CSE 332
Data Structures & Parallelism

Algorithm Analysis 2

Melissa Winstanley
Spring 2024

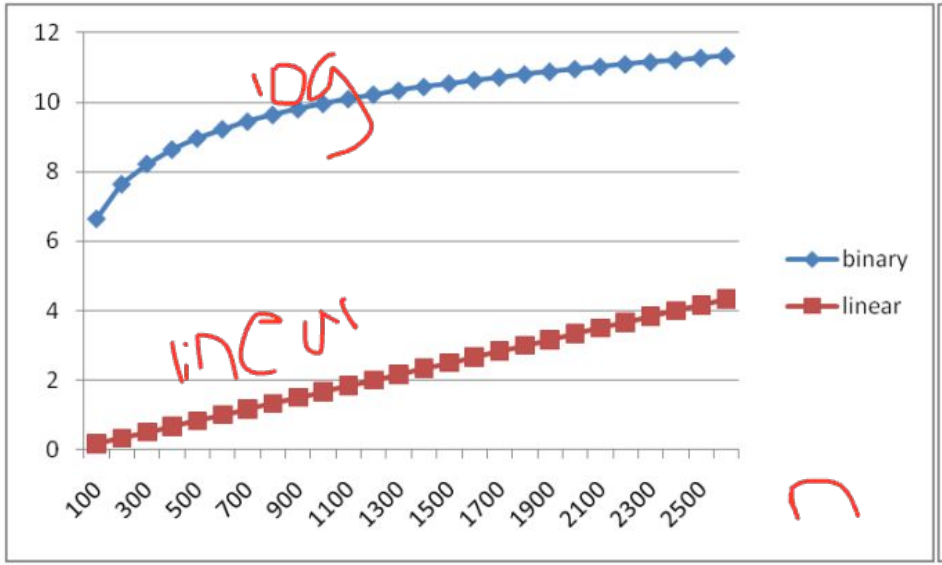
Today - Algorithm Analysis

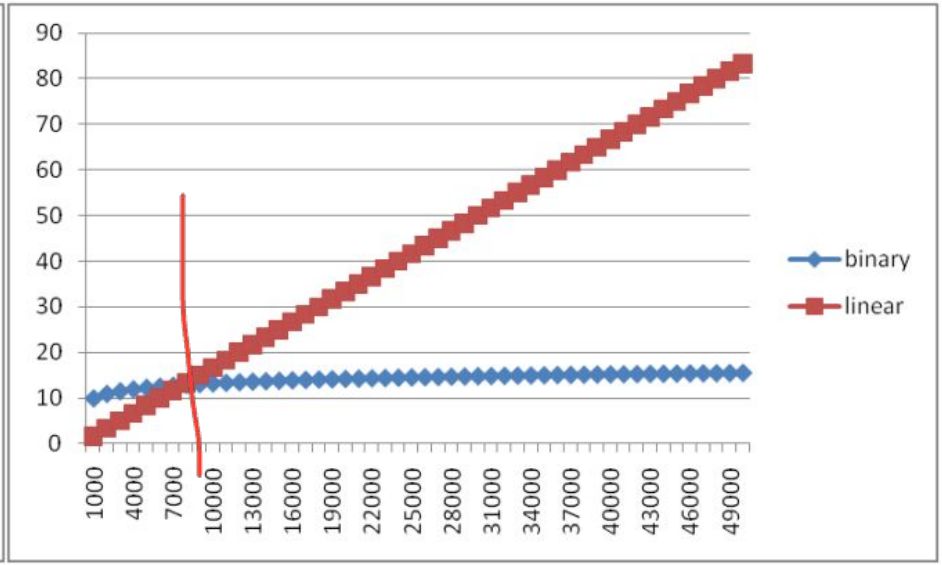
- What do we care about?
- How to compare two algorithms
- Analyzing code
  - How to count different code constructs
  - Best Case vs Worst Case
  - **Ignoring Constant Factors**
- Asymptotic Analysis
- Big-Oh Definition

