STAT243 - Problem Set 4

Cristobal Pais - cpaismz@berkeley.edu

October 11th, 2017

Problem 1

a) Number of copies: x vector

In order to determine the maximum number of copies of the x vector, we can analyze the provided code line by line. At the beginning, an explicit copy of x is initialized (1 copy). Then, function $f()$ creates a closure/environment for the nested $g()$ function that is declared inside, while passing the value of a data variable (referenced to the value of input) to the return clause inside the $g()$ function for performing its calculations. Thus, a second copy (2) of the vector x would be expected to be generated inside the $f()$ function if it is given as an input value $f(x)$ since it will be in the same environment that contains $g()$, function that will use that value for performing the necessary computations. However, no calculations/modifications are performed and thus, *data* remains as a pointer to the same object as x, without creating an explicit copy.

After the invocation of $f(x)$, no extra copies should be created. In order to check our initial analysis, we will perform a series of operations within the original code such that we can trace the memory usage of R and the extra copies that are being generated.

We clean the session in order to avoid any source of error when evaluating our code and we proceed to our analysis:

```r
# No warnings
options(warn=-1)

# Loading libraries
library("pryr")

# Check current memory usage and maximum using garbage collector information
gc()
```

```
##          used (Mb) gc trigger (Mb) max used (Mb)
## Ncells 377991 20.2     592000 31.7   460000 24.6
## Vcells 593681  4.6    1308461 10.0   791814  6.1
```

```r
# Initialize the first copy (large vector for easy tracking) and invoke gc()
x <- 1:1e8  # First copy (explicit)
gc()
```

```
##            used  (Mb) gc trigger  (Mb) max used  (Mb)
## Ncells   378304  20.3     750400  40.1   460000  24.6
## Vcells 50593830 386.1   73257576 559.0 50604015 386.1
```

```r
# Inspection and tracing
object_size(x)
```

```
## 400 MB
```

```r
tracemem(x)
```

```
## [1] "<00007FF5E7E30010>"
```

```r
.Internal(inspect(x))
```

```
## @0x00007ff5e7e30010 13 INTSXP g1c7 [MARK,NAM(2),TR] (len=100000000, tl=0) 1,2,3,4,5,...
```

```r
# Function f is declared
f <- function(input){
  # Second "copy" (argument is x and it is assigned to data explicitly)
  # Just a reference to x since no evaluation/computation is performed
  data <- input

  # Third object (not x copy): g needs to evaluate the return expression
  g <- function(param) return(param * data)
  return(g)
}

# Invoke the function f
myFun <- f(x) # At this point no new copies
gc()
```

```
##             used  (Mb) gc trigger  (Mb) max used  (Mb)
## Ncells   378590  20.3     750400  40.1   460000  24.6
## Vcells 50594323 386.1   73257576 559.0 50657559 386.5
```

```r
# Data and function execution (myFun)
data <- 100
gc()
```

```
##             used  (Mb) gc trigger  (Mb) max used  (Mb)
## Ncells   378605  20.3     750400  40.1   460000  24.6
## Vcells 50594342 386.1   73257576 559.0 50657559 386.5
```

```r
# Assign result to a variable for visualization purposes
y <- myFun(3)  # Second copy is generated inside the function
object_size(y) # Copy of x and return object
```

```
## 800 MB
```

```r
gc()
```

```
##              used   (Mb) gc trigger   (Mb)  max used   (Mb)
## Ncells     378644   20.3     750400   40.1    460000   24.6
## Vcells 150594388 1149.0  217258362 1657.6 150615344 1149.2
```

```r
# Cleaning the session
rm(list = ls()); gc(reset=TRUE);
```

```
##          used (Mb) gc trigger   (Mb) max used (Mb)
## Ncells 378791 20.3     750400   40.1   378791 20.3
## Vcells 594648  4.6  173806689 1326.1   594648  4.6
```

Based on the output, we can clearly understand what is happening inside the code:

1. An explicit copy of x is initialized. In the example $x = 1 : 1e8$ with an approximate size of 400MB.
2. No copies are generated until the invocation of $myFun(3)$. This is because in R variables are not copied inside the function unless they are being modified or needed inside a certain operation (promises and lazy evaluation inside function properties of R logic structure). When $myFun$ is initialized with $f(x)$, an environment/closure for the function $g()$ is generated, with a reference to the vector x created with the name *data*. However, no extra copies have been made at this point - easily seen by checking the $gc()$ outputs.
3. Once $myFun(3)$ is invoked, a copy of the x vector would be generated if the *data* variable (pointing to x inside the closure created by $f(x)$) is being used in a computation or modified, however, this does not happen.
4. The maximum memory usage is easily explained when we take into account the fact that a vector of the same length as x is generated as the main output from the function $g()$, but multiplied by a scalar equal to 3. This vector is not a copy of x but it has the same length and double size. This can be checked by calling the *object_size*() function and apply it to y. We can see that its size is exactly as twice as the size of the original x vector, indicating that our analysis is correct. As an extra check, we have:

```r
# Loading libraries
library("pryr")

# Vector and size
x <- 1:1e8
object_size(x)
```

```
## 400 MB
```

```r
tracemem(x)
```

```
## [1] "<00007FF5E7E30010>"
```

```r
# Scaling operation
x <- x * 3
object_size(x)
```

```
## 800 MB
```

```r
# Cleaning the session
rm(list = ls()); gc(reset=TRUE);
```

```
##          used (Mb) gc trigger   (Mb) max used (Mb)
## Ncells 382800 20.5     750400   40.1   382800 20.5
## Vcells 597409  4.6  139045351 1060.9   597409  4.6
```

Therefore, one explicit copies of the x vector is generated within the provided code.

Finally, we include a profiling of the code using the profvis/profmem [1] library in order to visualize all our previous results.

```r
# Profile for HTML
# Loading libraries
library("profvis")

# Perform the profiling
profvis({
  # First copy (explicit)
  x <- function(){x = 1:1e8}
  x = x()
  f <- function(input){
    # Second copy (argument is x and it is assigned to data explicitly)
    data <- input
    # Third object: g needs to evaluate the return expression
    g <- function(param) return(param * data)
    return(g)
  }

  myFun <- f(x) # At this point no new copies
  data <- 100
  myFun(3)
})
```

```r
# Loading libraries
library("profmem")

# Perform the profile
p <- profmem({
  # First copy (explicit): as a function for identifying it as an object in
  # the profile
  x <- function(){x = 1:1e8}
  x = x()
  f <- function(input){
    # Second copy (argument is x and it is assigned to data explicitly)
    data <- input
    # Third object: g needs to evaluate the return expression
    g <- function(param) return(param * data)
    return(g)
  }

  myFun <- f(x) # At this point no new copies
  data <- 100
  myFun(3)
})
```

[1] The profvis library is suitable for the HTML version of this document. In the pdf version, we are using the profmem library for simplicity. HTML file is included in the Github repository.

```
# Display results
p
```

```
## Rprofmem memory profiling of:
## {
##     x <- function() {
##         x = 1:1e+08
##     }
##     x = x()
##     f <- function(input) {
##         data <- input
##         g <- function(param) return(param * data)
##         return(g)
##     }
##     myFun <- f(x)
##     data <- 100
##     myFun(3)
## }
##
## Memory allocations:
##         bytes   calls
## 1     4.0e+08     x()
## 2     8.0e+08 myFun()
## total 1.2e+09
```

```
# Cleaning the session
rm(list = ls()); gc(reset=TRUE);
```

```
##          used (Mb) gc trigger   (Mb) max used (Mb)
## Ncells 386773 20.7     750400   40.1   386773 20.7
## Vcells 602924  4.6  144790259 1104.7   602924  4.6
```

b) Serialize() function analysis

In this case, we call the *serialize*() function in order to check our previous analysis:

```
# No warnings
options(warn=-1)

# Loading libraries
library("pryr")

# Initialize the first copy (large vector for easy tracking) and invoke gc()
x <- 1:1e8  # First copy (explicit)

# Function f is declared
f <- function(input){
  # Second "copy" (argument is x and it is assigned to data explicitly)
  # Just a reference to x since no evaluation/computation is performed
  data <- input
  # Third object (not x copy): g needs to evaluate the return expression
  g <- function(param) return(param * data)
  return(g)
}

# Invoke the function f
myFun <- f(x) # At this point no new copies
object_size(myFun)
```

```
## 400 MB
```

```
out <- serialize(myFun, NULL)

# Check the serialize size
object_size(out)
```

```
## 800 MB
```

```
length(out)
```

```
## [1] 800012750
```

```
# Data and function execution (myFun)
data <- 100

# Assign result to a variable for visualization purposes
y <- myFun(3)  # Second copy is generated inside the function

# Cleaning the session
rm(list = ls()); gc(reset=TRUE);
```

```
##          used (Mb) gc trigger   (Mb) max used (Mb)
## Ncells 386858 20.7     750400   40.1   386858 20.7
## Vcells 603384  4.7  240795756 1837.2   603384  4.7
```

Based on the previous results and the ones obtained in section (a), we can see that they match, reaching a value of approximately 800MB, corresponding to the vector of size $10e8$ (x) multiplied by some param returned from the $g()$ function. Therefore, it is the value we would expect to obtain given the answer in section a). In addition, notice that the *object_size*() function is equal to 400MB due to the fact that it takes into account the reference of the variable *data* to the vector x when the closure is created in the invocation of $f(x)$, but as we saw in the previous section using the $gc()$, no explicit copy is generated at this point.

From a different point of view, we should have had expected a very small value indicating that no copies of x are generated. However, a plausible explanation lies in the fact that the *serialization*() function is including the pointer *data* in its analysis, returning the size of the generated object rather than the current size of the function as we can easily check with $gc()$. Thus, it will tell us the amount of memory that is going to be needed when the function is actually called, but it is not indicating that the current object size is 800MB by that line of the code.

c) Arguments and evaluation

When $f(x)$ is called, a closure for $g()$ is created and variable *data* is pointed to the object x but no copy is made in place since no computations/modifications are performed inside $f()$. Hence, when *myFun* is initialized, the function $g()$ is expecting to access the location of x when an evaluation is performed (remembering the lazy evaluation and promises logical structure of R) and no local copies are created since none of these operations are done. Then, since the address associated with vector x is removed, we have that since no copy of x is inside the closure of $g()$, this function will look for the global environment since when x is passed to $f(x)$, but in this case, no x reference exists any longer and thus the function $g()$ it is not able to find its value and an error is risen.

Using the following code, we can check our previous analysis by noting that *data* and $g()$ are the elements inside the enclosing environment for *myFun*() function. Thus, since *data* inside the $g()$ function is pointing to the address associated with x, an error is obtained when R it is not able to find its value, beside that we are assigning a value of 100 to *data* (a different variable) outside the functions.

```
# Code provided
x <- 1:10
tracemem(x)
```

```
## [1] "<0000000013AAD4E0>"
```

```
f <- function(data){
  g <- function(param) return(param * data)
  return(g)
}

myFun <- f(x)
# Check elements in the environment for myFun
ls(environment(myFun))
```

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

```
rm(x)

data <- 100

# Commented out next line for compiling purposes
#myFun(3)

# Cleaning the session
rm(list = ls()); gc(reset=TRUE);
```

```
##          used (Mb) gc trigger   (Mb) max used (Mb)
## Ncells 386770 20.7     750400   40.1   386770 20.7
## Vcells 603029  4.7  192636604 1469.8   603029  4.7
```

d) No extra copies of x

Based on our previous analysis, the simplest solution consists of deleting the line where the address associated with x is removed. Thus, the function will be able to find the value of the vector x in the global environment, obtaining the desired result:

```r
# No warnings
options(warn=-1)

# Loading libraries
library("pryr")

# Code modified
x <- 1:1e8
tracemem(x)
```

```
## [1] "<00007FF5E7E30010>"
```

```r
f <- function(data){
  g <- function(param) return(param * data)
  return(g)
}

myFun <- f(x)

# Remove the following line: do not delete the address of the x vector
#rm(x)

# Serialize analysis
out <- serialize(myFun, NULL)
object_size(out)
```

```
## 11 kB
```

```r
object_size(myFun)
```

```
## 16.8 kB
```

```r
data <- 100
y <- myFun(3)

# Cleaning the session
rm(list = ls()); gc(reset=TRUE);
```

```
##          used (Mb) gc trigger   (Mb) max used (Mb)
## Ncells 386898 20.7     750400   40.1   386898 20.7
## Vcells 603527  4.7  154109283 1175.8   603527  4.7
```

From the previous output, it is clear that the current code is working and no copies are created. In this case, we are able to find a very small value when using the *serialize*() function due to the fact that no explicit pointers (like *data* in the original implementation) are included in the closure of the function and thus, it will return the current size of the function without being able of “predict” the memory usage associated with the vector to be processed since it is outside the closure (in fact, in the global environment).

