ps4

Malvika Rajeev

10/5/2018

Question 1

Explain what is going on in make_container() and bootmeans(). In particular, when one runs make_container() what is returned?

In make_container, the following happens:

1. The user provides an input (n).
2. x then equals input number of zeroes. Numeric creates a double-precision vector of the specified length with each element equal to 0.
3. 'i' is set to one.
4. Then a new function is created within the function: so its enclosing environment is the function make_container.
5. The argument of make_container is n. So basically by assigning bootstrap the value make_container(100), the parameter 'n' has been defined.
6. the '<<' commit the variables to the parent environment, which is the environment of function make_container in this case. This makes it possible to maintain a counter that records how many times a function has been called.

What happens when one executes bootmeans()

After assinging make_container(100) to bootstrap, if bootstrap has no argument it'll just print nboot number of zeroes. However, if we provide an argument to bootstrap, it won't return x or any other input, it'll just return the number of times, (including the present) the function has been called. This is because of the "i" that's been initialised outside bootstrap, and which gets incremented everytime the function is run (because of '<<-').

What are the various enclosing environments?

The enclosing environment for make_container is the global environment (or whatever environment in which it was created. For bootstrap(), it is the environment of make_container.

In what sense is this a function that ''contains" data?

Because bootstrap was assigned make_container(100), whenever it is called it'll always have a numeric vector of 100 zeroes referenced along with it, along with intialising counter i = 1.

How much memory does bootmeans use if n = 1000000?

Essentialy the difference is of changing a zero-valued vector of length 100 to 1000000.

```
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)

library(pryr)
mem_change(bootmeans <- make_container(1000000))
```

```
## 12.5 MB
```

In R script and the terminal, the memory change is 8MB and not 12. RMarkdown for some reason is showing 12.

```
##Comparing it to simply changing a numeric vector of length 100 to 1000000
x <- numeric(100)
mem_change(x <- numeric(1000000))
```

```
## 8 MB
```

Question 2

```
n <- 100000
p <- 5 ## number of categories

tmp <- exp(matrix(rnorm(n*p), nrow = n, ncol = p))
probs <- tmp / rowSums(tmp)
smp <- rep(0, n)

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

My approach was to find the row-wise cumulative sum of the probabilities, generate ‘n’ number of random numbers between 0 and 1, then see where the random numbers we generated lands within that sum. (FOR EACH COLUMN). I used the pryr package.

```r
##more efficient solution
library(pryr)

s <- seq_len(p) #The numbers we we mean to sample from

l = runif(n) ##generates 'n' random numbers between 0 and 1

sumrow <- probs %*% upper.tri(diag(p), diag = TRUE) / rowSums(probs)
##we get an n x p matrix with cumulative probabilities

i <- rowSums(l > sumrow) +1L  #sums the number of cumulative probilies less than the
 random number


selection <- s[i]
head(selection) ##selection is the required sample
```

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

Now, to test the efficiency of the new solution:

```r
microbenchmark({
               sumrow <- probs %*% upper.tri(diag(p), diag = TRUE) / rowSums(probs);
               i <- rowSums(l > sumrow) +1L;
               selection <- s[i]},  for(i in seq_len(n)){
                 smp[i] <- sample(p, 1, prob = probs[i, ])}, times = 25)
```

expr
<fctr>
{ sumrow <- probs %*% upper.tri(diag(p), diag = TRUE)/rowSums(probs) i <- rowSums(l > sumrow) + 1L sele
{ sumrow <- probs %*% upper.tri(diag(p), diag = TRUE)/rowSums(probs) i <- rowSums(l > sumrow) + 1L sele
{ sumrow <- probs %*% upper.tri(diag(p), diag = TRUE)/rowSums(probs) i <- rowSums(l > sumrow) + 1L sele
for (i in seq_len(n)) { smp[i] <- sample(p, 1, prob = probs[i,]) }
{ sumrow <- probs %*% upper.tri(diag(p), diag = TRUE)/rowSums(probs) i <- rowSums(l > sumrow) + 1L sele
for (i in seq_len(n)) { smp[i] <- sample(p, 1, prob = probs[i,]) }
{ sumrow <- probs %*% upper.tri(diag(p), diag = TRUE)/rowSums(probs) i <- rowSums(l > sumrow) + 1L sele
for (i in seq_len(n)) { smp[i] <- sample(p, 1, prob = probs[i,]) }
for (i in seq_len(n)) { smp[i] <- sample(p, 1, prob = probs[i,]) }
{ sumrow <- probs %*% upper.tri(diag(p), diag = TRUE)/rowSums(probs) i <- rowSums(l > sumrow) + 1L sele

1-10 of 50 rows | 1-1 of 2 columns   Previous **1** 2 3 4 5 Next

It’s a lot, lot faster.

Question 3

part(a)

Using the standard vapply function to calculate the denominator:

```
oneterm <- function(n){
  x <- n
  function(k){
  if (k > x){break}
  if (k == 0){
    return(exp(lchoose(x,k) + (((x-k)*log(x-k)) -
                               (x*log(x))) + 0.5*((x*log(x)) - ((x-k)*log(x-k))) + (0.5
*k*log(0.3)) + (0.5*(x-k)*log(0.7))))}
  else if (k == x){
    return(exp(lchoose(x,k) + ((k*log(k)) -
                               (x*log(x))) + 0.5*((x*log(x)) - (k*log(k))) + (0.5*k*log
(0.3)) + (0.5*(x-k)*log(0.7))))}
  else {
  return(exp(lchoose(x,k) + ((k*log(k)) + ((x-k)*log(x-k)) - (x*log(x))) + 0.5*((x*lo
g(x)) - (k*log(k))
                                                                                - ((x
-k)*log(x-k))) + (0.5*k*log(0.3)) + (0.5*(x-k)*log(0.7))))}
  }}

#Creating a function that sums all the terms
applyWay <- function(n){ return((sum(unlist(vapply(0:n, oneterm(n), 0))))))}

applyWay(1000)
```

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

part (b)

Now, to vectorise the function, I used to 'ifelse' function:

```
new <- function(k, n, p, phi) {
  x <- n
  ifelse(k==0, exp(lchoose(x,k) + (((x-k)*log(x-k)) - (x*log(x))) + 0.5*((x*log(x)) -
 ((x-k)*log(x-k))) + (0.5*k*log(0.3)) + (0.5*(x-k)*log(0.7))),
         ifelse(k==n, exp(lchoose(x,k) + ((k*log(k)) - (x*log(x))) + 0.5*((x*log(x))
 - (k*log(k))) + (0.5*k*log(0.3)) + (0.5*(x-k)*log(0.7))),
                exp(lchoose(x,k) + ((k*log(k)) + ((x-k)*log(x-k)) - (x*log(x))) + 0.5
*((x*log(x)) - (k*log(k)) - ((x-k)*log(x-k))) + (0.5*k*log(0.3))
                    + (0.5*(x-k)*log(0.7)))))}


n <- 200
microbenchmark(sum(new(0:n, n)), applyWay(n), times = 20)
```

expr <fctr>	**time** <dbl>
sum(new(0:n, n))	29830063
applyWay(n)	2260475
applyWay(n)	423119
sum(new(0:n, n))	159476

expr <fctr>	time <dbl>
applyWay(n)	1949853
sum(new(0:n, n))	124810
sum(new(0:n, n))	110369
sum(new(0:n, n))	111124
applyWay(n)	378801
applyWay(n)	316862

1-10 of 40 rows Previous **1** 2 3 4 Next

```
n <- 2000
microbenchmark(sum(new(0:n, n)), applyWay(n), times = 20)
```

expr <fctr>	time <dbl>
applyWay(n)	3219707
applyWay(n)	3061585
sum(new(0:n, n))	873768
applyWay(n)	3059941
applyWay(n)	3050617
applyWay(n)	3124231
sum(new(0:n, n))	865814
sum(new(0:n, n))	888107
sum(new(0:n, n))	920868
sum(new(0:n, n))	889609

1-10 of 40 rows Previous **1** 2 3 4 Next

```
n <- 1000
microbenchmark(sum(new(0:n, n)), applyWay(n), times = 20)
```

expr <fctr>	time <dbl>
sum(new(0:n, n))	481528
sum(new(0:n, n))	463057
sum(new(0:n, n))	444596
applyWay(n)	1536521
sum(new(0:n, n))	450734

expr <fctr>	time <dbl>
applyWay(n)	1527444
sum(new(0:n, n))	447179
applyWay(n)	1526023
applyWay(n)	1549427
applyWay(n)	1537283

The vectorised function is much faster.

Question 4

(a) Consider a list of vectors. Modify an element of one of the vectors. Can R make the change in place, without creating a new list or a new vector?

(b) Next, make a copy of the list and determine if there any copy-on-change going on. When a change is made to one of the vectors in one of the lists, is a copy of the entire list made or just of the relevant vector?

When I make a list of vectors and change any one of the vectors, R modifies the list itself (the address remains the same). When a copy is made, both lists then point to the same space in memory. When i change an element in one the lists, the one that I edit gets copied (in its entirety), but the original one stays the same.

