16 Performance

R is not a fast language. This is not an accident. R was purposely designed to make data analysis and statistics easier for you to do. It was not designed to make life easier for your computer. While R is slow compared to other programming languages, for most purposes, it's fast enough.

The goal of this part of the book is to give you a deeper understanding of R's performance characteristics. In this chapter, you'll learn about some of the trade-offs that R has made, valuing flexibility over performance. The following four chapters will give you the skills to improve the speed of your code when you need to:

- In Chapter 17, you'll learn how to systematically make your code faster. First you figure what's slow, and then you apply some general techniques to make the slow parts faster.
- In Chapter 18, you'll learn about how R uses memory, and how garbage collection and copy-on-modify affect performance and memory usage.
- For really high-performance code, you can move outside of R and use another programming language. Chapter 19 will teach you the absolute minimum you need to know about C++ so you can write fast code using the Rcpp package.
- To really understand the performance of built-in base functions, you'll need to learn a little bit about R's C API. In Chapter 20, you'll learn a little about R's C internals.

Let's get started by learning more about why R is slow.

16.1 Why is R slow?

To understand R's performance, it helps to think about R as both a language and as an implementation of that language. The R-language is abstract: it defines what R code means and how it should work. The implementation is concrete: it reads R code and computes a result. The most popular implementation is the one from r-project.org (`http://r-project.org`). I'll call that implementation GNU-R to distinguish it from R-language, and from the other implementations I'll discuss later in the chapter.

The distinction between R-language and GNU-R is a bit murky because the R-language is not formally defined. While there is the R language definition (`http://cran.r-project.org/doc/manuals/R-lang.html`), it is informal and incomplete. The R-language is mostly defined in terms of how GNU-R works. This is in contrast to other languages, like C++ (`http://isocpp.org/std/the-standard`) and javascript (`http://www.ecma-international.org/publications/standards/Ecma-262.htm`), that make a clear distinction between language and implementation by laying out formal specifications that describe in minute detail how every aspect of the language should work. Nevertheless, the distinction between R-language and GNU-R is still useful: poor performance due to the language is hard to fix without breaking existing code; fixing poor performance due to the implementation is easier.

In Section 16.3, I discuss some of the ways in which the design of the R-language imposes fundamental constraints on R's speed. In Section 16.4, I discuss why GNU-R is currently far from the theoretical maximum, and why improvements in performance happen so slowly. While it's hard to know exactly how much faster a better implementation could be, a >10x improvement in speed seems achievable. In Section 16.5, I discuss some of the promising new implementations of R, and describe one important technique they use to make R code run faster.

Beyond performance limitations due to design and implementation, it has to be said that a lot of R code is slow simply because it's poorly written. Few R users have any formal training in programming or software development. Fewer still write R code for a living. Most people use R to understand data: it's more important to get an answer quickly than to develop a system that will work in a wide variety of situations. This means that it's relatively easy to make most R code much faster, as we'll see in the following chapters.

Before we examine some of the slower parts of the R-language and GNU-R, we need to learn a little about benchmarking so that we can give our intuitions about performance a concrete foundation.

16.2 Microbenchmarking

A microbenchmark is a measurement of the performance of a very small piece of code, something that might take microseconds (Âţs) or nanoseconds (ns) to run. I'm going to use microbenchmarks to demonstrate the performance of very low-level pieces of R code, which help develop your intuition for how R works. This intuition, by-and-large, is not useful for increasing the speed of real code. The observed differences in microbenchmarks will typically be dominated by higher-order effects in real code; a deep understanding of subatomic physics is not very helpful when baking. Don't change the way you code because of these microbenchmarks. Instead wait until you've read the practical advice in the following chapters.

The best tool for microbenchmarking in R is the microbenchmark (http://cran.r-project.org/web/packages/microbenchmark/) package. It provides very precise timings, making it possible to compare operations that only take a tiny amount of time. For example, the following code compares the speed of two ways of computing a square root.

```
library(microbenchmark)

x <- runif(100)
microbenchmark(
  sqrt(x),
  x ^ 0.5
)
#> Unit: nanoseconds
#>     expr   min    lq median    uq    max neval
#>  sqrt(x)   592   612    666   703  6,460   100
#>    x^0.5 3,780 3,860  3,920 4,050 33,500   100
```

By default, `microbenchmark()` runs each expression 100 times (controlled by the `times` parameter). In the process, it also randomises the order of the expressions. It summarises the results with a minimum (`min`), lower quartile (`lq`), median, upper quartile (`uq`), and maximum (`max`). Focus on the median, and use the upper and lower quartiles (`lq` and `uq`) to get a feel for the variability. In this example, you can see that using the special purpose `sqrt()` function is faster than the general exponentiation operator.

