PART II

Numerical techniques

CHAPTER 9

Numerical accuracy and program efficiency

When using a computer to perform intensive numerical calculations, there are two important issues we should bear in mind: accuracy and speed.

In this chapter we will consider technical details about how computers operate, and their ramifications for programming practice, particularly within R. We look at how computers represent numbers, and the effect that this has on the accuracy of computation results. We also discuss the time it takes to perform a computation and programming techniques for speeding things up. Finally we consider the effects of memory limitations on computation efficiency.

9.1 Machine representation of numbers

Computers use switches to encode information. A single ON/OFF indicator is called a bit; a group of eight bits is called a byte. Although it is quite arbitrary, it is usual to associate 1 with ON and 0 with OFF.

9.1.1 Integers

A fixed number of bytes is used to represent a single integer, usually four or eight. Let k be the number of bits we have to work with (usually 32 or 64). There are a number of schema used to encode integers. We describe three of them: the *sign-and-magnitude*, *biased*, and *two's complement* schema.

In the sign-and-magnitude scheme, we use one bit to represent the sign $+/-$ and the remainder to give a binary representation of the magnitude. Using the sequence $\pm b_{k-2} \cdots b_2 b_1 b_0$, where each b_i is 0 or 1, we represent the decimal number $\pm(2^0 b_0 + 2^1 b_1 + 2^2 b_2 + \cdots + 2^{k-2} b_{k-2})$. For example, taking $k = 8$, -0100101 is interpreted as $-(2^5 + 2^2 + 2^0) = -37$. The smallest and largest integers we can represent under this scheme are $-(2^{k-1} - 1)$ and $2^{k-1} - 1$.

The disadvantage of this scheme is that there are two representations of 0.

In the biased scheme, we use the sequence $b_{k-1} \cdots b_1 b_0$ to represent the decimal number $2^0 b_0 + 2^1 b_1 + \cdots 2^{k-1} b_{k-1} - (2^{k-1} - 1)$. For example, taking $k = 8$,

00100101 is interpreted as $37 - 255 = -218$. The smallest and largest integers represented under this scheme are $-(2^{k-1} - 1)$ and 2^{k-1}.

The disadvantage of this scheme is that addition becomes a little more complex.

The most popular scheme for representing integers on a computer is the *two's complement* scheme. Given k bits, the numbers $0, 1, \ldots, 2^{k-1} - 1$ are represented in the usual way, using a binary expansion, but the numbers $-1, -2, \ldots, -2^{k-1}$ are represented by $2^k - 1, 2^k - 2, \ldots, 2^k - 2^{k-1}$. We will not go into the details, but it turns out that addition under this scheme is equivalent to addition modulo 2^k, and can be implemented very efficiently.

The representation of integers on your computer happens at a fundamental level, and R has no control over it. The largest integer you can represent on your computer (whatever encoding scheme is in use) is known as *maxint*; R records the value of maxint on your computer in the variable `.Machine`.

```
> .Machine$integer.max

[1] 2147483647
```

If you know a number is integer valued then it is efficient to store it as such. However in practice R almost always treats integers the same way it does real numbers, for which it uses *floating point representation*.

9.1.2 Real numbers

Floating point representation is based on binary scientific notation. In decimal scientific notation, we write $x = d_0.d_1d_2\cdots \times 10^m$, where $d_0, d_1, \ldots$ are digits, with $d_0 \neq 0$ unless $x = 0$. In binary scientific notation, we write $x = b_0.b_1b_2\cdots \times 2^m$, where $b_0, b_1, \ldots$ are all 0 or 1, with $b_0 = 1$ unless $x = 0$. The sequence $d_0.d_1d_2\cdots$ or $b_0.b_1b_2\cdots$ is called the mantissa and m the exponent.

R can use scientific `e` notation to represent numbers:

```
> 1.2e3

[1] 1200
```

The `e` should be read as 'ten raised to the power' and should not be confused with the exponential.

