Pre-workshop:

- Install R (or use it on the computers here):
  - http://cran.r-project.org/
- Install an R environment, such as R Studio:
  - http://rstudio.org/
- Download the slides and the .R script:
  - http://bit.ly/yRjShO
- **Review the exercise solutions from Part I**

Programming in R

PART II

Ben Haller

BGSA Stats Workshop 13 February 2012

Outline

- Flow control
- Subsetting and sorting
- Writing functions

- Coding for speed
- Debugging in R
- Caveats of R

Format

- Input to R:

```
> say.hello <- function(target)
+ {
+     cat("Hello, ", target,
+         "!", sep="")
+ }
> say.hello("world")
```

- Output from R:

```
Hello, world!
```

- R stuff you don't need to type:

```
> say.goodbye()
```

Help

- Don't forget you can get help using R's built-in help function:

```
> ?cat
> ?"["
```

help:
`?cat`

- Raise your hand at any time to get help; don't be shy!
- See last week's slides for recommended books & websites on R programming

Coding for Speed

Thanks to Phil Spector for examples used in this section
http://www.stat.berkeley.edu/~nolan/stat133/Fall05/lectures/profilingEx.html

Coding for Speed

- R is often sufficiently fast...
- ...but R is actually a very, very slow language
- At some point, you will want to speed up or "optimize" your code
- But don't optimize *all* your code:
  - "Premature optimization is the root of all evil" (Donald Knuth)
  - 90/10 rule: 90% of time is in 10% of the code

?system.time

- The simplest way to measure performance:

```
> system.time(<expression>)
```

- Example:

```
> busywork1 <- function(n)
+ {
+     total <- 0
+     for (i in 1:n)
+         total <- total + rnorm(1)
+     total
+ }
> system.time(busywork1(1000000))
```

```
   user  system elapsed
  4.857   0.656   5.464
```

1

- Why did this take almost 5 seconds?

- Hint: addition is fast, and generating random numbers is (fairly) fast:

```
> busywork2 <- function(n)
+ {
+     sum(rnorm(n))
+ }
> system.time(busywork2(1000000))
```

```
   user  system elapsed
  0.113   0.000   0.113
```

2

- So now we get 0.11 seconds, from 4.9 seconds, doing the same work. Why?
- Optimization technique #1: **vectorization**

Coding for Speed: Vectorization

- Vectorization is changing operations on individual values (usually in a loop) with operations on whole vectors
- This can leverage **vector processors**: special computational hardware units
- In R, it also avoids language-related overhead because R is an **interpreted language**

Coding for Speed: Vectorization

- In R, whenever you think about writing a loop, consider vectorization:
  - by using R functions that are vectorized, such as `rnorm` and `sum` (and most others)
  - by using `ifelse` to vectorize loops containing `if` / `else` constructs
- Note that `apply`, `sapply`, ... are not vectorized internally (but are good for other reasons)

Coding for Speed: Vectorization

- So compare the non-vectorized version:

```
> busywork1 <- function(n)
+ {
+     total <- 0
+     for (i in 1:n)
+         total <- total + rnorm(1)
+     total
+ }
```

to the vectorized version:

```
> busywork2 <- function(n)
+ {
+     sum(rnorm(n))
+ }
```

- Now consider this function:

```
> xx <- matrix(rnorm(200000), 10000, 20)
> xx[xx > 2] <- NA
> data <- as.data.frame(xx)
> omitNA_aggregation <- function(x)
+ {
+     res = NULL
+     for (i in 1:nrow(x))
+         if (!any(is.na(x[i,])))
+             res <- rbind(res, x[i,])
+     invisible(res)
+ }
> system.time(omitNA_aggregation(data))
```

```
   user  system elapsed
 69.779   2.653  71.769
```

3

- So `omitNA_aggregation()` took 70 seconds to remove rows containing `NA`. Why?

?Rprof

- A second way to measure performance:

```
> Rprof(filename, ...)
```

- Example:

```
> Rprof(tmp <- tempfile())
> omitNA_aggregation(data)
> Rprof()
> summaryRprof(tmp)
```

4

```
$by.self
...
$by.total
                total.time total.pct self.time self.pct
"omitNA_agg."        20.50    100.00      0.04     0.20
"rbind"              18.52     90.34     10.34    50.44
"unclass"             3.94     19.22      3.94    19.22
...
```

- Note that `rbind` is taking 90% of the time!

