ps3

Ramon Crespo

10/2/2018

Problem1

1. 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?

General Description make_container -> Takes in a variable n, assigns it as a numeric value to the variable x. Generates a variable i and assigns it the value 1. A new environment is generated when the new function is called. It will then return the value of the x if the condition that value is null is satisfied. Since value is not assigned at any point during the code, we can assume that this will allways be the case and the program will retur the value of x. If the value function is not empty, then the x[i] in the outer function will be filled with the value of i. bootmeans _> Calls the make_container environment and stores the output of this environment

What are the various enclosing environments? There are two enclosing environments, the first is the outer function make_container. The second environment is a function inside of this environment that just assigns NULL to the vairable value and then checks an if statement.

What happens when one excecutes bootmeans()? A new enclosing environment is generated to evaluate the function make_container. This function contains another enclosing environment which performs some other analysis. In short, when you excecute bootmeans a closing environment is generated (and then another enclosing environment within the original enclosing environment is generated), some calculations are generated and the outcome is stored in boormeans. What happens inside the closing environments is erased.

In what sense is this a function that contains data? make_container is a function receives data and then gives you a brief analysis of the data that was passed to it. The data that is passed into the make_container function disapears after the analysis is done. It just serches for data, compiles the results and then the results are stored in bootmeans. But means becomes a variable that acts as the key to access the results from the analysis that was done by make_container.

How much memory does bootmeans use if n=10000000? 30568 bytes according to the program. The value of n does not necesarily mean a larger value of bootmeans, because bootmeans just compiles the results of the analysis. So everything depends on the internal analysis done in the enclosing environment that is generated when the function is called.

```
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 <- 10000000
```

```
bootmeans <- make_container(nboot)
object.size(bootmeans)
```

```
## 13520 bytes
```

```
data <- faithful[ , 1] # Old Faithful geyser eruption lengths for (i in 1:nboot)
t<-bootmeans(mean(sample(data, length(data), replace=TRUE)))

object.size(t)
```

```
## 56 bytes
```

Problem2. In my computer, just by taking the sample n number of times wihout looping improved the performance of the loop and the time decreased from .685 to .17. The main difference is changin what was a row operation to a column operation. Looping takes time, so a rule of thumb is that the less you do it the more efficient the calculations.

```
n <- 100000
p <- 5 ## number of categories
p1 <- 1
## efficient vectorized way to generate a random matrix of ## row-normalized probabilities:
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)
system.time(
for(i in seq_len(n))
  smp[i] <- sample(p, 1, prob = probs[i, ])
)
```

```
##    user  system elapsed
##   0.641   0.052   0.699
```

```
temp <- runif(n)
m <- matrix( , nrow = n, ncol = 5)
cumulative_probs <- matrix(0, nrow = n, ncol = 1)
count <- 0
system.time(
for(i in seq_len(p)){
  #print(tmp2[i])
  cumulative_probs <- cumulative_probs + probs[,i]
  m[,i] <- cumulative_probs[i]<temp
  count <- count + 1
}
)
```

```
##    user  system elapsed
##   0.016   0.005   0.023
```

Problem 3 Question a When calculations are not done using the log function we get values that approach cero. This means that it gets to a point where the function will not work as the value will approach zero, the program will not be able to capture this precision so it will rount to cero and the function will not work as it will try to divide by zero and will just compute infinity

```
p = .3
phi = .5
#n <- matrix(seq(1,2000,100))
```

```
n <- matrix(2000,nrow = 1, ncol = 1)
current_value <- 0
system.time(
t<-apply(n,2,function(n){
  current_value <- 0
  for (k in 1:n){
      if (k != n){
        if (k == 0){
            factorial_term <- log(lchoose(n,k))
            second_term <- k*log(k)+ (n-k)*log(n-k) - n*log(n)
            third_term <- n*log(n) - phi*k*log(k) - phi*(n-k)*log(n-k)
            fourth_term <- k*phi*log(p)
            fifth_term <- (n-k)*phi*log(1-p)
            current_value <-factorial_term + second_term + third_term + fourth_term + fifth_term
        } else{
           factorial_term <- log(lchoose(n,k))
           second_term <- k*log(k)+ (n-k)*log(n-k) - n*log(n)
           third_term <- n*log(n) - phi*k*log(k) - phi*(n-k)*log(n-k)
           fourth_term <- k*phi*log(p)
           fifth_term <- (n-k)*phi*log(1-p)
           current_value <- current_value + factorial_term + second_term + third_term + fourth_term + f
  }
  }
  }
  return(current_value)
 }
)
)
```

```
##    user  system elapsed
##   0.043   0.001   0.045
```

```
t
```

```
## [1] 12647871
```