As with all microbenchmarks, pay careful attention to the units: each computation takes about 800 ns, 800 billionths of a second. To help calibrate the impact of a microbenchmark on run time, it's useful to think about how many times a function needs to run before it takes a second. If a microbenchmark takes:

- 1 ms, then one thousand calls takes a second
- 1 Âţs, then one million calls takes a second
- 1 ns, then one billion calls takes a second

The `sqrt()` function takes about 800 ns, or 0.8 Âţs, to compute the square root of 100 numbers. That means if you repeated the operation a million times, it would take 0.8 s. So changing the way you compute the square root is unlikely to significantly affect real code.

16.2.1 Exercises

1. Instead of using `microbenchmark()`, you could use the built-in function `system.time()`. But `system.time()` is much less precise, so you'll need to repeat each operation many times with a loop, and then divide to find the average time of each operation, as in the code below.

   ```
   n <- 1:1e6
   system.time(for (i in n) sqrt(x)) / length(n)
   system.time(for (i in n) x ^ 0.5) / length(n)
   ```

   How do the estimates from `system.time()` compare to those from `microbenchmark()`? Why are they different?

2. Here are two other ways to compute the square root of a vector. Which do you think will be fastest? Which will be slowest? Use microbenchmarking to test your answers.

   ```
   x ^ (1 / 2)
   exp(log(x) / 2)
   ```

3. Use microbenchmarking to rank the basic arithmetic operators (`+`, `-`, `*`, `/`, and `^`) in terms of their speed. Visualise the results. Compare the speed of arithmetic on integers vs. doubles.
4. You can change the units in which the microbenchmark results are expressed with the `unit` parameter. Use `unit = "eps"` to show the number of evaluations needed to take 1 second. Repeat the benchmarks above with the eps unit. How does this change your intuition for performance?

16.3 Language performance

In this section, I'll explore three trade-offs that limit the performance of the R-language: extreme dynamism, name lookup with mutable environments, and lazy evaluation of function arguments. I'll illustrate each trade-off with a microbenchmark, showing how it slows GNU-R down. I benchmark GNU-R because you can't benchmark the R-language (it can't run code). This means that the results are only suggestive of the cost of these design decisions, but are nevertheless useful. I've picked these three examples to illustrate some of the trade-offs that are key to language design: the designer must balance speed, flexibility, and ease of implementation.

If you'd like to learn more about the performance characteristics of the R-language and how they affect real code, I highly recommend "Evaluating the Design of the R Language" (`http://r.cs.purdue.edu/pub/ecoop12.pdf`) by Floreal Morandat, Brandon Hill, Leo Osvald, and Jan Vitek. It uses a powerful methodology that combines a modified R interpreter and a wide set of code found in the wild.

16.3.1 Extreme dynamism

R is an extremely dynamic programming language. Almost anything can be modified after it is created. To give just a few examples, you can:

- Change the body, arguments, and environment of functions.
- Change the S4 methods for a generic.
- Add new fields to an S3 object, or even change its class.
- Modify objects outside of the local environment with `<<-`.

Pretty much the only things you can't change are objects in sealed namespaces, which are created when you load a package.

