Stat243: Problem Set 4

Qingan Zhao
SID: 3033030808

11 Oct. 2017

Problem 1

(a)

Initially there is only 'x', When we run $myFun(3)$, at the point that we evaluate $data = input$, we execute the promise that creates $input$ as acopy of 'x', nut it can just point to the same memory as 'x'. Then $data$ can also point to that same memory. So only one copy need exist. We can confirm this by injecting a line that prints the internal address information.

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

## @7ff218fc32e0 13 INTSXP g0c4 [NAM(2)] (len=10, tl=0) 1,2,3,4,5,...

myFun <- f(x)

## @7ff218fc32e0 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
## @7ff218fc32e0 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
```

(b)

Let vector x be $1 : 1000000$ such that it is large enough and we can concentrate on the bytes involved in the vector.

First we generate a sequence of bytes that store the information in $myFun$. Then create a new environment with x and generate a sequence of bytes that store the information in the environment.

```
#function "myFun"
x <- 1:1000000
f <- function(input){
  data <- input
  g <- function(param) return(param * data)
  return(g)
```

```r
}
myFun <- f(x)
#generate bytes storing information in the closure
length(serialize(myFun, NULL))
```

```
## [1] 8008698
```

```r
#create a new environment with vector x
e <- new.env()
e$x <- 1:1000000
#generate bytes storing information in the environment
length(serialize(e, NULL))
```

```
## [1] 4000183
```

From the result we know that the serialized size of $myFun$ is about twice as much as e, which is as expected given the answer to **(a)**.

(c)

The reason is that the argument *data* is evaluated in the function rather than the frame. When $myFun$ is called, x is evaluated in the calling frame and *data* is assigned to x in the fuction frame. Hence, whether x is removed or not, assigning *data* outside the function is useless. So when x is removed, $myFun()$ cannot be executed because *data* cannot be evaluated outside the function.

(d)

To make the code in **(c)** work, we can just assign *data* in the environment of the closure.

```r
x <- 1:10
f <- function(data){
  g <- function(param) return(param * data)
  return(g)
}
myFun <- f(x)
rm(x)
#assign data in the environment of the closure
environment(myFun)$data <- 1:10
myFun(3)
```

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

Now let's see the serialized size when $1 : 1000000$ are assigned:

```r
environment(myFun)$data <- 1:1000000
length(serialize(myFun, NULL))
```

```
## [1] 4006694
```

Comparing to **(b)**, this time the size of the closure is almost the same as e, indicating that there is only one copy of the vector.

Problem 2

(a)

Let's create a list consists of 2 vectors, and modify the first element of the first vector.

```
#create a list of 2 vectors
vec_a <- 1:10
vec_b <- 1:20
list_1 <- list(vec_a, vec_b)
.Internal(inspect(list_1))
#output in R (not Rstudio)
##  @7fa1a08d5740 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
##    @7fa1a41da620 13 INTSXP g0c4 [NAM(2)] (len=10, tl=0) 1,2,3,4,5,...
##    @7fa1a37e8200 13 INTSXP g0c5 [NAM(2)] (len=20, tl=0) 1,2,3,4,5,...

#modify the first element of the first vector
list_1[[1]][1] <- 2
.Internal(inspect(list_1))
#output in R (not Rstudio)
##  @7fa1a08d5740 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
##    @7fa1a3c6c1f8 14 REALSXP g0c5 [] (len=10, tl=0) 2,2,3,4,5,...
##    @7fa1a37e8200 13 INTSXP g0c5 [NAM(2)] (len=20, tl=0) 1,2,3,4,5,...
```

It seems that the answer is no, because the address of the first vector has been changed (i.e., the first vector has been copied while the list and the second vector has not).

(b)

```
#make a copy of the list
list_2 <- list_1
.Internal(inspect(list_2))
#output in R (not Rstudio)
##  @7fa1a3c99990 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##    @7fa1a3c93150 14 REALSXP g0c5 [] (len=10, tl=0) 2,2,3,4,5,...
##    @7fa1a3c930a8 13 INTSXP g0c5 [NAM(2)] (len=20, tl=0) 1,2,3,4,5,...

