# Stat243: Problem Set 3, Due Wednesday Oct. 23

October 7, 2013

This covers material in Units 6 and 8 on R programming and parallel processing.
It's due at the start of class on 10/23.
Some guidelines on how to present your solutions:

1. Please use your knitr/Sweave/R Markdown solution from PS1, problem 5 as your template for how to format your solutions (only non-Statistics students are allowed to use R Markdown).

2. As with PS1, your solution should not just be code - you should have text describing how you approached the problem and what the various steps were.

3. In addition to the paper submission, which is the primary thing we will grade, please turn in your raw Latex/Markdown with embedded code electronically on bSpace (a file just containing the code with indication of what pieces of code are for what problems is fine too).

4. All your code should have comments indicating what each function or block of code does, and for any lines of code or code constructs that may be hard to understand, a comment indicating what that code does. You do not need to show exhaustive output but in general you should show short examples of what your code does to demonstrate its functionality. Depending on how long different pieces of your code are, you may want to have some of it be an appendix rather than part of the main solution.

Please note my comments in the syllabus about when to ask for help and about working together.

## Problems

1. The following example comes from a problem encountered by one of the current Statistics graduate students for his thesis work. He needed to compute the likelihood for an overdispersed binomial random variable with the following probability mass function (pmf):

$$\begin{aligned} P(Y &= y) = \frac{f(y; n, p, \phi)}{\sum_{k=0}^{n} f(k; n, p, \phi)} \\ f(k; n, p, \phi) &= \binom{n}{k} \frac{k^k (n-k)^{n-k}}{n^n} \left( \frac{n^n}{k^k (n-k)^{n-k}} \right)^{\phi} p^{k\phi} (1-p)^{(n-k)\phi}, \end{aligned}$$

   where the denominator serves as a normalizing constant to ensure this is a valid probability mass function. Your job is to write code to evaluate the denominator of $P(Y = y)$. In the graduate student's work, he needs to evaluate $P(Y = y)$ many many times, so efficient calculation of the denominator is important. For our purposes here you can take $p = 0.3$ and $\phi = 0.5$ when you need to actually run your function.

(a) First, write code to evaluate the denominator using *apply()/lapply()/sapply()*. Make sure to calculate all the terms in $f(k; n, p, \phi)$ on the log scale to avoid numerical issues, before exponentiating and summing. Describe briefly what happens if you don't do the calculation on the log scale. Hint: `?Special` in R will tell you about a number of useful functions. Also, recall that $0^0 = 1$.

(b) Now write code to do the calculation in a fully vectorized fashion with no loops or *apply()* functions. Compare the relative timing of (a) and (b) for some different values of $n$ ranging in magnitude from around 10 to around 2000.

(c) (Extra credit) Extra credit will be based on whether your code is as fast as my solution. My code on the SCF machine arwen uses approximately 0.11 seconds for 100 replications with $n = 2000$. I'd suggest using *Rprof()* to assess the steps in your code for (b) that are using the most time and focusing your efforts on increasing the speed of those parts of the code.

2. Function coding and efficiency. Consider a discrete random walk in two dimensions. At each step there is a 0.25 probability of moving left, right, up and down. An example of a random walk of three steps would be the first step is to move one unit to the right, the second step is to move one unit up, and the third step is to move one unit back down to the position attained after the first step.

   (a) Write a function that generates such a random walk. The input argument should be the number of steps to be taken. There should be an optional second argument (with a default) specifying whether the user wants the full path of the walk returned or just the final position. If the former, the result should be given in a reasonable format. Your code should check that the input is a valid integer (it should handle zero and negative numbers and non-integers gracefully). Finally, use *Rprof()* to assess where the bulk of the computation is in your code.

   (b) Now try a variety of ways to speed up your code and report the timing results (both for your successful and unsuccessful attempts) in your answer. Try to figure out at least one approach that avoids using any loops. Hint: the *cumsum()* function may come in handy.

