R Programming

Succinctly®

by James McCaffrey

R Programming Succinctly

By

James McCaffrey

Foreword by Daniel Jebaraj

Syncfusion®
Deliver innovation with ease®

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to "enable AJAX support with one click," or "turn the moon to cheese!"

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and "Like" us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

James McCaffrey works for Microsoft Research in Redmond, WA. He holds a B.A. in psychology from the University of California at Irvine, a B.A. in applied mathematics from California State University at Fullerton, an M.S. in information systems from Hawaii Pacific University, and a doctorate from the University of Southern California. James enjoys exploring all forms of activity that involve human interaction and combinatorial mathematics, such as the analysis of betting behavior associated with professional sports, machine learning algorithms, and data mining.

Chapter 1 Getting Started

The R programing language is designed to perform statistical analyses. Surveys of programming languages show that the use of R is increasing rapidly, apparently in conjunction with the increasing collection of data.

R can be used in two distinct ways. Most commonly, R is used as an interactive tool. For example, in an R console shell, a user could type commands such as:

```
> my_df <- read.table("C:\\Data\\AgeIncome.txt", header=TRUE)
> m <- lm(Income ~ Age, data=my_df)
> summary(m)
```

These commands would perform a linear regression data analysis of the age and income data stored in file AgeIncome.txt and would display the results of the analysis. R has hundreds of built-in functions that can perform thousands of statistical tasks.

However, you can also write R programs (i.e. scripts) by placing R commands and R language-control structures such as for-loops into a text file, saving the file, and executing the file. For example, a user can type commands such as:

```
> setwd("C:\\Data\\MyScripts")
> source("neuralnet.R")
```

These commands will run an R program named neuralnet.R, which is located in a C:\Data\MyScripts directory. You gain tremendous power and flexibility with the ability to extend the base R interactive functionality by writing R programs.

In this e-book, I will explain how to write R programs. I make no assumptions about your background and experience—even if you have no programming experience at all, you should be able to follow along with a bit of effort.

I will present a complete demo program in each section. Programmers learn how to program in a new language by getting an example program up and running, then experimenting by making changes. So, if you want to learn R programming, copy-paste the source code from a demo program, run the program, then make modifications to the program.

I will not present hundreds of one-line R examples. Instead, I will present short (but complete) programs that illustrate key R syntax examples and techniques. The code for all the demo programs can be found at https://github.com/jdmccaffrey/r-programming-succinctly.

In my experience, the most difficult part of learning any programming language or technology is getting a first program to run. After that, it's just details. But getting started can be frustrating, and the purpose of this first chapter is to make sure you can install R and run a program.

Enough chit-chat already. Let's get started.

1.1 Installing R

It's no secret that the best way to learn a programming language or technology is to use it. Although you can probably learn quite a bit about R simply by reading this e-book, you'll learn a lot more if you install R and run the demo programs that accompany each section.

Installing R is relatively quick and easy, and I'll walk you through each step of the installation process for a Windows system. If you're using a Linux system, the installation process varies quite a bit depending on which flavor of Linux you're running, but there are many step-by-step installation guides available on the Internet. With Mac systems, R installation is very similar to Windows installation.

Launch a browser, search the Internet for "Install R," and you'll find a link for the Windows installer. At the time of this writing, the Windows installation URL is https://cran.r-project.org/bin/windows/base. Clicking the link will direct your browser to a page that resembles Figure 1.
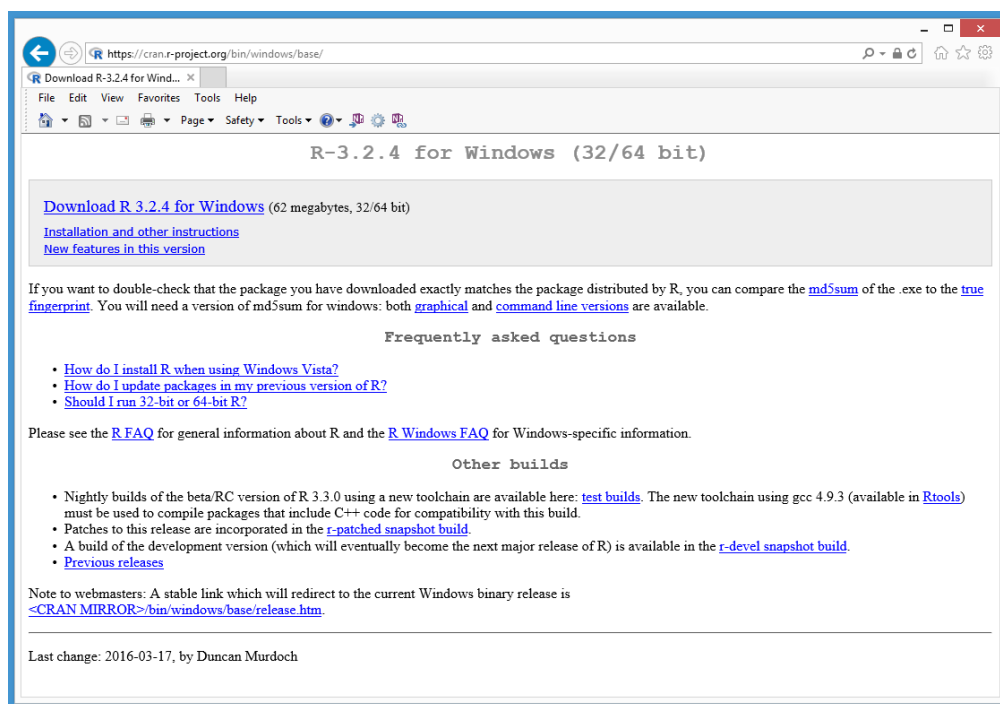
Figure 1: Windows Installation Page

The screenshot shows that I intend to install R version 3.2.4. By the time you read this, the most recent version of R will likely be different—however, the base R language is quite stable, and the code examples in this e-book have been designed to work with newer versions of R.

Notice the webpage title reads "(32/64 bit)." By default, on a 64-bit system (which you're almost certainly using), the R installer will give you a 32-bit version of R and also a 64-bit version. The 32-bit version is for old machines and for backward compatibility with older R add-on packages that can only use the 32-bit version of R.

Next, click the link that reads "Download R 3.x.y for Windows." The link points to a self-extracting executable installation program. Your browser will ask if you want to download the installer to your machine so that you can run the installation later, or it will ask if you want to run the install program immediately. Click "Run" to launch the installer.
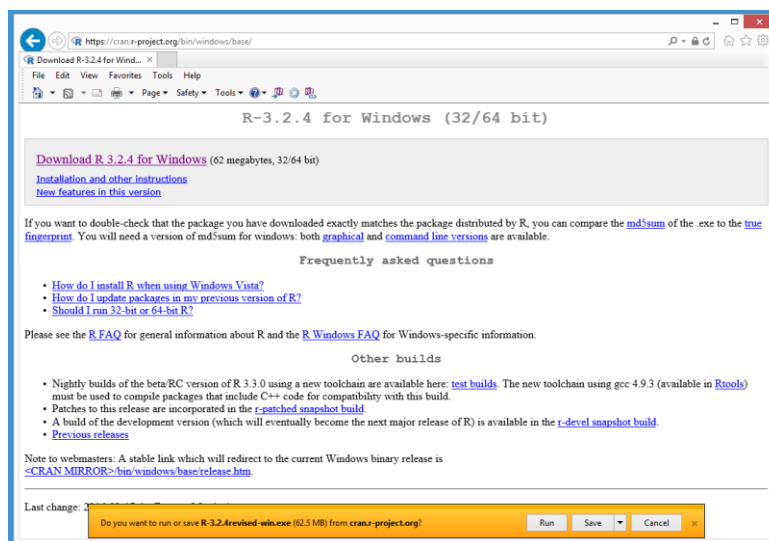
Figure 2: Run the Installer Now or Later

The installation program will start up and display a small dialog box asking you to select a language. One of R's strengths is that it supports many different spoken languages. Select your language from the drop-down list and click "OK."
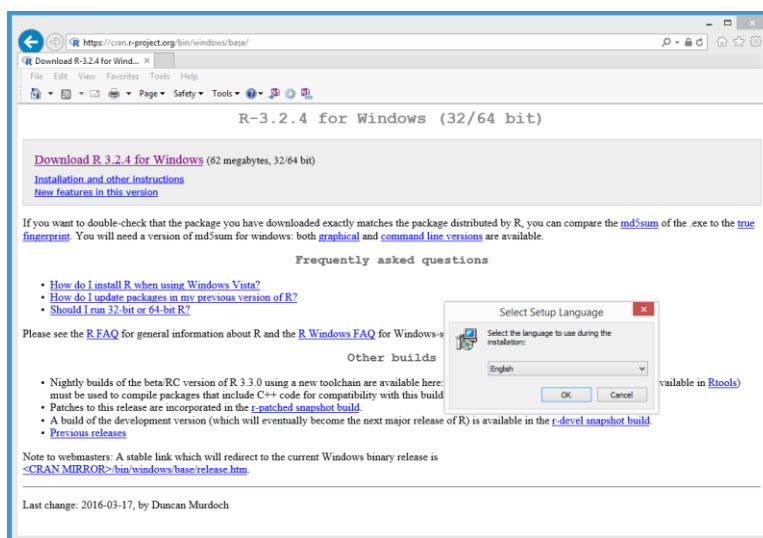
Figure 3: Select Language

Next, the installation program will display a Welcome window. Note that you will need administrative privileges on your machine in order to install R. Click "Next."
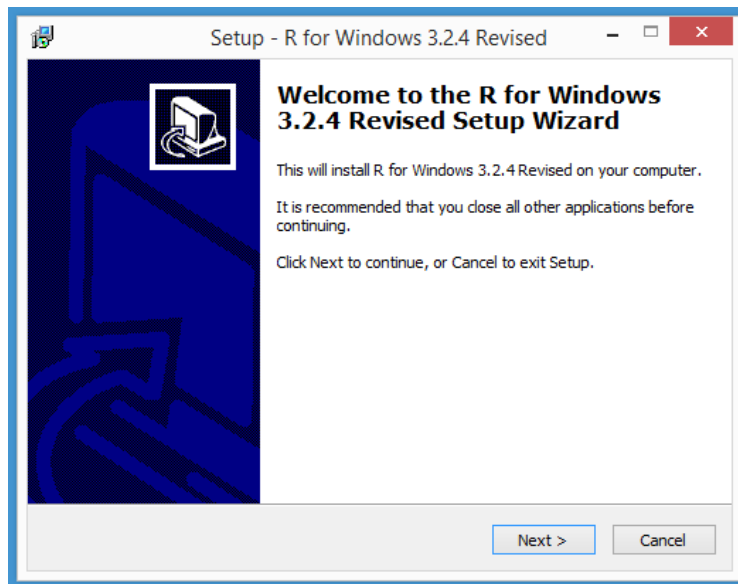
Figure 4: Installation Welcome Window

The installer will display the licensing information. If you look at the window scroll bar, you'll notice there's a lot of information. R runs under several different open source licenses. Read all the information if you're a glutton for legal punishment or just click "Next."
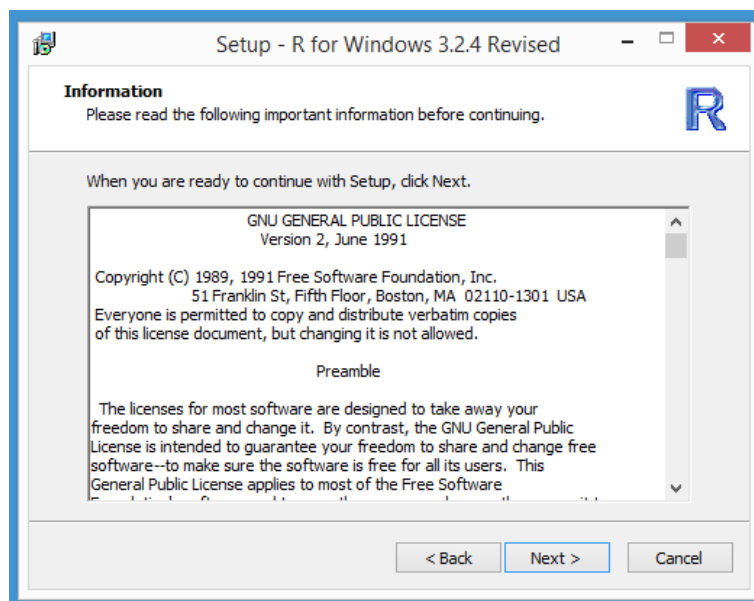
Figure 5: Licensing Information

Next, the installer will ask you to specify the installation directory. The default location is C:\Program Files\R\R-3.x.y, and I recommend you use the default location unless you have a good reason not to. Click "Next."
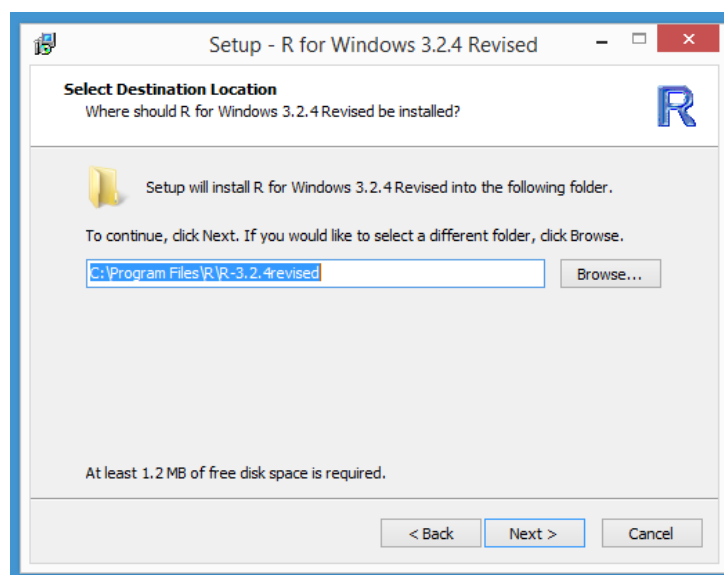
Figure 6: Specify the Installation Location

Next, the installer will ask you to select from four components. By default, all components are selected. Unless your machine has a memory shortage, you can leave all components selected and click "Next." You need at least the core files and either the 32-bit files or the 64-bit files component.
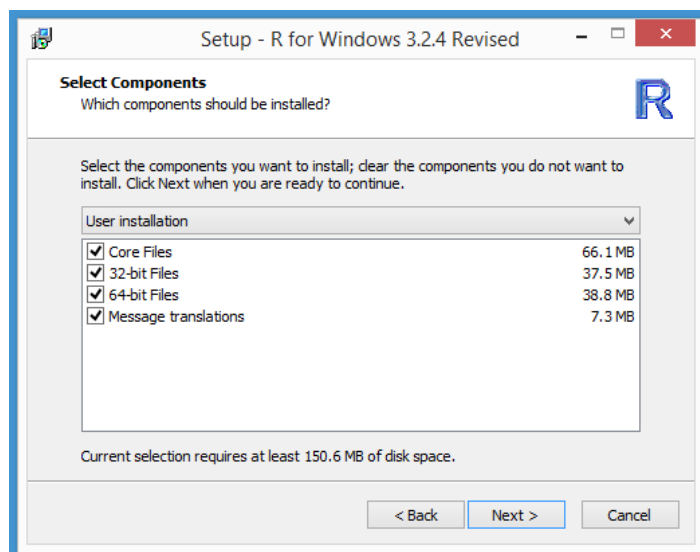
Figure 7: Select Components to Install

Next, the installer will give you the option to customize your startup options. These options are contained in a file named Rprofile.site, and they control items such as the default editor. I recommend accepting all the defaults and clicking "Next."
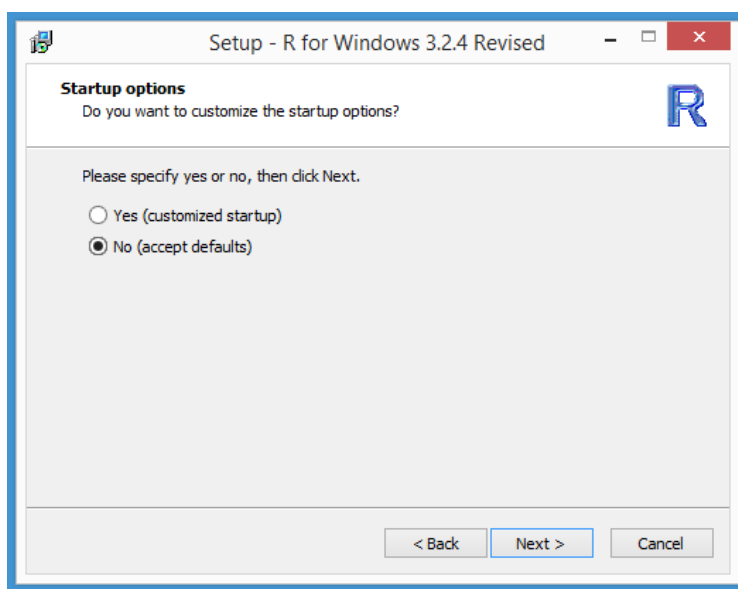
Figure 8: Specify the Startup Options

Next, the installer will ask where to place the program shortcuts. Accept the default location of "R" and click "Next."
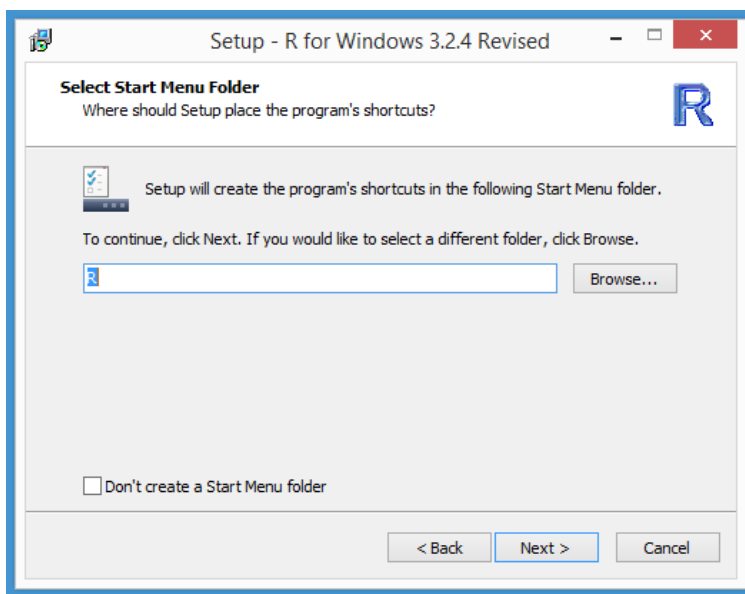
Figure 9: Select the Start Menu Folder

Next, the installer will ask you for some final options. The selected default options are fine—just click "Next."
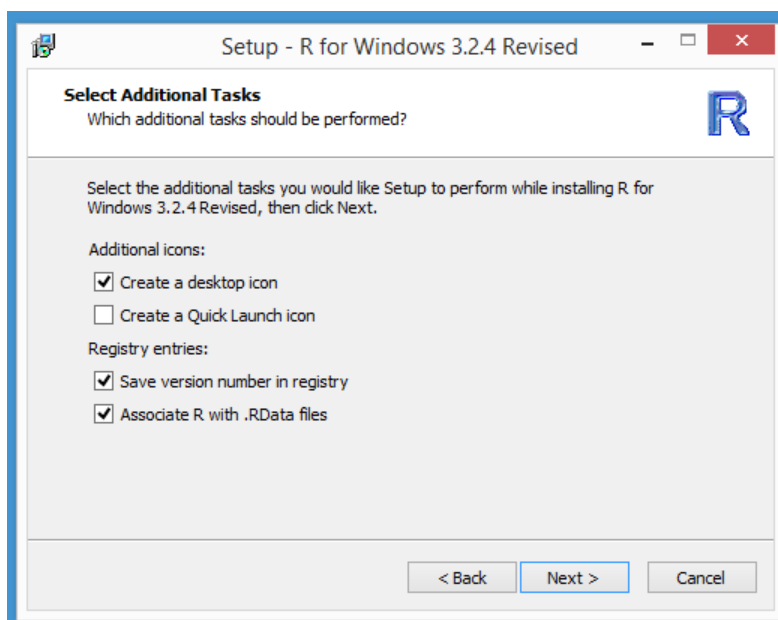
Figure 10: Additional Installation Options

Next, the actual installation process will begin. You'll see a window that displays the installation progress. Installation is very quick—it should take no more than a minute or two.
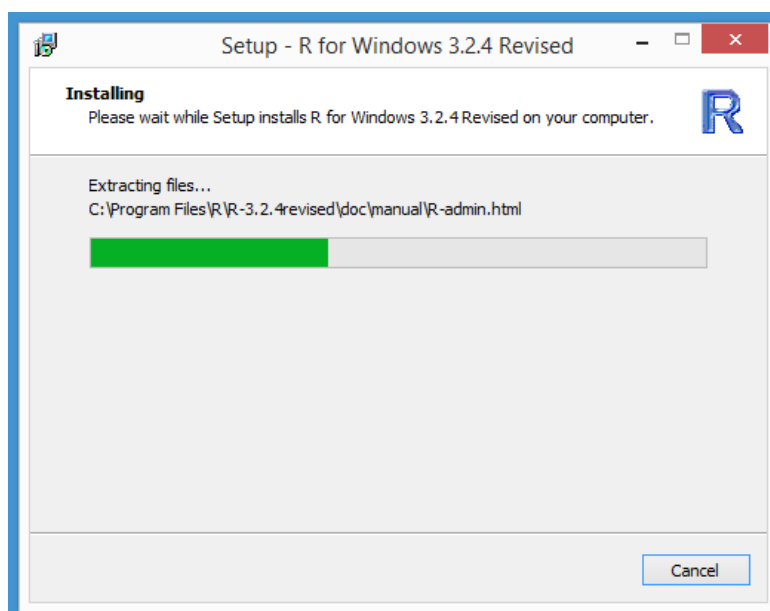
Figure 11: Installation Progress

When the install process completes, you'll see a final window. Click "Finish" and you'll be ready to start using R.
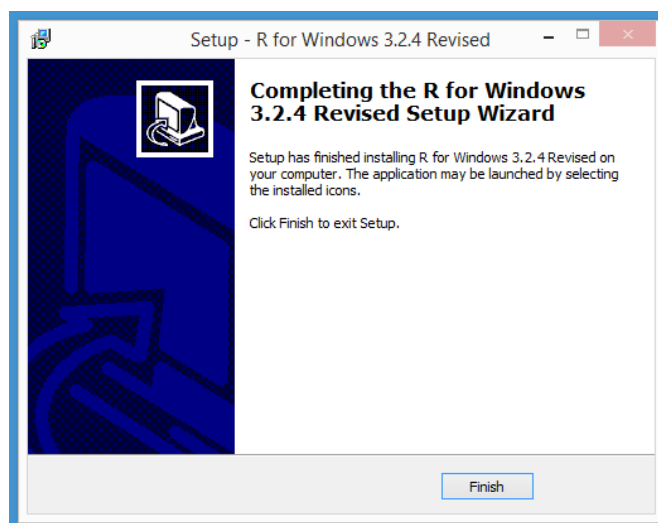
Figure 12: Installation is Complete

If you accepted the default install location, the R installation process places most of the key files at C:\Program Files\R\R-3.x.y\bin\x64. I recommend that you take a quick look there.
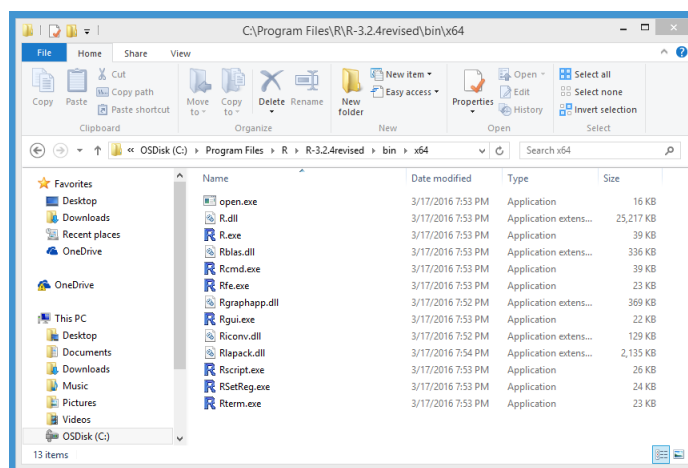
Figure 13: The Key R Files

In summary, installing R is relatively quick and easy. On a Windows system, the installer is a self-extracting executable. You can accept all the installation default options. The installation process will give you both a 32-bit version (primarily for backward compatibility with old add-on packages) and a 64-bit version.

