PS4

Mengying Yang

October 10, 2017

Problem 1

a)

What is the maximum number of copies that exist of the vector 1:10 during the first execution of myFun()? Why?

```
library(pryr)
x <- 1:10
f <- function(input){
  print(.Internal(inspect(input)))
data <- input
  print(.Internal(inspect(data)))
g <- function(param) return(param * data)
return(g)
}
print(.Internal(inspect(x)))

## @0x0000000018ceeaa0 13 INTSXP g0c4 [NAM(2)] (len=10, tl=0) 1,2,3,4,5,...
##  [1]  1  2  3  4  5  6  7  8  9 10

myFun <- f(x)

## @0x0000000018ceeaa0 13 INTSXP g0c4 [NAM(2)] (len=10, tl=0) 1,2,3,4,5,...
##  [1]  1  2  3  4  5  6  7  8  9 10
## @0x0000000018ceeaa0 13 INTSXP g0c4 [NAM(2)] (len=10, tl=0) 1,2,3,4,5,...
##  [1]  1  2  3  4  5  6  7  8  9 10

data <- 100
myFun(3)

##  [1]  3  6  9 12 15 18 21 24 27 30
```

The maximum number of copies that exist of the vector 1:10 during the first excution of myFun is 1. I use .Internal(inspect())check the address of input and data, and their address is same with the address x. These means the 'input' and 'data' point to the address of x, and, because they were not modified, there isn't a copy being made in the function. The only copy of the 1:10 vector is x itself.

b)

```
x <- 1:1e4
f <- function(input){
data <- input
```

```
g <- function(param) return(param * data)
return(g)
}
myFun <- f(x)
object.size(serialize(myFun, NULL))

## 86888 bytes

object.size(serialize(x, NULL))

## 40064 bytes
```

The serialize() result double the size of x, this because it also count the result in addtion to x.Therefore, it double the result

c)

Explain what is happening and why this doesn't work to embed a constant data value into the function. Recall our discussion of when function arguments are evaluated

```
x <- 1:10
f <- function(data){
  g <- function(param) return(param * data)
return(g)
}

myFun <- f(x)
rm(x)
data <- 100
myFun(3)

## Error in myFun(3):  object 'x' not found
```

myFun is assigned as a copy of function g. When we call myfun, g was called. Since g needs parameter 'data' and the data was setted to 'x', g will try to find 'x'. Therefore, there will be error when we delete x.

d)

```
x <- 1:10
f <- function(data){
  force(data)
  g <- function(param) return(param * data)
return(g)
}

myFun <- f(x)
rm(x)
data <- 100
myFun(3)

##  [1]  3  6  9 12 15 18 21 24 27 30
```

We can use force function to force the evaluation of argument 'data'

Problem 2

a)

```
#create three vectors and each vector have 100 random number from uniform distribution(0,1)
vec1 <- runif(100)
vec2 <- runif(100)
vec3 <- runif(100)

# put this three vector in to list and check the address where the list and vector store
overall <- list(vec1, vec2, vec3)
.Internal(inspect(overall))

## @0x000000001a8a7a10 19 VECSXP g0c3 [NAM(2)] (len=3, tl=0)
##   @0x0000000018d36828 14 REALSXP g0c7 [NAM(2)] (len=100, tl=0) 0.53281,0.303211,0.133021,0.765532,0.
##   @0x000000001610b4f0 14 REALSXP g0c7 [NAM(2)] (len=100, tl=0) 0.606027,0.783488,0.790505,0.295021,0
##   @0x0000000015f685f8 14 REALSXP g0c7 [NAM(2)] (len=100, tl=0) 0.478779,0.626007,0.845151,0.467984,0

#modify an element of one of vectors
overall[[2]][1] <- 1
.Internal(inspect(overall))

## @0x000000001a958ed0 19 VECSXP g0c3 [NAM(1)] (len=3, tl=0)
##   @0x0000000018d36828 14 REALSXP g0c7 [NAM(2)] (len=100, tl=0) 0.53281,0.303211,0.133021,0.765532,0.
##   @0x00000000185c1938 14 REALSXP g0c7 [] (len=100, tl=0) 1,0.783488,0.790505,0.295021,0.771427,...
##   @0x0000000015f685f8 14 REALSXP g0c7 [NAM(2)] (len=100, tl=0) 0.478779,0.626007,0.845151,0.467984,0
```

