Stat243: Problem Set 4

Linqing Wei

October 10, 2017

```
> #Problem 1(a)
> x <- 1:10
> .Internal(inspect(x))

@7ffd14ae7b48 13 INTSXP g0c4 [NAM(2)] (len=10, tl=0) 1,2,3,4,5,...

> f <- function(input){
+   data <- input
+   .Internal(inspect(data))
+   g <- function(param) return(param * data)
+   .Internal(inspect(data))
+   return(g)
+ }
> myFun <- f(x)

@7ffd14ae7b48 13 INTSXP g0c4 [NAM(2)] (len=10, tl=0) 1,2,3,4,5,...
@7ffd14ae7b48 13 INTSXP g0c4 [NAM(2)] (len=10, tl=0) 1,2,3,4,5,...

> data <- 100
> myFun(3)

 [1]  3  6  9 12 15 18 21 24 27 30
```

There's only one copy of vector x during the execution of myFun(). Using .Internal(inspect(x)) to check the "suspected copies of x" 's locations, we found out that they all have the same locations in memory. The reason could be that R only made one copy of x but several pointers pointing to x. Therefore, although other variables are assigned to vector x, they actually point to the same copy of x.

```
> #Problem 1(b)
> x <- 1:100000000
> f <- function(input){
+   data <- input
+   g <- function(param) return(param * data)
+   return(g)
+ }
> myFun <- f(x)
> data <- 100
> #myFun(3)
> length(serialize(myFun, NULL))

[1] 800006609
```

The size is 800000779 bytes, meaning that the object takes the size of only one copy. The serialize answer proves the assumption in part(a). Only one copy of x is made in memory while several pointers of different variables point to x.This construction in R saves memory space.

Problem 1(c) When myFun(3) is called, it'll look for myFun() within the environment. myFun() is assigned the value of f(x). Then f(x) will look for the value of x in the environment. However, rm(x) removed the only copy of x in the memory. Therefore, myFun() would not work.

```
> #Problem 1(d)
> x <- 1:10
> f <- function(data){
+   g <- function(param) return(param * data)
+   return(g)
+ }
> myFun <- f(x)
> data <- 100
> myFun(3)

 [1]  3  6  9 12 15 18 21 24 27 30

> length(serialize(myFun, NULL))

[1] 6582

>
```

Remove the code rm(x), and remove the assignment of data to output, the resulting length is 773.

```
> #Problem 2(a)
> y = list(c(1,2),c(3,4))
> .Internal(inspect(y))

@7ffd13f0bea0 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
  @7ffd13f0be30 14 REALSXP g0c2 [] (len=2, tl=0) 1,2
  @7ffd13f0be68 14 REALSXP g0c2 [] (len=2, tl=0) 3,4

> y[[2]][1] = 5
> .Internal(inspect(y))
```

```
@7ffd13926b20 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
  @7ffd13f0be30 14 REALSXP g0c2 [NAM(2)] (len=2, tl=0) 1,2
  @7ffd13926b90 14 REALSXP g0c2 [] (len=2, tl=0) 5,4
```