Ignoring constant factors

- So binary search is O(log n) and linear is O(n)
  - But which will actually be faster?
  - Depending on **constant factors** and **size of n**; in a particular situation, linear search could be faster….
- Could depend on constant factors
  - How *many* assignments, additions, etc. for each n
- And could depend on size of n – what if n is small?
- **But** there exists some n_0 such that for all $n > n_0$ binary search "wins"
- Let's look at a couple plots to get some intuition…

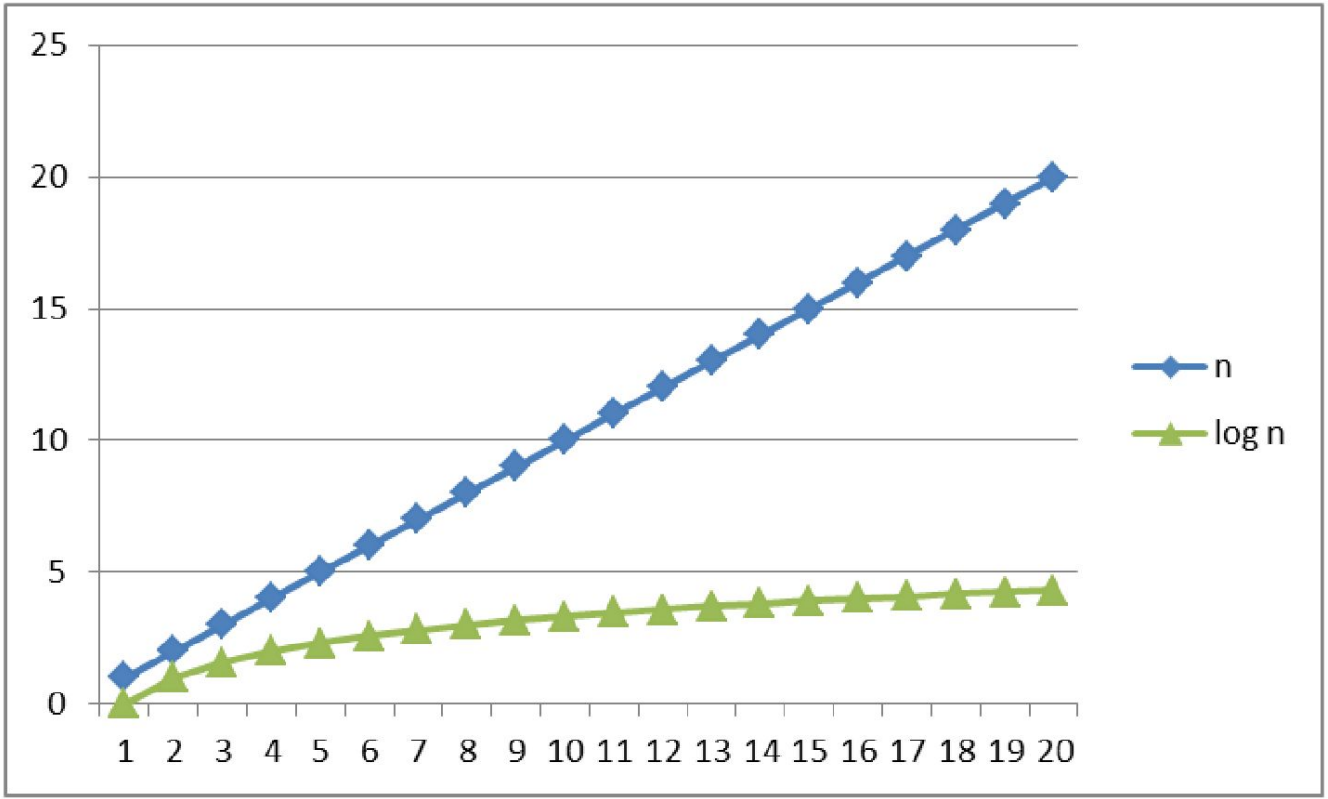
Linear search vs Binary search



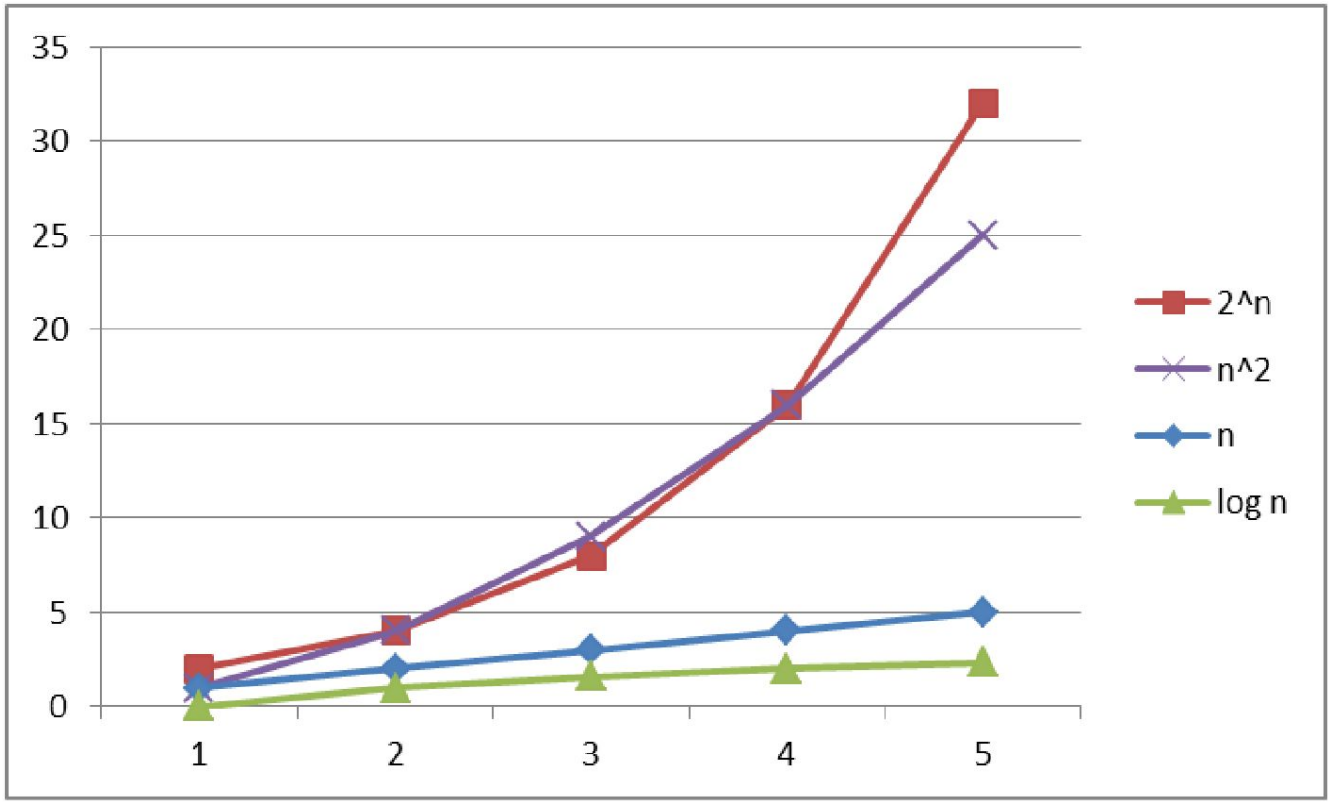


Let’s give linear search a boost (n / 600)

