ps4code

A.J. Torre

September 30, 2018

Comments: I worked with Huy, R.J., Andrea, Norae, and Phil.

Question 1

I ran the code to get a better idea of what was getting returned from running the make_container() function.

```
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)
bootmeans
```

```
## function(value = NULL) {
##         if (is.null(value)) {
##             return(x)
##         } else {
##             x[i] <<- value
##             i <<- i + 1
##         }
##     }
## <environment: 0x000000001c6af330>
```

So, from this part of the code, we can see that make_container returns a function, and now bootmeans is the inner function. From part of the unit 4 reading, we can learn about enclosures/ functions that return a function, which is happening here. While I've never seen coding where a function returns a function, from the reading, it seems like this can be helpful as the new function can access data/ objects of the enclosing function and different inputs won't change the value within the enclosure. For example, x and i here can never change as they're within the make_container function, which is nice because in your global environment, you may use x for the name of some other object and this function will still run with the x you intended. Also, if someone else wanted to use this code, they don't have to worry about what their global environment looks like or if they're overwriting/ renaming an already existing object. In this particular question, we only have one enclosing environment, which is where x and i are so they can't be changed by any input to bootmeans/ the inner function.

```
data <- faithful[, 1]
for (i in 1:nboot) bootmeans(mean(sample(data, length(data),
    replace = TRUE)))
## this will place results in x in the function env't and you
## can grab it out as
bootmeans()
```

```
##   [1] 3.465158 3.552978 3.416015 3.397654 3.585364 3.628827 3.623007
##   [8] 3.459423 3.408063 3.448309 3.496118 3.442699 3.522353 3.595037
##  [15] 3.471982 3.291956 3.488757 3.485040 3.529143 3.489011 3.534632
##  [22] 3.713456 3.583842 3.513136 3.489522 3.335261 3.451386 3.549316
##  [29] 3.509570 3.534180 3.634029 3.597408 3.500563 3.477956 3.640250
##  [36] 3.463702 3.313717 3.466404 3.511614 3.470022 3.509518 3.465110
##  [43] 3.506096 3.357588 3.402790 3.539147 3.437301 3.514338 3.531526
##  [50] 3.601456 3.426382 3.412651 3.405375 3.572342 3.466147 3.398357
##  [57] 3.468147 3.479585 3.510923 3.350643 3.568239 3.381963 3.315114
##  [64] 3.487232 3.401676 3.470934 3.563338 3.419728 3.375783 3.494088
##  [71] 3.528110 3.456191 3.444710 3.489676 3.359993 3.591952 3.490467
##  [78] 3.418375 3.493930 3.494787 3.498026 3.556974 3.566489 3.468423
##  [85] 3.425967 3.531489 3.558908 3.584665 3.546408 3.542515 3.492320
##  [92] 3.499875 3.492022 3.477298 3.637610 3.545673 3.532515 3.615165
##  [99] 3.446614 3.514607
```

```
object_size(bootmeans())
```

```
## 848 B
```

Now, bootmeans contains the data using sample on the information about the Geysers. I observed that when we call bootmeans() after running the for-loop, we get a vector of 100 decimals, resulting from the mean/ sample functions. So, bootmeans does now contain data. When we call object_size on bootmeans(), we get that it's 800 bytes, which makes sense as bootmeans() now has 100 numbers in it and each number is 8 bytes each. If n=1,000,000, we would see that bootmeans() should equal around 8,000,000 bytes as bootmeans() will then contain 1,000,000 numbers that are 8 bytes each.

