Homework 1

Avi Singh

Table of contents

Question 1 2
Question 2 3
Question 3 4

Appendix **9**

Link to the Github repository

❗ Due: Sun, Jan 29, 2023 @ 11:59pm

Please read the instructions carefully before submitting your assignment.

1. This assignment requires you to:
   - Upload your Quarto markdown files to a `git` repository
   - Upload a `PDF` file on Canvas
2. Don't collapse any code cells before submitting.
3. Remember to make sure all your code output is rendered properly before uploading your submission.

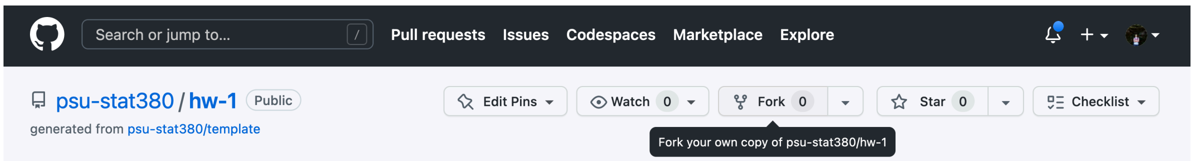
Please add your name to the the author information in the frontmatter before submitting your assignment.

Question 1

20 points

In this question, we will walk through the process of *forking* a `git` repository and submitting a *pull request*.

1. Navigate to the Github repository here and fork it by clicking on the icon in the top right

   Provide a sensible name for your forked repository when prompted.

2. Clone your Github repository on your local machine

   ```
   $ git clone <<insert your repository url here>>
   $ cd hw-1
   ```

   Alternatively, you can use Github codespaces to get started from your repository directly.

3. In order to activate the `R` environment for the homework, make sure you have `renv` installed beforehand. To activate the `renv` environment for this assignment, open an instance of the `R` console from within the directory and type

   ```
   renv::activate()
   ```

   Follow the instrutions in order to make sure that `renv` is configured correctly.

4. Work on the *remaining part* of this assignment as a `.qmd` file.

   - Create a `PDF` and `HTML` file for your output by modifying the YAML frontmatter for the Quarto `.qmd` document

5. When you're done working on your assignment, push the changes to your github repository.

6. Navigate to the original Github repository here and submit a pull request linking to your repository.

   Remember to **include your name** in the pull request information!

If you're stuck at any step along the way, you can refer to the official Github docs here

---

Question 2

30 points

Consider the following vector

```
my_vec <- c(
    "+0.07",
    "-0.07",
    "+0.25",
    "-0.84",
    "+0.32",
    "-0.24",
    "-0.97",
    "-0.36",
    "+1.76",
    "-0.36"
)
```

For the following questions, provide your answers in a code cell.

1. What data type does the vector contain?
2. Create two new vectors called `my_vec_double` and `my_vec_int` which converts `my_vec` to Double & Integer types, respectively,
3. Create a new vector `my_vec_bool` which comprises of:
    - `TRUE`if an element in `my_vec_double` is ≤ 0
    - `FALSE` if an element in `my_vec_double` is ≥ 0

    How many elements of `my_vec_double` are greater than zero?
4. Sort the values of `my_vec_double` in ascending order.

```
typeof(my_vec) # 1. The data type is character
```

```
[1] "character"
```

```r
my_vec_double <- as.double(my_vec) # 2. Converting data types
my_vec_int <- as.integer(my_vec)

typeof(my_vec_double) # Verifying the types of vectors are what we want
```

```
[1] "double"
```

```r
typeof(my_vec_int)
```

```
[1] "integer"
```

```r
my_vec_bool <- ifelse(my_vec_double >= 0, FALSE, TRUE)
summary(my_vec_bool)["FALSE"] # 3. Count number of False occurrences (4)
```

```
FALSE
    4
```

```r
my_vec_double_sorted = sort(my_vec_double) # 4. Sort values of my_vec_double
```

Question 3

> 50 points

In this question we will get a better understanding of how `R` handles large data structures in memory.

1. Provide `R` code to construct the following matrices:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & ... & 100 \\ 1 & 4 & 9 & 16 & 25 & ... & 10000 \end{bmatrix}$$

```r
mat1 <- matrix(c(1,2,3,4,5,6,7,8,9),nrow=3, ncol=3, byrow = TRUE) # Create first matrix
mat2row1 <- 1:100
mat2row2 <- (mat2row1)^2
mat2data <- c(mat2row1, mat2row2)
mat2 <- matrix(mat2data, nrow = 2, ncol = 100, byrow = TRUE) # Create second matrix
```

> **Tip**
>
> Recall the discussion in class on how `R` fills in matrices

In the next part, we will discover how knowledge of the way in which a matrix is stored in memory can inform better code choices. To this end, the following function takes an input n and creates an $n \times n$ matrix with random entries.

```r
generate_matrix <- function(n){
    return(
        matrix(
            rnorm(n^2),
            nrow=n
        )
    )
}
```

For example:

```r
generate_matrix(4)
```

```
           [,1]       [,2]       [,3]       [,4]
[1,] -0.9908804 -0.9185419 -0.3905955 -0.6917678
[2,] -0.5139997  0.6261928  0.9227220  0.3346173
[3,]  0.1690033 -0.2110162  1.3198771 -0.6603208
[4,] -1.2282286  0.9333946 -1.5091210 -0.5504451
```