R made a change in place rather than making a new vector.

```
> #Problem 2(b)
> y_2 = y
> .Internal(inspect(y))

@7ffd13926b20 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
  @7ffd13f0be30 14 REALSXP g0c2 [NAM(2)] (len=2, tl=0) 1,2
  @7ffd13926b90 14 REALSXP g0c2 [] (len=2, tl=0) 5,4

> .Internal(inspect(y_2))

@7ffd13926b20 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
  @7ffd13f0be30 14 REALSXP g0c2 [NAM(2)] (len=2, tl=0) 1,2
  @7ffd13926b90 14 REALSXP g0c2 [] (len=2, tl=0) 5,4

> #No new copy is made since they all have the same locations.
> y_2[[1]][1] = 5
> .Internal(inspect(y))

@7ffd13926b20 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
  @7ffd13f0be30 14 REALSXP g0c2 [NAM(2)] (len=2, tl=0) 1,2
  @7ffd13926b90 14 REALSXP g0c2 [NAM(2)] (len=2, tl=0) 5,4

> .Internal(inspect(y_2))

@7ffd13e74958 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
  @7ffd13e74990 14 REALSXP g0c2 [] (len=2, tl=0) 5,2
  @7ffd13926b90 14 REALSXP g0c2 [NAM(2)] (len=2, tl=0) 5,4

> #When one element is changed in y_1, only change in the relevant vector is made.

> #Problem 2(c)
> list_1 = list(y, y_2)
> list_2 = list_1
> .Internal(inspect(list_1))

@7ffd12971390 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
  @7ffd13926b20 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
    @7ffd13f0be30 14 REALSXP g0c2 [NAM(2)] (len=2, tl=0) 1,2
    @7ffd13926b90 14 REALSXP g0c2 [NAM(2)] (len=2, tl=0) 5,4
  @7ffd13e74958 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
    @7ffd13e74990 14 REALSXP g0c2 [] (len=2, tl=0) 5,2
    @7ffd13926b90 14 REALSXP g0c2 [NAM(2)] (len=2, tl=0) 5,4
```

```
> .Internal(inspect(list_2))

@7ffd12971390 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
  @7ffd13926b20 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
    @7ffd13f0be30 14 REALSXP g0c2 [NAM(2)] (len=2, tl=0) 1,2
    @7ffd13926b90 14 REALSXP g0c2 [NAM(2)] (len=2, tl=0) 5,4
  @7ffd13e74958 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
    @7ffd13e74990 14 REALSXP g0c2 [] (len=2, tl=0) 5,2
    @7ffd13926b90 14 REALSXP g0c2 [NAM(2)] (len=2, tl=0) 5,4

> list_2 = append(7,list_2)
> .Internal(inspect(list_1))

@7ffd12971390 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
  @7ffd13926b20 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
    @7ffd13f0be30 14 REALSXP g0c2 [NAM(2)] (len=2, tl=0) 1,2
    @7ffd13926b90 14 REALSXP g0c2 [NAM(2)] (len=2, tl=0) 5,4
  @7ffd13e74958 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
    @7ffd13e74990 14 REALSXP g0c2 [] (len=2, tl=0) 5,2
    @7ffd13926b90 14 REALSXP g0c2 [NAM(2)] (len=2, tl=0) 5,4

> .Internal(inspect(list_2))

@7ffd13eaac88 19 VECSXP g0c3 [NAM(2)] (len=3, tl=0)
  @7ffd13d78088 14 REALSXP g0c1 [] (len=1, tl=0) 7
  @7ffd13926b20 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
    @7ffd13f0be30 14 REALSXP g0c2 [NAM(2)] (len=2, tl=0) 1,2
    @7ffd13926b90 14 REALSXP g0c2 [NAM(2)] (len=2, tl=0) 5,4
  @7ffd13e74958 19 VECSXP g0c2 [NAM(2)] (len=2, tl=0)
    @7ffd13e74990 14 REALSXP g0c2 [] (len=2, tl=0) 5,2
    @7ffd13926b90 14 REALSXP g0c2 [NAM(2)] (len=2, tl=0) 5,4
```

When making a copy, the location of the two copies are the same. After appending an element to list2 The location of initial elements in list2 are not changed. However, A new location within list2, in relation to the new element, is created but list1 is not modified.

```
> #Problem 2(d)
> gc()

         used (Mb) gc trigger   (Mb)  max used   (Mb)
Ncells 246902 13.2     460000   24.6    350000   18.7
Vcells 448524  3.5  144645031 1103.6 150508515 1148.3

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

```
@7ffd13d80750 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
  @104a6a000 14 REALSXP g0c7 [NAM(2)] (len=10000000, tl=0) -0.216698,-1.11017,0.109788,-1.22
  @104a6a000 14 REALSXP g0c7 [NAM(2)] (len=10000000, tl=0) -0.216698,-1.11017,0.109788,-1.22

> object.size(tmp)

160000136 bytes

> gc()

           used (Mb) gc trigger  (Mb)  max used   (Mb)
Ncells   247167 13.3     460000  24.6    350000   18.7
Vcells 10449272 79.8  115716024 882.9 150508515 1148.3

>
```

The .internal(inspect()) function only shows that they have the same memory location, meaning that only one object with 80Mb is allocated. However, object.size() shows 1600Mb, meaning that two objects are allocated. By inspecting gc() results, the difference in Vcell value shows 80Mb of memory is allocated. The reason could be that object.size() only gives a rough estimate of allocated memory. It does not detect whether or not elements in a list are shared.

