STAT 243 PS4

Riv Jenkins

1 a)

There will be some memory initially allocated for x at the beginning. When the function is defined, and myFun is initialized, there are no changes being made to the data, so no copies will be made. There will be one copy made when myFun is actually executed with param=3 because param*data will create a new object.

b)

```r
x <- 1:1e7
print(paste(c('Initial object size', object.size(x))))
```

```
## [1] "Initial object size" "40000040"
```

```r
f <- function(input){
data <- input
g <- function(param) return(param * data)
return(g)
}
myFun <- f(x)
ser = serialize(myFun, NULL)
print(object.size(ser))
```

```
## 80008088 bytes
```

The serialized object is twice the size of the original which does not seem to concurr with my answer for part (a). It seems that when the object is serialized the enclosing function gets evaluated and all copies get made then.

c)

R has something called lazy evaluation where the function arguments are not evaluated until they are used. In the example code the argument data is not actually evaluted until myFun(3) is called because it is not needed for any evaluations up to that point. However, at this point the data (which is just a local name for x) has been removed, so we get an error.

d)

The only way to avoid having the function make a copy would be to remove the the "rm(x)" line. This will keep x in the global environment, so when the function is actually evaluated, x will still be there for the function to find.

2 a)

```
list_of_vec = list(1:1e7, 1:1e7)
object.size(list_of_vec)
```

```
## 80000136 bytes
```

```
mem_change(list_of_vec[[1]][1] <- 5)
```

```
## 40 MB
```

In this case, it looks like there is a copy being made. The memory change is half the size of the original list (40 MB compared to 80 MB originally), so it looks like only the vector being changed is copied.

b)

```
list_of_vec = list(1:1e7, 1:1e7, 1:1e7, 1:1e7)
object.size(list_of_vec)
```

```
## 160000232 bytes
```

```
mem_change(list_of_vec2 <- list_of_vec)
```

```
## 504 B
```

```
gc()
```

```
##            used  (Mb) gc trigger  (Mb) max used  (Mb)
## Ncells   562384  30.1     940480  50.3   750400  40.1
## Vcells 35954298 274.4   61413851 468.6 50975062 389.0
```

```
mem_change(list_of_vec2[[1]][1] <- 5)
```

```
## 80 MB
```

```
gc()
```

```
##            used  (Mb) gc trigger  (Mb) max used  (Mb)
## Ncells   562421  30.1     940480  50.3   750400  40.1
## Vcells 45954354 350.7   73776621 562.9 50975062 389.0
```

When we copy the entire list, there is basically no memory change (4.16 kB), so there is definitely some sort of copy on change going on. When we modify one element of one of the vectors for the second list, it looks like the entire list is being copied based on gc(), but there is also some sort of strange double copying goin on. We can see this with mem_change but also with the max value reported by gc().

c)

```
list_of_list = list(list(1:1e6, 1:1e6), list(1:1e6, 1:1e6))
object.size(list_of_list)
```

```
## 16000328 bytes
```

```
mem_change(list_of_list2 <- list_of_list)
```

```
## 296 B
```

```
mem_change(list_of_list2[[3]] <- 5)
```

```
## 800 B
```

Here, we made a list of lists which is around 16 MB. We then copy it to list_of_list2 which only results in a 3.95 kB change in memory indicating that there is no deep copy being made at that time. We then add a small element to the second list, and this again results in a tiny change in memory use indicating that the two lists are still sharing the memory for the first two elements of both lists.

d)

```r
gc()
```

```
##            used  (Mb) gc trigger  (Mb) max used  (Mb)
## Ncells   562404  30.1     940480  50.3   750400  40.1
## Vcells 47954351 365.9   73776621 562.9 50975062 389.0
```

