Profiling, timing, and code efficiency

Profile code to find where it is slow. Time competing code fragments. Revise code for speed.

Profile a program

To profile a slow `myScript.R` and find its bottleneck,

- Add `Rprof(filename="Rprof.out", line.profiling=TRUE)` to `myScript.R` to turn on line profiling: every `interval=.02` seconds, R will record in `filename` which code line is running.
- Add `Rprof(NULL)` to `myScript.R` to turn profiling off.
- Run `myScript.R` via `source(file="myScript.R", keep.source=TRUE)`.
- Run `summaryRprof(filename="Rprof.out", lines="show")` on the profiling output.

e.g. See `nflProfile1.R`.

Time a code fragment

- `system.time(expr)` runs `expr` and shows the CPU time spent in user instructions (our code), the CPU time spent in operating system instructions (for I/O and other things) on behalf of `expr`, and the elapsed (stopwatch) time. e.g.

```
system.time(Sys.sleep(3))
system.time(readLines("http://imdb.com/top250"))
system.time({n=100000; x=rnorm(n); y=2*x+3+rnorm(n,0,.1); m=lm(y~x)})
x = runif(50)
system.time(sqrt(x))
n = 1000000
system.time({ for (i in seq_len(n)) { sqrt(x) } })
```

- `microbenchmark(..., times=100)` runs the expressions in `...`, repeating each one `times` times. It returns data (a `data.frame` / `microbenchmark` object) on the run time. It has its own loop and uses greater precision than `system.time()`, so no explicit loop is needed. e.g.

```
require("microbenchmark")
a = 2
b = microbenchmark(2 + 2, 2 + a, sqrt(x), x ^ .5)
str(b)
head(as.data.frame(b), n=15)
print(b)
tapply(X=b$time, INDEX=b$expr, FUN=summary)
boxplot(b)
require("ggplot2"); autoplot(b)  # ggplot2 is a graphics package
```

Code efficiency: speed up the bottleneck

- Read the help files. e.g. `nflProfile2.R`
- Avoid unnecessary copying. e.g. Are loops slow in R? See `loopTiming.R`.

  R's loops are not terribly slow. However, incrementally growing a data structure in a loop causes wasteful copying and can turn a $O(n)$ algorithm into $O(n^2)$. Be wary of this with `[]` (above), `c()`, `cbind()`, `paste()`, etc.. Instead, allocate the data structure before the loop.

  One of the reasons R is slow is that it copies data in many more subtle situations.
- Vectorize, where possible, by coding in terms of a vector, not a scalar. Then R makes a call to C (fast) to run the implicit loop. Commonly used vectorized functions include `rowSums()`, `colSums()`, `rowMeans()`, `colMeans()`, `cumsum()`, `diff()`. e.g.

```
n=10000; m=matrix(data=as.numeric(1:(n^2)), nrow=n, ncol=n)
system.time(s <- apply(m, 1, sum))
system.time(rs <- rowSums(m))
```

- Use byte code compilation for a small gain. (It's already done to base R functions.) e.g.

```
baby.sapply = function(X, FUN) {
  n = length(X)
  values.FUN = numeric(n)
  for (i in seq_len(n)) {
    values.FUN[i] = FUN(X[i])
  }
  return(values.FUN)
}
x = rnorm(n=10000)
microbenchmark(baby.sapply(X=x, FUN=abs))
baby.sapply.compiled = compiler::cmpfun(baby.sapply) # Compilation here.
microbenchmark(baby.sapply.compiled(X=x, FUN=abs))
```

- Write the slow part in C++ (fast) and call it from R: coming soon.
- Learn "CS 367: Data Structures" and "CS 577: Algorithms" for much deeper improvements.

Before speeding up any code, especially code not in the bottleneck, consider that "premature optimization is the root of all evil." (Donald Knuth: TAOCP, TEX)

Avoid gathering data repeatedly

`save(..., list=character(), file)` saves object(s) in `...`, or in character vector `list`, to `file`. (By convention, `file` ends `".RData"`.) `load(file)` loads objects saved by `save()`. e.g.

```
x.file = "x.RData"
if (file.exists(x.file)) {
  load(file=x.file)
} else {
  x = data.frame(height=1:3, weight=4:6) # ... or read 250 web pages, etc. ...
  save(x, file=x.file)
}
```