STAT 243 PS 3

Junyuan Gao(SID:26484653)

October 11, 2017

1 Q1

1.1 a

```
library(pryr)

## Warning:  package 'pryr' was built under R version 3.4.2

x<- 1:10
f <- function(input){
  data <- input
  g <- function(param) return(param * data)
  return(g)
}
myFun <- f(x)
data <- 100
myFun(3)

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

x <- 100
myFun(3)

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

#maximum number of copies of the vector 1:10 is 1.
#Reason: At the first execution, when calling the function f(x),
# the program makes a variable "data", which is the copy of
# input vector x (1:10) and saved in some local address.
```

1.2 b

```r
library(pryr)
x <- 1:10000
f <- function(input){
  data <- input
  g <- function(param) return(param * data)
  return(g)
}
myFun <- f(x)
data <- 100
#myFun(3)
length(serialize(x, NULL)) #40022
```

```
## [1] 40022
```

```r
length(serialize(myFun, NULL)) #80778
```

```
## [1] 90261
```

```r
length(serialize(myFun(3), NULL)) #80022
```

```
## [1] 80022
```

```r
#Not match our expectation: expected that the size of seriazlized
#myFun() should be same as that of x, but indeed the size of
#serialized myFun() is approximately 2 times of the size of serialized x.

#Explanation: when calling f(), the program store "data" and "input" as local
#variables, each has the same length as x and same size as serialized x. Thus,
#the size of serialized myFun is 2 times of the size of serialized x.
```

1.3 c

```r
#(1)When executing myFun(), the program calls f(x), and then call x in global
#environment, however, since x is removed, R can't find where is x.

#(2)This doesnt work to embed a constant data value into the function since
#when running myFun(), the program first call the g() and evaluate 3 as param;
#then the program finds data in f(), and f calls data in the environment.
#However, x is removed, so the program can't find x, thus doesn't work.
```

1.4 d

The following modification make the code in part (c) work without explicitly creating a copy of the vector as in the original code.

```r
x <- 1:10
length(serialize(x, NULL))
```

```
## [1] 62
```

```r
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
```

```r
# size of the resulting serialized closure
length(serialize(myFun(3), NULL))
```

```
## [1] 102
```

2 Q2

For first 3 parts of this question, since there will be some bug kniting with Rstudio in printing the address, so I copy the result from R session as comment below the code.

2.1 a

```r
library(pryr) #execute together print the same output
y= list(c(1,2,3),c(4,5,6))
.Internal(inspect(y))
#@0x000000001f556710 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
  #@0x0000000022f4f798 14 REALSXP g0c3 [] (len=3, tl=0) 1,2,3
  #@0x0000000022f4f750 14 REALSXP g0c3 [] (len=3, tl=0) 4,5,6

y[[1]][2]=1
.Internal(inspect(y))
#@0x000000001f556710 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
  #@0x0000000022f4f798 14 REALSXP g0c3 [] (len=3, tl=0) 1,1,3
  #@0x0000000022f4f750 14 REALSXP g0c3 [] (len=3, tl=0) 4,5,6
```

```
#Conclusion:R can make the change in place, without creating
#a new list or a new vector
```

2.2 b

```
y= list(c(1,2,3),c(4,5,6))
yy =y
.Internal(inspect(y))
#@0x00000000204d0d08 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
  #@0x000000001dc562e0 14 REALSXP g0c3 [] (len=3, tl=0) 1,2,3
  #@0x000000001dc56400 14 REALSXP g0c3 [] (len=3, tl=0) 4,5,6
.Internal(inspect(yy))
#@0x00000000204d0d08 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
  #@0x000000001dc562e0 14 REALSXP g0c3 [] (len=3, tl=0) 1,2,3
  #@0x000000001dc56400 14 REALSXP g0c3 [] (len=3, tl=0) 4,5,6

## Conclusion: there is no copy-of-change going on

y[[1]][2]=1
.Internal(inspect(yy))
#@0x00000000204d0d08 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
  #@0x000000001dc562e0 14 REALSXP g0c3 [NAM(2)] (len=3, tl=0) 1,2,3
  #@0x000000001dc56400 14 REALSXP g0c3 [NAM(2)] (len=3, tl=0) 4,5,6
.Internal(inspect(y))
#@0x00000000204d0c98 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
  #@0x0000000005ef8d50 14 REALSXP g0c3 [] (len=3, tl=0) 1,1,3
  #@0x000000001dc56400 14 REALSXP g0c3 [NAM(2)] (len=3, tl=0) 4,5,6



# Conclusion:the copy of entire list is made since the entire
# address (first line of output) of y and yy are different.
# Thus, a copy is made.
```