1.2 Editing and running R programs

You can edit and execute an R program in several ways. If you are new to the R language, I recommend using the Rgui.exe program that is installed with R. Open a File Explorer window and navigate to the C:\Program Files\R\R-3.x.y\bin\x64 directory (or \i386 if you're using an old 32-bit machine).

Locate the Rgui.exe file. You can double-click on Rgui.exe to launch it, or you can right-click the file, then select the "Run as administrator" option. For editing and running simple programs, you can simply double-click. However, if you need to install add-on packages, you must run Rgui.exe with administrative privileges.

After Rgui.exe launches, you'll see an outer-shell window that contains an RConsole window. The RConsole window allows you to issue interactive commands directly to R. In order to create an R program, go to the menu bar and select **File | New Script**. This will create an R Editor window inside the shell window, where you can write an R program.
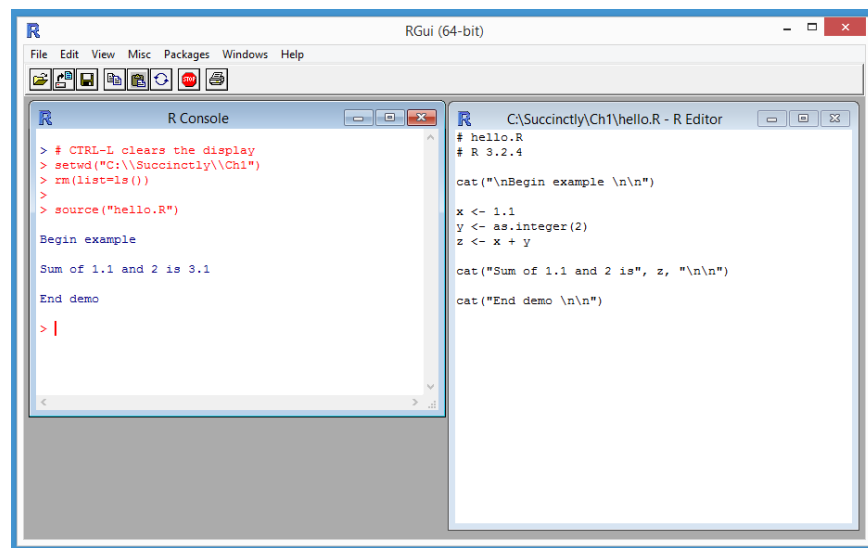
Figure 14: Using the Rgui.exe Program

In the R Editor window, type or copy-paste this code:

```
# hello.R
# R 3.2.4

cat("\nBegin example \n\n")

x <- 1.1
y <- as.integer(2)
z <- x + y

cat("Sum of 1.1 and 2 is", z, "\n\n")
cat("End demo \n\n")
```

You must save the program before running it. **Click File | Save As**, then navigate to any convenient directory and save the program as hello.R. I saved my program at C:\Succinctly\Ch1.

To run the program, click anywhere on the RConsole window, which will give it focus. If you wish to erase all the somewhat annoying R startup messaging, you can do so by typing CTRL-L. Next, type the following commands in the RConsole window:

```
> setwd("C:\\Succinctly\\Ch1")
> rm(list=ls())
> source("hello.R")
```

The `setwd()` command sets the working directory to the location of the R program. Notice the required C-family language syntax of double backward slashes. Because R is multiplatform, you can also use single forward slashes if you wish.

The `rm(list=ls())` command can be thought of as a magic R incantation that deletes all R objects currently in memory. You should always issue this command before running an R program. Failure to do so can lead to errors that are very difficult to track down.

The `source()` command is used to execute an R program. Technically, R programs are scripts because R is interpreted rather than compiled. However, I use the terms "program" and "script" interchangeably in this e-book.

After you make a change to your code, you can save using the standard shortcut CTRL-S or by selecting the **File | Save** option. When you close the Rgui.exe program, you'll be presented with a dialog box that asks if you want to "Save workspace image?" You can click "No."

A workspace image consists of the R objects currently in memory. Because a program recreates objects, saving the image isn't necessary. Typically, you save an image when you've been using R interactively and have typed dozens or even hundreds of commands, and you don't want to lose the state of all the objects you've created. If you do save an R image, the image is saved with a .RData file extension. Workspace images can be restored by using the **File | Load Workspace** menu option.

In summary, I recommend using the Rgui.exe program in order to write and execute an R program. You write and edit programs in an Editor window and run programs by issuing a `source()` command in the RConsole window. There are dozens of alternatives for editing R programs. I sometimes use the open source Notepad++ program to write R programs because it gives me nice source code coloring. Some of my colleagues use the open source RStudio program.

Resources

The `source()` function has many useful options to control running an R program. See:
https://stat.ethz.ch/R-manual/R-devel/library/base/html/source.html.

1.3 Using add-on packages

The R language is organized into packages. When you install R, you get about 30 packages with names like base, graphics, and stats. These built-in packages allow you to perform many programming tasks, but you can also install and use hundreds of packages that have been created by the R community.

Add-on packages are both a strength and weakness of R. Because anyone can write an R package, quality can vary greatly. And because packages can have dependencies on other packages, if you install lots of packages, it's possible that you'll run into versioning problems. In order to avoid this, in this e-book I use only the built-in R packages that come with the R install.

You can issue the command `installed.packages()` in the R Console window in order to see which packages are currently installed on your system. Note that unlike most programming languages, R often uses the "." character rather than the "_" character as part of a function name, which makes the name more readable. If you are an experienced programmer, this syntax can be surprisingly difficult at first.

In order to install a package, you use the `install.packages()` function. For example, there is the BigInteger data type. When installing the gmp add-on package, make sure you're connected to the Internet, then type the command `install.packages("gmp")`. This will launch a small window that allows you to select a website from which to install the gmp package.
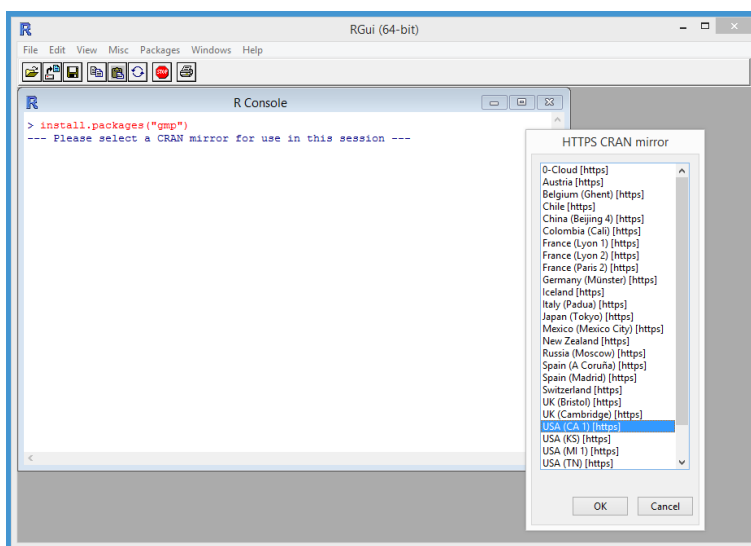
Figure 15: Selecting a Mirror when Installing an Add-On Package

You can select any of the mirror websites, then click "OK." The package installation process is silent and generally relatively quick. On rare occasions, the installation process will fail, typically because the mirror website is down or the mirror site doesn't have the requested package. If this happens, you'll get an error message. You can try a different mirror site by issuing the command `chooseCRANmirror()`.
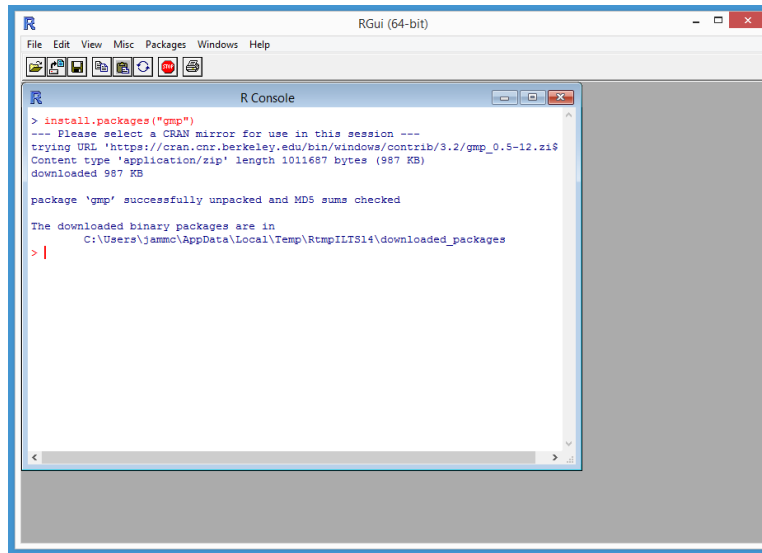
Figure 16: Package Installed Successfully

After your add-on package has been installed successfully, you access the package by using the **library()** or **require()** function. For example, in an interactive mode, you can type the command **library(gmp)**, then you can use all the functions and classes defined in the gmp package. For example, these commands will display the value of 20 factorial (as shown in Figure 17):

```
> library(gmp)
> f = factorialZ(20)
> f
```
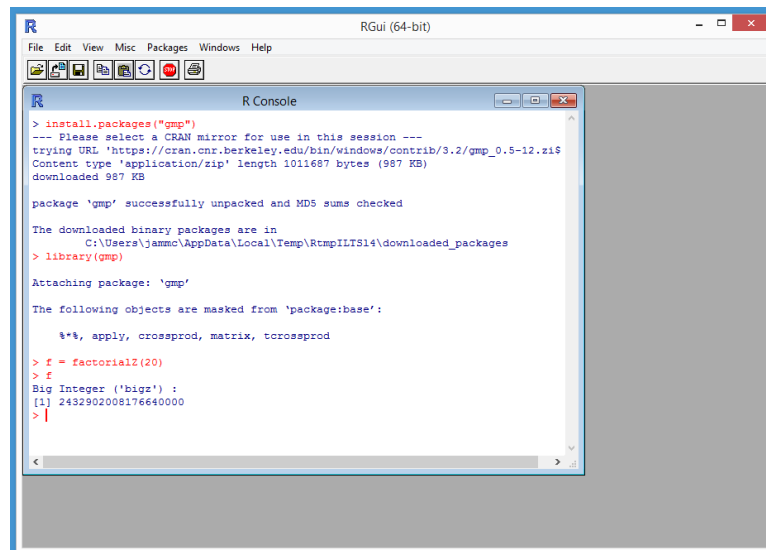

Figure 17: Accessing an Add-On Package

Both the **library()** function and the **require()** function load an installed package into the current context, but **library()** will return an error and halt execution if the target package is not installed. However, **require()** will return a value of **FALSE** and attempt to continue execution if the target package is not installed. After you finish using an add-on package, you should consider uninstalling the package in order to prevent your system from becoming overwhelmed with packages.

In order to remove a package, first list the installed packages using the **installed.packages()** function. In most cases, when removing packages, you'll want to list all installed packages because you might have several instances of the same package that contain different versions. You can list just a few of the installed packages or just some of the information about each by specifying the number of rows and columns in the output. For example, the command **installed.packages()[1:5,1:3]** will list rows 1-5 of the installed packages table with columns 1-3 of the table information. In order to remove a package, use the **remove.packages()** function; for example, **remove.packages("gmp")** or **remove.packages("gmp", version="0.5-12")**.
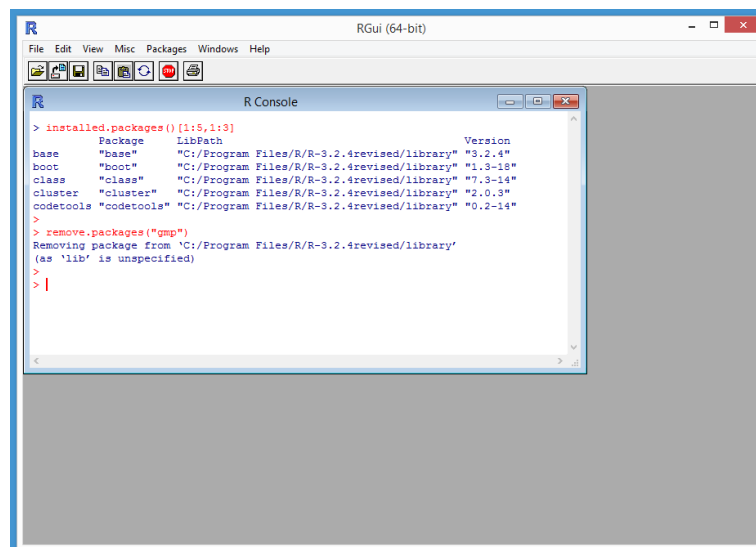
Figure 18: Removing a Package

In summary, installing R gives you about 30 built-in packages that contain most of the functionality you need for common programming tasks. In order to manage add-on packages, you can use the **installed.packages()**, **install.packages()**, and **remove.packages()** functions. In order to access an add-on package, you can use either the **library()** function, which will fail if the requested package is not installed on your system, or the **require()** function.

Resources

For details about the differences between the **library()** and **require()** functions, see:
https://stat.ethz.ch/R-manual/R-devel/library/base/html/library.html.

1.4 R Syntax and style reference program

The program in Code Listing 1 gives a quick reference for the syntax of common R language features such as if-then and for-loop statements.

Code Listing 1: R Language Syntax Examples

```r
# syntaxdemo.R                           # comments start with '#'
# R 3.2.4                                # filename and version

tri_max = function(x, y, z) {            # program-defined function
  if (x > y && y > z) {                  # logical AND
    return(x)                            # return() requires parens
  }
  else if (y > z) {                      # C-style braces OK here
    return(y)
  }
  else {
    return(z)
  }
}

my_display = function(v, dec=2) {        # default argument value
  # display vector v to console
  n <- length(v)                         # built-in length()
  for (i in 1:n) {                       # for-loop
    x <- v[i]                            # 1-based indexing
    xf <- formatC(x,                     # built-in formatC()
      digits=dec, format="f")            # you can break long lines
    cat(xf, " ")                         # basic display function
  }
  cat("\n\n")                            # print two newlines
}

my_binsearch = function(v, t) {          # program-defined function
  # search sorted integer vector v for t
  lo <- 1
  hi <- length(v)
  while (lo <= hi) {                     # while loop
    mid <- as.integer(round((lo + hi) / 2))  # built-in round()
    if (v[mid] == t) {                   # equality
      return(mid)
    } else if (v[mid] < t) {             # R-style braces optional
      lo <- mid + 1
    } else {
      hi <- mid - 1
    }
  }
  return(0) # not found                  # could just use 0 here
}

# -----                                  # functions must be defined
```

```r
cat("\nBegin R program syntax demo \n\n")

xx <- 4.4; yy <- 6.6; zz <- 2.2           # multiple values, one line
mx <- tri_max(xx, yy, zz)                 # function call
cat("Largest value is", mx, "\n\n")       # use ',' or paste()

v <- c(1:4)                               # make vector of integers
decimals <- 3                             # '<-' or '=' assignment
my_display(v, decimals)                   # override default 2 value

v <- vector(mode="integer", length=4)     # make vector of integers
v[1] <- 9; v[2] <- 6; v[3] <- 7; v[4] <- 8  # multiple statements
t <- 7
idx <- my_binsearch(v, t)
if (idx >= 1) {                           # R-style braces required here
  cat("Target ", t, "in cell", idx, "\n\n")
} else {
  cat("Target", t, "not found \n\n")
}

cat("End syntax demo \n\n")
```

If you're new to R, a few syntax quirks might be confusing. In R, there are two assignment operators: the `<-` operator and the `=` operator. In most situations, either assignment operator can be used.

The R language is one of the few languages in which vectors, arrays, and lists use 1-based indexing rather than 0-based indexing.

In an if-then-else statement, curly braces are required even in the case of a single then-statement. Inside a function definition, you can use the easier-to-read C-style in which the **else** keyword and the right curly brace from the **if** appear on separate lines. However, outside of a function definition, you must use R-style when the **else** and the right curly brace are on the same lines.

Although several R language style guidelines have been proposed and published, there is little agreement in the R community about what constitutes good R language style. In my opinion, consistency and common sense are more important than a slavish attention to any style guide.

In summary, R language syntax is quite similar to the C language. R uses both the `<-` and `=` operators for assignments (there's also a specialized `<<-` operator used with reference classes). R collections are 1-based rather than 0-based. You can use easy-to-read, if-then C-style syntax inside a function definition, but you must use R-style syntax outside a code block definition.

Resources

The official R language definition can be found at:
https://cran.r-project.org/doc/manuals/r-release/R-lang.html.

Chapter 2 Vectors and Functions

Arguably the five most fundamental objects in the R language are vectors, lists, arrays, matrices, and data frames. We should note, however, that R uses terminology somewhat differently than most other programming languages. An R vector corresponds to what is called an array in other languages, and R arrays and matrices correspond to an array-of-arrays style data structure in other languages. Being able to work with the five basic objects and write functions that operate on them are key skills.

Figure 19's screenshot gives you an idea of where this chapter is headed.
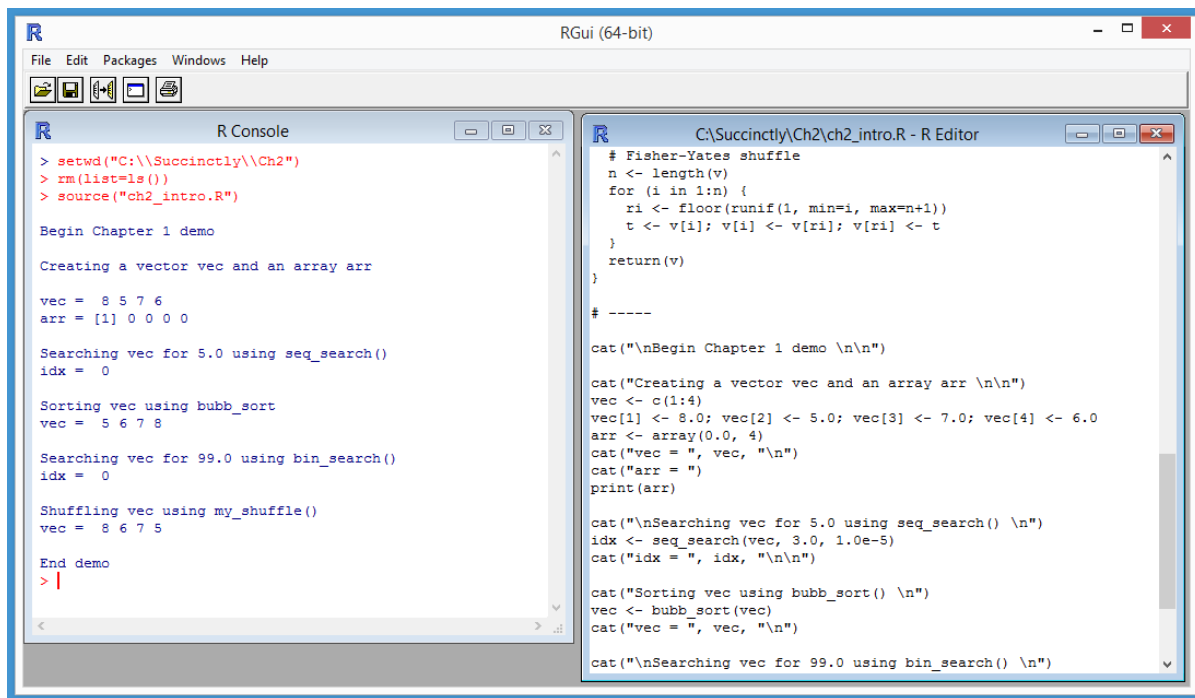
Figure 19: Vectors and Functions Demo

In section 2.1, we'll look at the five basic collection objects in R (vectors, lists, arrays, matrices, and data frames) and you'll learn how to write program-defined functions that work with these collection objects. In section 2.2, we'll see how to perform a sequential search on a vector using built-in functions such as **which.max()** and **is.element()** and also by using program-defined functions.

