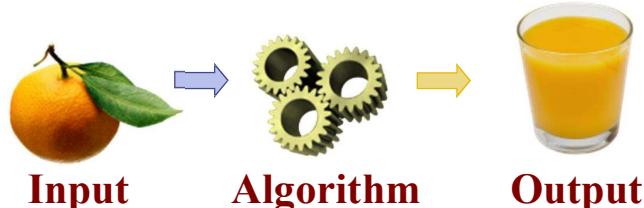
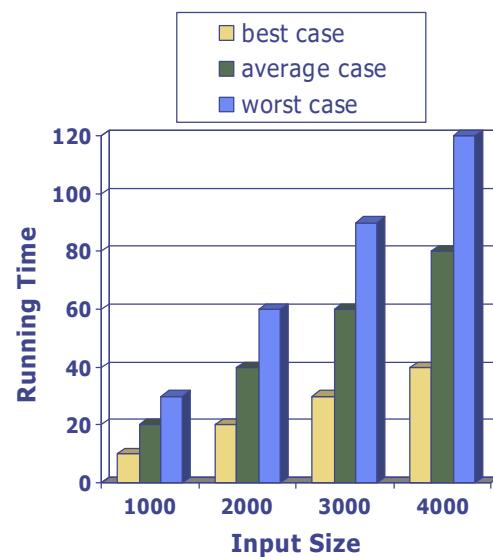


# Analysis of Algorithms



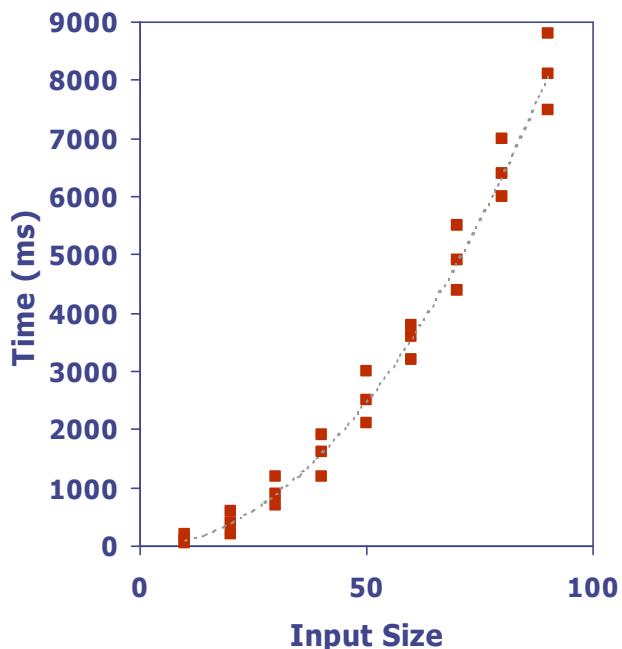
## Running Time

- ❑ Most algorithms transform input objects into output objects.
- ❑ 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, noting the time needed:
- ❑ Plot the results



```
1 long startTime = System.currentTimeMillis();           // record the starting time
2 /* (run the algorithm) */
3 long endTime = System.currentTimeMillis();           // record the ending time
4 long elapsed = endTime - startTime;               // compute the elapsed time
```

## Limitations of Experiments

- ❑ It is necessary to implement the algorithm, which may be difficult
- ❑ Results may not be indicative of the running time on other inputs not included in the experiment.
- ❑ In order to compare two algorithms, the same hardware and software environments must be used



# Theoretical Analysis



- ❑ Uses a high-level description of the algorithm instead of an implementation
- ❑ Characterizes running time as a function of the input size,  $n$ 
  - > The running time of an algorithmn typically grows with the input size.
- ❑ Takes into account all possible inputs
- ❑ Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

## Pseudocode

- ❑ High-level description of an algorithm
- ❑ More structured than English prose
- ❑ Less detailed than a program
- ❑ Preferred notation for describing algorithms
- ❑ Hides program design issues

