# Stats243 Problem Set 4

Mengling Liu

Oct 2017

Collaboration: I disscussed with Rui Chen and Fan Dong for this homework set.

1. a) 1 copy of the vector 1:10 is being made, because variable x in the global environment got passed into the local variable data inside of the function and stored. Data and x both point to the same address. No additional copy was being made.

b). If save 1 to 10000000 to an object x, then the object size is around 4000040 bytes. So serialize outputs an object size that is twice as much as the actual one. So when we call myFun, it calls on g function and g function will use the data variable as imported from input. No copies were made so serilize function was fooled to count into one additional copy.

```
x <- 1:10^6
f <- function(input){
data <- input
g <- function(param) return(param * data)
return(g)
}
myFun <- f(x)
# rm(x)
# data <- 100
# myFun(3)

length(serialize(myFun, NULL))

## [1] 8007090

# [1] 8000772

object.size(x)

## 4000040 bytes

# 4000040 bytes
```

(c) When we call myFun(3), we call function g and pass 3 into the param variable. When g function searches for data variable, it first does it in the local environement of g and then f, but it could not find then it goes to the global environment. X got assigned 1:10 as a vector but removed with rm(x), so the function is not able to locate x. If we remove rm(x), then the function is able to find x and therefore output results. If we add back data ¡- input, then x will be passed into data and stored inside of the function as closure which will be carried along with the function, so even if we remove the varible x in the global environment, we could still locate x in the function saved in data variable.

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

```
rm(x)
data <- 100
myFun(3)

## Error in myFun(3):  object 'x' not found
```

(d)

```
# add force(date) into the function
# force function forces the evaluation of a formal
# argument. This can be useful if the argument will
# be captured in a closure by the lexical scoping rule.
x <- 1:10
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
```

2 a) R makes change in place to an element of vectors/list without making copies.

```
# Pls refer to the comment part, results generated from terminal
# are copied here
listofvec <- list(c(1,2,3),c(4,5,6),c(7,8,9))
object.size(listofvec)

## 288 bytes

.Internal(inspect(listofvec))

## @0x0000000016d19da8 19 VECSXP g0c3 [NAM(2)] (len=3, tl=0)
##   @0x0000000016d1c320 14 REALSXP g0c3 [] (len=3, tl=0) 1,2,3
##   @0x0000000016d1c368 14 REALSXP g0c3 [] (len=3, tl=0) 4,5,6
##   @0x0000000016d19c40 14 REALSXP g0c3 [] (len=3, tl=0) 7,8,9

listofvec[3][2]<-10

## Warning in listofvec[3][2] <- 10:  number of items to replace is not a multiple of replacement
length

object.size(listofvec)

## 288 bytes

.Internal(inspect(listofvec))

## @0x0000000017173930 19 VECSXP g0c3 [NAM(2)] (len=3, tl=0)
##   @0x0000000016d1c320 14 REALSXP g0c3 [NAM(2)] (len=3, tl=0) 1,2,3
##   @0x0000000016d1c368 14 REALSXP g0c3 [NAM(2)] (len=3, tl=0) 4,5,6
##   @0x0000000016d19c40 14 REALSXP g0c3 [NAM(2)] (len=3, tl=0) 7,8,9
```

```
#Results:
# > object.size(listofvec)
# 288 bytes
# > .Internal(inspect(listofvec))
# @0x00000000126667c0 19 VECSXP g0c3 [NAM(2)] (len=3, tl=0)
# @0x0000000012666898 14 REALSXP g0c3 [] (len=3, tl=0) 1,2,3
# @0x0000000012666850 14 REALSXP g0c3 [] (len=3, tl=0) 4,5,6
# @0x0000000012666808 14 REALSXP g0c3 [] (len=3, tl=0) 7,8,9
```

b) When make a copy of the list, there is no copy-on-change going on because both lists point to the same address. When the change is made to one of the vectors in one of the list, then a copy is being made as both lists point to the same address. Need to create a new address to store the changed list. If there is a change being made to one of the vectors in the list, a copy of the list was being made and a copy of the relevant vector is being made. Only the address of the list and relevant vector under that list changed.

```
listofvec <- list(c(1,2,3),c(4,5,6),c(7,8,9))
object.size(listofvec)

## 288 bytes

.Internal(inspect(listofvec))

## @0x0000000017552130 19 VECSXP g0c3 [NAM(2)] (len=3, tl=0)
##   @0x0000000017552208 14 REALSXP g0c3 [] (len=3, tl=0) 1,2,3
##   @0x00000000175521c0 14 REALSXP g0c3 [] (len=3, tl=0) 4,5,6
##   @0x0000000017552178 14 REALSXP g0c3 [] (len=3, tl=0) 7,8,9

listofvec2 <- listofvec
object.size(listofvec2)

## 288 bytes

.Internal(inspect(listofvec2))

## @0x0000000017552130 19 VECSXP g0c3 [NAM(2)] (len=3, tl=0)
##   @0x0000000017552208 14 REALSXP g0c3 [] (len=3, tl=0) 1,2,3
##   @0x00000000175521c0 14 REALSXP g0c3 [] (len=3, tl=0) 4,5,6
##   @0x0000000017552178 14 REALSXP g0c3 [] (len=3, tl=0) 7,8,9

#Results:
# > object.size(listofvec2)
# 288 bytes
# > .Internal(inspect(listofvec2))
# @0x00000000186975f0 19 VECSXP g0c3 [NAM(2)] (len=3, tl=0)
#   @0x00000000186976c8 14 REALSXP g0c3 [] (len=3, tl=0) 1,2,3
#   @0x0000000018697680 14 REALSXP g0c3 [] (len=3, tl=0) 4,5,6
#   @0x0000000018697638 14 REALSXP g0c3 [] (len=3, tl=0) 7,8,9

listofvec <- list(c(1,2,3),c(4,5,6),c(7,8,9))
listofvec2 <- listofvec
object.size(listofvec)

## 288 bytes

.Internal(inspect(listofvec))

## @0x0000000017759e58 19 VECSXP g0c3 [NAM(2)] (len=3, tl=0)
##   @0x0000000017745b68 14 REALSXP g0c3 [] (len=3, tl=0) 1,2,3
##   @0x0000000017745b20 14 REALSXP g0c3 [] (len=3, tl=0) 4,5,6
##   @0x0000000017745ad8 14 REALSXP g0c3 [] (len=3, tl=0) 7,8,9
```

```
object.size(listofvec2)

## 288 bytes

.Internal(inspect(listofvec2))

## @0x0000000017759e58 19 VECSXP g0c3 [NAM(2)] (len=3, tl=0)
##   @0x0000000017745b68 14 REALSXP g0c3 [] (len=3, tl=0) 1,2,3
##   @0x0000000017745b20 14 REALSXP g0c3 [] (len=3, tl=0) 4,5,6
##   @0x0000000017745ad8 14 REALSXP g0c3 [] (len=3, tl=0) 7,8,9

listofvec[2][1] <- 10
object.size(listofvec)

## 264 bytes

.Internal(inspect(listofvec))

## @0x0000000017a82498 19 VECSXP g0c3 [NAM(2)] (len=3, tl=0)
##   @0x0000000017745b68 14 REALSXP g0c3 [NAM(2)] (len=3, tl=0) 1,2,3
##   @0x0000000017746c70 14 REALSXP g0c1 [] (len=1, tl=0) 10
##   @0x0000000017745ad8 14 REALSXP g0c3 [NAM(2)] (len=3, tl=0) 7,8,9

#Results:
# Result after changing listvec: a copy of the list was made
# and a copy of vector 2 under the list was made

#  object.size(listofvec)
# 288 bytes
# > .Internal(inspect(listofvec))
# @0x00000000186d0600 19 VECSXP g0c3 [NAM(2)] (len=3, tl=0)
#   @0x00000000186d06d8 14 REALSXP g0c3 [] (len=3, tl=0) 1,2,3
#   @0x00000000186d0690 14 REALSXP g0c3 [] (len=3, tl=0) 4,5,6
#   @0x00000000186d0648 14 REALSXP g0c3 [] (len=3, tl=0) 7,8,9
# > object.size(listofvec2)
# 288 bytes
# > .Internal(inspect(listofvec2))
# @0x00000000186d0600 19 VECSXP g0c3 [NAM(2)] (len=3, tl=0)
#   @0x00000000186d06d8 14 REALSXP g0c3 [] (len=3, tl=0) 1,2,3
#   @0x00000000186d0690 14 REALSXP g0c3 [] (len=3, tl=0) 4,5,6
#   @0x00000000186d0648 14 REALSXP g0c3 [] (len=3, tl=0) 7,8,9
# > listofvec[2][1] <- 10
# > object.size(listofvec)
# 264 bytes
# > .Internal(inspect(listofvec))
# @0x0000000017c7e088 19 VECSXP g0c3 [NAM(2)] (len=3, tl=0)
#   @0x00000000186d06d8 14 REALSXP g0c3 [NAM(2)] (len=3, tl=0) 1,2,3
#   @0x0000000017ddf1e0 14 REALSXP g0c1 [] (len=1, tl=0) 10
#   @0x00000000186d0648 14 REALSXP g0c3 [NAM(2)] (len=3, tl=0) 7,8,9
```

