1 Problem 1

1.1 Problem a

I expect the maximum number of copies to be 1, i.e. x itself.

1.2 Problem b

The size of the serialized object is twice what I would expect. I think it is the case where R has been fooled to sum up the size of "x" and "data" while there is actually only one copy. The conclusion can be verified by the fact that gc() shows only about 80MB has been used.

```
object.size(x)
##80000040 bytes

length(serialize(myFun,NULL))
##160000778

gc()
##           used (Mb) gc trigger  (Mb) max used  (Mb)
##Ncells   222394 11.9     467875  25.0   350000  18.7
##Vcells 10487566 80.1   32243889 246.1 30487660 232.7
```

1.3 Problem c

When evaluating g, we need "data", which has not been defined in g. So R searches in g's enclosing environment function f, where data has not been defined either because of lazy evaluation.

1.4 Problem d

I used "with" function to make the code work. And the resulting serialized closure is about the same size as x.

```
x <- rnorm(1e7)
myFun2 <- with(list(data = x),function(param) return(param * data))
invisible(myFun2(3))

##object.size(x)
##80000040 bytes

##length(serialize(myFun2,NULL))
##[1] 80000521
```

```
##gc()
##           used  (Mb) gc trigger  (Mb) max used  (Mb)
##Ncells   219902  11.8     467875  25.0   350000  18.7
##Vcells 20458209 156.1   22891870 174.7 20458203 156.1
```

2 Problem 2

2.1 Problem a

When we make a copy of myList, the use of memeo R can make the change in place without creating a new list. Instead, a new vector will be created. This can be verified by both the used memory and the address of the two vectors in the list. After we modify an element of the first vector, the used memory has increased by about 9MB, roughly the size of a vector. And only the address of the first vecotr changed.

```
myList<-list(a <- rnorm(1e6),b <- rnorm(1e6))

gc()
##           used (Mb) gc trigger (Mb) max used (Mb)
##Ncells  219370 11.8     467875 25.0   350000 18.7
##Vcells 2450774 18.7    3036483 23.2  2451893 18.8

.Internal(inspect(myList[[1]]))
##@0x000007ffff580010 14 REALSXP g1c7 [MARK,NAM(2)] (len=1000000, tl=0) -0.901841,0.480835,2
.Internal(inspect(myList[[2]]))
##@0x000007fffedd0010 14 REALSXP g0c7 [NAM(2)] (len=1000000, tl=0) -0.36836,0.0902418,-0.41


myList[[1]][[1]]<-1
gc()
##          used (Mb) gc trigger (Mb) max used (Mb)
##Ncells  219514 11.8     467875 25.0   350000 18.7
##Vcells 3453439 26.4    4143970 31.7  3453463 26.4


.Internal(inspect(myList[[1]]))
##@0x000007fffe620010 14 REALSXP g1c7 [MARK,NAM(1)] (len=1000000, tl=0) 1,0.480835,2.30725,
.Internal(inspect(myList[[2]]))
##@0x000007fffedd0010 14 REALSXP g1c7 [MARK,NAM(2)] (len=1000000, tl=0) -0.36836,0.0902418,
```

2.2 Problem b

As we can see from the memeory use situation below, when we change the first vector in myList2. Only 8MB more memory has been used, i.e. R doesn't make a copy of the whole list, but the relevant vector. The address of the vectors also verify this.

```
myList<-list(a <- rnorm(1e6),b <- rnorm(1e6))
gc()
##          used (Mb) gc trigger (Mb) max used (Mb)
##Ncells  218973 11.7     467875 25.0   350000 18.7
##Vcells 2444274 18.7    3031456 23.2  2445379 18.7




myList2 <- myList
gc()
##          used (Mb) gc trigger (Mb) max used (Mb)
##Ncells  220031 11.8     467875 25.0   350000 18.7
##Vcells 2461452 18.8    3263028 24.9  2461567 18.8

.Internal(inspect(myList))
##@0x00000000141fd3a8 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##  @0x000007ffff580010 14 REALSXP g1c7 [MARK,NAM(2)] (len=1000000, tl=0) -0.901841,0.480835
##  @0x000007fffedd0010 14 REALSXP g0c7 [NAM(2)] (len=1000000, tl=0) -0.36836,0.0902418,-0.
.Internal(inspect(myList2))
##@0x00000000141fd3a8 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##  @0x000007ffff580010 14 REALSXP g1c7 [MARK,NAM(2)] (len=1000000, tl=0) -0.901841,0.480835
##  @0x000007fffedd0010 14 REALSXP g0c7 [NAM(2)] (len=1000000, tl=0) -0.36836,0.0902418,-0.





myList2[[1]] <- rnorm(1e6)
gc()
##          used (Mb) gc trigger (Mb) max used (Mb)
##Ncells  221361 11.9     467875 25.0   350000 18.7
##Vcells 3484329 26.6    4178169 31.9  3484666 26.6

.Internal(inspect(myList2))
##@0x0000000005eab110 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
##  @0x000007fffe620010 14 REALSXP g0c7 [NAM(1)] (len=1000000, tl=0) 0.0048219,-1.59664,-1.5
##  @0x000007fffedd0010 14 REALSXP g1c7 [MARK,NAM(2)] (len=1000000, tl=0) -0.36836,0.0902418
```

