"Third programming exercise using for loops and if tests to check if a positive integer is a prime"

Alan E. Berger Sept 24, 2020

version 1

available at https://github.com/AlanBerger/Practice-programming-exercises-for-R

Introduction

The construction of a function to return all the prime numbers between 1 and a positive integer N will be done through a sequence of exercises.

This is the third in a sequence of programming exercises in "composing" an R function to carry out a particular task. The idea is to practice correct use of R constructs and built in functions (functions that "come with" the basic R installation), while learning how to "put together" a correct sequence of blocks of commands that will obtain the desired result.
Note these exercises are quite cumulative - one should do them in order.

In these exercises, there will be a statement of what your function should do (what are the input variables and what the function should return) and a sequence of "hints". To get the most out of these exercises, try to write your function using as few hints as possible.
Note there are often several ways to write a function that will obtain the correct result. For these exercises the directions and hints may point toward a particular approach intended to practice particular constructs in R and a particular line of reasoning, even if there is a more efficent way to obtain the same result.
There may also be an existing R function or package that will do what is stated for a given practice exercise, but here the point is to practice formulating a logical sequence of steps, with each step a section of code, to obtain a working function, not to find an existing solution or a quick solution using a more powerful R construct that is better addressed later on.

Motivation for this exercise

If statements and for loops are basic constructs for constructing a function that will carry out a specified computation. This exercise practices writing a code containing if statements and a for loop that through a sequence of steps computes the desired result. The specific function for this exercise is described below.

Background

The function to be written, getPrimeNumbers(N = 1000), in several steps with increasing amount of efficiency and use of logic in writing the code, is to return the all prime numbers between 1 and the integer N. For this exercise one will write a function, isItPrime, to test if a given positive integer is a prime. The isItPrime function will then be used in the next exercise to construct getPrimeNumbers.

Some basic definitions and results about prime numbers needed in writing these functions follow. A positive integer q **evenly divides** a positive integer n if there is a positive integer k such that n = k * q, for example 3 evenly divides 15; 6 evenly divides 24; but 4 does not evenly divide 9 (in integer arithmetic, since 9 = 2 * 4 with a **remainder** of 1). R provides the **mod** function **%%** such that n %% q gives the remainder **r** from integer dividing n by q (also phrased as **n equals r mod q**). So q evenly divides n is equivalent to n %% q = 0

Here are a few sample results from using the mod function

```
9 %% 4  # 9 mod 4 equals 1
```

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

```
15 %% 3  # 3 evenly divides 15
```

```
## [1] 0
```

```
24 %% 6  # 6 evenly divides 24
```

```
## [1] 0
```

```
32 %% 15  # 15 does not evenly divide 32
```

```
## [1] 2
```

A positive integer p is called **prime** if p > 1 and the only positive integers that evenly divide p are 1 and p (so the first several prime numbers are 2, 3, 5, 7, 11, 13). We can thus use the mod function to test whether a given positive integer is prime. There are more sophisticated approaches, see for example the Wikipedia article {"Primality Test"}(https://en.wikipedia.org/wiki/Primality_test#:~:text=9%20External%20links-,Simple%20methods,composite%2C%20otherwise%20it%20is%20prime.)

This is an example of it often being the case that one needs to learn something about the data and or the science that is relevant to the program one is writing in order to properly do an analysis or correctly carry out a calculation. This can sometimes be essential to avoid serious mistakes, or to write a program that does not take an impractical amount of time to do the calculation.

The function to be constructed by the end of this sequence of exercises is **getPrimeNumbers**, whose argument N is to be a positive integer greater than 1, and which should return, in a vector, call it for example primes_up_to_N, all the prime numbers between 2 and N (including 2, and if N is a prime number, N).

This sequence of exercises will work through the construction of several functions leading up to getPrimeNumbers, the point of which is to practice basic R constructs and "putting together" code to get a working function. This is also an example of writing several functions that together give a modular construction of code to obtain a desired result. It is often easier to test and debug a sequence of functions than one large function, and in some cases the individual functions can be used or quickly modified for use for another purpose.