Question b Below is an implementation of the same calculation using vector format. The results was .014 faster for a single calculation . If the analysis is replicated many times, the vectorized implementation has the potential to save a decent amount of time and computer power.

```
p = .3
phi = .5
k <- matrix(seq(1,1999,1))
n <- matrix(2000,nrow = 1999, ncol = 1)
ugly_denominator <- function(n,k){
  current_value <- 0
  factorial_term <- log(lchoose(n,k))
  second_term <- k*log(k)+ (n-k)*log(n-k) - n*log(n)
  third_term <- n*log(n) - phi*k*log(k) - phi*(n-k)*log(n-k)
  fourth_term <- k*phi*log(p)
  fifth_term <- (n-k)*phi*log(1-p)
  return(sum(factorial_term+second_term+third_term+fourth_term+fifth_term))
}
system.time(
t <- ugly_denominator(n,k)
)
```

```
##    user  system elapsed
##   0.015   0.000   0.016
```

```
t
```

```
## [1] 12647871
```

Problem 4 Question a. Can R make changes in place?

No, below we see how a simple list has to be rewriten into another object to create the changes. It is not inplace as the vector address is changed. The location for matrix is – @119371188 The location of a specific entry is – @11b2421f0 Then a change is made inside of the matrix and the locations are The location for matrix is – @103fcf218 The location of a specific entry is – @107224018 The tag remains the same

```
n <- 5
p <- 5 ## number of categories
p1 <- 1
## efficient vectorized way to generate a random matrix of ## row-normalized probabilities:
tmp <- exp(matrix(rnorm(n*p), nrow = n, ncol = p))
probs <- tmp / rowSums(tmp)
smp <- rep(0, n)
temp <- runif(n)
m <- matrix( , nrow = n, ncol = 5)
cumulative_probs <- matrix(0, nrow = n, ncol = 1)
count <- 0
for(i in seq_len(p)){
  #print(tmp2[i])
  cumulative_probs <- cumulative_probs + probs[,i]
  m[,i] <- cumulative_probs[i]<temp
  count <- count + 1
}
m
```

```
##       [,1]  [,2]  [,3]  [,4]  [,5]
## [1,]  TRUE  TRUE FALSE FALSE FALSE
## [2,]  TRUE  TRUE FALSE FALSE FALSE
## [3,] FALSE FALSE FALSE FALSE FALSE
## [4,]  TRUE  TRUE  TRUE FALSE FALSE
## [5,]  TRUE  TRUE FALSE FALSE FALSE
```

```
print("the location for matrix is")
```

```
## [1] "the location for matrix is"
```

```
.Internal(inspect(m))
```

```
## @7f84f81549f8 10 LGLSXP g0c5 [NAM(3),ATT] (len=25, tl=0) 1,1,0,1,1,...
## ATTRIB:
##   @7f84f6ce5c00 02 LISTSXP g0c0 []
##     TAG: @7f84f603e080 01 SYMSXP g1c0 [MARK,LCK,gp=0x4000] "dim" (has value)
##     @7f84fa2905f0 13 INTSXP g0c1 [NAM(3)] (len=2, tl=0) 5,5
```

```
print("the location for the 0 row column 1 is")
```

```
## [1] "the location for the 0 row column 1 is"
```

```
.Internal(inspect(m[0,]))
```

```
## @7f84f65ef238 10 LGLSXP g0c0 [ATT] (len=0, tl=0)
## ATTRIB:
```

```
##   @7f84f65ef270 02 LISTSXP g0c0 []
##     TAG: @7f84f603e080 01 SYMSXP g1c0 [MARK,LCK,gp=0x4000] "dim" (has value)
##     @7f84f674f700 13 INTSXP g0c1 [NAM(3)] (len=2, tl=0) 0,5
```

```
print("now lets make a change in the value from FALSE to TRUE")
```

```
## [1] "now lets make a change in the value from FALSE to TRUE"
```

```
m[0,1] <- TRUE

print("the new location of the matrix is")
```

```
## [1] "the new location of the matrix is"
```

```
.Internal(inspect(m))
```

```
## @7f84f83aa8c8 10 LGLSXP g0c5 [NAM(1),ATT] (len=25, tl=0) 1,1,0,1,1,...
## ATTRIB:
##   @7f84f6d20c00 02 LISTSXP g0c0 []
##     TAG: @7f84f603e080 01 SYMSXP g1c0 [MARK,LCK,gp=0x4000] "dim" (has value)
##     @7f84fa2905f0 13 INTSXP g0c1 [NAM(3)] (len=2, tl=0) 5,5
```

```
print("the new location of the component is")
```

```
## [1] "the new location of the component is"
```

```
.Internal(inspect(m[0,]))
```

```
## @7f84fb426e88 10 LGLSXP g0c0 [ATT] (len=0, tl=0)
## ATTRIB:
##   @7f84fb426e50 02 LISTSXP g0c0 []
##     TAG: @7f84f603e080 01 SYMSXP g1c0 [MARK,LCK,gp=0x4000] "dim" (has value)
##     @7f84f6bdf900 13 INTSXP g0c1 [NAM(3)] (len=2, tl=0) 0,5
```

