CSE 421: Intro Algorithms

2: Analysis

Winter 2012

Larry Ruzzo

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)

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"?)

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

Mathematically,

$T: N+ \rightarrow R+$

i.e., T is a function mapping positive integers (problem sizes) to positive real numbers (number of steps).

"Reals" so we can say, e.g., sqrt(n) instead of $\lceil \text{sqrt}(n) \rceil$

Asymptotic growth rate, i.e., characterize growth rate of worst-case run time as a function of problem size, up to a constant factor, e.g. $T(n) = O(n^2)$

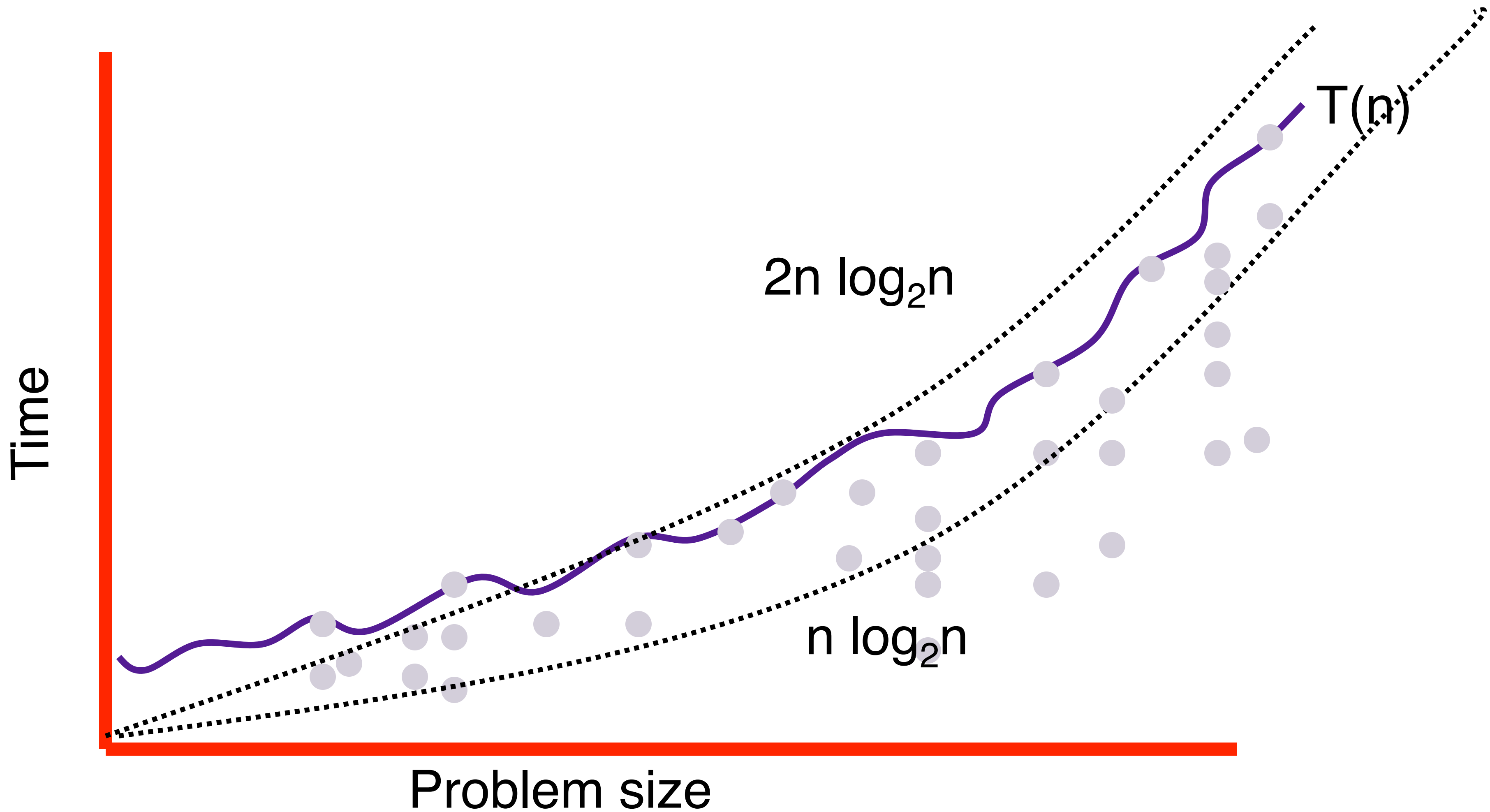
Why not try to be more precise?

