Zicheng Huang, STAT243 PS4

Collaborators: Gao Shan, Junyi Tang

10/11/2017

```r
library(microbenchmark)
library(ggplot2)
```

1(a) According to the result from .Internal(inspect), we can see that only one copy of the vector 1:10 is created, the two local variables input and data are pointing to the same copy of the vector 1:10 since their addresses are the same.

```r
x <- 1:10
f <- function(input){
  data <- input
  # addresses of data and input
  .Internal(inspect(data))
  .Internal(inspect(input))
  g <- function(param) return(param * data)
  return(g)
}
myFun <- f(x)

## @0x000000001254c850 13 INTSXP g0c4 [NAM(2)] (len=10, tl=0) 1,2,3,4,5,...
## @0x000000001254c850 13 INTSXP g0c4 [NAM(2)] (len=10, tl=0) 1,2,3,4,5,...
```

1(b) We make 'x' a large vector with size of around 40000 bytes. In part (a) we argue that only one copy of 'x' is made, we expect to see the number of bytes that store the information in the closure to be close to the size of 'x'. However, the actual size of the serialized 'myFun' is around 80000 bytes, twice of what we expect. This difference is due to the fact that under the function frame of myFun there exist the body text of the function, the variable 'input', and the variable 'data'. Even though both 'input' and 'data' are pointing to 'x', when R saves 'myFun' what happens is that 'input' and 'data' are saved seperated as two individual objects each takes up around 40000 bytes of size. Thus, the size of 'myFun' is 80000 bytes which is twice the size of what we expected.

```r
x <- 1:10000
# size of x
length(serialize(x, NULL))

## [1] 40022

f <- function(input){
  data <- input
  g <- function(param) return(param * data)
  return(g)
}
myFun <- f(x)
# size of myFun
length(serialize(myFun, NULL))

## [1] 86993
```

1(c) When 'myFun' is define by f(x), f(x) is not evaluated due to the lazy evaluation, so the user supplied argument 'data' in the function frame of 'f' does not take in the values in 'x'. When myFun(3) is called, we see that what is actually being called is g(3). When g(3) is evaluated, g will look for a variable 'data' to return (3 * data), but since no 'data' is defined in the function frame of 'g', it will go to the enclosing environment, function frame of 'f', to look for 'data'. Due to the lazy evaluation and the fact that 'x' is removed, g cannot find 'data' in the function frame of 'f' as well. Since there exists no 'data' in the global environment, myFun(3) will return error.

```r
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
```

1(d) Since we prevented the lazy evaluation from happening using force(), when we define myFun by passing in f(x), f(x) will be evaluated in a sense that user supplied argument 'data' will take in variable of 'x'. In this way, even 'x' is removed, when myFun(3) is called, g(3) will be able to find a local vaiable 'data' in the function frame of f to return (param * data), thus preventing the error from happening again. The resulting serialized closure has size around 40000 bytes.

```r
x <- 1:10000
f <- function(data){
  force(data) # add this line to prevent the lazy evaluation from happening
  g <- function(param) return(param * data)
  return(g)
}
myFun <- f(x)
rm(x)
head(myFun(3))

## [1]  3  6  9 12 15 18

# size of modified 'myFun'
length(serialize(myFun, NULL))

## [1] 47349
```

2(a) According to the results from .Internal(inspect) in a plain R session, we can see that the address of the two lists, before and after modification, are the same, so no new lists are created when the change is made.

```r
lst <- list(c(1, 2),c(3, 4))
.Internal(inspect(lst))
lst[[1]][1] <- 8
.Internal(inspect(lst))

## Results shown in plain R session

## @0x000000000399b3a8 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
##   @0x000000000399b418 14 REALSXP g0c2 [] (len=2, tl=0) 1,2
##   @0x000000000399b3e0 14 REALSXP g0c2 [] (len=2, tl=0) 3,4
## @0x000000000399b3a8 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
##   @0x000000000399b418 14 REALSXP g0c2 [] (len=2, tl=0) 8,2
##   @0x000000000399b3e0 14 REALSXP g0c2 [] (len=2, tl=0) 3,4
```

