6. Writing Big Loops

Ken Rice
Thomas Lumley

Universities of Washington and Auckland

Seattle, July 2014

Writing loops in R

We saw that `replicate()`, and `apply()`, `sapply()` are R's preferred way of looping (doing the same thing many times).

Even for expert useRs, their use requires some careful thought; debugging code may be complex. Also, there are some jobs (e.g. iteration) that these loops cannot do.

In this session we'll talk about some alternatives, and their application to genome-wide studies.

for() loops

Your first computer program?

```
for(i in 1:100){
    print("Hello world!")
    print(i*i)
}
```

- Everything inside the curly brackets {...} is done 100 times

- Looped commands can depend on `i` (or whatever you called the counter)

- `R` creates a vector `i` with `1:100` in it. You could use **any vector that's convenient**

`for()` loops

`for()` loops are very intuitive, but have some drawbacks;

- They can be **slow**;

  - 'growing' a large dataset is a bad idea;
    `mydata <- cbind(mydata, rnorm(1000, mean=i))`

  - so set up blank output **first**, then 'fill it in'

- `apply()` is interpreted slightly faster than `for()` − but typically this will not matter, contrary to some urban myths

- `for()` loops require more typing than `apply()`! For tasks which will be repeated, writing a function really is the Right Thing to do, in the long run.

Using `for(i in 1:N)` sets up a vector (`i`) of length `N`. Do you really need this?

for() loops

Two alternatives; (see ?Control for details)

```
i <- 1; my.mat <- matrix(NA, N, 3)
while(i <= N){
  z <- work.on.gene(i)
  my.mat[i,] <- summary(z)
  i <- i+1
  }
```
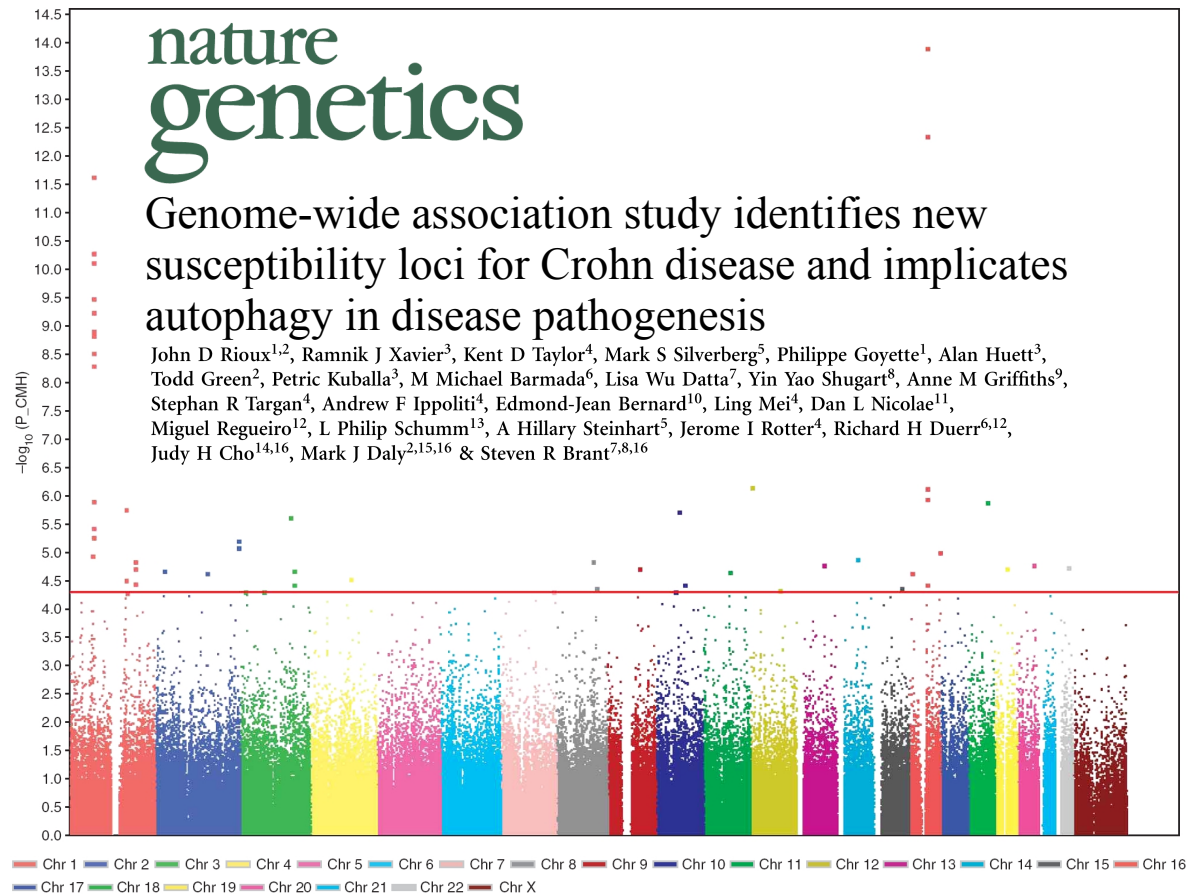
– note that we avoided 'growing' the output

```
i <- 1; my.mat <- matrix(NA, N, 3)
repeat{
  z <- work.on.gene(i)
  my.mat[i,] <- summary(z)
  i <- i+1
  if(i>=N) break
  }
```