Average-case, e.g., is hard to define, analyze

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 this today, how much longer will it take when my business is 2x larger? (E.g. today: cn^2, next year: $c(2n)^2 = 4cn^2$: 4 x longer.) Big-O analysis is adequate to address this.



T(n)
2n log2n
Time
n log2n
Problem size

O-notation, etc.

Given two functions f and g:N→R

f(n) is O(g(n)) iff there is a constant c>0 so that
f(n) is eventually always ≤ c g(n)

f(n) is Ω (g(n)) iff there is a constant c>0 so that
f(n) is eventually always ≥ c g(n)

f(n) is Θ (g(n)) iff there is are constants c_1, c_2>0 so that
eventually always $c_1g(n) \leq f(n) \leq c_2g(n)$

Examples

$10n^2$-$16n+100$ is $O(n^2)$ also $O(n^3)$

$10n^2-16n+100 \le 11n^2$ for all $n \ge 10$

$10n^2$-$16n+100$ is Ω (n^2) also Ω (n)

$10n^2-16n+100 \ge 9n^2$ for all $n \ge 16$

Therefore also $10n^2-16n+100$ is Θ (n^2)

$10n^2$-$16n+100$ is not $O(n)$ also not Ω (n^3)

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 $f + g = \Theta(h)$.

Working with O-Ω-Θ notation

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

$(n+a)^b \le (2n)^b$ for $n \ge |a|$

$= 2^b n^b$

$= cn^b$ for $c = 2^b$

so $(n+a)^b$ is $O(n^b)$

$(n+a)^b \ge (n/2)^b$ for $n \ge 2|a|$ (even if $a < 0$)

$= 2^{-b} n^b$

$= c'n$ for $c' = 2^{-b}$

so $(n+a)^b$ is $\Omega\ (n^b)$

Working with O-Ω-Θ notation

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

$$\log_a b = x \text{ means } a^x = b$$

$$a^{\log_a b} = b$$

$$(a^{\log_a b})^{\log_b n} = b^{\log_b n} = n$$

$$(\log_a b)(\log_b n) = \log_a n$$

$$c \log_b n = \log_a n \text{ for the constant } \mathrm{c} = \log_a b$$

So :

$$\log_b n = \Theta(\log_a n) = \Theta(\log n)$$

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:

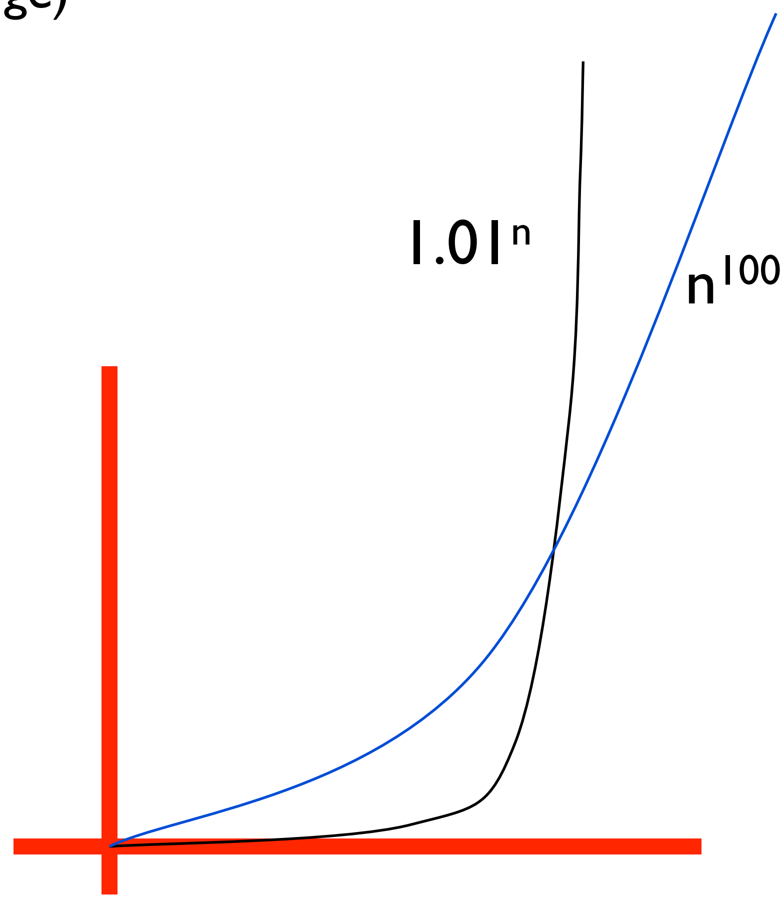
$O(\log_a n) = O(\log_b n)$ for any constants $a,b > 0$

