Homework 3

Stats 20 Lec 1

Winter 2023

General Guidelines

Please use R Markdown for your submission. Include the following files:

- Your .Rmd file.
- The compiled/knitted HTML document.

Name your .Rmd file with the convention `123456789_stats20_hw0.Rmd`, where `123456789` is replaced with your UID and `hw0` is updated to the actual homework number. Include your first and last name and UID in your exam as well. When you knit to HTML, the HTML file will inherit the same naming convention.

The knitted document should be clear, well-formatted, and contain all relevant R code, output, and explanations. R code style should follow the Tidyverse style guide: https://style.tidyverse.org/.

Any and all course material, including these homework questions, may not be posted online or shared with anyone at any time without explicit written permission by the instructor (Michael Tsiang), even after the quarter is over. Failure to comply is a breach of academic integrity.

Note: All questions on this homework should be done using only functions or syntax discussed in Chapters 1–4 of the lecture notes or Homeworks 1–3. No credit will be given for use of outside functions.

Basic Questions

Collaboration on basic questions must adhere to Level 0 collaboration described in the Stats 20 Collaboration Policy.

Several exercises throughout this course will ask you to code your own version of basic built-in functions from scratch. While you will likely not need to rewrite existing functions from scratch when using R in practice, the thought process of thinking through and writing these functions allows you to build your skills in breaking down complicated problems into simpler steps and more deeply understanding the fundamental toolkit you are building throughout the course.

A general strategy for how to think through writing functions from scratch:

1. Come up with small general examples and think about what your function ideally should do for those examples. Do you notice patterns or using similar logic for every example?
2. Formalize and generalize the logic you used in your small examples to the general case. How should your function work on the intended input argument(s) in general? Outline or describe the steps your function needs to do to output the desired result.
3. Consider any edge cases, i.e., valid inputs that your function is supposed to work on but may not use the same logic as the general case to work. See if the logic you used in general extends to the edge case. If not, think about whether the general case can be modified to accommodate the edge case.

Now that we have introduced flow control statements (loops and `if`/`if-else` statements), we can consider more complex algorithms and functions, but the general strategy remains the same.

Question 1

The objective of this question is to give practice with logical indexing.

Suppose Andy Dwyer tracks his commute time to his women's studies class for ten days and records the following times (in minutes):

14 12 20 19 15 20 28 20 20 18

(a)

Store Andy's commute times as a vector in your workspace called `commute_times`. Use a logical index to determine which days Andy had a commute time that was more than one standard deviation away (longer or shorter) from the average (mean) commute time? What were those commute times?

Hint: In addition to the lecture notes, you may use the `abs()` function, if necessary. The `abs()` function computes the absolute value of the elements of a numeric vector.

(b)

Using the same logical index from (a), determine which days Andy had a commute time that was within one standard deviation (longer or shorter) of the mean commute time? What were those commute times?

(c)

Using the same logical index from (a), what proportion of days did Andy have a commute time that was within one standard deviation of the mean commute time?

Hint: Can arithmetic operators/functions for numeric vectors work for logical vectors? What do `sum()` and `mean()` compute for logical vectors?

Question 2

The objective of this question is to help further your understanding of Boolean operators and what `NA` represents.

Logical vectors in R can contain `NA` in addition to `TRUE` and `FALSE`. An `NA` in a logical context can be interpreted as "unknown." Consider the following commands:

```
NA & TRUE
```

```
[1] NA
```

```
NA & FALSE
```

```
[1] FALSE
```

```
NA | TRUE
```

```
[1] TRUE
```

```
NA | FALSE
```

```
[1] NA
```

Explain why there is a difference in the output of these four commands. Why are they not all `NA`?

Hint: Does the output rely on knowing what the unknown value `NA` represents?

Question 3

The objective of this question is to review the concept of vectorization and practice using logical expressions.

Note: You may not use a loop for this question.

