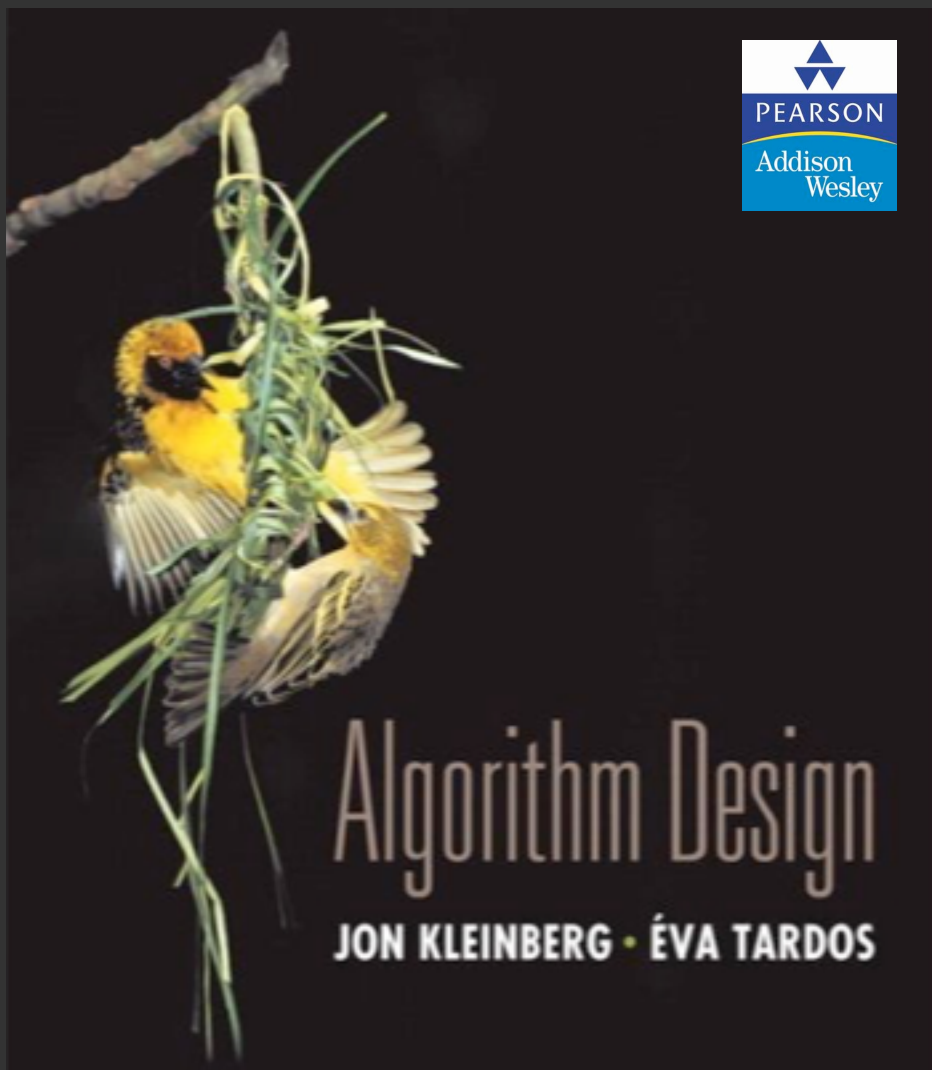


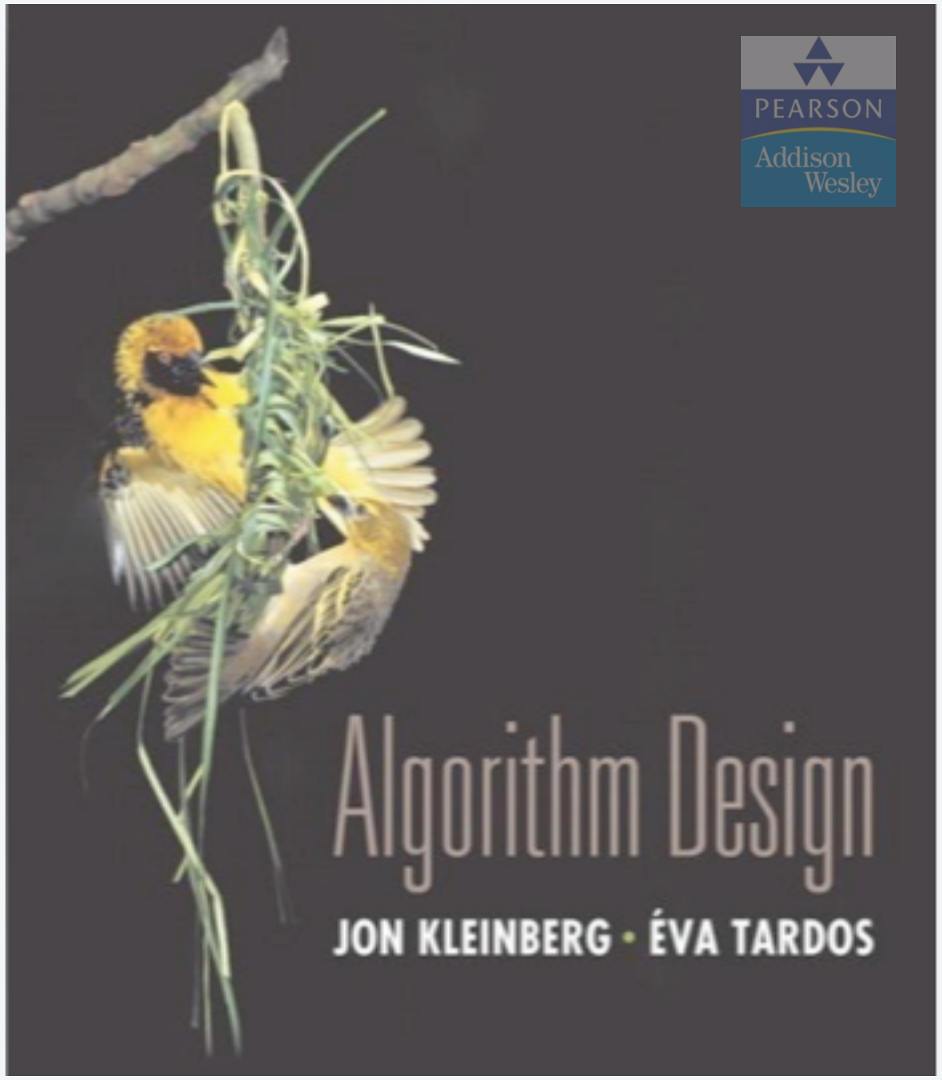
2. Algorithm Analysis

- *computational tractability*
- *asymptotic order of growth*
- *survey of common running times*