Problem 2

Here is Chris' "slow" method:

```
n <- 1e+05
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
##    0.67    0.02    0.69
```

```r
head(smp)
```

```
## [1] 4 3 2 1 5 3
```

I tried other ways to make sample faster, such as using replicate, apply and .internal sample. The results are below.

```r
n <- 1e+05
p <- 5
tmp <- exp(matrix(rnorm(n * p), nrow = n, ncol = p))
probs <- tmp/rowSums(tmp)
smp <- rep(0, n)
# using replicate instead of a for loop
set.seed(1)
system.time(replicate(n, sample(p, 1, prob = runif(5))))
```

```
##    user  system elapsed
##    0.95    0.02    0.97
```

```r
# using apply instead of a for loop
set.seed(1)
system.time(apply(probs, MARGIN = 1, FUN = function(x) sample(p,
    1, prob = x)))
```

```
##    user  system elapsed
##    0.79    0.00    0.78
```

```r
# using .Internal(sample) w/in a for loop
set.seed(1)
system.time(for (i in seq_len(n)) smp[i] <- .Internal(sample(p,
    1, replace = FALSE, prob = probs[i, ])))
```

```
##    user  system elapsed
##    0.25    0.00    0.25
```

So, similar to what I saw online about using replicate and apply, it may not always be faster than a for-loop. As we saw in class, sometimes functions call another function within them, which is what sample does so .Internal(sample) will call c code, which is faster than using just the sample function as we saw above.

I discussed with some classmates on a quicker way to do this as one of the hints was to try it without using the sample function, and they pointed me to try using an inverse transform, which I looked up online. The below code instead uses runif to form the matrix and then uses which.max to look at the maximum probabilities in the rows.

```r
# using inverse transform method
n <- 1e+05
p <- 5
tmp <- exp(matrix(runif(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] <- which.max(probs[i,
    ]))
```

```
##    user  system elapsed
##    0.08    0.00    0.08
```

```
head(smp)
```

```
## [1] 2 5 2 2 4 2
```

So, we see that this gives us a smp similar to the original code and does run considerably faster than the previous codes. From this problem set, I learned that making code faster doesn't always necessarily mean getting rid of a forloop, as seen in the other methods I tried as apply and replicate didn't prove to be significantly faster (and one was slower) than using a forloop. I also saw how using the .Internal code can be useful. When I looked into the sample functions, which runs two if-else statements to call other functions, and these if-else statements essentially check to make sure the data is in the right format to use sample. While .Internal did not show to be the fastest, I feel like it's a pretty quick fix to speed up code, especially if your function calls to faster c code.

Problem 3

a) To write a function to calculate the denominator, I first wrote a function that would take n,p,k, and q as inputs and uses logs to calculate each term separately. Then, we use the exponential of the sum to get the total product. As we learned in class, R can only be accurate to 16 digits in floating points or for integers, if it can't represent it, we can get a NaN answer. But, R can keep accuracy when working with logs and exponents so we want to take the log of each the first to avoid R going off to infinity for large n or not representing the digits accurately.

I wrote two separate functions, one to calculate all but when k=0 and k=n and one to calculate for the first and last k as these terms simplify to simply the last factors of the function. Then, I have a third function that uses sum and sapply while calling the two other functions in order to calculate the total.

```
# this function takes four inputs n,p,k,q, & calculates the
# terms for the cases when k is strictly greater than 0 and
# less than n
denominator_formula <- function(n, p, k, q) {
    assert_that(is.numeric(n))
    assert_that(is.numeric(p))
    assert_that(is.numeric(k))
    assert_that(is.numeric(q))
    first_term <- lchoose(n, k)
    second_term <- k * log(k) + (n - k) * log(n - k) - n * log(n)
    third_term <- q * (n * log(n) - k * log(k) - (n - k) * log(n -
        k))
    fourth_term <- k * q * log(p)
    fifth_term <- (n - k) * q * log(1 - p)
    total_product <- exp(first_term + second_term + third_term +
        fourth_term + fifth_term)
    return(total_product)
}

# this function takes 3 inputs n,p,q & calculates the first
# and last k needed for our sum
first_and_last_k <- function(n, p, q) {
    only_term5 <- exp(n * q * log(1 - p))
```

```
    only_term4 <- exp(n * q * log(p))
    return(sum(only_term4, only_term5))
}

# this function takes 3 inputs n,p,q & creates the k's from
# making a list from 1 to n-1
get_total <- function(n, p, q) {
    x <- list(1:(n - 1))
    final_sums <- sum(sapply(x, denominator_formula, n = n, p = p,
        q = q)) + first_and_last_k(n, p, q)
    return(final_sums)
}

# example output from function
get_total(2000, 0.3, 0.5)
```

```
## [1] 1.414436
```

```
get_total(10, 0.3, 0.5)
```

```
## [1] 1.475851
```

