Performance, Profiling, and Debugging in R

Kylie A. Bemis

Northeastern University
Khoury College of Computer Sciences

Northeastern University

Goals for this session

- Performance and profiling
- Debugging R code

PERFORMANCE AND PROFILING

Performance in R

- R is sometimes considered "slow"
- Most code can be written to be faster
- Benchmark to compare implementations

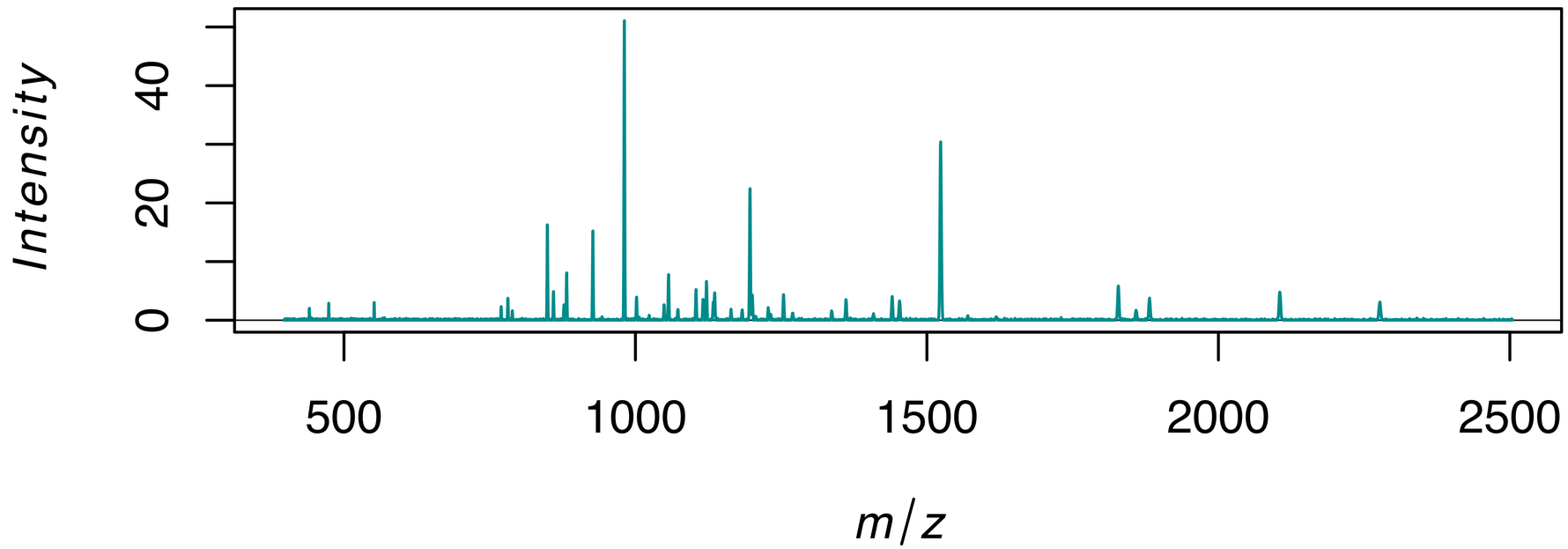
Avoid premature optimization

- First, write "correct" code
- Many times it will be "fast enough"
- Beware naive or early optimization
- Always benchmark to confirm

Case study: finding local maxima

```
library(MSExample)
set.seed(2020)
s <- simSpectra(n=200, by=50, units="ppm")

plot(s[[1]])
```



Version 1

```
locmax1 <- function(x, halfWindow = 2) {
      begin <- halfWindow + 1
      end <- length(x) - halfWindow
      out <- integer()
      if ( begin < 1L || end > length(x) )
        return(out)
      for ( i in begin:end ) {
        j1 <- i - halfWindow
        j2 <- i + halfWindow
        is_max <- TRUE
        for ( j in j1:j2 ) {
          if ( x[j] > x[i] )
            is_max <- FALSE
        }
        if ( is_max )
          out <- c(out, i)
      }
      out
}
```

Version 2

```
locmax2 <- function(x, halfWindow = 2) {
      begin <- halfWindow + 1
      end <- length(x) - halfWindow
      if ( begin < 1L || end > length(x) )
        return(integer())
      out <- logical(length(x)) # initialize output
      for ( i in begin:end ) {
        j1 <- i - halfWindow
        j2 <- i + halfWindow
        out[i] <- TRUE
        for ( j in j1:j2 ) {
          if ( x[j] > x[i] ) {
            out[i] <- FALSE    # remove c(x, y) call
            break
          }
        }
      }
      which(out)
}
```

Version 3

```
locmax3 <- function(x, halfWindow = 2) {
      begin <- halfWindow + 1
      end <- length(x) - halfWindow
      if ( begin < 1L || end > length(x) )
        return(integer())
      out <- vapply(begin:end, function(i) { # remove for loop
        j1 <- i - halfWindow
        j2 <- i + halfWindow
        if ( any(x[j1:j2] > x[i]) ) {
          FALSE
        } else {
          TRUE
        }
      }, logical(1))
      out <- c(rep(FALSE, halfWindow),
               out, rep(FALSE, halfWindow))
      which(out)
}
```

Benchmarking

```
microbenchmark(locmax1 = locmax1(intensity(s[[1]])),
               locmax2 = locmax2(intensity(s[[1]])),
               locmax3 = locmax3(intensity(s[[1]])))
```

```
## Unit: milliseconds
##     expr       min        lq      mean    median        uq      max
##  locmax1  8.798584  9.277475 10.947572  9.723257 10.857094 24.27297
##  locmax2  3.330522  3.440301  3.896457  3.572318  3.793416  7.73430
##  locmax3 13.420170 14.140362 16.323038 15.086706 17.035599 39.28341
```

Benchmarking

```
microbenchmark(locmax1 = locmax1(intensity(s[[1]])),
               locmax2 = locmax2(intensity(s[[1]])),
               locmax3 = locmax3(intensity(s[[1]])))
```

```
## Unit: milliseconds
##     expr       min        lq      mean    median        uq      max
##  locmax1  8.798584  9.277475 10.947572  9.723257 10.857094 24.27297
##  locmax2  3.330522  3.440301  3.896457  3.572318  3.793416  7.73430
##  locmax3 13.420170 14.140362 16.323038 15.086706 17.035599 39.28341
```

Tips for performant R code

- Use functions that call C/C++ code
- Use vectorized functions rather than loops
- Avoid creating copies/duplicating objects

DEBUGGING R CODE

Debugging tools in R

- `traceback()`
  - Prints the stack at the time the error occurred
- `browser()`
  - Creates a pseudo-breakpoint that enters debugger
- `options(error=recover)`
  - Allows entering debugger after any error
- `debug() and undebug()`
  - Enter debugger on a specific function call

Defensive programming

- "Fail fast" and "fail early"
  - Catch potential errors due to bad input ASAP
- Short, simple functions that do one thing
- Avoid non-standard evaluation
  - Useful in the tidyverse, but challenging to debug
- Avoid overly-flexible functions
  - Handling too many cases makes debugging difficult

BREAK