Profiling and Debugging
Statistics 650/750
Week 10 Thursday

Alex Reinhart and Christopher Genovese

02 Nov 2016

Announcements

- There are several SQL assignments available under `problem-bank/databases`
  - Remember, all assignments are tagged, along with being listed under `All/`
- Suggested entry-point exercises listed under `documents/Info/entry-point-exercises`
- Project grading is proceeding quickly; final revisions due Thursday, November 9
- You get two revisions for projects; answer TA comments quickly
- When resubmitting projects (or any assignment), remember to select the TA for review again
- Instructions for connecting to SQL with R or Python are in `documents/ClassFiles/weekA/commands-3.R` and `commands-3.py`

Debugging

Debugging

We've all done a lot of print-based debugging: if the code doesn't work, stick a `print` statement in the middle to see what it's doing.

This is a blunt tool, though a very easy one to use. For tricky cases, look to an interactive debugger before sticking in a few dozen print statements. Debugging is hard without the right tools:

"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it." – Brian Kernighan

Using your tests

A good first debugging step is to make sure you have tests for the function you're debugging.

You're going to be trying all sorts of changes, tinkering with things, refactoring, and generally messing with the function – if you have tests, you can easily check if you fixed the bug without introducing any new ones. If you don't, you have to laboriously try various inputs until you're satisfied.

Tests also ensure that you *actually know what the function is supposed to do.*

Interactive debuggers

An interactive debugger halts program execution and allows you to inspect the current state: display local variables, view the call stack, set breakpoints, and even run new code. You can step through the code line-by-line to examine how it works.

Debuggers can often be configured to open automatically when your program crashes or throws an exception (like Python's `pdb`). IDEs also let you set breakpoints and run debuggers whenever you'd like, or you can add code to invoke the debugger when desired.

(RStudio example time)

Debuggers also typically have command-line interfaces that let you interactively type commands and explore the program's state. (Common Lisp and Smalltalk even let you change the code from inside the debugger, editing as you work.)

An example:

```
# Instead of
python ingest_crimes.py -s 2707.1 data/example_data.txt
# Run
python -m pdb ingest_crimes.py -s 2707.1 data/example_data.txt
```

Some unit testing tools (like Python's pytest) can automatically open a debugger when a test fails.

Some debuggers support *remote* debugging: they can debug a program running on another machine, like a phone or a server.

Resources

- pdb, the Python debugger. If you use IPython, look at the %debug magic command.
- RStudio's debugging documentation
- gdb and lldb for compiled languages (C, C++, Objective-C, whatever GCC or LLVM support)

Profiling

Knowledge-based optimization

We often worry about whether code is fast. We vectorize algorithms, rewrite functions in C++, and try all sorts of little tricks which are allegedly "faster" than the naive implementation.

But without data to guide our optimization, it will be wasted effort. As Knuth says,

"Programmers waste enormous amounts of time thinking about, or worrying about, the speed of non-critical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We *should* forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil.** Yet we should not pass up our opportunities in that critical 3%."

Profiling is how we find that critical 3%.

Profiling happens *after* we've identified the best data structures and algorithms for our task. Profiling won't help an algorithm that's accidentally quadratic.

Performance profiling

There are two common kinds of profiling:

Deterministic profiling Every function call and function return is monitored, and the time spent inside each function recorded

Statistical profiling Execution is interrupted periodically and the function currently being executed recorded.

Deterministic profiling gives the most accurate data, but adds substantial overhead: every function call requires data to be stored. This overhead may slow down code or distort the profile results. Statistical profiling may miss frequently-called fast-running functions and gives less comprehensive results.

