Problem Set 4

github : stephaniesookim

October 15, 2014

/raggedright

Number 1

Since the result we are trying to return is x+y (5th line), we get the value of x first. Also, we can observe that the value of y is defined inside the declaration of x as 1. So in this case, x+y = 2 + 1 = 3.

```
f1 <- function(x={y <- 1
                                    2}
                          , y=0) {
  x+y
}


f1()

## [1] 3
```

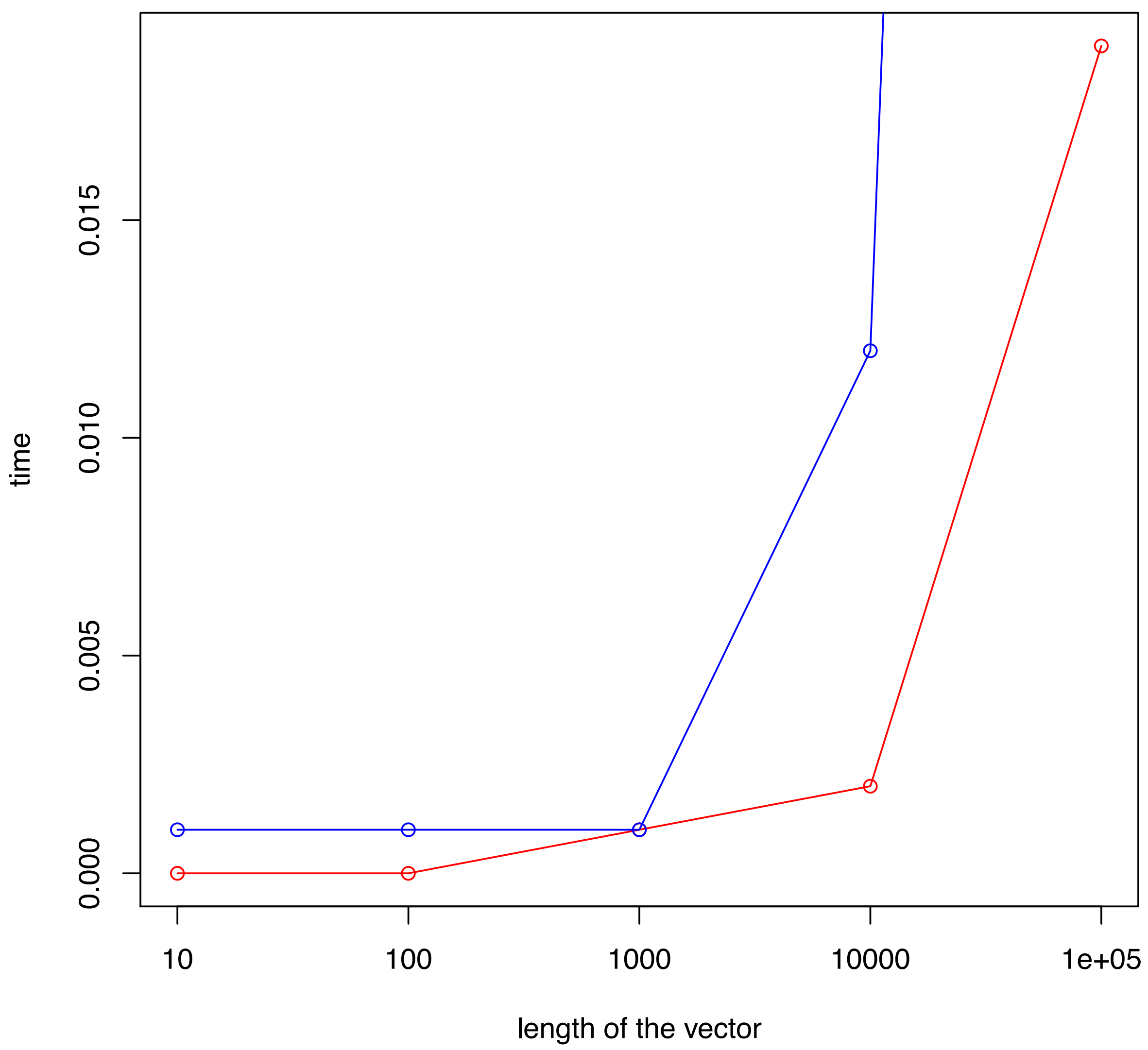
On the other hand in the below case, since the result we are trying to return is y+x, we get the value of y first. So in this case, y+x = 0 + 2 = 2

```
f2 <- function(x={y <- 1
                                    2}
                          , y=0) {
  y+x
}


f2()

## [1] 2
```

