ps4

Kehsin Su Esther 3033114294

October 8, 2017

```
knitr::opts_chunk$set(tidy = TRUE, cache = TRUE)
library(pryr)

## Warning:  package 'pryr' was built under R version 3.3.3
```

1 Q1

1.1 (a)

One time, when the input copy to data to run the g function.

1.2 (b)

As shows in the following, the memory used for calculate myFun(3) is 80022, which is nearly 2 times memory of x. It happened because the system would store data and input(x) in two different memories, SO the used the memory become twice.However, it still only copy one time.

```
mem_change(x <- 1:10000) #39.6 kB

## 53.1 kB

length( serialize(x, NULL) ) # 40022

## [1] 40022

mem_change( f <- function(input){
        data <- input
        g <- function(param) #return(param * data)
        print( length( serialize(g, NULL) ) ) #459
        return(g)
} ) #-2.44 kB

## 1.32 kB

length( serialize(f, NULL) ) #1517

## [1] 17617

mem_change(myFun <- f(x))  #-40.6 kB

## 1.66 kB

length( serialize(myFun, NULL) ) # 895

## [1] 97116
```

```
mem_change(data <- 100 ) #728 B
```

```
## 1.15 kB
```

```
length( serialize(data, NULL) ) #30
```

```
## [1] 30
```

```
mem_change(myFun(3)) #224 B
```

```
## [1] 97116
## 1.23 kB
```

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

```
## [1] 97116
## [1] 26
```

```
mem_change( x <- 100 ) #-16 B
```

```
## 1.1 kB
```

```
length( serialize(x, NULL) ) #30
```

```
## [1] 30
```

```
mem_change(myFun(3)) #1.3 kB
```

```
## [1] 97116
## 1.18 kB
```

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

```
## [1] 97116
## [1] 26
```

1.3 (c)

Because the memory that store x is already been free. The "data" is only a local variable, so when call myFun(3), it will turn to f(x), which is also g that require data(input of function f and in the case, it was x). However, there is no x in the memory, so it return an error.

1.4 (d)

Adding force function can inside the function can force it to evaluate a function argument(data). Hence, the input argument will exist. There won't exist such error happened in part(c).

```
mem_change(x <- 1:1e+06)
```

```
## 4 MB
```

```
mem_change(f <- function(data) {
    # data <<- data
    force(data)
    g <- function(param) {

        return(param * data)
```

```
    }


    return(g)
})

## 1.07 kB

mem_change(myFun <- f(x))

## 1.52 kB

mem_change(rm(x))

## 1.3 kB

mem_change(data <- 100)

## -2.71 kB

mem_change(myFun(3))

## 1.42 kB
```

2 Q2

```
# a
library(data.table)
mem_change(t <- data.table(a = c(1:500), b = c(501:1000)))
length(serialize(t, NULL))
.Internal(inspect(t))
add1 <- address(t)
tracemem(t)
t[1, `:=`(b, 777)]
add2 <- address(t)
identical(add1, add2)
length(serialize(a, NULL))
.Internal(inspect(t))
# b
mem_change(t1 <- rep(list(1:1000), 500))
.Internal(inspect(t1))
tracemem(t1)
mem_change(t2 <- copy(t1))
t1[[1]][1] = 11
.Internal(inspect(t1))
```

2.1 (a)

The results are following. When modifying one element of first vecoter, the address of the second vector didn't change. What would cahnge are the address that store the two addresses of vectors and the address that store vector one.Hence,the entire list won't be copy under this change.

```
mylista <- list(c(1:10000), c(100001:2e+05))
.Internal(inspect(mylista))

## @0x0000000019068f90 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##   @0x0000000019cb0010 13 INTSXP g0c7 [] (len=10000, tl=0) 1,2,3,4,5,...
##   @0x00007ff5ff8d0010 13 INTSXP g0c7 [] (len=100000, tl=0) 100001,100002,100003,100004,100005,...

mylista[[1]][[1]] <- 0
.Internal(inspect(mylista))

## @0x0000000018f2adc8 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
##   @0x0000000019cddf48 14 REALSXP g0c7 [] (len=10000, tl=0) 0,2,3,4,5,...
##   @0x00007ff5ff8d0010 13 INTSXP g0c7 [NAM(2)] (len=100000, tl=0) 100001,100002,100003,100004,100005,
```

