# HW1: R Programming

Due: 29 January 2017, 11:59PM UTC-12:00 on T-Square

In this assignment, you will get to know the R programming language by experimenting with examples from the reading and implementing an algorithm in multiple ways using R. You will then compare run times for your functions with a built-in R function.

## Instructions

- You must complete this assignment on your own, not in a group.
- All your code must be written in the R programming language, but feel free to use any tools/environment that you choose.
- You need to turn in 2 files for this assignment:
  - Code: R script file named **`hw1.r`** with code for the different problems.
  - Report: PDF file named **`hw1_report.pdf`** with responses to written portions.
- Each problem includes a Submit section that clearly mentions what you need to put in the Code and/or Report file.
- Make sure you include your GT account name at the beginning of both files (the one you use to log into T-Square or Passport).
- Total points in this assignment: 100.

## Problems

### 1. Get Familiar with R [10 points]

Familiarize yourself with the R code from the first reading assignment. Run the code in an R environment of your choice and observe the results.

Briefly describe one insight you learned about R in your observations. Illustrate it with a sample code snippet and observed output.

**Submit**

- Report: Observation about programming in R, with sample code snippet and output.

## 2. Log Gamma (Loop) [20 points]

Implement a function that computes and returns the natural logarithm of the gamma value of a positive integer <u>using an iterative loop</u> and has the following signature:

**`log_gamma_loop(n)`**

For instance, you should be able to call it as follows:

```
> print(log_gamma_loop(5))
[1] 3.178054
```

Notes:

- Log Gamma is defined as log((n - 1)!) for any positive integer n. You do not have to handle real or complex numbers.
- Implementing it as log(A · B · ⋯) can lead to numerical overflow even for relatively small values of n. Using log(A) + log(B) + ... can help you avoid that.

**Submit**

- Code: Implementation of `log_gamma_loop(n)`.

## 3. Log Gamma (Recursive) [20 points]

Implement a second version of the Log Gamma function that uses recursion instead of a for loop and has the following signature:

**`log_gamma_recursive(n)`**

**Submit**

- Code: Implementation of `log_gamma_recursive(n)`.

## 4. Sum of Log Gamma [20 points]

Using the functions you created in problems 2 & 3, create two additional functions (one for the loop implementation and one for the recursive implementation) that take an integer `n` and sum the Log Gamma results over the range `1` to `n`.

Use the following function signatures:

**`sum_log_gamma_loop(n)`**
**`sum_log_gamma_recursive(n)`**

**Submit**

- Code: Implementation of `sum_log_gamma_loop(n)` and `sum_log_gamma_recursive(n)`.

## 5. Compare Results to Built-In R Function [30 points]

Compare the execution times of your two implementations from problem 4 with an implementation based on the built-in R function `lgamma(n)`. You may use the function `system.time()` to measure execution time. What are the growth rates of the three implementations as `n` increases?

Note: Use the command `options(expressions=500000)` to increase the number of nested recursions allowed. Compare the timing of the recursive implementation as much as possible (until an overflow occurs), and continue beyond that for the other two implementations.

Optional: You may want to collect the running times and plot them using R. Note that `system.time()` returns a 5-element vector, the first of which is the user time that we are interested in (see `proc.time`).

**Submit**

- Code: To call the three functions with increasing values of `n` over a reasonable range and measure execution time.
- Report: A brief writeup explaining your observations, with a table or (optional) plot of running times for comparison.