Avoid copies

A pernicious source of slow R code is growing an object with a loop. Whenever you use `c()`, `append()`, `cbind()`, `rbind()`, or `paste()` to create a bigger object, R must first allocate space for the new object and then copy the old object to its new home. If you're repeating this many times, like in a for loop, this can be quite expensive. You've entered Circle 2 of the "R inferno".

Here's a little example that shows the problem. We first generate some random strings, and then combine them either iteratively with a loop using `collapse()`, or in a single pass using `paste()`. Note that the performance of `collapse()` gets relatively worse as the number of strings grows: combining 100 strings takes almost 30 times longer than combining 10 strings.

```
random_string <- function() {
  paste(sample(letters, 50, replace = TRUE), collapse = "")
}
strings10 <- replicate(10, random_string())
strings100 <- replicate(100, random_string())

collapse <- function(xs) {
  out <- ""
  for (x in xs) {
    out <- paste0(out, x)
  }
  out
}

microbenchmark(
  loop10  = collapse(strings10),
  loop100 = collapse(strings100),
  vec10   = paste(strings10, collapse = ""),
  vec100  = paste(strings100, collapse = "")
)
#> Unit: microseconds
#>     expr     min      lq   mean  median      uq     max neval
#>   loop10    49.5    51.0   54.8    52.0    58.4    89.7   100
#>  loop100 1,570.0 1,580.0 1649.2 1,590.0 1,670.0 2,180.0   100
#>    vec10    11.0    11.5   12.8    11.8    12.3    35.2   100
#>   vec100    79.6    80.2   83.5    80.6    84.7   104.0   100
```

Modifying an object in a loop, e.g., `x[i] <- y`, can also create a copy, depending on the class of `x`. Modification in place discusses this issue in more depth and gives you some tools to determine when you're making copies.

Byte code compilation

R 2.13.0 introduced a byte code compiler which can increase the speed of some code. Using the compiler is an easy way to get improvements in speed. Even if it doesn't work well for your function, you won't have invested a lot of time in the effort. The following example shows the pure R version of `lapply()` from functionals. Compiling it gives a considerable speedup, although it's still not quite as fast as the C version provided by base R.

```
lapply2 <- function(x, f, ...) {
  out <- vector("list", length(x))
  for (i in seq_along(x)) {
    out[[i]] <- f(x[[i]], ...)
  }
  out
}

lapply2_c <- compiler::cmpfun(lapply2)

x <- list(1:10, letters, c(F, T), NULL)
microbenchmark(
  lapply2(x, is.null),
  lapply2_c(x, is.null),
  lapply(x, is.null)
)
#> Unit: microseconds
#>                   expr  min    lq  mean median    uq  max neval
#>    lapply2(x, is.null) 9.92 11.30 13.42  11.80 12.40 49.2   100
#>  lapply2_c(x, is.null) 4.40  5.18  5.79   5.53  5.83 15.1   100
#>     lapply(x, is.null) 4.58  5.36  6.22   5.70  6.16 34.0   100
```

Byte code compilation really helps here, but in most cases you're more likely to get a 5-10% improvement. All base R functions are byte code compiled by default.

Case study: t-test

The following case study shows how to make t-tests faster using some of the techniques described above. It's based on an example in "Computing thousands of test statistics simultaneously in R" by Holger Schwender and Tina Müller. I thoroughly recommend reading the paper in full to see the same idea applied to other tests.

Imagine we have run 1000 experiments (rows), each of which collects data on 50 individuals (columns). The first 25 individuals in each experiment are assigned to group 1 and the rest to group 2. We'll first generate some random data to represent this problem:

```
m <- 1000
n <- 50
X <- matrix(rnorm(m * n, mean = 10, sd = 3), nrow = m)
grp <- rep(1:2, each = n / 2)
```

For data in this form, there are two ways to use `t.test()`. We can either use the formula interface or provide two vectors, one for each group. Timing reveals that the formula interface is considerably slower.

```
system.time(for(i in 1:m) t.test(X[i, ] ~ grp)$stat)
#>    user  system elapsed
#>     1.4     0.0     1.4
system.time(
  for(i in 1:m) t.test(X[i, grp == 1], X[i, grp == 2])$stat
)
#>    user  system elapsed
#>   0.348   0.000   0.354
```

Of course, a for loop computes, but doesn't save the values. We'll use `apply()` to do that. This adds a little overhead:

```
compT <- function(x, grp){
  t.test(x[grp == 1], x[grp == 2])$stat
}
system.time(t1 <- apply(X, 1, compT, grp = grp))
#>    user  system elapsed
#>   0.341   0.000   0.341
```

How can we make this faster? First, we could try doing less work. If you look at the source code of `stats:::t.test.default()`, you'll see that it does a lot more than just compute the t-statistic. It also computes the p-value and formats the output for printing. We can try to make our code faster by stripping out those pieces.

```
my_t <- function(x, grp) {
  t_stat <- function(x) {
    m <- mean(x)
    n <- length(x)
    var <- sum((x - m) ^ 2) / (n - 1)

    list(m = m, n = n, var = var)
  }

  g1 <- t_stat(x[grp == 1])
  g2 <- t_stat(x[grp == 2])

  se_total <- sqrt(g1$var / g1$n + g2$var / g2$n)
  (g1$m - g2$m) / se_total
}
system.time(t2 <- apply(X, 1, my_t, grp = grp))
#>    user  system elapsed
#>   0.049   0.000   0.049
stopifnot(all.equal(t1, t2))
```

This gives us about a 6x speed improvement.

Now that we have a fairly simple function, we can make it faster still by vectorising it. Instead of looping over the array outside the function, we will modify `t_stat()` to work with a matrix of values.
Thus, `mean()` becomes `rowMeans()`, `length()` becomes`ncol()`, and `sum()` becomes `rowSums()`. The rest of the code stays the same.

```
rowtstat <- function(X, grp){
  t_stat <- function(X) {
    m <- rowMeans(X)
    n <- ncol(X)
    var <- rowSums((X - m) ^ 2) / (n - 1)

    list(m = m, n = n, var = var)
  }

  g1 <- t_stat(X[, grp == 1])
  g2 <- t_stat(X[, grp == 2])

  se_total <- sqrt(g1$var / g1$n + g2$var / g2$n)
  (g1$m - g2$m) / se_total
}
system.time(t3 <- rowtstat(X, grp))
#>    user  system elapsed
#>   0.003   0.000   0.003
stopifnot(all.equal(t1, t3))
```

That's much faster! It's at least 40x faster than our previous effort, and around 1000x faster than where we started.

Finally, we could try byte code compilation. Here we'll need to use `microbenchmark()`instead of `system.time()` in order to get enough accuracy to see a difference:

```
rowtstat_bc <- compiler::cmpfun(rowtstat)

microbenchmark(
  rowtstat(X, grp),
  rowtstat_bc(X, grp),
  unit = "ms"
)
#> Unit: milliseconds
#>                 expr  min   lq mean median   uq  max neval
#>     rowtstat(X, grp) 2.58 2.62 2.89   3.01 3.04 3.83   100
#>  rowtstat_bc(X, grp) 2.56 2.60 2.90   2.99 3.05 4.90   100
```

In this example, byte code compilation doesn't help at all.