2.3 Problem c

When we first make the copy, R doesn't create a new list. And when we add an element to the second list, no copy of the original list or vectors has been made. The only difference is that the newly added element takes more memory use.

```
myList<-list(a <- rnorm(1e6),b <- rnorm(1e6))
myList2<-list(c <- rnorm(1e6),d <- rnorm(1e6))
gc()
##          used (Mb) gc trigger (Mb) max used (Mb)
##Ncells  219678 11.8     467875   25   350000 18.7
##Vcells 4456100 34.0    5239002   40  4457205 34.1

myList3 <- list(myList,myList2)
gc()
##          used (Mb) gc trigger (Mb) max used (Mb)
##Ncells  219794 11.8     467875 25.0   350000 18.7
##Vcells 4457735 34.1    5580952 42.6  4457851 34.1

.Internal(inspect(myList3))
##@0x00000000180bdc58 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
##  @0x00000000180bdb78 19 VECSXP g1c2 [MARK,NAM(2)] (len=2, tl=0)
##    @0x000007ffff590010 14 REALSXP g1c7 [MARK,NAM(2)] (len=1000000, tl=0) -0.901841,0.4808
##    @0x000007fffede0010 14 REALSXP g1c7 [MARK,NAM(2)] (len=1000000, tl=0) -0.36836,0.0902
##  @0x00000000180bdbe8 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##    @0x000007fffe620010 14 REALSXP g1c7 [MARK,NAM(2)] (len=1000000, tl=0) 0.0048219,-1.59
##    @0x000007fffde70010 14 REALSXP g0c7 [NAM(2)] (len=1000000, tl=0) -1.12537,0.379723,-0
>


myList4 <- myList3
gc()
##          used (Mb) gc trigger (Mb) max used (Mb)
##Ncells  219904 11.8     467875 25.0   350000 18.7
##Vcells 4459360 34.1    5939999 45.4  4459475 34.1

.Internal(inspect(myList4))
##@0x00000000180bdc58 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##  @0x00000000180bdb78 19 VECSXP g1c2 [MARK,NAM(2)] (len=2, tl=0)
##    @0x000007ffff590010 14 REALSXP g1c7 [MARK,NAM(2)] (len=1000000, tl=0) -0.901841,0.4808
##    @0x000007fffede0010 14 REALSXP g1c7 [MARK,NAM(2)] (len=1000000, tl=0) -0.36836,0.0902
##  @0x00000000180bdbe8 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##    @0x000007fffe620010 14 REALSXP g1c7 [MARK,NAM(2)] (len=1000000, tl=0) 0.0048219,-1.59
##    @0x000007fffde70010 14 REALSXP g0c7 [NAM(2)] (len=1000000, tl=0) -1.12537,0.379723,-0
```

```
myList4$e <- rnorm(1e6)
gc()
##          used (Mb) gc trigger (Mb) max used (Mb)
##Ncells  220021 11.8     467875 25.0   350000 18.7
##Vcells 5461167 41.7    6316998 48.2  5461600 41.7

.Internal(inspect(myList4))
##@0x000000000064e7e98 19 VECSXP g0c3 [NAM(1),ATT] (len=3, tl=0)
##  @0x00000000180bdb78 19 VECSXP g1c2 [MARK,NAM(2)] (len=2, tl=0)
##    @0x000007ffff590010 14 REALSXP g1c7 [MARK,NAM(2)] (len=1000000, tl=0) -0.901841,0.480
##    @0x000007fffede0010 14 REALSXP g1c7 [MARK,NAM(2)] (len=1000000, tl=0) -0.36836,0.0902
##  @0x00000000180bdbe8 19 VECSXP g1c2 [MARK,NAM(2)] (len=2, tl=0)
##    @0x000007fffe620010 14 REALSXP g1c7 [MARK,NAM(2)] (len=1000000, tl=0) 0.0048219,-1.59
##    @0x000007fffde70010 14 REALSXP g1c7 [MARK,NAM(2)] (len=1000000, tl=0) -1.12537,0.3797
##  @0x000007fffd6c0010 14 REALSXP g0c7 [NAM(1)] (len=1000000, tl=0) -1.55139,0.541609,-0.2
```

