

# Growth Functions and Asymptotic Analysis

Instructor: Krishna Venkatasubramanian

CSC 212

# Announcements

- **Go to office hours.**
- Next Quiz ---- we shall see
- Same format as today's quiz.

# Running Time of Algorithms

- Running time of algorithm determines how “**quickly**” it executes.
- **Computed based on # of basic steps** in the algorithm that are executed
  - Loops result in repeated computational sets leading to larger running time than those without
- **Size of the inputs can also affect number of basic steps**
  - Sorting longer arrays need more time than shorter ones!
  - In such cases, **size of input usually dictates # of basic steps!**

# Runtime affected by # of Basic Steps

- Two algorithms for performing the same tasks can have different running times **depending upon # of steps it has**
- Here, the size of the input is the same --- **1 number** --- but the presence of loops dictates running time.

```
import time

def SumOfN(n):
    start = time.time()
    theSUM = 0

    for i in range(1,n+1):
        theSUM +=i
    end = time.time()

    return theSUM, end-start

def directSumOfN(n):
    start = time.time()
    theSUM = (n*(n+1))/2
    end = time.time()

    return theSUM,end-start

def main():

    print("Sum of N with for loop")
    for i in range(5):
        print("Sum is %d required %10.7f seconds"%SumOfN(1000000))

    print("Sum of N function direct ")
    for i in range(5):
        print("Sum is %d required %10.7f seconds"%directSumOfN(1000000))
```

w/ FOR LOOP

DIRECT SUMMATION

Sum of N with for loop	
Sum is 500000500000	required 0.0627120 seconds
Sum is 500000500000	required 0.0636330 seconds
Sum is 500000500000	required 0.0593448 seconds
Sum is 500000500000	required 0.0563250 seconds
Sum is 500000500000	required 0.0615969 seconds
Sum of N function direct	
Sum is 500000500000	required 0.0000012 seconds
Sum is 500000500000	required 0.0000000 seconds
Sum is 500000500000	required 0.0000000 seconds
Sum is 500000500000	required 0.0000012 seconds
Sum is 500000500000	required 0.0000007 seconds

# Runtime affected by input size: Insertion Sort (Recap)

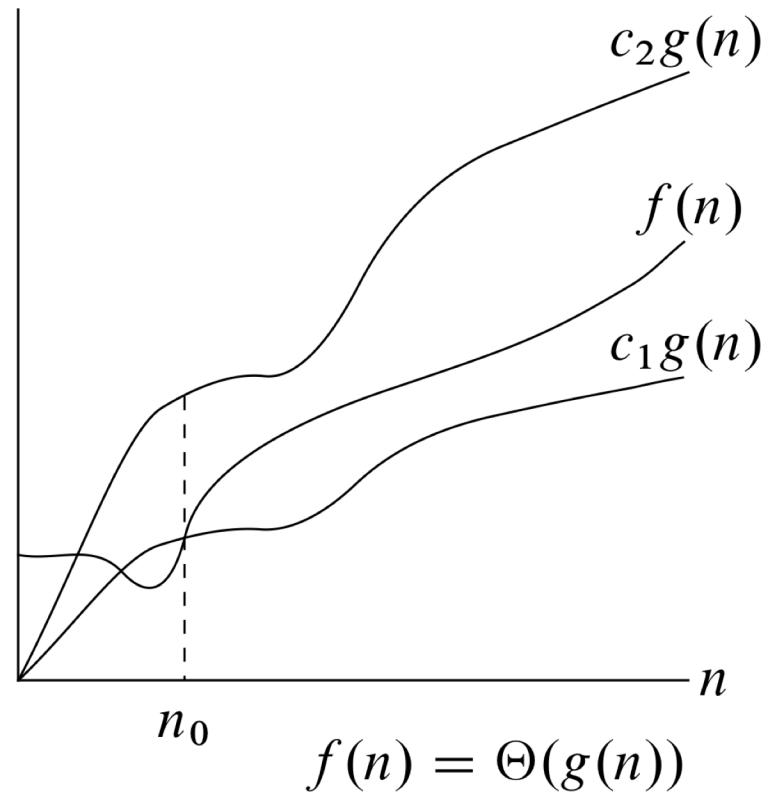
	Assume $n$ elements	Cost	Times
<code>def InsertionSort(A)</code>		$c_1$	$n$
<code>for k in range(1, len(A))</code>		$c_2$	$n-1$
<code>key = A[k]</code>		$c_4$	$n-1$
<code>i = k - 1</code>		$c_5$	$\sum_2^n t_j$ / WHY?
<code>while i &gt; 0 and A[i] &gt; key</code>		$c_6$	$\sum_2^n (tj - 1)$
<code>A[i+1] = A[i]</code>		$c_7$	$\sum_2^n (tj - 1)$
<code>i = i - 1</code>		$c_8$	$n-1$
<code>A[i+1] = key</code>			

