THE ART OF REASONING IS NOTHING MORE THAN A LANGUAGE WELL ARRANGED.

ÉTIENNE BONNOT DE CONDILLAC, 1790

THE VALIDITY OF ALL THE INDUCTIVE METHODS DEPENDS ON THE ASSUMPTION THAT EVERY EVENT, OR THE BEGINNING OF EVERY PHENOMENON, MUST HAVE SOME CAUSE; SOME ANTECEDENT, UPON THE EXISTENCE OF WHICH IT IS INVARIABLY AND UNCONDITIONALLY CONSEQUENT.

JOHN STUART MILL, 1911

I CAN'T BE AS CONFIDENT ABOUT COMPUTER SCIENCE AS I CAN ABOUT BIOLOGY. BIOLOGY EASILY HAS 500 YEARS OF EXCITING PROBLEMS TO WORK ON. IT'S AT THAT LEVEL.

DONALD KNUTH, 1993

SHAWN T. O'NEIL

BIO/RECURSION

AN EXPLORATION IN R

PUBLISHED BY —

SHAWNTONEIL.COM

First printing, February 2018

Contents

Dedicated to:

Dr. Andrew A. Poe

a.k.a. "Captain Recursion"

1 *Programming in R*

> The desire to economize time and mental effort in arithmetical computations, and to eliminate human liability to error is probably as old as the science of arithmetic itself.
>
> Howard Aiken, 1937

While the contents of this book may be implemented in almost any programming language, we'll be implementing them in R. R (a derivative of an earlier language called S) was originally designed for statistical computing: producing linear models, analyses of variance, that sort of thing. As we'll see though, R supports a full complement of programming techniques. It also includes some additional easy-to-use features for visualization (via the `ggplot2` and `TurtleGraphics` libraries) which will be useful for this material. In this chapter we'll briefly introduce the R language and some basic programming concepts; not enough to be considered a thorough introduction to R specifically, or programming generally, but enough for our own purposes.

For those with an existing background in programming, R can be described as a *dynamically typed*, *pass-by-value*, *multi-paradigm (but largely functional)*, and *vectorized* language.[1]

- *Dynamically typed:* Variables holding data in R can change their data (and the type of data they hold) over time; `a <- 5` sets `a` to be a "numeric" type, later a line like `a <- "XB"` can set it to a "character" type (similar to "strings" in other languages). Further, there is no need to indicate a variable's type when it is first created. This can be contrasted with Java's `int a = 5; String b = "XB";` where `a` is forever destined to hold an integer and `b` can only hold strings.

- *Pass-by-value:* Parameters given to functions effectively become *copies* during the function call, meaning that changes made inside

[1] For those without extensive programming experience, don't let these technical definitions intimidate you. Feel free to read them for their terminological value and focus on the code samples later in this chapter.

the function call are not saved after the call ends. If we consider lines such as `letters <- c("A", "b", "C")` and `answer <- tolower(letters)`, then `letters` will still hold `c("A", "b", "C")` while `answer` will hold `c("a", "b", "c")`. This is unsurprising–the point is that the `tolower()` function couldn't modify the original `letters` data *even if it wanted to*. This is in contrast to "pass-by-reference" languages like Python and Java, where functions are free to modify the data underlying the parameters so that those changes persist after the function ends.[2] For example, it is possible to write a Python function where `answer = tolower(letters)` results in `letters` being changed to `["a", "b", "c"]` and `answer` holding the number of changes made (`2` in this case).

[2] The terms "pass-by-value" and "pass-by-reference" are historically laden and indicate fairly specific behavior. In attempts to avoid confusion, the Python community uses the term "pass-by-name" and some Java programmers use the tortured "pass-by-value-of-the-reference" to specify what amounts to slight variations on the basic principle.

- *Multi-paradigm (but largely functional):* Although R supports most of the major programming paradigms, including object-oriented (where collections of data and functions form "objects" meant to represent real-world entities) and procedural (meaning we specify how the computer should perform tasks in step-by-step terms), its functional aspects edge out the others in typical usage. This means that functions are assigned to variables like any other type of data, and functions are free to take functions as parameters and return functions as answers. (R also supports other advanced functional concepts such as closures.) Because this book focuses on recursion but also loop-based techniques, we'll have a unique opportunity to explore the transition points between traditional procedural and more "functional" techniques.
- *Vectorized:* As a statistical language, R enforces vectorized operations as opposed to single-datapoint operations, a feature shared with only a few other languages (e.g. MATLAB). For example, `a <- c(1, 2, 3, 4)` stores in `a` a vector of four numbers, as does `b <- c(2, 4, 1, 7)`. The operation `a * b` returns the vector `c(2, 8, 3, 28)`, while `a < b` returns the vector of logical values `c(TRUE, TRUE, FALSE, TRUE)`. In fact, even `a <- 5` stores a vector in `a`, where the vector contains only a single numeric element. We'll be making some use of this; given a vector like `chars <- c("A", "T", "G", "C")`, we can get a subvector holding all but the last element as `subchars <- chars[1:(length(chars) - 1)]`, which reduces to `chars[1:3]`, returning the vector `c("A", "T", "G")`. This example highlights another difference between R and many other languages–indices in R start at `1` rather than `0` (so `chars[1]` holds `"A"`, and there is no `chars[0]`).

R is a large language, and unfortunately one with many special cases–although R functions are by default pass-by-value, it is *possible* to pass some data by reference. As such, these definitions and the

rest of this chapter constitute more of a rough field-guide to R than a definitive reference.

1.1 *Installing R*

The most straightforward way to write and execute R programs is to first download and install the R interpreter[3] itself from `http://r-project.org`, and then download the RStudio Integrated Development Environment (IDE) from `http://rstudio.org`. While R itself handles the execution of R code, RStudio provides a text editor aware of R's specialized syntax and facilitates sending R code to the R interpreter for execution.

[3] Interpreter: a program designed to interpret, or read and execute lines of code. This is in contrast to compiled languages, meaning the code is translated into machine code readable directly by the CPU.

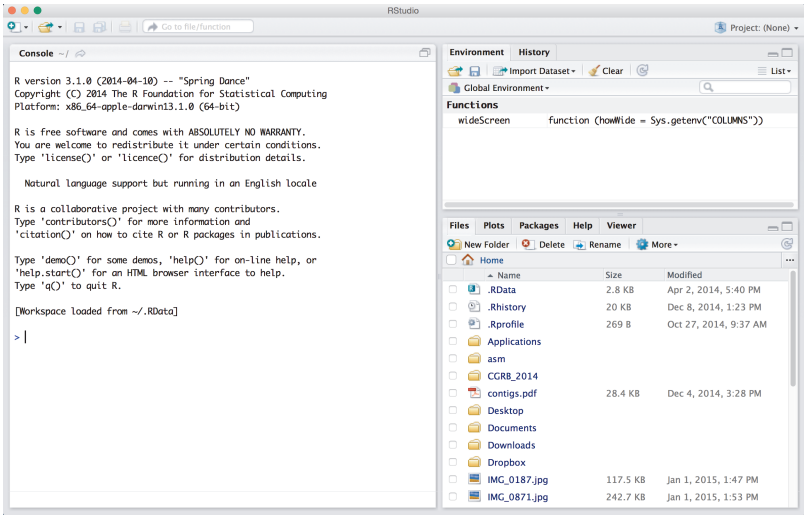


Figure 1.1: RStudio after initial installation.

When first opened, RStudio presents three panes (Figure 1.1). On the left is the R interpreter itself–here we can type a line of R code like `print("Hello World")` followed by the Enter key and that line will be executed. The upper-right pane shows some of the variables currently stored by the R interpreter; executing a line like `a <- c(1, 2, 3, 4)` will cause `a` to appear here. Later, a line like `print(a)` will print the stored contents of `a`. The lower-right pane shows a file browser, tab for help information, and a plots tab for the output of graphical function calls.

None of these panes, however, are where we'll be primarily working. Rather than executing lines of R code one at a time, we want to write R programs, or *scripts*: files containing many lines of code that can be executed as a batch. To edit such a file, we can select File → New File → R Script. This creates a new pane in the upper-left portion of the window with a text editor for writing the script (Figure 1.2).[4]

To execute lines in a script, we can either 1) highlight them (with the mouse) and click the Run button, which sends them to the

[4] Actually, it is a good idea to create a "Project" in RStudio, which is a simply a folder that might contain multiple R scripts and other files. RStudio also supports the creation of "R Markdown" files: documents containing human-readable text interleaved with chunks of code and their output, which can later be exported as reports in PDF or HTML format.

Figure 1.25: Parameters and variables created inside functions with <- are local variables; they shadow existing variables of the same name and are removed when the function call ends.

```
add_two_nums <- function(a, b) {
 result <- a + b
 return(result)
}

a <- "test a"
result <- "test result"
x <- 4
y <- 7
answer <- add_two_nums(x, y)
print(answer)                          # prints 11
print(a)                               # prints "test a"
print(result)                          # prints "test result"
print(b)                               # Error: object 'b' not found
```

Figure 1.26: Global variables may be read as any other (unless they are shadowed by a local variable) and are written to with <<-. Here we're using capital letters only as a reminder that they are global.

```
add_two_nums <- function(a, b) {
 CALL_COUNTER <<- CALL_COUNTER + 1
 result <- a + b
 return(result)
}

CALL_COUNTER <<- 0                     # initialize global
answer <- add_two_nums(2, 4)
print(CALL_COUNTER)                    # prints 1
answer <- add_two_nums(6, 5)
print(CALL_COUNTER)                    # prints 2
```

1.4 *Packages*

R packages are downloadable add-ons providing additional functions. Most packages are stored in an online repository called the Comprehensive R Archive Network, or CRAN. Installing a package from CRAN is straightforward (assuming an active internet connection is available). For example, to install the `stringr` package, in the interpreter window (at the > prompt, *not* in the script), run `install.packages("stringr")`. You may be you prompted via a popup window to select the nearest download location.

Once the process is complete, there are two ways to utilize functions provided by the package in a script. The first is to prefix the function name with the package name and `::`, as in `greeting <- stringr::str_c("hi", "there")`. This line runs the `str_c()` function from `stringr` (which concatenates character vectors). Alternatively, one may first "load" the functions in a package by calling `library()`; in this example we would first run `library(stringr)` and then `greeting <- str_c("hi", "there")`. Usually these calls to `library()` are located at the top of the script.

Throughout this book we will use functions from several packages. Rather than describe these functions and packages in detail

```
> a <- c(4, 5, 6)
> a
[1] 4 5 6
```

Figure 1.27: Values or variables that are not assigned are printed by default.

```
check_val <- function(val) {
  if(val < 10) {
    return()
  }
  print("Given value is at least 10.")
}

check_val(5)        # prints NULL
```

Figure 1.28: A function which should do nothing if a small value is given.

here, we'll just summarize them and note on which pages they are first introduced. Also, rather than prefix the function names, we will assume all scripts load these libraries at the top (Figure 1.29).

```
library(stringr)
library(rstackdeque)
library(hash)
library(TurtleGraphics)
library(ggplot2)

# Code below
# ...
```

Figure 1.29: Loading libraries needed for scripts.

- `stringr`: This package provides a number of functions for working with character vectors and their elements. Many of these functions have "regular" R equivalents–for example, `str_c()` from `stringr` is very similar to the built-in `paste()`–but the functions in `stringr` are more consistent in names and parameters. Functions from `stringr` we'll use include `str_sub()`, `str_length()`, `str_c()`, and `str_detect()`; these are described on page 39.

- `rstackdeque`: While the various data structures built into R (data frames, lists, matrices, etc.) are powerful, there are a couple we'll need that are not provided. The first are "stacks" (page 59) `rstackdeque` provides these with functions `rstack()`, `insert_top()`, `without_top()`, `peek_top()`, and `empty()`, as well as helper conversion functions `as.list()` and `as.data.frame()`. Other structures provided by this package are queues, through functions `rpqueue()`, `insert_back()`, `without_front()` and `peek_front()`. The functions provided by `rstackdeque` are described on page 69.

- `hash`: Another data structure we'll make heavy use of is the hash table. The `hash` package provides these through functions `hash()`, `has.key()`, `keys()`, and `values()` (page 72).

- `TurtleGraphics`: The `TurtleGraphics` package provides a simple interface for programmatic, line-based drawing. We'll use this only in the last chapter, where we'll learn how some of the techniques in other parts of the book can be visualized in beautiful ways. Functions provided by `TurtleGraphics` include `turtle_init()`, `turtle_forward()`, `turtle_turn()`, and several others described on page 133.

- `ggplot2`: This package provides a powerful toolset for plotting data stored in data frames. The main function provided by this package is `ggplot()`, which works in coordination with others like `geom_line()` and `geom_tile()`. Unfortunately, the `ggplot2` package is large and a useful tutorial for it would be outside the scope of this book, so we'll present and use `ggplot2` code without explanation. Excellent resources abound, including the official website `http://docs.ggplot2.org`, books such as *ggplot2: Elegant Graphics for Data Analysis* by Hadley Wickham, and *A Primer for Computational Biology* by Shawn T. O'Neil (yours truly) available as an open-access resource and in print.

1.5 Getting Help

Because of R's size and complexity, the official documentation and other unofficial resources are extremely helpful. The most important too in this is the `help()` function: when run at the interactive `>` prompt it will show a help page on a given function name. For example, `help("length")` will show information on the `length()` function.[17]. To get help on an installed package like `stringr`, try `help(package = "stringr")`

[17] A shortcut for `help()` is `?`; so `?length` is a shortcut for `help("length")`.

There also exist a number of excellent books and online resources. For readers new to programming and R, I would recommend (naturally) my own *A Primer for Computational Biology*, though many other books recently on the market are good as well. For beginner to intermediate programmers, I like *The Art of R Programming* by Norm Matloff. For a deeper look at R, *Advanced R* by Hadley Wickham is enlightening.

2 *Recursive Structures*

> A fool sees not the same tree that a wise man sees.
>
> William Blake, 1793

Let's consider a simple R list, as shown in Figure 2.1. Organizing data into lists is one of the most common features of programming, and R lists allow us to store any type of data in sequential order.

```
nums <- list(3, 5, 12)
```

Figure 2.1: A simple list of numbers.

How might a list like this be stored in the computer's memory? It might be stored simply as a contiguous sequence of binary numbers; in this case if the numbers were stored in an 8-bit representation, this could simply be `000000110000010100001100` (where `00000011` is 3, and so on).

Unfortunately, a strategy like this won't work for more general types of data. What if we wanted to store a list of more complicated objects, as in `names <- list("Mary", "Joe", "Allison")`? In this case, what might be stored is again a series of integers, but this time those integers indicate the "address" or location of the data in memory. Thus, a list like `100111000010110100101101` would indicate that `"Mary"` exists at address 156 (`10011100`), `"Joe"` at address 45 (`00101101`), and so on (Figure 2.2).[1] These address elements are sometimes known as "references" (though references also often keep track of the type of data at the given address). The primary advantage of a system like this is that data elements of arbitrary size can be "referenced" by the list elements.

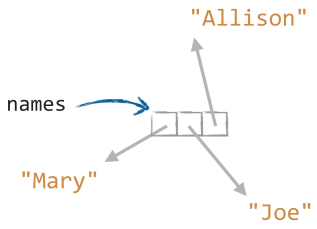


Figure 2.2: Visualizing the "address" model for lists. Here we are indicating that individual data element may occur anywhere in memory by distributing them visually in the figure.

Now, what if we desire to append a new element, say `"Katie"`, to the list? This would require creating the new data element for

[1] Most modern computers use 64 bits to store a simple integer; some programming languages like R define their own representation using many more to allow for much larger numbers. Using more bits for addresses also allows the computer to reference more places in memory; this is one of the primary advantages of using a 64-bit processor over a 32-bit processor.

"Katie" (which gets stored in memory anywhere it will fit by the interpreter working in conjunction with the operating system), and *appending* to the list the new address for it.[2] Let's assume it is fairly easy for the system find a sufficiently large chunk of memory, somewhere, for the new data element. But what about appending the new address? If the next few bits of memory (after the current list of addresses) are unused, then they can simply be written at the end of the current list and some bookkeeping updated to indicate the new length of the list. Frequently, however, we run into a situation where the next few bits of memory are already being used for some other important data!

[2] Although we're discussing lists in terms of R lists, in reality R lists and other data structures use a combination of these techniques and others mentioned later in this chapter.

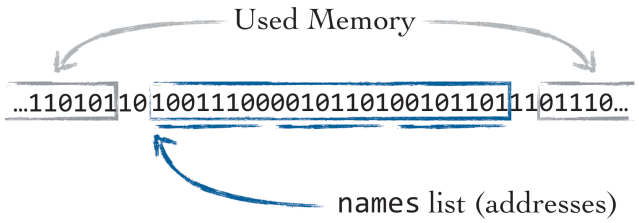


Figure 2.3: An example layout of memory. In this case, we can't easily append to the `names` list because not enough unused space exists at the end of it.

In this situation, there are a few options. First, lists (even lists of addresses) could be allowed to stretch over non-contiguous segments of memory, which would also require extra bookkeeping to manage where each portion of a list resides and how long it is. Second, the system might look for a new, larger segment of unused memory, and simply copy all of the addresses to that location so the append can be completed.[3]

As programmers, we can help the system out by considering a third, rather unusual option. Rather than keep lots of bookkeeping for lists split over arbitrary-sized pieces, or copy data when we run out of room, we can simply *restrict* lists to be of a given size, say two. If we know that a list will never have more than two elements, we can avoid the problem altogether![4]

[3] This copy-when-out-of-room strategy is used by R vectors, which can result in obnoxious amounts of copying for vectors that are appended to frequently. In general, it is not a good idea to "grow" native R types like vectors, lists, matrices, and data frames via many append operations for this reason.