From the result we can see, R didn't create a new list, but create a new vector which was changed

b)

```
overall2 <- overall
.Internal(inspect(overall2))

## @0x000000001a958ed0 19 VECSXP g0c3 [MARK,NAM(2)] (len=3, tl=0)
##   @0x0000000018d36828 14 REALSXP g0c7 [MARK,NAM(2)] (len=100, tl=0) 0.53281,0.303211,0.133021,0.7655
##   @0x00000000185c1938 14 REALSXP g0c7 [MARK] (len=100, tl=0) 1,0.783488,0.790505,0.295021,0.771427,.
##   @0x0000000015f685f8 14 REALSXP g0c7 [MARK,NAM(2)] (len=100, tl=0) 0.478779,0.626007,0.845151,0.467
```

As we can see, the address of the copy didn't change compare with 'overall', so no copy-on-change goes on.

```
overall2[[1]][1] <- 1
.Internal(inspect(overall2))

## @0x0000000017de32b8 19 VECSXP g0c3 [NAM(1)] (len=3, tl=0)
##   @0x00000000188afe48 14 REALSXP g0c7 [] (len=100, tl=0) 1,0.303211,0.133021,0.765532,0.161932,...
##   @0x00000000185c1938 14 REALSXP g0c7 [MARK,NAM(2)] (len=100, tl=0) 1,0.783488,0.790505,0.295021,0.7
##   @0x0000000015f685f8 14 REALSXP g0c7 [MARK,NAM(2)] (len=100, tl=0) 0.478779,0.626007,0.845151,0.467
```

The change on one of the vectors didn't make the copy change the entire list. The only change is the relevent vector(first vector at here)

c)

```
biglist <- list(overall, overall2)
.Internal(inspect(biglist))

## @0x0000000017700970 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##   @0x000000001a958ed0 19 VECSXP g0c3 [MARK,NAM(2)] (len=3, tl=0)
##     @0x0000000018d36828 14 REALSXP g0c7 [MARK,NAM(2)] (len=100, tl=0) 0.53281,0.303211,0.133021,0.765
##     @0x00000000185c1938 14 REALSXP g0c7 [MARK,NAM(2)] (len=100, tl=0) 1,0.783488,0.790505,0.295021,0
##     @0x0000000015f685f8 14 REALSXP g0c7 [MARK,NAM(2)] (len=100, tl=0) 0.478779,0.626007,0.845151,0.4
##   @0x0000000017de32b8 19 VECSXP g0c3 [NAM(2)] (len=3, tl=0)
##     @0x00000000188afe48 14 REALSXP g0c7 [] (len=100, tl=0) 1,0.303211,0.133021,0.765532,0.161932,...
##     @0x00000000185c1938 14 REALSXP g0c7 [MARK,NAM(2)] (len=100, tl=0) 1,0.783488,0.790505,0.295021,0
##     @0x0000000015f685f8 14 REALSXP g0c7 [MARK,NAM(2)] (len=100, tl=0) 0.478779,0.626007,0.845151,0.4

biglist2 <- biglist
.Internal(inspect(biglist2))

## @0x0000000017700970 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##   @0x000000001a958ed0 19 VECSXP g0c3 [MARK,NAM(2)] (len=3, tl=0)
##     @0x0000000018d36828 14 REALSXP g0c7 [MARK,NAM(2)] (len=100, tl=0) 0.53281,0.303211,0.133021,0.765
##     @0x00000000185c1938 14 REALSXP g0c7 [MARK,NAM(2)] (len=100, tl=0) 1,0.783488,0.790505,0.295021,0
##     @0x0000000015f685f8 14 REALSXP g0c7 [MARK,NAM(2)] (len=100, tl=0) 0.478779,0.626007,0.845151,0.4
##   @0x0000000017de32b8 19 VECSXP g0c3 [NAM(2)] (len=3, tl=0)
##     @0x00000000188afe48 14 REALSXP g0c7 [] (len=100, tl=0) 1,0.303211,0.133021,0.765532,0.161932,...
##     @0x00000000185c1938 14 REALSXP g0c7 [MARK,NAM(2)] (len=100, tl=0) 1,0.783488,0.790505,0.295021,0
##     @0x0000000015f685f8 14 REALSXP g0c7 [MARK,NAM(2)] (len=100, tl=0) 0.478779,0.626007,0.845151,0.4
```