```
l <- list()
n <- 6
for (i in seq_len(n)){
    l[[i]] <- c(seq_len(i))}
.Internal(inspect(l))
```

```
## @7f8b59701ac8 19 VECSXP g0c4 [NAM(1)] (len=6, tl=0)
##   @7f8b598e8b80 13 INTSXP g0c1 [NAM(1)] (len=1, tl=0) 1
##   @7f8b598e8c28 13 INTSXP g0c1 [NAM(1)] (len=2, tl=0) 1,2
##   @7f8b5aa7e508 13 INTSXP g0c2 [NAM(1)] (len=3, tl=0) 1,2,3
##   @7f8b5aa7e588 13 INTSXP g0c2 [NAM(1)] (len=4, tl=0) 1,2,3,4
##   @7f8b577a6e88 13 INTSXP g0c3 [NAM(1)] (len=5, tl=0) 1,2,3,4,5
##   ...
```

```
l[[3]] <- 4
.Internal(inspect(l))
```

```
## @7f8b596ced88 19 VECSXP g0c4 [NAM(1)] (len=6, tl=0)
##   @7f8b598e8b80 13 INTSXP g0c1 [NAM(3)] (len=1, tl=0) 1
##   @7f8b598e8c28 13 INTSXP g0c1 [NAM(3)] (len=2, tl=0) 1,2
##   @7f8b59cb4e78 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 4
##   @7f8b5aa7e588 13 INTSXP g0c2 [NAM(3)] (len=4, tl=0) 1,2,3,4
##   @7f8b577a6e88 13 INTSXP g0c3 [NAM(3)] (len=5, tl=0) 1,2,3,4,5
##   ...
```

```
#making a copy
k <- l
.Internal(inspect(k)) ##same as i(l)
```

```
## @7f8b596ced88 19 VECSXP g0c4 [NAM(3)] (len=6, tl=0)
##   @7f8b598e8b80 13 INTSXP g0c1 [NAM(3)] (len=1, tl=0) 1
##   @7f8b598e8c28 13 INTSXP g0c1 [NAM(3)] (len=2, tl=0) 1,2
##   @7f8b59cb4e78 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 4
##   @7f8b5aa7e588 13 INTSXP g0c2 [NAM(3)] (len=4, tl=0) 1,2,3,4
##   @7f8b577a6e88 13 INTSXP g0c3 [NAM(3)] (len=5, tl=0) 1,2,3,4,5
##   ...
```

```
##Changing just the copy

k[[2]] <- 5
.Internal(inspect(k))
```

```
## @7f8b5969de68 19 VECSXP g0c4 [NAM(1)] (len=6, tl=0)
##   @7f8b598e8b80 13 INTSXP g0c1 [NAM(3)] (len=1, tl=0) 1
##   @7f8b59cb4fc8 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 5
##   @7f8b59cb4e78 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 4
##   @7f8b5aa7e588 13 INTSXP g0c2 [NAM(3)] (len=4, tl=0) 1,2,3,4
##   @7f8b577a6e88 13 INTSXP g0c3 [NAM(3)] (len=5, tl=0) 1,2,3,4,5
##   ...
```

```
.Internal(inspect(l))
```

```
## @7f8b596ced88 19 VECSXP g0c4 [NAM(3)] (len=6, tl=0)
##   @7f8b598e8b80 13 INTSXP g0c1 [NAM(3)] (len=1, tl=0) 1
##   @7f8b598e8c28 13 INTSXP g0c1 [NAM(3)] (len=2, tl=0) 1,2
##   @7f8b59cb4e78 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 4
##   @7f8b5aa7e588 13 INTSXP g0c2 [NAM(3)] (len=4, tl=0) 1,2,3,4
##   @7f8b577a6e88 13 INTSXP g0c3 [NAM(3)] (len=5, tl=0) 1,2,3,4,5
##   ...
```

(c) Now make a list of lists. Copy the list. Add an element to the second list. Explain what is copied and what is not copied and what data is shared

between the two lists.

When a list is copied, the attributes are the same and the copied list gets the same address, but when a new element is added, the existing attributes have the same addresses but the list itself is copied.