2.3 c

```
y= list(list(1,2,3),list(4,5,6))
yyy= y
yyy[[3]]= list(7,8,9)
.Internal(inspect(y))
# @0x0000000022f2e690 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
```

```
#   @0x00000000233a6788 19 VECSXP g0c3 [NAM(2)] (len=3, tl=0)
#     @0x0000000020643b50 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 1
#     @0x0000000020643b80 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 2
#     @0x000000002062ab10 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 3
#   @0x00000000233a67d0 19 VECSXP g0c3 [NAM(2)] (len=3, tl=0)
#     @0x000000002062ab40 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 4
#     @0x000000002062ab70 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 5
#     @0x000000002062aba0 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 6
.Internal(inspect(yyy))
# @0x00000000231ebac0 19 VECSXP g0c3 [NAM(1)] (len=3, tl=0)
#   @0x00000000231edfd0 19 VECSXP g0c3 [NAM(2)] (len=3, tl=0)
#     @0x00000000216d7db0 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 1
#     @0x00000000216d7de0 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 2
#     @0x00000000216d7e10 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 3
#   @0x00000000231ee018 19 VECSXP g0c3 [NAM(2)] (len=3, tl=0)
#     @0x00000000216d7e40 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 4
#     @0x00000000216d7e70 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 5
#     @0x00000000216d7ea0 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 6
#   @0x00000000231eba78 19 VECSXP g0c3 [NAM(1)] (len=3, tl=0)
#     @0x00000000216d41d8 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 7
#     @0x00000000216d4208 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 8
#     @0x00000000216d4238 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 9



# Conclusion: linked list yyy(before adding the new element) is copied
# when adding the new element.
# the original vector y is not copied.
# the common elements in linked list yyy and linked list y are shared
```

2.4 d

```
gc()

##          used (Mb) gc trigger (Mb) max used (Mb)
## Ncells 483932 25.9     940480 50.3   750400 40.1
## Vcells 904127  6.9    1537628 11.8  1138576  8.7

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

```
## @0x000000001b5a8448 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
##   @0x00007ff5faca0010 14 REALSXP g0c7 [NAM(2)] (len=10000000, tl=0) 1.1465,0.942504,-0.88
##   @0x00007ff5faca0010 14 REALSXP g0c7 [NAM(2)] (len=10000000, tl=0) 1.1465,0.942504,-0.88

object.size(tmp)

## 160000136 bytes

gc()

##            used (Mb) gc trigger  (Mb) max used (Mb)
## Ncells   484679 25.9     940480  50.3   750400 40.1
## Vcells 10905990 83.3   16104525 122.9 10963024 83.7

# object.size() estunate the memory that being used to save tmp.
# Since tmp has 2 elements in list, so object.size() will go over
# both two elements and evaluate their usage of storage though
# they are both x and temporarily store at the same address.
# Thus, object,size() will count the use of storage twice, which
# produces a much larger storage than 80MB
```

3 Q3

Original code with run time at the end of the chunk.

```
load('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]
```

```r
        }
      }
    }
  }
  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
system.time(out <- oneUpdate(A, n, K, theta.init))

##    user  system elapsed
##   10.67    0.22   11.03

# in the real code, oneUpdate was called repeatedly in a while loop
# as part of an iterative optimization to find a maximum likelihood estimator
```

Modified code with run time at the end of the chunk.