In section 2.3, you'll learn how to perform a binary search using a program-defined function. In section 2.4, we'll examine how to sort vectors using the built-in **sort()** function and how to write a program-defined sorting function. Finally, in section 2.5, you'll learn how to shuffle the values in a vector and how to select samples of the values in a vector.

2.1 Vectors, lists, matrices, arrays, and data frames

An R **vector** has a fixed number of cells, and each cell holds a value of the same data type (integer, character, etc.). A **list** can hold values with different types, and the length of a list can change during runtime. A **matrix** has a fixed number of cells in exactly two dimensions (rows and columns), and each cell holds the same data type.

An **array** has a fixed number of cells in one, two, three, or more dimensions, and each cell holds the same data type. A **data.frame** has columns in which the number of values and the data types can be different.

Code Listing 2: The Five Basic Data Structures in R

```r
# vectorsVsArrays.R
# R 3.2.4

cat("\nBegin vectors vs. arrays demo \n\n")

cat("Creating three demo vectors \n\n")
v <- c(1:3)  # [1 2 3]
cat(v, "\n\n")

v <- vector(mode="numeric", 4)  # [0.0 0.0 0.0 0.0]
cat(v, "\n\n")

v <- c("a", "b", "x")
cat(v, "\n\n")

cat("Creating two demo lists \n\n")
ls <- list("a", 2.2)
ls[3] <- as.integer(3)
print(ls)
cat("\n")
cat("Cell [2] is: ", ls[[2]], "\n\n")

ls <- list(lname="Smith", age=22)
cat("Cells by cell names are: ", ls$lname, "-", ls$age)
cat("\n\n")

cat("Creating a 2x3 matrix \n\n")
m <- matrix(0.0, nrow=2, ncol=3)
print(m)
cat("\n")

cat("Creating 1 and 2-dim arrays \n\n")
arr <- array(0.0, 3)  # [0.0 0.0 0.0]
print(arr)
cat("\n")

arr <- array(0.0, c(2,3))  # 2x3 matrix
print(arr)
cat("\n")
```

```
# arr = array(0.0, c(2,5,4)) # 2x5x4 n-array
# print(arr)  # 40 values displayed
# cat("\n")

cat("Creating a data frame \n\n")
people <- c("Alex", "Barb", "Carl")
ages <- c(19, 29, 39)
df <- data.frame(people, ages)
names(df) <- c("NAME", "AGE")
print(df)

cat("\nEnd vectors vs. arrays demo \n\n")
```

```
> setwd("C:\\Succinctly\\Ch2")
> rm(list=ls())
> source("vectorsVsArrays.R")

Begin vectors vs. arrays demo

Creating three demo vectors

1 2 3

0 0 0 0

a b x

Creating two demo lists

[[1]]
[1] "a"

[[2]]
[1] 2.2

[[3]]
[1] 3


Cell [2] is:  2.2

Cells by cell names are:  Smith - 22

Creating a 2x3 matrix

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

Creating 1 and 2-dim arrays
```

```
[1] 0 0 0

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

Creating a data frame

  NAME AGE
1 Alex  19
2 Barb  29
3 Carl  39

End vectors vs. arrays demo
```

The demo program is launched by issuing a **setwd()** command to point to the source code directory, calling the **rm()** command to remove all existing objects in the current context and using the **source()** command to begin program execution. Next, the demo creates and displays three vectors:

```
v <- c(1:3)  # [1 2 3]
cat(v, "\n\n")

v <- vector(mode="numeric", 4)  # [0.0 0.0 0.0 0.0]
cat(v, "\n\n")

v <- c("a", "b", "x")
cat(v, "\n\n"
```

The **c()** function's name is short for "combine," and this function can be used to create a vector when you know the initial cell values.

The cell data type (**integer**, **numeric** / **double**, **character**, **logical**, **complex**, **raw**) of a vector is inferred by the values passed to the **c()** function. If you want to explicitly specify the cell type, you can use the **vector()** function with a **mode** parameter value as shown.

Vector cell values are accessed using 1-based indexing—for example, **v[1] <- 3.14** or x <- v[3]. Next, the demo program shows how to create, manipulate, and access a simple list:

```
ls <- list("a", 2.2)
ls[3] <- as.integer(3)
print(ls)
cat("Cell [2] is: ", ls[[2]], "\n\n")
```

The first statement creates a list with two values, the character *a* and the numeric 2.2. The second statement dynamically extends the length of the list by adding a new integer value of 3 at cell index [3]. As with almost all R composite objects, lists use 1-based indexing rather than 0-based indexing.

Notice that in order to access the value in cell [2] of the list, the demo uses double square brackets, `ls[[2]]`, rather than single square brackets (which would return a list object with one value). However, in order to store a value in cell [3] of the list, the demo uses single square brackets: `ls[3] <- as.integer(3)`. Two rules of thumb that are usually, but not always, the correct methods—first, use double square brackets on the right side of an assignment statement or when standing alone; and second, use single square brackets on the left side of an assignment statement or when accessing an item in the list.

Next, the demo creates a list in which cell values can be accessed by name as well as by index:

```
ls <- list(lname="Smith", age=22)
cat("Cells by cell names are: ", ls$lname, "-", ls$age)
```

The list named `ls` represents a person by storing the last name and age. Notice that the cell values are accessed using the `$` token. As another general rule of thumb, for clarity's sake it's preferable to access by cell name rather than by index when using a list to store related values.

Next, the demo creates and displays a 2x3 numeric matrix:

```
m <- matrix(0.0, nrow=2, ncol=3)
print(m)
```

The `matrix` function has several optional parameters, but the demo example uses the most common pattern. Next, the demo program illustrates the R array type:

```
arr <- array(0.0, 3)
print(arr)

arr <- array(0.0, c(2,3))
print(arr)
```

The first example creates a one-dimensional array that has three cells, each of which is initialized to 0.0. The second example creates a two-dimensional array that has two rows and three columns, with each of the six cells initialized to 0.0. Notice that a one-dimensional array can be used in situations in which a vector can be used, and a two-dimensional array can be used in situations in which a matrix can be used.

An array can have three or more dimensions. For example, `arr <- array(0.0, c(2,5,4))` creates an array with three dimensions—the first with two cells, the second with five cells, and the third with four cells, for a total of 40 cells.

Next, the demo program creates a `data.frame` object:

```
cat("Creating a data frame \n\n")
people <- c("Alex", "Barb", "Carl")
ages <- c(19, 29, 39)
df <- data.frame(people, ages)
names(df) <- c("NAME", "AGE")
print(df)
```

Although R data frames are somewhat similar to table objects in other languages, data frames are created manually by column rather than by row. The first two statements create a column vector with the names of three people and a second column vector with their ages. The `data.frame()` function has several optional parameters that give great flexibility for creating a data-frame object.

After the data frame is created, the demo supplies header names of "NAME" and AGE" for the two columns. Header names are optional, but they are useful because many built-in R functions use them.

In many situations, rather than programmatically constructing a data frame from column vectors, you'll want to create a data frame from the data stored in a text file using the `read.table()` function or the closely related `read.csv()` or `read.delim()` functions.

In summary, an R `vector` corresponds to what is called an array in most other programming languages. Vectors are accessed using 1-based indexing. An R `matrix` has exactly two dimensions. An R `array` can have one, two, or more dimensions and therefore can function like a vector or a matrix. An R `list` can hold values of different types and can change size dynamically. When accessing a list value using indexing, you should typically use double square brackets rather than single square brackets. An R `data.frame` is a table-like object, which often has columns that contain different data types.

Resources

For additional details about creating and using vectors, see:
https://stat.ethz.ch/R-manual/R-devel/library/base/html/vector.html.

For additional details about creating and using lists, see:
https://stat.ethz.ch/R-manual/R-devel/library/base/html/list.html.

For additional details about creating and using matrices, see:
https://stat.ethz.ch/R-manual/R-devel/library/base/html/matrix.html.

For additional details about creating and using arrays, see:
https://stat.ethz.ch/R-manual/R-devel/library/base/html/array.html.

For additional details about creating and using data frames, see:
https://stat.ethz.ch/R-manual/R-devel/library/base/html/data.frame.html.

2.2 Vector sequential search

The R language has several built-in ways to search a vector for a target value, including the **match()**, **is.element()**, and **which.max()** functions along with the **%in%** operator. In some situations with numeric vectors, you'll want to write a program-defined, sequential search function.

Code Listing 3: Sequential Search

```r
# seqsearching.R
# R 3.2.4

seq_search = function(v, t, eps) {
  # search vector v for target value t
  # eps is epsilon tolerance for equality

  n <- length(v)
  for (i in 1:n) {
    if (abs(v[i] - t) <= eps) {
      return(i)
    }
  }
  return(0)
}

my_print = function(v, dec) {
  n <- length(v)
  for (i in 1:n) {
    x <- v[i]
    xx <- formatC(x, digits=dec, format="f")
    cat(xx, " ")
  }
  cat("\n")
}

cat("\nBegin sequential search demo \n\n")

vec <- c(1.0, 5.0, 2.0, 3.0, 4.0)
target <- 2.0
epsilon <- 1.0e-5

cat("Vector is: ")
my_print(vec, dec=2)
cat("\n")

cat("Target is ")
cat(formatC(target, digits=1, format="f"), "\n")
cat("Epsilon is ", epsilon, "\n")
cat("Search using program-defined seq_search() \n")
idx <- seq_search(vec, target, epsilon)
cat("idx = ", idx, "\n\n")

cat("Search using base %in% operator \n")
```

```r
there <- target %in% vec
cat("there = ", there, "\n\n")

cat("Search using base which.max() \n")
idx <- which.max(vec)
cat("idx of largest = ", idx, "\n\n")

target <- c(5.0, 3.0)
cat("Target is ", target, "\n")
cat("Search using base match() \n")
matches <- match(vec, target)
cat("matches = ", matches, "\n\n")

cat("Search using base is.element() \n")
matches <- is.element(vec, target)
cat("matches = ", matches, "\n\n")

cat("End demo\n")
```

```
> source("seqsearching.R")

Begin sequential search demo

Vector is: 1.0   5.0   2.0   3.0   4.0

Target is 2.0
Epsilon is   1e-05
Search using program-defined seq_search()
idx =   3

Search using base %in% operator
there =   TRUE

Search using base which.max()
idx of largest =   2

Target is   5 3
Search using base match()
matches =   NA 1 NA 2 NA

Search using base is.element()
matches =   FALSE TRUE FALSE TRUE FALSE

End demo
```

The simplest way to search a vector for a value is to use the **%in%** operator. In the demo program, the key code is:

```
vec <- c(1.0, 5.0, 2.0, 3.0, 4.0)
target <- 2.0
cat("Search using base %in% operator \n")
there <- target %in% vec
cat("there = ", there, "\n\n")
```

The return result stored into the variable **there** is **TRUE** because the target value 2.0 is in the source vector at index [3]. Using the **%in%** operator is simple and effective in most situations. However, none of the R built-in search mechanisms give you control over exactly what equality means when you are comparing two floating-point values.

Because floating-point values are only approximations, comparing two floating-point values for exact equality can be very tricky. For example:

```
> x <- 0.15 + 0.15  # 0.30
> y <- 0.20 + 0.10  # 0.30
> if (x == y) { cat("equal") } else { cat("NOT equal") }
> "NOT equal"  # what the heck?
```

Instead of comparing two floating-point values for exact equality, it's usually preferable to use the built-in **abs()** (absolute value) function to check whether or not the two values are very close. The demo program defines a custom search function that uses this approach:

```
seq_search = function(v, t, eps) {
  n <- length(v)
  for (i in 1:n) {
    if (abs(v[i] - t) <= eps) {
      return(i)
    }
  }
  return(0)
}
```

The function walks through each cell of the source vector v using a for-loop. If the current value being checked is close to within a small value eps (epsilon), the function returns the index of the target value. If none of the cell values is close enough to the target value, the function returns 0, which is a nonvalid vector index.

Here is the key calling code:

```
vec <- c(1.0, 5.0, 2.0, 3.0, 4.0)
target <- 2.0
epsilon <- 1.0e-5

cat("Target is ")
cat(formatC(target, digits=1, format="f"), "\n")
cat("Epsilon is ", epsilon, "\n")
cat("Search using program-defined seq_search() \n")
idx <- seq_search(vec, target, epsilon)
cat("idx = ", idx, "\n\n")
```

Different problem domains tend to use different values of epsilon, but the value of 0.00001 used in the demo is quite common in machine learning. There are several ways to display floating-point values with nice formatting. My preference is to use the built-in `formatC()` function.

The built-in `which.max()` function returns the index that contains the largest value in a vector. For example:

```
vec <- c(1.0, 5.0, 2.0, 3.0, 4.0)
idx <- which.max(vec)
```

These statements return a value of 2 into variable `idx` because the largest value in `vec` is 5.0, which is located at cell [2]. There is a similar built-in `which.min()` function, too.

The built-in `match()` function is useful for searching a vector for multiple values. For example:

```
vec <- c(1.0, 5.0, 2.0, 3.0, 4.0)
target <- c(5.0, 3.0)
matches <- match(vec, target)
```

Here the return value stored into variable `matches` is the list **(NA, 1, NA, 2, NA)** because the first value in **vec** doesn't match any value in **target**, the second value in **vec** matches the value at [1] in **target**, and so on.

The built-in `is.element()` function is very similar to the `match()` function, except that the return list contains Boolean values rather than integer indices. For example:

```
vec <- c(1.0, 5.0, 2.0, 3.0, 4.0)
target <- c(5.0, 3.0)
matches <- is.element(vec, target)
```

These statements return a list with values **(FALSE, TRUE, FALSE, TRUE, FALSE)** because the cells in **vec** at [2] and [4] have values that match a value in **target**.

In summary, if you want to search a vector that contains floating-point values and you want to control the epsilon tolerance for equality, you can easily write a program-defined function. When searching a vector that contains values that are not floating-point types, using the **%in%** operator is simple and effective. In order to find the location of the largest value in a vector, you can use the `which.max()` function. When searching a vector for multiple values, the built-in `match()` and `is.element()` functions are good choices.

Resources

For additional details about the **match** function and the **%in%** operator, see:
https://stat.ethz.ch/R-manual/R-devel/library/base/html/match.html.

For more information about the `which.max()` and `which.min()` functions, see:
https://stat.ethz.ch/R-manual/R-devel/library/base/html/which.min.html.

2.3 Vector binary search

When searching a sorted vector or list that has many cells, a binary search algorithm is much faster than a sequential search. The base R language doesn't have a binary search algorithm. Instead, you can use the **binsearch()** function from the gtools package or write your own binary search function.

Code Listing 4: Binary Search

```r
# binsearching.R
# R 3.2.4

bin_search = function(v, t, eps) {
  # search sorted vector v for target value t
  # eps is epsilon tolerance for equality
  lo <- 1
  hi <- length(v)
  while (lo <= hi) {
    mid <- as.integer(round((lo + hi) / 2)) # always even!
    cat("lo, mid, hi = ", lo, mid, hi, "\n")

    if (abs(v[mid] - t) <= eps) {
      return(mid)
    }
    else if (v[mid] < t) { # C format OK in a function
      lo <- mid + 1
    }
    else {
      hi <- mid - 1
    }
  }
  return(0)
}

cat("\nBegin binary search demo \n\n")

vec <- c(1.5, 3.5, 5.5, 7.5, 9.5, 11.5, 13.5, 15.5, 17.5, 19.5)
target <- 17.5
epsilon <- 1.0e-5

cat("Vector is: \n")
print(vec)
cat("Target is ", target, "\n")
cat("Epsilon is ", epsilon, "\n\n")

cat("Begin search \n\n")
idx <- bin_search(vec, target, epsilon)
if (idx == 0) {
  cat("\nTarget not found \n\n")
} else {
  cat("\nTarget found at cell index ", idx, "\n\n")
}
```

```
cat("End demo \n")
```

```
> source("binsearching.R")

Begin binary search demo

Vector is:
 [1]  1.5  3.5  5.5  7.5  9.5 11.5 13.5 15.5 17.5 19.5
Target is  17.5
Epsilon is  1e-05

Begin search

lo, mid, hi =  1 6 10
lo, mid, hi =  7 8 10
lo, mid, hi =  9 10 10
lo, mid, hi =  9 9 9

Target found at cell index  9

End demo
```

The demo program begins by setting up a vector with 10 floating-point values, a target value to search for, and an epsilon tolerance that determines how close two floating-point values must be in order to be evaluated as equal:

```
vec <- c(1.5, 3.5, 5.5, 7.5, 9.5, 11.5, 13.5, 15.5, 17.5, 19.5)
target <- 17.5
epsilon <- 1.0e-5

cat("Vector is: \n")
print(vec)
cat("Target is ", target, "\n")
cat("Epsilon is ", epsilon, "\n\n")
```

The program-defined binary search function **bin_search()** is called this way:

```
idx <- bin_search(vec, target, epsilon)
if (idx == 0) {
  cat("\nTarget not found \n\n")
} else {
  cat("\nTarget found at cell index ", idx, "\n\n")
}
```

The return value is either 0 if the target value is not found, or it is the index of the first occurrence of the target value if the target is in the vector.

The structure of the **bin_search()** function looks like this:

```
bin_search = function(v, t, eps)
  lo <- 1
  hi <- length(v)
  while (lo <= hi) {
    # compute a midpoint
    # if target at midpoint return index
    # otherwise search left or sight part of v
  }
  return(0)
}
```

Although simple in principle, the binary search algorithm is quite subtle and has many possible implementation variations. In order to compute a midpoint between indices **lo** and **hi**, the demo function uses this statement:

```
mid <- as.integer(round((lo + hi) / 2))
```

Compared to many other programming languages, the R **round()** function is unusual because it uses IEEE 754-2008 "round ties to even." Here is the heart of the binary search implementation:

```
if (abs(v[mid] - t) <= eps) {  # found it!
  return(mid)
}
else if (v[mid] < t) {  # search left side
  lo <- mid + 1
}
else {  # search right side
  hi <- mid - 1
}
```

The Wikipedia entry on the binary search algorithm addresses several alternative variations of the core search-left, search-right implementation.

In summary, the base R language doesn't have a binary search function. The gtools package has a **binsearch()** function, but it doesn't allow you to control the epsilon tolerance for floating-point equality, so in some situations you'll want to write a custom binary search function.

Resources

For information about the gtools **binsearch()** function, see:
http://svitsrv25.epfl.ch/R-doc/library/gtools/html/binsearch.html.

For details about the surprisingly tricky **round()**, **ceiling()**, and related functions, see:
https://stat.ethz.ch/R-manual/R-devel/library/base/html/Round.html.

2.4 Vector sorting

The base R language has a **sort()** function that can be used to arrange the values in a vector. The **sort()** function can use three different sorting algorithms: shell sort, quicksort, and radix sort. You can also write a custom sorting function for special situations.

Code Listing 5: Sorting

```
# sorting.R
# R 3.2.4

bubb_sort = function(v) {
  # -----
  exchange = function(ii, jj) {
    tmp <<- v[ii]
    v[ii] <<- v[jj]
    v[jj] <<- tmp
  }
  # -----
  n <- length(v)
  for (i in 1:(n-1)) {
    for (j in (i+1):n) {
      if (v[i] > v[j]) {
        exchange(i, j)
        # tmp = v[i]; v[i] = v[j]; v[j] = tmp
      }
    }
  }
  return(v)
}

my_print = function(v, h, t) {
  n <- length(v)
  cat("[1]     ")
  cat(head(v, h), "\n")
  cat(" . . . ", "\n")
  idx <- n-t+1
  cat("[", idx, "]   ", sep="")
  cat(tail(v, t), "\n\n")
}

cat("\nBegin sorting demo \n\n")

cat("Generating 5,000 random values \n")
set.seed(0)
vec <- rnorm(5000)
cat("Vector to sort is : \n")
my_print(vec, 4, 4)
cat("\n")

cat("Sorting vector using built-in shell sort() \n")
start_t <- proc.time()
sorted <- sort(vec, method="shell")
```

```r
end_t <- proc.time()
times <- end_t - start_t
cat("Sorted vector is \n")
my_print(sorted, 4, 4)
e_time <- formatC(times[3], digits=2, format="f")
cat("Elapsed time =", e_time, "sec. \n")
cat("\n")

cat("Sorting with user-defined bubb_sort() \n")
start_t <- proc.time()
sorted <- bubb_sort(vec)
end_t <- proc.time()
times <- end_t - start_t

cat("Sorted vector is \n")
my_print(sorted, 4, 4)
cat("Elapsed time =", times[3], "sec.\n")
cat("\n")

cat("Verifying result is sorted using is.sorted() \n")
unsorted <- is.unsorted(sorted)
if (unsorted == T) {
  cat("Error. Result not sorted \n\n")
} else {
  cat("Result verified sorted \n\n")
}

cat("End demo \n")
```

```
> source("sorting.R")

Begin sorting demo

Generating 5,000 random values
Vector to sort is :
[1]    1.262954 -0.3262334 1.329799 1.272429
 . . .
[4997]  2.162363 1.175956 0.6190771 0.008463216


Sorting vector using built-in shell sort()
Sorted vector is
[1]    -3.703236 -3.236386 -3.115391 -3.082364
 . . .
[4997]  2.961743 3.023597 3.039033 3.266415

Elapsed time = 0.00 sec.

Sorting with user-defined bubb_sort()
Sorted vector is
[1]    -3.703236 -3.236386 -3.115391 -3.082364
```

```
   . . .
[4997]  2.961743 3.023597 3.039033 3.266415

Elapsed time = 27.57 sec.

Verifying result is sorted using is.sorted()
Result verified sorted

End demo
```

The demo program begins by creating a vector that has 5,000 random values:

```
set.seed(0)
vec <- rnorm(5000)
cat("Vector to sort is : \n")
my_print(vec, 4, 4)
```

Unlike some programming languages that allow you to create a random number generation object, R has one global random number generator. Calling the **set.seed()** function allows you to get reproducible results.