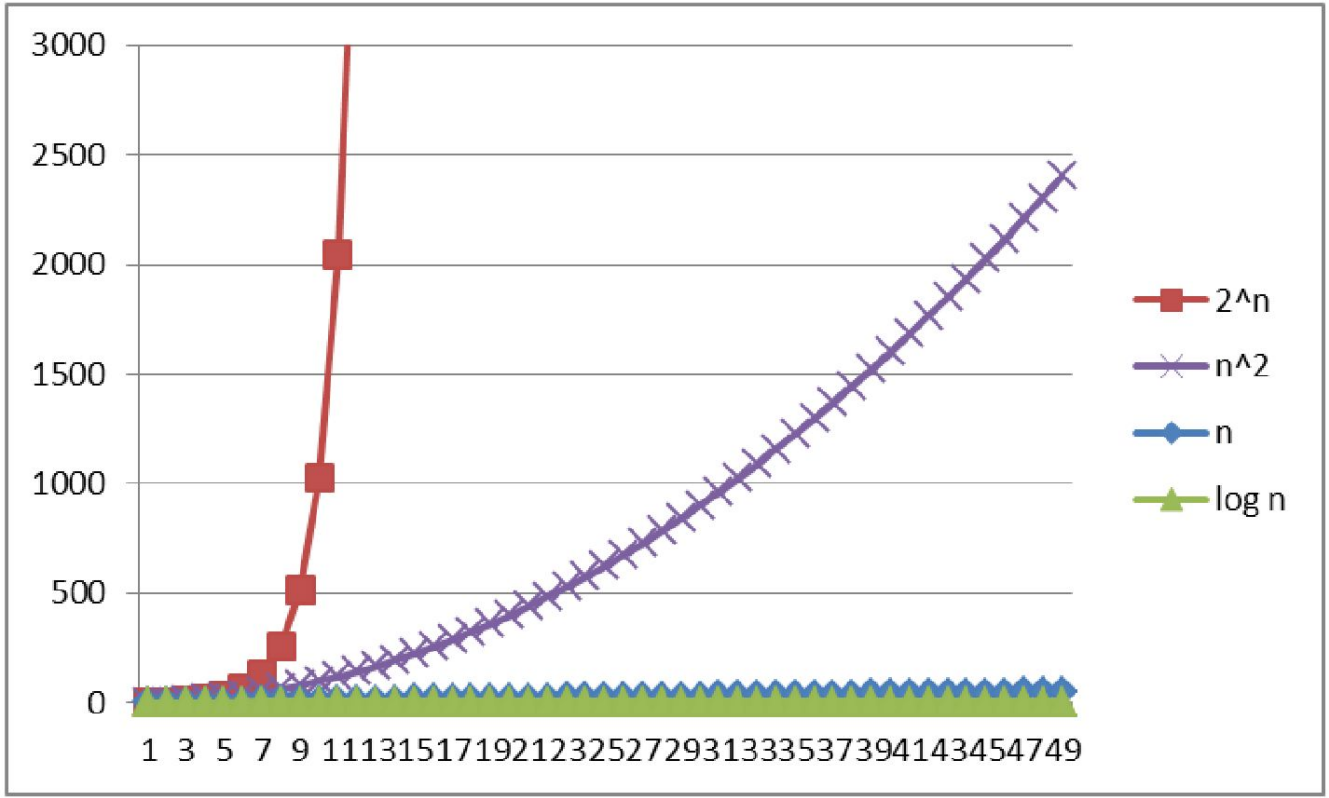
Logarithms and Exponents



Logarithms and Exponents



Logarithms and Exponents



Today - Algorithm Analysis

- What do we care about?
- How to compare two algorithms
- Analyzing code
- Asymptotic Analysis
- Big-Oh Definition

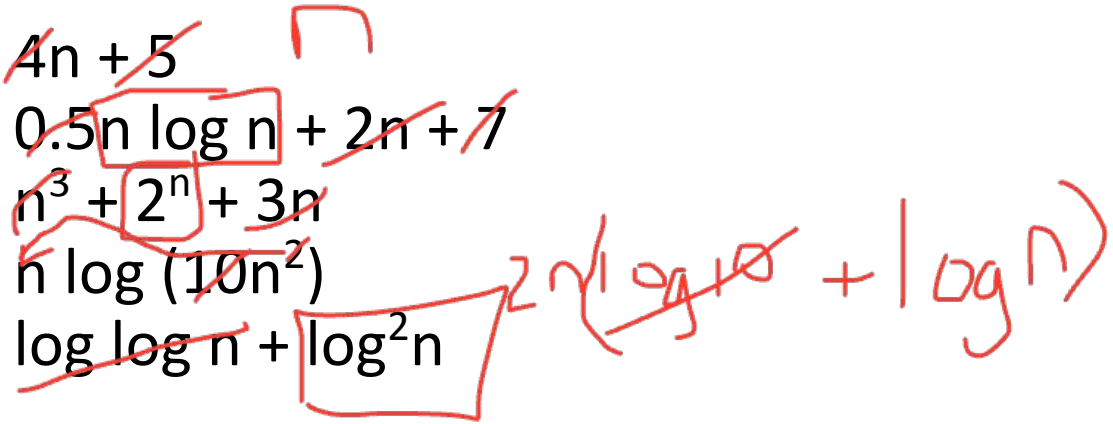
Asymptotic notation

About to show formal definition, which amounts to saying:

1. Eliminate low-order terms
2. Eliminate coefficients

Examples:

- $4n + 5$
- $0.5n \log n + 2n + 7$
- $n^3 + 2^n + 3n$
- $n \log (10n^2)$
- $\log \log n + \log^2 n$



Review: Properties of logarithms

- **$\log(A*B) = \log A + \log B$**
  - So **$\log(N^k) = k \log N$**
- **$\log(A/B) = \log A - \log B$**
- **$X = \log_2 2^X$**
- **log(log x)** is written **log log x**
  - Grows as slowly as **$2^{2^{\wedge}y}$** grows fast
  - Ex: **$\log_2 \log_2 4billion \sim \log_2 \log_2 2^{32} = \log_2 32 = 5$**
- **(log x)(log x)** is written **$\log^2 x$**
  - It is greater than **log x** for all **x > 2**

Today - Algorithm Analysis

- What do we care about?
- How to compare two algorithms
- Analyzing code
- Asymptotic Analysis
- Big-Oh Definition

Big-Oh relates to functions

We use O on a function f(n) (for example n^2) to mean *the set of functions with asymptotic behavior* ***less than or equal to*** *f(n)*

So $(3n^2+17)$ **is in** $O(n^2)$

- $3n^2+17$ and n^2 have the same **asymptotic behavior**

Confusingly, we also say/write:

- $(3n^2+17)$ **is** $O(n^2)$
- $(3n^2+17)$ **=** $O(n^2)$

But we would never say $O(n^2) = (3n^2+17)$

Formally Big-Oh

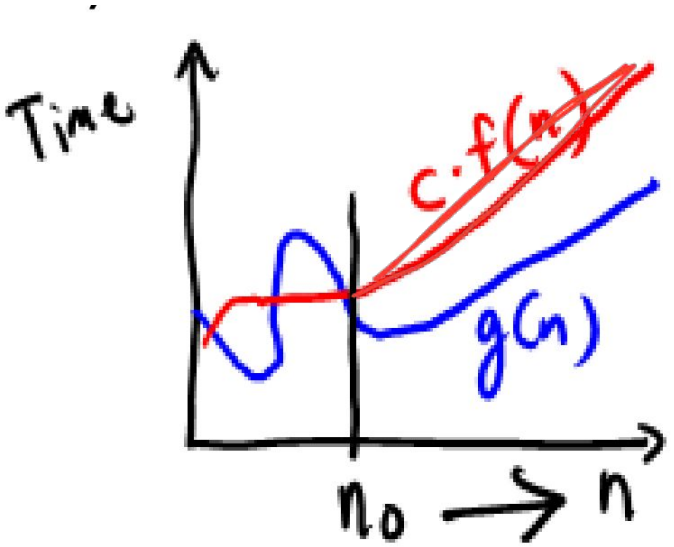
Definition: g(n) is in O(**f(n)**) iff there exist positive constants **c** and n_0 such that

$g(n) \leq c\, f(n)$ for all $n \geq n_0$

Note: $n_0 \geq 1$ (and a natural number) and $c > 0$

Or

$g(n) \in O(f(n))$



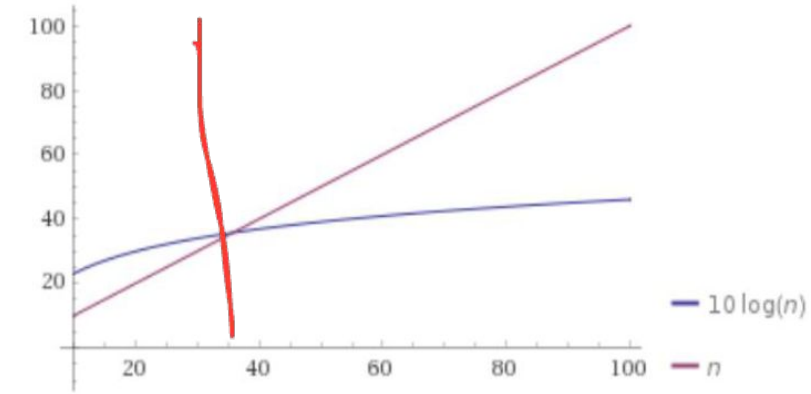
Why n_0? Why c?