# Pseudocode Details

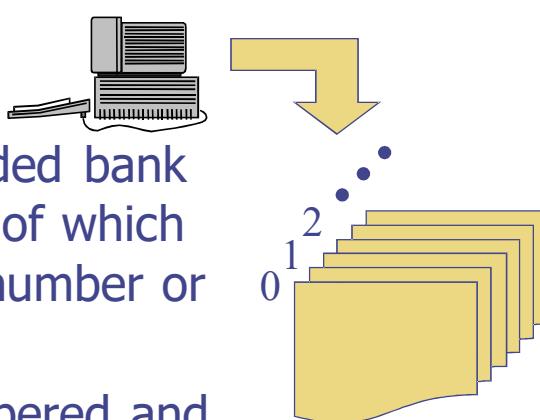


- Control flow
  - **if ... then ... [else ...]**
  - **while ... do ...**
  - **repeat ... until ...**
  - **for ... do ...**
  - Indentation replaces braces
- Method declaration  
**Algorithm** *method (arg [, arg...])*  
**Input** ...  
**Output** ...
  - Method call  
*method (arg [, arg...])*
  - Return value  
*return expression*
  - Expressions:  
 ← Assignment  
 = Equality testing
  - $n^2$  Superscripts and other mathematical formatting allowed

## The Random Access Machine (RAM) Model

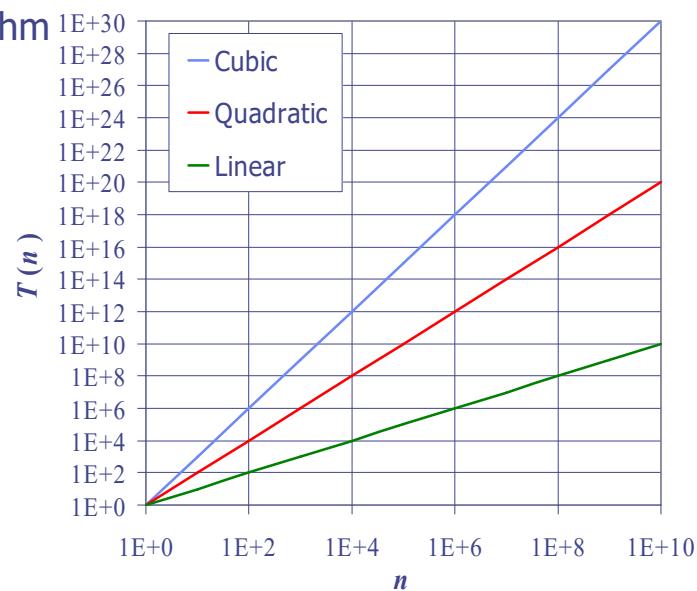
A RAM consists of

- A CPU
- An 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



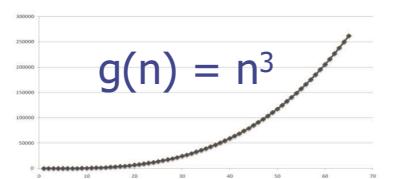
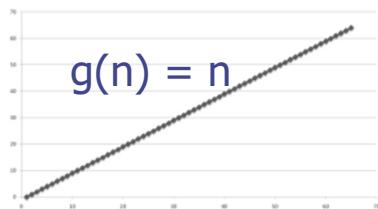
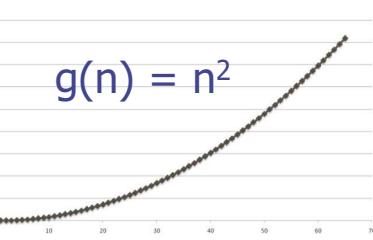
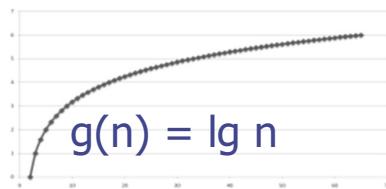
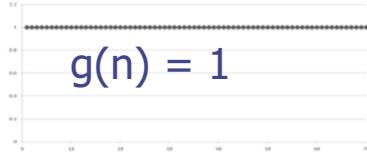
# Seven Important Functions

- Seven functions that often appear in algorithm analysis:
  - Constant  $\approx 1$
  - Logarithmic  $\approx \log n$
  - Linear  $\approx n$
  - N-Log-N  $\approx n \log n$
  - Quadratic  $\approx n^2$
  - Cubic  $\approx n^3$
  - Exponential  $\approx 2^n$
- 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.



