Problem Set 4

Hsieh Cheng Han

October 10, 2017

1.

In class we discussed the idea of a closure as a function that has data associated with it. We saw that the following code embeds the value of x inside the function.

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

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

(a)

What is the maximum number of copies that exist of the vector 1:10 during the first execution of myFun()? Why?

(Ans)

When we create f function, no copies of 1:10 are copied. Then, f(x) is assigned to myFun. x is evaluated as input in the f function. Afterward, a copy of 1:10 is created to data inside f. Finally, when command myFun(3) is running, g is executed and data(1:10) is evaluated but not copied, returning the output. As a result, the maximum number of copies existing of the vector 1:10 during the first execution of myFun() is **one**.

(b)

Use serialize() to generate a sequence of bytes that store the information in the closure. Is the size of the serialized object the size you would expect given your answer to (a)? If not, can you explain what is happening? For this part of the problem, make x a vector of large enough vector that your answer concentrates on the number of bytes involved in the numeric vector not in the function and any overhead for storing R objects.

(Ans)

```
object.size(serialize(environment(myFun),connection = NULL))

## 5776 bytes

ls(environment(myFun))

## [1] "data"  "g"     "input"
```

The size of the serialized object is 5784 bytes, which is apparently larger than what we expected. (The size of vector 1:10 is 88 bytes.) The difference comes from existence of g function in myFun. We reset the x to be larger in order to ignore the existence of other objects in myFun.

```
x <- 1:10000
myFun <- f(x)
object.size(serialize(environment(myFun),connection = NULL))

## 85696 bytes

object.size(serialize(environment(myFun)$data,connection = NULL))

## 40064 bytes
```

We can observe that the size of myFun is approximately twice the size of data inside it.

(c)

It seems unnecessary to have the data ¡- input line, so lets try the following.

```
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):   'x'
```

Explain what is happening and why this doesnt work to embed a constant data value into the function. Recall our discussion of when function arguments are evaluated.

(Ans)
R does not (usually) evaluate the arguments until they are needed, which is called lazy evaluation. When f(x) is assigned to myFun, vector 1:10 will not copied to data inside f since there is no need. Afterwards, myFun(3) is executed, data inside f(actually x) shoulde be executed. Since there's no such data in the calling environment, R looks for enclosing environment(where x is created). But we have removed x already. That's the reason of warning message : object 'x' not found.

(d)

(d) Can you figure out a way to make the code in part (c) work without explicitly creating a copy of the vector as in the original code? If you do that, how big is the resulting serialized closure?

(Ans)
We can try to redeem by not removing x :

```
x <- 1:10
f <- function(data){
  g <- function(param) return(param * data)
  return(g)
}
myFun <- f(x)
myFun(3)

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

object.size(serialize(environment(myFun),connection = NULL))

## 6320 bytes
```

The resulting serialized closure is 6320 bytes.

2.

This question explores memory use and copying with lists. In answering this question you can ignore what is happening with the list attributes, which are also reported by .Internal(inspect()).

(a)

Consider a list of vectors. Modify an element of one of the vectors. Can R make the change in place, without creating a new list or a new vector?

```
# Load the necessary package.
library(pryr)

##
## Attaching package:  'pryr'
## The following object is masked _by_ '.GlobalEnv':
##
##    f

# Create a list of vectors.
a <- list(c(1:10),c(1:10))
# Check the address of the list.
.Internal(inspect(a))

## @7fd0b2e076a8 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##   @7fd0b2effad8 13 INTSXP g0c4 [] (len=10, tl=0) 1,2,3,4,5,...
##   @7fd0b2effba8 13 INTSXP g0c4 [] (len=10, tl=0) 1,2,3,4,5,...

# Now we mofidy the first element of the first vector in the list.
a[[1]][1] <- 3
# Check again the address.
.Internal(inspect(a))
```

```
## @7fd0b29c6a38 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
##   @7fd0b1ce9350 14 REALSXP g0c5 [] (len=10, tl=0) 3,2,3,4,5,...
##   @7fd0b2effba8 13 INTSXP g0c4 [NAM(2)] (len=10, tl=0) 1,2,3,4,5,...
```

From the result we can conclude that the address of the two vectors' addresses changes as well as the address of the first vector. However, the address in which the second vector stores doesn't change. That is, the entire list won't be copied under the modificaiton.