Problem 2

a) List of vectors analysis

Based on our knowledge, when a modification if performed inside an object (vector in this case) that is referenced by another object (list that contains the vector) a copy of the vector should be generated in order to keep the original memory reference inside the container (list) and associate the modified vector to a new address in memory. Therefore, the change is not being made in place as we can see in the following example:

```
# No warnings
options(warn=-1)

# Loading libraries
library("pryr")

# Initializing vectors
x <- 1:1e8
y <- 1:1e7
z <- 1:1e7
gc()
```

```
##             used  (Mb) gc trigger   (Mb) max used  (Mb)
## Ncells    386976  20.7     750400   40.1   624568  33.4
## Vcells  60604637 462.4  154109283 1175.8 60761491 463.6
```

```
# Trace them
tracemem(x)
```

```
## [1] "<00007FF5E7E30010>"
```

```
tracemem(y)
```

```
## [1] "<00007FF5E5800010>"
```

```
tracemem(z)
```

```
## [1] "<00007FF5E31D0010>"
```

```
# Initialize the list L and trace it
L <- list(x,y,z)
tracemem(L)
```

```
## [1] "<0000000012253830>"
```

```
# Sizes
object_size(L)
```

```
## 480 MB
```

```
object_size(x)
```

```
## 400 MB
```

```
object_size(y)
```

```
## 40 MB
```

```
object_size(z)
```

```
## 40 MB
```

```
gc()
```

```
##            used  (Mb) gc trigger   (Mb) max used  (Mb)
## Ncells   387088  20.7     750400   40.1   624568  33.4
## Vcells 60604753 462.4  154109283 1175.8 60761491 463.6
```

```
# Inspect all elements
.Internal(inspect(L))
```

```
## @0x0000000012253830 19 VECSXP g1c3 [MARK,NAM(2),TR] (len=3, tl=0)
##   @0x00007ff5e7e30010 13 INTSXP g1c7 [MARK,NAM(2),TR] (len=100000000, tl=0) 1,2,3,4,5,...
##   @0x00007ff5e5800010 13 INTSXP g1c7 [MARK,NAM(2),TR] (len=10000000, tl=0) 1,2,3,4,5,...
##   @0x00007ff5e31d0010 13 INTSXP g1c7 [MARK,NAM(2),TR] (len=10000000, tl=0) 1,2,3,4,5,...
```

```
.Internal(inspect(x))
```

```
## @0x00007ff5e7e30010 13 INTSXP g1c7 [MARK,NAM(2),TR] (len=100000000, tl=0) 1,2,3,4,5,...
```

```
.Internal(inspect(y))
```

```
## @0x00007ff5e5800010 13 INTSXP g1c7 [MARK,NAM(2),TR] (len=10000000, tl=0) 1,2,3,4,5,...
```

```
.Internal(inspect(z))
```

```
## @0x00007ff5e31d0010 13 INTSXP g1c7 [MARK,NAM(2),TR] (len=10000000, tl=0) 1,2,3,4,5,...
```

```
# Modify vector x
x[1] <- 2
```

```
## tracemem[0x00007ff5e7e30010 -> 0x00007ff5cb450010]: eval eval withVisible withCallingHandlers handle
## tracemem[0x00007ff5cb450010 -> 0x00007ff59b950010]: eval eval withVisible withCallingHandlers handle
```

```r
gc()
```

```
##              used   (Mb) gc trigger   (Mb)  max used   (Mb)
## Ncells     387136   20.7     750400   40.1    624568   33.4
## Vcells 160604935 1225.4  252994624 1930.2 210615294 1606.9
```

```r
# Sizes
object_size(L)
```

```
## 480 MB
```

```r
object_size(x)
```

```
## 800 MB
```

```r
object_size(y)
```

```
## 40 MB
```

```r
object_size(z)
```

```
## 40 MB
```

```r
# Inspection is performed
.Internal(inspect(L))
```

```
## @0x0000000012253830 19 VECSXP g1c3 [MARK,NAM(2),TR] (len=3, tl=0)
##   @0x00007ff5e7e30010 13 INTSXP g1c7 [MARK,NAM(2),TR] (len=100000000, tl=0) 1,2,3,4,5,...
##   @0x00007ff5e5800010 13 INTSXP g1c7 [MARK,NAM(2),TR] (len=10000000, tl=0) 1,2,3,4,5,...
##   @0x00007ff5e31d0010 13 INTSXP g1c7 [MARK,NAM(2),TR] (len=10000000, tl=0) 1,2,3,4,5,...
```

```r
.Internal(inspect(x))
```

```
## @0x00007ff59b950010 14 REALSXP g1c7 [MARK,NAM(2),TR] (len=100000000, tl=0) 2,2,3,4,5,...
```

```r
.Internal(inspect(y))
```

```
## @0x00007ff5e5800010 13 INTSXP g1c7 [MARK,NAM(2),TR] (len=10000000, tl=0) 1,2,3,4,5,...
```

```r
.Internal(inspect(z))
```

```
## @0x00007ff5e31d0010 13 INTSXP g1c7 [MARK,NAM(2),TR] (len=10000000, tl=0) 1,2,3,4,5,...
```

```r
# Cleaning the session
rm(list = ls()); gc(reset=TRUE);
```

```
##          used (Mb) gc trigger   (Mb) max used (Mb)
## Ncells 387260 20.7     750400   40.1   387260 20.7
## Vcells 605093  4.7  202395699 1544.2   605093  4.7
```

Based on the output, we can clearly see that the list remains with the same address and elements inside (originals x, y, and z vectors) while a new copy of the vector x is created with a different address in comparison to its original one. However, from the *tracemem*() function and the garbage collector output, we can see that a two-step copying process is performed to the vector x: it starts with a certain address in memory, a copy is generated to a new address and then a copy from this last address is generated, without a formal logic explanation behind this behavior. This is the reason why the final memory usage is around 1230MB instead of the expected $480 + 400 = 880$MB obtained by adding a simple copy of x. This can be a problem with *RStudio* in Windows, creating extra copies as mentioned in Piazza. However, same results were obtained using the R version for UNIX installed inside Windows 10.

Therefore, a new vector is created, not performing the changing in place.

For completeness, we include another approach for the same analysis where vectors are being modified from inside the list. If the results are consistent with our initial impressions we should have the following: a new copy of the modified vector is generated but no other copies are created, obtaining a total memory usage equal to the memory used by the main list plus the extra copy of the modified vector.

```r
# No warnings
options(warn=-1)

# Loading libraries
library("pryr")

# Create a list of vectors in-situ
L <- list(1:1e7, 1:1e8)
tracemem(L)
```

```
## [1] "<00000000167A5A40>"
```

```r
object_size(L)
```

```
## 440 MB
```

```r
.Internal(inspect(L))
```

```
## @0x00000000167a5a40 19 VECSXP g0c2 [NAM(2),TR] (len=2, tl=0)
##   @0x00007ff5fd580010 13 INTSXP g0c7 [] (len=10000000, tl=0) 1,2,3,4,5,...
##   @0x00007ff5e5800010 13 INTSXP g0c7 [] (len=100000000, tl=0) 1,2,3,4,5,...
```

```r
gc()
```

```
##            used  (Mb) gc trigger   (Mb) max used  (Mb)
## Ncells   386799  20.7     750400   40.1   594987  31.8
## Vcells 55603892 424.3  161916559 1235.4 55750407 425.4
```

```r
# Modify first vector
L[[1]][1] <- 2
```

```
## tracemem[0x00000000167a5a40 -> 0x0000000012612ff0]: eval eval withVisible withCallingHandlers handle
```

```
gc()
```

```
##            used  (Mb) gc trigger   (Mb) max used  (Mb)
## Ncells   386796  20.7     750400   40.1   594987  31.8
## Vcells 60603910 462.4  161916559 1235.4 70624767 538.9
```

```
.Internal(inspect(L))
```

```
## @0x0000000012612ff0 19 VECSXP g1c2 [MARK,NAM(1),TR] (len=2, tl=0)
##   @0x00007ff5de580010 14 REALSXP g1c7 [MARK] (len=10000000, tl=0) 2,2,3,4,5,...
##   @0x00007ff5e5800010 13 INTSXP g1c7 [MARK,NAM(2)] (len=100000000, tl=0) 1,2,3,4,5,...
```

```
object_size(L)
```

```
## 480 MB
```

```
# Cleaning the session
rm(list = ls()); gc(reset=TRUE);
```

```
##          used (Mb) gc trigger  (Mb) max used (Mb)
## Ncells 386838 20.7     750400  40.1   386838 20.7
## Vcells 603995  4.7  129533247 988.3   603995  4.7
```

Looking at the results, we can confirm the previous analysis by checking the presence of a new address associated with the first vector inside the list and the total memory usage obtained by both the garbage collector and the $object_size()$ function. However, we can also notice the fact that a new address is associated with the list L and that its size is no longer 440MB but 480MB. This seems like some sort of mistake by R since the list contains exactly the same elements as before.

In order to complete our analysis, we add a third variant of the problem, where we have initial vectors x and y, we create a list using them, and we finally modify one of them from within the list as follows:

```
# No warnings
options(warn=-1)

# Loading libraries
library("pryr")

# Vectors
x <- 1:1e8
y <- 1:1e7
tracemem(x)
```

```
## [1] "<00007FF5E7E30010>"
```

```
tracemem(y)
```

```
## [1] "<00007FF5E5800010>"
```

```r
# List
L <- list(x, y)
tracemem(L)
```

```
## [1] "<000000001300EA58>"
```

```r
# Inspect and size
.Internal(inspect(x))
```

```
## @0x00007ff5e7e30010 13 INTSXP g0c7 [NAM(2),TR] (len=100000000, tl=0) 1,2,3,4,5,...
```

```r
.Internal(inspect(y))
```

```
## @0x00007ff5e5800010 13 INTSXP g0c7 [NAM(2),TR] (len=10000000, tl=0) 1,2,3,4,5,...
```

```r
.Internal(inspect(L))
```

```
## @0x000000001300ea58 19 VECSXP g0c2 [NAM(2),TR] (len=2, tl=0)
##   @0x00007ff5e7e30010 13 INTSXP g0c7 [NAM(2),TR] (len=100000000, tl=0) 1,2,3,4,5,...
##   @0x00007ff5e5800010 13 INTSXP g0c7 [NAM(2),TR] (len=10000000, tl=0) 1,2,3,4,5,...
```

```r
object_size(x)
```

```
## 400 MB
```

```r
object_size(y)
```

```
## 40 MB
```

```r
object_size(L)
```

```
## 440 MB
```

```r
gc()
```

```
##             used  (Mb) gc trigger  (Mb) max used  (Mb)
## Ncells   387047  20.7     750400  40.1   663024  35.5
## Vcells 55605102 424.3  129533247 988.3 55844766 426.1
```

```r
# Modification of x inside list L
L[[1]][1] <- 2
```

```
## tracemem[0x000000001300ea58 -> 0x0000000016f175f8]: eval eval withVisible withCallingHandlers handle
## tracemem[0x00007ff5e7e30010 -> 0x00007ff5cda80010]: eval eval withVisible withCallingHandlers handle
## tracemem[0x00007ff5cda80010 -> 0x00007ff59df80010]: eval eval withVisible withCallingHandlers handle
```

```r
# Inspect and size
.Internal(inspect(x))
```

```
## @0x00007ff5e7e30010 13 INTSXP g1c7 [MARK,NAM(2),TR] (len=100000000, tl=0) 1,2,3,4,5,...
```

```r
.Internal(inspect(y))
```

```
## @0x00007ff5e5800010 13 INTSXP g1c7 [MARK,NAM(2),TR] (len=10000000, tl=0) 1,2,3,4,5,...
```

```r
.Internal(inspect(L))
```

```
## @0x0000000016f175f8 19 VECSXP g1c2 [MARK,NAM(1),TR] (len=2, tl=0)
##   @0x00007ff59df80010 14 REALSXP g0c7 [TR] (len=100000000, tl=0) 2,2,3,4,5,...
##   @0x00007ff5e5800010 13 INTSXP g1c7 [MARK,NAM(2),TR] (len=10000000, tl=0) 1,2,3,4,5,...
```

```r
object_size(x)
```

```
## 400 MB
```

```r
object_size(y)
```

```
## 40 MB
```

```r
object_size(L)
```

```
## 840 MB
```

```r
gc()
```

```
##              used   (Mb) gc trigger   (Mb)  max used   (Mb)
## Ncells    387092   20.7     750400   40.1    663024   35.5
## Vcells 155605276 1187.2  246994846 1884.5 205678866 1569.3
```

```r
# Cleaning the session
rm(list = ls()); gc(reset=TRUE);
```

```
##          used (Mb) gc trigger   (Mb) max used (Mb)
## Ncells 387094 20.7     750400   40.1   387094 20.7
## Vcells 605285  4.7  197595876 1507.6   605285  4.7
```

Using this approach, results indicate that a new copy of the list L is created alongside with a new copy of the vector x. In addition, the size of the list L seems to be as twice as before. However, looking at the amount of memory in use ($gc()$), we can conclude that a copy of x is created in order to match the current total memory usage, but a series of middle steps are performed such that the maximum amount of memory used is larger than the current state.

b) List of vectors and copies

Making a copy of the list of vectors will not generate any copy-on-change. Regarding the copies generated when one vector is modified, we will use the same approach as in the previous section. A first version where modifications are performed directly on one vector and a second analysis where modifications are developed from within one of the lists will be analyzed as follows:

```r
# No warnings
options(warn=-1)

# Loading libraries
library("pryr")

# Vectors
x <- 1:1e8
y <- 1:1e7
tracemem(x)
```

```
## [1] "<00007FF5E7E30010>"
```

```r
tracemem(y)
```

```
## [1] "<00007FF5E5800010>"
```

```r
# Lists
L <- list(x, y)
L2 <- L
tracemem(L)
```

```
## [1] "<00000000142BA540>"
```

```r
tracemem(L2)
```

```
## [1] "<00000000142BA540>"
```

```r
# Inspect and size
.Internal(inspect(x))
```

```
## @0x00007ff5e7e30010 13 INTSXP g0c7 [NAM(2),TR] (len=100000000, tl=0) 1,2,3,4,5,...
```

```r
.Internal(inspect(y))
```

```
## @0x00007ff5e5800010 13 INTSXP g0c7 [NAM(2),TR] (len=10000000, tl=0) 1,2,3,4,5,...
```

```r
.Internal(inspect(L))
```

```
## @0x00000000142ba540 19 VECSXP g0c2 [NAM(2),TR] (len=2, tl=0)
##   @0x00007ff5e7e30010 13 INTSXP g0c7 [NAM(2),TR] (len=100000000, tl=0) 1,2,3,4,5,...
##   @0x00007ff5e5800010 13 INTSXP g0c7 [NAM(2),TR] (len=10000000, tl=0) 1,2,3,4,5,...
```

```r
.Internal(inspect(L2))
```

```
## @0x00000000142ba540 19 VECSXP g0c2 [NAM(2),TR] (len=2, tl=0)
##   @0x00007ff5e7e30010 13 INTSXP g0c7 [NAM(2),TR] (len=100000000, tl=0) 1,2,3,4,5,...
##   @0x00007ff5e5800010 13 INTSXP g0c7 [NAM(2),TR] (len=10000000, tl=0) 1,2,3,4,5,...
```

```r
object_size(x)
```

```
## 400 MB
```

```r
object_size(y)
```

```
## 40 MB
```

```r
object_size(L)
```

```
## 440 MB
```

```r
object_size(L2)
```

```
## 440 MB
```

```r
gc()
```

```
##            used  (Mb) gc trigger  (Mb) max used  (Mb)
## Ncells   379671  20.3     750400  40.1   492800  26.4
## Vcells 55599420 424.2   73264738 559.0 55769759 425.5
```

```r
# Modification of x
x[1] <- 2
```

```
## tracemem[0x00007ff5e7e30010 -> 0x00007ff5cda80010]: eval eval withVisible withCallingHandlers handle
## tracemem[0x00007ff5cda80010 -> 0x00007ff59df80010]: eval eval withVisible withCallingHandlers handle
```

```r
# Inspect and size
.Internal(inspect(x))
```

```
## @0x00007ff59df80010 14 REALSXP g0c7 [NAM(1),TR] (len=100000000, tl=0) 2,2,3,4,5,...
```

```r
.Internal(inspect(y))
```

```
## @0x00007ff5e5800010 13 INTSXP g1c7 [MARK,NAM(2),TR] (len=10000000, tl=0) 1,2,3,4,5,...
```

```r
.Internal(inspect(L))
```

```
## @0x00000000142ba540 19 VECSXP g1c2 [MARK,NAM(2),TR] (len=2, tl=0)
##   @0x00007ff5e7e30010 13 INTSXP g1c7 [MARK,NAM(2),TR] (len=100000000, tl=0) 1,2,3,4,5,...
##   @0x00007ff5e5800010 13 INTSXP g1c7 [MARK,NAM(2),TR] (len=10000000, tl=0) 1,2,3,4,5,...
```

```r
.Internal(inspect(L2))
```

```
## @0x00000000142ba540 19 VECSXP g1c2 [MARK,NAM(2),TR] (len=2, tl=0)
##   @0x00007ff5e7e30010 13 INTSXP g1c7 [MARK,NAM(2),TR] (len=100000000, tl=0) 1,2,3,4,5,...
##   @0x00007ff5e5800010 13 INTSXP g1c7 [MARK,NAM(2),TR] (len=10000000, tl=0) 1,2,3,4,5,...
```

```r
object_size(x)
```

```
## 800 MB
```

```r
object_size(y)
```

```
## 40 MB
```

```r
object_size(L)
```

```
## 440 MB
```

```r
object_size(L2)
```

```
## 440 MB
```

```r
gc()
```

```
##              used   (Mb) gc trigger   (Mb)  max used   (Mb)
## Ncells    379771   20.3     750400   40.1    492800   26.4
## Vcells 155599628 1187.2  246988040 1884.4 205694134 1569.4
```

```r
# Cleaning the session
rm(list = ls()); gc(reset=TRUE);
```

```
##          used (Mb) gc trigger   (Mb) max used (Mb)
## Ncells 379768 20.3     750400   40.1   379768 20.3
## Vcells 599627  4.6  197590432 1507.5   599627  4.6
```