Number 2

```
library(rbenchmark)


# plotfun is a function that plots the elapsed time of subsetting method
# by vector of indices and vector of logical values
# by observing the plot, we can see which method takes shorter time
```

```r
plotfun <- function(n) {

        x <- indicestime <- booleanstime <- time(as.numeric(NA), n)

        # run a for loop for 1,2,..,n
        # we set the length of vector be 10^i
        # indices is a function that subsets a vector based on vector of indices
        # booleans is a function that subsets a vector based on vector of logical values (booleans)
        # indicestime[i] is the elapsed time of the 1st method
        # booleanstime[i] is the elapsed time of the 2nd method

        for (i in 1:n) {
                x[i] <- y <- 10^i

        indices <- function(y) {
        ind <- c(1:y)
        ind[c(1:floor(y^1/2))]
        }

                booleans <- function(y) {
                  bool <- c(1:y)
                  subset(bool, bool <= floor(y^1/2))
        }

                time <- benchmark(indices(y), booleans(y), replications = 10,
                                                      columns=c('test', 'elapsed', 'replications'))

        indicestime[i] <- time[1,2]
        booleanstime[i] <- time[2,2]
        }

        # we plot elapsed time changing over the length of a vector for each method
        plot(booleanstime,  type="o", col="red", xlab="length of the vector",
                ylab="time", xaxt='n')
        points(indicestime,  type="o", col="blue")
        axis(1, at=1:length(x), labels=x)

        # then we return the plots
        return(list(t1=booleanstime, t2=indicestime))
}

# let's print out the plot when n=5
plotfun(5)
```



```
## $t1
## [1] 0.000 0.000 0.001 0.002 0.019
## attr(,"tsp")
## [1] 1 1 1
##
## $t2
## [1] 0.001 0.001 0.001 0.012 0.149
## attr(,"tsp")
## [1] 1 1 1

# from the plot, we can see that subsetting by indices is more efficient than subsetting by booleans
# this is because the 1st method only goes through 1,2,..,floor(y^1/2) elements
# while the 2nd method goes through the whole vector
# the gap will be greater as the length of a vector gets longer
```

Number 3 (a)

```
# we use .Internal(inspect(dat)); gc()
# to inspect into the structure of the data frame
# and where in memory different parts of the data frame are stored
# let's compare how it changes before and after we change one element

dat <- read.csv("cpds.csv", head = TRUE, sep = ",")
.Internal(inspect(dat)); gc()

datcopy[1,1] <- 2000
.Internal(inspect(datcopy)); gc()

# the entire data frame is not copied
# the element is simply replaced without the entire column being copied
# the memory address of the changed column change with others unchanged
```

