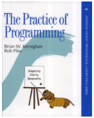
Run Time & Performance

Kurt Schmidt

Dept. of Computer Science, Drexel University

June 5, 2017

Examples are taken from Kernighan & Pike, *The Practice of Programming*, Addison-Wesley, 1999



Intro

Performance

Objective: To learn when and how to optimize the performance of a program.

> *The first principle of optimisation is don't.*

- Knowing how a program will be used, and the environment it runs in, is there any benefit to making it faster?
- Which areas of the program should we focus on?

Strategy

- Use the simplest, cleanest algorithms and data structures appropriate for the task
- Enable compiler options to generate the fastes possible code
  - Modern compilers optimise by default
  - Modern compilers are very good at their jobs
- Then, measure performance to see if changes are needed

Strategy (cont.)

- Assess what changes to the program will have the most effect
  - Use a profiler to find hotspots in your code
- Make changes incrementally, re-assess
  - Consider alternative algorithms
  - Tune the code
  - Consider a lower-level language
    - Maybe just for time-sensitive components
  - Always retest your code!

Profiling Code

Profiler – gprof

- A profiler watches your program run
- Reports back a bunch of information, including
  - How much time was spent in each function
  - How many times each function was called
- Use this information to find bottlenecks (areas worth improvement) in your code
- Profilers exist for most common languages, including Java, Python, and Haskell
- We will use `gprof`, which can be run with programs compiled with a gnu compiler

Using gprof

- Use the -p option to compile extra information into your program:

```
$ gcc -p driver.c quicksort.c -o mySort
```

- Now run the program once, to generate metrics:

```
$ ./mySort < ins.10000 > /dev/null
```

- Note, a new data file has appeared:[1]

```
$ ls -ot | head -n3
total 3864
-rw-r--r-- 1 kschmidt  747 Aug 4 16:05 gmon.out
-rwxrwxr-x 1 kschmidt 9102 Aug 4 16:05 mySort
```

[1] It's raw data. Don't look at it yet

Reading the gprof Report

- Supply gprof w/the name of the executable to see report on `stdout`:

```
$ gprof mySort
```

- Report contains 2 tables of data, with description of the information:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 61.41      0.25     0.25      500   503.60   805.76  quicksort
 36.85      0.40     0.15 45721062     0.00     0.00  swap
  1.23      0.41     0.01                             main
...
```

Reading the gprof Report (cont.)

