Solutions: Practising R Programming

YSC2210 - DAVis with R

Michael T. Gastner

1 Days of the week in the Shire calendar

(a) (I)

```r
library(dplyr)
shire_day_from_gregorian_a <- function(g) {
  if_else(
    g == "Monday",
    "Sterday",
    if_else(
      g == "Tuesday",
      "Sunday",
      if_else(
        g == "Wednesday",
        "Monday",
        if_else(
          g == "Thursday",
          "Trewsday",
          if_else(
            g == "Friday",
            "Hevensday",
            if_else(
              g == "Saturday",
              "Mersday",
              if_else(
                g == "Sunday",
                "Highday",
                NA_character_
              )
            )
          )
        )
      )
    )
  )
}
```

(II)

```r
shire_day_from_gregorian_b <- function(g) {
  case_when(
    g == "Monday" ~ "Sterday",
    g == "Tuesday" ~ "Sunday",
    g == "Wednesday" ~ "Monday",
    g == "Thursday" ~ "Trewsday",
```

```r
    g == "Friday" ~ "Hevensday",
    g == "Saturday" ~ "Mersday",
    g == "Sunday" ~ "Highday",
    TRUE ~ NA_character_
  )
}
```

(III)
```r
shire_day_from_gregorian_c <- function(g) {
  recode(
    g,
    Monday = "Sterday",
    Tuesday = "Sunday",
    Wednesday = "Monday",
    Thursday = "Trewsday",
    Friday = "Hevensday",
    Saturday = "Mersday",
    Sunday = "Highday",
    .default = NA_character_
  )
}
```

The implementation with `recode()` has the smallest amount of repeating code, but `?recode` reveals that the package managers of **dplyr** consider the function's life cycle as

> 'questioning because the arguments are in the wrong order [compared to other **dplyr** functions]... We don't yet know how to fix this problem, but it's likely to involve creating a new function then retiring or deprecating `recode()`.'

Therefore, I find the implementation with `case_when()` the best compromise at present. However, I accept that opinions about the best choice are likely to differ.

(b) The week day names in `bsts::weekday.names` start with Sunday; thus, the vector `shire_days` starts with `"Highday"`.

```r
library(bsts)
shire_days <- c(
  "Highday",
  "Sterday",
  "Sunday",
  "Monday",
  "Trewsday",
  "Hevensday",
  "Mersday"
)
```

The next code chunk returns `TRUE` if and only if all functions implemented in (a) return the correct Shire days.

```r
all(shire_days == shire_day_from_gregorian_a(weekday.names)) &
  all(shire_days == shire_day_from_gregorian_b(weekday.names)) &
  all(shire_days == shire_day_from_gregorian_c(weekday.names))
```

```
## [1] TRUE
```

2 Measuring run-times with the microbenchmark package

(a) Installing the **microbenchmark** package is straightforward. It is possible to automate the installation in an R script (see https://stackoverflow.com/questions/4090169/elegant-way-to-check-for-missing-packages-and-install-them/19873732). However, in this course, we assume that the user has already installed the packages. Still, we should not assume that the user has already loaded them in this R session; thus, we should include `library()` as part of the R script or R Markdown file, ideally at the top of the file.

```r
# Ideally, calls to library() would be at the top of the file. However,
# for the sake of clarity, I put it here.
library(microbenchmark)
abs_with_if_else <- function(x) {
  dplyr::if_else(x >= 0, x, -x)
}
abs_with_subsetting <- function(x) {
  neg <- (x < 0)
  x[neg] <- -x[neg]
  x
}
abs_with_data_type_conversion <- function(x) {
  ((x > 0) - (x < 0)) * x
}
abs_with_for_loop <- function(x) {
  for (i in seq_along(x)) {
    if (x[i] < 0) {
      x[i] <- -x[i]
    }
  }
  x
}
```

(b) We generate a moderately long vector `s` that keeps `abs()`, `abs_with_if_else()`, ... busy for a while.

```r
s <- rnorm(1e6)
microbenchmark(abs(s),
               abs_with_if_else(s),
               abs_with_subsetting(s),
               abs_with_data_type_conversion(s),
               abs_with_for_loop(s))
```

```
## Unit: microseconds
##                              expr       min        lq      mean    median
##                            abs(s)   701.896  1131.890  3725.559  1738.038
##               abs_with_if_else(s) 47700.035 58456.478 69373.498 64349.361
##            abs_with_subsetting(s) 14831.763 18587.574 21824.554 20937.432
##  abs_with_data_type_conversion(s)  7472.203  9735.736 15521.270 12027.646
##              abs_with_for_loop(s) 50285.978 52993.947 55597.340 54291.177
##         uq       max neval   cld
##   3949.577  65213.19   100 a
##  71276.757 131403.55   100     e
##  23619.179  37036.22   100   c
##  16822.391  74466.55   100  b
##  56105.865  80603.12   100    d
```

(c) The output of `microbenchmark()` shows summary statistics from 100 runs. The columns show the minimum (`min`), lower quartile (`lq`), mean, median, upper quartile (`uq`) and maximum (`max`). The unit (here microseconds) is stated on the first line. The primary piece of information is the median. The lower and upper quartile give us some indication of the variability.

The preinstalled function `abs()` is fastest. The functions `abs_with_data_type_conversion()` and `abs_with_subsetting()` are much slower than `abs()`, but still faster than `abs_with_for-loops()` and `abs_with_if_else()`.