(b) From the following result, we can observe that only the memory address of modified vector has change. The other vectors didn't change. A new memory was created to store the modified vector of mylista, but the address that store other vectors of mylista and mylistb still maintain the same.

```
mylistb <- mylista
# both mylista and mylistb store in the same address
.Internal(inspect(mylista))

## @0x0000000018f2adc8 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##   @0x0000000019cddf48 14 REALSXP g0c7 [] (len=10000, tl=0) 0,2,3,4,5,...
##   @0x00007ff5ff8d0010 13 INTSXP g0c7 [NAM(2)] (len=100000, tl=0) 100001,100002,100003,100004,100005,

.Internal(inspect(mylistb))

## @0x0000000018f2adc8 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##   @0x0000000019cddf48 14 REALSXP g0c7 [] (len=10000, tl=0) 0,2,3,4,5,...
##   @0x00007ff5ff8d0010 13 INTSXP g0c7 [NAM(2)] (len=100000, tl=0) 100001,100002,100003,100004,100005,

# make a change in mylista
mylista[[1]][[1]] <- 99
.Internal(inspect(mylista))

## @0x0000000017d12ec8 19 VECSXP g0c2 [MARK,NAM(1)] (len=2, tl=0)
##   @0x0000000019da5108 14 REALSXP g0c7 [MARK] (len=10000, tl=0) 99,2,3,4,5,...
##   @0x00007ff5ff8d0010 13 INTSXP g0c7 [MARK,NAM(2)] (len=100000, tl=0) 100001,100002,100003,100004,10

.Internal(inspect(mylistb))

## @0x0000000018f2adc8 19 VECSXP g0c2 [MARK,NAM(2)] (len=2, tl=0)
##   @0x0000000019cddf48 14 REALSXP g0c7 [MARK,NAM(2)] (len=10000, tl=0) 0,2,3,4,5,...
##   @0x00007ff5ff8d0010 13 INTSXP g0c7 [MARK,NAM(2)] (len=100000, tl=0) 100001,100002,100003,100004,10
```

2.2 (c)

What is actually copy is the first list of myList2. The second list of myList are still maintain in the same address and that is also the adodress of the second list of myList2. Other than the modified list, the other list in myList and myList2 still share the same address. Hence, we can conclud that when making part of change between two list that point to the same ddress, a copy will be made to the change part. The other part still maintain the same address.

```
myList <- list(list(1:10000), list(100001:2e+05))
tracemem(myList)
```

```
## [1] "<000000001474B190>"

.Internal(inspect(myList))

## @0x000000001474b190 19 VECSXP g0c2 [NAM(2),TR] (len=2, tl=0)
##   @0x000000001423e150 19 VECSXP g0c1 [] (len=1, tl=0)
##     @0x0000000015b22880 13 INTSXP g0c7 [] (len=10000, tl=0) 1,2,3,4,5,...
##   @0x000000001423e120 19 VECSXP g0c1 [] (len=1, tl=0)
##     @0x00007ff5ffb30010 13 INTSXP g0c7 [] (len=100000, tl=0) 100001,100002,100003,100004,100005,...

myList2 <- myList
# share the same memory
.Internal(inspect(myList2))

## @0x000000001474b190 19 VECSXP g0c2 [NAM(2),TR] (len=2, tl=0)
##   @0x000000001423e150 19 VECSXP g0c1 [] (len=1, tl=0)
##     @0x0000000015b22880 13 INTSXP g0c7 [] (len=10000, tl=0) 1,2,3,4,5,...
##   @0x000000001423e120 19 VECSXP g0c1 [] (len=1, tl=0)
##     @0x00007ff5ffb30010 13 INTSXP g0c7 [] (len=100000, tl=0) 100001,100002,100003,100004,100005,...

# add an element to myList
myList2[[1]] <- append(myList2[[1]][[1]], 100)

## tracemem[0x000000001474b190 -> 0x0000000013076300]: eval eval withVisible withCallingHandlers doTryC

.Internal(inspect(myList))

## @0x000000001474b190 19 VECSXP g0c2 [NAM(2),TR] (len=2, tl=0)
##   @0x000000001423e150 19 VECSXP g0c1 [NAM(2)] (len=1, tl=0)
##     @0x0000000015b22880 13 INTSXP g0c7 [NAM(2)] (len=10000, tl=0) 1,2,3,4,5,...
##   @0x000000001423e120 19 VECSXP g0c1 [NAM(2)] (len=1, tl=0)
##     @0x00007ff5ffb30010 13 INTSXP g0c7 [] (len=100000, tl=0) 100001,100002,100003,100004,100005,...

.Internal(inspect(myList2))

## @0x0000000013076300 19 VECSXP g0c2 [NAM(1),TR] (len=2, tl=0)
##   @0x0000000015b844c0 14 REALSXP g0c7 [NAM(2)] (len=10001, tl=0) 1,2,3,4,5,...
##   @0x000000001423e120 19 VECSXP g0c1 [NAM(2)] (len=1, tl=0)
##     @0x00007ff5ffb30010 13 INTSXP g0c7 [] (len=100000, tl=0) 100001,100002,100003,100004,100005,...
```

