

Homework 1

Jacob Adams

Table of contents

| | |
|------------------|---|
| | 2 |
| Question 1 | 2 |
| Question 2 | 3 |
| Question 3 | 5 |

| | |
|-----------------|-----------|
| Appendix | 11 |
|-----------------|-----------|

[Link to the Github repository](#)

! Due: Fri, Jan 26, 2024 @ 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.

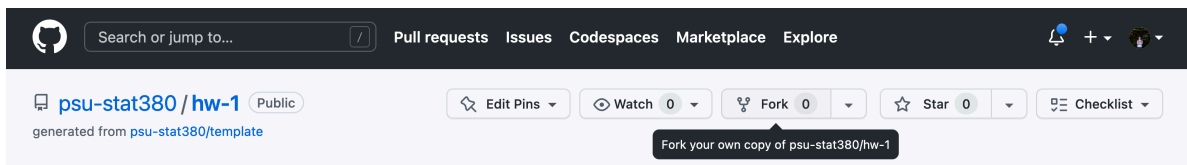
Please add your name to the the author information in the frontmatter before submitting your assignment.

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
```

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 instructions 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?

```
print(typeof(my_vec))
```

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

```
#The vector contains strings.
```

1. Create two new vectors called `my_vec_double` and `my_vec_int` which converts `my_vec` to Double & Integer types, respectively,

```
my_vec_double <- as.numeric(my_vec)
my_vec_int <- as.integer(my_vec)
print(my_vec_double)
```

```
[1] 0.07 -0.07 0.25 -0.84 0.32 -0.24 -0.97 -0.36 1.76 -0.36
```

```
print(my_vec_int)
```

```
[1] 0 0 0 0 0 0 0 0 1 0
```

1. 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

```
my_vec_bool <- logical(0)
for (i in my_vec_double)
{
  if (i <= 0)
  {
    my_vec_bool <- c(my_vec_bool, TRUE)
  }
  else
  {
    my_vec_bool <- c(my_vec_bool, FALSE)
  }
}
print(my_vec_bool)
```

```
[1] FALSE TRUE FALSE TRUE FALSE TRUE TRUE TRUE FALSE TRUE
```

How many elements of `my_vec_double` are greater than zero?

```
j <- 0
for(i in my_vec_double)
{
  if(i > 0)
  {
    j <- j +1
  }
}
print(j)
```


```
[1] 4
```

2. Sort the values of `my_vec_double` in ascending order.

```
sorted_my_vec_double <- sort(my_vec_double)
print(sorted_my_vec_double)
```

```
[1] -0.97 -0.84 -0.36 -0.36 -0.24 -0.07  0.07  0.25  0.32  1.76
```

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:

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

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.

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

For example:

```
generate_matrix(4)
```

```
      [,1]      [,2]      [,3]      [,4]
[1,] 0.7164655 -0.7649008  2.01170554  0.3982644
[2,] 1.6551530  1.1871248  0.78953146  0.4427765
[3,] 1.0523645 -0.2552305 -0.07042073 -2.9591633
[4,] 0.4543408 -0.4515581 -1.61251193  0.1748879
```

Let M be a fixed 50×50 matrix

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

```
[1] -0.006641796
```

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

  return(count)
}
```

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

```

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

  return(count)
  return(count)
}

```

```

# Actual code chunk
col_wise_scan <- function(x)
{
  count <- 0

```

```

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

return(count)
return(count)
}

```

You can check if your code is doing what it's supposed to using the function here¹

```
print(row_wise_scan(M))
```

```
[1] 1239
```

```
print(col_wise_scan(M))
```

```
[1] 1239
```

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

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

```

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

```



```
#I would expect them to be roughly the same.  
#The rows and columns are the same number,  
#thus most likely making it equal time
```

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 <- Sys.time()  
  f(M)  
  final_time <- Sys.time()  
  
  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(row_wise_scan, M)  
)
```

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

```
$row_wise_time  
Time difference of 0.0001020432 secs
```

```
$col_wise_time  
Time difference of 9.989738e-05 secs
```

Which 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

```
N <- generate_matrix(100)  
G <- generate_matrix(1000)  
H <- generate_matrix(5000)
```

```
lst <- list(  
  row_wise_time = time_scan(row_wise_scan, N),  
  col_wise_time = time_scan(row_wise_scan, N)  
)  
print(lst)
```

```
$row_wise_time  
Time difference of 0.0003361702 secs
```

```
$col_wise_time  
Time difference of 0.0004341602 secs
```

```
lst2 <- list(  
  row_wise_time = time_scan(row_wise_scan, G),  
  col_wise_time = time_scan(row_wise_scan, G)  
)  
print(lst2)
```

```
$row_wise_time  
Time difference of 0.04166698 secs
```

```
$col_wise_time  
Time difference of 0.04214191 secs
```

```
lst3 <- list(  
  row_wise_time = time_scan(row_wise_scan, H),  
  col_wise_time = time_scan(row_wise_scan, H)  
)  
print(lst3)
```

```
$row_wise_time  
Time difference of 1.303653 secs
```

```
$col_wise_time  
Time difference of 1.301507 secs
```

What can you conclude?

— I concluded that it doesn't really matter what you use the time difference is negligible.

Appendix

Print your R session information using the following command

```
sessionInfo()
```

```
R version 4.3.1 (2023-06-16 ucrt)  
Platform: x86_64-w64-mingw32/x64 (64-bit)  
Running under: Windows 11 x64 (build 22621)
```

```
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
```

```
time zone: America/New_York
```

```
tzcode source: internal
```

```
attached base packages:
```

```
[1] stats      graphics  grDevices datasets  utils      methods   base
```

```
loaded via a namespace (and not attached):
```

```
[1] compiler_4.3.1 fastmap_1.1.1 cli_3.6.2      htmltools_0.5.7  
[5] tools_4.3.1    yaml_2.3.8    rmarkdown_2.25 knitr_1.45  
[9] jsonlite_1.8.8 xfun_0.41     digest_0.6.34 rlang_1.1.3  
[13] renv_1.0.3     evaluate_0.23
```