Lecture slides by Kevin Wayne

http://www.cs.princeton.edu/~wayne/kleinberg-tardos

Last updated on 8/23/15 1:47 PM



SECTION 2.1

2. ALGORITHM ANALYSIS

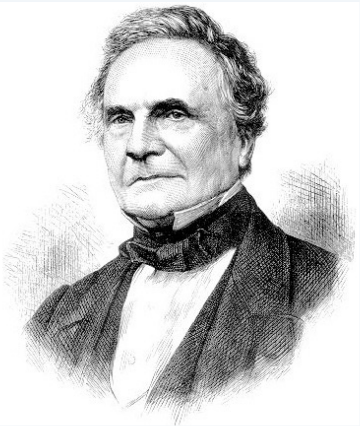
- *computational tractability*
- *asymptotic order of growth*
- *survey of common running times*

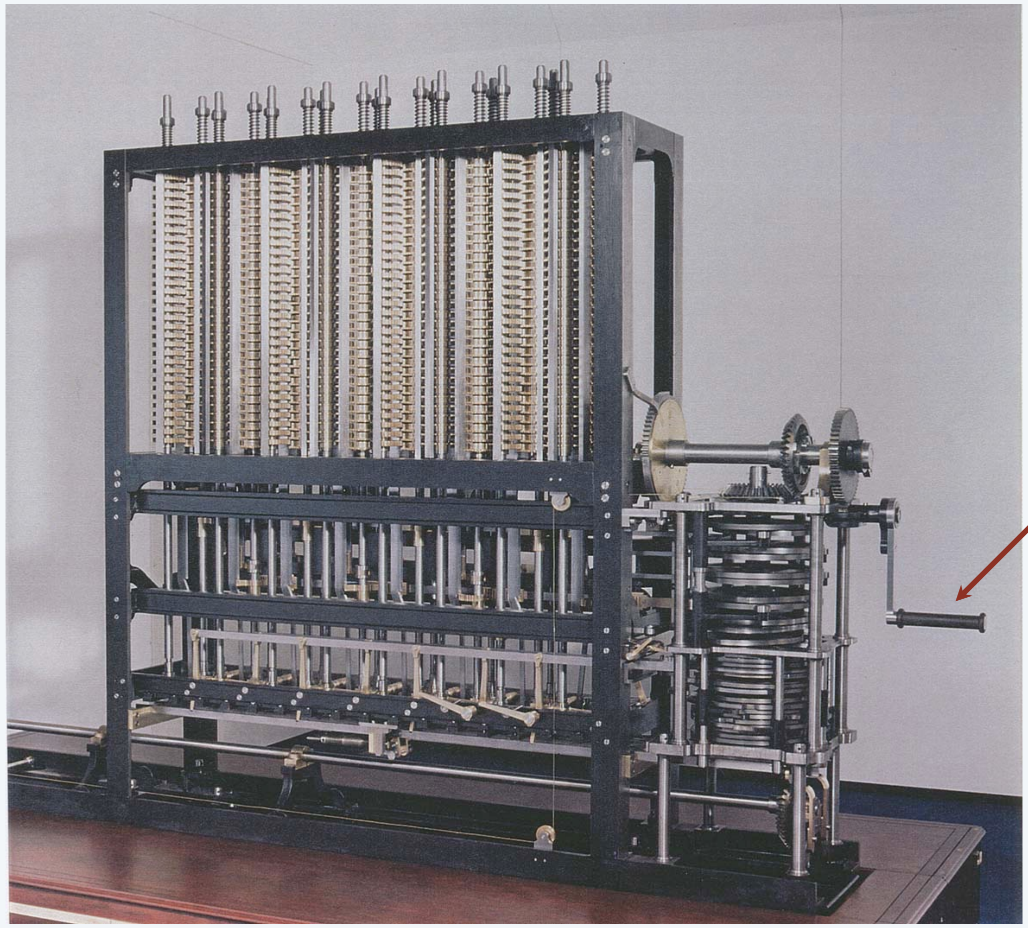
Brute force

"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." - FrancisSullivan

A strikingly modern thought

" 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 (1864)*





how many times do you have to turn the crank?

Analytic Engine

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.

Brute force

Brute force. For many nontrivial 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. When the input size doubles, the algorithm should only slow down by some constant factor C.

There exists constants $c > 0$ and $d > 0$ such that
on every input of size n, its running time is
bounded
by $c\ n^d$ primitive computational 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

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.

Polynomial running time

Def. We say that an algorithm is efficient if has a polynomial running time.

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 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.