2.3 (d)

The memory will become double when call object.size function is because x is copy 2 times when pointing both tmp[[1]] and tmp[[2]] to x. However, as they all point to the same address. In fact, it only increase one size of x, so it's about 8*1e+07 memory used as reveal in gc().

```
gc()
```

	used	(Mb)	gc trigger	(Mb)	max used	(Mb)
Ncells	587339	31.4	940480	50.3	940480	50.3
Vcells	1167586	9.0	2060183	15.8	2057986	15.8

```
tmp <- list()
x <- rnorm(1e+07)
tmp[[1]] <- x
tmp[[2]] <- x
.Internal(inspect(tmp))
```

```
## @0x0000000019157de0 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
##   @0x00007ff5fac10010 14 REALSXP g0c7 [NAM(2)] (len=10000000, tl=0) -0.0314003,-0.407658,-0.922024,0
##   @0x00007ff5fac10010 14 REALSXP g0c7 [NAM(2)] (len=10000000, tl=0) -0.0314003,-0.407658,-0.922024,0

object.size(tmp)

## 160000136 bytes

gc()
```

	used	(Mb)	gc trigger	(Mb)	max used	(Mb)
Ncells	588065	31.5	940480	50.3	940480	50.3
Vcells	11168607	85.3	16484004	125.8	11223966	85.7

3 Q3

```
# original one
load("C:/Users/Esther/Desktop/stat243-fall-2017-master/ps/ps4prob3.Rda")  # should have A, n, K

a <- proc.time()
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))
}
proc.time() - a  # elapsed

##    user  system elapsed
```

```
##       0       0       0
```

```
# initialize the parameters at random starting values temp <-
# matrix(runif(n*K), n, K)
temp <- matrix(runif(n * K), n, K)
theta.init <- temp/rowSums(temp)

# do single update
b <- proc.time()
out <- oneUpdate(A, n, K, theta.init)
proc.time() - b  # elapsed
```

```
##    user  system elapsed
##   92.42    0.19   92.71
```