The advantage of dynamism is that you need minimal upfront planning. You can change your mind at any time, iterating your way to a solution without having to start afresh. The disadvantage of dynamism is that it's difficult to predict exactly what will happen with a given function call. This is a problem because the easier it is to predict what's going to happen, the easier it is for an interpreter or compiler to make an optimisation. (If you'd like more details, Charles Nutter expands on this idea at On Languages, VMs, Optimization, and the Way of the World (http://blog.headius.com/2013/05/on-languages-vms-optimization-and-way.html).) If an interpreter can't predict what's going to happen, it has to consider many options before it finds the right one. For example, the following loop is slow in R, because R doesn't know that `x` is always an integer. That means R has to look for the right `+` method (i.e., is it adding doubles, or integers?) in every iteration of the loop.

```
x <- 0L
for (i in 1:1e6) {
  x <- x + 1
}
```

The cost of finding the right method is higher for non-primitive functions. The following microbenchmark illustrates the cost of method dispatch for S3, S4, and RC. I create a generic and a method for each OO system, then call the generic and see how long it takes to find and call the method. I also time how long it takes to call the bare function for comparison.

```
f <- function(x) NULL

s3 <- function(x) UseMethod("s3")
s3.integer <- f

A <- setClass("A", representation(a = "list"))
setGeneric("s4", function(x) standardGeneric("s4"))
setMethod(s4, "A", f)

B <- setRefClass("B", methods = list(rc = f))

a <- A()
b <- B$new()
```

```
microbenchmark(
  fun = f(),
  S3 = s3(1L),
  S4 = s4(a),
  RC = b$rc()
)
#> Unit: nanoseconds
#>  expr   min     lq median     uq     max neval
#>   fun   147    194    226    264     709   100
#>    S3 2,050  2,500  2,810  3,130  16,400   100
#>    S4 9,570 11,200 12,000 12,700  86,200   100
#>    RC 9,480 10,700 11,400 12,000 526,000   100
```

On my computer, the bare function takes about 200 ns. S3 method dispatch takes an additional 2,000 ns; S4 dispatch, 11,000 ns; and RC dispatch, 10,000 ns. S3 and S4 method dispatch are expensive because R must search for the right method every time the generic is called; it might have changed between this call and the last. R could do better by caching methods between calls, but caching is hard to do correctly and a notorious source of bugs.

16.3.2 Name lookup with mutable environments

It's surprisingly difficult to find the value associated with a name in the R-language. This is due to combination of lexical scoping and extreme dynamism. Take the following example. Each time we print `a` it comes from a different environment:

```
a <- 1
f <- function() {
  g <- function() {
    print(a)
    assign("a", 2, envir = parent.frame())
    print(a)
    a <- 3
    print(a)
  }
  g()
}
f()
#> [1] 1
#> [1] 2
#> [1] 3
```

This means that you can't do name lookup just once: you have to start from scratch each time. This problem is exacerbated by the fact that almost every operation is a lexically scoped function call. You might think the following simple function calls two functions: `+` and `^`. In fact, it calls four because `{` and `(` are regular functions in R.

```
f <- function(x, y) {
  (x + y) ^ 2
}
```

Since these functions are in the global environment, R has to look through every environment in the search path, which could easily be 10 or 20 environments. The following microbenchmark hints at the performance costs. We create four versions of `f()`, each with one more environment (containing 26 bindings) between the environment of `f()` and the base environment where `+`, `^`, `(`, and `{` are defined.

```
random_env <- function(parent = globalenv()) {
  letter_list <- setNames(as.list(runif(26)), LETTERS)
  list2env(letter_list, envir = new.env(parent = parent))
}
set_env <- function(f, e) {
  environment(f) <- e
  f
}
f2 <- set_env(f, random_env())
f3 <- set_env(f, random_env(environment(f2)))
f4 <- set_env(f, random_env(environment(f3)))

microbenchmark(
  f(1, 2),
  f2(1, 2),
  f3(1, 2),
  f4(1, 2),
  times = 10000
)
#> Unit: nanoseconds
#>      expr min  lq median  uq     max neval
#>   f(1, 2) 566 613    693 828  19,000 10000
#>  f2(1, 2) 606 661    741 877  22,700 10000
#>  f3(1, 2) 654 700    783 918 925,000 10000
#>  f4(1, 2) 695 748    829 963 823,000 10000
```

Each additional environment between `f()` and the base environment makes the function slower by about 30 ns.

It might be possible to implement a caching system so that R only needs to look up the value of each name once. This is hard because there are so many ways to change the value associated with a name: `<<-`, `assign()`, `eval()`, and so on. Any caching system would have to know about these functions to make sure the cache was correctly invalidated and you didn't get an out-of-date value.

