MA 354: Data Analysis I – Fall 2019
Homework 0:

Complete the following opportunities to use what we've talked about in class. These questions will be graded for correctness, communication and succinctness. Ensure you show your work and explain your logic in a legible and refined submission. If you make a mistake while completing this assignment do not delete it – instead, make a note about why it's a mistake, how you realized it, and how you fixed it. The best, most productive recovery from a mistake will receive a 5 point bonus.

0. **Complete weekly diagnostics.** Upon completion of this assignment reflect, in a few sentences, on what you've learned since the beginning of the semester. Think about an mention any concepts you want or need to continue working on and list at least one goal for the next assignment.

1. **Writing Functions and Loops.** Create a function called `f1` in `R`, in as few lines as possible that represents the following function.

$$f_1(x) = \ln\left(\sqrt{x * 10^3}\right)$$

Compare the runtime of your function to the one below using a loop that saves the runtime of each function over 1000 iterations. Summarize and compare the results.

```
> f2 <- function(x){
+   a<- x*10^3
+   b<- sqrt(a)
+   c<- log(b)
+   return(c)
+ }
> f2(1)
```

You can track the runtime of code using the following `R` code.

```
> start_time <- Sys.time()
> #do things you want to time here
> end_time <- Sys.time()
> runtime <-end_time - start_time
```

You might want to report this runtime in your .pdf document, but you'll find that the runtime changes each time you compile your document. You can pass variable values in `R` to the document using the `Sexpr` command. For example, the runtime of the comment above is 0.002 seconds.

Solution. The new and shorter function `f1` is shown below.

```
> f1 <- function(x) (log(sqrt(x * 10 ^ 3)))
```

Shortening the function entailed getting rid of the assignment operations in `f2`. Furthermore, I noticed that R implicitly returns the last value evaluated in the function, so the explicit `return` becomes unnecessary. There is an interesting conversation about this on stack overflow.

As suggested on the Moodle forum, I timed the 2 functions over 1,000,000 iterations.

```
> start_time_f1 <- Sys.time()
> for (j in 1:1000000) {
+   f1(1)
+ }
> end_time_f1 <- Sys.time()
> run_time_f1 <- end_time_f1 - start_time_f1
> start_time_f2 <- Sys.time()
> for (j in 1:1000000) {
+   f2(1)
+ }
> end_time_f2 <- Sys.time()
> run_time_f2 <- end_time_f2 - start_time_f2
```

The table 1 below summarizes the runtimes for the `f1` and `f2`, as well as their time differences.

number of iterations	f2 runtime	f1 runtime	time difference
10^6	0.87	0.62	0.25
10^7	6.13	4.29	1.84
10^8	58.44	43.13	15.30

Table 1: Runtimes for f1 and f2 at different numbers of iteration

In all cases, `f1` runs faster than `f2`. These runtime differences suggest that having more assignment operations within a function can affect its runtime significantly; especially if the function is called repeatedly and at high frequency. The takeaway here is that small tweaks like reducing the number of assignment operations/function calls can make your R code much more efficient. At the same time, I can't help but wonder whether there are real-life situations where this optimization would make a critical difference.

2. **Conditional Statements and Loops.** Assess which is faster.

   (a) Generate 1000 random integers between 1 and 1000 with replacement with the following `R` code.

   ```
   > random.ints<-sample(x=1:1000,size=1000,replace=TRUE)
   ```

   Then, using a loop, create a vector named `random.type` that is "odd" when the corresponding value in `random.ints` is odd and "even" otherwise.

   i. Add elements to the vector called `random.type` as they are decided; e.g., for an iteration where the value is odd we add an element using `random.type<-c(random.type,"odd")`.

   ii. Create an empty vector of size 1000 (with `NA` entries) to start and place elements into the vector; e.g., for iteration i where the value is odd we place the element using `random.type[i]`.

   Which is faster?