2(b) After making a copy of the original list 'lst', we see that there is no copy-on-change going on since the address of the two lists are the same.

```
lst = list(c(1, 2),c(3, 4))
lstCopy <- lst
.Internal(inspect(lst))

## @0x00000000127170c8 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##   @0x0000000012717058 14 REALSXP g0c2 [] (len=2, tl=0) 1,2
##   @0x0000000012717090 14 REALSXP g0c2 [] (len=2, tl=0) 3,4

.Internal(inspect(lstCopy))

## @0x00000000127170c8 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##   @0x0000000012717058 14 REALSXP g0c2 [] (len=2, tl=0) 1,2
##   @0x0000000012717090 14 REALSXP g0c2 [] (len=2, tl=0) 3,4
```

After modification is made to one of the vectors in one of the lists, a copy of the entire list is made in a sense that the entire vector of addresses (or vector of pointers) that stored the two sub-vectores contained in the original list is copied. The sub-vector that is about to be modified is also copied in order to make the modification.

```
lst = list(c(1, 2),c(3, 4))
.Internal(inspect(lst))

## @0x0000000012549a60 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##   @0x00000000125499f0 14 REALSXP g0c2 [] (len=2, tl=0) 1,2
##   @0x0000000012549a28 14 REALSXP g0c2 [] (len=2, tl=0) 3,4

lst[[1]][1] <- 8
.Internal(inspect(lst))

## @0x0000000012920470 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
##   @0x00000000129204a8 14 REALSXP g0c2 [] (len=2, tl=0) 8,2
##   @0x0000000012549a28 14 REALSXP g0c2 [NAM(2)] (len=2, tl=0) 3,4
```

2(c) lstC is a copy of the original list.

```
lst1 <- list(1, 2)
lst2 <- list(3, 4)
lst <- list(lst1, lst2)
lstC <- lst
.Internal(inspect(lst))

## @0x000000001290e3d0 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##   @0x00000000125a9398 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##     @0x00000000184e6300 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 1
##     @0x00000000184e6480 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 2
##   @0x0000000012910ff0 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##     @0x00000000184e6750 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 3
##     @0x00000000184e2d58 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 4
```

```
.Internal(inspect(lstC))

## @0x000000001290e3d0 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##   @0x00000000125a9398 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##     @0x00000000184e6300 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 1
##     @0x00000000184e6480 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 2
##   @0x0000000012910ff0 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##     @0x00000000184e6750 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 3
##     @0x00000000184e2d58 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 4
```

After adding an element to lstC, the vectors that stored the addresses of the two original sublists are copied, but the address of each of the original sublist itself is not copied. The data in the sublists stored in the two uncopied addresses are shared.

```
lstC[[3]] <- list(5, 6)
.Internal(inspect(lst))

## @0x000000001290e3d0 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##   @0x00000000125a9398 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##     @0x00000000184e6300 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 1
##     @0x00000000184e6480 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 2
##   @0x0000000012910ff0 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##     @0x00000000184e6750 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 3
##     @0x00000000184e2d58 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 4

.Internal(inspect(lstC))

## @0x0000000013900b88 19 VECSXP g0c3 [NAM(1)] (len=3, tl=0)
##   @0x00000000125a9398 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##     @0x00000000184e6300 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 1
##     @0x00000000184e6480 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 2
##   @0x0000000012910ff0 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##     @0x00000000184e6750 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 3
##     @0x00000000184e2d58 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 4
##   @0x00000000139016e8 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
##     @0x0000000013656fc0 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 5
##     @0x0000000013656f90 14 REALSXP g0c1 [NAM(2)] (len=1, tl=0) 6
```

2(d) The result from object.size() show that the size of the object 'tmp' is 160 Mb, however results from gc() shows that only 80 Mb of memeory is being used. Since 'tmp' is the list which contains two elements. Both elements are 'x' so they point to the same vector of size 80 Mb. The result from object.size() shows the size of 'tmp' being 160 Mb since it contains two 'x', while gc() reports the amount of memory being used to be 80 Mb since what is using up the memory is just the object 'x' in that 'tmp' is built by 'x'.

```
gc()

##          used (Mb) gc trigger (Mb) max used (Mb)
## Ncells 557375 29.8     940480 50.3   940480 50.3
## Vcells 968746  7.4    1926384 14.7  1334232 10.2

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

