Efficient *R* Programming

Martin Morgan
Fred Hutchinson Cancer Research Center

17-18 February, 2011

1 Pitfalls

These brief exercises are meant to illustrate some common obstacles to efficient R programming. The idea is that you'll follow along with the text, evaluating the instructions in your own R session.

The basic scenario is a genome-wide association study. There are 1000 individuals. Case versus control status, gender, and age were recorded for each, along with genotype at ≈100000 SNPs. The data is entirely synthetic.

1.1 Getting going

To begin:

1. Start an *R* session, and use `sessionInfo()` to confirm that you have version 2.12.1.

2. Install the *AdvancedR2011Data* package from the thumb drive distributed at the start of the class.

3. Install the course package with

```
source("http://bioconductor.org/course-packages/courseInstall.R")
courseInstall("AdvancedR2011")
```

4. Ensure that the installation has been successful by loading the *AdvancedR2011* package

```
> library(AdvancedR2011)
```

5. Read several pre-defined functions in to the R session

```
> fl <- system.file("script", "efficient.R",
+                   package="AdvancedR2011")
> source(fl)
```

1.2 Basic performance measurement and data I/O

The first activities are meant to illustrate the use of `system.time` to evaluate performance, `object.size` to investigate how much memory an object uses, and `identical` and `all.equal` to compare objects. We also look at how to more effectively read data in to R.

First, let's take a look at `fname0`, a file path to the GWAS data, and the function `f0`. Both `fname` and `f0` are defined in the `efficient.R` file. Here are the definitions, and the result assigned to the variable `gwas0`; we read in just 3 rows of data, to keep the exercise managable. Let's use `system.time` to evaluate how long this takes

```
> fname0

[1] "/Library/Frameworks/R.framework/Versions/2.12/Resources/library/AdvancedR2011Data/extda

> f0

function (fileName, nrows = 5, ...)
{
    read.csv(fileName, nrows = nrows, row.names = NULL, header = FALSE,
        ...)
}

> system.time(gwas0 <- f0(fname0, nrows=3))

   user  system elapsed
 26.584   0.074  26.772

> dim(gwas0)

[1]      3 113735
```

Note the way in which the `<-` assignment to variable `gwas0` occurs in `system.time`. See the help page `?system.time` to understand the output; we're usually interested in the `user.time`.

The function `f0` does its job, but perhaps we can improve on it. For instance, suppose we were interested in a preliminary investigation, and in particular reading just the first 100 SNPs. The following shows two different functions that allow us to read in just the information we're interested in

```
> f1

function (fileName, nrows = 5, ncols = .ncols, keep = 1:100,
    ...)
{
    colClasses <- rep("NULL", ncols)
    colClasses[keep] <- "integer"
    f0(fileName, nrows = nrows, colClasses = colClasses, ...)
}
```

```
> f2

function (fileName, nrows = 5, ncols = .ncols, keep = 1:100,
    ...)
{
    what <- rep(list(NULL), ncols)
    what[keep] <- list(integer())
    input <- scan(fileName, what = what, sep = ",", nmax = nrows,
        ..., quiet = TRUE)
    df <- as.data.frame(Filter(is.integer, input))
    names(df) <- paste("V", keep, sep = "")
    df
}
```

Compare these functions with each other and `f0`, and with the relevant help pages `?read.csv`, `scan` to identify the key steps for efficient data input. Now lets see how they perform in terms of evaluation time:

```
> system.time(gwas1 <- f1(fname0, nrows=3))

   user  system elapsed
 23.932   0.052  24.110

> system.time(gwas2 <- f2(fname0, nrows=3))

   user  system elapsed
  0.261   0.003   0.264
```

Since we've read in the just the data we're interested in, we should have saved quite a bit of space

```
> object.size(gwas0)

9099128 bytes

> object.size(gwas1)

8320 bytes
```

The functions `f1` and `f2` are meant to return the same result, just implemented in slightly different ways. We can verify this with `identical`

```
> identical(gwas1, gwas2)

[1] TRUE
```

1.3 Character manipulation

Here we'll look at one aspect of R that can sometimes have a surprising performance penalty and sometimes tricky semantics: character manipulation. We'll learn some additional techniques for monitoring performance, as well as gain an appreciation for the benefits of appropriate function choice.

Let's look at the the genotypic data only, by dropping the first three columns from the data frame; we'll take a peak at the first six rows (`head`) of the first five columns of the genotype information.

```
> gtype <- f2(fname0, keep=1:10000, nrows=-1)
> head(gtype[,1:5])

  V1 V2 V3 V4 V5
1  3  1  1  1  1
2  2  1  1  1  1
3  3  1  1  1  1
4  3  1  1  1  1
5  2  1  1  1  1
6  2  1  1  1  1
```

Note that the columns have names (e.g., `id_V1`).