Another simple fix would be to add more built-in constants that you can't override. This, for example, would mean that R always knew exactly what `+`, `-`, `{`, and `(` meant, and you wouldn't have to repeatedly look up their definitions. That would make the interpreter more complicated (because there are more special cases) and hence harder to maintain, and the language less flexible. This would change the R-language, but it would be unlikely to affect much existing code because it's such a bad idea to override functions like `{` and `(`.

16.3.3 Lazy evaluation overhead

In R, function arguments are evaluated lazily (as discussed in Section 6.4.4 and Section 13.1). To implement lazy evaluation, R uses a promise object that contains the expression needed to compute the result and the environment in which to perform the computation. Creating these objects has some overhead, so each additional argument to a function decreases its speed a little.

The following microbenchmark compares the runtime of a very simple function. Each version of the function has one additional argument. This suggests that adding an additional argument slows the function down by ~20 ns.

```
f0 <- function() NULL
f1 <- function(a = 1) NULL
f2 <- function(a = 1, b = 1) NULL
f3 <- function(a = 1, b = 2, c = 3) NULL
f4 <- function(a = 1, b = 2, c = 4, d = 4) NULL
f5 <- function(a = 1, b = 2, c = 4, d = 4, e = 5) NULL
```

```
microbenchmark(f0(), f1(), f2(), f3(), f4(), f5(), times = 10000)
#> Unit: nanoseconds
#>   expr min  lq median  uq     max neval
#>   f0() 118 139    145 171  28,800 10000
#>   f1() 142 164    172 222 123,000 10000
#>   f2() 162 185    192 298  22,700 10000
#>   f3() 185 209    219 387  52,200 10000
#>   f4() 202 228    260 436  42,100 10000
#>   f5() 224 249    308 491 957,000 10000
```

In most other programming languages there is little overhead for adding extra arguments. Many compiled languages will even warn you if arguments are never used (like in the above example), and automatically remove them from the function.

16.3.4 Exercises

1. `scan()` has the most arguments (21) of any base function. About how much time does it take to make 21 promises each time scan is called? Given a simple input (e.g., `scan(text = "1 2 3", quiet = T)`) what proportion of the total run time is due to creating those promises?
2. Read "Evaluating the Design of the R Language" (http://r.cs.purdue.edu/pub/ecoop12.pdf). What other aspects of the R-language slow it down? Construct microbenchmarks to illustrate.
3. How does the performance of S3 method dispatch change with the length of the class vector? How does performance of S4 method dispatch change with number of superclasses? How about RC?
4. What is the cost of multiple inheritance and multiple dispatch on S4 method dispatch?
5. Why is the cost of name lookup less for functions in the base package?

16.4 Implementation performance

The design of the R language limits its maximum theoretical performance, but GNU-R is currently nowhere near that maximum. There are many things that can (and will) be done to improve performance. This section discusses some aspects of GNU-R that are slow not because of their definition, but because of their implementation.

R is over 20 years old. It contains nearly 800,000 lines of code (about 45% C, 19% R, and 17% Fortran). Changes to base R can only be made by members of the R Core Team (or R-core for short). Currently R-core has twenty members (http://www.r-project.org/contributors.html), but only six are active in day-to-day development. No one on R-core works full time on R. Most are statistics professors who can only spend a relatively small amount of their time on R. Because of the care that must be taken to avoid breaking existing code, R-core tends to be very conservative about accepting new code. It can be frustrating to see R-core reject proposals that would improve performance. However, the overriding concern for R-core is not to make R fast, but to build a stable platform for data analysis and statistics.

Below, I'll show two small, but illustrative, examples of parts of R that are currently slow but could, with some effort, be made faster. They are not critical parts of base R, but they have been sources of frustration for me in the past. As with all microbenchmarks, these won't affect the performance of most code, but can be important for special cases.