By default, the **rnorm()** function returns values that are normal (Gaussian) distributed with mean = 0.0 and standard deviation = 1.0, which means the vast majority of the 5,000 demo values will be between -4.0 and +4.0.

When working with very large vectors, you will typically find it impractical to display all the cell values. It's often useful to use the built-in **head()** or **tail()** function to display a few of the values at the beginning or end of the vector. The demo program defines a custom **my_print()** function that allows you to see a few cells at both the beginning and end of a vector.

The demo program calls the **sort()** function this way:

```
start_t <- proc.time()
sorted <- sort(vec, method="shell")
end_t <- proc.time()
times <- end_t - start_t
```

The demo sandwiches the call to **sort()** with calls to the **proc.time()** function in order to determine how long the sorting will take. Notice that because R functions call by value rather than reference, the result is stored into a new vector object named **sorted**.

The **proc.time()** function returns a vector with three values. The value for determining an elapsed time is the value at cell [3]:

```
e_time <- formatC(times[3], digits=2, format="f")
cat("Elapsed time =", e_time, "sec. \n")
```

Interestingly, if a **method** parameter value is not supplied to the **sort()** function, the default algorithm depends on the data type of the cell values. For vectors that contain integers, Boolean values, or factor values, the default algorithm is the radix sort. For all other vector types (such as floating-point values and character values), the default is the shell sort algorithm.

The demo has a program-defined function that implements a crude bubble sort algorithm. In general, you'll only want to implement a custom sort function in unusual scenarios. Because the bubble sort algorithm is the slowest reasonable algorithm, it is sometimes used as a baseline.

```
bubb_sort = function(v) {
  # -----
  exchange = function(ii, jj) {
    tmp <<- v[ii]; v[ii] <<- v[jj]; v[jj] <<- tmp
  }
  # -----
  n = length(v)
  for (i in 1:(n-1)) {
    for (j in (i+1):n) {
      if (v[i] > v[j]) { exchange(i, j) }
    }
  }
  return(v)
}
```

The custom **bubb_sort()** function has a nested function named **exchange()** that swaps the values to be sorted in the array. Notice that **exchange()** has access to parameter **v**, but in order to change **v** the special **<<-** operator must be used. The **bubb_sort()** function is called this way:

```
sorted <- bubb_sort(vec)
unsorted <- is.unsorted(sorted)
if (unsorted == T) { cat("Error. Result not sorted \n\n") }
```

In summary, the base R language can sort vectors using the built-in **sort()** function, which can use the shell sort algorithm, the quicksort algorithm, or the radix sort algorithm. You can time function calls in R using the value in cell [3] of the return result from the **proc.time()** function. The R language supports nested-function definitions, but in my opinion those definitions are rarely worth the trouble. The built-in **is.unsorted()** function can be used to test whether or not a vector is sorted.

Resources

For detailed information about the base **sort()** function, see:
https://stat.ethz.ch/R-manual/R-devel/library/base/html/sort.html.

For details about the **rnorm()** function and related **dnorm()**, **pnorm()**, and **qnorm()**, see:
https://stat.ethz.ch/R-manual/R-devel/library/stats/html/Normal.html.

2.5 Vector sampling and shuffling

The base R language has a versatile **sample()** function that can select a subset of numbers from a given range and also shuffle the values in a vector to a random order. You can also write custom functions to sample and shuffle vector values when you want specialized behavior.

Code Listing 6: Sampling and Shuffling

```r
# sampling.R
# R 3.2.4

my_sample = function(N, k) {
  # select k random ints between 1 and N
  result <- c(1:k) # [1, 2, . . k]
  for (i in (k+1):N) {  # reservoir sampling algorithm
    j <- floor(runif(1, min=1.0, max=i+1))
    if (j <= k) {
      result[j] = i
    }
  }
  return(result)
}

my_shuffle = function(v) {
  # Fisher-Yates shuffle
  n = length(v)
  for (i in 1:n) {
    ri <- floor(runif(1, min=i, max=n+1))
    t <- v[i]; v[i] <- v[ri]; v[ri] <- t
  }
  return(v)
}

cat("\nBegin vector sampling demo \n\n")
set.seed(20) # arbitrary

N <- 9
k <- 4
cat("N = ", N, "\n")
cat("k = ", k, "\n\n")

cat("Sampling 4 items using program-defined my_sample() \n")
samp <- my_sample(N, k)
cat("Sample = ", samp, "\n\n")

cat("Sampling 4 items using built-in sample() \n")
samp <- sample(N, k)
cat("Sample = ", samp, "\n\n")

cat("Sampling 4 items using built-in sample(replace=TRUE) \n")
samp <- sample(N, k, replace=T)
cat("Sample = ", samp, "\n\n")
```

```r
cat("Shuffling (1,2,3,4,5,6) using built-in sample() \n")
v <- c(1:6)
vv <- sample(v)
cat("Shuffled v = ", vv, "\n\n")

cat("Shuffling (1,2,3,4,5,6) using program-defined my_shuffle() \n")
v <- c(1:6)
vv <- my_shuffle(v)
cat("Shuffled v = ", vv, "\n\n")

cat("End demo\n")
```

```
> source("sampling.R")

Begin vector sampling demo

N =  9
k =  4

Sampling 4 items using program-defined my_sample()
Sample =  1 7 3 4

Sampling 4 items using built-in sample()
Sample =  9 1 8 2

Sampling 4 items using built-in sample(replace=TRUE)
Sample =  4 7 7 1

Shuffling (1,2,3,4,5,6) using built-in sample()
Shuffled v =  5 1 2 6 3 4

Shuffling (1,2,3,4,5,6) using program-defined my_shuffle()
Shuffled v =  5 4 3 1 2 6

End demo
```

The demo program shows how to select a random subset of a range of values using the built-in **sample()** function:

```r
set.seed(20)
N <- as.integer(9)
k <- as.integer(4)
samp <- my_sample(N, k)
cat("Sample = ", samp, "\n\n")
```

The **set.seed()** function is used so that the demo results will be reproducible. The seed value of 20 is essentially arbitrary. Variable **N** is set to 9 and is the size of the source data. Variable **k**, set to 4, is the subset size. The call to **sample(N, k)** returns four random (by default, all different) integers that have values from 1 to 9, inclusive.

The demo program defines a custom-sampling function named **my_sample()** that uses a very clever technique called reservoir sampling:

```
my_sample = function(N, k) {
  result <- c(1:k) # [1, 2, . . k]
  for (i in (k+1):N) {
    j <- floor(runif(1, min=1.0, max=i+1))
    if (j <= k) {
      result[j] = i
    }
  }
  return(result)
}
```

Selecting k random and unique values from a set of values is surprisingly tricky. The reservoir algorithm starts with (1, 2, . . , k), then it probabilistically replaces these values. The **runif()** function (random uniform) call returns one random floating-point value from 1.0 to i+1 (exclusive). The **floor()** function strips away the decimal part of the return value from the **runif()** function.

The built-in **sample()** function can return a subset in which duplicate values are allowed by passing logical **True** to the **replace** parameter:

```
samp <- sample(N, k, replace=T)
cat("Sample = ", samp, "\n\n")
```

The behavior of the built-in **sample()** function is overloaded so that the function can also act as a shuffle function. For example:

```
cat("Shuffling (1,2,3,4,5,6) using built-in sample() \n")
v <- c(1:6)
vv <- sample(v)
cat("Shuffled v = ", vv, "\n\n")
```

If you pass **sample()** a vector but no subset size, the function will sample all values in the vector, which essentially performs a shuffle operation. This technique is useful when you want to traverse all the cells in a vector in a random order (a common task in many machine-learning algorithms). For example:

```
m <- matrix(0.0, nrow=9, ncol=3)
# populate the matrix with data
indices <- sample(1:9)
for (i in 1:9) {
  idx <- indices[i]
  # process row at [idx]
```

The demo program defines a custom function named **my_shuffle()** that can shuffle the values in a vector:

```
my_shuffle = function(v) {
  n = length(v)
  for (i in 1:n) {
    ri <- floor(runif(1, min=i, max=n+1))
    t <- v[i]; v[i] <- v[ri]; v[ri] <- t
  }
  return(v)
}
```

The custom function uses a neat mini-algorithm called the Fisher-Yates shuffle. Shuffling the values in a vector into random order can be challenging, and it's very easy to write code that produces results that appear random but in fact are heavily biased toward certain patterns. Fortunately, the Fisher-Yates algorithm is very simple and effective.

The demo program calls **my_shuffle()** this way:

```
cat("Shuffling (1,2,3,4,5,6) using program-defined my_shuffle() \n")
v <- c(1:6)
vv <- my_shuffle(v)
cat("Shuffled v = ", vv, "\n\n")
```

Recall that R functions pass parameters by value rather than by reference, which means that although function **my_shuffle()** appears to manipulate its input parameter **v**, behind the scenes a copy of **v** is made so that the actual argument passed to **my_shuffle()** does not change. If you want to shuffle the source vector, you can use this calling pattern:

```
v <- my_shuffle(v)
```

In summary, the built-in **sample()** function can generate a random subset with no duplicate values, a random subset with duplicate values allowed, or a random subset of all source values with no duplicates, which is a shuffle. If you need to define a function with custom sampling behavior, you can use the reservoir-sampling algorithm, and if you need to define a function with custom shuffling behavior, you can use the Fisher-Yates mini-algorithm.

Resources

For detailed information about the **sample()** function, see:
https://stat.ethz.ch/R-manual/R-devel/library/base/html/sample.html.

For information about **runif()** and three related random functions, see:
https://stat.ethz.ch/R-manual/R-devel/library/stats/html/Uniform.html.

For more information about the reservoir-sampling algorithm, see:
https://en.wikipedia.org/wiki/Reservoir_sampling.

Chapter 3 Object-Oriented Programming

Many programming languages, including Java, C#, and Python, support an object-oriented programming (OOP) model. Unlike most programming languages, R supports three distinct OOP models called S3, S4, and Reference Class (RC). The S3 and S4 models are so named because those models were developed in the S language, the predecessor to R.

In addition to the three formal OOP models of R, you can also create an informal model that uses a list object to encapsulate data fields and associated functions (often called methods when part of an OOP class). The screenshot in Figure 20 gives an idea of where this chapter is headed.
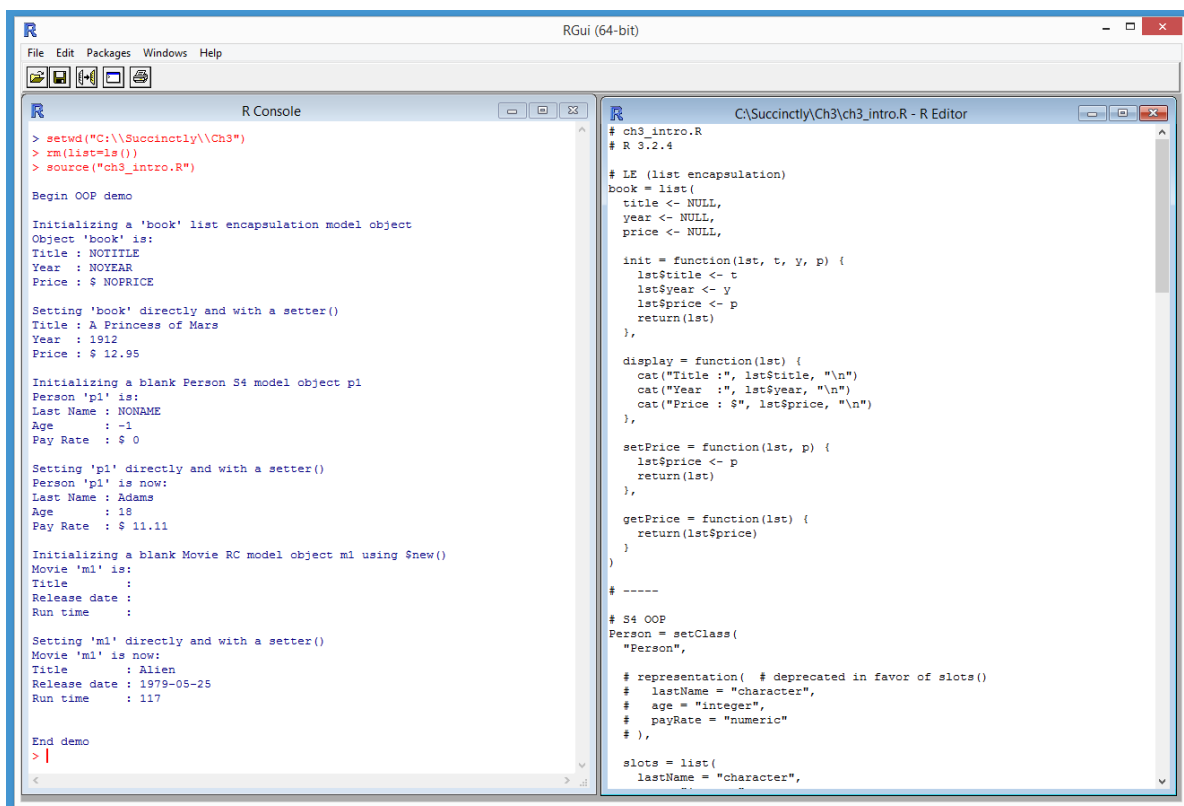
Figure 20: Object-Oriented Programming Demo

In each section of this chapter, we'll examine how to use a different model. In section 3.1, you'll learn how to create an informal, lightweight form of OOP using the R list type, called list encapsulation. In section 3.2, we'll see how to create OOP code using the old (but still useful) S3 model.

In section 3.3, you'll learn how to create OOP code using the S4 model. And, in section 3.4, we'll look at how to create OOP code using the RC model.

3.1 List Encapsulation OOP

Because R list objects can hold both variables and functions, the most basic way to implement OOP is to encapsulate related data and functions in a list. The main advantage of list encapsulation OOP compared to the alternatives of S3, S4, and RC OOP is simplicity. Because list encapsulation has no mechanism to avoid name collisions, the technique is most often useful for your personal code library rather than code intended to be used by others.

Code Listing 7: List Encapsulation Model OOP

```r
# books.R
# R 3.2.4

# LE (list encapsulation)
book = list(
  title <- NULL,
  year <- NULL,
  price <- NULL,

  init = function(lst, t, y, p) {
    lst$title <- t
    lst$year <- y
    lst$price <- p
    return(lst)
  },

  display = function(lst) {
    cat("Title :", lst$title, "\n")
    cat("Year  :", lst$year, "\n")
    cat("Price : $", lst$price, "\n")
  },

  setPrice = function(lst, p) {
    lst$price <- p
    return(lst)
  },

  getPrice = function(lst) {
    return(lst$price)
  }
)

# -----

cat("\nBegin OOP with list encapsulation demo \n\n")

# initialization
cat("Initializing a 'book' object \n")
book <- book$init(book, "NOTITLE", "NOYEAR", "NOPRICE")
cat("Object 'book' is: \n")
book$display(book)
cat("\n\n")
```

```r
# set fields
cat("Setting 'book' directly and with a setter() \n")
book$title <- "A Princess of Mars"
book$year <- as.integer(1912)
book <- book$setPrice(book, 12.95)
book$display(book)
cat("\n")

# get fields
cat("Getting title and price fields \n")
ti <- book$title
pr <- book$getPrice(book)
cat("Price of '")
cat(ti)
cat("' is ", pr, "\n")
cat("\n\n")

# assignment
cat("Calling 'tome <- book' \n")
tome <- book
cat("Object 'tome' is: \n")
tome$display(tome)
cat("\n")

cat("Modifying title of object 'tome' \n")
tome$title <- "xxxxxxxx"
cat("Object 'tome' is: \n")
tome$display(tome)

cat("\n")
cat("Original object 'book' is unchanged: \n")
book$display(book)

cat("\nEnd demo \n")
```

```
> source("books.R")

Begin OOP with list encapsulation demo

Initializing a 'book' object
Object 'book' is:
Title : NOTITLE
Year  : NOYEAR
Price : $ NOPRICE


Setting 'book' directly and with a setter()
Title : A Princess of Mars
Year  : 1912
Price : $ 12.95
```

```
Getting title and price fields
Price of 'A Princess of Mars' is  12.95


Calling 'tome <- book'
Object 'tome' is:
Title : A Princess of Mars
Year  : 1912
Price : $ 12.95

Modifying title of object 'tome'
Object 'tome' is:
Title : xxxxxxxx
Year  : 1912
Price : $ 12.95

Original object 'book' is unchanged:
Title : A Princess of Mars
Year  : 1912
Price : $ 12.95

End demo
```

The demo program defines a **book** object as a list that has a title, a year (of publication), a price, and four related functions. Notice there's no explicit way to enforce data types—for instance, the year field could be character, integer, or numeric:

```
book = list(
  title <- NULL,
  year <- NULL,
  price <- NULL,
. . .
```

With list encapsulation, you define a specific instance of an object rather than a template for creating objects. But, as I'll show shortly, you can create multiple objects using copy-by-value. The **book** object defines a function to display itself:

```
display = function(lst) {
  cat("Title :", lst$title, "\n")
  cat("Year  :", lst$year, "\n")
  cat("Price : $", lst$price, "\n")
},
```

The **book** object has an initializer function that supplies values to the three fields:

```
init = function(lst, t, y, p) {
  lst$title <- t
  lst$year <- y
  lst$price <- p
  return(lst)
},
```

Each **book** object function, including the **init()** function, has a parameter named **lst** that represents the **book** object. The purpose of this is illustrated by the calling statements:

```
cat("Initializing a 'book' object \n")
book <- book$init(book, "NOTITLE", "NOYEAR", "NOPRICE")
cat("Object 'book' is: \n")
book$display(book)
```

Notice that the **book** object appears three times in the call to **init()**. This triple-reference pattern is used in list encapsulation OOP by any function call that modifies the source object. The call to the **display()** function needs the **book** object only twice because **book** is not changed by the function call.

The demo program defines a function that modifies the price field of the **book** object:

```
setPrice = function(lst, p) {
  lst$price <- p
  return(lst)
},
```

Function **setPrice()** is called this way:

```
cat("Setting 'book' directly and with a setter() \n")
book$title <- "A Princess of Mars"
book$year <- as.integer(1912)
book <- book$setPrice(book, 12.95)
book$display(book)
```

In list encapsulation, all fields have public scope, which means they can be accessed directly by using the **$** operator, as shown here when the title and year fields are modified. Or, you can write a set-function and call it using the triple-reference pattern.

The demo defines an accessor get-function for the price field:

```
getPrice = function(lst) {
  return(lst$price)
}
```

This is the demo program code that calls **getPrice()**:

```
cat("Getting title and price fields \n")
ti <- book$title
pr <- book$Price(book)
cat("Price of '"); cat(ti); cat("' is ", pr, "\n")
```

List encapsulation object fields can be accessed directly or by using a get-function. Setting field values and getting field values directly, using the **$** operator, is much simpler than writing and calling set-functions and get-functions. So why use functions? My personal rule of thumb is that I write a list encapsulation OOP function only if the function uses two or more fields or performs a nontrivial calculation.

In this example, if the code is intended for actual use rather than as a demo, I would write the **display()** function because it uses all three book fields, but I wouldn't write the **setPrice()** and **getPrice()** functions because they only access one field.

In order to create multiple objects, you can use this kind of ordinary assignment:

```
cat("Calling 'tome <- book' \n")
tome <- book
cat("Object 'tome' is: \n")
tome$display(tome)
```

Because R assigns by value rather than by reference, object **tome** would be a complete, independent copy of the **book** object. The independence is demonstrated in the demo by these calling statements:

```
cat("Modifying title of object 'tome' \n")
tome$title <- "xxxxxxxx"
cat("Object 'tome' is: \n")
tome$display(tome)

cat("Original object 'book' is unchanged: \n")
book$display(book)
```

The title of object **tome** is changed, but because **tome** is not associated by reference to object **book**, a change to **tome** has no effect on the **book** object.

In summary, the simplest possible way to implement a form of object-oriented programming in R is to create a list object that holds related data and functions. If you want to access data, you can do so directly using the **$** operator or by writing get-functions and set-functions. A call to an encapsulated function that changes the source object will reference the object three times. A call to an encapsulated function that doesn't change the source object will use the object two times. List encapsulation creates a specific instance of an object, but you can create additional objects with the same data fields and function definitions by using ordinary assignment.

Resources

Using list encapsulation OOP is essentially using R lists. For information about the list type, see:
https://stat.ethz.ch/R-manual/R-devel/library/base/html/list.html.

A program-defined function is actually called a closure in R. See:
https://stat.ethz.ch/R-manual/R-devel/library/base/html/function.html.

3.2 OOP using S3

The R language supports an OOP model called S3. Much of the base R language was written using the S3 model. Although S3 has been replaced to some extent by the S4 and RC OOP models, S3 is perfectly viable and is the model of choice for many of my colleagues.

