

Reviewmath

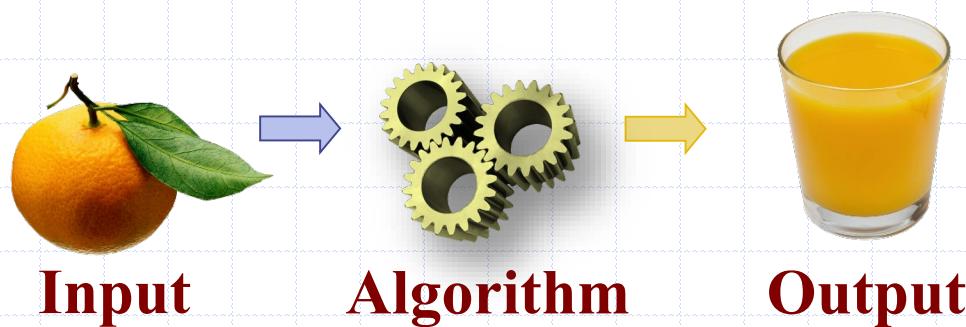
L7 Slide 25

L7 exercises ch3 first 10

Pseudo code- practical solution

- written in plain english
- goal: to process large input efficiently

Analysis of Algorithms



Sareh Taebi

- use to see if algorithm is efficient / good
- exponential = not good = not efficient (grows exponentially)

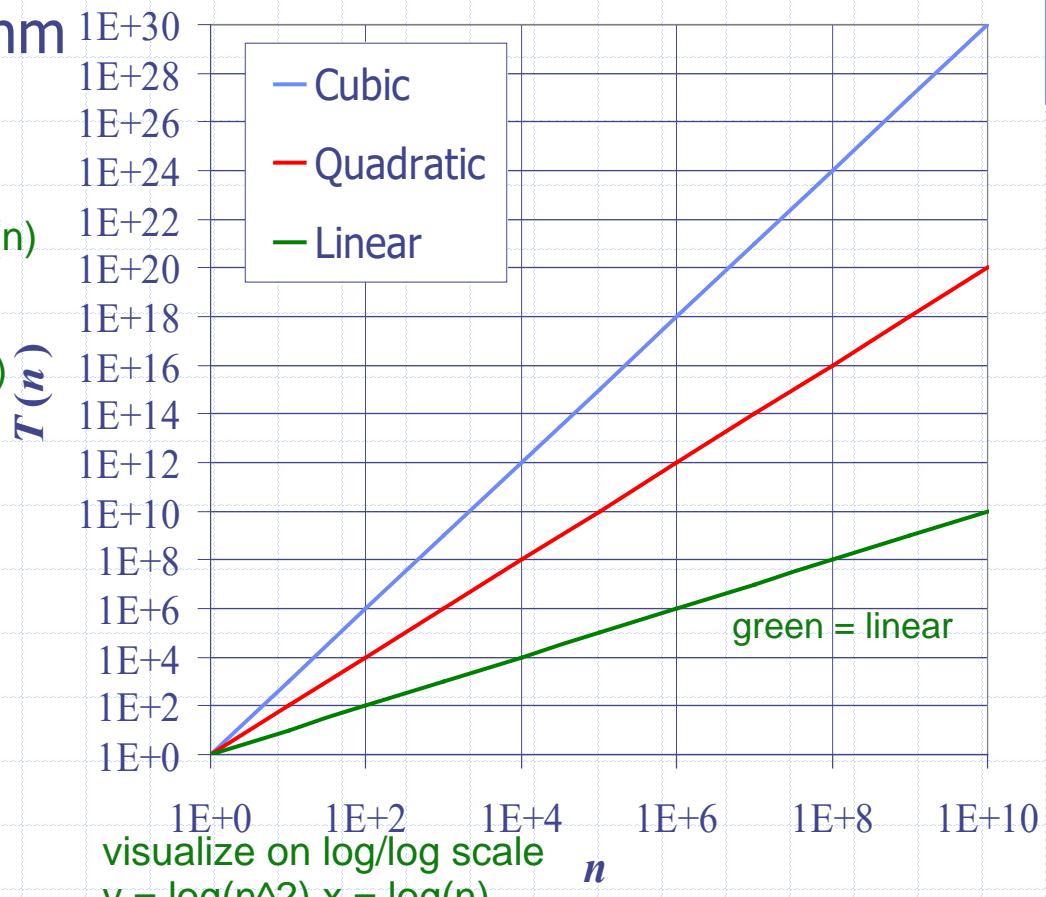
n = # inputs
 $f(n)$ = time it takes to compute algorithm
- units are relative
- don't worry about units for now

Seven Important Functions

- Seven functions that often appear in algorithm analysis:

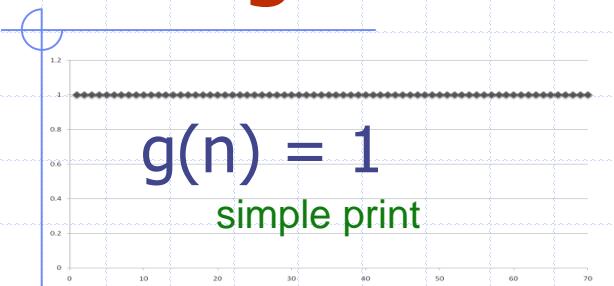
- Constant ≈ 1 $f(n) = c$
- Logarithmic $\approx \log n = f(n)$
- Linear $\approx n = f(n)$
- N-Log-N $\approx n \log n = f(n)$
- Quadratic $\approx n^2 = f(n)$
- Cubic $\approx n^3 = f(n)$
- Exponential $\approx 2^n = f(n)$

- In a log-log chart, the slope of the line corresponds to the growth rate



Functions Graphed Using “Normal” Scale

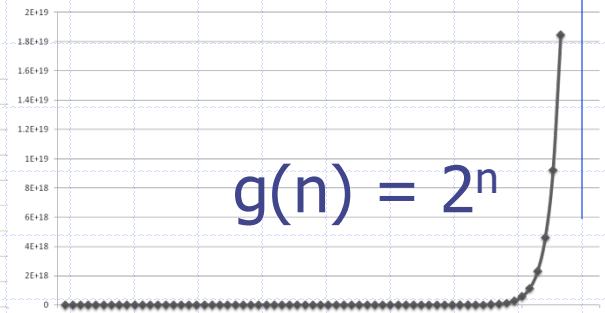
Slide by Matt Stallmann
included with permission.