[4] Although the examples in this section all consider data as simple strings like `"Mary"` and `"A"`, they could very well be more important data elements like DNA strings (e.g. `"CTAGAC"`) or even collections of multiple pieces of information (e.g. `list("CTAGAC", 6, 0.23)`).

Perhaps you are thinking, "how can lists restricted to holding only two elements possibly be of any use?" The answer involves the addressing strategy described above, which allows lists to hold data of arbitrary size and type (including other lists!) as elements. Thus, we can represent a list of `"A"`, `"B"`, `"C"`, and `"D"` as shown in Figure 2.4.

```
chars <- list("A", list("B", list("C", list("D", NULL))))
```

Figure 2.4: Storing an arbitrary number of elements in lists restricted to length two.

In this example, we are using nested lists in a "element, rest" structure: the first element of the list is the first data element, and the second element stores the "rest" of the list. We use the special data type `NULL` for the rest to indicate a list which has no elements (Figure 2.5). We're following our "rule" just fine: each list in the nested structure above has exactly two elements. For some general-

ity, we'll represent an empty list with something like `chars <- NULL`; thus a list either a "element, rest" pair, or `NULL` (empty).

Now, for the interpreter and operating system, appending a new piece of data like `"E"` simply requires finding some free space for a two-element list (`list("E", NULL)`) and modifying the address of the innermost `NULL` to address it instead.

Accessing a given element of this structure directly can be quite tedious. For example, we can access the third element as `chars[[2]][[2]][[1]]`. Soon we'll see more elegant ways to access individual elements. But first, what if we wanted to print all of the data elements in order? The first straightforward method is shown in Figure 2.6. In this strategy, we create a "working copy" of the data in a variable called `data`. In a loop, we'll print the first element of `data`, and then *set* `data` *to the second element of* `data` (so that it then contains the rest of the list). The output of this bit of code reliably prints `"A"`, `"B"`, `"C"`, and `"D"` (not shown).

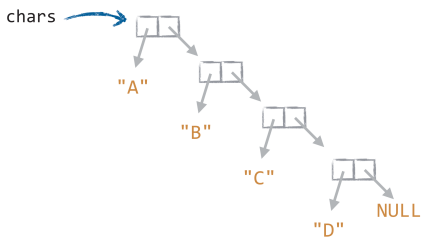


Figure 2.5: An "element, rest" list containing four data elements.

```
data <- chars
while(!is.null(data)) {
  el <- data[[1]]
  print(el)
  data <- data[[2]]
}
```

Figure 2.6: Printing a nested-list structure with a loop.

Figure 2.7 shows a more sophisticated strategy that uses a function to accomplish the same task.

```
print_list <- function(data) {
  if(is.null(data)) {
    return(invisible())
  }
  el <- data[[1]]
  rest <- data[[2]]
  print(el)
  print_list(rest)
}
```

Figure 2.7: Recursive method for printing the elements of a nested list in order.

Now this function is a bit trickier. In analyzing how it works, we should remember that the input to the function should be any list of the right "structure;" `data` could be `list("A", NULL)`, or `list("A", list("B", NULL))`, or even just `NULL`. First, it checks to see if the input list is `NULL` (representing an empty list). If so, no work needs to be done at all, and the function can simply `return(invisible())`, ending the function execution. Otherwise, it extracts the first element as `el` and the `rest` list; it prints `el` and then calls `print_list(rest)`!

This works because `data` is a *local variable* in the function, meaning each execution of the function works with its own `data` as passed to it, independent of any other `data` that might exist for other

What would happen if `print_list(rest)` was called *before* `print(el)`? Could the behavior be replicated easily using a looping strategy as above?

Figure 2.24: Storing an ordering "key" and corresponding "value" in a tree.

```
simple_tree <- list(NULL, list("C", 4.7), NULL)
```

Figure 2.25: Using a key/value tree, ordered by keys.

```
simple_tree <- list(NULL, list("C", 4.7), NULL)
simple_tree <- insert_tree(simple_tree, list("A", 9.2))
simple_tree <- insert_tree(simple_tree, list("D", 5.6))
print_tree(simple_tree)
```

Now, why are structures of this type known as binary *search* trees? Primarily because we can efficiently search for data by key (an operation also supported by hash tables, discussed in later chapters).

Figure 2.27: Retrieving a value by key.

```
get_value <- function(data, key) {
  if(is.null(data)) {
    return(NA)                          # not present
  }

  left <- data[[1]]
  el <- data[[2]]
  right <- data[[3]]

  if(key == el[[1]]) {                  # found it!
    return(el[[2]])                     # return the value
  }

  if(key < el[[1]]) {
    answer <- get_value(left, key)      # look left
    return(answer)
  } else {
    answer <- get_value(right, key)     # look right
    return(answer)
  }
}

print(get_value(simple_tree, "C"))      # prints 4.7
print(get_value(simple_tree, "Q"))      # prints NA
```

```
[1] "A 9.2"
[1] "C 4.7"
[1] "D 5.6"
```

Figure 2.26: Output for code in Figure 2.25.

In Figure 2.27, `get_value()` is a recursive function that returns the value associated with a given tree. It operates similarly to the `insert_tree()` function, but with an extra case. First, it checks to see if the tree being searched in is empty; if so, it can simply return `NA`, the standard value for "unknown" in R. Otherwise, it extracts the `left`, `el`, and `right`; if the key being searched for is the key being held in `el` at this node in the tree, it can simply return the value. If `el` is not the right value, the function can recurse left, *or* right, depending on whether the key being searched for is less than the central element or not.

Of particular interest is how quickly we can retrieve a value this way from very large tree. The number of steps is equal to the

number of recursive calls, which is at most the largest *depth* of the tree, since recursive calls are mode on left or right nodes (but not both) at each level (Figure 2.28). If we are lucky, our tree will be very wide and not very deep. What is the smallest depth we can hope for? Consider a "full" tree, where all leaf nodes are at the same depth and every node has two children (Figure 2.29).

Surely such a tree can't be made less deep. In this case, at every level the size of the sub-tree in consideration is reduced by approximately half; if at the start the number of nodes in consideration for the search is n, at the next level it is $n/2$, then $n/4$, and so on. How many times can a number be divided in half until 1 is reached? The answer is $\log_2(n)$, which grows much slower then n itself (Figure 2.30).

Note that essentially the same search process occurs for inserting an element, except in this case the search always goes to a leaf, and so for a "full" tree the time to insert an element is also approximately $\log_2(n)$.

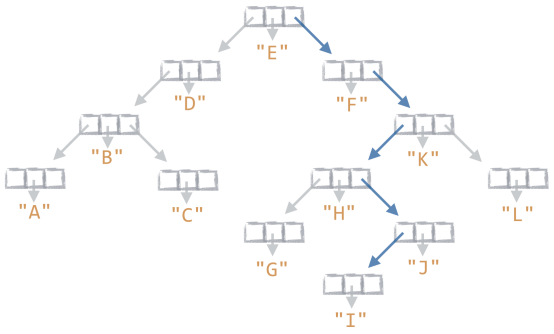
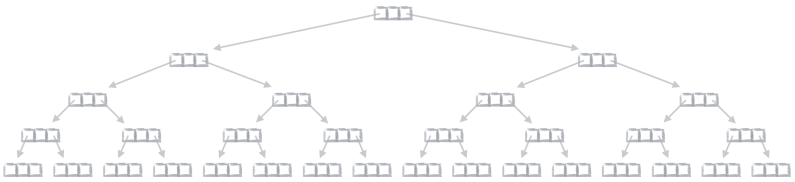


Figure 2.28: A tree of depth 6; data and arrows pointing to `NULL` have been removed for clarity. The search path to `"I"` is highlighted in blue: `"E"` is less than `"I"`, so the call is made on `right`, then `"F"` is less than `"I"`, so the call is made on `right` again, and so on.



Unfortunately, not all trees are full. If elements are inserted more or less randomly, the resulting tree will be quite likely to be close to full. But if elements are inserted in order, then they will stack up along the right (Figure 2.31, top). This can result in insertions or lookups to run in time much slower than $\log_2(n)$. More sophisticated tree types ensure that after every insertion or removal the tree stays "balanced," for example AVL trees may "rotate" triplets of nodes to ensure balance (Figure 2.31 bottom). There are a number of tree types that ensure balance and thus guarantee $\log_2(n)$-behavior, with names like 2-3 trees, red-black trees, and the aforementioned AVL trees.

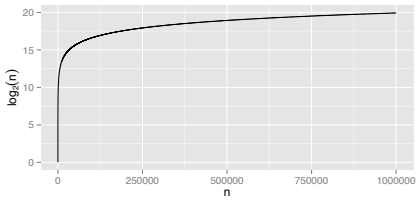


Figure 2.30: A plot of $\log_2(n)$. For even very large values of n the value of $\log_2(n)$ is small.

Exercises

1. Write a recursive function that returns the number of elements in a binary search tree.

2. Write a function that computes the depth of a tree, defined as the length of the longest path from the root to a leaf.

3. One problem with the key/value binary search tree we've imple-

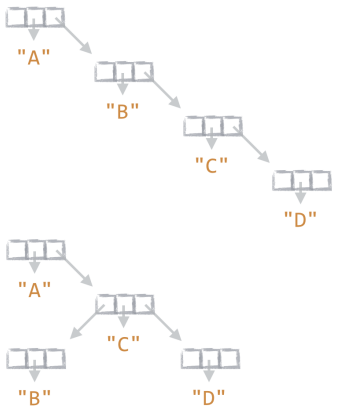


Figure 2.31: Top: an unbalanced tree resulting from elements being inserted in order. Bottom: the result of re-balancing the right sub-tree via rotation.

mented is that it allows duplicate keys to be stored in the tree, even if they are associated with different values. If duplicate keys are added, which entry will be returned by a search? Write a function that checks to see if a tree contains duplicate keys; it should return `TRUE` if it does, and `FALSE` if not. (You might first want to also implement `min_item()` and `max_item()` functions that return the smallest (resp. largest) items stored in a tree.)

4. Modify the `insert_tree()` function such that it refuses to insert a duplicate value and prints a warning if the user tries to do so.

5. The binary search trees discussed here are *persistent*, just like the nested linked lists of the last section. When performing an insertion, how many new nodes are created for the new tree, and how many are shared with the old version? You may want to draw a picture of node relationships before and after insertion, as in Figure 2.12 or Figure 2.13.

3 *Searching and Sorting*

> One essential object is to choose that arrangement which shall tend to reduce to a minimum the time necessary for completing the calculation.
>
> Ada Lovelace, 1843

To ground the discussion from the last chapters a bit, let's consider a common problem in bioinformatics, the *substring search* problem: given a long string (such as the DNA sequence for a chromosome, perhaps around 100 million letters), find matching locations of a short string (such as a DNA "read" produced from a sequencing machine, usually on the order of 100 letters).

Long sequence:

```
AGAGCCTAGAGCGAGAGTCCGTGAGACGACGAGAGACCCTGGACGAGACCG
```

Short sequence:

```
ACGA
```

Problem: find all locations where the short sequence matches the long sequence.

Figure 3.1: A small instance of the substring search problem.

To tackle this problem in R, we'll need to know about a few functions for working with character data, many of which are available in the `stringr` package. First, it's important to recall that in R the elements of a "character" vector are strings; if `names <- c("Johnny", "Marilyn")`, then `names[1]` returns the string `"John"`. Often, we'll be interested in vectors of a single string and working with substrings of it. The `str_sub()` function returns substrings of all elements of the vector: `str_sub(names, 2, 5)` returns the vector `c("ohnn", "aril")`, and if `name <- "Marilyn"` (which is just a character vector of length one) then `str_sub(name, 2, 5)` returns `"aril"` (another length-one vector).[1]

[1] Recall from Chapter 1 that the simplest form of data in R is the vector, and most functions take and return vectors.

Other useful functions in `stringr` include `str_length()` which returns the number of characters in a string, for example `str_length("Mary")` returns 4. The `str_c()` function concatenates strings using a given separator: `str_c("Mary", "O'Connor", sep = "_")` returns `"Mary_O'Connor"` while `str_c("Mary", "O'Connor")` returns `"MaryO'Connor"`.

The `str_detect()` function returns `TRUE` if a pattern is found in a string, and `FALSE` otherwise; to detect a pattern at the start or end of a string, we can use the regular expression patterns `^` and `$`. `str_detect("abracadabra", "^cad")` returns `FALSE` because `"cad"` does not occur at the start of `"abracadabra"`, while `str_detect("abracadabra", "abra$")` returns `TRUE` since `"abra"` occurs in the string at the end.[2]

[2] Regular expressions are a specialized and powerful syntax for finding patterns in strings. We'll be using only `^` and `$` from this syntax.

RETURNING TO SUBSTRING SEARCH, a straightforward way to solve this problem is to simply scan along the long sequence looking for matches with the shorter (or even using something like `str_detect()`). But this would be relatively slow, as it requires scanning over the entire longer string. For matching a single short sequence this might not be bad, but in practice we often have millions of short sequences (produced by the sequencing machine) that we wish to match against the chromosome. Better would be to "index" the large sequence in some way, potentially spending extra time up front to make each search go faster.

We'll start by indexing our chromosome via a binary search tree. Suppose we know that we'll never want to search for short sequences longer than, say, 10 characters. (In real life the number would be larger; we'll also use the 50 character sequence shown in Figure 3.1 as our genome.) Thus, we'll consider every 10-character substring of the genome, and store it in the binary tree as a key, along with the location of that substring as the value (Figure 3.2).

Figure 3.2: Adding substrings of the genome to a binary search tree.

```
genome <- "AGAGCCTAGAGCGAGAGTCCGTGAGACGACGAGAGACCCTGGACGAGACCG"

tree <- NULL

for(index in seq(1:str_length(genome))) {
  subseq <- str_sub(genome, index, index+9)
  element <- list(subseq, index)
  tree <- insert_tree(tree, element)
}

print_tree(tree)
```

```
[1] "ACCCTGGACG 36"
[1] "ACCG 48"
[1] "ACGACGAGAG 26"
[1] "ACGAGACCG 43"
[1] "ACGAGAGACC 29"
[1] "AGACCCTGGA 34"
[1] "AGACCG 46"
[1] "AGACGACGAG 24"
[1] "AGAGACCCTG 32"
[1] "AGAGCCTAGA 1"
[1] "AGAGCGAGAG 8"
[1] "AGAGTCCGTG 14"
```

Figure 3.3: Partial output for code shown in Figure 3.2.

Figure 3.3 shows the printed output, where entries are printed in dictionary order according to the sequences, along with the locations of those sub-sequences in the longer genome sequence. (Sub-sequences near the end are truncated to an appropriate length automatically by the `str_sub()` function, saving us from having to consider those as special cases.)

Given this tree, how can we find all locations where the short sequence `"ACGA"` occurs? Well, consider a tree (or sub-tree) with a root node holding the key `"AGAGCCTAGA"` and value 1 (Figure 3.4).

This sequence does not start with "ACGA", so "ACGA" does not occur at position 1 of the original sequence. Furthermore, "ACGA" is *less* than "AGAGCCTAGA" ("ACGA" < "AGAGCCTAGA" returns TRUE), and so if any nodes are to start with "ACGA" then they must be somewhere in the left sub-tree. (The same rules apply for query sequences greater, which would be found in the right sub-tree).

```
print_subseq_matches <- function(data, subseq) {
  if(is.null(data)) {
    return(invisible())
  }

  left <- data[[1]]
  el <- data[[2]]
  right <- data[[3]]
  pattern <- str_c("^", subseq)  # for str_detect at start of key

  if(!str_detect(el[[1]], pattern)) {     # no match
    if(subseq < el[[1]]) {                # continue search right
      print_subseq_matches(left, subseq)
    } else {           # continue search left
      print_subseq_matches(right, subseq) # continue search left
    }
  } else {                                # match!
    print(str_c(el[[1]], el[[2]], sep = " "))
    print_subseq_matches(left, subseq)    # continue search left
    print_subseq_matches(right, subseq)   # AND right
  }
}

print_subseq_matches(tree, "ACGA")
```

Figure 3.5: Finding matches for short sequences in the tree-index genome sequence.

On the other hand, if the sequence *did* start with "ACGA", then the value needs to be printed since "ACGA" occurs at that location. Further, in this case more matches might be found in both the left and right sub-trees, so the search must continue in both directions. Of course, if the tree is empty (NULL) then nothing needs to be done at all (Figure 3.5). This code will reliably indicate that "ACGA" occurs at positions 26, 29, and 43 of the original sequence.

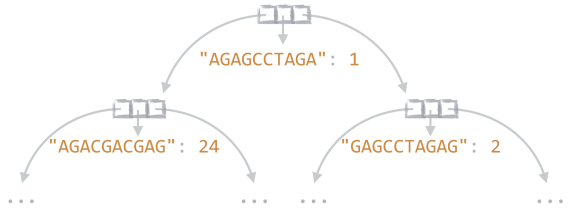


Figure 3.4: A key/value binary search tree, with subsequences of a long string indexing their locations within the string.

