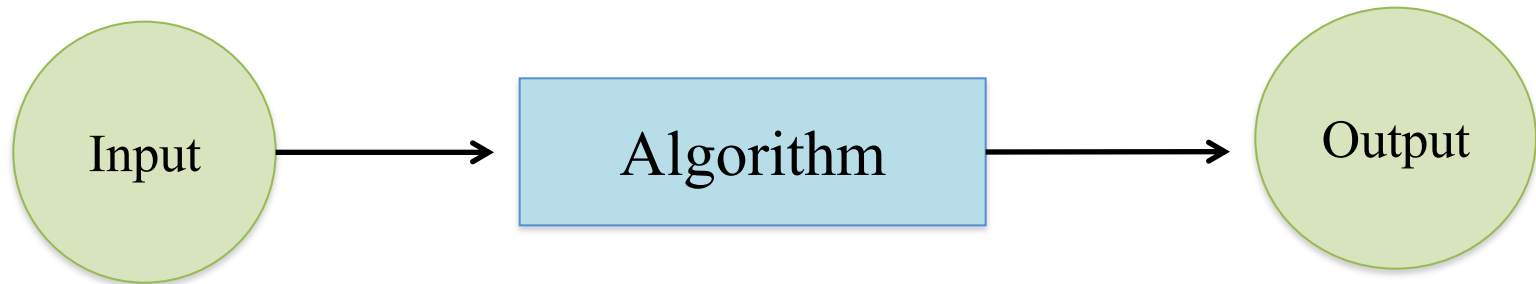


# Analysis of Algorithms

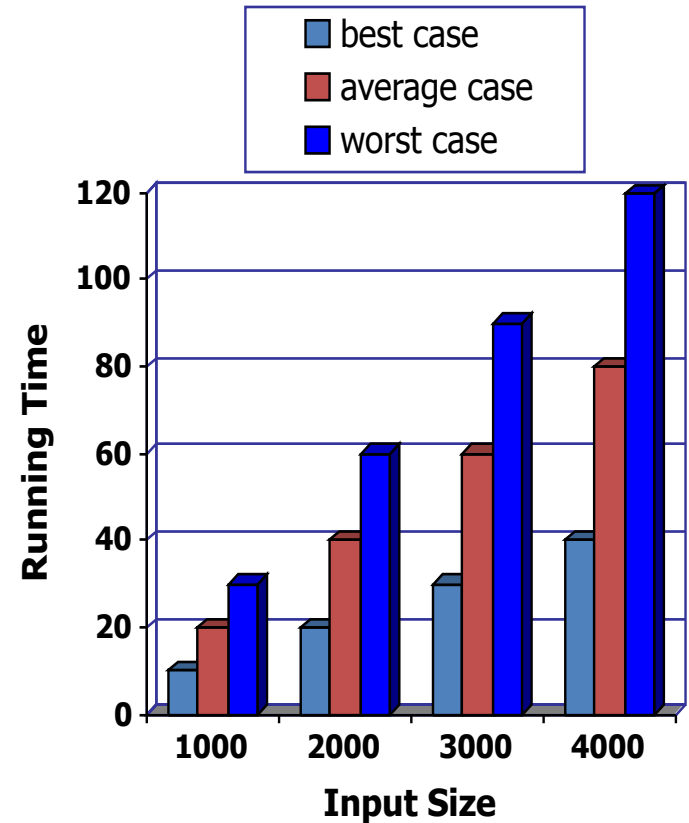
- An **algorithm** is a step-by-step procedure for performing some task (ex: sorting a set of integers) in a finite amount of time.



- We are concerned with the following properties:
  - Correctness
  - Efficiency (how fast it is, how many resources it needs)