3. Now embed the fastest version of your code from problem #3 in object-oriented code that nicely packages up the functionality. You can choose S3, S4, or Reference Classes, but if you already have a fair amount of experience with S3, you should try one of the other two. You should create a class, called '*rw*', with a constructor, a *print* method (for which the result should focus on where the final position is and some useful summary measures of the walk), a *plot* method, and a '*[*' operator than gives the position for the $i$th step. Also, create a replacement method called *start* that translates the origin of the random walk, e.g., `start(myWalk) = c(5, 7)` should move the origin and the entire walk so that it starts at the position *x=5, y=7*.

4. The following is the code (borrowed from a recently-graduated Statistics grad student) mentioned in Section 3 of Unit 6. The inputs to the algorithm need to be whole numbers.

```
fastcount <- function(xvar, yvar) {
    nalineX <- is.na(xvar)
    nalineY <- is.na(yvar)
    xvar[nalineX | nalineY] <- 0
    yvar[nalineX | nalineY] <- 0
    useline <- !(nalineX | nalineY)
    # Table must be initialized for -1's
    tablex <- numeric(max(xvar) + 1)
```

```
    tabley <- numeric(max(xvar) + 1)
    stopifnot(length(xvar) == length(yvar))
    res <- .C("fastcount", PACKAGE = "GCcorrect", tablex = as.integer(tablex),
        tabley = as.integer(tabley), as.integer(xvar), as.integer(yvar),
        as.integer(useline), as.integer(length(xvar)))
    xuse <- which(res$tablex > 0)
    xnames <- xuse - 1
    resb <- rbind(res$tablex[xuse], res$tabley[xuse])
    colnames(resb) <- xnames
    return(resb)
}
```

(a) Assume for the sake of concreteness that the user passes in two vectors, each of equal size (80 Mb), into the function and that example input vectors can be created with: `sample(c(seq(1, 20, by = 1), NA), n, replace = TRUE)` where `n <- 1e7`. Your job is to report when any new large objects are created in memory (even temporarily) and the amount of memory (in Mb) used at the point in the code that memory use reaches its maximum. You can ignore objects of negligible size relative to the larger objects. Note that numeric values (i.e, double precision floating point values) each take 8 bytes per value and integers and booleans 4 bytes per value. You can assume that negligible memory is used in the C code called with ".C" or in passing the result of `.C("fastcount",...)` back to R. Also assume that the object *res* and its subcomponents take up negligible memory.

(b) Rewrite the function to minimize memory use. You can also note whether there are ways to reduce memory use based on preprocessing before the values are passed into the function. I was able to keep memory use to about 200 Mb at the conclusion of any given line of code, with *gc()* reporting 240 Mb as the 'max used'.

(c) (Extra credit) Extra credit may be be granted for particularly nice solutions to (b).

Note that on even an old computer, the memory use in this example would not be something to be concerned about. But if one made each of those vectors 800 Mb or 8 Gb, then these considerations would be quite salient.

5. Here we'll consider the speed of matrix multiplication using parallel processing techniques.

(a) On a machine with multiple cores, compare the speed of multiplying an $n \times n$ matrix by another such matrix, for $n = 4000$, using a single core and using threaded linear algebra, where you should choose the maximum number of threads, $p$, based on the machine you are using (but don't use $p > 8$). The optimal speedup is $p$-fold - how close do you come to achieving that? Make a plot that illustrates the speed-up compared to the optimal speedup for a variety of values of $n$.

(b) Now write some code with *foreach* that breaks up the problem into $p$ chunks, where you use $p$ workers. Compare the speed to what you got in (a). Use multiple cores with *foreach*, but make sure you are not using threaded linear algebra. Make sure that *foreach* recombines the result into the correct final matrix.

To make things simpler in (b), feel free to choose $n$ such that it is divisible by $p$, but still with $n \approx 4000$ and use the same $n$ for (a) and (b).