```
> #problem 3
> load('ps4prob3.Rda')
> ll <- function(Theta, A) {
+   sum.ind <- which(A==1, arr.ind=T)
+   logLik <- sum(log(Theta[sum.ind])) - sum(Theta)
+   return(logLik)
+ }
> oneUpdate <- function(A, n, K, theta.old, thresh = 0.1) {
+   theta.old1 <- theta.old
+   Theta.old <- theta.old %*% t(theta.old)
+   L.old <- ll(Theta.old, A)
+   q <- array(0, dim = c(n, n, K))
+
+   #Changes are made here. Remove if/else and use vectors to replace " for (z in 1:k)"
+   for (i in 1:n) {
+     for (j in 1:n) {
+     q[i, j, 1:K] <- theta.old[i, 1:K]*theta.old[j, 1:K] /
+       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))
+ }
> # initialize the parameters at random starting values
> temp <- matrix(runif(n*K), n, K)
> theta.init <- temp/rowSums(temp)
> #out <- oneUpdate(A, n, K, theta.init)
> system.time(oneUpdate(A, n, K, theta.init))

   user  system elapsed
  4.199   0.428   4.654

>
```

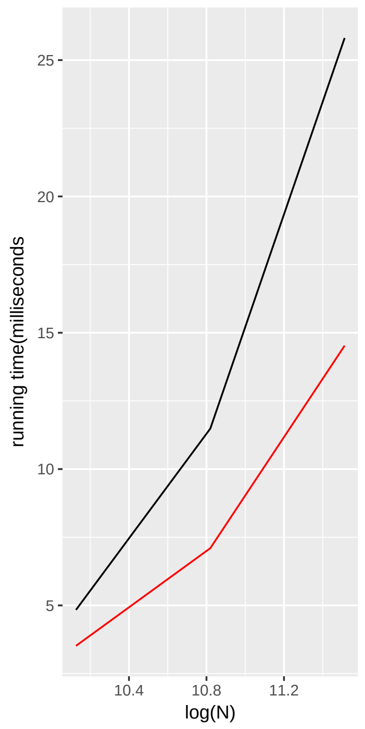
The initial running time was : user 88.140 system: 0.565 elapsed: 88.920. After modifying the code, the running time decreased by over 12 fold. The major modification was on the triple nested for loop. Since matrix q has been pre-assigned 0s, it is unnecessary to use if /else statement to check for 0s. Also, using vectorization to replace for loop will speed up the running time. Insteasd of looping through each z column, I replace all the Zs with a vector from 1:K. The reason I did not modify the for loop after the triple nested for loop is that: rowsum() functions as vectorization. It's already fast enough to carry out the computation.

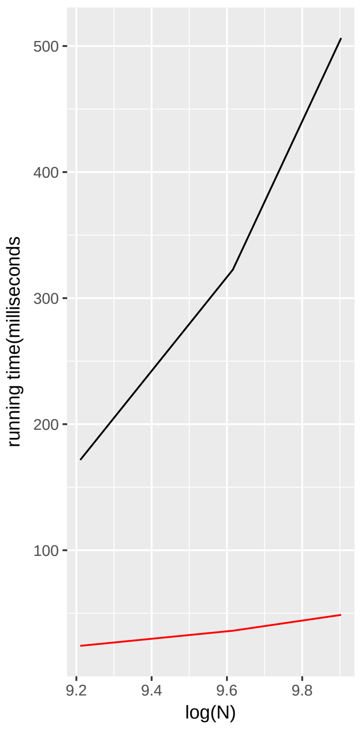
```
> #Problem 4
> library(microbenchmark)
> PIKK <- function(x, k) {
+ x[sort(runif(length(x)), index.return = TRUE)$ix[1:k]]
+ }
> #Add method = quick as a hashing option to speed up the running time.
> PIKK_new <- function(x, k) {
+   x[sort(runif(length(x)), method = "quick", index.return = TRUE)$ix[1:k]]
+ }
> microbenchmark(PIKK(1:10000,500), times = 100)

Unit: milliseconds
               expr      min       lq    mean   median       uq      max neval
 PIKK(1:10000, 500) 1.637094 1.716427 1.86203 1.785684 1.957095 2.531596   100

> microbenchmark(PIKK_new(1:10000,500), times = 100)

Unit: milliseconds
                   expr      min       lq     mean   median       uq      max
 PIKK_new(1:10000, 500) 1.080144 1.11831 1.304545 1.227859 1.393683 1.970791
 neval
   100

