MTH208: Worksheet 15

Coding efficiently and Stabily in R

Now that you've learned benchmarking and profiling a little bit, we will learn about coding efficiently in R. Before starting this worksheet, make sure to finish Worksheet 16.

Here are some general guidelines on making sure we code efficiently in R.

- Avoid loops when possible in R. If loops cannot be avoided, then make sure to allocate memory for the loop. This is a feature of high-level languages in R.

  Sometimes we cannot avoid loops. In such cases, we could switch to implementing C++ within R, which we will discuss in the next lab session.

- You should know how much memory a certain object will take. A single number takes roughly 8 bytes of memory. 1024 bytes = 1KB, 1024 KB = 1MB, 1024 MB = 1GB. Thus a vector of length `n` takes around $8n$ bytes of memory.

- Be careful about numerical instabilities. Sometimes numbers get too small or too large for R to handle. Make transformations to avoid this.

Problems:

1. Theoretically, assess roughly how much memory each of the objects below will take and then verify using `object.size()`.

```r
num1 <- numeric(length = 1e3)
num2 <- numeric(length = 1e6)

mat1 <- matrix(runif(100*1000), nrow = 100, ncol = 1000)
mat2 <- matrix(0, nrow = 100, ncol = 1000)

arr <- array(0, dim = c(100,100,100))
```

2. Sometimes, loading heavy datasets repeatedly can be very time consuming through the usual methods. Suppose I make a large dataset and save it in a csv file as follows

```r
n <- 1e4
p <- 1e2

# generating arbitrary data
dat <- matrix(runif(n*p), nrow = n, ncol = p)

# making csv file from data
write.csv(dat, file = "bigData.csv", row.names = FALSE)
```

The above code makes a csv file with randomly generated data of 10000 observations and 100 columns. Now, let's load the data from the csv file again. This is useful when you close Rstudio, and need to rerun everything again.

```r
temp <- read.csv("bigData.csv")
```

Notice how long it takes for it to load the data, at least a couple of seconds. This is when the data is not even that large. Instead, we can also do the following. We can save the object `dat` that stores the matrix as an R object, and then load the R object!

```r
# save dat
save(dat, file = "largeData.Rdata")
```

And then load this object:

```r
load("largeData.Rdata")
```

Note, we can save any number of R objects we want separated by commas in the `save` command.

3. Consider finding the norm of a vector $\|x\| = \sqrt{x_1^2 + x_2^2 + ... x_k^2}$.

   Compare `sqrt(sum(x^2))` with `norm(x, "2")`. Which is faster? Why?

4. Write two R functions that both take inputs `n` and `rho`. The functions should return an $n \times n$ matrix with the ijth element being $\rho^{|i-j|}$.
   Write one function using a loop and one without using loops. The function without using loop can be:

```r
rho_mat_2 <- function(n, rho)
{
  mat <- matrix(rho, nrow = n, ncol = n)
  mat <- mat^(abs(col(mat) - row(mat)))
  return(mat)
}
```

Benchmark the two functions for different values of `n = 10, 100, 1000`.

5. Stirling's Theorem says that

$$\lim_{n\to\infty} \frac{n!}{(n^n/e^n)\sqrt{2\pi n}} = 1\,.$$

"Verify" the above by plotting the fraction versus n for $n = 10, ..., 10^6$. **HINT:** you may face some unexpected errors. Think about why those errors are happening, and how you might resolve them.

6. Look at the code below. Figure out what it is trying to do, and try to write another function that is faster than this one. Make sure both functions return the same value!

```
func <- function(n = 1e3)
{
  nums <- 1:(n^2)
  mat <- matrix(nums, nrow = n, ncol = 2)
  means <- apply(mat, 2, mean)
  norm.means <- sqrt(sum(means^2))
  return(norm.means)
}
```

7. Consider a vector $x = (x_1, \dots, x_n)$, and suppose we want to calculate:

$$\frac{\log(x_i)}{\sum_{k=1}^{n} \log(x_k)}$$

The function `func` below calculates the above for a given vector `vec`.

```
func <- function(vec)
{
  n <- length(vec)

  # for tracking sum and log
  sum.log <- 0
  log.of.vec <- numeric(length(n))

  # calculating logs and sum for each element
  for(i in 1:n)
  {
    log.of.vec[i] <- log(vec[i])
    sum.log <- sum.log + log.of.vec[i]
  }

  # fraction
```

```
  frac <- log.of.vec/sum.log
  return(frac)
}
```

Write an alternative function `func2` so that when you run the following benchmark, it is *at least* 4 times faster:

```
benchmark(func(1:1e4), func2(1:1e4))
```