# Primitive Operations



- ❑ Basic computations performed by an algorithm
  - ❑ Identifiable in pseudocode
  - ❑ Largely independent from the programming language
  - ❑ Exact definition not important (we will see why later)
  - ❑ Assumed to take a constant amount of time in the RAM model
- ❑ Examples:
    - Evaluating an expression  
 $a*b+c$
    - Assigning a value to a variable  
 $a = b$
    - Indexing into an array  
 $A[i]$
    - Calling a method  
`facto(10);`
    - Returning from a method  
`return 10`

## 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  /** Returns the maximum value of a nonempty array of numbers. */
2  public static double arrayMax(double[ ] data) {
3      int n = data.length;
4      double currentMax = data[0];           // assume first entry is biggest (for now)
5      for (int j=1; j < n; j++)           // consider all other entries
6          if (data[j] > currentMax)        // if data[j] is biggest thus far...
7              currentMax = data[j];         // record it as the current max
8      return currentMax;
9  }

```

- ❑ Step 3: 2 ops, 4: 2 ops, 5:  $2n$  ops,  
6: \_\_\_\_\_ ops, 7: 0 to \_\_\_\_\_ ops, 8: 1 op

```

1  /** Returns the maximum value of a nonempty array of numbers. */
2  public static double arrayMax(double[ ] data) {
3      int n = data.length;
4      double currentMax = data[0];           // assume first entry is biggest (for now)
5      for (int j=1; j < n; j++)            // consider all other entries
6          if (data[j] > currentMax)        // if data[j] is biggest thus far...
7              currentMax = data[j];         // record it as the current max
8      return currentMax;
9  }

```

## Best Case

Step3: 2

Step4: 2

Step5:  $2n$

Step6:  $2(n-1)$

Step7: 0

Step8: 1

## Worst Case

2

2

$2n$

$2(n-1)$

$2(n-1)$

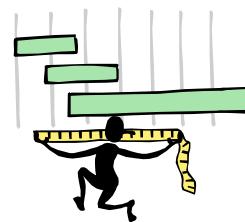
1

$4n+3$

$6n+1$

13

## Estimating Running Time



- Algorithm **arrayMax** executes  $6n + 1$  primitive operations in the worst case,  $4n + 3$  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 the running time of **arrayMax**. Then
 
$$a(4n + 3) \leq T(n) \leq b(6n + 1)$$
- Hence, the running time  $T(n)$  is bounded by two linear functions

# Growth Rate of Running Time

$$a(4n + 3) \leq T(n) \leq b(6n + 1)$$

- 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**

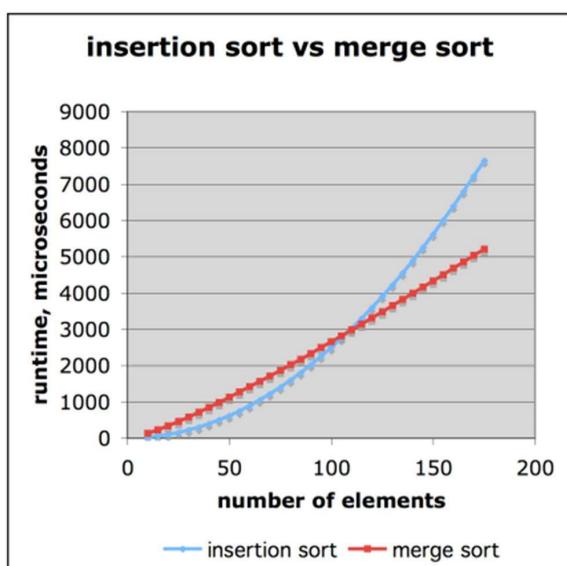


Analysis of Algorithms

15

Slide by Matt Stallmann  
