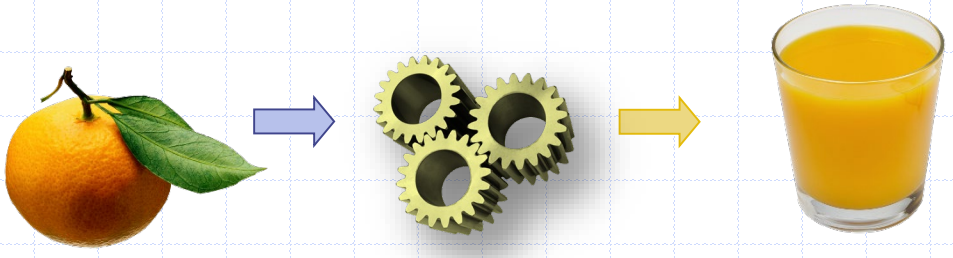
Review math
↳ slide 25
↳ exercises ch3 first 10

Pseudo code- practical solution
- written in plain english
- goal: to process large input efficiently

Analysis of Algorithms



Sareh Taebi

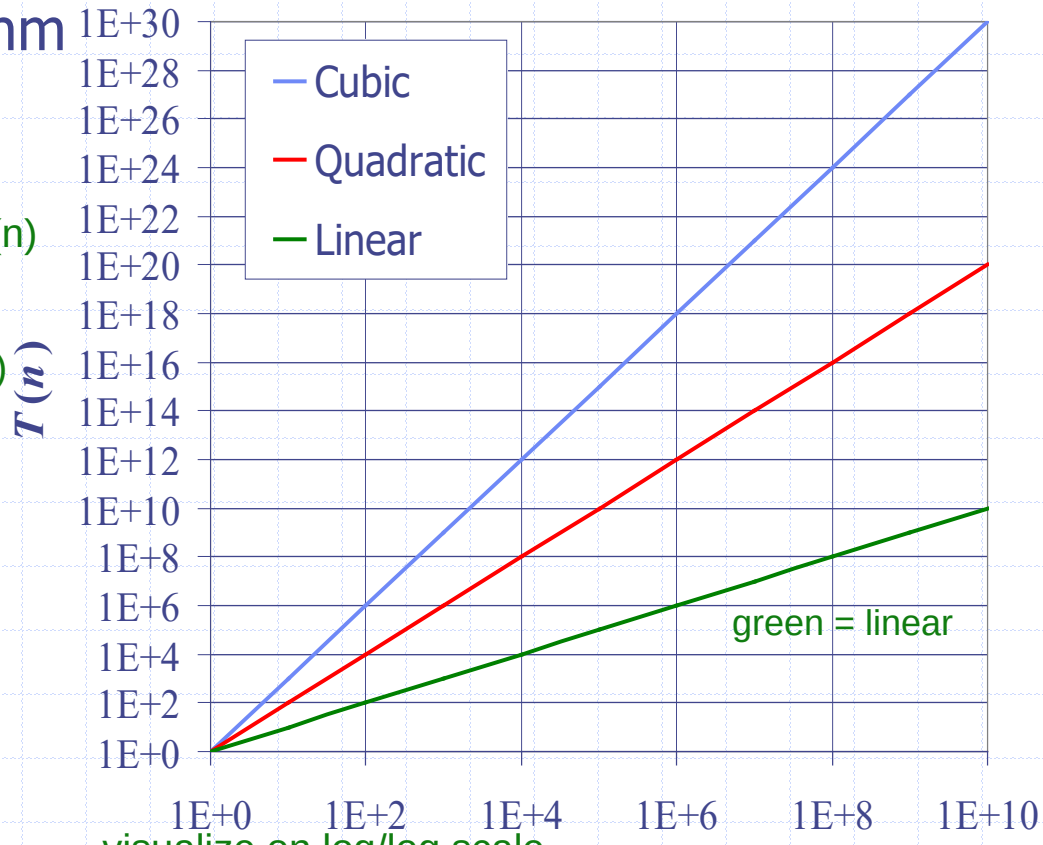
FAU
Florida Atlantic University

- 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

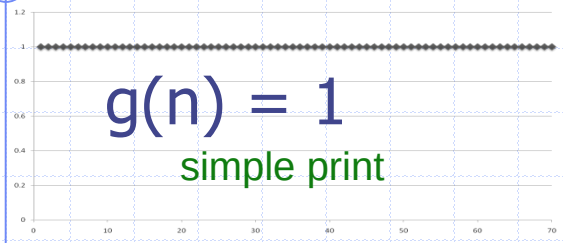
- Seven functions that often appear in algorithm analysis:
  - Constant ≈ 1 f(n) = c
  - Logarithmic $\approx \log n$ = f(n)
  - Linear $\approx n$ = f(n)