$$T(n)$$

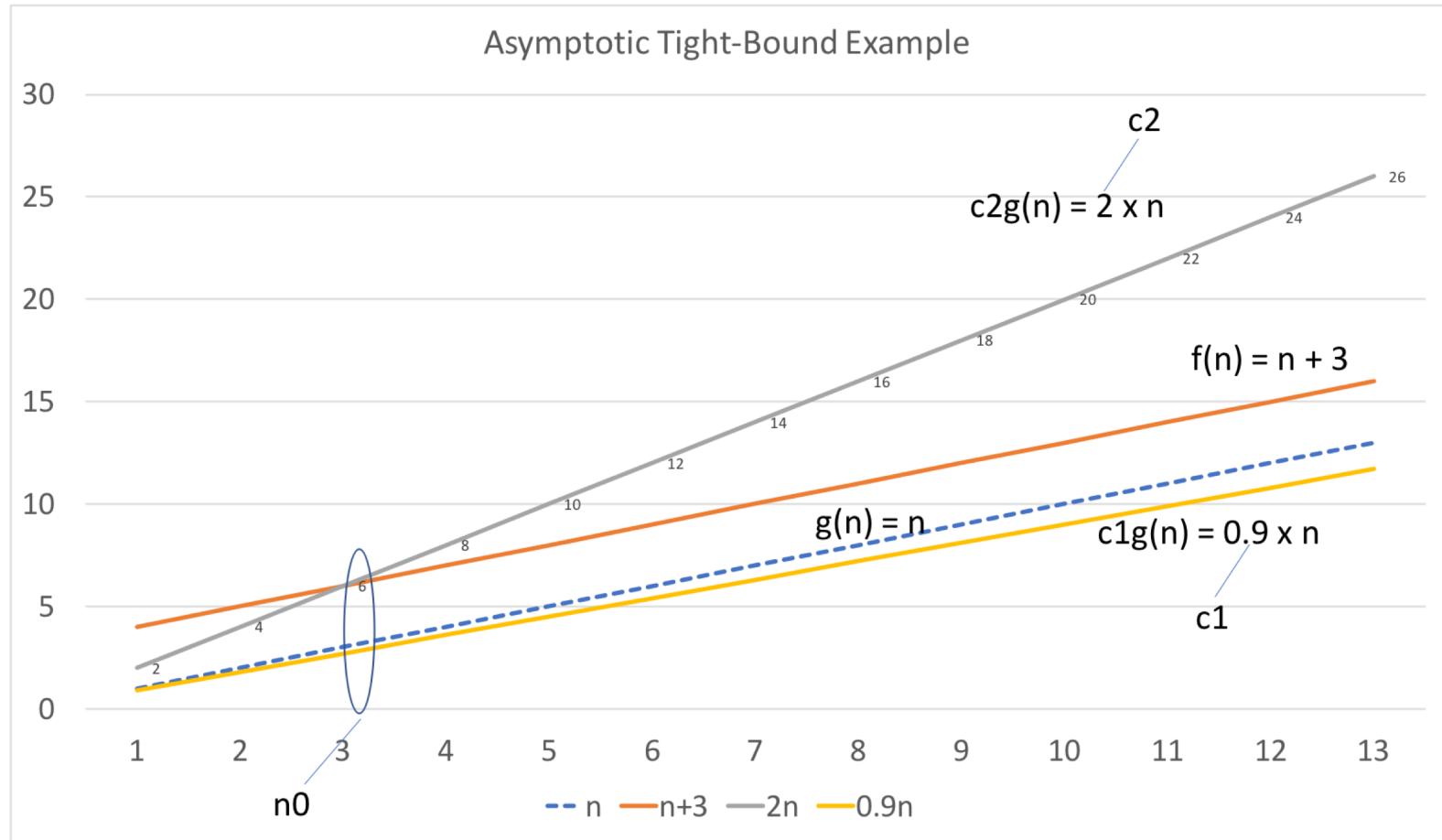
$$= c_1 * n + c_2(n - 1) + c_4(n - 1) + c_5 \sum_2^n t_j + c_6 \sum_2^n (tj - 1) + c_7 \sum_2^n (tj - 1) + c_8(n-1)$$

# $\Theta$ -notation

- **Definition:** For a given function  $g(n)$ ,  $\Theta(g(n))$  is a **set of functions** such that
  - $\Theta(g(n)) = \{ f(n) : \text{there exists positive constants } c_1, c_2, \text{ and } n_0 \text{ s.t. } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \}$
- This is called an asymptotic tight-bound for  $f(n)$ 
  - Really,  $f(n) \in \Theta(g(n))$
- For all values of  $n \geq n_0$  the value of  $f(n) = g(n)$  within a constant factor
- Focus on **large values of  $n$**



# $\Theta$ -notation: Graphical Example



# $\Theta$ -notation: Numerical Example

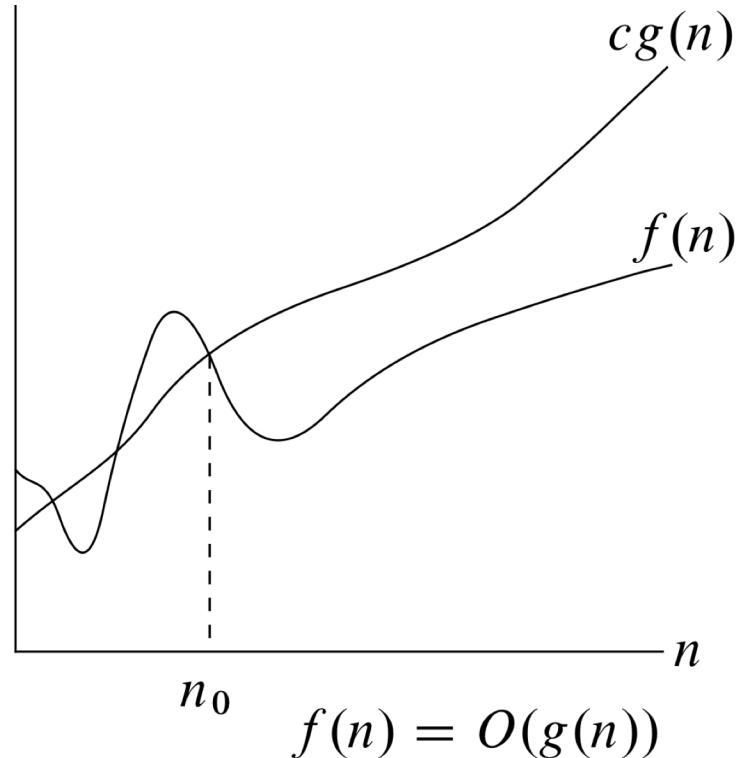
- Assume  $f(n) = 1/2n^2 - 3n$
  - We say  $f(n) = \Theta(n^2)$ , if this is true then
    - $c_1 n^2 \leq 1/2n^2 - 3n \leq c_2 n^2$
    - $c_1 \leq 1/2 - 3/n \leq c_2$
  - The right inequality is true for  $n \geq 1$  and  $c_2 \geq 1/2$
  - The left inequality is true for  $n \geq 7$  and  $c_1 \leq 1/14$
  - Thus if we choose
    - $c_1 = 1/14$ ,  $c_2 = 1/2$ , and  $n_0 = 7$  we can make the inequality true
  - Thus  $f(n) = \Theta(n^2)$
  - Note, other  $c_1$ ,  $c_2$ , and  $n_0$  may also exist that make the inequality true
  - Suffice it to say, that we can find one groups of values
- Remember:  
 $c_1, c_2$  and  $n$  are positive constants

