ps4

Yihuan Song

9/30/2018

question 1

When one runs make_container(), the function bootmeans() is returned. The enclosing environment for make_container() is the global environment, and bootmeans is defined inside of make_container(), and the enclosing environment for bootmeans() is the make_container() function. For the function bootmeans, the values of x and i will be looked up in the environment of make_container(). If we execute bootmeans(), the function will see if the value passed in(which, in this case, is the mean of 272 data sampled from data with replacement) is null, if it is null, the function returns x, which is a vector of length n(in this case, nboot = 100) consisting of zeros; if the value is not null, the value, which is the computed mean, will be the ith element of vector x. Since the bootmeans() function modifies the x vector in the environment of make_container, and carries the modified x vector as it returns, this is a function that "contains" data. If n = 1000000, bootmeans uses 8000848 bytes of memory.

```
library(pryr)
library(rbenchmark)
library(knitr)

make_container <- function(n) {
  x <- numeric(n)
  i <- 1
  function(value = NULL) {
    if (is.null(value)) {
      return(x) } else {
      x[i] <<- value
      i <<- i + 1
    } }
}
nboot <- 100
bootmeans <- make_container(nboot)
data <- faithful[ , 1]
for (i in 1:nboot)
  bootmeans(mean(sample(data, length(data), replace=TRUE)))
bootmeans()
```

```
##   [1] 3.507993 3.453952 3.471099 3.448592 3.439099 3.520938 3.409063
##   [8] 3.556382 3.374349 3.451107 3.326029 3.494136 3.462662 3.463629
##  [15] 3.444743 3.617890 3.421250 3.548563 3.476838 3.415467 3.449728
##  [22] 3.500757 3.519357 3.549445 3.425040 3.484099 3.386805 3.435085
##  [29] 3.366956 3.580092 3.462037 3.530360 3.457566 3.476191 3.389243
##  [36] 3.448706 3.472287 3.432522 3.429415 3.483765 3.562585 3.533474
##  [43] 3.520934 3.436754 3.558974 3.466279 3.557449 3.658798 3.343915
##  [50] 3.635926 3.450206 3.399636 3.580515 3.532882 3.486673 3.506349
##  [57] 3.563945 3.402103 3.517165 3.528721 3.474324 3.468662 3.530210
##  [64] 3.519452 3.448202 3.397279 3.536081 3.514335 3.475015 3.383915
##  [71] 3.508298 3.409048 3.417250 3.481140 3.475596 3.525511 3.464691
##  [78] 3.538026 3.433136 3.529871 3.447199 3.396489 3.422588 3.584165
##  [85] 3.465180 3.458926 3.390618 3.485949 3.382390 3.419901 3.519404
##  [92] 3.525140 3.515276 3.537915 3.546478 3.409349 3.537996 3.448574
```

```
##  [99] 3.511371 3.395772
```

```
#checking out the environment of functions
environment(make_container)
```

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

```
environment(bootmeans)
```

```
## <environment: 0x7fc8476b3b40>
```

```
#finding out size when n=1000000

num <- 1000000
bootmeans <- make_container(num)
for (i in 1:num)
  bootmeans(mean(sample(data, length(data), replace=TRUE)))
object_size(bootmeans())
```

```
## 8 MB
```

question 2

To make the solution faster, I first generated a matrix of cumulative probabilities using the probability matrix. To avoid using the sample() function, I used the "runif" to generate a random uniform vector of length 100000. Then, I looped by column to compare the uniform matrix with the columns in the cumulative probability matrix, and set the values of the smp to the column index whenever the uniform variable lies in the column of cumulative probability. I timed each method, and found significant improvement in efficiency.

```
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
##   0.409   0.026   0.435
```

```
#faster method
smp1 <- rep(0,n)
sums <-probs
set.seed(1)
sampler <- runif(n)
#transform probability matrix into cumulative probabilities
for(i in 1:4){
  sums[,i+1] <- rowSums(sums[ , c(i,i+1)])
}

#Compare each column with uniform sampler, and set the smp values to column index,
#if the uniform sampler lies in the range of the cumulative probability column
system.time(
  for(col in 1:p){
```

```
      smp1[sums[, col] > sampler & smp1 == 0 ] <- col
  }
)
```

```
##    user  system elapsed
##   0.013   0.001   0.014
```

```
#comparing efficiency
benchmark(
  for(i in seq_len(n))
      smp[i] <- sample(p, 1, prob = probs[i, ]),
  for(col in 1:p)
      smp1[sums[, col] > sampler & smp1 == 0 ] <- col,
  replications = 10)
```

```
##                                                          test
## 2 for (col in 1:p) smp1[sums[, col] > sampler & smp1 == 0] <- col
## 1 for (i in seq_len(n)) smp[i] <- sample(p, 1, prob = probs[i, ])
##   replications elapsed relative user.self sys.self user.child sys.child
## 2           10   0.118    1.000     0.094    0.024          0         0
## 1           10   4.348   36.847     4.080    0.268          0         0
```