2.4 Problem d

object.size(tmp) counts the size of each element in the tmp and then takes sum of them. But tmp[[1]], tmp[[2]] and x are actually the same thing by reference. So the real memory usage is 80MB, i.e. the usage of storing x.

3 Problem 3

In this problem, I refined three parts of the code. First, I used compact vectorized expression to replace the three nested for loop. Second, I simplifed the repeated calculation of the rowsum in the second for loop. Third, in the one Update function, theta.old has been assigned to theta.old1, which has not been used. So I moved the code ”theta.new ¡- theta.old” to the first line of the oneUpdate function. I manage to shorten the elapsed time to 5.79 seconds, compared with 388.75 seconds before.

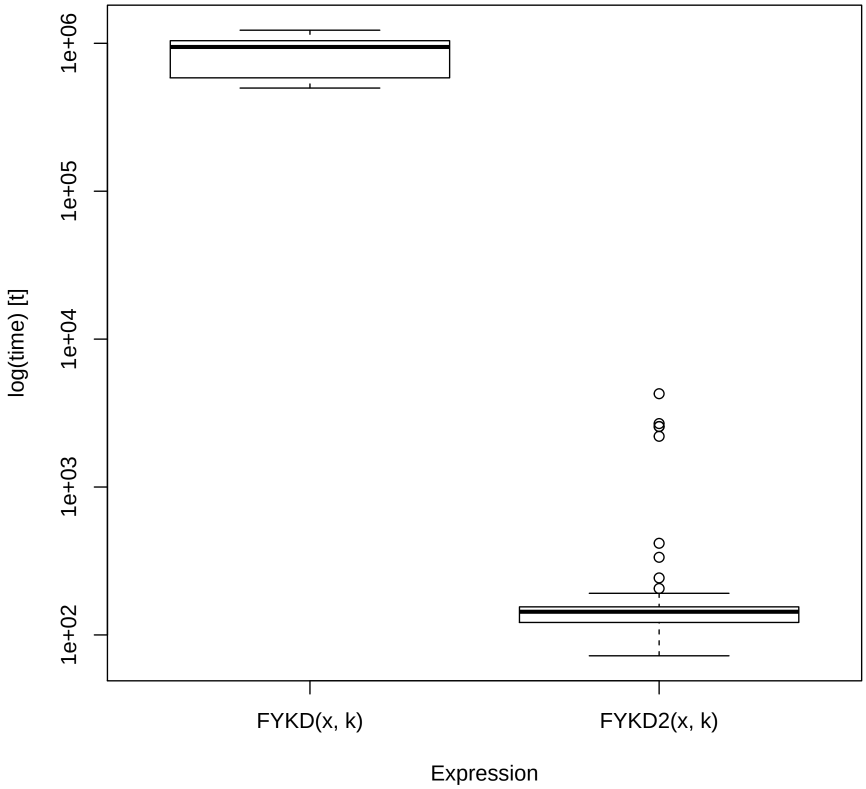
```
setwd("D:/Berkeley//stat243ps/ps4")
load('ps4prob3.Rda') # should have A, n, K
f1 <- function(){ll <- function(Theta, A) {
    sum.ind <- which(A==1, arr.ind=T)
    logLik <- sum(log(Theta[sum.ind])) - sum(Theta)
    return(logLik)
  }
  oneUpdate <- function(A, n, K, theta.old, thresh = 0.1) {
    theta.old1 <- theta.old
    print(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
  set.seed(0)
  temp <- matrix(runif(n*K), n, K)
  theta.init <- temp/rowSums(temp)
  # do single update
  out <- oneUpdate(A, n, K, theta.init)
}


##f2
f2 <- function(){
  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) {
    #the third change, remove theta.old1 <- theta.old, move "theta.new <- theta.old" forwar
    theta.new <- theta.old
    #end of the third change.
    print(theta.old)
    Theta.old <- theta.old %*% t(theta.old)
    L.old <- ll(Theta.old, A)
    q <- array(0, dim = c(n, n, K))
    #the first change: Using verctorized expression rather than three nested for loop
    for (z in 1:K){
      q[,,z] <- theta.old[,z] %*% t(theta.old[,z])/Theta.old
    }
    #end of the first change
    for (z in 1:K) {
      #the second change:Using a variable rowsum to store the value of "rowSums(A*q[,,z])" r
      rowsum <- rowSums(A*q[,,z])
      theta.new[,z] <- rowsum/sqrt(sum(rowsum))
      #end of the second change
    }
    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
  set.seed(0)
  temp <- matrix(runif(n*K), n, K)
  theta.init <- temp/rowSums(temp)
  # do single update
  out <- oneUpdate(A, n, K, theta.init)
}
system.time(f1())
##      user   system  elapsed
##    383.21     0.69   388.75

system.time(f2())
##      user   system  elapsed
##      5.25     0.53     5.79
```