Definition: g(n) ∈ O(**f(n)**) iff there exist positive constants **c** and n_0 such that

g(n) ≤ **c f(n)** for all $n \geq n_0$

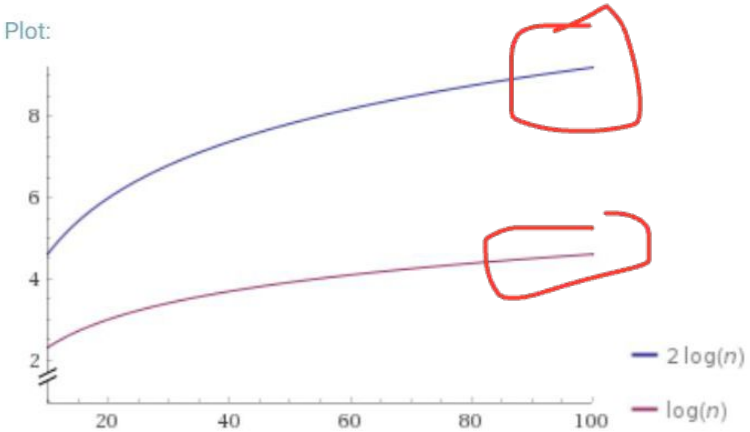
Why n_0?

Plot:

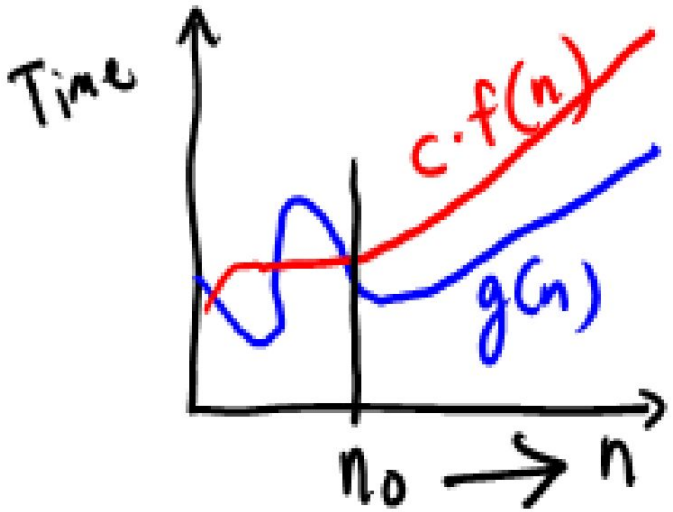


Why c?

Plot:



Formally Big-Oh



Definition: g(n) ∈ O(**f(n)**) iff there exist positive constants **c** and n_0 such that

$g(n) \leq c\, f(n)$ for all $n \geq n_0$

To show g(n) ∈ O(**f(n)**), pick a **c** large enough to "cover the constant factors" and n_0 large enough to "cover the lower-order terms"

Example: Let g(n) = 3n + 4 and **f(n)** = n

$c = 4$ and $n_0 = 5$ is one possibility

This is "less than or equal to"

- So 3n + 4 is also $O(n^5)$ and $O(2^n)$ etc.

$3n + 4 \leq 4n$

$4 \leq n$

Examples

True or false?