Coding for Speed: Thinking

- Optimization technique #2: **thinking**
- Which code is slow is often a surprise
  - This is why profiling is so useful
  - This is also why optimization should wait until the bottlenecks are clear
- So let's think: why would using `rbind` to build a new dataframe be so slow?
- What does R do when you execute a statement like this:

```
res <- rbind(res, x[i,])
```

?apply

- It turns out that growing an object is extremely inefficient; so let's avoid it:

```
> omitNA_apply <- function(x)
+ {
+     drop <- apply(is.na(x), MAR=1, FUN=any)
+     x[!drop, ]
+ }
> system.time(omitNA_apply(data))
```

```
   user  system elapsed
  0.111   0.002   0.112
```

5

- From 70 seconds to 0.11 seconds, not bad
- Why does `apply` do so well here?

- The key is that `apply` is smart about the way it builds the result object:

```
> apply
```

```
...
ans <- vector("list", d2)
...
for (i in 1L:d2) {
    tmp <- FUN(newX[, i], ...)
    if (!is.null(tmp))
        ans[[i]] <- tmp
}
```

- It allocates a result vector, `ans`, ahead of time, and then replaces items: much faster!

- A new problem:
- For each `x[i]` in `x`:
  - find distances `abs(x[i]-x[j])`
  - assemble a new vector of nearest neighbor distances for every value `x[i]`
- This is hard to optimize; for `x` of length `N`, finding the nearest neighbor for each `x[i]` involves `N-1` comparisons, so finding them all requires `N(N-1)` comparisons
- This is called an $O(N^2)$ problem: the execution time scales with N^2

Coding for Speed: Google

- Optimization technique #3: **Google**.

- A package called `class` has a function called `knn` that solves our problem, and is very fast
- There is so much help online for R! Use it!

Coding for Speed: External Calls

`?.C`

- But suppose that a **Google** search came up empty, and **thinking** didn't help, and **vectorization** wasn't applicable... then what?

- You can call out from R to code written in C, Fortran, or other languages!

Coding for Speed: External Calls

`?.C`

- Optimization technique #4: **external calls**
  - good introductions can be found via Google
  - the `inline` package looks really cool for this
- Pros:
  - these languages are compiled
  - this means they are much, much faster
- Cons:
  - you have to learn C or Fortran!
  - the call out from R is a bit tricky to do right

Coding for Speed: Multiprocessing

- Optimization technique #5: **parallelizing**
  - most computers have multiple processing cores, but normally, R uses only one
  - McGill also has computing clusters with many cores that can run R in parallel
  - parallelizing lets you utilize all available cores/ machines simultaneously!
- Many stats tasks parallelize very well:
  - cross-validation
  - bootstrapping

Coding for Speed: Multiprocessing

- There are several parallel computing packages:
  - `multicore` (simple, my favorite)
  - `snow` (more complex, more powerful)
  - `snowfall` (a higher-level wrapper around snow)
  - `foreach` (parallel language extensions)
- Some others can use parallelism internally:
  - `plyr` (a very powerful data-processing package)
- More information:
  - http://cran.r-project.org/web/views/HighPerformanceComputing.html

Coding for Speed: A Review

- Various optimization techniques:
  - vectorize loops
  - think about the problem
  - look for packages or help on Google
  - use external calls to C or Fortran
  - leverage parallel processing
- Only optimize when necessary, because:
  - it takes time and effort
  - it often introduces bugs
  - it often makes code harder to understand

Exercise 1

- Here's a slow function:

```
> row_means <- function(x)
+ {
+     means <- numeric(0)
+     for (i in 1:NROW(x))
+     {
+         total <- 0
+         for (j in 1:NCOL(x))
+             total <- total + x[i, j]
+         means <- c(means, total / NCOL(x))
+     }
+     invisible(means)
+ }
```

- Optimize it, and use `system.time` (or `Rprof`) to quantify your speedup; then compare your work with your neighbor

Debugging in R

Thanks to Mark Bravington and Roger Peng for examples used in this section
http://cran.r-project.org/doc/Rnews/Rnews_2003-3.pdf#page=29
http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf

Debugging in R

- Bugs are inevitable, so learn to find them fast!
- There are lots of simple debugging methods:
  - Execute your code one step at a time
  - Check intermediate results with `str`
  - Insert `print` or `cat` statements to show values
  - Stare at your code, wail, gnash your teeth
  - Ask for help from someone else
- We will focus on more advanced techniques

?debug

Debugging in R: debug

- Bugs can manifest as incorrect results:

```
> f <- function(i)  # factorials!
+ {
+     if (i < 0) return(NA)
+
+     # For i > 2 we will loop upward
+     # from 2 to (i-1), multiplying
+     for (j in 2:(i-1)) {
+         i <- i * j
+     }
+
+     i
+ }
> sapply(0:7, FUN=f)
```

```
[1]    0    0    4    6   24  120  720 5040
```