```
l <- list()
for (i in seq_len(n)){l[i] <- list(seq(1,i))}

##Copying the list

k <- l
k[[n+1]] <- c(1,2)

.Internal(inspect(l))
```

```
## @7f8b59282e88 19 VECSXP g0c4 [NAM(3)] (len=6, tl=0)
##   @7f8b58e75238 13 INTSXP g0c1 [NAM(3)] (len=1, tl=0) 1
##   @7f8b58f52510 13 INTSXP g0c0 [NAM(3)]  1 : 2 (compact)
##   @7f8b58f4f208 13 INTSXP g0c0 [NAM(3)]  1 : 3 (compact)
##   @7f8b58f4fe10 13 INTSXP g0c0 [NAM(3)]  1 : 4 (compact)
##   @7f8b58f50a18 13 INTSXP g0c0 [NAM(3)]  1 : 5 (compact)
##   ...
```

```
.Internal(inspect(k))
```

```
## @7f8b592f59c8 19 VECSXP g0c4 [NAM(1)] (len=7, tl=0)
##   @7f8b58e75238 13 INTSXP g0c1 [NAM(3)] (len=1, tl=0) 1
##   @7f8b58f52510 13 INTSXP g0c0 [NAM(3)]  1 : 2 (compact)
##   @7f8b58f4f208 13 INTSXP g0c0 [NAM(3)]  1 : 3 (compact)
##   @7f8b58f4fe10 13 INTSXP g0c0 [NAM(3)]  1 : 4 (compact)
##   @7f8b58f50a18 13 INTSXP g0c0 [NAM(3)]  1 : 5 (compact)
##   ...
```

(d) Run the following code in a new R session. The result of .Internal(inspect()) and of object.size() conflict with each other. In reality only ~80 MB is being used. Show that only ~80 MB is used and explain why this is the case.

```
tmp <- list()
x <- rnorm(1e7)
tmp[[1]] <- x
tmp[[2]] <- x
.Internal(inspect(tmp))
```

```
## @7f8b543f9348 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
##   @10df50000 14 REALSXP g0c7 [NAM(3)] (len=10000000, tl=0) -0.764692,0.326528,-1.0
2994,0.830095,0.691048,...
##   @10df50000 14 REALSXP g0c7 [NAM(3)] (len=10000000, tl=0) -0.764692,0.326528,-1.0
2994,0.830095,0.691048,...
```

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

```
## 160000160 bytes
```

```
object_size(tmp)
```

```
## 80 MB
```

```
compare_size(tmp)
```

```
##      base      pryr
## 160000160  80000112
```

Numeric vectors occupy 8 bytes for every element, integer vectors 4. So 10000000*8 .. it is infact 80 MB. Object.size() gives us an inflated answer because it doesn't account for shared objects. It doesn't take into account the fact that 'tmp' here is just a list of two objects referenced twice. In inspect.element, we see the two attributes are referring to the same piece of memory.

In the documentation, it is clearly mentioned that this function merely provides a rough indication: it should be reasonably accurate for atomic vectors, but does not detect if elements of a list are shared, for example. (Sharing amongst elements of a character vector is taken into account, but not that between character vectors in a single object). Sizes of objects using a compact internal representation may be over-estimated.

Question 5

Why does running tmp() not generate the same random number as earlier?

The seed number you choose is the starting point used in the generation of a sequence of random numbers, which is why (provided you use the same pseudo-random number generator) you'll obtain the same results given the same seed number. I think it's just a matter of which environment the file is being loaded in, If i create a function and do set.seed(1) inside it, I get the same result. If i create a function inside it, and do the same exercise, I won't get the same result.

```
tmp2 <- function(){
  set.seed(1)
  save(.Random.seed, file = 'tmp.Rda')
  rnorm(1)

}

tmp2()
```

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

As given in the problem set, tmp() just generates a new random number each time. To have tmp() display the same result, we have to set.seed(1) inside the tmp function as well, or load the file in the global environment (or wherever the file was intially stored): We should get the same random number.

```
rm(tmp)
set.seed(NULL)


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

tmp <-
  function() { load('tmp.Rda', .GlobalEnv)
    rnorm(1)
}

tmp()
```

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