NPTEL MOOC,JAN-FEB 2015
Week 1, Module 7

DESIGN AND ANALYSIS OF ALGORITHMS

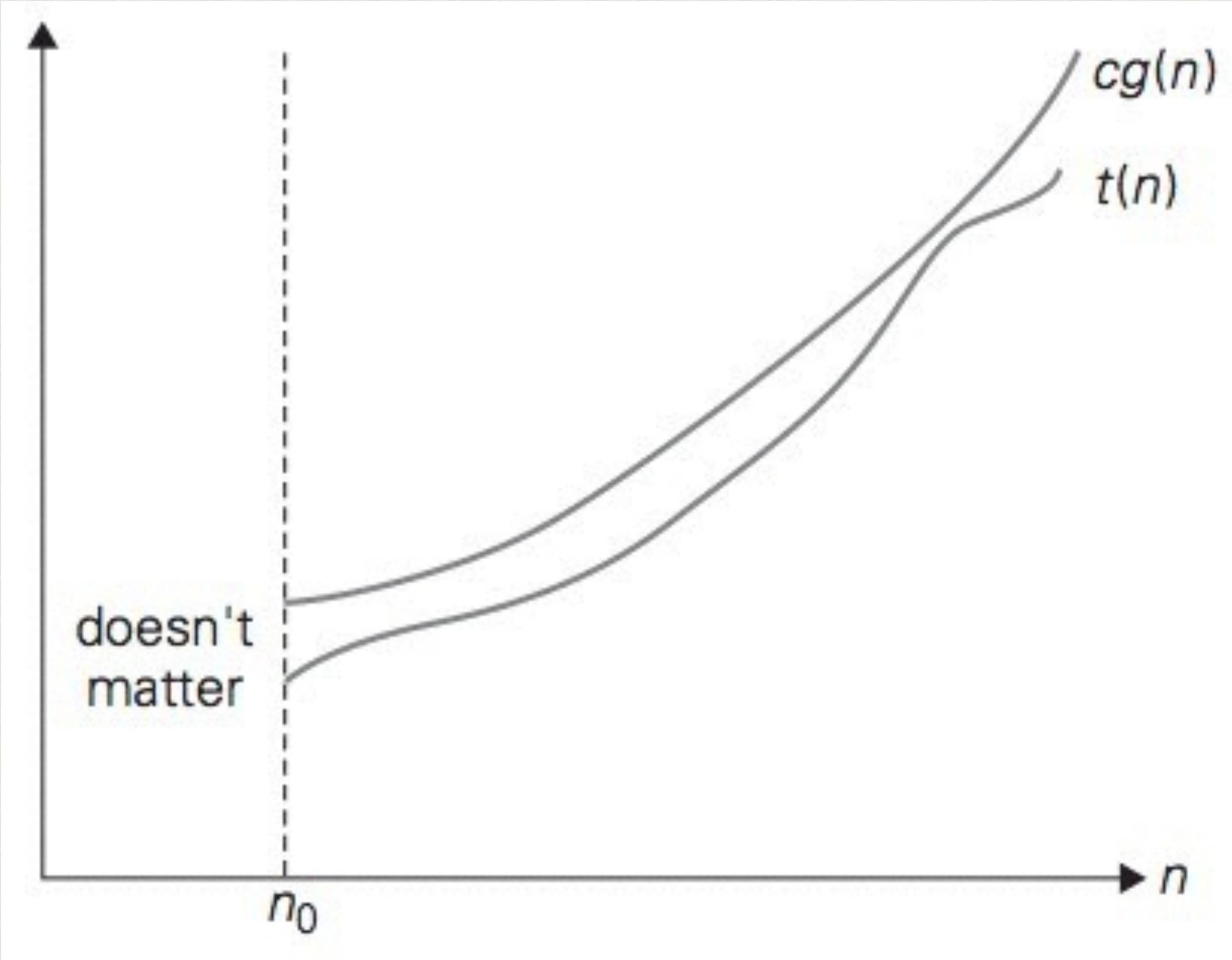
MADHAVAN MUKUND, CHENNAI MATHEMATICAL INSTITUTE
http://www.cmi.ac.in/~madhavan

Comparing time efficiency

- We measure time efficiency only upto an order of magnitude
  - Ignore constants
- How do we compare functions with respect to orders of magnitude?