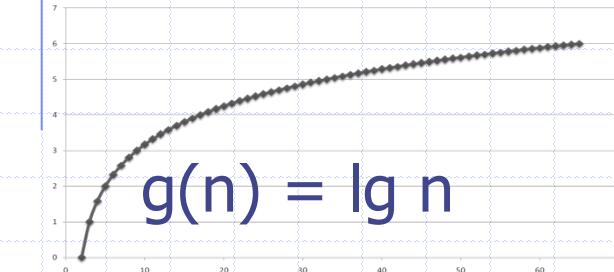
$$g(n) = 1$$

simple print

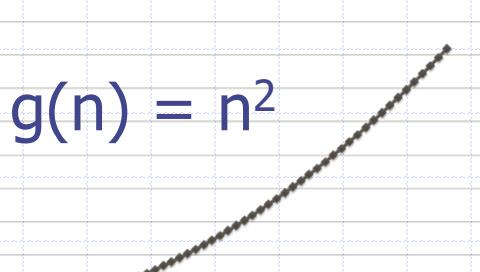
$$g(n) = n \lg n$$



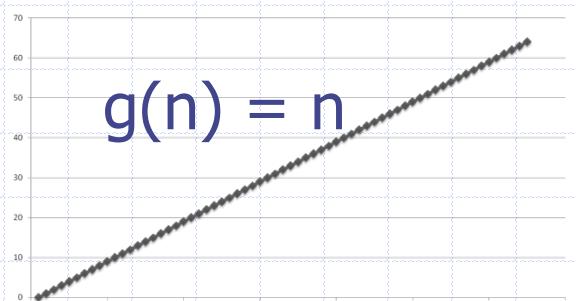
$$g(n) = 2^n$$



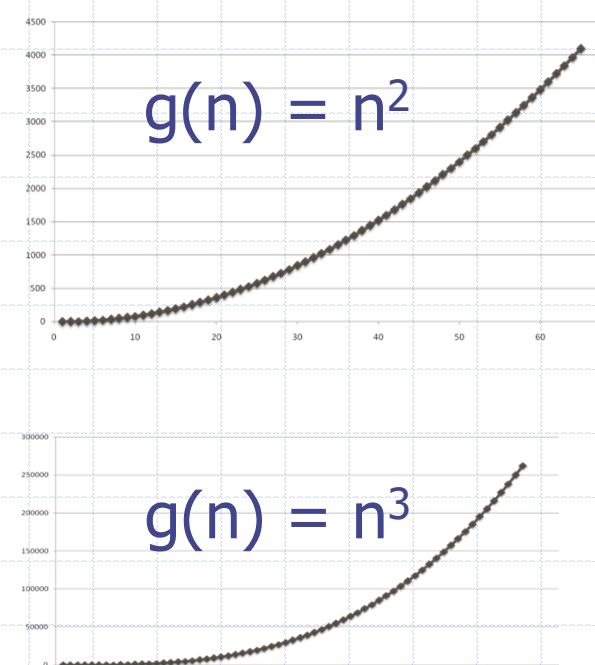
$$g(n) = \lg n$$



$$g(n) = n^2$$



$$g(n) = n$$



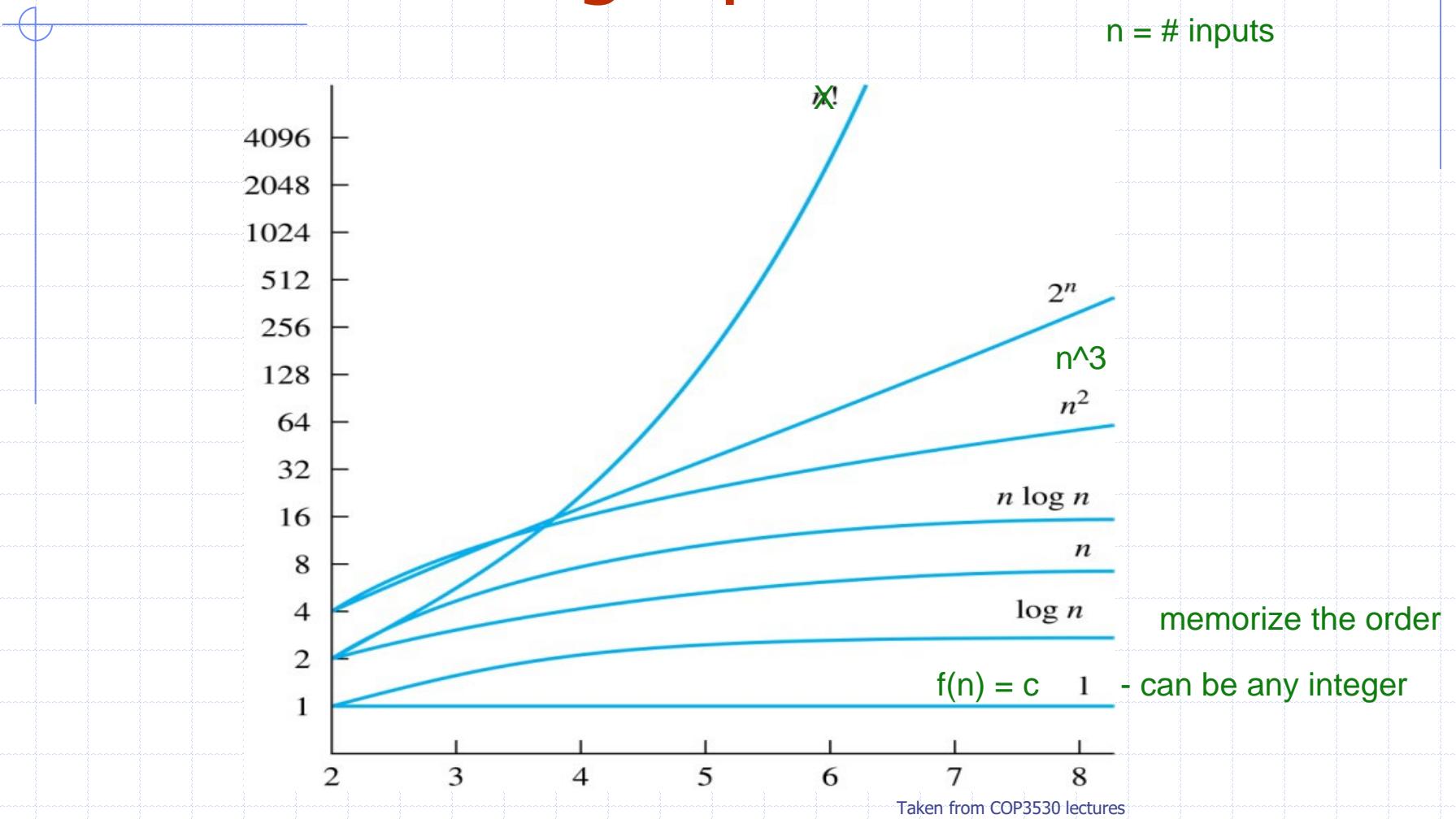
$$g(n) = n^3$$

visualize growth
- use log/log scale
- this class use log base 2
--- binary

loop $\rightarrow n$ $\sim n^2$
nested loop $\times n$

↳ add exponent for each nested loop
↳ avoid if you can
↳ else if only compares once
↳ logn

Display of Growth of Functions with increasing input size



for loop: knows size of list

- makes sure value is within bounds of list, index i should always be less than n
- comparison is burdensome for system, have to do n times

Counting Primitive Operations

- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

```
1 def find_max(data): [ , , , , , ]- compare each element to find largest
2     """Return the maximum element from a nonempty Python list."""
3     biggest = data[0] // update variable # The initial value to beat
4     for val in data: // goes through list # For each value:
5         if val > biggest //comparing current # if it is greater than the best so far,
6             biggest = val // update max # we have found a new best (so far)
7     return biggest # When loop ends, biggest is the max
```

1 op
+ n op
+ n
+ 1 ... n
1

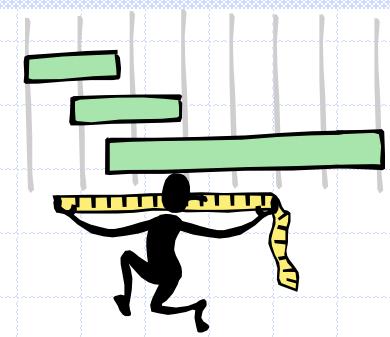
= $3n + 3$

$n = \# \text{ inputs}$
- size of data

- Step 1: 2 ops, 3: 2 ops, 4: 2n ops, 5: 2n ops, 6: 0 to n ops, 7: 1 op

if max = data[0] --> 2n+3

is there a big difference between $2n+3$ or $3n+3$? No both are linear functions



Estimating Running Time

- Algorithm `find_max` executes $5n + 5$ primitive operations in the worst case, $4n + 5$ in the best case. Define:
 - a = Time taken by the fastest primitive operation
 - b = Time taken by the slowest primitive operation
- Let $T(n)$ be worst-case time of `find_max`. Then
$$a(4n + 5) \leq T(n) \leq b(5n + 5)$$
- Hence, the running time $T(n)$ is bounded by two linear functions.

Growth Rate of Running Time

- Changing the hardware/ software environment
 - Affects $T(n)$ by a constant factor, but
 - Does not alter the growth rate of $T(n)$
- The **linear** growth rate of the running time $T(n)$ is an intrinsic property of algorithm **find_max**