Now, the most important question is, how much work is required to find these matches? This recursive function is not nearly as simple as the previous ones we've seen, which always branch left or right–this function sometimes branches left or right, and sometimes branches both ways. This means the amount of work could be larger. As an extreme example, consider a tree built from the sequence "AAAAAAAAAAAAA" and the search sequence "A". Here the query will match at every node, and the branching will always be to both sides; every node will be visited

done. But what about if $n > 1$? The run time depends on how long it takes to perform the steps listed above. First, in step one, a random element is selected; presented without argument, R uses $O(1)$ time for this simple operation. In step 2, elements are extracted in comparison with the pivot. This requires comparing each element with the pivot, and so this step takes $O(n)$ time. The third step performs two recursive calls, on vectors of size (by simplifying assumption) $n/2$. Since we don't have any way to describe this runtime other than via $T()$, we'll say this is $2T(n/2)$. Finally, the three sub-lists must be concatenated; R performs this in $O(n)$ time since each element must be included in the answer. Thus, the total time in the recursive case is $O(1) + O(n) + 2T(n) + O(n)$, or just $2T(n/2) + O(n)$.

$$T(n) = \begin{cases} 2T(n/2) + O(n), & \text{if } n > 1, \\ O(1), & \text{otherwise.} \end{cases}$$

It is possible to come up with a simple solution for recurrence relations like this, using a process not too dissimilar from recursion. First, we start by replacing $T(n/2)$ according to the definition $T(n/2) = 2T(n/4) + O(n/2)$:

$$\begin{aligned} T(n) =& 2T(n/2) + O(n) \\ =& 2\left(2T(n/4) + O(n/2)\right) + O(n) \\ =& 4T(n/4) + 2O(n/2) + O(n) \\ =& 4T(n/4) + O(n) + O(n)\ . \end{aligned}$$

Similarly, we can replace $T(n/4)$ with $2T(n/8) + O(n)$, and with some rearrangement obtain

$$T(n) = 8T(n/8) + O(n) + O(n) + O(n)\ .$$

This process will continue, until we get an equation like $xT(n/x) + O(n) + O(n) + \cdots + O(n)$. How many steps can we do this, until n/x is approximately 1 (wherein $T(n) = O(1)$ will apply)? Since x is being doubled at each iteration, the number of steps is $\log_2(n)$, thus there will be $O(\log_2(n))$ terms of $O(n)$ in the entire series.

$$T(n) = O(1) + \underbrace{O(n) + O(n) + \cdots + O(n)}_{O(\log_2(n)) \text{ terms}}.$$

Thus, the total runtime for the algorithm under the given assumptions is $T(n) = O(n \log_2(n))$, significantly better than the $O(n^2)$ of bubblesort (Figure 3.20).[6]

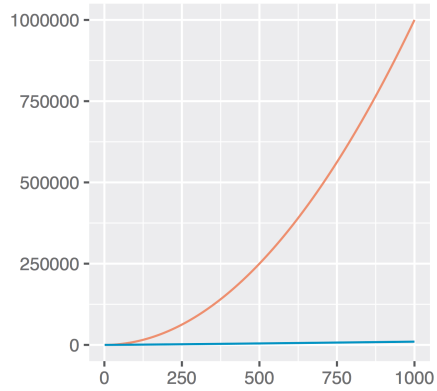


Figure 3.20: A plot of n^2 (red) and $n \log_2(n)$ (green) for values of n from 1 to 1,000.

Another interesting way to get a handle on the runtime of this function is with a visual diagram (Figure 3.21). We can illustrate the function calls as nodes in a "tree," wherein each node is sized

[6] We are taking some liberties in notation here, particularly in simplifying expressions like $2O(n/2)$ to $O(n)$. A more rigorous approach would not use order notation, but use variables for constants as in

$$T(n) = \begin{cases} 2T(n/2) + c_1 n + c_2, & \text{if } n > 1, \\ c_3, & \text{otherwise.} \end{cases}$$

You might try solving this recurrence yourself to verify it results in the same solution.

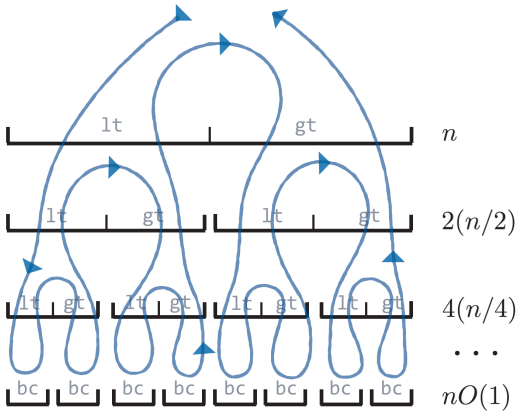


Figure 3.21: A "weighted call tree" for quicksort, under the assumption that every pivot splits `lessthan` and `greaterthan` (here represented by `lt` and `gt`, `bc` stands for "base case") into two roughly-equal halves. The blue line traces the path of execution: in any node a local `greaterthan` and `lessthan` are created, then `lessthan` is sorted recursively followed by `greaterthan` before the answer is concatenated and returned. In this path, going down a level represents a call, and going up a level represents a return.

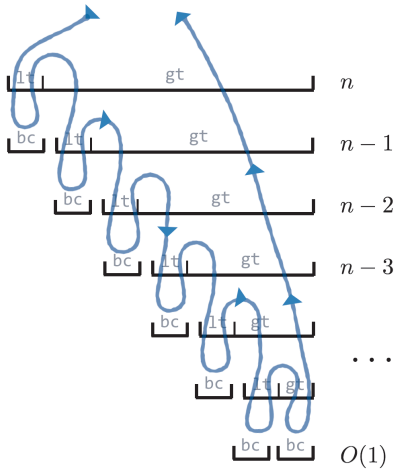


Figure 3.22: A "call tree" for quicksort, where the pivot is always the smallest or nearly the smallest element.

according to the amount of work done at that call (not counting the work done in recursive calls). Thus, the top node represents the $O(n)$ work done by the initial call, and this node calls two sub-nodes representing $O(n/2)$ work each; each of these make calls representing $O(n/4)$ work, and so on. But, at the level where each node requires $O(n/4)$ work, there are 4 such nodes; a similar argument applies to every level. The main question then is, how many levels are there? Notice the similarity to a balanced binary tree: there are $O(\log_2(n))$ levels, for a total amount of work of $O(n \log_2(n))$.

We've made a pretty big assumption here, which is that the randomly-chosen pivot is the median and always divides the input vector exactly in half. This will be true on average, but what happens if we get unlucky? Let's consider the case where the pivot always happens to be the smallest (or perhaps second-smallest) element of the list. In this case, `lessthan` will always be empty (and so sorting it will be an instance of the base case), and `greaterthan` will have $n-1$ elements. The recurrence relation for this case is

$$T(n) = \begin{cases} T(n-1) + O(n), & \text{if } n > 1, \\ O(1), & \text{otherwise.} \end{cases}$$

The corresponding graphical call-tree representation is shown in Figure 3.22. Now the amount of work at each level decreases incrementally, so it takes many more levels to reach the bottom. The total amount of work is $n + (n-1) + (n-2) + \cdots + 1 = n(n+1)/2$, which is $O(n^2)$ indicating that in the worst case quicksort, despite its name, can be as slow as bubblesort.

As we said, on average a randomly selected pivot will split the vector in half, and indeed it has been shown that with very high probability quicksort will run in $O(n \log_2(n))$ time. It is notable that quicksort does use random choices to guarantee this no matter the order of the input: any deterministic rule could result in $O(n^2)$ runtime if the data are "stacked against the rule." (For example, if the data are already sorted, always using the first element as the pivot is equivalent to being unlucky enough to always choosing the smallest element.)

Much as there are self-balancing binary trees that give strong guarantees on $O(\log_2(n))$ operations, there are other recursive sorting methods that guarantee $O(n \log_2(n))$ runtime without using randomization. Examples include mergesort and treesort (when using a balanced binary tree), which

will be covered in the exercises.

Exercises

1. Mergesort operates in a similar fashion as quicksort: the input vector is split into two equal-sized sub-vectors, these are sorted recursively, and then the answers are combined into the final answer. The main difference is in how the list is split in half.

   Mergesort requires a small auxiliary function called `merge()` which takes two sorted vectors and returns a single sorted vector with all of the elements. Write this function and argue that in runs in $O(n)$ time.

   With the `merge()` function written, `mergesort()` is simple: the base case is the same as for `quicksort()`, but otherwise split the input vector in two pieces called `first_half` (containing the first $\approx n/2$ elements) and `second_half` (containing the rest). Recursively call `mergesort()` on these halves, and use `merge()` to merge the sorted sub-vectors into a single answer to return. Since this method guarantees an equal split, its runtime is a guaranteed $O(n \log_2(n))$.

2. Treesort simply inserts all the elements of a vector into a binary search tree, and uses a recursive method to re-extract them to a vector (or list) in order. Argue that for a balanced binary tree, the runtime for treesort is also $O(n \log_2(n))$, and implement a treesort using the binary search trees discussed earlier.

3. A colleague thinks that if splitting the input vector into two parts works well for mergesort, then splitting into three (and recursing three times) must be even better. What would be the runtime of this modified sort? Is it in principle better, worse than, or equal to regular split-in-half version? Why? (It will help to have solved problem 5 on page 44.)

4 *Induction and Stacks*

> I have deeply regretted that I did not proceed far enough at least to understand something of the great leading principles of mathematics, for men thus endowed seem to have an extra sense.
>
> Charles Darwin, 1876

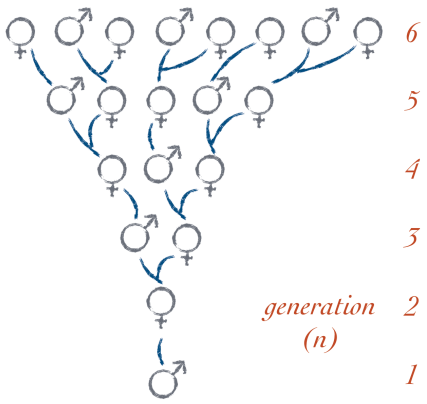


Figure 4.1: The family tree of a male bee.

Bees have an interesting biology. As you may know, a single female, the queen, lays eggs producing most of the male worker bees in a hive. These males have no father and are clones of the queen. Female (queen) bees on the other hand are rarer, and have both a mother and a father. The family tree for a male bee is illustrated in Figure 4.1. If we consider the number of ancestors of the bee at each generation (including himself as his "first" generation, as we'll count generations backward through time) then the sequence goes 1, 1, 2, 3, 5, 8, 13, and so on. You might recognize this as the famous Fibonacci sequence, after the 11th century mathematician who studied it. (Leonardo Fibonacci was also largely responsible for moving Europe away from the Roman Numeral counting system to the Arabic system we use today.) Apparently, the number of bees at a given generation n, which we'll call $bees(n)$ (and is often annotated as $fib(n)$, since the same equation describes the Fibonacci sequence), is the sum of the number of bees of the previous two generations ($1+1=2$, $1+2=3$, $2+3=5$, and so on):

$$bees(n) = \begin{cases} 1, & \text{if } n = 1, \\ 1, & \text{if } n = 2, \\ bees(n-1) + bees(n-2), & \text{otherwise.} \end{cases}$$

But, is it enough to simply assert this relationship? That the number of bees at a given generation is equal to the sum of the previous generations? Perhaps not. We've based this only on observation of a small number of cases, and we could envision an obstinate observer

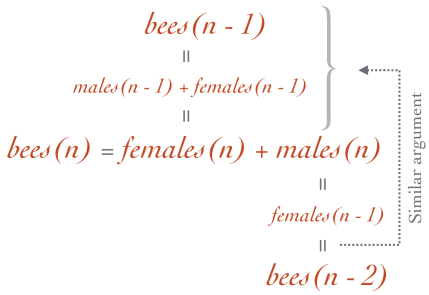


Figure 4.2: A sketch of the proof for $bees(n) = bees(n-1) + bees(n-2)$. The logic for $females(n-1)$ uses the same argumentation as for $females(n)$.

who might not agree that $bees(36) = bees(35) + bees(34)$ without a ridiculous extension of the drawing (there would be over 14 million bees in the figure at the 36th generation!) Rather, we should argue the relationship analytically by producing a proof.

Proof. Based on simple observation, we can argue conclusively that $bees(1) = 1$ and $bees(2) = 1$.

For any other n, let's *assume*, for the moment, that $bees(n-1)$ gives the correct number of ancestors at generation $n-1$ according to the formula, as does $bees(n-2)$ for generation $n-2$. If these facts are true (see discussion below), then all we need show is that $bees(n-1) + bees(n-2)$ gives the correct answer for generation n (since this is equated to $bees(n)$).

1. First, $bees(n) = females(n) + males(n)$, the number of females and males at generation n, respectively.

2. Consider $females(n)$: in the family tree, every bee (male and female) has exactly one mother. Further, every female in the tree gives birth to exactly one bee in the tree. Thus, the very rules of bee biology dictate that $females(n) = males(n-1) + females(n-1)$, which in turn equals $bees(n-1)$.

3. Now consider $males(n)$. Every male at generation n in the tree mates to contribute to exactly one female offspring at generation $n-1$ (in this family tree at least). Further, every female at generation $n-1$ has exactly one male parent at generation n. Thus, $males(n) = females(n-1)$.

4. By the same logic as step 2 above, $females(n-1) = bees(n-1)$.

Finally, we have that:

$$\begin{aligned} bees(n) &= females(n) + males(n) && \text{(step 1)} \\ &= bees(n-1) + males(n) && \text{(step 2)} \\ &= bees(n-1) + females(n-1) && \text{(step 3)} \\ &= bees(n-1) + bees(n-2) && \text{(step 4) .} \end{aligned}$$

□

THE STEPS ABOVE WILL WORK FINE for any given n, so long as we can actually *assume* that the formula produces good numbers for $bees(n-1)$ and $bees(n-2)$ as we did. Our simple observations showed that this is true for the first two generations, which means that the proof must hold for $n = 3$; that is, that $bees(3) = bees(2) + bees(1)$ $(2 = 1 + 1)$. This shows that the formula is further correct for $bees(3)$, and thus that the proof holds for $n = 4$ as well ($bees(4) =$

bees(3) + *bees*(2)). In a sense, our "proof" is an infinite series of sub-proofs, each relying on the correctness of the previous two (Figure 4.3).

Figure 4.3: Dependency of sub-proofs. We show the truth of $n = 1$ and $n = 2$ by observation, and these imply the truth of $n = 3$ by argumentation. Similarly, $n = 2$ and $n = 3$ imply the truth of $n = 4$, and so on.

This strategy of proving a "cascade of correctness" is known as *induction*. You might remember mathematical induction from a class in mathematics, for example to prove an identity like $1 + 2 + \cdots + n = n(n+1)/2$ (Figure 4.4). In simple mathematical induction, there is a base case where the identity is shown for a small value, and an inductive case where the cascade is described mathematically.

In the case of our bees, we're reasoning more generally about structures that exist in nature (or computer programs). We still have a base case (or cases), and we still have an inductive case (or cases). But we call this more general form of proof *structural induction*, as it relies on more than mere manipulation of formulae.

Base Case, $n = 1$:

$$1 = \frac{1(1+1)}{2} = \frac{2}{2} \checkmark = 1$$

Inductive Case, $n > 1$:

$$\underbrace{1 + 2 + \cdots + (n-1)}_{\text{Assuming true for } n-1:} + n = \frac{n(n+1)}{2}$$

$$\underbrace{\frac{(n-1)((n-1)+1)}{2}}_{\text{Rearrangment:}}$$

$$\frac{n(n+1)}{2} \qquad \checkmark \frac{n(n+1)}{2}$$

Figure 4.4: A sketch of the proof for $1 + 2 + \cdots + n = n(n+1)/2$ by mathematical induction.

4.1 Computing the Fibonacci Sequence

At this point, rather than using *bees*(n), let's switch to $fib(n)$ to emphasize that the sequence of numbers we're describing is indeed the Fibonacci sequence. How might we write a program to compute the n^{th} Fibonacci number? We can use recursion, and the code itself nearly mirrors the definition. Indeed, the contents of the `else` block in Figure 4.5 could simply be `return(fib(n-1) + fib(n-2))`.

```
fib <- function(n) {
  if(n == 1) {
    return(1)
  } else if(n == 2) {
    return(1)
  } else {
    a <- fib(n - 1)
    b <- fib(n - 2)
    answer <- a + b
    return(answer)
    # or simply: return(fib(n - 1) + fib(n - 2))
  }
}

print(fib(8))              # prints 21
```

Figure 4.5: A function for computing the n^{th} Fibonacci number.

The reason this code produces correct answers is due to the way functions and local variables work, as in the recursive functions from previous chapters. Here, we called `fib(8)`, an instance of the

case, the fact that `fib(1000)` is a ridiculously large number is a limitation in itself.)

Exercises

1. An earlier exercise (page 58) asked you to program a pair of recursive functions for a predator/prey model. Your task now is to memoize these functions with a pair of global hash tables. Your "key" should be a string that identifies all of the parameters, as in this skeleton code:

```
predators <- function(n, r, b, c, d) {
  this_call <- str_c(n, ";", r, ";", b, ";", c, ";", d)
  # ...
```

2. Next, create a dynamic programming solution for the predator/prey model. This will require two tables of the right design (or, alternatively, a matrix with two rows). You may need to draw a diagram of the subproblems and their dependencies: what does `predators(n)` depend on? What about `prey(n)`?