question 3

(a) The "denom" function will calculate f for each k, and we use sapply to loop through each k, and take the summation. We need to do the calculation on log scale, because if we derictly used the formula f, the calculation would fail because the numbers in the calculation are too large.

```
#initializing parameters
p = 0.3
phi = 0.5
n=1000
range <- 1:(n-1)

#function will take k as input, calculate the log-scale of all terms in f and then take the exponent
#function will return the result of f, based on different k calues
denom <- function(k){
  logpow <- k*log(k)+(n-k)*log(n-k)-(n*log(n))
  log_prob <- (k*phi)*log(p)+((n-k)*phi)*log(1-p)
  log_denom <- lchoose(n,k) + logpow - phi*logpow +log_prob
  return(exp(log_denom))
}

#define denominator when k = 0 and k=n
denom_1 <- (1-p)^(n*phi)
denom_n <- p^(n*phi)

sum(sapply( range, denom), denom_1, denom_n)
```

```
## [1] 1.414659
```

(b) By doing the calculation in a fully vectorized way and benchmarking the two methods, we found that the vectorized way is much more efficient than the apply method. After benchmarking for different values of n, we found that as n gets larger, the speed difference between the loop method and the

vectorized method is larger. In other words, the vactorized method's advantage in efficiency gets more and more valuable as n gets larger.

```
rm(n)
p = 0.3
phi = 0.5

cal_denom <- function(n){
  #define denominator when k = 0 and k=n
  denom_1 <- (1-p)^(n*phi)
  denom_n <- p^(n*phi)
  #set k and ln as matrices that hold values for k and n
  k <- matrix(1:(n-1))
  ln <- matrix(rep(n, n-1))
  #define terms in f using log scales & in a vectorized form
  logpow <- k*log(k)+(ln-k)*log(ln-k)-(ln*log(ln))
  log_prob <- (k*phi)*log(p)+((ln-k)*phi)*log(1-p)
  log_denom <- lchoose(ln,k) + logpow - phi*logpow +log_prob
  #calculate f and take summations
  f <- exp(log_denom)
  result <- sum(f, denom_1, denom_n)
  return(result)
}

cal_denom(1000)
```

```
## [1] 1.414659
```

```
#compare time it takes for method in a) and b) for n taking values from 10 to 2000
benchmark(cal_denom(10),
          {
            n = 10
            range <- 1:(n-1)
            denom_1 <- (1-p)^(n*phi)
            denom_n <- p^(n*phi)
            sum(sapply( range, denom),denom_1,denom_n)
          },
          replications = 100)
```

```
##
## 2 {\n    n = 10\n    range <- 1:(n - 1)\n    denom_1 <- (1 - p)^(n * phi)\n    denom_n <- p^(n * phi
## 1
##   replications elapsed relative user.self sys.self user.child sys.child
## 2          100   0.004        2     0.005        0          0         0
## 1          100   0.002        1     0.002        0          0         0
```

```
benchmark(cal_denom(100),
          {
            n = 100
            range <- 1:(n-1)
            denom_1 <- (1-p)^(n*phi)
            denom_n <- p^(n*phi)
            sum(sapply( range, denom),denom_1,denom_n)
          },
          replications = 100)
```

```
##
## 2 {\n    n = 100\n    range <- 1:(n - 1)\n    denom_1 <- (1 - p)^(n * phi)\n    denom_n <- p^(n * phi
## 1
##   replications elapsed relative user.self sys.self user.child sys.child
## 2          100   0.021      4.2     0.020        0          0         0
## 1          100   0.005      1.0     0.004        0          0         0
```

```
benchmark(cal_denom(1000),
          {
            n = 1000
            range <- 1:(n-1)
            denom_1 <- (1-p)^(n*phi)
            denom_n <- p^(n*phi)
            sum(sapply( range, denom),denom_1,denom_n)
          },
          replications = 100)
```

```
##
## 2 {\n    n = 1000\n    range <- 1:(n - 1)\n    denom_1 <- (1 - p)^(n * phi)\n    denom_n <- p^(n * ph
## 1
##   replications elapsed relative user.self sys.self user.child sys.child
## 2          100   0.165    9.167     0.164    0.001          0         0
## 1          100   0.018    1.000     0.017    0.000          0         0
```

```
benchmark(cal_denom(2000),
          {
            n = 2000
            range <- 1:(n-1)
            denom_1 <- (1-p)^(n*phi)
            denom_n <- p^(n*phi)
            sum(sapply( range, denom),denom_1,denom_n)
          },
          replications = 100)
```

```
##
## 2 {\n    n = 2000\n    range <- 1:(n - 1)\n    denom_1 <- (1 - p)^(n * phi)\n    denom_n <- p^(n * ph
## 1
##   replications elapsed relative user.self sys.self user.child sys.child
## 2          100   0.322   10.063     0.322    0.001          0         0
## 1          100   0.032    1.000     0.032    0.000          0         0
```