Question 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 you create a copy of the same vector, matrix (list of lists) a copy is not creater, rather R points both variables to the same memory. When a change is made to one of the vectors then the vectors (or matric or list of lists) are separated into two separates position (eachone has a different location on the memory). Bolow is the implementation of the code using .Interna(inspect()) to obtain the location and variables.

```
n <- 5
p <- 5 ## number of categories
p1 <- 1
## efficient vectorized way to generate a random matrix of ## row-normalized probabilities:
tmp <- exp(matrix(rnorm(n*p), nrow = n, ncol = p))
probs <- tmp / rowSums(tmp)
smp <- rep(0, n)
temp <- runif(n)
m <- matrix( , nrow = n, ncol = 5)
cumulative_probs <- matrix(0, nrow = n, ncol = 1)
count <- 0
for(i in seq_len(p)){
  #print(tmp2[i])
  cumulative_probs <- cumulative_probs + probs[,i]
  m[,i] <- cumulative_probs[i]<temp
  count <- count + 1
}
```

```
y <-m
.Internal(inspect(m))
```

```
## @7f84f8b18268 10 LGLSXP g0c5 [NAM(3),ATT] (len=25, tl=0) 1,1,1,1,1,...
## ATTRIB:
##   @7f84fb2345e0 02 LISTSXP g0c0 []
##     TAG: @7f84f603e080 01 SYMSXP g1c0 [MARK,LCK,gp=0x4000] "dim" (has value)
##     @7f84f76e1508 13 INTSXP g0c1 [NAM(3)] (len=2, tl=0) 5,5
```

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

```
## @7f84f8b18268 10 LGLSXP g0c5 [NAM(3),ATT] (len=25, tl=0) 1,1,1,1,1,...
## ATTRIB:
##   @7f84fb2345e0 02 LISTSXP g0c0 []
##     TAG: @7f84f603e080 01 SYMSXP g1c0 [MARK,LCK,gp=0x4000] "dim" (has value)
##     @7f84f76e1508 13 INTSXP g0c1 [NAM(3)] (len=2, tl=0) 5,5
```

```
print("now lets make a change to the whole 2nd row")
```

```
## [1] "now lets make a change to the whole 2nd row"
```

```
y[,1] <- c(1,2,3,4,5)
.Internal(inspect(y))
```

```
## @7f84f5df0c20 14 REALSXP g0c7 [NAM(1),ATT] (len=25, tl=0) 1,2,3,4,5,...
## ATTRIB:
##   @7f84fb064a88 02 LISTSXP g0c0 []
##     TAG: @7f84f603e080 01 SYMSXP g1c0 [MARK,LCK,gp=0x4000] "dim" (has value)
##     @7f84f76e1508 13 INTSXP g0c1 [NAM(3)] (len=2, tl=0) 5,5
```