Based on the results obtained, an explicit copy of the vector x is created as expected, without modifying any of the lists which are still using the same references. However, we again obtain an extra copy of the vector x as in the previous section (that can be seen in the previous output thanks to the tracemem command) reaching a final memory usage value around 1.2GB.

```r
# No warnings
options(warn=-1)

# Loading libraries
library("pryr")

# Lists
L <- list(1:1e7, 1:1e8)
L2 <- L
tracemem(L)
```

```
## [1] "<00000000120D22C8>"
```

```
tracemem(L2)
```

```
## [1] "<00000000120D22C8>"
```

```
# Inspect and size
.Internal(inspect(L))
```

```
## @0x00000000120d22c8 19 VECSXP g0c2 [NAM(2),TR] (len=2, tl=0)
##   @0x00007ff5fd580010 13 INTSXP g0c7 [] (len=10000000, tl=0) 1,2,3,4,5,...
##   @0x00007ff5e5800010 13 INTSXP g0c7 [] (len=100000000, tl=0) 1,2,3,4,5,...
```

```
.Internal(inspect(L2))
```

```
## @0x00000000120d22c8 19 VECSXP g0c2 [NAM(2),TR] (len=2, tl=0)
##   @0x00007ff5fd580010 13 INTSXP g0c7 [] (len=10000000, tl=0) 1,2,3,4,5,...
##   @0x00007ff5e5800010 13 INTSXP g0c7 [] (len=100000000, tl=0) 1,2,3,4,5,...
```

```
object_size(L)
```

```
## 440 MB
```

```
object_size(L2)
```

```
## 440 MB
```

```
gc()
```

```
##            used  (Mb) gc trigger   (Mb) max used  (Mb)
## Ncells   380470  20.4     750400   40.1   617486  33.0
## Vcells 55599220 424.2  158072345 1206.0 55791185 425.7
```

```
# Modification of second vector
L[[2]][1] <- 2
```

```
## tracemem[0x00000000120d22c8 -> 0x0000000016c73c50]: eval eval withVisible withCallingHandlers handle
```

```
# Inspect and size
.Internal(inspect(L))
```

```
## @0x0000000016c73c50 19 VECSXP g1c2 [MARK,NAM(1),TR] (len=2, tl=0)
##   @0x00007ff5fd580010 13 INTSXP g1c7 [MARK,NAM(2)] (len=10000000, tl=0) 1,2,3,4,5,...
##   @0x00007ff59df80010 14 REALSXP g0c7 [] (len=100000000, tl=0) 2,2,3,4,5,...
```

```
.Internal(inspect(L2))
```

```
## @0x00000000120d22c8 19 VECSXP g1c2 [MARK,NAM(2),TR] (len=2, tl=0)
##   @0x00007ff5fd580010 13 INTSXP g1c7 [MARK,NAM(2)] (len=10000000, tl=0) 1,2,3,4,5,...
##   @0x00007ff5e5800010 13 INTSXP g1c7 [MARK,NAM(2)] (len=100000000, tl=0) 1,2,3,4,5,...
```

```
object_size(L)
```

```
## 840 MB
```

```
object_size(L2)
```

```
## 440 MB
```

```
gc()
```

```
##              used   (Mb) gc trigger   (Mb)  max used (Mb)
## Ncells    380533   20.4     750400   40.1    617486   33
## Vcells 155599361 1187.2  246987810 1884.4 205651909 1569
```

```
# Cleaning the session
rm(list = ls()); gc(reset=TRUE);
```

```
##           used (Mb) gc trigger   (Mb) max used (Mb)
## Ncells 380541 20.4     750400   40.1   380541 20.4
## Vcells 599377  4.6  197590248 1507.5   599377  4.6
```

Similar to the results obtained in the previous section, the output indicates that a copy of the list is generated with a different address for the second element (vector), showing us that a copy of that element has been generated. However, since a modification has been done directly in the list L, its size is reported as $440 + 400 = 840$MB including the new copy of the vector of length $10e8$ generated.

Therefore, a copy of the relevant vector is generated, otherwise, we would end up with twice the size of the original list and that is not the case.

c) List of lists analysis

Similarly to the previous sections, in this case, we will analyze the situation by studying the output obtained from a pertinent code. In this case, we proceed as follows:

```
# Loading libraries
options(warn=-1)
library(pryr)

# Lists
L1 <- list(1:1e8)
L2 <- list(1:1e7)
L = list(L1, L2)

# Trace
tracemem(L1)
```

```
## [1] "<0000000015EDC2E0>"
```

```r
tracemem(L2)
```

```
## [1] "<00000000143F3D90>"
```

```r
tracemem(L)
```

```
## [1] "<0000000012CE00F8>"
```

```r
# Inspection and size
.Internal(inspect(L1))
```

```
## @0x0000000015edc2e0 19 VECSXP g0c1 [NAM(2),TR] (len=1, tl=0)
##   @0x00007ff5e7e30010 13 INTSXP g0c7 [] (len=100000000, tl=0) 1,2,3,4,5,...
```

```r
.Internal(inspect(L2))
```

```
## @0x00000000143f3d90 19 VECSXP g0c1 [NAM(2),TR] (len=1, tl=0)
##   @0x00007ff5e5800010 13 INTSXP g0c7 [] (len=10000000, tl=0) 1,2,3,4,5,...
```

```r
.Internal(inspect(L))
```

```
## @0x0000000012ce00f8 19 VECSXP g0c2 [NAM(2),TR] (len=2, tl=0)
##   @0x0000000015edc2e0 19 VECSXP g0c1 [NAM(2),TR] (len=1, tl=0)
##     @0x00007ff5e7e30010 13 INTSXP g0c7 [] (len=100000000, tl=0) 1,2,3,4,5,...
##   @0x00000000143f3d90 19 VECSXP g0c1 [NAM(2),TR] (len=1, tl=0)
##     @0x00007ff5e5800010 13 INTSXP g0c7 [] (len=10000000, tl=0) 1,2,3,4,5,...
```

```r
object_size(L1)
```

```
## 400 MB
```

```r
object_size(L2)
```

```
## 40 MB
```

```r
object_size(L)
```

```
## 440 MB
```

```r
gc()
```

```
##             used  (Mb) gc trigger   (Mb) max used  (Mb)
## Ncells    380587  20.4     750400   40.1   643678  34.4
## Vcells 55600091 424.2  158072198 1206.0 55828379 426.0
```

```r
# Add an element to the second list
L2[length(L2) + 1] <- 1
```

```
## tracemem[0x00000000143f3d90 -> 0x0000000016e08348]: eval eval withVisible withCallingHandlers handle
```

```r
# Check final status
.Internal(inspect(L1))
```

```
## @0x0000000015edc2e0 19 VECSXP g1c1 [MARK,NAM(2),TR] (len=1, tl=0)
##   @0x00007ff5e7e30010 13 INTSXP g1c7 [MARK] (len=100000000, tl=0) 1,2,3,4,5,...
```

```r
.Internal(inspect(L2))
```

```
## @0x0000000016e02db0 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
##   @0x00007ff5e5800010 13 INTSXP g1c7 [MARK,NAM(2)] (len=10000000, tl=0) 1,2,3,4,5,...
##   @0x0000000016e08468 14 REALSXP g0c1 [] (len=1, tl=0) 1
```

```r
.Internal(inspect(L))
```

```
## @0x0000000012ce00f8 19 VECSXP g1c2 [MARK,NAM(2),TR] (len=2, tl=0)
##   @0x0000000015edc2e0 19 VECSXP g1c1 [MARK,NAM(2),TR] (len=1, tl=0)
##     @0x00007ff5e7e30010 13 INTSXP g1c7 [MARK] (len=100000000, tl=0) 1,2,3,4,5,...
##   @0x00000000143f3d90 19 VECSXP g1c1 [MARK,NAM(2),TR] (len=1, tl=0)
##     @0x00007ff5e5800010 13 INTSXP g1c7 [MARK,NAM(2)] (len=10000000, tl=0) 1,2,3,4,5,...
```

```r
object_size(L1)
```

```
## 400 MB
```

```r
object_size(L2)
```

```
## 40 MB
```

```r
object_size(L)
```

```
## 440 MB
```

```r
gc()
```

```
##             used  (Mb) gc trigger   (Mb) max used  (Mb)
## Ncells   380655  20.4     750400   40.1   643678  34.4
## Vcells 55600268 424.2  158072198 1206.0 55828379 426.0
```

```r
# Cleaning the session
rm(list = ls()); gc(reset=TRUE);
```

```
##           used (Mb) gc trigger  (Mb) max used (Mb)
## Ncells 380651 20.4     750400  40.1   380651 20.4
## Vcells 600265  4.6  126457758 964.8   600265  4.6
```

In this case, we can easily check that no copies are generated across all the objects, both list are still sharing the original elements ($L1$ and $L2$ as they were initialized) and a modification is developed in place for the second list such that it changes its memory address and includes the new element added. Therefore, since a simple element was added to the second list, no changes in memory are registered after the operation.

d) Conflict with size of an object

The explanation behind the obtained results of the following code lies in the fact that the *object.size*() function, as indicated in its help file, *"provides a rough indication: it should be reasonably accurate for atomic vectors, but does not detect if elements of a list are shared, for example"*. Thus, due to its limitations, the function is not able to detect the fact that we are referring to the same vector *x* from within the same list *tmp*.

```
# Code provided
gc()
```

```
##          used (Mb) gc trigger  (Mb) max used (Mb)
## Ncells 386652 20.7     750400  40.1   490611 26.3
## Vcells 605111  4.7  101169218 771.9   669280  5.2
```

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

```
## @0x0000000016d264f0 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
##   @0x00007ff5faf60010 14 REALSXP g0c7 [NAM(2)] (len=10000000, tl=0) -2.10029,0.257841,2.04266,0.3893
##   @0x00007ff5faf60010 14 REALSXP g0c7 [NAM(2)] (len=10000000, tl=0) -2.10029,0.257841,2.04266,0.3893
```

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

```
## 160000136 bytes
```

```
gc()
```

```
##            used (Mb) gc trigger  (Mb) max used (Mb)
## Ncells   387294 20.7     750400  40.1   490611 26.3
## Vcells 10606048 81.0   80935374 617.5 10682132 81.5
```

In order to obtain the real size of the object, we can use the *object_size*() function from the pryr package since it is able to detect if elements inside an object are shared objects (same object). As said in its help file, *"object_size works similarly to object.size, but counts more accurately and includes the size of environments. While object.size doesn't keep track of shared elements in an object, object_size does"*.

```
# Using the object_size function from the pryr package
# gives us the real size
options(warn=-1)
library("pryr")

object_size(tmp)
```

```
## 80 MB
```

```
##          used (Mb) gc trigger  (Mb) max used (Mb)
## Ncells 387201 20.7     750400  40.1   387201 20.7
## Vcells 605741  4.7   64748299 494.0   605741  4.7
```

Therefore, we have checked our analysis.

Problem 3

In this problem We have two main functions that can be improved: (1) *ll()* for calculating the log-likelihood value of a current iteration and (2) *oneUpdate()* function that performs all the computations needed for updating (in an iterative way) all the relevant variables needed for calculating the new log-likelihood function until a convergence criterion (a certain threshold) is reached. The general strategy for improving both functions consists of:

1. Compact the *ll()* function in only one command line (expression) without explicitly calculating the indexes needed for the summation of the Θ matrix, saving both memory and computation time.
2. Eliminate as much as possible all the *for* loops (nested) inside the *oneUpdate()* function using vectorized operations in such a way that the same multidimensional array q is obtained in a far more efficient way, and delete all the extra (and unnecessary) calculations/variables/steps performed in order to obtain exactly the same final result as the original function.

In order to compare our proposed functions with the original ones, a series of microbenchmark comparisons will be performed for particular chunks of the code and for the overall function usage. Plots are provided for visualization purposes alongside with memory profilings.

a) Improving a log-likelihood function: AS-IS state

Before describing the modifications performed to both functions, we can analyze the current state of *ll()* and *oneUpdate()* in terms of memory usage and running time. In order to do this, we can use the useful "profvis" package that allows us to check the memory usage and running time of each line of the code in order to easily detect the bottlenecks and potential improvements of the current functions/steps. Therefore, we proceed as follows:

1. Relevant libraries are loaded in order to be able to analyze the code and obtain the same results by fixing a random seed for our analysis. We set the working directory and random testing parameters (*theta.init*) are generated using the data loaded from the .Rda file provided with the problem statement.

```r
# No warnings (visualization purposes)
options(warn=-1)

# Loading libraries
library(profvis)
library(microbenchmark)
library(ggplot2)
library(pryr)
require(stats)

# Setting random seed for reproducibility
set.seed(1)

# Set working directory and load the relevant data for A, n, K
setwd("C:/Users/chile/Desktop/Stats243/HW/HW4/Code/")
load('ps4prob3.Rda')

## Testing the functions
# Initialize the parameters at random starting values (remove temp, saving memory)
temp <- matrix(runif(n*K), n, K)
```

```
theta.init <- temp/rowSums(temp)
rm(temp)
```

2. The original function is loaded, a testing *Theta.init* value is generated (equivalent to *Theta.old* in the original code), and a microbenchmark running time analysis is performed for 100 times in order to be able to analyze the mean running time of the function.

```
## Log-likelihood function
# LL original function
ll <- function(Theta, A) {
  sum.ind <- which(A==1, arr.ind=T)
  logLik <- sum(log(Theta[sum.ind])) - sum(Theta)
  return(logLik)
}

# ll() function microbenchmark results
Theta.init = theta.init %*% t(theta.init)
mbmll <- microbenchmark(ll(Theta.init, A), times = 100)

# Show and plot (if required) results
print(mbmll)
```

```
## Unit: milliseconds
##               expr     min       lq     mean   median       uq     max
##  ll(Theta.init, A) 2.50733 2.603123 2.806627 2.689296 2.803479 8.56931
##  neval
##    100
```

```
#autoplot(mbmll)
```

Based on the previous results, we can see that the original *ll()* function mean running time is around 2.80 milliseconds and its range is represented by [2.50, 8.57] milliseconds approximately. Since the operations performed inside this function are very simple and direct, there is not so much margin to improve it, however, as we will see in the next section, we can still improve its performance.

3. Now, the same procedure is replicated for the *oneUpdate()* function: definition and microbenchmark running time analysis.

```
## Iteration function
# OneUpdate original function
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
```

```r
        } 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))
}

# oneUpdate() function microbenchmark results (10 runs for visualization)
mbmoneUp <- microbenchmark(oneUpdate(A, n, K, theta.init), times = 5)

# Show and plot (if required) results
print(mbmoneUp)
```

```
## Unit: seconds
##                            expr      min       lq     mean   median
##  oneUpdate(A, n, K, theta.init) 94.78753 95.22979 96.02513 96.18557
##       uq      max neval
##  96.8712 97.05155     5
```

```r
#autoplot(mbmoneUp)
```

In this case we can notice that the mean running time per iteration is around one minute and 30 seconds: very slow and poor performance for a function that we would need to run maybe hundred of times before reaching the algorithm convergence. Therefore, a series of improvements can be implemented as we will see in the next section (b).