Code Listing 8: S3 Model OOP

```r
# students.R
# R 3.2.4

# S3 OOP
Student = function(ln="NONAME", un=0, gpa=0.00) {
  this <- list(
    lastName = ln,
    units = un,
    gradePoint = gpa
  )
  class(this) <- append(class(this), "Student")
  return(this)
}

display = function(obj) {
  UseMethod("display", obj)
}

display.Student = function(obj) {
  cat("Last name : ", obj$lastName, "\n")
  cat("Units     : ", obj$units, "\n")
  cat("GPA       : ", obj$gradePoint, "\n")
}

setUnits = function(obj, u) {
  UseMethod("setUnits", obj)
}

setUnits.Student = function(obj, un) {
  obj$units <- un
  return(obj)
}

getUnits = function(obj) {
  UseMethod("getUnits", obj)
}

getUnits.Student = function(obj) {
  return(obj$units)
}

# -----
```

```r
cat("\nBegin OOP with S3 demo \n\n")

# initialization
cat("Initializing a default Student object s1 \n")
s1 <- Student()  # default param values
cat("Student 's1' is: \n")
display(s1)
cat("\n")

cat("Initializing a Student object s2 \n")
s2 <- Student("Barker", 27, 2.87)
cat("Student 's2' is: \n")
display(s2)
cat("\n")

# set fields
cat("Setting s1 directly and with a setter() \n")
s1$lastName <- "Archer"
s1 <- setUnits(s1, 19)
s1$gradePoint <- 1.99
cat("Student 's1' is now: \n")
display(s1)
cat("\n")

# get fields
cat("Getting lastName and units for 's1' \n")
ln <- s1$lastName
units <- getUnits(s1)
cat("Units for Student '")
cat(ln)
cat("' are ", units, "\n")
cat("\n")

# assignment
cat("Calling 's3 <- s1' \n")
s3 <- s1
cat("Object 's3' is: \n")
display(s3)
cat("\n")

cat("Modifying all fields of object 's3' \n")
s3$lastName <- "Coogan"
s3 <- setUnits(s3, 38)
s3$gradePoint <- 3.08
cat("Object 's3' is now: \n")
display(s3)
cat("\n")

cat("Original object 's1' is unchanged: \n")
display(s1)

cat("\nEnd demo \n")
```

```
> source("students.R")

Begin OOP with S3 demo

Initializing a default Student object s1
Student 's1' is:
Last name :  NONAME
Units     :  0
GPA       :  0

Initializing a Student object s2
Student 's2' is:
Last name :  Barker
Units     :  27
GPA       :  2.87

Setting s1 directly and with a setter()
Student 's1' is now:
Last name :  Archer
Units     :  19
GPA       :  1.99

Getting lastName and units for 's1'
Units for Student 'Archer' are  19

Calling 's3 <- s1'
Object 's3' is:
Last name :  Archer
Units     :  19
GPA       :  1.99

Modifying all fields of object 's3'
Object 's3' is now:
Last name :  Coogan
Units     :  38
GPA       :  3.08

Original object 's1' is unchanged:
Last name :  Archer
Units     :  19
GPA       :  1.99

End demo
```

The S3 OOP model is significantly different from the models used by other programing languages and even by the other R OOP models. If you examine the code listing, you'll see that S3 is essentially a collection of logically related but physically independent functions.

The demo program defines a Student class. The core definition is a single function named **Student()**, which is defined as:

```
Student = function(ln="NONAME", un=0, gpa=0.00) {
  this <- list(
    lastName = ln,
    units = un,
    gradePoint = gpa
  )
  class(this) <- append(class(this), "Student")
  return(this)
}
```

The **Student()** function contains a list that holds the field variables of **lastName**, **units**, and **gradePoint**. The list is assigned an object with the name **this**, but in S3 you can use any legal name for the field list. Common alternatives to **this** are **self** (as used in Python) and **me** (as used in Visual Basic).

The final two statements in the definition of function **Student()**, which use the keywords **class** and **append**, are, in my opinion, best thought of as an S3 magic incantation.

The demo program creates a first **Student** named **s1** with this code:

```
cat("Initializing a default Student object s1 \n")
s1 <- Student()
cat("Student 's1' is: \n")
display(s1)
```

Creating an S3 object looks exactly like calling an ordinary R function. Because no parameter values are passed to **Student()**, the default parameter values of "NONAME," 0, and 0.0 will be used for **s1** fields **lastName**, **units**, and **gradePoint**.

Notice that the **display()** function is called as though it were a built-in part of the R language, as opposed to a call such as **s1.display()** that you'd see in other programming languages. The S3 model registers functions with the global R runtime environment so that they can be called as if they were built-in functions.

The **display()** function is first registered, then defined, this way:

```
display = function(obj) {
  UseMethod("display", obj)
}
```

```
display.Student = function(obj) {
  cat("Last name : ", obj$lastName, "\n")
  cat("Units     : ", obj$units, "\n")
  cat("GPA       : ", obj$gradePoint, "\n")
}
```

The registration mechanism uses the **UseMethod()** function to tell R that a custom function named **display()** will be defined. The **display()** function is associated with the S3 Student class by using the naming convention of functionName.className. If you've used OOP in other languages, the S3 model may seem a bit odd at first.

The demo program defines a function named **setUnits()** in order to modify the units fields of a **Student** object:

```
setUnits = function(obj, un) {
  UseMethod("setUnits", obj)
}

setUnits.Student = function(obj, un) {
  obj$units <- un
  return(obj)
}
```

Again, the register-define pattern is used. Notice that because function **setUnits()** modifies its source **Student** object, the function must return the object. Function **setUnits()** is called with these statements:

```
cat("Setting s1 directly and with a setter() \n")
s1$lastName <- "Archer"
s1 <- setUnits(s1, 19)
s1$gradePoint <- 1.99
cat("Student 's1' is now: \n")
display(s1)
```

An S3 object can be modified directly using the **$** operator or by using a set-function. Notice that in order to call a function that changes an object, you must use the **object <- function(object, value)** pattern.

S3 objects are copied by value so that if you use assignment, the new object will be an independent copy and any changes made to one of the objects will not affect the other. For example:

```
s3 <- s1  # create a copy of s1
s3$lastName <- "Coogan"  # modify s3
s3 <- setUnits(s3, 38)
s3$gradePoint <- 3.08
display(s1)  # s1 has not been affected
```

In summary, S3 objects are defined as a collection of related functions. Each object has an initialization function containing a list that holds the fields. Auxiliary functions are defined in pairs, with one function to register the auxiliary function and one to define the function's behavior. Functions on S3 objects are called as if they were built-in functions. Functions that change an S3 object must be called using the object <- function(object, value) pattern.

Resources

For detailed information about the **UseMethod()** function, see:
https://stat.ethz.ch/R-manual/R-devel/library/base/html/UseMethod.html.

3.3 OOP using S4

In 2001, the R language introduced an OOP model called S4 that is quite a bit more sophisticated than the earlier S3 model. However, for many programming scenarios, the advanced features available in S4 (such as a form of inheritance and checking for input validity) are optional and not needed.

Code Listing 9: S4 Model OOP

```
# persons.R
# R 3.2.4

# S4 OOP
Person = setClass(
  "Person",

  slots = list(
    lastName = "character",
    age = "integer",
    payRate = "numeric"
  ),

  prototype = list(
    lastName = "NONAME",
    age = as.integer(-1),
    payRate = 0.00
  )

  # could define validity() here
)

setGeneric(name="display",
 def=function(obj) {
  standardGeneric("display")
 }
)

setMethod(f="display",
 signature="Person",
 definition=function(obj) {
    cat("Last Name :", obj@lastName, "\n")
    cat("Age       :", obj@age, "\n")
    cat("Pay Rate  : $", obj@payRate, "\n")
 }
)

setGeneric(name="setPayRate",
 def=function(obj, pRate) {
  standardGeneric("setPayRate")
 }
)

setMethod(f="setPayRate",
```

```r
  signature="Person",
  definition=function(obj, pRate) {
    obj@payRate <- pRate
    return(obj)
  }
)

setGeneric(name="getPayRate",
  def=function(obj) {
    standardGeneric("getPayRate")
  }
)

setMethod(f="getPayRate",
  signature="Person",
  definition=function(obj) {
    return(obj@payRate)
  }
)

# -----

cat("\nBegin OOP with S4 demo \n\n")

# initialization
cat("Initializing a blank Person object p1 \n")
p1 <- new("Person")  # calls prototype()
# p1 <- Person()  # non-preferred
cat("Person 'p1' is: \n")
display(p1)
cat("\n")

cat("Initializing a Person object p2 \n")
p2 <- new("Person", lastName="Baker", age=as.integer(22), payRate=22.22)
cat("Person 'p2' is: \n")
display(p2)
cat("\n")

# set fields
cat("Setting 'p1' directly and with a setter() \n")
p1@lastName <- "Adams"
p1@age <- as.integer(18)
p1 <- setPayRate(p1, 11.11)
cat("Person 'p1' is now: \n")
display(p1)
cat("\n")

# get fields
cat("Getting lastName and payRate of 'p1' \n")
ln <- p1@lastName
pr <- getPayRate(p1)
cat("Pay rate of '")
cat(ln)
cat("' is ", pr, "\n")
```

```
cat("\n")

# assignment
cat("Calling 'p3 <- p1' \n")
p3 <- p1
cat("Object 'p3' is: \n")
display(p3)
cat("\n")

cat("Modifying lastName, payRate of object 'p3' \n")
p3@lastName <- "Chang"
p3 <- setPayRate(p3, 33.33)
cat("Object 'p3' is now: \n")
display(p3)
cat("\n")

cat("Original object 'p1' is unchanged: \n")
display(p1)

cat("\nEnd demo \n")
```

```
> source("persons.R")

Begin OOP with S4 demo

Initializing a blank Person object p1
Person 'p1' is:
Last Name : NONAME
Age       : -1
Pay Rate  : $ 0

Initializing a Person object p2
Person 'p2' is:
Last Name : Baker
Age       : 22
Pay Rate  : $ 22.22

Setting 'p1' directly and with a setter()
Person 'p1' is now:
Last Name : Adams
Age       : 18
Pay Rate  : $ 11.11

Getting lastName and payRate of 'p1'
Pay rate of 'Adams' is  11.11

Calling 'p3 <- p1'
Object 'p3' is:
Last Name : Adams
Age       : 18
Pay Rate  : $ 11.11
```

```
Modifying lastName, payRate of object 'p3'
Object 'p3' is now:
Last Name : Chang
Age       : 18
Pay Rate  : $ 33.33

Original object 'p1' is unchanged:
Last Name : Adams
Age       : 18
Pay Rate  : $ 11.11

End demo
```

The S4 OOP model is significantly different from the models used by other programing languages, but S4 does share some similarities with the S3 model. If you examine the code listing, you'll see that S4, like S3, is essentially a collection of logically related but physically independent functions.

The demo program defines an S4 Person class using the **setClass()** function:

```
Person = setClass(
  "Person",

  slots = list(
    lastName = "character",
    age = "integer",
    payRate = "numeric"
  ),

  prototype = list(
    lastName = "NONAME",
    age = as.integer(-1),
    payRate = 0.00
  )
)
```

The class fields are defined in a list object with the name **slots**. You can specify the type of each field, which will allow R to perform validation checks. Using a **representation** parameter offers an alternative to **slots**, but the R documentation recommends using **slots**.

The optional **prototype** parameter is used to assign default values to an object. For example, the demo program creates a **Person** object named **p1** using the **new()** function this way:

```
cat("Initializing a blank Person object p1 \n")
p1 <- new("Person")
```

Because no values are specified for the **lastName**, **age**, and **payRate** fields, the default values of "NONAME," -1, and 0.00 specified in the prototype will be used. Object **p1** could have been created with the statement **p1 <- Person()**, but I recommend using the **new()** function.

Creating functions that are associated with an S4 class requires a call to **setGeneric()** to register the function name and a call to **setMethod()** to define the associated function:

```
setGeneric(name="setPayRate",
 def=function(obj, pRate) {
   standardGeneric("setPayRate")
 }
)

setMethod(f="setPayRate",
 signature="Person",
 definition=function(obj, pRate) {
    obj@payRate <- pRate
    return(obj)
 }
)
```

This two-step process is a bit cumbersome, and the syntax isn't obvious. Notice that because the **setPayRate()** function modifies the source **Person** object, the function must return a reference to the object. The **setPayRate()** function can be called this way:

```
p1@lastName <- "Adams"
p1@age <- as.integer(18)
p1 <- setPayRate(p1, 11.11)
```

The fields of an S4 object can be accessed directly by using the **@** operator or by defining a set-function. S4 objects are copied by value so that assignment creates an independent copy:

```
p2 <- new("Person", lastName="Baker", age=as.integer(22), payRate=22.22)
p1 <- p2  # any change to p1 or p2 leaves the other unchanged
```

In summary, an S4 class is defined using the **setClass()** function, which can accept a **slots** object that defines class fields and their types, and an optional **prototype** object that defines default field values. An S4 object is instantiated using the **new()** method. You can directly access fields using the **@** operator or by writing get-functions and set-functions. Each function associated with a class is defined by a **setGeneric()** function that registers the associated function and also with a **setMethod()** function that contains the associated function behavior definition.

Resources

For detailed information about creating S4 classes with the **setClass()** function, see:
https://stat.ethz.ch/R-manual/R-devel/library/methods/html/setClass.html.

For additional information about the S4 **prototype()** function for S4 object initialization, see:
https://stat.ethz.ch/R-manual/R-devel/library/methods/html/representation.html.

3.4 OOP using Reference Class

Remember that the R language has an OOP model called Reference Class (RC). RC is quite similar to the OOP models used by languages such as C#, Python, and Java. In my opinion, the RC model is significantly superior to the older S3 and S4 models; in fact, RC is my preferred model for most R programming scenarios when I use OOP.

Code Listing 10: Reference Class (RC) Model OOP

```r
# movies.R
# R 3.2.4

# RC (Reference Class) OOP
Movie <- setRefClass(
  "Movie",

  fields = list(
    title = "character",
    released = "character", # release date
    runTime = "integer"
  ),

  methods = list(
    display = function() {
      cat("Title        :", title, "\n")
      cat("Release date :", released, "\n")
      cat("Run time     :", runTime, "\n")
    },

    getAge = function() {
      rd = as.POSIXlt(released)
      today = as.POSIXlt(Sys.Date())

      age = today$year - rd$year
      if (today$mon < rd$mon ||
       (today$mon == rd$mon && today$mday < rd$mday)) {
        return(age - 1)
      } else {
        return(age)
      }
    },

    setReleased = function(rd) {
      released <<- rd
      # .self$released <- rd
    }
  ) # methods
)

# -----

cat("\nBegin OOP using RC demo \n\n")
```

```r
# initialization
cat("Initializing a blank Movie object using $new() \n")
m1 <- Movie$new()
cat("Movie 'm1' is: \n")
m1$display()
cat("\n")

cat("Initializing a Movie object m2 \n")
m2 <- Movie$new(title="Blade Runner",
 released="1982-06-25", runTime=as.integer(117))
cat("Movie 'm2' is: \n")
m2$display()
cat("\n")

# set fields
cat("Setting 'm1' directly and with a setter() \n")
m1$title <- "Alien"
m1$setReleased("1979-05-25")
m1$runTime <- as.integer(117)
cat("Movie 'm1' is now: \n")
m1$display()
cat("\n")

# get fields
cat("Getting title and age of 'p1' \n")
ti <- m1$title
age <- m1$getAge()
cat("The age of '")
cat(ti)
cat("' is ", age, "years\n")
cat("\n")

# assignment
cat("Calling m3 <- m1$copy() to make a copy \n")
m3 <- m1$copy()
cat("Object 'm3' is: \n")
m3$display()
cat("\n")

cat("Modifying all fields of object 'm3' \n")
m3$title <- "Cube"
m3$setReleased("1997-09-09")
m3$runTime <- as.integer(90)
cat("Object 'm3' is now: \n")
m3$display()
cat("\n")

cat("Original object 'm1' is unchanged: \n")
cat("Object m1 is: \n")
m1$display()
cat("\n")

cat("Calling 'm4 <- m1' (probably wrong!) \n")
m4 <- m1
```

```
cat("Object 'm4' is: \n")
m4$display()
cat("\n")

cat("Modifying all fields of object 'm4' \n")
m4$title <- "Dark City"
m4$setReleased("1998-02-27")
m4$runTime <- as.integer(100)
cat("Object 'm4' is now: \n")
m4$display()
cat("\n")

cat("BUT original object 'm1' has changed too: \n")
cat("Object m1 is: \n")
m1$display()

cat("\nEnd demo \n")
```

```
> source("movies.R")

Begin OOP using RC demo

Initializing a blank Movie object using $new()
Movie 'm1' is:
Title        :
Release date :
Run time     :

Initializing a Movie object m2
Movie 'm2' is:
Title        : Blade Runner
Release date : 1982-06-25
Run time     : 117

Setting 'm1' directly and with a setter()
Movie 'm1' is now:
Title        : Alien
Release date : 1979-05-25
Run time     : 117

Getting title and age of 'p1'
The age of 'Alien' is  36 years

Calling m3 <- m1$copy() to make a copy
Object 'm3' is:
Title        : Alien
Release date : 1979-05-25
Run time     : 117

Modifying all fields of object 'm3'
Object 'm3' is now:
```

```
Title        : Cube
Release date : 1997-09-09
Run time     : 90

Original object 'm1' is unchanged:
Object m1 is:
Title        : Alien
Release date : 1979-05-25
Run time     : 117

Calling 'm4 <- m1' (probably wrong!)
Object 'm4' is:
Title        : Alien
Release date : 1979-05-25
Run time     : 117

Modifying all fields of object 'm4'
Object 'm4' is now:
Title        : Dark City
Release date : 1998-02-27
Run time     : 100

BUT original object 'm1' has changed too:
Object m1 is:
Title        : Dark City
Release date : 1998-02-27
Run time     : 100

End demo
```

The demo program defines an RC Movie class using the **setRefClass()** function. The structure of the call to **setRefClass()** is:

```
Movie <- setRefClass(
  "Movie",

  fields = list(
    # fields go here
  ),

  methods = list(
    # class functions go here
  )
)
```

Unlike S3 and S4 classes, RC class functions (technically called methods) are physically defined inside the class definition. The Movie class fields are:

```
fields = list(
  title = "character",
  released = "character", # release date
  runTime = "integer"
),
```

The data types of the fields in an RC class are specified, and this allows R to perform runtime error checking.

RC class functions/methods

The Movie class has three associated functions/methods. The first is a display function defined as:

```
display = function() {
  cat("Title        :", title, "\n")
  cat("Release date :", released, "\n")
  cat("Run time     :", runTime, "\n")
},
```

The class fields **title**, **released**, and **runTime** can be accessed directly by the **display()** function. And you can make the field membership explicit by using the special **.self** variable with the **$** operator. For example:

```
cat("Title        :", .self$title, "\n")
```

The Movie class has a **getAge()** function that returns the age, in years, of a **Movie** object by calculating the difference between the current date and the release date:

```
getAge = function() {
  rd = as.POSIXlt(released)
  today = as.POSIXlt(Sys.Date())

  age = today$year - rd$year
  if (today$mon < rd$mon || (today$mon == rd$mon && today$mday < rd$mday)) {
    return(age - 1)
  } else {
    return(age)
  }
},
```

The **getAge()** function uses the built-in **as.POSIXlt()** function to convert the **released** field and the system date-time to a structure that has **year**, **mon**, and **mday** fields. The **year** fields are subtracted in order to get the age of the **Movie** object, although the age must be decremented by one year if the month and day of the release date are less than those of the system date (because the movie's "birthday" hasn't happened yet).

The third Movie class function is one that allows you to set the release date:

```
setReleased = function(rd) {
  released <<- rd
}
```

In order to modify an RC class field via a member function, you must use the special **<<-** operator rather than the regular **<-** or = assignment operators. This is very important—forgetting to use **<<-** is a common source of RC class implementation errors.

RC object instantiation

The demo program creates a **Movie** object using these statements:

```
cat("Initializing a blank Movie object using $new() \n")
m1 <- Movie$new()
cat("Movie 'm1' is: \n")
m1$display()
```

The pattern to create an RC object is **object <- className$new()**. Here, because no values are passed to **new()**, object **m1** has no values for the **title**, **released**, and **runTime** fields. These values must be supplied later, either directly or by using a set-function, as I'll demonstrate shortly.

Calling a member function/method of an RC object uses the pattern **object$function()**. This pattern is significantly different from the S3 and S4 calling patterns. For example, if a Movie class is defined using the S3 or S4 model, a display function will be called as **display(m1)**.

The demo creates a second **Movie** object with field values:

```
cat("Initializing a Movie object m2 \n")
m2 <- Movie$new(title="Blade Runner",  released="1982-06-25",
runTime=as.integer(117))
cat("Movie 'm2' is: \n")
m2$display()
```

When instantiating an RC object, if you have not defined the optional **initialize()** function, you cannot pass unnamed arguments to **new()**. For example:

```
m2 <- Movie$new("Blade Runner", "1982-06-25", as.integer(117))  # wrong
```

This code will run, but it will generate a waning message that "unnamed arguments to $new() must be objects from a reference class," and the class fields will not get values.

The demo program assigns values to the instantiated but uninitialized **m1** object this way:

```
cat("Setting 'm1' directly and with a setter() \n")
m1$title <- "Alien"
m1$setReleased("1979-05-25")
m1$runTime <- as.integer(117)
```

All RC object fields have public visibility, which means fields can be accessed directly using the **$** operator. Some programming languages, such as Java and C#, allow you to define private fields and functions, but other languages, including R and Python, do not. A recently released add-on package named R6 allows you to define R classes that can have private fields and functions.