In practice we must limit the size of the mantissa and the exponent; that is, we limit the precision and range of the real numbers we work with. In *double precision* eight bytes are used to represent floating point numbers: 1 bit is used for the sign, 52 bits for the mantissa, and 11 bits for the exponent. The biased scheme is used to represent the exponent, which thus takes on values from -1023 to 1024. For the mantissa, 52 bits are used for $b_1, \ldots, b_{52}$ while the value of b_0 depends on m:

- If $m = -1023$ then $b_0 = 0$, which allows us to represent 0, using $b_1 = \cdots = b_{52} = 0$, or numbers smaller in size than 2^{-1023} otherwise (these are called denormalised numbers).
- If $-1023 < m < 1024$ then $b_0 = 1$.
- If $m = 1024$ then we use $b_1 = \cdots = b_{52} = 0$ to represent $\pm\infty$, which R writes as `-Inf` and `+Inf`. If one of the $b_i \neq 0$ then we interpret the representation as `NaN`, which stands for Not a Number.

```
> 1/0

[1] Inf

> 0/0

[1] NaN
```

In double precision, the smallest non-zero positive number is 2^{-1074} and the largest number is $2^{1023}(2 - 2^{-53})$ (sometimes called realmax). More importantly, the smallest number x such that $1 + x$ can be distinguished from 1 is $2^{-52} \approx 2.220446 \times 10^{-16}$, which is called *machine epsilon*. Thus, in base 10, double precision is roughly equivalent to 16 significant figures, with exponents of size up to ± 308.

```
> 2^-1074 == 0

[1] FALSE

> 1/(2^-1074)

[1] Inf

> 2^1023 + 2^1022 + 2^1021

[1] 1.572981e+308

> 2^1023 + 2^1022 + 2^1022

[1] Inf

> x <- 1 + 2^-52
> x - 1

[1] 2.220446e-16

> y <- 1 + 2^-53
> y - 1

[1] 0
```

When arithmetic operations on double precision floating point numbers produce a result smaller in magnitude than 2^{-1074} or larger in magnitude than realmax, then the result is 0 or $\pm\infty$, respectively. We call this *underflow* or *overflow*.

The double precision scheme we have described here is part of the IEEE Standard for Binary Floating-Point Arithmetic IEEE 754-1985. This standard is used in practically all contemporary computers, though compliance cannot be guaranteed. The implementation of floating point arithmetic happens at a fundamental level of the computer and is not something R can control. It is something R is aware of however, and the constant `.Machine` will give you details about your local numerical environment.

9.2 Significant digits

Using double precision numbers is roughly equivalent to working with 16 significant digits in base 10. Arithmetic with integers will be exact for values from $-(2^{53}-1)$ to $2^{53}-1$ (roughly -10^{16} to 10^{16}), but as soon as you start using numbers outside this range, or fractions, you can expect to lose some accuracy due to *roundoff error*. For example, 1.1 does not have a finite binary expansion, so in double precision its binary expansion is rounded to $1.00011001100\cdots001$, with an error of roughly 2^{-53}.

To allow for roundoff error when comparing numbers, we can use `all.equal(x, y, tol)`, which returns `TRUE` if `x` and `y` are within `tol` of each other, with default given by the square root of machine epsilon (roughly 10^{-8}).

Let $\tilde{a}$ be an approximation of a, then the *absolute error* is $|\tilde{a}-a|$ and the *relative error* is $|\tilde{a}-a|/a$. Restricting $\tilde{a}$ to 16 significant digits is equivalent to allowing a relative error of 10^{-16}. When adding two approximations we add the absolute errors to get (a bound on) the absolute error of the result. When multiplying two approximations we add the relative errors to get (an approximation of) the relative error of result: suppose ϵ and δ are the (small) relative errors of a and b, then

$$\tilde{a}\tilde{b} = a(1+\epsilon)b(1+\delta) = ab(1+\epsilon+\delta+\epsilon\delta) \approx ab(1+\epsilon+\delta).$$

Suppose we add 1,000 numbers each of size around 1,000,000 with relative errors of up to 10^{-16}. Each thus has an absolute error of up to 10^{-10}, so adding them all we would have a number of size around 1,000,000,000 with an absolute error of up to 10^{-7}. That is, the relative error remains much the same at 10^{-16}. However, things can look very different when you start subtracting numbers of a similar size. For example, consider

$$1,234,567,812,345,678 - 1,234,567,800,000,000 = 12,345,678.$$

If the two numbers on the left-hand side have relative errors of 10^{-16}, then

the right-hand side has an absolute error of about 1, which is a relative error of around 10^{-8}: a dramatic loss in accuracy, which we call *catastrophic cancellation* error.

