ps4

Weijie Yuan

9/30/2018

Answer for Problem 1

Q:

Consider the function closure example in on page 65 of Section 6.10 of Unit 4. Explain what is going on in make_container() and bootmeans(). In particular, when one runs make_container() what is returned? What are the various enclosing environments? What happens when one executes bootmeans()? In what sense is this a function that "contains" data? How much memory does bootmeans use if n = 1000000?

A:

```
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)))
```

The following code helps explain things.

```
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 }
##    }
## }
```

```
make_container(n=10)
```

```
## function(value = NULL) {
##     if (is.null(value)) { return(x) } else {
##       x[i] <<- value
##       i <<- i + 1 }
##    }
## <bytecode: 0x7f9e16dd6268>
## <environment: 0x7f9e1921a760>
```

When we call ‘make_container()’, R returns a function with specified n. In ‘make_container()’ function, it generates a numeric object with n entries and set the initial index as 1. There is a nested function defined in the frame of ‘make_container()’ function call. So now ‘bootmeans()’ is assigned as the nested function of ‘make_container()’ with specified n=100. When one calls ‘bootmeans(Value)’ function, the bootmeans will check whether ‘Value’ is null. If it is null, the ‘bootmeans()’ function will directly return current ‘x’. If not, the ‘bootmeans(Value)’ will assign ‘Value’ to x[i] and move the index to the next position. This operation will change the variable ‘x’ defined in the frame of ‘make_contrainer()’ function call.

Then, in each loop of ‘for’, it calls ‘bootmeans()’ to assign the value of ‘mean(sample(data,length(data),replace=TRUE))’ to ‘x’ sequentially. The value of ‘mean(sample(data,length(data),replace=TRUE))’ is compute the mean of sample derived from ‘faithful[,1]’ of which sample size equals to length of ‘faithful[,1]’ and sampling method is with replacement.

As a result, now ‘x’ is a numeric object with n entries and its i(th) entry is mean of i(th) sample from ‘faithful[,1]’ derived above.

```r
bootmeans()
```

```
##   [1] 3.491360 3.419879 3.488092 3.512007 3.404456 3.404044 3.466261
##   [8] 3.442651 3.507324 3.478960 3.468489 3.469121 3.407717 3.586816
##  [15] 3.481331 3.498511 3.484816 3.454849 3.395978 3.559463 3.481695
##  [22] 3.476688 3.364702 3.562474 3.471919 3.423210 3.526320 3.457276
##  [29] 3.518798 3.579816 3.405621 3.588011 3.461618 3.660379 3.482268
##  [36] 3.511121 3.391963 3.521956 3.340342 3.580772 3.518504 3.401085
##  [43] 3.384827 3.455570 3.326007 3.569243 3.454574 3.499978 3.373757
##  [50] 3.417206 3.552349 3.506581 3.285827 3.521132 3.531732 3.471268
##  [57] 3.527746 3.463618 3.502640 3.353382 3.599173 3.546721 3.420570
##  [64] 3.616607 3.449114 3.557143 3.527349 3.405015 3.519643 3.511816
##  [71] 3.459176 3.518184 3.506636 3.484901 3.528195 3.508066 3.558184
##  [78] 3.580923 3.530235 3.550162 3.484551 3.501971 3.359162 3.487772
##  [85] 3.430489 3.325342 3.425338 3.449404 3.520691 3.463886 3.396739
##  [92] 3.321055 3.479621 3.675824 3.423294 3.395158 3.507125 3.524768
##  [99] 3.404327 3.535577
```

Now, we call function ‘bootmeans()’, because the input argument value is null, it returns ‘x’ showed above.

Various Enclosing Environments

```r
environment(make_container)
```

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

```r
environment(bootmeans)
```

```
## <environment: 0x7f9e1923f788>
```

- The enclosing environment for ‘make_container()’, which is global environment of R.
- The enclosing environment for ‘bootmeans()’ where ‘x’ is defined and can be called.

The function ‘make_container()’ contains data in the sense that the values of elements in ‘x’ are only enclosed in the function frame instead of the global enviroment. As a result, if we change the variable ‘x’ in the global enviroment, it will have no effect on the ‘x’ in the local enviroment of this function.

```r
nboot <- 0
bootmeans <- make_container(nboot)
data <- faithful[,1]
for (i in 1:nboot)
  bootmeans(mean(sample(data,length(data),replace=TRUE)))
bootmeans()
```

```
## [1] 3.528978 3.410687
```

```
gc()
```

```
##           used (Mb) gc trigger (Mb) limit (Mb) max used (Mb)
## Ncells  529614 28.3    1198110   64         NA   630602 33.7
## Vcells 1053605  8.1    8388608   64      16384  1767126 13.5
```

```
nboot <- 1000000
bootmeans <- make_container(nboot)
data <- faithful[,1]
for (i in 1:nboot)
  bootmeans(mean(sample(data,length(data),replace=TRUE)))
bootmeans()
```

```
gc()
```

```
##           used (Mb) gc trigger (Mb) limit (Mb) max used (Mb)
## Ncells  529705 28.3    1198110   64         NA  1198110   64
## Vcells 2053465 15.7    8388608   64      16384  8388539   64
```

If n = 1000000, bootmeans uses about 7.6 Mb of memory. We can regard this number is equal to 8 Mb. To be more specific, x contains 1000000 numbers, and each number occupies 8 bytes in memory.

Answer for Problem 2

Q:

However that is a lot slower than some other ways we might do it. How can we do it faster? Hint: think about how you could transform the matrix of probabilities such that you can do vectorized operations without using sample() at all. Hint #2: there is a very fast solution that uses a loop.

