

$$F = G \frac{m_1 m_2}{d^2}$$

# Design and Analysis of Algorithms

Bilal Khalid Dar

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}$$

$$\frac{df}{dt} = \lim_{h \rightarrow 0} \frac{f(t+h) - f(t)}{h}$$

$$\phi(x) = \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

$$dS \geq 0$$

$$F - E + V = 2$$

$$E = mc^2$$

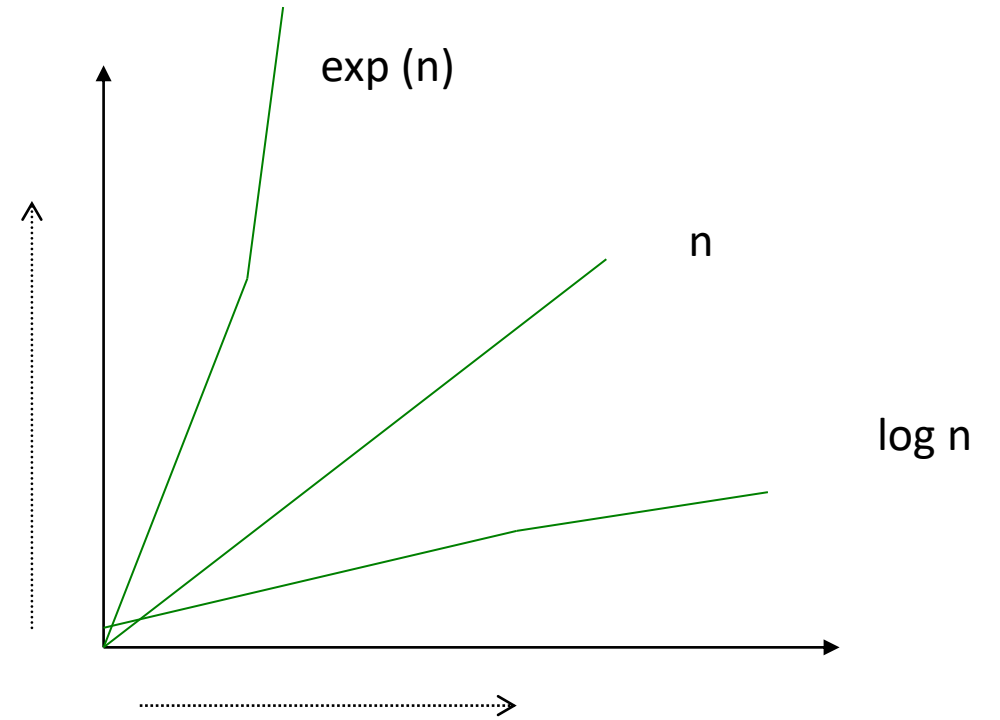




# Complexity Analysis

# Complexity

- In general, we are not so much interested in the time and space complexity for small inputs.
- For example, while the difference in time complexity between linear and binary search is meaningless for a sequence with  $n = 10$ , it is gigantic for  $n = 2^{30}$ .



# Complexity

- For example, let us assume two algorithms A and B that solve the same class of problems.
- The time complexity of A is  $5,000n$ , the one for B is  $\lceil 1.1^n \rceil$  for an input with  $n$  elements.
- For  $n = 10$ , A requires 50,000 steps, but B only 3, so B seems to be superior to A.
- For  $n = 1000$ , however, A requires 5,000,000 steps, while B requires  $2.5 \cdot 10^{41}$  steps.

