ps4

Baoyue Liang

10/6/2018

Problem 1

```
environment(bootmeans)
```

```
## <environment: 0x7fe135b72fc0>
```

```
environment(make_container)
```

```
## <environment: R_GlobalEnv>
```

"Error in numeric(n) : argument"n" is missing, with no default" is returned when we run make_container(). The bootmeans() itself is a function, which indicates that make_container returns the function bootmeans(). bootmeans()is the function defined in the make_container function, this could also be shown as we call out the environment of both function as above. The enclosing environment of make_container() is global environment.

The for loop will place results in x in the function env't and we can grab it out by bootmeans().
During the ith loop, the sample mean is assigned to the x[i], and i is updated to i + 1.

```
n <- 1000000
bootmeans <- make_container(n)
library(pryr)
object_size(bootmeans)
```

```
## 8.02 MB
```

As was shown above, the appximate size is 8 MB (8 bytes * 1e6 = 8 MB)

Problem 2

```
n <- 100000
p <- 5
tmp <- exp(matrix(rnorm(n*p), nrow = n, ncol = p))
probs <- tmp / rowSums(tmp)
smp <- rep(0, n)
set.seed(1)
system.time(
  for(i in seq_len(n))
    smp[i] <- sample(p, 1, prob = probs[i, ])
)
```

```
##    user  system elapsed
##   0.587   0.014   0.603
```

```
cdf<- probs
system.time({
  # construct the cdf matrix
  for(i in 2:5){
```

```
    cdf[,i] = cdf[,i-1] + cdf[,i]
  }
  # construct the uniform random vector
  u <- runif(n)
  # Assign True to elements that are smaller than cdf
  idx <- (u <= cdf)
  smp <- rowSums(idx)
})
```

```
##    user  system elapsed
##   0.018   0.005   0.024
```

```
function1 <- function(probs){
  for(i in seq_len(n))
    smp[i] <- sample(p, 1, prob = probs[i, ])
}

function2 <- function(probs){
    for(i in 2:5){
    probs[,i] = probs[,i-1] + probs[,i]
    }
    u <- runif(n)
    idx <- (u <= probs)
    smp <- rowSums(idx)
}

library(rbenchmark)
benchmark(function1(probs),function2(probs),replications = 100,
          columns = c("test", "replications", "elapsed",
                      "relative", "user.self", "sys.self"))
```

```
##               test replications elapsed relative user.self sys.self
## 1 function1(probs)          100  56.026    25.18    54.039    1.757
## 2 function2(probs)          100   2.225     1.00     1.819    0.400
```

What we are doing here is that we are using a cdf matrix, and a uniform random vector to decide which interval the each element of the vector falls into.

Problem 3

(a)

we need to calculate n^n and n!, whose result can be very large, if not overflowing. There we should use log calculation.

```
logFunction <- function(k, n, p, phi){
  a <- lchoose(n,k)
  if (n < k) a <- 0

  b <- (1-phi)*((n-k)*log(n-k)+k*log(k)-n*log(n))
  if (k==0 | k==n | n == 0 ) b <- 0

  c <- k*phi*log(p)+(n-k)*phi*log(1-p)
```

```
  return(a + b + c)
}


denominator <- function(n, p, phi){
  x <- sapply(seq(0,n), logFunction, n=n, p=p, phi=phi)
  return(sum(exp(x)))
}

# test function
denominator(50,0.3,0.5)
```

```
## [1] 1.424186
```

(b)

```
logFunction_v <- function(n, p, phi){
  k <- c(0:n)
  # This time a, b and c will all be n+1 numerics calculated based on c from 0 to n
  a <- lchoose(n,k)

  b <- (1-phi)*((n-k)*log(n-k)+k*log(k)-n*log(n))
  b[1] <- 0
  b[n+1] <- 0

  c <- k*phi*log(p)+(n-k)*phi*log(1-p)

  return(a+b+c)
}

denominator_v <- function(n, p, phi){
  x <- logFunction_v(n, p, phi)
  return(sum(exp(x)))
}


#test function
denominator(50,0.3,0.5)
```