# $\Theta$ -notation: Asymptotic Tight-bound

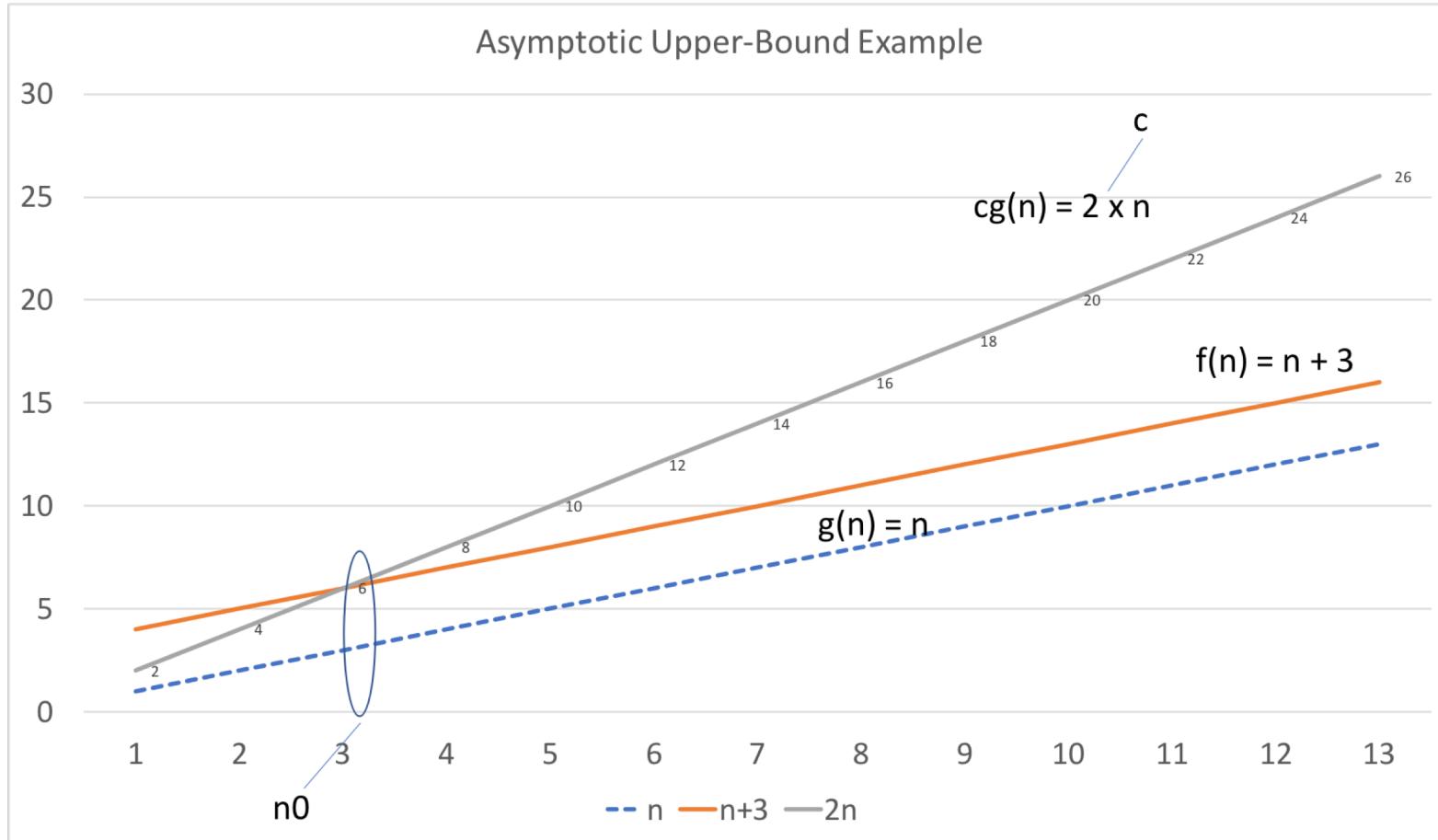
- Asymptotic means for large n
- Tight-bound because we have found the function that describes the algorithm's running time to with a constant multiple above and below
- In the running time of an algorithm, lower order terms are ignored.
  - For large values of n, the lower order terms become minuscule compared to the highest-order term
  - E.g., For  $T(n) = an^2 + bn + c$ , the **value of  $n^2$  will dominate** values of  $b*n$  or  $c$  or  $a$  for large values of n
- More generally, for any polynomial
$$p(n) = \sum_{i=0}^d a_i n^i \text{ where } a_i \text{ is a constant and } a_d > 0,$$
$$\mathbf{p(n) = \Theta(n^d)}$$
- Similarly, for a zero—degree polynomial  $q(n)$  or a constant function --- e.g., a given algorithm step
$$\mathbf{q(n) = \Theta(n^0) = \Theta(1)}$$

# $O$ -notation

- **Definition:** For a given function  $g(n)$ ,  $O(g(n))$  is a **set of functions** such that
  - $O(g(n)) = \{ f(n) : \text{there exists positive constants } c, \text{ and } n_0 \text{ s.t. } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \}$
- This is called an asymptotic upper-bound for  $f(n)$
- For all values of  $n \geq n_0$  the value of  $f(n)$  is always  $\leq g(n)$  within a constant factor
- Focus on **large values of  $n$**