Upper bounds, “big O”

- $t(n)$ is said to be $O(g(n))$ if we can find suitable constants c and n_0 so that $cg(n)$ is an upper bound for $t(n)$ for n beyond n_0
  - $t(n) \leq cg(n)$ for every $n \geq n_0$



Examples: Big O

- 100n + 5 is $O(n^2)$
  - 100n + 5
  - $\leq$ 100n + n, for $n \geq 5$
  - = 101n $\leq 101n^2$, so $n_0 = 5$, c = 101
- Alternatively
  - 100n + 5
  - $\leq$ 100n + 5n, for $n \geq 1$
  - = 105n $\leq 105n^2$, so $n_0 = 1$, c = 105
- n_0 and c are not unique!
- Of course, by the same argument, 100n+5 is also O(n)

Examples: Big O

- $100n^2 + 20n + 5$ is $O(n^2)$
  - $100n^2 + 20n + 5$
  - $\leq 100n^2 + 20n^2 + 5n^2$, for $n \geq 1$
  - $\leq 125n^2$
  - $n_0 = 1$, $c = 125$
- What matters is the highest term
  - $20n + 5$ dominated by $100n^2$

Examples: Big O

- n^3 is not $O(n^2)$
  - No matter what c we choose, cn^2 will be dominated by n^3 for $n \geq c$

Useful properties

- If
  - $f_1(n)$ is $O(g_1(n))$
  - $f_2(n)$ is $O(g_2(n))$
- then $f_1(n) + f_2(n)$ is $O(\max(g_1(n), g_2(n)))$

Proof

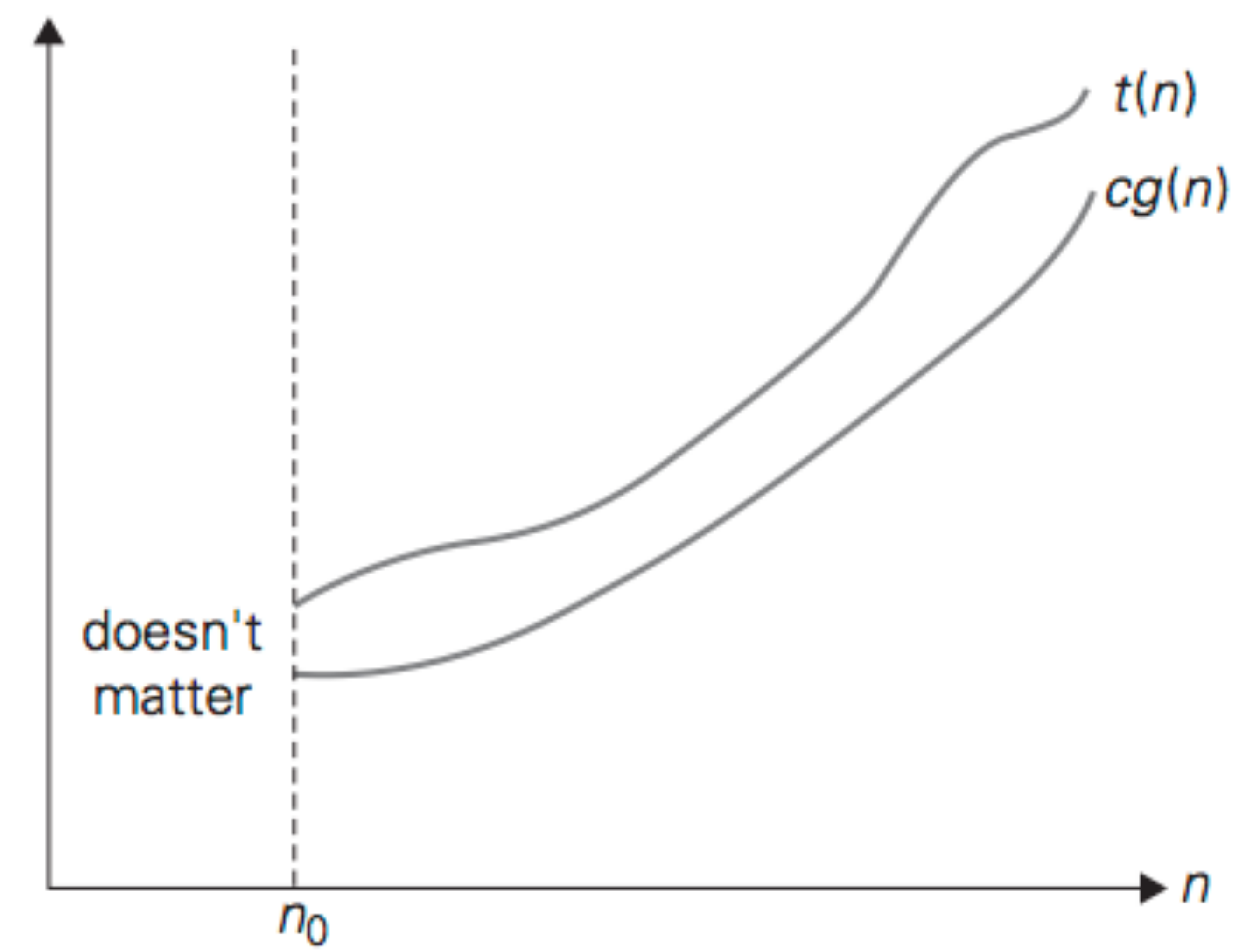
- $f_1(n) \leq c_1 g_1(n)$ for all $n > n_1$
- $f_2(n) \leq c_2 g_2(n)$ for all $n > n_2$