(b)

Next, make a copy of the list and determine if there any copy-on-change going on. When a change is made to one of the vectors in one of the lists, is a copy of the entire list made or just of the relevant vector?

```
# Make a copy of the original list.
b <- a
# Check whether the addresses are the same.
identical(.Internal(inspect(a)),.Internal(inspect(b)))

## @7fd0b29c6a38 19 VECSXP g0c2 [MARK,NAM(2)] (len=2, tl=0)
##   @7fd0b1ce9350 14 REALSXP g0c5 [MARK] (len=10, tl=0) 3,2,3,4,5,...
##   @7fd0b2effba8 13 INTSXP g0c4 [MARK,NAM(2)] (len=10, tl=0) 1,2,3,4,5,...
## @7fd0b29c6a38 19 VECSXP g0c2 [MARK,NAM(2)] (len=2, tl=0)
##   @7fd0b1ce9350 14 REALSXP g0c5 [MARK] (len=10, tl=0) 3,2,3,4,5,...
##   @7fd0b2effba8 13 INTSXP g0c4 [MARK,NAM(2)] (len=10, tl=0) 1,2,3,4,5,...
## [1] TRUE

# Now modify the original list again.
a[[1]][1] <- 2
identical(.Internal(inspect(a)),.Internal(inspect(b)))

## @7fd0b1aefaa8 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
##   @7fd0b1ce0f48 14 REALSXP g0c5 [] (len=10, tl=0) 2,2,3,4,5,...
##   @7fd0b2effba8 13 INTSXP g0c4 [MARK,NAM(2)] (len=10, tl=0) 1,2,3,4,5,...
## @7fd0b29c6a38 19 VECSXP g0c2 [MARK,NAM(2)] (len=2, tl=0)
##   @7fd0b1ce9350 14 REALSXP g0c5 [MARK,NAM(2)] (len=10, tl=0) 3,2,3,4,5,...
##   @7fd0b2effba8 13 INTSXP g0c4 [MARK,NAM(2)] (len=10, tl=0) 1,2,3,4,5,...
## [1] FALSE
```

From the result we can conclude that the address of the copied list remains while the address of the original list changes. That is, a new memory block was allocated to store the modified list but the copied list is not effected.

(c)

Now make a list of lists. Copy the list. Add an element to the second list. Explain what is copied and what is not copied and what data is shared between the two lists.

```
# Make a list of lists and copy it.
c <- list(list(1,2),list(4,5))
d <- c
# Check their addresses.
identical(.Internal(inspect(c)),.Internal(inspect(d)))
```

```
## @7fd0b3045a08 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##   @7fd0b3091f08 19 VECSXP g0c2 [] (len=2, tl=0)
##     @7fd0b2daa358 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 1
##     @7fd0b2da6408 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 2
##   @7fd0b3091f40 19 VECSXP g0c2 [] (len=2, tl=0)
##     @7fd0b2da6438 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 4
##     @7fd0b2da6468 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 5
## @7fd0b3045a08 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##   @7fd0b3091f08 19 VECSXP g0c2 [] (len=2, tl=0)
##     @7fd0b2daa358 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 1
##     @7fd0b2da6408 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 2
##   @7fd0b3091f40 19 VECSXP g0c2 [] (len=2, tl=0)
##     @7fd0b2da6438 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 4
##     @7fd0b2da6468 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 5
## [1] TRUE
```

```
# Add an element to the first list of the copied list.
d[[1]] <- append(d[[1]],7)
identical(.Internal(inspect(c)),.Internal(inspect(d)))
```

```
## @7fd0b3045a08 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##   @7fd0b3091f08 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##     @7fd0b2daa358 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 1
##     @7fd0b2da6408 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 2
##   @7fd0b3091f40 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##     @7fd0b2da6438 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 4
##     @7fd0b2da6468 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 5
## @7fd0b2958a98 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
##   @7fd0b324d360 19 VECSXP g0c3 [NAM(2)] (len=3, tl=0)
##     @7fd0b2daa358 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 1
##     @7fd0b2da6408 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 2
##     @7fd0b3036ea8 14 REALSXP g0c1 [] (len=1, tl=0) 7
##   @7fd0b3091f40 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##     @7fd0b2da6438 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 4
##     @7fd0b2da6468 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 5
## [1] FALSE
```

