Making R go Faster

LondonR Workshop

Date: 25 September 2018
Trainer: Graham Parsons

Sponsored by:

@ training@mango-solutions.com
+44 (0)1249 75 450
mango-solutions.com

Chapter 1
When to Optimise

1.1 Introduction to the Training

1.1.1 Course Aims

This course has been designed to help you learn when and how to make your R code go faster. It is assumed you already have a basic knowledge of R programming, you are familiar with R's basic data structures, and can write your own functions.

1.1.2 Course Materials

Items appearing in this material are sometimes given a special appearance to set them apart from regular text. Here's how they look:

```
> This is a section of code              # This is a comment
```

A warning, typically describing non-intuitive aspects of the R language

A tip: additional features of R or "shortcuts" based on user experience

Exercises to be performed during (or after) the training course

1.1.3 Course Script and Exercise Answers

A great deal of code will be executed within R during the delivery of this training. This includes the answers to each exercise, as well as other code written to answer questions that arise. You can access the scripts for the course here:

https://github.com/MangoTheCat/making-r-go-faster-workshop

1.2 When to Optimise

During this course we will learn how to write efficient R code that is optimised for speed. Whilst some best practice is easy to internalise and apply throughout our R code, other techniques take time to think about and apply, and come with certain trade-offs.

1.2.1 Speed vs. Legibility

It is possible to write faster code that is very hard to read, for example we can gain more control by using lower-level functions, but these often result in more verbose code where it is harder to see the programmer intended.

On the other hand, following a "Don't Repeat Yourself" (known as DRY) strategy can enhance readability and speed, producing more succinct code that does not repeat steps unnecessarily.

1.2.2 Speed vs. Flexibility

R is a very flexible language. We can do things like call the same function with different data structures and types and still get a meaningful result, or change the class of an object on-the-fly without a problem. For example:

```
> summary(faithful)
   eruptions        waiting
 Min.   :1.600   Min.   :43.0
 1st Qu.:2.163   1st Qu.:58.0
 Median :4.000   Median :76.0
 Mean   :3.488   Mean   :70.9
 3rd Qu.:4.454   3rd Qu.:82.0
 Max.   :5.100   Max.   :96.0

> summary(faithful$eruptions)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.600   2.163   4.000   3.488   4.454   5.100

> str(c(TRUE, 2L, 3.0))
 num [1:3] 1 2 3

> class(faithful) <- "not_a_data.frame"
> faithful  # now prints as a list not a data.frame
```

This flexibility comes at a cost. Whenever functions are called, the list of available methods is inspected in order to dispatch one appropriate to its argument's class. Often functions will include lots of input-checking and take sensible actions depending on the data

provided. Each of these decision points takes time, which can add up when a function is called many times.

We can avoid some of these overheads by writing code that is very specific to a certain purpose, but very brittle and unable to cope with anything but exactly the right inputs. On some occasions this may be worth the risk, but usually not.

1.2.3 Knowing When to Stop

Speeding up code can be very rewarding and feel like we are creating lots of value – but a very common trap to fall into is to optimise your code too early. Generally we should write code that is good enough in terms of performance whilst paying more attention to other issues such as maintainability, legibility, and ease of writing.

In the strategies we will cover in the course we will try to distinguish between those that are probably a good idea to employ regardless of speed, and others that involve trading off legibility or flexibility.

Before spending time on speeding up your code think about the following questions and consider whether it is worth it:

- What is the value of being faster?
- How often is the code going to be run?
- How much extra time will it take to write?
- How much extra complexity will it introduce?

1.3 Where to Optimise

If we have decided we are going to optimise some existing code the first thing we need to do is to decide where to start. There is an awful lot we *could* do to make our code go faster, but we should always keep in mind making it *good enough*. If we can improve the slowest, most frequently used parts then that's where we should start.

1. Take a look at the following example code. Assuming we want to make it go faster, what more information would you need in order to decide where to improve?
2. Without knowing how the functions are written, can you still recommend a good place to start optimising?

```
source("./utils.R")
data <- read.csv("records.csv")
vars <- read.csv("variablesForProcessing.csv")[[1]]
data <- preProcess(data)
result <- numeric()

for (i in 1:nrow(data)) {

  tmp <- applyNormalisation(data[i, ])

  for (var in vars) {
    result <- c(result, doCalculation(data = tmp, var = var))
  }
}

saveRDS(result, file = "result.rds")
```

It is hard to know where to start optimising the code example above, primarily for two reasons:

- We do not know speed characteristics for several of the function calls
- We do not know the size or shape of the data to be processed

Without knowing about the data or functions, an educated guess for where to focus our attention would be the `doCalculation` function. This is because we can see that it is called `nrow(data) * length(vars)` times, so any performance improvements in `doCalculation` will have a multiplicative effect.

Whilst it is a good idea to survey our code as we have in the example above in order to get a good understanding of its structure, to get a more accurate indication of where to focus our efforts we should use a profiler on some typical data to see definitively how long functions take and how often they are repeated.