# $O$ -notation: Graphical Example



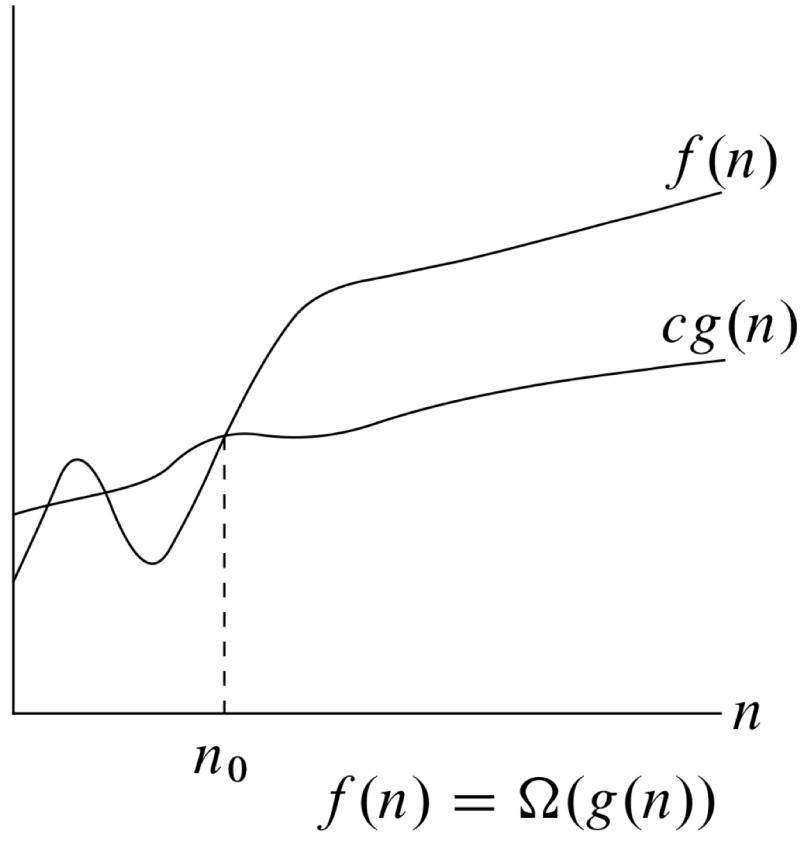
# $O$ -notation: Numerical Example

- Assume  $f(n) = 1/2n^2 - 3n$
- We say  $f(n) = O(n^2)$ , if this is true then
  - $0 \leq 1/2n^2 - 3n \leq cn^2$
  - $0 \leq 1/2 - 3/n \leq c$
- The right inequality is true for  $n \geq 1$  and  $c \geq 1/2$
- The left inequality is true for  $n \geq 4$
- Thus if we choose
  - $c = 1/2$ , and  $n_0 = 4$  we can make the inequality true
- Thus  $f(n) = O(n^2)$
- Called the **Asymptotic Upper-bound!**
  - Asymptotic means for large  $n$
  - Tight-bound because we have found the function that describes the algorithm's running time to with a constant multiple above

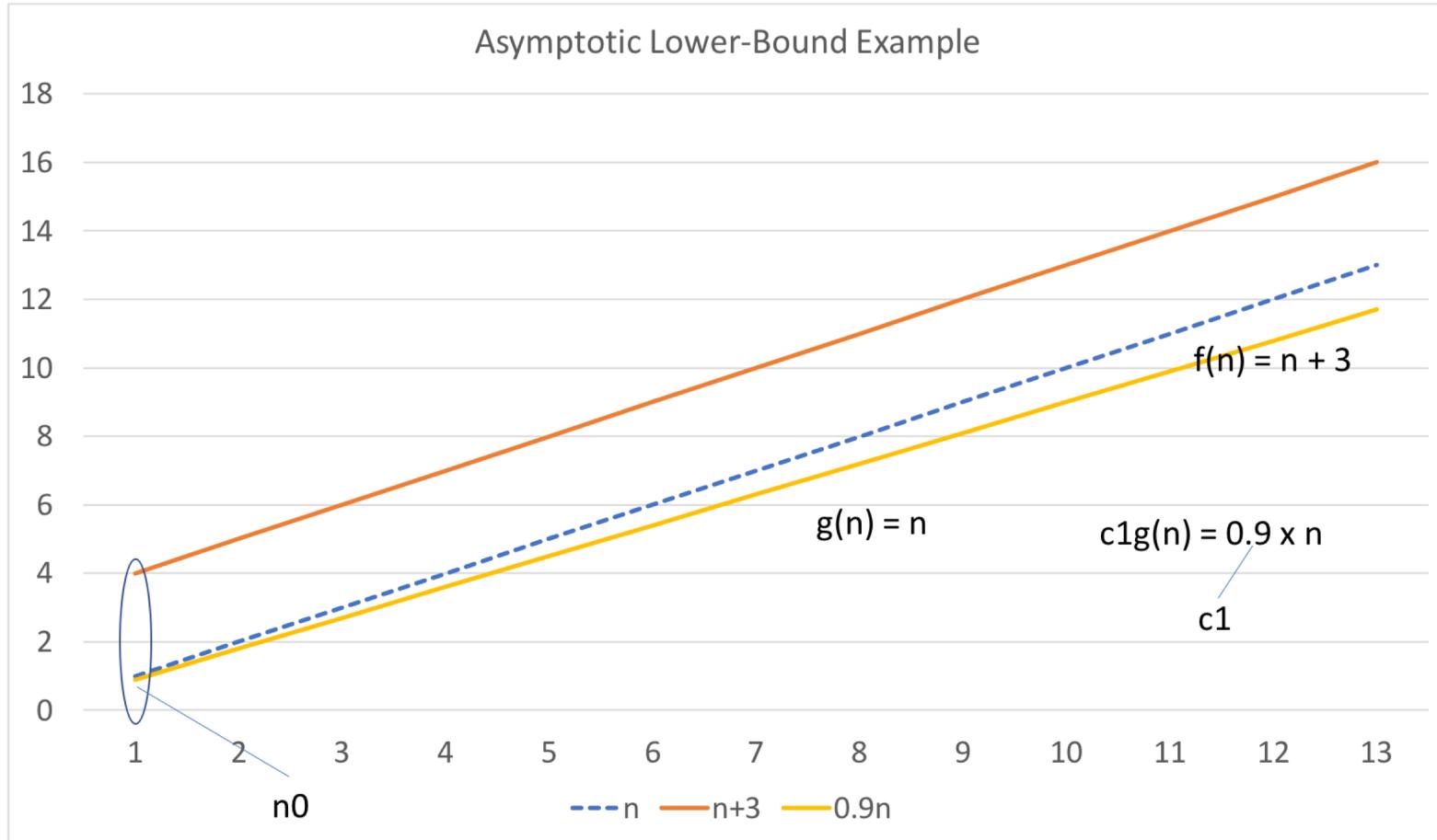
Remember:  
 $c$  and  $n$  are positive constants