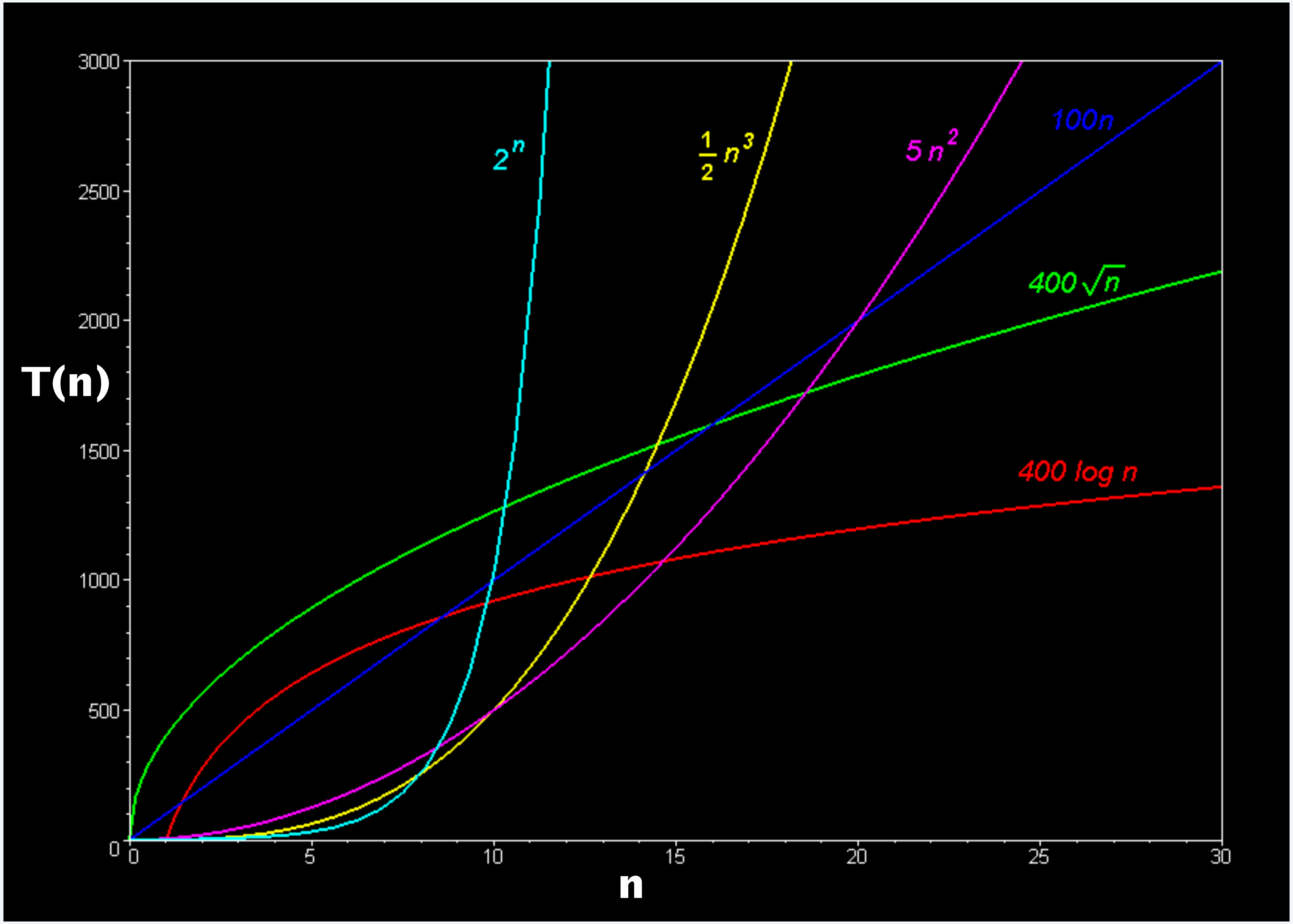
Khachiyan method for LP

- Some exponential-time (or worse) algorithms are widely used because the worst-case instances seem to be rare.