Chapter 2
The Optimiser's Toolset

2.1 Introduction

Before we can start optimising our code we need a good understanding of its performance characteristics. In this chapter we learn two important tools that will help us profile and accurately time our functions.

2.2 Profiling Code

If we find our code is running slowly we can run a *profiler* to understand which parts are causing the slowdown, so we can focus our attention correctly on optimising our code.

R already has a built-in tool for profiling called `Rprof`, which works by regularly recording which functions are running over time. The **profvis** package provides functionality to visualise the output of `Rprof` and allow us to diagnose slow code more easily.

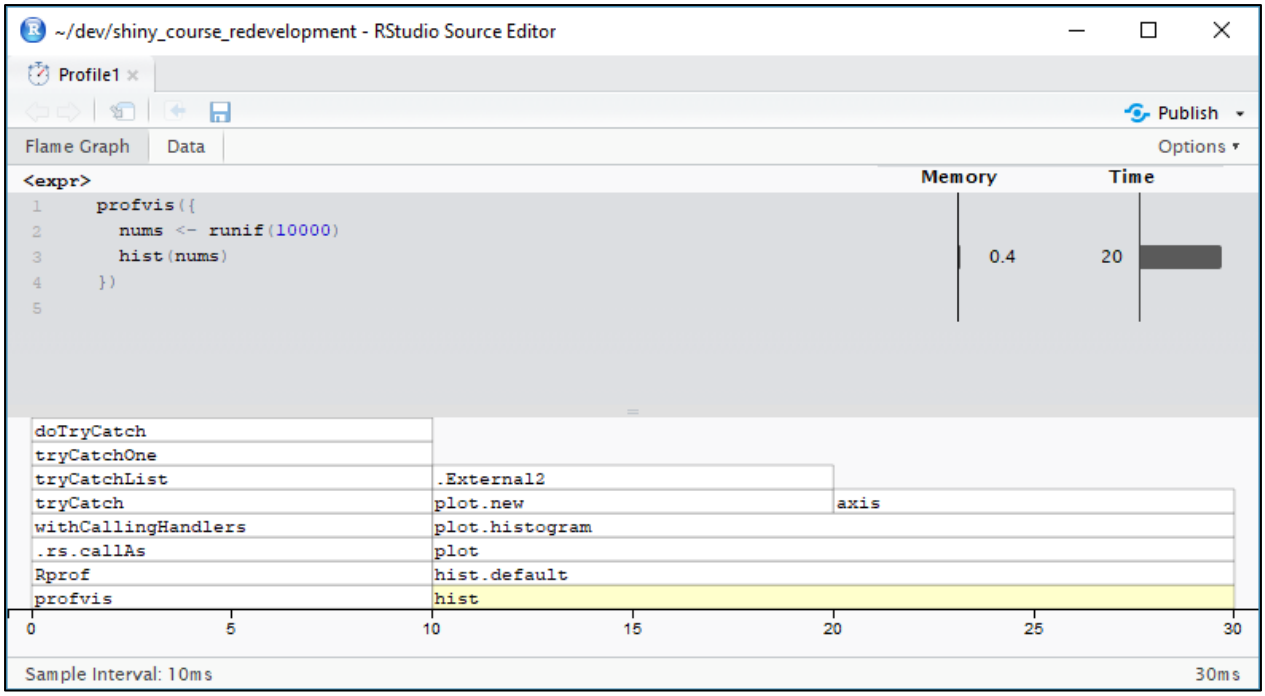
2.2.1 Running the Profiler

In RStudio profiling code is as simple as highlighting the code you want to profile and selecting from the menu Profile Selected Line(s).

We can also run the profiler by directly calling the `profvis` function, passing the code we want to evaluate as the first argument, e.g.

```
library(profvis)
profvis({
  nums <- runif(10000)
  hist(nums)
})
```

Once the code has finished `profvis` will open a code and "flame graph" visualisation that describes how much time was spent on each function call.



The top half of the chart shows bars next to each line of code representing how long each line took to complete. Here we can see that the `hist` function took 20ms to run.

The lower half of the chart shows horizontally how much time was spent on each function call, and shows vertically the callstack for each function.

The underlying data is produced by `Rprof`, which records the currently running function many times a second, but this is only a sample of what was running, so functions that complete very quickly may not even be captured.

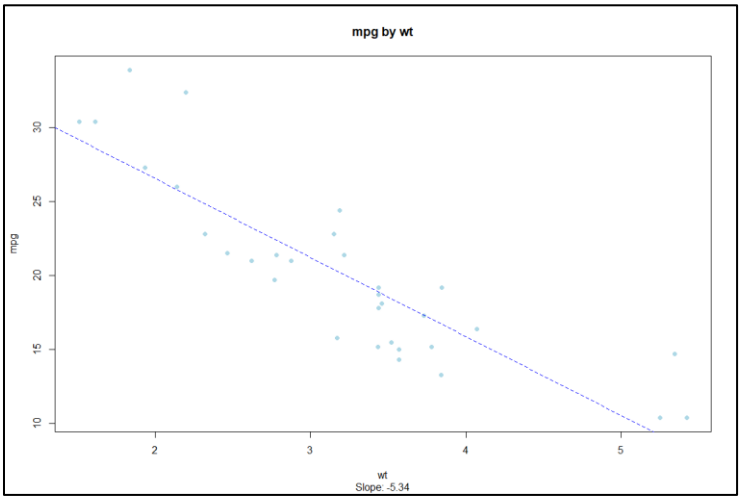
We can see in the example above, `runif` ran so quickly it was not captured by the output. Usually this is not a problem since we are only looking for slow functions!