#make a change in the first vector
list_2[[1]] <- 1:15
.Internal(inspect(list_2))
#output in R (not Rstudio)
##  @7fa1a3c99a00 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
##    @7fa1a3c9a140 13 INTSXP g0c4 [NAM(1)] (len=15, tl=0) 1,2,3,4,5,...
##    @7fa1a3c930a8 13 INTSXP g0c5 [NAM(2)] (len=20, tl=0) 1,2,3,4,5,...
```

It seems that there is a copy-on-change going on. When a change is made to one of the vectors in one of the lists, the copy of the list and the relevant vector are both made.

(c)

Let's create a list of 2 lists. Then copy the list and add an element to the second list.

```r
#create a list of 2 lists
list_3 <- list(list_1, list_2)
#copy the list
list_4 <- list_3
.Internal(inspect(list_3))
#output in R (not Rstudio)
## @7fa1a3c99a38 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##   @7fa1a3c99990 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##     @7fa1a3c93150 14 REALSXP g0c5 [NAM(2)] (len=10, tl=0) 2,2,3,4,5,...
##     @7fa1a3c930a8 13 INTSXP g0c5 [NAM(2)] (len=20, tl=0) 1,2,3,4,5,...
##   @7fa1a3c99a00 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##     @7fa1a3c9a140 13 INTSXP g0c4 [NAM(1)] (len=15, tl=0) 1,2,3,4,5,...
##     @7fa1a3c930a8 13 INTSXP g0c5 [NAM(2)] (len=20, tl=0) 1,2,3,4,5,...

#add an element 1:10 to the second list
list_4[[3]] <- list(1:10)
.Internal(inspect(list_4))
#output in R (not Rstudio)
## @7fa1a3c91650 19 VECSXP g0c3 [NAM(1)] (len=3, tl=0)
##   @7fa1a3c99990 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##     @7fa1a3c93150 14 REALSXP g0c5 [NAM(2)] (len=10, tl=0) 2,2,3,4,5,...
##     @7fa1a3c930a8 13 INTSXP g0c5 [NAM(2)] (len=20, tl=0) 1,2,3,4,5,...
##   @7fa1a3c99a00 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##     @7fa1a3c9a140 13 INTSXP g0c4 [NAM(1)] (len=15, tl=0) 1,2,3,4,5,...
##     @7fa1a3c930a8 13 INTSXP g0c5 [NAM(2)] (len=20, tl=0) 1,2,3,4,5,...
##   @7fa1a3c99218 19 VECSXP g0c1 [NAM(1)] (len=1, tl=0)
##     @7fa1a3c9a1a8 13 INTSXP g0c4 [] (len=10, tl=0) 1,2,3,4,5,...
```

From "$.Internal(inspect(list\_3))$" and "$.Internal(inspect(list\_4))$", it seems that the list has been copied and the lists in the first list has not been copied. So the data of lists (except the new added one) in the two lists are shared between the two lists.

(d)

```r
#code provided by the instructor
gc()
#output in R (not Rstudio)
##          used (Mb) gc trigger (Mb) max used (Mb)
## Ncells  422084 22.6     750400 40.1   423584 22.7
## Vcells 2222952 17.0    3276122 25.0  2233626 17.1
tmp <- list()
x <- rnorm(1e7)
tmp[[1]] <- x
tmp[[2]] <- x
.Internal(inspect(tmp))
#output in R (not Rstudio)
## @7f934c235408 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
##   @11acf8000 14 REALSXP g0c7 [NAM(2)] (len=10000000, tl=0)
##     -0.20553,-0.0927882,-2.31219,0.434319,-1.18064,...
##   @11acf8000 14 REALSXP g0c7 [NAM(2)] (len=10000000, tl=0)
##     -0.20553,-0.0927882,-2.31219,0.434319,-1.18064,...
object.size(tmp)
#output in R (not Rstudio)
```

```
##  160000136 bytes
gc()
#output in R (not Rstudio)
##             used (Mb) gc trigger  (Mb) max used (Mb)
##  Ncells   422986 22.6     750400  40.1   450643 24.1
##  Vcells 12224785 93.3   18003831 137.4 12241025 93.4
```

The reason is that *tmp*[[1]] and *tmp*[[2]] are sharing the memory as shown in ".*Internal*(*inspect*(*tmp*))", which is about 80 MB. However, the *object.size*() function does not account for the shared elements within an object, so the result is about $80 * 2 = 160$ MB. We can use *object_size*() instead to get the expected result:

```
object_size(tmp)
#output in R (not Rstudio)
##  80 MB
```

Problem 3

First, let's calculate the running time of the original code:

```
#original code provided by the instructor
load('/Users/franklin/Projects/Problem-sets-of-Statistical-Computing/R Programming/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
```

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

#calculate the running time of the original code
system.time(oneUpdate(A, n, K, theta.init))

##    user  system elapsed
##   8.584   0.319   9.091
```

Now let's refine the code to improve the efficiency. Basically I changed the 4 *for* loops in the original code into only one *for* loop and used matrix multiplication, and discarded some unnecessary code, etc.

```r
#the ll() function remains unchanged
oneUpdate <- function(A, n, K, theta.old, thresh = 0.1){
  #'theta.old1 <- theta.old' is not needed
  Theta.old <- theta.old %*% t(theta.old)
  L.old <- ll(Theta.old, A)
  #'q' can be replaced by a temporary value 'tmp' in the only for loop
  #discard 2 of the 4 for loops using matrix multiplication
  #combine the rest 2 for loops into one
  theta.new <- theta.old
  for (z in 1:K){
    #this is actually q[, , z]
    tmp <- theta.old[, z] %*% t(theta.old[, z]) / Theta.old
    #calculate theta.new using only one for loop
    theta.new[, z] <- rowSums(A * tmp) / sqrt(sum(A * tmp))
  }
  #the rest of the code remains unchanged
  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))
}
out_new <- oneUpdate(A, n, K, theta.init)

#calculate the running time of the original code
system.time(oneUpdate(A, n, K, theta.init))

##    user  system elapsed
##   0.459   0.369   0.840
```

From the result it seems that the efficiency of the refined code is greatly improved. Now let's check if the result is correct.

```r
identical(out, out_new)

## [1] TRUE
```

Problem 4

(a)(b)

First, let's run the code of the original PIKK and FYKD alorithms:

```
#PIKK and FYKU code provided by the instructor
PIKK <- function(x, k) {
  x[sort(runif(length(x)), index.return = TRUE)$ix[1:k]]
}

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])
}
```

For PIKK algorithm, I simply used *order*() function instead of *sort*(), and it turns out that the efficency is improved.

```
#refined PIKK code 'PIKK_pro'
PIKK_pro <- function(x, k) {
  order(runif(x))[1:k]
}
```

Now let's compare $PIKK()$ with $PIKK\_pro()$ given that $n = 10000$ and $k = 500$:

```
x <- 1:100000
k <- 5000
microbenchmark(PIKK(x, k), PIKK_pro(x, k))

## Unit: milliseconds
##            expr      min       lq     mean   median       uq      max
##      PIKK(x, k) 9.792397 11.31958 13.90442 12.59216 15.32217 27.32815
##  PIKK_pro(x, k) 7.565160  9.22534 11.28051 10.21533 11.70769 36.98466
##  neval
##    100
##    100
```

For FYKD algorithm, I changed the range of the for loop from n to k, and used a vector with pre-allocated memory to hold the sampled data. The for loop is also slightly changed to reduce the variable numbers.

```
FYKD_pro <- function(x, k){
  #Pre-allocate memory
  vec <- vector('numeric', k)
  n <- length(x)
  #similar to the FYKD code but using 'vec' rather than 'tmp'
  for(i in 1:k){
    j = sample(i:n, 1)
    vec[i] <- x[j]
```

