Problem Set 4

Dongyu Lang
SID: 24174288

October, 10, 2017

1 Question 1 (a)

```
gc()
```

```
##          used (Mb) gc trigger (Mb) max used (Mb)
## Ncells 440249 23.6     750400 40.1   592000 31.7
## Vcells 632191  4.9    1308461 10.0   845604  6.5
```

```
x <- 1:1e7
f <- function(input){
  data <- input
  g <- function(param) return(param * data)
  return(g)
}
myFun <- f(x)
data <- 100
result <- myFun(3)
gc()
```

```
##            used  (Mb) gc trigger  (Mb) max used  (Mb)
## Ncells   440660  23.6     750400  40.1   592000  31.7
## Vcells 15632405 119.3   22913146 174.9 15642647 119.4
```

From the result we can notice that myFun(3) actually is twice larger in memory use than that of x. The reason is that x, input and data all have the same address in function f, and for param * data, there is actually a copy of data, and counting the result itself makes myFun(3) twice the size; thus, the maximum number of copy is 1.

2 Question 1 (b)

```r
x <- 1:1e7
f <- function(input){
  data <- input
  g <- function(param) return(param * data)
  return(g)
}
myFun <- f(x)
data <- 100
result = myFun(3)
length(serialize(x, NULL))

## [1] 40000022

length(serialize(myFun(3), NULL))

## [1] 80000022

length(serialize(myFun, NULL))

## [1] 80008928
```

The results show that the size of serialized myFun(3) is twice the size of serialized x; thus, it supports the result we provided in part (a).

3 Question 1 (c)

```r
x <- 1:10
f <- function(data){
  g <- function(param) return(param * data)
  return(g)
}
myFun <- f(x)
rm(x)
data <- 100
environment(myFun)$data

## Error in eval(expr, envir, enclos):  object 'x' not found
```

The reason that the code fails is that when we remove x, we also remove data in the environment of myFun. And myFun(3) could not work, because it tried to find the x, not data. Since we removed x, the error occurred.

4 Question 1(d)

```r
x <- 1:10
f <- function(data){
  g <- function(param) return(param * data)
  return(g)
}
myFun <- f(x)
rm(x)
## Change "data <- 100" to "x <- 100".
x <- 100
myFun(3)

## [1] 300

length(serialize(x,NULL))

## [1] 30

length(serialize(myFun(3),NULL))

## [1] 30
```

I changed data = 100 to x = 100, and it worked. And the size of the serialized myFun(3) has the same size as the serialized x. Thus, I did not create a copy.

5 Question 2(a)

```r
testlist = list(a = rnorm(5), b = 1:10)
.Internal(inspect(testlist))

## @7fde1b9f1870 19 VECSXP g0c2 [NAM(2),ATT] (len=2, tl=0)
##   @7fde1b976fb0 14 REALSXP g0c4 [] (len=5, tl=0) 1.25439,-0.376587,-0.645499,0.511551,1.2
##   @7fde1b9771b8 13 INTSXP g0c4 [] (len=10, tl=0) 1,2,3,4,5,...
## ATTRIB:
##   @7fde1c4f22a8 02 LISTSXP g0c0 []
##     TAG: @7fde19802678 01 SYMSXP g1c0 [MARK,LCK,gp=0x4000] "names" (has value)
##     @7fde1b9f18a8 16 STRSXP g0c2 [] (len=2, tl=0)
##       @7fde1a12e4c8 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "a"
##       @7fde19a37208 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "b"

.Internal(inspect(testlist$a))

## @7fde1b976fb0 14 REALSXP g0c4 [NAM(2)] (len=5, tl=0) 1.25439,-0.376587,-0.645499,0.51155

.Internal(inspect(testlist$b))

## @7fde1b9771b8 13 INTSXP g0c4 [NAM(2)] (len=10, tl=0) 1,2,3,4,5,...
```

```
## Now change an element in a
testlist$a[2] = 0.123
## Check the address of the changed list
.Internal(inspect(testlist))

## @7fde1aa34e30 19 VECSXP g0c2 [NAM(1),ATT] (len=2, tl=0)
##   @7fde1a977020 14 REALSXP g0c4 [] (len=5, tl=0) 1.25439,0.123,-0.645499,0.511551,1.27163
##   @7fde1b9771b8 13 INTSXP g0c4 [NAM(2)] (len=10, tl=0) 1,2,3,4,5,...
## ATTRIB:
##   @7fde1a735a28 02 LISTSXP g0c0 []
##     TAG: @7fde19802678 01 SYMSXP g1c0 [MARK,LCK,gp=0x4000] "names" (has value)
##     @7fde1b9f18a8 16 STRSXP g0c2 [NAM(2)] (len=2, tl=0)
##       @7fde1a12e4c8 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "a"
##       @7fde19a37208 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "b"

.Internal(inspect(testlist$a))

## @7fde1a977020 14 REALSXP g0c4 [NAM(2)] (len=5, tl=0) 1.25439,0.123,-0.645499,0.511551,1.2

.Internal(inspect(testlist$b))

## @7fde1b9771b8 13 INTSXP g0c4 [NAM(2)] (len=10, tl=0) 1,2,3,4,5,...
```