From the result we can conclude that the address of the orginal doesn't change, while the modified part of the copied list(d) changes. That is, only the modified part of the list will be copied while the other part of the list remains the same.

(d)

Run the following code in a new R session. The result of .Internal(inspect()) and of object.size() conflict with each other. In reality only 80 MB is being used, as can be seen with gc(). Explain why this is the case.

```
gc()
```

```
##          used (Mb) gc trigger (Mb) max used (Mb)
## Ncells 363135 19.4     750400 40.1   592000 31.7
## Vcells 570386  4.4    1308461 10.0   975173  7.5
```

```
tmp <- list()
```

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

## @7fd0b2eeded8 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
##   @10b93c000 14 REALSXP g0c7 [NAM(2)] (len=10000000, tl=0) -0.863702,0.0685227,-0.415678,0.320101,-0
##   @10b93c000 14 REALSXP g0c7 [NAM(2)] (len=10000000, tl=0) -0.863702,0.0685227,-0.415678,0.320101,-0

object.size(tmp)

## 160000136 bytes

gc()

##            used (Mb) gc trigger  (Mb) max used (Mb)
## Ncells   363222 19.4     750400  40.1   592000 31.7
## Vcells 10570845 80.7   15623918 119.3 10622175 81.1
```

The reason why the object.size(tmp) is twice than the memory actually used is that x is copied twice when assigned to tmp[[1]] and tmp[[2]]. As both tmp[[1]] and tmp[[2]] point to the same address, the size of tmp only increases one time.

3.

Challenge 5 of Section 7.3 of Unit 4. The following is real code for maximizing a likelihood function of a statistical model, written by a Statistics grad student. The goal is to improve the efficiency of this R code. There are a number of improvements that can be made; in particular the code should not need three nested for loops. Consider also whether there are any calculations that are done repeatedly that need only be done once. Report the time it takes before and after your improvements. Compared to this code, I was able to achieve a 12-fold speedup. Also, the style of the code Ive given you could stand some improvement (though you should probably keep the names of objects somewhat similar to what they currently are to assist comparing be

```
load("/Users/henry50618/Desktop/ps4prob3.Rda")
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
temp <- matrix(runif(n*K), n, K)
theta.init <- temp/rowSums(temp)
# do single update
out <- oneUpdate(A, n, K, theta.init)
```

(Ans)
First we record the time for running the original code.

```
system.time(oneUpdate(A, n, K, theta.init))

##    user  system elapsed
##  83.627   1.857  99.197

q1 <- q
```

We try to rewrite the code without three nested for loop: First, the if-else conditional statement can be removed since the result will be equivalent in case of theta.old[i, z]*theta.old[j, z] == 0. Then, we remove i and j for loops and use matrix manipulation instead. Note that transpose should be used in order to fulfill matrix manipulation.

```
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))
  # Remove i and j for loops.
  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))
}
```

Then, we rerun the code :

```
system.time(oneUpdate(A, n, K, theta.init))

##    user  system elapsed
##   0.953   0.364   1.360

q2 <- q
# To Assure the advised algorithm gives the same result.
identical(q1,q2)

## [1] TRUE
```

Apparently the code runs much faster than the original one.

4.

This is a variation on Challenge 8 in Section 7.3 of Unit 4. The goal is to write a function to sample k values without replacement from a population of size n. The inputs to the function are a vector, x, of the values of the population, and k. The sample() function in R is quite fast using microbenchmark, I get a time of about 20 microseconds (for 100 evaluations) for n = 10000 and k = 500 on my computer. Now consider the PIKK and FYKD algorithms as implemented (naively) here:

```
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)
      # change the contents of x[i] and x[j]
      tmp <- x[i]
      x[i] <- x[j]
      x[j] <- tmp
    }
return(x[1:k])
}
```

(a)(b)

Figure out a way to speed up one of the two algorithms (or come up with a different approach) and demonstrate that your solution is faster than the code I present.