> 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]) }
> #move sampling function out of for loop to speed up the running time.
> FYKD_new <- function(x, k) {
+   n <- length(x)
+   j = sample(1:n, n)
+   for(i in 1:n) {
+     tmp <- x[i]
+     x[i] <- x[j[i]]
+     x[j[i]] <- tmp
+
+   }
+   return(x[1:k])
+   }
> microbenchmark(FYKD(1:10000,500), times = 100)

Unit: milliseconds
               expr      min       lq     mean   median       uq     max neval
 FYKD(1:10000, 500) 157.3138 167.0855 173.1414 171.1606 176.7843 218.001   100

> microbenchmark(FYKD_new(1:10000,500), times = 100)

Unit: milliseconds
                   expr      min       lq     mean   median       uq      max
 FYKD_new(1:10000, 500) 21.63364 22.18543 24.01566 23.50833 24.74172 58.84998
 neval
   100

>
```

(1)For PIKK function, I added ”method = quick.” Ideally, ”method = radix” would be the best modification but my program is not compatible with ”method = radix” for some reason. However, the reasons behind ”quick” or ”radix” are similar. Basically, it makes use of the concept of hash table. Hashing helps with linearizing the runnting time, since looking up at the hash table only takes one step (O(1)). If the object size is n, hashing will take O(n) running time.

(2) For FYKD function, since the algorithm inside of the for loop is ”swapping” x[i] with a random element in x. I moved sample function out of for loop to randomly sampling the index first. Then j would be a vector of indices. Then, j is used in the for loop to tell x[i] which element it should swap with. Basically, j is functioned as a hash table for x to look up. Therefore, the running time was decreased at an order scale.

```
> library(ggplot2)
> library(grid)
> library(gridExtra)
> #PIKK PLOTS
> t_1 = summary(microbenchmark(PIKK_new(1:25000,100), times = 100))$mean
> t_2 = summary(microbenchmark(PIKK_new(1:50000,100), times = 100))$mean
> t_3 = summary(microbenchmark(PIKK_new(1:100000,500), times = 100))$mean
> t_4 = summary(microbenchmark(PIKK(1:25000,100), times = 100))$mean
> t_5 = summary(microbenchmark(PIKK(1:50000,100), times = 100))$mean
> t_6 = summary(microbenchmark(PIKK(1:100000,500), times = 100))$mean
> x_axis = c(25000,50000,100000)
> x_axis = log(x_axis)
> Ynew_axis = c(t_1, t_2, t_3)
> Y_axis = c(t_4, t_5 , t_6)
> df = data.frame(x_axis,Y_axis)
> df2 = data.frame(x_axis,Ynew_axis)
> #FYKD PLOTS
> s_1 = summary(microbenchmark(FYKD_new(1:10000,500), times = 100))$mean
> s_2 = summary(microbenchmark(FYKD_new(1:15000,100), times = 100))$mean
> s_3 = summary(microbenchmark(FYKD_new(1:20000,100), times = 100))$mean
> s_4 = summary(microbenchmark(FYKD(1:10000,500), times = 100))$mean
> s_5 = summary(microbenchmark(FYKD(1:15000,100), times = 100))$mean
> s_6 = summary(microbenchmark(FYKD(1:20000,100), times = 100))$mean
> x_K = c(10000,15000,20000)
> x_K = log(x_K)
> Fnew_axis = c(s_1, s_2, s_3)
> F_axis = c(s_4, s_5 , s_6)
> df3 = data.frame(x_K,F_axis)
> df4 = data.frame(x_K,Fnew_axis)
> Plot_P = ggplot(data=df, aes(x=x_axis, y=Y_axis), color='green') +
+ geom_line() +
+ geom_line(data=df2, aes(x=x_axis, y=Ynew_axis), color='red')+
+    labs(x="log(N)", y = "running time(milliseconds")
> plot_K = ggplot(data=df3, aes(x=x_K, y=F_axis), color='blue') +
+ geom_line() +
+ geom_line(data=df4, aes(x=x_K, y=Fnew_axis), color='red')+
+ labs(x="log(N)", y = "running time(milliseconds")
> grid.arrange(Plot_P, plot_K, ncol=2)
```





Based on the plots, PIKK's modification reduces the running time by a factor of two. FYKD's modification reduces the running time by orders of magnitude.

Extra Credit: I found ways to speed up both of the algorithms PIKK and FYKD. Please see the above code, plots and explanations.