Instruction for the first function in this exercise

The first function in this sequence will be **isItPrime(n)** whose argument is a positive integer n less than 1,000,000 (just to avoid accidently starting an extremely time consuming calculation) which will return either TRUE if n is a prime and FALSE otherwise. To do this, treat the two cases n equal 1 (not a prime) and n equal 2 (a prime) separately (at the beginning of the function). Then for n greater than 2 simply check whether any integer between 2 and (n-1) evenly divides n (we will make this more efficient in the next version of isItPrime). Note my usage of the phrase: values "between X and Y" includes the "endpoints" X and Y. A skeleton of this funtion is:

```
isItPrimeV1 <- function(n) {
# determine whether the positive integer n is prime
# using the mod function, return TRUE or FALSE accordingly
```

```
# check that the function argument is "admissible"
# test that n is a positive integer (or a real number that equals a positive integer)
n.int <- as.integer(n)
# if n was a real number such as 3.2 then n.int will be n truncated
# to an integer (for this example, 3)

if(!(n.int == n)) stop("n is not an integer")
if(n < 1) stop("n is not positive")

# stop if n is "too large" to avoid a very long calculation
if(n > 1000000) stop("n is > a million")

# code to test if n is prime using R's mod function %%

# special cases
if(n == 1) return(FALSE)
if(n == 2) return(TRUE)

##### rest of code to test if n is prime when n is at least 3

}
```

Try programming this now.

First hint:

positive integer q evenly divides positive integer n if and only if n mod q is 0

Second hint:

positive integer n greater than 2 is a prime if and only if no integer between 2 and (n-1) evenly divides n; test using a for loop

note 2:(n-1) (what you want for the range of the for loop) is **not** the same as 2:n-1 which equals (2:n) - 1 and is 1,. . . ,(n-1)

A working code is given below.

```
isItPrimeV1 <- function(n) {
# determine whether the positive integer n is prime
# using the mod function

# check that the function argument is "admissible"
# test that n is a positive integer (or a real number that equals a positive integer)
n.int <- as.integer(n)
# if n was a real number such as 3.2 then n.int will be n truncated
# to an integer (for this example, 3)

if(!(n.int == n)) stop("n is not an integer")
if(n < 1) stop("n is not positive")

# stop if n is "too large" to avoid a very long calculation
if(n > 1000000) stop("n is > a million")

# code to test if n is prime using R's mod function %%
# return TRUE or FALSE
```

```
if(n.int == 1) return(FALSE)
if(n.int == 2) return(TRUE)
# if got to here, n is at least 3
# test if an integer between 2 and (n-1) evenly divides n

for (q in 2:(n-1)) {
   if((n %% q) == 0) return(FALSE)
   }

# if got to here, n is prime
return(TRUE)
# one could just as well have used an else statement
#
}

#####
# do a couple test runs
isItPrimeV1(2)
```

```
## [1] TRUE
```

```
isItPrimeV1(3)
```

```
## [1] TRUE
```

```
isItPrimeV1(4)
```

```
## [1] FALSE
```

```
isItPrimeV1(5)
```

```
## [1] TRUE
```

```
isItPrimeV1(6)
```

```
## [1] FALSE
```

```
# test several Mersenne numbers
# if you are curious, Google "Mersenne number"
isItPrimeV1(2^17 - 1)  # known to be prime
```

```
## [1] TRUE
```

```
isItPrimeV1(2^11 - 1)  # known to be not prime
```

```
## [1] FALSE
```

```
isItPrimeV1(2^6 - 1)    # known to be not prime
```

```
## [1] FALSE
```

Instruction for the second function in this exercise

Often one can improve the efficiency or accuracy of a program if one learns some more about the subject matter. Here, consider the possible positive integers q (q at least 2) that could evenly divide a positive integer n that is at least 3 (recall we are treating n equal 1 and n equal 2 "by hand"). If q is between 2 and (n-1) and evenly divides n, then n = q * k for some positive integer k, and k must be between 2 and (n-1) (since k equal 1 would be too small because q is at most (n-1), and k equal n would be too large because q is at least 2).
In fact, by similar reasoning, one of the values k and q must be at or below $\sqrt{n}$ since otherwise q * k would be > n. Hence, if n is not a prime, it must be evenly divisible by an integer between 2 and $\sqrt{n}$. So we only have to run the for loop in isItPrime from 2 through as.integer(sqrt(n)). While that doesn't make much practical difference in computation time for modest size n, in other circumstances a little analysis can make a substantial difference in run time and or accuracy.

Now modify the isItPrimeV1 function to take advantage of this additional information, call it isItPrime

Try writing it - and do the test runs. Did you get the correct answer that it is TRUE that 3 is prime? If not - What went wrong?