Python's built-in cProfile module uses deterministic profiling. It can be invoked on the the command line:

```
1 python -m cProfile -s time analyze_my_data.py
```

It can also be run inside code by importing cProfile and calling `cProfile.run`, or interactively inside IPython with the `%prun` magic. Example output:

```
         1330241 function calls in 121.351 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    30597   56.359    0.002   56.359    0.002 {crime.emtools.intensity}
       91   37.850    0.416   37.859    0.416 {crime.emtools.e_step}
       93   26.350    0.283   83.456    0.897 em.py:207(log_likelihood)
  1265544    0.510    0.000    0.510    0.000 {built-in method exp}
    30597    0.229    0.000   56.588    0.002 em.py:181(intensity)
        1    0.025    0.025  120.487  120.487 em.py:118(fit)
      278    0.007    0.000    0.007    0.000 {method 'reduce' of 'numpy.ufunc' objects}
      366    0.005    0.000    0.005    0.000 {built-in method empty_like}
      273    0.002    0.000    0.009    0.000 numeric.py:81(zeros_like)
      275    0.002    0.000    0.002    0.000 {built-in method copyto}
      183    0.001    0.000    0.008    0.000 fromnumeric.py:1852(all)
      273    0.001    0.000    0.001    0.000 {built-in method zeros}
       94    0.001    0.000    0.006    0.000 fromnumeric.py:1631(sum)
       ...
```

R's built-in `Rprof` uses statistical profiling. It writes profiling data to a file, which can then be analyzed with `summaryRprof`.

(Example interlude)

In a bigger program, Rprof can produce output like this:

```
> summaryRprof()
$by.self
                     self.time self.pct total.time total.pct
"specgram"              320.70    41.05     563.00     72.07
"seq.default"            48.40     6.20     114.02     14.60
"getPeaks"               41.02     5.25     780.98     99.97
":"                      34.38     4.40      34.38      4.40
"is.data.frame"          28.60     3.66      47.54      6.09
```

```
"pmin"                  25.70     3.29      58.62      7.50
"colSums"               25.52     3.27      78.02      9.99
"seq"                   20.52     2.63     135.28     17.32
"matrix"                20.04     2.57      21.58      2.76
"as.matrix"             19.40     2.48      50.44      6.46
...
```

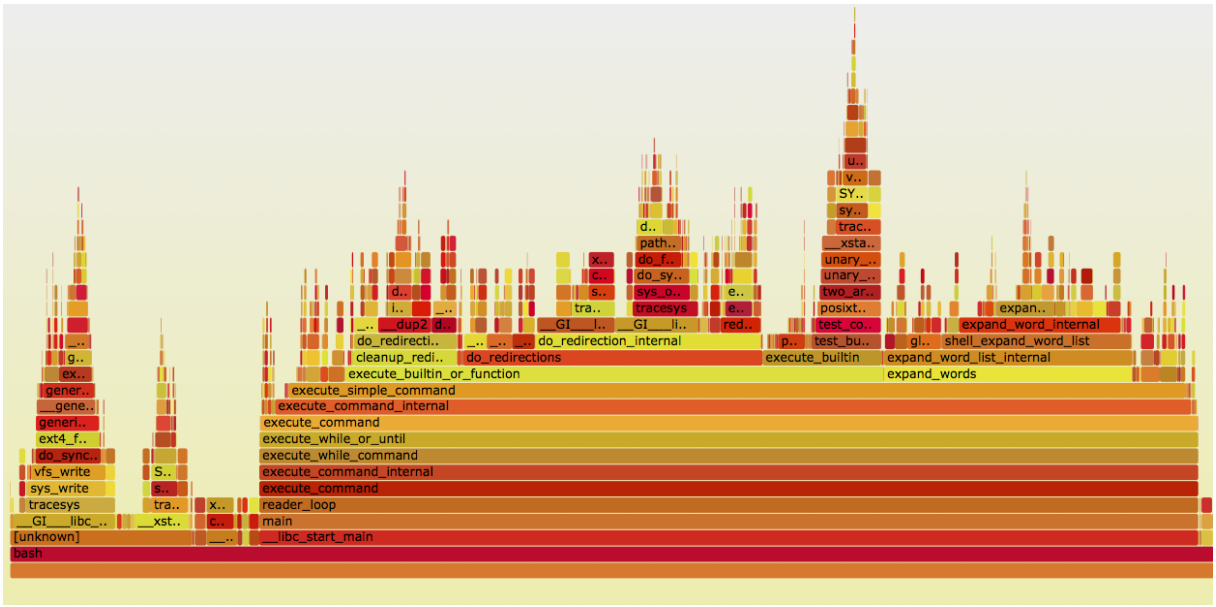
Notice the presence of ":". Big sequences are costly to construct – better to use iterators when possible. R's lazy evaluation can make profiling trickier to interpret.