The profiler also shows the memory usage for each call, which can sometimes be the target for optimisation rather than speed. In this course we are focussed on Time. The Memory output can be turned off using the Options dropdown.

When we are profiling functions we are developing interactively, we need to `source` the script first in order to show profile output for code inside the function.

1. Use the profiler to profile a single call to the `library` function, using a package that is not currently loaded (e.g. `library(ggplot2)`).
2. How long does the call take to run in total?
3. Within the call to `library`, which function occupies the most time?
4. Examine the code below that plots a simple linear regression model and displays the slope of the regression line. Can you guess which part of the code takes longest to run?
5. Find out for sure by profiling a call to the function, e.g.
   `simple_lm_plot(mpg ~ wt, data = mtcars)`

```
simple_lm_plot <- function(formula, data) {

  lm1 <- lm(formula, data = data)

  vars <- all.vars(formula)[1:2]  # get first 2 formula vars

  # make plot and annotations
  plot(lm1$model[rev(vars)], pch = 19, col = "lightblue")
  abline(lm1, col = "blue", lty = 2)
  slope = round(coefficients(lm1)[2], 2)
  title(main = paste(vars, collapse = " by "),
        sub = paste("Slope:", slope))

  # return the model object
  invisible(lm1)

}
```



2.3 Timing Functions Accurately

The profiler is a great tool for deciding where to focus our attention. Once we have identified a candidate function we can use the **microbenchmark** package to get more accurate readings for its execution speed. The `microbenchmark` function provides us with sub-millisecond accuracy with minimal overheads, giving us more accurate timings that R's built-in `system.time` function.

We can use `microbenchmark` on our selected function to generate a performance benchmark which we can then try to improve on.

2.3.1 Using microbenchmark

We can pass any number of expressions to the microbenchmark function. These will each be run several times (the default `times` argument is 100) and a summary of the timings will be output. For example, if we want to compare different sorting methods for a character vector:

```
library(microbenchmark)
chars <- sample(letters, 1e4, replace = TRUE)
res <- microbenchmark(
  sort(chars, method = "quick"),
  sort(chars, method = "radix"),
  chars[order(chars)])
```

Printing the result object will give us a summary of the timing results:

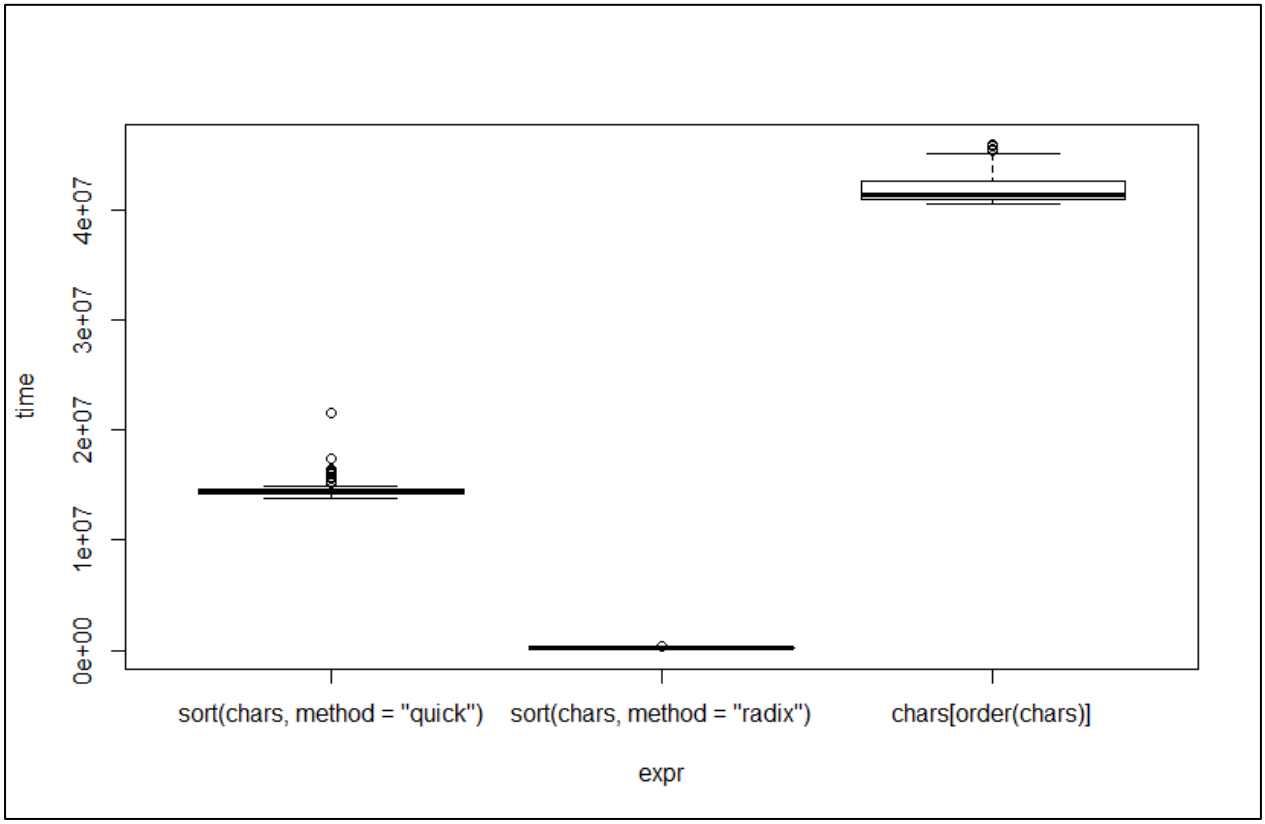
```
> res
Unit: microseconds
                        expr       min         lq       mean     median        uq       max neval cld
 sort(chars, method = "quick") 13835.331 14234.4345 14651.3289 14375.2715 14596.723 21526.660   100  b
 sort(chars, method = "radix")   157.829   190.6785   240.5273   251.8475   268.272   406.278   100 a
          chars[order(chars)] 40585.070 41017.4005 42020.0411 41421.4125 42681.024 45934.641   100   c
```