Let `M` be a fixed 50×50 matrix

```r
M <- generate_matrix(50)
mean(M)
```

```
[1] 0.02292089
```

2. Write a function `row_wise_scan` which scans the entries of `M` one row after another and outputs the number of elements whose value is ≥ 0. You can use the following **starter code**

```
row_wise_scan <- function(x){
    n <- nrow(x)
    m <- ncol(x)

    # Insert your code here
    count <- 0
    for(i in 1:n){
        for(value in x[i,]){
            if(value>=0){
                count <- count + 1
            }
        }
    }

    return(count)
}
```

3. Similarly, write a function `col_wise_scan` which does exactly the same thing but scans the entries of `M` one column after another

```
col_wise_scan <- function(x){
    count <- 0
    m <- ncol(x)
    for(i in 1:m){
      for(value in x[,i]){
        if(value>=0){
          count <- count + 1
        }
      }
    }

    return(count)
}
```

You can check if your code is doing what it's supposed to using the function here[1]

4. Between `col_wise_scan` and `row_wise_scan`, which function do you expect to take shorter to run? Why?

[1] If your code is right, the following code should evaluate to be `TRUE`

- I would expect row_wise_scan to take shorter to run since R handles row-wise operations faster than column-wise.

5. Write a function `time_scan` which takes in a method `f` and a matrix `M` and outputs the amount of time taken to run `f(M)`

```
time_scan <- function(f, M){
    initial_time <- ... # Write your code here
    f(M)
    final_time <- ...  # Write your code here

    total_time_taken <- final_time - initial_time
    return(total_time_taken)
}
```

```
time_scan <- function(f, M){
    initial_time <- Sys.time()
    f(M)
    final_time <- Sys.time()

    total_time_taken <- final_time - initial_time
    return(total_time_taken)
}
```

Provide your output to

```
list(
    row_wise_time = time_scan(row_wise_scan, M),
    col_wise_time = time_scan(col_wise_scan, M)
)
```

```
$row_wise_time
Time difference of 0.003197193 secs

$col_wise_time
Time difference of 0.002784967 secs
```

Which took longer to run?

row_wise_time took longer to run

6. Repeat this experiment now when:

- M is a 100×100 matrix
- M is a 1000×1000 matrix
- M is a 5000×5000 matrix ::: {.cell}

```
M_100 <- generate_matrix(100)
M_1000 <- generate_matrix(1000)
M_5000 <- generate_matrix(5000)
list(
    row_wise_time = time_scan(row_wise_scan, M_100),
    col_wise_time = time_scan(col_wise_scan, M_100)
    )
```

```
$row_wise_time
Time difference of 0.0002958775 secs

$col_wise_time
Time difference of 0.0002698898 secs
```

```
list(
    row_wise_time = time_scan(row_wise_scan, M_1000),
    col_wise_time = time_scan(col_wise_scan, M_1000)
    )
```

```
$row_wise_time
Time difference of 0.02502608 secs

$col_wise_time
Time difference of 0.0274241 secs
```

```
list(
    row_wise_time = time_scan(row_wise_scan, M_5000),
    col_wise_time = time_scan(col_wise_scan, M_5000)
    )
```

```
$row_wise_time
Time difference of 0.817682 secs

$col_wise_time
Time difference of 0.6657979 secs
```

:::

What can you conclude?

- Generally speaking, row_wise_time takes longer to run, and the amount of time it takes to run increases with dimensions of M

Appendix

Print your R session information using the following command

```
sessionInfo()
```

```
R version 4.2.2 (2022-10-31 ucrt)
Platform: x86_64-w64-mingw32/x64 (64-bit)
Running under: Windows 10 x64 (build 22000)

Matrix products: default

locale:
[1] LC_COLLATE=English_United States.utf8
[2] LC_CTYPE=English_United States.utf8
[3] LC_MONETARY=English_United States.utf8
[4] LC_NUMERIC=C
[5] LC_TIME=English_United States.utf8

attached base packages:
[1] stats     graphics  grDevices datasets  utils     methods   base

loaded via a namespace (and not attached):
 [1] digest_0.6.31   lifecycle_1.0.3 jsonlite_1.8.4  magrittr_2.0.3
 [5] evaluate_0.20   rlang_1.0.6     stringi_1.7.12  cli_3.6.0
 [9] renv_0.16.0-53  rstudioapi_0.14 vctrs_0.5.1     rmarkdown_2.20
[13] tools_4.2.2     stringr_1.5.0   glue_1.6.2      xfun_0.36
[17] yaml_2.3.6      fastmap_1.1.0   compiler_4.2.2  htmltools_0.5.4
[21] knitr_1.41
```

```
library(tidyverse)
```

```
-- Attaching packages --------------------------------------- tidyverse 1.3.2 --
v ggplot2 3.4.0      v purrr   1.0.1
v tibble  3.1.8      v dplyr   1.0.10
v tidyr   1.2.1      v stringr 1.5.0
v readr   2.1.3      v forcats 0.5.2
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
```

```r
sapply(1:100, function(i) {
    x <- generate_matrix(100)
    row_wise_scan(x) == col_wise_scan(x)
}) %>% sum == 100
```

```
[1] TRUE
```