b) To completely vectorize this function, I kept the same two functions for calculating the 2nd to n-1 term and the function for calculating with k=0 and k=n. Instead of using sapply, I created a vector from 1 to n-1 and applied and summed the function over that.

```
get_vector_total <- function(n, p, q) {
    x <- c(1:(n - 1))
    final_sums <- sum(denominator_formula(n = n, p = p, k = x,
        q = q)) + first_and_last_k(n, p, q)
    return(final_sums)
}

get_vector_total(2000, 0.3, 0.5)
```

```
## [1] 1.414436
```

To compare my sapply function to my vector function, I used the microbench function to plot the timing results of the functions.

```
# using microbenchmark for n=2000
microbenchmark(get_total(2000, 0.3, 0.5), get_vector_total(2000,
    0.3, 0.5), times = 100L)
```

```
## Unit: milliseconds
##                              expr      min       lq     mean   median
##         get_total(2000, 0.3, 0.5) 1.146455 1.180588 1.225679 1.203416
##  get_vector_total(2000, 0.3, 0.5) 1.124696 1.145389 1.248676 1.165656
##        uq      max neval
##  1.245442 1.632430   100
##  1.204268 6.937183   100
```

```
# unit in millieconds
```

```
# use microbenchmark for n=100
microbenchmark(get_total(100, 0.3, 0.5), get_vector_total(100,
    0.3, 0.5), times = 100L)
```

```
## Unit: microseconds
##                              expr     min      lq     mean  median       uq
##         get_total(100, 0.3, 0.5) 185.174 189.654 210.9534 193.708 203.3075
##  get_vector_total(100, 0.3, 0.5) 158.721 161.281 177.8526 163.841 169.6010
##      max neval
##  529.068   100
##  452.695   100
```

```
# unit in microseconds
```

```
# using microbenchmark for n=10
microbenchmark(get_total(10, 0.3, 0.5), get_vector_total(10,
    0.3, 0.5), times = 100L)
```

```
## Unit: microseconds
##                             expr     min      lq     mean   median      uq
##         get_total(10, 0.3, 0.5) 135.681 139.521 161.9848 144.0010 157.441
##  get_vector_total(10, 0.3, 0.5) 108.801 112.001 138.5310 114.3475 124.374
##      max neval
##  369.068   100
##  548.268   100
```

```
# unit in microseconds
```

So, here, we can compare the timing of each function and see that get_vector_total is somewhat quicker than using sapply in order calculate the denominator. I used the microbenchmark package to time the results and to compare the outputs of trying the functions with n=2000,100,and 10. Overall, get_vector_total, which isn't too much of a change from the original function but doesn't use sapply, is faster than using sum(sapply).

Problem 4

For this problem, I used R terminal and the outputs I got R terminal are in `#comments`.

