PS4

ShubeiWang

10/1/2018

1

When one runs make_container(n) it returns a function. The enclosing environment of make_container() is the global environment and that of bootmeans() is the execution environment of make_container().

In the loop, every time one runs *bootmeans(mean(sample(data, length(data), replace=TURE)))*, x[i] is assigned the mean of the samples from data and i is assigned i+1. So after this process when we executes *bootmeans()* it returns a vector x of length n which is in the enclosing environment. x[i] stores the mean of samples from data which is generated in the ith time of the loop. In that sense this function contains the data.

If n = 1000000, in the enclosing environment of bootmeans(), x is assigned a vector of length 1000000 and i is assigned 1 so they'll take about 1000001*8 = 8000008 bytes ≈ 8 MB.

2

For this question I used the cmf matrix and a uniform random vector to generate the random samples in a more vectorized way.

```
n <- 100000
p <- 5
tmp <- exp(matrix(rnorm(n*p), nrow = n, ncol = p))
probs <- tmp/rowSums(tmp)
cmf <- apply(probs,1,cumsum)
smp <- rep(0,n)
smp1 <- rep(1,n)
r<-runif(100000)

## orginal solution
# set.seed(1)
# for(i in seq_len(n))
#  smp[i] <- sample(p,1,prob = probs[i,])

## my solution
# r<-runif(100000)
# for(i in 1:5) smp1 <- smp1+(cmf[i,]<r)*1

library(rbenchmark)
benchmark(
  # orginal solution
  for(i in seq_len(n)) smp[i] <- sample(p,1,prob = probs[i,]),

  # my solution
  for(i in 1:5) smp1 <- smp1+(cmf[i,]<r)*1,

  replications = 10,
  columns = c( "test", "replications", "elapsed")
```

```
)
```

```
##                                                                  test
## 2               for (i in 1:5) smp1 <- smp1 + (cmf[i, ] < r) * 1
## 1 for (i in seq_len(n)) smp[i] <- sample(p, 1, prob = probs[i, ])
##   replications elapsed
## 2           10   0.111
## 1           10   4.360
```

3

(a)

Take log on both side we have:

$log(f(k;n,p,\phi)) = log(\binom{n}{k}) + k(1-\phi)log(k) + (n-k)(1-\phi)log(n-k) - n(1-\phi)log(n) + k\phi log(p) + (n-k)\phi log(1-p)$