When we create a list of lists and make a copy of it, they share the same address.

```
biglist2[[2]] <- append(biglist[[2]],runif(1))
.Internal(inspect(biglist2))

## @0x0000000018d27298 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
##   @0x000000001a958ed0 19 VECSXP g0c3 [MARK,NAM(2)] (len=3, tl=0)
##     @0x0000000018d36828 14 REALSXP g0c7 [MARK,NAM(2)] (len=100, tl=0) 0.53281,0.303211,0.133021,0.765
##     @0x00000000185c1938 14 REALSXP g0c7 [MARK,NAM(2)] (len=100, tl=0) 1,0.783488,0.790505,0.295021,0
##     @0x0000000015f685f8 14 REALSXP g0c7 [MARK,NAM(2)] (len=100, tl=0) 0.478779,0.626007,0.845151,0.4
##   @0x0000000017de2da8 19 VECSXP g0c3 [NAM(2)] (len=4, tl=0)
##     @0x00000000188afe48 14 REALSXP g0c7 [MARK,NAM(2)] (len=100, tl=0) 1,0.303211,0.133021,0.765532,0
##     @0x00000000185c1938 14 REALSXP g0c7 [MARK,NAM(2)] (len=100, tl=0) 1,0.783488,0.790505,0.295021,0
##     @0x0000000015f685f8 14 REALSXP g0c7 [MARK,NAM(2)] (len=100, tl=0) 0.478779,0.626007,0.845151,0.4
##     @0x0000000015f5e908 14 REALSXP g0c1 [] (len=1, tl=0) 0.384966
```

Through comparing with the address between biglist and new biglist2, we can find the addresses of all original list and original elements(those were not changed) inside the each list were not changed. However, the second list's address was pointed to a new address(although the original elements' address inside second list were not changed). The new element in the second list had a new address.

d)

```
gc()
```

```
##          used (Mb) gc trigger (Mb) max used (Mb)
## Ncells 360281 19.3     750400 40.1   592000 31.7
## Vcells 564555  4.4    1308461 10.0   786394  6.0

tmp <- list()
x <- rnorm(1e7)
gc()

##            used (Mb) gc trigger  (Mb) max used (Mb)
## Ncells   360317 19.3     750400  40.1   592000 31.7
## Vcells 10564605 80.7   15615500 119.2 10575140 80.7

tmp[[1]] <- x
tmp[[2]] <- x
.Internal(inspect(tmp))

## @0x00000000165d4da8 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
##   @0x00007ff5faf60010 14 REALSXP g1c7 [MARK,NAM(2)] (len=10000000, tl=0) -0.0948891,-0.365226,1.1561
##   @0x00007ff5faf60010 14 REALSXP g1c7 [MARK,NAM(2)] (len=10000000, tl=0) -0.0948891,-0.365226,1.1561

object.size(tmp)

## 160000136 bytes

gc()

##            used (Mb) gc trigger  (Mb) max used (Mb)
## Ncells   360366 19.3     750400  40.1   592000 31.7
## Vcells 10564716 80.7   15615500 119.2 10616267 81.0
```

The result of object.size is twice larger than the the result of the gc(). The object.size provides the estimate of the memory being used to store R object, according to the object.size(). Therefore, although the two element of tmp are store in the same address, object.size estimate there are two element in size x occupied the memory to store the tmp. However, the gc() shows the memory usage. Because all the element in tmp point to the address of X, the result of gc() is same with the .Interal(inspect()).

Problem 3

Original Code:

```
load("C:/Users/Renjun/Desktop/STAT243/ps4/ps4prob3.Rda") # should have A, n, K
ll <- function(Theta, A) {
  sum.ind <- which(A==1, arr.ind=T)
  logLik <- sum(log(Theta[sum.ind])) - sum(Theta)
  return(logLik)
  }
oneUpdate <- function(A, n, K, theta.old, thresh = 0.1) {
  theta.old1 <- theta.old
  Theta.old <- theta.old %*% t(theta.old)
  L.old <- ll(Theta.old, A)
  q <- array(0, dim = c(n, n, K))
  for (i in 1:n) {
    for (j in 1:n) {
      for (z in 1:K) {
        if (theta.old[i, z]*theta.old[j, z] == 0){
```

```
          q[i, j, z] <- 0
          } else {
            q[i, j, z] <- theta.old[i, z]*theta.old[j, z] / Theta.old[i, j]
            }
          }
        }
      }
  theta.new <- theta.old
  for (z in 1:K) {
    theta.new[,z] <- rowSums(A*q[,,z])/sqrt(sum(A*q[,,z]))
    }
  Theta.new <- theta.new %*% t(theta.new)
  L.new <- ll(Theta.new, A)
  converge.check <- abs(L.new - L.old) < thresh
  theta.new <- theta.new/rowSums(theta.new)
  return(list(theta = theta.new, loglik = L.new, converged = converge.check))
}
# initialize the parameters at random starting values
temp <- matrix(runif(n*K), n, K)
theta.init <- temp/rowSums(temp)
# do single update
ptm <- proc.time()
out <- oneUpdate(A, n, K, theta.init)
proc.time()-ptm

##    user  system elapsed
##  113.39    0.26  114.97
```