From the result, we can notice that R can make change in place without creating a new list or vector. (It should be the same addresses in plain R session, but it changed in Rstudio)

6 Question 2(b)

```
## create the list
testlist = list(a = rnorm(5), b = 1:10)
.Internal(inspect(testlist$a))

## @7fde1b4ba208 14 REALSXP g0c4 [NAM(2)] (len=5, tl=0) -0.153593,2.00993,1.05525,-0.203566,

.Internal(inspect(testlist$b))

## @7fde1b4fb088 13 INTSXP g0c4 [NAM(2)] (len=10, tl=0) 1,2,3,4,5,...

## Make a copy of the list
copylist = testlist
.Internal(inspect(copylist$a))

## @7fde1b4ba208 14 REALSXP g0c4 [NAM(2)] (len=5, tl=0) -0.153593,2.00993,1.05525,-0.203566,

.Internal(inspect(copylist$b))
```

```
## @7fde1b4fb088 13 INTSXP g0c4 [NAM(2)] (len=10, tl=0) 1,2,3,4,5,...

## Make a change to the copylist
copylist$a[3] = 0.1111

## Check the addresses for testlist and copylist
.Internal(inspect(testlist))

## @7fde19932ed8 19 VECSXP g0c2 [NAM(2),ATT] (len=2, tl=0)
##   @7fde1b4ba208 14 REALSXP g0c4 [NAM(2)] (len=5, tl=0) -0.153593,2.00993,1.05525,-0.20356
##   @7fde1b4fb088 13 INTSXP g0c4 [NAM(2)] (len=10, tl=0) 1,2,3,4,5,...
## ATTRIB:
##   @7fde1ad23f08 02 LISTSXP g0c0 []
##     TAG: @7fde19802678 01 SYMSXP g1c0 [MARK,LCK,gp=0x4000] "names" (has value)
##     @7fde19932f10 16 STRSXP g0c2 [NAM(2)] (len=2, tl=0)
##       @7fde1a12e4c8 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "a"
##       @7fde19a37208 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "b"

.Internal(inspect(testlist$a))

## @7fde1b4ba208 14 REALSXP g0c4 [NAM(2)] (len=5, tl=0) -0.153593,2.00993,1.05525,-0.203566,

.Internal(inspect(testlist$b))

## @7fde1b4fb088 13 INTSXP g0c4 [NAM(2)] (len=10, tl=0) 1,2,3,4,5,...

.Internal(inspect(copylist))

## @7fde1d03d590 19 VECSXP g0c2 [NAM(1),ATT] (len=2, tl=0)
##   @7fde1b5b1e80 14 REALSXP g0c4 [] (len=5, tl=0) -0.153593,2.00993,0.1111,-0.203566,0.374
##   @7fde1b4fb088 13 INTSXP g0c4 [NAM(2)] (len=10, tl=0) 1,2,3,4,5,...
## ATTRIB:
##   @7fde1b5dc978 02 LISTSXP g0c0 []
##     TAG: @7fde19802678 01 SYMSXP g1c0 [MARK,LCK,gp=0x4000] "names" (has value)
##     @7fde19932f10 16 STRSXP g0c2 [NAM(2)] (len=2, tl=0)
##       @7fde1a12e4c8 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "a"
##       @7fde19a37208 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "b"

.Internal(inspect(copylist$a))

## @7fde1b5b1e80 14 REALSXP g0c4 [NAM(2)] (len=5, tl=0) -0.153593,2.00993,0.1111,-0.203566,0

.Internal(inspect(copylist$b))

## @7fde1b4fb088 13 INTSXP g0c4 [NAM(2)] (len=10, tl=0) 1,2,3,4,5,...
```