RC object assignment

Unlike all other R objects and variables, RC objects are copied by reference rather than by value. This means you must be careful when using assignment with RC objects—you'll typically want to use the built-in **copy()** function. The idea is best explained by example. Suppose the **Movie** object **m1** has been instantiated and initialized. This code is probably not what you want:

```
m4 <- m1  # probably not what you intend
m4$title <- "Dark City"
m4$setReleased("1998-02-27")
m4$runTime <- as.integer(100)
# all fields of object m1 have also been changed too!
```

All RC objects inherit a built-in **copy()** function that will make a value copy instead of a reference copy:

```
m4 <- m1$copy()  # probably what you want to do
m4$title <- "Dark City"
m4$setReleased("1998-02-27")
m4$runTime <- as.integer(100)
# m1 is not affected
```

In summary, the Reference Class OOP model has some similarities to the OOP models used by languages such as C#, Java, and Python. You define an RC class using the **setRefClass()** function. All fields and class functions/methods are encapsulated in the class definition. This means you call class functions using the **object$function()** pattern rather than the **function(object)** pattern used by S3 and S4 objects. When defining a class function that modifies a field, you must use the special **<<-** assignment operator rather than the **<-** operator. All RC object fields have public visibility and can be accessed using the **$** operator. RC objects are copied by reference rather than by value, so that in most cases you'll want to use the built-in **copy()** function when performing object assignment.
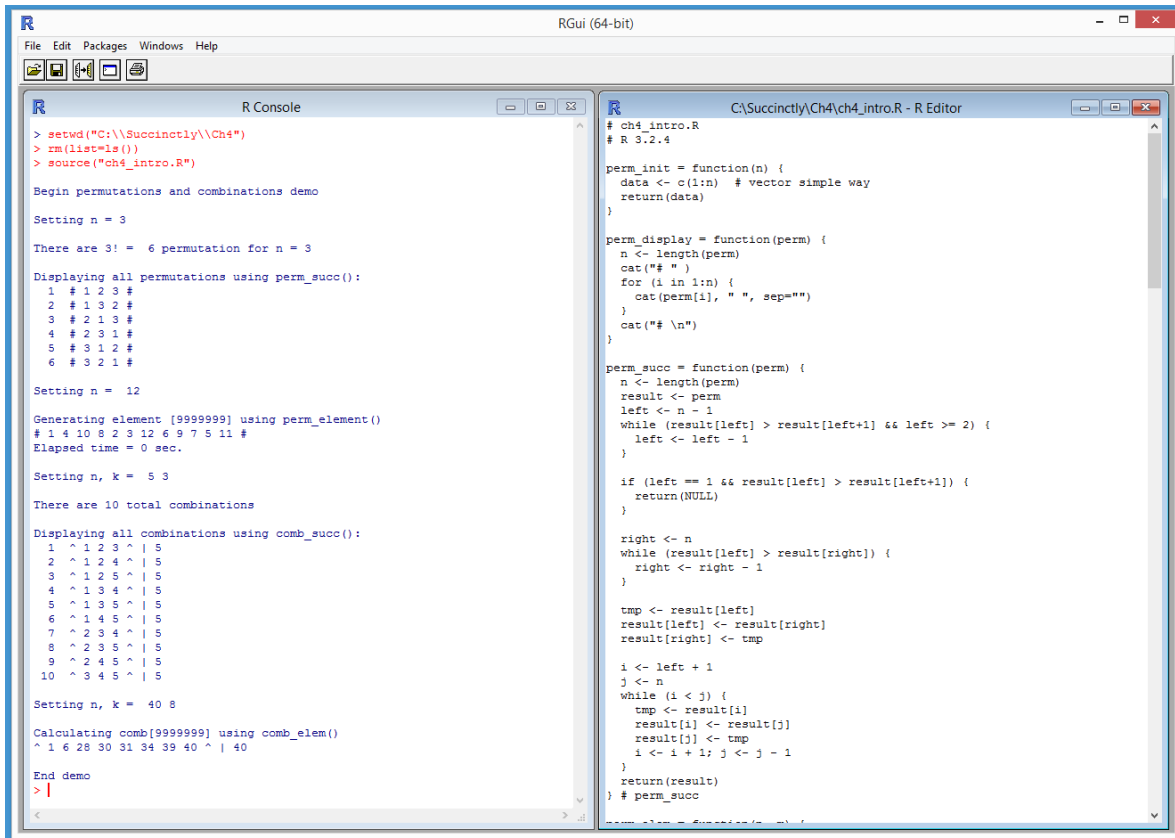
Resources

For detailed information about RC classes, including class inheritance, see:
https://stat.ethz.ch/R-manual/R-devel/library/methods/html/refClass.html.

For additional information about the many options available when creating an RC class, see:
https://stat.ethz.ch/R-manual/R-devel/library/methods/html/setClass.html.

Chapter 4 Permutations and Combinations

A mathematical permutation is a rearrangement of items. A mathematical combination is a subset of items. Together, combinations and permutations are sometimes called combinatorics. The R language has very limited built-in support for permutations and combinations, but it is possible to write your own program-defined functions. Figure 21 gives you an idea of where this chapter is headed.

Figure 21: Permutations and Combinations Demo

In section 4.1, we'll look at how to create a permutation library using only a list object and related functions such as **perm_init()** and **perm_display()**. In section 4.2, we will examine how to generate a specified permutation element directly (rather than by iterating) by writing a **perm_elem()** function that uses a clever idea called the factoradic of a number.

In section 4.3, you'll learn how to create a combination library using only a list object and related functions such as **comb_init()** and **comb_display()**. In section 4.4, we'll see how to generate a specified combination element directly (rather than by iterating) by writing a **comb_elem()** function that uses a clever idea called the combinadic of a number.

4.1 Permutations

Although permutations are used in many areas of statistics and machine learning, R has very limited built-in support for permutations. You can write program-defined functions to create, display, and manipulate permutations.

Code Listing 11: Permutations

```r
# permutations.R
# R 3.2.4

perm_init = function(n) {
  data <- c(1:n)  # vector simple way
  return(data)
}

perm_display = function(perm) {
  n <- length(perm)
  cat("# " )
  for (i in 1:n) {
    cat(perm[i], " ", sep="")
  }
  cat("# \n")
}

perm_succ = function(perm) {
  n <- length(perm)
  result <- perm
  left = n - 1
  while (result[left] > result[left+1] && left >= 2) {
    left <- left - 1
  }

  if (left == 1 && result[left] > result[left+1]) {
    return(NULL)
  }

  right <- n
  while (result[left] > result[right]) {
    right <- right - 1
  }

  tmp <- result[left]
  result[left] <- result[right]
  result[right] <- tmp

  i <- left + 1
  j <- n
  while (i < j) {
    tmp <- result[i]
    result[i] <- result[j]
    result[j] <- tmp
    i <- i + 1; j <- j - 1
```

```r
  }
  return(result)
} # perm_succ

# =====

cat("\nBegin permutations demo \n\n")

n <- as.integer(4)
cat("Setting n =", n, "\n\n")

cat("Displaying all permutations using perm_succ(): \n")
p <- perm_init(n)
i <- as.integer(1)
while (is.null(p) == FALSE) {
  cat(formatC(i, digits=2), " ")
  perm_display(p)
  p <- perm_succ(p)
  i <- i + 1
}
cat("\n")

cat("End permutations demo \n")
```

```
> source("permutations.R")

Begin permutations demo

Setting n = 4

Displaying all permutations using perm_succ():
   1  # 1 2 3 4 #
   2  # 1 2 4 3 #
   3  # 1 3 2 4 #
   4  # 1 3 4 2 #
   5  # 1 4 2 3 #
   6  # 1 4 3 2 #
   7  # 2 1 3 4 #
   8  # 2 1 4 3 #
   9  # 2 3 1 4 #
  10  # 2 3 4 1 #
  11  # 2 4 1 3 #
  12  # 2 4 3 1 #
  13  # 3 1 2 4 #
  14  # 3 1 4 2 #
  15  # 3 2 1 4 #
  16  # 3 2 4 1 #
  17  # 3 4 1 2 #
  18  # 3 4 2 1 #
  19  # 4 1 2 3 #
  20  # 4 1 3 2 #
```

```
 21  # 4 2 1 3 #
 22  # 4 2 3 1 #
 23  # 4 3 1 2 #
 24  # 4 3 2 1 #

End permutations demo
```

A mathematical permutation of order n is one possible arrangement of the numbers 1 through n. For example, one of the permutations of order n = 6 is (3, 1, 6, 5, 4, 2). When listed in lexicographical order, all 6 permutations of order n = 3 are:

1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1

There are several ways to design a custom library for permutations. The demo program defines a permutation as a vector of integers with three related functions: **perm_init()**, **perm_display()**, and **perm_succ()**. You should also consider using alternative, more sophisticated designs such as list encapsulation or an S3, S4, or RC class.

The **perm_init()** function is defined as:

```
perm_init = function(n) {
  data = c(1:n)
  return(data)
}
```

The demo program calls **perm_init()** this way:

```
n <- 4
p <- perm_init(n)
```

The **perm_display()** function prints a permutation vector with leading and trailing # characters and with values separated by blank spaces:

```
perm_display = function(perm) {
  n <- length(perm)
  cat("# " )
  for (i in 1:n) {
    cat(perm[i], " ", sep="")
  }
  cat("# \n")
}
```

Notice that the order of a permutation can be determined by the length of the underlying vector.

When working with permutations, you will often want to generate all the permutations or the successor to a given permutation. The demo program defines a **perm_succ()** function to do this. The function can be called this way in order to generate all permutations for an order n:

```
p <- perm_init(n)
while (is.null(p) == FALSE) {
  perm_display(p)
  p <- perm_succ(p)
}
```

The stopping condition assumes that **perm_succ()** has been defined so that the successor to the last permutation element is NULL. Notice that you must use the built-in **is.null()** function rather than the == operator to check if an object is NULL.

Before iterating through all permutations of order n, you should consider determining how many permutation elements there are by using the built-in **factorial()** function. For example:

```
numPerms <- factorial(n)
cat("Total number of permutations is", numPerms, "\n")
```

In most cases, it's not practical to iterate through all permutations when n is larger than about 9 because 9! = 362,880, but 10! = 3,628,800 and 15! = 1,307,674,368,000.

The integer values in a permutation are often used to map to vector or list indices. For example, you could write a function that applies a permutation to a list this way:

```
perm_applyTo = function(perm, lst) {
  result <- list()
  n <- length(lst)
  for (i in 1:n) {
    result[i] <- lst[[perm[i]]]
  }
  return(result)
}
```

In summary, permutations are rearrangements of the numbers 1 through n. Essentially, the only built-in support for permutations in R are the **sample()** and **factorial()** functions. It's relatively easy to implement your own permutation library in which a permutation is defined as a vector of integers plus related functions in order to initialize, display, and return a successor.

Resources

For information about **factorial()** and other specialized math functions, see:
https://stat.ethz.ch/R-manual/R-devel/library/base/html/Special.html.

For information about the **sample()** function, see:
https://stat.ethz.ch/R-manual/R-devel/library/base/html/sample.html.

4.2 Permutation element

A useful function when working with permutations is one that returns the mth lexicographical element. For example, if n = 4, there are 4! = 24 total elements. A call to **perm_element(1)** would return (1, 2, 3, 4), and a call to **perm_element(23)** would return (4, 3, 1, 2).

Code Listing 12: Direct Generation of a Permutation Element

```r
# permelements.R
# R 3.2.4

perm_init = function(n) {
  data <- c(1:n)
  return(data)
}

perm_display = function(perm) {
  n <- length(perm)
  cat("# " )
  for (i in 1:n) {
    cat(perm[i], " ", sep="")
  }
  cat("# \n")
}

perm_succ = function(perm) {
  n <- length(perm)
  result <- perm
  left <- n - 1
  while (result[left] > result[left+1] && left >= 2) {
    left <- left - 1
  }

  if (left == 1 && result[left] > result[left+1]) {
    return(NULL)
  }

  right <- n
  while (result[left] > result[right]) {
    right <- right - 1
  }

  tmp <- result[left]
  result[left] <- result[right]
  result[right] <- tmp

  i <- left + 1
  j <- n
  while (i < j) {
    tmp <- result[i]
    result[i] <- result[j]
    result[j] <- tmp
    i <- i + 1; j <- j - 1
```

```r
  }
  return(result)
} # perm_succ

perm_elem = function(n, m) {
  # mth element of perm order n
  m <- m - 1 # make m 0-based
  result <- c(1:n)
  factoradic <- c(1:n)

  for (j in 1:n) {
    factoradic[n-j+1] <- m %%j
    m <- m %/% j
  }

  for (i in 1:n) {
    factoradic[i] <- factoradic[i] + 1
  }

  result[n] <- 1 # last value to 1

  i <- n-1
  while(i >= 1) {
    result[i] <- factoradic[i]
    for (j in (i+1):n) {
      if (result[j] >= result[i]) {
        result[j] <- result[j] + 1
      }
    }
    i <- i - 1
  }

  return(result)
} # perm_element


# =====

cat("\nBegin permutation element demo \n\n")

n <- as.integer(12)
cat("Setting n = ", n, "\n\n")
cat("Generating element [9999999] using perm_element() \n")
start_t <- proc.time()
pe <- perm_elem(n, 9999999)
end_t <- proc.time()
times <- end_t - start_t
perm_display(pe)
cat("Elapsed time =", times[3], "sec.\n")
cat("\n\n")

cat("Generating element [9999999] using perm_succ() \n")
start_t <- proc.time()
p <- perm_init(n)
```

```
m <- 9999999
for (j in 1:(m-1)) {
  p = perm_succ(p)
}
end_t <- proc.time()
times <- end_t - start_t
perm_display(p)
cat("Elapsed time =", times[3], "sec.\n")
cat("\n")

cat("End permutations demo \n")
```

```
> source("permelements.R")

Begin permutation element demo

Setting n =  12

Generating element [9999999] using perm_element()
# 1 4 10 8 2 3 12 6 9 7 5 11 #
Elapsed time = 0 sec.


Generating element [9999999] using perm_succ()
# 1 4 10 8 2 3 12 6 9 7 5 11 #
Elapsed time = 98.33 sec.

End permutation element demo
```

There are 12! = 479,001,600 permutation elements for order n = 12. The demo program computes element 9,999,999 in two different ways. A naive approach for generating the mth element of a permutation starts with the first element, then calls a successor function m-1 times. However, this approach is only feasible for relatively small values of m. The demo program requires about 98 seconds using this iterative approach:

```
n <- as.integer(12)
p <- perm_init(n)
m <- 9999999
for (j in 1:(m-1)) {
  p = perm_succ(p)
}
perm_display(p)
```

We can also calculate the mth element of a permutation set directly by using the factoradic of a number. The factoradic of a number is a representation based on factorials. For example, in ordinary base 10 representation, the number 794 is (7 * 100) + (9 * 10) + (4 * 1).

Because 6! = 720, 5! = 120, 4! = 24, 3! = 6, 2! = 2, and 1! = 1, the number 794 can also be written as (103020) because 794 = (1 * 6!) + (0 * 5!) + (3 * 4!) + (0 * 3!) + (2 * 2!) + (0 * 1!). Although it's not immediately obvious, you can calculate the mth permutation element by calculating the factoradic of m, which then maps to the element.

In function **perm_element()**, here is the code that computes the factoradic of **m**:

```
m <- m - 1
factoradic <- c(1:n)
for (j in 1:n) {
  factoradic[n-j+1] <- m %%j
  m <- m %/% j
}

for (i in 1:n) {
  factoradic[i] <- factoradic[i] + 1
}
```

First, m is converted from the usual R 1-based indexing to 0-based indexing. The **%%** operator is integer modulus. For example, 13 %% 5 = 3 because 5 goes into 13 two times with 3 left. The **%/%** operator is integer division. For example, 7 / 2 = 3.5, but 7 %/% 2 = 3.

After the factoradic of **m** has been calculated, it is used to generate the result permutation element this way:

```
result <- c(1:n)
result[n] <- 1
i <- n-1
while(i >= 1) {
  result[i] <- factoradic[i]
  for (j in (i+1):n) {
    if (result[j] >= result[i]) {
      result[j] <- result[j] + 1
    }
  }
  i <- i - 1
}
```

In summary, if you want to generate the mth element of a permutation set, you can iterate using a successor function if m is small. For large values of m, you can write a function that calculates the mth element directly by using the factoradic of m.

Resources

For additional information about the factoradic of a number, see:
https://en.wikipedia.org/wiki/Factorial_number_system.

4.3 Combinations

Although mathematical combinations are used in many areas of statistics and machine learning, R has very limited built-in support for combinations. You can write functions to create, display, and manipulate combinations.

Code Listing 13: Combinations

```r
# combinations.R
# R 3.2.4

comb_init = function(n, k) {
  data <- c(1:(k+1))
  data[k+1] <- n  # store n in dummy last cell
  return(data)
}

comb_display = function(comb) {
  len <- length(comb)
  k <- len - 1
  n <- comb[len]

  cat("^ " )
  for (i in 1:k) {
    cat(comb[i], " ", sep="")
  }
  cat("^ |", n, "\n")
}

comb_succ = function(comb) {
  len <- length(comb)
  k <- len - 1
  n <- comb[len]

  if (comb[1] == n - k + 1) {
    return(NULL)
  }

  result <- comb

  i <- k
  while (i > 1 && (result[i] == (n-k+i))) {
    i <- i - 1
  }

  result[i] = result[i] + 1
  j <- i + 1
  while (j <= k) {
    result[j] <- result[j-1] + 1
    j <- j + 1
  }

  return(result)
```

```
} # comb_succ

# -----

cat("\nBegin combinations demo \n\n")

n <- as.integer(6)
k <- as.integer(4)
cat("Setting n, k = ", n, k, "\n\n")
nc <- choose(n, k)
cat("There are", nc, "total combinations \n\n")

cat("Displaying all combinations: \n")
cmb <- comb_init(n, k)
i <- as.integer(1)
while (is.null(cmb) == FALSE) {
  cat(formatC(i, digits=2), " ")
  comb_display(cmb)
  cmb <- comb_succ(cmb)
  i <- i + 1
}
cat("\n")

cat("End combinations demo \n")
```

```
> source("combinations.R")

Begin combinations demo

Setting n, k =  6 4

There are 15 total combinations

Displaying all combinations:
  1  ^ 1 2 3 4 ^ | 6
  2  ^ 1 2 3 5 ^ | 6
  3  ^ 1 2 3 6 ^ | 6
  4  ^ 1 2 4 5 ^ | 6
  5  ^ 1 2 4 6 ^ | 6
  6  ^ 1 2 5 6 ^ | 6
  7  ^ 1 3 4 5 ^ | 6
  8  ^ 1 3 4 6 ^ | 6
  9  ^ 1 3 5 6 ^ | 6
 10  ^ 1 4 5 6 ^ | 6
 11  ^ 2 3 4 5 ^ | 6
 12  ^ 2 3 4 6 ^ | 6
 13  ^ 2 3 5 6 ^ | 6
 14  ^ 2 4 5 6 ^ | 6
 15  ^ 3 4 5 6 ^ | 6

End combinations demo
```

A mathematical combination of order (n, k) is one possible subset of k numbers from the numbers 1 through n, where order doesn't matter. For example, one of the combinations of order (n = 7, k = 4) is (2, 3, 5, 7). The set (3, 2, 7, 5) is considered the same as (2, 3, 5, 7) because the order of the values doesn't matter.

Listed in lexicographical order, here are all 10 combinations of order (n = 5, k = 3):

```
1 2 3
1 2 4
1 2 5
1 3 4
1 3 5
1 4 5
2 3 4
2 3 5
2 4 5
3 4 5
```

Notice that if you see one combination element, you can infer k from the number of values in the element, but you can't infer n.

A custom library for combinations can be designed several ways. The demo program defines a combination of order (n, k) as a vector of k integers with an additional cell that holds the value of n plus three related functions: **comb_init()**, **comb_display()**, and **comb_succ()**. You should also consider using list encapsulation or using an S3, S4, or RC class.

The **comb_init()** function is defined as:

```
comb_init = function(n, k) {
  data <- c(1:(k+1))
  data[k+1] <- n  # store n in dummy last cell
  return(data)
}
```

Function **comb_init()** can be called this way:

```
n <- as.integer(6)
k <- as.integer(4)
cmb <- comb_init(n, k)
```

These statements would create an integer vector named **cmb** that has five cells. The first four cells would hold (1, 2, 3, 4) and the fifth cell would hold 6, the value of n.

The **comb_display()** function is defined as:

```
comb_display = function(comb) {
  len <- length(comb)
  k <- len - 1
  n <- comb[len]

  cat("^ " )
  for (i in 1:k) {
    cat(comb[i], " ", sep="")
  }

  cat("^ |", n, "\n")
}
```

The **length()** function directly yields the value of k (as length - 1) and indirectly yields the value of k (the value stored in cell [length]). The **comb_display()** function prints a combination element with leading and trailing ^ characters and with the value of n separated by a | character. You have many formatting alternatives available.

When working with combinations, you'll often want to generate all combinations or the successor to a given combination. The demo program defines a **comb_succ()** function to do this. The function can be called this way in order to generate all combinations of order (n, k):

```
cmb <- comb_init(n, k)
while (is.null(cmb) == FALSE) {
  comb_display(cmb)
  cmb <- cmb_succ(cmb)
}
```

The stopping condition assumes that **comb_succ()** has been defined so that the successor to the last combination element is NULL. Before iterating through all combinations of order (n, k), you should consider determining how many elements are using the built-in **choose()** function. For example:

```
numCombs <- choose(n, k)
```

In summary, combinations are subsets of the numbers 1 through n. The **choose()** function and the **sample()** function are essentially the only built-in support for combinations in R. It's relatively easy to implement your own combination library in which a combination is defined as a vector of integers plus related functions that initialize, display, and return a successor.

Resources

For information about **choose()** and other specialized math functions, see:
https://stat.ethz.ch/R-manual/R-devel/library/base/html/Special.html.

For information about the **sample()** function, see:
https://stat.ethz.ch/R-manual/R-devel/library/base/html/sample.html.

4.4 Combination element

When working with combinations, you'll find that useful functions are those that return the mth lexicographical element. For example, if n = 10 and k = 3, there are choose(10, 3) = 120 total elements. A call to **comb_element(1)** would return (1, 2, 3), and a call to **comb_element(120)** would return (8, 9, 10).