In the original code revise points: 1)the three nested for loop need to be revised by matrix muliplcation. 2)The if statement inside the for loop is unnecessary, because without the condition, the calculation result will be same. 3)The first line in the oneUpdate: the theta.old1 is a copy of theta.old, but it never be used in the function and following function, so redundant 4)The second for loop can combine together with the first for loop 5)A*q[,,z] be calculated twice and q never be used in the following, so we can let q equal to A*q[,,z] and save as q when calculate q.

Revise code:

```
load("C:/Users/Renjun/Desktop/STAT243/ps4/ps4prob3.Rda") # should have A, n, K
ll <- function(Theta, A) {
  sum.ind <- which(A==1, arr.ind=T)
  logLik <- sum(log(Theta[sum.ind])) - sum(Theta)
  return(logLik)
}
oneUpdatenew <- function(A, n, K, theta.old, thresh = 0.1) {
  #revise point 3) remove the first line
  Theta.old <- theta.old %*% t(theta.old)
  L.old <- ll(Theta.old, A)
  q <- array(0, dim = c(n, n, K))
  theta.new <- theta.old
  #revise point 1) 3) and 4) simplify the two for loop
  for (z in 1:K) {
          q[,, z] <- A*(theta.old[, z] %*% t(theta.old[, z]) / Theta.old) # revise point 5)
          theta.new[,z] <- rowSums(q[,,z])/sqrt(sum(q[,,z]))
          }
  Theta.new <- theta.new %*% t(theta.new)
```

```
  L.new <- ll(Theta.new, A)
  converge.check <- abs(L.new - L.old) < thresh
  theta.new <- theta.new/rowSums(theta.new)
  return(list(theta = theta.new, loglik = L.new, converged = converge.check))
}
# initialize the parameters at random starting values
temp <- matrix(runif(n*K), n, K)
theta.init <- temp/rowSums(temp)
# do single update
ptm <- proc.time()
out <- oneUpdatenew(A, n, K, theta.init)
proc.time()-ptm

##    user  system elapsed
##    1.03    0.28    1.38
```

The revised code achieved about more than 10-fold speed up.

Problem 4

a)

original code:

```
FYKD <- function(x, k) {
n <- length(x)
for(i in 1:n) {
j = sample(i:n, 1)
tmp <- x[i]
x[i] <- x[j]
x[j] <- tmp
}
return(x[1:k])
}
#
```

revised code: From the original code we can find the for loop generates 1:n permutation, which is unnecessary. We can only use first k permutation.

```
library(rbenchmark)
FYKDnew <- function(x, k) {
n <- length(x)
for(i in 1:k) { # Based on the reason above, I change the n to k
j = sample(i:n, 1)
tmp <- x[i]
x[i] <- x[j]
x[j] <- tmp
}
return(x[1:k])
}

x <- sample(1:1000000, 10000)
k <- 500
```

```
benchmark(FYKD(x,k),FYKDnew(x,k), replications=10, columns=c("test", "replications", "elapsed"))
```

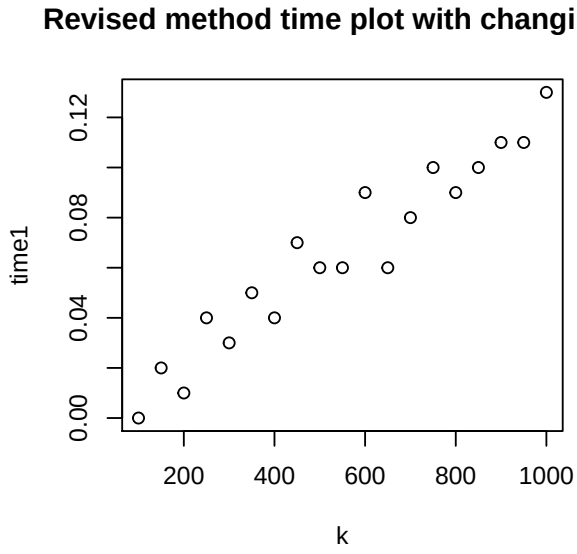
```
##            test replications elapsed
## 1    FYKD(x, k)           10    1.00
## 2 FYKDnew(x, k)           10    0.06
```