When I created a copy of the testlist, there is no copy-on-change going

on, and the addresses of vectors in testlist were the same as those in copylist. However, when I changed an element in vector a in copylist. There was a copy going on for vector a in copylist, but no change happened for vector b in copylist, since the address of vector a changed, but that of vector b did not change in copylist. In addition, the there is a copy of the two lists. (The result is a bit different in plain R session)

7 Question 2(c)

```
## Create a list of two lists, and each list contains two vectors
twolists = list(list1 = list(a = rnorm(5), b = 1:10),
            list2 = list(c = runif(4), d = c(2,5,8,1,4,9)))
## Copy the list
copytwolists = twolists

## Add an element to the second list in copytwolists
copytwolists$list2$e = 1:20

## Take a look at the address of each vector
.Internal(inspect(twolists$list1$a))

## @7fde1b8dbb40 14 REALSXP g0c4 [NAM(2)] (len=5, tl=0) 0.459816,0.260441,0.505083,0.877517,

.Internal(inspect(copytwolists$list1$a))

## @7fde1b8dbb40 14 REALSXP g0c4 [NAM(2)] (len=5, tl=0) 0.459816,0.260441,0.505083,0.877517,

.Internal(inspect(twolists$list1$b))

## @7fde1b8dbe18 13 INTSXP g0c4 [NAM(2)] (len=10, tl=0) 1,2,3,4,5,...

.Internal(inspect(copytwolists$list1$b))

## @7fde1b8dbe18 13 INTSXP g0c4 [NAM(2)] (len=10, tl=0) 1,2,3,4,5,...

.Internal(inspect(twolists$list2$c))

## @7fde1bf4a0d8 14 REALSXP g0c3 [NAM(2)] (len=4, tl=0) 0.237311,0.050223,0.399693,0.23273

.Internal(inspect(copytwolists$list2$c))

## @7fde1bf4a0d8 14 REALSXP g0c3 [NAM(2)] (len=4, tl=0) 0.237311,0.050223,0.399693,0.23273

.Internal(inspect(twolists$list2$d))

## @7fde1b8dbfb8 14 REALSXP g0c4 [NAM(2)] (len=6, tl=0) 2,5,8,1,4,...
```

```
.Internal(inspect(copytwolists$list2$d))

## @7fde1b8dbfb8 14 REALSXP g0c4 [NAM(2)] (len=6, tl=0) 2,5,8,1,4,...

.Internal(inspect(copytwolists$list2$e))

## @7fde1adf58a8 13 INTSXP g0c5 [NAM(2)] (len=20, tl=0) 1,2,3,4,5,...

.Internal(inspect(twolists$list2))

## @7fde1aa64788 19 VECSXP g0c2 [NAM(2),ATT] (len=2, tl=0)
##   @7fde1bf4a0d8 14 REALSXP g0c3 [NAM(2)] (len=4, tl=0) 0.237311,0.050223,0.399693,0.23273
##   @7fde1b8dbfb8 14 REALSXP g0c4 [NAM(2)] (len=6, tl=0) 2,5,8,1,4,...
## ATTRIB:
##   @7fde1c2f1dc8 02 LISTSXP g0c0 []
##     TAG: @7fde19802678 01 SYMSXP g1c0 [MARK,LCK,gp=0x4000] "names" (has value)
##     @7fde1aa58030 16 STRSXP g0c2 [NAM(2)] (len=2, tl=0)
##       @7fde19813d78 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "c"
##       @7fde1a37de68 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "d"

.Internal(inspect(copytwolists$list2))

## @7fde1bc197b8 19 VECSXP g0c3 [NAM(1),ATT] (len=3, tl=0)
##   @7fde1bf4a0d8 14 REALSXP g0c3 [NAM(2)] (len=4, tl=0) 0.237311,0.050223,0.399693,0.23273
##   @7fde1b8dbfb8 14 REALSXP g0c4 [NAM(2)] (len=6, tl=0) 2,5,8,1,4,...
##   @7fde1adf58a8 13 INTSXP g0c5 [NAM(2)] (len=20, tl=0) 1,2,3,4,5,...
## ATTRIB:
##   @7fde1a22df50 02 LISTSXP g0c0 []
##     TAG: @7fde19802678 01 SYMSXP g1c0 [MARK,LCK,gp=0x4000] "names" (has value)
##     @7fde1bc19800 16 STRSXP g0c3 [NAM(2)] (len=3, tl=0)
##       @7fde19813d78 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "c"
##       @7fde1a37de68 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "d"
##       @7fde1a02c8a8 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "e"
```

