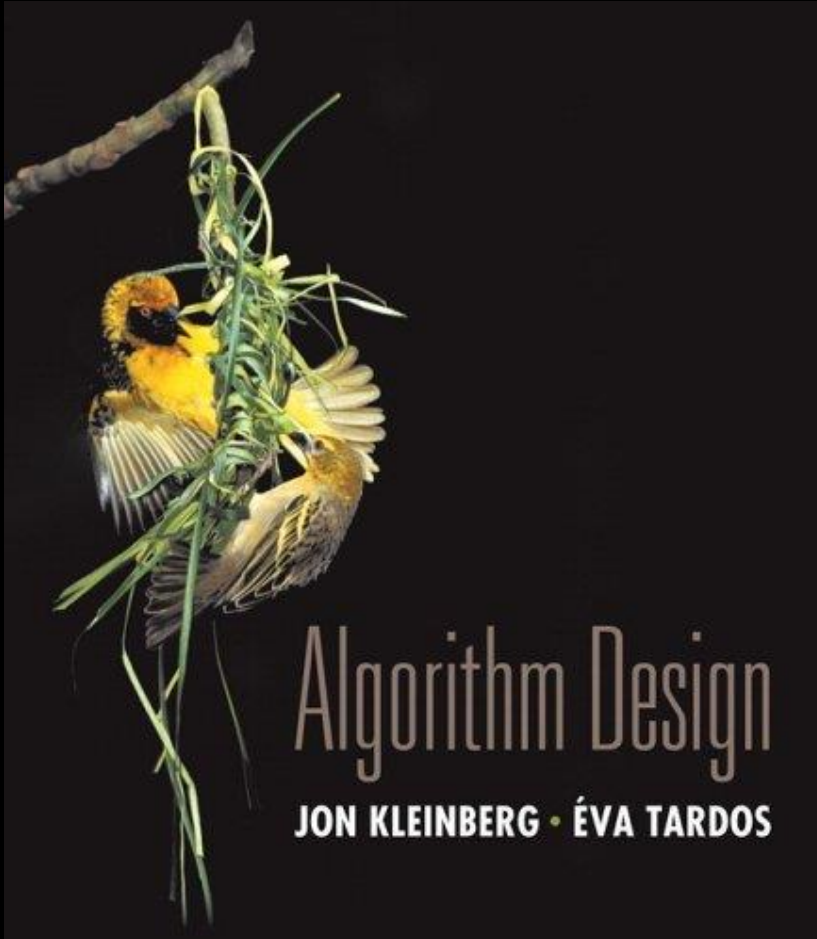


Chapter 2

Basics of Algorithm Analysis



Slides by Kevin Wayne.
Copyright © 2005 Pearson-Addison Wesley.
All rights reserved.
Updated by T. K. Prasad

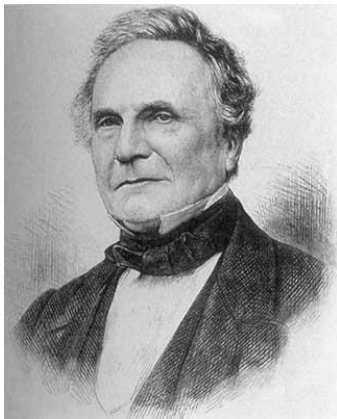
2.1 Computational Tractability

"For me, great algorithms are the poetry of computation. Just like verse, they can be terse, allusive, dense, and even mysterious. But once unlocked, they cast a brilliant new light on some aspect of computing." - *Francis Sullivan*

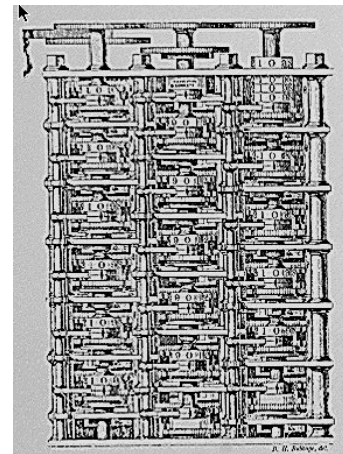
Computational Tractability

As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise - By what course of calculation can these results be arrived at by the machine in the shortest time?

- *Charles Babbage*



Charles Babbage (1864)



Analytic Engine (schematic)

Asymptotic Analysis of Computational Complexity of Algorithm: Motivation

To characterize the inherent nature of an algorithm, we need a *metric* that is *robust* w.r.t. hardware/language variations, ignores setup costs and depends only on input size.

That is, it *abstracts* from dependence on platform-specific details, *input instance specific* details, and *predicts the behavior on large inputs*.

Asymptotic Analysis of Computational Complexity of Algorithm: Motivation

1. Abstract from multiplicative constants.
2. Focus on **growth rates** of resource utilization (as a function of data size).



Scalability

3. Consider large data sizes.
4. PLUS: Abstract from specific input characteristics or distribution.

Polynomial-Time

Brute force. For many non-trivial problems, there is a natural brute force search algorithm that checks every possible solution.

- Typically takes 2^N time or worse for inputs of size N .
- Unacceptable in practice.

↖ $n!$ for stable matching
with n men and n women

Desirable scaling property

There exists constants $c > 0$ and $d > 0$ such that on every input of size N , its running time is bounded by $c N^d$ steps.

Def. An algorithm is **poly-time** if the above scaling property holds.

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

Growth Rates

	<i>constant</i>	<i>logarithmic</i>	<i>linear</i>	<i>N-log-N</i>	<i>quadratic</i>	<i>cubic</i>	<i>exponential</i>
<i>n</i>	$O(1)$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^3)$	$O(2^n)$
1	1	1	1	1	1	1	2
2	1	1	2	2	4	8	4
4	1	2	4	8	16	64	16
8	1	3	8	24	64	512	256
16	1	4	16	64	256	4,096	65536
32	1	5	32	160	1,024	32,768	4,294,967,296
64	1	6	64	384	4,069	262,144	1.84×10^{19}

Assume the machine is executing 1 billion (10^9) operations per second. Then the following table gives us how running times for algorithms in different complexity classes changes with problem size.

