Vectorise

If you've used R for any length of time, you've probably heard the admonishment to "vectorise your code". But what does that actually mean? Vectorising your code is not just about avoiding for loops, although that's often a step. Vectorising is about taking a "whole object" approach to a problem, thinking about vectors, not scalars. There are two key attributes of a vectorised function:

- It makes many problems simpler. Instead of having to think about the components of a vector, you only think about entire vectors.
- The loops in a vectorised function are written in C instead of R. Loops in C are much faster because they have much less overhead.

Functionals stressed the importance of vectorised code as a higher level abstraction. Vectorisation is also important for writing fast R code. This doesn't mean simply using `apply()` or `lapply()`, or even `Vectorise()`. Those functions improve the interface of a function, but don't fundamentally change performance. Using vectorisation for performance means finding the existing R function that is implemented in C and most closely applies to your problem.

Vectorised functions that apply to many common performance bottlenecks include:

- `rowSums()`, `colSums()`, `rowMeans()`, and `colMeans()`. These vectorised matrix functions will always be faster than using `apply()`. You can sometimes use these functions to build other vectorised functions.

  ```
  rowAny <- function(x) rowSums(x) > 0
  rowAll <- function(x) rowSums(x) == ncol(x)
  ```

- Vectorised subsetting can lead to big improvements in speed. Remember the techniques behind lookup tables (lookup tables) and matching and merging by hand (matching and merging by hand). Also remember that you can use subsetting assignment to replace multiple values in a single step. If `x` is a vector, matrix or data frame then `x[is.na(x)] <- 0` will replace all missing values with 0.
- If you're extracting or replacing values in scattered locations in a matrix or data frame, subset with an integer matrix. See matrix subsetting for more details.
- If you're converting continuous values to categorical make sure you know how to use `cut()` and `findInterval()`.
- Be aware of vectorised functions like `cumsum()` and `diff()`.

Matrix algebra is a general example of vectorisation. There loops are executed by highly tuned external libraries like BLAS. If you can figure out a way to use matrix algebra to solve your problem, you'll often get a very fast solution. The ability to solve problems with matrix algebra is a product of experience. While this skill is something you'll develop over time, a good place to start is to ask people with experience in your domain.

The downside of vectorisation is that it makes it harder to predict how operations will scale. The following example measures how long it takes to use character subsetting to lookup 1, 10, and 100 elements from a list. You might expect that looking up 10 elements would take 10x as long as looking up 1, and that looking up 100 elements would take 10x longer again. In fact, the following example shows that it only takes about 9 times longer to look up 100 elements than it does to look up 1.

```
lookup <- setNames(as.list(sample(100, 26)), letters)

x1 <- "j"
x10 <- sample(letters, 10)
x100 <- sample(letters, 100, replace = TRUE)

microbenchmark(
  lookup[x1],
  lookup[x10],
  lookup[x100]
)
#> Unit: nanoseconds
#>          expr    min     lq  mean median     uq    max neval
#>    lookup[x1]    960  1,100  1220  1,210  1,300  2,750   100
#>   lookup[x10]  3,040  3,160  3512  3,250  3,340 27,400   100
#>  lookup[x100] 10,600 10,900 11373 11,100 11,500 26,300   100
```

Vectorisation won't solve every problem, and rather than torturing an existing algorithm into one that uses a vectorised approach, you're often better off writing your own vectorised function in C++. You'll learn how to do so in Rcpp.