STAT 243: Problem Set 4

Jin Rou New

October 10, 2014

1 Problem 1

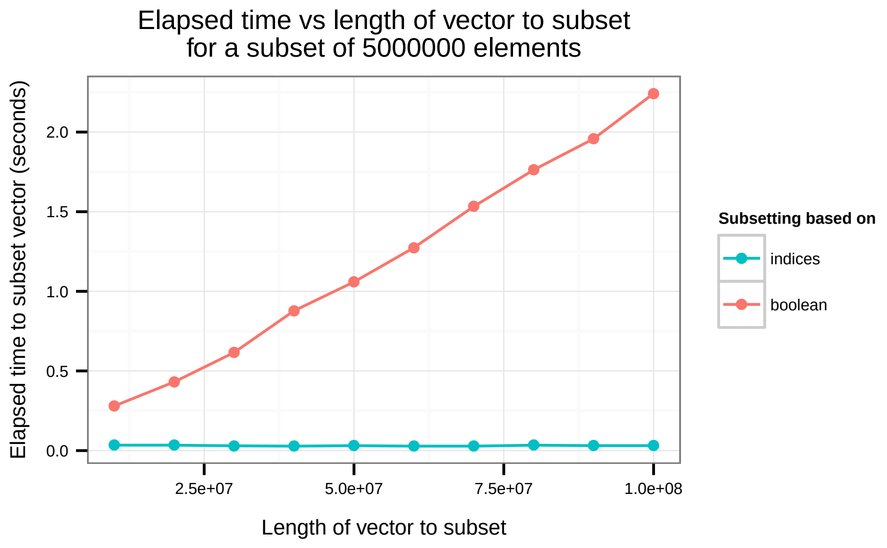
The value of `x` is 2 and that of `y` is 1. This is due to lazy evaluation, where `y = 0` is not evaluated until it is needed, and here it is not needed since `x` is first evaluated, during which `y` is assigned a value of 1.
When the function is modified such that we have `y + x` instead of `x + y` in the body of the function, the value of `y` is 0 (since `y = 0` is evaluated first) and the value of `x` is again 2.

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

## [1] 3

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

## [1] 2
```

2 Problem 2

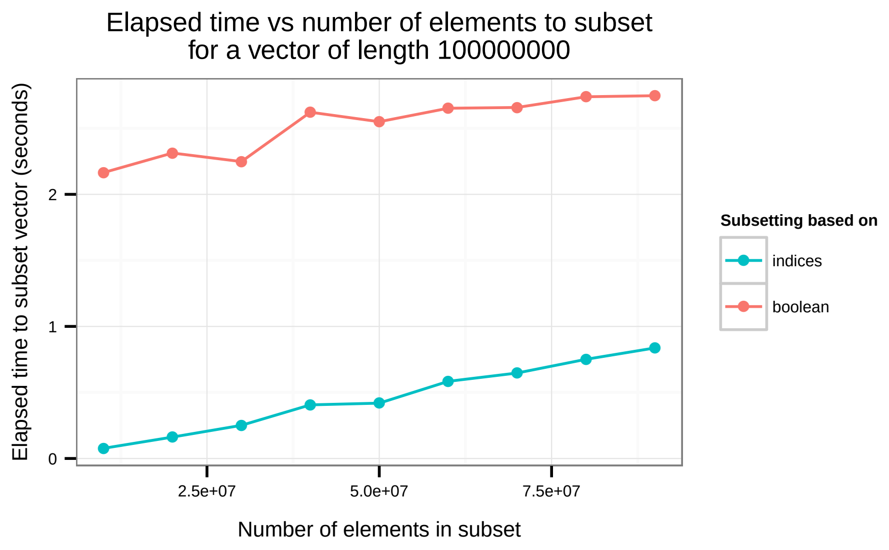
It is more efficient to subset a vector based on a vector of indices rather than booleans. When subsetting by a vector of indices, it is immediately clear which elements need to be extracted. On the other hand, when subsetting by a vector of booleans, it is first necessary to determine the indices of the vector for which elements need to be extracted, which is an additional step.
The timing when subsetting by indices changes little when the length of the vector increases, while the timing when subsetting by booleans increases with order n.