A:

```
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)

## slow approach: loop by row and use sample()
set.seed(1)
sample_slow <- function(){
  for(i in seq_len(n)) smp[i] <- sample(p, 1, prob = probs[i, ])
}
system.time(sample_slow())
```

```
##    user  system elapsed
##   0.410   0.021   0.459
```

```
for (i in 2:5){
  probs[,i]=probs[,i]+probs[,i-1]
}
sample_fast<-function(){
  rand <- runif(n)
  tf <- rand < probs
  return (max.col(tf,'first'))
}
system.time(sample_fast())
```

```
##    user  system elapsed
##   0.006   0.001   0.007
```

My solution has a speedup around 60 times the original loop above, which seems to satisfy the requirement. Then we use 'benchmark' to compare two solutions to make the result clearer.

```
benchmark(sample_slow(), sample_fast(),
replications = 20, columns=c('test', 'elapsed', 'replications','relative'))
```

```
##            test elapsed replications relative
## 2 sample_fast()   0.137           20    1.000
## 1 sample_slow()   9.306           20   67.927
```

Answer for Problem 3

(a)

```
sum_f <- function(n,p,phi){
  f<-function(k){
    log = lchoose(n,k)+(1-phi)*(k*log(k)+(n-k)*log(n-k)-n*log(n))+
      k*phi*log(p)+(n-k)*phi*log(1-p)

    # specify the situation of k=n and k=0
    if(k==n||k==0){return(p**(k*phi)*(1-p)**((n-k)*phi))}

    return (exp(log))
  }
  return(sum(sapply(seq(from=0,to=n),FUN = f)))
}

# test the case of n=10
sum_f(10,0.3,0.5)
```

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

If we don't do the calculation on the log scale, when n is large, some terms of summation will be extremely small so that it may go beyond the 16th decimal place accuracy of R representation of a number, which leads to tiny error. However, the accumulation of such small errors may leads to significant bias of our result.

On the other hand, 'choose(n,k)' will become extremely big when n increase up to 2000. It will generate large number that exceeds maximal number of R's handling.

And log scale can transform these two kinds of numbers to more smooth and modest numbers. At the meantime, all of original information will be stored. To sum up, our case benefits from log scale a lot.

(b)

There are two ways to address this calculation in fully vectorized fashion. We can compare their performance later.

- Method I

Build the nest function inside the summation function to carry out the vectorized calculation.

```
sum_f_vect <- function(n,p,phi){
  f<-function(k){
    log = lchoose(n,k)+(1-phi)*(k*log(k)+(n-k)*log(n-k)-n*log(n))
    log = log + k*phi*log(p)+(n-k)*phi*log(1-p)
    return (exp(log))
  }
  return(sum(f(1:(n-1))+(1-p)**(n*phi)+p**(n*phi)))
}
sum_f_vect(2000,0.3,0.5)
```

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

- Method II

Sum directly in fully vectorized way.

```
sum_vect <- function(n,p,phi){
k = 1:(n-1)
sum(exp(lchoose(n,k)+k*log(k) + (n-k) * log(n-k) - n*log(n) +
          phi * (n * log(n) - k * log(k) - (n-k) * log(n-k)) +
          k*phi*log(p) + (n-k)*phi*log(1-p)), exp(n*phi*log(1-p)), exp(n*phi*log(p)))
}
sum_vect(2000,0.3,0.5)
```

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

Rough Comparision

```
n<-500000;p<-0.3;phi=0.5
benchmark(sum_f_vect(n,p,phi), sum_f(n,p,phi), sum_vect(n,p,phi),
          replications = 20, columns=c('test', 'elapsed', 'replications','relative'))
```

```
##                     test elapsed replications relative
## 1 sum_f_vect(n, p, phi)   1.545           20    1.045
## 2      sum_f(n, p, phi)  30.308           20   20.492
## 3    sum_vect(n, p, phi)   1.479           20    1.000
```