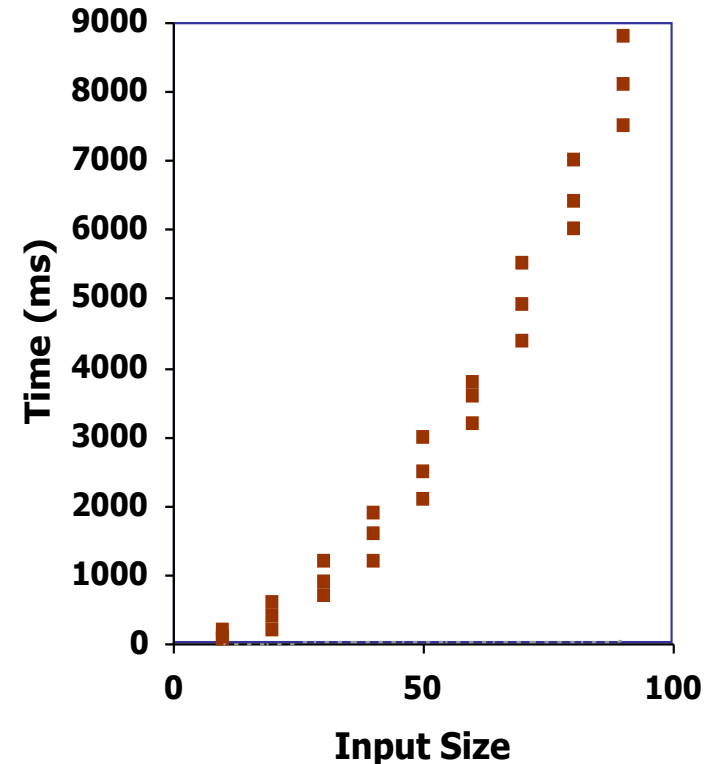
# Running Time

- The running time of an algorithm typically grows with the input size.
- Average case time is often difficult to determine.
- We focus on the **worst case** running time.
  - Easier to analyze
  - Crucial to applications such as games, finance, and robotics



# Experimental Studies

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
  - Use a method like `std::clock()` to get an accurate measure of the actual running time
- Plot the results



# Limitations of Experiments

- Need to implement the algorithm
  - may be difficult
- Experiments done on a limited set of test inputs
  - may not be indicative of running times on other inputs not included in the experiment
- Difficult to compare
  - same hardware and software environments must be used

# Theoretical Analysis

- Uses **pseudocode**, a high-level description of the algorithm
  - no implementation necessary
- Takes into account all possible inputs
- Characterizes running time by  $f(n)$ , a **function of the input size  $n$** 
  - allows us to evaluate the speed of an algorithm independent of hardware/software environment

# Pseudocode

- Mixture of natural language and high-level programming constructs that describe the main ideas behind an algorithm implementation
- Preferred notation for describing algorithms
- Hides program design issues

```
Algorithm arrayMax(A, n)  
  Input array A of n integers  
  Output maximum element of A  
  
  currentMax ← A[0]  
  for i ← 1 to n − 1 do  
    if A[i] > currentMax then  
      currentMax ← A[i]  
  return currentMax
```

# Pseudocode Details

- Control flow
  - **if ... then ... [else ...]**
  - **while ... do ...**
  - **repeat ... until ...**
  - **for ... do ...**
  - Indentation replaces braces
- Method declaration

**Algorithm** *method* (*arg* [, *arg...*])

**Input** ...

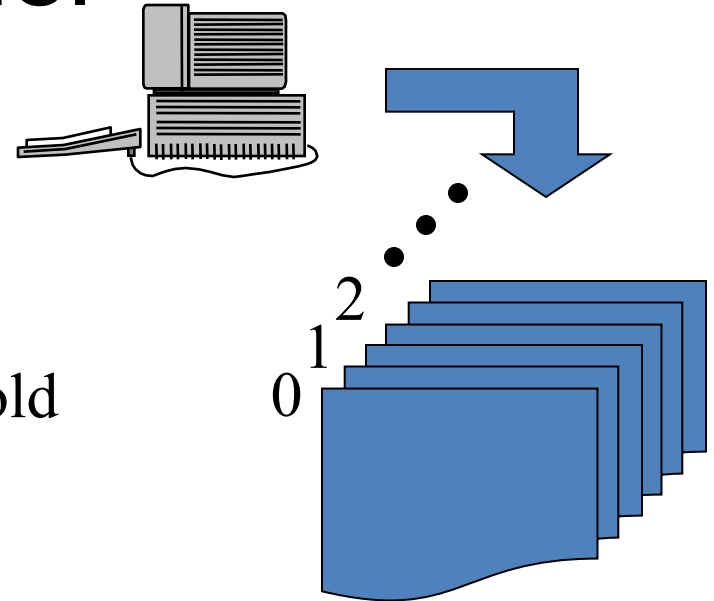
**Output** ...
- Method call

*var.method* (*arg* [, *arg...*])
- Return value

**return** *expression*
- Expressions
  - ← Assignment (like = in C++)
  - = Equality testing (like == in C++)
  - n*<sup>2</sup> Superscripts and other mathematical formatting allowed

# The Random Access Machine (RAM) Model

- Views a computer as:
  - a **CPU**, with
  - a potentially **unbounded** bank of **memory** cells, each of which can hold an arbitrary number or character



Memory cells are numbered and accessing any cell in memory takes unit time.