c) Now make a list of lists. Copy and list and add an element to the second list. So the address for the entire copied list changed as a copy is being made for the entire list object, but not for the unchanged sublist. The unchanged sublist under the new list still point to the original address. New address will be created for newly added element under the list.

```
listoflist <- list(list(c(1,2),c(3,4)),list(c(5,6),c(7,8)),list(c(9,10),c(11,12)))
object.size(listoflist)
```

```
## 576 bytes

.Internal(inspect(listoflist))

## @0x0000000017495018 19 VECSXP g0c3 [NAM(2)] (len=3, tl=0)
##   @0x00000000166d8358 19 VECSXP g0c2 [] (len=2, tl=0)
##     @0x00000000166d8278 14 REALSXP g0c2 [] (len=2, tl=0) 1,2
##     @0x00000000166d82e8 14 REALSXP g0c2 [] (len=2, tl=0) 3,4
##   @0x00000000166d8518 19 VECSXP g0c2 [] (len=2, tl=0)
##     @0x00000000166d83c8 14 REALSXP g0c2 [] (len=2, tl=0) 5,6
##     @0x00000000166d84a8 14 REALSXP g0c2 [] (len=2, tl=0) 7,8
##   @0x00000000166d8668 19 VECSXP g0c2 [] (len=2, tl=0)
##     @0x00000000166d8588 14 REALSXP g0c2 [] (len=2, tl=0) 9,10
##     @0x00000000166d85f8 14 REALSXP g0c2 [] (len=2, tl=0) 11,12

listoflist2 <- listoflist
object.size(listoflist2)

## 576 bytes

.Internal(inspect(listoflist2))

## @0x0000000017495018 19 VECSXP g0c3 [NAM(2)] (len=3, tl=0)
##   @0x00000000166d8358 19 VECSXP g0c2 [] (len=2, tl=0)
##     @0x00000000166d8278 14 REALSXP g0c2 [] (len=2, tl=0) 1,2
##     @0x00000000166d82e8 14 REALSXP g0c2 [] (len=2, tl=0) 3,4
##   @0x00000000166d8518 19 VECSXP g0c2 [] (len=2, tl=0)
##     @0x00000000166d83c8 14 REALSXP g0c2 [] (len=2, tl=0) 5,6
##     @0x00000000166d84a8 14 REALSXP g0c2 [] (len=2, tl=0) 7,8
##   @0x00000000166d8668 19 VECSXP g0c2 [] (len=2, tl=0)
##     @0x00000000166d8588 14 REALSXP g0c2 [] (len=2, tl=0) 9,10
##     @0x00000000166d85f8 14 REALSXP g0c2 [] (len=2, tl=0) 11,12

#Results:
#  object.size(listoflist2)
# 576 bytes
# > .Internal(inspect(listoflist2))
# @0x0000000018687e80 19 VECSXP g0c3 [NAM(2)] (len=3, tl=0)
#   @0x0000000017268308 19 VECSXP g0c2 [] (len=2, tl=0)
#     @0x00000000172684c8 14 REALSXP g0c2 [] (len=2, tl=0) 1,2
#     @0x0000000017268378 14 REALSXP g0c2 [] (len=2, tl=0) 3,4
#   @0x0000000017037470 19 VECSXP g0c2 [] (len=2, tl=0)
#     @0x0000000017268298 14 REALSXP g0c2 [] (len=2, tl=0) 5,6
#     @0x00000000170374e0 14 REALSXP g0c2 [] (len=2, tl=0) 7,8
#   @0x0000000017037320 19 VECSXP g0c2 [] (len=2, tl=0)
#     @0x0000000017037400 14 REALSXP g0c2 [] (len=2, tl=0) 9,10
#     @0x0000000017037390 14 REALSXP g0c2 [] (len=2, tl=0) 11,12

listoflist2[[4]]<-list(c(13,14),c(15,16))
object.size(listoflist2)

## 744 bytes

.Internal(inspect(listoflist2))

## @0x00000000176d4f78 19 VECSXP g0c3 [NAM(2)] (len=4, tl=0)
##   @0x00000000166d8358 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##     @0x00000000166d8278 14 REALSXP g0c2 [] (len=2, tl=0) 1,2
```

```
##     @0x00000000166d82e8 14 REALSXP g0c2 [] (len=2, tl=0) 3,4
##   @0x00000000166d8518 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##     @0x00000000166d83c8 14 REALSXP g0c2 [] (len=2, tl=0) 5,6
##     @0x00000000166d84a8 14 REALSXP g0c2 [] (len=2, tl=0) 7,8
##   @0x00000000166d8668 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##     @0x00000000166d8588 14 REALSXP g0c2 [] (len=2, tl=0) 9,10
##     @0x00000000166d85f8 14 REALSXP g0c2 [] (len=2, tl=0) 11,12
##   @0x0000000014d3aad8 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##     @0x0000000014d3bee0 14 REALSXP g0c2 [] (len=2, tl=0) 13,14
##     @0x0000000014d3c030 14 REALSXP g0c2 [] (len=2, tl=0) 15,16

#Results:
# > object.size(listoflist2)
# 744 bytes
# > .Internal(inspect(listoflist2))
# @0x00000000186c0f20 19 VECSXP g0c3 [NAM(2)] (len=4, tl=0)
#   @0x0000000017268308 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
#     @0x00000000172684c8 14 REALSXP g0c2 [] (len=2, tl=0) 1,2
#     @0x0000000017268378 14 REALSXP g0c2 [] (len=2, tl=0) 3,4
#   @0x0000000017037470 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
#     @0x0000000017268298 14 REALSXP g0c2 [] (len=2, tl=0) 5,6
#     @0x00000000170374e0 14 REALSXP g0c2 [] (len=2, tl=0) 7,8
#   @0x0000000017037320 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
#     @0x0000000017037400 14 REALSXP g0c2 [] (len=2, tl=0) 9,10
#     @0x0000000017037390 14 REALSXP g0c2 [] (len=2, tl=0) 11,12
#   @0x0000000016d93d48 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
#     @0x0000000016d93e60 14 REALSXP g0c2 [] (len=2, tl=0) 13,14
#     @0x0000000016d93db8 14 REALSXP g0c2 [] (len=2, tl=0) 15,16
```