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

```
## @0x0000000017683618 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
##   @0x00007ff5e3020010 14 REALSXP g0c7 [NAM(2)] (len=10000000, tl=0) -2.29738,0.640409,-1.68945,1.3723
##   @0x00007ff5e3020010 14 REALSXP g0c7 [NAM(2)] (len=10000000, tl=0) -2.29738,0.640409,-1.68945,1.3723
```

```r
object.size(tmp)
```

```
## 160000136 bytes
```

```r
gc()
```

```
##            used  (Mb) gc trigger  (Mb) max used  (Mb)
## Ncells   562497  30.1     940480  50.3   750400  40.1
## Vcells 57954828 442.2   88611945 676.1 58027123 442.8
```

Object size is counting the amount of memory used by each element of the list. It just so happens that these are the same two blocks of memory, so object.size is counting them twice. If one of the elements were changed, then the results would match up.

3)

```r
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
set.seed(1234)
temp <- matrix(runif(n*K), n, K)
theta.init <- temp/rowSums(temp)
# do single update
system.time(out1 <- oneUpdate(A, n, K, theta.init))
```

```
##    user  system elapsed
##   28.61    0.38   29.30
```

The time I get for running the original function is around 10 seconds.

```r
load('ps4prob3.Rda') # should have A, n, K

ll <- function(Theta, A) {
  logLik <- sum(log(Theta[A==1])) - sum(Theta)
  return(logLik)
}

oneUpdate <- function(A, n, K, theta.old, thresh = 0.1) {
  Theta.old <- theta.old %*% t(theta.old)
  L.old <- ll(Theta.old, A)
  #here is the largest change to the code (this replaces the 3 for loops)
  q = apply(theta.old, 2, function(x) return(x%*%t(x) / Theta.old))
  q <- array(q, dim = c(n, n, K))

  for (z in 1:K) {
   theta.old[,z] <- rowSums(A*q[,,z])/sqrt(sum(A*q[,,z]))
  }

  Theta.old <- theta.old %*% t(theta.old)
  L.new <- ll(Theta.old, 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))
}
# initialize the parameters at random starting values
set.seed(1234)
temp <- matrix(runif(n*K), n, K)
theta.init <- temp/rowSums(temp)
# do single update
system.time(out <- oneUpdate(A, n, K, theta.init))
```

```
##    user  system elapsed
##    2.37    0.69    3.11
```

The new version of the function runs in about 1/10th the time of the original, essentially giving one order of magnitude speedup. The major difference is vectorizing the nested for loops so that part is far more efficient. There was also some amount of unnecessary copying going on which I got rid of. This affected some of the variable names especially in the second half of the function.