Systematic Comparision Using Relative Timing

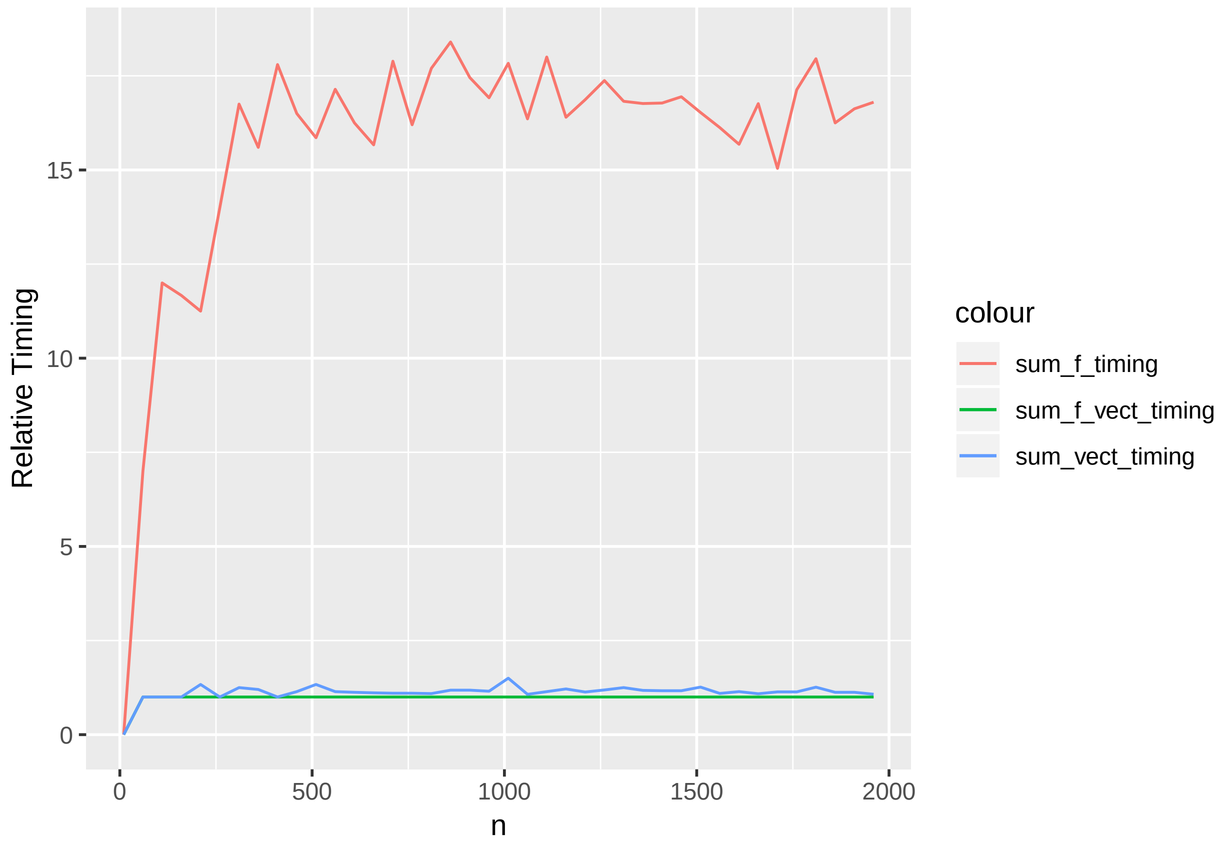
```
# take a seqence of n ranging from 10 to 2000 with step size 10
n = seq(from = 10, to = 2000, by = 50)

sum_f_time = c()
sum_f_vect_time = c()
sum_vect_time=c()

for (i in 1:length(n)){
  sum_f_vect_time[i] = benchmark(sum_f_vect(n[i],p,phi), sum_f(n[i],p,phi), sum_vect(n[i],p,phi),
                                 replications = 100,
                                 columns=c('test','elapsed','replications',
                                          'relative'))$relative[1]
  sum_f_time[i] = benchmark(sum_f_vect(n[i],p,phi), sum_f(n[i],p,phi), sum_vect(n[i],p,phi),
                              replications = 100,
                              columns=c('test','elapsed','replications',
                                        'relative'))$relative[2]
  sum_vect_time[i] = benchmark(sum_f_vect(n[i],p,phi), sum_f(n[i],p,phi), sum_vect(n[i],p,phi),
                              replications = 100,
                              columns=c('test','elapsed','replications',
                                        'relative'))$relative[3]
}
time_data <-
  data.frame(
    n = seq(from = 10, to = 2000, by = 50),
    sum_f_vect_timing = sum_f_vect_time,
    sum_f_timing = sum_f_time,
    sum_vect_timing = sum_vect_time)
```

When n is small, 'benchmark' generate some 'NA' data in 'relative' column. We assign zeros to these positions.

```
time_data[is.na(time_data)] <- 0
library(ggplot2)
ggplot(time_data, aes(n)) +
  geom_line(aes(y = sum_f_vect_timing, colour = "sum_f_vect_timing")) +
  geom_line(aes(y = sum_f_timing, colour = "sum_f_timing")) +
  geom_line(aes(y = sum_vect_timing, colour = "sum_vect_timing")) +
  ylab('Relative Timing')
```



According to the above graphic, we can know that Method I in (b) has the best performance in speed compared with the other two methods. As n goes from 10 to 2000, the relative timing increases from around 5 to around 20 fluctuating in the range between 15 and 20.

That is, when n is small, the difference between ‘sapply()’ and fully vectorized fashion in this case is not that obvious. And as n increases, the difference in speed becomes larger and stay relatively stable when n is fairly large.

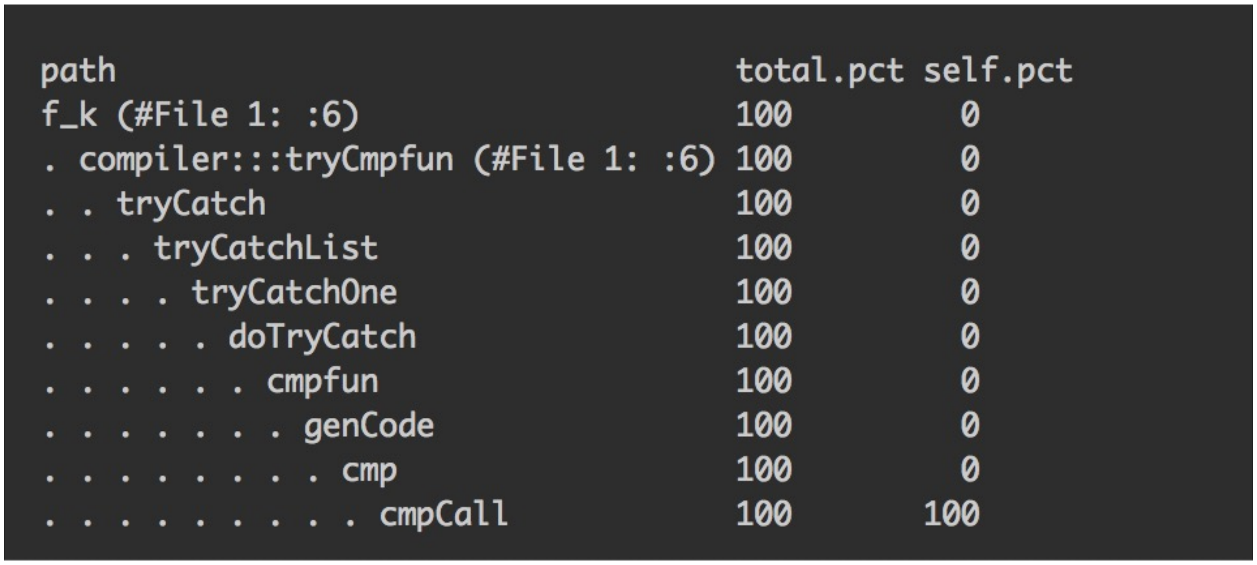
The relative timing will exceed 20 times when n is extremely large. It has been showed in ‘Rough Comparision’ section. But considering it’s time-consuming to draw a plot based on such large n, we do not show the entire result here.