```
## @0x000000001390ac48 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
##   @0x00007ff5facb0010 14 REALSXP g0c7 [NAM(2)] (len=10000000, tl=0) 0.968295,-1.00894,0.716222,-0.672995
##   @0x00007ff5facb0010 14 REALSXP g0c7 [NAM(2)] (len=10000000, tl=0) 0.968295,-1.00894,0.716222,-0.672995
```

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

```
## 160000136 bytes
```

```
gc()
```

```
##            used (Mb) gc trigger  (Mb) max used (Mb)
## Ncells   558111 29.9     940480  50.3   940480 50.3
## Vcells 10970602 83.7   16197569 123.6 11027686 84.2
```

3 In order to increase the efficiency in this chunk of codes, we replace the three nested for loops with a matrix multiplication which will do what is essentially the same thing. In addition, we pass in the result of the calculation of 'A*q[,,z]' to a variable so that this calculation will not be repeated twice in the second for loop.

```
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))
  # compress the three nested for loops to one for loop doing what is essentially
  # the matrix multiplication
  for (i in 1:K) {
    q[ , , i] <- theta.old[, i] %*% t(theta.old[, i]) / Theta.old
  }
  theta.new <- theta.old
  for (z in 1:K) {
    # create a variable B to prevent from calculating A*q[,,z] twice in every
    # for loop for K times
    B = A*q[,,z]
    theta.new[,z] <- rowSums(B)/sqrt(sum(B))
  }
  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)
```

```r
# time to execute the code
system.time(out <- oneUpdate(A, n, K, theta.init))

##    user  system elapsed
##    0.54    0.25    0.80
```

4. First Algorithm:

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

Use runif() to generate 2k numbers between 0 and 1. Multiply these 2k numbers with the length(x) to obtain 2k numbers ranging from 0 to the length(x). These 2k numbers now have decimal places, use round() to convert them to integers. Use unique() to filter and keep the unique numbers. This step makes sure the characteristic of without replacement. Select the first k numbers in the sequences of unique numbers we just obtain. Return these k numbers as sample without replacement.

```r
modPIKK <- function(x, k) {
  x[unique(round(runif(2*k)*length(x)))[1:k]]
}
```

Now we how how efficient is the modified algorithm comparing to the original algorithm as k and n vary. Here we use $k = \frac{n}{20}$ and $n = \{5000, 6000, 7000, 8000, 9000, 10000\}$.

```r
# different length of x
n <- c(5000, 6000, 7000, 8000, 9000, 10000)

# time for original PIKK
timePIKK <- c()

# time for modified PIKK
timePIKKmod <- c()

# obtain time for original PIKK
for (i in n) {
  x <- rnorm(i)
  k <- i/20
  timePIKK <- c(timePIKK, mean(microbenchmark(PIKK(x,k))$time))
}

# obtain time for modified PIKK
for (i in n) {
  x <- rnorm(i)
  k <- i/20
  timePIKKmod <- c(timePIKKmod, mean(microbenchmark(modPIKK(x,k))$time))
}

# report time of original algorithm takes in nanoseconds
# with n=(5000,6000,7000,8000,9000,10000), k=n/20
timePIKK

## [1] 586162.9 612326.9 672619.9 759508.0 856614.6 921727.3
```

```
# report time of modified algorithm takes in nanoseconds
# with n=(5000,6000,7000,8000,9000,10000), k=n/20
timePIKKmod