  - N-Log-N $\approx n \log n$ = f(n)
  - Quadratic $\approx n^2$ =f(n)
  - Cubic $\approx n^3$ =f(n)
  - Exponential $\approx 2^n$ =f(n)
- In a log-log chart, the slope of the line corresponds to the growth rate

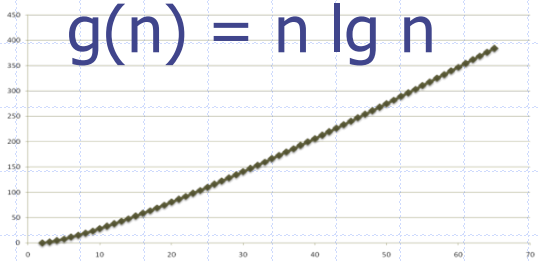


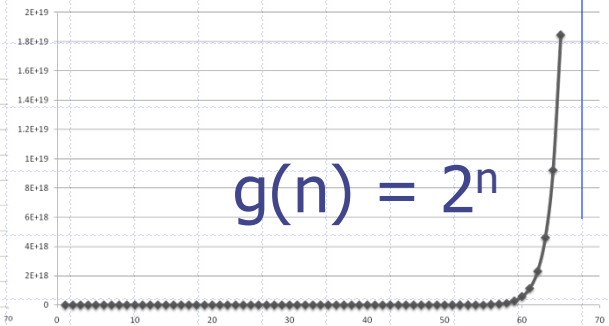
visualize on log/log scale
y = log(n^2) x = log(n)
- slope becomes linear- easier to see slope

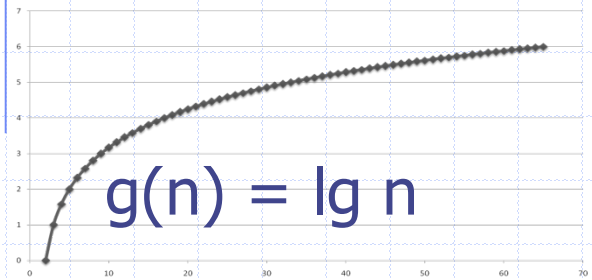
Functions Graphed Using "Normal" Scale

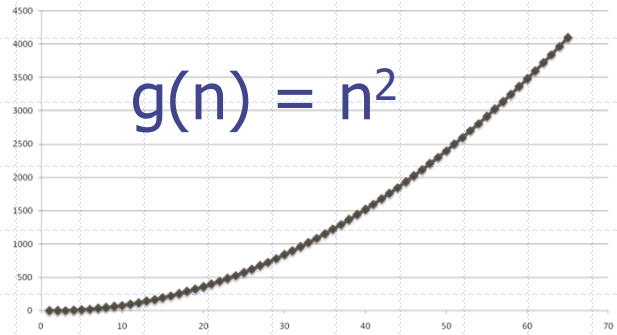
Slide by Matt Stallmann included with permission.