16.4.1 Extracting a single value from a data frame

The following microbenchmark shows seven ways to access a single value (the number in the bottom-right corner) from the built-in `mtcars` dataset. The variation in performance is startling: the slowest method takes 30x longer than the fastest. There's no reason that there has to be such a huge difference in performance. It's simply that no one has had the time to fix it.

```
microbenchmark(
  "[32, 11]"      = mtcars[32, 11],
  "$carb[32]"     = mtcars$carb[32],
  "[[c(11, 32)]]" = mtcars[[c(11, 32)]],
  "[[11]][32]"    = mtcars[[11]][32],
  ".subset2"      = .subset2(mtcars, 11)[32]
)
#> Unit: nanoseconds
#>            expr    min     lq median     uq     max neval
#>       [32, 11] 17,500 18,100 18,600 19,400 104,000   100
#>      $carb[32]  9,100 10,200 10,700 11,400  49,500   100
#>  [[c(11, 32)]]  7,540  8,210  8,720  9,330 500,000   100
#>     [[11]][32]  7,040  7,880  8,200  8,790  21,100   100
#>       .subset2    229    440    494    552   2,940   100
```

16.4.2 ifelse(), pmin(), and pmax()

Some base functions are known to be slow. For example, take the following three implementations of `squish()`, a function that ensures that the smallest value in a vector is at least `a` and its largest value is at most `b`. The first implementation, `squish_ife()`, uses `ifelse()`. `ifelse()` is known to be slow because it is relatively general and must evaluate all arguments fully. The second implementation, `squish_p()`, uses `pmin()` and `pmax()`. Because these two functions are so specialised, one might expect that they would be fast. However, they're actually rather slow. This is because they can take any number of arguments and they have to do some relatively complicated checks to determine which method to use. The final implementation uses basic subassignment.

```
squish_ife <- function(x, a, b) {
  ifelse(x <= a, a, ifelse(x >= b, b, x))
}
squish_p <- function(x, a, b) {
  pmax(pmin(x, b), a)
}
squish_in_place <- function(x, a, b) {
  x[x <= a] <- a
  x[x >= b] <- b
  x
}

x <- runif(100, -1.5, 1.5)
microbenchmark(
  squish_ife      = squish_ife(x, -1, 1),
  squish_p        = squish_p(x, -1, 1),
  squish_in_place = squish_in_place(x, -1, 1),
  unit = "us"
)
#> Unit: microseconds
#>             expr   min    lq median    uq   max neval
#>       squish_ife 73.40 86.40   89.2 114.0 212.0   100
#>         squish_p 18.90 21.90   24.3  35.4 465.0   100
#>  squish_in_place  7.92  9.73   11.1  14.5  62.9   100
```

Using `pmin()` and `pmax()` is about 3x faster than `ifelse()`, and using subsetting directly is about twice as fast again. We can often do even better by using C++. The following example compares the best R implementation to a relatively simple, if verbose, implementation in C++. Even if you've never used C++, you should still be able to follow the basic strategy: loop over every element in the vector and perform a different action depending on whether or not the value is less than `a` and/or greater than `b`. The C++ implementation is around 3x faster than the best pure R implementation.

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector squish_cpp(NumericVector x, double a, double b) {
  int n = x.length();
  NumericVector out(n);

  for (int i = 0; i < n; ++i) {
    double xi = x[i];
    if (xi < a) {
      out[i] = a;
    } else if (xi > b) {
      out[i] = b;
    } else {
      out[i] = xi;
    }
  }

  return out;
}
```

(You'll learn how to access this C++ code from R in Chapter 19.)