(c)

To assess the steps in your code for (b) that are using the most time.

```
n<-2000;p=0.3;phi=0.5
f_k <- function(n,k,p,phi){
  log = lchoose(n,k)+(1-phi)*(k*log(k)+(n-k)*log(n-k)-n*log(n))+
    k*phi*log(p)+(n-k)*phi*log(1-p)
  return (exp(log))
}

library(proftools)
pd <- profileExpr(
  {
    for(k in 1:(n-1)){
      result_zero <- (1-p)**(n*phi)
      result_n <- p**(n*phi)
      den <- sum(f_k(n,k,p,phi))}
  }
)
hotPaths(pd)
```



```
path                                  total.pct self.pct
f_k (#File 1: :6)                     100          0
. compiler:::tryCmpfun (#File 1: :6)  100          0
. . tryCatch                          100          0
. . . tryCatchList                    100          0
. . . . tryCatchOne                   100          0
. . . . . doTryCatch                  100          0
. . . . . . cmpfun                    100          0
. . . . . . . genCode                 100          0
. . . . . . . . cmp                   100          0
. . . . . . . . . cmpCall             100        100
```

According to the result showed above, I consider the Method I has best performance in speed as my final solution.

Answer for Problem 4

Q:

This question explores memory use and copying with lists. In answering this question you can ignore what is happening with the list attributes, which are also reported by .Internal(inspect()).

(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?

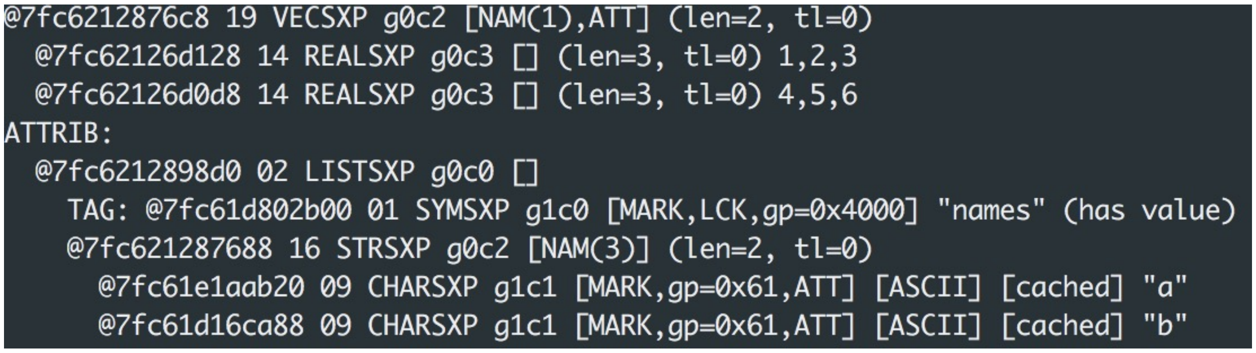
(c) Now make a list of lists. Copy the list. Add an element to the second list. Explain what is copied and what is not copied and what data is shared between the two lists.

(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.

(a)

```r
x<-list(a=c(1,2,3),b=c(4,5,6))
```

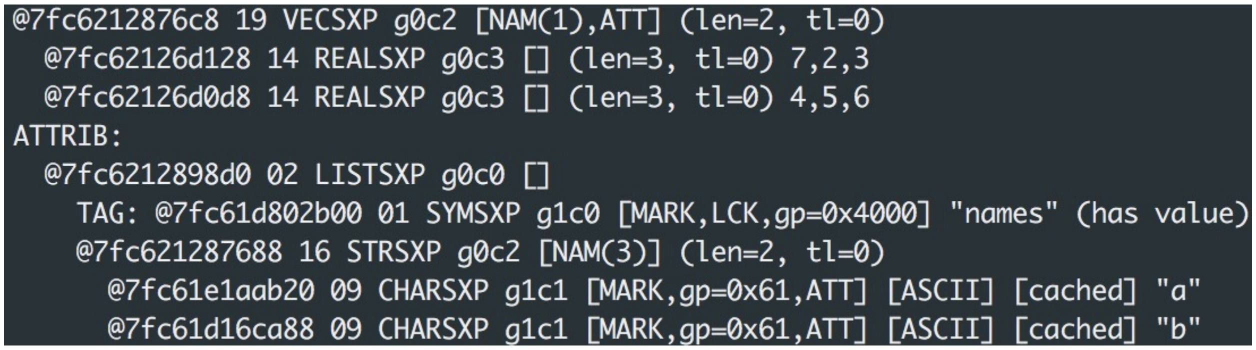
```r
.Internal(inspect(x))
```



```
@7fc6212876c8 19 VECSXP g0c2 [NAM(1),ATT] (len=2, tl=0)
  @7fc62126d128 14 REALSXP g0c3 [] (len=3, tl=0) 1,2,3
  @7fc62126d0d8 14 REALSXP g0c3 [] (len=3, tl=0) 4,5,6
ATTRIB:
  @7fc6212898d0 02 LISTSXP g0c0 []
    TAG: @7fc61d802b00 01 SYMSXP g1c0 [MARK,LCK,gp=0x4000] "names" (has value)
    @7fc621287688 16 STRSXP g0c2 [NAM(3)] (len=2, tl=0)
      @7fc61e1aab20 09 CHARSXP g1c1 [MARK,gp=0x61,ATT] [ASCII] [cached] "a"
      @7fc61d16ca88 09 CHARSXP g1c1 [MARK,gp=0x61,ATT] [ASCII] [cached] "b"
```

```r
x$a[1]<-7
```