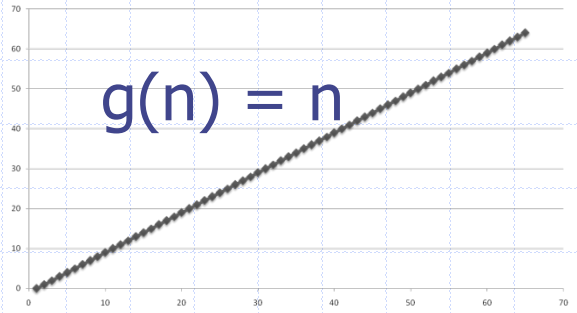


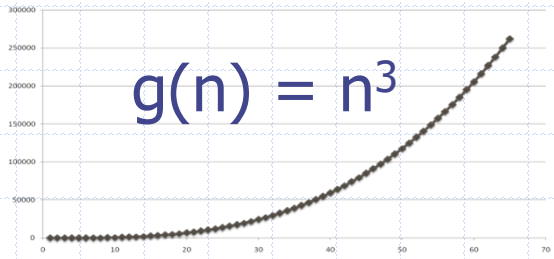




visualize growth
- use log/log scale
- this class use log base 2
--- binary

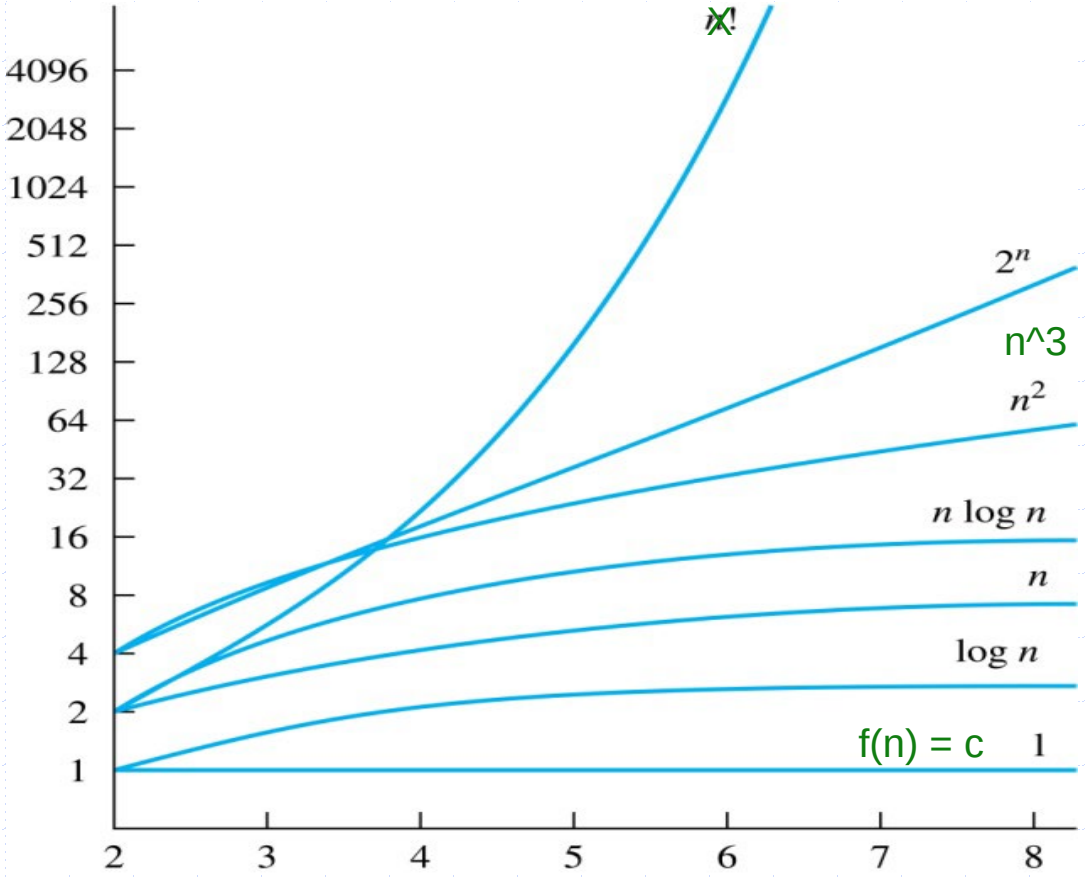
loop → n } n²
nested loop × n
↳ add exponent for each nested loop
↳ avoid if you can
↳ else/if only compares once
↳ log n