1. 4+3n is O(n)
2. n+2logn is O(logn)
3. logn+2 is O(1)
4. n^{50} is O(1.1^n)

Notes:

- Do NOT ignore constants that are not multipliers:
  - n^3 is O(n^2) : FALSE
  - 3^n is O(2^n) : FALSE
- When in doubt, refer to the definition

Big Oh: Common Categories

From fastest to slowest

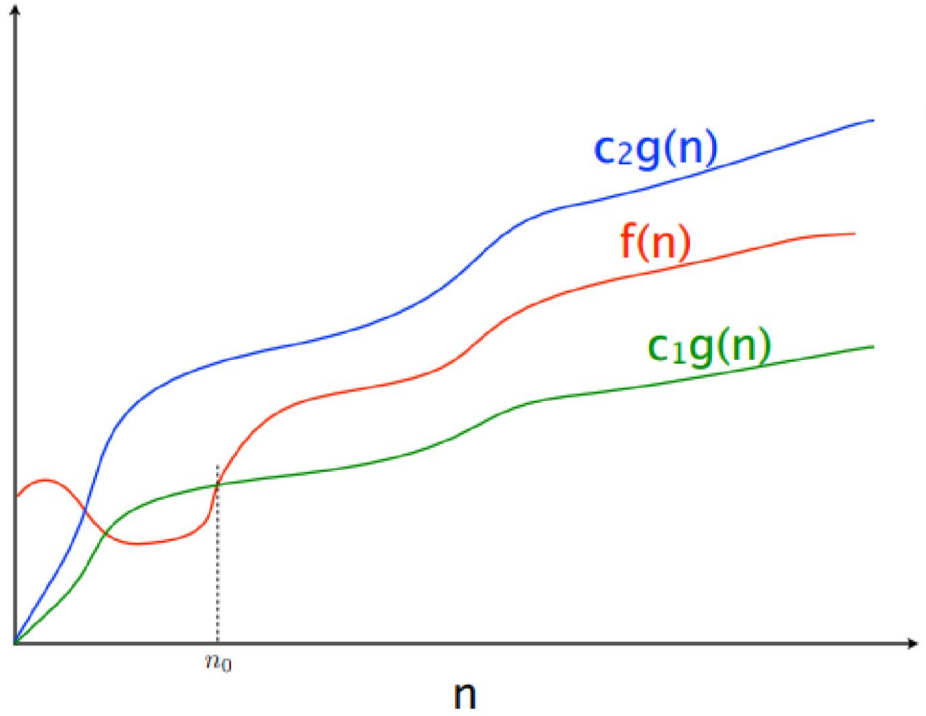
O(1)	constant (same as O(k) for constant k)
O(log n)	logarithmic
O(n)	linear
O(n log n)	“n log n”
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(n^k)$	polynomial (where is k is any constant > 1)
$O(k^n)$	exponential (where k is any constant > 1)

Usage note: “exponential” does not mean “grows really fast”, it means “grows at rate proportional to kn for some k>1”

More Asymptotic Notation

- **Upper bound**: O(**f(n)**) is the set of all functions asymptotically less than or equal to **f(n)**
  - g(n) ∈ O(**f(n)**) if there exist constants **c** and n_0 such that g(n) ≤ **c f(n)** for all n ≥ n_0
- **Lower bound**: **Ω**(**f(n)**) is the set of all functions asymptotically greater than or equal to **f(n)**
  - g(n) ∈ **Ω**(**f(n)**) if there exist constants **c** and n_0 such that g(n) ≥ **c f(n)** for all n ≥ n_0
- **Tight bound**: **θ**(**f(n)**) is the set of all functions asymptotically equal to **f(n)**
  - Intersection of O(**f(n)**) and **Ω**(**f(n)**) (can use different **c** values)

