Performance and Profiling

Stat 33B: Intro to Advanced Programming in R

Gaston Sanchez

Notes 13

About

In this lecture we will talk about performance in R, describing a simple example from different approaches, measuring their performance, and profiling them to identify bottlenecks in the code. Overall, we touch on:

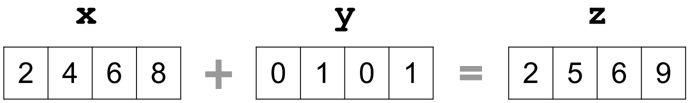
- Copy-on-modify policy
- Performance
- Profiling

```r
# suggested packages
library(lobstr)
library(microbenchmark)
library(profvis)
```

Motivation

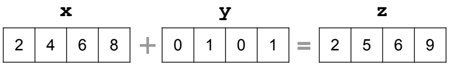
Toy Example

Consider this example:



Let's approach this with various coding options

Toy Example



A) For Loop 1

```
x = c(2,4,6,8)
y = c(0,1,0,1)
z = c()

for (i in 1:length(x)) {
  z = c(z, x[i] + y[i])
}
```

B) For Loop 2

```
x = c(2,4,6,8)
y = c(0,1,0,1)
z = rep(0, length(x))

for (i in 1:length(x)) {
  z[i] = x[i] + y[i]
}
```

C) Vectorized

```
x = c(2,4,6,8)
y = c(0,1,0,1)
z = x + y
```

What does R do in each case?

Measuring Performance with `system.time()`

Measuring Performance: case A)

```r
# a) For Loop 1
x <- runif(10000)
y <- runif(10000)
z <- c()

case_a <- system.time({
  for (i in seq_along(x)) {
    z <- c(z, x[i] + y[i])
  }
})
case_a
```

```
##    user  system elapsed
##   0.180   0.128   0.314
```

Measuring Performance: case B)

```r
# b) For Loop 2
x <- runif(10000)
y <- runif(10000)
z <- rep(0, 10000)

case_b <- system.time({
  for (i in seq_along(x)) {
    z[i] <- x[i] + y[i]
  }
})
case_b
```

```
##    user  system elapsed
##   0.004   0.000   0.004
```

Measuring Performance: case C)

```r
# c) Vectorized code
x <- runif(10000)
y <- runif(10000)

case_c <- system.time({
  z <- x + y
})
case_c
```

```
##    user  system elapsed
##       0       0       0
```

Comparison

```r
rbind(case_a, case_b, case_c)
```

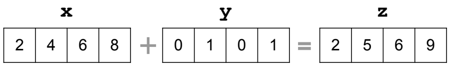
```
       user.self sys.self elapsed user.child sys.child
case_a     0.180    0.128   0.314          0         0
case_b     0.004    0.000   0.004          0         0
case_c     0.000    0.000   0.000          0         0
```

- As you can tell, vectorized code (C) beats both `for()` loops
- Moreover, case (A) is much slower than case (B)
- What's going on in each case?

Performance comparison

- Conceptually, all three options (A, B, C) are equivalent.
- In R, however, they all have different performances.
- To see why this is, we'll use functions from packages:
  - `"lobstr"`
  - `"microbenchmark"`

Motivation Example



A) For Loop 1

```
x = c(2,4,6,8)
y = c(0,1,0,1)
z = c()

for (i in 1:length(x)) {
  z = c(z, x[i] + y[i])
}
```

B) For Loop 2

```
x = c(2,4,6,8)
y = c(0,1,0,1)
z = rep(0, length(x))

for (i in 1:length(x)) {
  z[i] = x[i] + y[i]
}
```

C) Vectorized

```
x = c(2,4,6,8)
y = c(0,1,0,1)
z = x + y
```

Discussion example (A)

```r
x = c(2, 4, 6, 8)
y = c(0, 1, 0, 1)
z = c()

for (i in 1:length(x)) {
  z = c(z, x[i] + y[i])
}
```

- z has to grow at every iteration
- It seems that there are only two functions in the loop: `+` and `c()`
- In fact, `for()` is a function, as well as `":"`, `length()`, `z[i]=`, `x[i]`, `y[i]`, `"+"`

Discussion example (B)

```
x = c(2, 4, 6, 8)
y = c(0, 1, 0, 1)
z = rep(0, 4)

for (i in 1:length(x)) {
  z[i] = x[i] + y[i]
}
```

