concordance=TRUE

Problem Set 4

Yue Hu

Oct 2017

1 problem 1

1.1 a)

Since there is no change on the vector 1:10, ther is only one copy. Both name x and data points to the same memory location.

1.2 b)

```
x <- 1:1e6
f <- function(input){
  data <- input
  g <- function(param) return(param * data)
  return(g)
}
myFun <- f(x)
data <- 100
Binary <- serialize(myFun, NULL)
object.size(x)

## 4000040 bytes

object.size(Binary)

## 8007144 bytes
```

We can see although input vector is 4M, the total size of the closure is about 8M. Seems there is two copies, because serialize doesn't know they are from the same memory location, and write them into binary as two.

1.3 c)

When x is passed to f(x), it is not evaluated until being called, which is called lazy-evaluation. So not until myFun is acutally callled by myFun(3) does the program begin to search for what x is.But at this time x is removed so r can't find it.

1.4 d)

We can assign values to data directly in the environment of myFun. The closure is 4M, which is the size of the data.

```
f <- function(data){
  g <- function(param) return(param * data)
  return(g)
}
myFun <- f(x)
environment(myFun)$data <- 1:1e6
Binary <- serialize(myFun, NULL)
data <- 100
object.size(Binary)

## 4006800 bytes
```

In another way,no copy of data is in the closure now and when it's called it looked for f(1:1e6) to find that input argument is 1e6. the resulting closure is 824 bytes.

```
f <- function(data){
  g <- function(param) return(param * data)
  return(g)
}
myFun <- f(1:1e6)
Binary <- serialize(myFun, NULL)
data <- 100
object.size(x)

## 4000040 bytes

object.size(Binary)

## 6464 bytes
```

2 problem 2

2.1 a)

```
library(pryr)

##
## Attaching package:  'pryr'
## The following object is masked _by_ '.GlobalEnv':
##
##    f

l <- list(rnorm(1:1e8), rnorm(1:1e8))
.Internal(inspect(l))

## @0x00000000131cb488 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##   @0x00007ff5b7ca0010 14 REALSXP g1c7 [MARK] (len=100000000, tl=0) 0.820658,0.640552,0.112937,0.5602
##   @0x00007ff5881a0010 14 REALSXP g0c7 [] (len=100000000, tl=0) 1.97362,1.44647,1.23207,1.42036,0.588

mem_change(l[[1]][1] <- 4)

## 15.6 kB

.Internal(inspect(l))
```

```
## @0x00000000131cb9c8 19 VECSXP g1c2 [MARK,NAM(1)] (len=2, tl=0)
##   @0x00007ff5586a0010 14 REALSXP g1c7 [MARK] (len=100000000, tl=0) 4,0.640552,0.112937,0.560276,-0.08
##   @0x00007ff5881a0010 14 REALSXP g1c7 [MARK,NAM(2)] (len=100000000, tl=0) 1.97362,1.44647,1.23207,1.4
```

When we modify an element of one of the vectors, the change was crated in place and no new list or vector is made. From the result, no vector changed place. Only several KB memory change takes place, which is far less than the length of the vector.

2.2 b)

```
mem_change(lCopy <- l)

## 504 B

.Internal(inspect(l))

## @0x00000000131cb9c8 19 VECSXP g1c2 [MARK,NAM(2)] (len=2, tl=0)
##   @0x00007ff5586a0010 14 REALSXP g1c7 [MARK] (len=100000000, tl=0) 4,0.640552,0.112937,0.560276,-0.08
##   @0x00007ff5881a0010 14 REALSXP g1c7 [MARK,NAM(2)] (len=100000000, tl=0) 1.97362,1.44647,1.23207,1.4

.Internal(inspect(lCopy))

## @0x00000000131cb9c8 19 VECSXP g1c2 [MARK,NAM(2)] (len=2, tl=0)
##   @0x00007ff5586a0010 14 REALSXP g1c7 [MARK] (len=100000000, tl=0) 4,0.640552,0.112937,0.560276,-0.08
##   @0x00007ff5881a0010 14 REALSXP g1c7 [MARK,NAM(2)] (len=100000000, tl=0) 1.97362,1.44647,1.23207,1.4

mem_change(lCopy[[1]][1] <- 8)

## 800 MB

.Internal(inspect(l))

## @0x00000000131cb9c8 19 VECSXP g1c2 [MARK,NAM(2)] (len=2, tl=0)
##   @0x00007ff5586a0010 14 REALSXP g1c7 [MARK,NAM(2)] (len=100000000, tl=0) 4,0.640552,0.112937,0.56027
##   @0x00007ff5881a0010 14 REALSXP g1c7 [MARK,NAM(2)] (len=100000000, tl=0) 1.97362,1.44647,1.23207,1.4

.Internal(inspect(lCopy))

## @0x00000000131cb990 19 VECSXP g1c2 [MARK,NAM(1)] (len=2, tl=0)
##   @0x00007ff5cfa20010 14 REALSXP g1c7 [MARK] (len=100000000, tl=0) 8,0.640552,0.112937,0.560276,-0.08
##   @0x00007ff5881a0010 14 REALSXP g1c7 [MARK,NAM(2)] (len=100000000, tl=0) 1.97362,1.44647,1.23207,1.4
```

