Debugging and Profiling

Dom Owens

05/11/2019

Writing code inevitably leads to bugs and performance issues. Here, we address these through **debugging** and **profiling**.

Debugging

Bugs are flaws that lead to code not running as intended. Some can be identified by inspection; others require a more involved approach. We consider some general rules for operation:

- Understand what the error message is really telling you, and what the intended result is
- Obtain a reproducible, minimal, working example
- Identify the actual problem. Be systematic, test hypotheses
- After fixing the bug, add tests to ensure this does not reintroduce itself

Minimal working examples can be submitted to i.e. a package owner for bug fixing, or to StackExchange for advice. Incrementally writing small chunks of code, as opposed to a whole function, can be a good preventative measure.

Consider the following example, which raises the first argument to the power of the second:

```
to_power <- function(x,a){x^a}
100 == to_power(10,2) #this works as intended
```

```
## [1] TRUE
```

We wrap this inside another function, which sets the argument `x` to `5`

```
five_to_power <- function(a){return(to_power(5,a) )}
```

We specify a set of integers and letters to sample from.

```
set.seed(100)
bag <- c(1:100, letters) #1 to 100 and "a" to "z"
grab_bag <- sample(bag, 20) #select 20 from bag

for (i in 1:length(grab_bag)) {
  print(five_to_power(as.numeric(grab_bag[i]))) #raise 5 to each character from bag
}
```

```
## [1] 1.818989e+27
## [1] 1.164153e+23
## [1] 1.694066e+48
## [1] 78125
## [1] 3.469447e+40
## [1] 1.734723e+41
## [1] 3.155444e+68
## [1] 2.842171e+31
## [1] 2.710505e+45
## [1] 9.536743e+13
## [1] 1.058791e+51
```

```
## Warning in to_power(5, a): NAs introduced by coercion
```

```
## [1] NA
## [1] 2.328306e+22
## [1] 1.421085e+32
## [1] 1.29247e+60
## [1] 2.646978e+52
## [1] 1.192093e+16

## Warning in to_power(5, a): NAs introduced by coercion

## [1] NA

## Warning in to_power(5, a): NAs introduced by coercion

## [1] NA
## [1] 5.293956e+51
```

Here we have three errors (obviously, these correspond to letters sampled from the bag and were self-induced). Calling `browser()` will allow us to open a debugging envrionment, and diagnose the issue.

```r
for (i in 1:length(grab_bag)) {
  if(i==12)browser()
  print(five_to_power(as.numeric(grab_bag[i])))
}
```

```
## [1] 1.818989e+27
## [1] 1.164153e+23
## [1] 1.694066e+48
## [1] 78125
## [1] 3.469447e+40
## [1] 1.734723e+41
## [1] 3.155444e+68
## [1] 2.842171e+31
## [1] 2.710505e+45
## [1] 9.536743e+13
## [1] 1.058791e+51
## Called from: eval(expr, envir, enclos)
## debug at <text>#3: print(five_to_power(as.numeric(grab_bag[i])))

## Warning in to_power(5, a): NAs introduced by coercion

## [1] NA
## debug at <text>#2: if (i == 12) browser()
## debug at <text>#3: print(five_to_power(as.numeric(grab_bag[i])))
## [1] 2.328306e+22
## debug at <text>#2: if (i == 12) browser()
## debug at <text>#3: print(five_to_power(as.numeric(grab_bag[i])))
## [1] 1.421085e+32
## debug at <text>#2: if (i == 12) browser()
## debug at <text>#3: print(five_to_power(as.numeric(grab_bag[i])))
## [1] 1.29247e+60
## debug at <text>#2: if (i == 12) browser()
## debug at <text>#3: print(five_to_power(as.numeric(grab_bag[i])))
## [1] 2.646978e+52
## debug at <text>#2: if (i == 12) browser()
## debug at <text>#3: print(five_to_power(as.numeric(grab_bag[i])))
## [1] 1.192093e+16
## debug at <text>#2: if (i == 12) browser()
## debug at <text>#3: print(five_to_power(as.numeric(grab_bag[i])))
```

```
## Warning in to_power(5, a): NAs introduced by coercion
## [1] NA
## debug at <text>#2: if (i == 12) browser()
## debug at <text>#3: print(five_to_power(as.numeric(grab_bag[i])))
## Warning in to_power(5, a): NAs introduced by coercion
## [1] NA
## debug at <text>#2: if (i == 12) browser()
## debug at <text>#3: print(five_to_power(as.numeric(grab_bag[i])))
## [1] 5.293956e+51
```

Profiling

Profiling is a means of measuring the time and memory complexity of code. In R, we use the base function `Rprof` and the package **profvis**; these are *statistical profilers* which interrupt code to determine what is being executed - this is not too expensive to run. On the other hand, *instrumenting profilers*, which require explicit code commands to be added to the main body. These are expensive, but detect memory leaks and reference errors *(?)*.