Catastrophic cancellation is an inherent problem when you have a finite number of significant digits. However if you are aware of it, it can sometimes be avoided.

9.2.1 Example: $\sin(x) - x$ *near 0*

Since $\lim_{x\to 0} \sin(x)/x = 1$, we have that $\sin(x) \approx x$ near 0. Thus if we wish to know $\sin(x) - x$ near 0, then we expect catastrophic cancellation to reduce the accuracy of our calculation.

The Taylor expansion of $\sin(x)$ about 0 is $\sum_{n=0}^{\infty}(-1)^n x^{2n+1}/(2n+1)!$, thus

$$\sin(x) - x = \sum_{n=1}^{\infty}(-1)^n \frac{x^{2n+1}}{(2n+1)!}.$$

If we truncate this expansion to N terms, then the error is at most $|x^{2N+1}/(2N+1)!|$ (this can be proved using the fact that the summands oscillate in sign while decreasing in magnitude). Suppose we approximate $\sin(x)-x$ using two terms, namely

$$\sin(x) - x \approx -\frac{x^3}{6} + \frac{x^5}{120} = -\frac{x^3}{6}\left(1 - \frac{x^2}{20}\right).$$

If $|x| < 0.001$ then the error in the approximation is less than $0.001^5/120 < 10^{-17}$ in magnitude. If $|x| < 0.000001$ then the error is less than 10^{-302}. Since this formula does not involve subtraction of similarly sized numbers, it does not suffer from catastrophic cancellation.

```
> x <- 2^-seq(from = 10, to = 40, by = 10)
> x

[1] 9.765625e-04 9.536743e-07 9.313226e-10 9.094947e-13

> sin(x) - x

[1] -1.552204e-10 -1.445250e-19  0.000000e+00  0.000000e+00

> -x^3/6 * (1 - x^2/20)

[1] -1.552204e-10 -1.445603e-19 -1.346323e-28 -1.253861e-37
```

We see that for $x = 2^{-20} \approx 10^{-6}$, catastrophic cancellation when calculating $\sin(x) - x$ naively has resulted in an absolute error of around 10^{-23}, which may sound alright until we reflect that this is a relative error of around 10^{-4}. For $x = 2^{-30}$ the relative error is 1!

9.2.2 Example: range reduction

When approximating $\sin(x)$ using a Taylor expansion about 0, the further x is from 0, the more terms we need in the expansion to get a reasonable approximation. But $\sin(x)$ is periodic, so to avoid this, for large x we can just take k such that $|x - 2k\pi| \leq \pi$, then use $\sin(x) = \sin(x - 2k\pi)$.

Unfortunately this procedure can cause problems because of catastrophic cancellation.

Suppose we start with 16 significant digits. If x is large, say around 10^8, then the absolute error of x will be around 10^{-8} and thus the absolute error of $x - 2k\pi$ will be around 10^{-8}. This means the relative error of $x - 2k\pi$ has increased to (at least) 10^{-8} (more if $x - 2k\pi$ is close to 0).

9.3 Time

Ultimately we measure how efficient a program is by how long it takes to run, which will depend on the language it is written in and the computer it is run on. Also, computers typically are doing a number of things at once, such as playing music, watching the mouse or the mailbox, so the time taken to run a program will also depend on what else the computer is doing at the same time. To measure how many CPU (Computer Processing Unit) seconds are spent evaluating an expression, we use `system.time(expression)`. More generally, the expression `proc.time()` will tell you how much time you have used on the CPU since the current R session began.

```
> system.time(source("primedensity.r"))

   user  system elapsed
   0.08    0.03    0.19
```

The sum of the user and system time gives the CPU seconds spent evaluating the expression. The elapsed time also includes time spent on tasks unrelated to your R session.

In most cases of interest, the time taken to run a program depends on the nature of the inputs. For example, the time taken to sum a vector or sort a vector will clearly depend on the length of the vector, n say. Alternatively we may want a solution to the equation $f(x) = 0$, accurate to within some tolerance ϵ, in which case the time taken will depend on ϵ. Because we cannot test a program with all possible inputs, we need a theoretical method of assessing efficiency, which would depend on n or ϵ, and which will give us a measure of how long the program should take to run. We do this by counting the number of operations executed in running a program, where operations are tasks such as addition, multiplication, logical comparison, variable assignment, and calling built-in functions.

