Do as little as possible

The easiest way to make a function faster is to let it do less work. One way to do that is use a function tailored to a more specific type of input or ouput, or a more specific problem. For example:

- `rowSums()`, `colSums()`, `rowMeans()`, and `colMeans()` are faster than equivalent invocations that use `apply()` because they are vectorised (the topic of the next section).
- `vapply()` is faster than `sapply()` because it pre-specifies the output type.
- If you want to see if a vector contains a single value, `any(x == 10)` is much faster than `10 %in% x`. This is because testing equality is simpler than testing inclusion in a set.

Having this knowledge at your fingertips requires knowing that alternative functions exist: you need to have a good vocabulary. Start with the basics, and expand your vocab by regularly reading R code. Good places to read code are the R-help mailing list and stackoverflow.

Some functions coerce their inputs into a specific type. If your input is not the right type, the function has to do extra work. Instead, look for a function that works with your data as it is, or consider changing the way you store your data. The most common example of this problem is using `apply()` on a data frame. `apply()` always turns its input into a matrix. Not only is this error prone (because a data frame is more general than a matrix), it is also slower.

Other functions will do less work if you give them more information about the problem. It's always worthwhile to carefully read the documentation and experiment with different arguments. Some examples that I've discovered in the past include:

- `read.csv()`: specify known column types with `colClasses`.
- `factor()`: specify known levels with `levels`.
- `cut()`: don't generate labels with `labels = FALSE` if you don't need them, or, even better, use `findInterval()` as mentioned in the "see also" section of the documentation.
- `unlist(x, use.names = FALSE)` is much faster than `unlist(x)`.
- `interaction()`: if you only need combinations that exist in the data, use `drop = TRUE`.

Sometimes you can make a function faster by avoiding method dispatch. As we saw in (Extreme dynamism), method dispatch in R can be costly. If you're calling a method in a tight loop, you can avoid some of the costs by doing the method lookup only once:

- For S3, you can do this by calling `generic.class()` instead of `generic()`.
- For S4, you can do this by using `findMethod()` to find the method, saving it to a variable, and then calling that function.

For example, calling `mean.default()` quite a bit faster than calling `mean()` for small vectors:

```
x <- runif(1e2)

microbenchmark(
  mean(x),
  mean.default(x)
)
#> Unit: microseconds
#>             expr  min   lq mean median   uq  max neval
#>          mean(x) 6.28 6.48 7.13   6.62 6.77 21.3   100
#>  mean.default(x) 2.22 2.40 2.94   2.50 2.58 23.1   100
```

This optimisation is a little risky. While `mean.default()` is almost twice as fast, it'll fail in surprising ways if `x` is not a numeric vector. You should only use it if you know for sure what `x` is.

Knowing that you're dealing with a specific type of input can be another way to write faster code. For example, `as.data.frame()` is quite slow because it coerces each element into a data frame and then `rbind()`s them together. If you have a named list with vectors of equal length, you can directly transform it into a data frame. In this case, if you're able to make strong assumptions about your input, you can write a method that's about 20x faster than the default.

```
quickdf <- function(l) {
  class(l) <- "data.frame"
  attr(l, "row.names") <- .set_row_names(length(l[[1]]))
  l
}

l <- lapply(1:26, function(i) runif(1e3))
names(l) <- letters

microbenchmark(
  quick_df      = quickdf(l),
  as.data.frame = as.data.frame(l)
)
#> Unit: microseconds
#>           expr     min    lq   mean  median      uq     max neval
#>       quick_df    21.1    23   27.7    26.4    30.5    90.2   100
#>  as.data.frame 2,080.0 2,130 2284.5 2,160.0 2,220.0 4,200.0   100
```

Again, note the trade-off. This method is fast because it's dangerous. If you give it bad inputs, you'll get a corrupt data frame:

```
quickdf(list(x = 1, y = 1:2))
#> Warning in format.data.frame(x, digits = digits, na.encode = FALSE):
#> corrupt data frame: columns will be truncated or padded with NAs
#>   x y
#> 1 1 1
```

To come up with this minimal method, I carefully read through and then rewrote the source code for `as.data.frame.list()` and `data.frame()`. I made many small changes, each time checking that I hadn't broken existing behaviour. After several hours work, I was able to isolate the minimal code shown above. This is a very useful technique. Most base R functions are written for flexibility and functionality, not performance. Thus, rewriting for your specific need can often yield substantial improvements. To do this, you'll need to read the source code. It can be complex and confusing, but don't give up!

The following example shows a progressive simplification of the `diff()` function if you only want computing differences between adjacent values. At each step, I replace one argument with a specific case, and then check to see that the function still works. The initial function is long and complicated, but by restricting the arguments I not only make it around twice as fast, I also make it easier to understand.