**Random Access** refers to ability of CPU to access arbitrary memory cell with one **primitive operation**



# Primitive Operations

- **Basic computations** performed by an algorithm
  - Identifiable in pseudocode
  - Largely independent from the programming language
  - Exact definition not important (we'll see why later)
- Assumed to take a **constant amount of time** in the RAM model
- Includes:
  - evaluating an expression
  - indexing into an array
  - assigning a value to a variable
  - calling a method
  - returning from a method

# 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

<b>Algorithm</b> <i>arrayMax</i> ( <i>A</i> , <i>n</i> )	<u># operations</u>
<i>currentMax</i> ← <i>A</i> [0]	2
<b>for</b> <i>i</i> ← 1 <b>to</b> <i>n</i> - 1 <b>do</b>	2 + <i>n</i>
<b>if</b> <i>A</i> [ <i>i</i> ] > <i>currentMax</i> <b>then</b>	2( <i>n</i> - 1)
<i>currentMax</i> ← <i>A</i> [ <i>i</i> ]	2( <i>n</i> - 1)
{ increment counter <i>i</i> }	2( <i>n</i> - 1)
<b>return</b> <i>currentMax</i>	1
	-----
	<b>7<i>n</i> - 1</b>

# Estimating Running Time

- Algorithm *arrayMax* executes  $7n - 1$  primitive operations in the **worst 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 *arrayMax*. Then
$$a(7n - 1) \leq T(n) \leq b(7n - 1)$$

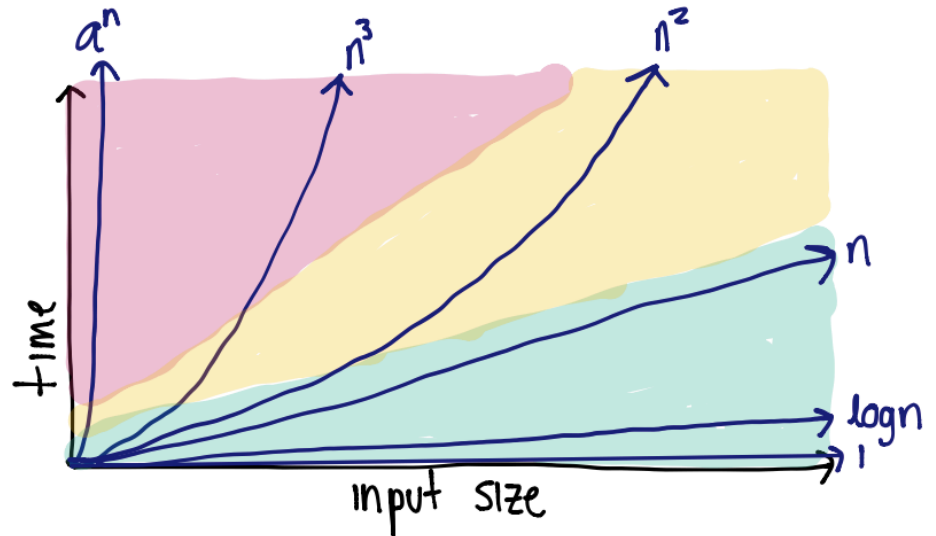
Hence, the running time  $T(n)$  is bounded by two linear functions.