7

- Let's use the `debug` function to investigate:

```
> debug(f)
> f(2)
```

```
debugging in: f(2)
debug: {
    if (i < 0)
        return(NA)
    for (j in 2:(i - 1)) {
        i <- i * j
    }
    i
}
```

```
Browse[2]>
```

8

- We are now running a debugging session!
- Enter commands at the `Browse[2]>` prompt

- What do we expect to see?

```
debugging in: f(2)
debug: {
    if (i < 0)
        return(NA)
    for (j in 2:(i - 1)) {
        i <- i * j
    }
    i
}
```

- We expect that `i == 2`, since we called `f(2)`
- We expect that the loop will be skipped, since `2:(i – 1)` should be `NULL`
- We thus expect `2` to be returned
- Now we can test these expectations!

- Some commands:
  - Enter `Q` to quit the session
  - Enter `n` to advance to the next step in the code
  - Most other commands get interpreted by R
- For example, we can print variables:

```
Browse[2]> i
```

```
[1] 2
```

- We can also do anything else we could normally do at an R prompt:
  - call functions
  - evaluate expressions

- OK, so let's continue:

```
Browse[2]> n
```

```
debug: if (i < 0) return(NA)
```

(About to execute the `if` statement)

```
Browse[2]> n
```

```
debug: NULL
```

(Stepping over the `if` statement yields `NULL` because the `if` was not executed; `else NULL` is always implied if no `else` clause is given)

- Now we get to the heart of the function:

```
Browse[2]> n
```

```
debug: for (j in 2:(i - 1)) {
    i <- i * j
}
```

```
Browse[2]> n
```

```
debug: j
```

(Meaning the `for` loop is about to set `j`)

```
Browse[2]> n
```

```
debug: i <- i * j
```

(About to execute the body of the `for` loop)

- Hmm, so we're executing the `for` loop:

```
debug: i <- i * j
```

- That's counter to our expectations
- Let's gather just a little more information:

```
Browse[2]> j
```

```
[1] 2
```

(`j == 2` for this iteration of the `for` loop)

- So one of our expectations – our hypotheses – turns out to be false. Now what?

- Now it's time to look again at our code and think:

```
> f <- function(i)  # factorials!
+ {
+     if (i < 0) return(NA)
+
+     # For i > 2 we will loop upward
+     # from 2 to (i-1), multiplying
+     for (j in 2:(i-1)) {
+         i <- i * j
+     }
+
+     i
+ }
```

- Our sequence `2:(i-1)` is not working as intended! `2:1` gives `2 1`, not `NULL`!

- Let's confirm the problem:

```
Browse[2]> 2:3
```

```
[1] 2 3
```

```
Browse[2]> 2:2
```

```
[1] 2
```

```
Browse[2]> 2:1
```

```
[1] 2 1
```

- This is a very common issue in R!
- A sequence like `x:y` is usually intended to count either upward or downward
- R will run the sequence in either direction!

?seq

8

- To avoid this problem, use `seq` instead of using the `:` operator:

```
Browse[2]> seq(from=2, to=3, by=1)
```

```
[1] 2 3
```

```
Browse[2]> seq(from=2, to=2, by=1)
```

```
[1] 2
```

```
Browse[2]> seq(from=2, to=1, by=1)
```

```
Error in seq.default(from = 2, to = 1, by = 1):
  wrong sign in 'by' argument
```

- The best way to debug is to use good coding practices to avoid writing bugs!

- Now that we know the problem, we can end our debugging session:

```
Browse[2]> Q
```

- Debugging is still enabled for `f` (the `debug` flag is persistent), so turn it off:

```
> undebug(f)
```

- Alternatively, we could have used `debugonce` to flag `f` for debugging only one time:

```
> debugonce(f)
```

?browser

Debugging in R: `browser`

- Suppose we wanted to debug the *end* of our function, for a call to f(35):

```
> f <- function(i)  # factorials!
+ {
+     if (i < 0) return(NA)
+
+     # For i > 2 we will loop upward
+     # from 2 to (i-1), multiplying
+     for (j in 2:(i-1)) {
+         i <- i * j
+     }
+
+     i
+ }
```

We want to debug HERE!

- Getting there with `debug` is tedious!