4. In order to finalize our AS-IS analysis, we perform a code profiling using the $profvis/profmem$ package that will allow us to detect the current bottlenecks and memory usage of each line of code in the original functions. Please check the online .html version of this document inside the GitHub repository.

As seen in the previous output (check online), the code profiling analysis is called using the profvis/profmem function. Based on the profiling results, we can see that the largest amount of time of the function (and hence, its bottleneck) is spent in the three for nested loops for generating the multidimensional array q. Alongside with this, it is clearly the operation that demands more memory from the system.

Thus, we will focus our improvements in trying to avoid the nested loops implementation using vectorized and algebraic operations for computing the relevant values (q) and perform an efficient iteration.

b) Improving a log-likelihood function: Proposed solution

Based on the AS-IS state, we perform a series of modifications to the original functions in order to improve the performance of both of them. A great performance improvement is obtained when the *oneUpdate()* function is reformulated in a way that it takes advantage of the structure of the problem using a series of algebraic (and vectorized) operations instead of a series of nested *for* loops (very slow implementation). In addition, some steps are eliminated or modified in order to perform the minimum number of calculations as possible.

Although the original *ll()* function is pretty efficient due to the fact that its operations are straightforward and direct, we are able to improve it by performing all the calculations in just one line.

1. Instead of computing, extracting, and recording the relevant indexes for the log-likelihood function summation, we can simply contract everything in one command line where we include the values of the A matrix that are equal to one. It is very easy to check that both functions perform exactly the same calculations but in our version, no intermediate steps are needed. In order to compare them, we perform a microbenchmark comparison using the loaded data for testing, and a value checking step is added for visualization purposes:

```
# LL new function: All computations in one line (no need to save indexes)
llNew <- function(Theta, A) {
  logLik <- sum(log(Theta[A == 1])) - sum(Theta)
  return(logLik)
}

## Log-likelihood fuctions comparison
mbmllcomp <- microbenchmark(llorig <- ll(Theta.init, A),
                            llnew <- llNew(Theta.init, A), times = 5)

# Print results
print(mbmllcomp)
```

```
## Unit: milliseconds
##                            expr      min       lq     mean   median
##     llorig <- ll(Theta.init, A) 2.595853 2.609966 2.793771 2.639047
##   llnew <- llNew(Theta.init, A) 1.988159 1.996285 2.170938 2.195998
##        uq      max neval
##  2.692931 3.431060     5
##  2.301200 2.373046     5
```

```
# Plot results (if required)
#autoplot(mbmllcomp)

# Check values
print(paste("Log-likelihood values:", llorig, ",", llnew, sep = ""))
```

```
## [1] "Log-likelihood values:-55574.3696913836,-55574.3696913836"
```

2. In the case of the *oneUpdate()* function, our main modification (and more relevant in terms of performance impact) consists of contract the series of for loops into one simple expression that exploits vectorized operations while using a simple call of the *lapply*() function. The expression is based on the structure of the computation: every iteration we are multiplying a specific entry of the *theta.init* (old in the

original notation) vector by another one, depending on the values of i, j, and z. Then, an unnecessary zero value assignment is performed since the q multidimensional array is initialized with all its entries equal to zero so we will be able to eliminate that step, and on the other hand, the operations performed when no null values are multiplied can be easily vectorized when we notice that a similar result can be obtained by simply using an outer multiplication between vectors and divide it by the relevant value of the *Theta.init* (.old in the original code) matrix.

As an extra safety step, we include an if clause in the proposed expression for detecting null values inside the *Theta.init* matrix such that we avoid divisions by zero (inf numbers) or NaN values. Important is to note that q is no longer a multidimensional array in the new function, but a multidimensional list: better memory performance and easy to manipulate or unlist if we need it.

```r
# Original function: q calculation step
qOneUpdate <- function(){
  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.init[i, z] * theta.init[j, z] == 0){
          q[i, j, z] <- 0
        } else {
          q[i, j, z] <- theta.init[i, z] * theta.init[j, z] / Theta.init[i, j]
        }
      }
    }
  }
  return(q)
}

# New function: q calculation step
qOneUpdateNew <- function(){

  # Replace of the nested for loops: one lapply call using 1:K vector as the
  # main input. New coordinate system q[[z]][i,j] instead of q[i, j, z]
  q <- lapply(1:K,
              function(x) {
                ifelse(Theta.init != 0,
                       (theta.init[, x] %o% theta.init[, x]) / Theta.init,
                       0)
                })
  # Return the q multidimensional list
  return(q)
}
```

3. In order to test our proposed function, we perform a set of specific comparisons for the code needed to obtain q in both the original and the new function versions as follows:

```r
# Microbenchmark analysis
mbmq <- microbenchmark(q <- qOneUpdate(), qNew <- qOneUpdateNew(), times = 5)

# Print results
print(mbmq)
```

```
## Unit: seconds
##                    expr       min        lq      mean    median       uq
##        q <- qOneUpdate() 94.425415 95.032712 95.347308 95.118156 96.04900
##  qNew <- qOneUpdateNew()  1.578509  1.583234  1.600028  1.605245  1.60705
##        max neval
##  96.111260     5
##   1.626104     5
```

```
# Checking results: dimension and comparison
qNewplain <- unlist(qNew)

# Comparison with original q
print(paste("Number of different elements: ", length(q[!(q == qNewplain)]),
            sep = ""))
```

```
## [1] "Number of different elements: 0"
```

```
print(paste("Lenght of original q: ", length(q), sep = ""))
```

```
## [1] "Lenght of original q: 25250000"
```

```
print(paste("Lenght of new q: ", length(qNewplain), sep = ""))
```

```
## [1] "Lenght of new q: 25250000"
```

```
# Checking random values qV2[[z]][i, j] = q[i, j, z] for completeness
q[1, 2, 3] == qNew[[3]][1, 2]
```

```
## [1] TRUE
```

```
q[24, 15, 77] == qNew[[77]][24, 15]
```

```
## [1] TRUE
```

```
q[444, 29, 61] == qNew[[61]][444, 29]
```

```
## [1] TRUE
```

```
# Plot results (uncomment if required)
#autoplot(mbmq)
```

Based on the results we can see that our proposed algorithm for calculating q obtains a far better performance than the original algorithm, reaching a **63.5-fold speedup** [2], a very impressive result. This great performance is related to the fact that we are exploiting the efficient vectorized operations in R (in a very smart way) while avoiding the poor performance nested *for* loops. We can see that the mean running time for the original function is around 96 seconds (1.5 minutes) while our new function just needs 1.62 seconds in average for solving exactly the same problem.

[2]Best result obtained across all our experiments.

3. Finally, we define our new function, including the following relevant changes:
   i) Deletion of unnecessary lines (commented in the code).
   ii) Replace of the nested *for* loops structure.
   iii) Non-explicit computation of *Theta.new* for calculating the new log-likelihood.

```r
# OneUpdate new function
oneUpdateNew <- function(A, n, K, theta.old, thresh = 0.1) {
  # Delete the Theta.old1 reference (unused), keep the next two lines
  Theta.old <- theta.old %*% t(theta.old)
  L.old <- llNew(Theta.old, A)

  # Replace of the nested for loops: one lapply call using 1:K vector as the
  # main input. New coordinate system q[[z]][i,j] instead of q[i, j, z]
  q <- lapply(1:K,
              function(x) {
                ifelse(Theta.old != 0,
                       (theta.old[, x] %o% theta.old[, x]) / Theta.old,
                       0)
                })

  # For loop is performed using list q (better performance than a/la/sapply
  # functions), modification in place without creating an extra copy theta.new
  for (z in 1:K) {
    theta.old[, z] <- rowSums(A * q[[z]]) / sqrt(sum(A * q[[z]]))
  }

  # New reference after modification (for simplicity)
  theta.new <- theta.old

  # Theta.new is created inside the llnew function call since Theta.new is not
  # used in the rest of the function, saving memory.
  L.new <- llNew(theta.new %*% t(theta.new), A)

  # Check convergence value
  converge.check <- abs(L.new - L.old) < thresh

  # Update theta.new
  theta.new <- theta.new / rowSums(theta.new)

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

4. Therefore, we perform a global comparison of both the original and new functions as follows:

```r
## Full functions comparison
# Individual calls
out <- oneUpdate(A, n, K, theta.init)
out2 <- oneUpdateNew(A, n, K, theta.init)
print(paste("Number of different Theta values: ",
            length(out$theta[!(out$theta == out2$theta)]),
            sep = ""))
```

```
## [1] "Number of different Theta values: 0"
```

```
print(paste("Value of LLs obtained: ", out$loglik, ",",
            out2$loglik, sep = ""))
```

```
## [1] "Value of LLs obtained: -41655.6856747477,-41655.6856747477"
```

```
print(paste("Convergence status: ", out$converged, ",",
            out2$converged, sep = ""))
```

```
## [1] "Convergence status: FALSE,FALSE"
```

```
# Perform a single update
system.time(out <- oneUpdate(A, n, K, theta.init))
```

```
##    user  system elapsed
##   94.12    0.14   94.30
```

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

```
##    user  system elapsed
##    1.06    0.85    1.91
```

```
# Benchmark for multiple calls
mbm <- microbenchmark(oneUpdate(A, n, K, theta.init),
                      oneUpdateNew(A, n, K, theta.init),
                      times = 5)

# Print results
print(mbm)
```

```
## Unit: seconds
##                               expr       min        lq     mean   median
##     oneUpdate(A, n, K, theta.init) 93.985003 94.127366 94.41984 94.43247
##  oneUpdateNew(A, n, K, theta.init)  1.891545  1.898465  1.90873  1.90400
##         uq       max neval
##  94.653673 94.900700     5
##   1.910926  1.938715     5
```

```
# Plots (uncomment if required)
#autoplot(mbm)
#boxplot(mbm, log = FALSE, unit = "s", xlab = "Functions")
```

Based on the results obtained, we can easily see that we were able to improve the performance of the original function in a very significant way, reaching speedups up to 63.5-folds [3] thanks to the smart vectorized operations instead of a series of nested *for* loops inside the original function.

[3] Best result obtained during our tests.

Problem 4

In this problem, we will improve two functions: (1) The *PIKK()* function that returns k random values without replacement from a vector x, where $k \leq n$, and n the size of the x vector, and (2) the *FYKD()* function that returns the same output as the previous function but using an iterative shuffling algorithm in order to generate and extract the k random values from a vector x without replacement.

In order to improve them, different approaches and modifications of the original functions are implemented. The main strategies consist of eliminating extra calculations/steps as well as modifying/replacing inefficient computations by more efficient expressions. Due to their structure and AS-IS state, both functions can be easily improved by taking into account the previous points.

a.1) Improving PIKK function

Looking at the original function, we can clearly identify some performance issues that can be tackled in order to improve the formulation: (1) Besides k values are returned as the main output of the function, n random numbers are generated, (2) Sorting function is needed in order to obtain a random permutation based on the random numbers generated. These two operations can be easily improved using a stochastic approach such that the expected number of iterations (and therefore running time) is clearly less than the original formulation.

We develop three new functions that we are going to compare with the original $PIKK()$ function in terms of running time:

i) $PIKKV1()$: Poor performance (slow) implementation using loops for comparison purposes. The idea is to show how we can develop a very bad formulation of the same problem depending on the approach used to tackle the problem.

ii) $PIKKV2()$: High performance implementation based on $runif()$ function (random number generations) transformed into a k-indexes array.

iii) $PIKKVF()$: High performance implementation based $PIKKV2()$ with an extra checking step.

In addition, we include the R *sample*() function version in our benchmarks for comparison purposes (more details in section b).

1. The logic of our first function consists of an iterative approach where each element inside the x vector has a chance of being selected equal to $\frac{k_i}{n_i}$ where k_i is equal to the remaining number of elements at iteration i that must be selected in order to get k entries and n_i is equal to the remaining number of entries inside the vector x (not yet iterated). Thus, $n_i = n - i + 1$ starting from iteration $i = 1$ and the value of k_i will be updated every iteration depending on the probability of picking that entry.

   Clearly, this function will have a poor performance due to the number of steps, calculations, and inefficient use of a for loop included. However, we are including it for comparison purposes in our analysis.

```
# First new function: PIKKV1
PIKKV1 <- function(x, k){
  # Initialization of selected numbers array and size variable
  sel = integer(k)
  size = k

  # Variable for keeping track of the current not visited entries of vector x
  left = length(x)

  # For loop: select elements based on k_{i}/n_{i} probability, ending up with the
  # desire number of entries k
  aux = 1
  for (i in 1:length(x)){

    # Probability of selecting an element
    if (runif(1) <= size / left){
      size = size - 1
      sel[aux] <- x[i]
      aux = aux + 1
    }

    # Break if all k entries have been selected
    if (size == 0){
      break
    }

    # Update the remaining number of entries
    left = left - 1
  }

  # Return selected values (k)
  return(sel)
}
```

2. The second function $PIKKV2()$ is very simple but powerful. In this case, we are generating a set of k random numbers between 1 and the length of the x vector, such that we end up with a k size array. Then, a ceiling operation is performed to all the elements of the vector, obtaining integer values. This allows us to use the $unique()$ function in order to keep only the unique elements of the array. If the size of the resulting vector is k, we have k different indexes and the resulting vector is returned, otherwise, a new vector is generated until the k unique elements condition is satisfied.

   Clearly, this function will not have a very good performance when increasing the value of k (when it tends to $length(x)$) since the probabilities of obtaining k unique values are decreasing. However, since we are going to concentrate on values of k where k is much less than n (as indicated in the problem statement), it will reach a very good performance in comparison to the original implementation.

   Note that we use the ceiling function for showing purposes only since we would be able to skip it by remembering that non-integer values are rounded down (floor operation) automatically when they are used as indexes. In the same logic, all functions can be binarized (using the R compiler) in order to obtain an extra speed-up, however, since we want to test pure R code we are not implementing this "trick" in our analysis.

```
# PIKK Version 2 function
PIKKV2 <- function(x, k){
  # While loop: True until condition is satisfied
  while(TRUE){

    # Indexes array (unique elements) is generated
    indexes <- unique(ceiling(runif(k, 0, length(x))))

    # If size is k: stop, otherwise repeat procedure
```

```r
    if (length(indexes) == k) {
      break
    }
  }

  # Return relevant (k) entries
  return(x[indexes])
}
```

3. Based on the previous function, we define $PIKKVF()$ function with a different approach for dealing with the remaining indexes where no unique k indexes are generated. In this case, we include two hyper-parameters that will add flexibility to the proposed function: (1) A *threshold* value that determines when to generate more than k numbers based on the proportion of total elements (size of x) and k in such a way that if k is larger, then we will have a better chance of getting k unique random numbers from a larger set, and (2) *tune* parameter that determines the number of extra random numbers to be generated. Its current value (6) was obtained based on a series of experiments as can be seen in the optimizePIKKVF.R file inside the Github repository.

   Based on its structure, it would clearly have a better performance than the previous algorithms when dealing with higher values for k, thanks to the new extra checking step.