When a copy of list is made, the rule of copy-on-change goes on and no new copys of the data is made. Memory does't change. When one of the vectors of one of the list is modified, only a copy of the relevant vector is made. From the result, only the place of modified vector changed. 800mb memory change takes place, which is the length of the vector.

2.3 c)

```
library(pryr)
l <- list( a= rnorm(1e6),
           b= list(b1 = rnorm(1e6),b2 = rnorm(1e6)),
           c= list(c1 = rnorm(1e6)))
lCopy <- l
```

```
.Internal(inspect(l))
```

```
## @0x000000001226fc80 19 VECSXP g0c3 [NAM(2),ATT] (len=3, tl=0)
##   @0x00007ff5cf270010 14 REALSXP g0c7 [] (len=1000000, tl=0) 1.15092,-1.61235,-1.42706,1.29121,-0.907
##   @0x000000001738c210 19 VECSXP g0c2 [ATT] (len=2, tl=0)
##     @0x00007ff5ceac0010 14 REALSXP g0c7 [] (len=1000000, tl=0) -0.323739,-1.3498,-0.424786,-1.28727,-
##     @0x00007ff5ce310010 14 REALSXP g0c7 [] (len=1000000, tl=0) -0.514394,1.02459,-0.441407,0.151066,0
##   ATTRIB:
##     @0x00000000125a9550 02 LISTSXP g0c0 []
##       TAG: @0x00000000105d2630 01 SYMSXP g1c0 [MARK,NAM(2),LCK,gp=0x4000] "names" (has value)
##       @0x000000001738c280 16 STRSXP g0c2 [] (len=2, tl=0)
##  @0x0000000017733128 09 CHARSXP g0c1 [gp=0x61] [ASCII] [cached] "b1"
##  @0x00000000174568c0 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "b2"
##   @0x0000000017358290 19 VECSXP g0c1 [ATT] (len=1, tl=0)
##     @0x00007ff5cdb60010 14 REALSXP g0c7 [] (len=1000000, tl=0) 1.14646,-0.431174,0.709113,-1.44903,-0
##   ATTRIB:
##     @0x00000000125a92b0 02 LISTSXP g0c0 []
##       TAG: @0x00000000105d2630 01 SYMSXP g1c0 [MARK,NAM(2),LCK,gp=0x4000] "names" (has value)
##       @0x00000000173582c0 16 STRSXP g0c1 [] (len=1, tl=0)
##  @0x00000000144b6358 09 CHARSXP g1c1 [MARK,gp=0x61,ATT] [ASCII] [cached] "c1"
## ATTRIB:
##   @0x00000000125a9240 02 LISTSXP g0c0 []
##     TAG: @0x00000000105d2630 01 SYMSXP g1c0 [MARK,NAM(2),LCK,gp=0x4000] "names" (has value)
##     @0x000000001226fc38 16 STRSXP g0c3 [] (len=3, tl=0)
##       @0x0000000013921bb8 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "a"
##       @0x0000000013cc2ac0 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "b"
##       @0x00000000105d9000 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "c"
```

```
.Internal(inspect(lCopy))
```

```
## @0x000000001226fc80 19 VECSXP g0c3 [NAM(2),ATT] (len=3, tl=0)
##   @0x00007ff5cf270010 14 REALSXP g0c7 [] (len=1000000, tl=0) 1.15092,-1.61235,-1.42706,1.29121,-0.907
##   @0x000000001738c210 19 VECSXP g0c2 [ATT] (len=2, tl=0)
##     @0x00007ff5ceac0010 14 REALSXP g0c7 [] (len=1000000, tl=0) -0.323739,-1.3498,-0.424786,-1.28727,-
##     @0x00007ff5ce310010 14 REALSXP g0c7 [] (len=1000000, tl=0) -0.514394,1.02459,-0.441407,0.151066,0
##   ATTRIB:
##     @0x00000000125a9550 02 LISTSXP g0c0 []
##       TAG: @0x00000000105d2630 01 SYMSXP g1c0 [MARK,NAM(2),LCK,gp=0x4000] "names" (has value)
##       @0x000000001738c280 16 STRSXP g0c2 [] (len=2, tl=0)
##  @0x0000000017733128 09 CHARSXP g0c1 [gp=0x61] [ASCII] [cached] "b1"
##  @0x00000000174568c0 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "b2"
##   @0x0000000017358290 19 VECSXP g0c1 [ATT] (len=1, tl=0)
##     @0x00007ff5cdb60010 14 REALSXP g0c7 [] (len=1000000, tl=0) 1.14646,-0.431174,0.709113,-1.44903,-0
##   ATTRIB:
##     @0x00000000125a92b0 02 LISTSXP g0c0 []
##       TAG: @0x00000000105d2630 01 SYMSXP g1c0 [MARK,NAM(2),LCK,gp=0x4000] "names" (has value)
##       @0x00000000173582c0 16 STRSXP g0c1 [] (len=1, tl=0)
##  @0x00000000144b6358 09 CHARSXP g1c1 [MARK,gp=0x61,ATT] [ASCII] [cached] "c1"
## ATTRIB:
##   @0x00000000125a9240 02 LISTSXP g0c0 []
##     TAG: @0x00000000105d2630 01 SYMSXP g1c0 [MARK,NAM(2),LCK,gp=0x4000] "names" (has value)
##     @0x000000001226fc38 16 STRSXP g0c3 [] (len=3, tl=0)
##       @0x0000000013921bb8 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "a"
##       @0x0000000013cc2ac0 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "b"
##       @0x00000000105d9000 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "c"
```