Display of Growth of Functions with increasing input size

n = # inputs



Taken from COP3530 lectures

memorize the order

- can be any integer

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

[, , , , ,]- compare each element to find largest
- assume first is largest, then when you hit the next largest
--- that one becomes the max

```
1  def find_max(data):
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

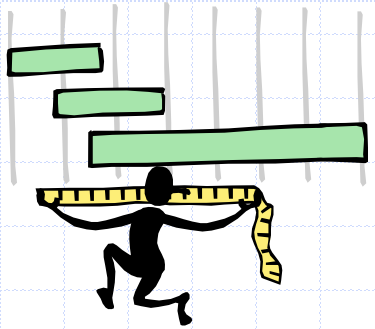
- Step 1: 2 ops, 3: 2 ops, 4: 2n ops, 5: 2n ops, 6: 0 to n ops, 7: 1 op

n = # inputs
- size of data

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

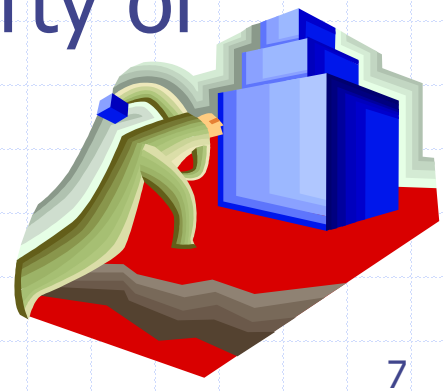
$$a(4n + 5) \leq T(n) \leq b(5n + 5)$$

- 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	~ c n lg n + c n	2c n lg n + 2cn	4c n lg n + 4cn
c n^2	~ c n^2 + 2c n	**4c n^2**	16c n^2
c n^3	~ 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

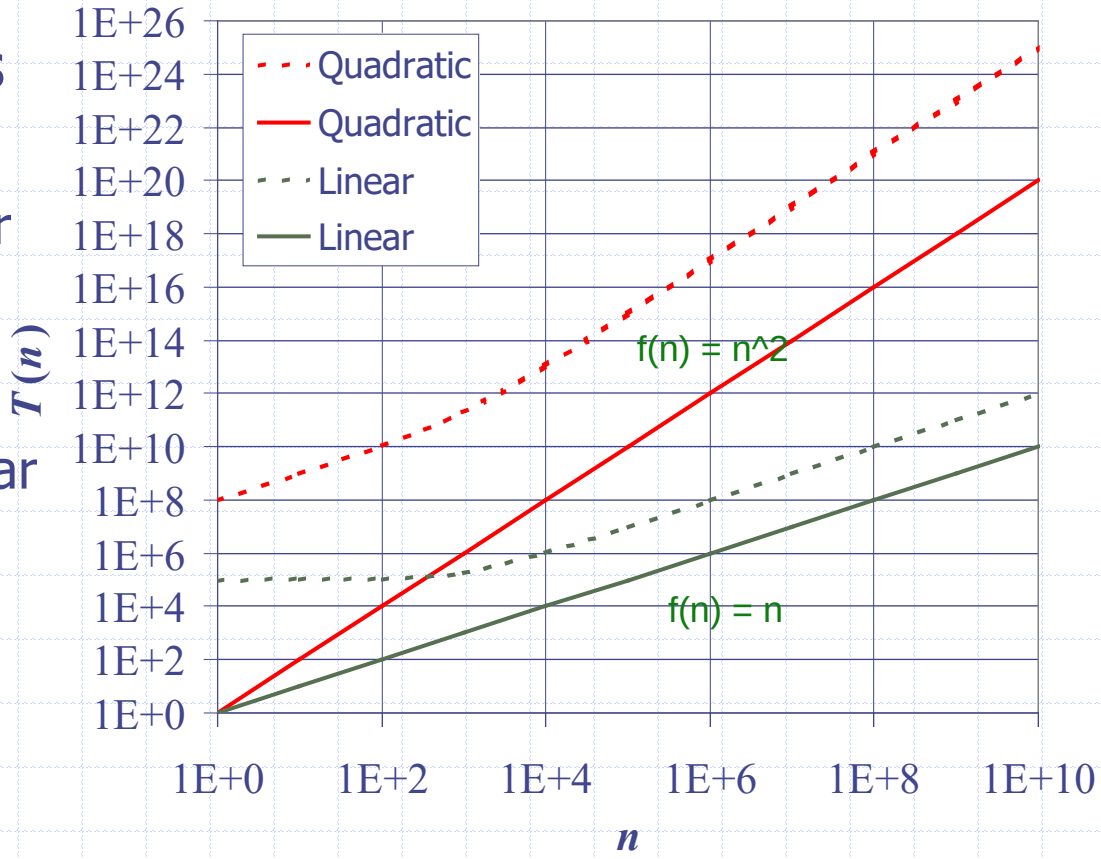
charges more \ A and B - more stable
↳ only evaluates ↳ always true

Constant Factors

Short circuit reduces # operations
↳ first is true → gets out