For example, the following program will sum the elements of a vector `x`:

```
S <- 0
for (a in x) S <- S + a
```

Let n be the length of `x`, then when we run this program we carry out n addition operations and $2n + 1$ variable assignments (we assign a value to `a` and to `S` each time we go around the `for` loop).

For a second example, suppose we are using the Taylor series $\sum_{n=1}^{N}(-1)^{n+1}x^n/n$ to approximate $\log(1+x)$, for $0 \leq x \leq 1$, and we want an error of at most $\pm\epsilon$. It can be shown that the approximation error is no greater in magnitude than the last term in the sum, which suggests the following code:

```
eps <- 1e-12
x <- 0.5

n <- 0
log1x <- 0
while (n == 0 || abs(last.term) > eps) {
  n <- n + 1
  last.term <- (-1)^(n+1)*x^n/n
  log1x <- log1x + last.term
}
```

How many arithmetic operations are performed when running this program? When we go around the loop the n-th time we perform three additions and $2n+3$ multiplications/divisions, noting that x^n requires n multiplications. We loop until $x^n/n < \epsilon$. Putting $x = 1$ we get $n = \lceil 1/\epsilon \rceil$, which is an upper bound on n for all $x \in (0, 1]$. (Here $\lceil 1/\epsilon \rceil$ is the *ceiling* of $1/\epsilon$, obtained by rounding up to the nearest integer.) Thus the total number of additions will be bounded by $3\lceil 1/\epsilon \rceil$ and the total number of multiplications/divisions bounded by

$$\sum_{n=1}^{\lceil 1/\epsilon \rceil} (2n+3) = \lceil 1/\epsilon \rceil (\lceil 1/\epsilon \rceil + 1) + 3\lceil 1/\epsilon \rceil = \lceil 1/\epsilon \rceil^2 + 4\lceil 1/\epsilon \rceil.$$

In this example, a simple modification to the program will improve the efficiency. Change the line `last.term <- (-1)^(n+1)*x^n/n` to

```
last.term <- -last.term*x*(1 - 1/n)
```

We now have just three multiplications/divisions each time we go around the loop, so the total number will be bounded by $3\lceil 1/\epsilon \rceil$. (Multiplying by -1 does not count as a multiplication.)

In practice, if we know the number of operations grows like an^b where n is the problem size (the length of the vector or the inverse tolerance in our examples), then the value of b is *much* more important than the value of a. For this reason rather than count operations exactly, it is usually sufficient to ascertain how fast they grow as a function of n. Let f and g be functions of n, then we say that $f(n) = O(g(n))$ as $n \to \infty$ if $\lim_{n\to\infty} f(n)/g(n) < \infty$, and $f(n) = o(g(n))$ as $n \to \infty$ if $\lim_{n\to\infty} f(n)/g(n) = 0$. Our first example

required $O(n)$ operations to sum a vector of length n. In its initial form, our second example required $O(1/\epsilon^2)$ operations to calculate $\log(1+x)$ to within tolerance ϵ.

Some operations take much longer than others. Variable assignment (to an existing variable), addition and subtraction are quick. Multiplication and division take a bit longer, and powers take longer still. Transcendental functions such as sin and log have to be calculated and so take even longer, but not as long as user-defined functions.

As we have already seen in Example 3.3.4, creating or changing the size of a vector (also called redimensioning an array) is relatively slow, which is why, when we know how big a vector is going to be, it is better to initialise it fully grown (but full of zeros) than to increase it incrementally. We can compare the relative speeds using `system.time`:

```
> n <- 10000
> x <- rep(0, n)
> system.time(for (i in 1:n) x[i] <- i^2)

   user  system elapsed
  0.023   0.000   0.024

> x <- c()
> system.time(for (i in 1:n) x[i] <- i^2)

   user  system elapsed
  0.515   0.044   0.621
```

In practice what we do is identify the longest or most important operation in a program, and count how many times it is performed. For example, for numerical integration, root-finding, and optimisation, we are working with a user-defined function f, and we count how many times $f(x)$ is evaluated, for different x. For numerical sorting algorithms (see Exercise 7), we count how many comparisons of the form $x < y$ are made. For advanced uses, the function `Rprof` can be used to capture a lot of information about a program as it runs.