```
...
granularity: each sample hit covers 2 byte(s) for 2.45% of 0.41 sec

index % time   self  children   called     name
                                             <spontaneous>
[1]    100.0   0.01    0.40                 main [1]
               0.25    0.15    500/500          quicksort [2]
-----------------------------------------------
                            6666956             quicksort [2]
               0.25    0.15    500/500          main [1]
[2]     98.8   0.25    0.15    500+6666956 quicksort [2]
               0.15    0.00 45721062/45721062   swap [3]
                            6666956             quicksort [2]
-----------------------------------------------
               0.15    0.00 45721062/45721062   quicksort [2]
[3]     37.0   0.15    0.00 45721062        swap [3]
-----------------------------------------------
...
```

Run-time Analysis

Asymptotic Run Time

- We would like to be able to compare algorithms, for arbitrarily large inputs
- We want to evaluate the growth of algorithms vs. input
  - We don't care about, e.g., processor speed, the leading coefficient, nor lower-order terms
  - E.g., linear search. If the array size doubles, so does the run-time $\implies \Theta(n)$
  - Selection sort is a quadratic sort, $\Theta(n^2) \implies$ doubling input size will quadruple run time (for large n)
  - Accessing an element of an array is a constant-time operation, $\Theta(1)$, regardless of size of array

Lower Bound (Loose) – Little Omega

Consider function $T_n = 5n^2 + 17n - 12$

$$\lim_{n \to \infty} \frac{5n^2 + 17n - 12}{n} = \infty \implies T_n \in \omega(n)$$

- We say T_n is *bound below* (loosely) by n
- We say T_n grows *strictly faster* (asymptotically) than n
- T_n can not be bound above by n

More generally:

$$\lim_{n \to \infty} \frac{T_n}{f_n} = \infty \implies T_n \in \omega(f_n)$$

Upper Bound (Loose)– Little Oh

$$\lim_{n\to\infty} \frac{5n^2 + 17n - 12}{n^3} = 0 \implies T_n \in \mathsf{o}(n^3)$$

- We say T_n is *bound above* (loosely) by n^3
- We say T_n grows *strictly more slowly* (asymptotically) than n^3
- T_n can not be bound below by n^3

More generally:

$$\lim_{n\to\infty} \frac{T_n}{f_n} = 0 \implies T_n \in \mathsf{o}(f_n)$$

Asymptotically Equivalent – Theta

$$\lim_{n\to\infty} \frac{5n^2 + 17n - 12}{n^2} = 5 \implies T_n \in \Theta(n^2)$$

- We say T_n grows like n^2 (asymptotically)
- T_n can bound below, and above, by n^2

More generally:

$$\lim_{n\to\infty} \frac{T_n}{f_n} = c, c \in \mathbb{R}^+ \implies T_n \in \Theta(f_n)$$

Note: This does not mean that T_n is polynomic

Notes on Asymptotic Equivalence

- Two equivalent functions needn't look the same
- Just grow similarly
- Consider $y = x + \sin x$
  - y grows like a line
    - Bound above by $y = x + 1$
    - Bound below by $y = x - 1$
  - Clearly not a line

Quick Observations

$$f \in \mathsf{o}(g) \iff g \in \omega(f)$$

$$T \in \Theta(f) \implies T \notin \mathsf{o}(f)$$

$$T \in \Theta(f) \implies T \notin \omega(f)$$

$$T \in \omega(f) \implies T \notin \Theta(f)$$

$$T \in \mathsf{o}(f) \implies T \notin \Theta(f)$$

Big-Oh, -Omega

Upper (lower) bound, may or may not be tight

$$\mathrm{O}(f) = \mathsf{o}(f) \cup \Theta(f)$$

$$\Omega(f) = \omega(f) \cup \Theta(f)$$

We have these observations:

$$\mathsf{o}(f) \subset \mathrm{O}(f)$$

$$\omega(f) \subset \Omega(f)$$

Finally,

$$T \in \Theta(f) \iff T \in \mathrm{O}(f) \wedge T \in \Omega(f)$$

Qualitative Statements

T is $\mathrm{O}(f)$

- "T grows no faster than f"
- "T is bound above by f"
- "f is an upper bound for T"

T is $\Omega(f)$

- "T grows no slower than f"
- "T is bound below by f"
- "f is a lower bound for T"

Ranking of Common Functions

For reference, here are some common functions, in increasing order:

1 (constant)
$\log n$
$\sqrt{n}$
n
$n \log n$
$n\sqrt{n}$
n^2
$\vdots$
n^p
c^n
$n^n \approx n!$

Note, $\log n \in \mathrm{o}(n^p), p \in \mathbb{R}, \forall p > 0$

Timing Programs

Timing – `time`

- If we can't evaluate the algorithm, we can run the program with various inputs, time each run
- Various languages may provide their own mechanism for timing from within the program
- The `time` utilitiy takes a program, with arguments, to run
  - You now know why you type `date` to see what the time is

Using the `time` Utility

```
time [options] cmd [cmd_args]
```

- **options** Options to modify behavior of `time`. Must precede cmd
- **cmd** The program run you want to time
- **cmd_args** Arguments, including options, to be passed to cmd

Note: This is a built-in in Bash, Tenex C-Shell, and others.

`time` example

From our previous example:

```
$ time ./mySort < ins.10000 > /dev/null
```

- Output to screen is expensive
- We're not interested in that time

Output from `time` (to `stderr`):