d) Object.size outputs a result (160Mb) that is twice as much as the actual number (80Mb). x has been stored twice into tmp[[1]] and tmp[[2]], but the two objects points to the same address, so there is only one copy being made. In this case, the actual size is 80Mb and object.size function was fooled to believe that two copies have been generated so output a result that's double the size of the regular amount.

```
gc()

##          used (Mb) gc trigger (Mb) max used (Mb)
## Ncells 412695 22.1     750400 40.1   750400 40.1
## Vcells 811322  6.2    2983612 22.8  2516436 19.2

#Results:
#          used (Mb) gc trigger (Mb) max used (Mb)
# Ncells 245090 13.1     460000 24.6   350000 18.7
# Vcells 481038  3.7    1023718  7.9   786411  6.0

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

## @0x00000000173c1c90 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
##   @0x00007ff5faca0010 14 REALSXP g0c7 [NAM(2)] (len=10000000, tl=0) 0.216744,1.14689,-1.0229,1.10026,0.
##   @0x00007ff5faca0010 14 REALSXP g0c7 [NAM(2)] (len=10000000, tl=0) 0.216744,1.14689,-1.0229,1.10026,0.

#Results:
#  .Internal(inspect(tmp))
```

```
# @0x0000000017ff6280 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
#   @0x00007ff5fadd0010 14 REALSXP g0c7 [NAM(2)] (len=10000000, tl=0) 0.0411239,0.539126,-1.14483,-0.12411
#   @0x00007ff5fadd0010 14 REALSXP g0c7 [NAM(2)] (len=10000000, tl=0) 0.0411239,0.539126,-1.14483,-0.12411

object.size(tmp)

## 160000136 bytes

#Results:
# 160000136 bytes (= 160Mb)

gc()

##            used (Mb) gc trigger  (Mb) max used (Mb)
## Ncells   412882 22.1     750400  40.1   750400 40.1
## Vcells 10811855 82.5   15970913 121.9 10863930 82.9

#Results:
#            used (Mb) gc trigger  (Mb) max used (Mb)
# Ncells   245922 13.2     460000  24.6   350000 18.7
# Vcells 10483096 80.0   15495182 118.3 10490732 80.1


# x <- rnorm(1e7)
# object.size(x)
# 80000040 bytes (=80Mb)
```