9.4 Loops versus vectors

In R, vector operations are generally faster than equivalent loops. However, if you just count operations there appears to be no reason for this. R is a very high-level language in which it is relatively easy to create and manipulate variables. The price we pay for this flexibility is speed. When you evaluate an expression in R, it is 'translated' into a faster lower-level language before being evaluated, then the result is translated back into R. It is the translation that

takes much of the time, and vectorisation saves on the amount of translation required.[1]

For example, take the following code to square each element of x:

```
for (i in 1:length(x)) x[i] <- x[i]^2
```

Each time we evaluate the expression `x[i] <- x[i]^2`, we have to translate `x[i]` into our lower-level language, and then translate the result back. In contrast, to evaluate the expression `x <- x^2`, we translate `x` all at once and then square it, before translating the answer back: all the work takes place in our faster lower-level language.

Many of R's functions are vectorised, which means that if the first argument is a vector, then the output will be a vector of the same length, computed by applying the function elementwise to the input vector. Vectorisation allows for faster executing code that is easier to read. User-defined functions can also be vectorised if they comprise functions that are innately vectorised, or are invoked using one of the `apply` family of functions (see Sections 5.4 and 6.4).

When we have a numerically intensive algorithm that uses lots of loops and cannot be vectorised, then R allows us to encode the algorithm in C or Fortran (which are faster lower-level languages) and then access this as a function. Section 8.6 gives some pointers as to how this is done.

9.4.1 Example: column sums of a matrix

We demonstrate a collection of different approaches to solving the problem of summing across the columns of a matrix. We confine ourselves to R code, and we order the solutions from the least to the most efficient.

```
> big.matrix <- matrix(1:1e+06, nrow = 1000)
> colsums <- rep(NA, dim(big.matrix)[2])
```

We compare

1. A double loop of summations,

```
> system.time({
+     for (i in 1:dim(big.matrix)[2]) {
+         s <- 0
+         for (j in 1:dim(big.matrix)[1]) {
+             s <- s + big.matrix[j, i]
+         }
+         colsums[i] <- s
+     }
+ })
```

[1] This is a rather simplified view of what is going on. Nonetheless, it provides us with a workable cognitive model in which we can express the problem.

```
   user  system elapsed
  1.727   0.000   1.903
```

2. The use of `apply`,

```
> system.time(colsums <- apply(big.matrix, 2, sum))

   user  system elapsed
  0.035   0.008   0.044
```

3. A single loop of sums, and

```
> system.time(for (i in 1:dim(big.matrix)[2]) {
+     colsums[i] <- sum(big.matrix[, i])
+ })

   user  system elapsed
  0.029   0.001   0.030
```

4. Using the dedicated R function:

```
> system.time(colsums <- colSums(big.matrix))

   user  system elapsed
  0.004   0.000   0.003
```

We note that using `apply` is not faster than using a `for` loop. This is because, in R, `apply` creates its own for loop.

9.5 Memory

Computer memory comes in a variety of forms. For most purposes it is sufficient to think in terms of RAM (random access memory), which is fast, and the hard disk, which is slow.

Variables require memory. By default they are stored in RAM, but if you have too many they will be stored on the hard disk then swapped into RAM when needed, which takes time. Historically RAM was expensive and in short supply, and keeping memory use to a minimum was important if you wanted your programs to run quickly. In the present day however, RAM is relatively cheap, and programmers seldom have to worry about how many variables they use. Moreover, because accessing an existing variable is invariably quicker than recalculating it, it is usual to store commonly used quantities for reuse.

For example, consider the function `prime` from Example 5.4.1:

```
# program spuRs/resources/scripts/prime.r

prime <- function(n) {
    # returns TRUE if n is prime
    # assumes n is a positive integer
```

```
    if (n == 1) {
        is.prime <- FALSE
    } else if (n == 2) {
        is.prime <- TRUE
    } else {
        is.prime <- TRUE
        m <- 2
        m.max <- sqrt(n)  # only want to calculate this once
        while (is.prime && m <= m.max) {
            if (n %% m == 0) is.prime <- FALSE
            m <- m + 1
        }
    }
    return(is.prime)
}
```

Calculating $\sqrt{n}$ is relatively slow, so we do this once and store the result. An alternative is for the main while loop to start as follows:

```
while (is.prime && m <= sqrt(n))
```

Coding the loop this way would require us to recalculate $\sqrt{n}$ each time we check the loop condition, which is inefficient.