Recall the `get_minimum_coins()` function from Homework 1 Question 7(c), which inputs a positive (whole) number of cents (call the argument `cents`) and outputs the minimum number of coins required to equal that number of cents.

(a)

Is the `get_minimum_coins()` function (as you wrote it in Homework 1) vectorized? Why or why not?

(b)

In Homework 1 Question 7(e), you used reasoning to explain that the numbers of cents less than 100 which require the most coins are 94 and 99. With a single command (i.e., one line of code), verify your answer with your `get_minimum_coins()` function.

Hint 1: The output of your single command should be a vector with the two values 94 and 99. You should not display/print the results for every number of cents less than 100.

Hint 2: If the single command is too challenging to come up with all at once, first find a solution using several commands, then condense the operations into a single line.

Question 4

The objective of this question is to give practice with writing a function using a `for` loop and an `if` statement, as well as to help your understanding of how to deal with `NA` values.

(a)

Write a function called `my_min()` that computes the minimum of a vector without the `min()`, `max()`, `range()`, `fivenum()`, `summary()`, or `sort()` functions. Include an optional logical argument called `na.rm` that specifies whether to remove `NA`s. The output of `my_min(x)` and `min()` should be identical for any vector `x`.

Hint: Optional arguments (in general) have default values so that they do not need to be specified by the user. What should be the default value of the `na.rm` argument? That is, should the default behavior be to remove `NA`s or keep them?

(b)

Test your `my_min()` function from (a) with the following inputs:

(i) `c(4, 1, 0, 2, -3, -5, -4)`

(ii) `c("bears", "beets", "Battlestar Galactica")`

(iii) `7`

(iv) `c("Pawnee", "rules", "Eagleton", NA)`, with `na.rm = TRUE` and `na.rm = FALSE`

(v) `NA`, with `na.rm = TRUE` and `na.rm = FALSE`

Question 5

The objective of this question is to give practice with `while()` loops and writing cleaner code.

Consider the `while()` loop below that computes all Fibonacci numbers less than 500.

```
fib1 <- 1
fib2 <- 1
full_fib <- c(fib1, fib2)
while (fib1 + fib2 < 500) {
  old_fib2 <- fib2
  fib2 <- fib1 + fib2
  full_fib <- c(full_fib, fib2)
  fib1 <- old_fib2
}
full_fib
```

```
 [1]   1   1   2   3   5   8  13  21  34  55  89 144 233 377
```

(a)

The variable `old_fib2` is not actually necessary. Rewrite the `while()` loop with the update of `fib1` based on just the current values of `fib1` and `fib2`.

(b)

In fact, `fib1` and `fib2` are not necessary either. Rewrite the `while()` loop without using *any* variables except `full_fib`.

(c)

Determine the number of Fibonacci numbers less than 1,000,000,000 $= 10^9$.

Intermediate Questions

Collaboration on intermediate questions must adhere to Level 1 collaboration described in the Stats 20 Collaboration Policy.

Question 6

The objective of this question is to introduce and give practice with the expanded order of operations.

Recall the standard PEMDAS order of operations:

- Parentheses `()`
- Exponents `^`
- Multiplication and Division `*`, `/`
- Addition and Subtraction `+`, `-`

We have now learned several additional operations in R to consider. The combined order of operations is:

- Parentheses `()`
- Exponents `^`
- Unary operators `-`, `+` (changing the sign of a number, e.g. `-1`)
- Colon operator `:` (making a regular sequence)
- Infix operators of the form `%xyz%` (e.g., mod `%%`, integer division `%/%`, or matrix multiplication `%*%`)
- Multiplication and Division `*`, `/`
- Addition and subtraction `+`, `-`
- Relational operators `>`, `>=`, `<`, `<=`, `==`, `!=`
- Logical negation `!`
- Logical AND `&`, `&&`
- Logical OR `|`, `||`
- Assignment operator `<-`