Code Listing 14: Direct Generation of a Combination Element

```
# combelements.R
# R 3.2.4

comb_init = function(n, k) {
  data <- c(1:(k+1))
  data[k+1] <- n
  return(data)
}

comb_display = function(comb) {
  len <- length(comb)
  k <- len - 1
  n <- comb[len]

  cat("^ " )
  for (i in 1:k) {
    cat(comb[i], " ", sep="")
  }
  cat("^ |", n, "\n")
}

comb_succ = function(comb) {
  len <- length(comb)
  k <- len - 1
  n <- comb[len]

  if (comb[1] == n - k + 1) {
    return(NULL)
  }

  result <- comb

  i <- k
  while (i > 1 && (result[i] == (n-k+i))) {
    i <- i - 1
  }

  result[i] = result[i] + 1
  j <- i + 1
  while (j <= k) {
    result[j] <- result[j-1] + 1
    j <- j + 1
  }
```

```r
    return(result)
}

comb_elem = function(n, k, m) {
  # mth element, combinadic

  m <- m - 1  # zero-based m
  maxM <- choose(n, k) - 1  # largest z-index
  ans <- c(1:(k+1))  # extra cell for [0]

  a <- n  # look for a v less than this
  b <- k
  x <- maxM - m  # x is the dual of m

  for (i in 1:k) {
    v <- a - 1
    while (choose(v, b) > x) {
      v <- v - 1
    }

    ans[i] <- v
    x <- x - choose(v, b)
    a <- v
    b <- b - 1
  }

  for (i in 1:k) {
    ans[i] <- n - ans[i]  # (+1) to [1]-based
  }
  ans[k+1] <- n  # recall n goes in last cell

  return(ans)
}

# -----

cat("\nBegin combination element demo \n\n")

n <- as.integer(40)
k <- as.integer(8)
cat("Setting n, k = ", n, k, "\n\n")

cat("Calculating comb[9999999] using comb_elem() \n")
m <- as.integer(9999999)

cmb <- comb_elem(n, k, m)
comb_display(cmb)
cat("\n\n")


cat("Calculating comb[9999999] using comb_succ() \n")
cmb <- comb_init(n, k)
for (i in 1:(m-1)) {
  cmb <- comb_succ(cmb)
```

```
}
comb_display(cmb)

cat("\nEnd combination element demo \n")
```

```
> source("combelements.R")

Begin combination element demo

Setting n, k =  40 8

Calculating comb[9999999] using comb_elem()
^ 1 6 28 30 31 34 39 40 ^ | 40


Calculating comb[9999999] using comb_succ()
^ 1 6 28 30 31 34 39 40 ^ | 40

End combination element demo
```

For order n = 40 and k = 8, there are choose(40, 8) = 76,904,685 different combination elements. The demo program computes element 9,999,999 in two different ways. Starting with the first element, then calling a successor function m-1 times, is a naive approach for generating the mth element of a combination. However, this approach is only feasible for relatively small values of m. The demo program requires about 25 seconds using this iterative approach:

```
n <- as.integer(40)
k <- as.integer(8)
cmb <- cmb_init(n, k)
m <- 9999999
for (j in 1:(m-1)) {
  cmb = cmb_succ(cmb)
}
cmb_display(cmb)
```

You can also calculate the mth element of a combination set directly using what is called the combinadic of a number. The combinadic of a number is a representation based on the choose() function. For example, in ordinary base 10 representation, the number 27 is (2 * 10) + (7 * 1).

Suppose n = 7 and k = 4. The number 27 can also be written as (6, 5, 2, 1) because:

27 = choose(6, 4) + choose(5, 3) + choose(2, 2) + choose(1, 1) = 15 + 10 + 1 + 1

Although it's not immediately obvious, you can calculate the mth combination element by calculating the combinadic of m, which then maps to the associated combination element.

In function **comb_element()**, the code that computes the combinadic of m begins with:

```
m <- m - 1
maxM <- choose(n, k) - 1
ans <- c(1:(k+1))
a <- n
b <- k
x <- maxM - m  # x is the dual of m
```

The input argument m is converted to 0-based indexing. Rather than computing the combinadic of m, the algorithm actually computes the combinadic of x, the 0-based dual of m. But the demo implementation of **comb_elem()** has a limitation—for large values of n and k, the built-in **choose()** function might overflow. A more robust implementation would use a BigInteger version of a choose function such as the one in the gmp add-on package.

Function **comb_elem()** computes a preliminary result into vector **ans** with this code:

```
for (i in 1:k) {
  v <- a - 1
  while (choose(v, b) > x) {
    v <- v - 1
  }
  ans[i] <- v
  x <- x - choose(v, b)
  a <- v
  b <- b - 1
}
```

When this code finishes execution, the desired combination element values are 0-based and stored as complements to n, which means they are converted to 1-based values and de-complemented. Also, the value of n is placed into the last cell of **ans**:

```
for (i in 1:k) {
  ans[i] <- n - ans[i]
}
ans[k+1] <- n
```

In summary, if you want to generate the mth element of a combination of order (n, k), and if m is small, you can iterate using a successor function. For large values of m, you can write a function that calculates the mth element directly by using the combinadic of m.
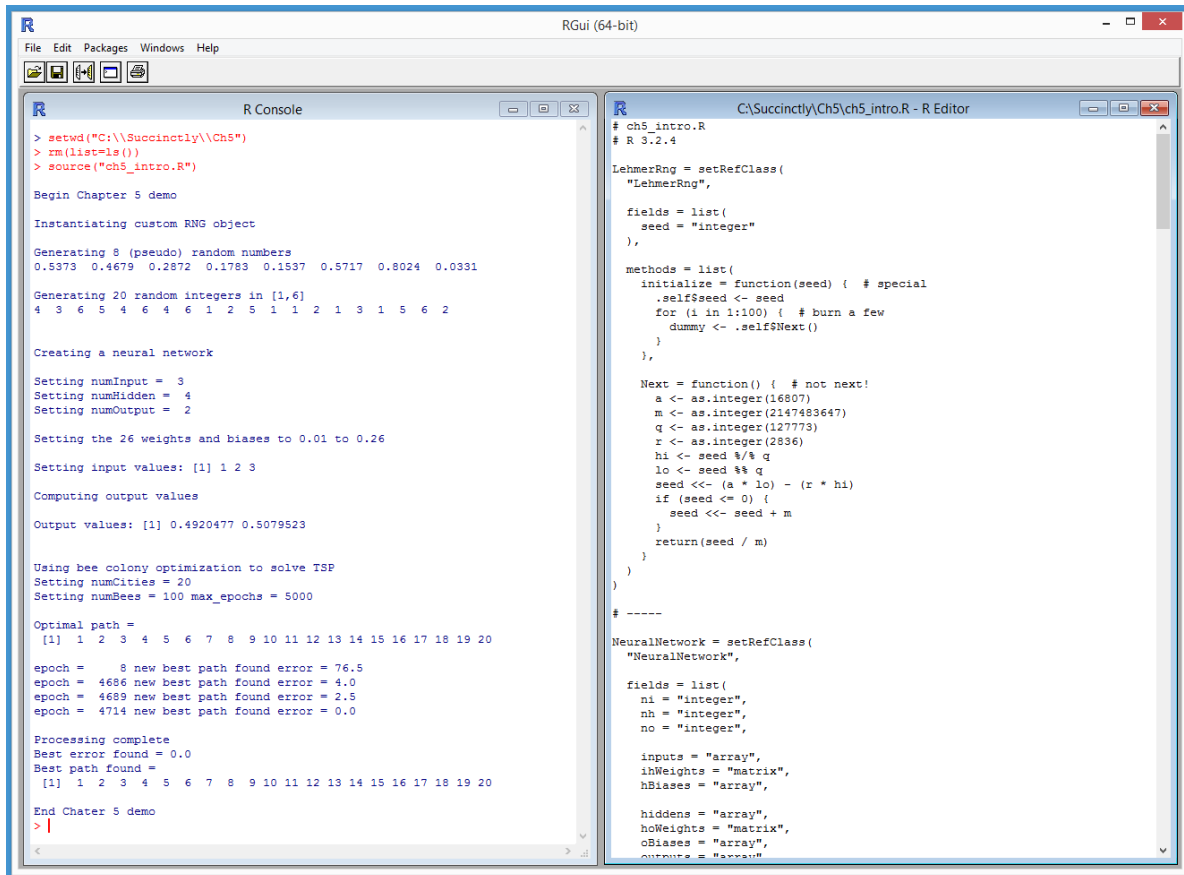
Resources

For additional information about the combinadic of a number, see:
https://en.wikipedia.org/wiki/Combinatorial_number_system.

For information about an add-on R package named combinat, see:
https://cran.r-project.org/web/packages/combinat/combinat.pdf.

Chapter 5 Advanced R Programming

The R programming language contains all the features needed to create very sophisticated programs. This chapter presents three topics (random number generation, neural networks, and combinatorial optimization) that illustrate the power of R, demonstrate useful R programming techniques, and give you starting points for a personal code library. Figure 22 gives you an idea of where this chapter is headed.

Figure 22: Advanced R Programming Topics

In section 5.1, you'll learn how to create a program-defined random number generator (RNG) object using an RC model and the Lehmer random number generation algorithm.

In section 5.2, we'll look at how to design and create a neural network and how to write code that implements the neural network feed-forward, input-process-output mechanism.

In section 5.3, you'll learn how to solve combinatorial optimization problems using a bio-inspired algorithm that models the behavior of honey bees. In particular, you'll come to understand how to solve the traveling salesman problem, arguably the most well-known combinatorial-optimization problem.

5.1 Program-defined RNGs

Most programming languages allow you to create multiple instances of random number generators. The R language uses a single, global-system RNG that is somewhat limiting. However, you can write your own RNG class in R.

Code Listing 15: A Random Number Generator Class

```r
# randoms.R
# R 3.2.4

LehmerRng = setRefClass(
  "LehmerRng",

  fields = list(
    seed = "integer"
  ),

  methods = list(
    initialize = function(seed) {  # special
      .self$seed <- seed
      for (i in 1:100) {  # burn a few
        dummy <- .self$Next()
      }
    },

    Next = function() {  # not next!
      a <- as.integer(16807)
      m <- as.integer(2147483647)
      q <- as.integer(127773)
      r <- as.integer(2836)
      hi <- seed %/% q
      lo <- seed %% q
      seed <<- (a * lo) - (r * hi)
      if (seed <= 0) {
        seed <<- seed + m
      }
      return(seed / m)
    }
  )
)

# -----

cat("\nBegin custom RNG demo \n\n")

cat("Instantiating custom RNG object \n\n")
my_rng <- LehmerRng$new(as.integer(1))

cat("Generating 8 (pseudo) random numbers \n")
for (i in 1:8) {
  x <- my_rng$Next()
  xx <- formatC(x, digits=4, format="f")
```

```
   cat(xx, " ")
}
cat("\n\n")

cat("Generating 20 random integers in [1,6] \n")
lo <- as.integer(1)
hi <- as.integer(7)
for (i in 1:20) {
  n <- floor((hi - lo) * my_rng$Next() + lo)
  cat(n, " ")
}
cat("\n\n")

cat("End custom RNG demo \n")
```

```
> source("randoms.R")

Begin custom RNG demo

Instantiating custom RNG object

Generating 8 (pseudo) random numbers
0.5373  0.4679  0.2872  0.1783  0.1537  0.5717  0.8024  0.0331

Generating 20 random integers in [1,6]
4  3  6  5  4  6  4  6  1  2  5  1  1  2  1  3  1  5  6  2

End custom RNG demo
```

Here is an example of creating RNGs with other programming languages:

```
Random my_rng = new Random(0);  // C# and Java
my_rnd = random.Random(0)  # Python
```

There are many different RNG algorithms. Among the most common are the linear congruential algorithm, the Lehmer algorithm, the Wichmann-Hill algorithm, and the Lagged Fibonacci algorithm. The demo program uses the Lehmer algorithm and creates a Reference Class definition named **LehmerRng**. Here is the structure of the class:

```
LehmerRng = setRefClass(
  "LehmerRng",
  fields = list( . . . ),
  methods = list( . . . )
)
```

The Lehmer algorithm is $X(i+1) = a * X(i)$ mod m. In English, that is "the next random value is some number *a* times the current value modulo some number *m*." For example, suppose the current random value is 104, where a = 3 and m = 100. The next random value would be 3 * 104 mod 100 = 312 mod 100 = 12. It's standard practice for RNGs to return a floating-point value between 0.0 and 1.0, so the next-function of the RNG would return 12 / 100 = 0.12.

The next random value would be 3 * 12 mod 100 = 36, and the next-function would emit 0.36. The X(i) in most RNG algorithms is named "seed." The **fields** part of the **LehmerRng** class definition is simply:

```
fields = list(
  seed = "integer"
),
```

The custom RNG contains two methods. The **Next()** function implements the Lehmer algorithm but uses an algebra trick to prevent arithmetic overflow:

```
Next = function() {
  a <- as.integer(16807)
  m <- as.integer(2147483647)
  q <- as.integer(127773); r <- as.integer(2836)
  hi <- seed %/% q; lo <- seed %% q
  seed <<- (a * lo) - (r * hi)
  if (seed <= 0) { seed <<- seed + m }
  return(seed / m)
}
```

Integer values *a* and *m* are the multiplier and modulo constants. Their values come from math theory, and you should not change them. The q value is (m / a) and the r value is (m mod a). By using q and r, the algorithm prevents arithmetic overflow during the a * X(i) multiplication part of the algorithm. Local variables a, m, q, and r could have been defined as class fields.

I use **Next()** upper case rather than **next()** lower case because **next** is a reserved word in R (used to short-circuit to the next iteration of a for-loop). The class definition also has the special Reference Class **initialize()** function:

```
initialize = function(seed) {
  .self$seed <- seed
  for (i in 1:100) {   # burn a few
    dummy <- .self$Next()
  }
},
```

If you define an **initialize()** function in an S4 or RC class, control will be transferred to **initialize()** when you instantiate an object of the class using the **new()** function. Here, the **initialize()** function sets the **seed** field, then generates and discards the first 100 values. Notice the use of the **.self** keyword to access the **seed** field and the **Next()** function.

The seed value could have been set with:

```
seed <<- seed
```

Here, the **seed** on the left side of the **<<-** assignment operator is the class field, and the **seed** on the right side of the assignment is the input parameter. But that syntax looks a bit odd. Whenever I set a field value using a parameter with the same name, I like to use the explicit **.self** syntax. Notice, however, that when you do assign an RC class field value using **.self**, you must use the regular **<-** assignment operator rather than the **<<-** assignment operator.

The demo program instantiates an object this way:

```
my_rng <- LehmerRng$new(as.integer(1))
```

An initial seed value of 1 is passed to the **new()** function. Some RNG algorithms can use an initial seed value of 0, but the initial seed must be 1 or greater for the Lehmer algorithm, which means you could add an error-check to the **initialize()** function. Because the class has a defined **initialize()** function, control is passed from **new()** to **initialize()**. The input parameter value of 1 is assigned to the **seed** field, and then the first 100 random values are generated and discarded. Burning a few hundred or a few thousand initial values is a common technique in RNG implementations.

The demo program uses the RNG object to generate a few random floating-point values:

```
for (i in 1:8) {
  x <- my_rng$Next()
  xx <- formatC(x, digits=4, format="f")
  cat(xx, " ")
}
```

The demo program shows how to map random floating-point values between 0.0 and 1.0 to integers between [1, 6] inclusive this way:

```
lo <- as.integer(1)
hi <- as.integer(7)  # note: 1 more than 6
for (i in 1:20) {
  n <- floor((hi - lo) * my_rng$Next() + lo)
  cat(n, " ")
}
```

In summary, if you want multiple, independent random-number generator objects, you can define your own RC class. If you define the optional **initialize()** function, when assigning a field value that has the same name as an input parameter, consider using the **.self** syntax.

Resources

An excellent, famous paper that describes how to implement the Lehmer RNG algorithm is:
http://www.firstpr.com.au/dsp/rand31/p1192-park.pdf.

For additional information about the Lehmer RNG algorithm, see:
https://en.wikipedia.org/wiki/Lehmer_random_number_generator.

5.2 Neural networks

This section explains the input-process-output mechanism of an R language implementation of a neural network. This mechanism is called neural network feed-forward.

Code Listing 16: A Neural Network

```r
# neuralnets.R
# R 3.2.4

NeuralNetwork = setRefClass(
  "NeuralNetwork",

  fields = list(
    ni = "integer",
    nh = "integer",
    no = "integer",

    inputs = "array",
    ihWeights = "matrix",
    hBiases = "array",

    hiddens = "array",
    hoWeights = "matrix",
    oBiases = "array",
    outputs = "array"
  ),

  methods = list(
    initialize = function(ni, nh, no) {
      .self$ni <- ni
      .self$nh <- nh
      .self$no <- no

      inputs <<- array(0.0, ni)
      ihWeights <<- matrix(0.0, nrow=ni, ncol=nh)
      hBiases <<- array(0.0, nh)

      hiddens <<- array(0.0, nh)
      hoWeights <<- matrix(0.0, nrow=nh, ncol=no)
      oBiases <<- array(0.0, no)
      outputs <<- array(0.0, no)
    }, # initialize()

    setWeights = function(wts) {
      numWts <- (ni * nh) + nh + (nh * no) + no
      if (length(wts) != numWts) {
        stop("FATAL: incorrect number weights")
      }

      wi <- as.integer(1) # weight index
      for (i in 1:ni) {
        for (j in 1:nh) {
```

```r
      ihWeights[i,j] <<- wts[wi]
      wi <- wi + 1
    }
  }
  for (j in 1:nh) {
    hBiases[j] <<- wts[wi]
    wi <- wi + 1
  }
  for (j in 1:nh) {
    for (k in 1:no) {
      hoWeights[j,k] <<- wts[wi]
      wi <- wi + 1
    }
  }
  for (k in 1:no) {
    oBiases[k] <<- wts[wi]
    wi <- wi + 1
  }
}, # setWeights()

computeOutputs = function(xValues) {
  hSums <- array(0.0, nh) # hidden nodes scratch array
  oSums <- array(0.0, no) # output nodes scratch

  for (i in 1:ni) {
    inputs[i] <<- xValues[i]
  }

  for (j in 1:nh) { # pre-activation hidden node sums
    for (i in 1:ni) {
      hSums[j] <- hSums[j] + (inputs[i] * ihWeights[i,j])
    }
  }

  for (j in 1:nh) { # add bias
    hSums[j] <- hSums[j] + hBiases[j]
  }

  for (j in 1:nh) { # apply activation
    hiddens[j] <<- tanh(hSums[j])
  }

  for (k in 1:no) { # pre-activation output node sums
    for (j in 1:nh) {
      oSums[k] <- oSums[k] + (hiddens[j] * hoWeights[j,k])
    }
  }

  for (k in 1:no) { # add bias
    oSums[k] <- oSums[k] + oBiases[k]
  }

  outputs <<- my_softMax(oSums) # apply activation
  return(outputs)
```

```r
    }, # computeOutputs()

    my_softMax = function(arr) {
      n = length(arr)
      result <- array(0.0, n)
      sum <- 0.0
      for (i in 1:n) {
        sum <- sum + exp(arr[i])
      }
      for (i in 1:n) {
        result[i] <- exp(arr[i]) / sum
      }
      return(result)
    }

  ) # methods
) # class

# -----

cat("\nBegin neural network demo \n")

numInput <- as.integer(3)
numHidden <- as.integer(4)
numOutput <- as.integer(2)
cat("\nSetting numInput = ", numInput, "\n")
cat("Setting numHidden = ", numHidden, "\n")
cat("Setting numOutput = ", numOutput, "\n")

cat("\nCreating neural network \n")
nn <- NeuralNetwork$new(numInput, numHidden, numOutput)

cat("\nSetting input-hidden weights to 0.01 to 0.12 \n")
cat("Setting hidden biases to 0.13 to 0.16 \n")
cat("Setting hidden-output weights to 0.17 to 0.24 \n")
cat("Setting output biases to 0.25 to 0.26 \n")
wts <- seq(0.01, 0.26, by=0.01)
nn$setWeights(wts)

xValues <- c(1.0, 2.0, 3.0)
cat("\nSetting input values: ")
print(xValues)

cat("\nComputing output values \n")
outputs <- nn$computeOutputs(xValues)
cat("\nDone \n")
cat("\nOutput values: ")
print(outputs)

cat("\nEnd demo\n")
```

```
> source("neuralnets.R")

Begin neural network demo

Setting numInput =  3
Setting numHidden =  4
Setting numOutput =  2

Creating neural network

Setting input-hidden weights to 0.01 to 0.12
Setting hidden biases to 0.13 to 0.16
Setting hidden-output weights to 0.17 to 0.24
Setting output biases to 0.25 to 0.26

Setting input values: [1] 1 2 3

Computing output values

Done

Output values: [1] 0.4920477 0.5079523

End demo
```

Let's think of a neural network as a complicated math function that accepts numeric input values, does some processing in a way that loosely models biological neurons, and produces numeric output values that can be interpreted as predictions.

Here are the key lines of demo program code that create a neural network:

```
numInput <- as.integer(3)
numHidden <- as.integer(4)
numOutput <- as.integer(2)

nn <- NeuralNetwork$new(numInput, numHidden, numOutput)
```

The number of input and output nodes is determined by the prediction problem. The number of hidden nodes is a free parameter and must be determined by trial and error.

The demo neural network has three input nodes, four hidden nodes, and two output nodes. For example, if you're trying to predict whether a person is a political conservative or liberal (two output possibilities) based on income level, education level, and age level (three input values).

After the neural network object is created, it is initialized with these lines of code:

```
wts <- seq(0.01, 0.26, by=0.01)
nn$setWeights(wts)
```

A neural network has constants called weights and biases that, along with the input values, determine the output values. A neural network that has x input nodes, y hidden nodes, and z output nodes has (x * y) + y + (y * z) + z weights and biases. That means the demo 3-4-2 network has (3 * 4) + 4 + (4 * 2) + 2 = 26 weights and biases. The demo generates 26 values from 0.01 to 0.26 using the built-in **seq()** function, then uses a class function/method **setWeights()** to copy the values into the neural network's weights and biases.