```
CompareTime <- function(length_vector, num_select) {
  test_vector <- 1:length_vector
  indices_vector <- 1:num_select
  boolean_vector <- c(rep(TRUE, num_select), rep(FALSE, length_vector-num_select))
  time_indices <- system.time(test_vector[indices_vector])["elapsed"]
  time_boolean <- system.time(test_vector[boolean_vector])["elapsed"]
  return(c(time_indices = time_indices, time_boolean = time_boolean))
}
library(ggplot2)
# Examine how length of vector to subset affects timing
lengths <- c(seq(1, 9)*10000000, 100000000)
times_list <- lapply(lengths, CompareTime, num_select = 5000000)
theme_set(theme_bw(base_size = 8) +
            theme(plot.title = element_text(vjust = 1),
                  axis.title.x = element_text(vjust = -0.3),
                  axis.title.y = element_text(vjust = 1.1)))
p <- ggplot(data.frame(length = rep(lengths, 2),
                  type = rep(c("indices", "boolean"), each = length(lengths)),
                  system_time = c(sapply(times_list, "[[", 1), sapply(times_list, "[[", 2))),
            aes_string(x = "length", y = "system_time", colour = "type")) +
  geom_line() + geom_point() +
  ggtitle("Elapsed time vs length of vector to subset\nfor a subset of 5000000 elements") +
  xlab("Length of vector to subset") + ylab("Elapsed time to subset vector (seconds)") +
  scale_colour_discrete(name = "Subsetting based on",
                        breaks = c("indices", "boolean"),
                        labels = c("indices", "boolean"))
print(p)
```



Other than the length of the vector, an increase in the number of elements in the subset also increases the time taken to subset with order n.

```
# Examine how number of elements in subset affects timing
nums_select <- c(seq(1, 9)*10000000)
times_list2 <- lapply(nums_select, CompareTime, length_vector = 100000000)
theme_set(theme_bw(base_size = 8) +
            theme(plot.title = element_text(vjust = 1),
                  axis.title.x = element_text(vjust = -0.3),
                  axis.title.y = element_text(vjust = 1.1)))
p2 <- ggplot(data.frame(num_select = rep(nums_select, 2),
                        type = rep(c("indices", "boolean"), each = length(nums_select)),
                        system_time = c(sapply(times_list2, "[[", 1), sapply(times_list2, "[[", 2))),
             aes_string(x = "num_select", y = "system_time", colour = "type")) +
  geom_line() + geom_point() +
  ggtitle("Elapsed time vs number of elements to subset\nfor a vector of length 100000000") +
  xlab("Number of elements in subset") + ylab("Elapsed time to subset vector (seconds)") +
  scale_colour_discrete(name = "Subsetting based on",
                        breaks = c("indices", "boolean"),
                        labels = c("indices", "boolean"))
print(p2)
```



3 Problem 3

(a) Memory addresses of all components of the data frame except for that changed column remain the same, which means that the column is copied to a new memory address.

```
data_file <- "data/cpds.csv"
data <- read.csv(data_file)
.Internal(inspect(data))
data[1, colnames(data) == "year"] <- data[1, colnames(data) == "year"] + 1
.Internal(inspect(data))
```

(b) For a list, only the changed vector is copied.

```
data <- read.csv(data_file)
data_list <- as.list(data)
.Internal(inspect(data_list))
data_list[["year"]][1] <- data_list[["year"]][1] + 1
.Internal(inspect(data_list))
```

4 Problem 4

The possible values that can be taken on each axis at each time step is 0 (with probability 0.5), -1 and 1 (with probability 0.25 each). Hence I created two vectors `units_x` and `units_y`, both of length 4 with these values (with one repeated 0) and made sure that for the same index, when `units_x` is nonzero, `units_y` is zero and vice versa. I then sampled the indices 1 to 4 with replacement for a user-defined number of steps `num_steps` to get the step taken at each time step, then used `cumsum` to obtain the full random walk.