- Insert a call to `browser`:

```
> f <- function(i)  # factorials!
+ {
+   if (i < 0) return(NA)
+
+   # For i > 2 we will loop upward
+   # from 2 to (i-1), multiplying
+   for (j in 2:(i-1)) {
+       i <- i * j
+   }
+   browser()
+   i
+ }
> f(35)
```

```
Called from: f(35)
```

```
Browse[1]> i
```

```
[1] 1.033315e+40
```

```
Browse[1]> Q
```

Debugging in R: Warnings

- Bugs can also manifest as warnings:

```
> x <- log(-1)
```

```
Warning message:
In log(-1) : NaNs produced
```

```
> x
```

```
[1] NaN
```

10

- A warning indicates a non-fatal condition that may or may not be an error (but it probably is)
- Use `options(warn=2)` to convert warnings into errors, to make them easier to debug

Debugging in R: Errors

- Finally, bugs can manifest as errors:

```
> f(x)
```

```
Error in if (i < 0) return(NA) : missing value
where TRUE/FALSE needed
```

11

- Errors indicate fatal conditions that cannot be recovered from gracefully; execution stops
- Note that ignoring the earlier warning, and passing `x` (which is `NaN`) to our factorial function `f`, has now led to an error

?traceback

Debugging in R: `traceback`

- What to do when you hit an error? If you don't know which part of your code caused it, step one is to use `traceback`:

```
> traceback()
```

```
1: f(x)
```

12

- In this case, the error occurred in `f`
- More: we now know that it didn't happen in something `f` called; it happened in `f` itself, which makes it easy to find and fix

- Let's look at a more complex situation:

```
> f <- function(x)
+ {
+     h(x) - g(x-3)
+ }
> g <- function(x)
+ {
+     r <- sqrt(x) * h(x)
+     sum(1:r)
+ }
> h <- function(x)
+ {
+     r <- sqrt(x)
+     sum(1:r)
+ }
> f(1)
```

```
Error in 1:r : NA/NaN argument
In addition: Warning messages:
1: In sqrt(x) : NaNs produced
2: In sqrt(x) : NaNs produced
```

- If you write code like this, you deserve your fate!
  - Give variables explanatory names!
  - Comment your code!
  - Test for error conditions using `stopifnot`!
- That said...

```
> traceback()
```

```
4: 1:r
3: h(x)
2: g(x - 3)
1: f(1)
```

- The `traceback` shows us the error is in `h`, so we can focus; more on this later...

?debug

Exercise 2

- Here's a function with three bugs in it:

```
> myPlot <- function(fuzz, hiod, eh)
+ {
+     cx <- sapply(0:fuzz, FUN=function(x)
                { c(cos(x), sin(x)) })
+     pts <- matrix(data=cm, ncol=2, byrow=TRUE)
+     plot(x=pts[,1], y=pts[2], pch=19, cex=1.5)
+     points(x=c(hiod, -hoid), y=c(eh, eh),
+         pch=21, bg="yellow", cex=10, lwd=3)
+     segments(x0=-0.5, y0=-0.5, x1=0.5,
+         y1=-0.3, lwd=10)
+ }
> myPlot(43, 0.3, 0.4)
```

- Find and fix the bugs; talk to your neighbor or raise your hand if you get stuck!

?recover

Debugging in R: `recover`

- Let's return to the last example:

```
> f <- function(x)
+ {
+     h(x) - g(x-3)
+ }
> g <- function(x)
+ {
+     r <- sqrt(x) * h(x)
+     sum(1:r)
+ }
> h <- function(x)
+ {
+     r <- sqrt(x)
+     sum(1:r)
+ }
```

Debugging in R: `recover`

- We got an error in a nested function call:

```
> f(1)
```

13

```
Error in 1:r : NA/NaN argument
In addition: Warning messages:
1: In sqrt(x) : NaNs produced
2: In sqrt(x) : NaNs produced
```

```
> traceback()
```

14

```
4: 1:r
3: h(x)
2: g(x - 3)
1: f(1)
```

- We could use `debug` on `h`... but what if the root of the problem is elsewhere?

Debugging in R: `recover`

- To solve this, let's try a new method:

```
> options(error=recover)
> f(1)
```

```
Error in 1:r : NA/NaN argument
In addition: Warning messages:
1: In sqrt(x) : NaNs produced
2: In sqrt(x) : NaNs produced

Enter a frame number, or 0 to exit
1: f(1)
2: g(x - 3)
3: h(x)
```

- R caught the error and we've entered debugging using a new function, `recover`

- Let's look in frame 3 first, since that's where the error occurred:

```
Selection: 3
```

```
Called from: g(x - 3)
```

```
Browse[1]>
```

- Now we are in the "browser" interface, just like when we used `debug`
- Recall function `h`:

```
> h <- function(x)
+ {
+     r <- sqrt(x)
+     sum(1:r)
+ }
```

- Why would it give an error?