# $\Omega$ -notation

- **Definition:** For a given function  $g(n)$ ,  $\Omega(g(n))$  is a **set of functions** such that
  - $\Theta(g(n)) = \{ f(n) : \text{there exists positive constants } c, \text{ and } n_0 \text{ s.t. } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0 \}$
- This is called an **asymptotic lower-bound** for  $f(n)$
- For all values of  $n \geq n_0$  the value of  $g(n)$  is always  $\leq f(n)$  within a constant factor
- Again, focus on **large values of n**



# $\Omega$ -notation: Graphical Example



# Theta, O and Omega

If  $f(n) = \Theta(g(n))$   
*Then*

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

# Note: Correct but Meaningless

You could say

$$3n^2 + 2 = O(n^6) \text{ or } 3n^2 + 2 = O(n^7)$$

$O(n^2)$

is a tighter asymptotic upper bound

But this is like answering:

- What is the world's record for running one mile?
  - Less than 3 days.
- How long does it take to drive from here to Chicago?
  - Less than 11 years.

# Do not get confused: O-Notation

## O(1) or “Order One”

- DOES NOT mean that it takes only one operation
- DOES mean that the work doesn't change as N changes
- Is notation for “constant work”

## O(N) or “Order N”

- DOES NOT mean that it takes N operations
- DOES mean that the work changes in a way that is proportional to N
- Is a notation for “work grows at a linear rate”

# Recap: Best Case Analysis of Insertion Sort

$$T(n)$$

$$= c1 * n + c2(n - 1) + c4(n - 1) + c5 \sum_2^n t_j + c6 \sum_2^n (t_j - 1) + c7 \sum_2^n (t_j - 1) + c8(n-1)$$

- When will the algorithm take the least amount of time?
  - When the array is already sorted
  - So the  $T(n)$  will be

$$T(n) = c1 * n + c2(n - 1) + c4(n - 1) + c5(n - 1) + c8(n - 1)$$

- Or  $T(n)$  is of the form **An +B**
- **Linear function** of input size, which is **n**

# Recap: Worst Case Analysis of Insertion Sort

$$T(n)$$

$$= c1 * n + c2(n - 1) + c4(n - 1) + c5 \sum_2^n t_j + c6 \sum_2^n (t_j - 1) + c7 \sum_2^n (t_j - 1) + c8(n-1)$$

- When will the algorithm take the most amount of time?

- When the array sorted in inverse

- So the  $T(n)$  will be

$$T(n)$$

$$= c1 * n + c2(n - 1) + c4(n - 1) + c5 \sum_2^n j + c6 \sum_2^n (j - 1) + c7 \sum_2^n (j - 1) + c8(n-1)$$

$$\begin{array}{c} [n(n+1)/2] - 1 \\ \diagup \\ \sum_2^n j \end{array}$$

$$n(n-1)/2$$

- Or  $T(n)$  is of the form  $An^2 + Bn + C$
  - Quadratic function of input size, which is  $n$

# Running Time for insertion sort

- BEST CASE

$$T(n) = c1 * n + c2(n - 1) + c4(n - 1) + c8(n-1)$$

- $T(n) = An + B = \mathbf{O(n)}$
- Where  $A = c1+c2+c4+c8$  and  $B = -(c2+c4+c8)$

- WORST CASE

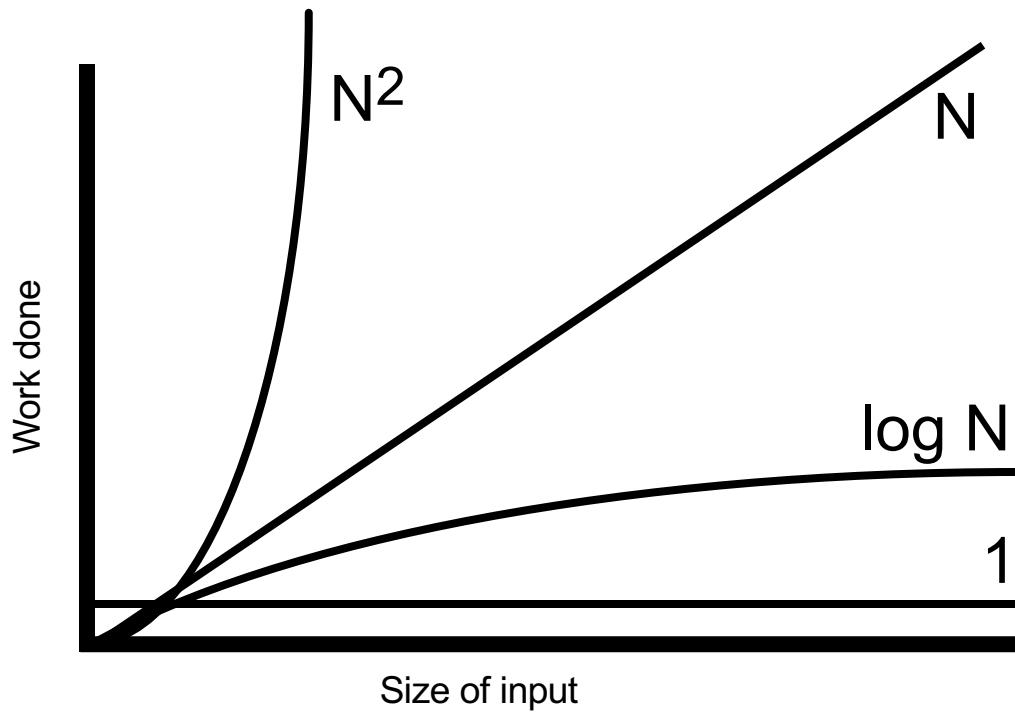
$$T(n) = c1 * n + c2(n - 1) + c4(n - 1) + c5 \frac{n(n+1)}{2} + c6 \frac{n(n-1)}{2} + c7 \frac{n(n-1)}{2} + c8 (n-1)$$

