R Exercises

When doing the following exercises, keep in mind that typing `?funcnames` into the console will being up documentation corresponding to the function called `funcname()`.

- Conditionals are very helpful in structuring the logic of your code. It's best to just illustrate how it works with a code example:

  ```
  x = 1
  if (x > 0) {
      print("x is positive")
  } else if (x < 0) {
      print("x is negative")
  } else {
      print("x is zero")
  }
  ```

  If `if-else` statements are new to you, change the values of `x` and see how the output of this code varies. Now, change this code so that it distinguishes if `x` is greater than, lesser than, or equal to 10.

- Run the following code:

  ```
  x = runif(1)
  if (x < 0.5) {
    print(x)
  }
  else {
    print('big')
  }
  ```

  What's wrong? Fix it. What does `runif` do? Try to guess before looking it up. (*Hint:* It's `r unif`, not `run if`.)

- Read about how to use `for` loops in R (just familiarize yourself with the syntax if you already know what `for` loops are). Now use the `paste()` function to make a vector of length 30 that looks like `("label 1", "label 2", ..., "label 30")`.

- The `rnorm()` function is used to sample from a normal distribution. Write code that generates 10 random samples from the normal distribution, loops through them, and for each value `x` prints `x` if `x` is less than 0.5 and prints `"big"` otherwise.

- Calculate the sums $\sum_{i=10}^{100}\left(i^3+4i^2\right)$ and $\sum_{i=1}^{25}\left(\frac{2^i}{i}+\frac{3^i}{i^2}\right)$ using the `sum()` function.

- Read about how to time function calls in R. Set `x = 1:10` and play around with the different ways in which you can access parts of `x`, like with

`x[1:10]`, `x[2:3]`, etc. Using the `tail()`, `length()`, and `seq()` functions, come up with as many different ways as you can to access the last 5 elements of `x`, and time them to figure out which ones are the fastest. Afterward, check your answers.

- Create a vector of the values of $e^x \cos(x)$ at $x = 3, 3.1, 3.2, \ldots, 6$.
- Using two nested `for` loops, print out all pairs of integers from 1 to 20 without repeats (order doesn't matter). E.g., don't print out both `(1,2)` and `(2,1)`.

Read about how to define functions in R. Refer back to this page as necessary when doing the following exercises. Initially, don't worry too much about the performance of your code—just focus on making it work.

- Make a function `collatz(n)` that takes in a positive integer `n` and returns `n/2` if `n` is even or `3n+1` if `n` is odd. Play around with this—what's the limiting behavior as you apply this function repeatedly, e.g. taking `collatz(collatz(collatz(collatz(collatz(n)))))`?
- For the first 100 integers, calculate the number of iterations of `collatz()` required before the output stops changing and use `hist()` to make a histogram of these results. (For the first part, you might find while loops useful.)
- Write a function that takes in a positive integer `n` and calculates the number of ways in which `n` is expressible as the sum of two cubes of positive integers. What is the smallest integer expressible as the sum of two cubes in two different ways?
- Write a function `fib(n)` that returns the `n`th Fibonacci number, with `fib(1) == fib(2) == 1`. Then write a different function `fib_test()` that takes in two parameters, `n` and `k`, and for the first `n` Fibonacci numbers calculates whether or not they're divisible by `k`. (Think about what this function should return!) Play around and see if you can find any patterns (hint: try `k = 3`).

A **hint** that may be useful: if you have a variable defined in an *outer* environment and you want to change its value while you're inside an *inner* environment, you can use the `<<-` operator to do so. (Don't worry if this doesn't make much sense – this is a relatively advanced topic and you can complete these problems without needing to do this.)

In general, instead of recalculating the same values over and over again, you can store the results of computations the first time you do them and then look for the precomputed results if you need them again. (This is called memoization, related to the broader method of dynamic programming.)

Try to speed up of some of the code you've written with this technique, and quantify the improvements in runtime using the `tictoc` package. In particular, look at `fib_test()` and `collatz()`.