↳ range is good

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

    O(n^2)
    - grows faster



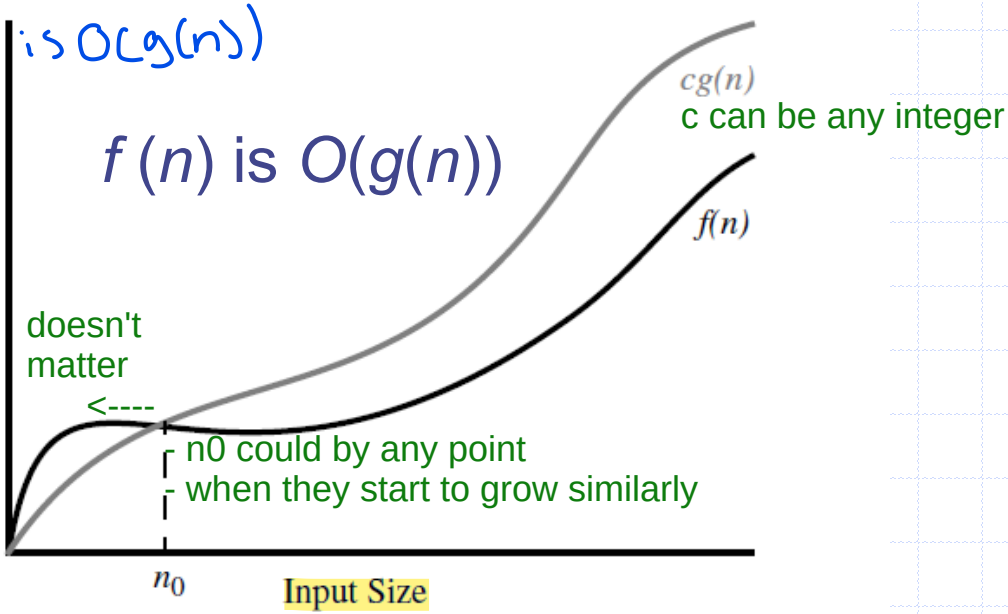
The "Big-Oh" Notation

- 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 $n0 \geq 1$
  - $f(n) \leq cg(n)$, for $n \geq n0$.

find n0 & c
- building upper bound for speed of algorithm

memorize:
- f(n) <= cg(n)

c = positive
n = positive
--- n = # inputs



c can be any integer

doesn't matter
<----

more operations
=
more running time

n0 could by any point
- when they start to grow similarly

Big-Oh Notation → prove order of n
↳ proof of complexity → 1 of the 7

- 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 is O(n)
how to prove? put in equation:

$2n + 10 \leq cn$ (12n)

$10 \leq (c-2)n$

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

$10 \leq (c-2)$

True ← $12 \leq c$

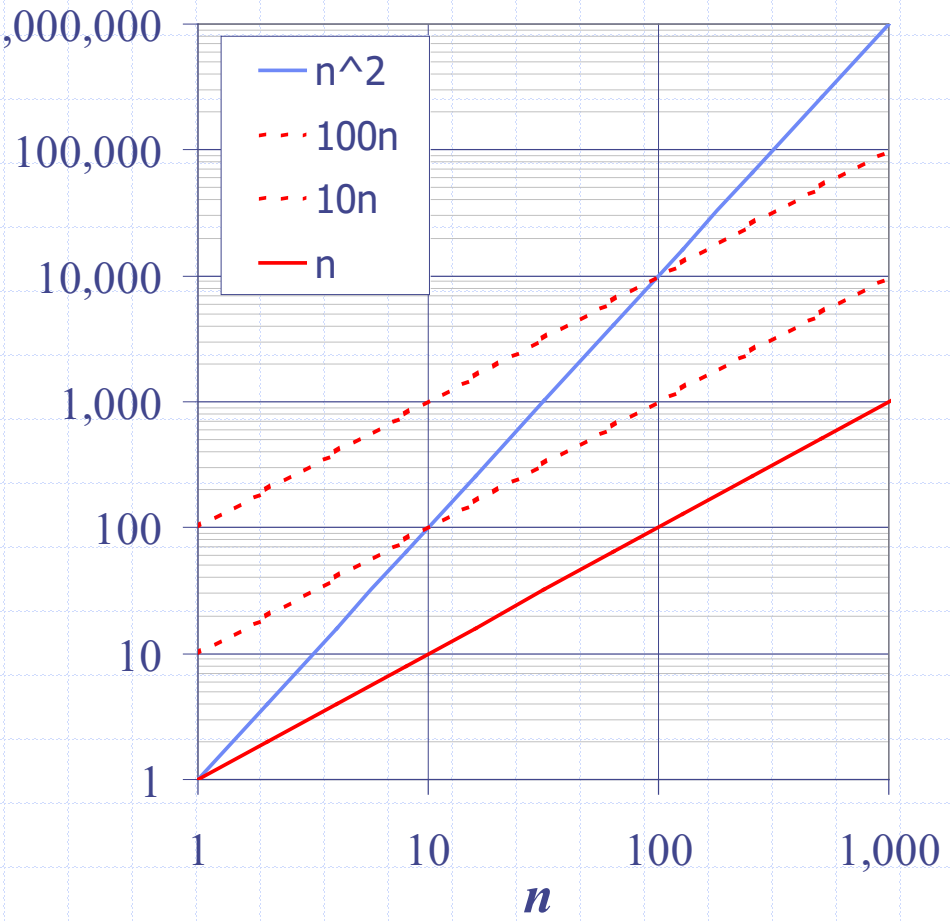
values for c & n can change
↳ not just 1 pair of values will be true

Never write $2n+2 = O(n)$
↳ not equal → belongs to order
↳ use $\in$ instead of $=$
↳ or set notation

Big-Oh Example