```r
# Setting an rw class
setClass("rw",
         slots = c(x = "numeric", y = "numeric", num_steps = "numeric"),
         prototype = list(x = numeric(), y = numeric(), num_steps = numeric()),
         validity = function(object) {
           if (length(object@num_steps) != 1)
             return("Error: num_steps must be a scalar.")
           if (object@num_steps <= 0)
             stop("num_steps has to be a positive integer.")
           if (object@num_steps %% 1 != 0)
             stop("num_steps has to be an integer.")
           if (!(all(length(object@x) == length(object@y), length(object@x) == (object@num_steps + 1)))
             return("Error: Lengths of x and y must be equal to num_steps+1.")
           return(TRUE)
         })
#' Generate a 2D random walk.
#'
#' Generate a discrete random walk in two dimensions with uniform probability of moving up, down,
#' left, right by 1 unit at each step (+/- 1 on the x- or y-axis).
#'
#' @param num_steps Number of steps of the random walk to generate.
#' @param full_walk \code{TRUE} return full random walk, \code{FALSE} to return final position.
#' @return A list object containing:
#' \describe{
#'   \item{\code{x}}{Integer vector of length \code{num_steps+1} of x-coordinates of the random walk
#'   if \code{full_walk} is TRUE and of length \code{1} of final x-coordinate if \code{full_walk} is FA
#'   \item{\code{y}}{Integer vector of length \code{num_steps+1} of y-coordinates of the random walk
#'   if \code{full_walk} is TRUE and of length \code{1} of final y-coordinate if \code{full_walk} is FA
#' }
Generate2DRandomWalk <- function(num_steps, full_walk = TRUE) {
  # Return error if num_steps is not a scalar, is negative or zero and warning if it is not an integer
  if (length(num_steps) != 1)
    stop("num_steps must be a scalar.")
  if (num_steps <= 0)
    stop("num_steps has to be a positive integer.")
  if (num_steps %% 1 != 0) {
    num_steps <- as.integer(num_steps)
    warning("num_steps has been converted to an integer.")
  }
  # Return error if full_walk is not logical
  if (!is.logical(full_walk))
    stop("full_walk has to be TRUE or FALSE")
  # Do random walk
  units_x <- c(0, 0, -1, 1)
  units_y <- c(-1, 1, 0, 0)
  step_indices <- sample(1:4, num_steps, replace = TRUE)
  steps_x <- units_x[step_indices]
  steps_y <- units_y[step_indices]
  x <- c(0, cumsum(steps_x))
  y <- c(0, cumsum(steps_y))
  if (full_walk) {
    rw <- new("rw", x = x, y = y, num_steps = num_steps)
  } else {
    rw <- list(x = tail(x, n = 1), y = tail(y, n = 1), num_steps = num_steps)
  }
  return(rw)
}

# Method print
print.rw <- function(x) {
  cat(paste0("Final x-coordinate: ", tail(x@x, n = 1), "\n",
             "Final y-coordinate: ", tail(x@y, n = 1), "\n",
             "Maximum distance moved along x-axis: ",
             max(x@x) - min(x@x), "\n",
             "Maximum distance moved along y-axis: ",
             max(x@y) - min(x@y), "\n"))
}
setMethod("print", signature(x = "rw"), definition = print.rw)
```

```
## Creating a generic function for 'print' from package 'base' in the global environment
```

```
## [1] "print"
```

```r
# Method plot
plot.rw <- function(x, y) {
  theme_set(theme_bw(base_size = 10) +
            theme(plot.title = element_text(size = 10, vjust = 1),
                  axis.title.x = element_text(vjust = -0.3),
                  axis.title.y = element_text(vjust = 1.1)))
  p <- ggplot(data.frame(x = x@x, y = x@y),
              aes(x = x, y = y)) + geom_path(alpha = 0.5) +
    geom_hline(yintercept = 0) + geom_vline(xintercept = 0) +
    geom_point(x = x@x[1], y = x@y[1]) + # starting position
    geom_point(x = x@x[x@num_steps+1], y = x@y[x@num_steps+1], colour = "red") + # ending position
    ggtitle(paste0("2D random walk of \nlength ", x@num_steps)) +
    xlim(range(x@x)) + ylim(range(x@y)) +
    coord_fixed()
  print(p)
}
setMethod("plot", signature(x = "rw", y = "missing"), definition = plot.rw)
```