Solution: As specified in the question, the first method of creating the "odd/even" vector involves appending to the vector as the parity of a value is decided:

```
> # appending approach
> start_time <- Sys.time()
> # random.type1 -> values will be appended as they are decided
> random.type1 <- c()
> # populating by appending
> for (i in 1:length(random.ints)) {
+   if (random.ints[i] %% 2 == 0) {
+     random.type1 <- c(random.type1, "even")
+   }
+   else {
+     random.type1 <- c(random.type1, "odd")
+   }
+ }
> end_time <- Sys.time()
> run_time1 <- as.numeric(end_time - start_time)
```

The second method involves creating an NA vector of size 1000, then indexing into it to place values into the vector:

```
> # indexing approach
>
> start_time <- Sys.time()
> # random.type2 -> vector will be indexed
> random.type2 <- rep(NA, length(random.ints))
> # populating by indexing
> for (i in 1:length(random.ints)) {
+   if (random.ints[i] %% 2 == 0) {
+     random.type2[i] <- "even"
+   }
+   else{
+     random.type2[i] <- "odd"
+   }
+ }
> end_time <- Sys.time()
> run_time2 <- as.numeric(end_time - start_time)
```

Even with a vector of size 1000, the time difference is significant. I went ahead and timed these 2 methods for bigger-sized vectors. The results are summarized in table 2 below.

number of iterations	method1 runtime	method2 runtime	time difference
10^3	0.247	0.0040	0.243
10^4	25.79	0.0479	25.742
10^5	2632.868	1.733	2631.137

Table 2: Runtime comparison of the 2 methods of creating a vector

The time difference grows quite significant at higher levels of iteration; at 10^5 iterations, the running time of method1 is about 44 minutes, while method2 only takes 1.733s.
The operation `random.type1 <- c(random.type1, "even")` essentially copies values in the vector `random.type1` into a new vector, appends the value "even" to it, then assigns this new vector to the variable name `random.type1`. As the vector grows, so does the size of the vector being copied in each operation. The takeaway here is that you should never grow a vector if you can avoid it; instead, create a vector of the expected size and index into it, placing values as needed.

3. **Conditional Statements and Loops** Each new term in the Fibonacci sequence is generated by adding the previous two terms. By starting with 1 and 2, the first 10 terms will be 1, 2, 3, 5, 8, 13, 21, 34, 55, 89. Find the sum of the even terms in the Fibonacci sequence whose values do not exceed one million.

   Solution: My solution to this question is shown below:

```
> fib.vector <- c(1,2)
> sum.fib <- 2
> repeat {
+   # get the value of the next fibonacci number
+   next.fib = sum(fib.vector)
+   # break if the value of next.fib exceeds 1,000,000
+   if (next.fib > 1000000) {
+     break
+   }
+   # add the next fib number to sum.fib is it's an even term
+   else if (next.fib %% 2 == 0) {
+     sum.fib <- sum.fib + next.fib
+   }
+
+   # update the fib vector to hold the current 2 fib numbers.
+   fib.vector[1] = fib.vector[2]
+   fib.vector[2] = next.fib
+ }
```

First, a vector of size 2 called `fib.vector` is created, which initially holds the values of the first 2 fibonacci terms. Notice that in order to calculate the n^{th} terms of a fibonacci sequence F_n, only the previous 2 terms, F_{n-1} and F_{n-2}, are needed. Therefore, it is possible to solve this problem without having to store every value in the sequence.

The variable `sum.fib` holds the sum of even terms found so far. It is initialized with the value 2 since we've encountered the second term in the fibonacci sequence, 2.

Inside the `repeat` loop, the sum of the numbers in `fib.vector`, which represents the value of the next number in the sequence is calculated and stored in `next.fib`. If it exceeds 1,000,000, then the script breaks out of the loop. Otherwise, if the `next.fib` is even, it is added to `sum.fib`.

Finally, `fib.vector` is updated to hold the 2 most recent fibonacci numbers.