```r
load('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))
## modification occurs at here. Use Vectorized expression
  #rather than multiple for-loops
  for (z in 1:K) {
    q[ , , z]= theta.old[ ,z] %*% t(theta.old[, z])/Theta.old
  }
  theta.new <- theta.old
```

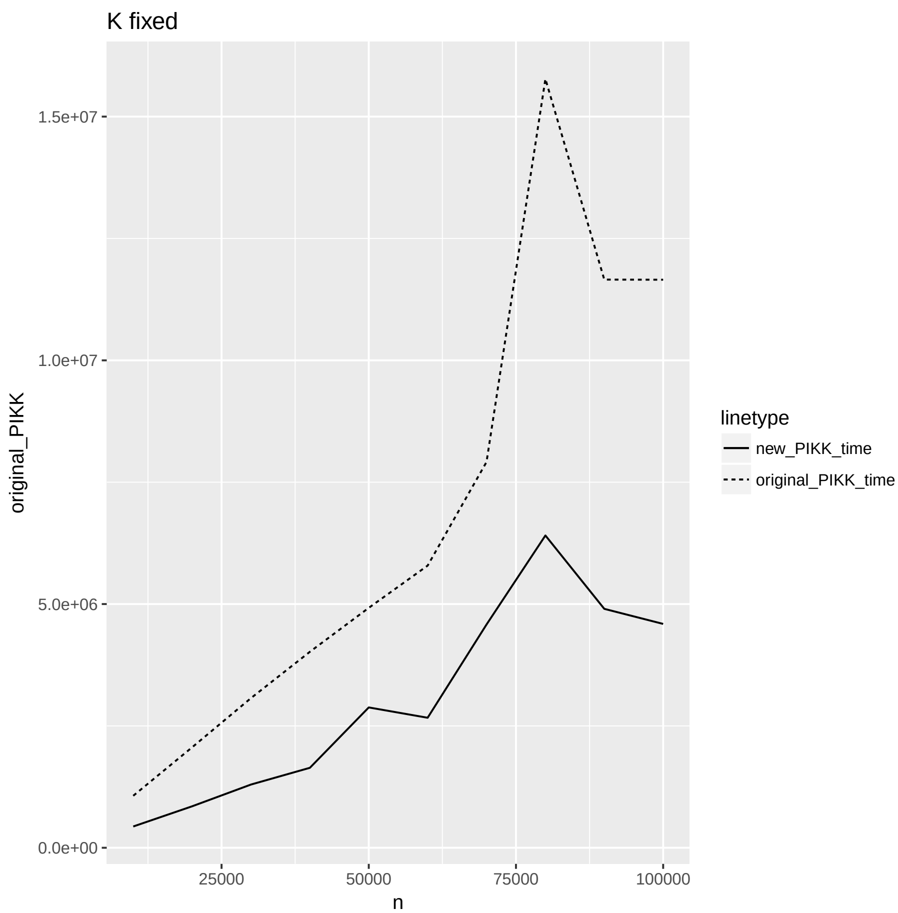
```r
  for (z in 1:K) {
    theta.new[,z] <- rowSums(A*q[,,z])/sqrt(sum(A*q[,,z]))
  }
  Theta.new <- theta.new %*% t(theta.new)
  L.new <- ll(Theta.new, A)
  converge.check <- abs(L.new - L.old) < thresh
  theta.new <- theta.new/rowSums(theta.new)
  return(list(theta = theta.new, loglik = L.new,
  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