- not cubic, quadratic, exponential

$O(n)$ - order of n

- order of complexity of algorithm



Why Growth Rate Matters

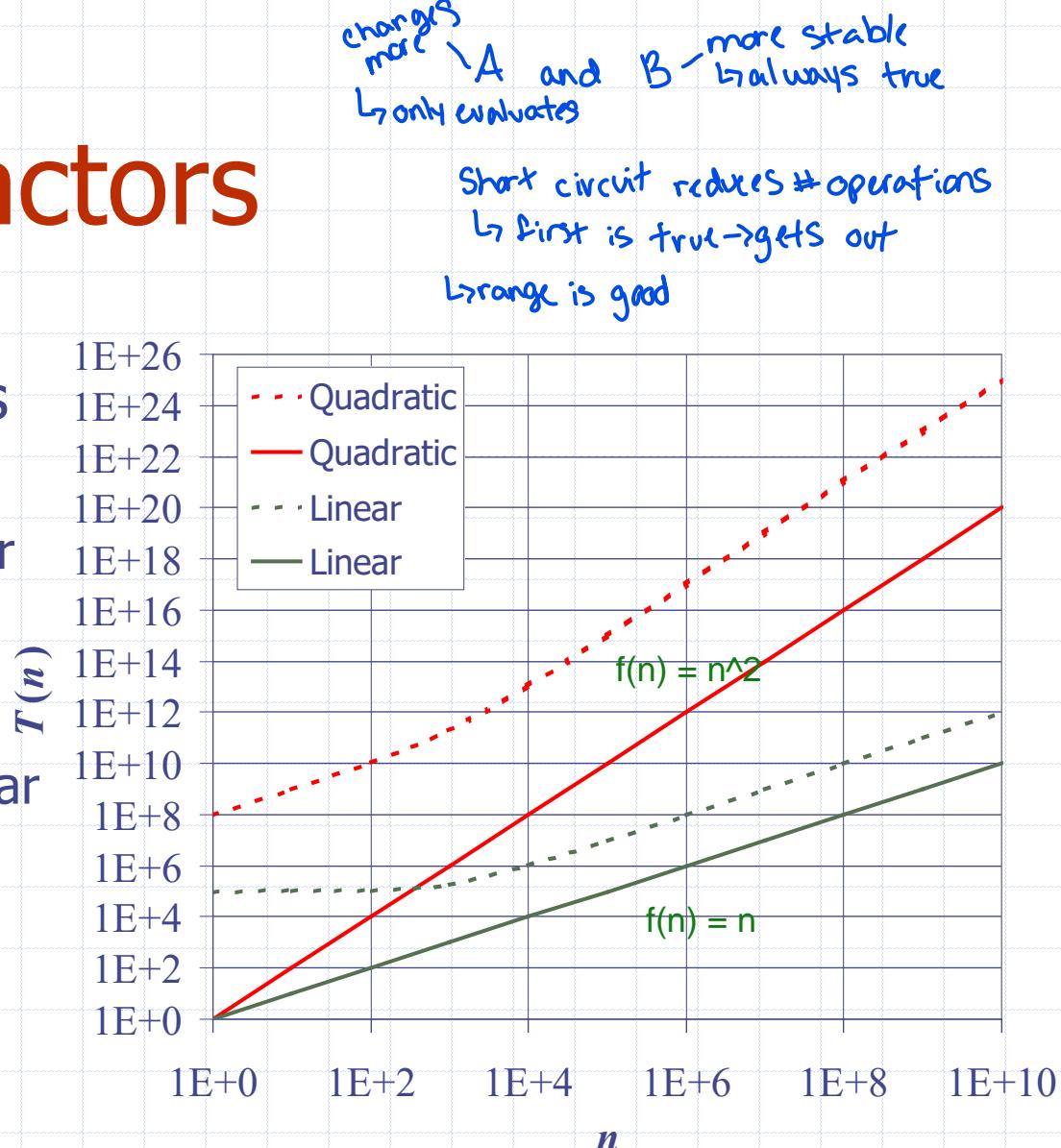
Slide by Matt Stallmann included with permission.

if runtime is...	time for n + 1	time for 2 n	time for 4 n
$c \lg n$	$c \lg (n + 1)$	$c (\lg n + 1)$	$c(\lg n + 2)$
$c n$	$c (n + 1)$	$2c n$	$4c n$
$c n \lg n$	$\sim c n \lg n + c n$	$2c n \lg n + 2cn$	$4c n \lg n + 4cn$
$c n^2$	$\sim c n^2 + 2c n$	$4c n^2$	$16c n^2$
$c n^3$	$\sim c n^3 + 3c n^2$	$8c n^3$	$64c n^3$
$c 2^n$	$c 2^{n+1}$	$c 2^{2n}$	$c 2^{4n}$

runtime quadruples when problem size doubles

Constant Factors

- The growth rate is not affected by
 - constant factors or
 - lower-order terms
- Examples
 - $10^2n + 10^5$ is a linear function $O(n)$
 - $10^5n^2 + 10^8n$ is a quadratic function $O(n^2)$
- grows faster



The "Big-Oh" Notation

- Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers.
- $f(n)$ is $O(g(n))$ if for a real constant $c > 0$ and integer constant $n_0 \geq 1$
 - $f(n) \leq cg(n)$, for $n \geq n_0$.

find n_0 & c
- building upper bound for speed of algorithm

memorize:
- $f(n) \leq cg(n)$

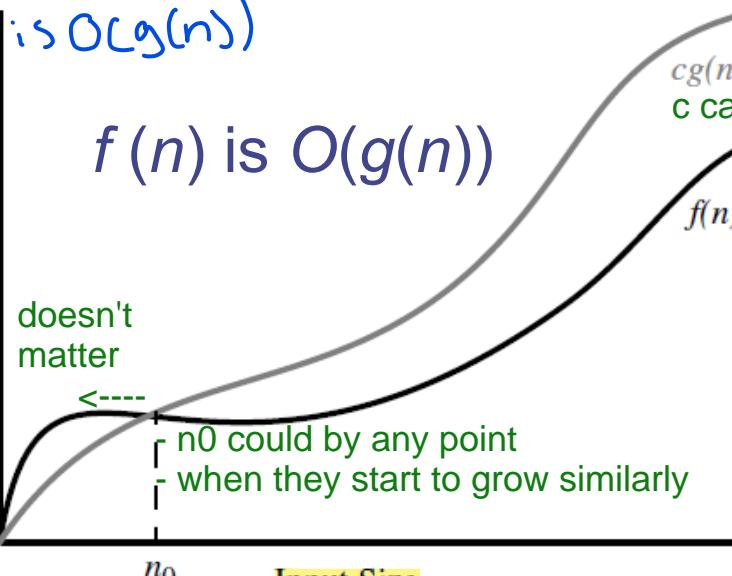
$c = \text{positive}$
 $n = \text{positive}$
--- $n = \# \text{ inputs}$

more operations
=
more running time

then $f(n)$ is $O(g(n))$

$f(n)$ is $O(g(n))$

$c g(n)$
 c can be any integer



Big-Oh Notation

→ prove order of n
 ↳ proof of complexity → \log & the γ

Example: $2n + 10$ is $O(n)$

- $2n + 10 \leq cn$
- $(c - 2)n \geq 10$
- $n \geq 10/(c - 2)$
- Pick $c = 3$ and $n_0 = 10$

$$f(n) = 2n + 10 \quad \text{is } O(n)$$

how to prove? put in equation:

$$2n + 10 \leq cn$$

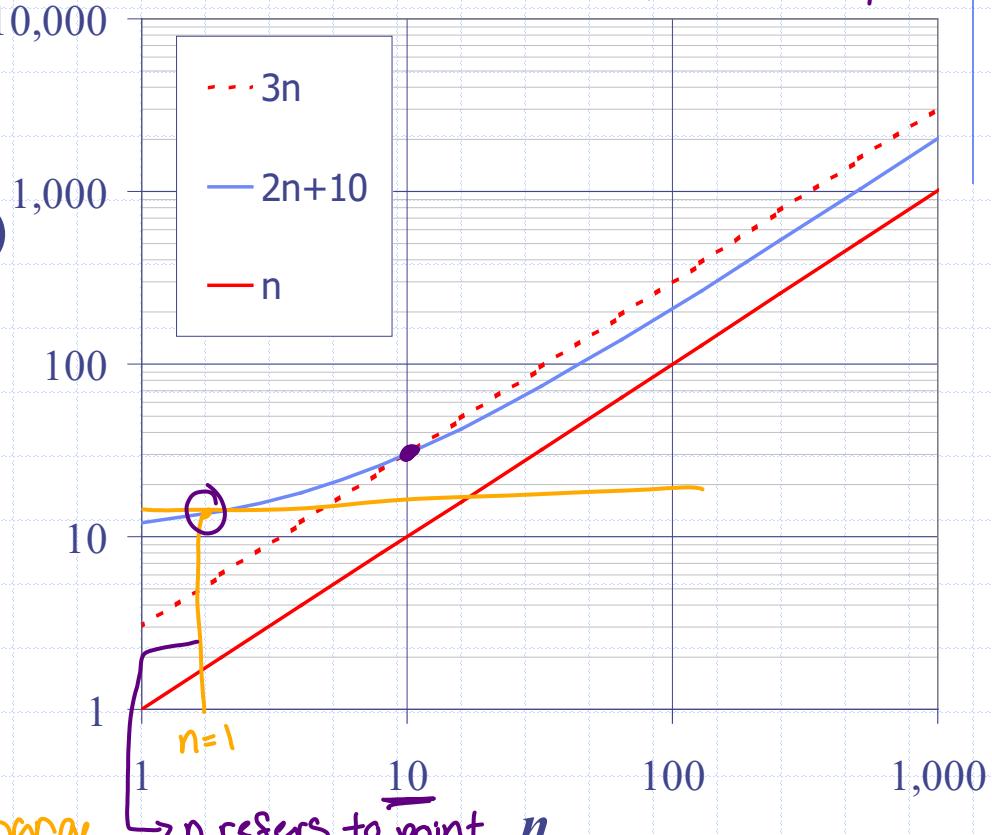
$$10 \leq (c-2)n$$

$$\text{if } n=1: 10 \leq (c-2)1$$

$$10 \leq (c-2)$$

$$\text{True} \leftarrow 10 \leq c$$

values for
 c & n can change
 ↳ not just 1 pair
 of values will
 be true



Big-Oh Example

Never write $2n+2 = O(n)$

↳ not equal \rightarrow belongs to order

↳ $\forall \epsilon$ is instead of =
↳ or set notation

↳ trying to prove $n^2 \neq O(n)$

- Example: the function n^2 is not $O(n)$

- $n^2 \leq cn$
- $n \leq c$
- The above inequality cannot be satisfied since c must be a constant

↳ plug into equation:

$$n^2 \leq cn$$

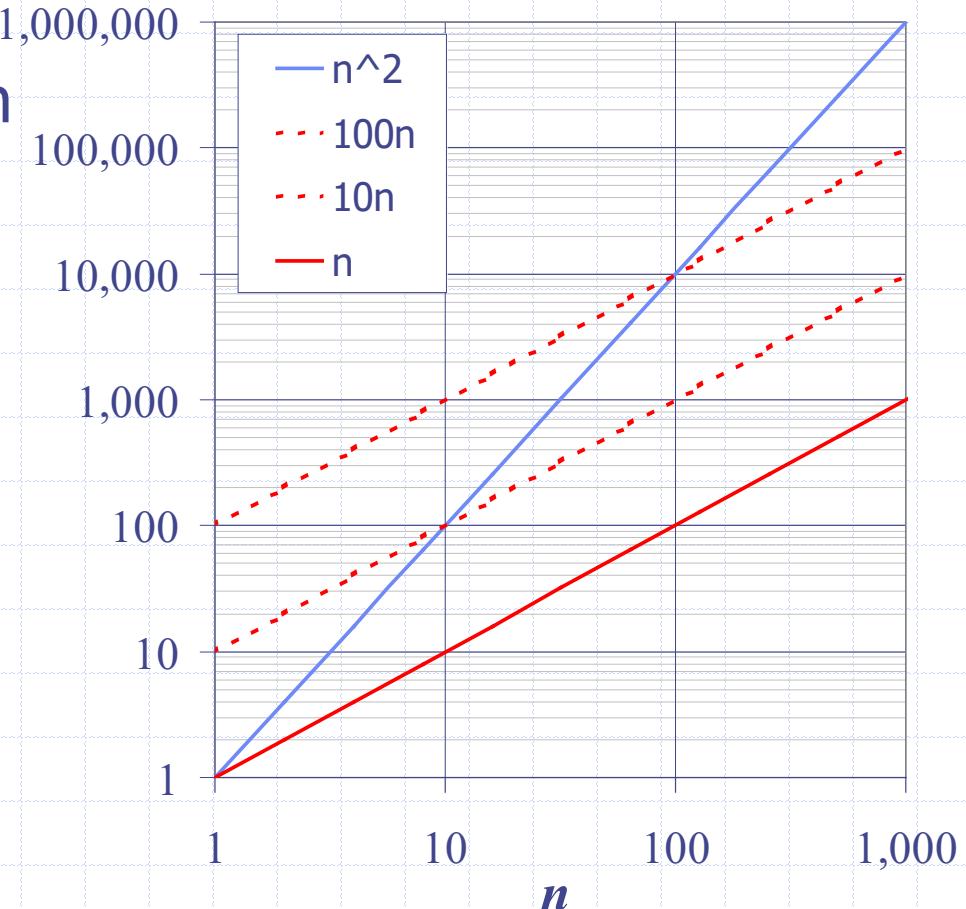
$$n \leq c = 12$$

↳ n can be anything

↳ not just ≤ 12

↳ false

↳ inequality that can't be satisfied $\rightarrow n^2 \neq O(n)$



More Big-Oh Examples



◆ $7n - 2$: linear, $O(n)$

$7n - 2$ is $O(n)$ ✓

need $c > 0$ and $n_0 \geq 1$ such that $7n - 2 \leq cn$ for $n \geq n_0$

this is true for $c = 7$ and $n_0 = 1$

$$f(\infty) \leq cg(n) \leftarrow$$

$$7n - 2 \leq cn \rightarrow \text{pick } 2 \text{ values to be true}$$

~~$7n - 2 \leq 7n$~~

$\hookrightarrow n=1, c=5$

\hookrightarrow when $c = 7$, true for any n

will be larger than original

$$3n^3 + 20n^2 + 5 \leq cn^3$$

$$(3+20+5)n^3 \leq c$$

$$\begin{cases} c=28 \\ n=1 \end{cases} \quad \begin{cases} c=4 \\ n=21 \end{cases} \quad \begin{cases} c \downarrow = \text{lower order} \\ n \uparrow = \text{cross later on graph} \end{cases}$$

\hookrightarrow tighter data

■ $3n^3 + 20n^2 + 5$ \rightarrow only highest order matters
 $O(n^3)$

$3n^3 + 20n^2 + 5$ is $O(n^3)$

need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq cn^3$ for $n \geq n_0$

this is true for $c = 4$ and $n_0 = 21$

$$\log_2 2 = 1$$

$$\begin{cases} n=2 \\ c=8 \end{cases}$$

■ $3 \log n + 5$ is $O(\log n) \rightarrow 3 \log n + 5 \leq c \log n$

$3 \log n + 5$ is $O(\log n)$

~~$\log n(3+5) \leq c \log n$~~

need $c > 0$ and $n_0 \geq 1$ such that $3 \log n + 5 \leq c \log n$ for $n \geq n_0$

this is true for $c = 8$ and $n_0 = 2$

Big-Oh and Growth Rate

- The big-Oh notation gives an upper bound on the growth rate of a function
worst case scenario
- The statement “ $f(n)$ is $O(g(n))$ ” means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$
- We can use the big-Oh notation to rank functions according to their growth rate $\frac{g(n)}{f(n)}$ probably grows $> f(n)$

	$f(n)$ is $O(g(n))$	$g(n)$ is $O(f(n))$
$g(n)$ grows more	Yes	No
$f(n)$ grows more	No	Yes
Same growth	Yes	Yes

Big-Oh Rules



- If $f(n)$ is a polynomial of degree d , then $f(n)$ is $O(n^d)$, i.e., $f(n) = n^d + n^{d-1} + \dots + n + c$
 1. Drop lower-order terms
 2. Drop constant factors \downarrow $O(n^d) \rightarrow$ highest growth
- Use the smallest possible class of functions
 - Say “ $2n$ is $O(n)$ ” instead of “ $2n$ is $O(n^2)$ ”
 $\xrightarrow{\text{go w/ smaller}}$ $2n \leq Cn^2$
 $2=n, c=1$
- Use the simplest expression of the class
 - Say “ $3n + 5$ is $O(n)$ ” instead of “ $3n + 5$ is $O(3n)$ ”

Growth rates of the seven functions

- Look at the big picture. For small n, slower algorithms might still perform fine.

units of time slower algorithms →

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,096	262,144	1.84×10^{19}
128	7	128	896	16,384	2,097,152	3.40×10^{38}
256	8	256	2,048	65,536	16,777,216	1.15×10^{77}
512	9	512	4,608	262,144	134,217,728	1.34×10^{154}

Importance of Asymptotic Analysis

$$2n^2 = 10^6 \Rightarrow n^2 = \frac{10^6}{2} \Rightarrow n = \sqrt{500,000} = 707$$

$$\log_2 \cdot 2^n = 10^6 \cdot \log_2 = n = 6 \cdot \log_2 10 \Rightarrow 6 \cdot 3 \sim 18$$

↓
19

Good algorithm design matters!

$$400n = 60 \times 60e^6$$

Running Time (μs)	Maximum Problem Size (n) $1 \text{e}6 \text{ us}$ 1 second	Maximum Problem Size (n) $60 \text{e}6 \text{ us}$ 1 minute	Maximum Problem Size (n) $60 \times 60 \text{e}6 \text{ us}$ 1 hour
$400 \text{ loops} \rightarrow 400n \text{ } O(n)$	2,500	150,000	9,000,000
$2n^2$	707	5,477	42,426
2^n	19	25	31

$$1 \text{ microsecond} = 10^6 \text{ ms}$$

$$400n = 10^6 \Rightarrow n = \frac{10^6}{400} \Rightarrow n = \frac{10^4}{4} \Rightarrow n = 2500 \text{ per 1 sec.}$$

Faster Computer Hardware?