4. **Conditional Statements and Loops** A palindromic number reads the same both ways. The largest palindrome made from the product of two 2-digit numbers is $9009 = 91 \times 99$. Find the largest palindrome made from the product of two 3-digit numbers.

   You can get the reverse of a number using the "stringr" package for `R` (practice Sweave by putting by the citation here) as follows.

```
> #install.packages("stringr",repos = "https://cloud.r-project.org")
> library(stringr)
> number.to.reverse <- 1234
> #creates a list containing a vector of characters
> (split.number<-str_split(number.to.reverse,pattern=""))

[[1]]
[1] "1" "2" "3" "4"

> (split.number<-split.number[[1]]) #takes just the vector

[1] "1" "2" "3" "4"

> (split.number.reversed <- rev(split.number)) #reverses the order of the vector

[1] "4" "3" "2" "1"

> (reversed.number <- paste(split.number.reversed, collapse="")) #paste items back together

[1] "4321"

> (reversed.number<-as.numeric(reversed.number)) #treat it like a number

[1] 4321
```

   Remark: You'll find that the solution to this question will elongate the compiling time for the .pdf. That's okay. Once you're happy with the solution you can type the output into your code chunk and set eval=FALSE between the left and right arrows.

   Solution: First, I wrapped the code provided above into a function called `reverse.string`.

```
> reverse.string <- function(number.to.reverse) {
+   split.number <- str_split(number.to.reverse, pattern = "")
+   split.number.reversed <- rev(split.number[[1]])
+   reversed.number <-
+     as.numeric(paste(split.number.reversed, collapse = ""))
+   reversed.number
+ }
```

To find the largest palindrome, I thought up 2 approaches. The first approach was to multiply all $i * j$ where $100 \leq i, j \leq 999$, shown below.

```
> largest.palindrome <- 0
> find.largest.palindrome <- function() {
+   for (num.1 in 100:999) {
+     for (num.2 in 100:999) {
+       product <- num.1 * num.2
+       if (reverse.string(product) == product & product > largest.palindrome) {
+         largest.palindrome <- product
+       }
+     }
+   }
+   largest.palindrome
+ }
```

I maintained a `largest.palindrome` variable that stores the largest palindromic number so far. If `product` is a palindrome and is greater than `largest.palindrome`, then `largest.palindrome` is updated to hold `product`, which is the new largest palindromic number. The lone `largest.palindrome` statement right before the closing brace of the function ensures that it is the last evaluated statement, which makes it the function's implicit return value.

The second approach involved counting down from the largest product possible, $999 * 999 = 998001$ to the least product possible, $100 * 100 = 10000$ in a for loop, shown below:

```
> find.largest.palindrome.v2 <- function() {
+   max.product <- 999 * 999
+   min.product <- 100 * 100
+   for (current.number in max.product:min.product) {
+     if (reverse.string(current.number) == current.number & check.if.product(current.number)) {
+       return (current.number)
+     }
+   }
+ }
```

In each iteration, I first check if the `current.number` is a palindrome. If it is, I then check if it is also a valid product (meaning that it is a product of two 3-digit numbers). To check this, I used a helper function `check.if.product()`, shown below:

```
> check.if.product <- function(current.number) {
+   for (i in 999:100) {
+     quotient <- current.number / i
+     if (current.number %% i == 0 & quotient <= 999 & quotient >= 100) {
+       return (TRUE)
+     }
+     else if (quotient > 999) {
+       return (FALSE)
+     }
+   }
+ }
```

The `check.if.product()` function takes `current.number` as input. It then checks if the number is a valid product by dividing it by `i`, which starts at 999 and counts down to 100, in a `for` loop. The `if` block gets executed if the remainder is 0 and the quotient lies in the range 100:999; if it does, then the number is a valid product and the `check.if.product()` function returns TRUE. The `else if` condition checks if the quotient exceeds 999. In this case, the product is no longer valid because one of the factors has more than 3 digits; the function returns FALSE.
Lastly, I called the functions, storing their return values in separate variables:

```
> (largest.palindrome<-find.largest.palindrome())
> (largest.palindrome.v2<-find.largest.palindrome.v2())
```