```
mem_change(lCopy$b <- c(l$b, b3 = list(rnorm(1e6))))

## 8 MB

.Internal(inspect(l))

## @0x000000001226fc80 19 VECSXP g1c3 [MARK,NAM(2),ATT] (len=3, tl=0)
##   @0x00007ff5cf270010 14 REALSXP g1c7 [MARK,NAM(2)] (len=1000000, tl=0) 1.15092,-1.61235,-1.42706,1.1
##   @0x0000000001738c210 19 VECSXP g1c2 [MARK,NAM(2),ATT] (len=2, tl=0)
##     @0x00007ff5ceac0010 14 REALSXP g1c7 [MARK,NAM(2)] (len=1000000, tl=0) -0.323739,-1.3498,-0.42478
##     @0x00007ff5ce310010 14 REALSXP g1c7 [MARK,NAM(2)] (len=1000000, tl=0) -0.514394,1.02459,-0.44140
##   ATTRIB:
##     @0x00000000125a9550 02 LISTSXP g1c0 [MARK]
##       TAG: @0x00000000105d2630 01 SYMSXP g1c0 [MARK,NAM(2),LCK,gp=0x4000] "names" (has value)
##       @0x0000000001738c280 16 STRSXP g1c2 [MARK,NAM(2)] (len=2, tl=0)
##  @0x0000000017733128 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "b1"
##  @0x00000000174568c0 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "b2"
##   @0x0000000017358290 19 VECSXP g1c1 [MARK,NAM(2),ATT] (len=1, tl=0)
##     @0x00007ff5cdb60010 14 REALSXP g1c7 [MARK] (len=1000000, tl=0) 1.14646,-0.431174,0.709113,-1.449
##   ATTRIB:
##     @0x00000000125a92b0 02 LISTSXP g1c0 [MARK]
##       TAG: @0x00000000105d2630 01 SYMSXP g1c0 [MARK,NAM(2),LCK,gp=0x4000] "names" (has value)
##       @0x00000000173582c0 16 STRSXP g1c1 [MARK] (len=1, tl=0)
##  @0x00000000144b6358 09 CHARSXP g1c1 [MARK,gp=0x61,ATT] [ASCII] [cached] "c1"
## ATTRIB:
##   @0x00000000125a9240 02 LISTSXP g1c0 [MARK]
##     TAG: @0x00000000105d2630 01 SYMSXP g1c0 [MARK,NAM(2),LCK,gp=0x4000] "names" (has value)
##     @0x000000001226fc38 16 STRSXP g1c3 [MARK,NAM(2)] (len=3, tl=0)
##       @0x0000000013921bb8 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "a"
##       @0x0000000013cc2ac0 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "b"
##       @0x00000000105d9000 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "c"

.Internal(inspect(lCopy))

## @0x00000000126528f8 19 VECSXP g1c3 [MARK,NAM(1),ATT] (len=3, tl=0)
##   @0x00007ff5cf270010 14 REALSXP g1c7 [MARK,NAM(2)] (len=1000000, tl=0) 1.15092,-1.61235,-1.42706,1.1
##   @0x0000000012652868 19 VECSXP g1c3 [MARK,NAM(1),ATT] (len=3, tl=0)
##     @0x00007ff5ceac0010 14 REALSXP g1c7 [MARK,NAM(2)] (len=1000000, tl=0) -0.323739,-1.3498,-0.42478
##     @0x00007ff5ce310010 14 REALSXP g1c7 [MARK,NAM(2)] (len=1000000, tl=0) -0.514394,1.02459,-0.44140
##     @0x00007ff5fed70010 14 REALSXP g1c7 [MARK,NAM(2)] (len=1000000, tl=0) -0.688094,0.119279,0.66408
##   ATTRIB:
##     @0x00000000125e20d8 02 LISTSXP g1c0 [MARK]
##       TAG: @0x00000000105d2630 01 SYMSXP g1c0 [MARK,NAM(2),LCK,gp=0x4000] "names" (has value)
##       @0x00000000126528b0 16 STRSXP g1c3 [MARK] (len=3, tl=0)
##  @0x0000000017733128 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "b1"
##  @0x00000000174568c0 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "b2"
##  @0x000000001772b560 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "b3"
##   @0x0000000017358290 19 VECSXP g1c1 [MARK,NAM(2),ATT] (len=1, tl=0)
##     @0x00007ff5cdb60010 14 REALSXP g1c7 [MARK] (len=1000000, tl=0) 1.14646,-0.431174,0.709113,-1.449
##   ATTRIB:
##     @0x00000000125a92b0 02 LISTSXP g1c0 [MARK]
##       TAG: @0x00000000105d2630 01 SYMSXP g1c0 [MARK,NAM(2),LCK,gp=0x4000] "names" (has value)
##       @0x00000000173582c0 16 STRSXP g1c1 [MARK] (len=1, tl=0)
##  @0x00000000144b6358 09 CHARSXP g1c1 [MARK,gp=0x61,ATT] [ASCII] [cached] "c1"
## ATTRIB:
```

```
##   @0x00000000125e2148 02 LISTSXP g1c0 [MARK]
##     TAG: @0x00000000105d2630 01 SYMSXP g1c0 [MARK,NAM(2),LCK,gp=0x4000] "names" (has value)
##     @0x000000001226fc38 16 STRSXP g1c3 [MARK,NAM(2)] (len=3, tl=0)
##       @0x0000000013921bb8 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "a"
##       @0x0000000013cc2ac0 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "b"
##       @0x00000000105d9000 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "c"
```