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)
}
oldoneUpdate <- 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)
```

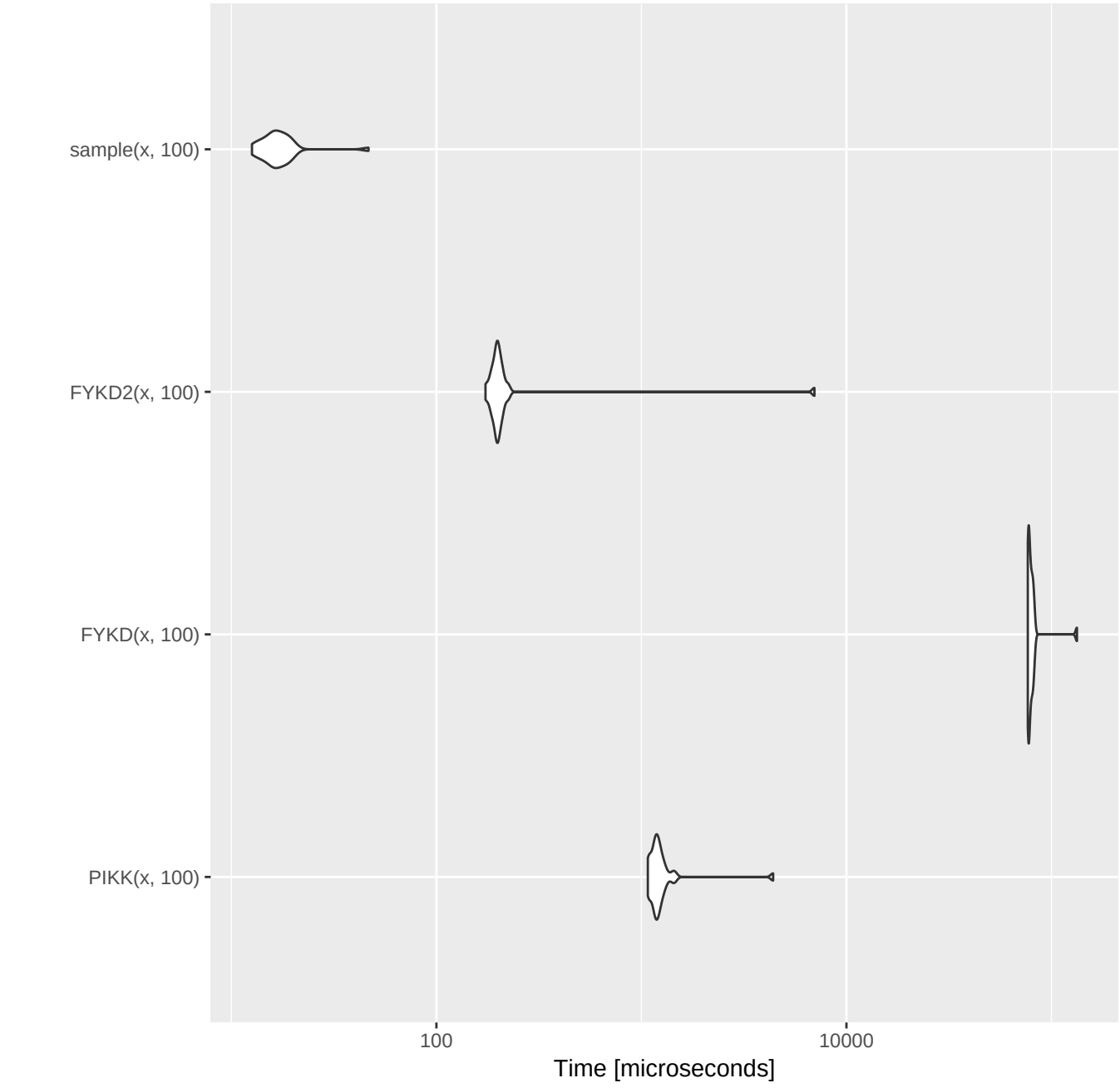
```
converge.check <- abs(L.new - L.old) < thresh
theta.new <- theta.new/rowSums(theta.new)
return(list(theta = theta.new, loglik = L.new,
converged = converge.check))
}
# # initialize the parameters at random starting values
# temp <- matrix(runif(n*K), n, K)
# theta.init <- temp/rowSums(temp)
# # do single update
# oldoutput <- oldoneUpdate(A, n, K, theta.init)

##-------------------------------------------------
## Turn the three for-loops into matrix mulplication

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)
}
newoneUpdate <- 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))

# theta.old[,z] multiply by transpose of theta.old[,z] for zth column
# will return a matrix containing multiplicaitons of each two numbers
# in that column, and then loop over all the columns
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) {
    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
# newoutput <- oneUpdate(A, n, K, theta.init)


##-------------------------------------------------------------
#Use benchmark to test out the time spent on runing each code
library(rbenchmark)
temp <- matrix(runif(n*K), n, K)
theta.init <- temp/rowSums(temp)
```

```
oldoutput <- oldoneUpdate(A, n, K, theta.init)
newoutput <- newoneUpdate(A, n, K, theta.init)
identical(oldoutput, newoutput)

## [1] TRUE

benchmark(oldoneUpdate(A, n, K, theta.init),
          newoneUpdate(A, n, K, theta.init), replications = 10,
          columns = c('test','elapsed','replications'))

##                                test elapsed replications
## 2 newoneUpdate(A, n, K, theta.init)   13.30           10
## 1 oldoneUpdate(A, n, K, theta.init)   97.89           10

# Results:
#                                test elapsed replications
# 2 newoneUpdate(A, n, K, theta.init)   14.36           10
# 1 oldoneUpdate(A, n, K, theta.init)  102.83           10
```

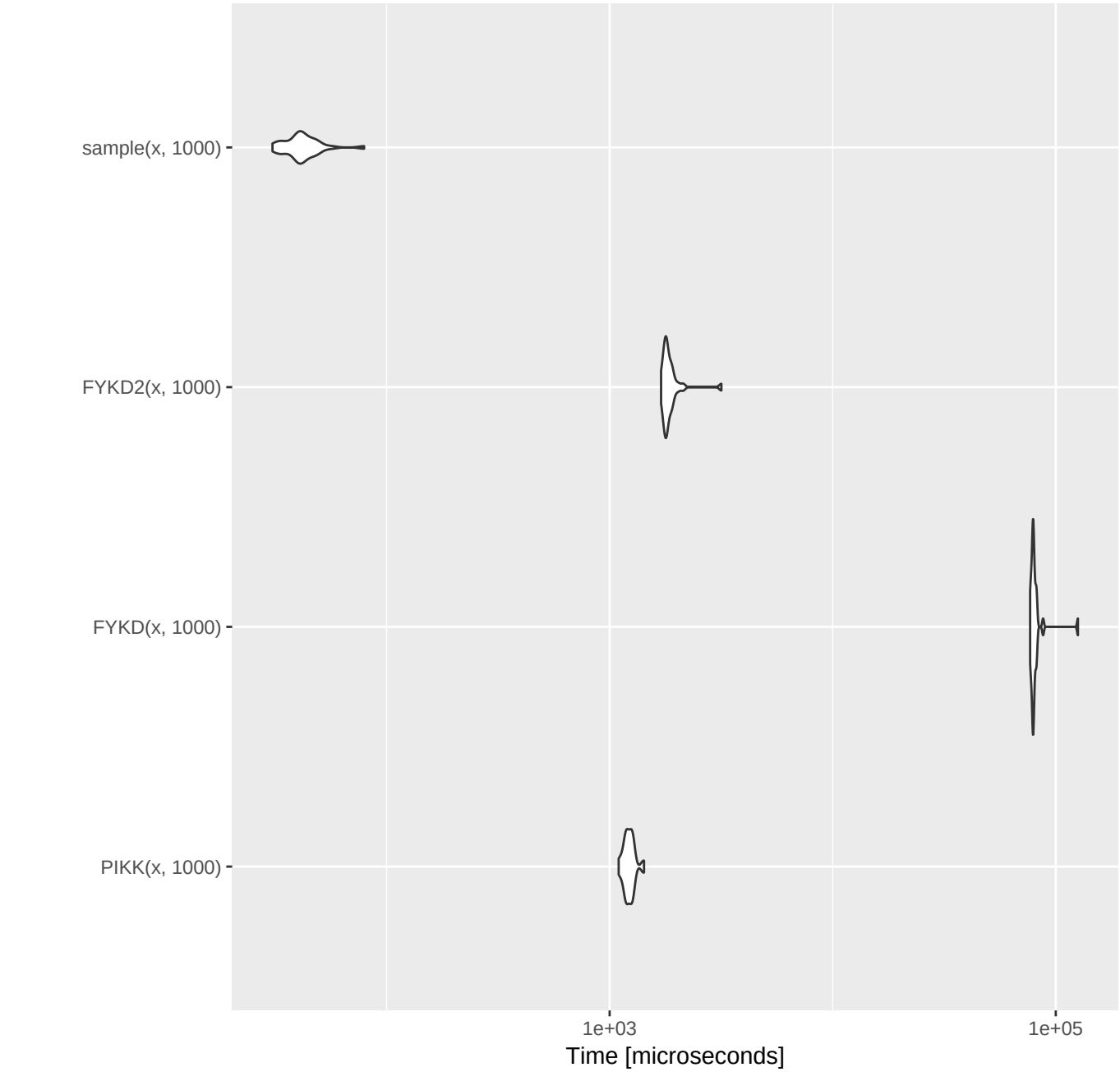
4. a)