3. Complete the hash table example, by first adjusting `append_end()` so that duplicate keys cannot be stored within the same nested list, and if a user attempts to do so the value for that key is overwritten. Further, implement a `has_key()` method which takes a list of buckets, and returns `TRUE` if a given key is present, and `FALSE` otherwise. Try using your custom hash table to memoize `fib()`.

5.2 *Notes on Software Engineering*

Many programmers dislike the use of global variables that are updated across function calls, and for good reason: they add complexity and "state" within the program for the programmer to keep track of. A function that reads and writes global data can no longer be thought of as a black box that only turns given parameters into required outputs. Even our relatively benign memoization caches will be cumbersome, since we need to remember to initialize them before using them.

Functional languages like R provide some unique tricks to help manage this complexity. As you've perhaps surmised, functions in R are types of data like any other, and their names are the variables holding the data. Consider the syntax for our `fib()` function: `fib <- function(n) ...`. Here, `fib` is a variable name that we've assigned data (the function) to using `<-`. This allows us to do interesting

things like pass function as parameters to other functions–we saw an example of this when we wrote `nested_lapply()` in Chapter 2.

Additionally, we know that variables assigned to within a function with <- are *local*, meaning they are unique to that function call and disappear when the call ends. This important feature not only enables recursion, but also keeps code clean and manageable.

Combining these two facts, we find that we can define *local functions* within other functions. Figure 5.16 show an outer function definition, and within it an inner function definition.

```
fib <- function(n) {

  fib_inner <- function(n_inner) {
    if(n_inner == 1 | n_inner == 2) {
      answer_inner <- 1
      return(answer_inner)
    } else {
      answer_inner <- fib_inner(n_inner - 1) +
                      fib_inner(n_inner - 2)
      return(answer_inner)
    }
  }

  answer <- fib_inner(n)
  return(answer)
}

print(fib(20))     # 6765
```

Figure 5.16: A function defined locally within another.

Here, `fib()` defines a local variable `fib_inner` (which happens to be a function), and then later calls `fib_inner()` to produce `answer`. If we call `fib()`, the inner function `fib_inner()` will be created, but it will only exist as long as the outer `fib()` is executing.

Perhaps you can see where we're headed: we're going to memoize `fib_inner()`, and the cache will be a local variable of the outer `fib()` (Figure 5.17).

Now when we call `fib()`, it 1) defines a local `fib_inner()` function, 2) defines a local `fib_cache` hash table, and 3) calls `fib_inner()`, which makes use of the cache as a "global" variable. How does this work, given that earlier in the book we said that <<- assigns to a global variable, but now we are assigning to a local variable of the outer function? Actually, we fibbed a bit earlier. What <<- really does is assign "up" the hierarchy of calls (up the variables in the call stack!) Even if we are deep within a nest of calls, <<- will assign to the first outer variable of that name that it can find.

Thus, from the perspective of outside the main `fib()` call, there are no global variables and no extra state to keep track of; `fib()` itself handles initializing the cache, using it, destroying it when

Figure 5.17: Memoizing an inner function.

```
fib <- function(n) {

  fib_inner <- function(n_inner) {
    key <- str_c("n_inner: ", n_inner)
    if(has.key(key, fib_cache)) {
      return(fib_cache[[key]])
    }

    if(n_inner == 1 | n_inner == 2) {
      answer_inner <- 1
      fib_cache[[key]] <<- answer_inner
      return(answer_inner)
    } else {
      answer_inner <- fib_inner(n_inner - 1) +
                      fib_inner(n_inner - 2)
      fib_cache[[key]] <<- answer_inner
      return(answer_inner)
    }
  }

  fib_cache <- hash()
  answer <- fib_inner(n)
  return(answer)
}

print(fib(20))     # 6765
```

done. In fact, if we are comfortable enough with local variables shadowing external variables, we don't even need the awkward names of `fib_inner`, `n_inner`, and `answer_inner` (which we used purely for clarity). We could replace these with `fib`, `n`, and `answer` and the code would work identically.

In future chapters we won't be encapsulating our memoized functions along with their caches in outer functions. We'll instead opt to keep the code simpler–but potentially more difficult to manage–by using global variables and focusing on the theoretical topics at hand.

6 *Alignment*

> Beware of bugs in the above code; I have only proved it correct, not tried it.
>
> Donald Knuth, 1977

The topics covered in the previous chapters–recursion, induction, memoization, dynamic programming–are important and powerful, but using them to compute Fibonacci numbers isn't that useful. In this chapter we'll bring these ideas together to solve one of the most important problems in bioinformatics: sequence alignment. The problem is defined as follows: given two DNA strings α and β of length n and m, produce a "good" alignment of these by inserting `"-"` ("gap") characters so that they are the same length. As an example, consider Figure 6.1; the output produces modified versions of α and β by inserting gaps.

Input:

α : ACTAGC

β : ATACC

Output:

ACTAGC
| | | |
A-TACC
+2 -4 +2 +2 -3 +2

Total Score: +1

Figure 6.1: An example sequence alignment.

What makes a "good" alignment? Intuitively, a good alignment produces a large number of matches (four in this example). But this is not the only possible criteria, and the details of defining goodness depend on what the goal is. We are often interested in aligning DNA sequences because we want to compare their sequences after evolutionary divergence. Consider the hemoglobin protein, which carries oxygen in red blood cells. This protein is defined by an A/C/G/T DNA sequence in animal genomes, from fish to humans to mice. All of these animals share a common ancestor, and this ancient ancestor also had an "original" hemoglobin sequence. As time passed and species diverged, the hemoglobin sequence changed via mutation in different ways for different species, and some of these changes were kept or lost by evolutionary pressures (Figure 6.2).

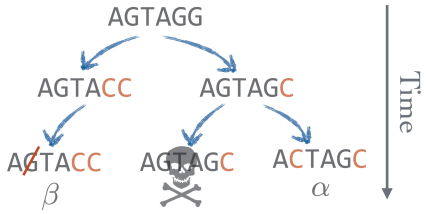


Figure 6.2: Sequence divergence over evolutionary time. Modern species shared common ancestors in the past, and thus also common DNA sequences. Over time the species diverged (and some went extinct), and so did the sequence, leaving us to find only α and β.

So, we would expect the hemoglobin sequence to be very similar, but not identical, between humans and mice. Further, we now know certain things about how DNA sequences mutate over time. For example, the loss of a DNA base (which would require inserting a gap in the alignment) is rarer than a DNA-base switch. As such, we should prefer alignments with fewer gaps even if in some cases it means accepting more mismatches. In practice this means alignments are evaluated based on a scoring scheme: for example, $+2$ points for a match, -3 for a mismatch, and -4 for a gap.[1] With these rules in place, the goal is to produce an alignment that maximizes the score. We call such an alignment *optimal* with respect to the scoring scheme. (There may be a number of different optimal alignments with the same score; we're interested in finding any one.[2])

It should be noted that a scoring scheme like this makes sense only if matching bases are worth more than mismatches and gaps: gap score < mismatch score < `0` < match score. (Imagine what would happen if the gap score was $+2$ and the others were -2: the score could be maximized by adding millions of gap characters!)

In very simple cases, finding an optimal alignment is easy. Because gaps are scored lower than mismatches (scored lower than matches), if α and β are only one character each, then the optimal alignment is simply the two inputs. Although odd to consider, if one of the sequences is "empty" (i.e. `""`), then the optimal alignment simply pads out the empty sequence with gaps to make it the same length as the other (Figure 6.3).

What about more complex cases? Consider some optimal alignments as shown in Figure 6.4 (which we've perhaps computed by hand). In all three cases, the end of the alignments have either a gap (`-`) or the last base of one or both of the input sequences. And after all, how could they be anything else?

In fact, if we go so far as to say that all sequences begin with an identical common character like `@`, then even some of the simple cases become complex cases. For example, aligning just `"A"` with just `"T"` can be recast as aligning `"@A"` with `"@T"`, a complex case. We consider this now as it will make the code later much simpler, as we have only two types of cases to deal with: simple cases (where one or both of the sequences are just `"@"`) and complex ones (all others, Figure 6.5). For scoring purposes, we'll assume that `"@"` aligned with `"@"` has a score of `0`.

For convenience, let's create some definitions. Given two strings α and β, let p_α be the "prefix" of α (containing all but the last character), and e_α be the "end" of α (the last character). Define p_β and e_β similarly for β (Figure 6.6). We'll also define a function $S()$ that represents the score of an alignment or a pair of characters as defined

[1] Determining scoring rules that match evolutionary history is a large area of study in itself. For example, we also know that A/T and C/G switches are more common than others, so we might score those mismatches slightly higher. In protein sequences we have strings of 20 letters (e.g. `"LPEAF..."`) with varying probabilities of switching. Large sets of genes are analyzed to determine these probabilities empirically, and the scoring points are determined from them.

[2] Computer scientists and mathematicians use the term "optimal" precisely: it is almost exclusively used to describe the best possible solution, or an algorithm that provably delivers one.

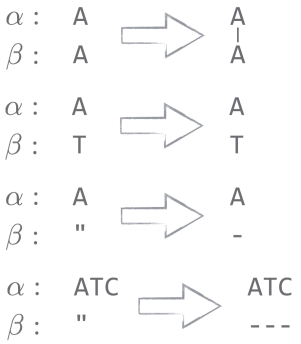


Figure 6.3: Easy cases of the sequence alignment problem: when both are only a single character, or one sequence is empty.

Figure 6.4: Any alignment can only end in one of three possible ways.

by whatever scoring scheme we're using.

Based on the observation above about how alignments can end, there are three possibilities for the alignment of α and β and their scores, based on sub-alignments that we'll call *left*, *center*, and *right* for reasons that will become clearer in a bit:

$$\begin{aligned}
&\underbrace{\begin{pmatrix} p_\alpha \text{ aligned w/} \\ p_\beta e_\beta \end{pmatrix}}_{left} \begin{matrix} e_\alpha \\ - \end{matrix}, \quad score = S(left) + S(e_\alpha, -) \\
&\underbrace{\begin{pmatrix} p_\alpha \text{ aligned w/} \\ p_\beta \end{pmatrix}}_{center} \begin{matrix} e_\alpha \\ e_\beta \end{matrix}, \quad score = S(center) + S(e_\alpha, e_\beta) \\
&\underbrace{\begin{pmatrix} p_\alpha e_\alpha \text{ aligned w/} \\ p_\beta \end{pmatrix}}_{right} \begin{matrix} - \\ e_\beta \end{matrix}, \quad score = S(right) + S(-, e_\beta)
\end{aligned} \tag{6.1}$$

The optimal alignment (the answer) is thus the best scoring of these three options, if the *left*, *center*, and *right* sub-alignments are themselves optimal. This relationship is often written in a more obtuse (but precise) mathematical notation:

$$S(\alpha_{1,i}, \beta_{1,j}) = \max \begin{cases} S(\alpha_{1,i-1}, \beta_{1,j}) + S(\alpha_i, \text{-}), \\ S(\alpha_{1,i-1}, \beta_{1,j-1}) + S(\alpha_i, \beta_j), \\ S(\alpha_{1,i}, \beta_{1,j-1}) + S(\text{-}, \beta_j), \end{cases}$$

where $S()$ is the scoring function, $X_{a,b}$ represents the subsequence of X from position a to position b, and X_i is just the i^{th} base of sequence X.

THERE IS A SIMILARITY HERE between solving the alignment problem and the "bees" problem of previous chapters. We've identified the patterns that we think we can use, in terms of very simple cases and more complex ones that depend on solving subcases (*left*, *center*, and *right*). In fact, what we have is a recursive definition! The argument we've put together for the solution of the problem is pretty strong. But does it constitute a proof? Could there be some outside possibility that using a recursive method in this way won't generate *optimal* alignments in the sense of maximizing the score? To be thorough, we can provide a proof-by-induction for this methodology.

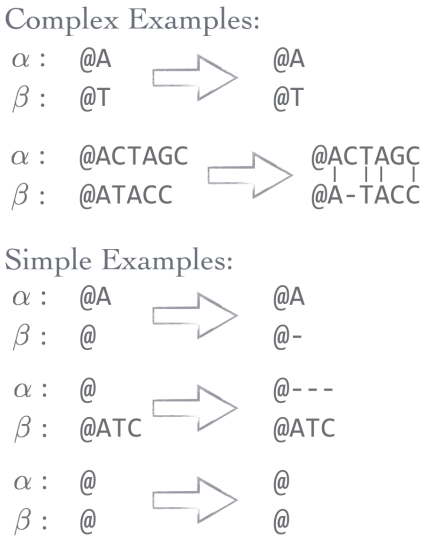


Figure 6.5: With the appropriate representation, "simple" cases of the alignment problem are those where at least one sequence is just "@"; all others can be considered as a complex case.

Figure 6.6: Definitions for p_α, p_β, e_α and e_β for sequences α and β.

```
base_case <- function(a_in, b_in) {
  answer <- list(a_in = a_in, b_in = b_in,
                 a_out = "", b_out = "", score = 0)

  if(length(a_in) == 1 & length(b_in) == 1) {   # empty/empty
    answer$a_out <- "@"
    answer$b_out <- "@"
  } else if(length(a_in) == 1) {                # a_in is just @
    answer$a_out <- rep("-", length(answer$b_out))
    answer$b_out <- b_in
    answer$a_out[1] <- "@"
  } else {                                      # b_in is just @
    answer$a_out <- a_in
    answer$b_out <- rep("-", length(answer$a_out))
    answer$b_out[1] <- "@"
  }

  answer$score <- score_aln(answer$a_out, answer$b_out)
  return(answer)
}

str(base_case(char_vec("@"), char_vec("@")))
str(base_case(char_vec("@"), char_vec("@A")))
str(base_case(char_vec("@TAC"), char_vec("@")))
```

Figure 6.11: A function that computes an answer for simple base-cases: those where either one or both input sequences are empty.

```
List of 5
 $ a_in : chr "@"
 $ b_in : chr "@"
 $ a_out: chr "@"
 $ b_out: chr "@"
 $ score: num 0
List of 5
 $ a_in : chr "@"
 $ b_in : chr [1:2] "@" "A"
 $ a_out: chr [1:2] "@" "-"
 $ b_out: chr [1:2] "@" "A"
 $ score: num -4
List of 5
 $ a_in : chr [1:4] "@" "T" "A" "C"
 $ b_in : chr "@"
 $ a_out: chr [1:4] "@" "T" "A" "C"
 $ b_out: chr [1:4] "@" "-" "-" "-"
 $ score: num -12
```

Figure 6.12: Output for Figure 6.11, showing base-case answers encoded as named R lists.

With the necessary tools in place, now we can work on the fun part: the main recursive function. Like `base_case()`, it will take in two character vectors `a_in` and `b_in`, and return an answer list containing the alignment and its score. We'll call our recursive function `global_aln()`. The first thing it can do is check to see if the inputs represent a base case: if so, then the answer can simply be returned from the `base_case()` function (Figure 6.13.1).

```
global_aln <- function(a_in, b_in) {
  if(length(a_in) == 1 | length(b_in) == 1) {
    return(base_case(a_in, b_in))
  }
  # to be continued...
```

Figure 6.13.1: The start of the recursive alignment function. If the inputs represent a simple base case, then the answer from `base_case()` can be returned.

If the inputs aren't a base case, we need to follow the recursive pattern by extracting p_α, e_α, p_β, and e_β, which we'll call `p_a_in`, `e_a_in`, `p_b_in`, and `e_b_in` (Figure 6.13.2). (The syntax `x[1:m]` returns the sub-vector of `x` from index `1` to index `m`, inclusive.)

```
  p_a_in <- a_in[1:(length(a_in) - 1)]
  e_a_in <- a_in[length(a_in)]
  p_b_in <- b_in[1:(length(b_in) - 1)]
  e_b_in <- b_in[length(b_in)]
  # to be continued...
```

Figure 6.13.2: Extracting p_α, e_α, p_β, and e_β from α and β.

Let's revisit our recursive definition from Equation 6.1:

$$\underbrace{\begin{pmatrix} p_\alpha \text{ aligned w/} \\ p_\beta e_\beta \end{pmatrix}}_{left} \begin{matrix} e_\alpha \\ - \end{matrix}, \quad score = S(left) + S(e_\alpha, -)$$

$$\underbrace{\begin{pmatrix} p_\alpha \text{ aligned w/} \\ p_\beta \end{pmatrix}}_{center} \begin{matrix} e_\alpha \\ e_\beta \end{matrix}, \quad score = S(center) + S(e_\alpha, e_\beta)$$

$$\underbrace{\begin{pmatrix} p_\alpha e_\alpha \text{ aligned w/} \\ p_\beta \end{pmatrix}}_{right} \begin{matrix} - \\ e_\beta \end{matrix}, \quad score = S(right) + S(-, e_\beta)$$

We need to produce the sub-alignments `left`, `center`, and `right` recursively. For `left`, for example, we do this by aligning p_α against $p_\beta e_\beta$, which is also the entirety of β. We could do this by simply using `b_in`, or we could concatenate the pieces we've already produced with `c(p_b_in, e_b_in)`. To stick with our notation, we'll choose the latter, even though re-concatenating the vectors is less efficient (Figure 6.13.3).

Figure 6.13.3: Recursive calls for computing *left*, *center*, and *right* sub-alignments.

```
left <- global_aln(p_a_in, c(p_b_in, e_b_in))
center <- global_aln(p_a_in, p_b_in)
right <- global_aln(c(p_a_in, e_a_in), p_b_in)
# to be continued...
```

Just as in the `fib()` function, notice the similarity between our code and the conceptual definition! Now, we can use the answers for these sub-alignments to produce three potential overall answers for the problem, accessing parts of the answers to sub-problems as needed using $-notation (Figure 6.13.4).