# Complexity

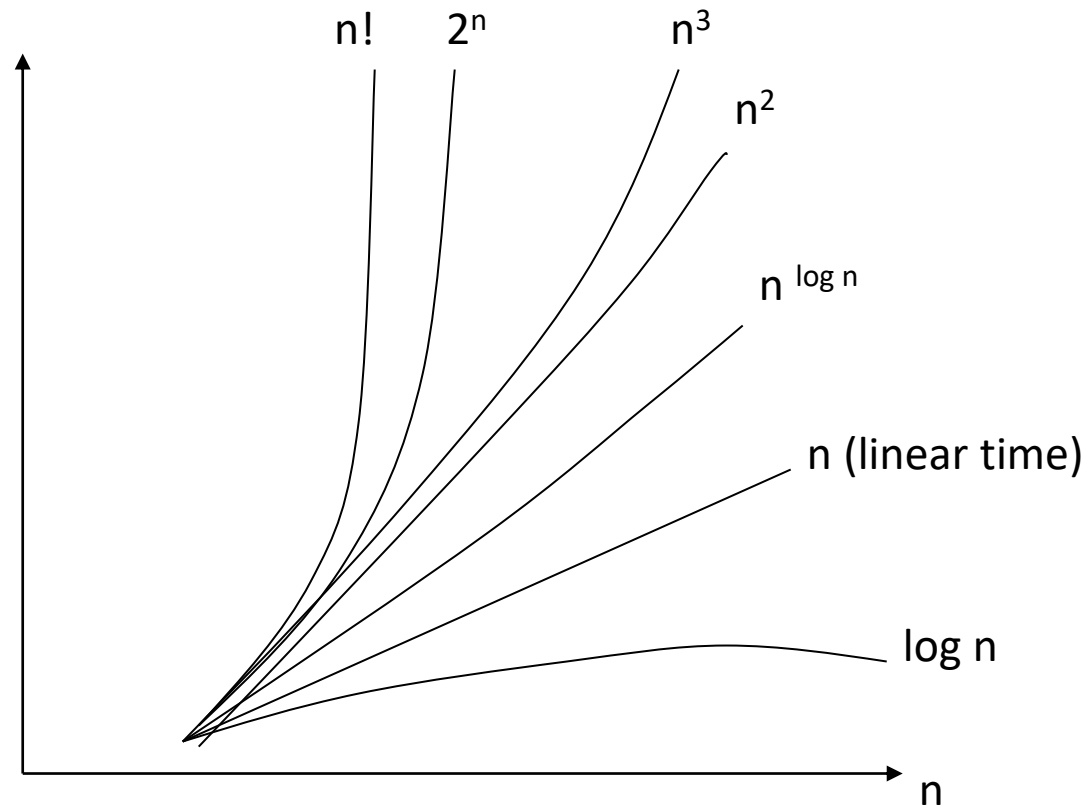
- Comparison: time complexity of algorithms A and B

Input Size	Algorithm A	Algorithm B
n	$5,000n$	$1.1^n$
10	50,000	3
100	500,000	13,781
1,000	5,000,000	$2.5 \times 10^{41}$
1,000,000	$5 \times 10^9$	$4.8 \times 10^{41392}$

# Complexity

- This means that algorithm B cannot be used for large inputs, while algorithm A is still feasible.
- So what is important is the growth of the complexity functions.
- The growth of time and space complexity with increasing input size  $n$  is a suitable measure for the comparison of algorithms.

# Complexity classes $f(n)$



**Figure 2.** Growth rates of some important complexity classes

# Clicker Question 1

- “A program finds all the prime numbers between 2 and 1,000,000,000 from scratch in 0.37 seconds.”
- Is this a fast solution?
  - A. no
  - B. yes
  - C. it depends



## Clicker Question 2

- What is output by the following code?

```
int total = 0;
for (int i = 0; i < 13; i++)
    for (int j = 0; j < 11; j++)
        total += 2;
System.out.println(total);
```

- A. 24
- B. 120
- C. 143
- D. 286
- E. 338

# Standard Analysis Techniques

- Constant time statements
- Analyzing Loops
- Analyzing Nested Loops
- Analyzing Sequence of Statements
- Analyzing Conditional Statements

# Analysis Example

- How many statements are executed by method total as a function of values.length
- Let  $N = \text{values.length}$
- $N$  is commonly used as a variable that denotes the amount of data

```
public int total(int[] values) {  
    int result = 0;  
    for (int i = 0; i < values.length; i++)  
        result += values[i];  
    return result;  
}
```

# Counting Up Statements

- `int result = 0;` 1
- `int i = 0;` 1
- `i < values.length;`  $N + 1$
- `i++`  $N$
- `result += values[i];`  $3N$
- `return total;` 1
- **$T(N) = 5N + 4$**
- $T(N)$  is the number of executable statements in method `total` as function of `values.length`

```
public int total(int[] values) {  
    int result = 0;  
    for (int i = 0; i < values.length; i++)  
        result += values[i];  
    return result;  
}
```

# Analyzing an Algorithm

// Input: int A[N], array of N integers

// Output: Sum of all numbers in array A

```
int Sum(int A[], int N)
{
    int s=0;
    for (int i=0; i< N; i++)
        s = s + A[i];
    return s;
}
```

How should we analyse this?