Question 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 the list of list is generated it is given a specifi location on the disc. Then we make a copy of this list and no new memory is allocated, instead the new variable just maps to the same location as the original memory. Then we append a value to the list of lists. When we append a new list, the location of the list is changed, nonetheless the list that where copied to a new variable still keep their location. In other words, lets call the list of lists "A" and the vectors inside this list "a" "b". When you copy "A" into "B", no new memory is generated and both variables point to the same memory. When you add "c" to "B", then "B" gets a new assignment in memory, nontheless "a" and "b" still point to the same location (same location as what they point to in "A") and "c" is given a new space on memory.

```
a <- list(1,2,3)
b <- list(4,5,6)
e <- list(1,2,5)
c <- append(list(a), list(b))
print("below the information of c")
```

```
## [1] "below the information of c"
```

```
.Internal(inspect(c))
```

```
## @7f84f85cfc08 19 VECSXP g0c2 [NAM(3)] (len=2, tl=0)
##   @7f84fad4a7a8 19 VECSXP g0c3 [NAM(3)] (len=3, tl=0)
##     @7f84fa34e848 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 1
##     @7f84fa34e880 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 2
##     @7f84fa34e8b8 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 3
##   @7f84fad4be28 19 VECSXP g0c3 [NAM(3)] (len=3, tl=0)
##     @7f84fa34e928 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 4
```

```
##     @7f84fa34e960 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 5
##     @7f84fa34e998 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 6
```

```
d <- c
print("below the information of d")
```

```
## [1] "below the information of d"
```

```
.Internal(inspect(d))
```

```
## @7f84f85cfc08 19 VECSXP g0c2 [NAM(3)] (len=2, tl=0)
##   @7f84fad4a7a8 19 VECSXP g0c3 [NAM(3)] (len=3, tl=0)
##     @7f84fa34e848 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 1
##     @7f84fa34e880 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 2
##     @7f84fa34e8b8 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 3
##   @7f84fad4be28 19 VECSXP g0c3 [NAM(3)] (len=3, tl=0)
##     @7f84fa34e928 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 4
##     @7f84fa34e960 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 5
##     @7f84fa34e998 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 6
```

```
d <- append(list(d),list(e))
print("below the information of d after change")
```

```
## [1] "below the information of d after change"
```

```
.Internal(inspect(d))
```

```
## @7f84fa0b2ec8 19 VECSXP g0c2 [NAM(3)] (len=2, tl=0)
##   @7f84f85cfc08 19 VECSXP g0c2 [NAM(3)] (len=2, tl=0)
##     @7f84fad4a7a8 19 VECSXP g0c3 [NAM(3)] (len=3, tl=0)
##       @7f84fa34e848 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 1
##       @7f84fa34e880 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 2
##       @7f84fa34e8b8 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 3
##     @7f84fad4be28 19 VECSXP g0c3 [NAM(3)] (len=3, tl=0)
##       @7f84fa34e928 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 4
##       @7f84fa34e960 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 5
##       @7f84fa34e998 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 6
##   @7f84fad062a8 19 VECSXP g0c3 [NAM(3)] (len=3, tl=0)
##     @7f84fa34ea08 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 1
##     @7f84fa34ea40 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 2
##     @7f84fa34ea78 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 5
```

Question 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.

The conflicts arrise because the size of the object is close to 160MB, as object.size(tmp) states, nonetheless the two vectors that make up tmp are the same, thus R will allocate the same memory space to both vectors. This decreases the amount of memory needed from 160 to 80MB. Proof of this is changin one of the lists, thus forcing R to generate a different copy in each of the arrays. By doing this R will be forced to allocate new memory to each of the lists making it 160MB of size.

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

```
## @7f84fc190f48 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
```

```
##   @116f67000 14 REALSXP g0c7 [NAM(3)] (len=10000000, tl=0) 1.6725,1.07992,0.576919,-0.341494,-0.5212
##   @116f67000 14 REALSXP g0c7 [NAM(3)] (len=10000000, tl=0) 1.6725,1.07992,0.576919,-0.341494,-0.5212
```

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

```
## 160000160 bytes
```