Because R works much more quickly with vectors than loops, it is usual to try to vectorise R programs. This will occasionally produce very large vectors. As soon as a vector (or list) is too large to store in RAM all at once, the speed at which you can use it will drop dramatically. If it is sufficiently large, then it may not be possible to store it at all, in which case you are said to have run out of memory.

R has an absolute limit on the length of a vector of $2^{31} - 1 = 2{,}147{,}483{,}647$ (the result of using signed 32-bit integers to index vectors), however, if you run out of memory it is more likely that the problem is that you have reached the limits of your computing environment. If you find this happening then you will need to break your vectors down into smaller subvectors and deal with each in turn. In extreme cases it may be necessary to save a variable and then delete it from the workspace, using `save` and `rm`, to free up enough memory for your program to keep running.

9.6 Caveat

In this chapter we have considered programming efficiency only from the point of view of code execution. A more useful approach is to consider programming efficiency from the point of view of code creation as well as execution; that is, to include the cost of code development. It may well be that judicious refining can trim an hour off the execution of your code, but if it takes two hours to do so, then perhaps there is no real gain. This is the evaluation that a

programmer must make: what are the short-term and long-term benefits of code optimisation against the short-term cost of programming time?

For large projects involving more than one programmer other considerations become important, such as the clarity and stability of your code. That is, can others easily understand what the code does, and does it produce sensible answers no matter what sort of input is provided (even if that is just an informative error message). The practice of systematically developing and maintaining large complicated programs is often referred to as *software engineering*.

9.7 Exercises

1. In single precision four bytes are used to represent a floating point number: 1 bit is used for the sign, 8 for the exponent, and 23 for the mantissa.

   What are the largest and smallest non-zero positive numbers in single precision (including denormalised numbers)?

   In base 10, how many significant digits do you get using single precision?
2. What is the relative error in approximating π by $22/7$? What about $355/113$?
3. Suppose x and y can be represented without error in double precision. Can the same be said for x^2 and y^2?

   Which would be more accurate, $x^2 - y^2$ or $(x - y)(x + y)$?
4. To calculate $\log(x)$ we use the expansion

$$\log(1 + x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \cdots.$$

   Truncating to n terms, the error is no greater in magnitude than the last term in the sum. How many terms in the expansion are required to calculate $\log 1.5$ with an error of at most 10^{-16}? How many terms are required to calculate $\log 2$ with an error of at most 10^{-16}?

   Using the fact that $\log 2 = 2 \log \sqrt{2}$, suggest a better way of calculating $\log 2$.
5. The sample variance of a set of observations $x_1, \ldots, x_n$ is given by $S^2 = \sum_{i=1}^n (x_i - \overline{x})^2/(n-1) = (\sum_{i=1}^n x_i^2 - n\overline{x}^2)/(n-1)$, where $\overline{x} = \sum_{i=1}^n x_i/n$ is the sample mean.

   Show that the second formula is more efficient (requires fewer operations) but can suffer from catastrophic cancellation. Demonstrate catastrophic cancellation with an example sample of size $n = 2$.
6. Horner's algorithm for evaluating the polynomial $p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$ consists of re-expressing it as

$$a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + x a_n) \cdots)).$$

How many operations are required to evaluate $p(x)$ in each form?

7. This exercise is based on the problem of sorting a list of numbers, which is one of the classic computing problems. Note that R has an excellent sorting function, `sort(x)`, which we will not be using.

   To judge the effectiveness of a sorting algorithm, we count the number of *comparisons* that are required to sort a vector `x` of length `n`. That is, we count the number of times we evaluate logical expressions of the form `x[i] < x[j]`. The fewer comparisons required, the more efficient the algorithm.

   Selection sort The simplest but least efficient sorting algorithm is selection sort. The selection sort algorithm uses two vectors, an unsorted vector and a sorted vector, where all the elements of the sorted vector are less than or equal to the elements of the unsorted vector. The algorithm proceeds as follows:

   1. Given a vector x, let the initial unsorted vector u be equal to x, and the initial sorted vector s be a vector of length 0.
   2. Find the smallest element of u then remove it from u and add it to the end of s.
   3. If u is not empty then go back to step 2.

   Write an implementation of the selection sort algorithm. To do this you may find it convenient to write a function that returns the *index* of the smallest element of a vector.