question 4

(a) From the R demonstration below, we can see that if we modify an element of one of the vectors, R will not make copy of the list, since the address is the same before and after the modification. Therefore, R will change in place.

```
mylist <- list(c(1,2,3),c(4,5,6),c(7,8,9))
.Internal(inspect(mylist))
mylist[[1]][[1]] <- 100
.Internal(inspect(mylist))
```

result from r

```
> mylist <- list(c(1,2,3),c(4,5,6),c(7,8,9))
> .Internal(inspect(mylist))
@7fbded96b308 19 VECSXP g0c3 [NAM(1)] (len=3, tl=0)
  @7fbded96b3f8 14 REALSXP g0c3 [] (len=3, tl=0) 1,2,3
  @7fbded96b3a8 14 REALSXP g0c3 [] (len=3, tl=0) 4,5,6
  @7fbded96b358 14 REALSXP g0c3 [] (len=3, tl=0) 7,8,9
> mylist[[1]][[1]] <- 100
> .Internal(inspect(mylist))
@7fbded96b308 19 VECSXP g0c3 [NAM(1)] (len=3, tl=0)
  @7fbded96b3f8 14 REALSXP g0c3 [] (len=3, tl=0) 100,2,3
  @7fbded96b3a8 14 REALSXP g0c3 [] (len=3, tl=0) 4,5,6
  @7fbded96b358 14 REALSXP g0c3 [] (len=3, tl=0) 7,8,9
```

(b) As we can see from the demonstration in r below, the address did not change when we make a copy of the list, the two lists had the same address. If we change one of the elements of the copied list, the address changed for the modified list, so copy was made when we modified the copied list. However, the address only changed for the modified element in the copied list. From the inspection, we can see that since we changed only the first vector of the copied list, the address for the second and third vector did not change. Therefore, R will only make copy of the vector where modification was specified, and copies of the remaining vectors in the list will not be made.

```
mylist <- list(c(1,2,3),c(4,5,6),c(7,8,9))
address(mylist)
newlist <- mylist
address(newlist)
newlist[[1]] <- c(100, 200,300)
.Internal(inspect(mylist))
.Internal(inspect(newlist))
```

result from r

```
> mylist <- list(c(1,2,3),c(4,5,6),c(7,8,9))
> address(mylist)
[1] "0x7fbded96b088"
> newlist <- mylist
> address(newlist)
[1] "0x7fbded96b088"
> newlist[[1]] <- c(100, 200,300)
> .Internal(inspect(mylist))
@7fbded96b088 19 VECSXP g0c3 [NAM(2)] (len=3, tl=0)
  @7fbded96b178 14 REALSXP g0c3 [NAM(3)] (len=3, tl=0) 1,2,3
  @7fbded96b128 14 REALSXP g0c3 [NAM(3)] (len=3, tl=0) 4,5,6
  @7fbded96b0d8 14 REALSXP g0c3 [NAM(3)] (len=3, tl=0) 7,8,9
> .Internal(inspect(newlist))
@7fbded96aea8 19 VECSXP g0c3 [NAM(1)] (len=3, tl=0)
  @7fbded96aef8 14 REALSXP g0c3 [NAM(1)] (len=3, tl=0) 100,200,300
  @7fbded96b128 14 REALSXP g0c3 [NAM(3)] (len=3, tl=0) 4,5,6
  @7fbded96b0d8 14 REALSXP g0c3 [NAM(3)] (len=3, tl=0) 7,8,9
```