5. **(Working with Data)** Below you will load and summarize a dataset containing 575 observations of drug treatments. The data includes the following

   - ID – Identification Code (1 - 575)
   - AGE – Age at Enrollment (Years)
   - BECK – Beck Depression Score (0.000 - 54.000)
   - HC – Heroin/Cocaine Use During 3 Months Prior to Admission (1 = Heroin & Cocaine; 2 = Heroin Only, 3 = Cocaine Only; 4 = Neither Heroin nor Cocaine)
   - IV – History of IV Drug Use (1 = Never; 2 = Previous; 3 = Recent)
   - IV3 – Recent IV use (1 = Yes; 0 = No)
   - NDT – Number of Prior Drug Treatments (0 - 40)
   - RACE – Subject's Race (0 = White; 1 = Non-White)
   - TREAT – Treatment Randomization (0 = Short Assignment; 1 = Long Assignment)
   - SITE – Treatment Site (0 = A; 1 = B)
   - LEN.T – Length of Stay in Treatment (Days Admission Date to Exit Date)
   - TIME – Time to Drug Relapse (Days Measured from Admission Date)
   - CENSOR – Event for Treating Lost to Follow-Up as Returned to Drugs (1 = Returned to Drugs or Lost to Follow-Up; 0 = Otherwise)
   - etc.

   (a) Load the data provided in the "quantreg" package for `R` (Koenker, 2018).

   ```
   > install.packages("quantreg",repos = "http://cloud.r-project.org/")
   > library("quantreg")
   > data("uis")
   ```

   (b) Numerically summarize the Beck Depression Score of the observed drug treatment patients. Note the following designations of the test when interpreting your results.

   - 0-13: minimal depression
   - 14-19: mild depression
   - 20-28: moderate depression
   - 29-63: severe depression.

   (c) Graphically summarize the Beck Depression Score of the observed drug treatment patients. Interpret the results through the lens of the scale above.

   (d) List three questions about drug use that we might be able to answer based on the data we have.

6. **(Working with Data)** Hepatitis C is a disease that affects the liver. The virus that causes hepatitis C is spread through blood or bodily fluids of an infected person. The virus is often difficult to diagnose because there are few unique symptoms. Those infected, however, sometimes experience jaundice – a condition that causes yellowing of the skin or eyes, as the liver is infected.

   Bracht et al. (2016) consider the human microfibrillar-associated protein 4, or MFAP4, and its role in disease-related tissue. Stage 0–no fibrosis; Stage 1–enlarged, fibrotic portal tracts; Stage 2–periportal fibrosis or portal-portal septa, but intact architecture; Stage 3–fibrosis with architectural distortion, but no obvious cirrhosis; and Stage 4–probable or definite cirrhosis.

   Previously, it has been shown that MFAP4 is a biomarker candidate for hepatic fibrosis and cirrhosis in hepatitis C patients. The analysis of Bracht et al. (2016) aimed to consider the ability of MFAP4 to differentiate between stages of the disease – fibrosis stages (0-2) and cirrhosis (3-4) based on the Scheuer scoring system.