Consider this example, in which we identify prime factors (in particular, the greatest prime factor) of $N = 600851475143$ (Project Euler problem 3):

```
# Sieve of Eratosthenes for generating primes 2:n
esieve <- function(n) {
    if (n==1) return(NULL)
    if (n==2) return(n)
    # Create a list of consecutive integers {2,3,...,N}.
    l <- 2:n
    # Start counter
    i <- 1
    # Select p as the first prime number in the list, p=2.
    p <- 2
    while (p^2<=n) {
        # Remove all multiples of p from the l.
        l <- l[l==p | l%%p!=0]
        # set p equal to the next integer in l which has not been removed.
        i <- i+1 # Repeat steps 3 and 4 until p2 > n, all the remaining numbers in the list are primes
        p <- l[i]
    }
    return(l)
}

# Prime Factors
prime.factors <- function (n) { #get prime factors of n
    factors <- c() # Define list of factors
    primes <- esieve(floor(sqrt(n))) # Define primes to be tested
    d <- which(n%%primes == 0) # Idenitfy prime divisors
    if (length(d) == 0) # No prime divisors
        return(n)
    for (q in primes[d]) { # Test candidate primes
        while (n%%q == 0) { # Generate list of factors
            factors <- c(factors, q)
            n <- n/q } }
    if (n > 1) factors <- c(factors, n)
```

```r
    return(factors)
}

max(prime.factors(600851475143)) #return greatest prime factor
```

```
## [1] 6857
```

Using `profvis` we obtain a flame graph showing time useage of the function.

```r
#install.packages("profvis")
library(profvis)
profvis(prime.factors(600851475143))
```

```
## PhantomJS not found. You can install it with webshot::install_phantomjs(). If it is installed, please
```

Here, we have evidenced which parts of the code impact performance. In larger-scale computing this will be even more useful, as we will be able to identify which functions should be rewritten.

Performance

Given R is an interpreted language, operations will be slower than they otherwise would be in a lower-level language (consider,for example, matrix operations).

Memory management

We see by example that small changes in code can sharply affect performance.

```r
a1 <- runif(5000, 0.5, 1.5) #generate samples from unif[0.5, 1.5]
a2 <- runif(10000, 0.5, 1.5)

multiply.bad <- function(x) {
  y <- 1:length(x) #integers 1 to length of x
  for (i in 1:length(x)) {
    x[i] <- x[i]*y[i] #multiply elementwise

  }
  x
}

multiply.index <- function( x, i) i*x[i] #multiply x by index

multiply.slow <- function(x) { #perform elementwise multiplication as above

  for (i in 1:length(x)) {
    x[i] <- multiply.index( x, i)
  }
  x
}

#compare run times
system.time(multiply.bad(a1))
```

```
##    user  system elapsed
##   0.006   0.000   0.006
```

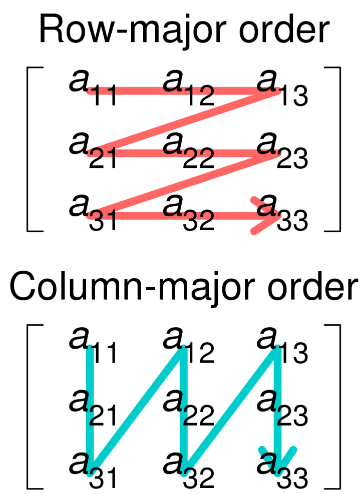

Figure 1: Row and Column Major Order

```
system.time(multiply.bad(a2))
```

```
##    user  system elapsed
##   0.001   0.000   0.002
```

```
system.time(multiply.slow(a1))
```

```
##    user  system elapsed
##   0.053   0.003   0.056
```

```
system.time(multiply.slow(a2))
```

```
##    user  system elapsed
##   0.201   0.000   0.201
```

Column-major storage

In R, matrices are stored according to column-major order, so elements in each column are stored in one block (see Figure 1). Thus, writing functions on column operations is preferable to functions with row operations.

```
fill.by.column <- function(n) { #populate matrix by column
  x <- rep(1,n)
  X <- matrix(0,n,n)
  for (i in 1:n) {
    X[,i] <- x
  }
  X
}

fill.by.row <- function(n) {#populate matrix by row
  x <- rep(1,n)
  X <- matrix(0,n,n)
  for (i in 1:n) {
    X[i,] <- x
  }
  X
```

```r
}
n <- 10000
system.time(Y1 <- fill.by.column(n)) #compare run times
```

```
##    user  system elapsed
##   0.455   0.315   0.770
```

```r
system.time(Y2 <- fill.by.row(n))
```

```
##    user  system elapsed
##   1.289   0.296   1.585
```

Example: 3-matrix multiplication

We calculate \$ D = ABC \$ where A is n by p, B is p by m, and C is m by q

```r
n <- 10000
p <- 200
m <- 500
q <- 300
#specify A,B,C as above using standard normal samples for each entry
A <- matrix(rnorm(n*p), n, p)
B <- matrix(rnorm(p*m), p, m)
C <- matrix(rnorm(m*q), m, q)


foo <- function(A, B, C) {
  AB <- matrix(0, n, m)
  for (i in 1:m) { #populate AB by column
    AB[,i] <- A%*%B[,i]
  }
  D <- matrix(0, n, q)
  for (i in 1:q) { #populate D by column
    D[,i] <- AB%*%C[,i]
  }
  D
}

#compare optimised base multiplication with our column-wise populator
system.time(D <- A %*% B %*% C)
```

```
##    user  system elapsed
##   0.689   0.052   0.097
```

```r
system.time(D_alt <- foo(A,B,C))
```

```
##    user  system elapsed
##  27.776  16.418   5.699
```

Note the performance divergence between the base matrix multiplication, and our multiplication using for loops.