```
library(microbenchmark)

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

library(ggplot2)

## Warning:  package 'ggplot2' was built under R version 3.4.2

x<- 1:10000

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])
}

##-----------------------------------------------------
# Edits to the second algorithm
# change 1:n to 1:k as we only outpout the first k numbers,
# so only need to shuffle the first k numbers
FYKD2 <- function(x, k) {
n <- length(x)
for(i in 1:k) {
  j <- ceiling(runif(1, min=0, max=n))
  tmp <- x[i]
  x[i] <- x[j]
  x[j] <- tmp
}
```

```
return(x[1:k])
}

timedata1<-microbenchmark(PIKK(x,100), FYKD(x,100), FYKD2(x,100),sample(x,100),times=30L)
# Unit: microseconds
#            expr       min        lq        mean    median         uq        max neval
#    PIKK(x, 100)  1082.821  1163.298  1328.77621  1205.285  1284.3625   4383.532   100
#    FYKD(x, 100) 77972.379 82126.378 91270.01644 85200.122 92833.9800 170761.339   100
#   FYKD2(x, 100)   163.753   177.283   285.97052   188.713   201.5420   6457.733   100
#  sample(x, 100)    11.197    13.996    18.07858    16.096    20.0615     56.917   100
plot1 <- autoplot(timedata1)
plot1
```

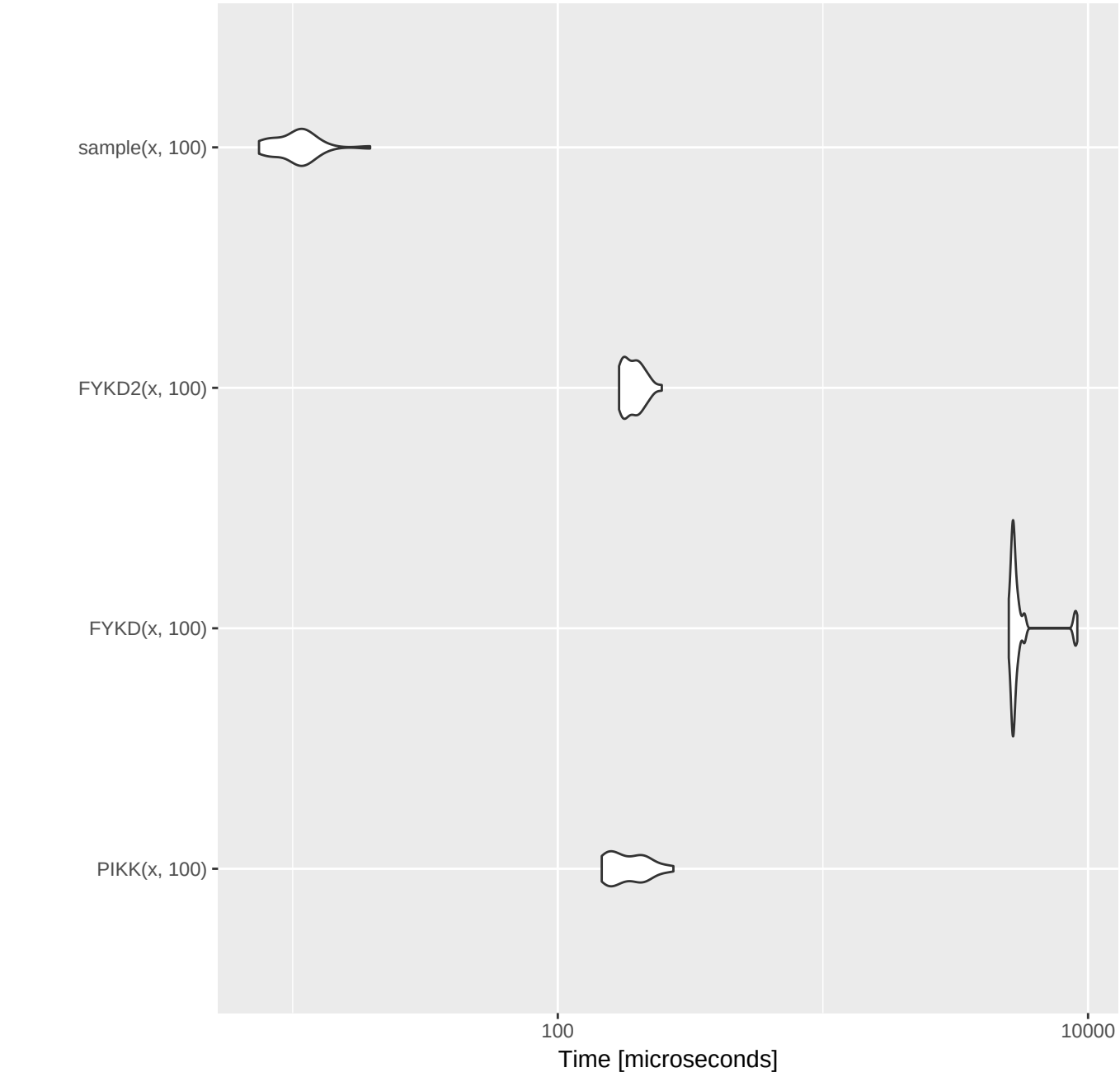