- Let's examine our variables:

```
Browse[1]> r
```

```
[1] NaN
```

```
Browse[1]> x
```

```
[1] -2
```

- And consider `h` once more:

```
> h <- function(x)
+ {
+     r <- sqrt(x)
+     sum(1:r)
+ }
```

- So now we can see that `h` was called with `x = -2`, that caused `r <- NaN`, and then trying to do `1:NaN` caused the error

- But what if that doesn't tell us where the bug is? Why was `h` called with `x = -2`?

```
Browse[1]> c
```

```
Enter a frame number, or 0 to exit

1: f(1)
2: g(x - 3)
3: h(x)
```

- Now we can go look at the frame for `g`:

```
Selection: 2
```

```
Called from: f(1)
```

```
Browse[1]>
```

- We can examine variables in this frame, go look at the frame for `f`, or whatever

- Let's assume we've done that and we've found the bug. Here's how we finish up:

```
Browse[1]> Q
> options(error=NULL)
```

- That last line turns off "recover" mode
- Many people, however, run a lot of the time with `options(error=recover)`

?trace

Debugging in R: `trace`

- Finally, suppose you wanted to debug an R core function like `abs`:

```
> abs
```

```
function (x)  .Primitive("abs")
```

```
> debug(abs)
> abs(-17)
```

```
[1] 17
```

Well, that didn't work. Why not?

```
> isdebugged(abs)
```

```
[1] TRUE
```

```
> undebug(abs)
```

- There is only one way to do this: `trace`

```
> trace(abs)
> sapply(-2:2, FUN=abs)
```

```
trace: FUN(-2:2[[1L]], ...)
trace: FUN(-2:2[[2L]], ...)
trace: FUN(-2:2[[3L]], ...)
trace: FUN(-2:2[[4L]], ...)
trace: FUN(-2:2[[5L]], ...)
[1] 2 1 0 1 2
```

```
> untrace(abs)
```

- So now we get a console log every time `abs` is called, showing who called it
- But suppose we actually want to debug a specific call to `abs`, not just monitor it?

- The `trace` command can do this too:

```
> trace(abs, tracer=quote(if (...[[1]]==-1)
recover()), print=FALSE)
```

```
Tracing function "abs" in package "base"
[1] "abs"
Warning message: ...
```

```
> abs(-7)
```

```
[1] 7
```

```
> abs(-1)
```

```
Enter a frame number, or 0 to exit

1: abs(-1)
```

```
Selection: 0
```

```
[1] 1     # the return value from abs(-1)
```

```
untrace(abs)
```

- That last example may seem over the top
- But when you hit a really complicated bug, these tools can be invaluable
- The `trace` command is incredibly powerful, so keep it in your toolbox:
  - it can debug even R core functions
  - it can tell which package the traced function was called from ("break into the debugger when `sqrt` is called from `nlme`")
  - it can insert arbitrary debugging code at any position in the traced function

Debugging in R: Defensive Coding

- The best debugging tactic is to avoid writing bugs in the first place:
  - break up complex logic into functions that can be independently tested and reused
  - use comments and good, consistent formatting to improve code readability
  - be aware of language issues that make certain constructs particularly bug-prone
  - check for unexpected conditions and generate warnings and errors as appropriate

Debugging in R: Defensive Coding

- To generate a warning in your code:

```
> if ((x < 0) || (x > 1))
+     warning("x outside expected range")
```

- To generate an error:

```
> if ((x < 0) || (x > 1))
+     stop("x outside expected range")
```

19

- Best, to make an **assertion**:

?stopifnot

```
> stopifnot((x >= 0) && (x <= 1))
> stopifnot(x > 1000)
```

```
Error: x > 1000 is not TRUE
```

Debugging in R: Defensive Coding

- Sprinkle your code liberally with assertions of things you "know":
  - the type or class of objects
  - the ranges of values
  - the number of elements in vectors
  - the absence of `NA` or `NULL` (if known)
- This style of coding is particularly important in R because R does so many unexpected things silently!