```
## Creating a generic function for 'plot' from package 'graphics' in the global environment
```

```
## [1] "plot"
```

```r
# Method [
setMethod("[", signature(x = "rw"),
          function(x, i) {
            if (i < 0 | i > x@num_steps)
              stop("i must be between 0 and num_steps (inclusive).")
            return(list(x = x@x[i+1], y = x@y[i+1]))
          })
```

```
## [1] "["
```

```r
# Method start
setGeneric("start<-", function(x, ...) {
  standardGeneric("start<-")
})
```

```
## [1] "start<-"
```

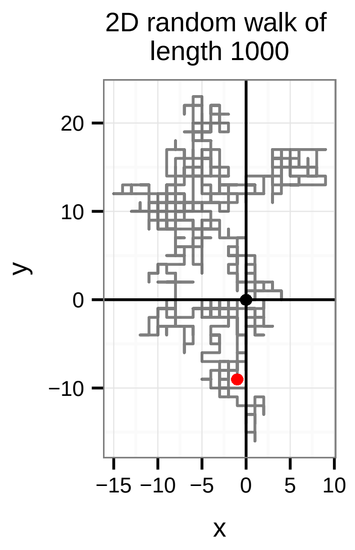
```
`start.rw<-` <- function(x, value) {
  x@x <- x@x + value[1]
  x@y <- x@y + value[2]
  return(x)
}
setReplaceMethod("start", signature(x = "rw"),
                 definition = `start.rw<-`)
```

```
## [1] "start<-"
```

```
# Test function and class-specific methods
rw_object <- Generate2DRandomWalk(num_steps = 1000)
# Call class-specific methods
print(rw_object)
```

```
## Final x-coordinate: -1
## Final y-coordinate: -9
## Maximum distance moved along x-axis: 24
## Maximum distance moved along y-axis: 39
```

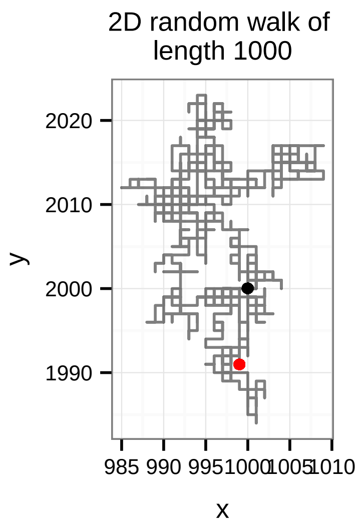
```
plot(rw_object)
```



```
rw_object[rw_object@num_steps]
```

```
## $x
## [1] -1
##
## $y
## [1] -9
```

```
start(rw_object) <- c(1000, 2000)
plot(rw_object)
```



5 Problem 5

(a) I inserted calls to `gc` and `object.size` to check memory usage at different points in the function.

```
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 <- '
tablex[0] = 7;
'

dummyFun <- cfunction(signature(tablex = "integer", tabley = "integer",
                               xvar = "integer", yvar = "integer", useline = "integer",
                               n = "integer"), src, convention = ".C")