all original data is shared between two lists of lists, only one copy of new vector of the second list is made. The othter elements of the second list as well as other elements of the two lists is not changed or copied. From the result, only the place of appended vector changed. 8mb memory change takes place, which is the length of the vector.

2.4 d)

```
gc()

##           used (Mb) gc trigger   (Mb)  max used   (Mb)
## Ncells  479423 25.7     940480   50.3    797561   42.6
## Vcells 6392114 48.8  185546329 1415.7 305566051 2331.3

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

## @0x00000000131a7628 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
##   @0x00007ff5fa120010 14 REALSXP g0c7 [NAM(2)] (len=10000000, tl=0) -0.77895,-1.42905,-0.545173,0.18
##   @0x00007ff5fa120010 14 REALSXP g0c7 [NAM(2)] (len=10000000, tl=0) -0.77895,-1.42905,-0.545173,0.18

object.size(tmp)

## 160000136 bytes

gc()

##            used  (Mb) gc trigger   (Mb)  max used   (Mb)
## Ncells   479589  25.7     940480   50.3    797561   42.6
## Vcells 15892420 121.3  148437063 1132.5 305566051 2331.3
```

When we inspect we see that two vectors in the tmp list share the same location, so they take up 80MB in reality. But as documentation for object.size goes, this function merely provides a rough indication: it should be reasonably accurate for atomic vectors, but does not detect if elements of a list are shared, for example. So it only adds up the size of two vectors not knowing they point to the same loaction.