##    user  system elapsed
##    1.03    0.27    1.30
```

4 Q4

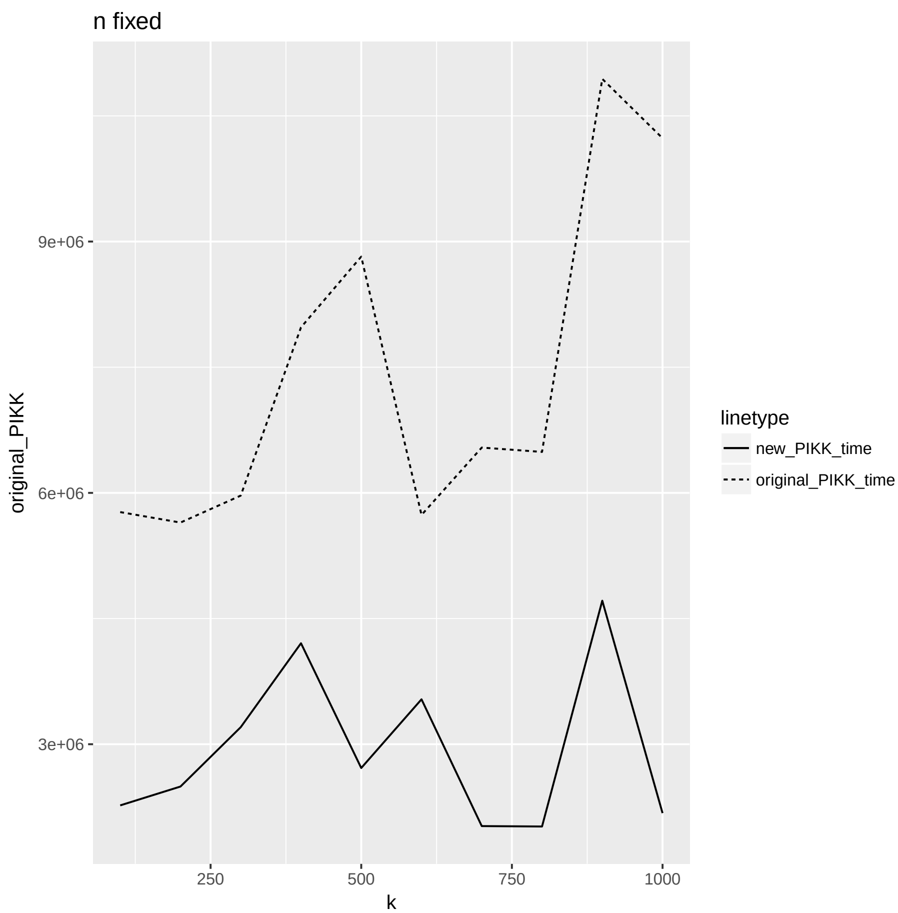
4.1 a

```r
library(microbenchmark)

## Warning:  package 'microbenchmark' was built under R version 3.4.2

x = runif(10000)
microbenchmark::microbenchmark(sample(x, size=500, replace=TRUE))

## Unit: microseconds
##                                   expr    min     lq   mean median     uq
##  sample(x, size = 500, replace = TRUE) 11.054 12.238 13.312 12.633 13.423
##     max neval
##  37.108   100

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

mean(microbenchmark::microbenchmark(PIKK(x, 500))$time)

## [1] 1120170

# the sort() is useless and have O(nlog(n)) complexity
# delete sort() and modify the algorithm
```

```
PIKK_new <- function(x, k) {
  x[floor(runif(length(x))*length(x))[1:k]]
}
mean(microbenchmark::microbenchmark(PIKK_new(x, 500))$time)

## [1] 445244.3

# the modified make 2+ times faster; but sample() make 79 times faster

## plot
# various n, fixed k=500
library(ggplot2)
n_vec <- seq(from=10000, to=100000, by=10000)
time_original1=c()
time_new1=c()

for (i in 1:10){
  samp= rnorm(n_vec[i])
  time_original1[i]=mean(microbenchmark::microbenchmark(PIKK(samp, 500))$time)
  time_new1[i] = mean(microbenchmark::microbenchmark(PIKK_new(samp, 500))$time)
}
df1 <- data.frame(n=n_vec, original_PIKK= time_original1, new_PIKK= time_new1)
k_fixed <- ggplot(df1, aes(x=n))+
          geom_line(aes(y=original_PIKK, lty="original_PIKK_time"))+
          geom_line(aes(y=new_PIKK, lty="new_PIKK_time"))
k_fixed + labs(title="K fixed")
```




```
#fixed n=50000, various k
k_vec <- seq(from=100, to=1000, by=100)
time_original2=c()
time_new2=c()

for (i in 1:10){
  samp= rnorm(50000)
  time_original2[i]=mean(microbenchmark::microbenchmark(PIKK(samp, k_vec[i]))$time)
  time_new2[i] = mean(microbenchmark::microbenchmark(PIKK_new(samp, k_vec[i]))$time)
}
df2 <- data.frame(k=k_vec, original_PIKK= time_original2, new_PIKK= time_new2)
n_fixed <- ggplot(df2, aes(x=k))+
  geom_line(aes(y=original_PIKK, lty="original_PIKK_time"))+
  geom_line(aes(y=new_PIKK, lty="new_PIKK_time"))
n_fixed + labs(title="n fixed")
```



4.2 b

Speed up the algorithm by only make k iterations rather than n iterations.

```
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])
}

FYKD_new <- function(x, k) {
```

```r
  n <- length(x)
  for(i in 1:k) {
    j = sample(i:n, 1)
    tmp <- x[i]
    x[i] <- x[j]
    x[j] <- tmp
  }
  return(x)
}
microbenchmark::microbenchmark(FYKD(x, 500))

## Unit: milliseconds
##           expr      min       lq     mean   median       uq      max neval
##   FYKD(x, 500) 67.69862 74.21568 85.99601 78.69855 90.31548 156.6869   100

microbenchmark::microbenchmark(FYKD_new(x, 500))

## Unit: milliseconds
##               expr      min       lq     mean   median       uq      max
##   FYKD_new(x, 500) 4.088113 4.819797 7.559852 6.190792 8.159253 62.30307
##   neval
##     100

microbenchmark::microbenchmark(sample(x, size=500, replace=TRUE))

## Unit: microseconds
##                                     expr    min     lq     mean median
##   sample(x, size = 500, replace = TRUE) 11.448 12.239 15.48316 12.633
##        uq     max neval
##   13.2255 105.006   100

# modified FYKD make 10 times faster than before, but still much slower
# than sample()
```