- here, `z` does not have to grow at every iteration
- There are still several function calls: `for()`, `":"`, `length()`, `x[i]`, `y[i]`, `"+"`, `"="`

Discussion example (C)

```r
x = c(2, 4, 6, 8)
y = c(0, 1, 0, 1)
z = x + y
```

- In this case, to obtain z there are only two function calls: `"+"` and `"="`
- Both are **primitive** functions which means they are written in C

Profiling

Profiling

- To understand code performance we use a **profiler**
- R uses a fairly simple type called a *sampling* or *statistical profiler*
- A sampling profiler stops the execution of code every few milliseconds and records the call stack
- The default profiling resolution is quite small, so if your function takes even a few seconds it will generate hundreds of samples
- We'll use the `"profvis"` package to visualise aggregates

Profiling with Rprof(): example (A)

```
x = runif(100000)
y = runif(100000)
z = c()

Rprof()
invisible(
  for (i in 1:length(x)) {
    z = c(z, x[i] + y[i])
  }
)
Rprof(NULL)
```

Note: I've increased the size of x and y for profiling purposes.

Profiling with Rprof(): example (A)

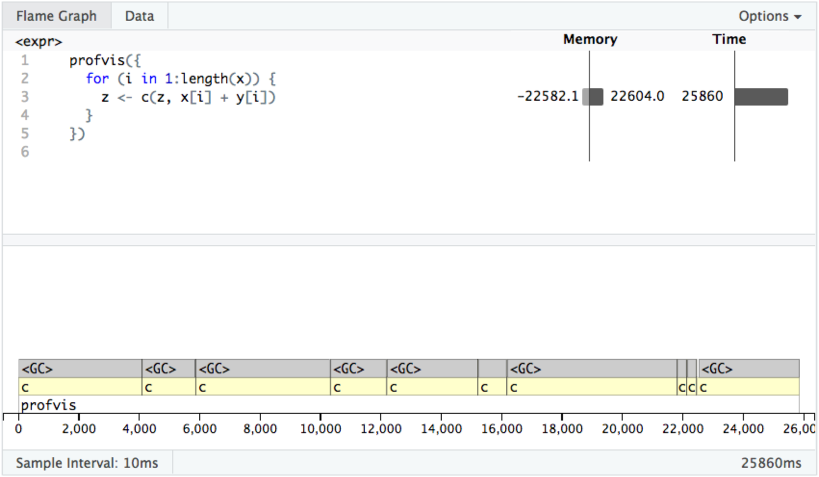
```
summaryRprof()
```

```
$by.self
    self.time self.pct total.time total.pct
"c"     25.76      100      25.76       100

$by.total
    total.time total.pct self.time self.pct
"c"      25.76       100     25.76      100

$sample.interval
[1] 0.02

$sampling.time
[1] 25.76
```

Profiling with `profvis()`: Example A

```r
x = runif(100000)
y = runif(100000)
z = c()

profvis({
  for (i in 1:length(x)) {
    z <- c(z, x[i] + y[i])
  }
})
```

Profiling with `profvis()`: Example A (cont'd)



```
profvis({
  for (i in 1:length(x)) {
    z <- c(z, x[i] + y[i])
  }
})
```

```
-22582.1  22604.0   25860
```

Sample Interval: 10ms

25860ms

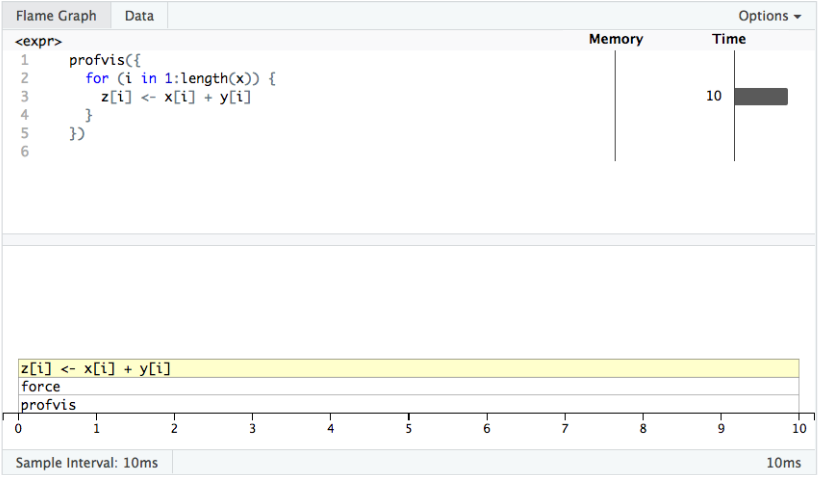
Profiling with `profvis()`: Example A (cont'd)