Logarithms:

For all $x > 0$, $\log n = O(n^x)$

log grows slower than every polynomial

For all $r > 1$ (no matter how small)
and all $d > 0$, (no matter how large)
$n^d = O(r^n)$.



In short, every exponential grows faster than every polynomial!

Domination

f(n) is o(g(n)) iff $\lim_{n \to \infty} f(n)/g(n) = 0$

that is g(n) *dominates* f(n)

If $a \le b$ then n^a is $O(n^b)$

If $a < b$ then n^a is $o(n^b)$

Note:

if f(n) is Θ (g(n)) then it cannot be o(g(n))

Working with little-o

$n^2 = o(n^3)$ [Use algebra]:

$$\lim_{n\to\infty} \frac{n^2}{n^3} = \lim_{n\to\infty} \frac{1}{n} = 0$$

$n^3 = o(e^n)$ [Use L'Hospital's rule 3 times]:

$$\lim_{n\to\infty} \frac{n^3}{e^n} = \lim_{n\to\infty} \frac{3n^2}{e^n} = \lim_{n\to\infty} \frac{6n}{e^n} = \lim_{n\to\infty} \frac{6}{e^n} = 0$$

P: Running time $O(n^d)$ for some constant d

(d is independent of the input size n)

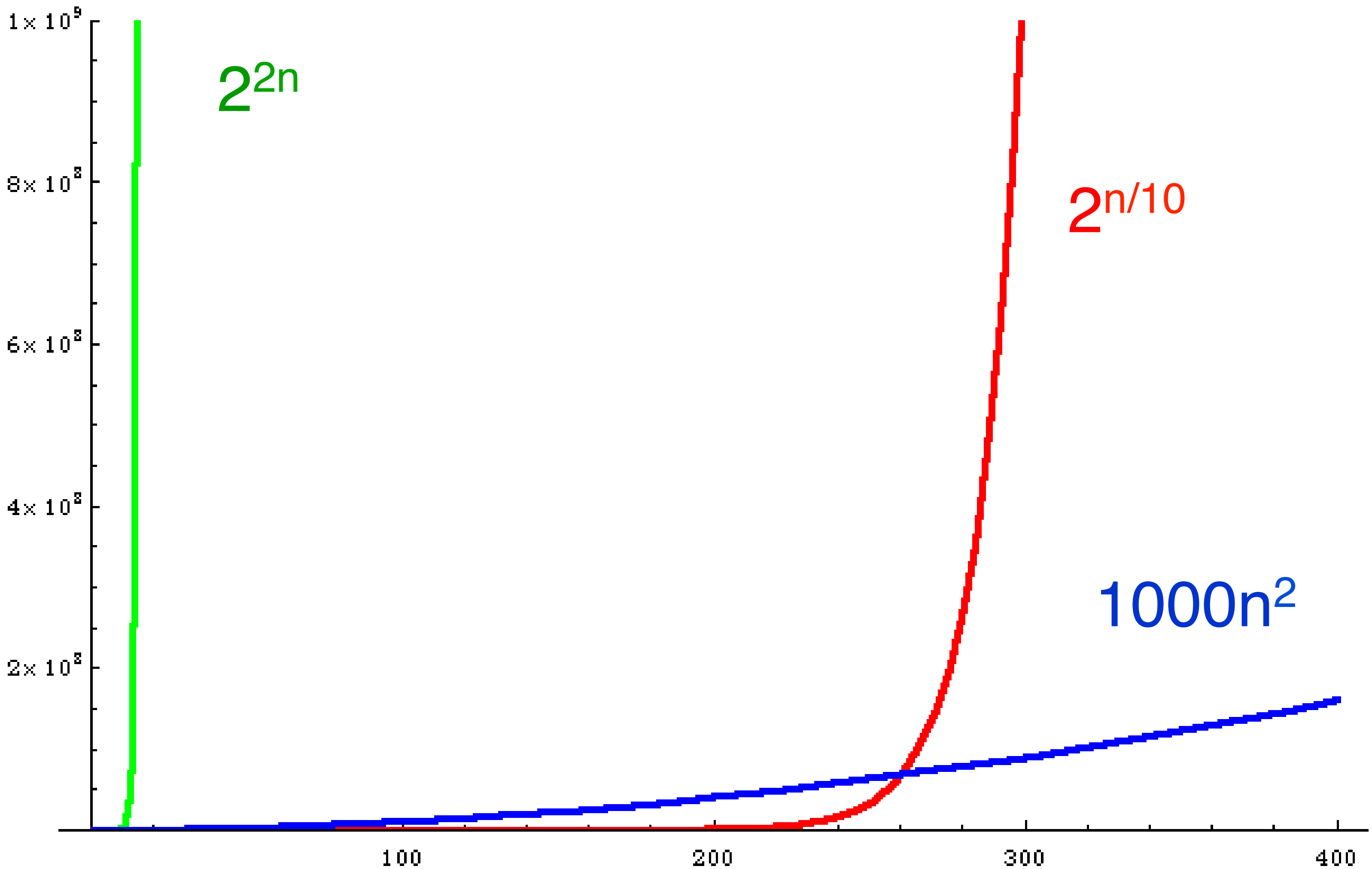
Nice scaling property: there is a constant c s.t. *doubling* n, time increases only by a factor of c.

(E.g., $c \sim 2^d$)

Contrast with exponential: For any constant c, there is a d such that $n \rightarrow n+d$ increases time by a factor of more than c.

(E.g., c = 100 and d = 7 for 2^n vs 2^{n+7})

polynomial vs exponential growth