↳ trying to prove $n^2 \notin O(n)$

- 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 → $n^2 \notin O(n)$

More Big-Oh Examples

- 7n-2 : linear, O(n)

  7n-2 is O(n) ✓

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

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

- $3n^3 + 20n^2 + 5$ → 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 $3n^3 + 20n^2 + 5 \leq c \bullet n^3$ for $n \geq n_0$

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

- 3 log n + 5 is O(log n)

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

	$f(n)$ 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 is $f(n)$ a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e., $f(n) = n^d + n^{d-1} + \ldots + n + c$
  1. Drop lower-order terms
  2. Drop constant factors

  $O(n^d) \Rightarrow$ highest growth
- Use the smallest possible class of functions
  - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$" $2n \leq cn^2$, $2=n$, $c=1$
- 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 →

Input Size ↓

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) 1e6 us 1 second	60e6 us 1 minute	60*60e6 us 1 hour
400 loops → $400n$ O(n)	2,500	150,000	9,000,000
$2n^2$	707	5,477	42,426
2^n	19	25	31

1microsecond = 10^6 ms

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

Faster Computer Hardware?

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

$400n \cdot 256 = 256 \cdot m$ → previous input size

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

multiply the running time by 256 (16^2 or 2^8)

$2n^2 \cdot 256 = 2n^2 \cdot 16^2 = 2(16n)^2 \Rightarrow 2m^2$

$m \rightarrow$ new input max

$2^n \cdot 256 \Rightarrow 2^n \cdot 2^8 \Rightarrow 2^{n+8}$ } only 8 + inputs

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"
- Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations

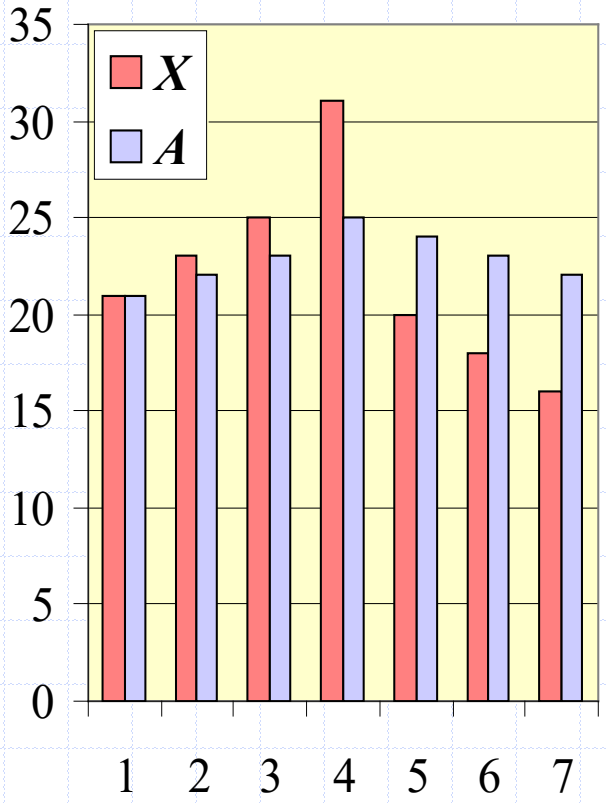
list size

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:

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

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



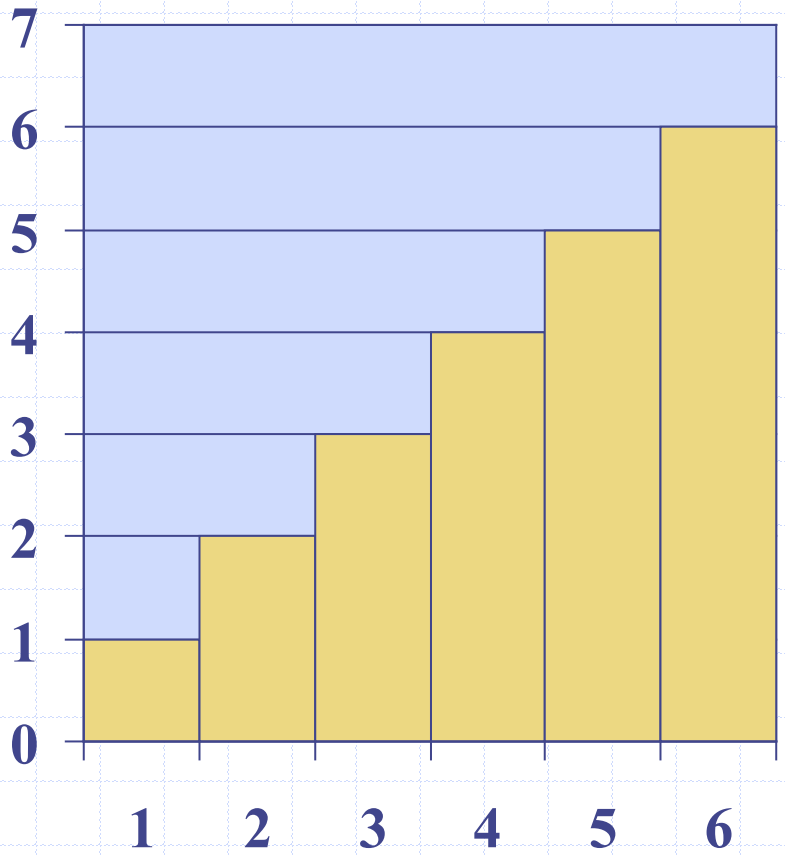
Prefix Averages (Quadratic)

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