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



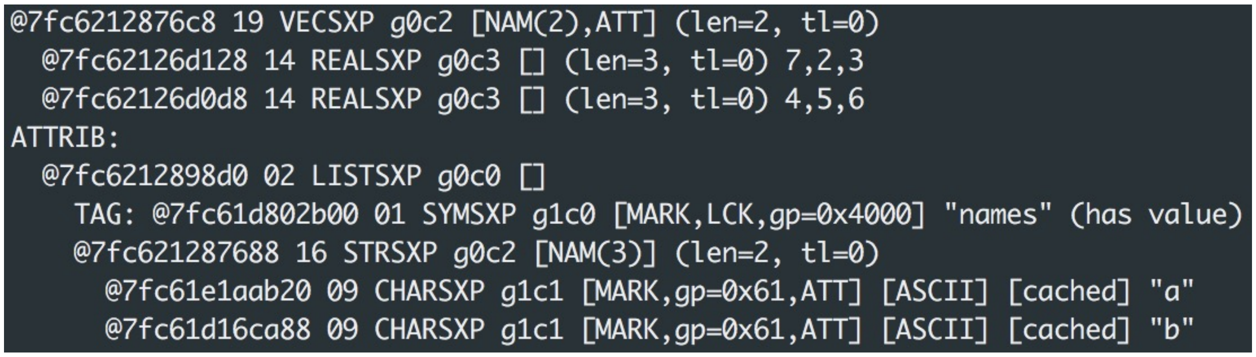
```
@7fc6212876c8 19 VECSXP g0c2 [NAM(1),ATT] (len=2, tl=0)
  @7fc62126d128 14 REALSXP g0c3 [] (len=3, tl=0) 7,2,3
  @7fc62126d0d8 14 REALSXP g0c3 [] (len=3, tl=0) 4,5,6
ATTRIB:
  @7fc6212898d0 02 LISTSXP g0c0 []
    TAG: @7fc61d802b00 01 SYMSXP g1c0 [MARK,LCK,gp=0x4000] "names" (has value)
    @7fc621287688 16 STRSXP g0c2 [NAM(3)] (len=2, tl=0)
      @7fc61e1aab20 09 CHARSXP g1c1 [MARK,gp=0x61,ATT] [ASCII] [cached] "a"
      @7fc61d16ca88 09 CHARSXP g1c1 [MARK,gp=0x61,ATT] [ASCII] [cached] "b"
```

R make the change in place without creating a new list or a new vector.

(b)

```
y<-x
```

```
.Internal(inspect(y))
```



```
@7fc6212876c8 19 VECSXP g0c2 [NAM(2),ATT] (len=2, tl=0)
  @7fc62126d128 14 REALSXP g0c3 [] (len=3, tl=0) 7,2,3
  @7fc62126d0d8 14 REALSXP g0c3 [] (len=3, tl=0) 4,5,6
ATTRIB:
  @7fc6212898d0 02 LISTSXP g0c0 []
    TAG: @7fc61d802b00 01 SYMSXP g1c0 [MARK,LCK,gp=0x4000] "names" (has value)
    @7fc621287688 16 STRSXP g0c2 [NAM(3)] (len=2, tl=0)
      @7fc61e1aab20 09 CHARSXP g1c1 [MARK,gp=0x61,ATT] [ASCII] [cached] "a"
      @7fc61d16ca88 09 CHARSXP g1c1 [MARK,gp=0x61,ATT] [ASCII] [cached] "b"
```

```
library(pryr)
address(x)
```

```
## [1] "0x7f9e1b4ffc88"
```

```
address(y)
```

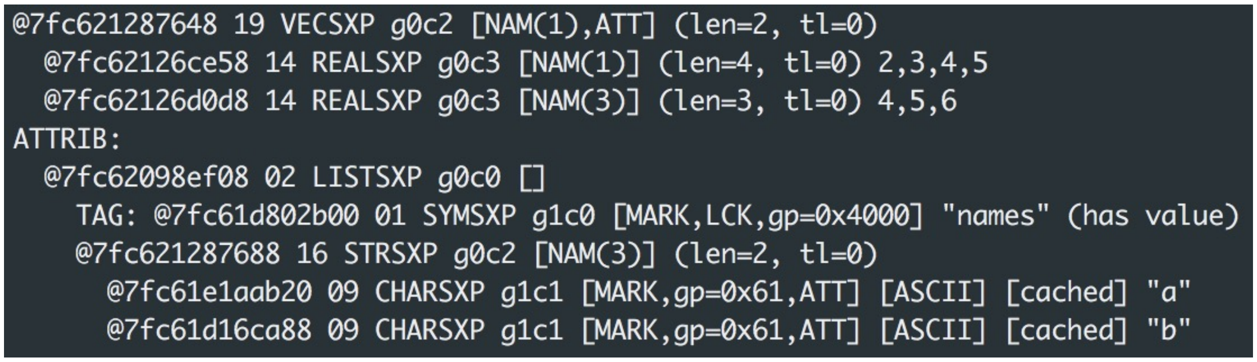
```
## [1] "0x7f9e1b4ffc88"
```

There is no copy-on-change going on up to now.