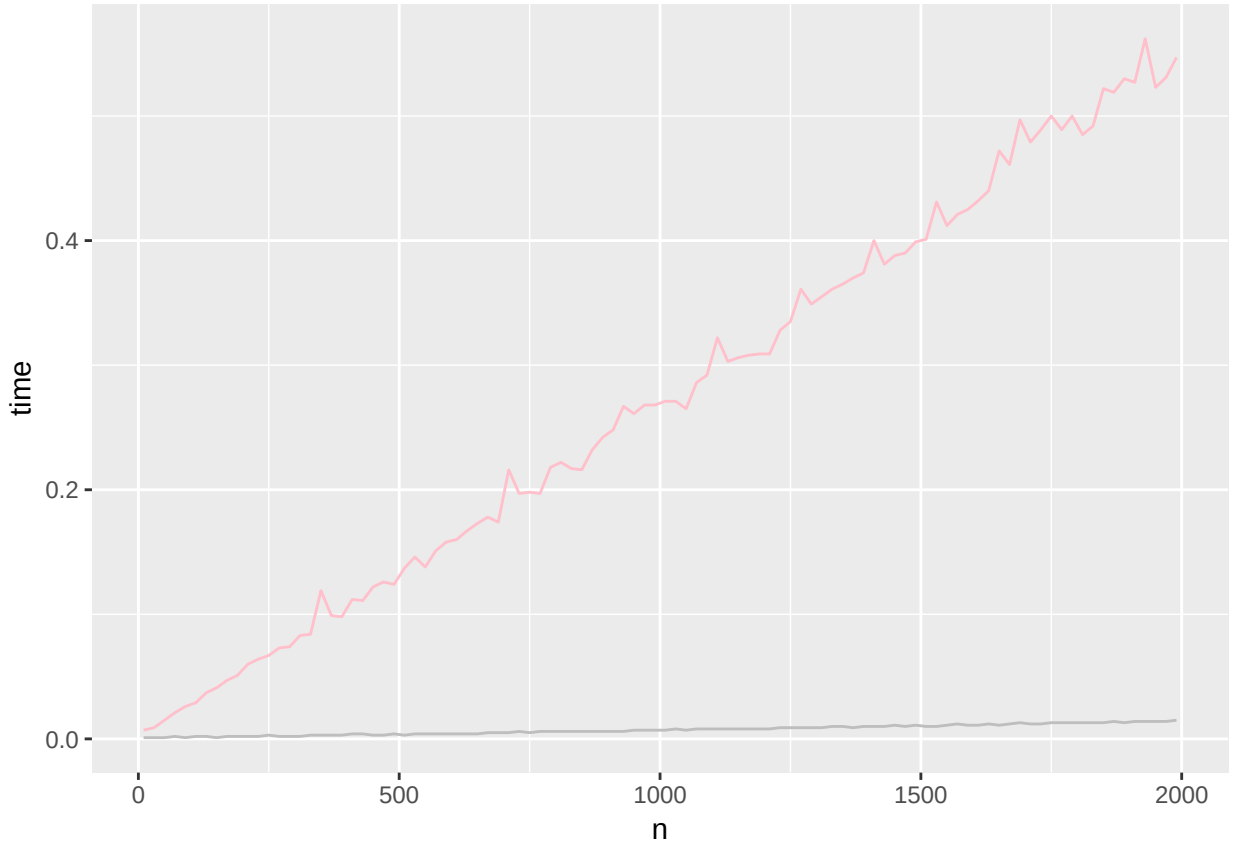
```
## [1] 1.424186
```

```
# Slice the magnitude of n(from 10 to 2000)  by 5
itv <- seq(10, 2000, 20)

# Time the system time using denominator function, where argument n will be then replaced by itv
tm <- function(n, p, phi){
  tm <- system.time(for(i in 1:50) denominator(n, p, phi))
  return(tm[[3]])
  }
# Time the system time using denominator_v function, where argument n will be then replaced by itv
tm_v <- function(n, p, phi){
  tm <- system.time(for(i in 1:50) denominator_v(n, p, phi))
  return(tm[[3]])
```

```
  }
tm1 <- sapply(itv, tm, 0.3, 0.5)
tm2 <- sapply(itv, tm_v, 0.3, 0.5)
df <- data.frame(itv, tm1, tm2)
```

```
library(ggplot2)
ggplot(df, aes(x=itv)) +
  geom_line(aes(y=tm1), colour="pink") +
  geom_line(aes(y=tm2), colour="grey") +
  ylab("time") + xlab("n")
```



Problem 4

(a)

```
set.seed(1)
# data_list is a list that contains 3 lists, and each of the list is a vector of 10 random variable
data_list <- lapply(1:3, function(x) as.vector(x=rnorm(10)))
#check address before manipulation
rapply(data_list,address)
```

```
## [1] "0x7fe135311950" "0x7fe1353118a8" "0x7fe135311800"
```

```
address(data_list)
```

```
## [1] "0x7fe137ec7110"
```

```
# we want to replace the first vector with its max value
max_x <- pmax(data_list[[1]])
data_list[[1]] <- max_x
#check address after manipulation
rapply(data_list,address)
```

```
## [1] "0x7fe137d21f58" "0x7fe1353118a8" "0x7fe135311800"
```

```
address(data_list)
```

```
## [1] "0x7fe13752fba8"
```

```
address(max_x)
```

```
## [1] "0x7fe137d21f58"
```

What we can see here is that the other two vector in the data_list kept the same address. In general, only the address of the modified piece and the data structure will have to be replaced. In this example, the max_x vector had to be created anyway, so it is not that inefficient.