3 problem 3

first check the original time

```
library(rbenchmark)
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]
        }
      }
    }
  }
  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 and check the time
system.time(out <- oneUpdate(A, n, K, theta.init))

##    user  system elapsed
##    8.17    0.10    8.27
```

the rivised version uses matirx calculation to substitute 3 forloop with one forloop. Also, it reduced the repeatedly calculated A*q[,,z], also chaged some names to improve readability. This can improve by about 11 folds.

```
load('ps4prob3.Rda') # should have A, n, K
# Change the functiuon name to indicate its purpose
GetLoglik <- 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)
  # use Loglik instead of L to indecate its meaning
  Loglik.old <- GetLoglik(Theta.old, A)
  q <- array(0, dim = c(n, n, K))
  # use matirx calculation to substitute 3 forloop with one forloop
  for (z in 1:K){
    q[,,z] <- theta.old[,z]%*% t(theta.old[,z])/Theta.old
      }
  theta.new <- theta.old
  for (z in 1:K) {
    # A*q[,,z] is repeatedly calculated. calculate once and store in temp matrx
    temp <- A*q[,,z]
    theta.new[,z] <- rowSums(temp)/sqrt(sum(temp))
  }
  Theta.new <- theta.new %*% t(theta.new)
  Loglik.new <- GetLoglik(Theta.new, A)
  converge.check <- abs(Loglik.new - Loglik.old) < thresh
  theta.new <- theta.new/rowSums(theta.new)
  return(list(theta = theta.new, loglik = Loglik.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 and check the time
system.time(out <- oneUpdate(A, n, K, theta.init))

##    user  system elapsed
##     0.5     0.2     0.7
```

to further improve, two forloop can be merged together. Original q is a list of z (n*n)matrix,but we can just use one (n*n)matrix and overwright it z times to save space. This can improve it by 20 folds

```
load('ps4prob3.Rda') # should have A, n, K
# Change the functiuon name to indicate its purpose
GetLoglik <- function(Theta, A) {
  sum.ind <- which(A==1, arr.ind=T)
  logLik <- sum(log(Theta[sum.ind])) - sum(Theta)
  return(logLik)
}
oneUpdate1 <- function(A, n, K, theta.old, thresh = 0.1) {

  theta.old1 <- theta.old
  Theta.old <- theta.old %*% t(theta.old)
  # use Loglik instead of L to indecate its meaning
  Loglik.old <- GetLoglik(Theta.old, A)
  q <- matrix(0,n,n)
  theta.new <- theta.old
  # use matirx calculation to substitute 2 forloop
  # and two forloop can be merged together. original q is a list of z (n*n)matrix, but we can just use
  for (z in 1:K){
    q <- theta.old[,z]%*% t(theta.old[,z])/Theta.old
    temp <- A*q
    theta.new[,z] <- theta.new[,z] <- rowSums(temp)/sqrt(sum(temp))
```

```
  }

  Theta.new <- theta.new %*% t(theta.new)
  Loglik.new <- GetLoglik(Theta.new, A)
  converge.check <- abs(Loglik.new - Loglik.old) < thresh
  theta.new <- theta.new/rowSums(theta.new)
  return(list(theta = theta.new, loglik = Loglik.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(out1 <- oneUpdate1(A, n, K, theta.init))

##    user  system elapsed
##    0.22    0.16    0.38

benchmark(out <- oneUpdate1(A, n, K, theta.init), replications = 5)

##                                    test replications elapsed relative
## 1 out <- oneUpdate1(A, n, K, theta.init)            5    1.87        1
##   user.self sys.self user.child sys.child
## 1      1.01     0.84         NA        NA
```

4 problem 4

4.1 a)