## [1]  93679.14  72811.04  99336.41  94249.79 108935.05 116329.00
```

Second Algorithm

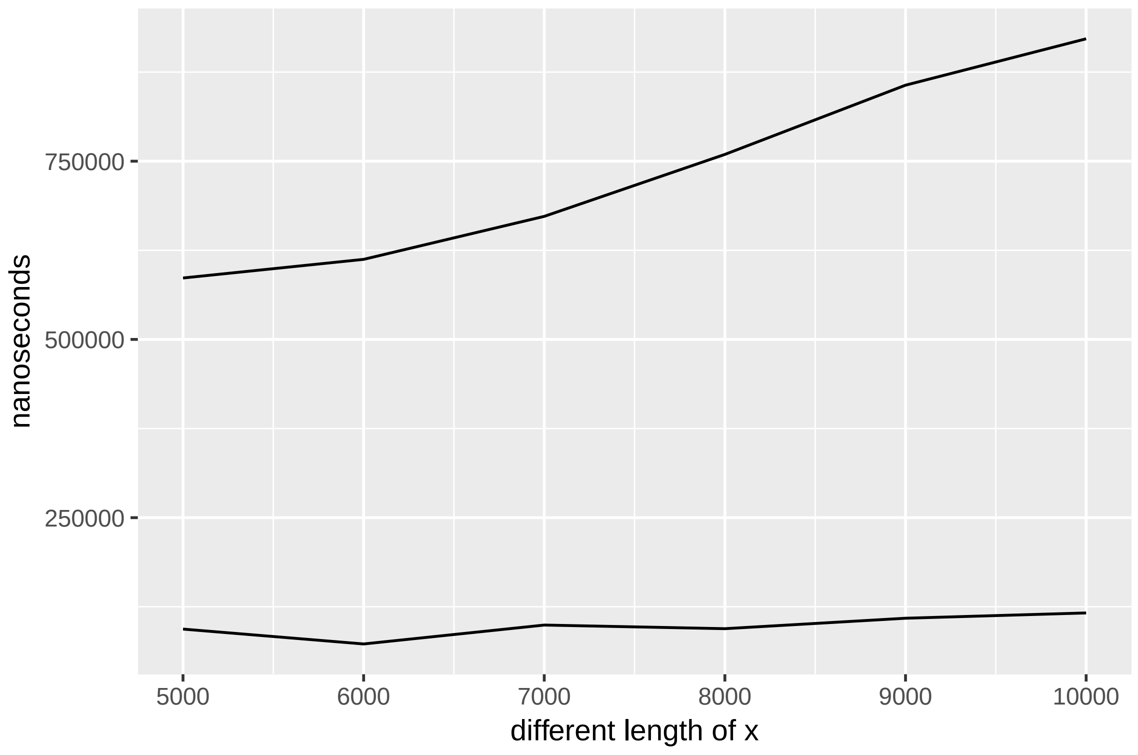
```
# original algorithm
FYKD <- function(x, k) {
  n <- length(x)
  for(i in 1:n) {
    j = sample(i:n, 1)
    tmp <- x[i]
    x[i] <- x[j]
    x[j] <- tmp
  }
  return(x[1:k])
}
```

Modify FYKD by only swapping the first k numbers in x, instead of swapping all numbers in x. Then select the first k numbers.

```
# modified FYKD
modFYKD <- function(x, k) {
  n <- length(x)
  # instead of 1:n, use 1:k
  for(i in 1:k) {
    j = sample(i:n, 1)
    tmp <- x[i]
    x[i] <- x[j]
    x[j] <- tmp
  }
  return(x[1:k])
}

# time for original FYKD
timeFYKD <- c()

# time for modified FYKD
timeFYKDmod <- c()

# obtain time for original FYKD
for (i in n) {
  x <- rnorm(i)
  k <- i/20
  timeFYKD <- c(timeFYKD, mean(microbenchmark(FYKD(x,k))$time))
}

# obtain time for modified FYKD
for (i in n) {
  x <- rnorm(i)
  k <- i/20
  timeFYKDmod <- c(timeFYKDmod, mean(microbenchmark(modFYKD(x,k))$time))
}

# report time of original FYKD algorithm takes in nanoseconds
# with n=(5000,6000,7000,8000,9000,10000), k=n/20
timeFYKD

## [1] 25319710 32114333 39847967 49853127 57730063 67152896

# report time of modified FYKD algorithm takes in nanoseconds
# with n=(5000,6000,7000,8000,9000,10000), k=n/20
timeFYKDmod

## [1] 1702981 2055268 2729377 3114125 3757013 4338618

# create a dataframe collecting all information
df <- data.frame(xLength = n, PIKKtime = timePIKK,
                 modPIKKtime = timePIKKmod, FYKDtime = timeFYKD,
                 modFYKDtime = timeFYKDmod)