Then, we make change to one of the vectors in one of the list.

```
y$a <- c(2,3,4,5)
```

```
.Internal(inspect(y))
```



```
@7fc621287648 19 VECSXP g0c2 [NAM(1),ATT] (len=2, tl=0)
  @7fc62126ce58 14 REALSXP g0c3 [NAM(1)] (len=4, tl=0) 2,3,4,5
  @7fc62126d0d8 14 REALSXP g0c3 [NAM(3)] (len=3, tl=0) 4,5,6
ATTRIB:
  @7fc62098ef08 02 LISTSXP g0c0 []
    TAG: @7fc61d802b00 01 SYMSXP g1c0 [MARK,LCK,gp=0x4000] "names" (has value)
    @7fc621287688 16 STRSXP g0c2 [NAM(3)] (len=2, tl=0)
      @7fc61e1aab20 09 CHARSXP g1c1 [MARK,gp=0x61,ATT] [ASCII] [cached] "a"
      @7fc61d16ca88 09 CHARSXP g1c1 [MARK,gp=0x61,ATT] [ASCII] [cached] "b"
```

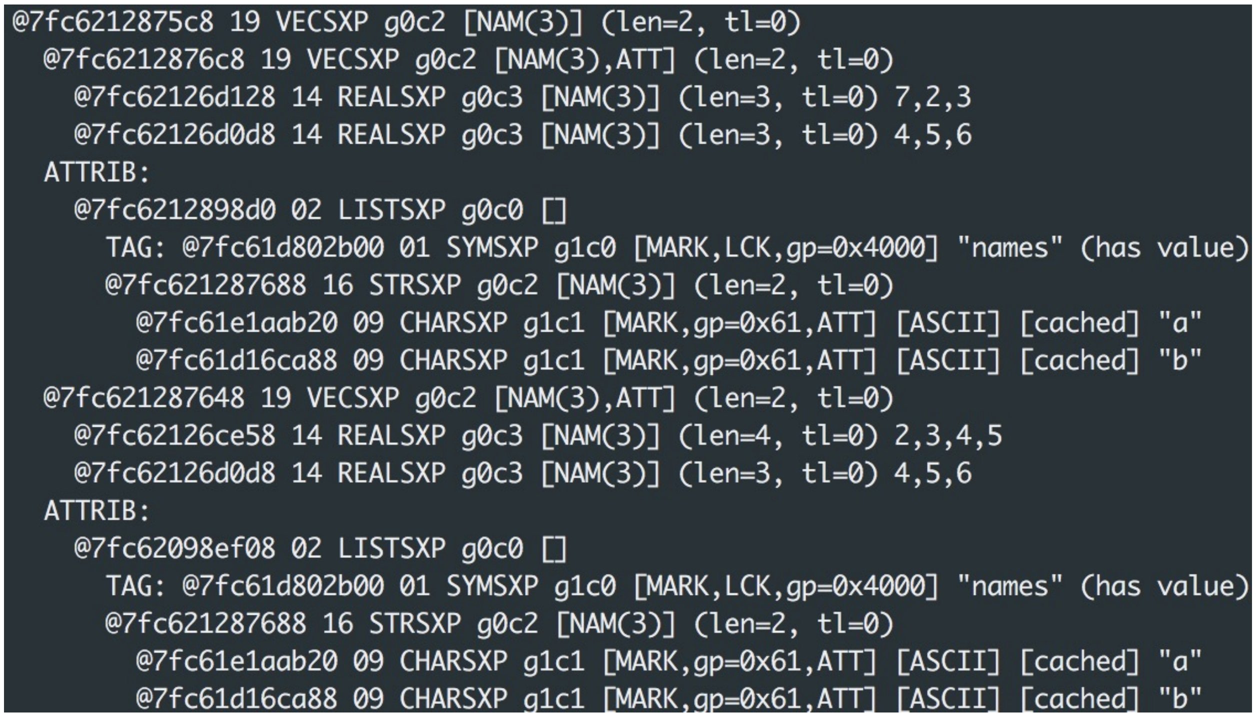
A copy of the entire list is made and the relevant vector also gets its copy with other data shared between the original version and the copy one.

(c)

```
list1<-list(x,y)
list1
```

```
## [[1]]
## [[1]]$a
## [1] 7 2 3
##
## [[1]]$b
## [1] 4 5 6
##
##
## [[2]]
## [[2]]$a
## [1] 2 3 4 5
##
## [[2]]$b
## [1] 4 5 6
```

```r
.Internal(inspect(list1))
```



