STAT243 - Problem Set 4 P1, P2

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 376269 20.1     592000 31.7   460000 24.6
## Vcells 579787  4.5    1308461 10.0   786432  6.0
```

```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   376612  20.2     750400  40.1   460000  24.6
## Vcells 50579986 385.9   73237641 558.8 50590171 386.0
```

```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   376898  20.2     750400  40.1   460000  24.6
## Vcells 50580479 385.9   73237641 558.8 50643715 386.4
```

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

```
##            used  (Mb) gc trigger  (Mb) max used  (Mb)
## Ncells   376913  20.2     750400  40.1   460000  24.6
## Vcells 50580498 385.9   73237641 558.8 50643715 386.4
```

```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    376952   20.2     750400   40.1    460000   24.6
## Vcells 150580544 1148.9  217238427 1657.4 150601500 1149.0
```

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

```
##          used (Mb) gc trigger   (Mb) max used (Mb)
## Ncells 377099 20.2     750400   40.1   377099 20.2
## Vcells 580804  4.5  173790741 1326.0   580804  4.5
```

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:

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

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

```
## 400 MB
```

```
tracemem(x)
```

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

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

```
## 800 MB
```

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

```
##          used (Mb) gc trigger   (Mb) max used (Mb)
## Ncells 381135 20.4     750400   40.1   381135 20.4
## Vcells 583545  4.5  139032592 1060.8   583545  4.5
```

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.

```
})
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 384721 20.6     750400   40.1   384721 20.6
## Vcells 588467  4.5  144776339 1104.6   588467  4.5
```

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] 800012736
```

```
# 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 384819 20.6     750400   40.1   384819 20.6
## Vcells 588945  4.5  240781892 1837.1   588945  4.5
```

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] "<0000000014B437E0>"
```

```
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 384731 20.6     750400   40.1   384731 20.6
## Vcells 588590  4.5  192625513 1469.7   588590  4.5
```

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:

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

# Loading libraries
library("pryr")

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

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

```
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
```

```
object_size(myFun)
```

```
## 16.8 kB
```

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

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

```
##          used (Mb) gc trigger   (Mb) max used (Mb)
## Ncells 384859 20.6     750400   40.1   384859 20.6
## Vcells 589088  4.5  154100410 1175.7   589088  4.5
```

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   384937  20.6     750400   40.1   622529  33.3
## Vcells 60590198 462.3  154100410 1175.7 60747053 463.5
```

```
# 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] "<00000000122528E8>"
```

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

```
## 480 MB
```

```r
object_size(x)
```

```
## 400 MB
```

```r
object_size(y)
```

```
## 40 MB
```

```r
object_size(z)
```

```
## 40 MB
```

```r
gc()
```

```
##            used  (Mb) gc trigger   (Mb) max used  (Mb)
## Ncells   385049  20.6     750400   40.1   622529  33.3
## Vcells 60590314 462.3  154100410 1175.7 60747053 463.5
```

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

```
## @0x00000000122528e8 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))
```

```
## @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(z))
```

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

```r
# 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    385097   20.6     750400   40.1    622529   33.3
## Vcells 160590496 1225.3  252977297 1930.1 210600855 1606.8
```

```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))
```

```
## @0x00000000122528e8 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 385221 20.6     750400   40.1   385221 20.6
## Vcells 590654  4.6  202381837 1544.1   590654  4.6
```

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] "<000000001669DC00>"
```

```r
object_size(L)
```

```
## 440 MB
```

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

```
## @0x000000001669dc00 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   384760  20.6     750400   40.1   592949  31.7
## Vcells 55589453 424.2  161905469 1235.3 55735964 425.3
```

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

```
## tracemem[0x000000001669dc00 -> 0x00000000126fbfd0]: eval eval withVisible withCallingHandlers handle
```

```
gc()
```

```
##            used  (Mb) gc trigger   (Mb) max used  (Mb)
## Ncells   384757  20.6     750400   40.1   592949  31.7
## Vcells 60589471 462.3  161905469 1235.3 70610328 538.8
```

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

```
## @0x00000000126fbfd0 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 384799 20.6     750400  40.1   384799 20.6
## Vcells 589556  4.5  129524375 988.2   589556  4.5
```

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] "<00000000134BFC88>"
```

```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))
```

```
## @0x00000000134bfc88 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   385008  20.6     750400  40.1   660985  35.4
## Vcells 55590663 424.2  129524375 988.2 55830325 426.0
```

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

```
## tracemem[0x00000000134bfc88 -> 0x0000000016ed5020]: eval eval withVisible withCallingHandlers handle
## tracemem[0x00007ff5e7e30010 -> 0x00007ff5cda80010]: eval eval withVisible withCallingHandlers handle
## tracemem[0x00007ff5cda80010 -> 0x00007ff59df80010]: eval eval withVisible withCallingHandlers handle
```

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

```
## @0x0000000016ed5020 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,...
```

```
object_size(x)
```

```
## 400 MB
```

```
object_size(y)
```

```
## 40 MB
```

```
object_size(L)
```

```
## 840 MB
```

```
gc()
```

```
##              used   (Mb) gc trigger   (Mb)  max used   (Mb)
## Ncells    385053   20.6     750400   40.1    660985   35.4
## Vcells 155590837 1187.1  246977519 1884.3 205664427 1569.1
```

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

```
##          used (Mb) gc trigger   (Mb) max used (Mb)
## Ncells 385055 20.6     750400   40.1   385055 20.6
## Vcells 590846  4.6  197582015 1507.5   590846  4.6
```

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:

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

# Loading libraries
library("pryr")

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

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

```
tracemem(y)
```

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

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

```
## [1] "<0000000014B2BB28>"
```

```
tracemem(L2)
```

```
## [1] "<0000000014B2BB28>"
```

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

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

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

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

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

```
## @0x0000000014b2bb28 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))
```

```
## @0x0000000014b2bb28 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   377731  20.2     750400  40.1   490905  26.3
## Vcells 55583723 424.1   73242243 558.8 55754137 425.4
```

```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))
```

```
## @0x0000000014b2bb28 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,...
```

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

```
## @0x0000000014b2bb28 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,...
```

```
object_size(x)
```

```
## 800 MB
```

```
object_size(y)
```

```
## 40 MB
```

```
object_size(L)
```

```
## 440 MB
```

```
object_size(L2)
```

```
## 440 MB
```

```
gc()
```

```
##              used   (Mb) gc trigger   (Mb)  max used   (Mb)
## Ncells    377831   20.2     750400   40.1    490905   26.3
## Vcells 155583931 1187.1  246969203 1884.3 205678437 1569.3
```

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

```
##          used (Mb) gc trigger   (Mb) max used (Mb)
## Ncells 377828 20.2     750400   40.1   377828 20.2
## Vcells 583930  4.5  197575362 1507.4   583930  4.5
```

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.

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

# Loading libraries
library("pryr")

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

```
## [1] "<00000000168EF3C8>"
```

```
tracemem(L2)
```

```
## [1] "<00000000168EF3C8>"
```

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

```
## @0x00000000168ef3c8 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))
```

```
## @0x00000000168ef3c8 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   378530  20.3     750400   40.1   615546  32.9
## Vcells 55583523 424.1  158060289 1206.0 55775488 425.6
```

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

```
## tracemem[0x00000000168ef3c8 -> 0x0000000016c0ca08]: eval eval withVisible withCallingHandlers handle
```

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

```
## @0x0000000016c0ca08 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))
```

```
## @0x00000000168ef3c8 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,...
```

```r
object_size(L)
```

```
## 840 MB
```

```r
object_size(L2)
```

```
## 440 MB
```

```r
gc()
```

```
##              used   (Mb) gc trigger   (Mb)  max used   (Mb)
## Ncells    378593   20.3     750400   40.1    615546   32.9
## Vcells 155583664 1187.1  246968974 1884.3 205636212 1568.9
```

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

```
##          used (Mb) gc trigger   (Mb) max used (Mb)
## Ncells 378601 20.3     750400   40.1   378601 20.3
## Vcells 583681  4.5  197575179 1507.4   583681  4.5
```

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:

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

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

# Trace
tracemem(L1)
```

```
## [1] "<000000001210C888>"
```

```r
tracemem(L2)
```

```
## [1] "<0000000016780FE8>"
```

```r
tracemem(L)
```

```
## [1] "<000000001390B390>"
```

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

```
## @0x000000001210c888 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))
```

```
## @0x0000000016780fe8 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))
```

```
## @0x000000001390b390 19 VECSXP g0c2 [NAM(2),TR] (len=2, tl=0)
##   @0x000000001210c888 19 VECSXP g0c1 [NAM(2),TR] (len=1, tl=0)
##     @0x00007ff5e7e30010 13 INTSXP g0c7 [] (len=100000000, tl=0) 1,2,3,4,5,...
##   @0x0000000016780fe8 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   378647  20.3     750400   40.1   641738  34.3
## Vcells 55584395 424.1  158060143 1206.0 55812694 425.9
```

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

```
## tracemem[0x0000000016780fe8 -> 0x0000000016dc40d8]: eval eval withVisible withCallingHandlers handle
```

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

```
## @0x000000001210c888 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))
```

```
## @0x0000000016dd9028 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,...
##   @0x0000000016dc41f8 14 REALSXP g0c1 [] (len=1, tl=0) 1
```

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

```
## @0x000000001390b390 19 VECSXP g1c2 [MARK,NAM(2),TR] (len=2, tl=0)
##   @0x000000001210c888 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,...
##   @0x0000000016780fe8 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   378715  20.3     750400   40.1   641738  34.3
## Vcells 55584572 424.1  158060143 1206.0 55812694 425.9
```

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

```
##          used (Mb) gc trigger  (Mb) max used (Mb)
## Ncells 378711 20.3     750400  40.1   378711 20.3
## Vcells 584569  4.5  126448114 964.8   584569  4.5
```

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 384613 20.6     750400  40.1   488572 26.1
## Vcells 590672  4.6  101162120 771.9   654850  5.0
```

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

```
## @0x0000000016cd62f0 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
##   @0x00007ff5faf60010 14 REALSXP g0c7 [NAM(2)] (len=10000000, tl=0) 1.17482,0.18097,-1.22453,-1.2144
##   @0x00007ff5faf60010 14 REALSXP g0c7 [NAM(2)] (len=10000000, tl=0) 1.17482,0.18097,-1.22453,-1.2144
```

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

```
## 160000136 bytes
```

```
gc()
```

```
##            used (Mb) gc trigger  (Mb) max used (Mb)
## Ncells   385255 20.6     750400  40.1   488572 26.1
## Vcells 10591609 80.9   80929696 617.5 10667694 81.4
```

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)
```

```
## 160 MB
```

```
##          used (Mb) gc trigger  (Mb) max used (Mb)
## Ncells 381483 20.4     750400  40.1   381483 20.4
## Vcells 587206  4.5  101158491 771.8   587206  4.5
```

Therefore, we have checked our analysis.