Here is a version **that fails for n = 3** (but works for other positive integers)

```
isItPrime <- function(n) {
# determine whether the positive integer n is prime
# using the mod function, Version 2

# check that the function argument is "admissible"
# test that n is a positive integer (or a real number that equals a positive integer)
n.int <- as.integer(n)
# if n was a real number such as 3.2 then n.int will be n truncated
# to an integer (for this example, 3)

if(!(n.int == n)) stop("n is not an integer")
if(n < 1) stop("n is not positive")

# stop if n is "too large" to avoid a very long calculation
if(n > 1000000) stop("n is > a million")

# code to test if n is prime using R's mod function %%
# return TRUE or FALSE

if(n.int == 1) return(FALSE)
if(n.int == 2) return(TRUE)
# if got to here, n is at least 3
# test if an integer between 2 and (n-1) evenly divides n

lastq <- as.integer(sqrt(n))
for (q in 2:lastq) {
   if((n %% q) == 0) return(FALSE)
   }
```

```
# if got to here, n is prime
return(TRUE)
# one could just as well have used an else statement
}


# do a couple test runs
isItPrime(2)
```

```
## [1] TRUE
```

```
isItPrime(3)  #  this should return TRUE, did it?, if not why not?
```

```
## [1] FALSE
```

```
isItPrime(4)
```

```
## [1] FALSE
```

```
isItPrime(5)
```

```
## [1] TRUE
```

```
isItPrime(6)
```

```
## [1] FALSE
```

```
# test several Mersenne numbers
isItPrime(2^17 - 1)  # known to be prime
```

```
## [1] TRUE
```

```
isItPrime(2^11 - 1)  # known to be not prime
```

```
## [1] FALSE
```

```
isItPrime(2^6 - 1)   # known to be not prime
```

```
## [1] FALSE
```

Debugging syntax errors (such as forgetting a parenthesis or bracket or curly brace or typing one of these when another is required, or typing a left one when a right one is needed etc.) or using the $ form to extract a column from a data frame with a **variable** that contains a column name but not an actual name of a column (which doesn't even give an error message! - R just returns **NULL**), and debugging errors in the logical construction of the code as in this case, can be a frustrating part of programming, but is a necessary skill that one learns with practice (more on this in later exercises).

In the test cases, this code failed for n = 3. So look at each line of the code (from the top) and think about (or, in general (but not needed here), print out, or for larger objects use head or tail or str (structure) to

look at) what takes place in each line (that is, what was the new value of the variable that was created or modified in that line). Or, in this case, since the code worked before, look at the effect of what was changed, which was to run the for loop from 2 to lastq equal to (in the case that failed) as.integer(sqrt(3)). Well, sqrt(3) is 1.732 (to 4 significant digits) so lastq is 1 when n = 3, and the range of the for loop in this case is the two values {2, 1} and 1 evenly divides any integer, so that is why the code failed (returned FALSE when n was 3). This is an example of the type of reasoning used to track down a coding error. The failure only occurs with n equal 3 since for n larger than 3, as.integer(sqrt(n)) is at least 2. One easy fix is to do the case n = 3 "by hand": add the if test: if(n.int == 3) return(TRUE)
Another way is to increase lastq by 1: lastq <- as.integer(sqrt(n)) + 1L (With some algebra one can check that this value of lastq is < n when n is at least 3, so it is alright to use this value of lastq in isItPrime). Here is a version that works:

```
isItPrime <- function(n) {
# determine whether the positive integer n is prime
# using the mod function, Version 2

# check that the function argument is "admissible"
# test that n is a positive integer (or a real number that equals a positive integer)
n.int <- as.integer(n)
# if n was a real number such as 3.2 then n.int will be n truncated
# to an integer (for this example, 3)

if(!(n.int == n)) stop("n is not an integer")
if(n < 1) stop("n is not positive")

# stop if n is "too large" to avoid a very long calculation
if(n > 1000000) stop("n is > a million")

# code to test if n is prime using R's mod function %%
# return TRUE or FALSE

if(n.int == 1) return(FALSE)
if(n.int == 2) return(TRUE)
# if got to here, n is at least 3
# test if an integer between 2 and sqrt(n) + 1 evenly divides n

lastq <- as.integer(sqrt(n)) + 1L
# the L in 1L "tells" R is treat 1 as an
# integer value rather than a real (numeric) value
# this could also have equivalently been done by
# lastq <- as.integer(sqrt(n) + 1)
for (q in 2:lastq) {
   if((n %% q) == 0) return(FALSE)
   }

# if got to here, n is prime
return(TRUE)
# one could just as well have used an else statement
}


# do a couple test runs
isItPrime(2)
```

```
## [1] TRUE
```

```
isItPrime(3)
```

```
## [1] TRUE
```

```
isItPrime(4)
```

```
## [1] FALSE
```

```
isItPrime(5)
```

```
## [1] TRUE
```

```
isItPrime(6)
```

```
## [1] FALSE
```

```
# test several Mersenne numbers
isItPrime(2^17 - 1)  # known to be prime
```

```
## [1] TRUE
```

```
isItPrime(2^11 - 1)  # known to be not prime
```

```
## [1] FALSE
```

```
isItPrime(2^6 - 1)   # known to be not prime
```

```
## [1] FALSE
```