```
microbenchmark(
  squish_in_place = squish_in_place(x, -1, 1),
  squish_cpp      = squish_cpp(x, -1, 1),
  unit = "us"
)
#> Unit: microseconds
#>             expr  min   lq median   uq  max neval
#>   squish_in_place 7.52 7.81   8.04 8.51 40.3   100
#>        squish_cpp 2.46 2.72   2.88 3.07 38.1   100
```

16.4.3 Exercises

1. The performance characteristics of `squish_ife()`, `squish_p()`, and `squish_in_place()` vary considerably with the size of `x`. Explore the differences. Which sizes lead to the biggest and smallest differences?
2. Compare the performance costs of extracting an element from a list, a column from a matrix, and a column from a data frame. Do the same for rows.

16.5 Alternative R implementations

There are some exciting new implementations of R. While they all try to stick as closely as possible to the existing language definition, they improve speed by using ideas from modern interpreter design. The four most mature open-source projects are:

- pqR (`http://www.pqr-project.org/`) (pretty quick R) by Radford Neal. Built on top of R 2.15.0, it fixes many obvious performance issues, and provides better memory management and some support for automatic multithreading.
- Renjin (`http://www.renjin.org/`) by BeDataDriven. Renjin uses the Java virtual machine, and has an extensive test suite (`http://packages.renjin.org/`).
- FastR (`https://github.com/allr/fastr`) by a team from Purdue. FastR is similar to Renjin, but it makes more ambitious optimisations and is somewhat less mature.
- Riposte (`https://github.com/jtalbot/riposte`) by Justin Talbot and

Zachary DeVito. Riposte is experimental and ambitious. For the parts of R it implements, it is extremely fast. Riposte is described in more detail in Riposte: A Trace-Driven Compiler and Parallel VM for Vector Code in R (`http://www.justintalbot.com/wp-content/uploads/2012/10/pact080talbot.pdf`).

These are roughly ordered from most practical to most ambitious. Another project, CXXR (`http://www.cs.kent.ac.uk/projects/cxxr/`) by Andrew Runnalls, does not provide any performance improvements. Instead, it aims to refactor R's internal C code in order to build a stronger foundation for future development, to keep behaviour identical to GNU-R, and to create better, more extensible documentation of its internals.

R is a huge language and it's not clear whether any of these approaches will ever become mainstream. It's a hard task to make an alternative implementation run all R code in the same way as GNU-R. Can you imagine having to reimplement every function in base R to be not only faster, but also to have exactly the same documented bugs? However, even if these implementations never make a dent in the use of GNU-R, they still provide benefits:

- Simpler implementations make it easy to validate new approaches before porting to GNU-R.
- Knowing which aspects of the language can be changed with minimal impact on existing code and maximal impact on performance can help to guide us to where we should direct our attention.
- Alternative implementations put pressure on the R-core to incorporate performance improvements.

One of the most important approaches that pqR, Renjin, FastR, and Riposte are exploring is the idea of deferred evaluation. As Justin Talbot, the author of Riposte, points out: "for long vectors, R's execution is completely memory bound. It spends almost all of its time reading and writing vector intermediates to memory". If we could eliminate these intermediate vectors, we could improve performance and reduce memory usage.

The following example shows a very simple example of how deferred evaluation can help. We have three vectors, `x`, `y`, `z`, each containing 1 million elements, and we want to find the sum of `x` + `y` where `z` is TRUE. (This represents a simplification of a pretty common sort of data analysis question.)

```
x <- runif(1e6)
y <- runif(1e6)
z <- sample(c(T, F), 1e6, rep = TRUE)

sum((x + y)[z])
```

In R, this creates two big temporary vectors: `x + y`, 1 million elements long, and `(x + y)[z]`, about 500,000 elements long. This means you need to have extra memory available for the intermediate calculation, and you have to shuttle the data back and forth between the CPU and memory. This slows computation down because the CPU can't work at maximum efficiency if it's always waiting for more data to come in.

However, if we rewrote the function using a loop in a language like C++, we only need one intermediate value: the sum of all the values we've seen:

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
double cond_sum_cpp(NumericVector x, NumericVector y,
                    LogicalVector z) {
  double sum = 0;
  int n = x.length();

  for(int i = 0; i < n; i++) {
    if (!z[i]) continue;
    sum += x[i] + y[i];
  }

  return sum;
}
```

On my computer, this approach is about eight times faster than the vectorised R equivalent, which is already pretty fast.

```
cond_sum_r <- function(x, y, z) {
  sum((x + y)[z])
}

microbenchmark(
  cond_sum_cpp(x, y, z),
  cond_sum_r(x, y, z),
  unit = "ms"
)
#> Unit: milliseconds
#>                    expr   min    lq median    uq   max neval
#>  cond_sum_cpp(x, y, z)  4.07  4.11    4.4  4.74  5.74   100
#>    cond_sum_r(x, y, z) 30.80 32.30   33.3 37.20 67.20   100
```

The goal of deferred evaluation is to perform this transformation automatically, so you can write concise R code and have it automatically translated into efficient machine code. Sophisticated translators can also figure out how to make the most of multiple cores. In the above example, if you have four cores, you could split `x`, `y`, and `z` into four pieces performing the conditional sum on each core, then adding together the four individual results. Deferred evaluation can also work with for loops, automatically discovering operations that can be vectorised.

This chapter has discussed some of the fundamental reasons that R is slow. The following chapters will give you the tools to do something about it when it impacts your code.