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

As show in the following result, when modified the 3 for loop into doing one time, and change the if else case into a single indicator, the running time decreased to 1/92.

```
# new one
load("C:/Users/Esther/Desktop/stat243-fall-2017-master/ps/ps4prob3.Rda")  # should have A, n, K
a <- proc.time()

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))
    # as the matrix will directly product each corresponding element, it's
    # unnecessary to do the n by n time for loop.

    for (z in 1:K) {

        # remove the repeating for loop to do it only k times reduce the if as a
        # indicator to check zero case
        q[, , z] <- (theta.old[, z] != 0) * theta.old[, z] * theta.old[, z]/Theta.old

    }

    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))
```

```
}
proc.time() - a  # elapsed

##    user  system elapsed
##       0       0       0

# initialize the parameters at random starting values temp <-
# matrix(runif(n*K), n, K)
temp <- matrix(runif(n * K), n, K)
theta.init <- temp/rowSums(temp)

# do single update
b <- proc.time()
out <- oneUpdate(A, n, K, theta.init)
proc.time() - b  # elapsed

##    user  system elapsed
##    0.81    0.23    1.05

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

4 Q4

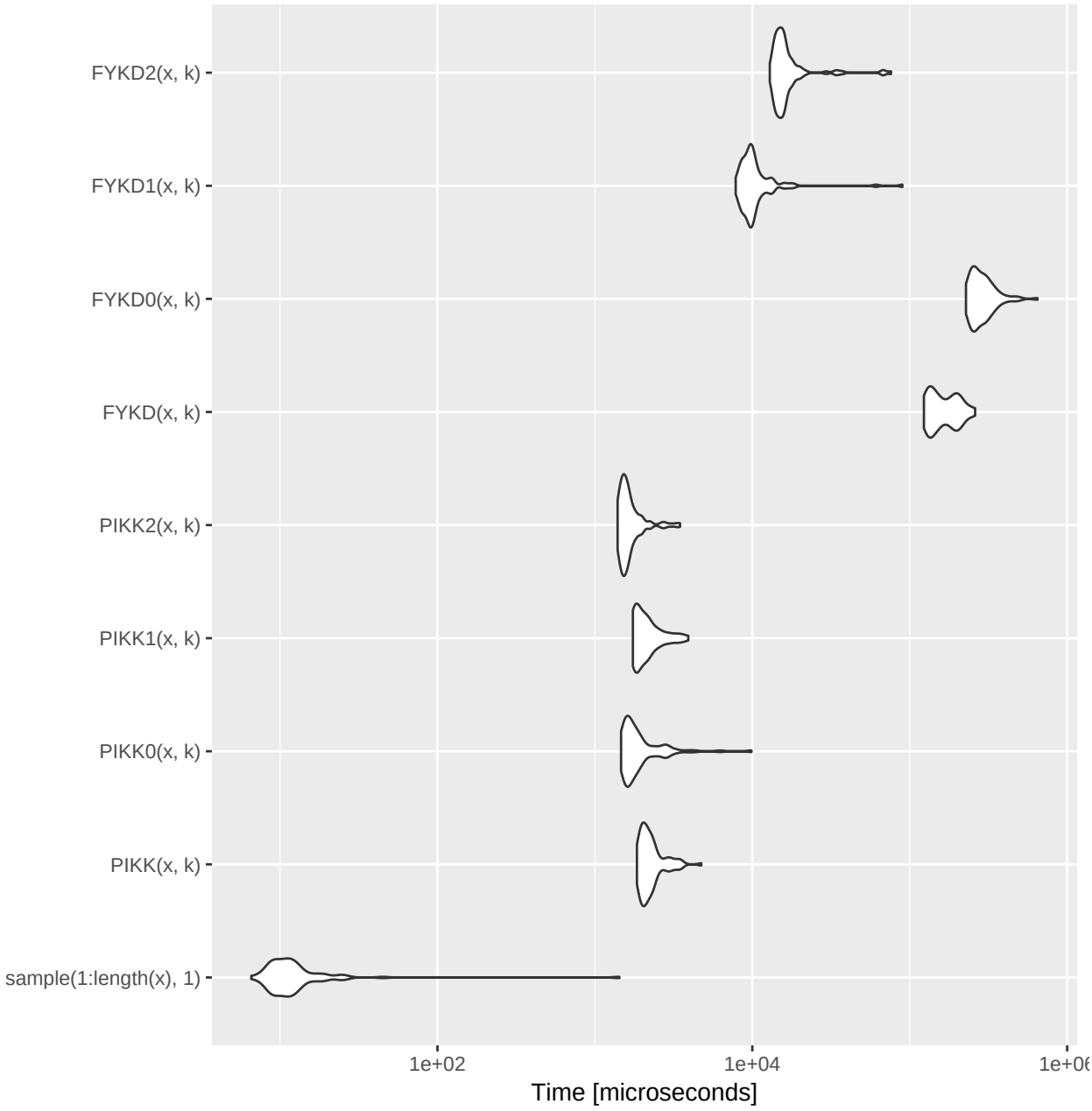
```
# old
library(rbenchmark)
library(microbenchmark)
PIKK <- function(x, k) {
    x[sort(runif(length(x)), index.return = TRUE)$ix[1:k]]
}

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])
}
x <- c(1:10000)
k = 500
# benchmark(PIKK(x,k)) benchmark(FYKD(x,k))
```

4.1 (a)(b)

```
# new
library(grr)
stime <- as.numeric(Sys.time())
set.seed(stime)

# original one but change the function sort into sort2
```

```
PIKK0 <- function(x, k) {
    x[rank(sort2(runif(length(x))))[1:k]]
}


# change the method into order to get the rank of data
PIKK1 <- function(x, k) {
    x[order(runif(x))[1:k]]
}


# change use the same method as PIKK2, but change the function from order to
# order2
PIKK2 <- function(x, k) {
    x[order2(runif(x))[1:k]]
}