From the results, we can notice that the unchanged vectors are shared between the two lists. There is no copy for the vectors that are unchanged. There is a new copy for the newly added vector. In addition, there is a copy for these two lists.

8 Question 2(d)

```
gc()

##            used (Mb) gc trigger  (Mb) max used  (Mb)
## Ncells   441851 23.6     750400  40.1   750400  40.1
```

```
## Vcells 10636559 81.2   29850996 227.8 35821168 273.3

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

## @7fde1c8ccfc0 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
##   @1140d6000 14 REALSXP g0c7 [NAM(2)] (len=10000000, tl=0) -0.225882,-2.58529,0.357297,0.
##   @1140d6000 14 REALSXP g0c7 [NAM(2)] (len=10000000, tl=0) -0.225882,-2.58529,0.357297,0.

object.size(tmp)

## 160000136 bytes

gc()

##            used  (Mb) gc trigger  (Mb) max used  (Mb)
## Ncells   442481  23.7     750400  40.1   750400  40.1
## Vcells 20637478 157.5   29850996 227.8 35821168 273.3
```

The reason that object.size() generates a different result is that this fucntion provides estimates of the memory use but does not detect if the list is shared or not. Since there is no memory copy made we just copy x to tmp1 and tmp2; thus, object.size() overestimates the memory use.

9 Question 3

```
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)

# This is the updated function for oneUpdate
oneUpdate2 <- function(A, n, K, theta.old, thresh = 0.1) {
  Theta.square <- theta.old %*% t(theta.old)
  L.old <- ll(Theta.square, A)
  ## I deleted all the three nested for-loop
  ## And I add a new matrix qz in repalce for the old 3 dimensional q
  ## it turns out my qz is the same as q[,,z] for each z in 1:k
  ## Let's suppose z = 1, a = theta.old and A = Theta.square
  ## Thus, q[i,j,1]=a[i,1]*a[j,1]/A[i,j].
  ## So we can actually write the q matrix into some matrix multiplication
  ## q[,,z] is just theta.old[,z] %*% t(theta.old[,z]) / Theta.square
  for (z in 1:K) {
    qz = theta.old[,z] %*% t(theta.old[,z]) / Theta.square
    theta.old[,z] <- rowSums(A*qz)/sqrt(sum(A*qz))
  }
  Theta.square.new <- theta.old %*% t(theta.old)
  L.new <- ll(Theta.square.new, A)
  converge.check <- abs(L.new - L.old) < thresh
  theta.old <- theta.old/rowSums(theta.old)
  return(list(theta = theta.old, loglik = L.new,
              converged = converge.check))
}
```

```
# do single update
originaltime = system.time(out <- oneUpdate(A, n, K, theta.init))
newtime = system.time(out2 <- oneUpdate2(A, n, K, theta.init))
## Are the results for old and new method identical?
identical(out,out2)

## [1] TRUE

originaltime

##    user  system elapsed
##  80.397   0.559  81.252

newtime

##    user  system elapsed
##   0.448   0.377   0.827

## Speedup
speedup = as.numeric(originaltime[3]/newtime[3])
speedup

## [1] 98.24909
```

I explained my code and changes inside the code chunk. Beside these changes, I also changed some style of the code to make it clearer, and deleted some memory copy. The result shows that my new method achieve a 98.2490931 folds speedup