   Below, I load the data and calculate the age of patients using the "lubridate" package for `R` (Grolemund and Wickham, 2011).

```
> fn<-"http://cipolli.com/students/data/biomarker.csv"
> dat <- read.csv(file=fn, header=TRUE, sep=",")
> head(dat)

  Patient.ID Year.of.Birth Gender Date.of.sampling Fibrosis.Stage HCV.Genotype
1       1112          1958 female         2/1/2005              0            1
2       3403          1946 female        1/18/2005              2             
3       2841          1954 female         1/3/2005              3            1
4        654          1958   male         2/1/2005              3            1
5       2788          1960   male        12/9/2004              0            3
6       2242          1954 female        5/12/2004              0            1
  MFAP4.U.mL
1        5.1
2        5.3
3       12.9
4        6.2
5        3.3
6        7.5

> ###Calculate the age of each subject
> #install.packages("lubridate",repos = "http://cloud.r-project.org/")
> library(lubridate)
> ###Create Date Variable for Date Sampled
> dos<-mdy(dat$Date.of.sampling)
> dos.year<-year(dos)
> ###Create age Variable
> age <- dos.year - dat$Year.of.Birth
> ###Add age to original dataset
> dat<-data.frame(dat,age)
```

   (a) Recreate Table 1 in https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4932744/. Add a row to this table labeled "Median age IQR." I've provided the LaTeX code for the table and made the first entry so you can see how it works.

   Solution: I divided my solution into 3 (conceptual) tasks.
   The first task was to calculate the mean age, age sd, and age IQR. First, I created 3 vectors: `ages.mean`, `ages.sd`, `ages.iqr`. I initialized them with 6 NA entries; the first 5 indices correspond to fibrosis stage 0, or column `F0` in the table, to column `F4` The last index corresponds to the `All` column in the table. Per their names, these vectors hold the mean age, age sd, and Median age IQR, respectively.

```
> ages.mean <- rep(NA, 6)
> ages.sd <- rep(NA, 6)
> ages.iqr <- rep(NA, 6)
```

Next, I used a for loop to subset the `age` column by fibrosis stage. The value `i` ranges from 1 to 5; therefore, `i-1` corresponds to fibrosis stages 0-4 and the required data can be extracted neatly in a for loop. During each iteration, I filtered the age column by fibrosis stage using `dat$age[which(dat$Fibrosis.Stage == i-1)]`. I then used the `mean()`, `sd()`, `IQR()` functions to calculate the required values, placing them in their respective vectors.
Since the column `All` corresponds to the entire dataset, I chose to calculate its values outside the for loop.

```
> for (i in 1:5) {
+   fi.ages <- dat$age[which(dat$Fibrosis.Stage == i-1)]
+   ages.mean[i] <- round(mean(fi.ages), 2)
+   ages.sd[i] <- round(sd(fi.ages), 2)
+   ages.iqr[i] <- round(IQR(fi.ages), 2)
+ }
> # mean, iqr, and sd for ages for the entire dataset.
> ages.mean[6] <- round(mean(dat$age), 2)
> ages.sd[6] <- round(sd(dat$age), 2)
> ages.iqr[6] <- round(IQR(dat$age), 2)
```

The second task was to get the patient count by gender, as well as a total patient count. The process is analogous to that for calculating ages above. 2 vectors, `female.count` and `male.count`, with 6 entries corresponding to fibrosis stages 0-4, and the last entry representing the entire data set.

```
> female.count <- rep(NA, 6)
> male.count <- rep(NA, 6)
```

The for loop here is similar to what I described for the 1st task; in this case I filter out the columns by gender and by fibrosis stage, as shown below. Similar to task 1, `j-1` corresponds to Fibrosis Stages 0-4. Since the total patient count is just a sum of the male and female patient counts, I opted to calculate it directly when passing the values table **??** using `Sexpr`. Code below:

```
> ## Select and count patients with fibrosis stage == j-1 and
> ##   update the female.count and male.count vectors
> for (j in 1:5) {
+   female.count[j] <-
+     length(dat$Patient.ID[which(dat$Gender == "female" &
+                                 dat$Fibrosis.Stage == j - 1)])
+   male.count[j] <-
+     length(dat$Patient.ID[which(dat$Gender == "male" &
+                                 dat$Fibrosis.Stage == j - 1)])
+ }
> ## count for all patients
> female.count[6] <-
+   length(dat$Patient.ID[which(dat$Gender == "female")])
> male.count[6] <-
+   length(dat$Patient.ID[which(dat$Gender == "male")])
```