# used the original function, but change the function sample to sample2 to
# compare the speed between two functions
FYKD0 <- function(x, k) {
    n <- length(x)
    for (i in 1:n) {
        j = sample2(i:n, 1)
        tmp <- x[i]
        x[i] <- x[j]
        x[j] <- tmp
    }
    return(x[1:k])
}
# change the function to reduce the for loop time
FYKD1 <- function(x, k) {
    n <- length(x)
    tmp <- vector("list", k)  # a list to put in selected items
    # pick one item each time and do it k times to pick totally k items
    for (i in 1:k) {
        j = sample(1:n, 1)  # randomly select the postion of which item to pick
        tmp[i] <- x[j]  #put the picked item into tmp list
        # put the last item of the sample into the position where the item has
        # already been pick
        x[j] <- x[n]
        # The last item has already been copy into the selected position. We do not
        # need to consider it when when randomly choose a position next time. The
        # range we choose for next time will only between the n-1 items. After
        # repeating, the range will continuously reducing, so the efficency will
        # increase.
        n = n - 1
    }
    return(tmp[1:k])
}
# same as FYKD2, but try the sample2 function
FYKD2 <- function(x, k) {
    n <- length(x)
    tmp <- vector("list", k)
    for (i in 1:k) {
        j = sample2(1:n, 1)
        tmp[i] <- x[j]
```

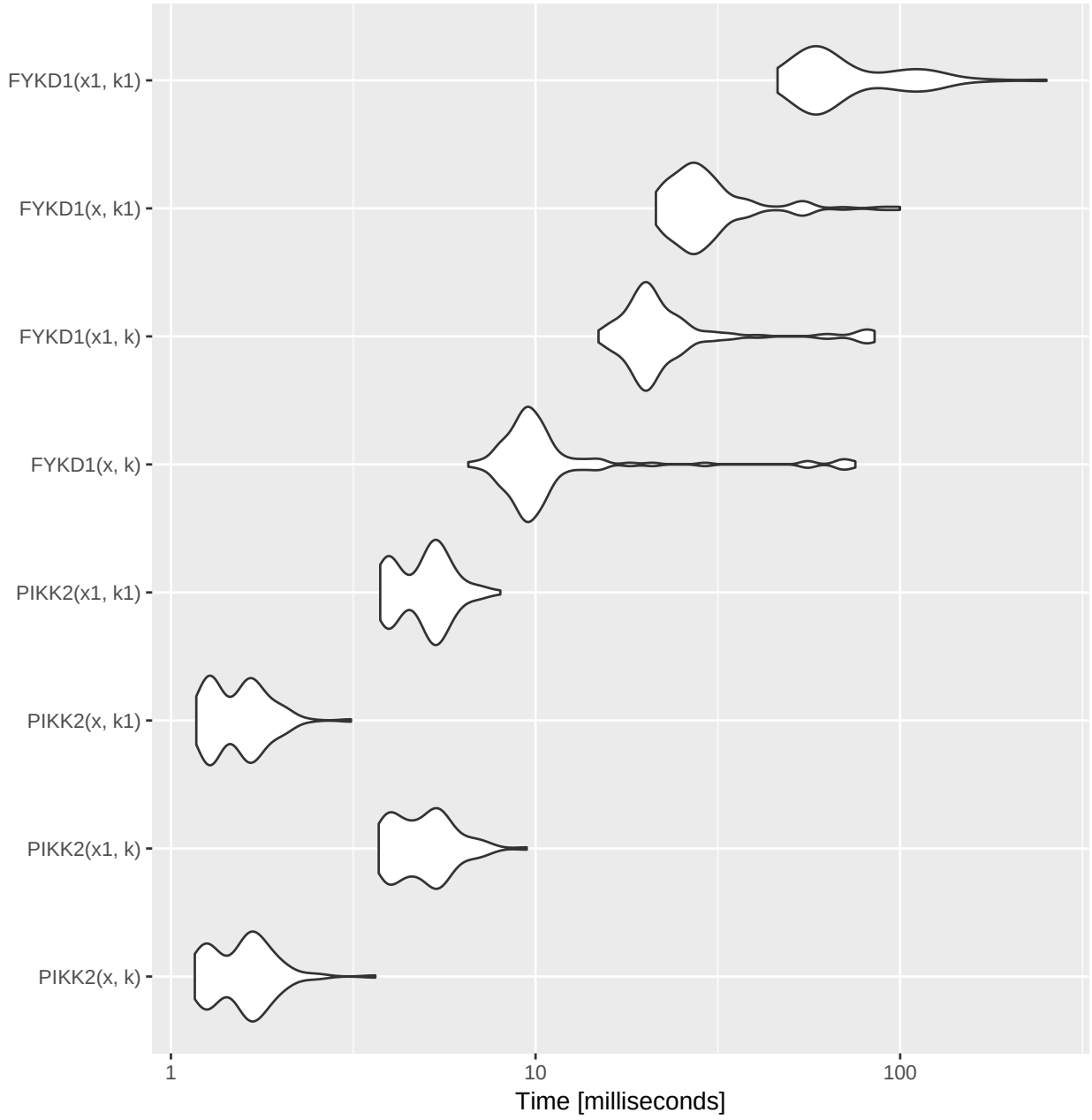
```
        x[j] <- x[n]
        n = n - 1
    }
    return(tmp[1:k])
}
x <- c(1:10000)
k = 500
# benchmark(PIKK0(x,k)) benchmark(FYKD0(x,k)) benchmark(PIKK1(x,k))
# benchmark(FYKD1(x,k)) benchmark(PIKK2(x,k)) benchmark(FYKD2(x,k))