First, I take the code of `diff()` and reformat it to my style:

```
diff1 <- function (x, lag = 1L, differences = 1L) {
  ismat <- is.matrix(x)
  xlen <- if (ismat) dim(x)[1L] else length(x)
  if (length(lag) > 1L || length(differences) > 1L ||
      lag < 1L || differences < 1L)
    stop("'lag' and 'differences' must be integers >= 1")

  if (lag * differences >= xlen) {
    return(x[0L])
  }

  r <- unclass(x)
  i1 <- -seq_len(lag)
  if (ismat) {
    for (i in seq_len(differences)) {
      r <- r[i1, , drop = FALSE] -
        r[-nrow(r):-(nrow(r) - lag + 1L), , drop = FALSE]
    }
  } else {
    for (i in seq_len(differences)) {
      r <- r[i1] - r[-length(r):-(length(r) - lag + 1L)]
    }
  }
  class(r) <- oldClass(x)
  r
}
```

Next, I assume vector input. This allows me to remove the `is.matrix()` test and the method that uses matrix subsetting.

```
diff2 <- function (x, lag = 1L, differences = 1L) {
  xlen <- length(x)
  if (length(lag) > 1L || length(differences) > 1L ||
      lag < 1L || differences < 1L)
```

```
    stop("'lag' and 'differences' must be integers >= 1")

  if (lag * differences >= xlen) {
    return(x[0L])
  }

  i1 <- -seq_len(lag)
  for (i in seq_len(differences)) {
    x <- x[i1] - x[-length(x):-(length(x) - lag + 1L)]
  }
  x
}
diff2(cumsum(0:10))
#>  [1]  1  2  3  4  5  6  7  8  9 10
```

I now assume that `difference = 1L`. This simplifies input checking and eliminates the for loop:

```
diff3 <- function (x, lag = 1L) {
  xlen <- length(x)
  if (length(lag) > 1L || lag < 1L)
    stop("'lag' must be integer >= 1")

  if (lag >= xlen) {
    return(x[0L])
  }

  i1 <- -seq_len(lag)
  x[i1] - x[-length(x):-(length(x) - lag + 1L)]
}
diff3(cumsum(0:10))
#>  [1]  1  2  3  4  5  6  7  8  9 10
```

Finally I assume `lag = 1L`. This eliminates input checking and simplifies subsetting.

```
diff4 <- function (x) {
  xlen <- length(x)
  if (xlen <= 1) return(x[0L])

  x[-1] - x[-xlen]
}
diff4(cumsum(0:10))
#>  [1]  1  2  3  4  5  6  7  8  9 10
```

Now `diff4()` is both considerably simpler and considerably faster than `diff1()`:

```
x <- runif(100)
microbenchmark(
  diff1(x),
  diff2(x),
  diff3(x),
  diff4(x)
)
#> Unit: microseconds
#>      expr   min    lq  mean median    uq  max neval
#>  diff1(x) 13.40 13.90 15.37  14.10 14.60 33.8   100
#>  diff2(x) 10.80 11.30 12.49  11.60 11.80 33.6   100
#>  diff3(x)  8.84  9.26 10.02   9.51  9.77 19.6   100
#>  diff4(x)  6.11  6.47  6.92   6.67  6.99 22.4   100
```

You'll be able to make `diff()` even faster for this special case once you've readRcpp.

A final example of doing less work is to use simpler data structures. For example, when working with rows from a data frame, it's often faster to work with row indices than data frames. For instance, if you wanted to compute a bootstrap estimate of the correlation between two columns in a data frame, there are two basic approaches: you can either work with the whole data frame or with the individual vectors. The following example shows that working with vectors is about twice as fast.

```
sample_rows <- function(df, i) sample.int(nrow(df), i,
  replace = TRUE)

# Generate a new data frame containing randomly selected rows
boot_cor1 <- function(df, i) {
  sub <- df[sample_rows(df, i), , drop = FALSE]
  cor(sub$x, sub$y)
}

# Generate new vectors from random rows
boot_cor2 <- function(df, i ) {
  idx <- sample_rows(df, i)
  cor(df$x[idx], df$y[idx])
}

df <- data.frame(x = runif(100), y = runif(100))
microbenchmark(
  boot_cor1(df, 10),
  boot_cor2(df, 10)
)
#> Unit: microseconds
#>               expr min  lq mean median  uq max neval
#>  boot_cor1(df, 10) 179 185  200    194 199 768   100
#>  boot_cor2(df, 10) 113 116  122    119 126 192   100
```