The median speed is a good metric to keep an eye on, but also pay attention to the spread of the readings as this could indicate erratic behaviour for your function.

With the **multicomp** package installed the output will show an additional "cld" (compact letter display) column giving a quick indication of which expressions differ significantly from each other. Expressions that are not significantly different will share the same letter.

We can also visualise the output calling `plot` on the result object:

```
plot(res)
```



We can see more easily from the plot that the "radix" `sort` method is the fastest, and that using `order` combined with square brackets is the slowest.

Pay attention to function arguments and how they affect performance. Often default arguments cater for flexibility over speed. For example, `order` appears slower, but it is being called with its default arguments in the example above.

1. Use microbenchmark to find the fastest function with which to filter a `data.frame`. Try the extract operator "`[`", the `subset` function, and `dplyr::filter`. Use the `quakes` dataset to test. Which is fastest?
2. Plot the results to see which functions are most consistent.
3. In some scenarious 100 repeats of each expression would take too long. Change the number of times the calls are repeated to 50 and plot the results.

Extension:

4. Does it make a difference to the **dplyr** `filter` results if it is called using the pipe operator? E.g. `quakes %>% filter(mag > 6)` vs. `filter(quakes, mag > 6)`
5. Have you found the fastest way to filter a `data.frame`, or are there any other methods you might use, or other packages?

There are often many functions and approaches to solving problems in R. If you are interested in speed, get into the habit of using `microbenchmark` to test alternative implementations using your own data to inform your choice.

Chapter 3
Optimisation Strategies

3.1 Introduction

In this chapter we will look at some common mistakes that cause R to run more slowly than it should and learn better alternatives. We will also look at choosing the right packages, functions, and data structures in order to achieve higher performance.

3.2 Avoiding Common Mistakes

3.2.1 Avoid Growing Objects

In R, functions are called by "value" rather than by "reference", which means that function calls will return a new object, rather than modifying existing objects in place. For example, in the code below, the `append` function adds the number 11 to the vector `x`, returning a new vector, with `x` being unmodified.

```
> x <- 1:10
> append(x, 11)
 [1]  1  2  3  4  5  6  7  8  9 10 11
> x
 [1]  1  2  3  4  5  6  7  8  9 10
```

This behaviour can make code easier to reason about, particularly when debugging as functions work on their own copies of data and don't have the potential to interfere with each other. However, creating new objects, rather than modifying existing ones, requires some overhead in terms of memory allocation.

3.2.1.1 Growing a Result Set

A frequent data processing task is to iterate over items in a dataset performing a calculation and then adding it to a growing result set. For example, writing a simple function that outputs a numeric vector from one to `n`:

```
make_sequence_grow <- function(n) {
  vec <- numeric()
  for (i in 1:n) vec <- c(vec, i)
  vec
}

> make_sequence_grow(10)
 [1]  1  2  3  4  5  6  7  8  9 10
```

Each iteration of the `for` loop creates a new, slightly larger, object where the result of `c(vec, i)` is copied. This memory allocation overhead is trivial for a few iterations, but adds up as the frequency of iterations increases.