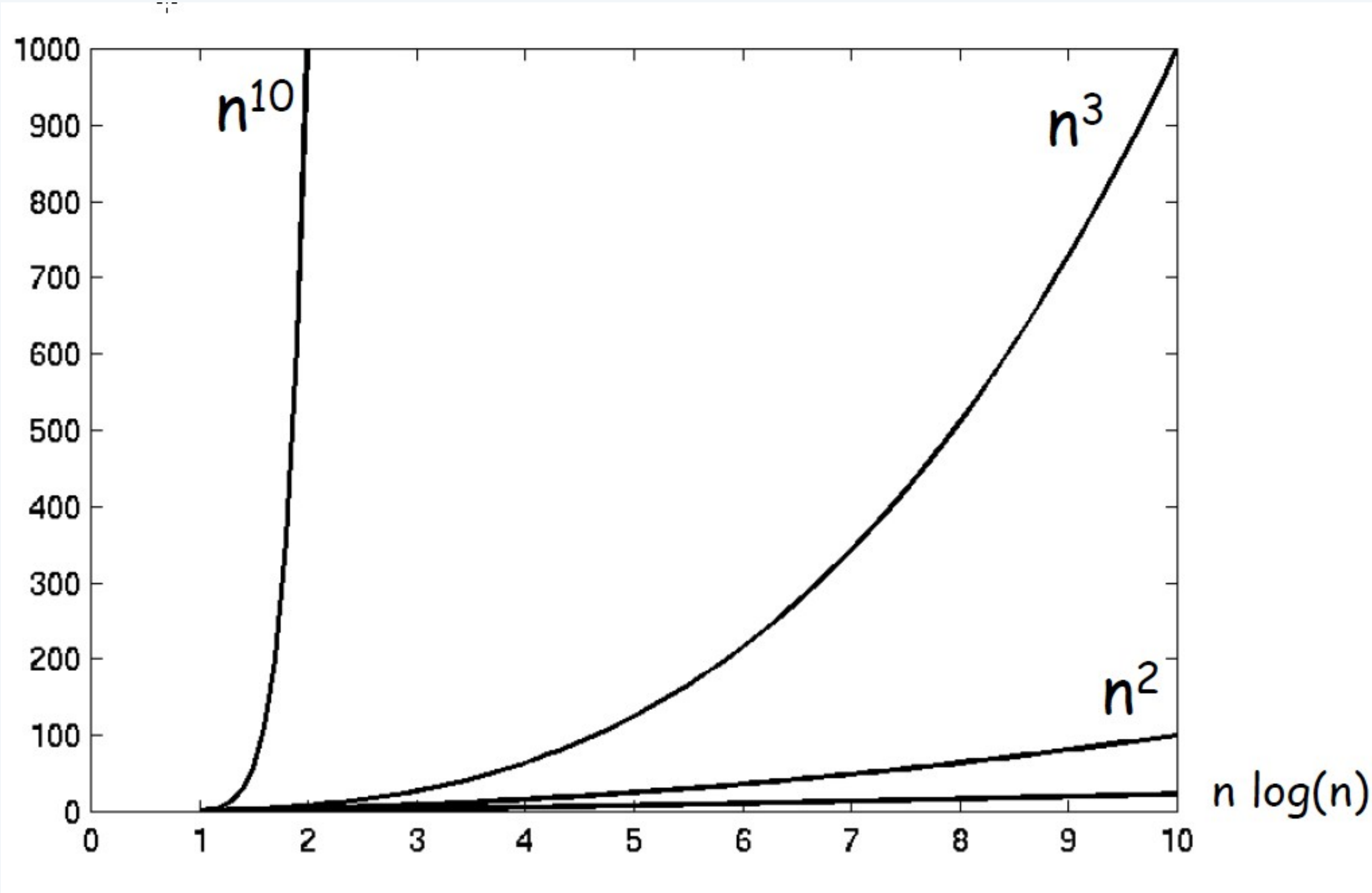
simplex method
Unix grep

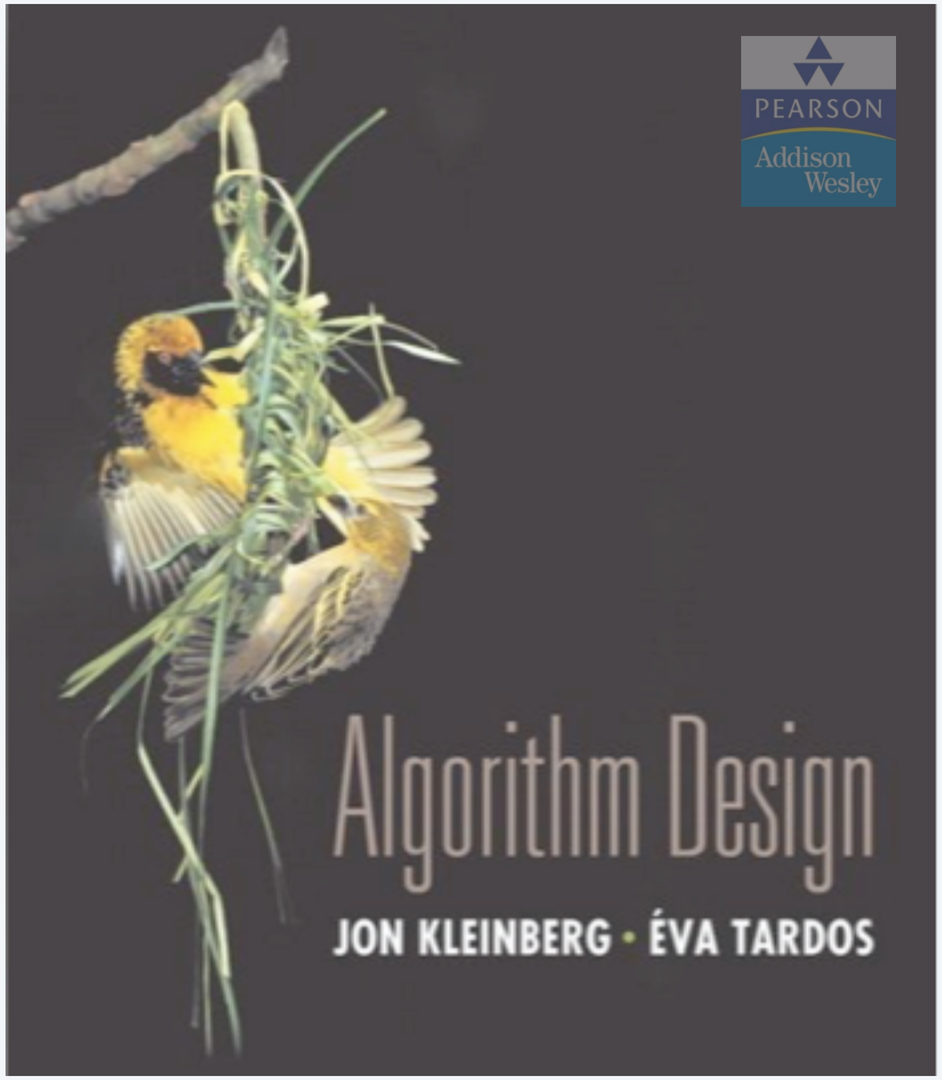
Complexity Graphs

- Linear -- O(*n*)
- Quadratic -- O(n^2)
- Cubic -- O(n^3)
- Logarithmic -- O(log *n*)
- Exponential -- O(2^n)
- Square root -- O(sqrt *n*)



Complexity Graphs





SECTION 2.2

2. ALGORITHM ANALYSIS

- *computational tractability*
- *asymptotic order of growth*
- *survey of common running times*

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 $T(n) \leq c \cdot f(n)$.

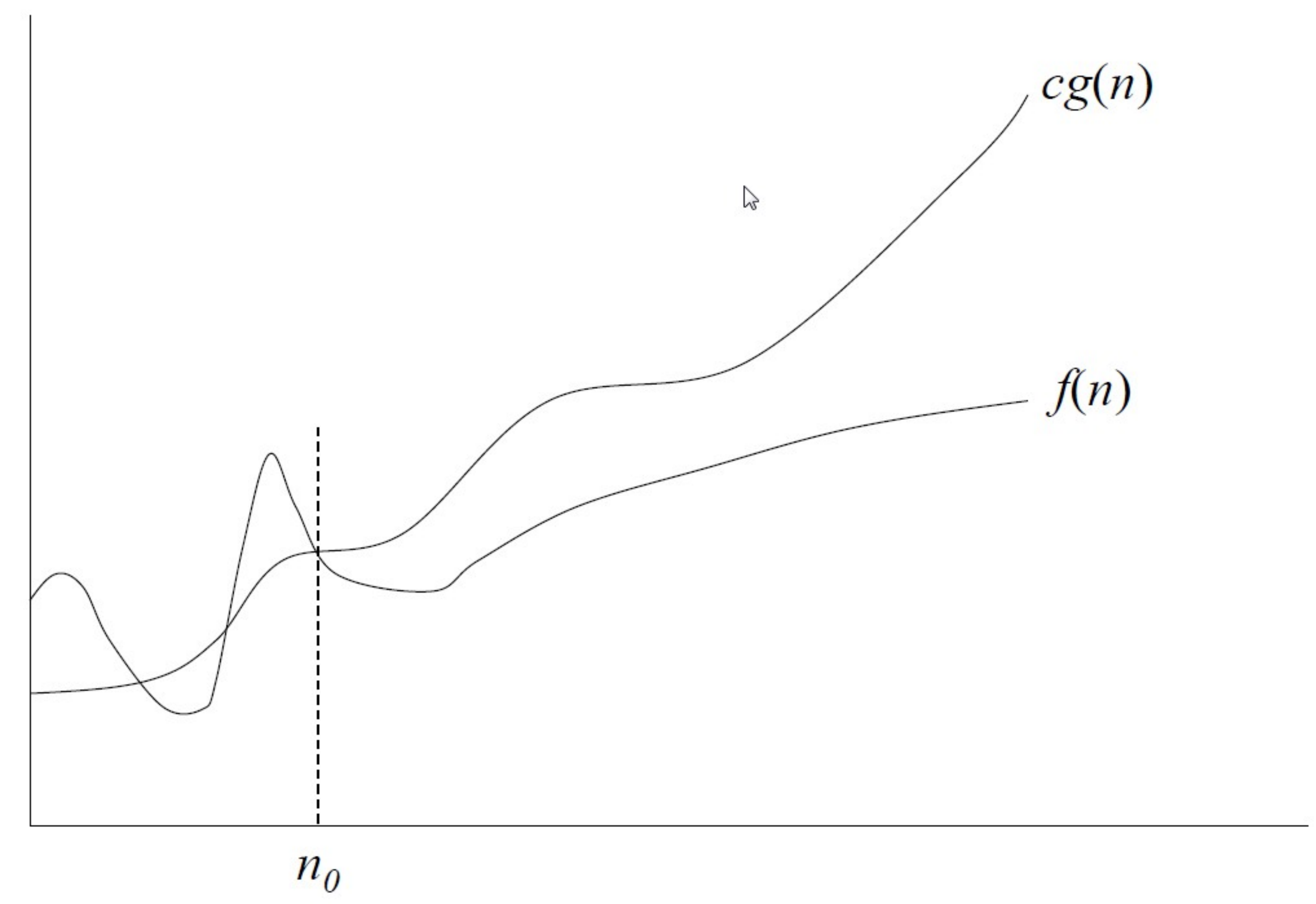
Lower bounds. $T(n)$ is $\Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot f(n)$.