Complexity Class	N = 10	N = 100	N = 1,000	...	N = 1,000,000
$O(1)$	1×10^{-7} seconds	1×10^{-7} seconds	1×10^{-7} seconds	...	1×10^{-7} seconds
$O(\log_2 N)$	3.3×10^{-7} seconds	6.6×10^{-7} seconds	10×10^{-7} seconds	...	20×10^{-7} seconds
$O(N)$	1×10^{-7} seconds	1×10^{-6} seconds	1×10^{-5} seconds	...	1×10^{-3} seconds
$O(N \log_2 N)$	3.3×10^{-7} seconds	6.6×10^{-6} seconds	10×10^{-5} seconds	...	20×10^{-3} seconds
$O(N^2)$	1×10^{-6} seconds	1×10^{-4} seconds	1×10^{-2} seconds	...	2.7 hours
$O(N^3)$	1×10^{-5} seconds	1×10^{-2} seconds	10 seconds	...	3×10^3 years
$O(2^N)$	1×10^{-5} seconds	4×10^{21} centuries	fuggidaboutit	...	fuggidaboutit

Worst-Case Analysis

Worst case running time. Obtain bound on **largest possible** running time of algorithm on input of size N .

- Generally captures efficiency in practice.
- Draconian view, but hard to find effective alternative.

Average case running time. Obtain bound on running time of algorithm on **random** input as a function of input size N .

- Hard (or impossible) to accurately model real instances by random distributions.

Worst-Case Polynomial-Time

Def. An algorithm is **efficient** if its running time is polynomial.

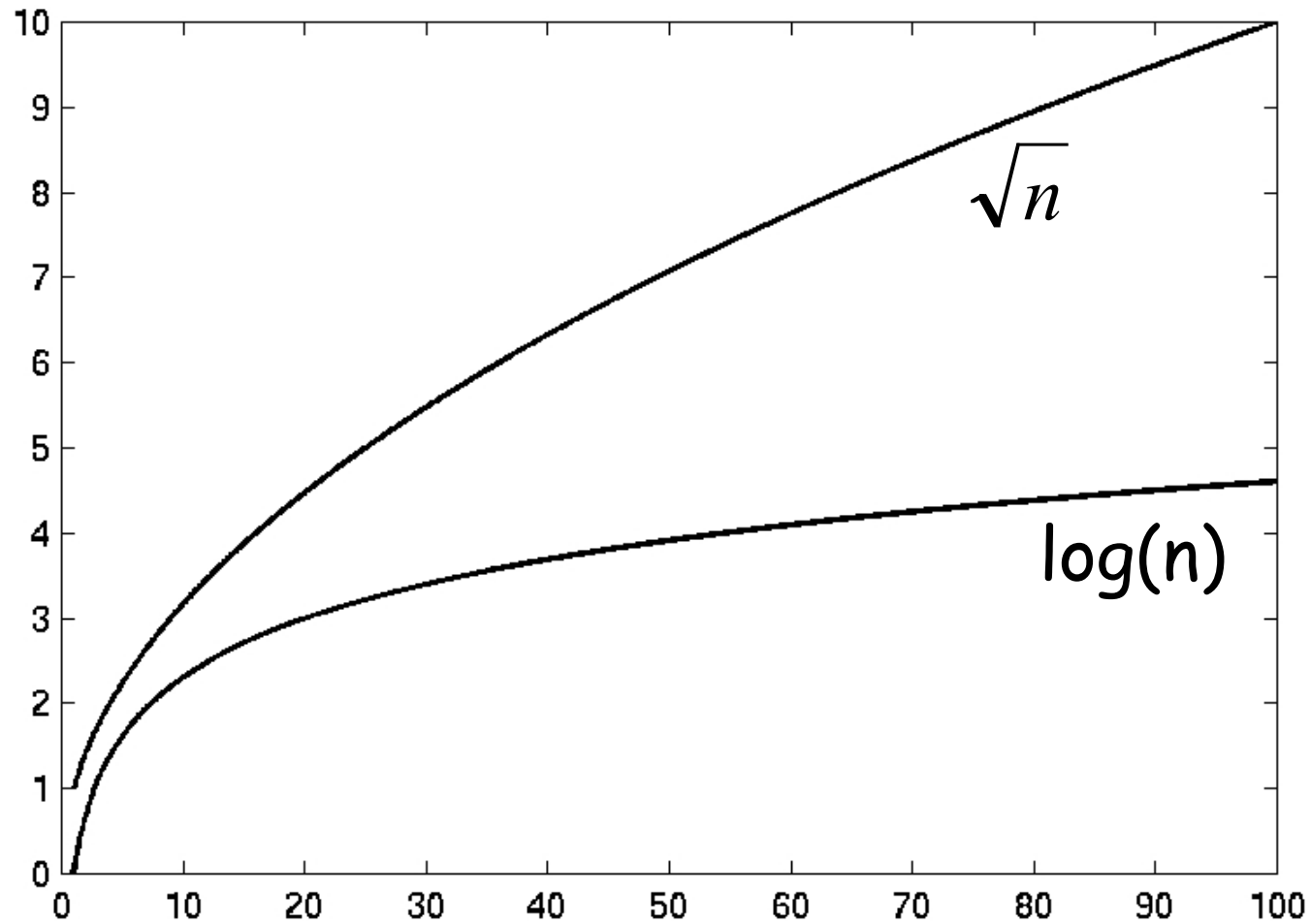
Justification: **It really works in practice!**