Why is this important?

- Algorithm has two phases
  - Phase A takes time $O(g_A(n))$
  - Phase B takes time $O(g_B(n))$
- Algorithm as a whole takes time
  - $\max(O(g_A(n)),O(g_B(n)))$
- For an algorithm with many phases, least efficient phase is an upper bound for the whole algorithm

Lower bounds, Ω (omega)

- t(n) is said to be Ω(g(n)) if we can find suitable constants c and n_0 so that cg(n) is an lower bound for t(n) for n beyond n_0
  - $t(n) \geq cg(n)$ for every $n \geq n_0$

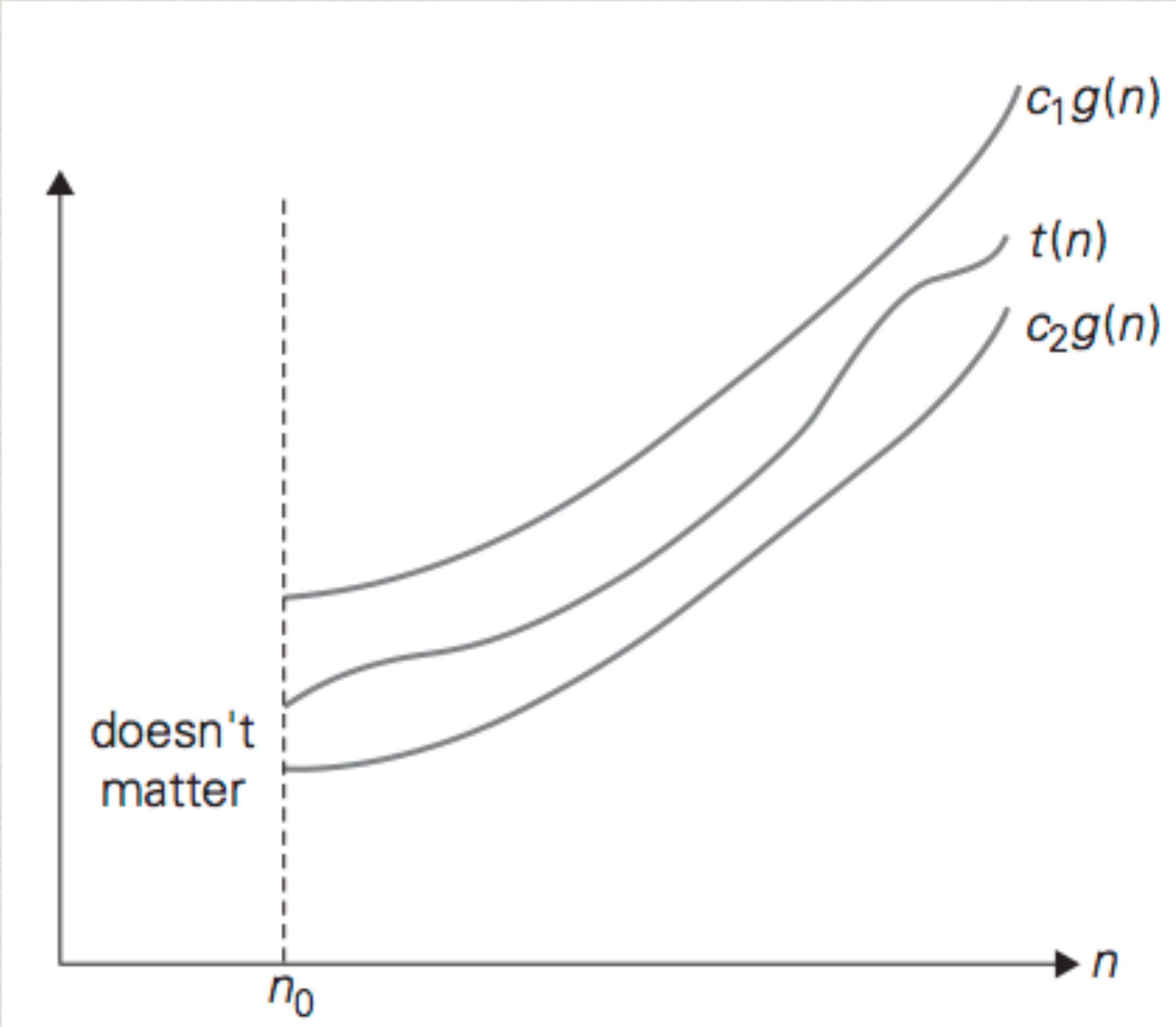


Lower bounds

- n^3 is $\Omega(n^2)$
  - $n^3 \geq n^2$ for all n
  - $n_0 = 0$ and $c = 1$
- Typically we establish lower bounds for problems as a whole, not for individual algorithms
  - Sorting requires $\Omega(n \log n)$ comparisons, no matter how clever the algorithm is

Tight bounds, Θ (theta)

- t(n) is Θ(g(n)) if it is both O(g(n)) and Ω(g(n))
- Find suitable constants c_1, c_2, and n_0 so that
  - $c_2 g(n) \le t(n) \le c_1 g(n)$ for every $n \ge n_0$



Tight bounds

- $n(n-1)/2$ is $\Theta(n^2)$
  - Upper bound

    $n(n-1)/2 = n^2/2 - n/2 \leq n^2/2$, for $n \geq 0$
  - Lower bound

    $n(n-1)/2 = n^2/2 - n/2 \geq n^2/2 - (n/2 \times n/2) \geq n^2/4$, for $n \geq 2$
- Choose $n_0 = \max(0,2) = 2$, $c_1 = 1/2$ and $c_2 = 1/4$

Summary

- f(n) = O(g(n)) means g(n) is an upper bound for f(n)
  - Useful to describe limit of worst case running time for an algorithm
- f(n) = Ω(g(n)) means g(n) is a lower bound for f(n)
  - Typically used for classes of problems, not individual algorithms
- f(n) = Θ(g(n)): matching upper and lower bounds
  - Best possible algorithm has been found