a) Below, I created a list of vectors and used Internal inspect to look at the addresses and see if there were changes made without making a copy.

```
list_vectors <- list(c(1, 2, 3, 4), c(5, 6, 7, 8), c(9, 10, 11,
    12))
.Internal(inspect(list_vectors))
# output from R terminal: @0x000000001ac2a2a8 19 VECSXP g0c3
# [NAM(1)] (len=3, tl=0) @0x000000001ac2a398 14 REALSXP g0c3
# [] (len=4, tl=0) 1,2,3,4 @0x000000001ac2a348 14 REALSXP
# g0c3 [] (len=4, tl=0) 5,6,7,8 @0x000000001ac2a2f8 14
# REALSXP g0c3 [] (len=4, tl=0) 9,10,11,12
```

```
list_vectors[[1]][2] <- 9
.Internal(inspect(list_vectors))
# output from R terminal: @0x000000001ac2a2a8 19 VECSXP g0c3
# [NAM(1)] (len=3, tl=0) @0x000000001ac2a398 14 REALSXP g0c3
# [] (len=4, tl=0) 1,9,3,4 @0x000000001ac2a348 14 REALSXP
# g0c3 [] (len=4, tl=0) 5,6,7,8 @0x000000001ac2a2f8 14
# REALSXP g0c3 [] (len=4, tl=0) 9,10,11,12
```

So, we see that R can make a change in place as an element of the first vector changed, but the address did not so R did not have to make a copy in order to make this change, which is good as it's likely what the user wants.However, as we discussed in class if we have a list or vector of characters, the address will change as with characters or letters, each one seems to have a unique address so we could not make character changes in place.

b) Next, I made a copy of the list first and looked at the addresses of each and the mem_used before and after making the copy.

```
library(pryr)
mem_used()
# 30.7 MB
temp <- list_vectors
mem_used()
# 30.9 MB
.Internal(inspect(list_vectors))
# @0x000000001ac2a2a8 19 VECSXP g1c3 [MARK,NAM(2)] (len=3,
# tl=0) @0x000000001ac2a398 14 REALSXP g1c3 [MARK] (len=4,
# tl=0) 1,9,3,4 @0x000000001ac2a348 14 REALSXP g1c3 [MARK]
# (len=4, tl=0) 5,6,7,8 @0x000000001ac2a2f8 14 REALSXP g1c3
# [MARK] (len=4, tl=0) 9,10,11,12
.Internal(inspect(list_vectors))
# @0x000000001ac2a2a8 19 VECSXP g1c3 [MARK,NAM(2)] (len=3,
# tl=0) @0x000000001ac2a398 14 REALSXP g1c3 [MARK] (len=4,
# tl=0) 1,9,3,4 @0x000000001ac2a348 14 REALSXP g1c3 [MARK]
# (len=4, tl=0) 5,6,7,8 @0x000000001ac2a2f8 14 REALSXP g1c3
# [MARK] (len=4, tl=0) 9,10,11,12
```

So, it looks like the memory increases slightly while making the copy, but looking at temp, we see that its addresses point to the exact same place as list_vectors does so that implies that when R makes a copy, it assigns our copy to point to the same addresses as our original.