```r
# PIKK VF function
PIKKVF <- function(y, k, threshold = 800, tune = 0){
  # Check if extra terms are added for increasing the probability of success
  ifelse(length(y)/k <= threshold, tune <- k/6, tune <- 0)

  # Indexes loop
  while (TRUE){
    indexes = unique(ceiling(runif(k + tune, 0, length(y)) ))
    if (length(indexes) >= k){
      return(y[indexes[1:k]])
    }
  }
}
```

4. For completeness and comparison purposes, we define a wrapper for the R *sample*() function, such that we can compare all the proposed functions and the original one provided in the statement of the problem with this internal (and very efficient) sampling function. This will be the first step before performing a detailed comparison between our best function and the default R *sample*() function in section b.1).

```r
# R Sample function wrapper
RSample <- function(x, k){
  return(sample(x = x, k, replace = FALSE))
}
```

5. Finally, we declare the original function in order to be able to produce the comparisons.

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

6. In order to perform the comparisons, we proceed as follows:

   i) Preliminary microbenchmark tests and plot are developed comparing all functions (including the PIKKV1 poor performance function).

   ii) A formal comparison for a series of values of k and n are performed for: $PIKK$, $PIKKV2$, $PIKKVF$, and $RSample()$ functions. Different values of n and k will be tested, focusing our analysis in values of k that are much less than n.

   iii) Specific comparison between our best performance function (as we will see, $PIKKVF()$) and the very efficient $RSample()$ function will be performed following the same analysis for k and n values (section b).

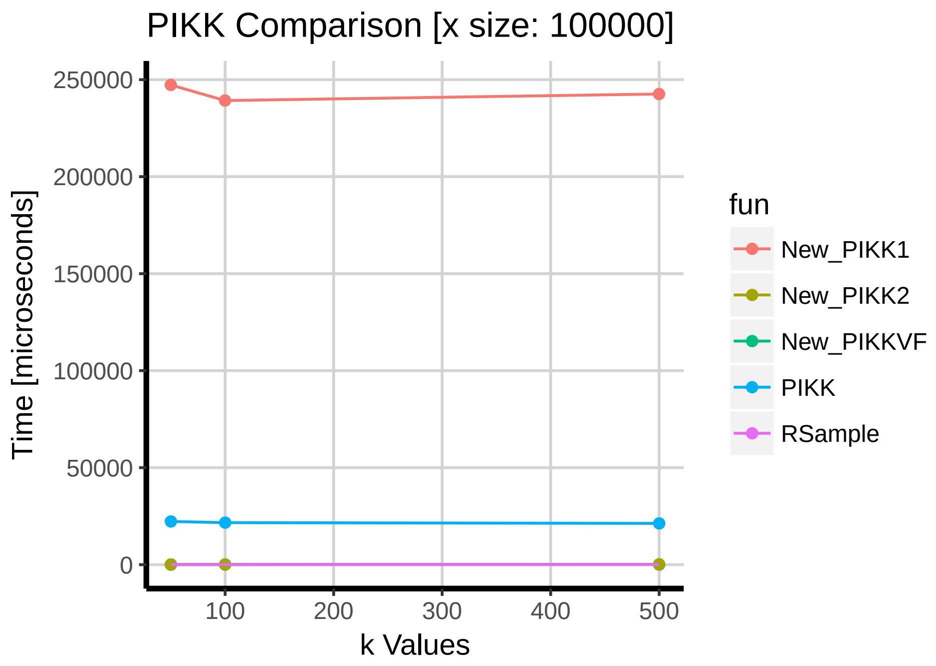
   Thus, we will declare and load a series of plotting functions (for comparison purposes) that are available inside the file plotPIKKVF.R in the Github repository. These functions can be found in the Appendix section. For completeness, we also include a microbenchmark wrapper function that can be easily used with a *lapply*() command for generating a series of comparisons (check appendix for more details).

```r
# Load relevant plot functions
options(warn=-1)
setwd("C:/Users/chile/Desktop/Stats243/HW/HW4/Code/Submitting/")
source("plotPIKKVF.R")

# Microbenchmark wrapper for comparisons: inputs(vector, number of elements,
# time unit string, times)
MicroExp <- function(x, k, tu = NULL, ntimes = 100){
  if (!is.null(tu)){
    # Print results using specific time unit
    print(microbenchmark(PIKK(x, k), PIKKV1(x, k), PIKKV2(x, k),
                         PIKKVF(x, k), RSample(x, k), unit = tu,
                         times = ntimes))
  }

  # Print results using default time unit
  else{
    print(microbenchmark(PIKK(x, k), PIKKV1(x, k), PIKKV2(x, k),
                         PIKKVF(x, k), RSample(x, k), times = ntimes))
  }
}
```

7. Preliminary comparisons are performed following the values recommendation from the problem statement. Plots are generated for visualization purposes.

```r
# Loading libraries
library("microbenchmark")
library("ggplot2")
library("tidyr")

## Preliminary Test
#Testing Data: x size 1e5
x <- 1:1e5

# Up to k = 500
ks  <- c(50, 100, 500)
```

```r
# All values plot & microbenchmark comparison
plotComparisonPIKKAll_A(x, ks, ntimes = 5)
```

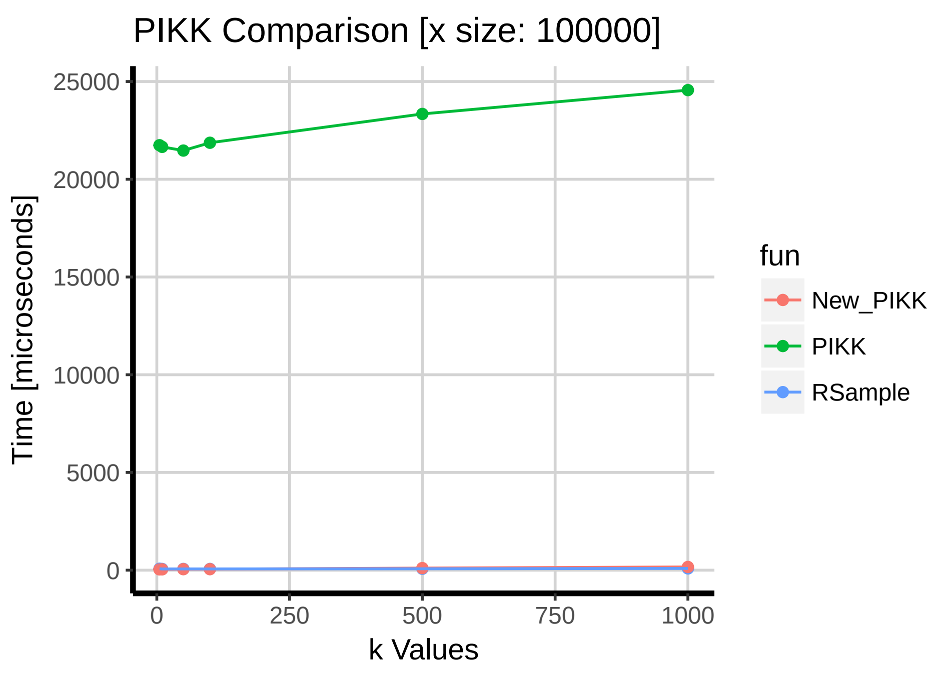


Looking at the results, we can easily check our initial thought about our $PIKKV1()$ function: it is by far the less efficient of all the tested ones. Important is to notice that both $PIKKV2()$ and $PIKKVF()$ reached a far better performance than the original $PIKK()$ function. In addition, we can clearly see that these two functions follow a similar pattern to the one showed by the $RSample()$ function, telling us that we should focus our analysis on them.

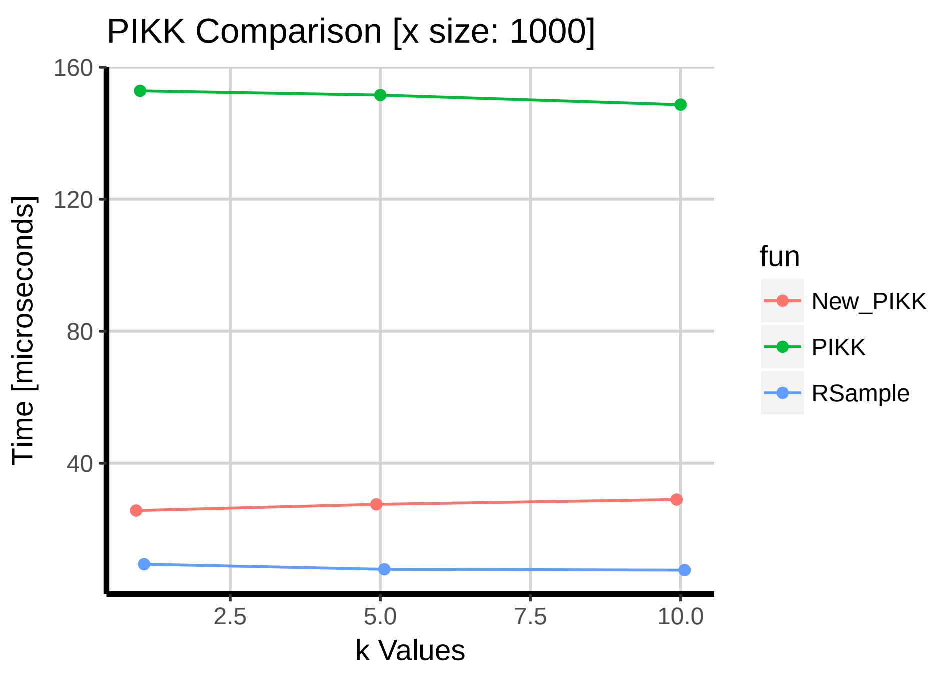
The microbenchmark analysis are very clear: (1) $PIKKV1()$ is the worst function (in terms of performance) by 1 order of magnitude in comparison to the original $PIKK()$ function, (2) the original function is the second worst in terms of running time, reaching mean values around 24 milliseconds while (3) $PIKKV2()$ needs around 11 milliseconds and (4) $PIKKVF()$ only needs around 140 microseconds, pretty similar to the performance reached by the very efficient $RSample()$ function.

Thanks to this preliminary analysis, we are able to illustrate how important is the structure of a function and the approach for solving the problem under study in order to reach the best possible performance. Now, we will perform the rest of our analysis without taking $PIKKV1()$ function into account, due to its poor performance. Under the same logic, since $PIKKVF()$ is an enhanced version of $PIKKV2()$, we will also drop this last function from the rest of our analysis.

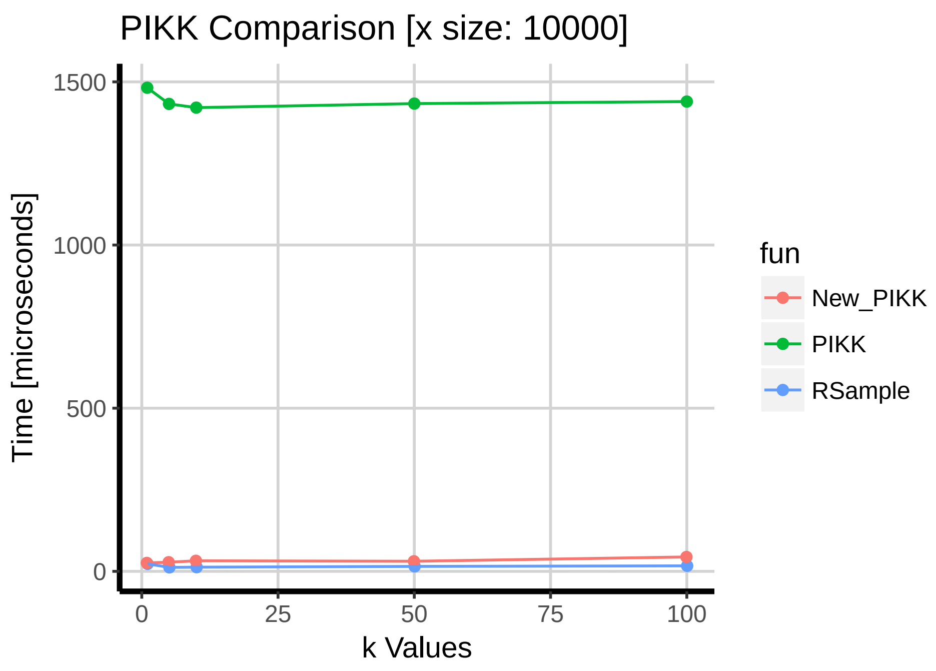
```r
# Focus on cases up to 1/100 of the total size without PIKKV1()
ks  <- c(5, 10, 50, 100, 500, 1000)

# All values plot & microbenchmark comparison for the largest k
plotComparisonAll(x, ks, ntimes = 5)
```



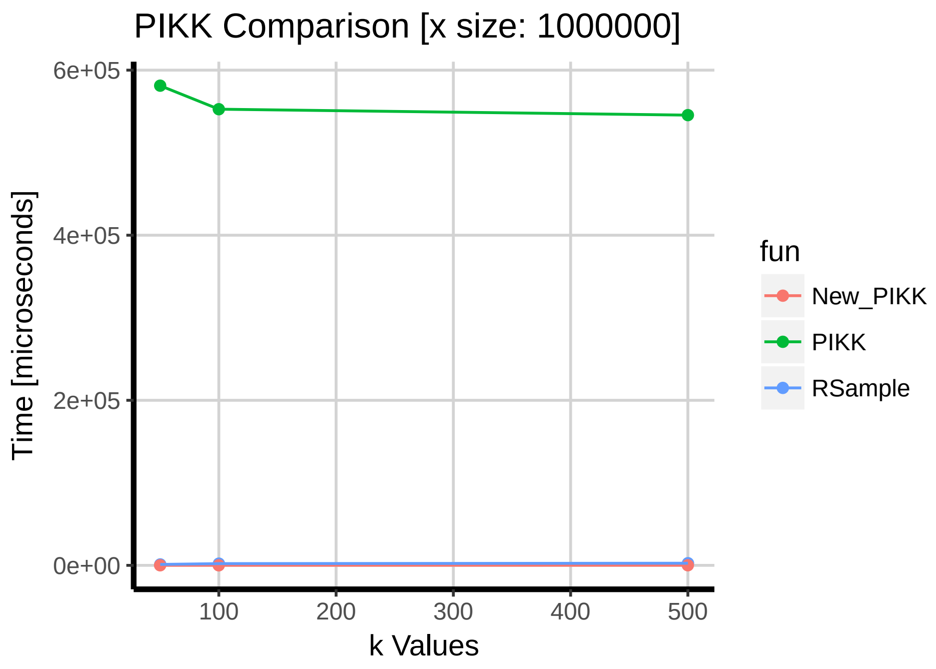
```r
# Testing Data: x size 1e3
x <- 1:1e3

