title: “Homework 1” author: “ArjunLaxman{style=‘background-color: yellow;’}” toc: true title-block-banner: true title-block-style: default format: pdf #format: pdf

Link to the Github repository

::: {.callout-important} ## Due: Fri, Jan 26, 2024 @ 11:59pm

```
install.packages("magrittr", repos = "https://cran.r-project.org")
```

```
The downloaded binary packages are in
    /var/folders/kr/zyr76x5136x61sbwmb2_w_qr0000gn/T//RtmpnlHjLY/downloaded_packages
```

```
library(magrittr)
```

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

1. Don’t collapse any code cells before submitting.

2. 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 <<https://github.com/arjunlaxman/hw1>>
$ 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 instrutions in order to make sure that `renv` is configured correctly.

4. Work on the *reminaing 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?

```
data_type <- typeof(my_vec)
```

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)  # Converts character to double
my_vec_int <- as.integer(my_vec_double)  # Converts double to int
```

1. Create a new vector `my_vec_bool` which comprises of:
    - `TRUE` if an element in `my_vec_double` is ≤ 0

```
my_vec_bool <- my_vec_double <= 0
# TRUE if an element is <= 0
# FALSE if an element is > 0
```

```
* ``FALSE`` if an element in `my_vec_double` is $\ge 0$


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

```
count_greater_than_zero <- sum(my_vec_double > 0)
```

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

```
my_vec_sorted <- sort(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:

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

Tip

```r
# Creatig the first 3x3 matrix
matrix_1 <- matrix(1:9, nrow=3, ncol=3, byrow=TRUE)

# Prints the matrix to check
print(matrix_1)
```

```
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

```r
# Creating the second matrix with integers and their squares
matrix_2 <- matrix(c(1:100, (1:100)^2), nrow=2, byrow=TRUE)

# Print the matrix to check
print(matrix_2)
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13] [,14]
[1,]    1    2    3    4    5    6    7    8    9    10    11    12    13    14
[2,]    1    4    9   16   25   36   49   64   81   100   121   144   169   196
     [,15] [,16] [,17] [,18] [,19] [,20] [,21] [,22] [,23] [,24] [,25] [,26]
[1,]    15    16    17    18    19    20    21    22    23    24    25    26
[2,]   225   256   289   324   361   400   441   484   529   576   625   676
     [,27] [,28] [,29] [,30] [,31] [,32] [,33] [,34] [,35] [,36] [,37] [,38]
[1,]    27    28    29    30    31    32    33    34    35    36    37    38
[2,]   729   784   841   900   961  1024  1089  1156  1225  1296  1369  1444
     [,39] [,40] [,41] [,42] [,43] [,44] [,45] [,46] [,47] [,48] [,49] [,50]
[1,]    39    40    41    42    43    44    45    46    47    48    49    50
[2,]  1521  1600  1681  1764  1849  1936  2025  2116  2209  2304  2401  2500
     [,51] [,52] [,53] [,54] [,55] [,56] [,57] [,58] [,59] [,60] [,61] [,62]
[1,]    51    52    53    54    55    56    57    58    59    60    61    62
[2,]  2601  2704  2809  2916  3025  3136  3249  3364  3481  3600  3721  3844
     [,63] [,64] [,65] [,66] [,67] [,68] [,69] [,70] [,71] [,72] [,73] [,74]
```

```
 [1,]    63    64    65    66    67    68    69    70    71    72    73    74
 [2,]  3969  4096  4225  4356  4489  4624  4761  4900  5041  5184  5329  5476
      [,75] [,76] [,77] [,78] [,79] [,80] [,81] [,82] [,83] [,84] [,85] [,86]
 [1,]    75    76    77    78    79    80    81    82    83    84    85    86
 [2,]  5625  5776  5929  6084  6241  6400  6561  6724  6889  7056  7225  7396
      [,87] [,88] [,89] [,90] [,91] [,92] [,93] [,94] [,95] [,96] [,97] [,98]
 [1,]    87    88    89    90    91    92    93    94    95    96    97    98
 [2,]  7569  7744  7921  8100  8281  8464  8649  8836  9025  9216  9409  9604
      [,99] [,100]
 [1,]    99    100
 [2,]  9801  10000
```