c) Next, I made a list of lists and used .Internal inspect once agin to see more information about the list.

```
list_list <- list(list("a", "b", "c"), list("d", "e", "f"), list("g",
    "h", "i"))
.Internal(inspect(list_list))
# output from R terminal: @0x00000000047717738 19 VECSXP g0c3
# [NAM(1)] (len=3, tl=0) @0x00000000047717328 19 VECSXP g0c3
# [] (len=3, tl=0) @0x0000000004847558 16 STRSXP g0c1
# [NAM(3)] (len=1, tl=0) @0x0000000018aaa6f0 09 CHARSXP g1c1
# [MARK,gp=0x61] [ASCII] [cached] 'a' @0x0000000004847590 16
# STRSXP g0c1 [NAM(3)] (len=1, tl=0) @0x0000000018e6b720 09
# CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] 'b'
```

```
# @0x000000000048475c8 16 STRSXP g0c1 [NAM(3)] (len=1, tl=0)
# @0x000000000185b2a28 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII]
# [cached] 'c' @0x000000000047174b8 19 VECSXP g0c3 [] (len=3,
# tl=0) @0x0000000000484760016 STRSXP g0c1 [NAM(3)] (len=1,
# tl=0) @0x0000000018fa7f30 09 CHARSXP g1c1 [MARK,gp=0x61]
# [ASCII] [cached] 'd' @0x0000000004847638 16 STRSXP g0c1
# [NAM(3)] (len=1, tl=0) @0x000000000187f1d18 09 CHARSXP g1c1
# [MARK,gp=0x61] [ASCII] [cached] 'e' @0x000000000048476701 6
# STRSXP g0c1 [NAM(3)] (len=1, tl=0) @0x0000000018903050 09
# CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] 'f'
# @0x000000000047175a8 19 VECSXP g0c3 [] (len=3, tl=0)
# @0x000000000048476a8 16 STRSXP g0c1 [NAM(3)] (len=1, tl=0)
# @0x000000001987a9d0 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII]
# [cached] 'g' @0x000000000048476e0 16 STRSXP g0c1 [NAM(3)]
# (len=1, tl=0) @0x000000001895d330 09 CHARSXP g1c1
# [MARK,gp=0x61] [ASCII] [cached] 'h' @0x0000000004847718 16
# STRSXP g0c1 [NAM(3)] (len=1, tl=0) @0x000000000187f1df8 09
# CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] 'i'

list_copy <- list_list
.Internal(inspect(list_copy))
# output from R terminal: @0x0000000004717738 19 VECSXP g0c3
# [NAM(2)] (len=3, tl=0) @0x0000000004717328 19 VECSXP g0c3
# [] (len=3, tl=0) @0x0000000004847558 16 STRSXP g0c1
# [NAM(3)] (len=1, tl=0) @0x0000000018aaa6f0 09 CHARSXP g1c1
# [MARK,gp=0x61] [ASCII] [cached] 'a' @0x0000000004847590 16
# STRSXP g0c1 [NAM(3)] (len=1, tl=0) @0x0000000018e6b720 09
# CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] 'b'
# @0x000000000048475c8 16 STRSXP g0c1 [NAM(3)] (len=1, tl=0)
# @0x000000000185b2a28 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII]
# [cached] 'c' @0x000000000047174b8 19 VECSXP g0c3 [] (len=3,
# tl=0) @0x0000000000484760016 STRSXP g0c1 [NAM(3)] (len=1,
# tl=0) @0x0000000018fa7f30 09 CHARSXP g1c1 [MARK,gp=0x61]
# [ASCII] [cached] 'd' @0x0000000004847638 16 STRSXP g0c1
# [NAM(3)] (len=1, tl=0) @0x000000000187f1d18 09 CHARSXP g1c1
# [MARK,gp=0x61] [ASCII] [cached] 'e' @0x000000000048476701 6
# STRSXP g0c1 [NAM(3)] (len=1, tl=0) @0x0000000018903050 09
# CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] 'f'
# @0x000000000047175a8 19 VECSXP g0c3 [] (len=3, tl=0)
# @0x000000000048476a8 16 STRSXP g0c1 [NAM(3)] (len=1, tl=0)
# @0x000000001987a9d0 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII]
# [cached] 'g' @0x000000000048476e0 16 STRSXP g0c1 [NAM(3)]
# (len=1, tl=0) @0x000000001895d330 09 CHARSXP g1c1
# [MARK,gp=0x61] [ASCII] [cached] 'h' @0x0000000004847718 16
# STRSXP g0c1 [NAM(3)] (len=1, tl=0) @0x000000000187f1df8 09
# CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] 'i'

list_copy[[1]][2] <- "z"
.Internal(inspect(list_copy[[1]]))
# @0x000000000047179b8 19 VECSXP g0c3 [NAM(1)] (len=3, tl=0)
# @0x0000000004847558 16 STRSXP g0c1 [NAM(3)] (len=1, tl=0)
# @0x0000000018aaa6f0 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII]
# [cached] 'a' @0x000000000048479b8 16 STRSXP g0c1 [] (len=1,
```

```
# tl=0) @0x00000000188cf7b8 09 CHARSXP g1c1 [MARK,gp=0x61]
# [ASCII] [cached] 'z' @0x00000000048475c8 16 STRSXP g0c1
# [NAM(3)] (len=1, tl=0) @0x000000000185b2a28 09 CHARSXP g1c1
# [MARK,gp=0x61] [ASCII] [cached] 'c' see that it does
# change!

.Internal(inspect(list_list[[1]]))
# @0x0000000004717328 19 VECSXP g0c3 [NAM(3)] (len=3, tl=0)
# @0x0000000004847558 16 STRSXP g0c1 [NAM(3)] (len=1, tl=0)
# @0x0000000018aaa6f0 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII]
# [cached] 'a' @0x0000000004847590 16 STRSXP g0c1 [NAM(3)]
# (len=1, tl=0) @0x0000000018e6b720 09 CHARSXP g1c1
# [MARK,gp=0x61] [ASCII] [cached] 'b' @0x00000000048475c8 16
# STRSXP g0c1 [NAM(3)] (len=1, tl=0) @0x000000000185b2a28 09
# CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] 'c'
```