Number 3 (b)

```
v1 <- 1:10
v2 <- 1:10
v3 <- 1:10

vlist <- list(v1, v2, v3)
.Internal(inspect(vlist)); gc()

## @7fd33dccae88 19 VECSXP g0c3 [NAM(2)] (len=3, tl=0)
##   @7fd33da6ecf0 13 INTSXP g0c4 [NAM(2)] (len=10, tl=0) 1,2,3,4,5,...
##   @7fd33d976878 13 INTSXP g0c4 [NAM(2)] (len=10, tl=0) 1,2,3,4,5,...
##   @7fd33c0ae288 13 INTSXP g0c4 [NAM(2)] (len=10, tl=0) 1,2,3,4,5,...
##          used (Mb) gc trigger (Mb) max used (Mb)
## Ncells 160557  8.6     350000 18.7   350000 18.7
## Vcells 362061  2.8    1031040  7.9  1031040  7.9

vlist[[1]][1] <- 10
.Internal(inspect(vlist)); gc()

## @7fd33dccad68 19 VECSXP g0c3 [NAM(1)] (len=3, tl=0)
##   @7fd33dbedaa8 14 REALSXP g0c5 [] (len=10, tl=0) 10,2,3,4,5,...
##   @7fd33d976878 13 INTSXP g1c4 [MARK,NAM(2)] (len=10, tl=0) 1,2,3,4,5,...
##   @7fd33c0ae288 13 INTSXP g1c4 [MARK,NAM(2)] (len=10, tl=0) 1,2,3,4,5,...
##          used (Mb) gc trigger (Mb) max used (Mb)
## Ncells 160583  8.6     350000 18.7   350000 18.7
## Vcells 362143  2.8    1031040  7.9  1031040  7.9

# the entire list is not copied
# the element is simply replaced without the entire list being copied
# the memory address of the changed vector (v1) change with others unchanged
```

Number 4 (a), (c)

```r
# randomwalk is a function that shows the (x,y) coordinate of n random walk
# it will print the whole path only when Full=T

randomwalk <- function(n, Full = F){

    #for no step
        if (n==0){
        paste0("no step: (0,0)")
        }

        #for negative input
        else if (n<0){
        paste0("invalid (negative) input argument")
        }

        else{

        #cast as integer if n is not integer
        n = as.integer(n)

        # gostop = vector of T/F
        # randomly generate n-1 number for 0 to 1 (uniform) if greater than 0.5 True or False
        gostop = runif(n, min = 0, max = 1) > 0.5

        # dir = vector of 1/-1
        # T = 1, F = 0, multiply by 2 - 1 become 1 and -1
        dir = 2*(runif(n, min = 0, max = 1) > 0.5)-1

        # ifelse (gostop,dir,0)
        # for every nth element of list gostop of T/F
        # if T => nth element of dir
        # if F => nth element of 0
        # cumsum = cumulative sum
        x = c(0,cumsum(ifelse(gostop,dir,0)))
        y = c(0,cumsum(ifelse(gostop,0, dir)))

        #print Full
        if (Full == TRUE) {
                printx <- function(x,n) {
                        char = (paste0("(x,y) coordinate : (", x[n]))
                }
                printy <- function(y,n) {
                        char = (paste0(",", y[n], ")"))
                }
                pathx <- sapply(x,printx)
                pathy <- sapply(y,printy)
                print(paste0(pathx, pathy))
                }

        #print result
        else if (Full == FALSE) {
                char = (paste0("(x,y) coordinate : (", x[n+1], ",", y[n+1], ")"))
                print(char)
                }
        }
}

#some examples
randomwalk(0)

## [1] "no step: (0,0)"

randomwalk(-3.9)

## [1] "invalid (negative) input argument"

randomwalk(1,Full=T)

## [1] "(x,y) coordinate : (0,0)"  "(x,y) coordinate : (-1,0)"

randomwalk(1)

## [1] "(x,y) coordinate : (1,0)"

randomwalk(4.9, Full=T)

## [1] "(x,y) coordinate : (0,0)"  "(x,y) coordinate : (0,-1)"
## [3] "(x,y) coordinate : (1,-1)" "(x,y) coordinate : (1,0)"
## [5] "(x,y) coordinate : (1,-1)"

randomwalk(5,Full=T)

## [1] "(x,y) coordinate : (0,0)"  "(x,y) coordinate : (0,1)"
## [3] "(x,y) coordinate : (-1,1)" "(x,y) coordinate : (-2,1)"
## [5] "(x,y) coordinate : (-3,1)" "(x,y) coordinate : (-3,2)"

randomwalk(5)

## [1] "(x,y) coordinate : (-2,-1)"
```

