# An Introduction to Programming in R: Exercises and Proposed Solutions

Drew Schmidt

November 13, 2015

## Contents

## Part I

# Foreword

The exercises here accompany the lecture *An Introduction to Programming in R.* Some problems will be presented without full motivation or information made available to you. This is deliberate. The only way to learn is through struggle. If I tell you that in R, you have to cast a factor as character before casting as integer/numeric, then you have already forgotten. But if you experience it in a live problem, you are much more likely to remember.

If you aren't confused, you aren't learning.

The proposed solutions are not meant to be perfect or ideal. In the lecture, we almost exclusively deal with base R, even if superior options exist in a package. The exercises here too exclusively deal with base R. In some cases, better solutions even in base R exist; for example, most of the string problems can be easily solved elegantly with regular expressions. But we do not discuss regular expressions, so we do not use them in the solutions. This, too, is part of your programming journey. First learn how to do things; then after you know they can be done, learn how to do them better.

Good luck.

# Part II

# Exercises

## 1 R Basics

## 1.1 R Basics

*"Begin at the beginning", the King said gravely, "and go on till you come to the end: then stop."*

—Lewis Carroll, *Alice in Wonderland*

**Ex. 1** — Download and install R from the CRAN `http://cran.r-project.org/`. Bonus points if you compile it yourself.

**Ex. 2** — Download and install Rstudio from `http://www.rstudio.com/`.

**Ex. 3** — Install the **devtools** package from the CRAN using R's `install.packages()` function.

**Ex. 4** — Enter:

```
print("Hello, world!")
```

Into the R terminal. Is there any difference between this and simply entering `"Hello, world!"`?

**Ex. 5** — Create an R script that contains the code

```
print("Hello, world!")
```

Save the file as `hw.r` and run it from from an interactive R session using the command `source("hw.r")`.

**Ex. 6** — In each case, what is the value of `x`? (Try to think it through before you try it in R)

```
x <- 2 - 1 * 2
x <- 6/3-2+1*0+3/3-3
x <- 19%%17%%13 # compare to (19%%17)%%13 and 19%%(17%%13)
x <- 2^17%%17
x <- 3-2%%5+3*2-4/2
```

**Ex. 7** — In each case, what is the value of `x`?

```
x <- sum(1:10-5)
x <- sum(1:10)-5
x <- 1:10-10:1
x <- sum(1:10-10:1)
```

**Ex. 8** — Russian Roulette is a game played with a revolver and a single bullet. Write an R script which simulates the game of Russian Roulette, returning `"click"` 5 out of 6 times, and `"BANG"` 1 out of 6 times (Hint: see `help("sample")`).

## 1.2 IO

*Good design adds value faster than it adds cost.*

—Thomas C. Gale

Throughout, let `x <- data.frame(a=1:5, b=5:1)`.

**Ex. 9** — Save `x` to disk using `write.csv()` and `write.table()`. Experiment with the different options.

**Ex. 10** — Use `read.csv()` and `read.table()` to read the file(s) you wrote in Exercise 9.

**Ex. 11** — Repeat Exercises 9 and 10 using R's `save()` and `load()` functions.

**Ex. 12** — Use `scan()` to read the file(s) you wrote in Exercise 9.

## 1.3 Strings

*Most of you are familiar with the virtues of a programmer. There are three, of course: laziness, impatience, and hubris.*

—Larry Wall

**Ex. 13** — What does this do?

```
x <- rnorm(1000, mean=10)

cat(sprintf("mean:\t%.2f\nvar:\t%.2f\n", mean(x), var(x)))
```

**Ex. 14** — Use paste() on these character vectors:

```
a <- c("o", "a", "a", "n", "Y", "f", "d", "e", "c", " ", "s", ".")
b <- c("g", "u", "i", "!", "u", "u", "t", "s", "e", "e", "g", "")
c <- c("n", "t", "t", "s", "o", "o", " ", " ", "r", "m", "a", "")
d <- c("C", "r", "l", "o", " ", " ", "n", "h", "e", "t", "s", "e")
```

to decode the secret message. For example, `paste(a, b, c, d, collapse="", sep="")`.