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,] -0.8463432 -0.8555337 -0.2711772 -0.8549438
[2,] -0.1123393  2.1796750 -0.3168731  1.1889898
[3,] -0.1948057  0.8855451  0.1722450 -0.7926015
[4,]  0.1310771  0.2764002  0.3573066 -0.2179375
```

Let `M` be a fixed 50×50 matrix

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

```
[1] -0.01957708
```

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)  # Number of rows
    m <- ncol(x)  # Number of columns
```

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

    count <- 0

    for(j in 1:m){
        for(i in 1:n){
            if(x[i, j] >= 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?

Since R stores matrices in column-major order, accessing data column-wise aligns with continuous memory locations, enhancing cache efficiency. Thus, the col_wise_scan function should perform faster than row_wise_scan due to more optimal memory access.

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()  # to Capture start time

    f(M)  # Executes function f on matrix M

    final_time <- Sys.time()  # Captures the end time

    total_time_taken <- final_time - initial_time  # Calculates the duration
```

```
    return(total_time_taken)
}
```

Provide your output to

```
M <- generate_matrix(50)  # Create a 50x50 matrix

# Measure the time taken by each scan function
row_wise_time <- time_scan(row_wise_scan, M)
col_wise_time <- time_scan(col_wise_scan, M)

# Output the times
list(
    row_wise_time = row_wise_time,
    col_wise_time = col_wise_time
)
```

```
$row_wise_time
Time difference of 0.002444983 secs

$col_wise_time
Time difference of 0.002273083 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

```
# List of matrix sizes
sizes <- c(100, 1000, 5000)
results <- list()

# Loop through each size and measure performance
for (size in sizes) {
    M <- generate_matrix(size)  # Generates matrix
    row_time <- time_scan(row_wise_scan, M)  # Time raken row-wise scanning
    col_time <- time_scan(col_wise_scan, M)  # Time taken column-wise scanning
    results[[paste(size, "x", size)]] <- list(row_time = row_time, col_time = col_time)
}

# Print results
results
```

```
$`100 x 100`
$`100 x 100`$row_time
Time difference of 0.0003459454 secs
```

```
$`100 x 100`$col_time
Time difference of 0.0003318787 secs


$`1000 x 1000`
$`1000 x 1000`$row_time
Time difference of 0.0330019 secs

$`1000 x 1000`$col_time
Time difference of 0.03213406 secs


$`5000 x 5000`
$`5000 x 5000`$row_time
Time difference of 1.069341 secs

$`5000 x 5000`$col_time
Time difference of 0.8146741 secs
```

What can you conclude?

size of the matrix affects the performance of row-wise and column-wise scanning in R

Appendix

Print your `R` session information using the following command

```
sessionInfo()
```

```
R version 4.3.3 (2024-02-29)
Platform: aarch64-apple-darwin20 (64-bit)
Running under: macOS Sonoma 14.4.1

Matrix products: default
BLAS:   /Library/Frameworks/R.framework/Versions/4.3-arm64/Resources/lib/libRblas.0.dylib
LAPACK: /Library/Frameworks/R.framework/Versions/4.3-
arm64/Resources/lib/libRlapack.dylib;  LAPACK version 3.11.0

locale:
[1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

time zone: America/New_York
tzcode source: internal

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

other attached packages:
[1] magrittr_2.0.3
```

```
loaded via a namespace (and not attached):
 [1] compiler_4.3.3    fastmap_1.1.1     cli_3.6.2         tools_4.3.3
 [5] htmltools_0.5.8.1 rmarkdown_2.26    knitr_1.45        jsonlite_1.8.8
 [9] xfun_0.43         digest_0.6.35     rlang_1.1.3       evaluate_0.23
```

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

```
[1] TRUE
```

Footnotes

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