10 Question 4 (a,b)

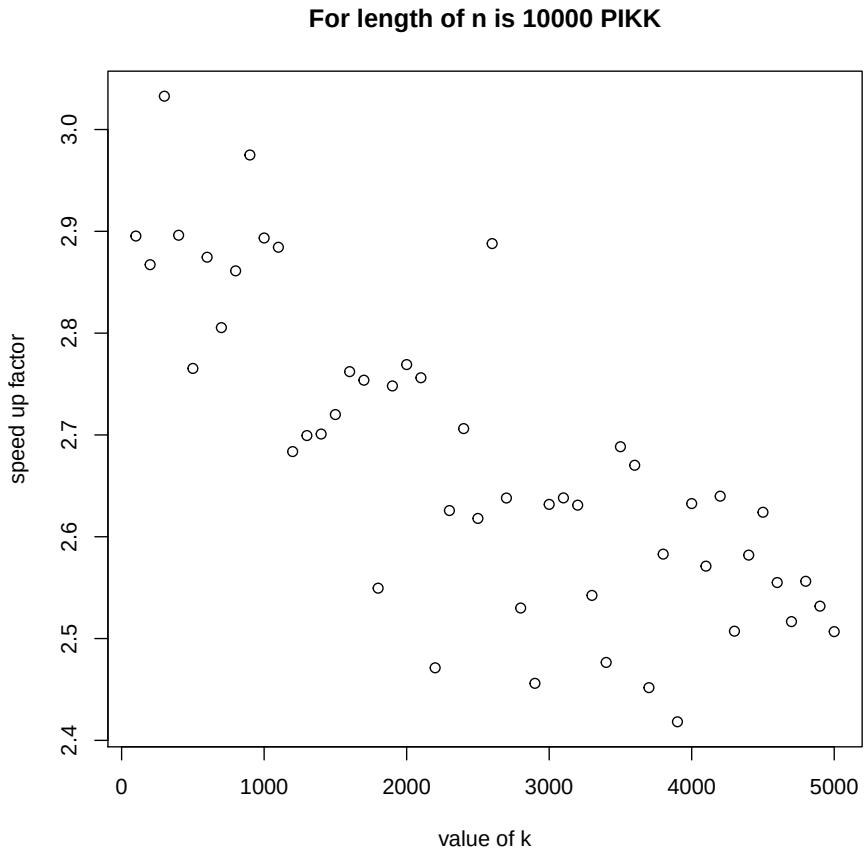
For PIKK

```
## The original one is slow because that first it stores not only index,
## but also the original numbers. Secondly, it sorts all the n runif;
## however, we only need the first k
PIKK <- function(x, k) {
  x[sort(runif(length(x)), index.return = TRUE)$ix[1:k]]
}

## The new method only compares the smallest k numbers, and it saves time
## by not comparing all the n numbers.
PIKKnew <- function(x,k) {
  randomx = runif(length(x))
  x[randomx <= sort(randomx, partial=k)[k]]
}
```

```
mbmPIKK = microbenchmark("original" = {
  a <- PIKK(1:10000,500)
}, "new" = {
  b <- PIKKnew(1:10000,500)
})

mbmPIKK

## Unit: microseconds
##      expr      min        lq     mean    median        uq      max neval
##  original 1595.721 1737.0440 1850.834 1767.7445 1815.9555 3593.024   100
##       new  503.514  640.3395  722.670  682.3215  711.5665 1545.663   100

## Speed up factor
speedupfactor = summary(mbmPIKK)$mean[1]/summary(mbmPIKK)$mean[2]
speedupfactor

## [1] 2.561105
```

We can notice that the time was saved by a factor of 2.5611049.
Plot for PIKK

```
## For length of n is 10000
set.seed(100)
speedup10000 = NULL
for (i in seq(100,5000,100)) {
  mbmPIKK10000 = microbenchmark("original" = {
    a <- PIKK(1:10000,i)
    }, new = {
    b <- PIKKnew(1:10000,i)
  })
  index = which(i == seq(100,5000,100))
  speedup10000[index] = summary(mbmPIKK10000)$mean[1]/summary(mbmPIKK10000)$mean[2]
}
plot(x = seq(100,5000,100), y = speedup10000, xlab="value of k",
     ylab="speed up factor", main = "For length of n is 10000 PIKK")
```



When I fixed the value of n, and took different values of k, the speed up factors were consistantly over 2. And as k increased, there was a slightly decreasing trend.