```r
timedata2<-microbenchmark(PIKK(x,1000), FYKD(x,1000), FYKD2(x,1000),sample(x,1000),times=30L)
# Unit: microseconds
#              expr       min         lq        mean    median         uq        max neval
#    PIKK(x, 1000)  1089.819  1170.0620   1284.8149  1218.815   1275.032   2725.946   100
#    FYKD(x, 1000) 88534.657 92886.2315 104430.8062 95046.739 101243.681 172968.966   100
#   FYKD2(x, 1000)  1637.527  1680.2150   1975.6831  1718.937   1815.742   3641.748   100
#  sample(x, 1000)    29.859    34.0575     56.1756    40.589     43.155   1631.928   100
plot2 <- autoplot(timedata2)
plot2
```



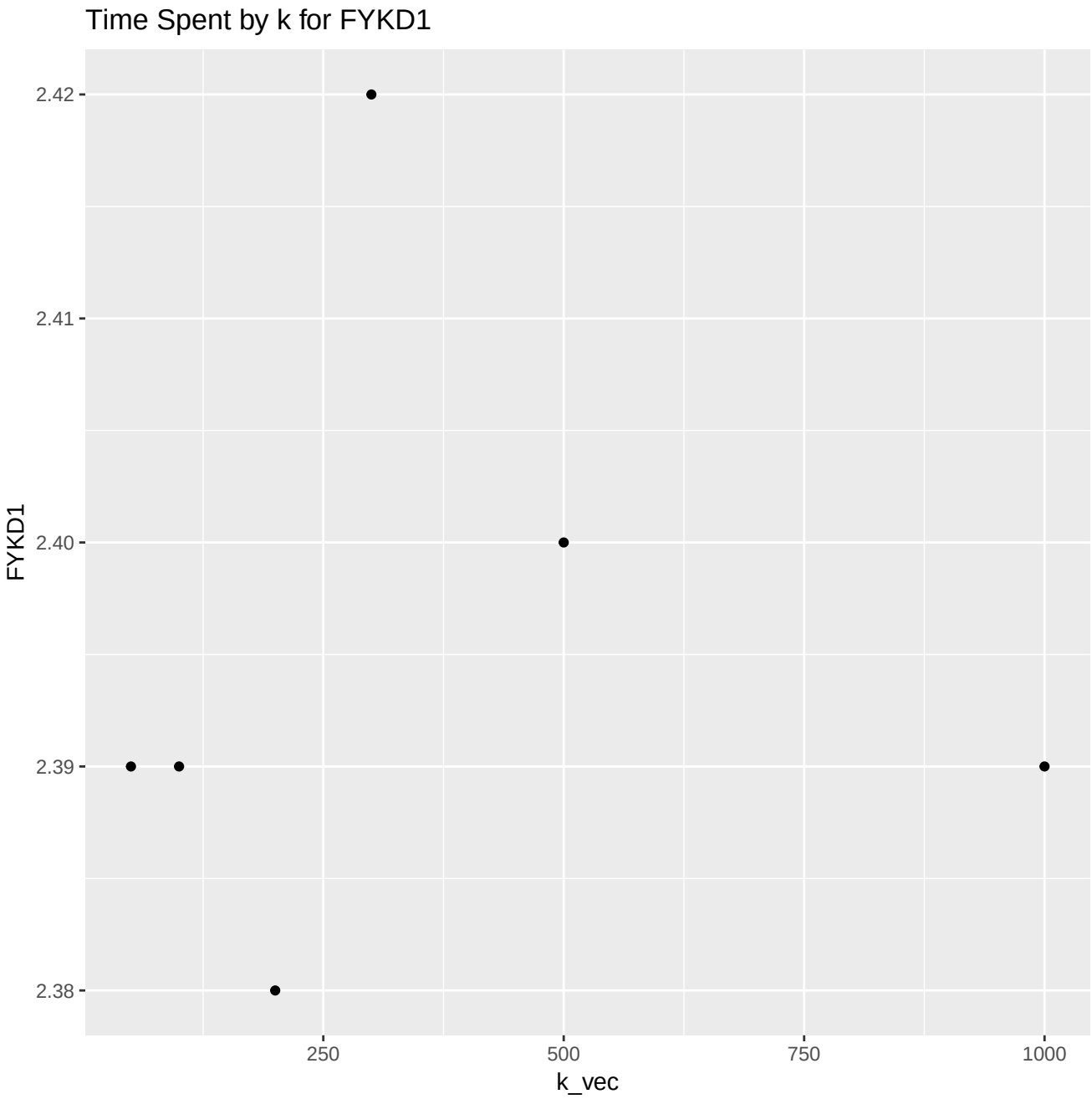

```r
x<-1:1000
timedata3<-microbenchmark(PIKK(x,100), FYKD(x,100), FYKD2(x,100),sample(x,100),times=30L)
# Unit: microseconds
#             expr      min        lq       mean    median        uq      max neval
```