3.2.1.2 Pre-allocating a Result Set

The correct way to implement this pattern is to pre-allocate a result set of appropriate size first, then slot in the results as they are calculated:

```
make_sequence_preallocate <- function(n) {
  vec <- numeric(n)
  for (i in 1:n) vec[i] <- i
  vec
}
```

We can use microbenchmark to examine the relative speeds of each approach:

```
library(microbenchmark)
n <- 1000
res <- microbenchmark(
  make_sequence_grow(n),
  make_sequence_preallocate(n)
)

> print(res, signif = 2)
Unit: microseconds
                          expr  min   lq mean median   uq  max neval
        make_sequence_grow(n) 1100 1200 1700   1200 1300 8500   100
 make_sequence_preallocate(n)   62   63  120     67   71 4400   100
```

We can see that pre-allocating the right size result object has increased the speed by over ten times.

Watch out for this "growing the result set" pattern. It applies to other data structures too, e.g. using `rbind` or `cbind` to append a `data.frame` or `matrix` result set.

3.2.2 Make Use of Vectorisation

Many of R's functions are designed to work with vectors rather than scalar values, for example, the addition function is vectorised – we can pass it two vectors and it will add them together element-wise.

```
x <- 1:1000
y <- 1000:1
x + y
```

This is much faster (and less verbose) than an element-wise approach, e.g.

```
add <- function(x, y) {
  result <- numeric(length(x))
  for (i in seq_along(x)) result[i] <- x[i] + y[i]
  result
}
```

Examining the timings shows the vectorised version is about 40 times faster:

```
> res <- microbenchmark(x + y, add(x, y))
> print(res, signif = 2)
Unit: microseconds
      expr   min    lq  mean median    uq max neval
     x + y   1.5   2.3   3.9      3   4.2  19   100
 add(x, y) 100.0 100.0 160.0    120 220.0 330   100
```

One reason the vectorised version is faster is due to R's flexibility of data types. In the element-wise version each iteration adds a single x value to a single y value and each time the addition function incurs some overhead, for example checking that the x and y values are of data types that make sense to add.

In the vectorised version `x` and `y` are verified as numeric just once, reducing the overhead.

Consult a function's documentation to find out if it accepts vector arguments.

This may come as second nature to us in using some of R's existing functions, but it is easy to drop into element-wise thinking when designing solutions to our own problems.

3.2.3 Don't Reinvent a Slower Wheel

Many existing functions for common processing operations that are highly-optimised for a particular task, and are usually written in C, so run very quickly. If what you are doing seems common, or is composed of a set of common operations, spend some time searching for existing implementations.

3.2.3.1 Sequences

We have written our own sequence generator above to generate an ascending sequence of integers. The built-in `seq` function is highly optimised for this purpose and will perform much faster than our implementation.

As a general rule, more specialised functions will be faster than their generic equivalents, for example, `seq.int` is faster than `seq`, but has a few restrictions. Note, the "int" in `seq.int` stands for "internal" not "integer". Always read the manual for a base function to see if there are more specialised versions.

From the benchmarking below we can see that `seq` is over ten times faster than our implementation, and `seq.int` is several orders of magnitude faster that `seq`.

```
n <- 1000
res <- microbenchmark(make_sequence_preallocate(n),
                      seq(n),
                      seq.int(n))
> print(res, signif = 2)
Unit: nanoseconds
                        expr   min    lq  mean median    uq    max neval
 make_sequence_preallocate(n) 62000 63000 67000  64000 69000 110000   100
                      seq(n)  4900  5300  6800   5900  6400  53000   100
                  seq.int(n)     0     0   170      1   380   3000   100
```

3.2.3.2 Row/Column Operations

A common task is to summarise datasets by row or by column, e.g. to calculate the row sums or averages. There are more general approaches, such as using the apply function, but using more specialised functions, which have been designed for speed, will be faster. The main row/column functions are:

- `colSums`
- `rowSums`
- `colMeans`
- `rowMeans`

For example, we can measure the difference between using `apply`/`mean` versus `colMeans`, and see that the specialised function is about twice as fast:

```
res <- microbenchmark(
  apply(mtcars, 2, mean),
  colMeans(mtcars)
)
> print(res, signif = 2)
Unit: microseconds
                  expr min  lq mean median  uq   max neval
 apply(mtcars, 2, mean) 170 180  450    190 200 25000   100
       colMeans(mtcars)  92  94  100     99 100   170   100
```

1. Write a function that takes a vector of input and using a loop iterates over all of the values calculating the sum up to that value (i.e. the cumulative sum). For example, `myfun(1:10)` returns:

   ```
    1  3  6  10  15  21  28  36  45  55
   ```

2. Determine how long it takes to run your function.
3. Use the initialization and vectorization techniques to improve the speed of your function. Check that this is in fact more efficient.
4. Can you find a function in R that will do this for you? Compare the speed of that function to your most efficient version.