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))

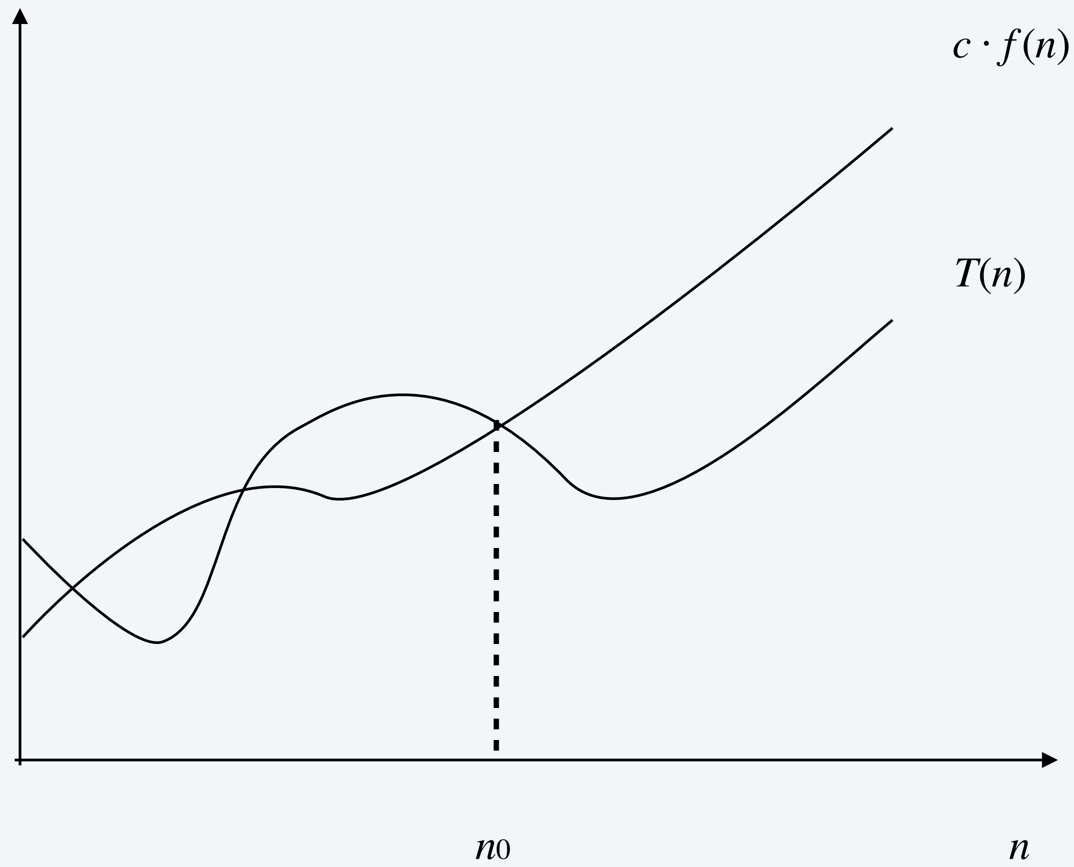


Big-Oh notation

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

Ex. $T(n) = 32n^2 + 17n + 1$.

- $T(n)$ is $O(n^2)$. ⟵ choose $c = 50$, $n_0 = 1$
- $T(n)$ is also $O(n^3)$.
- $T(n)$ is neither $O(n)$ nor $O(n \log n)$.



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

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

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

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

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

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

Big-O (Example)

- 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 $n > 4$

Big-O (Another Example)

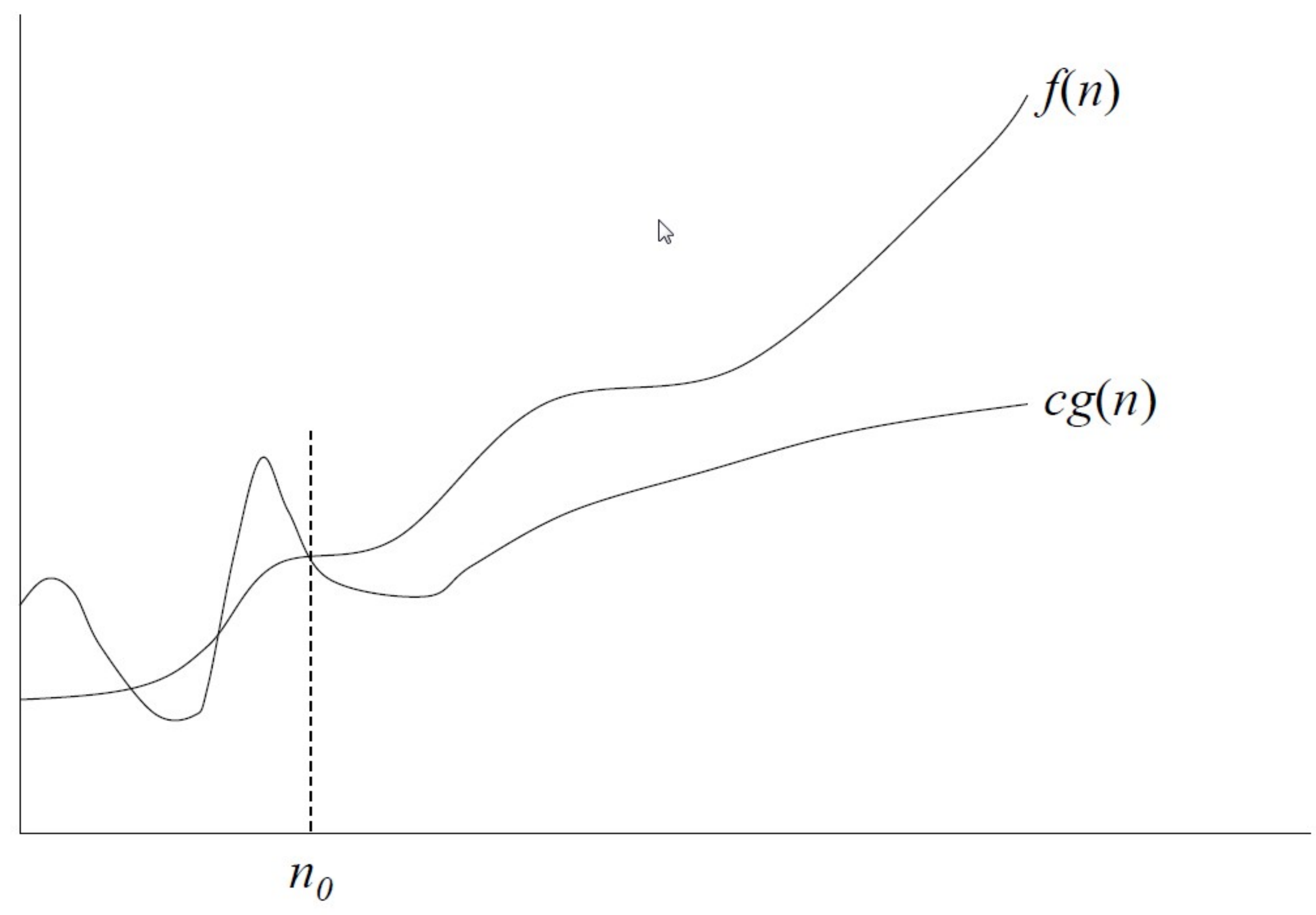
- Show that $2^n + n^2 = O(2^n)$
- Let $c = 2$ and $n_0 = 5$