included with permission.

## Comparison of Two Algorithms

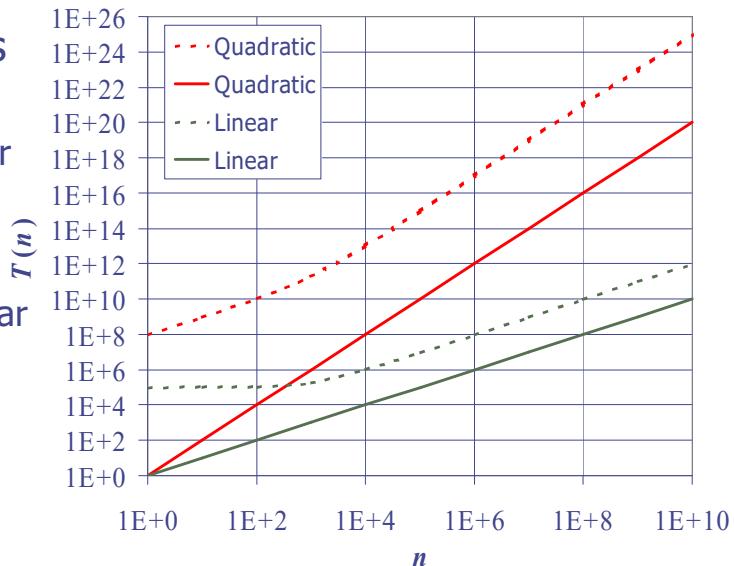


insertion sort is  $n^2 / 4$   
 merge sort is  $2 n \lg n$   
 sort a million items?  
 insertion sort takes roughly **70 hours**  
 while  
 merge sort takes roughly **40 seconds**

This is a slow machine, but if 100 x as fast then it's **40 minutes** versus less than **0.5 seconds**

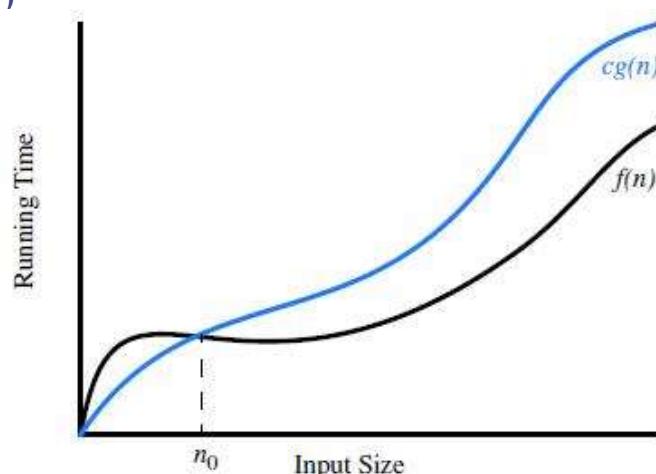
# Constant Factors

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



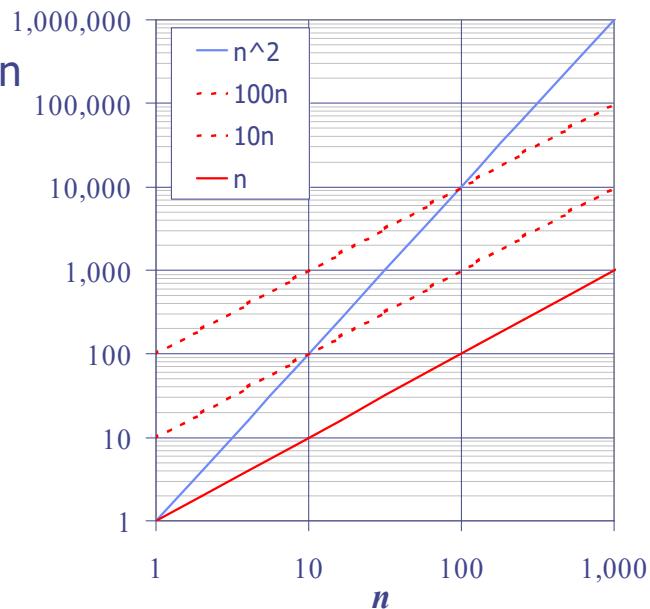
# Big-Oh Notation

- Given functions  $f(n)$  and  $g(n)$ , we say that  $f(n)$  is  $O(g(n))$  if there are positive constants  $c$  and  $n_0$  such that  $f(n) \leq cg(n)$  for  $n \geq n_0$
- Example:  $2n + 10$  is  $O(n)$ 
  - $2n + 10 \leq cn$
  - $(c - 2)n \geq 10$
  - $n \geq 10/(c - 2)$
  - Pick  $c = 3$  and  $n_0 = 10$



# Big-Oh Example

- 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



# More Big-Oh Examples



- $7n - 2$

$7n-2$  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  $n_0 = 1$

- $3n^3 + 20n^2 + 5$

$3n^3 + 20n^2 + 5$  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  $n_0 = 21$

- $3 \log n + 5$

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

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 **under 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  $f(n)$  is a polynomial of degree  $d$ , then  $f(n)$  is  $O(n^d)$ , i.e.,
  1. Drop lower-order terms
  2. Drop constant factors
- Use the smallest possible class of functions
  - Say “ $2n$  is  $O(n)$ ” instead of “ $2n$  is  $O(n^2)$ ”
- Use the simplest expression of the class
  - Say “ $3n + 5$  is  $O(n)$ ” instead of “ $3n + 5$  is  $O(3n)$ ”

# 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 **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

# 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

1	2	3	4	5
1	1.5	2	2.5	3

# Prefix Averages (Quadratic)

- ◆ The following algorithm computes prefix averages in quadratic time 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**

$s \leftarrow 0$

**for**  $j \leftarrow 0$  **to**  $i$  **do**

$s \leftarrow s + X[j]$

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

**return**  $A$

*prefixAverages1* runs  
in  $O(n^2)$  time

25

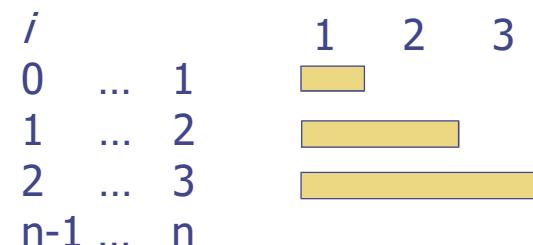
# Arithmetic Progression

- 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