Extension

5. The function below is designed to take a vector of starting values and project each value forward by a number of periods, multiplying the starting value by a growth rate each period. Profile the code.
6. How many times faster can you make it using the techniques in this chapter?

```
#' Project Investments
#' @param startingAmounts numeric vector of starting balances
#' @param growthRate single growth rate, e.g. 1.1 for 10% growth
#' @param nPeriods number of periods to project forwards
#' @return matrix with one row per startingAmount, one col per period
#' @examples projectInvestments(1:100, growth = 1.1, nPeriods = 1000)
#'
projectInvestments <- function(startingAmounts, growthRate, nPeriods)
{
  result <- numeric()
  for (startingAmount in startingAmounts) {
    # Use startingAmount as the base value and create next period
    # by multiplying by the growthRate
    projectedAmounts <- startingAmount
    for (i in seq_len(nPeriods)) {
      projectedAmounts[i + 1] <- projectedAmounts[i] * growthRate
    }
    result <- rbind(result, projectedAmounts)
  }
  unname(result)
}
```

3.3 For Loops

It is a myth that looping over data structures using `for` is slower than other iteration approaches, such as `lapply`, or `purrr::map`.

Using a `for` loop is not inherently slower, and can sometimes be the best approach, but they do have some downsides:

- Using for loops encourages us to think about problems per element instead of using vectorised functions
- Using for loops makes it easy to apply bad practice such as growing result sets

Provided we can avoid falling into these traps there is not a significant difference in performance. The main benefit to alternative iteration methods, such as `apply` or `map`, is that they encourage us to think at a higher level, rather than focusing on mechanical details. For example, it is common to need to apply a function to a list of elements.

With a `for` loop we explicitly tell R to:

- "Generate a list of indices
- Take the first element, apply the function, add it to the results
- Take the next element, and apply the function…"

```
result <- numeric(length(x))
for (i in seq_along(x)) {
  result[i] <- aFunction(x[i])
}
result
```

With `purrr::map` or `apply` it is more like saying "Apply this to those". It's more succinct and allows us to think and code faster (even if the code doesn't run any faster).

```
purrr::map(x, aFunction)
```

Note, `map` and `apply` do not specify the ordering for processing. This has benefits when we consider parallel processing, with potentially each element of `x` being processed by a different computer and the results combined afterwards. Where the ordering of processing is important, e.g. where a process depends on the previous result, `for` loops are still useful.

3.4 Choosing the Right Data Structures

So far we have seen a recurring theme: the more flexible a function is, usually the slower it is. The same rule of thumb holds for data structures. Therefore, if we can solve our problem using more primitive data structures we should glean a performance boost.

A `data.frame` is a `list` with an extra metadata and a few additional rules, such as each element of the `list` (each column in the `data.frame`) must be the same length and data type. These rules are checked whenever we manipulate the `data.frame`, so slow things down:

```
> mtcars$newCol <- 1:100
Error in `$<-.data.frame`(`*tmp*`, newCol, value = 1:100) :
  replacement has 100 rows, data has 32
```

We can often trade off some of the flexibility for speed. A common case is when we have a `data.frame` with all the same data type, which could be represented as a more primitive `matrix`.

For example we can create equivalent `data.frame` and `matrix` objects and time how long each takes to be updated and to be computed on.

```
df1 <- data.frame(A = seq_len(1e3), B = 0)
# Convert to matrix
mt1 <- data.matrix(df1)

# Define an updating function
updateColumn <- function(x) {
  for (i in seq_len(nrow(x))) {
    x[i, "B"] <- i
  }
  x
}

res <- microbenchmark(
  updateColumn(df1), updateColumn(mt1),
  rowSums(df1), rowSums(mt1))
> print(res, signif = 2)
Unit: microseconds
              expr     min    lq  mean median    uq   max neval
 updateColumn(df1) 22000.0 24000 26000  25000 27000 50000   100
 updateColumn(mt1)   270.0   290   360    310   330  4000   100
      rowSums(df1)    63.0    77   120    140   150   490   100
      rowSums(mt1)     9.8    12    20     15    27    84   100
```

We can see that both our `updateColumn` function and the built-in `rowSums` function are many times faster when working with matrices.

3.5 Using Existing Implementations Effectively

One of the strengths of R is the huge community and multitude of extension packages. This means that for common tasks, either base R has an optimised function readily available, or somebody has already written one in a package. We should get to know the features of existing implementations and familiarise ourselves with their documentation in order to use them most effectively.

3.5.1 Reading Tables

For reading text data, `read.csv` is the standard function, but is considered slow compared to other implementations. However, this function is very flexible and can be made to go faster if we give it more information. For example, `read.csv` infers the data types of columns by reading in the first 1000 rows of data and applying some heuristics. Usually we know the intended data types, so we can provide this in the `colClasses` argument.

In the example below we create a dummy CSV file and read it in with and without the `colClasses` argument to test the difference in speed:

```
dat <- as.data.frame(matrix(rnorm(1e6), ncol = 20))
dat[,1] = "a"
headings <- LETTERS[1:20]
write.table(dat,
            file = "dat.csv",
            sep = ",",
            col.names = headings,
            row.names = FALSE)

res <- microbenchmark(times = 10, list = alist(
  withoutCols = read.csv("dat.csv", header = TRUE),
  withCols = read.csv("dat.csv", header = TRUE,
                      colClasses = c("character", rep("numeric",19)))
))

> print(res, signif = 2)
Unit: milliseconds
        expr  min   lq mean median   uq  max neval
 withoutCols 1000 1100 1300   1300 1400 2000    10
    withCols  490  520  590    540  620  920    10
```

In this case we have more than doubled the speed of the function by making it do less work.