```
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                          # create new list of n zeros
5    for j in range(n):
6      total = 0                          # begin computing S[0] + ... + S[j]
7      for i in range(j + 1):
8        total += S[i]
9      A[j] = total / (j+1)               # record the average
10   return A
```

Arithmetic Progression

- The running time of *prefixAverage1* is $O(1 + 2 + \ldots + n)$
- 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
```

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

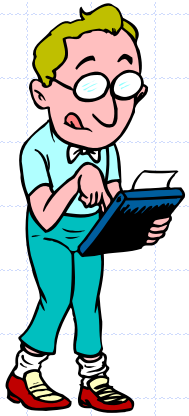
Prefix Averages 3 (Linear Time)

- 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                    # create new list of n zeros
5    total = 0                      # compute prefix sum as S[0] + S[1] + ...
6    for j in range(n):
7      total += S[j]                # update prefix sum to include S[j]
8      A[j] = total / (j+1)         # compute average based on current sum
9    return A
```

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

Math you need to Review



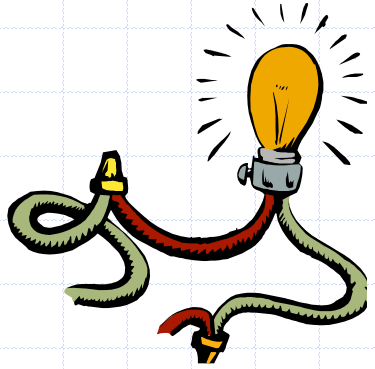
- Summations
- Logarithms and Exponents
- Proof techniques
- Basic probability

- **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**:
  - $a^{(b+c)} = a^b a^c$
  - $a^{bc} = (a^b)^c$
  - $a^b / a^c = a^{(b-c)}$
  - $b = a^{\log_a b}$
  - $b^c = a^{c*\log_a b}$

Relatives of Big-Oh

- **big-Omega**
  - f(n) is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \bullet g(n)$ for $n \geq n_0$
- **big-Theta**
  - 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 $c' \bullet g(n) \leq f(n) \leq c'' \bullet g(n)$ 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 Ω(g(n)) if f(n) is asymptotically **greater than or equal** to g(n)

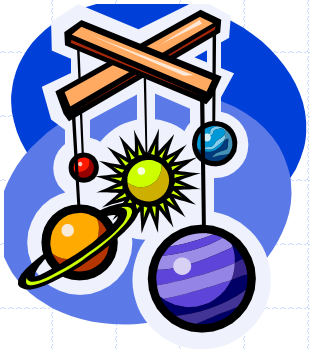
best-case growth

big-Theta

- f(n) is Θ(g(n)) if f(n) is asymptotically **equal** to g(n)

asymptotically the same growth

Example Uses of the Relatives of Big-Oh



- **$5n^2$ 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 $f(n) \geq c \bullet g(n)$ for $n \geq n_0$

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

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

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

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

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

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

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