Use only parentheses, the order of operations, and coercion rules (i.e., the mode hierarchy) to change the following line of code so that the `jerry` object will contain the numeric vector `c(2, 1)`, and the statement should not produce a warning. You may not use type casting functions for this question.

```r
jerry <- 2:8 * 5 %% 3^-2:7 > 2
jerry
```

```
[1] FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

Hint 1: First use parentheses to understand what is the order of operations as written, then consider what computations/coercions need to occur to get a numeric vector of length 2.

Hint 2: A minimal/ideal solution contains no more than three sets of parentheses.

Question 7

The objective of this question is to introduce the `ifelse()` function and give practice with vectorization.

(a)

Write a function called `my_ifelse()` that implements a vector form of the if-else statement without the `ifelse()` function. That is, the `i`th element of `my_ifelse(test, yes, no)` will be `yes[i]` if `test[i]` is `TRUE` and `no[i]` if `test[i]` is `FALSE`. Values of `yes` or `no` should be recycled if either is shorter than `test`.

Hint: This can be written as a loop, but a vectorized solution is better.

(b)

Verify that your `my_ifelse()` function works by executing the following commands:

```r
x <- (1:10) * pi
my_ifelse(x %% 1 >= 0.5, x %/% 1 + 1, x %/% 1)
```

(c)

Use your `my_ifelse()` function to write `my_abs()` and `my_sign()` functions that, respectively, compute the absolute values and signs of the elements of a numeric vector. The respective outputs of `my_abs(x)` and `my_sign()` should be identical to `abs(x)` and `sign(x)` for any numeric vector `x`.

Hint: It may be helpful to use `my_abs()` when writing `my_sign()`.

Question 8

The objective of this question is to introduce the concept of recursion, as well as give further practice with flow control statements and following instructions given in pseudocode.

There are many ways to sort a vector. In this problem, you will implement one of the more interesting: the **merge sort**.

Merge sort is an example of a **"divide and conquer"** algorithm, which means it solves the problem it is working on by breaking it up into smaller and smaller pieces so that by the time the smaller problems are solved, the larger problem will seem to have magically resolved itself. It is also interesting because it is an example of a ***recursive*** algorithm, which means it calls itself (generating Fibonacci numbers is another example of a recursive algorithm).

One way to think of a merge sort is like this: Suppose you were responsible for sorting a stack of papers. You could just sort them all yourself, but who has time for that? So, you find two friends who owe you a favor and you split the stack of papers in half, giving one half to each of your friends and tell them to sort their stacks. Your plan is to simply "merge" their two sorted stacks together after they give them back to you by repeatedly looking at the first paper on each stack and putting the appropriate one of those two stacks next in a new stack, until only that stack remains. The magic part comes in when each of your friends does the same thing, splitting their stacks in half and each giving those smaller stacks to two other friends, and so on. If anyone gets a stack of length 1 or 0 papers, they simply hand it back up the chain.

Here is an example:

```
Level 1: You
         Dasul, Chengyi, Bianca, Hyunjee, Giorgia, Filippo, Eva, Albert
Level 2: You split the vector in two and give it to two others
         Dasul, Chengyi, Bianca, Hyunjee             Giorgia, Filippo, Eva, Albert
Level 3: They each split their vectors in two and give them two two others each
         Dasul, Chengyi         Bianca, Hyunjee       Giorgia, Filippo      Eva, Albert
Level 4: Each of them split their vectors again and give them to two others
         Dasul      Chengyi    Bianca     Hyunjee    Giorgia    Filippo    Eva        Albert
```

Everybody at level 4 was given a stack of just one paper, so they have already "sorted" their stack and they have them back up the chain:

```
Level 3: Get back:
         Dasul & Chengyi        Bianca & Hyunjee      Giorgia & Filippo     Eva & Albert
         Merge to and pass up:
         Chenyi, Dasul          Bianca, Hyunjee       Filippo, Giorgia      Albert, Eva
Level 2: Get back:
         Chenyi, Dasul  &  Bianca, Hyunjee            Filippo, Giorgia  &  Albert, Eva
         Merge to and pass up:
         Bianca, Chenyi, Dasul, Hyunjee               Albert, Eva, Filippo, Giorgia
Level 1: Get back:
         Bianca, Chenyi, Dasul, Hyunjee   &   Albert, Eva, Filippo, Giorgia
         Merge to and pass up:
         Albert, Bianca, Chenyi, Dasul, Eva, Filippo, Giorgia, Hyunjee
```

All any one person in the system needs to do is: Take the stack they are given. If the stack is length 1 or zero, immediately return it to the person who gave it to them, otherwise split their stack of papers in two and give each half to someone else to sort. When they get back the two sorted halves, merge them into one sorted stack and pass that back.

(a)

Read through the following pseudocode:

```
FUNCTION: merge()
INPUTS: left, right, two sorted numeric vectors
OUTPUT: A single combined sorted vector

merged <- numeric vector of length 0.
WHILE length of left > 0
    IF length right > 0
        IF the first element of left < the first element of right
            merged <- combination of merged and the first element of left
            remove the first element of left
        ELSE
            merged <- combination of merged and the first element of right
            remove the first element of right
    ELSE
        merged <- combination of merged and left
        remove all the elements of left
OUTPUT combination of merged and right


FUNCTION: merge_sort()
INPUTS:   x, a numeric vector
OUTPUT:   A vector containing the elements of x, sorted from smallest to largest.

IF length x > 1
    split x roughly into half, as two vectors named left and right
    sorted_left <- left sorted by merge_sort()
    sorted_right <- right sorted by merge_sort()
    x <- merge sorted_left and sorted_right with the merge() function
OUTPUT x
```

Using the provided pseudocode, write the `merge()` and `merge_sort()` functions.

(b)

Test your `merge_sort()` function on the following vectors:

(i) `numeric(0)`
(ii) `7`
(iii) `10:1`

Advanced Questions

Collaboration on advanced questions must adhere to Level 1 collaboration described in the Stats 20 Collaboration Policy.

Note: Advanced Questions are intended for further enrichment and a deeper challenge, so they will not count against your grade if they are not completed or attempted.

Question 9

Download the `dna.RData` file from BruinLearn and save it to your working directory. Then run the following command to load the objects `dna1` and `dna2` in your workspace:

```r
load("dna.RData")
```

The `dna1` and `dna2` vectors represent nucleotide sequences of DNA (deoxyribonucleic acid). The letters `A`, `C`, `G`, and `T` respectively represent the four nucleotide bases of a DNA strand: adenine, cytosine, guanine, and thymine.

Note: Do *not* print the entire `dna1` and `dna2` objects. It is *extremely* bad practice/style to output an entire object with more than about 100 values in a vector.

(a)

Write a function called `locate_motif()` with two character vector arguments `strand` and `motif` that outputs the index of the start of the `motif` sequence located in the `strand` vector. If the motif is not found, the `locate_motif()` function should output `integer(0)`.

Use your `locate_motif()` function to find the sequence `c("G", "A", "T", "T", "A", "C", "A")` in the `dna1` vector.

(b)

Consider the following two DNA sequences:

```r
seq1 <- c("A", "C", "A", "G", "T")
seq2 <- c("T", "A", "G", "T", "A")
```

A substring is a subset of contiguous values in a sequence. The longest shared substring between `seq1` and `seq2` is `"A", "G", "T"`, starting at index 3 in `seq1` and starting at index 2 in `seq2`.

Using your `locate_motif()` function from (a), write a function called `extract_longest_substring()` with two character vector arguments `strand1` and `strand2` that outputs the longest shared substring. If there is no shared substring, the `extract_longest_substring()` function should output `character(0)`.

Use your `extract_longest_substring()` function to find the longest shared substring for the vectors `dna1` and `dna2`.