# 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 *arrayMax*

# Growth Rates

- Constant  $\approx 1$
- Logarithmic  $\approx \log n$
- Linear  $\approx n$
- Quadratic  $\approx n^2$
- Cubic  $\approx n^3$
- Polynomial  $\approx n^k$  (for  $k \geq 1$ )
- Exponential  $\approx a^n$  ( $a > 1$ )



Growth rate is not affected by

- constant factors or
- lower-order terms

Ex:  $10^2n + 10^5$  is a **linear** function

Ex:  $10^5n^2 + 10^8n$  is a **quadratic** function

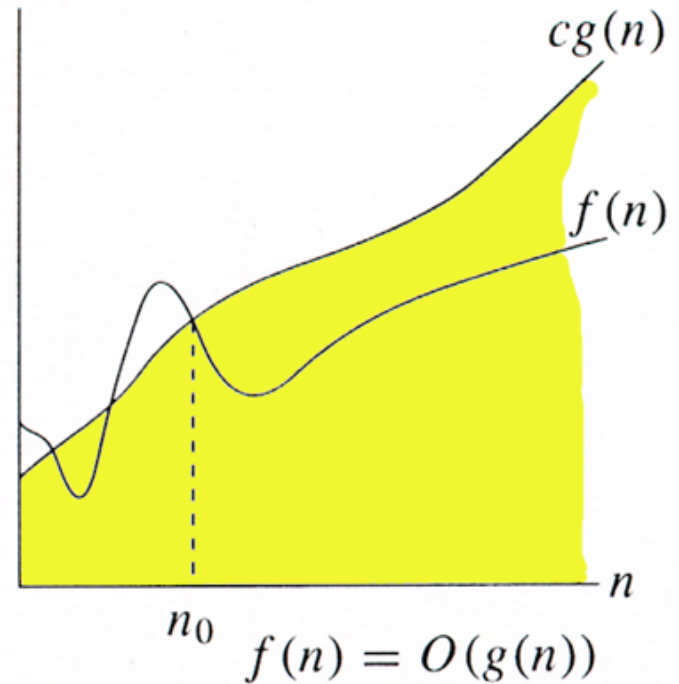
# Asymptotic Complexity

- Worst case running time of an algorithm as a function of input size  $n$  **for large  $n$** .
- Expressed using only the **highest-order term** in the expression for the exact running time.
  - Instead of exact running time, say  $O(n^2)$
- Written using **asymptotic notation** ( $\mathcal{O}$ ,  $\Omega$ ,  $\Theta$ ,  $o$ ,  $\omega$ )
  - Ex:  $f(n) = O(n^2)$
  - Describes how  $f(n)$  grows in comparison to  $n^2$
- The notations describe different rate-of-growth relations between the defining function and the defined **set** of functions

# O-notation

For functions  $g(n)$ , we define  $O(g(n))$ , **big-O** of  $g(n)$ , as the set:

$$O(g(n)) = \{ f(n) : \\ \exists \text{ positive constants } c \text{ and } n_0, \\ \text{such that } \forall n \geq n_0 \\ \text{we have } 0 \leq f(n) \leq cg(n) \}$$



Technically,  $f(n) \in O(g(n))$ .  
Older usage,  $f(n) = O(g(n))$ .

*Intuitively*: Set of all functions whose *rate of growth* is the same as or lower than that of  $g(n)$ .

$g(n)$  is an *asymptotic upper bound* for  $f(n)$

# Examples

$O(g(n)) = \{ f(n) : \exists \text{ positive constants } c \text{ and } n_0, \text{ such that } \forall n \geq n_0, \text{ we have } 0 \leq f(n) \leq cg(n) \}$