```
real  0m7.572s
user  0m7.555s
sys   0m0.004s
```

Description of Output

real
: The wall clock. Total time elapsed. Keeps ticking, even if your program is sliced out

user
: The actual time your program spent running, in user mode

sys
: The actual time your program spent running, in kernel mode

- The sum of the user and sys times is probably what you want

Using Bash's `time` in a Script

- The `-p` option be handier for parsing:

```
$ /usr/bin/time ./mySort < ins.10000 &> /dev/null
```

```
real 7.572
user 7.555
sys 0.004
```

- Storing it in a variable takes some contortions:

```
$ result=$({ time -p ./$prog <ins.10000 >/dev/null ; } 2>&1 )
$ echo $result
```

```
real 7.572 user 7.555 sys 0.004
```

`time` Built-in v. Utility

- There is a utility (not a shell built-in)
  - On tux, installed as `/usr/bin/time`
  - Can also report on other metrics
  - Output values and format can be customised (see `-f`)
- Bash, Tenex C Shell, and others have a built-in `time` command, which doesn't allow for customising the output format
  - The built-ins are a bit slicker, parsing up the command line
  - For e.g., the following produces no output (why?), but works fine w/the shell built-in

```
$ /usr/bin/time ./mySort < ins.10000 &> /dev/null
```

Alternatives to Timing Program

- Computers have gotten fast
  - Makes it a little harder to grab good numbers
- Alternatively, we could be creative, use metrics from a profiler
  - E.g., we could count the number of calls to `swap` for various sized inputs to our sort
  - Or, we might use a function to compare elements in a sort, use a profiler to count the number of times items are compared
- We can use this data in the same way, plot it, see how it grows

Program Growth

Estimating Program Growth

- Run your program on various-sized inputs, collect times
  - Get a good number of points
  - Discard very small results
  - Provide inputs large enough to get past lower-order noise
- Find an upper and lower bound
- Consider $\frac{T_n}{f_n}$ for various functions f
  - Identify upper and lower bounds
  - Try to pinch in, get the upper and lower bounds closer to each other
  - You might not get them to meet

Estimating Program Growth – e.g.

Consider times T discovered for various input sizes n:

n	$T(n)$
10	3908.51
20	20657.40
30	55954.53
40	113992.17
50	198284.36
60	311920.28
70	457689.75
80	638156.74
90	855706.82
100	1112580.00

- $T(n)$ appears to be increasing w/out bound
  - So, T is not constant
  - Maybe. We don't *know* this

Let's compare to $f(n) = n$

E.g. – Line is a Lower Bound

n	$T(n)/n$
10	390.85
20	1032.87
30	1865.15
40	2849.80
50	3965.69
60	5198.67
70	6538.43
80	7976.96
90	9507.85
100	11125.80

- $T(n)/n$ also appears to be increasing, w/out bound
- So, $f(n) = n$ looks like a lower bound
  - I.e., $T(n) \in \Omega(n)$
  - If not tight, if it increases w/out bound, then $T(n) \in \omega(n)$

Let's try $f(n) = n^2$:

E.g. – Quadratic is a Lower Bound

n	$T(n)/n^2$
10	39.09
20	51.64
30	62.17
40	71.25
50	79.31
60	86.64
70	93.41
80	99.71
90	105.64
100	111.26

- $T(n)/n^2$ also appears to be increasing
- $f(n) = n^2$ looks like a lower bound
  - I.e., $T(n) \in \Omega(n^2)$
  - If not tight, if it increases w/out bound, then $T(n) \in \omega(n^2)$