- Although $6.02 \times 10^{23} \times N^{20}$ is technically poly-time, it would be useless in practice.
- **In practice, the poly-time algorithms that people develop almost always have low constants and low exponents.**
- **Breaking through the exponential barrier of brute force typically exposes some crucial structure of the problem.**

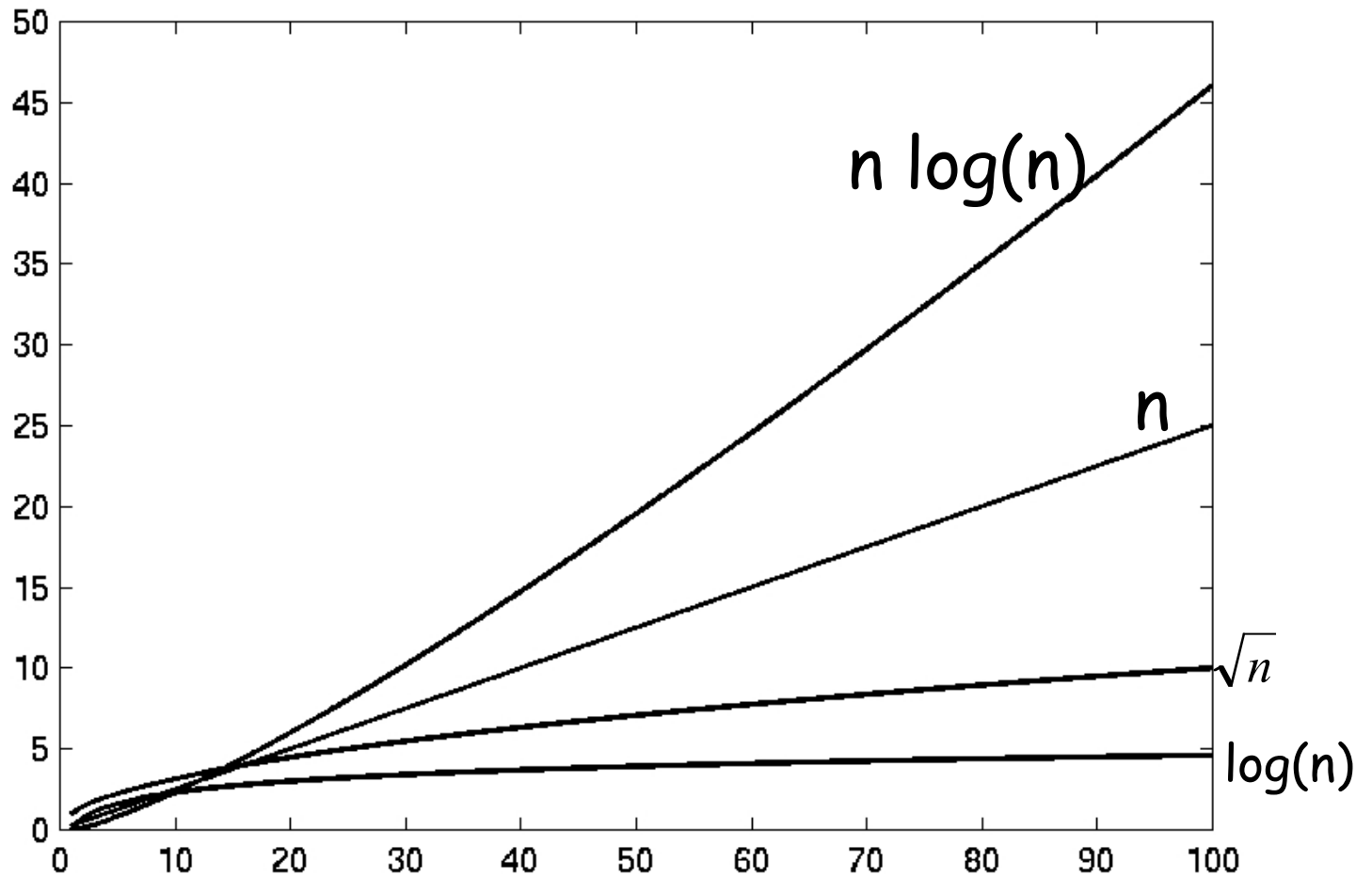
Exceptions.

- Some poly-time algorithms do have high constants and/or exponents, and are useless in practice.  Khachiyan method for LP
- Some exponential-time (or worse) algorithms are widely used because the worst-case instances seem to be rare.  simplex method
Unix grep