For FYKD

```
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])
}

## The new function tries to get the index for the minimum value
```

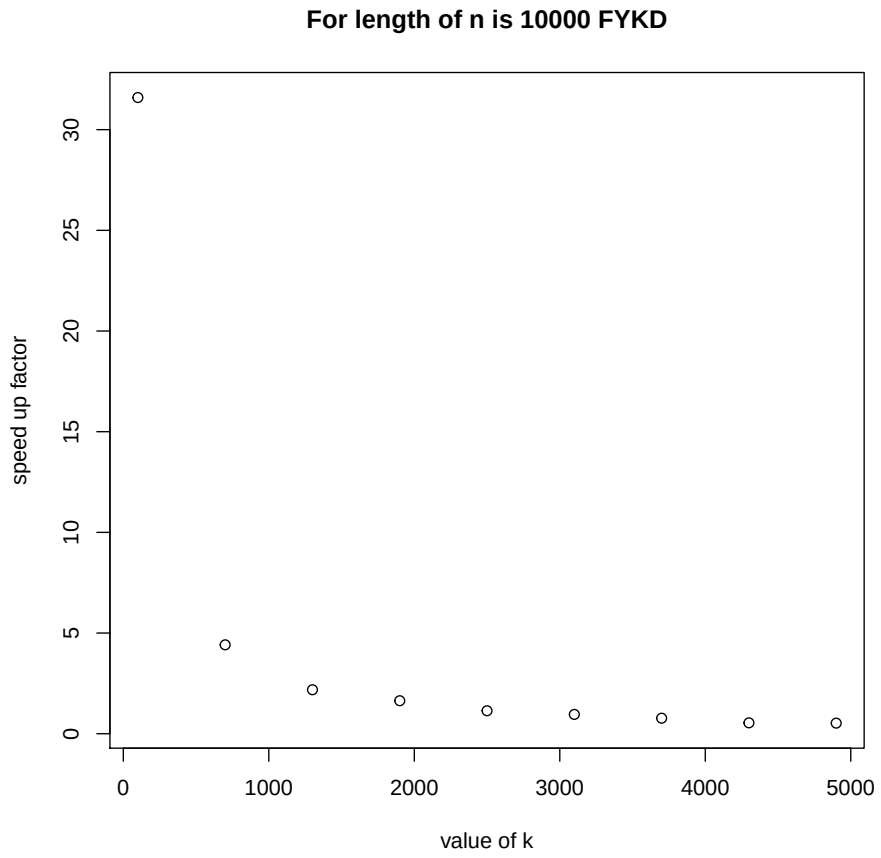
```r
## in every for-loop, and set the value to infinity after taking
## the value to our result.
FYKDnew <- function(x, k) {
  n <- length(x)
  randomx = runif(n)
  ksample = NULL
  for(i in 1:k) {
    tmp <- which.min(randomx)
    ksample[i] <- x[tmp]
    randomx[tmp] <- Inf
  }
  return(ksample)
}

mbmFYKD = microbenchmark("original" = {
  a <- FYKD(1:10000,500)
}, "new" = {
  b <- FYKDnew(1:10000,500)
})

mbmFYKD

## Unit: milliseconds
##      expr       min       lq      mean    median       uq       max neval
##  original 142.50505 171.4358 184.40077 179.36501 194.3956 280.31274   100
##       new  21.42572  23.5222  27.49191  24.78549  28.7779  57.96218   100

## Speed up factor
speedupFYKD = summary(mbmFYKD)$mean[1]/summary(mbmFYKD)$mean[2]
speedupFYKD

## [1] 6.707456
```

We can notice that the time was saved by a factor of 6.7074558. However, this function is still slower than the revised PIKK one.

Plot for FYKD

```r
## For length of n is 10000
set.seed(100)
speedup10000FYKD = NULL
for (i in seq(100,5000,600)) {
  mbmFYKD10000 = microbenchmark("original" = {
    a <- FYKD(1:10000,i)
    }, new = {
    b <- FYKDnew(1:10000,i)
  })
```

```
  indexFYKD = which(i == seq(100,5000,600))
  speedup10000FYKD[indexFYKD] = summary(mbmFYKD10000)$mean[1]/summary(mbmFYKD10000)$mean[2]
}
plot(x = seq(100,5000,600), y = speedup10000FYKD, xlab="value of k",
     ylab="speed up factor", main = "For length of n is 10000 FYKD")
```



This is an interesting graph. When I increase the size of K, the performance for the revised function goes down. Thus, it is good to use this function when k is relatively small.