$$2 \times 2^n > 2^n + n^2$$

$$2^n > n^2 \quad \forall n \geq 5$$

- *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 $\Omega(g(n))$

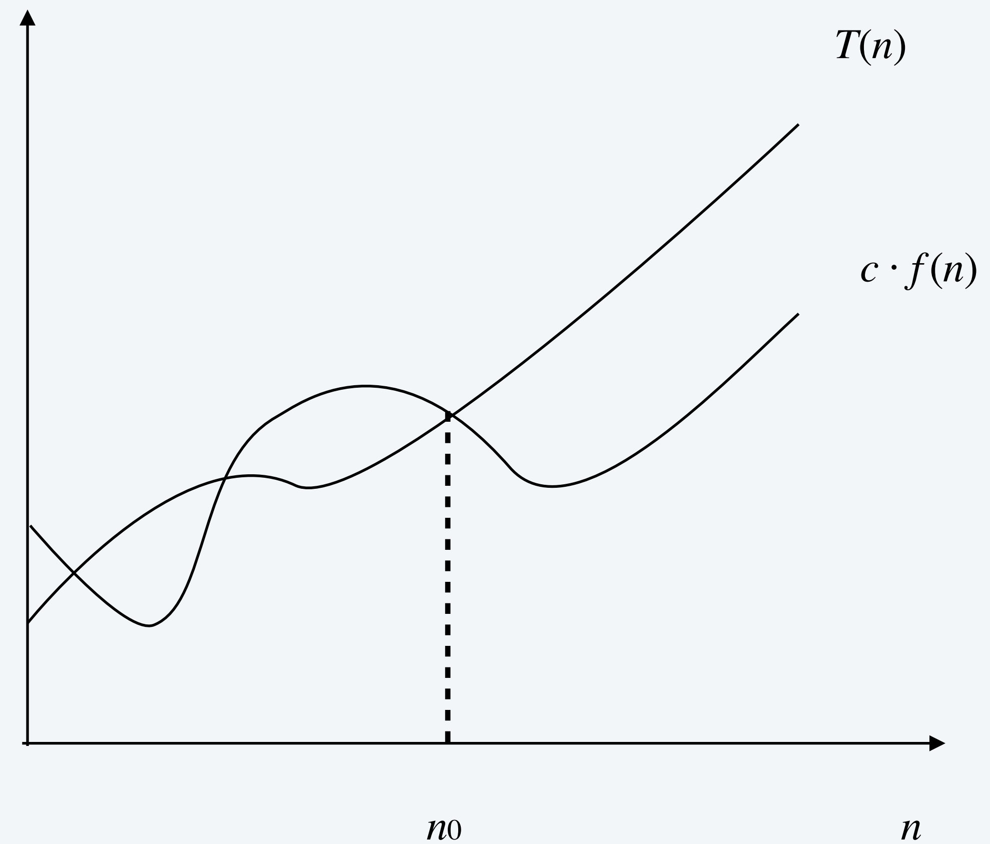


Big-Omega notation

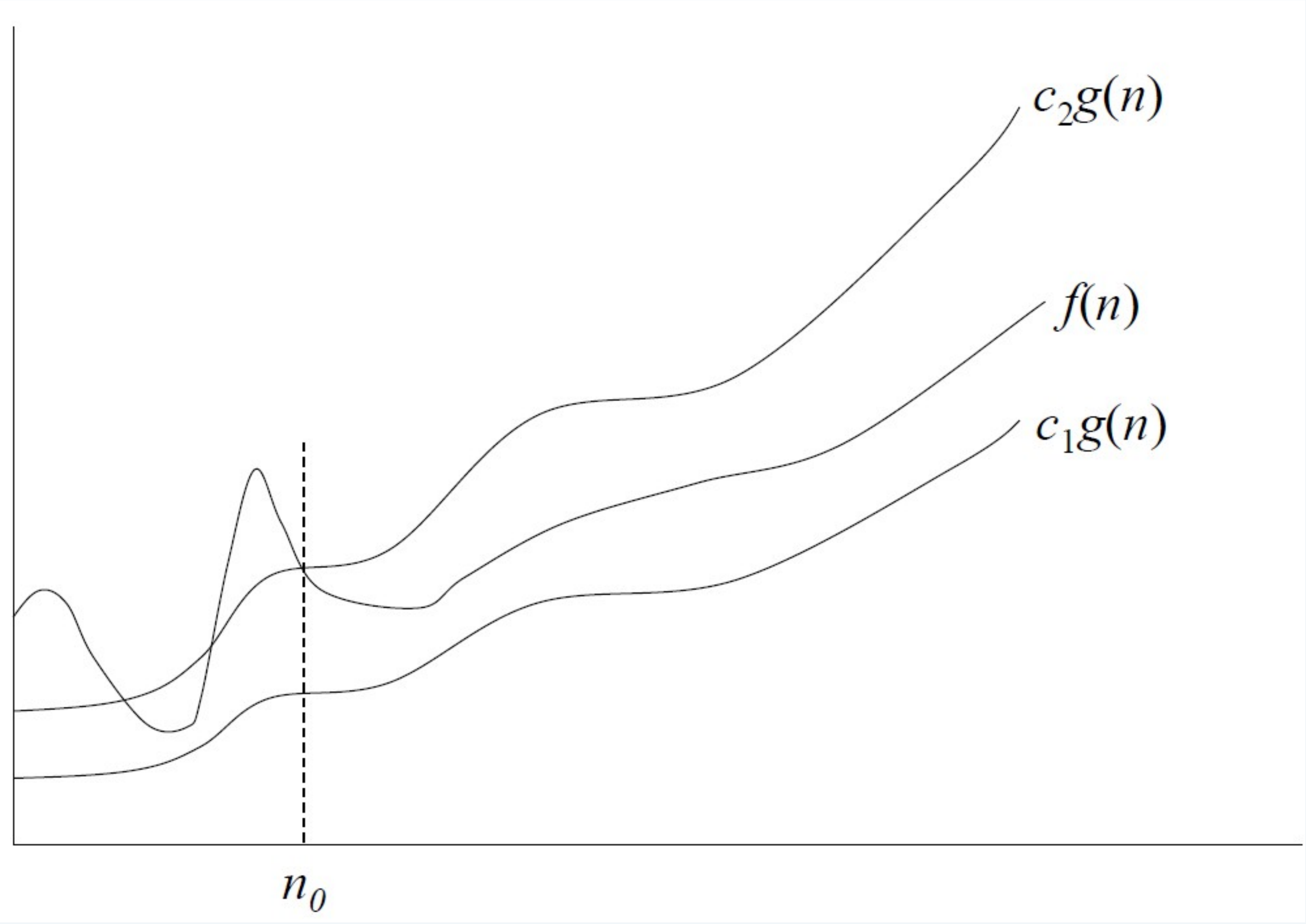
Lower bounds. $T(n)$ is $\Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that $T(n) \geq c \cdot f(n)$ for all $n \geq n_0$.

