STAT243-PS4

Jinhui Xu

September 2017

1 Other students

I discuss some problems with Xin Shi.

2 Question 1

2.1 (a)

There is only one copy. Because we can see that data and input share same address.

```
x <- 1:10
f <- function(input){
  print(.Internal(inspect(input)))       #check the address of input
  data <- input
  print(.Internal(inspect(data)))        #check the address of data
        g <- function(param) return(param * data)
        return(g)
}
data<-100
myFun <- f(x)

## @11745fe10 13 INTSXP g0c4 [NAM(2)] (len=10, tl=0) 1,2,3,4,5,...
##  [1]  1  2  3  4  5  6  7  8  9 10
## @11745fe10 13 INTSXP g0c4 [NAM(2)] (len=10, tl=0) 1,2,3,4,5,...
##  [1]  1  2  3  4  5  6  7  8  9 10
```

2.2 (b)

The size of the serialized object is doubled. The reason is that R store both input and data even though they have same address.

```
x <- rnorm(1e5)
f <- function(input){
  data <- input
        g <- function(param) return(param * data)
        return(g)
}
myFun <- f(x)
object.size(x)

## 800040 bytes

length(serialize(myFun,NULL))
```

```
## [1] 1606454
```

2.3 (c)

When the function contains the command: data=input. myFun can get the value of data even x is removed. However, when we delete that command, myFun need the value of input of f which means that myFun needs the value of x. So if we rm x, there would be an error.

```
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'

ls(envir=environment(myFun))

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

2.4 (d)

We can use force to force the value of data.

```
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
```

3 Question 2

3.1 (a)

When change the a vector of list, I find that the address of the relevant changes and the other one does not change. Therefore, R would create a new vector.

```
list1=list(a=rnorm(1e5),b=rnorm(1e5))
.Internal(inspect(list1))
```

```
## @116e802a0 19 VECSXP g0c2 [NAM(2),ATT] (len=2, tl=0)
##   @11301d000 14 REALSXP g0c7 [] (len=100000, tl=0) 0.260864,1.72475,-1.39273,-0.308712,0.459819,...
##   @109baf000 14 REALSXP g0c7 [] (len=100000, tl=0) 0.720782,-0.750896,-0.00154814,-0.591199,-0.46508
## ATTRIB:
##   @110353bf0 02 LISTSXP g0c0 []
##     TAG: @103824140 01 SYMSXP g1c0 [MARK,NAM(2),LCK,gp=0x6000] "names" (has value)
##     @116e802d8 16 STRSXP g0c2 [] (len=2, tl=0)
##       @103021b98 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "a"
##       @1032eab98 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "b"
```

```
list1[[1]][1]<-100
.Internal(inspect(list2))
```

```
## @11864b118 19 VECSXP g1c2 [MARK,NAM(2),ATT] (len=2, tl=0)
##   @10b47e000 14 REALSXP g1c7 [MARK,NAM(2)] (len=100000, tl=0) -1.05714,0.0947067,-1.0191,1.53438,-2.
##   @10b700000 14 REALSXP g1c7 [MARK,NAM(2)] (len=100000, tl=0) 0.19425,-0.332612,0.753014,0.510845,-0
## ATTRIB:
##   @135938190 02 LISTSXP g1c0 [MARK]
##     TAG: @103824140 01 SYMSXP g1c0 [MARK,NAM(2),LCK,gp=0x6000] "names" (has value)
##     @11864b188 16 STRSXP g1c2 [MARK,NAM(2)] (len=2, tl=0)
##       @103021b98 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "a"
##       @1032eab98 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "b"
```

3.2 (b)

According to the adddress of two lists before any change, we know that there is no copy-on-change.When the change is made, only the address of the relevant vector changes. Therefore, only a copy of the relevant vector is made.