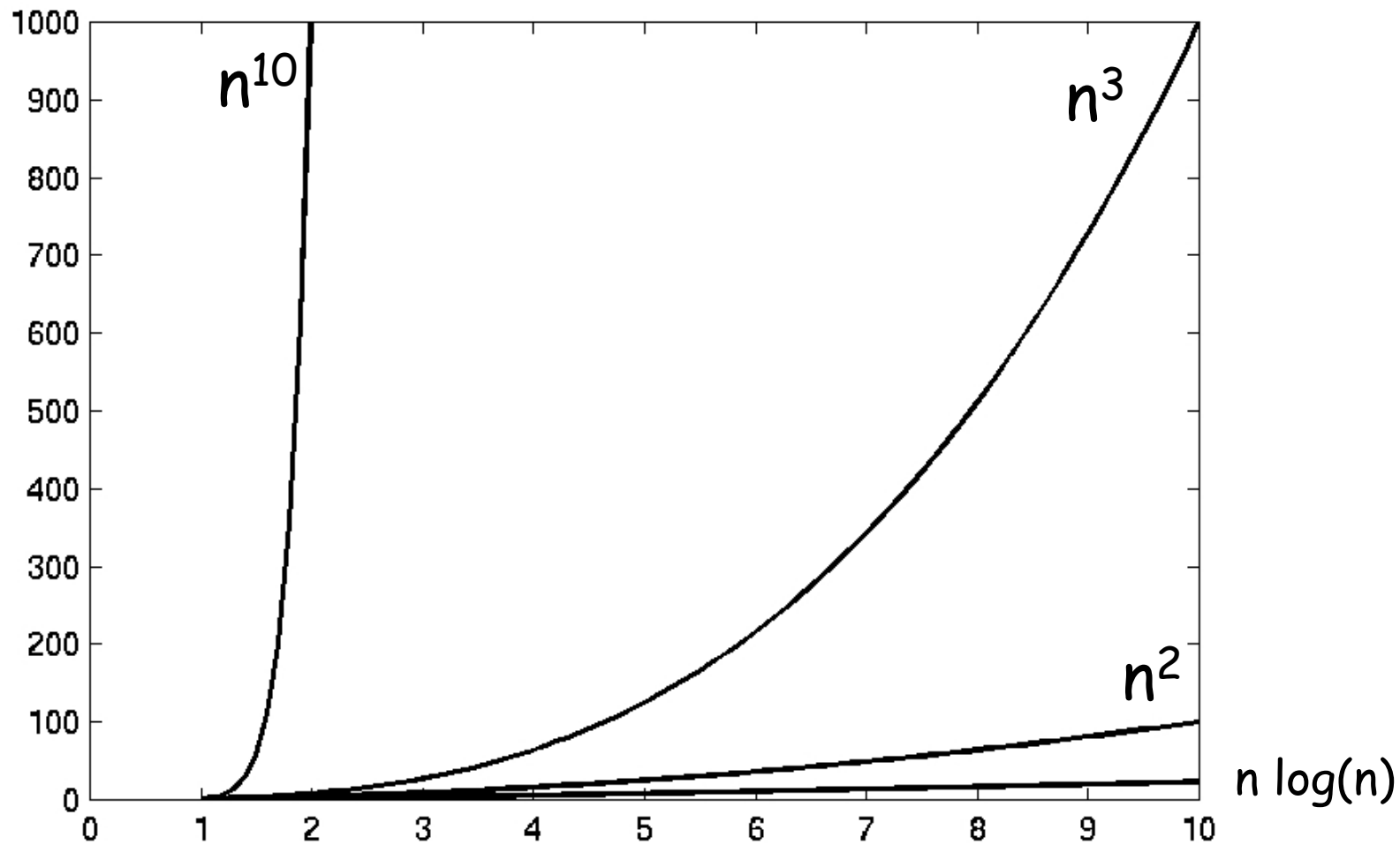
Complexity Graphs



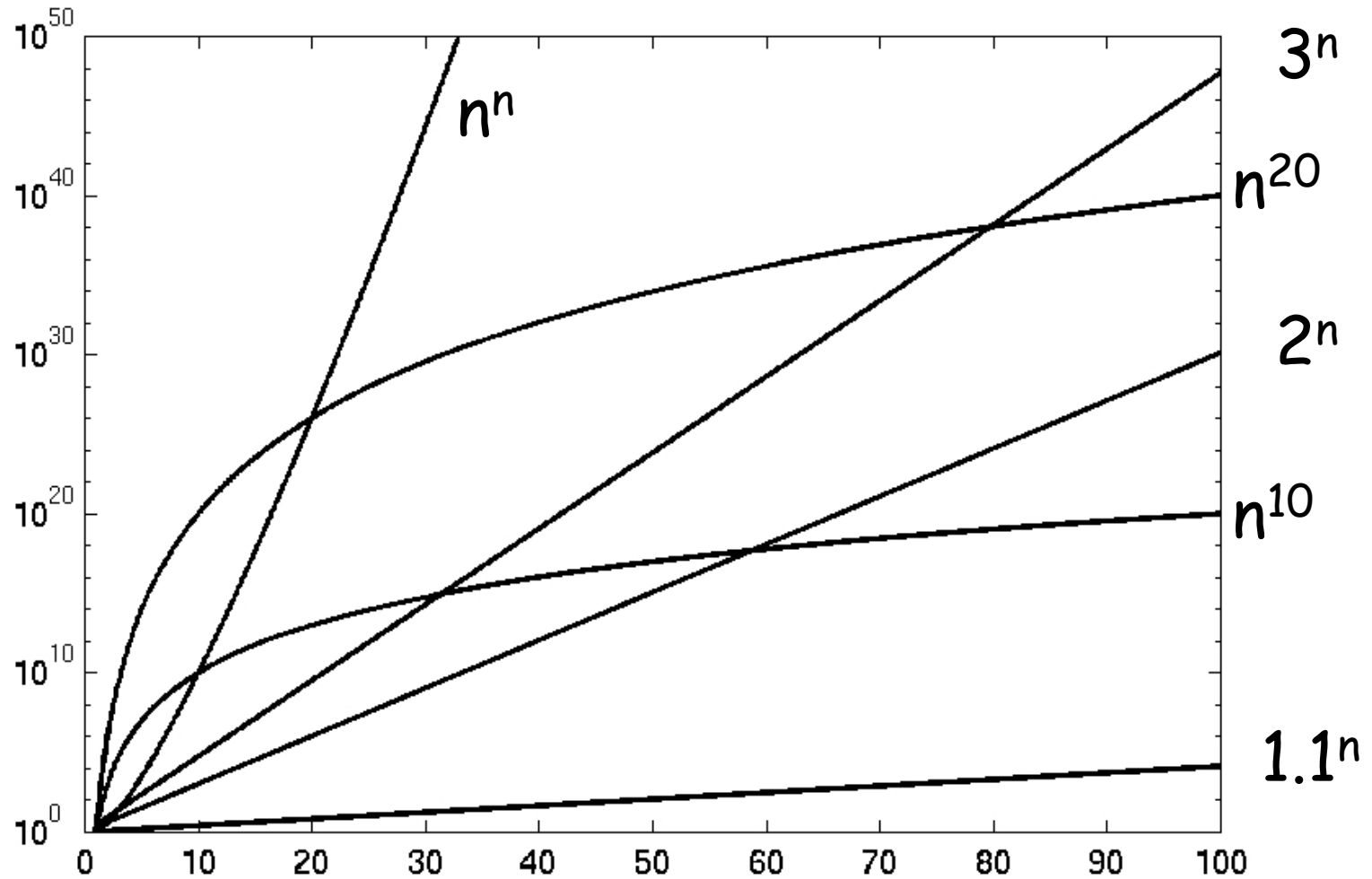
Complexity Graphs



Complexity Graphs



Complexity Graphs (log scale)



Asymptotic Order of Growth : Formalization

Upper bounds. $T(n)$ is $O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq c \cdot f(n)$.

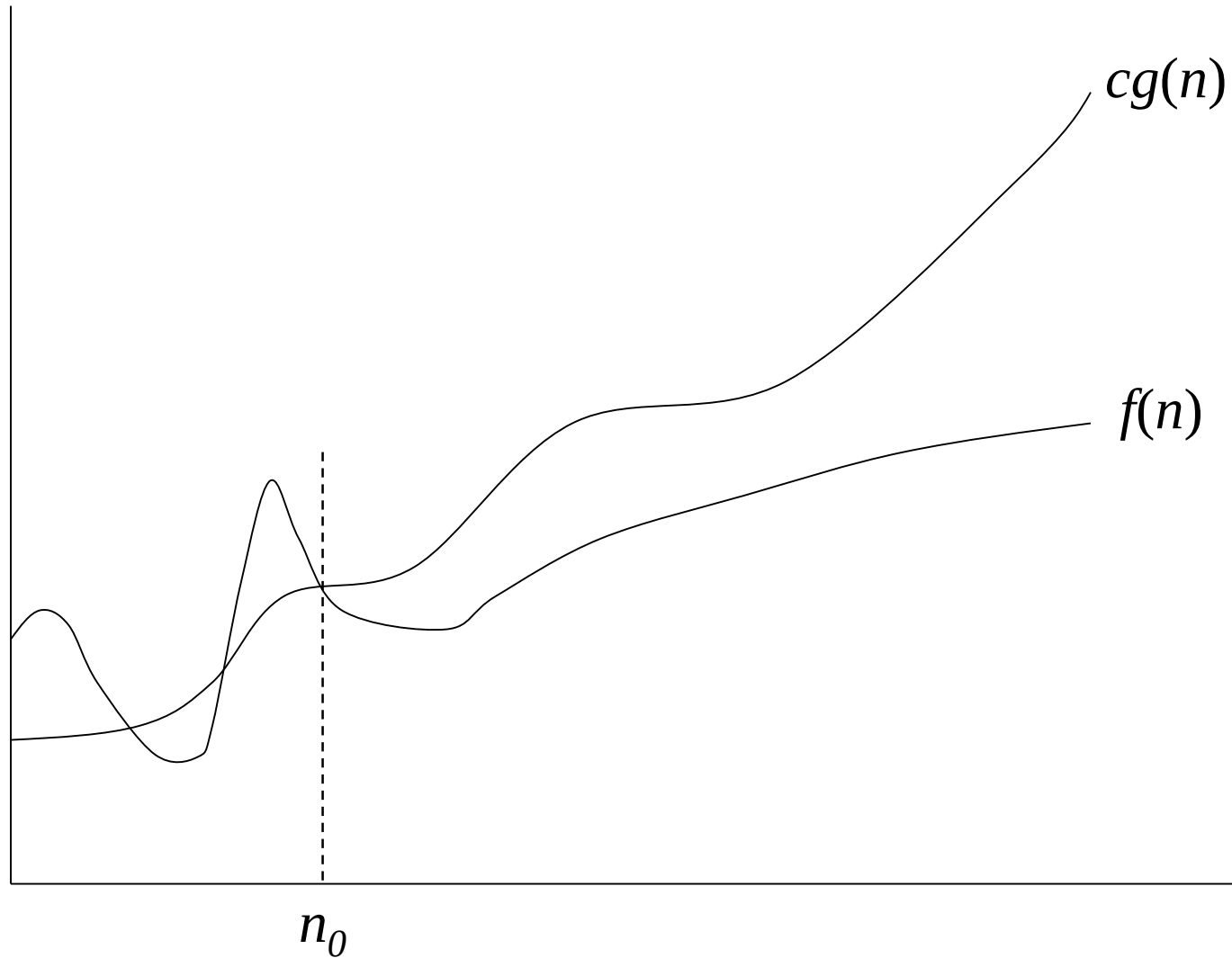
Lower bounds. $T(n)$ is $\Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot f(n)$.

Tight bounds. $T(n)$ is $\Theta(f(n))$ if $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$.

E.g.: $T(n) = 32n^2 + 17n + 32$.

- $T(n)$ is $O(n^2)$, $O(n^3)$, $\Omega(n^2)$, $\Omega(n)$, and $\Theta(n^2)$.
- $T(n)$ is not $O(n)$, $\Omega(n^3)$, $\Theta(n)$, or $\Theta(n^3)$.

Visualization of $O(g(n))$



Notation

Slight abuse of notation. $T(n) = O(f(n))$.

- Asymmetric:

- $f(n) = 5n^3$; $g(n) = 3n^2$

- $f(n) = O(n^3) = g(n)$

- but $f(n) \neq g(n)$.

- Better notation: $T(n) \in O(f(n))$.

