell". The function in the "change" slot should take one string argument which should change the secret message. The second function in the "tell" slot should take no arguments and print the secret message with a "tell count" ahead of it. The tell count should be the number of times the tell function has been called.

```
ll <- sekrit_msg("Nothing here")
change <- ll$change
tell <- ll$tell
tell()

[ 1 ] Nothing here

change("Meet me under the bridge")
tell()

[ 2 ] Meet me under the bridge

change("No, changed my mind, let's make it in the bar")
tell()

[ 3 ] No, changed my mind, let's make it in the bar

tell()

[ 4 ] No, changed my mind, let's make it in the bar
```

Suggested solution:

```
function(msg) {
tellcnt <- 0
change <- function(newmsg) {
     msg <<- newmsg
}
tell <- function() {
tellcnt <<- tellcnt + 1
     cat(paste("[",tellcnt,"]",msg))
}
list(change=change,tell=tell)
}
```

b) Describe, in terms of different types of environments, where the variables and functions involved in implementing and executing `sekrit_msg` "live".

Suggested solution:
The sekrit_msg function is bound to a name in the Global environment, The two closures "tell" and "change" are declared in the execution environment of the sekrit_msg function, so is the "tell count" variable. The two closures "tell" and "change" returned from the sekrit_msg function are later also bound to names in the Global environment. They are also bound, together with the input argument "msg" and "tell count", to names in the captured execution environment of the sekrit_msg function. The calling environment of the sekrit_msg, change and tell functions is the Global environment. The execution environment of sekrit_msg is the captured enclosing environment of the two closures.

c) You changed your mind. You are now fighting on the machine side, having realized they will win anyway. Implement a function called `double_agent()` that takes two aguments, a captured `change` function and a new `fake_msg` that changes the captured change functions message, *without calling the* `change` *function*! You are allowed to use your knowledge about the implementation of `sekrit_msg` to solve your task.

```
double_agent(change, "No, changed my mind again, let's make it in the garage")
tell()

[ 5 ] No, changed my mind again, let's make it in the garage
```

Suggested solution:

```
function(change_function, fake_msg) {
  chg_env <- environment(change_function)
  chg_env$msg <- fake_msg
}
```

Good luck!