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

not only get very big, but do so abruptly, which likely yields erratic performance on small instances

Next year's computer will be 2x faster. If I can solve problem of size n_0 today, how large a problem can I solve in the same time next year?

Complexity	Increase	E.g. $T=10^{12}$		
$O(n)$	$n_0 \to 2n_0$	10^{12}	$\to$	2×10^{12}
$O(n^2)$	$n_0 \to \sqrt{2}\, n_0$	10^6	$\to$	1.4×10^6
$O(n^3)$	$n_0 \to \sqrt[3]{2}\, n_0$	10^4	$\to$	1.25×10^4
$2^{n/10}$	$n_0 \to n_0+10$	400	$\to$	410
2^n	$n_0 \to n_0+1$	40	$\to$	41

Point is not that n^{2000} is a nice time bound, or that the differences among n and 2n and n^2 are negligible.

Rather, simple theoretical tools may not easily capture such differences, whereas exponentials are qualitatively different from polynomials, so more amenable to theoretical analysis.

- "My problem is in P" is a starting point for a more detailed analysis
- "My problem is *not* in P" may suggest that you need to shift to a more tractable variant, or otherwise readjust expectations

Typical initial goal for algorithm analysis is to find an

- asymptotic
- upper bound on
- 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 - concentrate on the good ones!