Stat 243 - Problem Set #4

Todd Faulkenberry

10/7/2018

Problem 1

Here, make_container() is a closure, i.e. a function that contains its own environment. This allows us to access the local variables in that environment, even after the environment has finished executing the code and has otherwise closed. We see that in action here. Make_container() is a function that requires a numerical argument - if we run it without one, we get an error. When you do pass a number into make_container() and assign it to a value, that value returns a function where that contains an enclosing environment where information can be stored and accessed if we pass some data into that function. Those are two of the enclosing environments with which we deal.

We see another enclosing environment in play when we assign nboot to 100 and bootmeans to make_container(nboot). Once we do this, if we run bootmeans() without any information, we just get a copy of make_container(). If we simply run bootmeans(nboot), we will get a number that is equal to nboot. We can now, however, iterate over bootmeans, allowing to store values (in this case, the mean of a bootstrapped samples) in each of those positions. So, once we iterate over bootmeans, calling bootmeans() will now display the means of each of those 100 bootstrapped samples. We can use bootmeans()[i] to access the value we want, and we now have those values stored in the aforementioned third enclosing environment, where the values are protected from future function calls.

This function "contains" data in the sense that the function stores n number of variables, which each variable representing a value from the function we ran when we iterate. So the function contains data in a very literal sense - those local variables (i.e. the means) are values by themselves but also variables connected to the function, and will continue to exist as long as the function itself exists. If the function is deleted, so are those values. Using object_size(), we can determine that bootmeans() takes up 8MB of space when n = 1000000.

Problem 2

While looking for a way to approach this problem online, I found an excellent solution to the same basic problem on StackOverflow from a user named Flodel (discussion is here: https://stackoverflow.com/questions/20508658/sampling-repeatedly-with-different-probability-in-r):

```
n <- 100000
p <- 5

tmp <- exp(matrix(rnorm(n*p), nrow = n, ncol = p))
probs <- tmp / rowSums(tmp)
smp <- rep(0, n)


set.seed(1)

system.time(
  for(i in seq_len(n))
    smp[i] <- sample(p, 1, prob = probs[i, ])
)
```

```
##    user  system elapsed
##   0.752   0.041   0.809
```

```
f_runif <- function(Weight, y) {
  x <- runif(nrow(Weight))
  cumul.w <- Weight %*% upper.tri(diag(ncol(Weight)), diag = TRUE) /
    rowSums(Weight)
  i <- rowSums(x > cumul.w) + 1L
  y[i]
}

system.time(f_runif(probs, smp))
```

```
##    user  system elapsed
##   0.019   0.003   0.022
```

In lieu of writing code for this particular solution – I didn't think I could improve on this – I looked into this well-designed function to understand what it's doing at each individual part. The function takes in two arguments, a matrix of probabilities where each row sums to one, and a list of numbers from which to randomly select. The function then has a handful of steps:

1. the first variable created (x) is a matrix of random deviates from a uniform distribution between 0 and 1, with a row for each value we ultimately want to collect (so, in this case, x has five times the amount of rows that probs does.)
2. Next, the function calculates the cumulative sum and turns the data into a matrix of 100000 rows where each row sums to 1. What's notable here is the use of matrix multiplication by a triangular matrix – instead of using apply, this makes use of a vectorized function, which creates efficiency.
3. The function then a rowSums comparing x and the rows from cumul.w, counting the number of times that X is bigger and ultimately giving a number between 0 and 4. The 1L is added for two reasons: To get the distribution between the wanted values of 1 and 5, and to ensure that we get an integer vector back instead of a float vector, which will cut down the memory usage and save time.

Problem 3

Don't know how to do this! Going to come talk to you in Office Hours soon about questions like these.

Problem 4

Part A

To test this code, I created a very simple list of vectors and changed one value in the first vector.

```
test <- list(c(1,2,3), c(4,5,6), c(7,8,9))
.Internal(inspect(test))

test[[1]][[1]] <- 2
.Internal(inspect(test))
```

As the above output shows, R can modify an element of a vector without changing the location of any of the elements. Both the list of vectors and the individual vectors themselves stay in the same place.

Part B

Code:

```
> test <- list(c(1,2,3), c(4,5,6), c(7,8,9))
> .Internal(inspect(test))
@7fcffc1d5a08 19 VECSXP g0c3 [NAM(1)] (len=3, tl=0)
  @7fcffc1d5af8 14 REALSXP g0c3 [] (len=3, tl=0) 1,2,3
  @7fcffc1d5aa8 14 REALSXP g0c3 [] (len=3, tl=0) 4,5,6
  @7fcffc1d5a58 14 REALSXP g0c3 [] (len=3, tl=0) 7,8,9
>
> test[[1]][[1]] <- 2
> .Internal(inspect(test))
@7fcffc1d5a08 19 VECSXP g0c3 [NAM(1)] (len=3, tl=0)
  @7fcffc1d5af8 14 REALSXP g0c3 [] (len=3, tl=0) 2,2,3
  @7fcffc1d5aa8 14 REALSXP g0c3 [] (len=3, tl=0) 4,5,6
  @7fcffc1d5a58 14 REALSXP g0c3 [] (len=3, tl=0) 7,8,9
```

