

Homework 1

Danny Dawson

Table of contents

Question 1	2
Question 2	3
Question 3	4

Appendix	10
-----------------	-----------

[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 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?

this vector contains character 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.double(my_vec)
```

```
my_vec_double
```

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

```
my_vec_int <- as.integer(my_vec)
```

```
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 <- my_vec_double <= 0
```

```
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?


4 elements are greater than zero

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

```
sort(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

```
matrix_one <-  
  matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9),  
        nrow = 3,  
        byrow = TRUE)
```

```
matrix_one
```

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

```
row_one <- 1:100      #need list of numbers from 1 to 100  
row_two <- row_one^2  #this squares the values of row_one
```

```
matrix_two <-  
  matrix(c(row_one, row_two),  
        nrow = 2,  
        byrow = TRUE)
```

```
matrix_two
```

```
      [,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

```

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,] -0.12880432  1.0089632  1.6861397155  1.2729850
[2,] -1.82145507 -0.3032200 -1.2937207659  0.9445901
[3,] -0.03123503 -0.3451563  0.0007674688  0.8302483
[4,] -0.01756789 -1.2676363  2.0503731130 -0.6578059

```

Let M be a fixed 50×50 matrix

```

M <- generate_matrix(50)
mean(M)

```

[1] 0.002857536

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(M){      #needs to be a function of M
  n <- nrow(M)
  m <- ncol(M)

  count <- 0
  for(i in 1:length(n)){          #start by going through rows
    for(j in 1:length(m)){        #goes through each element in the row
      if(j >= 0){
        count <- count + 1      #updates the count if element j is >= 0
      }
    }
  }

  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(M){      #needs to be a function of M
  count <- 0
  n <- nrow(M)
  m <- ncol(M)

  for(j in 1:length(m)){          #swap order of row_wise_scan, cols first.
    for(i in 1:length(n)){        #goes through each element in the column
      if(i >= 0){
        count <- count + 1      #updates count if element i is >= 0
      }
    }
  }

  return(count)
}
```

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

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

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

I wouldn't expect either of these functions to run faster than the other. The matrix `M` that we are working with is a 50×50 matrix, so the amount of time to loop through the columns should be the same amount of time it takes to loop through the rows.

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)  
}
```

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 0.006191015 secs
```

```
$col_wise_time  
Time difference of 1.096725e-05 secs
```

Which took longer to run?

`row_wise_scan` took slightly 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


```

X <- generate_matrix(100)
Y <- generate_matrix(1000)      #just generating the matrices that the
Z <- generate_matrix(5000)      #time_scan function will test

list(
  row_wise_time = time_scan(row_wise_scan, X),    #testing a 100 x 100 matrix
  col_wise_time = time_scan(row_wise_scan, X)
)

```

```

$row_wise_time
Time difference of 1.40667e-05 secs

```

```

$col_wise_time
Time difference of 8.106232e-06 secs

```

```

list(
  row_wise_time = time_scan(row_wise_scan, Y),    #testing a 1000 x 1000 matrix
  col_wise_time = time_scan(row_wise_scan, Y)
)

```

```

$row_wise_time
Time difference of 1.096725e-05 secs

```

```

$col_wise_time
Time difference of 7.152557e-06 secs

```

```

list(
  row_wise_time = time_scan(row_wise_scan, Z),    #testing a 5000 x 5000 matrix
  col_wise_time = time_scan(row_wise_scan, Z)
)

```

```

$row_wise_time
Time difference of 9.059906e-06 secs

```

```

$col_wise_time
Time difference of 7.152557e-06 secs

```

What can you conclude?

While the time difference is very minuscule, `col_wise_scan()` runs faster than `row_wise_scan()`.

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 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
```

```
attached base packages:
```

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

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

```
[1] compiler_4.2.2 fastmap_1.1.0 cli_3.6.0      htmltools_0.5.4
[5] tools_4.2.2    rstudioapi_0.14 yaml_2.3.7     rmarkdown_2.20
[9] knitr_1.42     xfun_0.36      digest_0.6.31 jsonlite_1.8.4
[13] rlang_1.0.6    renv_0.16.0-53 evaluate_0.20
```

```
install.packages("tidyverse")
```

```
Installing tidyverse [1.3.2] ...
```

```
OK [linked cache in 1.9 milliseconds]
```

```
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   1.3.0      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()
```

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

```
[1] TRUE
```

```
#loaded in tidyverse to prevent render error
```