- The top pane shows the source code, overlaid with bar graphs for memory and execution time for each line of code.
- This display gives you a good overall feel for the bottlenecks. You can see that the whole process takes about 25.86 seconds.
- The bottom pane displays a flame graph showing the full call stack. This allows you to see that `c()` is constantly called.
- `GC` indicates that the garbage collector is running. Here, `GC` is taking a lot of time, which is usually an indication that we're creating many short-lived objects.

Profiling with profvis(): Example B

```
x = runif(100000)
y = runif(100000)
z = rep(0, 100000)

profvis({
  for (i in 1:length(x)) {
    z[i] <- x[i] + y[i]
  }
})
```

Profiling with `profvis()`: Example B (cont'd)



```r
profvis({
  for (i in 1:length(x)) {
    z[i] <- x[i] + y[i]
  }
})
```

```
z[i] <- x[i] + y[i]
force
profvis
```

As you can tell, example (B) "only" takes about 10 milliseconds

Profiling with `profvis()`: Example C

```
x = runif(100000)
y = runif(100000)

profvis({
  z <- x + y
})
```

This will fail because the code runs too fast for the profiler. And also because the profiling tool in R does not extend to C code.

Microbenchmarks

Microbenchmarking

- A microbenchmark is a measurement of the performance of a very small piece of code.
- We are going to use microbenchmarks (from the package `"microbenchmark"`) to illustrate the performance of various pieces of R code.
- Keep in mind that we will review simple examples so that you gain intuition of how R works.
- In general, don't change the way you code because of these microbenchmarks.

Microbenchmarking

Here's an example comparing two ways of calculating square root: `sqrt(x)` against `x^0.5`

```r
library(microbenchmark)

x <- runif(100)
microbenchmark(
  sqrt(x),
  x^0.5
)
```

```
## Unit: nanoseconds
##     expr  min     lq    mean median   uq   max neval
##  sqrt(x)  315  336.0  634.71    370  425 19876   100
##    x^0.5 2233 2260.5 2940.36   2314 2382 18153   100
```

Microbenchmarking

- By default, `microbenchmark()` runs each expression 100 times
- It randomizes the order of the expressions
- It summarizes the results with a minimum (`min`), lower quartile (`lq`), median, upper quartile (`uq`), and maximum (`max`)
- Focus on the median, and the quartiles to get a feel for the variability

About Performance in R

Performance of R

- R is not a particularly fast language.
- This is not an accident but mainly a consequence of R being inspired by S.
- R (like S) has been designed to make data analysis and statistics easier for you to do, not to make life easier for your computer.
- By design, R privileges flixibility and convenience (for you) over performance (for your computer).
- For most applications and purposes, R is fast enough.
- After all, for really high-performance code, you must always use languages that are closer to the computer (e.g. C, C++, Fortran).

Why is R slow?

Some of the main reasons why R can be slow:

- Because of poorly written code.
- Because R is an extremely dynamic programming language.
- Because of its lexical scoping rules—together with extreme dynamism.
- Because of lazy evaluation overhead.

Some of the reasons why R is not particularly fast

No scalars in R

- There are no scalars in R
- This is intentional: because S didn't have scalars too

This has implications because something apparently simple as

```
2 + 3
```

involves adding two vectors instead of adding two scalars.

- For most applications, though, I don't think this is a serious concern.