Meaningless statement. Any comparison-based sorting algorithm requires at least $O(n \log n)$ comparisons.

- Statement doesn't "type-check."

- Use Ω for lower bounds.

Big-O

$$0.5n^2 \in O(n^2)$$

$$1,000,000n^2 + 150,000 \in O(n^2)$$

$$5n^2 + 7n + 20 \in O(n^2)$$

$$2n^3 + 2 \notin O(n^2)$$

$$n^{2.1} \notin O(n^2)$$

More Big-O

- Prove that: $20n^2 + 2n + 5 = O(n^2)$

we want to check if there is a c and n_0 , so that
 $cn^2 > 20n^2 + 2n + 5$ for $n > n_0$

Let $c = 21$ and $n_0 = 4$

$21n^2 > 20n^2 + 2n + 5$ for all $n > 4$

$n^2 > 2n + 5$ for all $n > 4$

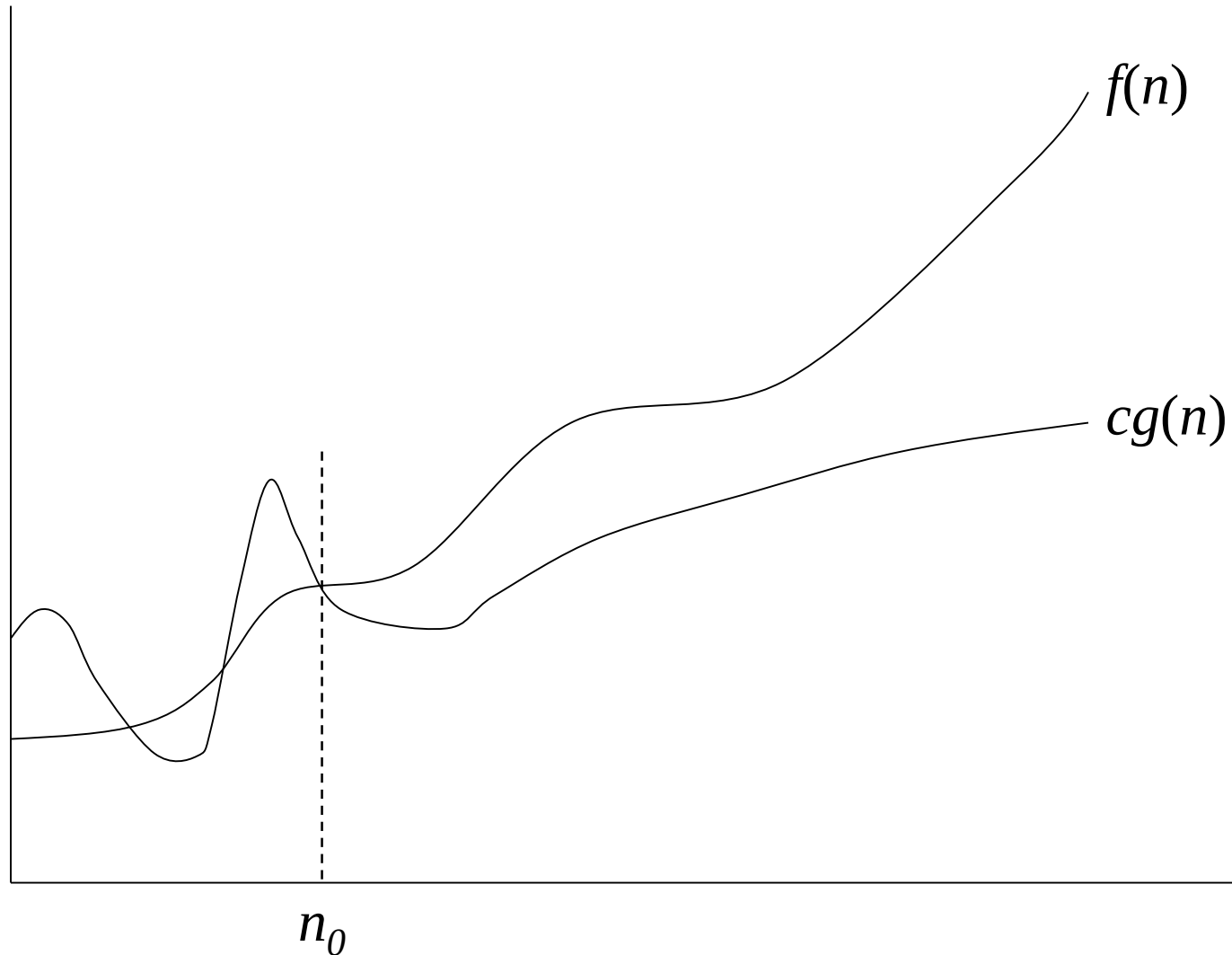
QED

Another Example

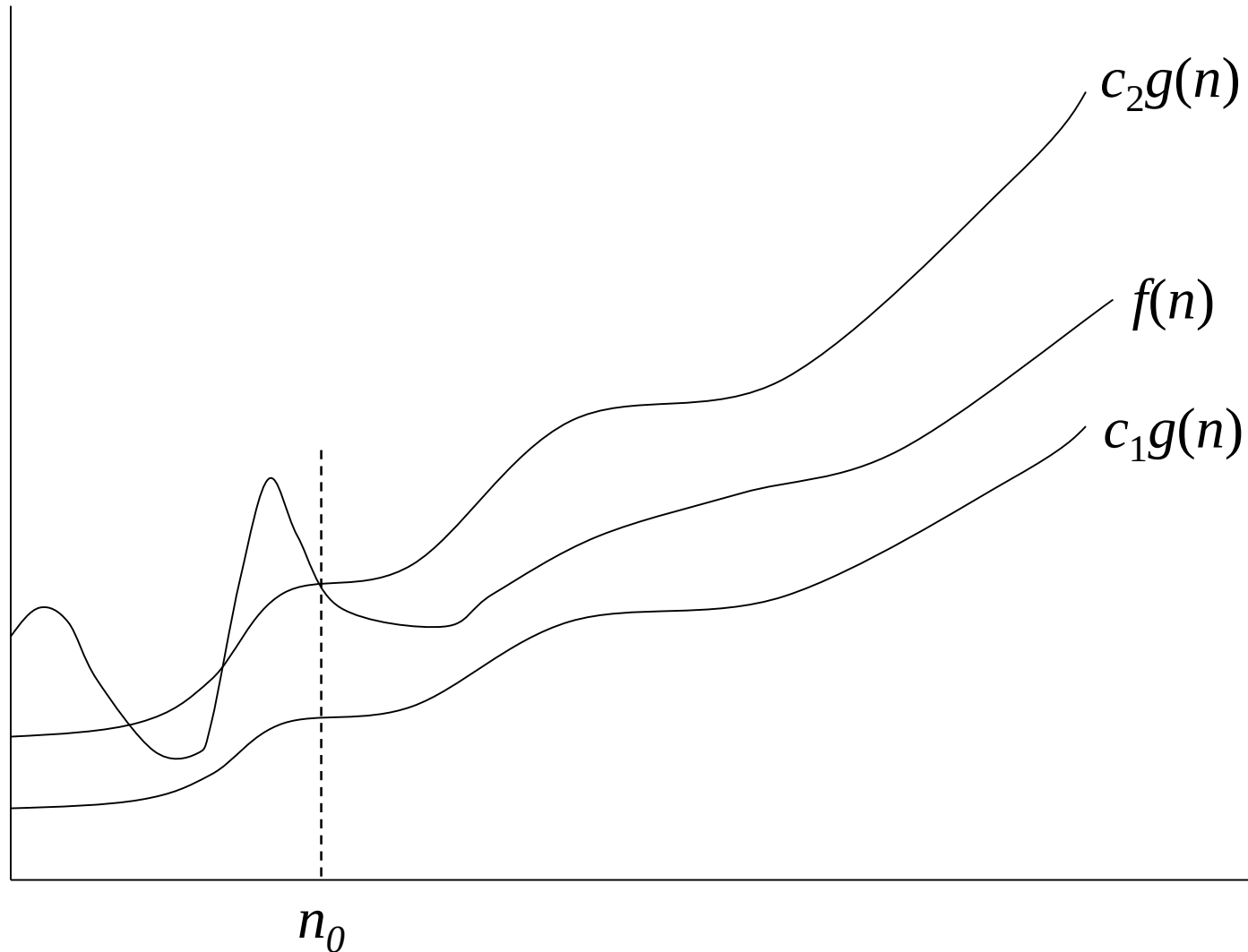
- Show that $2^n + n^2 = O(2^n)$ $2 \times 2^n > 2^n + n^2$
- Let $c = 2$ and $n_0 = 5$ $2^n > n^2 \quad \forall n \geq 5$

- *Tight Bounds*
 - We generally want the tightest bound we can find.
 - While it is true that $n^2 + 7n$ is in $O(n^3)$, it is more interesting to say that it is in $O(n^2)$

Visualization of $\Omega(g(n))$



Visualization of $\Theta(g(n))$



A Few More Examples

- $n \in O(n^2)$ $n \notin \Omega(n^2)$ $n \notin \Theta(n^2)$
- $200n^2 \in O(n^2)$ $200n^2 \in \Omega(n^2)$ $200n^2 \in \Theta(n^2)$
- $n^{2.5} \notin O(n^2)$ $n^{2.5} \in \Omega(n^2)$ $n^{2.5} \notin \Theta(n^2)$

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

Justification

If $f = O(g)$ and $g = O(h)$ then $f = O(h)$.

$$f = O(g)$$

$$g = O(h)$$

$$f(n) \leq c g(n) \quad \text{for } n > n_0$$

$$g(n) \leq d h(n) \quad \text{for } n > n_1$$

$\implies$

$$f = O(h)$$

$$\text{because } f(n) \leq c g(n) \leq c d h(n)$$

$$\text{for } n > \max(n_0, n_1)$$

Justification

If $f = \Omega(h)$ and $g = \Omega(h)$ then $f + g = \Omega(h)$.

