Writing Loops

Ken Rice
Thomas Lumley

UW Biostatistics

Seattle, June 2009
Writing loops in R

We saw (Day 2 AM 1) that apply, sapply are R’s preferred way of looping (doing the same thing many times)

Even for expert users, their use requires thinking hard, and debugging code is complex. (A “write-only” language?)

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

Your first computer program?

```r
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)

- \textit{R} creates a vector `i` with `1:100` in it. You could use \textbf{any vector that’s convenient}
for loops

for loops are very intuitive, but have some drawbacks;

- Can be **slow**;
  - ‘growing’ the dataset is a bad idea;
    ```r
    mydata <- cbind(mydata, rnorm(1000, mean=i))
    ```
  - 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** requires more typing than **apply**! For tasks which will be repeated, writing a function 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)

```r
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

```r
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
Whole genome studies look very intimidating ...

Genome-wide association study identifies new susceptibility loci for Crohn disease and implicates autophagy in disease pathogenesis

Application to whole-genome study

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

There are 304,413 tests in total; if each one takes $1/10$ sec, the analysis is done in under an hour:

<table>
<thead>
<tr>
<th>Time per test</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01 sec</td>
<td>51 mins</td>
</tr>
<tr>
<td>0.1 sec</td>
<td>8 hours 27 mins</td>
</tr>
<tr>
<td>1 sec</td>
<td>3 days 12.5 hrs</td>
</tr>
<tr>
<td>5 sec</td>
<td>17 days 15 hrs (!)</td>
</tr>
<tr>
<td>5 mins</td>
<td>3 yrs 11 months (!!!)</td>
</tr>
</tbody>
</table>

Cutting time per test from 1 sec $\rightarrow$ 0.1 sec is **clearly worthwhile**

**Proposing** analyses where each test takes $> 5$ secs is **silly.**
Some easy ‘streamlining’ ideas;

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

- 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` [see examples]

- Use vectorized operations, where possible

- Store data as matrices, not data.frames
Streamlining, for ‘experts-only’

- 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
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 users 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