# Up to k = 10
ks  <- c(1, 5, 10)

# All values plot & microbenchmark comparison for the largest k
plotComparisonAll(x, ks, ntimes = 5)
```



```
# Testing Data: x size 1e4
x <- 1:1e4

# Cases up to k = 100
ks  <- c(1, 5, 10, 50, 100)

# All values plot & microbenchmark comparison
plotComparisonAll(x, ks, ntimes = 5)
```



```
## Testing Data: x size 1e6
x <- 1:1e6

# Up to k = 500
ks  <- c(50, 100, 500)

# All values plot & microbenchmark comparison
plotComparisonAll(x, ks, ntimes = 5)
```

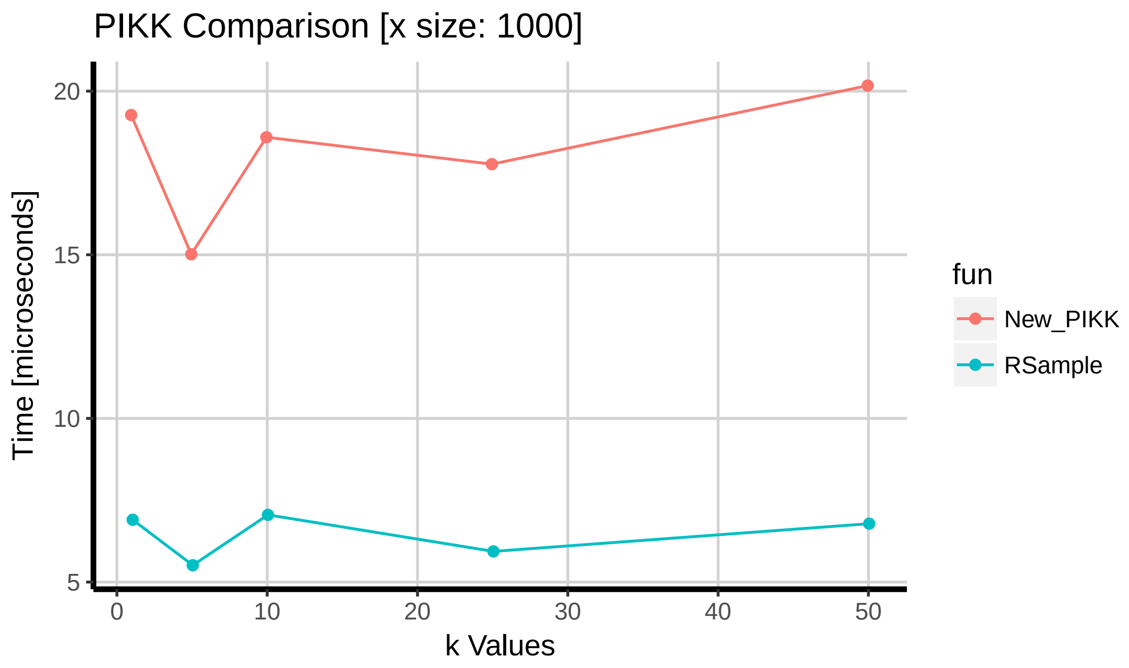


Based on all the previous results, we can conclude that the $PIKKVF()$ function is far better in terms of running time performance in comparison to the original $PIKK()$ function. Furthermore, we see that it follows a very similar performance pattern to the one obtained by the $RSample()$ function, allowing us to perform a specific analysis of these two functions in the next section, where we will check that $PIKKVF()$ will never take longer than 3-4 times the time of the $RSample()$ function when solving instances where k is much less than n as expected in the problem statement.

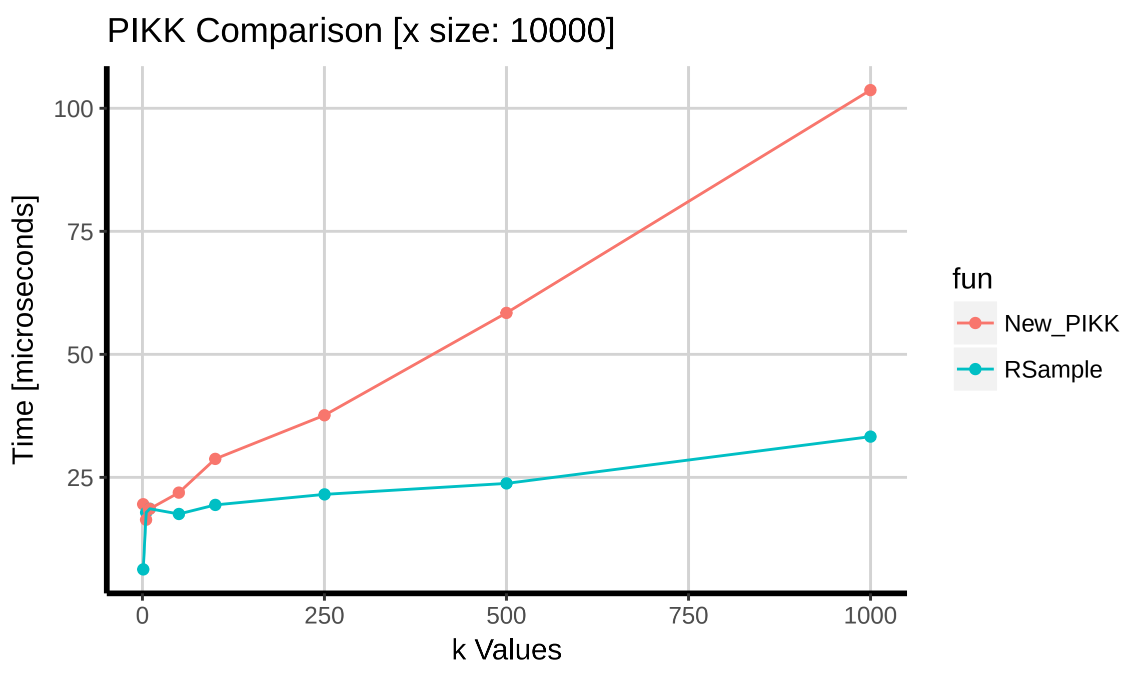
b.1) PIKKVF vs RSample function

In order to perform a specific comparison analysis between these two functions, we perform similar experiments to the ones shown above for n and k. A series of plots are included for simplicity instead of multiple microbenchmark outputs.

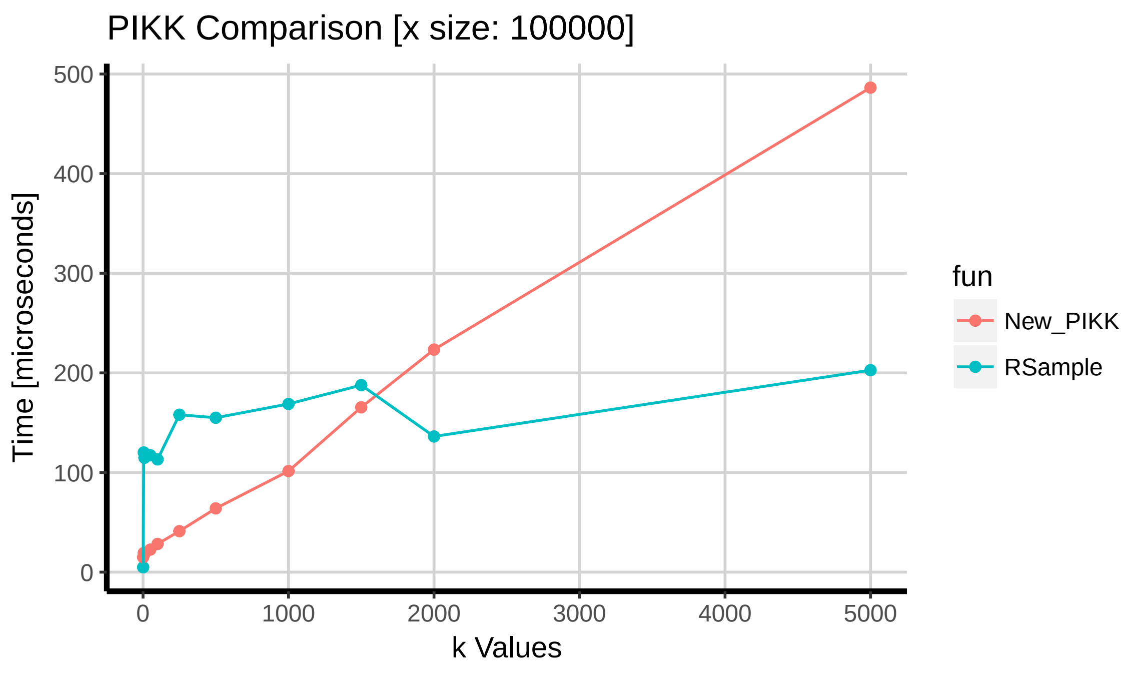
```
# Testing Data: x size 1e3
x <- 1:1e3

# Up to k = 50
ks  <- c(1, 5, 10, 25, 50)

# Plot comparison vs RSample
plotComparison(x, ks, ntimes = 100)
```



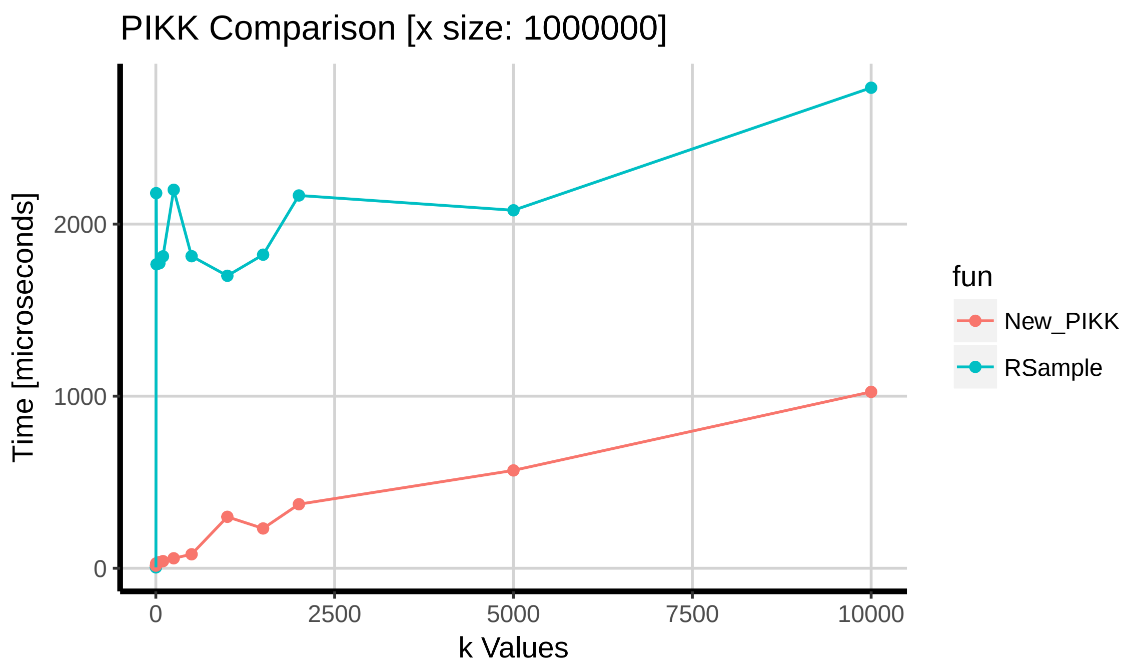
```
# Testing Data: x size 1e4
x <- 1:1e4

# Up to k = 1000
ks  <- c(1, 5, 10, 50, 100, 250, 500, 1000)

# Plot & microbenchmark comparisons
plotComparison(x, ks, ntimes = 100)
```



```r
# Testing Data: x size 1e5
x <- 1:1e5

# Up to k = 5000
ks  <- c(1, 5, 10, 50, 100, 250, 500, 1000, 1500, 2000, 5000)

# Plot & microbenchmark comparisons
plotComparison(x, ks, ntimes = 100)
```



```
## Testing Data: x size 1e6
x <- 1:1e6

# Up to k = 10000
ks  <- c(1, 5, 10, 50, 100, 250, 500, 1000, 1500, 2000, 5000, 10000)

# Plot & microbenchmark comparisons
plotComparison(x, ks, ntimes = 100)
```



Based on the previous results, it is clear that our $PIKKVF()$ function is able to "defeat" the $RSample()$ function for some instances and it is always below 3-4 times the running time of this very efficient function for instances where k is much less than n, as asked and expected. In addition, we can notice that our function performance is better in comparison to the $RSample()$ function when we increase the size of the vector x (and vice versa).

a.2) Improving FYKD function

Looking at the original function, we can clearly identify some performance issues that can be tackled in order to improve the formulation: (1) n random numbers following a specific rule are generated inside a for loop that clearly is not the most efficient implementation since it can be easily vectorized and (2) the swap operation (exchanging indexes i and j values) is not being done in place, requiring extra step calculations and memory usage instead of performing the replacement in one simple expression by using an auxiliary function or a basic R function such as $replace()$. As with $PIKK()$, these two operations can be easily improved (in terms of running time), obtaining a better formulation than the original (naive) one.

Following the same analysis as before, we implement three alternative/modified functions for performing similar operations as the original function $FYKD()$.

i) $FYKDV1()$: Simple improvement based on the needed calculations for generating the same output as the original function.

ii) $FYKDV2()$: Change value-in-place function using an auxiliary $swap()$ function.

iii) $FYKDV3()$: Alternative version of $FYKDV2()$ function using an R replacement function for changing values-in-place.

1. In this first improved function, we exploit the fact that the function returns/shows only the first k values, then no extra calculations are needed for $k+1$, ..., n. Clearly, this small improvement will not be useful when k is similar to n but can save a significant amount of time when $k << n$ since the for loop would be far shorter than in the original formulation.

```
# First new version: FYKDV1 function
FYKDV1 <- function(x, k) {

  # Length of the vector
  n <- length(x)

  # Loop up to k instead of n
  for(i in 1:k) {
    j = sample(i:n, 1)
    tmp <- x[i]
    x[i] <- x[j]
    x[j] <- tmp
  }

  # Return the relevant values
  return(x[1:k])
}
```

2. Using a different approach, we define an auxiliary function that allows us to swap values inside a vector without creating copies of it (change-in-place values). In addition, k random numbers are generated outside the original for loop exploiting the vectorized operations inside the $runif()$ function, obtaining an equivalent expression to the one presented in the original function. Hence, we perform a loop for swapping the values but not for creating any new variables/parameters.

```
# Auxiliary swap function
swap <- function(x,i,j) {

  # Swap the values of x[i] and x[j]
  x[c(i,j)] <- x[c(j,i)]

  # Return the new vector
  return(x)
}

# FYDV second version
FYKDV2 <- function(x, k) {

  # Generate k random numbers outside the loop, following the same rules as in
  # the original function.
  j = runif(k, 1:k, length(x))

  # Swap in place loop
  for(i in 1:k) {
    if(i != j[i]) {
      x = swap(x, i, j[i])
      }
  }

  # Return relevant values
  return(x[1:k])
}
```

3. Similar to the previous function, we use the basic *replace*() function of R instead of our auxiliary function *swap*().