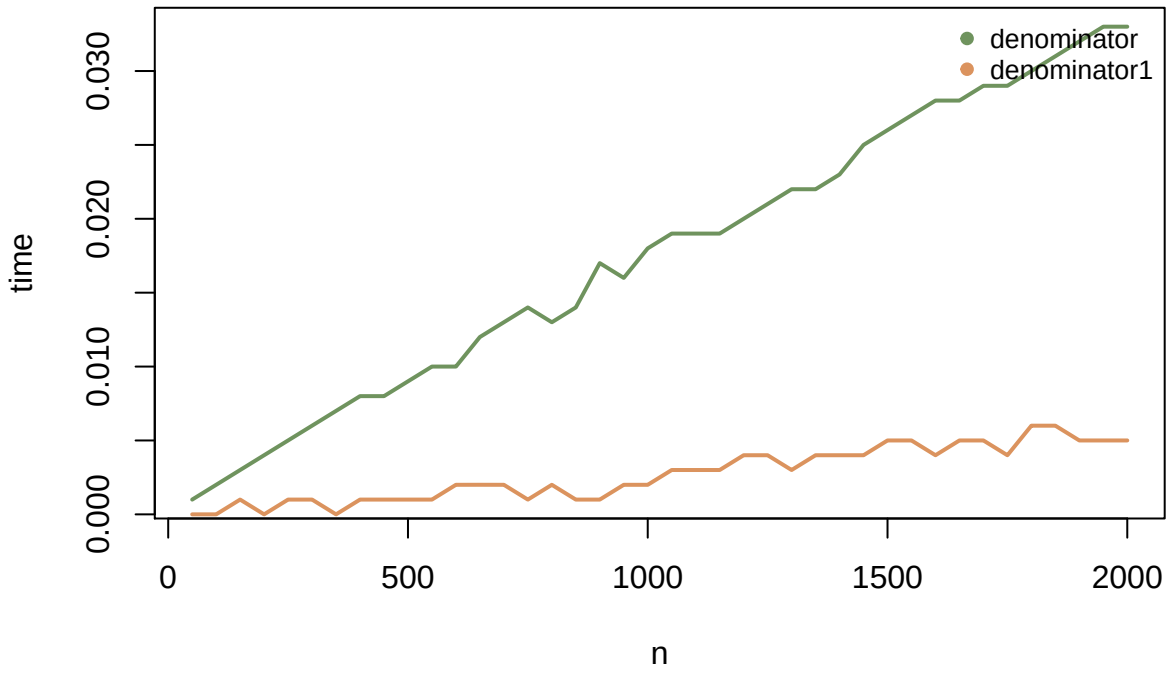
I used this expression to evaluate the denominator. If the calculation is not done on the log scale, we can encounter overflows or underflows when n is large. For example, there is a n^n term in $f(k;n,p,\phi)$. If n =200 this term cannot be calculated since 200^200 is Inf in R. Also $1/(n^n)$ will be 0 when n is large and that will lead to wrong calculations.

```
## evaluate the denominator
denominator <- function(n,p,phi){
  # create a vector for future use
  iterator <- 0:n

  ## calculate f(k;n,p,phi)
  pmf <- function(k,n,p,phi){
    # when k=0 or n, log(k) or log(n-k) will be -Inf, so we need to
    # evaluate these two terms seperately
    if(k==0) return((1-p)^(n*phi))
    else if(k==n) return(p^(n*phi))

    lg <- log(choose(n,k))+k*(1-phi)*log(k)+(n-k)*(1-phi)*log(n-k)-
      n*(1-phi)*log(n)+k*phi*log(p)+(n-k)*phi*log(1-p)
    value <- exp(lg)
    return(value)
  }

  # calculate the denominator
  denominator <- sum(sapply(iterator,pmf,n,p,phi))
  return(denominator)
}
```

(b)

For this question I wrote another function to calculate the denominator in a fully vectorized way and timed both functions with n = 50 to 1950, by step 50. The results are as follows. We can see that the time *denominator* uses is much greater than *denominator1*.

```r
## calculate the denominator in a fully vectorized way
denominator1 <- function(n,p,phi){
  if(n==1) return((1-p)^phi+p^phi)

  #calculate f(k;n,p,phi) for k from 2 to n-1
  temp <- 1:(n-1)
  term1 <- log(choose(n,temp))
  term2 <- temp*(1-phi)*log(temp)
  term3 <- (n-temp)*(1-phi)*log(n-temp)
  term4 <- n*(1-phi)*log(n)
  term5 <- temp*phi*log(p)
  term6 <- (n-temp)*phi*log(1-p)
  temp <- term1+term2+term3-term4+term5+term6
  value <- c((1-p)^(n*phi), exp(temp), p^(n*phi))
  denominator <- sum(value)
  return(denominator)
}

library(rbenchmark)

time_denom <- c()
time_denom1 <- c()
index <- 1

for(i in seq(50,2000,50)){
  time_denom[index] <- benchmark(denominator(i,0.3,0.5), replications = 10,  columns = c("test", "repli
  time_denom1[index] <- benchmark(denominator1(i,0.3,0.5), replications = 10,  columns = c("test", "rep
  index <- index+1
}

plot(time_denom~seq(50,2000,50), col=rgb(0.2,0.4,0.1,0.7),type='l', lwd=2, pch=17,xlab="n", ylab="time"
lines(time_denom1~seq(50,2000,50),col=rgb(0.8,0.4,0.1,0.7),lwd=2,pch=19)
legend("topright",legend=c("denominator","denominator1"),
       col= c(rgb(0.2,0.4,0.1,0.7), rgb(0.8,0.4,0.1,0.7)),
       pch=c(19,19),
       bty="n",
       cex=0.8)
```