(c) After adding an element to the second list of the copied list, the addresses of the copied list and the second list of the copied list changed, and a new address created for the new element added, but all

others did not change. Therefore, only the second list was copied when we were modifying the copied list, and the two lists share all data except for the element being added.

```
ls_list <- list(list(1,2,3), list(4,5,6), list(7,8,9))
address(ls_list)
snd_list <- ls_list

#Add an element to the second list
snd_list[[2]] <- append(snd_list[[2]], 100)

.Internal(inspect(ls_list))

.Internal(inspect(snd_list))
```

result from r

```
> ls_list <- list(list(1,2,3), list(4,5,6), list(7,8,9))
> address(ls_list)
[1] "0x7fa092a17c98"
> snd_list <- ls_list
>
> #Add an element to the second list
> snd_list[[2]] <- append(snd_list[[2]], 100)
>
> .Internal(inspect(ls_list))
@7fa092a17c98 19 VECSXP g0c3 [NAM(3)] (len=3, tl=0)
  @7fa092a17d88 19 VECSXP g0c3 [NAM(3)] (len=3, tl=0)
    @7fa099d1b038 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 1
    @7fa099d1b000 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 2
    @7fa099d1afc8 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 3
  @7fa092a17d38 19 VECSXP g0c3 [NAM(3)] (len=3, tl=0)
    @7fa099d1af90 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 4
    @7fa099d1af58 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 5
    @7fa099d1af20 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 6
  @7fa092a17ce8 19 VECSXP g0c3 [NAM(3)] (len=3, tl=0)
    @7fa099d1aee8 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 7
    @7fa099d1aeb0 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 8
    @7fa099d1ae78 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 9
>
> .Internal(inspect(snd_list))
@7fa092a17888 19 VECSXP g0c3 [NAM(1)] (len=3, tl=0)
  @7fa092a17d88 19 VECSXP g0c3 [NAM(3)] (len=3, tl=0)
    @7fa099d1b038 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 1
    @7fa099d1b000 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 2
    @7fa099d1afc8 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 3
  @7fa092a178d8 19 VECSXP g0c3 [NAM(1)] (len=4, tl=0)
    @7fa099d1af90 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 4
    @7fa099d1af58 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 5
    @7fa099d1af20 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 6
    @7fa099d1a858 14 REALSXP g0c1 [] (len=1, tl=0) 100
  @7fa092a17ce8 19 VECSXP g0c3 [NAM(3)] (len=3, tl=0)
    @7fa099d1aee8 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 7
    @7fa099d1aeb0 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 8
    @7fa099d1ae78 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 9
```

(d) According to "?size.object", "sizes of objects using a compact internal representation may be over-estimated" for the function object.size().

From the function .Internal(inspect()) of tmp, we can see that the two elements of tmp share the same address, so the x is NOT copied twice while we were assigning x to tmp[[1]] and tmp[[2]], we only had x to point to both of the elements in the tmp list, so the two elements are from the same place in memory. Since we did not actually make two copies, the size of tmp would not be ~160 MB, it would actually be ~80 MB

instead.

```
tmp <- list()
x <- rnorm(1e7)
tmp[[1]] <- x
tmp[[2]] <- x
object.size(tmp)
```

```
## 160000160 bytes
```

```
#internal inspection to see addresses
.Internal(inspect(tmp))
```

```
## @7fc846399d48 19 VECSXP g0c2 [NAM(3)] (len=2, tl=0)
##   @10b5e4000 14 REALSXP g0c7 [NAM(3)] (len=10000000, tl=0) 0.103694,-0.202456,-0.531656,-0.978496,1.5
##   @10b5e4000 14 REALSXP g0c7 [NAM(3)] (len=10000000, tl=0) 0.103694,-0.202456,-0.531656,-0.978496,1.5
```

```
.Internal(inspect(x))
```

```
## @10b5e4000 14 REALSXP g0c7 [NAM(3)] (len=10000000, tl=0) 0.103694,-0.202456,-0.531656,-0.978496,1.567
```

```
#try object_size to show 80 MB used
object_size(tmp)
```

```
## 80 MB
```

question 5

In the original question, inside the tmp function, the "load('tmp.Rda')" did not actually load the seed of 1 into the function, which means that it did not work in the environment of the tmp() function. If we set the seed inside the function, we can see that the number became the same as the beginning.

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

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

```
load('tmp.Rda')
rnorm(1)   ## same random number
```

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

```
#original
tmp <- function() {
  load('tmp.Rda')
  print(rnorm(1))
}
tmp()      ##not the same number
```

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

```
#set seed in the function
tmp <- function() {
  set.seed(1)
  load('tmp.Rda')
  print(rnorm(1))
}
tmp()    ##the same number
```

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