“Fourth R programming exercise find prime integers less than or equal N”

Alan E. Berger November 22, 2020

version 1

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

Finish the construction of a function to return all the prime numbers between 1 and a positive integer N

Introduction

This is the fourth 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

For this exercise, we will finish constructing the function getPrimeNumbers(N = 1000) which will return all the prime numbers between 1 and the positive integer N. We will use the isItPrime(n) function constructed in the previous exercise, which tests whether the positive integer n is a prime number. This illustrates construction of a function in several steps and in a modular fashion, allowing for flexibility and easier testing and debugging.

Background

Recall the definitions and results about prime numbers from the previous exercise:
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

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). In the previous exercise we used the mod function to construct the isItPrime(n) function.

The function to be constructed 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).

Instructions for constructing getPrimeNumbers

In the previous exercise we constructed **isItPrime(n)** whose argument is a positive integer n that is at most 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. This is a copy of isItPrime, the same as in the previous exercise, except that here I have commented out the check for N being too large since that will be done in getPrimeNumbers:

```
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)
}
```

Use a for loop and use isItPrime(n) to test each positive integer n between 2 and N to see if it is prime. Return the integers that are found to be prime in a vector called, for example, primes_up_to_N

For the first version of getPrimeNumbers, use the following simple comstruction to obtain primes_up_to_N: initialize primes_up_to_N to be integer(0), then in a for loop whose index, call it n, runs from 2L to N, use isItPrime to test if n is a prime. If n is prime, append n to primes_up_to_N via the statement

primes_up_to_N <- c(primes_up_to_N, n)

Try writing getPrimeNumbers now.

If you do getPrimeNumbers(N = 111) you should get

```
getPrimeNumbers(111)
 [1]   2   3   5   7  11  13  17  19  23  29  31  37  41  43  47
[16]  53  59  61  67  71  73  79  83  89  97 101 103 107 109
```

The number of values printed on each line in an R session depends on the width of the R console window.

A working version of getPrimeNumbers follows:

```r
getPrimeNumbers <- function(N) {
# N should be a positive integer that is at least 2
# return a vector containing all the prime numbers between 2 and N
# (including 2 and including N if N is a prime)

# 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 < 2) stop("N is not at least 2")

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

# initialize primes_up_to_N
primes_up_to_N <- integer(0)

for (n in 2L:N.int) {
   if(isItPrime(n)) {
      primes_up_to_N <- c(primes_up_to_N, n)
   }
}

return(primes_up_to_N)
}
```

Using a running index with a preset vector to obtain primes_up_to_N

In the next version of getPrimeNumbers, instead of doing

primes_up_to_N <- c(primes_up_to_N, n)

to "accumulate" the prime numbers in a vector, you are to initialize the integer vector primes_up_to_N to be of length N to contain the prime numbers between 2 and N. Obviously this vector will generally be larger than needed, but we can place each prime number as it is found into successive entries of primes_up_to_N using a **running index**, call it k. How this works is one intializes k to 0 and then each time inside the for loop an integer n is found to be prime, one increases k by 1 and then sets primes_up_to_N[k] <- n When the for loop is completed, k will be the number of primes that were found between 2 and N, and so one then "trims" primes_up_to_N by doing

primes_up_to_N <- primes_up_to_N[1:k]

This takes more initial storage space, but is "cleaner" than successively creating new vectors by doing primes_up_to_N <- c(primes_up_to_N, n) and is a technique one should be familiar with.

Try writing a version of getPrimeNumbers that uses a predefined primes_up_to_N integer vector (of length N) and a running index to fill in its entries, and then trim it to the correct length before returning it. A working version is given below.

```r
getPrimeNumbers <- function(N) {
# N should be a positive integer that is at least 2
# return a vector containing all the prime numbers between 2 and N
# (including 2 and including N if N is a prime)

# for this version use a predefined integer vector primes_up_to_N of
# length N and a running index k to fill in entries, and then trim it
# after the for loop is completed

# 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 < 2) stop("N is not at least 2")

# if N is "too large" (> 1,000,000) then stop
if(N > 1000000) {
   cat("N = ", N, "\n")  # print N and also include going to a new output line
   stop("N is > a million")
}

# initialize primes_up_to_N
primes_up_to_N <- integer(N)
k <- 0  # the running index

for (n in 2L:N.int) {
   if(isItPrime(n)) {
      k <- k + 1  # get next location in primes_up_to_N
      primes_up_to_N[k] <- n
   }
}

primes_up_to_N <- primes_up_to_N[1:k]  # trim to correct length
return(primes_up_to_N)
}

# do a test run
getPrimeNumbers(111)
```

```
##  [1]   2   3   5   7  11  13  17  19  23  29  31  37  41  43  47  53  59  61  67
## [20]  71  73  79  83  89  97 101 103 107 109
```

On my computer the latter version of getPrimeNumbers runs a bit faster than the former version (for N = 1000000 the former takes about 18 seconds and the latter 12 seconds).

Using the readline function to let the user decide whether to continue a run if N > a million

The next version of getPrimeNumbers is the same as the one immediately above except that instead of stopping with an error if N is > 1,000,000 this version asks the user to decide whether or not to continue with running the function by replying "yes" or "no" using the **readline** function if N is > 1,000,000 as illustrated below.

```r
getPrimeNumbers <- function(N) {
# N should be a positive integer that is at least 2
# return a vector containing all the prime numbers between 2 and N
# (including 2 and including N if N is a prime)

# for this version use a predefined integer vector primes_up_to_N of
# length N and a running index k to fill in entries, and then trim it
# after the for loop is completed

# 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 < 2) stop("N is not at least 2")


# if N is "large" (> 1,000,000) check with the user to see if the user wants to proceed
if(N > 1000000) {
   cat("N = ", N, "\n")  # print N and also include going to a new output line
   yes.or.no <- readline("this N is large, do you want to continue, type yes or no: ")
   if(yes.or.no != "yes") return("N was large so exited getPrimeNumbers")
}


# initialize primes_up_to_N
primes_up_to_N <- integer(N)
k <- 0  # the running index

for (n in 2L:N.int) {
   if(isItPrime(n)) {
      k <- k + 1  # get next location in primes_up_to_N
      primes_up_to_N[k] <- n
   }
}

primes_up_to_N <- primes_up_to_N[1:k]  # trim to correct length
return(primes_up_to_N)
}

# do a test run
getPrimeNumbers(111)
```

```
## [1]   2   3   5   7  11  13  17  19  23  29  31  37  41  43  47  53  59  61  67
```

```
## [20]  71  73  79  83  89  97 101 103 107 109
```

```
# do a second test run
primes.for.N.equal.a.million <- getPrimeNumbers(1000000)
length(primes.for.N.equal.a.million) # should be 78498
```

```
## [1] 78498
```

```
primes.for.N.equal.a.million[1000] # should be 7919
```

```
## [1] 7919
```

```
primes.for.N.equal.a.million[10000] # should be 104729
```

```
## [1] 104729
```

```
tail(primes.for.N.equal.a.million) # the last value should be 999983
```

```
## [1] 999931 999953 999959 999961 999979 999983
```

Hope this programming exercise was informative and good practice. The next set of exercises will get into using data frames.

= =