```
#    PIKK(x, 100)  140.427  161.8870  213.74198  198.9765  221.1365 2009.353   100
#    FYKD(x, 100) 4970.429 5164.0400 5962.25712 5313.0965 7027.3680 9318.971   100
#   FYKD2(x, 100)  160.021  170.7515  187.44371  182.4140  193.3780  361.563   100
#  sample(x, 100)    6.998    8.6310   10.85199   10.2645   11.6640   30.791   100
plot3 <- autoplot(timedata3)
plot3
```

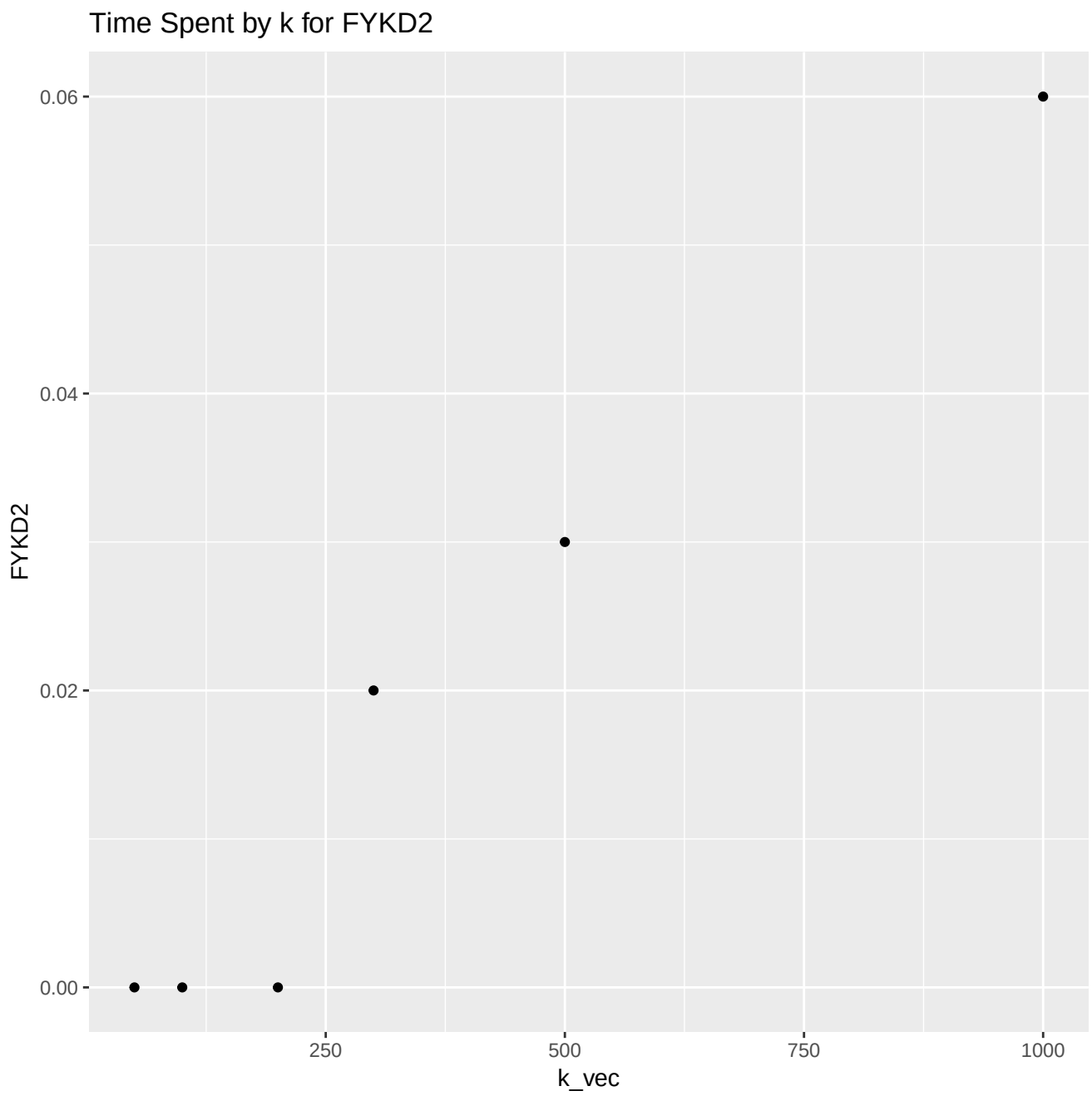