Figure 1: R Output for Part A.

```
test <- list(c(1,2,3), c(4,5,6), c(7,8,9))
.Internal(inspect(test))

test_2 <- test
test_2[[1]][[1]] <- 2
.Internal(inspect(test_2))
```

Output:

```
> test <- list(c(1,2,3), c(4,5,6), c(7,8,9))
> .Internal(inspect(test))
@7fcffac13fb8 19 VECSXP g0c3 [NAM(1)] (len=3, tl=0)
  @7fcffac140a8 14 REALSXP g0c3 [] (len=3, tl=0) 1,2,3
  @7fcffac14058 14 REALSXP g0c3 [] (len=3, tl=0) 4,5,6
  @7fcffac14008 14 REALSXP g0c3 [] (len=3, tl=0) 7,8,9
>
> test_2 <- test
> test_2[[1]][[1]] <- 2
> .Internal(inspect(test_2))
@7fcffac13e78 19 VECSXP g0c3 [NAM(1)] (len=3, tl=0)
  @7fcffac13e28 14 REALSXP g0c3 [] (len=3, tl=0) 2,2,3
  @7fcffac14058 14 REALSXP g0c3 [NAM(3)] (len=3, tl=0) 4,5,6
  @7fcffac14008 14 REALSXP g0c3 [NAM(3)] (len=3, tl=0) 7,8,9
```

Figure 2: R Output for Part B.

Here, when we make a copy of a list of vectors and change one element in a vector, we see some changes. The two vectors that weren't changed remain at the same location, but the modified vector has a new location, as does the copied list. Of course, the change only happens to the list of vectors after we change the initial vector - if we had just copied the list and not changed any individual element, that list of vectors would have the same location as the original list of vectors.

Part C

Here is the code I ran directly into the R:

```
test <- list(list(1,2,3), list(4,5,6), list(7,8,9))
.Internal(inspect(test))

test_2 <- test

test_2[[2]][[4]] <- 7
.Internal(inspect(test))
```

And here is the output:

```
> test <- list(list(1,2,3), list(4,5,6), list(7,8,9))
> .Internal(inspect(test))
@7fcffac13c48 19 VECSXP g0c3 [NAM(1)] (len=3, tl=0)
  @7fcffac13d38 19 VECSXP g0c3 [] (len=3, tl=0)
    @7fcffb02e6d0 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 1
    @7fcffb02e698 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 2
    @7fcffb02e660 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 3
  @7fcffac13ce8 19 VECSXP g0c3 [] (len=3, tl=0)
    @7fcffb02e628 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 4
    @7fcffb02e5f0 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 5
    @7fcffb02e5b8 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 6
  @7fcffac13c98 19 VECSXP g0c3 [] (len=3, tl=0)
    @7fcffb02e580 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 7
    @7fcffb02e548 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 8
    @7fcffb02e510 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 9
>
> test_2 <- test
> test_2[[1]][[4]] <- 7
> .Internal(inspect(test_2))
@7fcffac13b08 19 VECSXP g0c3 [NAM(1)] (len=3, tl=0)
  @7fcffac13a68 19 VECSXP g0c3 [] (len=4, tl=0)
    @7fcffb02e6d0 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 1
    @7fcffb02e698 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 2
    @7fcffb02e660 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 3
    @7fcffb02e3c0 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 7
  @7fcffac13ce8 19 VECSXP g0c3 [NAM(3)] (len=3, tl=0)
    @7fcffb02e628 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 4
    @7fcffb02e5f0 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 5
    @7fcffb02e5b8 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 6
  @7fcffac13c98 19 VECSXP g0c3 [NAM(3)] (len=3, tl=0)
    @7fcffb02e580 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 7
    @7fcffb02e548 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 8
    @7fcffb02e510 14 REALSXP g0c1 [NAM(3)] (len=1, tl=0) 9
```

Figure 3: R Output for Part C.

The results here are similar to the results for Part B. The two lists that were not touched remain at the same location in memory, but the original list does not. Neither does the entire list of lists - it has been moved to a new location. Within the modified list, however, all three elements that weren't changed have the same location. The only element with a new location is the element that was added. So, in total, the two lists share the exact same lists if they aren't changed and the elements of any changed list where the elements weren't changed. They do not share the first list overall, nor the entire list of lists.

Part D

Here, using .Internal(inspect()) and object.size() in R reveal that, while object.size() says the the object is taking up 160MB, .Internal(inspect()) shows that both lists of large numbers are actually the same list located in the same memory. So, while object_size() counts the aggregate size of everything in a given object, it does not consider the memory location of the object, meaning it won't realize when an object has something repeated in the memory. This means object_size() can easily overstate the size of the file, and it should be used in conjunction with .Internal(inspect()), which can corroborate if no objects are repeated and thus if object_size() is accurate.