- Good algorithm still plays an important role
- Even with a computer which is 256 times faster!

$$400n \cdot 256 = 256 \cdot m \quad \text{L7 previous : input size}$$

multiply the running time by 256

(16^2 or 2^8)

Running Time	New Maximum Problem Size
$400n$	$256m$
$2n^2$	$16m$
2^n	$m + 8$

$$2n^2 \cdot 256 = 2n^2 \cdot 16^2 = 2(\underbrace{16n}_m)^2 \Rightarrow 2m^2$$

$m \rightarrow$ new input max

$$2^n \cdot 256 \Rightarrow 2^n \cdot 2^8 \quad \begin{cases} \text{only 8} \\ \text{* inputs} \end{cases}$$

Asymptotic Algorithm Analysis

- The asymptotic analysis of an algorithm determines the running time in big-Oh notation
- To perform the asymptotic analysis
 - We find the worst-case number of primitive operations executed as a function of the input size
 - We express this function with big-Oh notation
- Example:
 - We say that algorithm `find_max` “runs in $O(n)$ time”
list size
- Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations

array/set \times []

$\text{L} \geq \text{each element} = \text{average}$

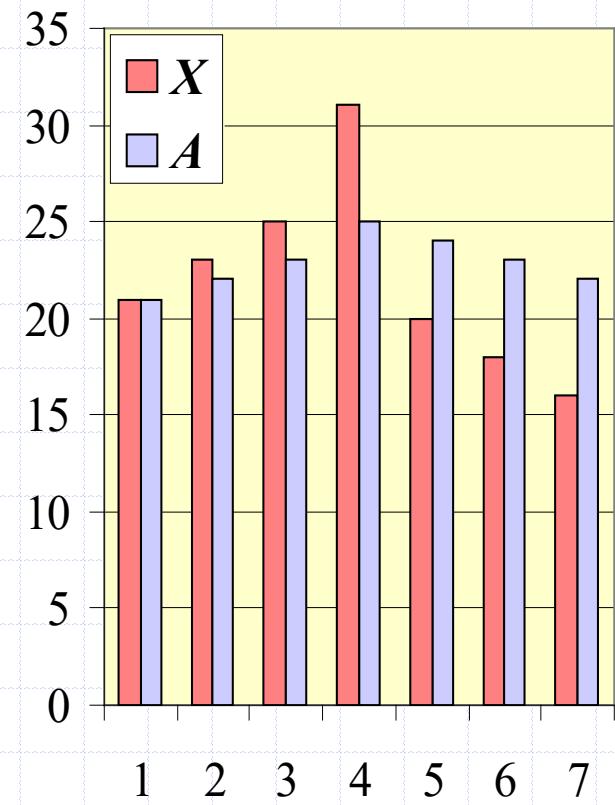
$$A[0] = \underbrace{\cancel{X[0]}}_1; A[i] = \frac{\cancel{X[0]} + X[1]}{2}$$

Computing Prefix Averages

- We further illustrate asymptotic analysis with two algorithms for prefix averages
- The i -th prefix average of an array X is average of the first $(i + 1)$ elements of X :

$$A[i] = (X[0] + X[1] + \dots + X[i])/(i+1)$$

- Computing the array A of prefix averages of another array X has applications to financial analysis



Prefix Averages (Quadratic)

- The following algorithm computes prefix averages in quadratic time by applying the definition

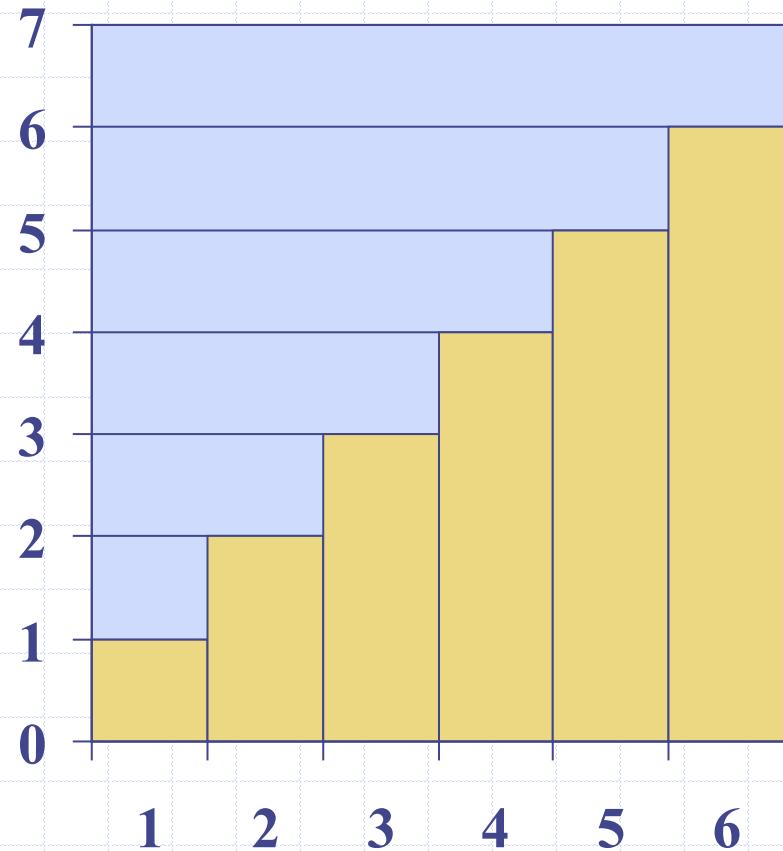
- inner loop that depends
on outer loop = always quadratic

```
1 def prefix_average1(S):
2     """ Return list such that, for all j, A[j] equals average of S[0], ..., S[j]."""
3     n = len(S)
4     A = [0] * n
5     for j in range(n): ntimes
6         total = 0
7         for i in range(j + 1): depends on j
8             total += S[i]
9         A[j] = total / (j+1) # record the average
10    return A
```

$$1 + 2 + 3 + \dots + n = \frac{(1+n) \cdot n}{2} = n^2$$

Arithmetic Progression

- The running time of *prefixAverage1* is $O(1 + 2 + \dots + n)$
- The sum of the first n integers is $n(n + 1) / 2$
 - There is a simple visual proof of this fact
- Thus, algorithm *prefixAverage1* runs in $O(n^2)$ time



Prefix Averages 2 (Looks Better)

- ◆ The following algorithm uses an internal Python function to simplify the code

```
1 def prefix_average2(S):
2     """ Return list such that, for all j, A[j] equals average of S[0], ..., S[j]."""
3     n = len(S)
4     A = [0] * n                      # create new list of n zeros
5     for j in range(n):
6         A[j] = sum(S[0:j+1]) / (j+1)  # record the average
7     return A
    |slicing ; times
```

- ◆ Algorithm *prefixAverage2* still runs in $O(n^2)$ time!