In the example above, we are calling `microbenchmark` providing the expressions as a named list of unevaluated expressions using the `alist` function, which we pass to the `list` argument. This is a convenient way to provide longer expressions whilst maintaining a short printout of the results.

3.5.1.1 Other Packages

The two main alternatives to `read.csv` are `read_csv` from the **readr** package, and `fread` from the **data.table** package, which are usually significantly faster.

```
res <- microbenchmark(times = 10,
                      read.csv("dat.csv"),
                      readr::read_csv("dat.csv"),
                      data.table::fread("dat.csv")
)
> print(res, signif = 2)
Unit: milliseconds
                         expr  min   lq mean median   uq  max neval
          read.csv("dat.csv") 1100 1300 1400   1400 1500 1600    10
   readr::read_csv("dat.csv")  200  230  420    390  510  780    10
 data.table::fread("dat.csv")   33   34   45     35   39  100    10
```

For this example data we can see that `read_csv` is much faster than `read.csv`, and `fread` faster still. Bear in mind that these functions are not completely equivalent, for example `read.csv` will convert strings to factors by default whereas the other functions will not.

Just as with read.csv before, we should be able to make further speed improvements to `read_csv` or `fread` by studying their documentation.

Get to know R's existing functions in detail by studying their documentation. Providing the right argument to an existing function is often the easiest way to obtain a performance boost.

3.5.2 Other Common Cases

The table below illustrates some common tasks and some potential functions/packages to investigate in order to improve performance. However, this is just a starting point, you are encouraged to research other existing solutions!

Task	Useful packages	Tips
Read text files	`readr, data.table, feather`	
Manipulate data.frames	`dplyr, data.table`	**data.table** is often faster than **dplyr**, but its terse syntax is less readable than **dplyr**.
Sort data structures	`base, data.table`	Consult the `base::sort` documentation for faster sort options.
Write data	`readr, data.table, feather`	Pay attention to function defaults, e.g. compression can be turned on/off for a filespace/speed trade-off.
If/else conditions	`dplyr`	`dplyr::if_else` is faster than base::if_else. For non-vectorised scenarios `base::if` is faster.
Time series	`xts`	The **xts** package extends functionality provided by **base** and **zoo**, and is often much faster.

There is usually no definitive answer in terms of which package/function is fastest. The only way to know for sure is to benchmark functions using your own data.

1. Create a matrix with one million elements and 20 columns. Compare the speed of saving the file in ".Rds" format between `base::saveRDS` and `readr::write_rds`. Which function is faster and why?
2. The `createRefs` function below generates random ten-letter reference codes, returning them as a sorted character vector.
   a. Benchmark the existing function, recording its typical speed when n = 1000?
   b. What is the slowest part of the function?
   c. Using the techniques in this chapter, how many times faster can you make the function?

```
#' Create References
#' Creates sorted random 10-letter reference codes
#' @param nRef Number of references to generate
#' @return sorted (ascending) character vector of 10-letter codes
createRefs <- function(nRef = 1000) {
  result <- data.frame()
  for (i in seq_len(nRef)) {

    # Generate a ten-letter reference code
    ref <- character()
    for (j in seq_len(10)) {
      ref <- paste0(ref, LETTERS[floor(runif(1, min = 0, max = 27))])
    }
    result <- rbind(result, data.frame(ref = ref,
                                       stringsAsFactors = FALSE))
  }

  result <- result[order(result$ref), ]
  return(result)
}
```

Chapter 4
Parallel Computing

4.1 When to Use Parallel Computing

When computing in parallel we break our problem down into discrete pieces and process each piece on a separate processor at the same time. Some problems are well-suited to parallel processing whereas others are not.

For example, playing a game of chess is not well-suited to parallel processing; each move has to happen sequentially and we can't move multiple pieces at once! However, we could parallelise a computer's calculation of the next move, perhaps running multiple independent simulations of potential moves and selecting the one with the best outcome.

Parallel computing works best when our problems are “embarrassingly parallel”, meaning that there are no interdependencies between our data, so we can break them up into pieces, process them separately, and bring the result together at the end. In R, these are often the same problems that we can use `apply`, or `map`, functions for. For example:

- Producing a summary statistic for each group in a dataset
- Repeatedly sampling a dataset with replacement and computing a statistic (known as “bootstrapping”)

We will look at an example of an embarrassingly parallel problem for a password-guessing utility. Passwords are usually stored as the result of a one-way hashing algorithm, for example the SHA1 algorithm:

```
> digest::sha1("mypassword")
[1] "16e663f3783feac03eac667f409aaaae21c16f82"
```

We can’t determine the actual password from the hashed value. When we log in, the password we provide is passed through the hashing algorithm and compared to the stored hashed values – if they match then we are authenticated.

If we know the hashed value but don’t know the password, one approach to retrieving it is a “brute force” approach, hashing every possible password and checking to see if it matches. Each potential password can be checked independently of the others so this problem will parallelise well.