```
# FYKDV third version
FYKDV3 <- function(x, k) {

  # Generate k random numbers outside the loop, following the same rules as in
  # the original function.
  j = runif(k, 1:k, length(x))

  # Replace in place loop
  for(i in 1:k) {
    if (i != j[i]){
      x = replace(x, c(i, j[i]), x[c(j[i], i)])
    }
  }

  # Return relevant values
  return(x[1:k])
}
```

4. As before, we define a wrapper function for microbenchmark including the functions we want to compare and we perform a series of tests.

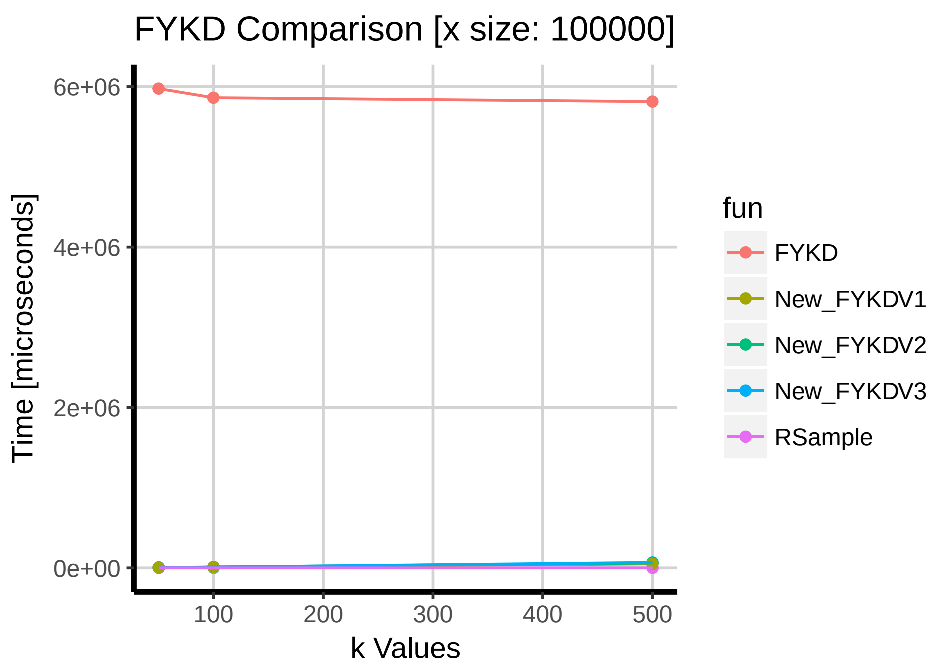
```
# Load relevant plot functions
setwd("C:/Users/chile/Desktop/Stats243/HW/HW4/Code/Submitting/")
source("plotFYKDVF.R")
source("RSample.R")

# Microbenchmark wrapper for comparisons: inputs(vector, number of elements,
# time unit string, times)
MicroExp2 <- function(x, k, tu = NULL){
  if (!is.null(tu)){
    print(microbenchmark(FYKD(x, k), FYKDV1(x, k), FYKDV2(x, k),
                         FYKDV3(x, k), RSample(x, k), unit = tu))
  }
  else{
    print(microbenchmark(FYKD(x, k), FYKDV1(x, k), FYKDV2(x, k),
                         FYKDV3(x, k), RSample(x, k)))
  }
}
```

5. Before performing the comparisons, we declare the original function (AS-IS, no improved style).

```r
# Original function
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]) }
```

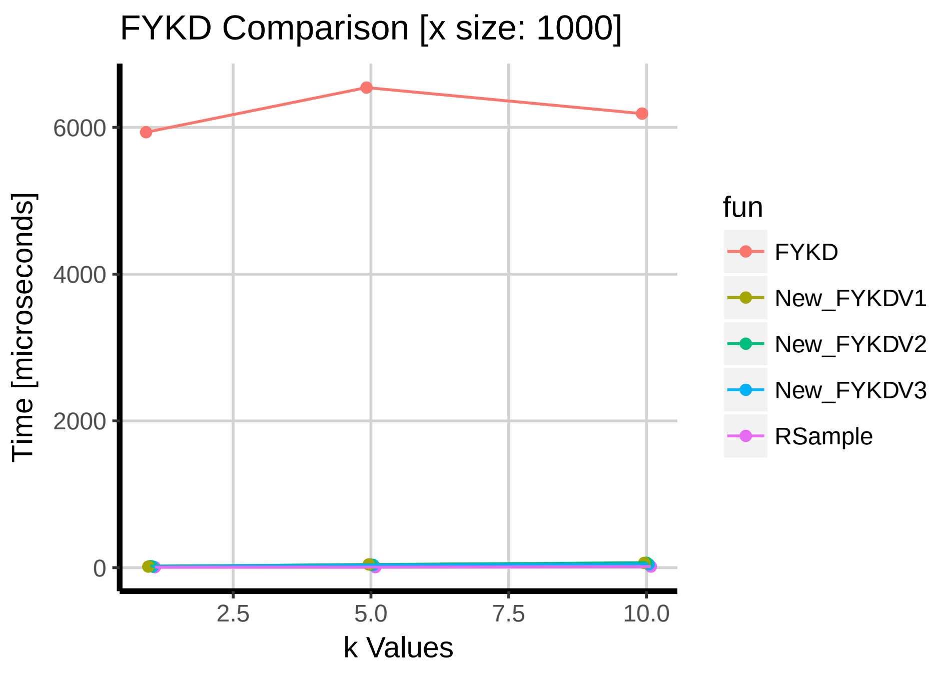
6. Comparisons are performed. Plots are generated for visualization purposes.

```r
# Loading libraries
library("microbenchmark")
library("ggplot2")
library("tidyr")

## Preliminar Test
#Testing Data: x size 1e5
x <- 1:1e5

# Up to k = 500
ks  <- c(50, 100, 500)

# All values plot & microbenchmark comparison
plotComparisonFYKD_All_A(x, ks, ntimes = 5)
```

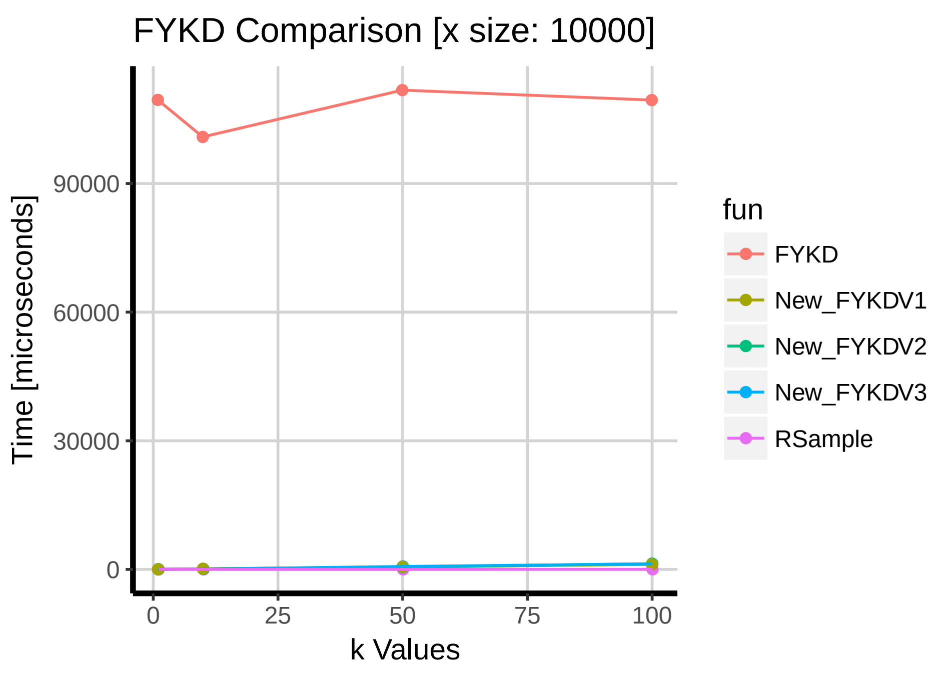


```r
# Testing Data: x size 1e3
x <- 1:1e3
```

```
# Up to k = 10
ks  <- c(1, 5, 10)

# All values plot & microbenchmark comparison
plotComparisonFYKD_All_A(x, ks, ntimes = 5)
```



```
# Testing Data: x size 1e4
x <- 1:1e4

# Up to k = 100
ks  <- c(1, 10, 50, 100)

# All values plot & microbenchmark comparison
plotComparisonFYKD_All_A(x, ks, ntimes = 5)
```



Based on all the previous results, we can conclude that the $FYKDV2()$ and $FYKDV3()$ functions are far better in terms of running time performance in comparison to the original $FYKD()$ function. However, we do not see that they follow a very similar performance pattern to the one obtained by the $RSample()$ function. In order to tackle this situation, we will perform further modifications/improvements to both functions, allowing us to perform a specific analysis in the next section, where we will check that our new implementations will never take longer than 3-4 times the time of the $RSample()$ function when solving instances where k is much less than n as expected in the problem statement.

b.2) Improved FYKDV2 and V3 versions vs RSample function

Based on the results above, it is clear that both functions $FYKDV2()$ and $FYKDV3()$ cannot reach the same performance as the very efficient $RSample()$ function. Therefore, here we propose a new version (improved) of these two functions taking advantage of the very efficient $PIKKVF()$ function we developed in the previous section. In this case, we will generate a series of k random numbers thanks to this function and then we will swap/replace values inside the original vector x in order to replicate the shuffling approach of the original function, obtaining a subset of a permutation of the original vector given as an input.

In order to perform a specific comparison analysis between these three functions, we perform similar experiments to the ones shown above including new values for n and k. A series of plots are included for simplicity instead of multiple microbenchmark outputs.

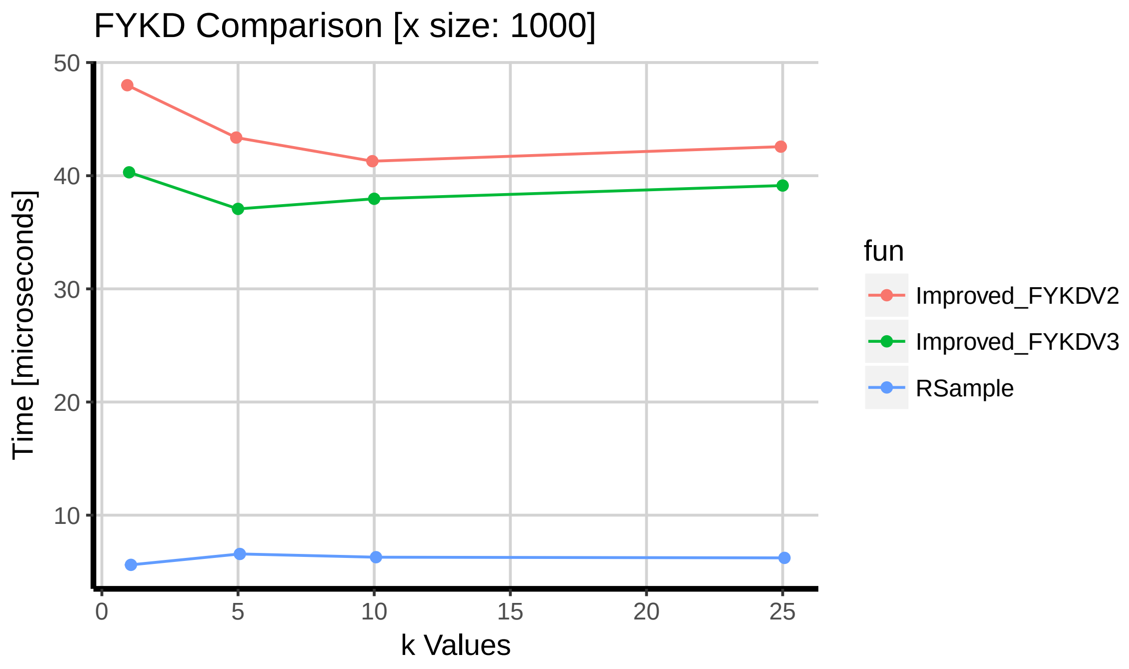
1. The improved versions are using $PIKKVF()$ as an auxiliary variable for generating k unique random numbers. Once they are generated, a vectorized version of $swap()/replace()$ functions is applied avoiding the inefficient for loop originally included in our functions (and the original method). An extra checking step is needed in order to take into account the possibility of repeating numbers due to the shuffling structure of our $swap()$ or R basic $replace()$ functions.

```r
# FYDV second version improved
FYKD2_V2 <- function(x, k, threshold = 800) {

  while (TRUE){
    # Generate k random numbers outside the loop, using the efficient PIKKVF
    # function already implemented.
    j = PIKKVF(x, k, threshold)

    # List with indexes to exchange
    i = 1:k
    l1 = c(rbind(i, j))
    l2 = c(rbind(j, i))

    # Replace in place (vectorized swap version)
    x = swap(x, l1, l2)

    # If we have k different numbers (at least), break
    if (length(unique(x[1:k])) >= k){
      break
    }
  }
  # Return relevant values
  return(x[1:k])
}
```

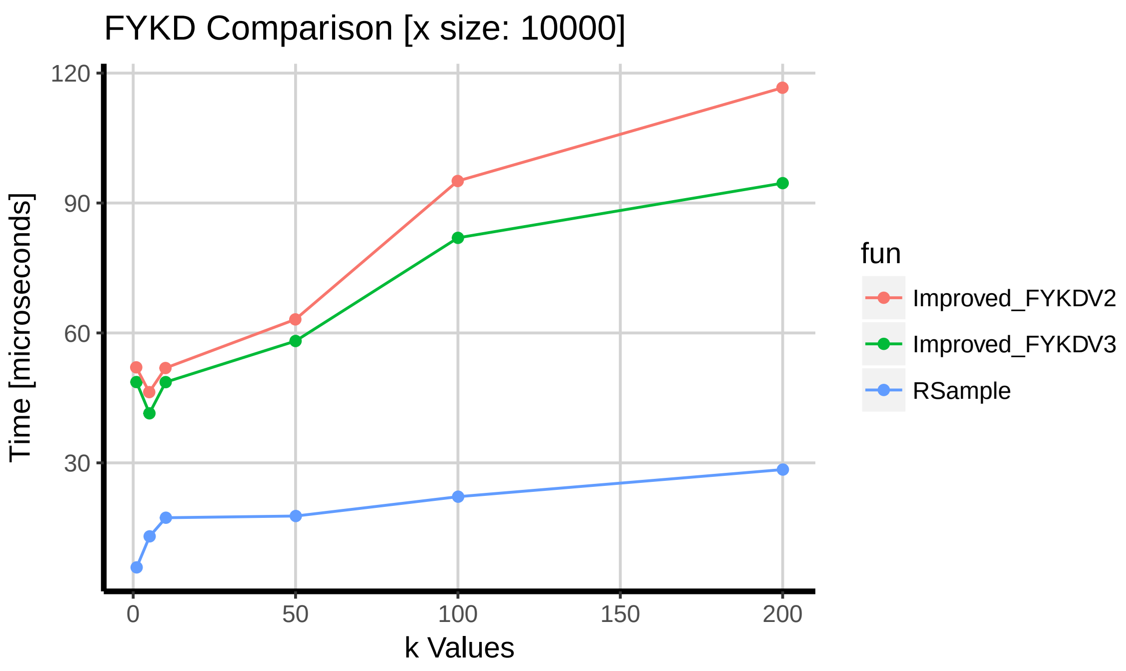
```r
# FYDV third version improved
FYKD3_V3 <- function(x, k, threshold = 800) {

  while (TRUE){
    # Generate k random numbers outside the loop, using the efficient PIKKVF
    # function already implemented.
    j = PIKKVF(x, k, threshold)