Figure 6.13.4: Computing three potential answers to aligning α and β according to the recursive definition.

```
answer_left <- list(a_in = a_in, b_in = b_in,
                    a_out = c(left$a_out, e_a_in),
                    b_out = c(left$b_out, "-"),
                    score = left$score +
                            score_pair(e_a_in, "-"))
answer_center <- list(a_in = a_in, b_in = b_in,
                      a_out = c(center$a_out, e_a_in),
                      b_out = c(center$b_out, e_b_in),
                      score = center$score +
                              score_pair(e_a_in, e_b_in))
answer_right <- list(a_in = a_in, b_in = b_in,
                     a_out = c(right$a_out, "-"),
                     b_out = c(right$b_out, e_b_in),
                     score = right$score +
                             score_pair("-", e_b_in))
# to be continued...
```

All that is left is to determine which of these three (`answer_right`, `answer_center`, or `answer_left`) is the best, defined as having the largest score, and return it. We can do this in a straightforward manner, and of course we need to include the final brace that closes our function (Figure 6.13.5).

```
  best <- answer_left
  best_score <- answer_left$score
  if(answer_center$score > best_score) {
    best <- answer_center
    best_score <- answer_center$score
  }
  if(answer_right$score > best_score) {
    best <- answer_right
    best_score <- answer_right$score
  }

  return(best)
}
```

Figure 6.13.5: Determining and returning the best answer by selecting the one with the largest score.

And that's it! We've turned our recursive definition (which we also proved correct via induction) into a recursive algorithm. (The full function is simply the concatenation of Figures 6.13.1, 6.13.2, 6.13.3, 6.13.4, and 6.13.5.) Figure 6.14 shows a usage attempt, and Figure 6.15 reveals the printed output.

```
a <- char_vec("@TATCGG")
b <- char_vec("@TTCG")
answer <- global_aln(a, b)
str(answer)                        # See margin
print(unvec_char(answer$a_out))    # @TATCGG
print(unvec_char(answer$b_out))    # @T-TCG-
```

Figure 6.14: Using the recursive alignment function.

```
List of 5
 $ a_in : chr [1:7] "@" "T" "A" "T" ...
 $ b_in : chr [1:5] "@" "T" "T" "C" ...
 $ a_out: chr [1:7] "@" "T" "A" "T" ...
 $ b_out: chr [1:7] "@" "T" "-" "T" ...
 $ score: num 0
[1] "@TATCGG"
[1] "@T-TCG-"
```

Figure 6.15: Output for Figure 6.14.

Exercises

1. Implement the code in this chapter, but modify the scoring rules such that A/T and C/G mismatches are scored at -2 rather than -3.

2. What is the maximum depth of the call stack in relation to `length(a_in)` and `length(b_in)` for `global_aln()`? Prove that your answer is correct.

3. Prove the correctness of the recursive definition in equation 6.2, and implement the recursive method it describes. You may need to create an alternate representation as well, since in this case a sequence like `"TAGC"` should be represented the vector `c("T", "A", "G", "C", "@")`.

7 *Fast Alignment, Local Alignment*

> Scientists often have a naïve faith that if only they could discover enough facts about a problem, these facts would somehow arrange themselves in a compelling and true solution.
>
> Theodosius Dobzhansky, 1962

In the previous chapter we developed and proved correct a recursive method for computing what are known as *global* sequence alignments. A global alignment of two sequences requires that all portions of both sequences are used in the alignment–later in this chapter we'll explore other types of alignments that are particularly good for finding matching *subsequences* of two sequences, known as local alignments.

Before we can do that, however, we should evaluate the speed of our algorithm. If we try to use this method to align two longish sequences (as in Figure 7.1), we'll find that we must wait for a surprisingly long time.

```
a <- char_vec("@TATCGGCGATCGATTAGCCC")
b <- char_vec("@TTGGCGATCGACCATCC")
answer <- global_aln(a, b)       # waiting...
```

Figure 7.1: Attempting to use the recursive method from the last chapter on long sequences takes a very long time.

The recursive `global_aln()` function operates not too dissimilarly from the `fib()` function of Chapter 4, except instead of making two recursive calls it makes *three*! And, just like the `fib()` function, the recursive calls only get closer to a base case one step at a time: if α has n characters and β has m, computing just the *center* sub-alignment requires aligning $n-1$ and $m-1$ characters. Contrast this to many of the methods in Chapter 3, where recursive calls were made to subproblems of *half* the original size. Indeed, just as we wrote a recurrence relation for `quicksort()` (page 51), we can write a recurrence relation for `global_aln()`. Let $T(n,m)$ be the time needed to align two sequences of length n and m. Then:

$$T(n,m) = \begin{cases} T(n-1,m) + T(n-1,m-1) + T(n,m-1) + O(n+m), & \text{if } n > 1 \text{ and } m > 1, \\ O(m), \text{ if } n = 1, \\ O(n), \text{ if } m = 1. \end{cases}$$

We won't solve this recurrence exactly, other than to say that just like the basic `fib()` function, the runtime is *exponential* in m and n.[1] But, also like the `fib()` function, many of the recursive calls solve overlapping subproblems. Consider the call tree for an alignment where α = `@TATC` and β = `@TTGT` (Figure 7.2; hopefully it's clear now why we named the subproblems *left*, *center*, and *right*!)

[1] The $O(n+m)$ comes from lines like `a_out = c(center$a_out, e_a_in)`, which requires copying the data from `center$a_out` into a new vector. Even without an $O(n+m)$ term in the recurrence (which can be avoided with the creative use of stacks), the method would still be exponentially slow.

In Figure 7.2 we've highlighted the largest overlapping and redundant subproblems, but there are many in the full tree even for short input sequences. Obviously, memoizing our global alignment function using the techniques of Chapter 4 will speed things up considerably. We'll use as keys into the memoization cache the result of concatenating the `unvec_char()`'d versions of `a_in` and `b_in` (Figure 7.3).

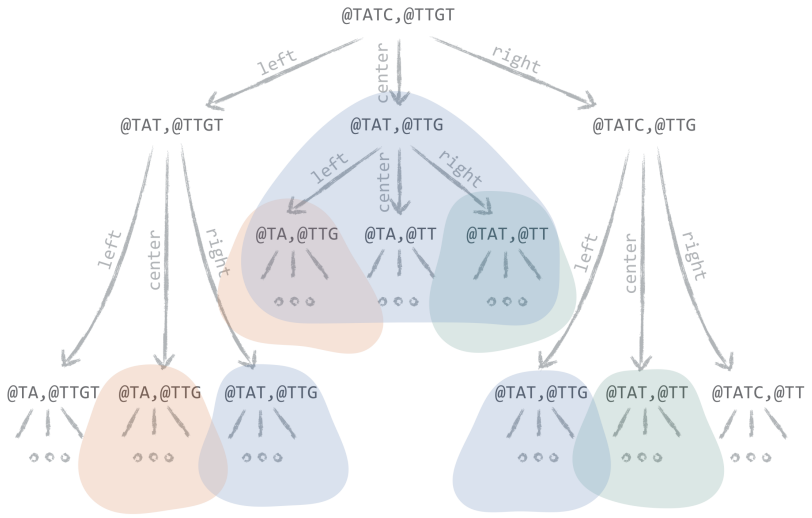


Figure 7.2: Call tree for `global_aln()` on sequences `@TATC` and `@TTGT`, with several redundant computations highlighted.

With memoization our recursive function runs much faster, even for long input sequences. Just *how* fast remains to be seen.

7.1 *Inspecting the Cache*

In an effort to determine how much time and effort the memoized alignment algorithm takes, we can take a detailed look at the cache. After all, each entry in the cache represents a chunk of work performed by some function call. We're going to visualize the contents of the cache using the `ggplot2` R package.

But in order to plot anything with `ggplot2`, the data to plot must exist in a dataframe–a table of columns and rows. As shown in Figure 4.10 on page 60, the contents of an `rstack` can easily be converted to a data frame. Unfortunately, our data exists as entries of a `hash` table, for which no such convenient conversion exists. Our strategy will be to use an `rstack` as an intermediary (Figure 7.4, ignoring the highlighted lines).

As we do this, we'll also (using an internal loop) run `unvec_char()` on any elements of `answer` that are character vectors, effectively converting entries like `a_out` so they are represented as `c("@CATG")` instead of `c("@", "C", "A", "T", "G")` (highlighted lines in Figure

```
GLOBAL_ALN_CACHE <<- hash()

global_aln <- function(a_in, b_in) {
  key <- str_c(unvec_char(a_in), unvec_char(b_in), sep = ",")
  if(has.key(key, GLOBAL_ALN_CACHE)) {
    return(GLOBAL_ALN_CACHE[[key]])
  }

  if(length(a_in) == 1 | length(b_in) == 1) {
    answer <- base_case(a_in, b_in)
    GLOBAL_ALN_CACHE[[key]] <- answer
    return(answer)
  }

  # rest of function...

  GLOBAL_ALN_CACHE[[key]] <<- best
  return(best)
}
```

Figure 7.3: Modifications to memoize the `global_aln()` function.

7.4).

```
hash_values_as_dataframe <- function(thehash) {
  tempstack <- rstack()

  for(key in keys(thehash)) {
    answer <- thehash[[key]]

    for(index in seq(1, length(answer))) {
      if(is.character(answer[[index]])) {
        answer[[index]] <- unvec_char(answer[[index]])
      }
    }

    tempstack <- insert_top(tempstack, answer)
  }

  return(as.data.frame(tempstack, stringsAsFactors = FALSE))
}
```

Figure 7.4: Modifying `hash_values_as_dataframe()` so that any character vectors are run through `unvec_char()` for the resulting data frame.

Of course, if we are interested in inspecting the cache for a given run of `global_aln()`, we should be sure to clear it out before we call `global_aln()` (Figure 7.5).

```
a <- char_vec("@TATCTGCAACGA")
b <- char_vec("@TTGTGC")
GLOBAL_ALN_CACHE <<- hash()
answer <- global_aln(a, b)

cache_df <- hash_values_as_dataframe(GLOBAL_ALN_CACHE)
print(head(cache_df))
```

Figure 7.5: Using `hash_values_to_dataframe()` to summarize the memoization cache. Notice that we reset `GLOBAL_ALN_CACHE` before each call to `global_aln()`.

A beautiful approached for global alignment that uses much less memory ($O(n)$ as opposed to $O(mn)$) is called *Hirschberg's algorithm*. This method fills out the score matrix for the dynamic program and traceback, but does so recursively.[6] The essense of the algorithm lies in identifying–as efficiently (in memory used) as possible–which cells in the middle two rows of the table the traceback path will intersect. This can be done by computing scores (and "from" information) in a row-by-row fashion, but only storing the most recent row in order to compute the next. This proceeds in the forward fashion to the middle row. Next, the same process happens in the reverse starting at the bottom right (using the "alternative formulation" discussed above), and where these two collide provides one small part of the overall traceback path (Figure 7.34, top).

With these neighboring cells identified, the same process can be recursively computed on the upper-left qaudrant of the matrix defined by the cells found, and then again on the lower-right quadrant of the matrix (Figure 7.34, bottom). In this way, at most one or two full rows of information need to stored, even though cell scores will be computed many times. Fortunately, only $1/2$ of the scores will be computed twice, only $1/4$ of the cell scores need to be computed three times, $1/8$ four times, and so on. Because $1 + 1/2 + 1/4 + \cdots \leq 2$, the total runtime of Hirshberg's algorithm is only twice that of normal global alignment (still $O(nm)$) but uses much less memory ($O(n)$, or "linear space").

This is only a sketch of the algorithm; for details we refer the reader to a more comprehensive bioinformatics text such as *An Introduction to Bioinformatics Algorithms* by Neil C. Jones and Pavel A. Pevzner.

[6] Hirschberg's algorithm is particularly fascinating because it recursively solves a dynamic program, itself representing a recursively defined process!

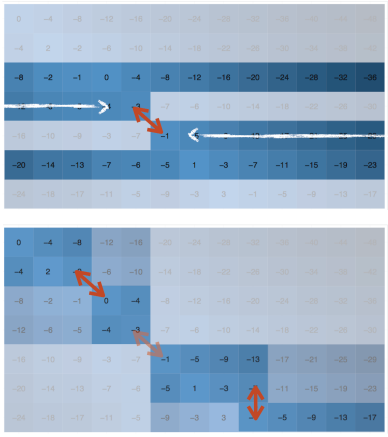


Figure 7.34: Hirschberg's algorithm finds the traceback path in the dynamic programming tables recursively. To start, pairs of rows are scanned from the top and bottom to find a small part of the traceback path near the middle row (top). Then the upper-left quadrant and lower-right quadrants are solved recursively (bottom).

7.6 *Alternative Scoring Rules, Multiple Alignment*

As we saw, only simple modifications to the scoring and traceback rules were needed to convert global alignment into local alignment. Other adjustments can be used to produce other alignment types.

The first modification is for *end-gap-free alignments*–these allow for overhangs at the start or end of α and/or β to be scored as 0, or "free." These are useful when both sequences are sourced from similar template sequences and potentially overlap (Figure 7.35). This is common in genome assemply applications where short fragments are sequenced from a longer chromosome and need to be pieced back together.

To allow for free end-gaps, the dynamic program is adjusted so that scores along the top row and left-hand column are all 0 (rather than increasingly negative gap costs). This allows the traceback to "stop" anywhere along either sequence (and free gaps are assumed

Figure 7.35: In end-gap-free alignment, gaps occurring at the beginning or end of the alignment (shown in purple) do not count against the alignment score, allowing the algorithm to align two sequences drawn from different portions of similar template sequences. The four general types are shown.

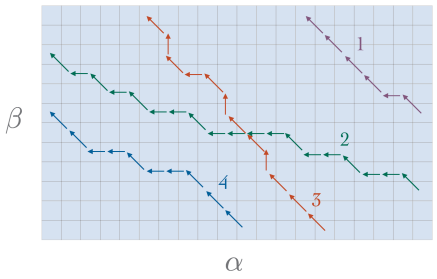


Figure 7.36: Traceback paths for the four potential types of end-gap-free alignments, which start at the highest-scoring cell along the bottom or right, and end at the first cell encountered along the top or left.

to fill out the rest). The traceback starts not in the lower-right corner, but rather at the largest scoring cell along the bottom row or right-hand column, allowing the free gaps at the other end. Figure 7.36 shows the four possible types of traceback paths corresponding to the four types of alignments in Figure 7.35.

A more sophisticated modification provides for *affine-gap* alignments, which consider a tricky biological reality that not all gaps are created equal. In an evolutionary tree (see Figure 6.2, page 83), once a gap has been introduced by the loss of a DNA base or addition of a new base, it can be more likely that future losses or additions may occur at the same location.[7] To account for this, affine-gap scoring associates a large cost with the first in a series of gaps, but smaller costs with subsequent gaps (Figure 7.37).

There are at least a couple of ways to implement affine-gap penalties. The easiest (and slowest) extends the idea of "from" arrows so that an `"up"` arrow (or `"left"` as the case may be) might be longer than a single cell, with the associated cost computed as desired from the length (Figure 7.38). This means, however, that when filling out the score table each cell depends on not just three neighboring subproblems, but many. A more efficient solution makes use of multiple scoring tables with "from" arrows that might move between them. For more information on this topic we again refer to *An Introduction to Bioinformatics Algorithms* by Jones and Pevzner.

[7] The particulars of this are best left to advanced courses in evolutionary biology. In some cases repeated additions or deletions may not be causal, but an artifact of the non-uniform nature of mutation. In other cases a single modification could increase the likelihood of future mutations. For example, if a removal causes a gene sequence to become non-functional (and this doesn't cause the lineage to go extinct), then future mutations to the same region carry less risk of ill effects.

Figure 7.37: Affine-gap alignments score gaps in a series differently that standard alignment; usually the first gap incurs a severe penalty, and subsequent gaps less so to account for evolutionary patterns.

For this chapter and the last, we've considered aligning only two sequences, α and β. The idea can be extended to three sequences; consider the seven possible ending configurations for sequences α, β, and δ. When computing a multi-way aligment, we consider all combinations for scoring. (An alignment of `"T"` with `"A"` with `"-"`, for example, incurs the `T/A` cost, the `A/-` cost, and the `-/T` cost.)