- $T(n) = An^2 + Bn + C = \mathbf{O(n^2)}$
- What are A, B, and C ???

# Comparing Algorithms

- We will use O-notation from now on to describe algorithm running time.
- We can compare different algorithms that solve the same problem:
  1. Determine the O(.) for the time complexity of each algorithm
  2. Compare them and see which has “better” performance

# Comparing Asymptotic Growth

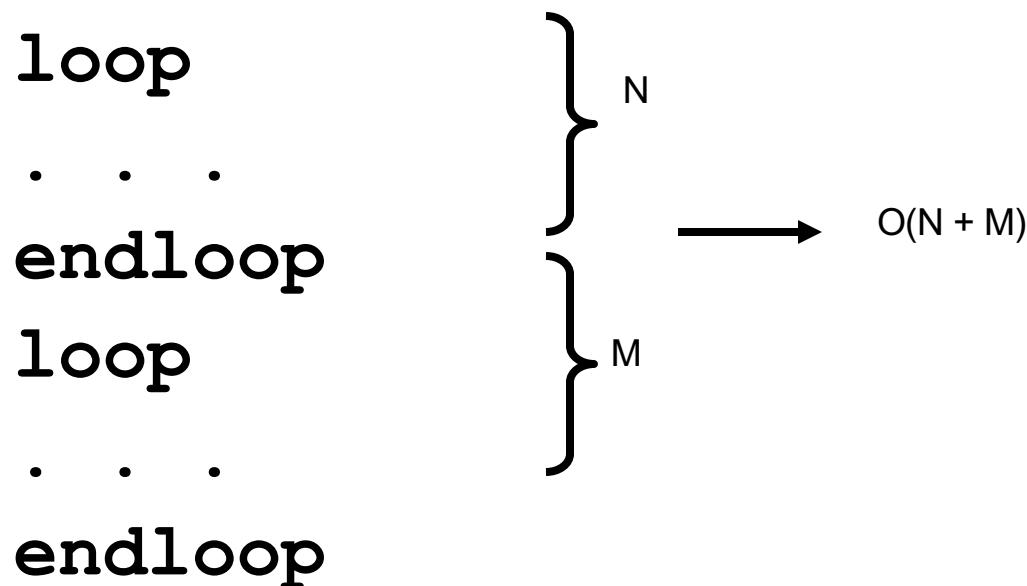


# Modular Analysis

- Algorithms typically consist of a sequence of logical steps/sections/modules
- We need a way to analyze these more complex algorithms...
- It's easy – analyze the sections and then combine them

# Sequential Steps

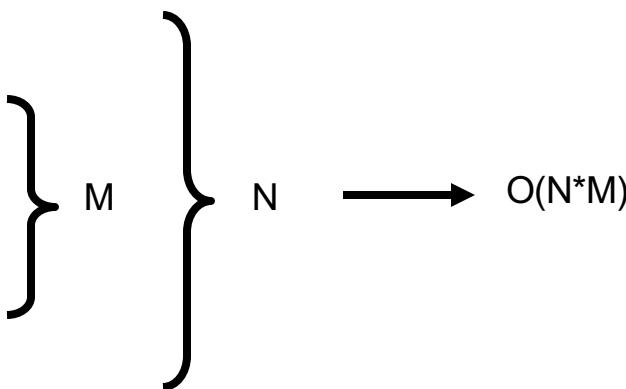
- If steps appear sequentially (one after another), then add their respective  $O()$ .



# Embedded Steps

- If steps appear embedded (one inside another), then **multiply** their respective O().

```
loop
    loop
        .
        .
        .
    endloop
endloop
```



The diagram illustrates the time complexity of nested loops. It shows a sequence of code: 'loop', 'loop', '...', 'endloop', 'endloop'. Two curly braces are placed over the code. The inner brace groups the two 'loop' statements and is labeled 'M'. The outer brace groups the entire sequence from 'loop' to 'endloop' and is labeled 'N'. An arrow points from the braces to the formula  $O(N*M)$ , indicating that the total time complexity is the product of the number of iterations of each loop.

# Correctly Determining Big-O

- **Can have multiple factors** (variables that measure input size)
  - $O(N * M)$
  - $O(\log P + N^2)$
- **But keep only the dominant factors:**
  - $O(N + N \log N) \rightarrow O(N \log N)$
  - $O(N * M + P) \rightarrow$  remains the same
  - $O(V^2 + V \log V) \rightarrow O(V^2)$
- **Drop constants:**
  - $O(2N + 3N^2) \rightarrow O(N + N^2) \rightarrow O(N^2)$

# Growth Functions

- Using O-notation, we are characterizing an algorithm's running time using a polynomial of some kind!
  - $O(n)$  --- worst case sumOfN function (in slide 3)
  - $O(n^2)$  --- worst case insertion sort
- One can have algorithms that have other running times as well such as
  - $O(2^n)$  --- worst case traveling salesman problem
  - $O(nlgn)$  – worst case merge sort
  - ....
- So, which of the running times are ok, and which are not?