```
list2=list(a=rnorm(1e5),b=rnorm(1e5))
list2_cp=list2
.Internal(inspect((list2)))
```

```
## @116103c38 19 VECSXP g0c2 [NAM(2),ATT] (len=2, tl=0)
##   @110900000 14 REALSXP g0c7 [] (len=100000, tl=0) -0.315374,1.05064,1.03944,0.722292,-0.809419,...
##   @110d00000 14 REALSXP g0c7 [] (len=100000, tl=0) 0.0720925,0.810187,0.415644,-0.576821,3.24783,...
## ATTRIB:
##   @127c22808 02 LISTSXP g0c0 []
##     TAG: @103824140 01 SYMSXP g1c0 [MARK,NAM(2),LCK,gp=0x6000] "names" (has value)
##     @116103ca8 16 STRSXP g0c2 [] (len=2, tl=0)
##       @103021b98 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "a"
##       @1032eab98 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "b"
```

```
.Internal(inspect((list2_cp)))
```

```
## @116103c38 19 VECSXP g0c2 [NAM(2),ATT] (len=2, tl=0)
##   @110900000 14 REALSXP g0c7 [] (len=100000, tl=0) -0.315374,1.05064,1.03944,0.722292,-0.809419,...
##   @110d00000 14 REALSXP g0c7 [] (len=100000, tl=0) 0.0720925,0.810187,0.415644,-0.576821,3.24783,...
## ATTRIB:
##   @127c22808 02 LISTSXP g0c0 []
##     TAG: @103824140 01 SYMSXP g1c0 [MARK,NAM(2),LCK,gp=0x6000] "names" (has value)
##     @116103ca8 16 STRSXP g0c2 [] (len=2, tl=0)
##       @103021b98 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "a"
##       @1032eab98 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "b"
```

```
#the address of list2_cp is same with that of list2
list2_cp[[1]][1]<-100
.Internal(inspect(list2_cp))

## @1163d1460 19 VECSXP g0c2 [NAM(1),ATT] (len=2, tl=0)
##   @110dc4000 14 REALSXP g0c7 [] (len=100000, tl=0) 100,1.05064,1.03944,0.722292,-0.809419,...
##   @110d00000 14 REALSXP g0c7 [NAM(2)] (len=100000, tl=0) 0.0720925,0.810187,0.415644,-0.576821,3.2478
## ATTRIB:
##   @119702d58 02 LISTSXP g0c0 []
##     TAG: @103824140 01 SYMSXP g1c0 [MARK,NAM(2),LCK,gp=0x6000] "names" (has value)
##     @116103ca8 16 STRSXP g0c2 [NAM(2)] (len=2, tl=0)
##       @103021b98 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "a"
##       @1032eab98 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "b"

#after change, we can find only the address of relevant vector changes
```

3.3 (c)

Notice the change of address after adding a vector into the second list. The address of two lists becomes different, but two original vectors still have original addresses. So the only change is that the second list creates a new vector while other vectors still share original addresses.

```
list3=list(a=list(rnorm(1e5)),b=list(rnorm(1e5)))
list3_cp=list3
.Internal(inspect(list3))

## @1336c3940 19 VECSXP g0c2 [NAM(2),ATT] (len=2, tl=0)
##   @127ff1af8 19 VECSXP g0c1 [] (len=1, tl=0)
##     @109900000 14 REALSXP g0c7 [] (len=100000, tl=0) -0.902474,0.657533,-1.1857,-0.844848,0.079948,.
##   @127ff1b28 19 VECSXP g0c1 [] (len=1, tl=0)
##     @11301d000 14 REALSXP g0c7 [] (len=100000, tl=0) -0.789741,0.481347,-0.815168,-0.121418,1.07905,
## ATTRIB:
##   @133723230 02 LISTSXP g0c0 []
##     TAG: @103824140 01 SYMSXP g1c0 [MARK,NAM(2),LCK,gp=0x6000] "names" (has value)
##     @1336c2a40 16 STRSXP g0c2 [] (len=2, tl=0)
##       @103021b98 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "a"
##       @1032eab98 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "b"

.Internal(inspect(list3_cp))

## @1336c3940 19 VECSXP g0c2 [NAM(2),ATT] (len=2, tl=0)
##   @127ff1af8 19 VECSXP g0c1 [] (len=1, tl=0)
##     @109900000 14 REALSXP g0c7 [] (len=100000, tl=0) -0.902474,0.657533,-1.1857,-0.844848,0.079948,.
##   @127ff1b28 19 VECSXP g0c1 [] (len=1, tl=0)
##     @11301d000 14 REALSXP g0c7 [] (len=100000, tl=0) -0.789741,0.481347,-0.815168,-0.121418,1.07905,
## ATTRIB:
##   @133723230 02 LISTSXP g0c0 []
##     TAG: @103824140 01 SYMSXP g1c0 [MARK,NAM(2),LCK,gp=0x6000] "names" (has value)
##     @1336c2a40 16 STRSXP g0c2 [] (len=2, tl=0)
##       @103021b98 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "a"
##       @1032eab98 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "b"