(b)

```
set.seed(2)
# data_list is a list that contains 3 lists, and each of the list is a vector of 10 random variable
data_list <- lapply(1:3, function(x) as.vector(x=rnorm(10)))
data_list_copy = data_list
#check address before manipulation
rapply(data_list,address)
```

```
## [1] "0x7fe1374127f0" "0x7fe137412748" "0x7fe1374126a0"
```

```
address(data_list)
```

```
## [1] "0x7fe1393f03f0"
```

```
rapply(data_list_copy,address)
```

```
## [1] "0x7fe1374127f0" "0x7fe137412748" "0x7fe1374126a0"
```

```
address(data_list_copy)
```

```
## [1] "0x7fe1393f03f0"
```

```
# we want to replace the first vector with its max value
max_x <- pmax(data_list[[1]])
data_list[[1]] <- max_x
#check address after manipulation
rapply(data_list,address)
```

```
## [1] "0x7fe137e9cc98" "0x7fe137412748" "0x7fe1374126a0"
```

```
address(data_list)
```

```
## [1] "0x7fe137e61f68"
```

```
rapply(data_list_copy,address)
```

```
## [1] "0x7fe1374127f0" "0x7fe137412748" "0x7fe1374126a0"
```

```
address(data_list_copy)
```

```
## [1] "0x7fe1393f03f0"
```

We can see that the address of the copy stayed the same before and after the manipulation.But apparently, when we make change to the original data_list, no change has been made to data_list_copy.

(c)

```
set.seed(3)
# data_list is a list that contains 3 lists, and each of the list is a vector of 10 random variable
data_list <- lapply(1:3, function(x) as.vector(x=rnorm(10)))
data_list_copy = data_list
#check the before address
rapply(data_list,address)
```

```
## [1] "0x7fe137c7bea0" "0x7fe137c7bdf8" "0x7fe137c7bd50"
```

```
address(data_list)
```

```
## [1] "0x7fe1393effb8"
```

```
rapply(data_list_copy,address)
```

```
## [1] "0x7fe137c7bea0" "0x7fe137c7bdf8" "0x7fe137c7bd50"
```

```
address(data_list_copy)
```

```
## [1] "0x7fe1393effb8"
```

```
data_list_copy[[4]] <- as.vector(x=rnorm(10))
#check the after address
rapply(data_list,address)
```

```
## [1] "0x7fe137c7bea0" "0x7fe137c7bdf8" "0x7fe137c7bd50"
```

```
address(data_list)
```

```
## [1] "0x7fe1393effb8"
```

```
rapply(data_list_copy,address)
```

```
## [1] "0x7fe137c7bea0" "0x7fe137c7bdf8" "0x7fe137c7bd50" "0x7fe137edda98"
```

```
address(data_list_copy)
```

```
## [1] "0x7fe137c63170"
```

Wwhen we make a copy of the data_list, the original and the copy share the same address. And after we add an element to the copied list, the address of the copied list changed, which means we make a copy of the data_list_copy and add an element to the list. But the data_list is not copied, it still share two elements with the modified copied list.

(d)

```
tmp <- list()
x <- rnorm(1e7)
```

```
tmp[[1]] <- x
tmp[[2]] <- x
.Internal(inspect(tmp))
```

```
## @7fe139406dc0 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
##   @10e99a000 14 REALSXP g0c7 [NAM(2)] (len=10000000, tl=0) 0.786507,-0.310463,1.69888,-0.794594,0.348
##   @10e99a000 14 REALSXP g0c7 [NAM(2)] (len=10000000, tl=0) 0.786507,-0.310463,1.69888,-0.794594,0.348
```

```
object.size(tmp)
```

```
## 160000136 bytes
```

```
gc(tmp)
```

```
##            used  (Mb) gc trigger  (Mb) max used  (Mb)
## Ncells   513710  27.5     940480  50.3   750400  40.1
## Vcells 13153906 100.4   19342886 147.6 13209291 100.8
```

The reason why only 80 MB is used is because that both elements in the list tmp point at the same vector. Therefore, there is only one vector occupied the memory.

Problem 5

```
set.seed(1)
save(.Random.seed, file = 'tmp.Rda')
rnorm(1)
```

```
## [1] -0.6264538
```

```
load('tmp.Rda')
rnorm(1)
```

```
## [1] -0.6264538
```

```
tmp <- function() {
  load('tmp.Rda')
  print(rnorm(1))
}

tmp()
```

```
## [1] 0.1836433
```

Although we load the .Rda file, it just seet seed in the function environment of tmp, but the original variable .Random.seed in the global environment is not changed. Since the function rnorm() is defined in the global environment, the rnorm() is corresponded to global variables. Therefore, the random number generated by tmp() is not the same number.