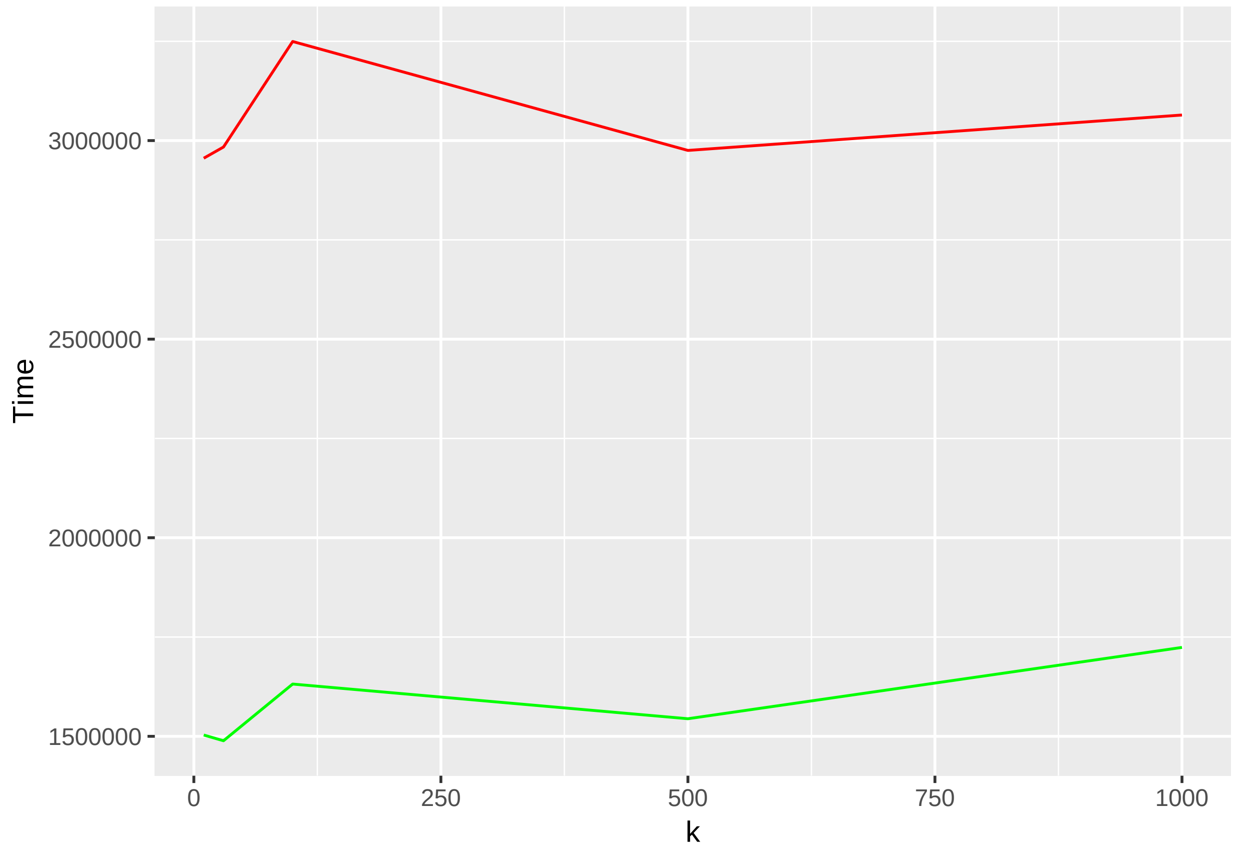
4

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

PIKKnew <- function(x, k) {
  #this modified function just sorts the pth element to avoid having to sort the entire vector
  nx <- length(x)
  p <- nx-k
  ranvec <- runif(nx)
  return(x[which(ranvec > sort(ranvec, partial=p)[p])])
}