For a simplified example we will use a five-digit numeric PIN rather than a password. We know the hashed value is “600272d88d7961888d606497e8380280c2e95314”. The code below shows the serial version to compute all possible PIN hashes and recover the original PIN.

```
> library(digest)
> hashedPIN <- "600272d88d7961888d606497e8380280c2e95314"
>
> findPIN <- function(hashedPIN) {
+   potentialPINs <- formatC(seq.int(0,99999),
+                            width = 5,format = "d", flag= "0")
+   allHashes <- lapply(potentialPINs, FUN = sha1)
+   allHashes <- unlist(allHashes)
+   names(allHashes) <- allHashes
+   return(names(allPINs[which(allPINs == hashedPIN)]))
+ }
```

The `findPIN` function will take a few seconds to complete, depending on your computer's processor. In this version each hash is calculated sequentially, but since each calculations is independent we could process them in parallel to find the PIN faster. We will use the **parallel** package to divide the processing task between our computer's processor cores.

4.2 Creating a Cluster

When we typically run R it only uses a single core on your computer. However, most computers now have multiple cores available. By using more of them, we can reduce our processing time. However, there will still be some overheads to set up the R session in each core and passing data back and forth so it's not as simple as half the time for each extra core.

To create a cluster on our own machines we first need to determine the number of cores available to us. We can then create a cluster that uses some, or all, of those cores.

```
> library(parallel)
> nCores <- detectCores()
> cl <- makeCluster(nCores)
```

If you are using the computer for other tasks whilst performing your parallel calculations you should allocate one fewer than what is returned by `detectCores`, e.g.

```
cl <- makeCluster(detectCores() - 1)
```

4.3 Adapting Code for Parallel Processing

Once we have our cluster set up we need to modify our code so that it takes advantage of all of the clusters. If we are already using an apply function, this is relatively straightforward as there are a number of parallel equivalents.
The parallel version of `lapply`, is the `parLapply` function. `parLapply` will automatically distribute our data processing across however many cores are in our cluster.

The arguments to `parLapply` are slightly different to lapply. The first argument is the cluster, and there are some inconsistencies between argument cases:

```
lapply(X, FUN, …)
parLapply(cl, X, fun, …)
```

The parallelised version of our function becomes:

```
> parFindPIN <- function(cl, hashedPIN) {
>   potentialPINs <- formatC(seq.int(0, 99999),
+                            width = 5, format = "d", flag = "0")
+   allHashes <- parLapply(cl = cl, X = potentialPINs, fun = sha1)
+   allHashes <- unlist(allHashes)
+   names(allHashes) <- allHashes
+   return(names(allPINs[which(allPINs == hashedPIN)]))
+ }
```

Just like with all our other code we can test the performance difference with `microbenchmark`:

```
> res <- microbenchmark(times = 3,
+                findPIN(hashedPIN),
+                parFindPIN(cl, hashedPIN))

> print(res, signif = 2)
Unit: seconds
                     expr min  lq mean median  uq max neval
        findPIN(hashedPIN) 6.9 7.0  7.3    7.1 7.5 7.9     3
 parFindPIN(cl, hashedPIN) 3.5 3.5  3.6    3.6 3.7 3.8     3
```

The benchmark results were obtained using a cluster of four cores, but we did not achieve a speedup of four times. As mentione above, this is because it takes time to initialise the cluster, partition the data, and combine the results. However, we did double our performance with only a few extra lines of code.

4.4 Pre-Configuring a Cluster

In the example above we used a function from the **digest** package during the parallel processing to calculate hash values. The `parLapply` function automatically passed this function across to each sub-process before starting to perform any calculations. This is part of the overhead to initialise the cluster. But what if we needed to pass other values, or there were a number of packages that needed to be loaded?

We can do all of this before we run the parallel functions. Suppose we wanted to compare the `hashedPIN` to the calculated hash on the cluster. We would need to export the `hashedPIN` object as well as the `sha1` function to each R process.

```
> clusterExport(cl, varlist = c("hashedPIN", "sha1"))
```

We can also use `clusterEvalQ` to evaluate a series of lines of code, such as loading entire packages.

```
> clusterEvalQ(cl, {
+  library(digest)
+  })
```

4.5 Stopping a Cluster

Just like when we connect to a database, once we have finished our processing it is good practice to shut down the cluster:

```
> stopCluster(cl)
```

4.6 Types of Cluster

The type of cluster that we have used here is a “sockets” or PSOCK cluster. This type of cluster creates a new R session on each core, with each session communicteing via sockets. The benefit of this approach is that it works with any operating system and even across networks of computers. The trade-off for is that we have had to perform additional steps, essentially to set up the cluster and ensure that we have passed all of the objects that we require, and the inter-process communication is slower.

However, if you are using Mac or Linux based systems there is an alternative, forking. This approach simply copies your entire R session to each available core. This approach is generally faster, although more restrictive as it is not available to Windows users. To make use of this approach take a look at the `mclapply` function, in the **parallel** package.