Number 4 (b)

```r
## formal s4 class OOP
library(methods)
setClass("rw",representation(x = "vector", y = "vector", n = "numeric"))

# constructor
# this is function that generates the class, rw
random <- function(x, y, n) {
  n = as.integer(n)
  gostop = runif(n, min = 0, max = 1) > 0.5
  dir = 2*(runif(n, min = 0, max = 1) > 0.5) - 1
  x = c(0,cumsum(ifelse(gostop,dir,0)))
  y = c(0,cumsum(ifelse(gostop,0, dir)))
  new("rw", x = x, y = y, n = n)
}
```

```
# print method
print <- function(object) {
  n = as.integer(object@n)
  paste0("final (x,y) coordinate : (", (object@x)[n+1], "," , (object@y)[n+1], ")")
}


# operator
"|" <- function(object, i) {
  paste0("position at ", i ,"th step : (", object@x[i], ",", object@y[i], ")")
}

# start method
start <- function(object, n1,n2) {
  n = as.integer(object@n)
  object@x <- object@x + n1
  object@y <- object@y + n2
  return(object)
}

# some examples
rw1 <- random(x,y,5)
print(rw1)

## [1] "final (x,y) coordinate : (-2,1)"

rw1 <- start(rw1,3,1)
print(rw1)

## [1] "final (x,y) coordinate : (1,2)"
```

Number 5 (a)

In this question, we use gc() function to observe how the memory usage has changed.

```
install.packages("inline")

## Installing package into '/Users/stephaniekim/Library/R/3.1/library'
## (as 'lib' is unspecified)
## Error:  trying to use CRAN without setting a mirror

library(inline)

# this code is simply a placeholder to demonstrate that I can
# modify the input arguments as desired in C;
# in reality 'src' would contain substantive computations

src <- "\n tablex[0] = 7;\n"

dummyFun <- cfunction(signature(tablex = "integer", tabley = "integer", xvar = "integer", yvar = "integ

n<-1e7
gc()
```

```
##          used (Mb) gc trigger (Mb) max used (Mb)
## Ncells 365472 19.6     531268 28.4   467875 25.0
## Vcells 660539  5.1    1300721 10.0  1031040  7.9

# Used Mb = 5.1

xvar1<-sample(c(seq(1,20,by=1),NA),n,replace=TRUE)
gc()

##            used (Mb) gc trigger  (Mb) max used  (Mb)
## Ncells   366019 19.6     667722  35.7   467875  25.0
## Vcells 10661521 81.4   21939752 167.4 20672994 157.8

# Used Mb = 81.4
# Used Mb increases by 76.3 (approximately 80) after xvar1 is created
# integers take 4 btyes per value
# 4*20=80

yvar1<-sample(c(seq(1,20,by=1),NA),n,replace=TRUE)
gc()

##            used  (Mb) gc trigger  (Mb) max used  (Mb)
## Ncells   366065  19.6     667722  35.7   467875  25.0
## Vcells 20661571 157.7   32439805 247.5 30673044 234.1

# Used Mb = 157.7
# Used Mb increases by 76.3 (approximately 80) after yvar1 is created
# integers take 4 bytes per value
# 4*20=80


# fastcount is a function that prints out a list called result
fastcount <- function(xvar,yvar) {
  nalineX <- is.na(xvar)
  gc1 <- gc()
  nalineY <- is.na(yvar)
  gc2 <- gc()
  xvar[nalineX | nalineY] <- 0
  gc3 <- gc()
  yvar[nalineX | nalineY] <- 0
  gc4 <- gc()
  useline <- !(nalineX | nalineY);
  gc5 <- gc()
  tablex <- numeric(max(xvar)+1)
  gc6 <- gc()
  tabley <- numeric(max(yvar)+1)
  gc7 <- gc()
  stopifnot(length(xvar) == length(yvar))
  gc8 <- gc()
 res <- dummyFun(     tablex = as.integer(tablex), tabley = as.integer(tabley),     as.integer(xvar), a
  gc9 <- gc()
  xuse <- which(res$tablex > 0)
  gc10 <- gc()
  xnames <- xuse - 1
  gc11 <- gc()
  resb <- rbind(res$tablex[xuse], res$tabley[xuse])
  gc12 <- gc()
  colnames(resb) <- xnames
  gc13 <- gc()
  result <- list(gc1, gc2, gc3, gc4, gc5, gc6, gc7, gc8, gc9, gc10, gc11, gc12, gc13,resb)
  result
}


fastcount(xvar1,yvar1)
```

```
## Error:  trying to get slot "x" from an object of a basic class ("logical") with no slots
```

```
# 1st observation:
# Used Mb increases by approximately 40 after nalineX and nalineY are created
# this is because booleans take 4 bytes per value

# 2nd observation:
# Used Mb increases by approximately 80
# after xvar[nalineX | nalineY] and yvar[nalineX | nalineY] are created
# this is because the xvar and yvar are copied (used memory doubles)

# 3rd observation:
# Used Mb reaches its maximum after res is created


gc()
```

```
##            used  (Mb) gc trigger  (Mb) max used  (Mb)
## Ncells   366604  19.6     667722  35.7   467875  25.0
## Vcells 20661914 157.7   42940107 327.7 40675364 310.4
```

```
# Used Mb returns back to the amount before the function
# when the function is over
```