fastcount <- function(xvar,yvar) {
  print(gc())
  print(paste("Object size of xvar:", object.size(xvar)))
  print(paste("Object size of yvar:", object.size(yvar)))
  nalineX <- is.na(xvar)
  print("nalineX <- is.na(xvar)")
  print(gc())
  print(paste("Object size of nalineX:", object.size(nalineX)))
  nalineY <- is.na(yvar)
  print("nalineY <- is.na(yvar)")
  print(gc())
  print(paste("Object size of nalineY:", object.size(nalineY)))
  xvar[nalineX | nalineY] <- 0
  print("xvar[nalineX | nalineY] <- 0")
  print(gc())
  print(paste("Object size of nalineX | nalineY:", object.size(nalineX | nalineY)))
  print(paste("Object size of xvar[nalineX | nalineY]:", object.size(xvar[nalineX | nalineY])))
  yvar[nalineX | nalineY] <- 0
  print("yvar[nalineX | nalineY] <- 0")
  print(gc())
  print(paste("Object size of yvar[nalineX | nalineY]:", object.size(yvar[nalineX | nalineY])))
  useline <- !(nalineX | nalineY)
  print("useline <- !(nalineX | nalineY)")
  print(gc())
  print(paste("Object size of useline:", object.size(useline)))
  tablex <- numeric(max(xvar)+1)
  print("tablex <- numeric(max(xvar)+1)")
  print(gc())
  print(paste("Object size of tablex:", object.size(tablex)))
  tabley <- numeric(max(yvar)+1)
  print("tabley <- numeric(max(yvar)+1)")
  print(gc())
  print(paste("Object size of tabley:", object.size(tabley)))
  stopifnot(length(xvar) == length(yvar))
  res <- dummyFun(
    tablex = as.integer(tablex), tabley = as.integer(tabley),
    as.integer(xvar), as.integer(yvar), as.integer(useline),
    as.integer(length(xvar)))
  print("res <- dummyFun()")
  print(gc())
  print(paste("Object size of as.integer(tablex):", object.size(as.integer(tablex))))
  print(paste("Object size of as.integer(tabley):", object.size(as.integer(tabley))))
  print(paste("Object size of as.integer(xvar):", object.size(as.integer(xvar))))
  print(paste("Object size of as.integer(yvar):", object.size(as.integer(yvar))))
  print(paste("Object size of as.integer(useline):", object.size(as.integer(useline))))
  xuse <- which(res$tablex > 0)
  print("xuse <- which(res$tablex > 0)")
  print(gc())
  print(paste("Object size of res$tablex > 0:", object.size(res$tablex > 0)))
  print(paste("Object size of xuse:", object.size(xuse)))
  xnames <- xuse - 1
  print("xnames <- xuse - 1")
  print(gc())
  print(paste("Object size of xnames:", object.size(xnames)))
  resb <- rbind(res$tablex[xuse], res$tabley[xuse])
  print("resb <- rbind(res$tablex[xuse], res$tabley[xuse])")
  print(gc())
  print(paste("Object size of res$tablex[xuse]:", object.size(res$tablex[xuse])))
  print(paste("Object size of res$tabley[xuse]:", object.size(res$tabley[xuse])))
  print(paste("Object size of resb:", object.size(resb)))
  colnames(resb) <- xnames
  print("colnames(resb) <- xnames")
  print(gc())
  return(resb)
}

n <- 1e7
set.seed(1)
xvar <- sample(c(seq(1, 20, by = 1), NA), n, replace = TRUE)
yvar <- sample(c(seq(1, 20, by = 1), NA), n, replace = TRUE)
resb <- fastcount(xvar, yvar)
```

The amount of memory used as its maximum is 653.4 Mb at the end of the code. From the output, it is clear that all the objects of similar size to `xvar` and `yvar` are large.