Prefix Averages 3 (Linear Time)

- The following algorithm computes prefix averages in linear time by keeping a running sum

```
1 def prefix_average3(S):
2     """ Return list such that, for all j, A[j] equals average of S[0], ..., S[j]. """
3     n = len(S)
4     A = [0] * n
5     total = 0  $\nearrow n$  # create new list of n zeros
6     for j in range(n):  $O(n)$  # compute prefix sum as S[0] + S[1] + ...
7         total += S[j]  $O(1)$  # update prefix sum to include S[j]
8         A[j] = total / (j+1)  $O(1)$  # compute average based on current sum
9     return A
```

$$\text{total} = S[0]$$

$$\begin{aligned}\text{Total} &= \underbrace{S[0]}_{\text{initial}} + \underbrace{S[1]}_{\text{added}} \\ A[1] &= \frac{\text{Total}}{2}\end{aligned}$$

- Algorithm *prefixAverage3* runs in $O(n)$ time

Math you need to Review



- ◆ Summations
- ◆ Logarithms and Exponents

- **properties of logarithms:**

$$\log_b(xy) = \log_b x + \log_b y$$

$$\log_b(x/y) = \log_b x - \log_b y$$

$$\log_b x^a = a \log_b x$$

$$\log_b a = \log_x a / \log_x b$$

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

- **properties of exponentials:**

$$a^{(b+c)} = a^b a^c$$

$$a^{bc} = (a^b)^c$$

$$a^b / a^c = a^{(b-c)}$$

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

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

- ◆ Proof techniques
- ◆ Basic probability

Big Oh \rightarrow worst case

Relatives of Big-Oh



◆ big-Omega

best case

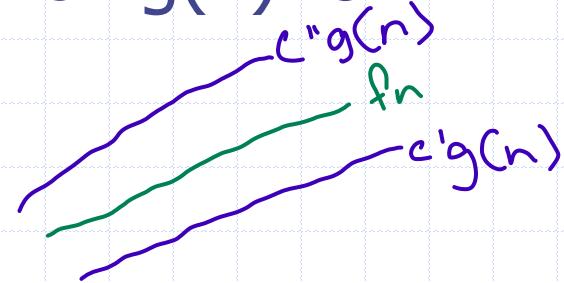


- $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$

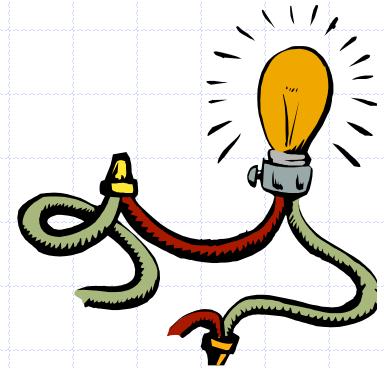
◆ big-Theta

behave the same

- $f(n)$ is $\Theta(g(n))$ if there are constants $c' > 0$ and $c'' > 0$ and an integer constant $n_0 \geq 1$ such that $c' \cdot g(n) \leq f(n) \leq c'' \cdot g(n)$ for $n \geq n_0$



Intuition for Asymptotic Notation



Big-Oh

- $f(n)$ is $O(g(n))$ if $f(n)$ is asymptotically **less than or equal** to $g(n)$

worst-case growth

big-Omega

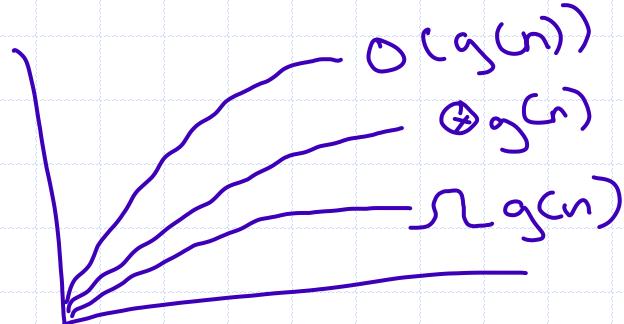
- $f(n)$ is $\Omega(g(n))$ if $f(n)$ is asymptotically **greater than or equal** to $g(n)$

best-case growth

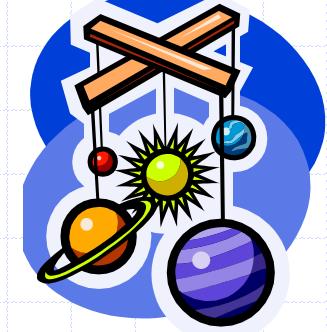
big-Theta

- $f(n)$ is $\Theta(g(n))$ if $f(n)$ is asymptotically **equal** to $g(n)$

asymptotically the same growth



Example Uses of the Relatives of Big-Oh



$$5n^2 \geq cn^2$$

- $5n^2$ is $\Omega(n^2)$

$f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$

let $c = 5$ and $n_0 = 1$

- $5n^2$ is $\Omega(n)$

$$5n^2 \geq cn$$

$f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$

let $c = 1$ and $n_0 = 1$

- $5n^2$ is $\Theta(n^2)$

$$5n^2 \leq cn^2$$

$f(n)$ is $\Theta(g(n))$ if it is $\Omega(n^2)$ and $O(n^2)$. We have already seen the former, for the latter recall that $f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq c \cdot g(n)$ for $n \geq n_0$

Let $c = 5$ and $n_0 = 1$