Instruction for the third function in this exercise

Suppose we are curious to see, for values of n that were not prime, what was the first value of q in the for loop that evenly divided n. Modify isItPrime (call the modified function isItPrimeV2) so that it returns a vector with 3 named entries:

c(is_n_prime = 1, n = n, firstq = n) when n is prime, and

c(is_n_prime = 0, n = n, firstq = q) when n is not a prime

Running these test cases should give the results that follow:

```
# do a couple test runs
isItPrimeV2(2)
isItPrimeV2(3)
isItPrimeV2(4)
isItPrimeV2(5)
isItPrimeV2(6)
```

```
# test several Mersenne numbers
isItPrimeV2(2^17 - 1)  # known to be prime
isItPrimeV2(2^11 - 1)  # known to be not prime
isItPrimeV2(2^6 - 1)   # known to be not prime

##### should get these results:

isItPrimeV2(2)
is_n_prime          n     firstq
         1          2          2

isItPrimeV2(3)
is_n_prime          n     firstq
         1          3          3

isItPrimeV2(4)
is_n_prime          n     firstq
         0          4          2

isItPrimeV2(5)
is_n_prime          n     firstq
         1          5          5

isItPrimeV2(6)
is_n_prime          n     firstq
         0          6          2

# test several Mersenne numbers
isItPrimeV3(2^17 - 1)  # known to be prime
is_n_prime          n     firstq
         1     131071     131071

isItPrimeV3(2^11 - 1)  # known to be not prime
is_n_prime          n     firstq
         0       2047         23

isItPrimeV3(2^6 - 1)   # known to be not prime
is_n_prime          n     firstq
         0         63          3
```

A working version of the code is:

```
isItPrimeV2 <- function(n) {
# determine whether the positive integer n is prime
# using the mod function, Version 2

# check that the function argument is "admissible"
# test that n is a positive integer (or a real number that equals a positive integer)
n.int <- as.integer(n)
# if n was a real number such as 3.2 then n.int will be n truncated
# to an integer (for this example, 3)

if(!(n.int == n)) stop("n is not an integer")
```

```
if(n < 1) stop("n is not positive")

# stop if n is "too large" to avoid a very long calculation
if(n > 1000000) stop("n is > a million")

# code to test if n is prime using R's mod function %%
# return c(is_n_prime = 1, n = n, firstq = n)  when n is prime
# return c(is_n_prime = 0, n = n, firstq = q)  when n is not a prime
# where firstq is the first (smallest) integer
# greater than 1 that evenly divides n (firstq is set to 1 if n is 1)
# (q is the index of the for loop below)

if(n.int == 1) return(c(is_n_prime = 0, n = 1, firstq = 1))
if(n.int == 2) return(c(is_n_prime = 1, n = 2, firstq = 2))
# if got to here, n is at least 3
# test if an integer between 2 and sqrt(n) + 1 evenly divides n

lastq <- as.integer(sqrt(n)) + 1L
# the L in 1L "tells" R is treat 1 as an
# integer value rather than a real (numeric) value
# this could also have equivalently been done by
# lastq <- as.integer(sqrt(n) + 1)
for (q in 2:lastq) {
   if((n %% q) == 0) return(c(is_n_prime = 0, n = n.int, firstq = q))
   }

# if got to here, n is prime
return(c(is_n_prime = 1, n = n.int, firstq = n.int))
}


# do a couple test runs
isItPrimeV2(2)
```

```
## is_n_prime          n     firstq
##          1          2          2
```

```
isItPrimeV2(3)
```

```
## is_n_prime          n     firstq
##          1          3          3
```

```
isItPrimeV2(4)
```

```
## is_n_prime          n     firstq
##          0          4          2
```

```
isItPrimeV2(5)
```

```
## is_n_prime          n     firstq
##          1          5          5
```

```r
isItPrimeV2(6)
```

```
## is_n_prime          n     firstq
##          0          6          2
```

```r
# test several Mersenne numbers
isItPrimeV2(2^17 - 1)  # known to be prime
```

```
## is_n_prime          n     firstq
##          1     131071     131071
```

```r
isItPrimeV2(2^11 - 1)  # known to be not prime
```

```
## is_n_prime          n     firstq
##          0       2047         23
```

```r
isItPrimeV2(2^6 - 1)   # known to be not prime
```

```
## is_n_prime          n     firstq
##          0         63          3
```

Hope this programming exercise was informative and good practice. The next programming exercise will be to use your isItPrime function as the "engine" for writing getPrimeNumbers(N = 1000), which will return all the prime numbers between 1 and the positive integer N.

= =