The demo computes output values for an input set of (1.0, 2.0, 3.0) with this code:

```
xValues <- c(1.0, 2.0, 3.0)
outputs <- nn$computeOutputs(xValues)
print(outputs)
```

The two output values are (0.4920, 0.5080). This diagram shows the conceptual structure of the demo neural network:
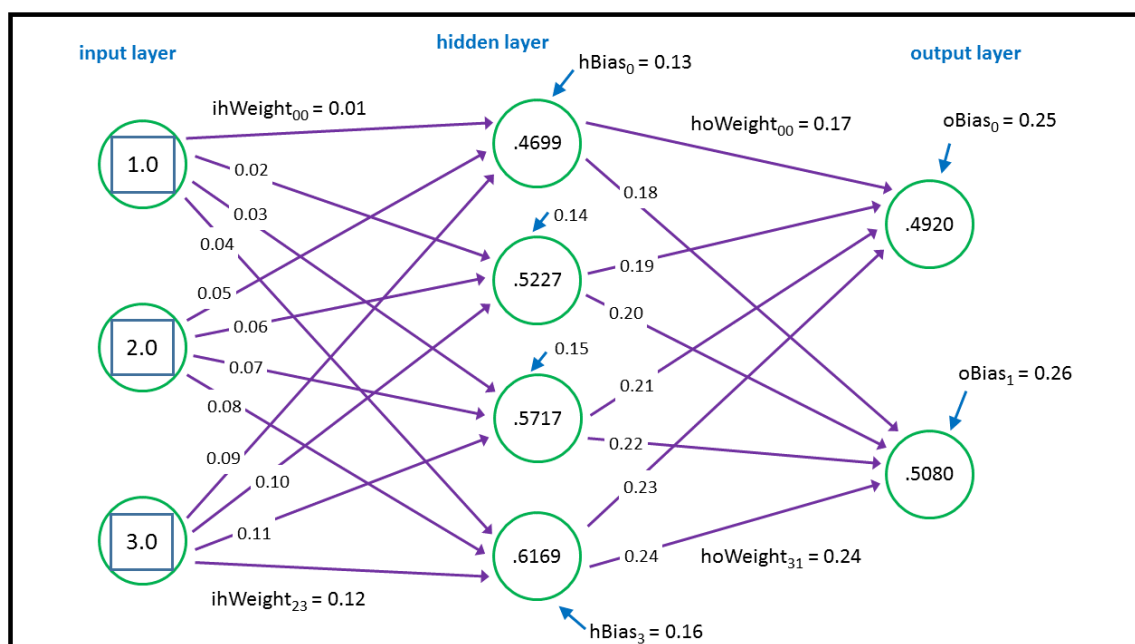
Figure 23: Neural Network Architecture

Each purple line connecting a pair of nodes represents a weight. Additionally, each of the hidden and output nodes (but not input nodes) has a blue arrow that represents a bias. The neural network calculates its outputs from left to right. The top-most hidden node value is calculated:

tanh((1.0 * 0.01) + (2.0 * 0.05) + (3.0 * 0.09) + 0.13) = tanh(0.51) = 0.4699

The tanh (hyperbolic tangent) is called the activation function in neural network terminology. In words, we'd say "in order to calculate a hidden node value, you multiply each input value times the associated input-to-hidden weight, sum, add the hidden node bias value, and then take the tanh of the sum."

The values of the two output nodes are calculated a bit differently. First, preliminary sums are calculated:

oSums[1] = (0.46 * 0.17) + (0.52 * 0.19) + (0.57 * 0.21) + (0.61 * 0.23) + 0.25 = 0.69
oSums[2] = (0.46 * 0.18) + (0.52 * 0.20) + (0.57 * 0.22) + (0.61 * 0.24) + 0.26 = 0.72

Then these values are used to determine the final output values by using a special function called the **softmax** function:

outputs[] = softmax(oSums[1], oSums[2]) = (0.4920, 0.5080)

These output values can be interpreted as probabilities. In this case, because the second value is (just barely) larger than the first value, the neural network predicts that a person with an income level of 1.0, an education level of 2.0, and an age level of 3.0 is a liberal (the second outcome possibility).

Here is the structure of the demo neural network RC class:

```
NeuralNetwork = setRefClass(
  "NeuralNetwork",
  fields = list( . . . ),
  methods = list( . . .)
)
```

These are the class fields:

```
fields = list(
  ni = "integer", nh = "integer", no = "integer",
  inputs = "array",
  ihWeights = "matrix",
  hBiases = "array",

  hiddens = "array",
  hoWeights = "matrix",
  oBiases = "array",
  outputs = "array"
),
```

Fields **ni**, **nh**, and **no** are the number of input, hidden, and output nodes. Matrix **ihWeights** holds the input-to-hidden weights where the row index corresponds to an input node index, and the column index corresponds to a hidden node index. Array **hBiases** holds the hidden node bias values. Matrix **hoWeights** holds the hidden-to-output weights, and array **oBiases** holds the output node biases.

The **initialize()** function is defined as:

```
initialize = function(ni, nh, no) {
  .self$ni <- ni
  .self$nh <- nh
  .self$no <- no
```

```
    inputs <<- array(0.0, ni)
    ihWeights <<- matrix(0.0, nrow=ni, ncol=nh)
    hBiases <<- array(0.0, nh)

    hiddens <<- array(0.0, nh)
    hoWeights <<- matrix(0.0, nrow=nh, ncol=no)
    oBiases <<- array(0.0, no)
    outputs <<- array(0.0, no)
},
```

Although all the fields defined as array types could be made vectors, it's more principled to use the array type when interoperating with matrix objects.

If you modify a field in an RC class using the **.self** syntax, you use the **<-** assignment operator, but if you don't qualify a field with **.self**, you must use the **<<-** assignment operator.

The definition of class method **setWeights()** begins as:

```
setWeights = function(wts) {
  numWts <- (ni * nh) + nh + (nh * no) + no
  if (length(wts) != numWts) {
    stop("FATAL: incorrect number weights")
  }
. . .
```

Function **setWeights()** accepts an array or vector of all the weights and biases. If the number of weights and biases in the input **wts** parameter isn't correct, the function calls the built-in **stop()** function to halt program execution immediately. Function **setWeights()** continues by setting an index, **wi**, into the **wts** array/vector parameter, then populates the input-to-hidden weights matrix in row-order form (left to right, top to bottom):

```
wi <- as.integer(1)
for (i in 1:ni) {
  for (j in 1:nh) {
    ihWeights[i,j] <<- wts[wi]
    wi <- wi + 1
  }
}
```

Notice that the **<<-** assignment operator is needed to modify the member matrix, but ordinary **<-** assignment is used for the **wi** local variable. Function **setWeights()** finishes by copying value from the **wts** parameter into the hidden node biases, followed by the hidden-to-output weights, and finally to the output node biases:

```
for (j in 1:nh) {
  hBiases[j] <<- wts[wi]
  wi <- wi + 1
}
```

```
for (j in 1:nh) {
  for (k in 1:no) {
    hoWeights[j,k] <<- wts[wi]
    wi <- wi + 1
  }
}

for (k in 1:no) {
  oBiases[k] <<- wts[wi]
  wi <- wi + 1
}
```

In short, the **setWeights()** function deserializes its input array/vector values first to the input-to-hidden weights matrix in row-order fashion, then to the hidden node biases, then to the hidden-to-output weights matrix in row-order fashion, and finally to the output node biases. There is no standard order for copying weights, and if you're working with another neural network implementation, you must check to see exactly which ordering is being used.

The definition of member function **computeOutputs()** begins with:

```
computeOutputs = function(xValues) {
  hSums <- array(0.0, nh) # hidden nodes scratch array
  oSums <- array(0.0, no) # output nodes scratch
  for (i in 1:ni) {
    inputs[i] <<- xValues[i]
  }
. . .
```

Local array **hSums** holds the pre-tanh activation sums for the hidden nodes. Local array **oSums** holds the pre-softmax activation sums for the output nodes. The values in the input parameter **xValues** array/vector are then copied into the neural network's **input** array.

The values for the hidden nodes are computed with these lines of code:

```
for (j in 1:nh) {
  for (i in 1:ni) {
    hSums[j] <- hSums[j] + (inputs[i] * ihWeights[i,j])
  }
}
for (j in 1:nh) {
  hSums[j] <- hSums[j] + hBiases[j]
}
for (j in 1:nh) {
  hiddens[j] <<- tanh(hSums[j])
}
```

As explained earlier, first a sum of the products of input value times associated weight is computed, then the bias is added, then the **tanh()** activation function is applied.

The output values are calculated and returned this way:

```
for (k in 1:no) {
  for (j in 1:nh) {
    oSums[k] <- oSums[k] + (hiddens[j] * hoWeights[j,k])
  }
}
for (k in 1:no) {
  oSums[k] <- oSums[k] + oBiases[k]
}
outputs <<- my_softMax(oSums)
return(outputs)
```

Notice that unlike the hidden nodes, in which values are calculated one at a time, the output node values are calculated together by the **my_softMax()** program-defined function. The **softmax** function of one of a set of values is the exp() of that value divided by the sum of the exp() of all the values. For example, in the demo, the pre-softmax activation-output sums are (0.6911, 0.7229). The softmax of the two sums is:

output[1] = exp(0.6911) / (exp(0.6911) + exp(0.7229)) = 0.4920
output[2] = exp(0.7229) / (exp(0.6911) + exp(0.7229)) = 0.5080

The purpose of the softmax output-activation function is to coerce the output values so that they sum to 1.0, then can be loosely interpreted as probabilities, which in turn can be used to determine a prediction.

The feed-forward, input-process-output mechanism is just one part of a neural network. The next key part, which is outside the scope of this e-book, is called training the network. In the example neural network, the weights and biases values were set to 0.01 through 0.26 in order to demonstrate how feed-forward works. In a real neural network, you must determine the correct values of the weights and biases so that the network generates accurate predictions.

Training is accomplished by using a set of data that has known input values and known, correct output values. Training searches for the values of the weights and biases so that the computed outputs closely match the known, correct output values. There are several neural network training algorithms, but the most common is called the back-propagation algorithm.

In summary, it is entirely possible to implement a custom neural network, and an R Reference Class is well suited for an implementation. When writing an **initialize()** function, use the **.self** syntax for fields that have the same name as their associated parameters and use the **<-** assignment operator rather than the **<<-** operator.

Resources

For details about the relationship between the **new()** and **initialize()** functions, see:
https://stat.ethz.ch/R-manual/R-devel/library/methods/html/refClass.html.

5.3 Bee colony optimization

A combinatorial-optimization problem occurs when the goal is to arrange a set of items in the best way. The traveling salesman problem (TSP) is arguably the most well-known combinatorial-optimization problem, and bee colony optimization (BCO), an algorithm that loosely models the behavior of honey bees, is used to find a solution for TSP.

Code Listing 17: Bee Colony Optimization

```
# beecolony.R
# R 3.2.4

Bee = setClass(  # S4 class
  "Bee",

  slots = list(
    type = "integer", #1 = worker, 2 = scout
    path = "vector",
    error = "numeric"
  )

) # Bee

# -----

make_data = function(nc) {
  result <- matrix(0.0, nrow=nc, ncol=nc)
  for (i in 1:nc) {
    for (j in 1:nc) {
      if (i < j) {
        result[i,j] <- 1.0 * (j - i)
      }
      else if (i > j) {
        result[i,j] <- 1.5 * (i - j)
      }
      else {
        result[i,j] <- 0.0
      }
    }
  }
  return(result)
}

error = function(path, distances) {
  nc <- nrow(distances)
  mindist <- nc - 1
  actdist <- 0
  for (i in 1:(nc-1)) {
    d <- distances[path[i], path[i+1]]
    actdist <- actdist + d
  }
  return(actdist - mindist)
}
```

```r
solve = function(nc, nb, distances, max_epochs) {
  # create nb random bees
  numWorker <- as.integer(nb * 0.80)
  numScout <- nb - numWorker
  hive <- list()

  for (i in 1:nb) {
    b <- new("Bee")

    # set type
    if (i <= numWorker) {
      b@type <- as.integer(1) # worker
    }
    else {
      b@type <- as.integer(2) # scout
    }
    # set a random path and its error
    b@path <- sample(1:nc)
    b@error <- error(b@path, distances)

    hive[[i]] <- b # place bee in hive
  }

  # find initial best (lowest) error
  best_error <- 1.0e40 # really big
  best_path <- c(1:nc) # placeholder path
  for (i in 1:nb) {
    if (hive[[i]]@error < best_error) {
      best_error <- hive[[i]]@error
      best_path <- hive[[i]]@path
    }
  }

  cat("Best initial path = \n")
  print(best_path)
  cat("Best initial error = ")
  cat(formatC(best_error, digits=1, format="f"))
  cat("\n\n")

  # main processing loop
  epoch <- 1
  while (epoch <= max_epochs) {

    # cat("epoch =", epoch, "\n")
    if (best_error <= 1.0e-5) { break }

    # process each bee
    for (i in 1:nb) {
      if (hive[[i]]@type == 1) { # worker
        # get a neighbor path and its error
        neigh_path <- hive[[i]]@path
        ri <- sample(1:nc, 1)
        ai <- ri + 1
```

```
      if (ai > nc) { ai <- as.integer(1) }
      tmp <- neigh_path[[ri]]
      neigh_path[[ri]] <- neigh_path[[ai]]
      neigh_path[[ai]] <- tmp
      neigh_err <- error(neigh_path, distances)

      # is neighbor path better?
      p <- runif(1, min=0.0, max=1.0)
      if (neigh_err < hive[[i]]@error || p < 0.05) {
        hive[[i]]@path <- neigh_path
        hive[[i]]@error <- neigh_err

        # new best?
        if (hive[[i]]@error < best_error) {
          best_path <- hive[[i]]@path
          best_error <- hive[[i]]@error
          cat("epoch =", formatC(epoch, digits=4))
          cat(" new best path found ")
          cat("error = ")
          cat(formatC(best_error, digits=1, format="f"))
          cat("\n")
        }
      } # neighbor is better


    }
    else if (hive[[i]]@type == 2) { # scout
      # try random path
      hive[[i]]@path <- sample(1:nc)
      hive[[i]]@error <- error(hive[[i]]@path, distances)
      # new best?
      if (hive[[i]]@error < best_error) {
        best_path <- hive[[i]]@path
        best_error <- hive[[i]]@error
        cat("epoch =", formatC(epoch, digits=4))
        cat(" new best path found ")
        cat("error = ")
        cat(formatC(best_error, digits=1, format="f"))
        cat("\n")
      }

      # waggle dance to a worker
      wi <- sample(1:numWorker, 1)  # random worker
      if (hive[[i]]@error < hive[[wi]]@error) {
        hive[[wi]]@error <- hive[[i]]@error
        hive[[wi]]@path <- hive[[i]]@path
      }
    } # scout

  } # each bee
  epoch <- epoch + 1
} # while

cat("\nProcessing complete \n")
cat("Best error found = ")
```

```
    cat(formatC(best_error, digits=1, format="f"))
    cat("\n")

    cat("Best path found = \n")
    print(best_path)
} # solve

# -----

cat("\nBegin TSP using bee colony optimization demo \n\n")

set.seed(7) #
numCities <- as.integer(20)
numBees <- as.integer(100)
max_epochs <- as.integer(5000)

cat("Setting numCities =", numCities, "\n")
cat("Setting numBees =", numBees, "max_epochs =", max_epochs, "\n\n")

distances <- make_data(numCities ) # city-city distances

optPath <- c(1:numCities)
cat("Optimal path = \n")
print(optPath)
cat("\n")

solve(numCities, numBees, distances, max_epochs)

cat("\nEnd demo \n")
```

```
> source("beecolony.R")

Begin TSP using bee colony optimization demo

Setting numCities = 20
Setting numBees = 100 max_epochs = 5000

Optimal path =
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20

Best initial path =
 [1] 20 12  9 10  3  5  4  2  7  6  1 17 16 18 19 11  8 15 13 14
Best initial error = 77.5

epoch =      8 new best path found error = 76.5
epoch =     10 new best path found error = 71.5
epoch =     14 new best path found error = 59.0
epoch =     34 new best path found error = 57.5
epoch =     45 new best path found error = 45.0
epoch =     55 new best path found error = 40.0
epoch =    124 new best path found error = 37.5
```

```
epoch =    288 new best path found error = 35.5
epoch =    292 new best path found error = 25.5
epoch =    520 new best path found error = 23.5
epoch =    541 new best path found error = 21.0
epoch =    956 new best path found error = 20.5
epoch =   3046 new best path found error = 19.0
epoch =   3053 new best path found error = 16.5
epoch =   3056 new best path found error = 14.0
epoch =   3059 new best path found error = 11.5
epoch =   3066 new best path found error = 10.0
epoch =   3314 new best path found error = 7.5
epoch =   4665 new best path found error = 6.0
epoch =   4686 new best path found error = 4.0
epoch =   4689 new best path found error = 2.5
epoch =   4714 new best path found error = 0.0

Processing complete
Best error found = 0.0
Best path found =
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20

End demo
```

In using BCO to find a solution to TSP, the demo sets up a problem in which 20 cities exist and the goal is to find the shortest path that visits all 20 cities. The demo creates a problem by defining a **make_data()** function and an associated **error()** function. Code Listing 18 shows the function **make_data()** definition.

Code Listing 18: Programmatically Generating TSP City-to-City Distances

```
make_data = function(nc) {
  result <- matrix(0.0, nrow=nc, ncol=nc)
  for (i in 1:nc) {
    for (j in 1:nc) {
      if (i < j) {
        result[i,j] <- 1.0 * (j - i)
      }
      else if (i > j) {
        result[i,j] <- 1.5 * (i - j)
      }
      else {
        result[i,j] <- 0.0
      }
    }
  }
  return(result)
}
```

The function accepts a parameter **nc**—the number of cities. The return object is a matrix that defines the distance between two cities in which the row index is the "from" city and the column index is the "to" city. Every city is connected to every other city. The distance between two consecutive cities is best explained with examples: distance(3, 4) = 1.0; distance(4, 3) = 1.5; and distance(5, 5) = 0.0.

When defined this way, the optimal path that visits all 20 cities is 1 -> 2 -> . . -> 20. This path has a total distance of 19.0. In a realistic scenario, the distances between cities would probably be stored in a text file, and you'd write a helper function for reading the data from the file into a matrix.

Here is the associated **error()** function:

```
error = function(path, distances) {
  nc <- nrow(distances)
  mindist <- nc - 1
  actdist <- 0
  for (i in 1:(nc-1)) {
    d <- distances[path[i], path[i+1]]
    actdist <- actdist + d
  }
  return(actdist - mindist)
}
```

The function accepts a path that has 20 city indices, walks through the path, and accumulates the total distance using the information in the distances matrix. The optimal-path distance is subtracted from the actual-path distance to give an error value.

In most realistic scenarios, you won't know the optimal solution to your problem, which means you can't define an error function as optimal minus actual. An alternative approach is to define error as a very large value (such as .Machine$integer.max) minus actual.

BCO is a metaheuristic rather than a prescriptive algorithm. This means BCO is a set of general design guidelines that can be implemented in many ways. The demo program models a collection of synthetic bees. Each bee has a path to a food source and an associated measure of that path's quality. There are two kinds of bees—worker bees try different paths in a controlled fashion, and scout bees try random paths.

The demo program defines a **Bee** object using an S4 class:

```
Bee = setClass(
  "Bee",
  slots = list(
    type = "integer", #1 = worker, 2 = scout
    path = "vector",
    error = "numeric"
  )
)
```

The Bee class has data fields but no class functions/methods. The **Bee** object types are hard-coded as 1 for a worker and 2 for a scout. In many programming languages you'd use a symbolic constant, but R does not support this feature.

All of the work is done by a program-defined **solve()** function. Expressed in high-level pseudocode, this is the algorithm used:

```
initialize a collection of bees with random path values
loop max_epochs times
  loop each bee
    if bee is a worker then
      examine an adjacent path
      if adjacent path is better, update bee
    else if bee is a scout then
      examine a random path
      pick a worker bee at random
      communicate the random path to the worker ("waggle")
  end loop each bee
end loop max_epochs
display best path found
```

The definition of function **solve()** begins with:

```
solve = function(nc, nb, distances, max_epochs) {
  numWorker <- as.integer(nb * 0.80)
  numScout <- nb - numWorker
  hive <- list()
. . .
```

The total number of bees, **nb**, is passed in as a parameter. The number of worker bees is set to 80% of the total, and the rest are scout bees, although you can experiment with other percentages. The collection of **Bee** objects is stored in a list object. Even though the total number of bees is known, when working with collections of S4 or RC objects, I find it preferable to use a variable-length list object rather than a fixed-length vector object.

The **Bee** objects are initialized this way:

```
for (i in 1:nb) {
  b <- new("Bee")
  if (i <= numWorker) {
    b@type <- as.integer(1) # worker
  }
  else {
    b@type <- as.integer(2) # scout
  }

  b@path <- sample(1:nc)
  b@error <- error(b@path, distances)
  hive[[i]] <- b # place bee in hive
}
```

The built-in **sample()** function is used to generate a random permutation. The associated error of the random path is computed and stored. Notice the double-square bracket notation.

After all the **Bee** objects have been initialized, they are examined to find the best initial path and associated error:

```
best_error <- 1.0e40 # really big
best_path <- c(1:nc) # placeholder path
for (i in 1:nb) {
  if (hive[[i]]@error < best_error) {
    best_error <- hive[[i]]@error
    best_path <- hive[[i]]@path
  }
}
```

This initialize-best-path logic could have been placed inside the preceding initialization loop, but it's a bit cleaner to use separate loops. The main while-loop processes each bee in the hive object. Here is the code in which a worker bee generates a neighbor path to the current path:

```
neigh_path <- hive[[i]]@path
ri <- sample(1:nc, 1)
ai <- ri + 1
if (ai > nc) { ai <- as.integer(1) }
tmp <- neigh_path[[ri]]
neigh_path[[ri]] <- neigh_path[[ai]]
neigh_path[[ai]] <- tmp
neigh_err <- error(neigh_path, distances)
```

Variable **ri** is a random index. Variable **ai** is an adjacent index that is only one greater than the random index. You might want to investigate an alternative design by defining a neighbor path as one in which the cities at two randomly selected indices are exchanged. Or you could define two different types of worker bees that have different neighbor-generation strategies.

Here is the code in which a **Bee** object's path is updated:

```
p <- runif(1, min=0.0, max=1.0)
if (neigh_err < hive[[i]]@error || p < 0.05) {
  hive[[i]]@path <- neigh_path
  hive[[i]]@error <- neigh_err
. . .
```

A probability is generated and, 5% of the time, the path is updated even if the neighbor path is not better than the current path. This helps prevent bees from getting stuck at local minima.

In summary, BCO is a bio-inspired, swarm-intelligence metaheuristic that can be used to solve combinatorial problems. Swarm algorithms can use a list of S4 objects.

Resources

For additional information about bee colony optimization, see:
https://en.wikipedia.org/wiki/Bees_algorithm.