result <- microbenchmark(sample(1:length(x), 1), PIKK(x, k), PIKK0(x, k), PIKK1(x,
    k), PIKK2(x, k), FYKD(x, k), FYKD0(x, k), FYKD1(x, k), FYKD2(x, k))
library(ggplot2)
autoplot(result)
```



```
print(result)
```

```
## Unit: microseconds
##                     expr        min         lq         mean     median
##  sample(1:length(x), 1)      6.565      9.231     26.48222     11.077
##              PIKK(x, k)   1849.026   1993.026   2303.20438   2162.052
##             PIKK0(x, k)   1466.667   1576.001   2042.83510   1732.309
##             PIKK1(x, k)   1743.590   1810.871   2189.54280   2021.539
##             PIKK2(x, k)   1394.052   1493.744   1699.60237   1575.385
##              FYKD(x, k) 123016.615 134924.924 167802.88442 154083.078
##             FYKD0(x, k) 227510.975 249160.410 294764.74687 278533.539
##             FYKD1(x, k)   7842.462   8981.744  11578.37153   9802.051
##             FYKD2(x, k)  12908.308  14355.897  17980.04535  15496.001
##          uq        max neval
##      12.718   1431.385   100
##    2374.360   4742.564   100
##    2008.000   9874.872   100
##    2312.001   3920.820   100
##    1745.231   3474.051   100
##  200246.154 260094.359   100
##  315925.539 647808.411   100
##   10615.794  89840.000   100
##   16660.924  75988.103   100
```

```
# by the plot, we know that PIKK2 and FYKD1 are fastest, so use them to
# observe how they change when n and k change
x1 <- c(1:30000)
k1 = 1500

result2 <- microbenchmark(PIKK2(x, k), PIKK2(x1, k), PIKK2(x, k1), PIKK2(x1,
    k1), FYKD1(x, k), FYKD1(x1, k), FYKD1(x, k1), FYKD1(x1, k1))
autoplot(result2)
```




```
print(result2)
```

```
## Unit: milliseconds
##           expr       min        lq      mean    median        uq
##    PIKK2(x, k)  1.162667  1.288821  1.606536  1.614769  1.765539
##   PIKK2(x1, k)  3.714462  4.065436  5.010786  5.051283  5.529436
##   PIKK2(x, k1)  1.173744  1.293744  1.561502  1.597949  1.721231
##  PIKK2(x1, k1)  3.749744  4.047591  5.021190  5.114462  5.536205
##    FYKD1(x, k)  6.539487  8.930667 13.552648  9.669949 10.701333
##   FYKD1(x1, k) 14.881642 19.086154 25.704489 20.660513 24.026667
##   FYKD1(x, k1) 21.386256 24.979692 31.317383 27.462359 31.555077
##  FYKD1(x1, k1) 46.107898 56.726564 76.176993 61.401231 96.155487
##         max neval
##    3.636923   100
##    9.460923   100
##    3.117949   100
```

```
##    8.005333   100
##   75.357539   100
##   85.096205   100
##   99.881436   100
##  251.930667   100
```

As above comparsion, the running time reduced signifcantly when choose change to use order2 funciton in PIKK and reduce the for loop time in FYKD. From above graph, we can conclude that Whern the population or sample size increase, the running time will also increase.