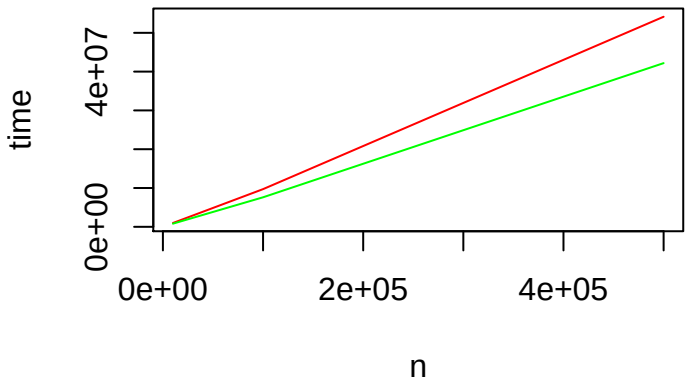
use order instead of sort, since it only returns the index, and sort returns both index and original number, it cam speed up 2 times.

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

# revised version
PIKK2 <- function(x, k) {
 x[order(runif(length(x)))[1:k]]
}

# test on x whith length 1e4 to 5e5 and plot the time of two functions
time <- list()
time2 <- list()
n <- c(10000, 100000, 500000)
for (i in n){
  x <- rnorm(i)
  k <- i/100
  benchm <- microbenchmark(PIKK(x,k), times = 100L)
  benchm2 <- microbenchmark(PIKK2(x,k), times = 100L)
  time <- c(time,mean(benchm$time))
  time2 <- c(time2, mean(benchm2$time))
```

```
}
plot(n,time, type = "l", col = "red")
lines(n, time2, col = "green")
```



another way for a is to only find the index of first k largest, insterad of sorting them all and exctract the first k.

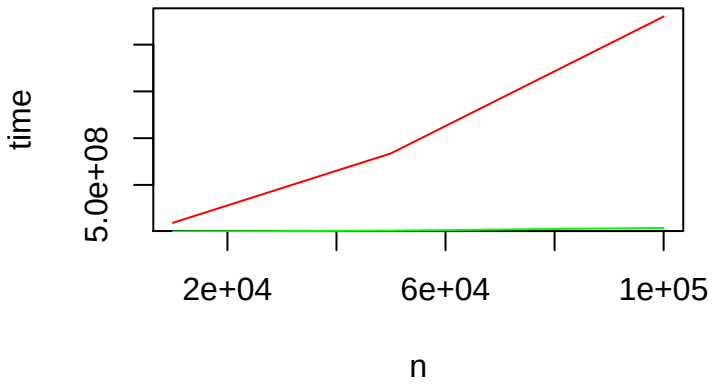
for b, only need to shuffle the first k elements and extract them. This can speed up 10 times.

```
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])
}

FYKD2 <- function(x, k) {
  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[1:k])
}

# test on x whith length 1e4 to 5e5 and plot the time of two functions
time <- list()
time2 <- list()
n <- c(10000, 50000, 100000)
for (i in n){
```

```
  x <- rnorm(i)
  k <- i/100
  benchm <- microbenchmark(FYKD(x,k), times = 10L)
  benchm2 <- microbenchmark(FYKD2(x,k), times = 10L)
  time <- c(time,mean(benchm$time))
  time2 <- c(time2, mean(benchm2$time))
}
plot(n,time, type = "l", col = "red")
lines(n, time2, col = "green")
```



4.2 b)

Another way to think about, which generate the random numbers in a vectorized fashion. this can spped up to 10 times of the original sample().

```
FYKD3 <- function(x, k) {
  sample <- array(0,dim = k)
  n <- length(x)
  # generate a ramdom int array (index) with a max of n, n-1,ect.
  scale <- c(n:(n-k+1))
  index2 <- floor(runif(k)*scale)+1
  # for loop i, extract element according to index, (the pisition shold be between 1 and n-i+1) and rep
  for(i in 1:k) {
    sample[i] <- x[index2[i]]
    x[index2[i]] <- x[n-i+1]
  }
  return(sample)
}

microbenchmark(FYKD(x,k), times =50L)

## Unit: seconds
##        expr      min       lq     mean   median       uq      max neval
##  FYKD(x, k) 2.205703 2.258611 2.486203 2.373855 2.508721 3.576228    50
```

```r
microbenchmark(FYKD3(x,k), times = 50L)
```

```
## Unit: microseconds
##         expr     min      lq     mean   median      uq      max neval
##  FYKD3(x, k) 339.016 370.088 668.8206 512.4655 551.885 9971.497    50
```