4 Problem 4

4.1 Problem a

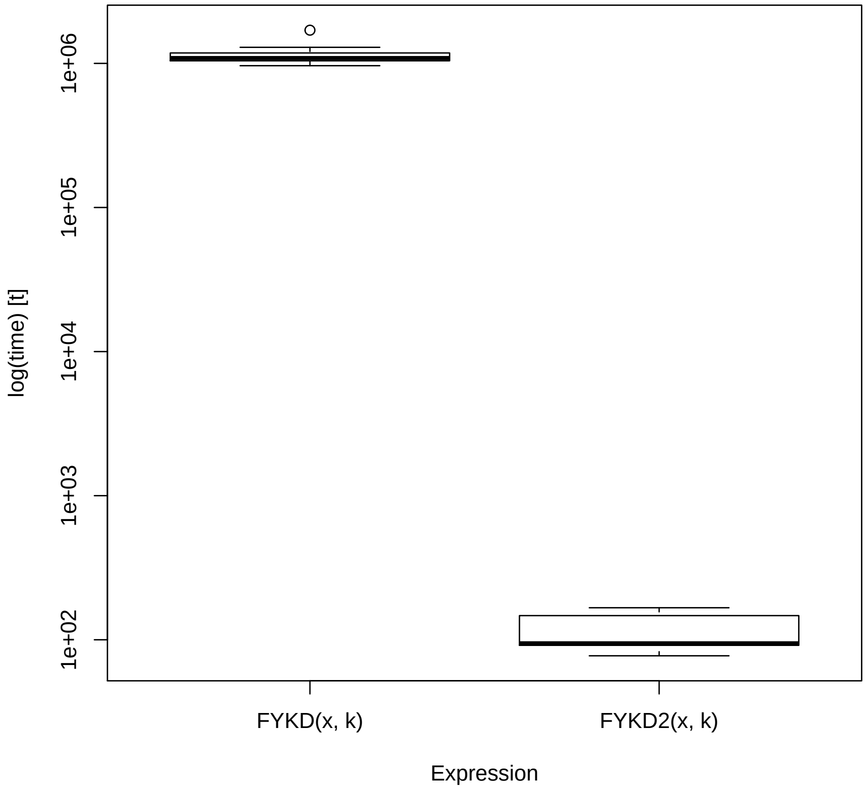
I speed up the FYKD function by replcing the unnecessary for loop and usin g the sample function more efficiently.