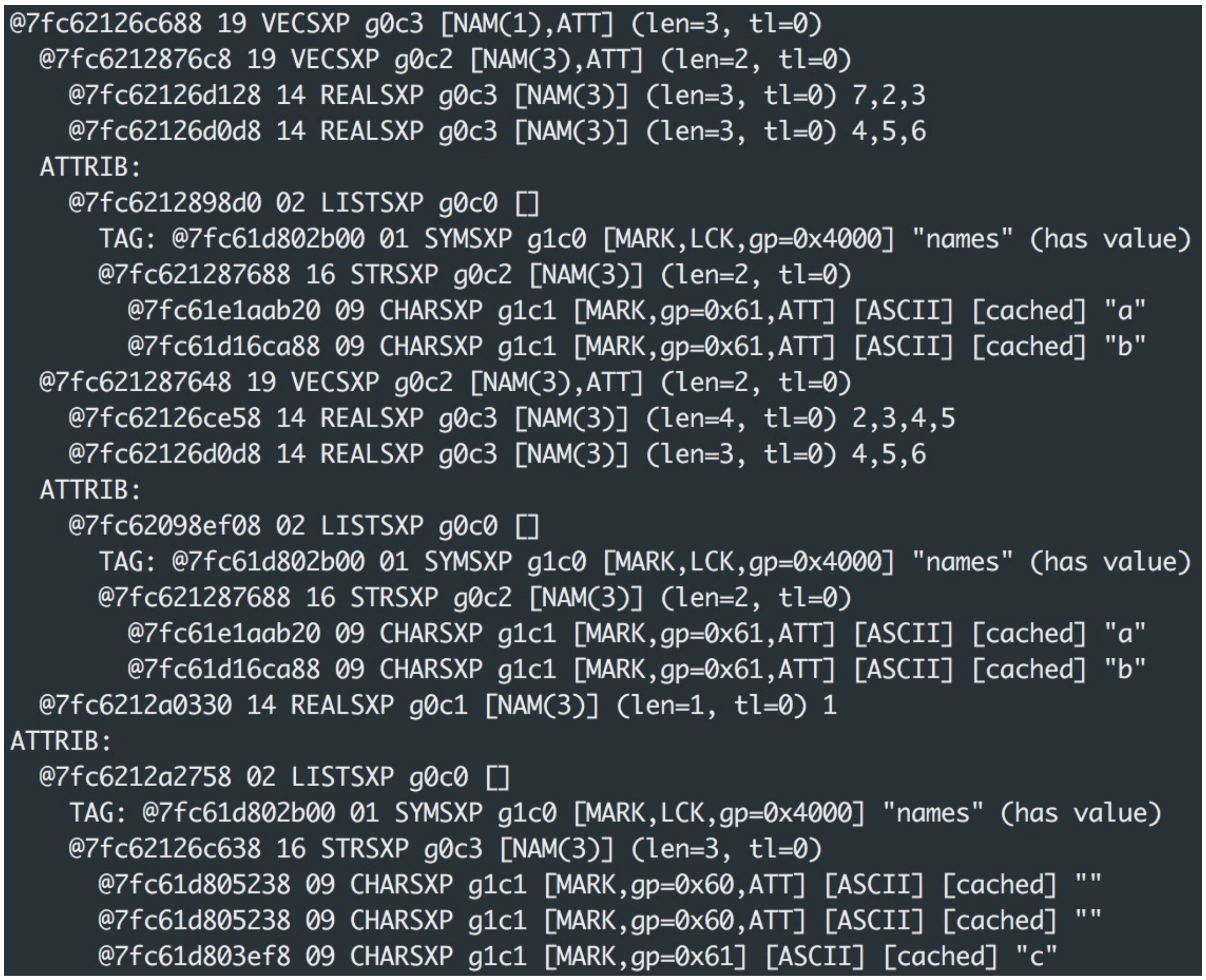
```
@7fc6212875c8 19 VECSXP g0c2 [NAM(3)] (len=2, tl=0)
  @7fc6212876c8 19 VECSXP g0c2 [NAM(3),ATT] (len=2, tl=0)
    @7fc62126d128 14 REALSXP g0c3 [NAM(3)] (len=3, tl=0) 7,2,3
    @7fc62126d0d8 14 REALSXP g0c3 [NAM(3)] (len=3, tl=0) 4,5,6
  ATTRIB:
    @7fc6212898d0 02 LISTSXP g0c0 []
      TAG: @7fc61d802b00 01 SYMSXP g1c0 [MARK,LCK,gp=0x4000] "names" (has value)
      @7fc621287688 16 STRSXP g0c2 [NAM(3)] (len=2, tl=0)
        @7fc61e1aab20 09 CHARSXP g1c1 [MARK,gp=0x61,ATT] [ASCII] [cached] "a"
        @7fc61d16ca88 09 CHARSXP g1c1 [MARK,gp=0x61,ATT] [ASCII] [cached] "b"
  @7fc621287648 19 VECSXP g0c2 [NAM(3),ATT] (len=2, tl=0)
    @7fc62126ce58 14 REALSXP g0c3 [NAM(3)] (len=4, tl=0) 2,3,4,5
    @7fc62126d0d8 14 REALSXP g0c3 [NAM(3)] (len=3, tl=0) 4,5,6
  ATTRIB:
    @7fc62098ef08 02 LISTSXP g0c0 []
      TAG: @7fc61d802b00 01 SYMSXP g1c0 [MARK,LCK,gp=0x4000] "names" (has value)
      @7fc621287688 16 STRSXP g0c2 [NAM(3)] (len=2, tl=0)
        @7fc61e1aab20 09 CHARSXP g1c1 [MARK,gp=0x61,ATT] [ASCII] [cached] "a"
        @7fc61d16ca88 09 CHARSXP g1c1 [MARK,gp=0x61,ATT] [ASCII] [cached] "b"
```

```r
list2<-list1
list2$c<-1
list2
```

```
## [[1]]
## [[1]]$a
## [1] 7 2 3
##
## [[1]]$b
## [1] 4 5 6
##
##
## [[2]]
## [[2]]$a
## [1] 2 3 4 5
##
```

```
## [[2]]$b
## [1] 4 5 6
##
##
## $c
## [1] 1
```

```
.Internal(inspect(list2))
```



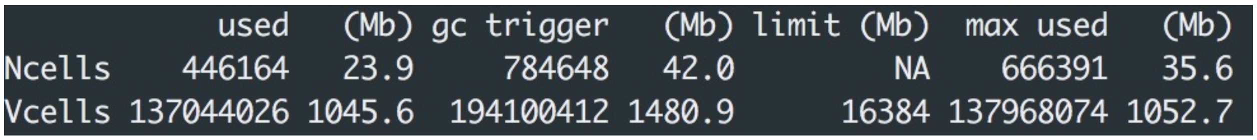
```
@7fc62126c688 19 VECSXP g0c3 [NAM(1),ATT] (len=3, tl=0)
  @7fc6212876c8 19 VECSXP g0c2 [NAM(3),ATT] (len=2, tl=0)
    @7fc62126d128 14 REALSXP g0c3 [NAM(3)] (len=3, tl=0) 7,2,3
    @7fc62126d0d8 14 REALSXP g0c3 [NAM(3)] (len=3, tl=0) 4,5,6
  ATTRIB:
    @7fc6212898d0 02 LISTSXP g0c0 []
      TAG: @7fc61d802b00 01 SYMSXP g1c0 [MARK,LCK,gp=0x4000] "names" (has value)
      @7fc621287688 16 STRSXP g0c2 [NAM(3)] (len=2, tl=0)
        @7fc61e1aab20 09 CHARSXP g1c1 [MARK,gp=0x61,ATT] [ASCII] [cached] "a"
        @7fc61d16ca88 09 CHARSXP g1c1 [MARK,gp=0x61,ATT] [ASCII] [cached] "b"
  @7fc621287648 19 VECSXP g0c2 [NAM(3),ATT] (len=2, tl=0)
    @7fc62126ce58 14 REALSXP g0c3 [NAM(3)] (len=4, tl=0) 2,3,4,5
    @7fc62126d0d8 14 REALSXP g0c3 [NAM(3)] (len=3, tl=0) 4,5,6
  ATTRIB:
    @7fc62098ef08 02 LISTSXP g0c0 []
      TAG: @7fc61d802b00 01 SYMSXP g1c0 [MARK,LCK,gp=0x4000] "names" (has value)
      @7fc621287688 16 STRSXP g0c2 [NAM(3)] (len=2, tl=0)
        @7fc61e1aab20 09 CHARSXP g1c1 [MARK,gp=0x61,ATT] [ASCII] [cached] "a"
        @7fc61d16ca88 09 CHARSXP g1c1 [MARK,gp=0x61,ATT] [ASCII] [cached] "b"
  @7fc6212a0330 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 1
ATTRIB:
  @7fc6212a2758 02 LISTSXP g0c0 []
    TAG: @7fc61d802b00 01 SYMSXP g1c0 [MARK,LCK,gp=0x4000] "names" (has value)
    @7fc62126c638 16 STRSXP g0c3 [NAM(3)] (len=3, tl=0)
      @7fc61d805238 09 CHARSXP g1c1 [MARK,gp=0x60,ATT] [ASCII] [cached] ""
      @7fc61d805238 09 CHARSXP g1c1 [MARK,gp=0x60,ATT] [ASCII] [cached] ""
      @7fc61d803ef8 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "c"
```