# Puzzle

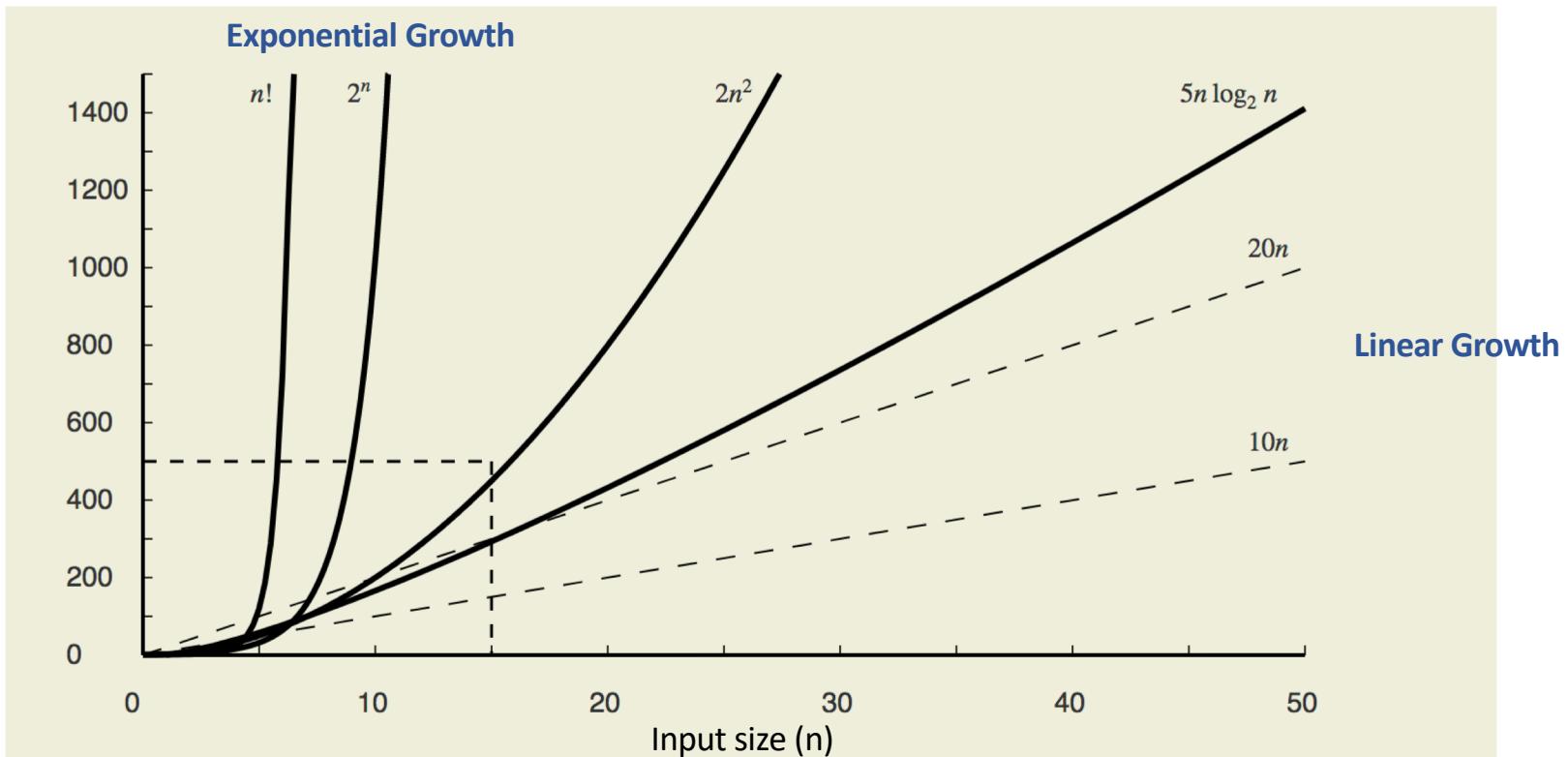
- Imagine a pond.
- Moss starts to grow on it and doubles in size every day.
- On the 10<sup>th</sup> day the moss fully covers the pond.

On what day, was the pond half-covered with moss?

**Exponential Growth and Linear Growth are very different**

We are used to thinking in terms of linear functions, exponential functions behave differently

# More Asymptotic Growth Rates



# Quiz

- You are given this set of functions:
- $n!$
- $2^n$
- $2n^2$
- $5n\log n$
- $20n$
- $10n$