```
x<-1:10000
k_vec <- c(50,100,200,300,500,1000)
timeplotk <-sapply(k_vec,function(k){benchmark(FYKD(x,k),FYKD2(x,k),replications = 30L)$elapsed})
timeplot1<-data.frame(t(timeplotk))
names(timeplot1)[1] <- paste("FYKD1")
names(timeplot1)[2] <- paste("FYKD2")
timeplot1 <- cbind(k_vec,timeplot1)
```

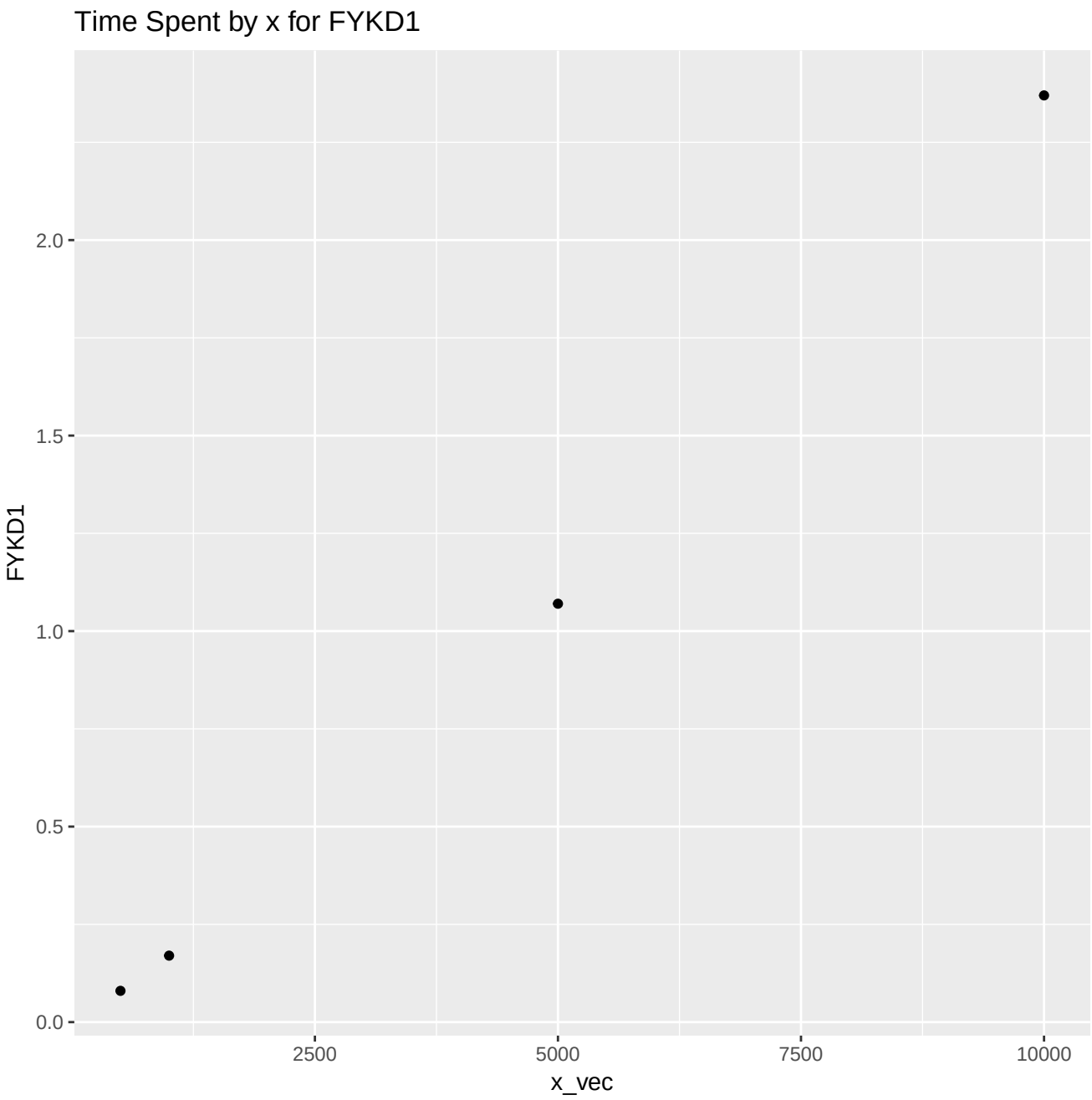
```
ggplot(timeplot1, aes(x=k_vec, y=FYKD1))+geom_point()+ggtitle("Time Spent by k for FYKD1")
```



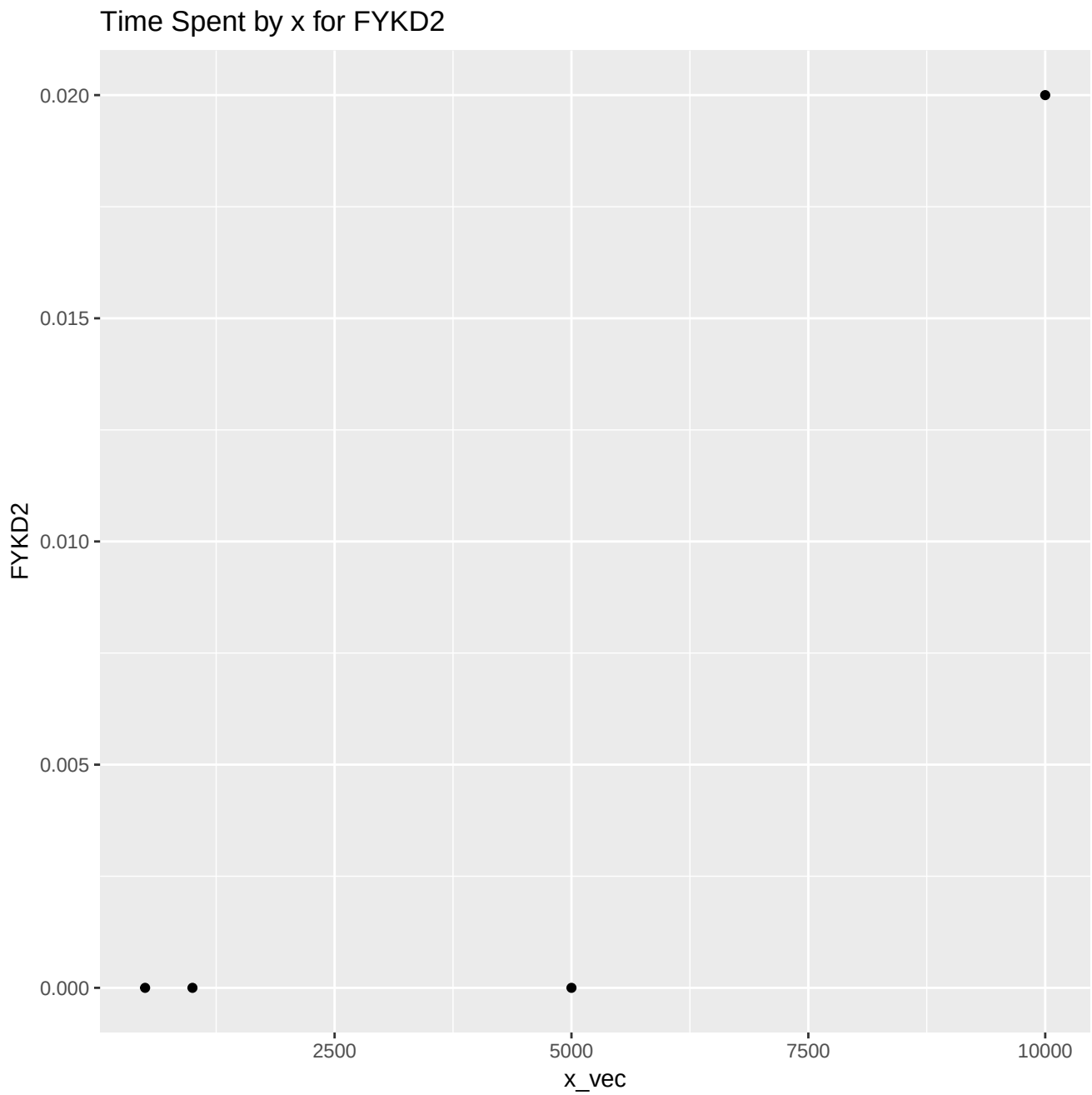

```
ggplot(timeplot1, aes(x=k_vec, y=FYKD2))+geom_point()+ggtitle("Time Spent by k for FYKD2")
```




```
k <- 100
x_vec <- c(500,1000,5000,10000)
timeplotx <- sapply(x_vec, function(x){benchmark(FYKD(1:x,k),FYKD2(1:x,k), replications=30L)$elapsed})
timeplot2<-data.frame(t(timeplotx))
names(timeplot2)[1] <- paste("FYKD1")
names(timeplot2)[2] <- paste("FYKD2")
timeplot2 <- cbind(x_vec,timeplot2)
ggplot(timeplot2, aes(x=x_vec, y=FYKD1))+geom_point()+ggtitle("Time Spent by x for FYKD1")
```




```
ggplot(timeplot2, aes(x=x_vec, y=FYKD2))+geom_point()+ggtitle("Time Spent by x for FYKD2")
```




b) Write a new method to generate a random sample of size k

```
# randomly sample k numbers with uniform distribution
# then take these k numbers as index to extract the actual number
# from the vector x. The output would be randomly generated
# number as well
method <- function(x, k) {
n <- length(x)
for(i in 1:k) {
  j <- ceiling(runif(i, min=0, max=n))
}
return(x[j])
}

x <- 1:1000
```

```r
k <- 100
method(x,k)
```

```
##   [1] 189 238 137 948 185 399 898 424 676  43 232 204 584 144 440 487  62
##  [18] 297 169 113 138 870 977 936 534 411 794 239 619 550 851 517  57 530
##  [35] 391 123 119 604 329 156 981 574 132 270 142 331 478 463 388 766 825
##  [52] 354 576 570 579 566 569 968 289 464 491 661 807 616 115 601 618 321
##  [69] 589 868 858 452 228 308 235  81 654  18 147  78 815 224 397 882 743
##  [86] 402 383 587 357 690 840  38  89 149 158 711 489 329 315 309
```