$$f = \Omega(h)$$

$$f(n) \geq c h(n) \quad \text{for } n > n_0$$

$$g = \Omega(h)$$

$$g(n) \geq d h(n) \quad \text{for } n > n_1$$

$\implies$

$$f + g = \Omega(h)$$

because $f(n) + g(n) \geq \min(c, d) h(n)$
for $n > \max(n_0, n_1)$

Asymptotic Bounds for Some Common Functions

Polynomials. $a_0 + a_1n + \dots + a_dn^d$ is $\Theta(n^d)$ if $a_d > 0$.

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

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



can avoid specifying the
base

Logarithms. For every $x > 0$, $\log n = O(n^x)$.



log grows slower than every polynomial

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



every exponential grows faster than every polynomial

Factorial grows faster than exponential

Stirling's Approximation

$$n! = (\sqrt{2\pi n}) \left(\frac{n}{e}\right)^n (1 + o(1)) = 2^{\Theta(n \log n)}$$

↙
grows faster than every exponential

Ramanujan's Approximation

$$n! \sim \sqrt{\pi} \left(\frac{n}{e}\right)^n \sqrt[6]{8n^3 + 4n^2 + n + \frac{1}{30}}$$

Comparing Complexity

Limit Method Process

Say we have functions $f(n)$ and $g(n)$. We set up a limit quotient between f and g as follows:

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \begin{cases} 0 & \text{then } f(n) \in \mathcal{O}(g(n)) \\ c > 0 & \text{then } f(n) \in \Theta(g(n)) \\ \infty & \text{then } f(n) \in \Omega(g(n)) \end{cases}$$

- ▶ Justifications for the above can be proven using calculus, but for our purposes the limit method will be sufficient for showing asymptotic inclusions.
- ▶ Always try to look for algebraic simplifications *first*.
- ▶ If f and g *both* diverge or converge on zero or infinity, then you need to apply l'Hôpital's Rule.