A function `shuffle0` might come in handy if one were wanting to randomize genotypes, and to return the randomized genotypes as a matrix.

```
> shuffle0

function (genotypes, seed = 123L)
{
    set.seed(seed)
    samp <- sample(genotypes)
    g <- unlist(samp)
    matrix(g, ncol = ncol(genotypes))
}
```

We use `seed` to make the results reproducible across invocations. `sample(genotypes)` permutes the columns of the `gtype` data frame. The `unlist` and `matrix` commands are meant as a first attempt at creating a matrix from our permuted data frame – we 'collapse' the sampled genotypes into a vector, and then shape the vector into a matrix. Let's measure how long this takes:

```
> system.time(s0 <- shuffle0(gtype))

   user  system elapsed
 35.152   0.531  38.547
```

Wow, that seems like a fairly long time for a relatively simple set of operations. I wonder what's going on?

```
> profFile <- tempfile()
> Rprof(profFile)      # start gathering profile information
> s0 <- shuffle0(gtype)
> Rprof(NULL)          # stop
> head(summaryRprof(profFile)$by.self)
```

```
               self.time self.pct
"unlist"           13.18    94.96
"structure"         0.40     2.88
"matrix"            0.14     1.01
"as.vector"         0.12     0.86
"attributes<-"      0.04     0.29
               total.time total.pct
"unlist"            13.18     94.96
"structure"          0.44      3.17
"matrix"             0.26      1.87
"as.vector"          0.12      0.86
"attributes<-"       0.04      0.29
```

A large fraction of the time is spent on the `unlist` function. A little experimentation suggests what the problem might be:

```
> gsubset <- gtype[1:2, 1:3]
> unlist(gsubset)
```

```
V11 V12 V21 V22 V31 V32
  3   2   1   1   1   1
```

Notice that the value of `unlist` is a vector with names, and that the names have been constructed to be unique. We can reference the help page `?unlist`, and arrive at a better solution `shuffle1` that avoids creating names.

```
> shuffle1
```

```
function (genotypes, seed = 123L)
{
    set.seed(seed)
    samp <- sample(genotypes)
    g <- unlist(samp, use.names = FALSE)
    matrix(g, ncol = ncol(genotypes))
}
```

This almost trivial change has a big influence on performance, without changing the result:

```
> system.time(s1 <- shuffle1(gtype))
```

```
   user  system elapsed
  0.156   0.116   0.272
```

```
> identical(s0, s1)

[1] TRUE
```

Finally, in *R* its common to be able to 'cast' from one data structure to another. `shuffle2` does this

```
> shuffle2

function (genotypes, seed = 123L)
{
    set.seed(seed)
    as.matrix(sample(genotypes))
}

> system.time(s2 <- shuffle2(gtype))

   user  system elapsed
  0.444   0.087   0.530
```

Note that the performance of `as.matrix` is comparable to our `shuffle1`. Are the results the same?

```
> identical(s1, s2)

[1] FALSE
```

Oh oh! This doesn't look good. But maybe it's just that our results `s1` do not have `dimnames`, whereas `s2` might?

```
> all.equal(s1, s2)

[1] "Attributes: < Length mismatch: comparison on first 1 components >"

> all.equal(s1, s2, check.attributes=FALSE)

[1] TRUE
```

(The result of the first call to `all.equal` is fairly cryptic; this is, unfortunately, typical.) The built-in function `as.matrix` is performing as well as our version, and is doing a better job of tracking important information (row and column names) through the analysis.

2 Data I/O: Streaming

It can be difficult to fit large data into memory, and it is increasingly necessary to think of 'streaming' algorithms that process a small 'chunk' of data at a time. The following defines a function `.fapply` that (tries to!) stream data from a file through a function, much like `lapply` streams elements of a list through a function.

```
> .fapply

function (con, FUN, ..., .get, .reduce)
{
    result <- list()
    it <- 1
    while (nrow(chunk <- .get(con))) {
        message("chunk ", it)
        result[[it]] <- FUN(chunk, ...)
        it <- it + 1
    }
    .reduce(result)
}
```

The function takes an open file connecton, a function to be applied to each chunk, and parameters influencing how the chunks are input and processed. Let's give it a whirl, by reading the GWAS file in chunks of 100 rows at a time. For each chunk, we'll calculate the number of individuals in the chunk, and the heterozygotes for each SNP. Here's our function

```
> heterozygosity <- function(chunk, ...)
+     list(N=nrow(chunk), Het=colSums(chunk == 2))
```

The `.get` argument to `.fapply` is meant to retrieve a chunk of data, and takes as its single argument an open *R* connection. Here's our `get`:

```
> get <- function(con)
+     f2(con, nrows=100, ncols=.ncols, keep=1:100)
```

A common operation, both in stream processing and in distribution of tasks for parallel evaluation, is to 'reduce' the results from different chunks / tasks into a single meaningful object. The `.reduce` argument to `.fapply` is meant to be a function that performs the reduction. It expects a list, with each element of the list the result of the function operating on a chunk, e.g., each element being the result of `heterozygosity`. The result should be a vector of heterozygosities that we can manipulate or visualize in the usual way. Here's our simple reduce function:

```
> reduce <- function(lst)
+ {
+     n <- sum(sapply(lst, "[[", "N"))
+     het <- Reduce("+", lapply(lst, "[[", "Het"))
+     browser()
+     het / n
+ }
```

Let's give this a go...

```
> con <- file(fname0, open="r")
> res <- .fapply(con, heterozygosity, .get=get, .reduce=reduce)
```

Figure 1: Individual heterozygosity, 'stream' processing

```
Called from: .reduce(result)

> close(con)
> length(res)

[1] 100

> library(lattice)
> print(densityplot(res, plot.points=FALSE, xlab="Heterozygosity",
+                   main="Stream"))
```

As exercises:

1. Explore different functions that might usefully be used in a streaming context. For instance, suppose the goal is to calculate heterozygosities for each individual, rather than each SNP. What might the `FUN` and `.reduce` arguments to `.fapply` look like?
2. As an advanced exercise, consider how `.fapply` might be modified to work in parallel, for instance using the *multicore* or *Rmpi* packages.
3. The `.reduce` argument takes the complete list of results, and reduces it. This will not be a good strategy if the result of `FUN` is itself large. Revise `.fapply` so that `.reduce` is a function that takes as its first argument the current reduction, and as its second argument the current chunk with `FUN` already applied.