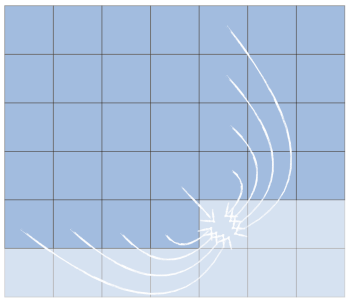


Figure 7.38: The simplest way to implement affine-gap alignment costs is to consider different-length "jumps" for each subproblem, with each having a different cost associated as desired.

$$\underbrace{\begin{pmatrix} p_\alpha e_\alpha \text{ aligned w/} \\ p_\beta e_\beta \text{ aligned w/} \\ p_\delta \end{pmatrix}}_{aln_1} \begin{matrix} - \\ - \\ e_\delta \end{matrix}, \; score = S(aln_1) + S(-,-) + S(-,e_\delta) + S(e_\delta,-)$$

$$\underbrace{\begin{pmatrix} p_\alpha \text{ aligned w/} \\ p_\beta e_\beta \text{ aligned w/} \\ p_\delta e_\delta \end{pmatrix}}_{aln_2} \begin{matrix} e_\alpha \\ - \\ - \end{matrix}, \; score = S(aln_2) + S(e_\alpha,-) + S(-,-) + S(-,e_\alpha)$$

$$\underbrace{\begin{pmatrix} p_\alpha e_\alpha \text{ aligned w/} \\ p_\beta \text{ aligned w/} \\ p_\delta e_\delta \end{pmatrix}}_{aln_3} \begin{matrix} - \\ e_\beta \\ - \end{matrix}, \; score = S(aln_3) + S(-,e_\beta) + S(e_\beta,-) + S(-,-)$$

$$\underbrace{\begin{pmatrix} p_\alpha e_\alpha \text{ aligned w/} \\ p_\beta \text{ aligned w/} \\ p_\delta \end{pmatrix}}_{aln_4} \begin{matrix} - \\ e_\beta, \\ e_\delta \end{matrix} \; score = S(aln_4) + S(-, e_\beta) + S(e_\beta, e_\delta) + S(e_\delta, -)$$

$$\underbrace{\begin{pmatrix} p_\alpha \text{ aligned w/} \\ p_\beta e_\beta \text{ aligned w/} \\ p_\delta \end{pmatrix}}_{aln_5} \begin{matrix} e_\alpha \\ -, \\ e_\delta \end{matrix} \; score = S(aln_5) + S(e_\alpha, -) + S(-, e_\delta) + S(e_\delta, e_\alpha)$$

$$\underbrace{\begin{pmatrix} p_\alpha \text{ aligned w/} \\ p_\beta \text{ aligned w/} \\ p_\delta e_\beta \end{pmatrix}}_{aln_6} \begin{matrix} e_\alpha \\ e_\beta, \\ - \end{matrix} \; score = S(aln_6) + S(e_\alpha, e_\beta) + S(e_\beta, -) + S(-, e_\alpha)$$

$$\underbrace{\begin{pmatrix} p_\alpha \text{ aligned w/} \\ p_\beta \text{ aligned w/} \\ p_\delta \end{pmatrix}}_{aln_7} \begin{matrix} e_\alpha \\ e_\beta, \\ e_\delta \end{matrix} \; score = S(aln_7) + S(e_\alpha, e_\beta) + S(e_\beta, e_\delta) + S(e_\delta, e_\alpha)$$

Since there are 3 input sequences, there are $2^3 - 1$ ending configurations to consider–2 choices for how we end the α portion (e_α and $-$), times 2 for β, times 2 for δ, minus the obviously unnecessary `-/-/-` option. A memoized, recursive solution would follow much the same strategy as in the last chapter. Similarly, a dynamic program would follow in the footsteps of this chapter: these seven options can be cast as potential "from" arrows in a 3-dimensional table! (Figure 7.39.) Although more difficult to visualize, aligning four sequences can be accomplished with 4-dimensional matrices, each cell of which depends on $2^4 - 1$ neighboring cells.[8]

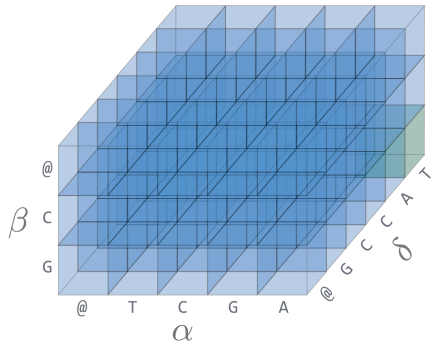


Figure 7.39: The dynamic programming solution for three-sequence multiple alignment would utilize 3-dimensional `score` and `from` matrices. The global alignment would start the traceback from the green cell; neighboring solutions are represented by the seven cells touching it.

While such *multiple alignments* allow biologists to compare many sequences simultaneously, in practice this method is far too slow. If α has n characters, β has m, and δ has l, the time needed to fill out the 3-dimensional table is $O(nml)$. Adding a fourth sequence of length k increases the runtime to $O(nmlk)$. In general, the time necessary grows explonentially with the number of input sequences. Real multiple-alignment programs turn away from this optimal approach and again turn to much faster (but not gauranteed to produce score-maximizing alignment) heuristics.

[8] I find it interesting that "base cases" of higher-dimensional alignments are instances of lower-dimensional alignments. For example, any subproblem along a "wall" of the 3-dimensional table is an instance of 2-dimensional alignment, with the third sequence filled out by gaps. From this, it is possible to design a general n-way dynamic program that relies on itself (recursively) to compute solutions for lower-dimensional walls in dimension $n - 1$.

Exercises

1. Try implementing end-gap-free alignment.
2. Try implementing affine-gap alignment with the naïve algorithm described.
3. Try implementing 3-sequence global alignment, using the algorithm for 2-sequence alignment if one of the three inputs is just `"@"`.

8 *Hidden Markov Models*

> On two occasions I have been asked, 'Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?' I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.
>
> Charles Babbage, 1864

Many topics in computer science constitute a sort of "algorithmic forensics:" given a set of observations, what happened to produce them? In some cases, we have a good idea of the underlying "model" and our goal is to figure out how our observations fit into the model. Hidden Markov Models (HMMs, after late 19^{th} century Russian mathematician Andrey Markov) are a common example.

Suppose, for the moment, that we are security guards working deep underground for months at a time at the NORAD compound (North American Aerospace Defense Command), inside Cheyenne Mountain in Colorado. As amateur meteorologists, based on historical newspapers we've built a simple weather model: the weather tomorrow will most likely be like the weather today. To quantify this, if any given day is sunny then it will be sunny the next day with 75% probability and rainy with the remaining 25% probability. On the other hand, if it is rainy, then the next day will be rainy 70% of the time and sunny 30% of the time. These two *states*–sunny (s) and rainy (r)–represent our model of the world, and our model satisfies the *Markov property*: that the probability of moving to a state depends only on the immediately previous state. After five rainy days the sixth will be rainy with 70% probability; similarly after ten rainy days the eleventh will also be rainy with 70% probability.

But this is not the entire story. As mere security guards stuck inside, we can't actually observe the weather.[1] Fortunately, our friend the General lives outside the base, and every day he comes to work passing by our security station either carrying an umbrella,

[1] We should probably pick a different hobby. Programming, maybe.

or not. From talking to his friends, we know that he tends to carry an umbrella on rainy days, but he forgets it on about 20% of those rainy days. Similarly, he usually doesn't carry it on sunny days, but brings it just in case 10% of the time. All of this information together represents our Hidden Markov Model, where "umbrella" (`u`) and "no umbrella" (`n`) are possible observations influenced by the (hidden from us) states 8.1.

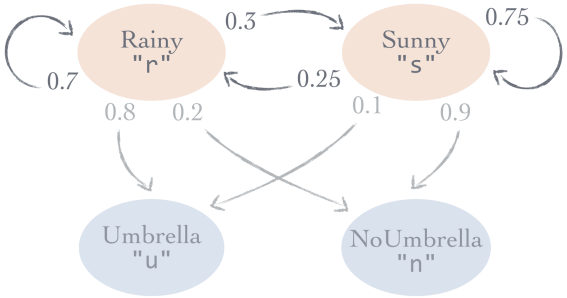


Figure 8.1: Graphical representation for the rainy/sunny Hidden Markov Model, with states shown in pink, and possible observations shown in blue.

Given a model like this, there are a number of questions we might ask. Supposing we start on a sunny day, what is the probability it will be rainy exactly 10 days later? What is the probability that we'll see an umbrella on day 20? In the long term, what percentage of days will be sunny?

All of these questions can be answered using the many beautiful mathematical theories developed for Hidden Markov Models.[2] For a practical perspective, we'll focus in this chapter on a single question: given a sequence of Umbrella/No Umbrella observations like `n, n, n, u, n, u, u, u, u, n, u, n, n, n`, what is the *most likely* sequence of underlying states? (More plainly: what was most probably the weather on those days?)

This is a well-formed question. Consider a shorter sequence like `n, u`. The likelihood that the underlying states were `s, s` is [probability of `n` being observed from state `s`] $\times$ [probability that state `s` transitions to state `s`] $\times$ [probability of `u` being observed from state `s`], or $(0.9)(0.75)(0.1) = 0.0675$. A similar calculation shows that the likelihood of `s, r` is $(0.9)(0.25)(0.8) = 0.18$.[3] Given an observation sequence, we simply want to find a state sequence maximizing this likelihood.

But this is also not an easy question. One possible strategy for finding the most likely state sequence is to consider *all* possible state sequences, compute the likelihood for each as described above, and keep the most likely one. However, the number of possible state sequences is $(\text{number of states})^{(\text{number of observations})}$ (two possible states for day 1, times two for day 2, and so on). If our sequence was 20 days long, there would be 1,048,576 possible state sequences to check; at 30 days, there are over a billion! Clearly we need a better strategy.

HIDDEN MARKOV MODELS can be used to represent a huge variety of processes. Those involving chance immediately come to mind, such as measurements made with scientific instruments prone to error. Many games like poker and roulette can be modeled probabilistically, though they may or may not have a "hidden" component.

[2] The excellent book *Probability and Computing* by Mitzenmacher and Upfal covers a variety of these results.

[3] Although "likelihood" and "probability" are closely related, they are not quite the same thing. Probability measures the chance of an outcome from a random process, while likelihood measures the probability that a hypothesized model produced some outcome (for discrete situations at least, such as our weather model).

HMMs have also been successfully applied to processes that we don't usually think of as random, such as text-to-speech applications where the observations are sound waves, and the hidden states are words or letter combinations. In the realm of bioinformatics, HMMs are commonly used to identify regions of DNA with specific properties.

The "central dogma" of molecular biology holds that "gene" regions of DNA sequences are transcribed into messenger RNA (potentially with some subsections, called introns, removed). These are then translated in three-base chunks into an amino acid sequence that folds in 3 dimensions to become a building block of life: a protein (Figure 8.2).

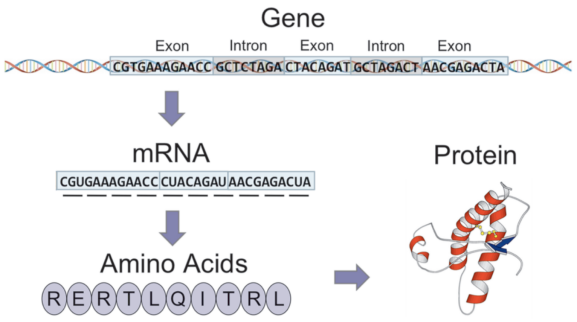


Figure 8.2: The "central dogma" of molecular biology describes the transcription/translation process whereby genic regions in the DNA are turned into functional proteins.

In many species, the overall makeup of the DNA is biased, for example consisting of 80% As and Ts and only 20% Cs and Gs. Furthermore, there are often patterns and trends that distinguish gene sequences from the surrounding DNA. In real gene finding applications, these patterns and the models that represent them are amazingly complex and are generated by analyzing many sequences in related species.

For this discussion we'll assume an extremely simple model: a genome that is 80% As and Ts in non-gene ("n") regions, and 25% As, Ts, Cs, and Gs in gene ("g") regions. Further, we'll assume that the genome starts in a non-gene region, and that each position can transition from a non-gene position to a gene position with 1% probability, and gene positions can transition to non-gene positions with 5% probability (Figure 8.3).

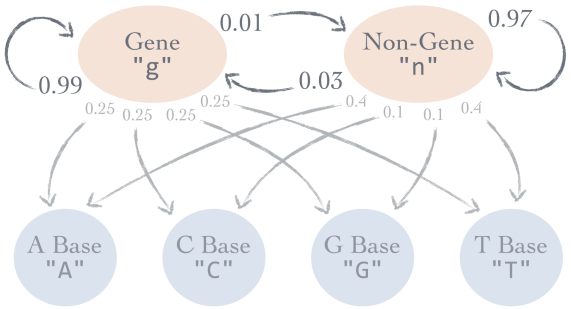


Figure 8.3: Graphical representation for the gene/non-gene Hidden Markov Model.

8.1 *Generating Sequences*

How might we represent a Hidden Markov Model like the above in code? There are a number of possibilities. We might store the state transition probabilities in a hash table, with a key for each state and the corresponding probabilities as values. Alternatively we could could use a matrix, with each row/column combination representing a particular transition probability. Since R supports matrices natively, we'll use this representation (Figure 8.4).

Figure 8.4: Representing state transition probabilities with an R matrix, including named rows and columns.

```
trans_probs <- matrix(nrow = 2, ncol = 2,
                      dimnames = list(c("g", "n"), c("g", "n")))

trans_probs["g", "g"] <- 0.97
trans_probs["g", "n"] <- 0.03
trans_probs["n", "n"] <- 0.99
trans_probs["n", "g"] <- 0.01

print(trans_probs)
```

```
     g    n
g 0.97 0.03
n 0.01 0.99
```

Figure 8.5: Output for Figure 8.4, representing transition probabilities (from row to column).

The code and printed output (Figure 8.5) reveal that we can name the rows and columns, and access individual entries by name, as in `trans_probs["g", "n"]`, storing the probability of transitioning from a gene to a non-gene state. Similarly, we'll encode the observation probabilities in a two by four matrix (Figure 8.6, output in Figure 8.7).

Figure 8.6: Representing observation probabilities.

```
obs_probs <- matrix(nrow = 2, ncol = 4,
                    dimnames = list(c("g", "n"),
                                    c("A", "C", "G", "T")))
obs_probs["g", "A"] <- 0.25
obs_probs["g", "C"] <- 0.25
obs_probs["g", "G"] <- 0.25
obs_probs["g", "T"] <- 0.25
obs_probs["n", "A"] <- 0.4
obs_probs["n", "C"] <- 0.1
obs_probs["n", "G"] <- 0.1
obs_probs["n", "T"] <- 0.4

print(obs_probs)
```

```
     A    C    G    T
g 0.25 0.25 0.25 0.25
n 0.40 0.10 0.10 0.40
```

Figure 8.7: Output for Figure 8.6, representing observation probabilities.

R provides convenient syntax for working with named matrices. For example, `probs <- trans_probs["g", ]` assigns to `probs` the `"g"` row as a named vector; `names(probs)` thus returns the character vector `c("g", "n")` while `probs` itself contains the numeric vector `c(0.97, 0.03)`. The `sample()` function returns a random sample from a vector; it takes an optional `size` parameter specifying the sample size, and a `prob` parameter specifying the sampling distribution vector. Thus, `sample(names(probs), size = 1, prob = probs)` returns `"g"` with 97% probability and `"n"` with 3% probability (representing a random transition from the original `"g"`). We can use these to write a function `random_next_state()` that, given the `trans_probs` matrix and a "current" state, randomly selects a new state and returns it. A very similar function `random_obs()` generates a random observation from a given state 8.8).

With these functions in hand, we can write another that generates a random sequence of states and corresponding outputs. The function will need both the `trans_probs` and `obs_probs` matrices,

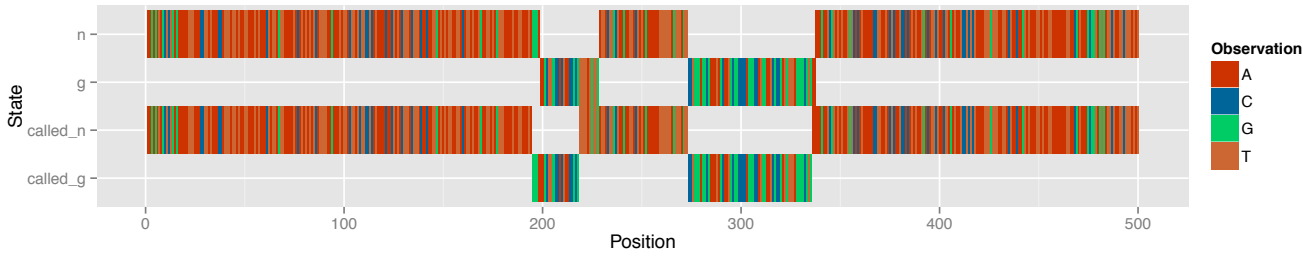


Figure 8.23: Plotted output for Figure 8.22.

```
    pos state obs   called
500   1     n   A called_n
499   2     n   A called_n
498   3     n   A called_n
497   4     n   T called_n
496   5     n   A called_n
495   6     n   C called_n
```

Figure 8.24: Printed output for Figure 8.22.

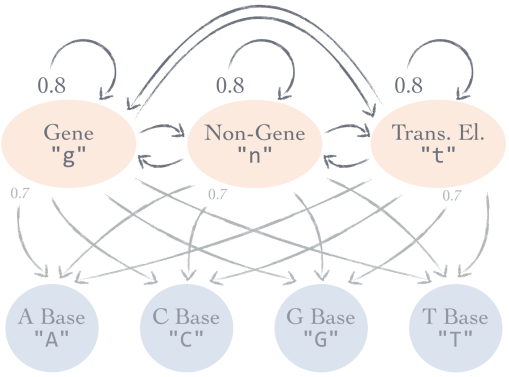


Figure 8.25: Graphical representation for a three-state Hidden Markov Model; all unlabeled edges have weight of 0.1.

Figure 8.26: Generating a moderate-size observation sequence for the three-state model.

8.4 *Sub-problems and Dynamic Programming*

As with sequence alignment, even though the memoized recursive solution is fast (compared to a non-memoized solution), solving large instances is still impractical due to limitations of the call stack (see page 64). Fortunately, finding most-likely state sequences lends itself well to a dynamic-programming, table-based solution. (As mentioned earlier, the Viterbi algorithm was originally developed as a dynamic program.)