Judging from this output, it is not worth writing our own alternative to `abs()`. In general, the mathematical functions in R's base installation are so fast that we do not need to spend our own time re-inventing the wheel.

When we have to write our own function, data type conversion or subsetting are the preferred strategies. It is best to avoid `for`-loops because they are slow and cumbersome to write. Although `if_else()` can be as slow as a `for`-loop, the syntax is straightforward, so we should keep `if_else()` in our toolbox.

3 Comparing two functions for calulating Pythagorean sums

(a) The line

```
ratio[is.nan(ratio)] <- 1
```

ensures that, even for $a = b = 0$ and $a = b =$ `Inf`, the returned value is correct (zero and `Inf`, respectively). Otherwise, `ratio` would have been `NaN` because of the division `q / p` one line earlier.

(b)

```
library(microbenchmark)
pythag_1 <- function(a, b) {
  sqrt(a^2 + b^2)
}
pythag_2 <- function(a, b) {
  absa <- abs(a)
  absb <- abs(b)
  p <- pmax(absa, absb)
  q <- pmin(absa, absb)
  ratio <- q / p
  ratio[is.nan(ratio)] <- 1
  p * sqrt(1.0 + ratio^2)
}

# Long random vectors that keep `pythag_1()` and `pythag_2()` busy for a
# while
x <- sample(-10000:10000, 1e6, replace = TRUE)
y <- sample(-10000:10000, 1e6, replace = TRUE)
microbenchmark(pythag_1(x, y), pythag_2(x, y))
```

```
## Unit: milliseconds
##            expr       min        lq     mean   median       uq       max neval
##  pythag_1(x, y)  5.586894  7.776487 13.32199 11.08600 13.70036  64.27665   100
##  pythag_2(x, y) 26.233682 33.838599 42.00928 36.67267 40.78477 106.56651   100
##  cld
##   a
##    b
```

If speed were our only concern, then `pythag_1()` would be the winner. Its median calculation time is approximately one third of the time needed by `pythag_2()`.

(c) Speed is only one of the concerns when writing computer code. Numerical robustness is another serious concern. In terms of numerical robustness, `pythag_1()` has serious shortcomings. It overflows and underflows for arguments that can still be handled adequately by `pythag_2()`. Overflow for `pythag_1()` begins with input numbers of the order of 10^{154}, whereas `pythag_2()` only starts overflowing at around 10^{308}.

```
pythag_1(c(3e153, 3e154), c(4e153, 4e154))
```

```
## [1] 5e+153    Inf
```

```
pythag_2(c(3e154, 3e307, 3e308), c(4e154, 4e307, 4e308))
```

```
## [1] 5e+154 5e+307    Inf
```

Underflow for `pythag_1()` starts with input of the order of 10^{-159}. For numbers below 10^{-162}, `pythag_1()` cannot distinugish the Pythagorean sum from zero.

```
pythag_1(c(3e-159, 3e-160, 3e-162, 3e-163),
        c(4e-159, 4e-160, 4e-162, 4e-163))
```

```
## [1] 5.000000e-159 4.999972e-160 4.970240e-162  0.000000e+00
```

By contrast, `pythag_2()` only underflows for input that is smaller than 10^{-317}.

```
pythag_2(c(3e-317, 3e-318, 3e-324, 3e-325),
        c(4e-317, 4e-318, 3e-324, 3e-325))
```

```
## [1] 5.000000e-317 4.999999e-318 4.940656e-324  0.000000e+00
```

(d) The function `pythag_1()` is a straightforward translation of the formula for the Pythagorean sum into a properly vectorised expression. Whoever wrote `pythag_2()` should do a better job at commenting the code![1] Upon closer inspection, however, `pythag_2()` has a clear numerical advantage. It computes an auxiliary variable `ratio` that is always ≤ 1 and can, consequently, never overflow. It does not really matter if `ratio` underflows. In that case, `1.0` is much larger than `ratio^2` in the last line of the function body; thus, a long sequence of the returned value's leading figures is still correct.

By contrast, calculating the squares of `a` and `b` directly as in `pythag_1()` can cause overflow and underflow in intermediate results, from which `pythag_1()` cannot recover.

In summary, the code of `pythag_1()` is shorter and its execution faster. However, in my opinion the improved numerical stability of `pythag_2()` makes it worth paying the price of waiting a little bit longer for results.

4 Floating-point accuracy

(a) Floating-point numbers are only represented with finite precision on a computer, following the IEEE 754 standard. The internal representation of these numbers is in binary format. Numbers that have an exact decimal representation (e.g. 0.2) cannot be represented accurately as a binary number with finite precision. Consequently, there are round-off errors, which lead to `0.1 + 0.2` and `0.3` to differ in the least significant bits.

```
0.1 + 0.2 - 0.3
```

```
## [1] 5.551115e-17
```

(b) Instead of `==`, we should use `all.equal()`, which is designed to test whether two objects are equal or almost equal.

```
all.equal(0.1 + 0.2, 0.3)
```

```
## [1] TRUE
```

[1] Oh, now I remember . . . it was me. So, I retract this complaint.

There are special packages (e.g. **gmp**) to widen the limits of the IEEE 754 standard, but they slow down run-times tremendously. For most data analysis problems, it is sensible to accept the limits of the standard 64-bit double-precision floating-point format and avoid comparisons with `==`.