Analysis

Imran Rashid

University of Washington

Jan 9, 2008

Defining Efficiency

- "Runs fast on typical real problem instances"
- Pro:
  - sensible, bottom-line-oriented
- Con:
  - moving target (diff computers, compilers, Moore's law)
  - highly subjective (how fast is "fast"? what's "typical"?)

Efficiency

- Our correct TSP algorithm was incredibly slow
- Basically slow no matter what computer you have
- We want a general theory of “efficiency” that is
    - Simple
    - Objective
    - Relatively independent of changing technology
    - But still predictive - “theoretically bad” algorithms should be bad in practice and vice versa (usually)

Measuring efficiency

- Time # of instructions executed in a simple programming language
  - only simple operations (+,*,-,=,if,call,. . .)
  - each operation takes one time step
  - each memory access takes one time step
  - no fancy stuff (add these two matrices, copy this long string,. . .) built in; write it/charge for it as above
- No fixed bound on the memory size

We left out things but...

- Things we've dropped

Complexity analysis

- Problem size n
  - Worst-case complexity: max # steps algorithm takes on any input of size n
  - Best-case complexity: min # steps algorithm takes on any input of size n
  - Average-case complexity: avg # steps algorithm takes on inputs of size n

Pros and cons:

- Best-case
- Average-case
- Worst-case

Why Worst-Case Analysis?

General Goals

- Characterize growth rate of (worst-case) run time as a function of problem size, up to a constant factor
- Why not try to be more precise?
  - Technological variations (computer, compiler, OS, ...) easily 10x or more
  - Being more precise is a ton of work
  - A key question is "scale up": if I can afford to do it today, how much longer will it take when my business problems are twice as large? (E.g. today: cn^2, next year: $c(2n)^2 = 4cn^2$: 4 x longer.)

Complexity

- The complexity of an algorithm associates a number $T(n)$, the worst-case time the algorithm takes, with each problem size n.

O-notation etc

- Given two functions f and $g : N \rightarrow R$
  - $f(n)$ is $O(g(n))$ iff there is a constant $c > 0$ so that $f(n)$ is eventually always $\leq c\ g(n)$
  - $f(n)$ is $\Omega(g(n))$ iff there is a constant $c > 0$ so that $f(n)$ is eventually always $\geq c\ g(n)$
  - $f(n)$ is $\Theta(g(n))$ iff there is are constants $c_1, c_2 > 0$ so that eventually always $c_1\ g(n) \leq f(n) \leq c_2\ g(n)$

Examples

$10n^2 - 16n + 100$

Properties

- Transitivity.
  - If $f = O(g)$ and $g = O(h)$ then $f = O(h)$.
  - If $f = \Omega(g)$ and $g = \Omega(h)$ then $f = \Omega(h)$.
  - If $f = \Theta(g)$ and $g = \Theta(h)$ then $f = \Theta(h)$.
- Additivity.
  - If $f = O(h)$ and $g = O(h)$ then $f + g = O(h)$.
  - If $f = \Omega(h)$ and $g = \Omega(h)$ then $f + g = \Omega(h)$.
  - If $f = \Theta(h)$ and $g = O(h)$ then $f + g = \Theta(h)$.

Asymptotic Bounds for Some Common Functions

- Polynomials:
  $a_0 + a_1 n + \ldots + a_d n_d$ is $\Theta(n^d)$ if $a_d > 0$
- Logarithms:
  $O(log_a n) = O(log_b n)$ for any constants $a, b > 0$
- Logarithms:
  For all $x > 0$, $logn = O(n^x)$

"One-Way Equalities"

$2 + 2$ is 4	$2n^2 + 5n$ is $O(n^3)$
$2 + 2 = 4$	$2n2 + 5n = O(n^3)$
$4 = 2 + 2$	~~$O(n^3) = 2n^2 + 5n$~~
All dogs are mammals	~~All mammals are dogs~~

- Bottom line:
  - OK to put big-O in R.H.S. of equality, but not left.
  - (Better, but uncommon, notation: $2n^2 + 5n \in O(n^3)$)

Working with O-Ω-Θ notation

- Claim: For any a, and any $b > 0$, $(n + a)^b$ is $\Theta(n^b)$

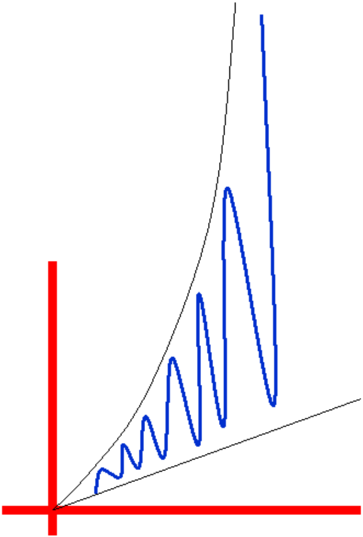
Working with O-Ω-Θ notation (2)

- Claim: For any $a, b > 1$, $log_a n$ is $\Theta(log_b n)$

Big-Theta, etc. not always "nice"

$$f(x) = \begin{cases} n^2 & \text{if } n \text{ even,} \\ n & \text{else} \end{cases}$$

- $f(n) \neq \Theta(n^a)$ for any a.
- Fortunately, this is rare

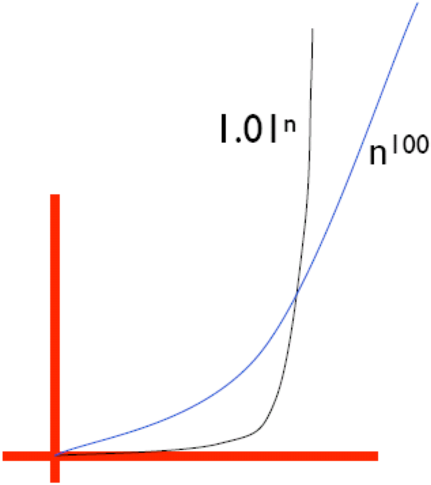


A Possible Misunderstanding?

- We have looked at
  - type of complexity analysis
    - worst-, best-, average-case
  - types of function bounds
    - O, Ω, Θ
- These two considerations are independent of each other
  - one can do any type of function bound with any type of complexity analysis - measuring different things with same yardstick

Asymptotic Bounds for Some Common Functions

- Exponentials. For all $r > 1$ and all $d > 0$, $n^d = O(r^n)$.



Polynomial time

- Running time is $O(n^d)$ for some constant d independent of the input size n.

Why It Matters

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Example: Studying Protein Interactions

- Yeast: $\approx 6k$ proteins

- Human: $\approx 25k$ proteins

Key Facts

Geek-speak Faux Pas du Jour

- “Any comparison-based sorting algorithm requires at least O(n log n) comparisons.”
  - Statement doesn't "type-check."
  - Use Ω for lower bounds.

Summary

- Typical initial goal for algorithm analysis is to find a
  reasonably tight (i.e., Θ if possible)
  asymptotic (i.e., O or Θ)
  bound on (usually upper bound)
  worst case running time
  as a function of problem size
- This is rarely the last word, but often helps separate good algorithms from blatantly poor ones - so you can concentrate on the good ones!