```
used  (Mb) gc trigger  (Mb) max used  (Mb)
Ncells   362856  19.4     667722  35.7   467875  25.0
Vcells 20630517 157.4   32407041 247.3 30632195 233.8
[1] "Object size of xvar: 80000040"
[1] "Object size of yvar: 80000040"
[1] "nalineX <- is.na(xvar)"
used  (Mb) gc trigger  (Mb) max used  (Mb)
Ncells   362948  19.4     667722  35.7   467875  25.0
Vcells 25630642 195.6   34107393 260.3 30632195 233.8
[1] "Object size of nalineX: 40000040"
[1] "nalineY <- is.na(yvar)"
used  (Mb) gc trigger  (Mb) max used  (Mb)
Ncells   362950  19.4     667722  35.7   467875  25.0
Vcells 30630642 233.7   35892762 273.9 30632195 233.8
[1] "Object size of nalineY: 40000040"
[1] "xvar[nalineX | nalineY] <- 0"
used  (Mb) gc trigger  (Mb) max used  (Mb)
Ncells   362953  19.4     667722  35.7   467875  25.0
Vcells 40630642 310.0   56157671 428.5 51095785 389.9
[1] "Object size of nalineX | nalineY: 40000040"
[1] "Object size of xvar[nalineX | nalineY]: 7442152"
[1] "yvar[nalineX | nalineY] <- 0"
used  (Mb) gc trigger  (Mb) max used  (Mb)
Ncells   362954  19.4     667722  35.7   467875  25.0
Vcells 50630642 386.3   67182671 512.6 61095785 466.2
[1] "Object size of yvar[nalineX | nalineY]: 7442152"
[1] "useline <- !(nalineX | nalineY)"
used  (Mb) gc trigger  (Mb) max used  (Mb)
Ncells   362957  19.4     667722  35.7   467875  25.0
Vcells 55630642 424.5   70621804 538.9 67026247 511.4
[1] "Object size of useline: 40000040"
[1] "tablex <- numeric(max(xvar)+1)"
used  (Mb) gc trigger  (Mb) max used  (Mb)
Ncells   362962  19.4     667722  35.7   467875  25.0
Vcells 55630663 424.5   74232894 566.4 67026247 511.4
[1] "Object size of tablex: 208"
[1] "tabley <- numeric(max(yvar)+1)"
used  (Mb) gc trigger  (Mb) max used  (Mb)
Ncells   362964  19.4     667722  35.7   467875  25.0
Vcells 55630684 424.5   78024538 595.3 67026247 511.4
[1] "Object size of tabley: 208"
[1] "res <- dummyFun()"
used  (Mb) gc trigger  (Mb) max used  (Mb)
Ncells   362990  19.4     667722  35.7   467875  25.0
Vcells 70630738 538.9   94745321 722.9 85630790 653.4
[1] "Object size of as.integer(tablex): 168"
[1] "Object size of as.integer(tabley): 168"
[1] "Object size of as.integer(xvar): 40000040"
[1] "Object size of as.integer(yvar): 40000040"
[1] "Object size of as.integer(useline): 40000040"
[1] "xuse <- which(res$tablex > 0)"
used  (Mb) gc trigger  (Mb) max used  (Mb)
```

```
Ncells   362994  19.4     667722  35.7   467875  25.0
Vcells 70630739 538.9   99562587 759.7 85631439 653.4
[1] "Object size of res$tablex > 0: 168"
[1] "Object size of xuse: 48"
[1] "xnames <- xuse - 1"
used  (Mb) gc trigger  (Mb) max used  (Mb)
Ncells   362997  19.4     667722  35.7   467875  25.0
Vcells 70630740 538.9  104620716 798.2 85631439 653.4
[1] "Object size of xnames: 48"
[1] "resb <- rbind(res$tablex[xuse], res$tabley[xuse])"
used  (Mb) gc trigger  (Mb) max used  (Mb)
Ncells   363002  19.4     667722  35.7   467875  25.0
Vcells 70630742 538.9  104620716 798.2 85631439 653.4
[1] "Object size of res$tablex[xuse]: 48"
[1] "Object size of res$tabley[xuse]: 48"
[1] "Object size of resb: 208"
[1] "colnames(resb) <- xnames"
used  (Mb) gc trigger  (Mb) max used  (Mb)
Ncells   363148  19.4     667722  35.7   467875  25.0
Vcells 70630992 538.9  104620716 798.2 85631439 653.4
```

(b) My rewritten function to minimize memory use is given below. Many objects were simply slightly-modified versions of `nalineX` and `nalineY`, so I recoded those lines to remove the redundant objects to reduce memory use.