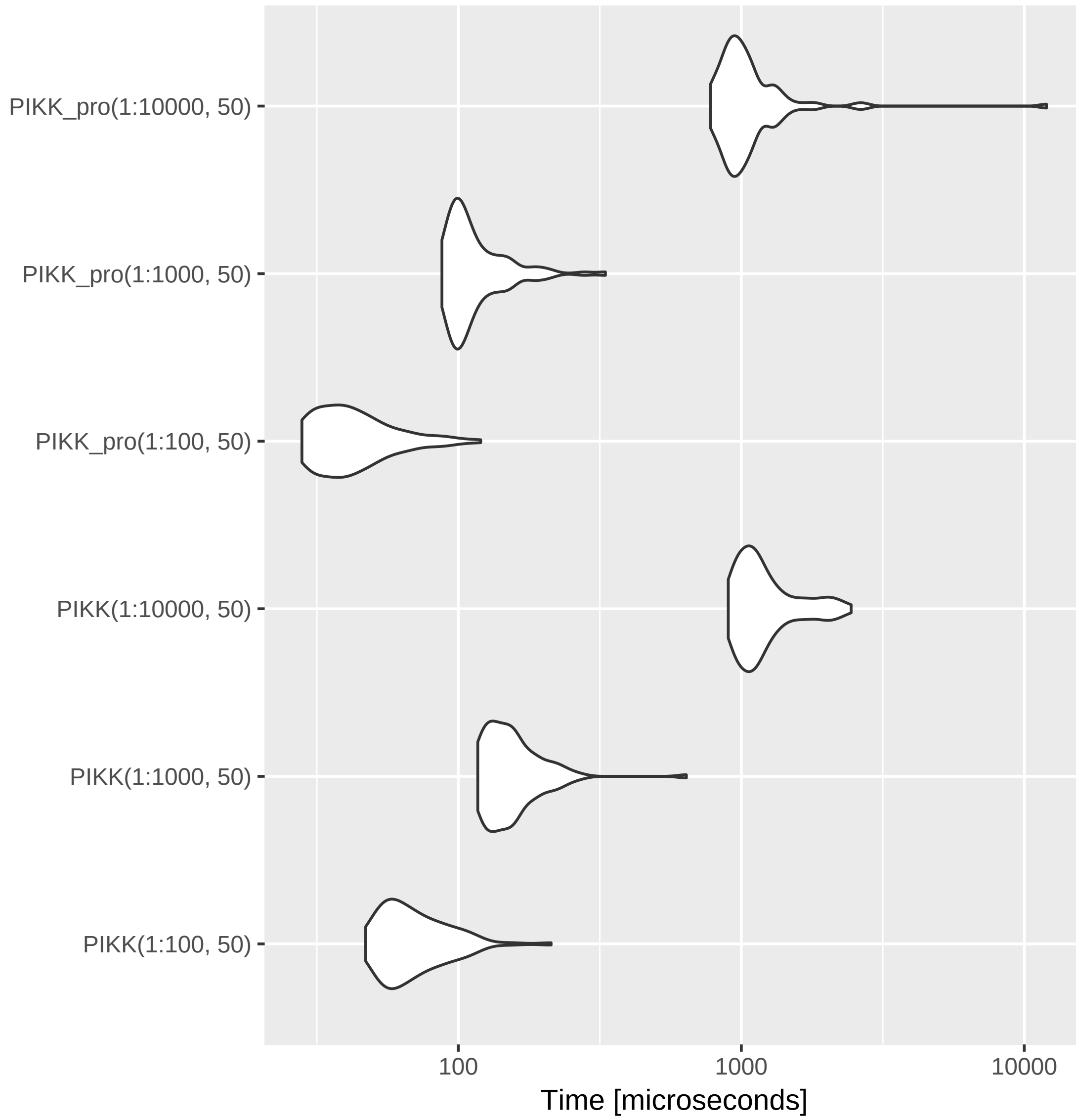
```
    x[j] <- x[i]
    x[i] <- vec[i]
  }
  return(vec)
}
```

Now let's compare $FYKD()$ with $FYKD\_pro()$ given that $n = 10000$ and $k = 500$:

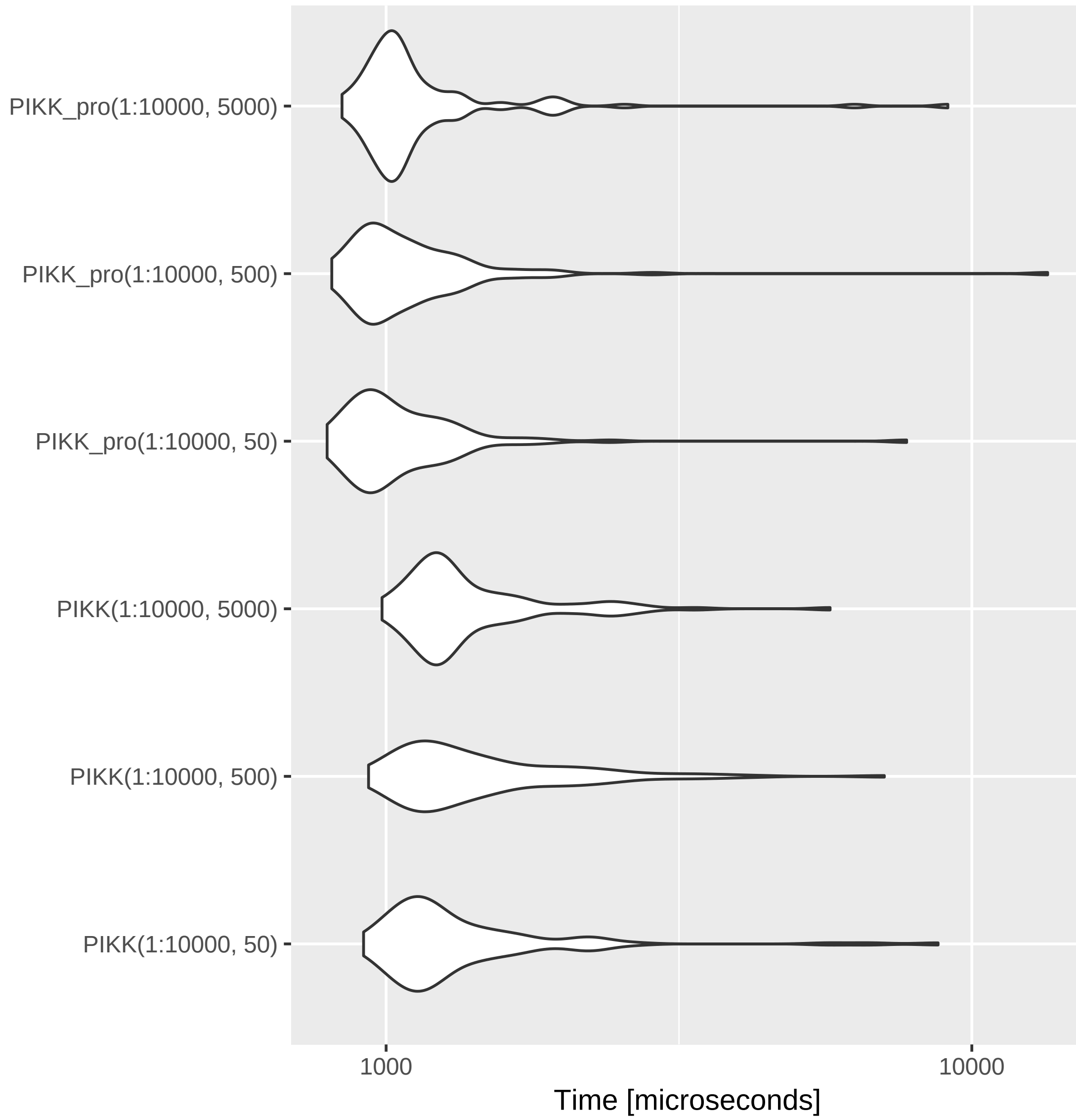
```
x <- 1:10000
k <- 500
microbenchmark(FYKD(x, k), FYKD_pro(x, k))

## Unit: milliseconds
##            expr        min         lq      mean     median        uq
##      FYKD(x, k) 118.940924 141.040413 151.72037 148.955632 157.13814
##  FYKD_pro(x, k)   5.966581   7.491117  12.83031   8.665091  10.97397
##        max neval
##  210.72286   100
##   72.74627   100
```

Finally, Let's produce some plots showing the efficiency as k and n vary.