- $O(n)$

- $f(n) = 2n + 10$

- $f(n) = n + 1$

- $f(n) = 10000n$

- $f(n) = 10000n + 300$

- $O(n^2)$

- $f(n) = n^2 + 1$

- $f(n) = n^2 + n$

- $f(n) = 10000n^2 + 10000n + 300$

- $f(n) = n^{1.99}$

- The function  $n^2$  is **not**  $O(n)$

- the inequality  $n^2 \leq cn$  cannot be satisfied since  $c$  is constant



# Big-Oh Rules

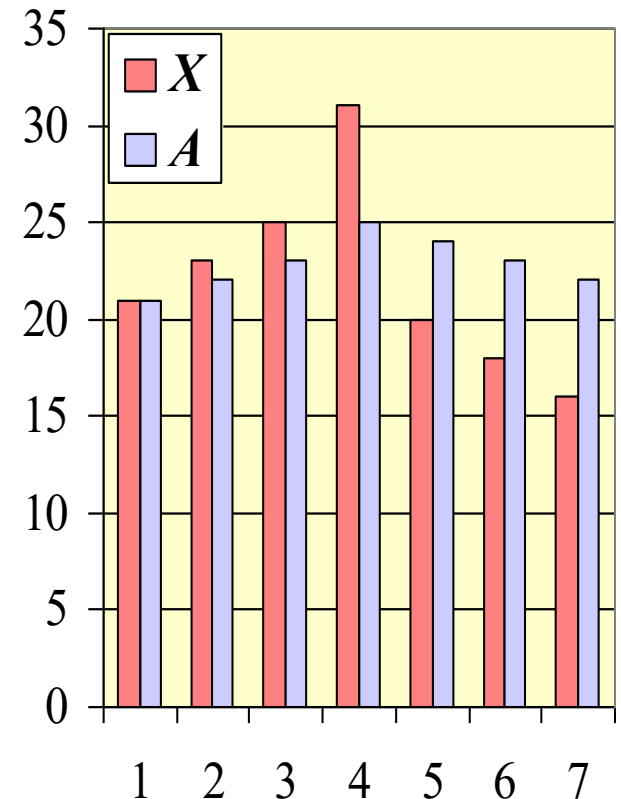
- Drop lower-order terms
  - Ex: if  $f(n)$  is a polynomial of degree  $d$ , then  $f(n)$  is  $O(n^d)$
- Drop constant factors, using the simplest expression of the class
  - Say “ $3n + 5$  is  $O(n)$ ” instead of “ $3n + 5$  is  $O(3n)$ ”
- Use the smallest possible class of functions
  - Say “ $2n$  is  $O(n)$ ” instead of “ $2n$  is  $O(n^2)$ ”
- See Theorem 1.7 in your book

# Asymptotic Algorithm Analysis

- The **asymptotic analysis** of an algorithm determines the running time in big-Oh notation
- To perform the asymptotic analysis
  - Find the **worst-case** number of primitive operations executed as a function of the input size
  - We express this function with **big-Oh notation**
- Ex:
  - *arrayMax* executes at most  $7n - 1$  primitive operations
  - *arrayMax* “runs in  $O(n)$  time”
- Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations

# Ex: 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  $X$ :  
$$A[i] = (X[0] + X[1] + \dots + X[i]) / (i + 1)$$
- Prefix average has applications in economic and statistics



# Prefix Averages V1 $O(n^2)$ - Quadratic!

The following algorithm computes prefix averages by applying the definition

**Algorithm** *prefixAverages1*( $X, n$ )

**Input** array  $X$  of  $n$  integers

**Output** array  $A$  of prefix averages of  $X$

$A \leftarrow$  new array of  $n$  integers

**for**  $i \leftarrow 0$  to  $n - 1$  **do**

$s \leftarrow X[0]$

**for**  $j \leftarrow 1$  to  $i$  **do**

$s \leftarrow s + X[j]$

$A[i] \leftarrow s / (i + 1)$

**return**  $A$

rough # operations

$n$

$n$

$n$

$1 + 2 + \dots + (n-1)$

$1 + 2 + \dots + (n-1)$

$n$

$1$

# Prefix Averages V2

$O(n)$  - Linear!