The final task was to summarise data about HCV genotype. I created a data frame in which columns represent the HCV Genotype categories in the table **??**.

```
> type.df <- data.frame(
+   type.1 = rep(NA, 6),
+   type.2 = rep(NA, 6),
+   type.3 = rep(NA, 6),
+   type.4 = rep(NA, 6),
+   type.Other = rep(NA, 6),
+   type.NA = rep(NA, 6)
+ )
```

In order to populate the `type.df` data frame, I used a nested for loop. The outer loop iterates over the HCV genotypes, where `k=1-4` corresponds to genotypes 1-4, `k=5` represents type `Other`, and `k=6` represents type NA.

```
> for (k in 1:6) {
+   if (k %in% c(1, 2, 3, 4)) {
+     fibrosis.stage <- dat$Fibrosis.Stage[which(dat$HCV.Genotype == k)]
+   }
+   else if (k == 5) {
+     fibrosis.stage <-
+       dat$Fibrosis.Stage[which(dat$HCV.Genotype == "andere")]
+   }
+   else {
+     fibrosis.stage <-
+       dat$Fibrosis.Stage[which(dat$HCV.Genotype == "")]
+   }
+   # l-1 in this case represents the Fibrosis stages.
+   # l==1 to 5 ==> fibrosis stages 0 to 4. l==6 ==> whole data set
+   for (m in 1:6) {
+     if (m == 6)
+     {
+       type.df[m, k] <- length(fibrosis.stage)
+     }
+     else
+     {
+       type.df[m, k] <-
+         length(fibrosis.stage[which(fibrosis.stage == m-1)])
+     }
+   }
+ }
```

Inside this loop, I have an `if:else-if:else` block that selects rows from the Fibrosis Stage column of the data set that correspond to the current `HCV Genotype`. Notice that the genotype `Other` in the table corresponds to the string `■andere"` in the data set, while genotype `NA` corresponds to an empty string. The selected rows are stored in a vector called `fibrosis.stage`, as in the above code.
In the inner for loop, the variable `m=1-5` represents the table columns F0-F4, and `m=6` represents the `All` column. This loop selects entries in the `fibrosis.stage` vector that corresponds to the current value of `m`. It then gets the length of that subset and places it in the `type.df` data set. Notice that when `m==6`, we want length of the entire `fibrosis.stage` vector.

Fibrosis Stage	F0	F1	F2	F3	F4	All
Age						
Mean age SD	43.75 ± 12.11	44.83 ± 12.64	51.24 ± 12.04	57.1 ± 10.49	57.33 ± 11.31	49.3 ± 13.08
Median age IQR	16	17.5	19	14.5	16.5	18
Gender						
Women	52	85	66	36	28	267
Men	45	91	69	31	39	275
Number of patients	97	176	135	67	67	542
HCV genotype						
1	70	126	100	50	43	389
2	2	12	1	1	3	19
3	11	25	12	5	7	60
4	0	4	4	1	0	9
Other	0	1	0	0	0	1
NA	14	8	18	10	14	64

Table 3: Patient cohorts characteristics, subdivided by fibrosis stage

(b) Create several graphs that may be helpful for the researchers.

References

Bracht, T., Molleken, C., Ahrens, M., Poschmann, G., Schlosser, A., Eisenacher, M., Stuhler, K., Meyer, H. E., Schmiegel, W. H., Holmskov, U., Sorensen, G. L., and Sitek, B. (2016). Evaluation of the biomarker candidate mfap4 for non-invasive assessment of hepatic fibrosis in hepatitis C patients. *Journal of Translational Medicine*, 14.

Grolemund, G. and Wickham, H. (2011). Dates and times made easy with lubridate. *Journal of Statistical Software*, 40(3):1–25.

Koenker, R. (2018). *quantreg: Quantile Regression.* R package version 5.35.