STAT 33B Homework 5

Ziang Gao(3033669851)

This assignment is due **April 14, 2020** by 11:59pm.

The purpose of this assignment is to practice using closures and displaying error messages.

Edit this file, knit to PDF, and:

- Submit the Rmd file on bCourses.
- Submit the PDF file on Gradescope.

If you think you'll need help with submission, please ask in office hours *before* the assignment is due.

Answer all questions with complete sentences, and put code in code chunks. You can make as many new code chunks as you like. Please do not delete the exercises already in this notebook, because it may interfere with our grading tools.

Exercise 1

The Fibonacci sequence is formed by adding successive pairs of numbers together to get the next number. The first ten numbers in the sequence are:

```
1  1  2  3  5  8  13  21  34  55
```

Because each number in the Fibonacci sequence is the sum of the previous two numbers, it is especially easy to compute the sequence using recursion. Here's a simple recursive function to compute the `n`-th Fibonacci number:

```r
slow_fib = function(n) {
  if (n < 3)
    return (1)

  Recall(n - 1) + Recall(n - 2)
}
```

Unfortunately, this function is quite slow for large values of n. The problem is that the number of recursive computations grows exponentially with `n`.

For instance, consider calling `slow_fib(10)`. This results in two calls, to `slow_fib(9)` and `slow_fib(8)`. Each of those calls also results in two calls, for a total of four more calls. Each of those four calls also results in two calls, and so on. The first three levels of calls are:

```
slow_fib(9) + slow_fib(8)

slow_fib(8) + slow_fib(7) + slow_fib(7) + slow_fib(6)

slow_fib(7) + slow_fib(6) + slow_fib(6) + slow_fib(5) +
    slow_fib(6) + slow_fib(5) + slow_fib(5) + slow_fib(4)
...
```

As you can see, many of the calls are identical, so the function wastes time recomputing Fibonacci numbers it has already computed.

The Fibonacci numbers can be computed recursively in a more efficient way by keeping a record of each number that's already been computed. Using the record, each number in the sequence only has to be computed once.

This strategy of recording values that have been already computed and reusing them (rather than recomputing them) is called "memoization". Memoization is a useful technique for improving efficiency in many programming problems.

Implement a memoized Fibonacci function `fib()`. Like `slow_fib()`, your function should have a parameter `n` and should return the `n`-th Fibonacci number. However, your function should "remember" numbers that have already been computed so they do not have to be recomputed.

Test your function by computing `fib(40)`. If your function is working correctly, it should be able to compute this number in less than 5 seconds, and the number should be `102334155`.

Hint 1: To get started, you need to create a factory function that will provide the environment for the closure. The factory function should not have any parameters, and should return your recursive Fibonacci function.

Hint 2: Use a variable called `memo` local to the factory function (not the Fibonacci function) to store the computed Fibonacci values. The variable can be a vector or list. Initialize the variable with 1 at positions 1 and 2; these are the first two Fibonacci numbers.

Hint 3: Your Fibonacci function will need to test whether the requested Fibonacci number is already in `memo`, and act accordingly.

```
# Your code goes here.
memo = c(1, 1)

fib = function(n) {
  if (n <= length(memo)) {
    return (memo[n])
  } else {
    while (length(memo) < n){
      i = length(memo)
      memo[i + 1] = memo[i-1] + memo[i]
    }
    return (memo[n])
  }
}

fib(40)
```

```
## [1] 102334155
```

Exercise 2

Use the microbenchmark package's `microbenchmark()` function to compare the speed of `slow_fib()` and `fib()`. Benchmark how long it takes each function to compute the 20th Fibonacci number.

Memoization is a tradeoff. Although memoization increases the speed of a computation, can you think of any potential drawbacks? Explain in 1-3 sentences.

```
# Your code goes here.
library(microbenchmark)

n = 20
microbenchmark(slow_fib(n), fib(n))
```

```
## Unit: microseconds
##         expr      min        lq      mean   median       uq      max neval
##  slow_fib(n) 6008.787 6418.2135 7104.4668 6657.395 7662.191 13712.84   100
##       fib(n)    9.776   11.3105  261.1168   18.014   22.706 24338.94   100
```

YOUR WRITTEN ANSWER GOES HERE: I think there are two major drawbacks of memoization: 1. It is hard to implement and therefore reduce the readibilty of code 2. It takes more space (compared with slow_fib) in order to store information

Exercise 3

Make a new version of your `fib()` function that includes error handling. If the user supplies a non-positive `n`, the function should print the error message `"n must be positive (got n = N)"` where N is replaced by the value of `n`.

Test your function for `fib(-1)` and `fib(5)` (to confirm that it still works for positive values).

Hint 1: See this week's lecture video for how to generate error messages.

```
# Your code goes here.
memo = c(1, 1)

fib = function(n) {
  if (n < 0) {
    warning("n mush be posituve (got n =", n, ")")
    return ()
    }
  if (n <= length(memo)) {
    return (memo[n])
  } else {
    while (length(memo) < n){
      i = length(memo)
      memo[i + 1] = memo[i-1] + memo[i]
    }
    return (memo[n])
  }
}
```

```
# Use this cell to test.
#
# The `error = TRUE` setting tells knitr to knit despite any errors in this
# cell.
fib(-1)
```

```
## Warning in fib(-1): n mush be posituve (got n =-1)
```

```
## NULL
```

```
fib(40)
```

```
## [1] 102334155
```