**Ex. 15** — Write a function to reverse the case of a string (all caps are sent to lowercase and all lowercase are capitalized).

**Ex. 16** — A *palindrome* is a string which is the same even after it is reversed, such as "12321" or "racecar". Determine whether or not the following two numbers are palindromes:

- 73132514168488803306526700353904092183768124612171068341
  38522783887608393519025808100767202597280997030718964589
  85469817030799082795202767001808520915393806788387225831
  4386017121642186738129040935300762560330888486141523137

- 73132514168488803306526700353904092183768124612171068341
  38522783887608393519025808100767202597280997030718964589
  85469817030799082795202767007808520915393806788387225831
  4386017121642186738129040935300762560330888486141523137

(Hint: treat them as strings, not numbers).

**Ex. 17** — Write a function `chomp()` that, given a string, removes from the string any occurrence of the character `&`, as well as the character to the left of each `&` character. So for example, your function should return:

```
> chomp("a&c")
[1] "c"
> chomp("a&")
[1] ""
> chomp("abc")
[1] "abc"
```

## 1.4 Dates

*Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.*

—Martin Golding

**Ex. 18** — Construct valid times using lubridate's `ymd()`, `dmy()`, . . . etc commands.

```
"2-3-13" # month first (American style)
"2/3/13" # day first (European style)
"January 1, 1970"
```

**Ex. 19** — Find the week of the year and day of the week for each of these dates:

```
"2-3-13" # month first
"2/3/13" # day first
"January 1, 1970"
```

## 1.5 Dealing with Dataframes

*On two occasions I have been asked [by members of Parliament]: 'Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?' I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.*

—Charles Babbage

We will use a small, sample dataset from Bob Muenchen[1]. The dataset is a fabrication of some survey data from software workshops, where the variables are:

- workshop – software introduced at the workshop
- gender – gender of participant
- q1 – The instructor was well prepared.
- q2 – The instructor communicated well.
- q3 – The course materials were helpful.
- q4 – Overall, I found this workshop useful.

We can recreate the data in R by entering:

```
workshop <- c("R", "SPSS", NA, "SPSS", "STATA", "SPSS")
gender <- factor(c("Female", "Male", NA, "Female", "Female",
   "Female"))
q1 <- c(4, 3, 3, 5, 4, 5)
q2 <- c(3, 4, 2, 4, 4, 4)
q3 <- c(4, 3, NA, 5, 3, 3)
q4 <- c(5, 4, 3, 3, 4, 5)

df <- data.frame(workshop, gender, q1, q2, q3, q4)
```

**Ex. 20** — Create a dataframe consisting of only the first two columns.

**Ex. 21** — Create a dataframe consisting of only the first and last row.

**Ex. 22** — What happens when you enter `as.list(df)`? `unlist(df)`?

**Ex. 23** — Create a dataframe called `df2` where every entry in the q3 and q4 columns is 0.

**Ex. 24** — Sort `df` by gender.

**Ex. 25** — Does `df` have any duplicate rows? What about `df2` from exercise 23?

[1]See `r4stats.com`

**Ex. 26** — What does `na.omit(df)` do?

# 2 Programming

## 2.1 Data Structures

*I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships.*

—Linus Torvalds

**Ex. 27** — Give the R code required to produce this list:

```
$a
[1] 1 2 3 4 5

$b
[1] "a" "b"

$c
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

**Ex. 28** — Take the vector

```
set.seed(1234)
x <- c(rep(NA, 3), rnorm(5))
```

and use the `sum()` function on this vector. Experiment with its options for handling `NA`'s (see `?sum` for details).

**Ex. 29** — The sample mean of a vector $x = [x_i]_{i=1}^n$ is defined as

$$\mu_x = \sum_{i=1}^{n} \frac{x_i}{n}$$

and the unbiased sample variance is defined as

$$\sigma_x^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - mn_x)^2$$

Write an R script which will compute the mean and variance of the vector `x <- 1:100`. Compare with R's internal `mean()` and `var()` functions.

**Ex. 30** — R has a full suite of matrix-vector and matrix-matrix arithmetic operations, including `t()`, `+`, `*`, and `%*%`. Let