In order to inspect the relationships among sub-problems, we'll plot the sub-problems on a grid as we did for sequence alignment in Figure 7.8 on page 98. To get a better sense of the process, we'll consider a more complex model with another state possibility, `"t"` for "transposable element." In real genomes, transposable elements are regions of the DNA prone to being replicated in other parts of the genome. For our test we'll represent genic regions as being rich in As, non-gene regions as being rich in Cs, and transposable elements as being rich in Gs. Further, all three states will only transition to another state with 10% probability (Figure 8.25).

After generating transition probability and observation matrices `trans_probs2` and `obs_probs2` (code not shown), we generate a sequence of 50 observations (Figure 8.26).

However, before running `decode()` we need to modify the function so that the answer lists generated store which sub-answer produced the most likely overall answer. This "from" information consists of the sub-answer's input (`obs_vec` and `end_state`). For base cases these will be set to `NA`. These modifications are found in Figures 8.27.1 and 8.27.2.

```
seq_df2 <- generate_seq(trans_probs2, obs_probs2, "n", 50)
```

Now we can this modified `decode()` function to compute the most likely state sequence for the given observation sequence, `seq_df2$obs`. Rather than consider all three possible answers and

```
  # ...
  # Base case:
  if(length(obs_vec) == 1) {
    answer <- list(state_vec = end_state,
                   lhood = log(obs_probs[end_state, obs_vec]),
                   obs_vec = obs_vec,
                   end_state = end_state,
                   from_obs_vec = NA,
                   from_end_state = NA)
    return(answer)
  }
  # ...
```

Figure 8.27.1: Modifications to Figure 8.20.1 for keeping track of "from" information in base-case answers.

```
  # ...
  for(s in possible_states) {
    subanswer <- decode(trans_probs, obs_probs, obs_subvec, s)

    answer <- list(state_vec = c(subanswer$state_vec, end_state),
                   lhood = subanswer$lhood +
                     log(trans_probs[s, end_state]) +
                     log(obs_probs[end_state, last_obs]),
                   obs_vec = obs_vec,
                   end_state = end_state,
                   from_obs_vec = subanswer$obs_vec,
                   from_end_state = subanswer$end_state)
    # ...
```

Figure 8.27.2: Modifications to Figure 8.20.2 for keeping track of "from" information in recursive-case answers.

checking for the most likely, we'll compute just `answer_n`. Further, we'll assume that `VITERBI_CACHE` is the name of the memoization cache, and use the `hash_values_as_dataframe()` function (page 97) to convert the cache into a data frame for plotting (Figure 8.28).

```
DECODE_CACHE <<- hash()   # memoization cache
answer_n <- decode(trans_probs2, obs_probs2, seq_df2$obs, "n")

cache_df <- hash_values_as_dataframe(DECODE_CACHE)
print(tail(cache_df, n = 20))
```

Figure 8.28: Converting the memoization cache for `decode()` into a data frame.

Using `ggplot2`, we can then plot a cell for each subproblem solved, colored by the best log-likelihood computed for that subproblem. We organize the cells along the x-axis according to the length of the observation sequence solved for, and we also draw "from" arrows, indicating which state transitions proved most likely (Figure 8.30).

The resulting plot shows each desired ending state on the y-axis and each observation sequence solved for on the x-axis (Figure 8.31). We've also annotated the plot with the true state sequence and the called state sequence contained in `answer_n`.

```
    state_vec      lhood obs_vec end_state from_obs_vec from_end_state
129   nnnpppn -9.9403791 CCCGGGG         n       CCCGGG              p
130   nnnpppg -9.9403791 CCCGGGG         g       CCCGGG              p
131    nnnppp -5.3352090  CCCGGG         p        CCCGG              p
132    nnnnnn -9.0934979  CCCGGG         n        CCCGG              n
133    nnnppg -9.3605607  CCCGGG         g        CCCGG              p
134     nnnpp -4.7553905   CCCGG         p         CCCG              p
135     nnnnn -6.5677692   CCCGG         n         CCCG              n
136     nnnng -8.6472108   CCCGG         g         CCCG              n
137      nnnp -4.1755720    CCCG         p          CCC              n
138      nnnn -4.0420406    CCCG         n          CCC              n
139      nnng -6.1214821    CCCG         g          CCC              n
140       nnp -5.5416636     CCC         p           CC              n
141       nnn -1.5163119     CCC         n           CC              n
142       nng -5.5416636     CCC         g           CC              n
143        pp -4.8283137      CC         p            C              p
144        nn -0.9364934      CC         n            C              n
145        gg -4.8283137      CC         g            C              g
146         p -2.3025851       C         p         <NA>           <NA>
147         n -0.3566749       C         n         <NA>           <NA>
148         g -2.3025851       C         g         <NA>           <NA>
```

Figure 8.29: Printed output for Figure 8.28.

Figure 8.30: Using `ggplot2` to plot the memoization cache contents, with sub-problems organized by observation sequence length.

```
p <- ggplot(cache_df) +
  geom_tile(aes(x = reorder(obs_vec, nchar(obs_vec)),
                y = end_state, fill = lhood)) +
  geom_segment(aes(x = obs_vec, y = end_state,
                   xend = from_obs_vec, yend = from_end_state),
               arrow = arrow(length = unit(0.2,"cm")),
               position = position_jitter(width = 0.1,
                                          height = 0.1),
               color = "red") +
  scale_y_discrete(name = "State") +
  theme_bw(14) +
  theme(axis.text.x = element_text(angle = 25, hjust = 1))

plot(p)
```

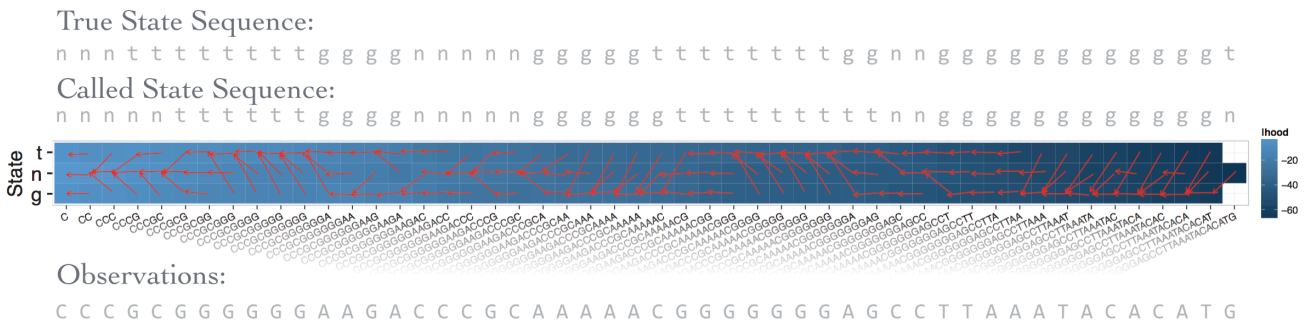


Figure 8.31: Plot inspecting the contents of the memoization cache for `decode()`. Also shown are the true and called state sequences (top) and the observation sequence (bottom). Values along the x-axis are observation sequences considered in the solution for each sub-problem cell.

Based on this visualization, it is clear that the number of unique sub-problems solved is $O(nk)$, where n is the length of the observation sequence and k is the number of states in the model. However, the total amount of *work* performed by `decode()` is actually much larger, even for the memoized version. This is because each sub-problem requires considering each possible "from" state in the for-loop, of which there are $O(k)$ (which is to say, for each red arrow visualized in Figure 8.31, k arrows were computed and considered). Thus, the total runtime for the memoized solution is $O(nk^2)$, as will be the runtime for a dynamic programming solution.

Exercises

1. Our gene/non-gene model assumes that "gene" and "non-gene" are equally likely first states, even though the base-case (Figure 8.13.2) considers the relative probability of observations. (To see this clearly, consider what would happen if there was an observation that could be produced by either with equal probability.) In reality, it is extremely unlikely that the first base in a genome is part of a gene.

One easy way to work around this is to introduce an additional "start" state into the model, such that it is only possible to move out of the start state into other states, and the start state only produces a special observation that can be placed at the beginning of an observation sequence (Figure 8.32).

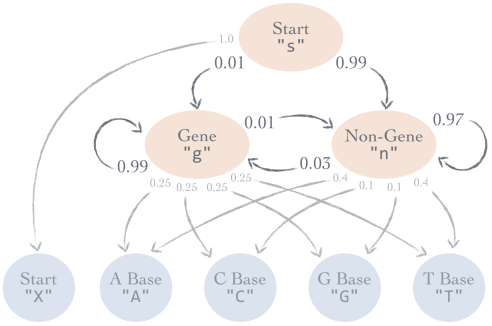


Figure 8.32: A gene/non-gene Hidden Markov Model with an explicit starting state.

Code an HMM with such a start state, and generate a random sequence of states and observations starting from it. Try decoding the observation sequence.

2. Visualize the non-memoized call tree for `decode(trans_probs, obs_probs, obs_vec, end_state)`, with `obs_vec <- c("C", "A", "T")` and `end_state <- "g"`. (You only need to visualize the contents of `obs_vec` and `end_state` at each node, since `trans_probs` and `obs_probs` don't change.)

3. Implement a memoized version of the `decode()` function, using information from `obs_vec` and `end_state` for the memoization cache keys. (The `trans_probs` and `obs_probs` parameters don't need to be part of the keys, since they don't change.)

4. Implement the dynamic-programming Viterbi algorithm for decoding. As hinted at by Figure 8.31 (and the techniques of Chapter 7), you'll need to produce a table with a column for each observation and a row for each state. A for-loop will be required in computing the log-likelihood for each cell, considering potential log-likelihoods based on all the cells in the column to the left.

   Once the log-likelihood and "from" tables are complete, computing the most-likely sequence as a traceback is straightforward.

5. Generate a formal proof of correctness for the recursive `decode()` function. You will likely need to use an inductive proof similar in structure to that used for global sequence alignment (Chapter 6).

9 *Turtle Drawing, L-Systems*

> The programmer, like the poet, works only slightly removed from pure thought-stuff. He builds his castles in the air, from air, creating by exertion of the imagination. Few media of creation are so flexible, so easy to polish and rework, so readily capable of realizing grand conceptual structures...
>
> Fred Brooks, 1975

Nearly all of the techniques covered in this book have a self-similar beauty to them, and the many figures we've seen show that these ideas can be visualized graphically. The goal of this chapter is to more explicitly explore the idea of *drawing* via recursive and self-similar processes, and as a result will be more whimsical than practical. First though, we will need a way to draw simple lines and shapes programmatically. As it turns out, this functionality will be provided by a turtle–a virtual turtle–provided by the R package `TurtleGraphics`.

After loading the library (along with others, see page 23), we can run `turtle_init(mode = "clip")` to initiate and display our turtle in the center of a 100 by 100 "terrarium" (display box). In this box, the lower-left hand corner is at coordinate $(0, 0)$. The `turtle_getpos()` function returns the turtle's current position as a length-2 vector in x, y coordinates. Similarly, `turtle_getangle()` returns a length-1 vector of his current angle (where 0 is up, -90 is left, 90 is right and 180 is down; Figure 9.2).

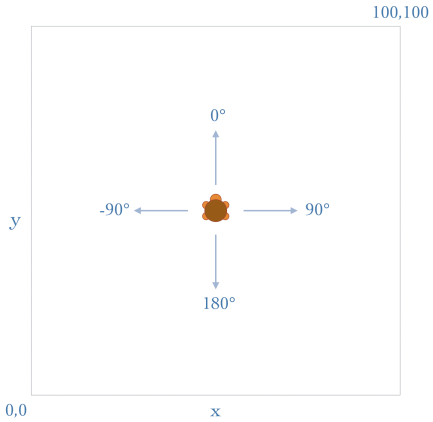


Figure 9.1: The turtle's "terrarium" after calling `turtle_init()` is 100 units on the x (horizontal) axis and 100 units on the y (vertical), with $0, 0$ in the lower-left and $100, 100$ in the upper-right.

```
turtle_init(mode = "clip")
print(turtle_getpos())    # 50 50
print(turtle_getangle())  # 0
```

Figure 9.2: Initiating a new turtle and printing his initial location and angle.

Our turtle has a pen, which by default is "down" on the paper. We can tell him to move forward 10 units by calling `turtle_forward(10)`, causing him to move and draw a line with his pen. We could then

call `turtle_up()` and `turtle_forward(10)` to have him lift his pen and then move forward, followed by `turtle_down()` and another `turtle_forward(10)` to have him to put his pen down and move again while drawing.

Figure 9.3: Basic moves for turtle-based drawing.

```
turtle_init(mode = "clip")    # initiate turtle at 50,50

turtle_setpos(50, 20)  # set x,y location
turtle_forward(10)     # move forward 10
turtle_up()            # lift drawing pen
turtle_forward(10)     # move forward 10
turtle_down()          # set drawing pen down
turtle_lwd(3)          # set pen size to 3.0
turtle_forward(10)     # move forward 10
turtle_right(45)       # turn 45 degrees right
turtle_col("red")      # set pen color to red
turtle_backward(20)    # move backward 20
```

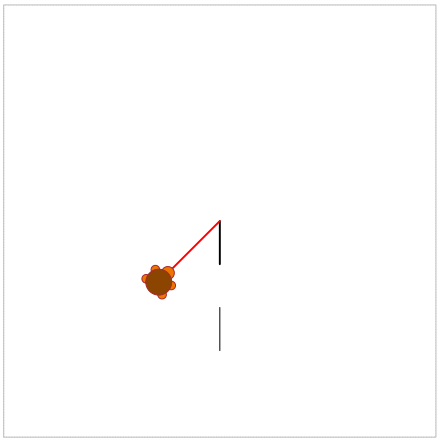


Figure 9.4: Displayed output for Figure 9.3.

The `mode = "clip"` in the `turtle_init()` function controls what happens if the turtle moves outside the 100×100 window: by default an error occurs, using `mode = "clip"` lets him simply move outside the box but we won't see any drawing. Figure 9.3 summarizes these basic moves and a few more; the output is shown in Figure 9.4.

Unfortunately, after every instruction, the turtle himself needs to be redrawn (at least in the R `TurtleGraphics` package). Since we will be drawing many lines (in loops and with functions), this dramatically slows down the drawing process. So although the turtle is cute, after running `turtle_init(mode = "clip")` we'll generally then run `turtle_hide()` to indicate that we aren't interested in seeing the turtle himself. When we hide the turtle we can efficiently create interesting images, such as a spiral produced by a loop (Figures 9.5 and 9.6).

Figure 9.5: If we are willing to "hide" the turtle himself, we can efficiently produce more complex drawings.

```
turtle_init(mode = "clip")
turtle_hide()

for(dist in seq(1,20)) {
  turtle_forward(dist)
  turtle_left(30)
}
```

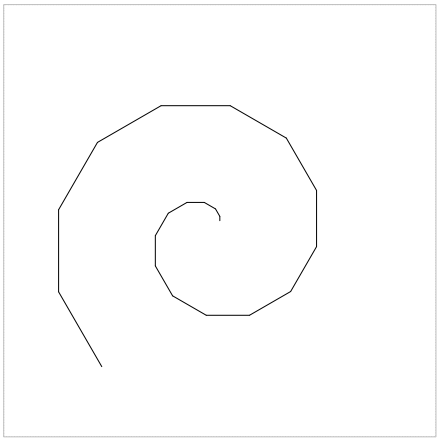


Figure 9.6: Displayed output for Figure 9.5.

Finally, we're also going to write three helper functions. The first, `turtle_getstate()` will return the turtle's current x and y location as well as his angle as a 3-element vector. Conversely, `turtle_setstate()` will take such a 3-element vector and set the turtle's current state to that position and angle. The `TurtleGraphics` package provides no support for having the turtle draw text, so we'll include a custom `turtle_text()` function that takes a length-1 character vector and plots it at the turtle's current location and

angle. This function makes use of the same underlying graphical system used by the `TurtleGraphics` package (Figures 9.7 and 9.8).

```
turtle_getstate <- function() {
  state <- c(turtle_getpos(), turtle_getangle())
  return(state)
}

turtle_setstate <- function(state) {
  turtle_setpos(state[1], state[2])
  turtle_setangle(state[3])
}

turtle_text <- function(label, col = "black", fontsize = 20) {
  grid.text(label,
            turtle_getpos()[1],
            turtle_getpos()[2],
            rot = -1*turtle_getangle(),
            default.units = "native",
            gp = gpar(fontsize = fontsize, col = col))
}

# Example usage:
turtle_init(mode = "clip")
turtle_hide()

start_state <- turtle_getstate()
turtle_forward(20)
turtle_text("After move")
turtle_setstate(start_state)
turtle_text("Back at start")
```

Figure 9.7: Helpful functions for retrieving and setting the turtle's state, as well as drawing text.

Figure 9.8: Displayed output for Figure 9.7.

9.1 *Graphical Recursion*

Turtle graphics are too simple to be used for many applications, like designing user interfaces. On the other hand, they are an excellent tool for visualizing computational processes. Let's start by writing a function that draws a "tree" of a given size (taken as a parameter), where the tree has only two branches, and the turtle is returned to the starting position before the function ends. The overall strategy will be to store the current state, move forward by the size given, turn left, move forward by some fraction of the size given, return to the branching point (without drawing), turn right, move forward again by some fraction of the size given, and finally return to the stored state (Figures 9.9 and 9.10).