Exercise 3

`?stopifnot`

`?warning`

- Write a function `safe_power` that takes two parameters, `base` and `exponent`, and uses the `^` operator to raise the base to the exponent
- You know that you will be calling this function only with positive values, and only with single values (not vectors of length greater than 1)
- You wish to produce a warning before returning `NULL`, `NA`, `NaN`, or `Inf` (infinity)
- Test your function against the given test suite

Debugging in R: A Review

- Debugging techniques:
  - `debug`, `undebug`, `debugonce`
  - `browser`
  - `traceback`
  - `recover`
  - `trace`
- Defensive coding:
  - `warning`, `stop`, `stopifnot`
  - use functions and comments
  - be aware of dangerous language constructs

Caveats of R

Thanks to Patrick Burns for examples used in this section
http://www.burns-stat.com/pages/Tutor/R_inferno.pdf

Caveats in R

- R is a weird and tricky language
  - It often screws you by trying to be "helpful"
  - It has many surprising little quirks of behavior
- The definitive source on difficulties in using R is The R Inferno by Patrick Burns:
  - http://www.burns-stat.com/pages/Tutor/R_inferno.pdf
  - "If you are using R and you think you're in hell, this is a map for you."
- I'll cover only a few of the biggest issues

Caveats in R: Floating Point

- Numbers in computers are represented in a way that can cause unexpected results:

```
> 0.1 == 0.3 / 3
```

```
[1] FALSE
```

20

- What's going on? How do they differ?

```
> 0.1 - 0.3 / 3
```

```
[1] 1.387779e-17
```

- The upshot:
  - Numerical error is omnipresent
  - Don't compare non-integers for equality!

?Methods

Caveats in R: Methods

- R is object-oriented; but this part of R is very strangely designed, so beware!

```
> mean
```

```
function (x, ...)
UseMethod("mean")
<environment: namespace:base>
```

21

- `mean` is an example of a "method": a function whose implementation depends upon the type of object passed to it
- The main caveat here is that code for many functions is confusingly hidden

?Syntax

Caveats in R: Precedence

- This one bites everybody sooner or later:

```
> n <- 5
> 1:n-1
```

```
[1] 0 1 2 3 4
```

22

- The `:` operator has higher precedence than the `-` operator, which means it binds first; so the above is `(1:n) - 1`
- Use parentheses to fix this:

```
> 1:(n-1)
```

```
[1] 1 2 3 4
```

?is.na

Caveats in R: NA

- You want to know if `x` is `NA`:

```
> x <- 5
> x == NA
```

```
[1] NA
```

```
> x <- NA
> x == NA
```

```
[1] NA
```

- Comparing `NA` to anything gives `NA`!
- Use `is.na` to address this:

```
> is.na(NA)
```

```
[1] TRUE
```

?is.null

Caveats in R: NULL

- Similarly, comparing `x` to `NULL`:

```
> x <- 5
> x == NULL
```

```
logical(0)
```

```
> x <- NULL
> x == NULL
```

```
logical(0)
```

- Comparing `NULL` to things: `logical(0)`
- Use `is.null` to address this:

```
> is.null(NULL)
```

```
[1] TRUE
```

?coerce

Caveats in R: Coercion

- Calculations that mix types can surprise:

```
> "50" < 7
```

```
[1] TRUE
```

25

- In such calculations, variables are "promoted" to the most general type used in the calculation:
  - logical (least general)
  - integer
  - numeric
  - complex
  - character (most general)

Caveats in R: Coercion

- So for our test case, it becomes:

```
> "50" < "7"
```

```
[1] TRUE
```

(alphabetical order)

- To control this process, use the `as.*` functions to explicitly coerce variables:

```
> as.numeric("50") < 7
```

```
[1] FALSE
```

Caveats in R: Spaces

- A simple test:

```
> x <- -5
> (x>-7) && (x<-3)
```

```
[1] TRUE
```

26

- Seems fine, right?

```
> x
```

```
[1] 3
```

- Deadly. `x<-3` reads as `x <- 3`, not as `x < -3`! Always use spaces around `>` `<` `->` `<-` to avoid this issue!

Caveats in R: Dropping

- Suppose we make a matrix:

```
> M <- matrix(1:9, nrow=3, ncol=3); M
```

```
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```


- We can verify that it's a matrix:

```
> class(M)
```

```
[1] "matrix"
```

- A subset can give you another matrix:

```
> M[1:2, 1:2]
```

```
     [,1] [,2]
[1,]    1    4
[2,]    2    5
```

```
> class(M[1:2, 1:2])
```

```
[1] "matrix"
```

- Or it can give you a vector:

```
> M[1:2, 2]
```

```
[1] 4 5
```

```
> class(M[1:2, 2])
```

```
[1] "integer"
```

- This behavior is called "dropping": the dimensionality drops to the lowest dimension needed for the data. You can override this with `drop=FALSE` :