Call graphs

Sophisticated profilers can produce *call graphs*: a graph representing which functions call which other functions, and hence breaking down which callers contribute most to the use of a function. This can be useful if you spot a slow function but aren't sure which function is responsible for calling it the most.

The prof.tree package for R can produce handy call graphs from Rprof output files.

Call graphs can be visualized as *flame graphs*, which show which parts of your code are hot. Tools like DTrace and SystemTap allow you to instrument code to extract this information:



See `http://www.brendangregg.com/flamegraphs.html` for details on how to produce such graphs for compiled code. Julia's ProfileView.jl can build these automatically for Julia code, and the profvis package can produce an interactive web page for R profiles. (See the example here.) RStudio integrates profvis to make it easy to profile R code with a single button.

Other profiling tools are Callgrind and gperftools.

Line profiling

Some profiling tools can measure individual lines instead of whole function calls. This can be useful if your code isn't cleanly split into many small functions. (But it should be!)

Line profiling isn't as commonly used as function profiling; look at the profvis package for an R implementation, available in RStudio.

Always measure your changes!

It is too easy to do optimization voodoo: tweak one line, then another, then another, without knowing what is working and what is not.

Use your profiler to test if the optimizations are worthwhile. (Most optimizations add a complexity cost to your code.)

Remember that program execution time is a random variable with noise: you need more than one run to tell if a change mattered. Remember that a profiler adds overhead: code is slower when it is being profiled.

Many languages provide microbenchmarking packages to do this for you. For example, in Python:

```
>>> import timeit
>>> timeit.timeit('"-".join(str(n) for n in range(100))', number=10000)
0.8187260627746582
>>> timeit.timeit('"-".join([str(n) for n in range(100)])', number=10000)
0.7288308143615723
>>> timeit.timeit('"-".join(map(str, range(100)))', number=10000)
0.5858950614929199
```

R has the microbenchmark package to do the same thing.

Memory profiling

Memory allocation has a cost. In my research, I had to fit a large mixture model, which required two ~10000x1000 (80 MB) mixing matrices *on every iteration*. As the data size grew, Python eventually ran out of memory on my 1GB server account, despite the matrices technically fitting in RAM.

Why is allocation a problem?

Garbage collection

R, Python, Julia, Ruby, Java, JavaScript, and all other dynamic languages are *garbage collected* (GCed): at runtime, the interpreter must determine which variables are accessible (live) and which are not (garbage) and free memory accordingly.

In C and some other compiled languages, memory management is manual. There is a distinction between the *stack* and the *heap*:

Stack When a function is called, its arguments are pushed onto the stack, as well as any local variables it declares, in a stack *frame*. When the function returns, its frame is popped off the stack, destroying the local variables. But the stack frame has a fixed size, and can only contain variables whose sizes are known in advance.

Heap A global pool of explicitly-allocated memory. Can contain arbitrary objects shared between functions, but requires explicit management to allocate and deallocate space.

```c
// I am not a C programmer. Forgive me.

double fit_big_model(double *data, int p, double tuning_param, ...) {
  double *betas = malloc(p * sizeof(double));

  // do stuff

```

```
8   free(betas);
9 }
```

More advanced languages like C++, Rust, D and so on add additional ways to manage memory, like RAII, which make it less cumbersome and less error-prone.

In dynamic languages, any object can be any size, and may live arbitrarily long. A typical strategy is *tracing*: the language keeps track of reachable objects, those referenced by local variables or global variables. It then traces out a graph: any object contained inside a local variable (e.g. inside a list in R) or accessible from one.

This produces the set of "live" objects. Any other objects are garbage and can be deallocated, since they are no longer accessible.

(This is essentially a graph traversal problem, and so there are many variations with different performance characteristics in different use cases.)

This traversal takes time. Most garbage collectors "stop the world": execution stops while they collect. If there's lots of garbage or lots of allocation, GC can be slow.