We see that list_copy has the same addresses/ "points to" the same places as our original list, list_list. We also see that when we change our copy, the address of the 2nd element within the first list changes, which suggests that R needs to make a copy in order to change an element. This is consistent with what we learned in class about making trying to changes with character elements as R doesn't do changes in place with characters as each character address points to a different place so to change a character element, the address needs to change as well.

d) Here's the code that we're given to run:

```
tmp <- list()
x <- rnorm(1e+07)
tmp[[1]] <- x
tmp[[2]] <- x
.Internal(inspect(tmp))

# output from R terminal: @0x0000000004fc3b18 19 VECSXP g0c2
# [NAM(1)] (len=2, tl=0) @0x00007ff4f8db0010 14 REALSXP g0c7
# [NAM(3)] (len=10000000, tl=0)
# 0.0773031,-0.296869,-1.18324,0.0112927,0.991601,...
# @0x00007ff4f8db0010 14 REALSXP g0c7 [NAM(3)] (len=10000000,
# tl=0) 0.0773031,-0.296869,-1.18324,0.0112927,0.991601,...

object.size(tmp)

# output in R terminal: 160000160 bytes

# since there 1000000 bytes in 1 megabyte
160000160/(1e+06)

object_size(tmp)
# 80MB
```

So, looking at the output of .Internal(inspect), it looks like that a separate copy isn't made for tmp[[1]] and tmp[[2]] as both have the same address. But, when you run object.size(tmp), the output is 160 bytes, which is double what we would think from looking at the addresses. Object.size() seems to think that tmp[[1]] and tmp[[2]] are different values and it calculates the memory of each and then adds them separately where as

the addresses are exactly the same for each of them when using inspect internal which agrees with the 80MB as it says that a copy isn't made.

When we use object_size in the pryr package, we see that we do indeed get 80 MB as the output. So, object.size can't recognize when two objects share the same address, but object_size can, which we have discussed in class.

Problem 5

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

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

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

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

```
temp1 <- function() {
    load("tmp.Rda")
    a <- rnorm(1)
    print(a)
}

temp1()
```

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

So, we observe that the first two ways of using set.seed(1) and rnorm(1) give us the same output, but using it within a function does not. I worked with a classmate (Huy) on this to observe that when we commented out the load part of the function we still got the same output. We then looked into the load function and it says that there's a part of the load function to specify where to load the file in from. The tmp.Rda file is created in the global environment. Since we don't specificy this within the function, it actually doesn't load in the file and doesn't know where to look for it so when you call temp1() it just goes to the next number of the random seed as set.seed(1) was set in the global environment. If we want to generate the same results, we should specify where to load the file in from. As we see below, once we specify the parent frame in the load command, we get the same output.

```
temp1 <- function() {
    load("tmp.Rda", .GlobalEnv)
    a <- rnorm(1)
    print(a)
}

temp1()
```

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