    # List with indexes to exchange
    i = 1:k
    l1 = c(rbind(i, j))
    l2 = c(rbind(j, i))

    # Replace in place (vectorized replace version)
    x = replace(x, l1, x[l2])

    # If we have k different numbers (at least), break
    if (length(unique(x[1:k])) >= k){
      break
    }
  }
  # Return relevant values
  return(x[1:k])
}
```

2. Now, we are ready to perform the comparisons:

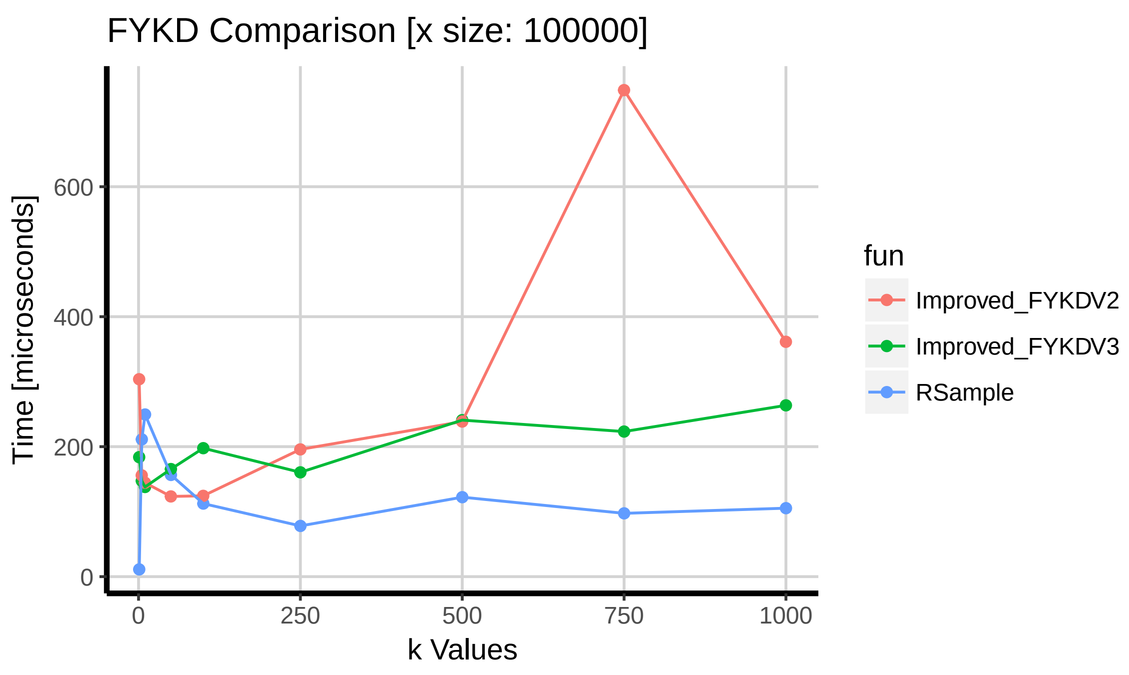
```
# Loading PIKKVF function
setwd("C:/Users/chile/Desktop/Stats243/HW/HW4/Code/Submitting/")
source("PIKKVF.R")

# Testing Data: x size 1e3
x <- 1:1e3

# Up to k = 25
ks  <- c(1, 5, 10, 25)

# Plot comparison vs RSample
plotComparisonFYKD_B(x, ks, ntimes = 100)
```



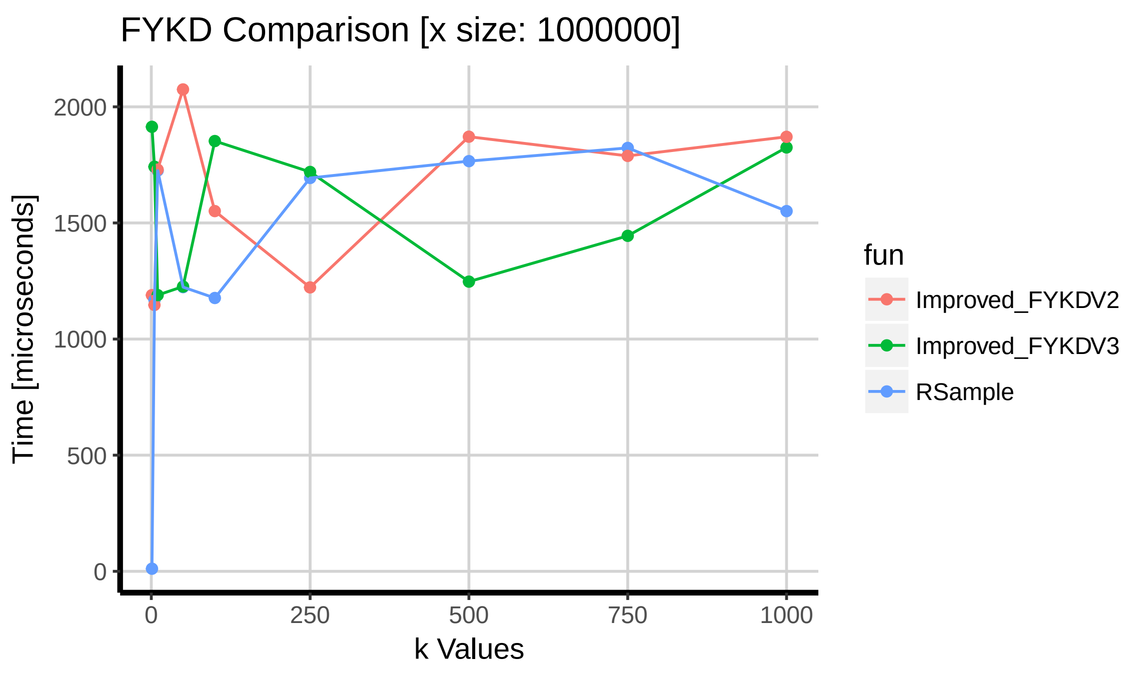
```
# Testing Data: x size 1e4
x <- 1:1e4

# Up to k = 100
ks  <-  c(1, 5, 10, 50, 100, 200)

# Plot & microbenchmark comparisons
plotComparisonFYKD_B(x, ks, ntimes = 100)
```



```
# Testing Data: x size 1e5
x <- 1:1e5

# Up to k = 500
ks  <- c(1, 5, 10, 50, 100, 250, 500, 750, 1000)

# Plot & microbenchmark comparisons
plotComparisonFYKD_B(x, ks, ntimes = 100)
```



```r
## Testing Data: x size 1e6
x <- 1:1e6

# Up to k = 500
ks  <- c(1, 5, 10, 50, 100, 250, 500, 750, 1000)

# Plot & microbenchmark comparisons
plotComparisonFYKD_B(x, ks, ntimes = 100)
```



Based on the previous results, it is clear that our our improved functions are do not take longer than 3-4 times the time of the $RSample()$ function for instances where k is much less than n, as asked and expected. In addition, we can notice that our function performance is better in comparison to the $RSample()$ function when we increase the size of the vector x (and vice versa).

Appendix

a) Plot Functions

```r
# Plot function: PIKKVF vs RSample
plotComparison <- function(x, ks, ntimes = 100, threshold = 800){

  # Define times as x-labels for each k value
  times <- sapply(ks, function(k) {

    # Create labels and extract the relevant information
    op <- microbenchmark(
      New_PIKK = PIKKVF(x, k, threshold),
      RSample  = RSample(x, k),
      times = ntimes
    )

    by(op$time, op$expr, function(t) mean(t) / 1000)
  }
  )

  # Transponse k-values and create a data frame
  times <- t(times)
  times <- as.data.frame(cbind(times, k = ks))

  # Define the time values and keys of the plot
  times <- gather(times, -k, key = "fun", value = "time")
  pd <- position_dodge(width = 0.2)

  # Create the plot
  ggplot(times, aes(x = k, y = time, group = fun, color = fun)) +
    theme(axis.line = element_line(size = 1, colour = "black"),
          panel.grid.major = element_line(colour = "#d3d3d3"),
          panel.grid.minor = element_blank(),
          panel.border = element_blank(), panel.background = element_blank()) +

    geom_point(position = pd) +
    geom_line(position = pd) +

    xlab("k Values") +
    ylab("Time [microseconds]") +

    ggtitle(paste("PIKK Comparison [x size: ", as.character(length(x)), "]",
                  sep = ""))
}
```

```r
# Plot function (With original PIKK)
plotComparisonAll <- function(x, ks, ntimes = 100, threshold = 800){

  # Define times as x-labels for each k value
  times <- sapply(ks, function(k) {

    # Create labels and extract the relevant information
    op <- microbenchmark(
      PIKK     = PIKK(x, k),
      New_PIKK = PIKKVF(x, k, threshold),
      RSample  = RSample(x, k),
      times = ntimes
    )

    by(op$time, op$expr, function(t) mean(t) / 1000)
  }
  )

  # Transponse k-values and create a data frame
  times <- t(times)
  times <- as.data.frame(cbind(times, k = ks))

  # Define the time values and keys of the plot
  times <- gather(times, -k, key = "fun", value = "time")
  pd <- position_dodge(width = 0.2)

  # Create the plot
  ggplot(times, aes(x = k, y = time, group = fun, color = fun)) +
    theme(axis.line = element_line(size = 1, colour = "black"),
          panel.grid.major = element_line(colour = "#d3d3d3"),
          panel.grid.minor = element_blank(),
          panel.border = element_blank(), panel.background = element_blank()) +

    geom_point(position = pd) +
    geom_line(position = pd) +

    xlab("k Values") +
    ylab("Time [microseconds]") +

    ggtitle(paste("PIKK Comparison [x size: ", as.character(length(x)), "]",
                  sep = ""))
}
```

```r
# Plot function (All PIKK functions)
plotComparisonPIKKAll_A <- function(x, ks, ntimes = 100, threshold = 800){

  # Define times as x-labels for each k value
  times <- sapply(ks, function(k) {

    # Create labels and extract the relevant information
    op <- microbenchmark(
      PIKK     = PIKK(x, k),
      New_PIKK1 = PIKKV1(x, k),
      New_PIKK2 = PIKKV2(x, k),
      New_PIKKVF = PIKKVF(x, k, threshold),
      RSample  = RSample(x, k),
      times = ntimes
    )

    by(op$time, op$expr, function(t) mean(t) / 1000)
  }
  )

  # Transponse k-values and create a data frame
  times <- t(times)
  times <- as.data.frame(cbind(times, k = ks))

  # Define the time values and keys of the plot
  times <- gather(times, -k, key = "fun", value = "time")
  pd <- position_dodge(width = 0.2)

  # Create the plot
  ggplot(times, aes(x = k, y = time, group = fun, color = fun)) +
    theme(axis.line = element_line(size = 1, colour = "black"),
          panel.grid.major = element_line(colour = "#d3d3d3"),
          panel.grid.minor = element_blank(),
          panel.border = element_blank(), panel.background = element_blank()) +

    geom_point(position = pd) +
    geom_line(position = pd) +

    xlab("k Values") +
    ylab("Time [microseconds]") +

    ggtitle(paste("PIKK Comparison [x size: ", as.character(length(x)), "]",
                  sep = ""))
}
```

```r
# Plot function
plotComparisonFYKD_B <- function(x, ks, ntimes = 100, threshold = 800){

  # Define times as x-labels for each k value
  times <- sapply(ks, function(k) {

    # Create labels and extract the relevant information
    op <- microbenchmark(
      Improved_FYKDV2 = FYKD2_V2(x, k, threshold),
      Improved_FYKDV3 = FYKD3_V3(x, k, threshold),
      RSample  = RSample(x, k),
      times = ntimes
    )

    by(op$time, op$expr, function(t) mean(t) / 1000)
  }
  )

  # Transponse k-values and create a data frame
  times <- t(times)
  times <- as.data.frame(cbind(times, k = ks))

  # Define the time values and keys of the plot
  times <- gather(times, -k, key = "fun", value = "time")
  pd <- position_dodge(width = 0.2)
  # Create the plot
  ggplot(times, aes(x = k, y = time, group = fun, color = fun)) +
    theme(axis.line = element_line(size = 1, colour = "black"),
          panel.grid.major = element_line(colour = "#d3d3d3"),
          panel.grid.minor = element_blank(),
          panel.border = element_blank(), panel.background = element_blank()) +

    geom_point(position = pd) +
    geom_line(position = pd) +

    xlab("k Values") +
    ylab("Time [microseconds]") +

    ggtitle(paste("FYKD Comparison [x size: ", as.character(length(x)), "]",
                  sep = ""))
}
```

```r
# Plot function (With original FYKD and improved V2, V3 functions)
plotComparisonFYKD_All_AB <- function(x, ks, ntimes = 100, threshold = 800){

  # Define times as x-labels for each k value
  times <- sapply(ks, function(k) {

    # Create labels and extract the relevant information
    op <- microbenchmark(
      FYKD     = FYKD(x, k),
      Improved_FYKDV2 = FYKD2_V2(x, k, threshold),
      Improved_FYKDV3 = FYKD3_V3(x, k, threshold),
      RSample   = RSample(x, k),
      times = ntimes
    )

    by(op$time, op$expr, function(t) mean(t) / 1000)
  }
  )

  # Transponse k-values and create a data frame
  times <- t(times)
  times <- as.data.frame(cbind(times, k = ks))

  # Define the time values and keys of the plot
  times <- gather(times, -k, key = "fun", value = "time")
  pd <- position_dodge(width = 0.2)

  # Create the plot
  ggplot(times, aes(x = k, y = time, group = fun, color = fun)) +
    theme(axis.line = element_line(size = 1, colour = "black"),
          panel.grid.major = element_line(colour = "#d3d3d3"),
          panel.grid.minor = element_blank(),
          panel.border = element_blank(), panel.background = element_blank()) +

    geom_point(position = pd) +
    geom_line(position = pd) +
    xlab("k Values") +
    ylab("Time [microseconds]") +

    ggtitle(paste("FYKD Comparison [x size: ", as.character(length(x)), "]", sep = ""))
}
```

```r
# Plot function (With original FYKD)
plotComparisonFYKD_All_A <- function(x, ks, ntimes = 100, threshold = 800){

  # Define times as x-labels for each k value
  times <- sapply(ks, function(k) {

    # Create labels and extract the relevant information
    op <- microbenchmark(
      FYKD     = FYKD(x, k),
      New_FYKDV1 = FYKDV1(x, k),
      New_FYKDV2 = FYKDV2(x, k),
      New_FYKDV3 = FYKDV3(x, k),
      RSample   = RSample(x, k),
      times = ntimes
    )

    by(op$time, op$expr, function(t) mean(t) / 1000)
  }
  )

  # Transponse k-values and create a data frame
  times <- t(times)
  times <- as.data.frame(cbind(times, k = ks))

  # Define the time values and keys of the plot
  times <- gather(times, -k, key = "fun", value = "time")
  pd <- position_dodge(width = 0.2)

  # Create the plot
  ggplot(times, aes(x = k, y = time, group = fun, color = fun)) +
    theme(axis.line = element_line(size = 1, colour = "black"),
          panel.grid.major = element_line(colour = "#d3d3d3"),
          panel.grid.minor = element_blank(),
          panel.border = element_blank(), panel.background = element_blank()) +

    geom_point(position = pd) +
    geom_line(position = pd) +

    xlab("k Values") +
    ylab("Time [microseconds]") +

    ggtitle(paste("FYKD Comparison [x size: ", as.character(length(x)), "]",
                  sep = ""))
}
```