Tracking memory use

Some languages provide simple blunt instruments to see how much memory is allocated by code:

```
julia> @time f(10^6)
elapsed time: 0.04123202 seconds (32002136 bytes allocated)
2.5000025e11
```

Memory profiling (sometimes called "heap profiling") is not as common as ordinary profiling, but can still be very useful. There are a number of tools for different languages. The granddaddy might be Massif, part of the Valgrind suite of tools for analyzing compiled code.

Massif takes snapshots of the heap as your program executes, then provides detailed analysis showing which functions allocated memory and in what amounts. It uses a detailed call graph to see how they were used. Here's an example from the Massif documentation:

```
--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
 10         10,080           10,080           10,000            80            0
 11         12,088           12,088           12,000            88            0
 12         16,096           16,096           16,000            96            0
 13         20,104           20,104           20,000           104            0
 14         20,104           20,104           20,000           104            0
99.48% (20,000B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->49.74% (10,000B) 0x804841A: main (example.c:20)
|
->39.79% (8,000B) 0x80483C2: g (example.c:5)
| ->19.90% (4,000B) 0x80483E2: f (example.c:11)
| | ->19.90% (4,000B) 0x8048431: main (example.c:23)
| |
| ->19.90% (4,000B) 0x8048436: main (example.c:25)
|
->09.95% (2,000B) 0x80483DA: f (example.c:10)
  ->09.95% (2,000B) 0x8048431: main (example.c:23)
```

Python's memory_profiler module offers line-by-line memory profiling features. A simple example from its documentation:

```
@profile
def my_func():
    a = [1] * (10 ** 6)
    b = [2] * (2 * 10 ** 7)
    del b
    return a

if __name__ == '__main__':
    my_func()
```

```
Line #    Mem usage  Increment   Line Contents
================================================
     3                           @profile
     4      5.97 MB    0.00 MB   def my_func():
     5     13.61 MB    7.64 MB       a = [1] * (10 ** 6)
     6    166.20 MB  152.59 MB       b = [2] * (2 * 10 ** 7)
     7     13.61 MB -152.59 MB       del b
     8     13.61 MB    0.00 MB       return a
```

These results are not wholly reliable, since they rely on the OS to report memory usage instead of tracking specific allocations, and garbage collection can occur unpredictably.

R's `Rprof` has memory profiling, but the output is poorly documented and difficult to understand.

Reducing allocations

Common memory pitfalls include unnecessary copying:

```
foo <- function(x) {
    x$weights <- calculate_weights(x)

    s <- sample_by_weight(x)

    ...
}
```

Because we've written to `x`, `x` is copied. (This is peculiar to R's copy-on-write scheme.) Another example:

```
for (x in data) {
    results <- c(results, calculate_stuff(x))
}
```

The same happens with repeated `rbind` or `cbind` calls. Allocate `results` in advance, or use `Map` or `vapply` instead:

```r
results <- numeric(nrow(data))

for (i in seq_along(data)) {
    results[i] <- calculate_stuff(data[i])
}
```

Intermediate results also require allocations:

```r
pois.grad <- function(y, X) {
    function(beta) {
        t(X) %*% (exp(X %*% beta) - y)
    }
}
```

In Numpy, we can specify the output array to avoid these kinds of problems, with extra tedium:

```python
def pois_grad(y, X):
    def grad(beta):
        tmp = np.empty((X.shape[0], 1))
        np.dot(X, beta, out=tmp)
        np.exp(tmp, out=tmp)
        np.subtract(tmp, y, out=tmp)

        return X.T * tmp

    return grad
```

I do not recommend writing code this ugly unless absolutely necessary. Numba is a smart optimizing compiler for Python code which can perform these kinds of optimizations automatically.

Resources

- Python's profiling documentation
- RStudio's profiling documentation
- Rprof documentation
- Advanced R's Profiling chapter
- RStudio's profvis package for interactive profiling
- gprof
- Valgrind for a variety of memory and profiling tools for compiled code (typically C and C++)

General performance tips

- The Cython optimizing compiler for Python
- Rcpp for R and C++ integration
- The R Inferno: "If you are using R and you think you're in hell, this is a map for you."
- Why Python is Slow, a blog post on the complicated abstractions buried inside Python that make it slow. Applies in part to other dynamic languages like R.