# all time are in nanoseconds
df

##   xLength PIKKtime modPIKKtime FYKDtime modFYKDtime
## 1    5000 586162.9    93679.14 25319710     1702981
## 2    6000 612326.9    72811.04 32114333     2055268
## 3    7000 672619.9    99336.41 39847967     2729377
## 4    8000 759508.0    94249.79 49853127     3114125
## 5    9000 856614.6   108935.05 57730063     3757013
## 6   10000 921727.3   116329.00 67152896     4338618
```

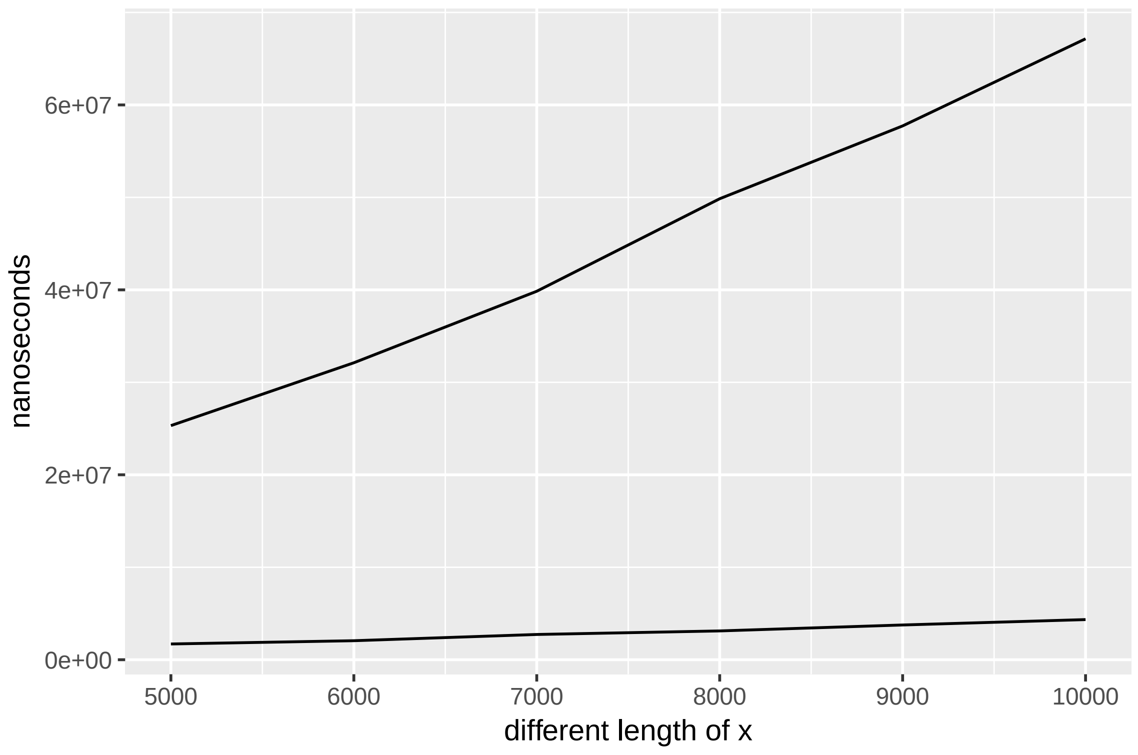
Plot the comparison of the PIKK algorithm as follows. Line on top is original algorithm, line on bottom is the modified algorithm.

```r
# plot comparison for first algorithm
g1 <- ggplot(df) +
  geom_line(aes(x = xLength, y = PIKKtime)) +
  geom_line(aes(x = xLength, y = modPIKKtime)) +
  xlab("different length of x") +
  ylab("nanoseconds")
g1
```



Plot the comparison of the FYKD algorithm as follows. Line on top is original algorithm, line on bottom is the modified algorithm.

```r
# plot comparison for second algorithm
g2 <- ggplot(df) +
  geom_line(aes(x = xLength, y = FYKDtime)) +
  geom_line(aes(x = xLength, y = modFYKDtime)) +
  xlab("different length of x") +
  ylab("nanoseconds")
g2
```


6e+07
4e+07
2e+07
0e+00
nanoseconds
different length of x