```

for  $i \leftarrow 0$  to  $n - 1$  do
     $s \leftarrow 0$ 
    for  $j \leftarrow 0$  to  $i$  do
         $s \leftarrow s + X[j]$ 
     $A[i] \leftarrow s / (i + 1)$ 
return  $A$ 

```



# Prefix Averages (Linear)

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

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

```

Input array  $X$  of  $n$  integers
Output array  $A$  of prefix averages of  $X$ 
 $A \leftarrow$  new array of  $n$  integers
 $s \leftarrow 0$ 
for  $i \leftarrow 0$  to  $n - 1$  do
     $s \leftarrow s + X[i]$ 
     $A[i] \leftarrow s / (i + 1)$ 
return  $A$ 

```

- ◆ Algorithm *prefixAverages2* runs in  $O(n)$  time

27

## Math you need to Review



- Summations
- Powers
- Logarithms
- Proof techniques
- Basic probability

- Properties of powers:

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

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

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

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

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

- Properties of logarithms:

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

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

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

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

# Big-Oh Rules (Sum Rule)

**Theorem** If  $f_1(n) = O(g_1(n))$  and  $f_2(n) = O(g_2(n))$ , then

$$f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n))).$$

```

int sum = 0;
for (int i = 0; i < m; i++)
    sum = sum + i;
} } O(m)

int product = 1;
for (int j = 0; j < n; j++)
    product = product * j;
} } O(n)
O(max(m, n))

```

Analysis of Algorithms

29

# Big-Oh Rules (Product Rule)

**Theorem** If  $f_1(n) = O(g_1(n))$  and  $f_2(n) = O(g_2(n))$ , then

$$f_1(n) \times f_2(n) = O(g_1(n) \times g_2(n)).$$