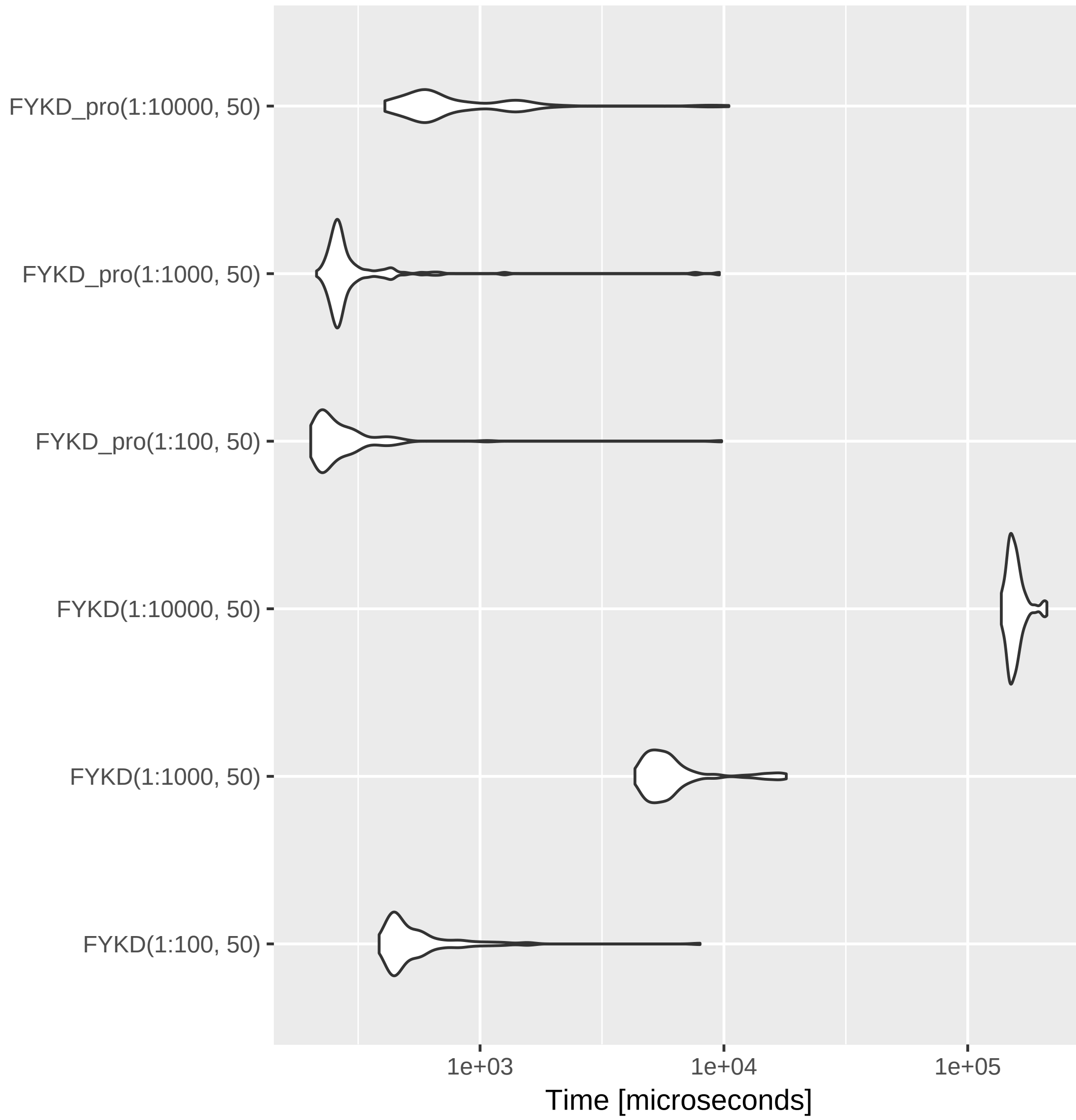
```
#let k be 50, n vary from 100 to 10000, and compare 'PIKK' with 'PIKK_pro'
plot_1 <- microbenchmark(PIKK(1:100, 50),
                         PIKK(1:1000, 50),
                         PIKK(1:10000, 50),
                         PIKK_pro(1:100, 50),
                         PIKK_pro(1:1000, 50),
                         PIKK_pro(1:10000, 50))
autoplot(plot_1)
```