We can use this function to draw two simple trees, one in the lower-left with size of 20, and one in the lower-right with size of 0.5 (Figure 9.11).

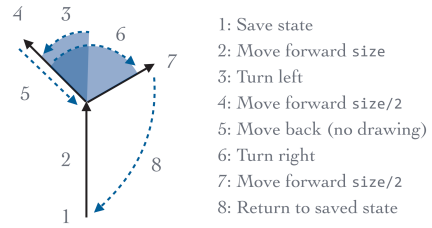


Figure 9.9: Turtle moves for drawing a simple tree with two branches.

Figure 9.52: Generating a complex L-System sentence and interpreting it with the `draw_sentence()` function.

```
rules <- hash()
rules[["F"]] <- char_vec("F[-F]F[+F]F")

sentence <- c("F")
sentence <- lproduce(sentence, rules)
sentence <- lproduce(sentence, rules)
print(unvec_char(sentence))

turtle_init(mode = "clip")
turtle_hide()

turtle_setstate(c(50, 20, 0))
draw_sentence(sentence, 6, 30, 30)
```

```
[1] "F[-F]F[+F]F[-F[-F]F[+F]F]F[-F]F[+F]F
[+F[-F]F[+F]F]F[-F]F[+F]F"
```

Figure 9.53: Printed output for Figure 9.52, showing the sentence used to draw Figure 9.54.

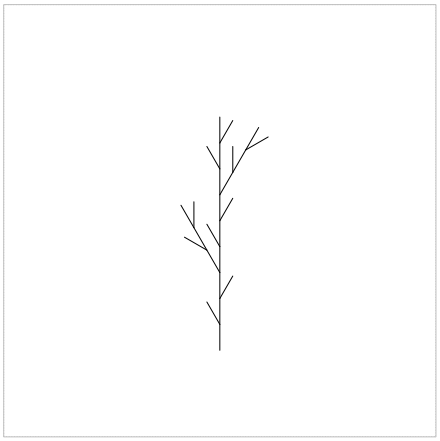


Figure 9.54: Displayed output for Figure 9.52.

Perhaps the most remarkable property of L-Systems is their flexibility: they provide almost endless opportunities for creativity. We could easily envision using random choices for sizes or angles (like `runif(1, min = size / 2, max = size * 2)`); these types of modifications often produce more life-like images.

For another example, rather than draw the entire structure smaller to fit all of the `F` moves in the turtle's drawing window, we could specify that each jump into a substructure (indicated by an addition to the state stack with a `[` character) could reduce the current `size` by 1/3, while each jump out (removal with a `]` character) reverses the adjustment. Other potential strategies include reducing the size at each `F` symbol encountered, setting the size according to the current size of the state stack, or increasing the symbol set with explicit "smaller" and "larger" symbols.

```
    # ...
    } else if(symbol == "]") {
      turtle_setstate(peek_top(pos_stack))
      pos_stack <- without_top(pos_stack)
    } else if(symbol == "s") {
      size <- size * (2/3)
    } else if(symbol == "l") {
      size <- size / (2/3)
    } else {
    # ...
```

Figure 9.55: Potential adjustments for the `draw_sentence()` function (Figure 9.48), incorporating specific symbols for "smaller" and "larger."

Figures 9.55, 9.56 and 9.57 show an example of the latter, a modification of the earlier L-System where `s` is interpreted as `size <- size * (2/3)` and `l` is interpreted as `size <- size / (2/3)` in the `draw_sentence()` function. (Note that since we've specified no replacement rules for `s` and `l`, the `lproduce()` function simply copies existing instances of them into each successive generation.)

```
rules <- hash()
rules[["F"]] <- char_vec("Fs[l-Fs]F[l+Fs]Fl")

sentence <- c("F")
for(i in seq(1,4)) {
  sentence <- lproduce(sentence, rules)
}

turtle_init(mode = "clip")
turtle_hide()

turtle_setstate(c(50, 10, 0))
draw_sentence(sentence, 2.5, 30, 30)
```

Figure 9.56: Drawing a more sophisticated L-System including s and l symbols interpreted as in Figure 9.55.

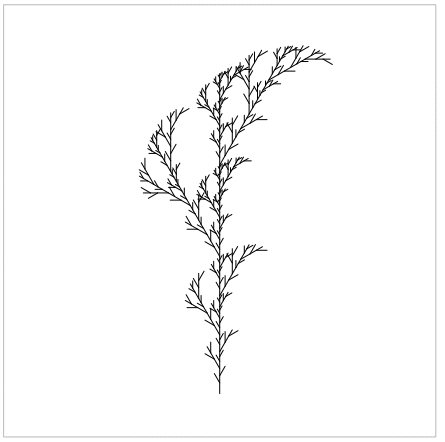


Figure 9.57: Displayed output for Figure 9.56.

```
[1] "Fs[l-Fs]F[l+Fs]Fls[l-Fs[l-Fs]F[l+Fs]Fls]Fs[l-Fs]F[l+Fs]Fl[l+Fs[l-Fs]F
[l+Fs]Fls]Fs[l-Fs]F[l+Fs]Fl[ls[l-Fs[l-Fs]F[l+Fs]Fls[l-Fs[l-Fs]F[l+Fs]Fls]Fs
[l-Fs]F[l+Fs]Fl[l+Fs[l-Fs]F[l+Fs]Fls]Fs[l-Fs]F[l+Fs]Fl[ls]Fs[l-Fs]F[l+Fs]Fl
s[l-Fs[l-Fs]F[l+Fs]Fls]Fs[l-Fs]F[l+Fs]Fl[l+Fs[l-Fs]F[l+Fs]Fls]Fs[l-Fs]F[l+
Fs]Fl[l+Fs[l-Fs]F[l+Fs]Fls[l-Fs[l-Fs]F[l+Fs]Fls]Fs[l-Fs]F[l+Fs]Fl[l+Fs[l-
Fs]F[l+Fs]Fls]Fs[l-Fs]F[l+Fs]Fl[ls]Fs[l-Fs]F[l+Fs]Fls[l-Fs[l-Fs]F[l+Fs]Fls]
Fs[l-Fs]F[l+Fs]Fl[l+Fs[l-Fs]F[l+Fs]Fls]Fs[l-Fs]F[l+Fs]Fl[ls[l-Fs[l-Fs]F[l+
Fs]Fls[l-Fs[l-Fs]F[l+Fs]Fls]Fs[l-Fs]F[l+Fs]Fl[l+Fs[l-Fs]F[l+Fs]Fls]Fs[l-Fs
]F[l+Fs]Fl[ls[l-Fs[l-Fs]F[l+Fs]Fls[l-Fs[l-Fs]F[l+Fs]Fls]Fs[l-Fs]F[l+Fs]Fl[l
+Fs[l-Fs]F[l+Fs]Fls]Fs[l-Fs]F[l+Fs]Fl[ls]Fs[l-Fs]F[l+Fs]Fls[l-Fs[l-Fs]F[l+F
s]Fls]Fs[l-Fs]F[l+Fs]Fl[l+Fs[l-Fs]F[l+Fs]Fls]Fs[l-Fs]F[l+Fs]Fl[l[l+Fs[l-Fs]
F[l+Fs]Fls[l-Fs[l-Fs]F[l+Fs]Fls]Fs[l-Fs]F[l+Fs]Fl[l+Fs[l-Fs]F[l+Fs]Fls]Fs[
l-Fs]F[l+Fs]Fl[ls]Fs[l-Fs]F[l+Fs]Fls[l-Fs[l-Fs]F[l+Fs]Fls]Fs[l-Fs]F[l+Fs]Fl
[l+Fs[l-Fs]F[l+Fs]Fls]Fs[l-Fs]F[l+Fs]Fl[ls]Fs[l-Fs]F[l+Fs]Fls[l-Fs[l-Fs]F[
l+Fs]Fls]Fs[l-Fs]F[l+Fs]Fl[l+Fs[l-Fs]F[l+Fs]Fls]Fs[l-Fs]F[l+Fs]Fl[ls]Fs[l-
-Fs]F[l+Fs]Fls[l-Fs[l-Fs]F[l+Fs]Fls]Fs[l-Fs]F[l+Fs]Fl[l+Fs[l-Fs]F[l+Fs]Fls
]Fs[l-Fs]F[l+Fs]Fl[ls]Fs[l-Fs]F[l+Fs]Fls[l-Fs[l-Fs]F[l+Fs]Fls]Fs[l-Fs]F[l+F
s]Fl[l+Fs[l-Fs]F[l+Fs]Fls]Fs[l-Fs]F[l+Fs]Fl[ls[l-Fs[l-Fs]F[l+Fs]Fls[l-Fs[l-F
s]F[l+Fs]Fls]Fs[l-Fs]F[l+Fs]Fl[l+Fs[l-Fs]F[l+Fs]Fls]Fs[l-Fs]F[l+Fs]Fl[ls]Fs
[l-Fs]F[l+Fs]Fls[l-Fs[l-Fs]F[l+Fs]Fls]Fs[l-Fs]F[l+Fs]Fl[l+Fs[l-Fs]F[l+Fs]F
ls]Fs[l-Fs]F[l+Fs]Fl[ls[l-Fs]F[l+Fs]Fls[l-Fs[l-Fs]F[l+Fs]Fls]Fs[l-Fs]F
[l+Fs]Fl[l+Fs[l-Fs]F[l+Fs]Fls]Fs[l-Fs]F[l+Fs]Fl[ls]Fs[l-Fs]F[l+Fs]Fl[l+Fs[l-
+Fs]Fls]Fs[l-Fs]F[l+Fs]Fl[ls]Fs[l-Fs]F[l+Fs]Fl[l+Fs[l-Fs]F[l+Fs]Fls]Fs[l-
Fs]F[l+Fs]Fl[l+Fs[l-Fs]F[l+Fs]Fls]Fs[l-Fs]F[l+Fs]Fl[ls]Fs[l-Fs]F[l+Fs]Fl
-Fs[l-Fs]F[l+Fs]Fls]Fs[l-Fs]F[l+Fs]Fl[l+Fs[l-Fs]F[l+Fs]Fls]Fs[l-Fs]F[l+Fs]
Fl[ls]Fs[l-Fs]F[l+Fs]Fls[l-Fs[l-Fs]F[l+Fs]Fls]Fs[l-Fs]F[l+Fs]Fl[l+Fs[l-Fs]
F[l+Fs]Fls]Fs[l-Fs]F[l+Fs]Fl[ls[l-Fs[l-Fs]F[l+Fs]Fls[l-Fs[l-Fs]F[l+Fs]Fls]F
s[l-Fs]F[l+Fs]Fl[l+Fs[l-Fs]F[l+Fs]Fls]Fs[l-Fs]F[l+Fs]Fl[ls]Fs[l-Fs]F[l+Fs]F
ls[l-Fs[l-Fs]F[l+Fs]Fls]Fs[l-Fs]F[l+Fs]Fl[l+Fs[l-Fs]F[l+Fs]Fls]Fs[l-Fs]F[l
+Fs]Fl[l+Fs[l-Fs]F[l+Fs]Fls[l-Fs[l-Fs]F[l+Fs]Fls]Fs[l-Fs]F[l+Fs]Fl[l+Fs[l
-Fs]F[l+Fs]Fls]Fs[l-Fs]F[l+Fs]Fl[ls]Fs[l-Fs]F[l+Fs]Fls[l-Fs[l-Fs]F[l+Fs]Fls
]Fs[l-Fs]F[l+Fs]Fl[l+Fs[l-Fs]F[l+Fs]Fls]Fs[l-Fs]F[l+Fs]Fl[l[l+Fs]"
```

Figure 9.58: The sentence underlying Figure 9.57.

Parameterized L-Systems use symbols that encode additional information. For example `F(20)` might represent a forward movement of size 20, while `+(30)` might represent a right turn of 30°. Similarly, replacement rules could take this information into account; as in `F(n)` → `F(n/2)[+F(n/2)]-F(n/2)`. Such rules might be encoded as R lists (or our own nested lists) as in `s <- list(list("F", 20), list("+", 30), list("F", 20))`. Corresponding versions of `lproduce()` and `draw_sentence()` would need to be created to work with such lists-of-lists.

What about rules that are not deterministic? Perhaps with 25% probability we use the rule `F` → `F[+F]F` and with the remaining 75% probability we use `F` → `F[-F]F`. Symbols could be occasionally deleted (say, with 50% probability we neglect to put anything in the produced sentence for the symbol). "Context-sensitive" L-Systems take multiple symbols into account in sentence generation, as in `F+` → `F+F` and `F-` → `F-F`.

Finally, even basic L-Systems can include symbols that are completely ignored in drawing but are used in the sentence generation process. Consider the following L-System:

start: `X`

`X` → `F-[[X]+X]+F[+FX]-X`

`F` → `FF`

Figure 9.59 illustrates five iterations of `lproduce()`, where `X` is interpreted as "do nothing," left and right turns are 20° and 25°.

A vast array of geometric figures can be produced and drawn with L-Systems as well, including the Koch curve described by the rule `F` → `F-F++F-F` (with 60° turns). There are many resources to be found for L-Systems on the web. Excellent books that cover the topic include *The Nature of Code* by Daniel Shiffman and *The Algorithmic Beautify of Plants* by Przemyslaw Prusinkiewicz and Aristid Lindenmayer, both available for free online.

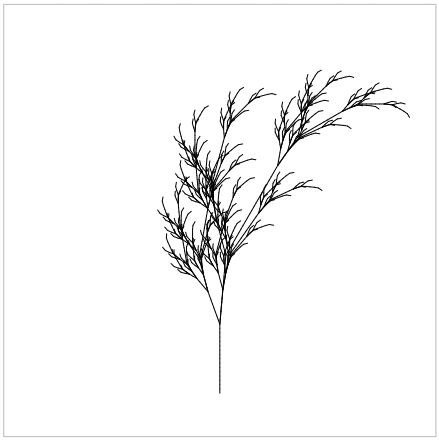


Figure 9.59: Drawing for an L-System with start symbol `X` and rules `X` → `F-[[X]+X]+F[+FX]-X`, `F` → `FF`.

Exercises

1. Consider the L-System sentence `F[-Fc]F[+Fc]Fc`, where `c` is interpreted as "draw a small circle." This would add circular "leaves" to Figure 9.50. Create a modified substitution rule such that after each sentence generation, leaves are present at the ends of all branches (but not at internal branching points).

2. Add randomness to the `draw_sentence()` function, such that each `F`, `+`, or `-` is interpreted and drawn slightly differently.

3. Research some of the properties of "space-filling curves," and implement one (such as the Hilbert curve) as an L-System.

4. See if you can implement a parameterized L-System, with "symbols" made up of 2-element lists and sentences as lists of symbols (e.g. `list(list("F", 10), list("+", 36))`). Rules will likely also need to be stored as lists; in fact, they could store *anonymous functions* (functions not given a name). For example, the rule $F(n) \rightarrow F(n/2)$ might be encoded as `rules[["F"]] <- function(value) { return(value/2) }`. Using the rule would require first extracting the function, with something like `if(symbol[[1]] == "F") { func <- rules[["F"]] }` and then calling it with something like `newsymbol <- list("F", func(symbol[[2]]))`.

Epilogue

This book is an outgrowth of a short class I taught at Oregon State University through the Center for Genome Research and Biocomputing, "Recursion and Dynamic Programming for Sequence Analysis." My goal for the class was to introduce the beauty of computer science to life scientists, via the algorithms and tools they encounter frequently. Too often (in my opinion) are methods in bioinformatics reduced to mechanistic table-filling, when fundamental ideas about computing and mathematics lie so close at hand.

This audience drove the use of R; in the first offering of the class, R was a comfortable language for most, whereas other more common choices (like Python) would have required additional background. Soon what was a pedagogical challenge became an opportunity, as R's flexible graphical utilities allowed us to explore the internals of these algorithms in ways I had not previously. Additionally, I discovered that R's nature as a functional language with procedural capabilities worked well with the recursion $\rightarrow$ memoization $\rightarrow$ dynamic programming exposition I prefer. Indeed, my own experiments in R for this work have helped me explore the fascinating interface between functional and procedural paradigms.

A secondary goal for this book was to produce an homage to these topics I love so much, as an artifact of elegance itself worthy of display. Hopefully my few skills in typesetting and design (with the help of some excellent LaTeX packages) have been up to the challenge. I apologize that in some sections the reader must play "hunt the figure," a consequence of the LaTeX layout engine in figure-dense regions.

With respect to content, in a few ways the current edition is over-complete, and in others under-complete. Some of the material in Chapters 3 and 4 was not included in the original class and is tangential to the main topics. Yet, for example, it felt irresponsible to cover stacks and depth-first-search in trees, but not depth- and breadth-first search in graphs. In the future I'd like to add some discussion of phylogenetic trees. I also think context-free-grammars,

the CYK dynamic programming algorithm, and their application to RNA secondary structure prediction would be a nice followup (or precursor) to L-Systems.

THIS BOOK WAS TYPESET with the excellent Tufte-LaTeX package, with Palatino and Helvetica typefaces for the main text, and Inconsolata for code. Other packages of note include `minted` for styling code blocks (with a custom scheme for R code), `epigraph` for chapter epigraphs, `wrapfig` for placing text-wrapping figures, and `subfloat` for numbering of sub-figures. Most figures were produced either as R output, or with Apple Keynote. The cover image is courtesy Wikimedia Commons.

Index