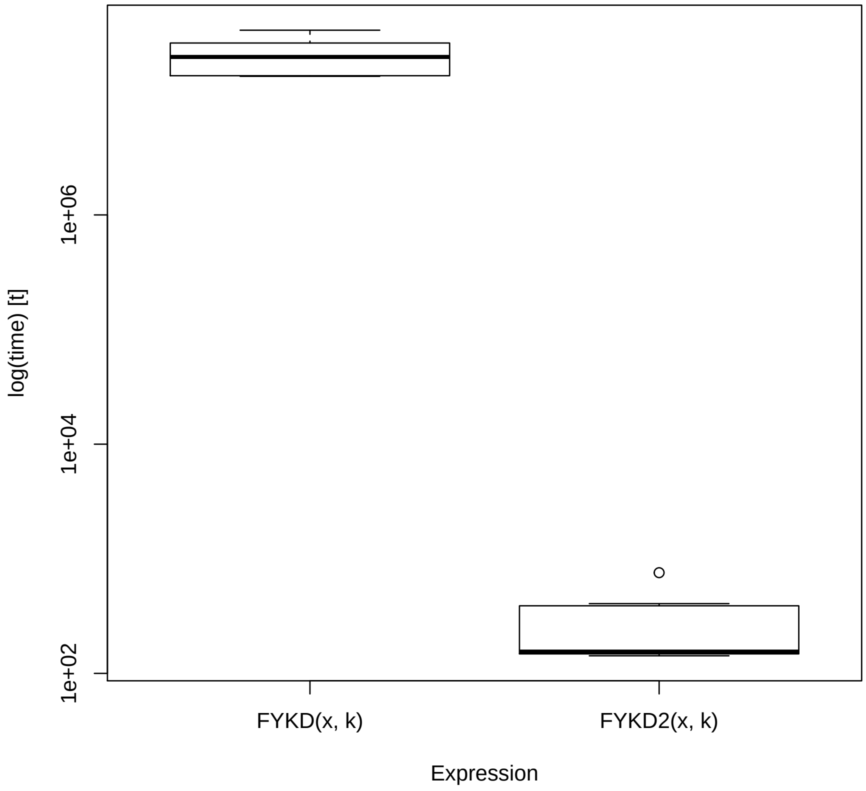
```r
library(microbenchmark)
n <- 10000
k <- 500
x <- 1:n
FYKD <- function(x,k){
  n <- length(x)
  for(i in 1:n){
    j = sample(1:n,1)
    tmp <- x[i]
    x[i] <- x[j]
    x[j] <- tmp
  }
  return(x[1:k])
}

#Refined one
FYKD2 <- function(x,k){
  n <- length(x)
  j = sample(1:n,k)
  return(x[j])
}

res <- microbenchmark(FYKD(x,k),FYKD2(x,k))
boxplot(res)
```