Ex. $T(n) = 32n^2 + 17n + 1$.

- $T(n)$ is both $\Omega(n^2)$ and $\Omega(n)$. ⟵ choose c = 32, n_0 = 1
- $T(n)$ is neither $\Omega(n^3)$ nor $\Omega(n^3 \log n)$.

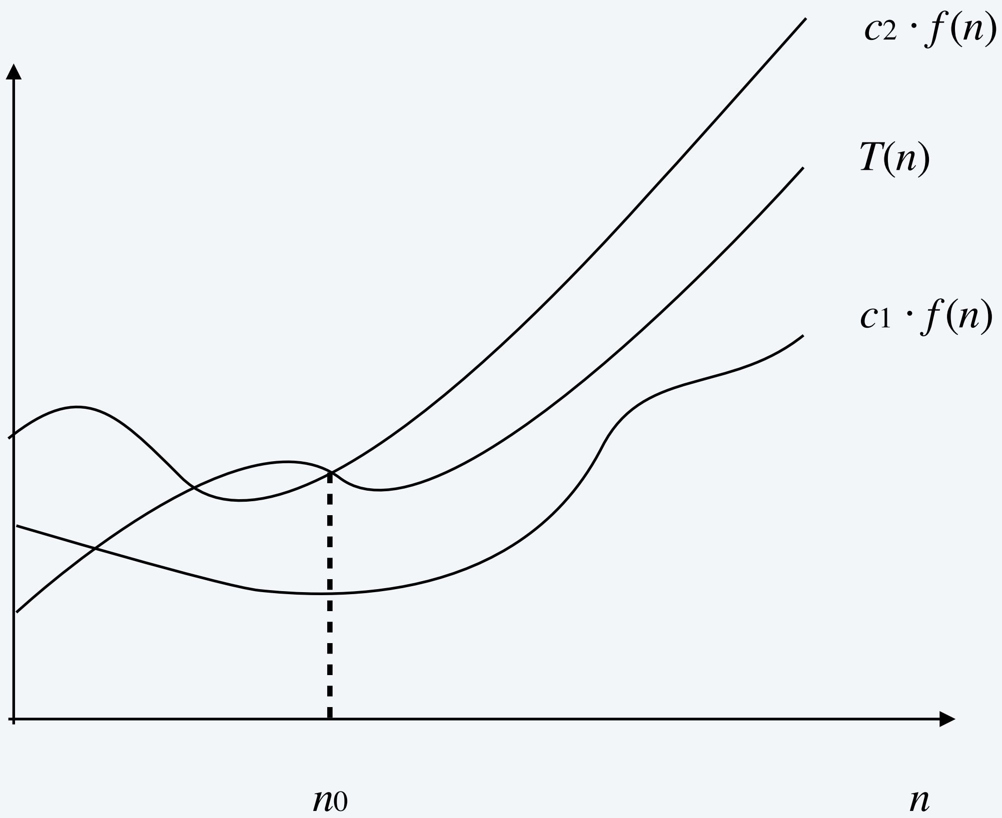


Visualization of $\Theta(g(n))$



Big-Theta notation

Tight bounds. $T(n)$ is $\Theta(f(n))$ if there exist constants $c_1 > 0$, $c_2 > 0$, and $n_0 \geq 0$ such that $c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n)$ for all $n \geq n_0$.



Ex. $T(n) = 32n^2 + 17n + 1$.

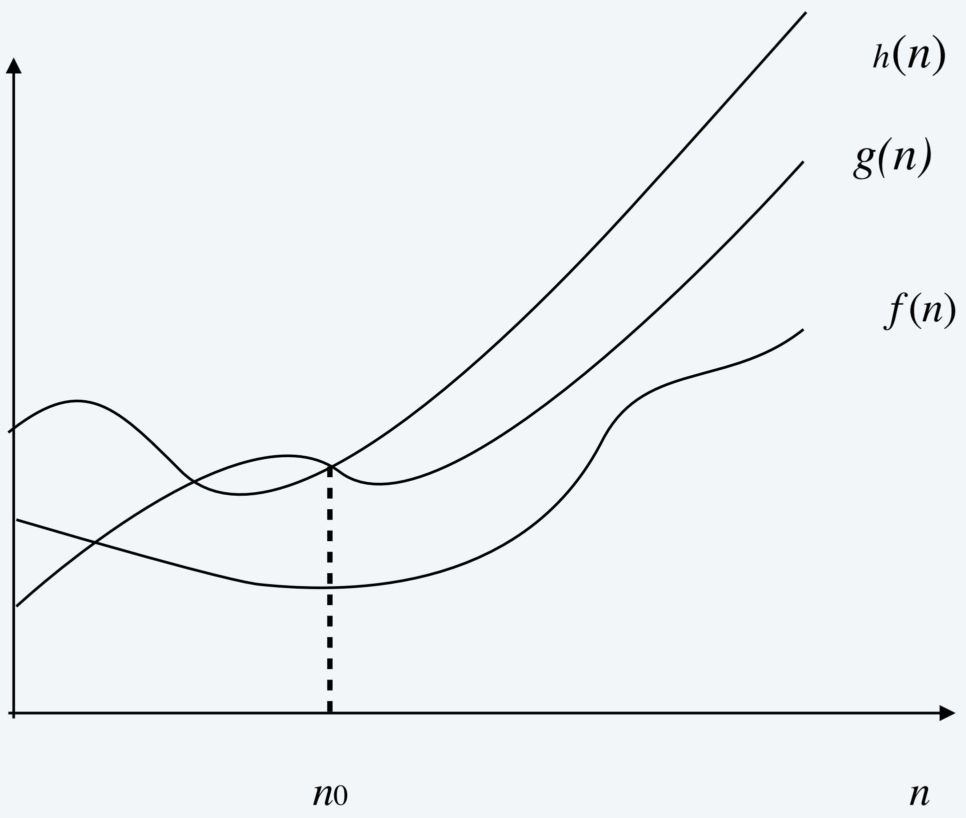
- $T(n)$ is $\Theta(n^2)$. ⟵ choose $c_1 = 32$, $c_2 = 50$, $n_0 = 1$
- $T(n)$ is neither $\Theta(n)$ nor $\Theta(n^3)$.

Big-Theta notation

Tight bounds. $T(n)$ is $\Theta(f(n))$ if there exist constants $c_1 > 0$, $c_2 > 0$, and $n_0 \geq 0$ such that $c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n)$ for all $n \geq n_0$.

Ex. $T(n) = 32n^2 + 17n + 1$.

- $T(n)$ is $\Theta(n^2)$. ⟵ choose $c_1 = 32$, $c_2 = 50$, $n_0 = 1$
- $T(n)$ is neither $\Theta(n)$ nor $\Theta(n^3)$.