Let's try $f(n) = n^3$:

E.g. – Cubic is an Upper Bound

n	$T(n)/n^3$
10	3.91
20	2.58
30	2.07
40	1.78
50	1.59
60	1.44
70	1.33
80	1.25
90	1.17
100	1.11

- $T(n)/n^3$ is decreasing
- $f(n) = n^3$ looks like an upper bound
  - I.e., $T(n) \in \mathrm{O}(n^3)$
  - If not tight, if the values tend towards 0, then $T(n) \in \mathrm{o}(n^3)$
- We know that T grows no slower than a quadratic, and no faster than a cubic
- We *might* be able to improve one or both of those bounds

E.g. – $n^2 \log n$

Consider $f_n = n^2 \log n$

n	$\frac{T(n)}{n^2 \log n}$
10	16.974
20	17.239
30	18.279
40	19.313
50	20.274
60	21.162
70	21.986
80	22.755
90	23.477
100	24.159

- $\frac{T(n)}{n^2 \log n}$ also seems to be increasing
- So, we have a new (better) lower bound
  - $T(n) \in \Omega(n^2 \log n)$

Let's try moving up a bit more:

E.g. – $n^{2.3}$

Consider $f_n = n^{2.3}$

n	$T(n)/n^{2.3}$
10	19.589
20	21.024
30	22.411
40	23.558
50	24.528
60	25.369
70	26.112
80	26.781
90	27.388
100	27.947

- I'm comfortable saying $n^{2.3}$ is a lower bound
  - $T(n) \in \Omega(n^{2.3})$

Let's try moving up a bit more:

E.g. – $n^{2.5}$

Consider $f_n = n^{2.5}$

n	$T(n)/n^{2.5}$
10	12.360
20	11.548
30	11.351
40	11.265
50	11.217
60	11.186
70	11.164
80	11.148
90	11.136
100	11.126

- This looks like an upper bound
  - $T_n \in \mathrm{O}(n^2\sqrt{n})$
- Is it tight?
- Let's try something a little lower:

E.g. – $n^{2.4}$

Consider $f_n = n^{2.4}$

n	$T(n)/n^{2.4}$
10	15.560
20	15.581
30	15.949
40	16.290
50	16.587
60	16.845
70	17.074
80	17.279
90	17.464
100	17.633

- Increasing, so, lower bound
  - $T_n \in \Omega(n^{2.4})$

E.g. – Conclusion

- We have T_n bound below by $n^{2.4}$ and bound above by $n^{2.5}$
- We maybe didn't find it exactly, but we have a very good idea how this algorithm grows
- Only push in each direction while you're comfortable w/the data
- None of this is proof
  - We need to choose input size sufficiently large to get past lower-order terms
  - No way to *know*
  - But, a program, on a given computer, has a practical upper limit on input size

Comparing Functions, Small n

- You are generally not going to be able to find tight bounds, using the method above
- E.g., you can show that $3n \ln n \in \mathrm{o}(n^{1.1})$
  - Hint: Use L'Hopital's Rule
- However $3n \ln n > n^{1.1}, \ \forall\ 3 < n < 1.2 \times 10^{16}$
- $\frac{3n \ln n}{n^{1.1}}$ will be increasing until around 66,000
- Remember, there's no such thing as a big number
- So, don't slice powers of n too finely

Notes on Scale

Leading Zeros

Remember your Physics or Chemistry lab lessons

- As you divide by larger magnitudes, your data pick up leading zeros
  - These are *not* significant digits
  - Do not make a statement such as “There is only a difference of 0.00001”
  - This is meaningless, neither large nor small
- Scale the column:

n	$T(n)/n^2$	$T(n)/n^2(*10000)$
10000	0.000238	2.380
20000	0.0002576	2.576
30000	0.0002691	2.691
40000	0.0002764	2.764
50000	0.0002829	2.829
60000	0.0002856	2.856

Scaling, Scientific Notation

- The scale is irrelevant
  - As long as it's consistent for a given column
- You can use scientific notation
  - Keep the exponent the same

n	$T(n)/f(n)$
10000	2.384 e-8
20000	1.288 e-8
30000	0.8970e-8
40000	0.6917e-8
50000	0.5659e-8
60000	0.4762e-8