Organize them by ascending order of growth rate

$$5n\log n > 10n > 20n > 2n^2 > 2^n > n!$$

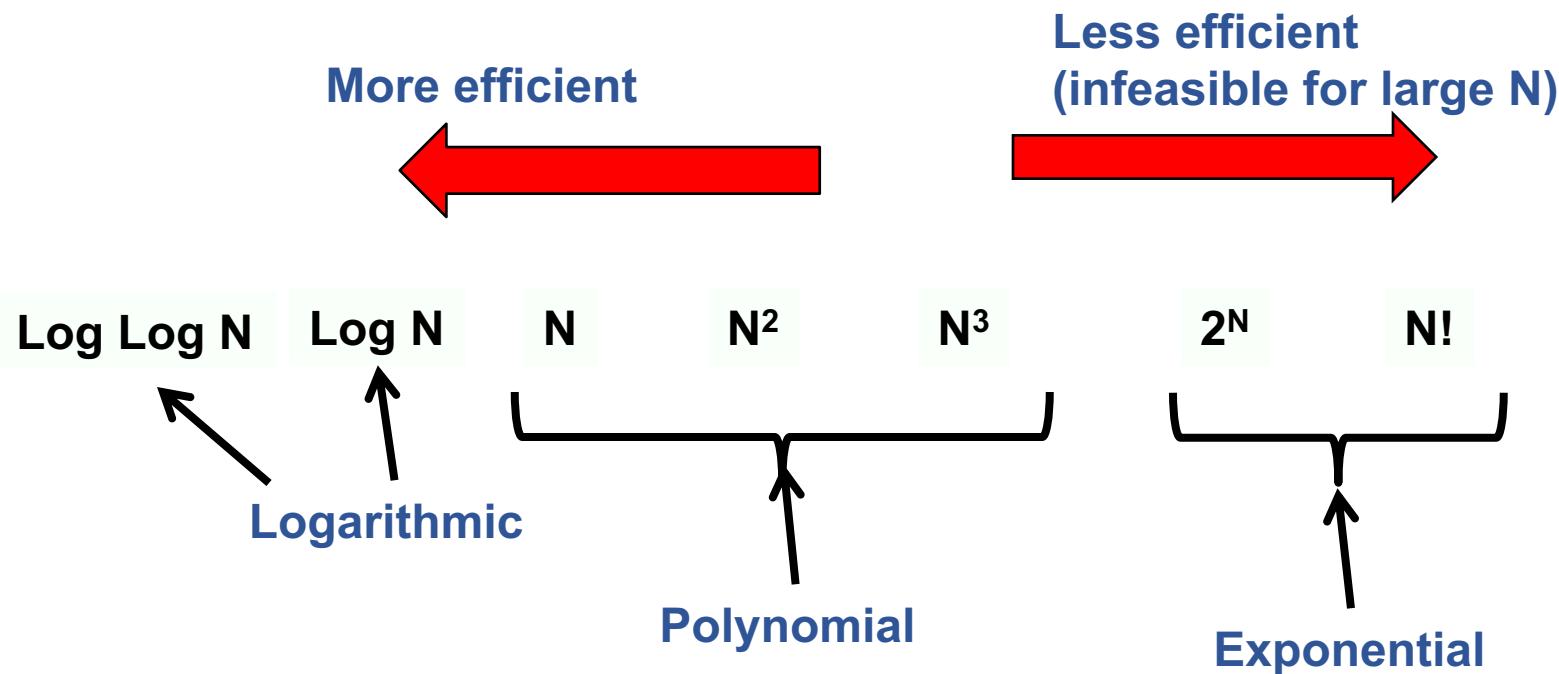
# Growth Rates Table

$n$	$\log \log n$	$\log n$	$n$	$n \log n$	$n^2$	$n^3$	$2^n$
16	2	4	$2^4$	$4 \cdot 2^4 = 2^6$	$2^8$	$2^{12}$	$2^{16}$
256	3	8	$2^8$	$8 \cdot 2^8 = 2^{11}$	$2^{16}$	$2^{24}$	$2^{256}$
1024	$\approx 3.3$	10	$2^{10}$	$10 \cdot 2^{10} \approx 2^{13}$	$2^{20}$	$2^{30}$	$2^{1024}$
64K	4	16	$2^{16}$	$16 \cdot 2^{16} = 2^{20}$	$2^{32}$	$2^{48}$	$2^{64K}$
1M	$\approx 4.3$	20	$2^{20}$	$20 \cdot 2^{20} \approx 2^{24}$	$2^{40}$	$2^{60}$	$2^{1M}$
1G	$\approx 4.9$	30	$2^{30}$	$30 \cdot 2^{30} \approx 2^{35}$	$2^{60}$	$2^{90}$	$2^{1G}$

# Comparing Computational Cost

Size of Input	$2^n$	$n^3$	$n^2$	$n$	$n \log n$	$\log n$
1	2	1	1	1	1	0
10	1024	1000	100	10	33.21928095	3.321928095
100	1.26765E+30	1000000	10000	100	664.385619	6.64385619
1000	1.0715E+301	10000000000	1000000	1000	9965.784285	9.965784285
10000	Are you crazy!	1E+12	100000000	10000	132877.1238	13.28771238
100000	Stop it!	1E+15	10000000000	100000	1660964.047	16.60964047
1000000	NO!	1E+18	1E+12	1000000	19931568.57	19.93156857
10000000	This is nuts!	1E+21	1E+14	10000000	232534966.6	23.25349666
100000000	I give up!	1E+24	1E+16	100000000	2657542476	26.57542476

# Order Of Growth



# Practice

- Consider this program. What is it doing?

```
sum = 0;  
for (i=1; i<=n; i++)  
    for (j=1; j<=n; j++)  
        sum++;
```

Input size = n

- What is the running time here?
  - The basic operation here is  $\text{sum}++ \rightarrow$  can be done in constant time, say  $c$
  - Ignore the operation  $\text{sum} = 0 \rightarrow$  it's so simple  $\rightarrow$  the time taken is  $\ll c$
- $T(n) = O(?)$ 
  - For a given input size  $n$ , how many steps will be taken?

# Practice: What is the O-notation for the following?

```
for i in range(n):
    for j in range(n):
        for k in range(n):
            k = 2+j+i
```

```
i = n
while i > 0:
    k = 2+2
    i = i//2
```

```
for i in range(n):
    k = k + i
```

```
for i in range(n):
    k = k+i
for j in range(n):
    k = k + j
for k in range(n):
    k = k+ k
```

```
i = 2
i = i*i+2*(5^6)/(i*9)
```

# Practice

- What is the Asymptotic Relationship ( $O$  or  $\Theta$  – *notation*) between
- $n^k$  *in terms of*  $c^n$  (assuming  $c > 1$  and  $k > 1$ )
- $\lg n^{\lg 17}$  *in terms of*  $\lg 17^{\lg n}$
- $\log_2 n$  *in terms of*  $\log_8 n$
- $3n\log_8 n$  *in terms of*  $n^3 \lg n$



*That's all Folks!*  
*Any Question?*