More Examples

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

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

Justification

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

$f = O(g)$ $\quad$ $f(n) \le c\, g(n)$ for $n > n_0$

$g = O(h)$ $\quad$ $g(n) \le d\, h(n)$ for $n > n_1$

$$\Longrightarrow$$

$f = O(h)$

because $f(n) \le c\, g(n) \le c\, d\, h(n)$

for $n > \max(n_0, n_1)$

Justification

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

$f = \Omega(g)$ $\quad f(n) >= c\ h(n)$ $\quad$ for $n > n_0$

$g = \Omega(h)$ $\quad g(n) >= d\ h(n)$ $\quad$ for $n > n_1$

$\Longrightarrow$

$f + g = \Omega(h)$

because $f(n) + g(n) >= \min(c,d)\ h(n)$

for $n > \max(n_0, n_1)$

Asymptotic Bounds for Some Common Functions

Polynomials. $a_0 + a_1 n + \ldots + a_d n^d$ is $\Theta(n^d)$ if $a_d > 0$.

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 $a, b > 0$.

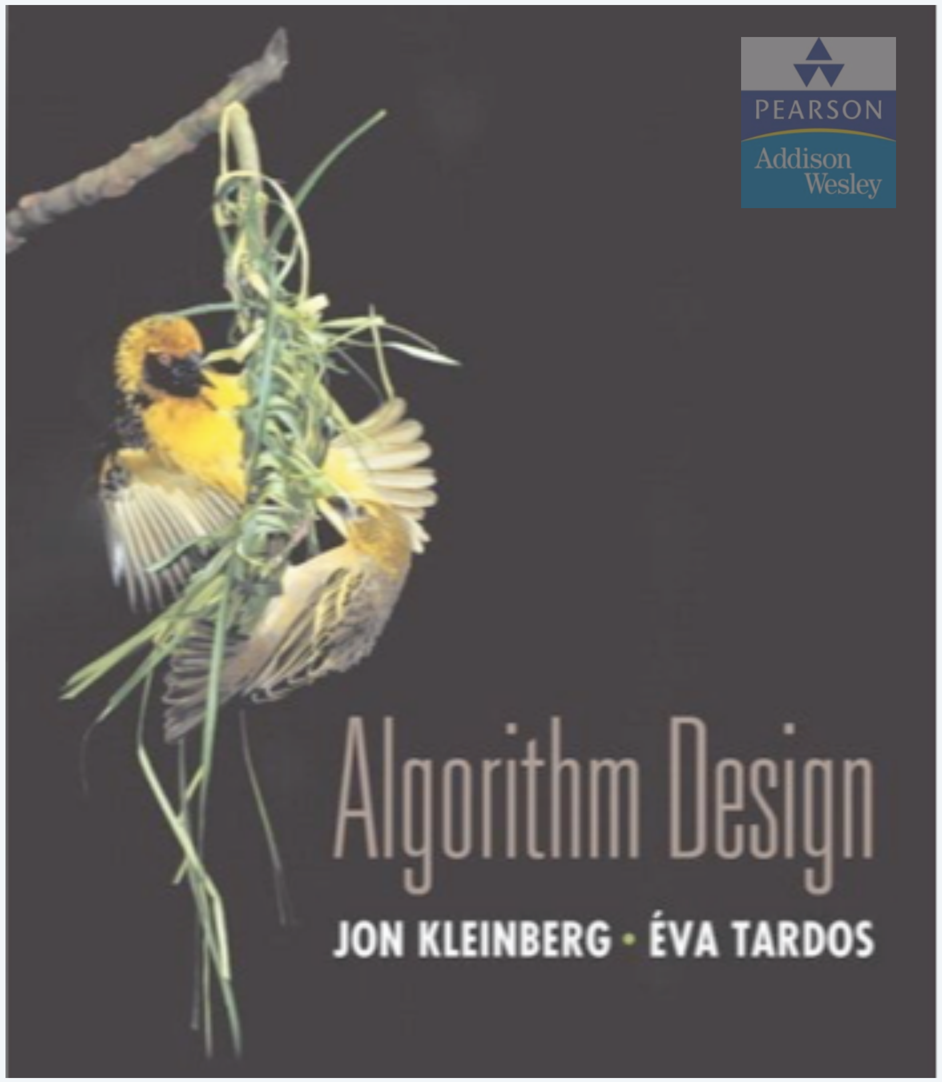
can avoid specifying the base

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



SECTION 2.4

2. ALGORITHM ANALYSIS

- *computational tractability*
- *asymptotic order of growth*
- *survey of common running times*

Linear time: O(n)

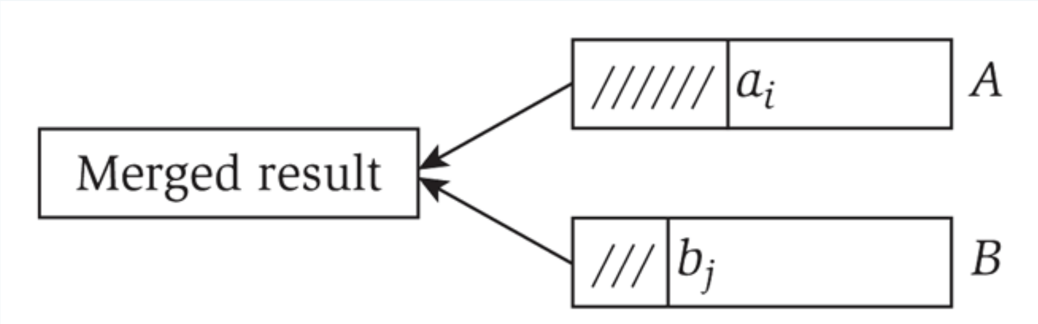
Linear time. Running time is proportional to input size.

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

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

Linear time: O(n)

Merge. Combine two sorted lists $A = a_1, a_2, \ldots, a_n$ with $B = b_1, b_2, \ldots, b_n$ 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 compare, the length of output list increases by 1.

Linearithmic time: O(n log n)

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

also referred to as linearithmic time

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

Largest empty interval. Given n time-stamps $x_1, \ldots, x_n$ on which copies of a file arrive at a server, what is largest interval when no copies of 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(n²)

Ex. Enumerate all pairs of elements.

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

O(n²) solution. Try all pairs of points.

```
min <- (x1 - x2)^2 + (y1 - y2)^2
for i = 1 to n {
   for j = i+1 to n {
      d <- (xi - xj)^2 + (yi - yj)^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 $S_1, \ldots, S_n$ each of which is a subset of $1, 2, \ldots, n$, is there some pair of these which are disjoint?

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

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

Polynomial time: O(n^k)

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 in an independent set
   if (S is an independent set)
      report S is an independent set
   }
}
```

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

poly-time for k=17,
but not practical

Exponential time

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

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

```
S* <- φ
foreach subset S of nodes {
   check whether S in an independent set
   if (S is largest independent set seen so far)
      update S* <- S
   }
}
```