O, Theta, Omega



$f(n) = O(g(n))$

$f(n) = \Theta(g(n))$

$f(n) = \Omega(g(n))$

Regarding use of terms

A common error is to say **O(f(n))** when you mean **θ(f(n))**

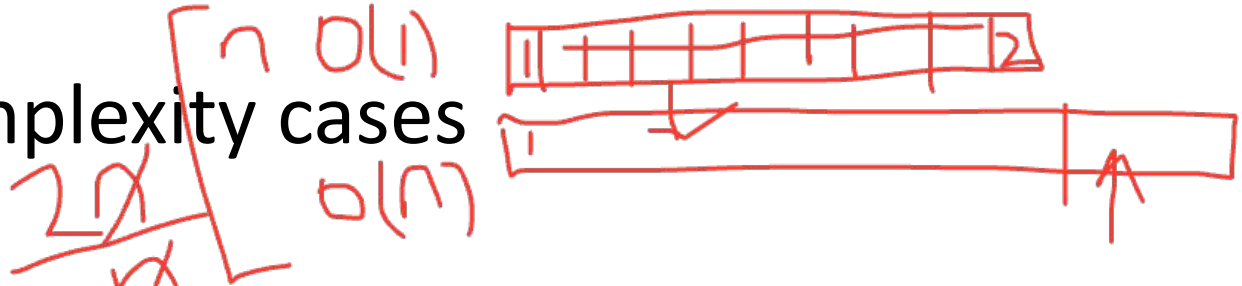
- People often say O() to mean a tight bound
- Say we have f(n)=n; we could say f(n) is in O(n), which is true, but only conveys the upper-bound
- Since f(n)=n is also $O(n^5)$, it's tempting to say "*this algorithm is exactly O(n)*"
- Somewhat incomplete; instead say it is **θ(n)**
- That means that it is not, for example **O(log n)**

Less common notation:

- "little-oh": like "big-Oh" but strictly less than
  - Example: sum is $o(n^2)$ but not o(n)
- "little-omega": like "big-Omega" but strictly greater than
  - Example: sum is ω(log n) but not ω(n)

o (n)

Summary of Complexity cases



Problem size **N**

- **Worst-case complexity: max** # steps algorithm takes on "most challenging" input of size **N**
- **Best-case complexity: min** # steps algorithm takes on "easiest" input of size **N**

- **Average-case complexity: avg** # steps algorithm takes on random inputs of size **N**
- **Amortized complexity: max** total # steps algorithm takes on **M** "most challenging" consecutive inputs of size **N**, divided by **M** (i.e., divide the max total by **M**).

What we are analyzing

- The most common thing to do is give an O or **θ** bound to the worst-case running time of an algorithm

- Example: True statements about binary-search algorithm
  - *Common*: **θ**(log n) running-time in the worst-case
  - *Less common*: **θ**(1) in the best-case (item is in the middle)
  - *Less common*: Algorithm is **Ω**(log log n) in the worst-case (it is not really, really, really fast asymptotically)
  - *Less common* (but very good to know): the find-in-sorted-array ***problem*** is **Ω**(log n) in the worst-case
    - *No* algorithm can do better (without parallelism)
    - A ***problem*** cannot be O(f(n)) since you can always find a slower algorithm, but can mean ***there exists*** an algorithm

Summary

Analysis can be about:

- The problem or the algorithm (usually algorithm)
- Time or space (usually time)
  - Or power or dollars or …
- Best-, worst-, or average-case (usually worst)
- Upper-, lower-, or tight-bound (usually upper or tight)

Summary

	O	Ω	θ
Best			
Worst			💖
Average			

Addendum: Timing vs Big-Oh?

- At the core of CS is a backbone of theory & mathematics
  - Examine the algorithm itself, mathematically, not the implementation
  - Reason about performance as a function of n
  - Be able to mathematically prove things about performance
- Yet, timing has its place
  - In the real world, we do want to know whether implementation A runs faster than implementation B on data set C
  - Ex: Benchmarking graphics cards
- Evaluating an algorithm? Use asymptotic analysis
- Evaluating an implementation of hardware/software? Timing can be useful

Asymptotic Notation Example

Show $10n + 100 \in O(n^2)$

- Technique: find values $c > 0$ and $n_0 > 0$ such that $10n + 100$ is less than $c * n^2$

Asymptotic Notation Example

Show $10n + 100 \in O(n^2)$

- *Technique: find values $c > 0$ and $n_0 > 0$ such that $10n + 100$ is less than or equal to $c * n^2$*

Let $c = 10$, $n_0 = 6$

$10n + 100 <= 10n^2$

$n + 10 <= n^2$

$10 <= n^2 - n$

$10 <= n*(n-1)$

$n*(n-1)$ is strictly increasing, and

$10 < 6*(6-1)$