Comparing Complexity

l'Hôpital's Rule

Theorem

(l'Hôpital's Rule) Let f and g , if the limit between the quotient $\frac{f(n)}{g(n)}$ exists, it is equal to the limit of the derivative of the denominator and the numerator.

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{f'(n)}{g'(n)}$$

Comparing Complexity

Useful Identities & Derivatives

Some useful derivatives that you should memorize include

- ▶ $(n^k)' = kn^{k-1}$
- ▶ $(\log_b(n))' = \frac{1}{n \ln(b)}$
- ▶ $(f_1(n)f_2(n))' = f_1'(n)f_2(n) + f_1(n)f_2'(n)$ (product rule)
- ▶ $(c^n)' = \ln(c)c^n \leftarrow \text{Careful!}$

Log Identities

- ▶ Change of Base Formula: $\log_b(n) = \frac{\log_c(n)}{\log_c(b)}$
- ▶ $\log(n^k) = k \log(n)$
- ▶ $\log(ab) = \log(a) + \log(b)$

2.4 A Survey of Common Running Times

Linear Time: $O(n)$

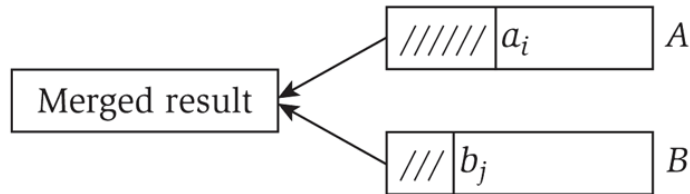
Linear time. Running time is at most a constant factor times the size of the input.

Computing the maximum. Compute maximum of n numbers $a_1, \dots, a_n$.

```
max ← a1
for i = 2 to n {
  if (ai > max)
    max ← ai
}
```

Linear Time: $O(n)$

Merge. Combine two sorted lists $A = a_1, a_2, \dots, a_n$ with $B = b_1, b_2, \dots, b_n$ into sorted whole.



```
i = 1, j = 1
while (both lists are nonempty) {
    if (ai ≤ bj) append ai to output list and increment i
    else          append bj to output list and increment j
}
append remainder of nonempty list to output list
```

Claim. Merging two lists of size n takes $O(n)$ time.

Pf. After each comparison, the length of output list increases by 1.

$O(n \log n)$ Time

$O(n \log n)$ time. Arises in divide-and-conquer algorithms.



also referred to as linearithmic time

Sorting. Mergesort, quicksort, and heapsort are sorting algorithms that perform $O(n \log n)$ comparisons.

Largest empty interval. Given n time-stamps $x_1, \dots, x_n$ on which copies of a file arrive at a server, what is largest interval of time when no copies of the file arrive?

$O(n \log n)$ solution. Sort the time-stamps. Scan the sorted list in order, identifying the maximum gap between successive time-stamps.

Quadratic Time: $O(n^2)$

Quadratic time. Enumerate all pairs of elements.

Closest pair of points. Given a list of n points in the plane $(x_1, y_1), \dots, (x_n, y_n)$, find the pair that is closest.

$O(n^2)$ solution. Try all pairs of points.

```
min ←  $(x_1 - x_2)^2 + (y_1 - y_2)^2$ 
for i = 1 to n {
  for j = i+1 to n {
    d ←  $(x_i - x_j)^2 + (y_i - y_j)^2$ 
    if (d < min)
      min ← d
  }
}
```

← don't need to
take square roots

Remark. $\Omega(n^2)$ seems inevitable, but this is just an illusion. ← see chapter 5

Cubic Time: $O(n^3)$

Cubic time. Enumerate all triples of elements.

Set disjointness. Given n sets $S_1, \dots, S_n$ each of which is a subset of $1, 2, \dots, n$, is there some pair of these which are disjoint?

$O(n^3)$ solution. For each pairs of sets, determine if they are disjoint.

```
foreach set  $S_i$  {  
    foreach other set  $S_j$  {  
        foreach element  $p$  of  $S_i$  {  
            determine whether  $p$  also belongs to  $S_j$   
        }  
        if (no element of  $S_i$  belongs to  $S_j$ )  
            report that  $S_i$  and  $S_j$  are disjoint  
    }  
}
```

Polynomial Time: $O(n^k)$ Time

Independent set of size k . Given a graph, are there k nodes such that no two are joined by an edge?

k is a constant

$O(n^k)$ solution. Enumerate all subsets of k nodes.

```
foreach subset S of k nodes {  
    check whether S is an independent set  
    if (S is an independent set)  
        report S is an independent set  
    }  
}
```

- Check whether S is an independent set = $O(k^2)$.
- Number of k element subsets = $\binom{n}{k} = \frac{n(n-1)(n-2)\cdots(n-k+1)}{k(k-1)(k-2)\cdots(2)(1)} \leq \frac{n^k}{k!}$
- $O(k^2 n^k / k!) = O(n^k)$.

poly-time for $k=17$,
but not practical

Exponential Time

Independent set. Given a graph, what is maximum size of an independent set?

$O(n^2 2^n)$ solution. Enumerate all subsets.

```
S* ← ∅  
foreach subset S of nodes {  
    check whether S is an independent set  
    if (S is largest independent set seen so far)  
        update S* ← S  
}  
}
```