```
x <- matrix(1:30, 10)
y <- matrix(1:10, 2)
a <- 1:3
b <- 1:5
```

and experiment with these different operations, such as `x+a`, `x*x`, `t(x) %*% x`, etc.

**Ex. 31** — A square matrix $A$ is said to be invertible if there exists a matrix $B$ such that

$$AB = BA = I$$

Where $I$ is the identity matrix (1 along the diagonal, 0 elsewhere). Use R to compute the inverse of this matrix:

```r
set.seed(1234)
x <- matrix(rnorm(25), nrow=5, ncol=5)
```

(Hint: see `?solve`).

## 2.2 Control Flow

*Computer science education cannot make anybody an expert programmer any more than studying brushes and pigment can make somebody an expert painter.*

—Eric S. Raymond

**Ex. 32** — In each case, what is the value of x?

```
x <- 1==1
x <- 1==0
x <- !1==1
x <- if (1==0) 0 else 1
x <- ifelse(1==0, 1, 0)
x <- if (1==TRUE) FALSE else TRUE
x <- if (10) 1 else 0
x <- if (1==0) 0
```

**Ex. 33** — In each case, what is the value of x?

```
x <- 1==0 || 1>0
x <- 1==0 && 1 > 0
x <- !(1==0) || !(1>0)
x <- (!1==0 && !1>0) == !(1==0 || 1>0)
x <- if (x <- 10 > 0 && TRUE < 11) TRUE else FALSE
```

**Ex. 34** — In each case, what is the value of x?

```
x <- NaN == NaN
x <- Inf == Inf
x <- Inf == -Inf
x <- NA == NA
x <- NULL == NULL
```

## 2.3 Loops

*When someone says: 'I want a programming language in which I need only say what I wish done', give him a lollipop.*

—Alan J. Perlis

**Ex. 35** — A `while()` loop can emulate a for loop. Rewrite

```
for (i in 1:n) dothing()
```

as a while loop. You can make `dothing()` the function of your choice, or just write the pseudocode.

**Ex. 36** — An *infinite loop* is a bug where your program is trapped inside of a loop forever. The simplest infinite loop is

```
while(TRUE){}
```

So that we can see what's really going on, here is a more instructive example:

```
i <- 0
while(TRUE){
  i <- i+1
  print(i)
}
```

Try running this code, and remember that the key sequence `ctrl+c` breaks computation (including infinite loops).

**Ex. 37** — Suppose we wish to take the square root of the numbers 1 to 1000. Do this in each of the following ways, timing each one:

- for loop without initialization
- for loop with initialization
- lapply
- vectorized

**Ex. 38** — Project Euler is a website that contains hundreds of programming challenges, and is an invaluable resource for learning a new language (or programming in general). Go to `https://projecteuler.net`, register an account, and solve the first problem using R: `https://projecteuler.net/problem=1`. Make sure that your program solves the problem in under a second.

**Ex. 39** — "Fizzbuzz" is a simple programming challenge often used at interviews to test very basic programming skill. Your goal is the following: for the numbers 1 to 100, print "fizz" if the number is a multiple of 3, "buzz" if the number is a multiple of 5, "fizzbuzz" if the number is a multiple of *both* 3 and 5, and simply print the number otherwise.

**Ex. 40** — The Fibonacci Numbers are an important integer sequence. The sequence is usually recursively defined as follows:

- $F_0 = F_1 = 1$
- For $n > 1$, $F_n = F_{n-1} + F_{n-2}$

So for example $F_2 = F_1 + F_0 = 1 + 1 = 2$. Write an R script that, for any non-negative integer $n$ produces the $n$'th Fibonacci Number. Test that your script works by verifying that $F_{25} = 121393$.

**Ex. 41** — Find all primes smaller than 1000.

- Hint: start with all numbers and figure out how to eliminate all non-primes.
- Strong hint: `https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes`

**Ex. 42** — Sorting a list of numbers is a shockingly important problem in programming. Consider the vector `x <- c(10, 5, 3, 6, 1, 4, 2, 8, 7, 9)`.

- Sort the vector using the `sort()` function.
- Sort the vector in reverse order using the `sort()` function (Hint: see `help("sort")`).
- (Difficult challenge) Sort the vector using the `sample()` function.
- (Very difficult challenge) Sort the vector using the `Sys.sleep()` function.

**Ex. 43** — Newton's Method is an algorithm for finding approximations to roots (values where a function is 0). It says that if $f$ is a real-valued function of a real variable, with derivative $f'$. Take initial guess $x_0$, and for each $n > 0$, define

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}$$

Then it can be shown that under certain conditions, the sequence of $x$'s will converge to a root of $f$.

Let $f(x) = x^4 - 9x^3 - 334x^2 + 4416x - 10080$, so that $f'(x) = 4x^3 - 27x^2 - 668x + 4416$. Take an initial, uniform random guess from $-10$ to 10 and perform 50 iterations of Newton's Method to find an approximation of a root. This polynomial has 4 real roots; which did you find?

**Ex. 44** — Imagine a high school with 1000 lockers all in a row, numbered 1 to 1000 in order. At the start, all of them are closed. 1000 students are sent, one after the other, to change the state of a set of lockers (from open to closed or closed to open). The first student changes the state of all lockers. The second changes the state of every other one (2, 4, 6, 8, ...). The third changes the state of every third one (3, 6, 9, 12, ...). This process continues until all 1000 students have gone. Which lockers are open at the end of this process? For bonus points, explain *why* these are the remaining lockers.

## 2.4 Functions

*Controlling complexity is the essence of computer programming.*

—Brian Kernighan

**Ex. 45** — Write a function that takes 3 numbers $a$, $b$, and $c$ as inputs and returns the smallest number of the three.

**Ex. 46** — Write a function that takes an array of numbers $x$ and returns the smallest number in the array.

**Ex. 47** — Dungeons and Dragons is a game played with polyhedral dice beyond the standard 6-sided dice. For example, you might need to roll a 20-sided die 3 times and sum the result; this is referred to as "3d20" as a kind of shorthand. Write a function which will roll $m$d$n$ (for integers $m$ and $n$).

**Ex. 48** — Take the code from Exercise 39 and turn it into a function, where instead of stopping at 100, the printing will stop at a user-supplied input `n`.

**Ex. 49** — Take the code from Exercise 40 and turn it into a function that produces the n'th Fibonacci number.

**Ex. 50** — Take the code from Exercise 44 and turn it into a function that produces a vector of the open lockers, starting from an initial set of `n` lockers, where `n` is a user-supplied input.

**Ex. 51** — The factorial of a non-negative integer $n$, notated $n!$, can be algebraically defined as

$$\begin{aligned} n! &= \prod_{i=0}^{n-1} (n-i) \\ &= n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1 \end{aligned}$$

In the case where $n = 0$, $n! = 1$ (empty products are always 1). Write a function which recursively computes the factorial. Write a second function which computes the factorial without recursion.

**Ex. 52** — Write a function which recursively computes the n'th Fibonacci number.

**Ex. 53** — Zeller's congruence is an algorithm for taking a date and returning the day of the week (Monday, Tuesday, ...). It says that the day of the week $h$ is given by[2]:

$$h = \left(q + \left\lfloor \frac{13(m+1)}{5} \right\rfloor + K + \left\lfloor \frac{K}{4} \right\rfloor + \left\lfloor \frac{J}{4} \right\rfloor - 2J\right) \mod 7$$

Where $h$ is the index of the day of the week (0 for Saturday, 1 for Sunday, ...), $q$ is the day of the month, $m$ is an indexing of the month (3 for March, 4 for April, ..., 14 for February — so January and February are counted with the actual previous year), $K$ is the year $\mod 100$, and

[2]Formula from `https://en.wikipedia.org/wiki/Zeller%27s_congruence`

$J$ is $\left\lfloor \frac{year}{100} \right\rfloor$. Write a function which takes in the day, month and year (by ordinary accounting), and returns the day of the week. Test that your function is working with the fact that January 1, 2013 was on a Tuesday, and March 1, 2013 was on a Friday.

## 2.5 Debugging

*One of my most productive days was throwing away 1000 lines of code.*

—Ken Thompson

**Ex. 54** — Find the bug:

```
x <- 0:9
if (x[1] = 1){
  print(x)
}
```

**Ex. 55** — Find the bug:

```
x <- 0:9
if (x[0] == 1){
  print(x)
}
```

**Ex. 56** — Find the bug:

```
myfactorial <- function (x)
{
  if (x==1)
    return(1)
  else
    return( x*myfactorial(x) )
}
```

**Ex. 57** — Use the `debug()` function to debug this function:

```
f <- function(X)
{
  scl <- sum(as.numeric(X$a))
  ans <- scl * (as.numeric(X$a)+X$b)
  ans <- crossprod(ans)

  return(ans)
}

X <- list(a=factor(-2:2), b=matrix(1:30, nrow=10))
f(X)
```

So that for input

```
X <- matrix(1:30, 10)
```

it produces the correct output:

```
     [,1] [,2] [,3]
[1,]    0    0    0
[2,]    0    0    0
[3,]    0    0    0
```

**Ex. 58** — Find the bug:

```
f <- function(n)
{
  if (n==1)
    return(1)
  else {
    if (n%%2==0)
      return(n/2)
    else
      return(3*x)
  }
}

x <- 1
f(x)
n <- 3
f(n)
```

**Ex. 59** — Find the bug:

```
f <- function(n, p=NULL)
{
  if (n-p)
    1
  else
    0
}

f(1)
```

**Ex. 60** — Find the bug:

```
f <- function(n, p=-1)
{
  if (sqrt(p) == 1)
    1
  else
    0
}

f(1)
```

**Ex. 61** — Find the bug:

```r
x <- factor(c(1,3,5))
as.numeric(x)
```

**Ex. 62** — Find the bug:

```r
f <- function(.)
{
  if (runif(1) > 0.5)
    x <- 1

  return(x)
}

x <- 0
sapply(1:10, f)
x
```

# Part III
# Hints and Solutions

**Answer (Ex. 4)** — No difference. The default thing for R to do with an object (in this case, a string) is to print it.

**Answer (Ex. 6)** — The values of `x` are:

```
[1] 0
[1] -2
[1] 2
[1] 2
[1] 5
```

**Answer (Ex. 7)** — The values of `x` are:

```
[1] 5
[1] 50
[1] -9 -7 -5 -3 -1  1  3  5  7  9
[1] 0
```

**Answer (Ex. 8)** — One possible solution is:

```
sample(c("BANG", "click"), size=1, prob=c(1/6, 5/6))
```

**Answer (Ex. 13)** — `cat()` "concatenates representations"; sort of like function evaluation for strings. `sprintf()` is the **s**tring formatted **print**. `\n` is a newline character and `\t` is a tab. Unpacking the rest should be simple.

**Answer (Ex. 14)** — The answer is `Congratulations! You found the secret message.`, which you can see for yourself by entering:

```
paste(d, a, c, b, collapse="", sep="")
```

**Answer (Ex. 15)** — There are more sophisticated ways to do this, but one simple solution is:

```
swap_case_char <- function(char)
{
  if (char %in% letters)
    return( LETTERS[which(letters==char)] )
  else if (char %in% LETTERS)
    return( letters[which(LETTERS==char)] )
  else
    return(char)
}

swap_case <- function(s)
{
  if (!is.character(s))
    stop("Input 's' must be of character type.")

  ltrs <- unlist(strsplit(s, split=""))
  ltrs <- sapply(ltrs, swap_case_char)
  swapped <- paste(ltrs, collapse="")

  return(swapped)
}
```

**Answer (Ex. 16)** — Yes and no, respectively:

```
x <-
"731325141684888033065267003539040921837681246121710683413852278388760839
80810076720259728099703071896458985469817030799082795202767001808520915393
387225831438601712164218673812904093530076256033088848614152313 7"

y <-
"731325141684888033065267003539040921837681246121710683413852278388760839
80810076720259728099703071896458985469817030799082795202767007808520915393
3872258314386017121642186738129040935300762560330888486141523137"

is.palindrome <- function(str)
{
  paste(rev(unlist(strsplit(str, split=""))), collapse="") == str
}

is.palindrome(x)
is.palindrome(y)
```

**Answer (Ex. 17)** — One possible solution is:

```
chomp <- function(s)
{
  ltrs <- unlist(strsplit(s, split=""))

  stars <- which(ltrs=="&")

  leftofs <- stars-1
  leftofs <- leftofs[which(leftofs>0)]

  removes <- c(leftofs, stars)

  if (length(removes) > 0)
    ltrs <- ltrs[-removes]

  ret <- paste(ltrs, collapse="")

  return(ret)
}
```

**Answer (Ex. 18)** —

```
mdy("2/3/13")
dmy("2/3/13")
mdy("January 1, 1970")
```

**Answer (Ex. 19)** —

```
date <- mdy("2-3-13")
week(date)
wday(date, label=TRUE)

date <- dmy("2/3/13")
week(date)
wday(date, label=TRUE)

date <- mdy("January 1, 1970")
date <- mdy("2-3-13")
week(date)
```

**Answer (Ex. 20)** —

```
df[, 1:2]
```

**Answer (Ex. 21)** —

```
df[c(1, nrow(df)), ]
```

**Answer (Ex. 23)** —

```
df2 <- df
df2$q3 <- 0
df2$q4 <- 0
```

**Answer (Ex. 24)** —

```
df[order(df$gender), ]
```

**Answer (Ex. 25)** — No and yes, respectively:

```
any(duplicated(df))
any(duplicated(df2))
```

**Answer (Ex. 29)** — A possible solution is:

```
x <- 1:100
mn <- sum(x)/length(x)
var <- sum((x-mn)^2)/(length(x)-1)
```

**Answer (Ex. 32)** — The values of x are:

```
[1] TRUE
[1] FALSE
[1] FALSE
[1] 1
[1] 0
[1] FALSE
[1] 1
[1] NULL
```

**Answer (Ex. 33)** — The values of x are:

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

**Answer (Ex. 34)** — The values of x are:

```
[1] NA
```

```
[1] TRUE
[1] FALSE
[1] NA
logical(0)
```

The final value is a length zero logical vector. Be careful when comparing numbers and use the `is._` functions!

**Answer (Ex. 35)** —

```r
# for (i in 1:n)
n <- 10

i <- 1
while (i <= n){
  dothing()
  i <- i+1
}
```

**Answer (Ex. 37)** — See the lecture notes.

**Answer (Ex. 39)** — One possible solution is:

```r
for (i in 1:100){
  if (i%%3==0) {
    if (i%%5==0){
      print("fizzbuzz")
    } else {
      print("fizz")
    }
  } else if (i%%5==0) {
    print("buzz")
  } else {
    print(i)
  }
}
```

**Answer (Ex. 40)** — One possible solution is:

```r
n <- 26

if (n==0 || n==1){
  print(1)
} else {
  prev1 <- prev2 <- 1
  i <- 2
  while(i < n){
    fib <- prev1 + prev2
    prev2 <- prev1
    prev1 <- fib
    i <- i+1
  }
}

print(fib)
```

**Answer (Ex. 41)** — Using the Sieve of Eratosthenes:

```
x <- 2:1000

n <- sqrt(max(x))

for (i in x[1:n]){
  for (j in x[which(x>i)]){
    if (j%%i==0){
      x[j-1] <- 0
    }
  }
}

x[which(x>0)]
```

**Answer (Ex. 42)** —

```
sort(x)
```

```
sort(x, decreasing=TRUE)
```

- This is referred to as the slowsort or bogosort:

```
slowsort <- function(x){
    while ( FALSE %in% (diff(x) >= 0)){
      print(x)
      x <- sample(x)
    }

    return(x)
}
slowsort(x)
```

- This is referred to as the sleepsort. To do this, you will need access to something like the the system fork command. For all platforms but Windows you can do:

```
library(parallel)

sleepn <- function(n)
```

```
{
     Sys.sleep(time=n)
     return(print(n))
}

sleepsort <- function(x) invisible(mclapply(x, FUN=sleepn,
   mc.cores=length(x)))
```

**Answer (Ex. 43)** — You probably found $x = 3$:

```
1  x <- runif(1, -10, 10)
2
3  i <- 1
4  while(i<20){
5    x <- x -
        (x^4-9*x^3-334*x^2+4416*x-10080)/(4*x^3-27*x^2-668*x+4416)
6    i <- i+1
7  }
8
9  x
```

It would really be better to take line 5 above and turn the two evaluations (at the function and the derivative) and turn them into R functions.

**Answer (Ex. 44)** — One possible solution is:

```
lockers <- rep(0, 1000)

for (i in 1:1000){
  for (j in seq(from=i, to=1000, by=i)){
    lockers[j] <- 1-lockers[j]
  }
}

which(lockers>0)
```

Notice that these are the square of the values 1 to 31. Do you see why? Try thinking through it mathematically. This is a useful demonstration of how simulation can be used to answer "difficult" (maybe not too difficult here) mathematical questions, but not necessarily give any insight into the solution.

**Answer (Ex. 45)** — One possible solution is:

```
f <- function(a, b, c)
{
  ans <- min(a, b, c)
  return(ans)
}
```

Another more instructive one is:

```
g <- function(a, b, c)
{
  ans <- a
  if (b < ans){
    ans <- b
  }

  if (c < ans){
    ans <- c
  }

  return(ans)
}
```

**Answer (Ex. 46)** — One possible solution is:

```
f <- function(x)
{
  ans <- min(x)
  return(ans)
}
```

Another more instructive one is:

```
g <- function(x)
{
  ans <- x[1]
  for (i in 2:length(x)){
    if (x[i] < ans){
      ans <- x[i]
    }
  }

  return(ans)
}
```

**Answer (Ex. 47)** — One possible solution is:

```
dice <- function(m, n)
{
  ans <- sum(sample(1:n), size=m, replace=TRUE)
  return(ans)
}
```

**Answer (Ex. 51)** — For the first:

```
factorial1 <- function(n)
{
  if (n <= 2){
    return(n)
  } else {
    return(n*factorial1(n-1))
  }
}
```

And for the second:

```
factorial2 <- function(n)
{
  if (n <= 2){
    ans <- n
  } else {
    ans <- 2
    while(n > 2){
      ans <- ans*n
      n <- n-1
    }
  }

  return(ans)
}
```

**Answer (Ex. 52)** — One possible solution is:

```
fibonacci <- function(n)
{
  if (n==0 || n==1) {
    return(n)
  } else {
    return( fibonacci(n-1) + fibonacci(n-2) )
  }
}
```

**Answer (Ex. 54)** — Use `==` for comparison, not `=` (which can be used for assignment).

**Answer (Ex. 55)** — Vectors in R are indexed from 1, not 0 (like in C). The vector `x` contains no 0'th element.

**Answer (Ex. 56)** — Calling `f(x)` from inside any function `f` will cause infinite recursion. The call should instead be `x*myfactorial(x-1)`.

**Answer (Ex. 57)** — Remember that conversion of factors to numeric data is often not straightforward. Try casting the factor as character first in the `scl <-` assignment.

**Answer (Ex. 58)** — Type `rm(x)` then re-run `f(n)`. Now look at your variable names in the function definition...

**Answer (Ex. 59)** — `n-p` is a length 0 numeric vector. You might instead want to do `if (isTRUE(n-p))`, or choose a better value for `p` and screen for "bad" values (like `NULL`).

**Answer (Ex. 60)** — `sqrt(p)` is `NaN`, and a comparison of `NaN` with a number by `==` is `NA`, which is not valid for conditionals. Probably the solution in this case is to replace `==` with `identical(sqrt(p), 1)`, although you will receive a warning upon function evaluation.

**Answer (Ex. 61)** — This is well-known, documented behavior that eventually catches everyone by surprise. Factors are actually stored as integer vectors, and so casts (like `as.numeric()`) will operate on the *data*, not the data substituted for its coded level. The easiest way to convert to numeric is `as.numeric(as.character(x))`.

**Answer (Ex. 62)** — This is a famous example from Ross Ihaka, one of the co-creators of the R language. It demonstrates R's "lexical scoping" rules. Here inside the function's scope, the object `x` is *randomly* locally defined to `f` (as 1) or a global value (as 0). You are generally encouraged to avoid relying on global variables in R, as it can make your programs difficult to "reason about" (which is programmer/jerk speak for "understand").