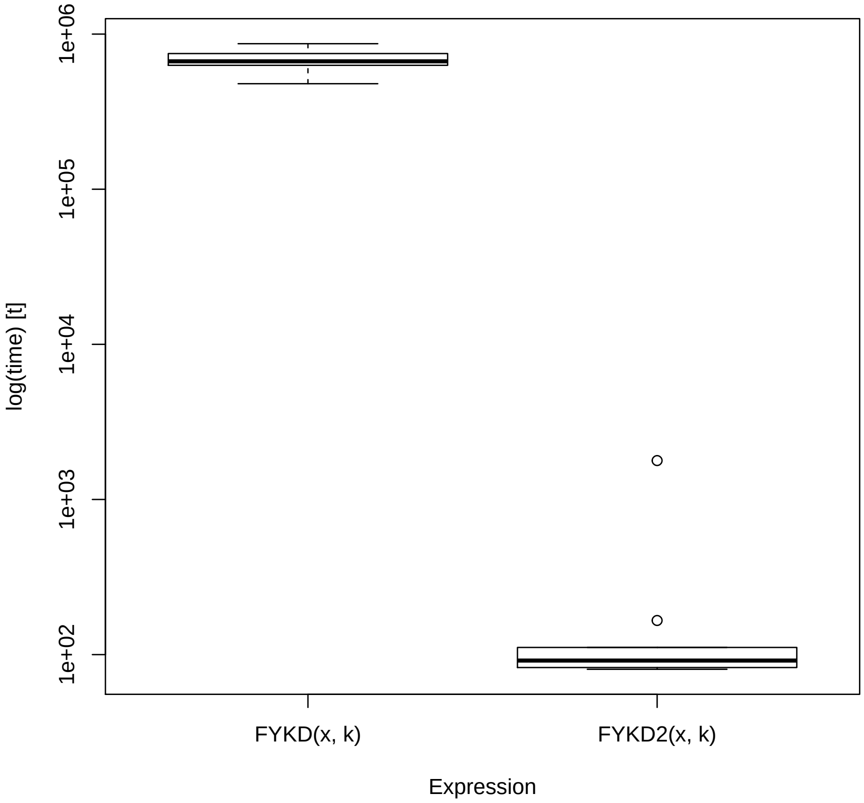


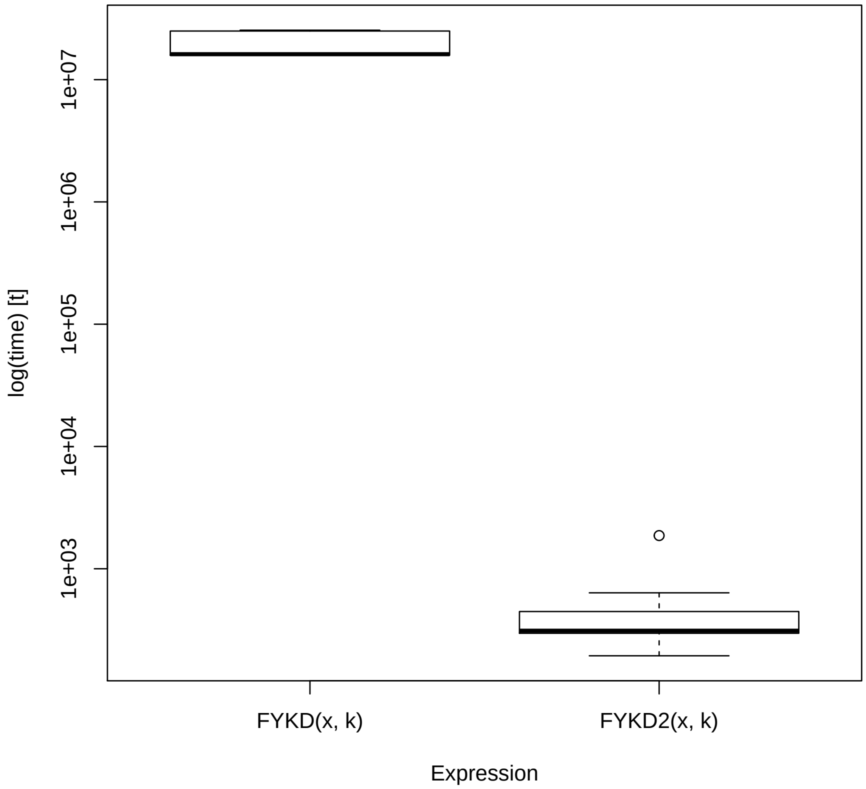
We can also figure out the performance when n and k changes.To save the waiting time, I set times = 10 in the microbenchmark function.

```
N <- seq(10000,100000,50000)
K <- seq(100,1000,500)

for(k in K){
  for(n in N){
    x <- 1:n
    res <- microbenchmark(FYKD(x,k),FYKD2(x,k),times=10)
    boxplot(res)
  }
}
```


log(time) [t]
1e+06
1e+05
1e+04
1e+03
1e+02
FYKD(x, k)
FYKD2(x, k)
Expression


log(time) [t]
1e+02
1e+04
1e+06
FYKD(x, k)
FYKD2(x, k)
Expression


1e+06
1e+05
1e+04
1e+03
1e+02
log(time) [t]
FYKD(x, k)
FYKD2(x, k)
Expression



4.2 Problem b

I manage to get the time for FYKD down to less than two times the time of sample (n=10000,k=500). And the detials can be seen in Problem a.

```
n <- 10000
k <- 500
x <- 1:n
microbenchmark(FYKD2(x,k),sample(x,k))
```

```
## Unit: microseconds
##          expr    min     lq      mean  median      uq      max neval
##   FYKD2(x, k) 69.706 73.555 122.58361 75.2650 76.9755 2022.737   100
##  sample(x, k) 40.626 44.475  50.73593 45.9725 47.6825  106.910   100
```