set.seed(123)
x=runif(10000)
k=30
print(microbenchmark(PIKK(x,k), PIKKnew(x,k)))
```

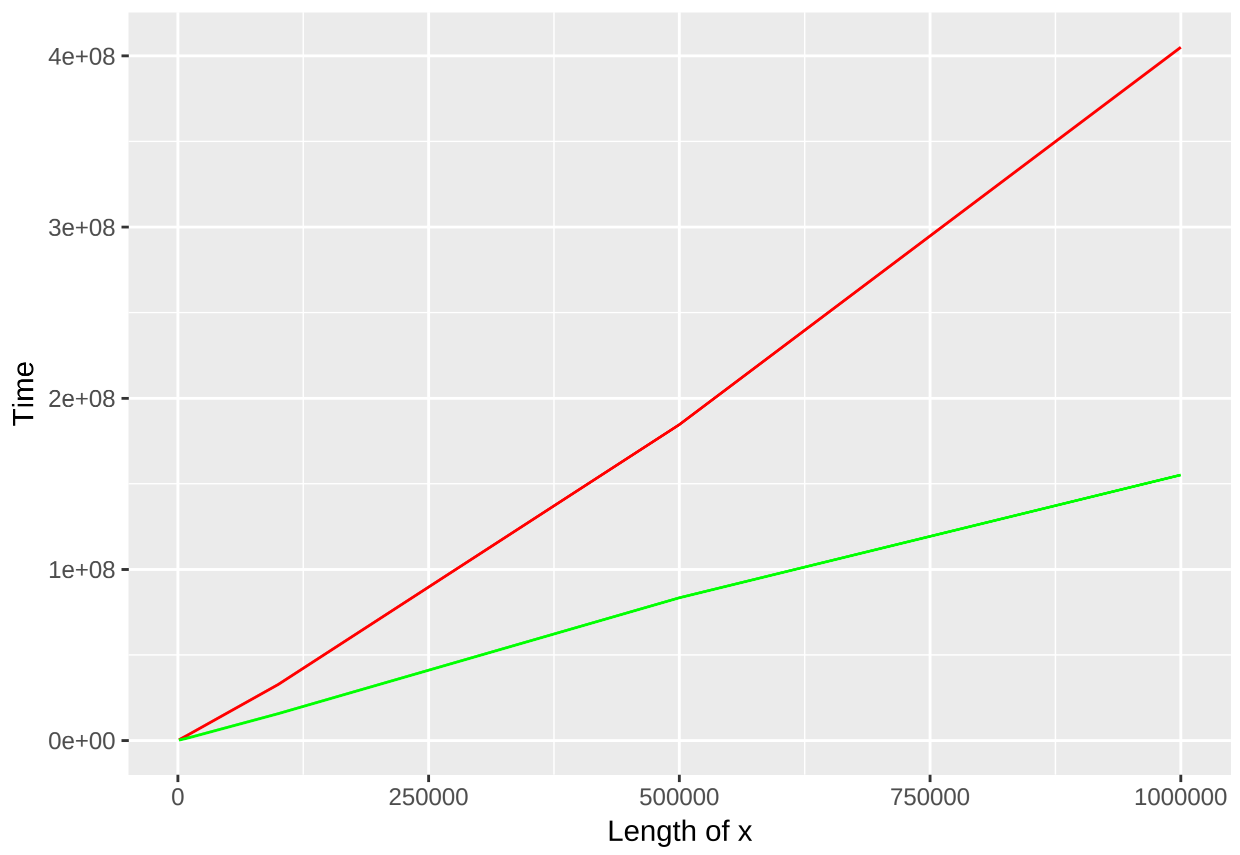
```
## Unit: milliseconds
##           expr      min       lq     mean   median       uq      max neval
##     PIKK(x, k) 2.908947 3.175344 3.714724 3.294709 4.039461 15.43585   100
##  PIKKnew(x, k) 1.314135 1.600836 1.973577 1.730465 1.970757 16.35864   100
```

```
x = runif(10000)
res1 = integer(5)
res2 = integer(5)
k = c(10, 30, 100, 500, 1000)
for(i in 1:5){
  res1[i] = mean(microbenchmark(PIKK(x,k[i]))$time)
  res2[i] = mean(microbenchmark(PIKKnew(x,k[i]))$time)
}
```

```r
df = data.frame(k, res1, res2)
g <- ggplot(df, aes(k)) +                    # basic graphical object
  geom_line(aes(y=res1), colour="red") +  # first layer
  geom_line(aes(y=res2), colour="green")
g <- g + ylab("Time")
g
```



```r
x = list(runif(1000), runif(10000), runif(100000), runif(500000), runif(1000000))
res1 = integer(5)
res2 = integer(5)
k = 100
for(i in 1:5){
  res1[i] = mean(microbenchmark(PIKK(x[[i]],k))$time)
  res2[i] = mean(microbenchmark(PIKKnew(x[[i]],k))$time)
}
lx = sapply(x, length)
df = data.frame(lx, res1, res2)
g <- ggplot(df, aes(lx)) +
  geom_line(aes(y=res1), colour="red") +
  geom_line(aes(y=res2), colour="green")
g <- g + ylab("Time") + xlab("Length of x")
g
```



The graphs show that the new version of the algorithm seems to follow the same trend as the original as k varies, but it gets significantly better than the original as the size of x increases.