(c)

```
## assess the time cosumption of each step

file <- "denominator1.out"
Rprof(file, interval = 0.0001)
# calculate f(1000,0.3,0.5) as an example
n <- 1000
p <- 0.3
phi <- 0.5
{
  temp <- 1:(n-1)
  term1 <- log(choose(n,temp))
  term2 <- temp*(1-phi)*log(temp)
  term3 <- (n-temp)*(1-phi)*log(n-temp)
  term4 <- n*(1-phi)*log(n)
  term5 <- temp*phi*log(p)
  term6 <- (n-temp)*phi*log(1-p)
  temp <- term1+term2+term3-term4+term5+term6
  value <- c((1-p)^(n*phi), exp(temp), p^(n*phi))
  denominator <- sum(value)
}
Rprof(NULL)
head(summaryRprof(file)$by.total,10)


## result
#                          total.time   total.pct
#"<Anonymous>"               0.018       97.34
#"tryCatch"                  0.016       82.98
```

```
#"doTryCatch"                  0.015        81.38
#"tryCatchList"               0.015       81.38
#"tryCatchOne"                0.015       81.38
#".rs.valueFromStr"           0.014       73.94
#".rs.withTimeLimit"            0.014        73.40
#"capture.output"               0.013        67.02
#"eval"                       0.013       67.02
#".rs.valueDescription"       0.010       51.06
```

4

(a)

By using .Internal(inspect()) we can see that when an element of one of the vectors of the list is modified, the address of the list remains the same while that of the vector has changed. So R created a new vector but kept the orginal list.

```
library(pryr)
list <- list('1'= c(1:5), '2'=c('a','b','c'))
address(list)
address(list[[1]])
list[[1]][1] <- 2
address(list)
address(list[[1]])

# results in plain R

# > .Internal(inspect(list))
# @110b92248 19 VECSXP g0c2 [NAM(1),ATT] (len=2, tl=0)
#  @10d208d28 13 INTSXP g0c3 [] (len=5, tl=0) 1,2,3,4,5
#  @10d208c88 16 STRSXP g0c3 [] (len=3, tl=0)
#    @10401f120 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "a"
#    @1021d5ce0 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "b"
#    @103007cf8 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "c"
#  ...(omit attriutes)

# > .Internal(inspect(list))
# @110b92248 19 VECSXP g0c2 [NAM(1),ATT] (len=2, tl=0)
#  @1122fcb68 14 REALSXP g0c4 [] (len=5, tl=0) 2,2,3,4,5
#  @10d208c88 16 STRSXP g0c3 [] (len=3, tl=0)
#    @10401f120 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "a"
#    @1021d5ce0 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "b"
#    @103007cf8 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "c"
#   ...(omit attriutes)
```

(b)

From the results we can see that copies will be made for both the list and the vector in that case.

```
copy <- list
.Internal(inspect(list))
list[[1]][1] <- 1
```

```
.Internal(inspect(list))

# results in plain R

# .Internal(inspect(list))
# @110b92248 19 VECSXP g0c2 [MARK,NAM(3),ATT] (len=2, tl=0)
#   @1122fcb68 14 REALSXP g0c4 [MARK,NAM(3)] (len=5, tl=0) 2,2,3,4,5
#   @10d208c88 16 STRSXP g0c3 [MARK,NAM(3)] (len=3, tl=0)
#     @10401f120 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "a"
#     @1021d5ce0 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "b"
#     @103007cf8 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "c"
# ATTRIB: ...

# .Internal(inspect(list))
# @10f5162c8 19 VECSXP g0c2 [NAM(1),ATT] (len=2, tl=0)
#   @10f517ec8 14 REALSXP g0c4 [] (len=5, tl=0) 1,2,3,4,5
#   @10d208c88 16 STRSXP g0c3 [MARK,NAM(3)] (len=3, tl=0)
#     @10401f120 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "a"
#     @1021d5ce0 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "b"
#     @103007cf8 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "c"
# ATTRIB: ...
```

(c)

