Circle 2

Growing Objects

We made our way into the second Circle, here live the gluttons.

Let's look at three ways of doing the same task of creating a sequence of numbers. Method 1 is to grow the object:

```
vec <- numeric(0)
for(i in 1:n) vec <- c(vec, i)
```

Method 2 creates an object of the final length and then changes the values in the object by subscripting:

```
vec <- numeric(n)
for(i in 1:n) vec[i] <- i
```

Method 3 directly creates the final object:

```
vec <- 1:n
```

Table 2.1 shows the timing in seconds on a particular (old) machine of these three methods for a selection of values of n. The relationships for varying n are all roughly linear on a log-log scale, but the timings are drastically different.

You may wonder why growing objects is so slow. It is the computational equivalent of suburbanization. When a new size is required, there will not be

Table 2.1: Time in seconds of methods to create a sequence.

n	grow	subscript	colon operator
1000	0.01	0.01	.00006
10,000	0.59	0.09	.0004
100,000	133.68	0.79	.005
1,000,000	18,718	8.10	.097

enough room where the object is; so it needs to move to a more open space. Then that space will be too small, and it will need to move again. It takes a lot of time to move house. Just as in physical suburbanization, growing objects can spoil all of the available space. You end up with lots of small pieces of available memory, but no large pieces. This is called fragmenting memory.

A more common—and probably more dangerous—means of being a glutton is with rbind. For example:

```
my.df <- data.frame(a=character(0), b=numeric(0))
for(i in 1:n) {
    my.df <- rbind(my.df, data.frame(a=sample(letters, 1),
        b=runif(1)))
}
```

Probably the main reason this is more common is because it is more likely that each iteration will have a different number of observations. That is, the code is more likely to look like:

```
my.df <- data.frame(a=character(0), b=numeric(0))
for(i in 1:n) {
    this.N <- rpois(1, 10)
    my.df <- rbind(my.df, data.frame(a=sample(letters,
        this.N, replace=TRUE), b=runif(this.N)))
}
```

Often a reasonable upper bound on the size of the final object is known. If so, then create the object with that size and then remove the extra values at the end. If the final size is a mystery, then you can still follow the same scheme, but allow for periodic growth of the object.

```
current.N <- 10 * n
my.df <- data.frame(a=character(current.N),
    b=numeric(current.N))
count <- 0
for(i in 1:n) {
    this.N <- rpois(1, 10)
    if(count + this.N > current.N) {
        old.df <- my.df
        current.N <- round(1.5 * (current.N + this.N))
        my.df <- data.frame(a=character(current.N),
            b=numeric(current.N))
        my.df[1:count,] <- old.df[1:count, ]
    }
    my.df[count + 1:this.N,] <- data.frame(a=sample(letters,
        this.N, replace=TRUE), b=runif(this.N))
    count <- count + this.N
}
my.df <- my.df[1:count,]
```

Figure 2.1: The giants by Sandro Botticelli.

Often there is a simpler approach to the whole problem—build a list of pieces and then scrunch them together in one go.

```
my.list <- vector('list', n)
for(i in 1:n) {
   this.N <- rpois(1, 10)
   my.list[[i]] <- data.frame(a=sample(letters, this.N
       replace=TRUE), b=runif(this.N))
}
my.df <- do.call('rbind', my.list)
```

There are ways of cleverly hiding that you are growing an object. Here is an example:

```
hit <- NA
for(i in 1:one.zillion) {
    if(runif(1) < 0.3) hit[i] <- TRUE
}
```

Each time the condition is true, `hit` is grown.

Eliminating the growth of objects can be one of the easiest and most dramatic ways of speeding up R code.

If you use too much memory, R will complain. The key issue is that R holds all the data in RAM. This is a limitation if you have huge datasets. The up-side is flexibility—in particular, R imposes no rules on what data are like.

You can get a message, all too familiar to some people, like:

```
Error: cannot allocate vector of size 79.8 Mb.
```

This is often misinterpreted along the lines of: "I have xxx gigabytes of memory, why can't R even allocate 80 megabytes?" It is because R has already allocated a lot of memory successfully. The error message is about how much memory R was going after at the point where it failed.

The user who has seen this message logically asks, "What can I do about it?" There are some easy answers:

1. Don't be a glutton by using bad programming constructs.
2. Get a bigger computer.
3. Reduce the problem size.

If you've obeyed the first answer and can't follow the second or third, then your alternatives are harder. One is to restart the R session, but this is often ineffective.

Another of those hard alternatives is to explore where in your code the memory is growing. One method (on at least one platform) is to insert lines like:

```
cat('point 1 mem', memory.size(), memory.size(max=TRUE), '\n')
```

throughout your code. This shows the memory that R currently has and the maximum amount R has had in the current session.

However, probably a more efficient and informative procedure would be to use Rprof with memory profiling. Rprof also profiles time use.

Another way of reducing memory use is to store your data in a database and only extract portions of the data into R as needed. While this takes some time to set up, it can become quite a natural way to work.

A "database" solution that only uses R is to save (as in the `save` function) objects in individual files, then use the files one at a time. So your code using the objects might look something like:

```
for(i in 1:n) {
   objname <- paste('obj.', i, sep='')
   load(paste(objname, '.rda', sep=''))
   the_obj <- get(objname)
   rm(list=objname)
   # use the_obj
}
```

Are tomorrow's bigger computers going to solve the problem? For some people, yes—their data will stay the same size and computers will get big enough to hold it comfortably. For other people it will only get worse—more powerful computers means extraordinarily larger datasets. If you are likely to be in this latter group, you might want to get used to working with databases now.

If you have one of those giant computers, you may have the capacity to attempt to create something larger than R can handle. See:

```
?'Memory-limits'
```

for the limits that are imposed.