```

for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        a[i][j] = i + j
    }
}
= O(n * m)
} } O(n) } } O(n)
} } = O(n) } } O(1)
} } = O(n) * O(m)

```

30

# 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

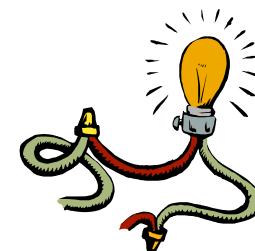
$$f(n) \geq c g(n) \text{ for } n \geq n_0$$

## 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

$$c'g(n) \leq f(n) \leq c''g(n) \text{ for } n \geq n_0$$

# 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  $g(n)$

## BIG IDEA! Asymptotic Analysis

1. Ignore machine dependent constants
2. Look at growth of running time  $T(n)$  as  $n \rightarrow \infty$

Let  $n$  be size of program's input.

Let  $f(n)$  be function for running time

Let  $g(n)$  be another function ~~—~~ preferably simple.

Let  $f$  and  $g$  are non-negative functions over non-negative integers.

**Definition:** [Big-Oh]  $f(n) = O(g(n))$  iff there exist positive constants  $c$  and  $n_0$  such that  $f(n) \leq cg(n)$  for all  $n, n \geq n_0$ .

33

**Example:**

The function  $T(n) = 3n^3 + 2n^2$  is  $O(n^3)$ .

To see this, let  $n_0 = 0$  and  $c = 5$ .  $3n^3 + 2n^2 \leq 5n^3$

You need to show that for all  $n \geq 0$ ,

$$3n^3 + 2n^2 \leq 5n^3 \Rightarrow n^2 \leq n^3$$

if  $n = 0, 0 \leq 0$

otherwise,  $1 \leq n \quad \therefore 3n^3 + 2n^2 = O(n^3) \square$

**Example:**

$T(n) = 3^n$  is not  $O(2^n)$ .

Suppose that there were constants  $n_0$  and  $c$  such that for all  $n \geq n_0$ ,

we had  $3^n \leq c2^n$ . Then  $c \geq (3/2)^n$  for any  $n \geq n_0$ . But

$\lim_{n \rightarrow \infty} (3/2)^n = \infty$ . Therefore, no constant  $c$  can exceed  $(3/2)^n$  for all

$n$ .  $\square$

- The statement  $f(n) = O(g(n))$  only states that  $g(n)$  is an upper bound on the growth rate of  $f(n)$  for all  $n, n \geq n_0$ .

34

Set definition:

$O(g(n)) = \{ f(n) : \text{there are positive constants } c \text{ and } n_0 \text{ such that } f(n) \leq cg(n) \text{ for all } n \geq n_0 \}$

*Example:*

The  $n = O(n^2)$ ,  $n = O(n^{2.5})$ ,  $n = O(n^3)$ ,  $n = O(2^n)$ .

In order for the statement  $f(n) = O(g(n))$  to be informative,  $g(n)$  should be as small a function of n as one can come up with for which  $f(n) = O(g(n))$ .

$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$  means  $g(n)$  is asymptotically bigger than  $f(n)$  and  $f(n)$  is asymptotically smaller than  $g(n)$ .

$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$  means vice versa.

To prove  $f(n) = O(g(n))$ , we only need to show  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$  or Constant C

Example)

$$\begin{aligned}\lim_{n \rightarrow \infty} \frac{n \log_2 n}{n^2} &= \frac{1}{\ln 2} \lim_{n \rightarrow \infty} \frac{n \ln n}{n^2} = \frac{1}{\ln 2} \lim_{n \rightarrow \infty} \frac{\ln n + 1}{2n} \\ &= \frac{1}{\ln 2} \left( \lim_{n \rightarrow \infty} \frac{\ln n}{2n} + \lim_{n \rightarrow \infty} \frac{1}{2n} \right) = \frac{1}{\ln 2} \lim_{n \rightarrow \infty} \frac{1}{2n} = 0\end{aligned}$$

Therefore,  $n \log_2 n = O(n^2)$ .

37

## Summary

- Programs can be evaluated by comparing their Big-Oh functions.
  - Drop low-order terms and ignore leading constants.
- Ex)  $3n^2 + n \log_2 n = O(n^2)$
- The common Big-Oh functions provide a "yardstick" for classifying different algorithms. Algorithms of the same Big-Oh can be considered as equally good.
  - For example, a program with  $O(\log n)$  is better than one with  $O(n)$ .
- Warnings!

38

- Warnings!

1. Fallacious proof:

$n^2 = O(n)$  Proof: Choose  $c = n$ ,  $n_0 = 1$ . Then  $n^2 \leq n^2$ .

$\Rightarrow$  WRONG!  $c$  must be constant.

2. " $e^{3n} = O(e^n)$  because constant factors don't matter"  $\Rightarrow$

Bigger by a factor of  $e^{2n}$

" $10^n = O(2^n) \Rightarrow$  WRONG!"  $\Rightarrow$  Bigger by a factor of  $5^n$

39

3. Big-Oh notation doesn't always tell whole story.

Big-Oh is meaningful only when  $n$  is sufficiently large ( $n \geq n_0$ ).

This implies that we only care about large size problems.

$$T(n) = n \log_2 n, \quad U(n) = 100n$$

$T(n)$  dominates  $U(n)$  asymptotically.

But, if  $\log_2 n < 50$  in practice, what algorithm would you like to choose?

40

**Definition:** [Big-Theta]  $f(n) = \Theta(g(n))$  iff there exist positive constants  $c_1$  and  $c_2$  and an  $n_0$  such that  $c_1g(n) \leq f(n) \leq c_2g(n)$  for all  $n, n \geq n_0$ .

Example:  $3n^2 + 4n = \Theta(n^2)$

Proof) choose  $c_1 = 3$  and  $c_2 = 7$  and  $n_0 = 0$ . We have  $3n^2 \leq 3n^2 + 4n \leq 7n^2$  for all  $n, n \geq n_0$ .

41

- $\Omega(f(n))$ : Big Omega

**Definition:** [Big-Omega]  $f(n) = \Omega(g(n))$  iff there exist positive constants  $c$  and  $n_0$  such that  $f(n) \geq cg(n)$  for all  $n, n \geq n_0$ .

Example:  $f(n) = 3n+2 \geq 3n$  for all  $n$ , so  $f(n) = \Omega(n)$

42