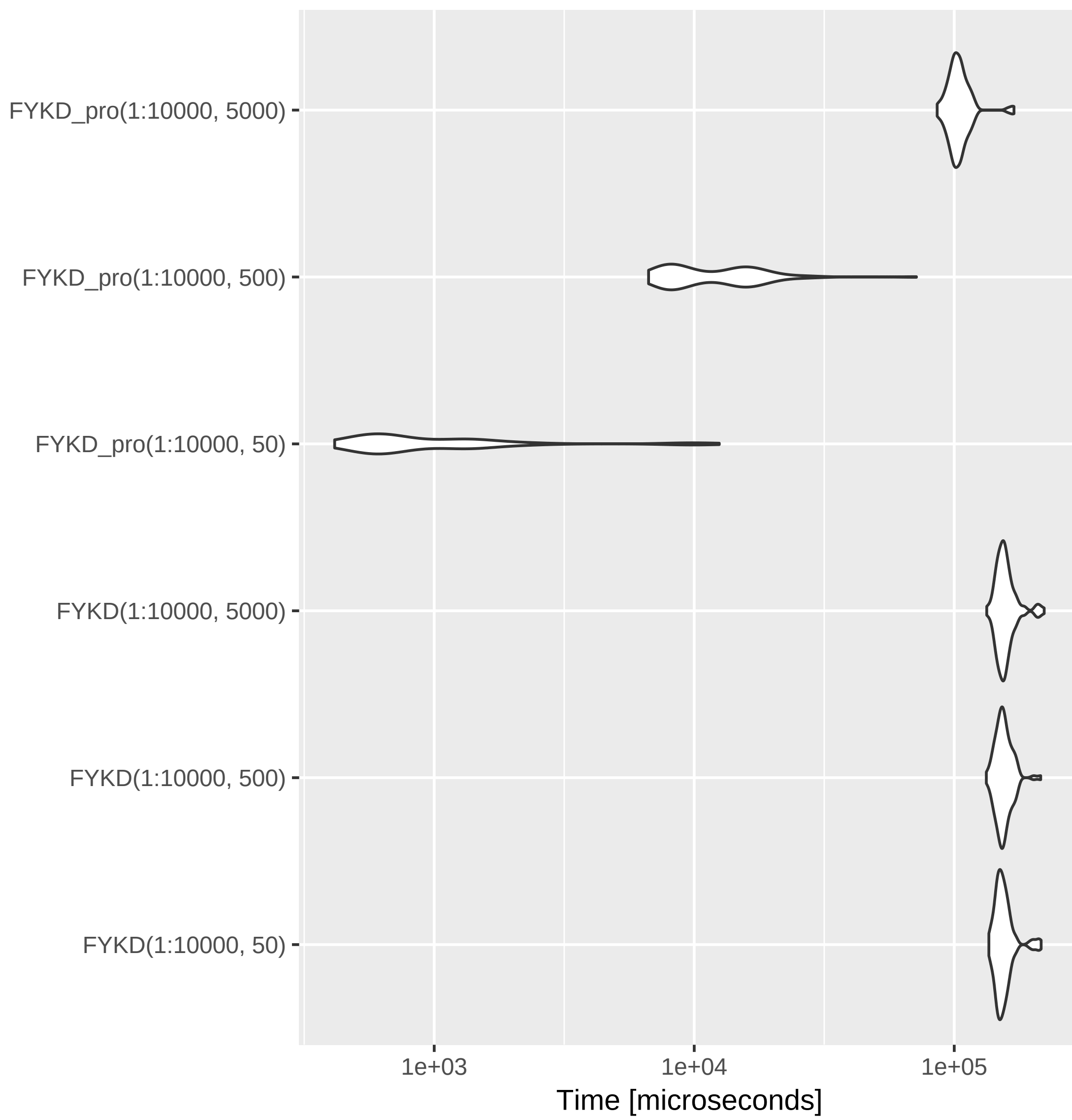

```
#let n be 10000, k vary from 50 to 5000, and compare 'PIKK' with 'PIKK_pro'
plot_2 <- microbenchmark(PIKK(1:10000, 50),
                         PIKK(1:10000, 500),
                         PIKK(1:10000, 5000),
                         PIKK_pro(1:10000, 50),
                         PIKK_pro(1:10000, 500),
                         PIKK_pro(1:10000, 5000))
autoplot(plot_2)
```




```r
#let k be 50, n vary from 100 to 10000, and compare 'FYKD' with 'FYKD_pro'
plot_3 <- microbenchmark(FYKD(1:100, 50),
                         FYKD(1:1000, 50),
                         FYKD(1:10000, 50),
                         FYKD_pro(1:100, 50),
                         FYKD_pro(1:1000, 50),
                         FYKD_pro(1:10000, 50))
autoplot(plot_3)
```




```r
#let n be 10000, k vary from 50 to 5000, and compare 'FYKD' with 'FYKD_pro'
plot_4 <- microbenchmark(FYKD(1:10000, 50),
                         FYKD(1:10000, 500),
                         FYKD(1:10000, 5000),
                         FYKD_pro(1:10000, 50),
                         FYKD_pro(1:10000, 500),
                         FYKD_pro(1:10000, 5000))
autoplot(plot_4)
```


FYKD_pro(1:10000, 5000)
FYKD_pro(1:10000, 500)
FYKD_pro(1:10000, 50)
FYKD(1:10000, 5000)
FYKD(1:10000, 500)
FYKD(1:10000, 50)
1e+03
1e+04
1e+05
Time [microseconds]