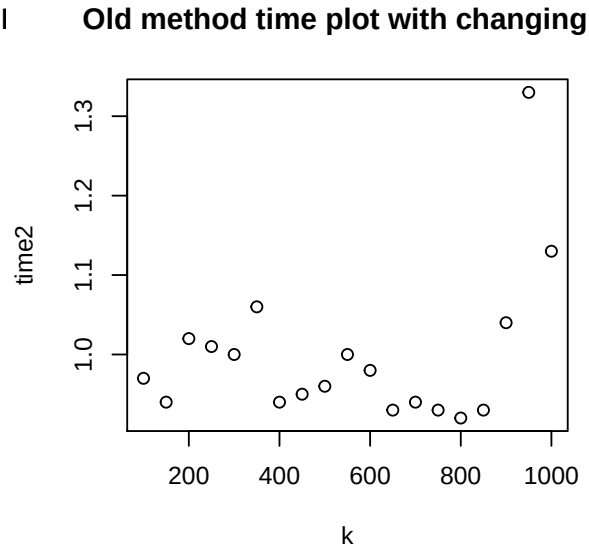
b)

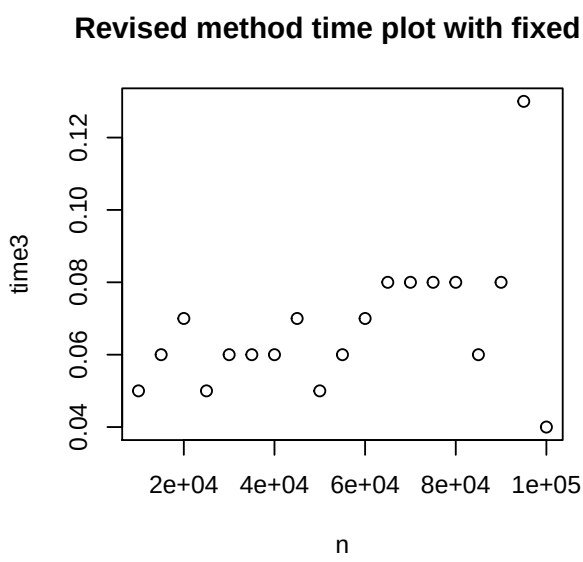
```
#plot
nfix <- sample(1:1e6, 1e4)
k <- seq(1e2, 1e3, 50)
n <- seq(1e4, 1e5, 5000)

time1 <- 1:length(k)
time2 <- 1:length(k)
time3 <- 1:length(k)
time4 <- 1:length(k)
x <- list()
for(i in 1:length(k)){
  #fix n and change k
  time1[i] <- benchmark(FYKDnew(nfix,k[i]),replications=10)$elapsed
  time2[i] <- benchmark(FYKD(nfix,k[i]),replications=10)$elapsed
  #fix k(pick k = 500) and change n
  x[[i]] <- sample(1:1e6, n[1])
  time3[i] <- benchmark(FYKDnew(x[[i]],500),replications=10)$elapsed
  time4[i] <- benchmark(FYKD(x[[i]],500),replications=10)$elapsed
}

par(mfrow=c(2,2))
plot(k, time1, main = "Revised method time plot with changing K")
plot(k, time2, main = "Old method time plot with changing k")
plot(n, time3, main = "Revised method time plot with fixed K")
plot(n, time4, main = "Old method time plot with fixed k")
```







bonus)

original code: if we want to speed up the function, we need to find a way to avoid use sort() function

```
PIKK <- function(x, k){
x[sort(runif(length(x)), index.return = TRUE)$ix[1:k]]
}
```

revised code:

```
PIKKnew <- function(x,k){
  x[unique(round(runif(1.5*k,0,length(x))))[1:k]]
}
```

```
x <- rnorm(1e4)
benchmark(PIKK(x,500),PIKKnew(x,500), replications=1000, columns=c("test", "replications", "elapsed"))

##              test replications elapsed
## 1    PIKK(x, 500)         1000    1.83
## 2 PIKKnew(x, 500)         1000    0.11
```

more than 15 times faster