- ◆ The following algorithm computes prefix averages by keeping a running sum

**Algorithm** *prefixAverages2*( $X, n$ )

**Input** array  $X$  of  $n$  integers

**Output** array  $A$  of prefix averages of  $X$

rough # operations

$A \leftarrow$  new array of  $n$  integers

$n$

$s \leftarrow 0$

$1$

**for**  $i \leftarrow 0$  **to**  $n - 1$  **do**

$n$

$s \leftarrow s + X[i]$

$n$

$A[i] \leftarrow s / (i + 1)$

$n$

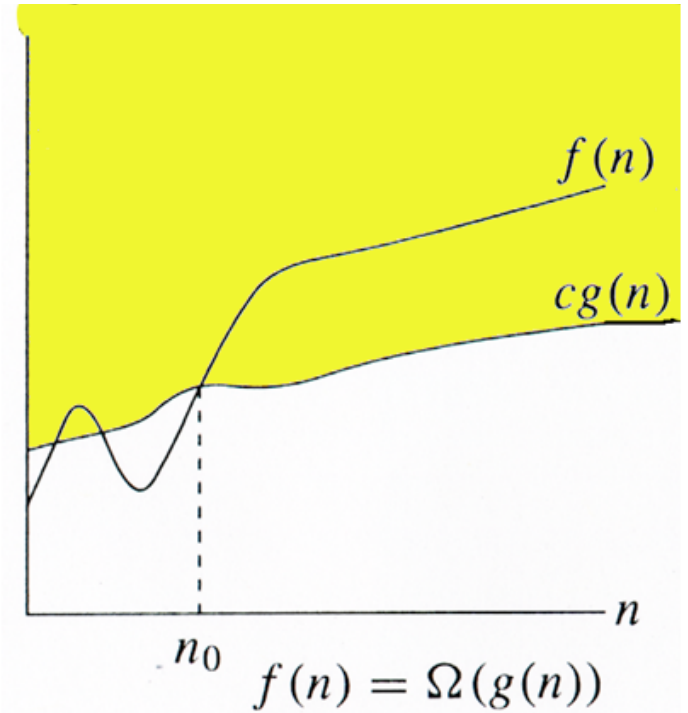
**return**  $A$

$1$

# $\Omega$ -notation

For functions  $g(n)$ , we define  $\Omega(g(n))$ , **big-Omega** of  $g(n)$ , as the set:

$$\Omega(g(n)) = \{ f(n) : \\ \exists \text{ positive constants } c \text{ and } n_0, \\ \text{such that } \forall n \geq n_0 \\ \text{we have } 0 \leq cg(n) \leq f(n) \}$$



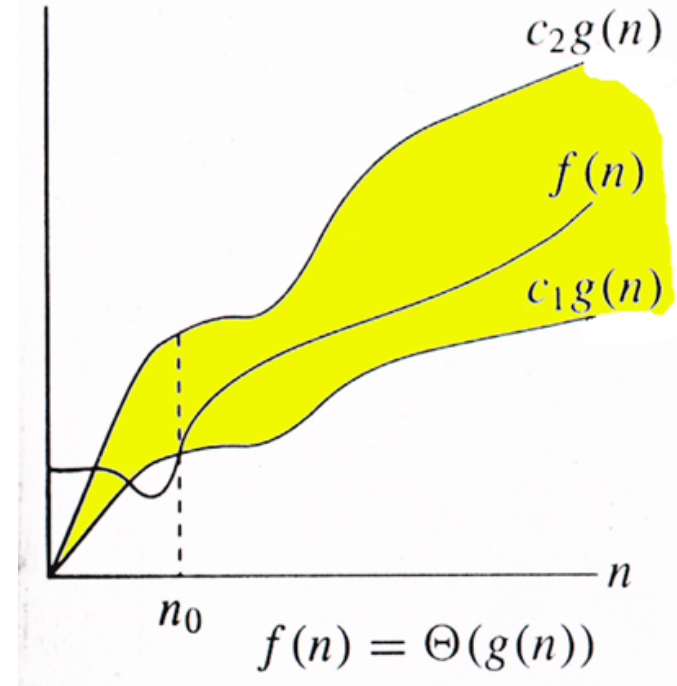
*Intuitively*: Set of all functions whose *rate of growth* is the same as or higher than that of  $g(n)$ .

$g(n)$  is an *asymptotic lower bound* for  $f(n)$

# $\Theta$ -notation

For functions  $g(n)$ , we define  $\Theta(g(n))$ , **big-Theta** of  $g(n)$ , as the set:

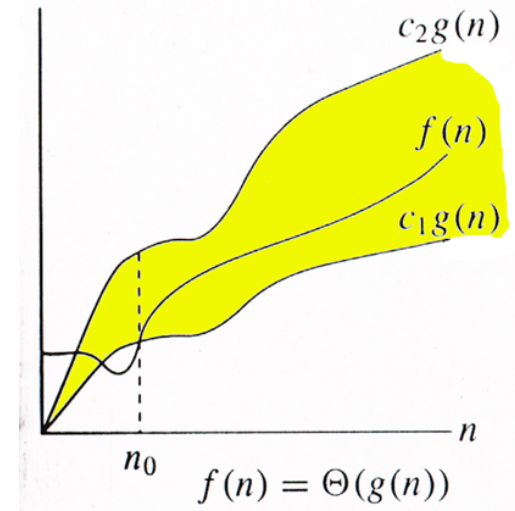
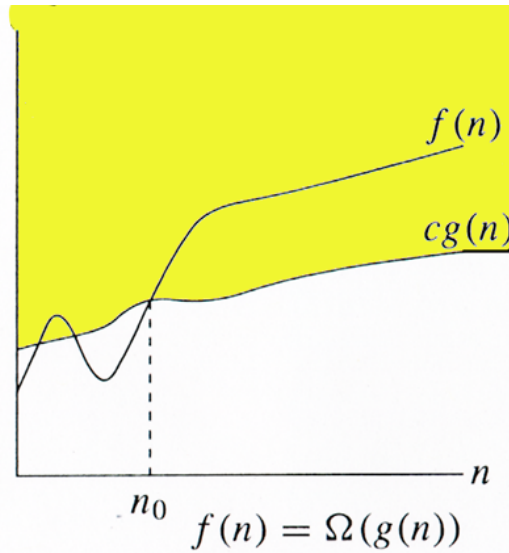
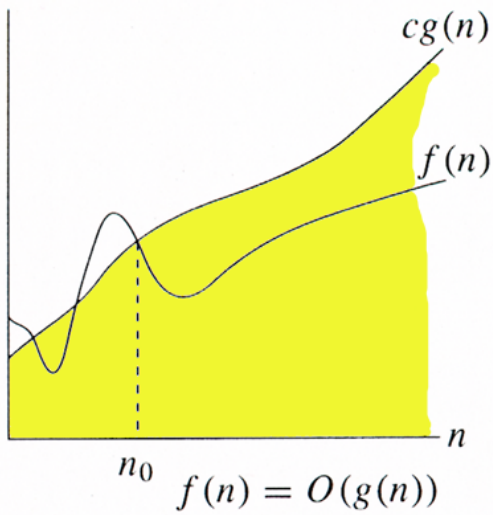
$$\Theta(g(n)) = \{ f(n) : \\ \exists \text{ positive constants } c_1, c_2, \text{ and } n_0, \\ \text{such that } \forall n \geq n_0 \\ \text{we have } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \}$$



*Intuitively*: Set of all functions that have the same *rate of growth* as  $g(n)$ .

$g(n)$  is an *asymptotically tight bound* for  $f(n)$

# Relationship between $O$ , $\Omega$ , $\Theta$





# Relatives of $O$ and $\Omega$

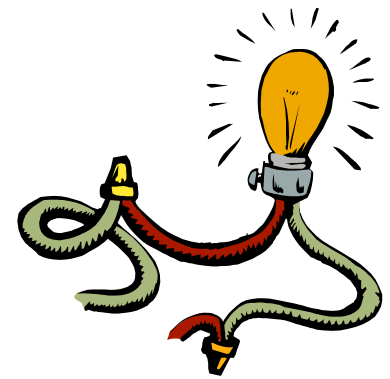
## Little-oh

- $f(n)$  is  $o(g(n))$  if  $\forall c > 0, \exists n_0 \geq 0$  such that  $f(n) \leq cg(n)$  for  $n \geq n_0$

## Little-omega

- $f(n)$  is  $\omega(g(n))$  if  $\forall c > 0, \exists n_0 \geq 0$  such that  $cg(n) \leq f(n)$  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  $g(n)$

## Big-Omega

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

## Big-Theta

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

## little-oh

- $f(n)$  is  $o(g(n))$  if  $f(n)$  is asymptotically **strictly less** than  $g(n)$

## little-omega

- $f(n)$  is  $\omega(g(n))$  if  $f(n)$  is asymptotically **strictly greater** than  $g(n)$

# Math you need to review



- ◆ Summations (Sec. 1.3.1)
- ◆ Logarithms and Exponents (Sec. 1.3.2)

$$\log_b a = c \quad \text{if} \quad a = b^c$$

## properties of logarithms:

$$\begin{aligned}\log_b(xy) &= \log_b x + \log_b y \\ \log_b(x/y) &= \log_b x - \log_b y \\ \log_b x^a &= a \log_b x \\ \log_b a &= \log_x a / \log_x b\end{aligned}$$

## properties of exponentials:

$$\begin{aligned}a^{(b+c)} &= a^b a^c \\ a^{bc} &= (a^b)^c \\ a^b / a^c &= a^{(b-c)} \\ b &= a^{\log_a b} \\ b^c &= a^{c \cdot \log_a b}\end{aligned}$$

- ◆ Proof techniques (Sec. 1.3.3)
- ◆ Basic probability (Sec. 1.3.4)