There is a copy of the entire list. Shared data between the two lists includes other original internal lists and the names of internal lists. These data is not copied. list2$c has a distict memory reference showed above.

Particularly, there is another attribution appearing in the result of new list. It shows that the information of names of external list, which does not appear in former inspect information.

(d)

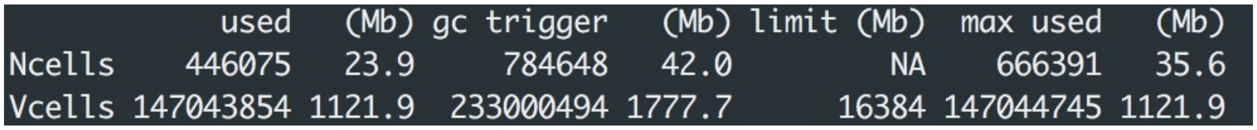
```
gc()
```



```
          used   (Mb) gc trigger   (Mb) limit (Mb)  max used   (Mb)
Ncells    446164   23.9     784648   42.0         NA    666391   35.6
Vcells 137044026 1045.6  194100412 1480.9      16384 137968074 1052.7
```

```r
tmp <- list()
x <- rnorm(1e7)
tmp[[1]] <- x
tmp[[2]] <- x
```

```r
gc()
```



```
          used   (Mb) gc trigger   (Mb) limit (Mb)  max used   (Mb)
Ncells    446075   23.9     784648   42.0         NA    666391   35.6
Vcells 147043854 1121.9  233000494 1777.7      16384 147044745 1121.9
```

The difference between two results of 'gc()' shows that only ~80 MB is used. However, when we execute following code:

```r
.Internal(inspect(tmp))
```

```
## @7f9e1acef7c8 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
##   @10df1a000 14 REALSXP g0c7 [NAM(3)] (len=10000000, tl=0) 0.172943,-0.444284,0.416906,-1.64464,-1.0
##   @10df1a000 14 REALSXP g0c7 [NAM(3)] (len=10000000, tl=0) 0.172943,-0.444284,0.416906,-1.64464,-1.0
```

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

```
## 160000160 bytes
```

The result of 'object.size()' shows that 'tmp' has size of 160MB.

Explanation:

If we use 'object_size()' instead of 'object.size()', we get right answer.

```r
object_size(tmp)
```

```
## 80 MB
```

This function is better than the built-in object.size() because it accounts for shared elements within an object and includes the size of environments. That is, 'object.size()' function does not accounts for shared elements within an object, which is the case in this question. The result of '.Internal(inspect(tmp))' suggests that memory references of two elements in 'tmp' is the same. In other words, 'tmp[[1]]' and 'tmp[[2]]' are shared elements with in 'tmp'. They don't take memory repeatedly. However, 'object.size()' counts twice.

Answer for Problem 5

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

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

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

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

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

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

- Explanation

The following cade helps explain:

```r
set.seed(1)
rnorm(1)
```

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

```r
rnorm(1)
```

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

The seed number you choose is just the starting point used in the generation of a sequence of random numbers. So once we set our seed number in a certain environment (Global environment in this case), we can get the same sequence of 'random' numbers from the start point when calling 'rnorm' several times.

And then, we try the following code:

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

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

From the code chunk above, we can realize the fact that if we set the seed number anew, we can get the same number '-0.6264538' as the result of first two chunks in this question because it corresponds the first 'random' number of seed(1).

When we call 'tmp()' function, it is called in global environment, so when the code encounter 'print(rnorm(1))'

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

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

The function 'rnorm()' and 'set.seed()' are both defined in global environment. So when we call 'tmp()' function, it calls 'rnorm()' which defined in global environment. The behaviour of 'rnorm()' will follow the random state of global environment. And the 'load('tmp.Rda')'changes random state of the frame of 'tmp()' which is a local environment.

We can try the following chunk to validate the conclusion.

```r
tmp123 <- function() {
  set.seed(123)
  print(rnorm(1))
}
tmp123()
```

```
## [1] -0.5604756
```

```r
rnorm(1)
```

```
## [1] -0.2301775
```

It shows that the random state has been changed by ‘set.seed(123)’ inside the frame of ‘tmp123()’ because ‘set.seed()’ is also defined in global environment. That is, when ‘set.seed()’ is called inside of ‘tmp123()’, it changes the random state of global environment. By contrast, ‘load(’tmp.Rda’)’ only changes the random state of local environment.