Number 5 (b)

```
# do not create nalineX and nalineY, use is.na instead

fastcount2 <- function(xvar,yvar) {
  xvar[is.na(xvar) | is.na(yvar)] <- 0
  gc1 <- gc()
  yvar[is.na(xvar) | is.na(yvar)] <- 0
  gc2 <- gc()
  useline <- !(is.na(xvar) | is.na(yvar))
  gc3 <- gc()
  tablex <- numeric(max(xvar)+1)
  gc4 <- gc()
  tabley <- numeric(max(yvar)+1)
  gc5 <- gc()
  stopifnot(length(xvar) == length(yvar))
  gc6 <- gc()
  res <- dummyFun(
    tablex = as.integer(tablex), tabley = as.integer(tabley),     as.integer(xvar), as.integer(yvar), a
  gc7 <- gc()
  xuse <- which(res$tablex > 0)
  gc8 <- gc()
  xnames <- xuse - 1
  gc9 <- gc()
  resb <- rbind(res$tablex[xuse], res$tabley[xuse])
  gc10 <- gc()
  colnames(resb) <- xnames
  gc11 <- gc()
  result2 <- list(gc1, gc2, gc3, gc4, gc5, gc6, gc7, gc8, gc9, gc10, gc11,resb)
  result2
}
```

Number 6 (original code)

```
load('~/stat243-fall-2014/ps/ps4prob6.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)
```

```r
  return(list(theta = theta.new, loglik = L.new,
              converged = converge.check))
}

temp <- matrix(runif(n*K), n, K)
theta.init <- temp/rowSums(temp)

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

The original code takes 132 seconds.

Number 6

I improved the code by fixing the three nested for loops.

```r
load('~/stat243-fall-2014/ps/ps4prob6.Rda') # should have A, n, K
ll <- function(Theta, A) {
  logLik <- sum(log(Theta[which(A==1, arr.ind=T)])) - 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) { q[i,,] <- t(theta.old[i, ]*t(theta.old))/Theta.old[i, ]}
  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))
}

set.seed(1)
temp <- matrix(runif(n*K), n, K)
set.seed(1)
theta.init <- temp/rowSums(temp)

system.time(out <- oneUpdate(A, n, K, theta.init))

##    user  system elapsed
##   3.936   0.616   4.592
```

Now the updated code takes approximately 4 seconds