```r
fastcountnew <- function(xvar, yvar) {
  print(gc())
  select.na <- is.na(xvar + yvar)
  print("select.na <- is.na(xvar + yvar)")
  print(gc())
  xvar[select.na] <- yvar[select.na] <- 0
  print("stopifnot(length(xvar) == length(yvar))")
  print(gc())
  stopifnot(length(xvar) == length(yvar))
  print("stopifnot(length(xvar) == length(yvar))")
  print(gc())
  res <- dummyFun(
    tablex = as.integer(max(xvar)+1), tabley = as.integer(max(yvar)+1),
    xvar = as.integer(xvar), yvar = as.integer(yvar),
    useline = as.integer(!select.na), n = length(xvar))
  print("res <- dummyFun()")
  print(gc())
  xuse <- which(res$tablex > 0)
  xnames <- xuse - 1
  resb <- rbind(res$tablex[xuse], res$tabley[xuse])
  colnames(resb) <- xnames
  print("colnames(resb) <- xnames")
  print(gc())
  return(resb)
}
resbnew <- fastcountnew(xvar, yvar)
```

The memory use at different steps of the rewritten function is given below.

```
used  (Mb) gc trigger  (Mb) max used  (Mb)
```

```
Ncells   326742  17.5     597831  32.0   407500  21.8
Vcells 20575576 157.0   32349347 246.9 30576588 233.3
[1] "select.na <- is.na(xvar + yvar)"
used  (Mb) gc trigger  (Mb) max used  (Mb)
Ncells   326835  17.5     597831  32.0   407500  21.8
Vcells 25575696 195.2   37599611 286.9 35575707 271.5
[1] "stopifnot(length(xvar) == length(yvar))"
used  (Mb) gc trigger (Mb) max used  (Mb)
Ncells   326838  17.5     597831   32   407500  21.8
Vcells 45575696 347.8   56097093  428 51040839 389.5
[1] "stopifnot(length(xvar) == length(yvar))"
used  (Mb) gc trigger (Mb) max used  (Mb)
Ncells   326841  17.5     597831   32   407500  21.8
Vcells 45575696 347.8   58981947  450 51040839 389.5
[1] "res <- dummyFun()"
used  (Mb) gc trigger  (Mb) max used  (Mb)
Ncells   326866  17.5     597831  32.0   407500  21.8
Vcells 60575720 462.2   83659630 638.3 75575742 576.6
[1] "colnames(resb) <- xnames"
used  (Mb) gc trigger  (Mb) max used  (Mb)
Ncells   327041  17.5     597831  32.0   407500  21.8
Vcells 60576003 462.2   87922611 670.8 75575742 576.6
```

6 Problem 6

My main improvement of the code lies in using `outer` to reduce the number of `for` loops used from 3 to 1, which vastly reduced the time taken. I also removed some redundant lines of code.

```r
# Improved and optimised code
likelihood <- function(Theta, A) {
  sum.ind <- which(A == 1)
  log.likelihood <- sum(log(Theta[sum.ind])) - sum(Theta)
  return(log.likelihood)
}

update_once <- function(A, n, K, theta.old, threshold = 0.1) {
  Theta.old <- theta.old %*% t(theta.old)
  L.old <- likelihood(Theta.old, A)
  q <- array(0, dim = c(n, n, K))
  theta.new <- theta.old
  for (z in 1:K) {
    q[, , z] <- outer(theta.old[, z], theta.old[, z])/Theta.old
    theta.new[, z] <- rowSums(A*q[, , z])/sqrt(sum(A*q[, , z]))
  }
  Theta.new <- theta.new %*% t(theta.new)
  L.new <- likelihood(Theta.new, A)
  converge.check <- abs(L.new - L.old) < threshold
  theta.new <- theta.new/rowSums(theta.new)
  return(list(theta = theta.new, loglik = L.new,
              converged = converge.check))
}

load('ps4prob6.Rda') # should have A, n, K
```

```r
# Initialize the parameters at random starting values
set.seed(1234)
temp <- matrix(runif(n*K), n, K)
theta.init <- temp/rowSums(temp)

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

##    user  system elapsed
##  80.762   0.493  81.341

system.time(out2 <- update_once(A, n, K, theta.init))

##    user  system elapsed
##   1.865   0.270   2.135

identical(out, out2)

## [1] TRUE
```