list3_cp$b[[2]]<-rnorm(1e5)
.Internal(inspect(list3_cp))
```

```
## @1336bdb58 19 VECSXP g0c2 [NAM(1),ATT] (len=2, tl=0)
##   @127ff1af8 19 VECSXP g0c1 [NAM(2)] (len=1, tl=0)
##     @109900000 14 REALSXP g0c7 [] (len=100000, tl=0) -0.902474,0.657533,-1.1857,-0.844848,0.079948,.
##   @1336bdb90 19 VECSXP g0c2 [] (len=2, tl=0)
##     @11301d000 14 REALSXP g0c7 [NAM(2)] (len=100000, tl=0) -0.789741,0.481347,-0.815168,-0.121418,1.
##     @110e88000 14 REALSXP g0c7 [NAM(2)] (len=100000, tl=0) -0.311266,0.221599,0.132626,-0.370482,0.2
## ATTRIB:
##   @1335a5440 02 LISTSXP g0c0 []
##     TAG: @103824140 01 SYMSXP g1c0 [MARK,NAM(2),LCK,gp=0x6000] "names" (has value)
##     @1336c2a40 16 STRSXP g0c2 [NAM(2)] (len=2, tl=0)
##       @103021b98 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "a"
##       @1032eab98 09 CHARSXP g1c1 [MARK,gp=0x61] [ASCII] [cached] "b"
```

3.4 (d)

Object.size is twice large as the result of gc. I guess that it is because two elements of list is stored in the same address, but object.size estimates the size of list equels to sum of size of each element.

```
gc()

##            used  (Mb) gc trigger  (Mb)  max used  (Mb)
## Ncells  1650699  88.2    2637877 140.9   2637877 140.9
## Vcells 30261474 230.9   97834876 746.5 109412705 834.8

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

## @119390740 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
##   @143800000 14 REALSXP g0c7 [NAM(2)] (len=10000000, tl=0) -0.0151081,-0.752749,-0.294608,2.24558,0.
##   @143800000 14 REALSXP g0c7 [NAM(2)] (len=10000000, tl=0) -0.0151081,-0.752749,-0.294608,2.24558,0.

object.size(tmp)

## 160000136 bytes

gc()

##            used  (Mb) gc trigger  (Mb)  max used  (Mb)
## Ncells  1650748  88.2    2637877 140.9   2637877 140.9
## Vcells 30261575 230.9   97834876 746.5 109412705 834.8
```

4 Question 3

Notice that in the original code,firstly, the if else is not necessary at all. So I directly calculate q without if else. Secondly I replace three nested for loops with simple computation of vector

```
ll <- function(Theta, A) {
  sum.ind <- which(A==1, arr.ind=T)
  logLik <- sum(log(Theta[sum.ind])) - sum(Theta)
```

```r
  return(logLik)
}
########################################
##original code########################
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))
##############the following part would be revised
  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))
}

########################################
##revised code#########################
oneUpdate_new <- 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))
####################begin.revised part
  for (z in 1:K) {
       q[ , , z] <- theta.old[, z]%*%t(theta.old[ , z]) /Theta.old
  }
####################end.revised part
  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)
#compare the time used
system.time(out <- oneUpdate(A, n, K, theta.init))

##    user  system elapsed
##   5.458   0.227   5.759

system.time(out_new <- oneUpdate_new(A, n, K, theta.init))

##    user  system elapsed
##   0.736   0.335   1.075

all.equal(out,out_new)

## [1] TRUE
```

The results are same while the time used by revised code decreases.

5 Question 4

Notice that in the function FYKD, the for loop is aim to generate vector x. However, the algorithm only need the first k value of vector x. So we can only calculate that part rather than entire vector.

```
PIKK <- function(x, k) {
x[sort(runif(length(x)), index.return = TRUE)$ix[1:k]]
}
FYKD <- function(x, k) {
  n <- length(x)
#in the original code , the following code is to generate entire n values in vector 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])  # while we only need first k values
}

#so revised code do not calculate the latter part of the vector
FYKD_new <- function(x, k) {
  n <- length(x)
  for(i in 1:k) {
     j = sample(i:n, 1)
     tmp <- x[i]
     x[i] <- x[j]
     x[j] <- tmp
  }
return(x[1:k])
}
```