```
# Load necessary packages. grr has modified basic R functions such as
# sort2, order2 and sample2.
library(rbenchmark)
library(microbenchmark)
library(grr)

# First we try to rewrite PIKK algorithm. We use order2 function to get the index
# directly.
myPIKK <- function(x, k) {
    return(order2(runif(x))[1:k])
```

```r
}

# Then we try to rewrite FYKD algorithm.
myFYKD <- function(x, k) {
    # Pre-allocate the memory for the output.
    tmp <- vector("list", k)
    n <- length(x)
    # Fill up the output list by each element.
    for(i in 1:k) {
      j <- sample(n, 1)
      tmp[[i]] <- x[j]
      # We need to fill x[j] with any element, choose the last one.
      x[j] <- x[n]
      # n - 1 reflects "without replacement".
      n <- n - 1
    }
return(tmp)
}
```

Now we compare our algorithms with original ones.

```r
x <- 1:10000
k <- 500
microbenchmark(PIKK(x,k),myPIKK(x,k))

## Unit: microseconds
##          expr      min       lq     mean   median       uq      max neval
##    PIKK(x, k) 1381.273 1444.225 1565.431 1530.232 1653.362 2037.618   100
##  myPIKK(x, k)  980.536 1019.791 1103.134 1074.778 1156.861 1501.011   100

microbenchmark(FYKD(x,k),myFYKD(x,k))

## Unit: milliseconds
##          expr       min         lq       mean     median         uq
##    FYKD(x, k) 139.27232 171.224010 194.634673 182.661880 211.515667
##  myFYKD(x, k)   2.67615   3.024382   5.014736   3.694587   4.938133
##        max neval
##  297.01904   100
##   24.05054   100
```

Apparently myFYKD is much efficient than FYKD. Now we consider x and k vary.

```r
# Change x.
x <- 1:20000
microbenchmark(PIKK(x,k),myPIKK(x,k))

## Unit: milliseconds
##          expr      min       lq    mean   median       uq       max neval
##    PIKK(x, k) 3.224845 3.665810 4.74803 4.226855 5.682663  9.220841   100
##  myPIKK(x, k) 2.158802 2.369496 3.48080 2.682743 3.600978 17.202451   100

microbenchmark(FYKD(x,k),myFYKD(x,k))

## Unit: milliseconds
##          expr        min         lq       mean    median         uq
##    FYKD(x, k) 414.566313 456.124976 570.256222 496.32206 646.798922
##  myFYKD(x, k)   2.686765   2.916362   4.132098   3.40596   4.471783
##         max neval
##  1220.84242   100
##    12.70797   100
```

```r
# Change k.
k <- 1500
microbenchmark(PIKK(x,k),myPIKK(x,k))
```

```
## Unit: milliseconds
##          expr      min       lq     mean   median       uq      max neval
##    PIKK(x, k) 3.487903 5.645828 7.937335 6.380612 7.853016 83.05152   100
##  myPIKK(x, k) 2.247910 3.228395 5.179448 3.991914 5.675959 27.39247   100
```

```r
microbenchmark(FYKD(x,k),myFYKD(x,k))
```

```
## Unit: milliseconds
##          expr        min         lq      mean    median        uq
##    FYKD(x, k) 424.494727 466.720278 587.07655 523.78011 695.97666
##  myFYKD(x, k)   8.350164   9.824525  15.20083  13.58295  17.78699
##         max neval
##  1026.57955   100
##    36.91667   100
```