From the results we can see that the list(mylist) and the second list(list(1,2,3)) are copied. The first list and the first three elements in the second list are shared between the two lists.

```
mylist <- list(list("a","b","c"), list(1,2,3))
copy <- mylist
.Internal(inspect(copy))
mylist[[2]] <- c(mylist[[2]],4)
.Internal(inspect(mylist))

# results in plain R

# > .Internal(inspect(copy))
# @11134cac8 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
#   @1172c2538 19 VECSXP g0c3 [] (len=3, tl=0)
#     @110165970 16 STRSXP g0c1 [NAM(3)] (len=1, tl=0)
#       @10401f120 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "a"
#     @110165938 16 STRSXP g0c1 [NAM(3)] (len=1, tl=0)
#       @1021d5ce0 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "b"
#     @110165900 16 STRSXP g0c1 [NAM(3)] (len=1, tl=0)
#       @103007cf8 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "c"
#   @1172c24e8 19 VECSXP g0c3 [] (len=3, tl=0)
#     @1101658c8 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 1
#     @110165890 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 2
#     @110165858 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 3


# > .Internal(inspect(mylist))
# @11134cbc8 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
#   @1172c2538 19 VECSXP g0c3 [NAM(3)] (len=3, tl=0)
```

```
#     @110165970 16 STRSXP g0c1 [NAM(3)] (len=1, tl=0)
#       @10401f120 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "a"
#     @110165938 16 STRSXP g0c1 [NAM(3)] (len=1, tl=0)
#       @1021d5ce0 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "b"
#     @110165900 16 STRSXP g0c1 [NAM(3)] (len=1, tl=0)
#       @103007cf8 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "c"
#   @1172c23a8 19 VECSXP g0c3 [NAM(1)] (len=4, tl=0)
#     @1101658c8 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 1
#     @110165890 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 2
#     @110165858 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 3
#     @110165698 14 REALSXP g0c1 [] (len=1, tl=0) 4
```

(d)

From the results above we can see that in reality 97.5-21.2=76.3(~80) MB is used. That's because x, tmp[1] and tmp[2] has the same address, actually only one large number is stored and it takes about 80 MB.

```
## run the code
tmp <- list()
x <- rnorm(1e7)
tmp[[1]] <- x
tmp[[2]] <- x
.Internal(inspect(tmp))
object.size(tmp)

# result
# > .Internal(inspect(tmp))
# @117483888 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
#   @11cc4c000 14 REALSXP g0c7 [NAM(3)] (len=10000000, tl=0) 0.0372227,-0.0691961,-1.39104,-0.271885,0.
#   @11cc4c000 14 REALSXP g0c7 [NAM(3)] (len=10000000, tl=0) 0.0372227,-0.0691961,-1.39104,-0.271885,0.
# > object.size(tmp)
# 160000160 bytes

## find out real memory usage
gc()
tmp <- list()
x <- rnorm(1e7)
tmp[[1]] <- x
tmp[[2]] <- x
gc()

# result

# > gc()
#          used (Mb) gc trigger  (Mb) limit (Mb)
# Ncells 1305292 69.8    2371484 126.7         NA
# Vcells 2775649 21.2   15065094 115.0      16384
#       max used  (Mb)
# Ncells  2371484 126.7
# Vcells 22832488 174.2

# > gc()
#           used (Mb) gc trigger  (Mb) limit (Mb)
```

```
# Ncells  1305291 69.8    2371484 126.7         NA
# Vcells 12775632 97.5   18158112 138.6      16384
#        max used  (Mb)
# Ncells  2371484 126.7
# Vcells 22832488 174.2
```

5

This is because *rnorm* searches for random seed in global environment while *load* can only change current environment. And *set.seed* changes random seed in global environment wherever it's executed. We can see that as below.

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

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

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

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

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

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

```
## when we change random seed in global environment, tmp1 gives the same result.
## This shows rnorm looks for random seed in the global environment.
tmp1 <- function(){
  print(rnorm(1))
}
set.seed(1)
tmp1()
```

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

```
## We can see that after we execute tmp2 the random seed is set as 1.
## This shows that set.seed() changes the random seed in global environment.
tmp2 <- function(){
  set.seed(1)
}
tmp2()
rnorm(1)
```

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