```
> M[1:2, 2, drop=FALSE]
```

```
     [,1]
[1,]    4
[2,]    5
```

```
> class(M[1:2, 2, drop=FALSE])
```

```
[1] "matrix"
```

- If you know you want a matrix as a result, use `drop=FALSE` to prevent dropping

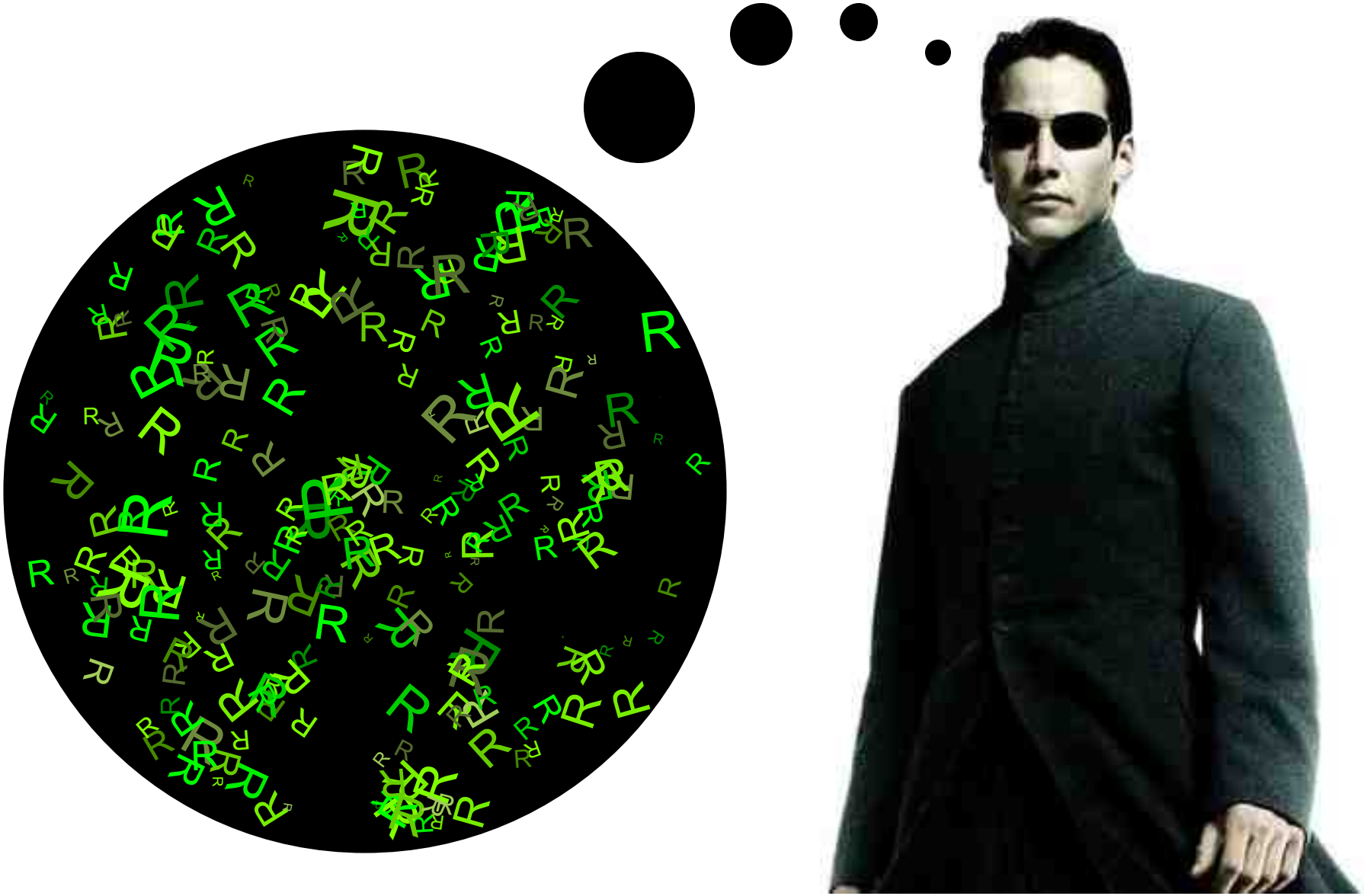
Caveats in R: Others

- I've mentioned a few caveats earlier:
  - `Error: unexpected 'else' in "else"`
  - reversal of the intended direction of a sequence when using the `:` operator
  - the extreme slowness of growing objects
  - automatic repetition of vectors
  - the dangers of ignoring warnings
- <u>The R Inferno</u> has lots more!

We are all together in The Matrix...

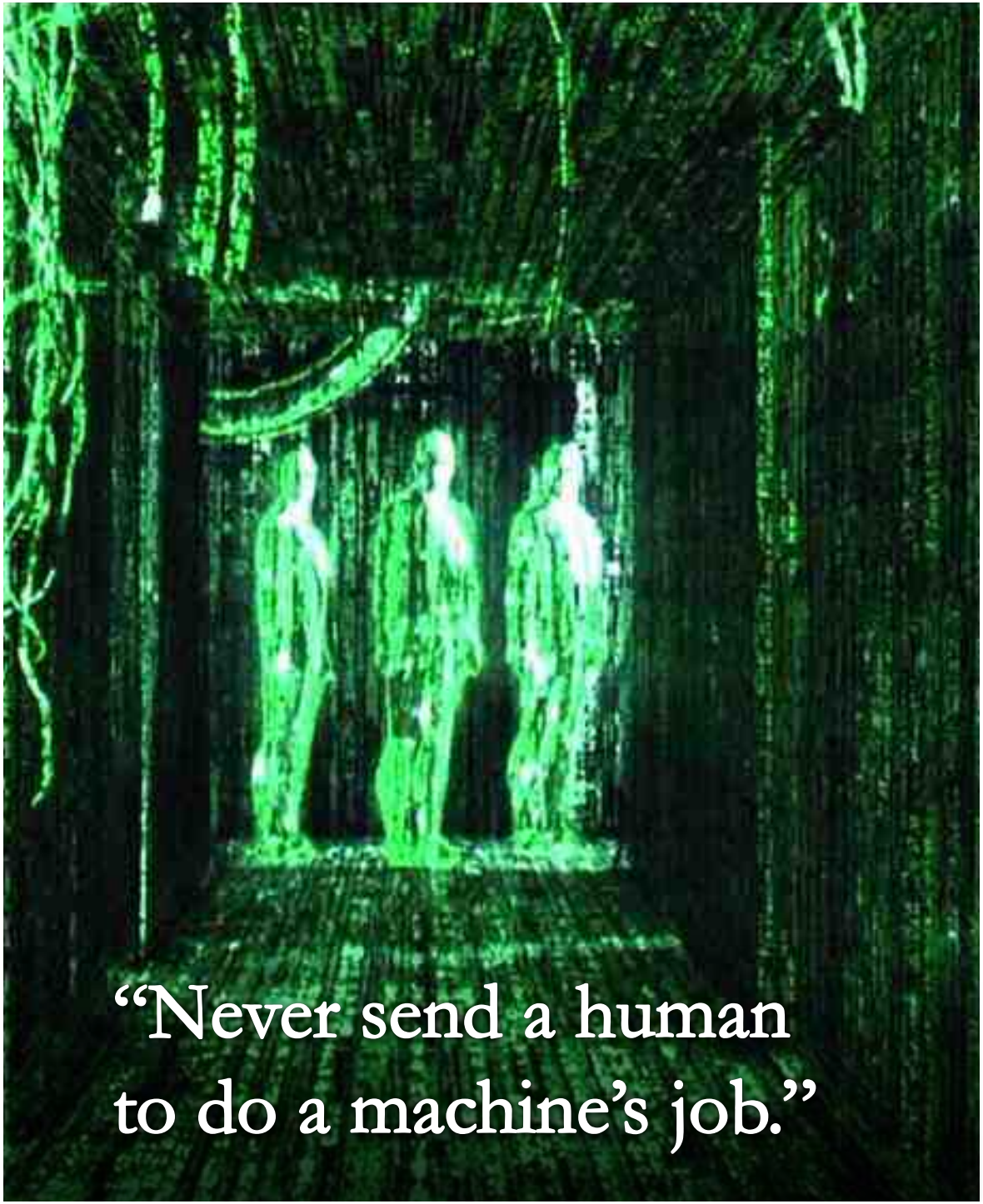
"I know why you're here, Neo. I know what you've been doing... why you hardly sleep, why you live alone, and why night after night, you sit by your computer... It's the question that drives us, Neo. It's the question that brought you here."

...and it's implemented in R



Exercises now online!

- Sample solutions to the exercises are now posted online:
  - http://bit.ly/yRjShO
  - https://sites.google.com/site/mcgillbgsa/workshops/programming_in_r
- I'll stick around now to work on them
- You can also email me:
  - ben dot haller [at] mail dot mcgill dot ca
- Remember, there are always many ways to solve any problem in R!



- Thanks to:
  - Jonathan Whiteley
  - Kiyoko Gotanda
  - Etienne Low-Décarie
  - Zofia Taranu
  - Corey Chivers
  - Morpheus

National Science Foundation
WHERE DISCOVERIES BEGIN

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. 1038597.