Extreme Dynamism

- R is an extremely dynamic programming language
- This means that you can change almost any object.
- This is the reason why you can write code like this:

```r
a = TRUE    # start with a logical vector

a[2] = 3L  # add a second element, and change 'a' into integer

a[1] = -2  # modify 1st elem, and change 'a' into double

a = as.character(a)  # coerce it into character

# make it into a function, or 'closure'
a = function(a) {
  a*a
}
```

Extreme Dynamism

Pros:

- you need minimal upfront planning
- gives you plenty of flexibility
- you can change your mind at any time, iterating your way to a solution without having to start afresh

Cons:

- too much flexibility has a cost for the language interpreter
- it's difficult to predict exactly what will happen with a given function call
- R has to constantly look for the right names and their methods

Name lookup with mutable environments

In this example, each time we print a it comes from a different environment:

```r
a = 1

f = function() {
  g = function() {
    print(a)
    assign("a", 2, envir = parent.frame())
    print(a)
    a = 3
    print(a)
  }
  g()
}
f()
```

Name lookup with mutable environments

- It’s surprisingly difficult to find the value associated with a name in R
- This is due to combination of lexical scoping and extreme dynamism
- This means that R can’t do name lookup just once, it has to start from scratch each time
- This issue is accentuated by the fact that almost every operation is a lexically scoped function call

Name lookup with mutable environments

```r
make_taco = function(tortilla, carne) {
  (tortilla + carne)^2
}
```

- It seems that `make_taco()` calls two functions: + and ^
- In fact, it calls four because { and (are regular functions in R

Lazy evaluation overhead

- Lazy evaluation means that function arguments are evaluated when needed. In other words, if an argument is not used, then it won't be evaluated.
- To implement lazy evaluation, R uses a **promise** object that contains the expression needed to compute the result and the environment in which to perform the computation.
- Creating promise objects has some overhead, so each additional argument to a function decreases its speed a little.

Lazy evaluation overhead

Consider this example in which each version of the function has one additional argument:

```r
g0 = function() NULL

g1 = function(a = 1) NULL

g2 = function(a = 1, b = 2) NULL

g3 = function(a = 1, b = 2, c = 3) NULL

g4 = function(a = 1, b = 2, c = 3, d = 4) NULL

g5 = function(a = 1, b = 2, c = 3, d = 4, e = 5) NULL
```

Lazy evaluation overhead

```r
microbenchmark(g0(), g1(), g2(), g3(), g4(), g5(), times = 10000)
```

```
## Unit: nanoseconds
##   expr min  lq     mean median  uq    max neval
##   g0() 108 118 254.5652    124 133 929729 10000
##   g1() 155 170 282.2761    178 191 444477 10000
##   g2() 195 215 349.4013    226 244 452049 10000
##   g3() 237 261 425.8221    274 302 407385 10000
##   g4() 276 300 473.1978    315 360 408547 10000
##   g5() 313 344 573.9759    362 425 433378 10000
```

Various Examples

Example

```r
# median-centering a data frame
dat = data.frame(matrix(runif(100 * 1000), ncol = 100))
medians = vapply(dat, median, numeric(1))

for (j in seq_along(medians)) {
  dat[ ,j] = dat[ ,j] - medians[j]
}
```

- every iteration of the loop copies the data frame
- this is because `[<-data.frame` is not a primitive function
- so every replacement produces an entire copy

Example

```r
# median-centering a data frame
dat = data.frame(matrix(runif(100 * 1000), ncol = 100))
medians = vapply(dat, median, numeric(1))
lis = as.list(dat)

for (j in seq_along(medians)) {
  lis[[j]] = lis[[j]] - medians[j]
}
```

- using a list instead of data frame improves performance
- having a list, we can use `[[<-list` which is a primitive function
- here, every replacement occurs in place (without any copies)

Some Comments

My Advice

- **Good code is clear**: it's better to be clear than clever
- **Time and practice**: it takes time and practice to be able to write code in R that performs well, so don't obsess too much about it (if you don't need to)
- **Is it worth your time?** Don't spend hours of your time to save seconds of computer time when you can get the job done without worrying about performance.
- **Know when to quit**: At some point, if you are seriously concerned about the speed of your code, you will reach a limit within R, and will have to switch to "faster" languages (e.g. C, C++, Fortran)

My Advice

- Start small (baby steps), writing code that gets the job done
- Gradually work your way up, developing and testing your code under different circumstances
- The more you understand R, you'll tend to have more opportunity to improve your code
- The more time you have to think about your code (special cases, make it more robust, more user friendly), the higher the chances to make it faster
- What's the ROI? Sometimes you don't have time to polish your code, so even if you know your code is not optimized, this won't matter because you need to move to next stage(s) of your project(s)