   How many comparisons are required to sort a vector of length n using the selection sort algorithm?

   Insertion sort Like selection sort, insertion sort uses an unsorted vector and a sorted vector, moving elements from the unsorted to the sorted vector one at a time. The algorithm is as follows:

   1. Given a vector x, let the initial unsorted vector u be equal to x, and the initial sorted vector s be a vector of length 0.
   2. Remove the last element of u and insert it into s so that s is still sorted.
   3. If u is not empty then go back to step 2.

   Write an implementation of the insertion sort algorithm. To insert an element a into a sorted vector $s = (b_1, \ldots, b_k)$ (as per step 2 above), you do not usually have to look at every element of the vector. Instead, if you start searching from the end, you just need to find the first i such that $a \geq b_i$, then the new sorted vector is $(b_1, \ldots, b_i, a, b_{i+1}, \ldots, b_k)$.

   Because inserting an element into a sorted vector is usually quicker than finding the minimum of a vector, insertion sort is usually quicker than selection sort, but the actual number of comparisons required depends on the initial vector x. What are the worst and best types of vector x, with respect to sorting using insertion sort, and how many comparisons are required in each case?

Bubble sort Bubble sort is quite different from selection sort and insertion sort. It works by repeatedly comparing adjacent elements of the vector $x = (a_1, \ldots, a_n)$, as follows:

1. For $i = 1, \ldots, n-1$, if $a_i > a_{i+1}$ then swap a_i and a_{i+1}.
2. If you did any swaps in step 1, then go back and do step 1 again.

Write an implementation of the bubble sort algorithm and work out the minimum and maximum number of comparisons required to sort a vector of length n.

Bubble sort is not often used in practice. Its main claim to fame is that it does not require an extra vector to store the sorted values. There was a time when the available memory was an important programming consideration, and so people worried about how much storage an algorithm required, and bubble sort is excellent in this regard. However at present computing speed is more of a bottleneck than memory, so people worry more about how many operations an algorithm requires.

If you like bubble sort then you should look up the related algorithm *gnome sort*, which was named after the garden gnomes of Holland and their habit of rearranging flower pots.

Quick sort The quick sort algorithm is (on average) one of the fastest sorting algorithms currently available and is widely used. It was first described by C.A.R. Hoare in 1960. Quick sort uses a 'divide-and-conquer' strategy: it is a recursive algorithm that divides the problem into two smaller (and thus easier) problems. Given a vector $x = (a_1, \ldots, a_n)$, the algorithm works as follows:

1. If $n = 0$ or 1 then x is sorted so stop.
2. If $n > 1$ then split $(a_2, \ldots, a_n)$ into two subvectors, l and g, where l consists of all the elements of x less than a_1, and g consists of all the elements of x greater than a_1 (ties can go in either l or g).
3. Sort l and g. Call the sorted subvectors $(b_1, \ldots, b_i)$ and $(c_1, \ldots, c_j)$, respectively, then the sorted vector x is given by $(b_1, \ldots, b_i, a_1, c_1, \ldots, c_j)$.

Implement the quick sort algorithm using a recursive function.

It can be shown that on average the quick sort algorithm requires $O(n \log n)$ comparisons to sort a vector of length n, though its worst-case performance is $O(n^2)$. Also, it is possible to implement quick sort so that it uses memory efficiently while remaining quick.

Two other sorting algorithms that also require on average $O(n \log n)$ comparisons are *heap sort* and *merge sort*.

8. Use the `system.time` function to compare the programs `primedensity.r` and `primesieve.r`, given in Chapter 5.
9. For $\mathbf{x} = (x_1, \ldots, x_n)^T$ and $\mathbf{y} = (y_1, \ldots, y_n)^T$, the *convolution* of $\mathbf{x}$ and $\mathbf{y}$ is

the vector $\mathbf{z} = (z_1, \ldots, z_{2n})^T$ given by

$$z_k = \sum_{i=\max\{1,k-n\}}^{\min\{k,n\}} x_i \cdot y_{k-i}.$$

Write two programs to convolve a pair of vectors, one using loops and the other using vector operations, then use `system.time` to compare their speed.

10. Use the `system.time` function to compare the relative time that it takes to perform addition, multiplication, powers, and other simple operations. You may wish to perform each more than once!