Use apply(), sapply() to avoid the 'setup' stage

Application to whole-genome study

Whole genome studies look **very intimidating** …

Application to whole-genome study

... however, each p-value on that picture comes from a single logistic regression.

There may be 2,500,000 tests in total; if each one takes 1/10 sec, the analysis is done in under an hour;

Time per test	Total time	
0.01 sec	7 hrs	
0.1 sec	2 days 22 hrs	
1 sec	28 days	(!)
5 sec	5 months	(!!)
5 mins	24 yrs	(!!!)

Cutting time per test from 1 sec $\rightarrow$ 0.1 sec is clearly worthwhile − but *proposing* analyses where each test takes > 5 secs may be silly.

Making code run faster, part 1

Some easy 'streamlining' ideas;

- Write a function to do just the analysis you want
  ```
  > my.output <- apply(my.data, 1, my.function)
  ```

- Make a compiled version of it − use `cmpfun()` in the (standard) `compiler` package

- Pre-process/'clean' your data before analysis; e.g. `sum(x)/length(x)` doesn't error-check like `mean(x)`

- Similarly, you can streamline `glm()` to just `glm.fit()` [as we'll see, in some examples]

- Use vectorized operations, where possible

- Store data as matrices, not data.frames

Making code run faster, part 2

Streamlining, for experts;

- Write small but *important* pieces of code in `C`, and call these from `R`

- **Batch mode** processing lets you break down e.g. the whole genome into 23 chromosomes − great if you have 23 processors to use.

  - Save your analysis in 23 output files

  - read in the answers

  - finally produce e.g. multi-color pictures

  Various packages help implement this, but use is platform-specific

Timing

"Premature optimization is the root of all evil"

Donald Knuth

Do you need to optimize your code? Running 2 or 3 times faster may **not be worth the time spent coding/debugging!**

But going an **order of magnitude** faster is A Good Thing.

After you have code that works, you may need to speed it up. Experienced useʀs may be able to 'eyeball' the problem; measurement is an **easier and more reliable** approach

Timing

- `proc.time()` returns the current time. Save it before a task and subtract from the value after a task.

- `system.time()` times the evaluation of expression

- `R` has a **profiler**; this records which functions are being run, many times per second. `Rprof(filename)` turns on the profiler, `Rprof(NULL)` turns it off. `summaryRprof(filename)` reports how much time was spent in each function.

Remember that a 1000-fold speedup in a function used 10% of the time is **less helpful** than a 30% speedup in a function used 50% of the time.

High-throughput code — caveats

We saw yesterday that 'weird' datasets can crash your code. These **will appear** in genome-wide studies, and a crash at SNP 299,999 will be **very frustrating**.

- Some 'weirdness' is easy to spot;

  - Everyone is homozygous
  - All cases missing
  - No variation in outcome ...

- In more complex models, it's easier to 'try it and see'. Use `tryCatch()`

- When 'weirdness' is found, high-throughput code should;
  - Produce sensible output (`NA, -999` etc)
  - Handle these appropriately in summary output