# Analyzing an Algorithm

```
// Input: int A[N], array of N integers
// Output: Sum of all numbers in array A
```

```
int Sum(int A[], int N){
```

```
    int s=0; ← ①
```

```
    for (int i=0; i< N; i++)
```

```
        ② → i=0; i< N; i++ ← ③ ← ④
```

```
        s = s + A[i];
```

```
        ⑤ → s ← ⑥ ← ⑦
```

```
    return s;
```

```
    } ← ⑧
```

1,2,8: Once

3,4,5,6,7: Once per each iteration  
of for loop, N iteration

Total:  $5N + 4$

The *complexity function* of the  
algorithm is :  $f(N) = 5N + 4$



# Another Simplification

- When determining complexity of an algorithm we want to simplify things
  - hide some details to make comparisons easier
- Like assigning your grade for course
  - At the end of of degree your transcript won't list all the details of your performance in the course
  - it won't list scores on all assignments, quizzes, and tests
  - simply a letter grade, B- or A or D+
- So we focus on the dominant term from the function and ignore the coefficient

# Constant time statements

- Simplest case:  $O(1)$  time statements
- Assignment statements of simple data types  
`int x = y;`
- Arithmetic operations:  
`x = 5 * y + 4 - z;`
- Array referencing:  
`A[j] = 5;`
- Most conditional tests:  
`if (x < 12) ...`

# Analyzing Loops

- Any loop has two parts:
- How many iterations are performed?
- How many steps per iteration?

```
int sum = 0,j;
```

```
for (j=0; j < N; j++)
```

```
sum = sum +j;
```

# Analyzing Loops

- Any loop has two parts:
  - How many iterations are performed?
  - How many steps per iteration?  

```
int sum = 0,j;  
for (j=0; j < N; j++)  
    sum = sum +j;
```
  - Loop executes N times (0..N-1)
  - $O(1)$  steps per iteration
- Total time is  $N * O(1) = O(N*1) = O(N)$

# Analyzing Nested Loops

- Treat just like a single loop and evaluate each level of nesting as needed:

```
int j,k;
```

```
for (j=0; j<n; j++)
```

```
    for (k=0; k<n; k++)
```

```
        sum += k+j;
```

# Analyzing Nested Loops

- Treat just like a single loop and evaluate each level of nesting as needed:

```
int j,k;
```

```
for (j=0; j<n; j++)
```

```
    for (k=0; k<n; k++)
```

```
        sum += k+j;
```

$n+1$  times

$(n) * n+1$  times

$n * n$  times

$F(n) = 2n^2 + 2n + 1$

$F(n) = O(n^2)$





## Clicker Question 3

- What is output when method `sample` is called?

// pre:  $n \geq 0, m \geq 0$

```
public static void sample(int n,  
int m) {
```

```
    int total = 0;
```

```
    for (int i = 0; i < n; i++)
```

```
        for (int j = 0; j < m;
```

```
            j++)
```

```
                total += 5;
```

```
    System.out.println(total);
```

```
}
```

A. 5

B.  $n * m$

C.  $n * m * 5$

D.  $n^m$

E.  $(n * m)^5$

# Analyzing an Algorithm

- Simple statement sequence

$S_1; S_2; \dots; S_k$

- Basic Step = 1 as long as  $k$  is constant

- Simple loops

`for (i=0; i<n; i++) { s; }`

where  $s$  is Basic Step = 1

- Basic Steps :  $n$

- Nested loops

`for (i=0; i<n; i++)`

`for (j=0; j<n; j++) { s; }`

- Basic Steps :  $n^2$

# Analyzing Nested Loops

- Treat just like a single loop and evaluate each level of nesting as needed:

```
int j,k;  
for (j=0; j<N; j++)  
    for (k=N; k>0; k--)  
        sum += k+j;
```

- Start with outer loop:
  - How many iterations?  $N$
  - How much time per iteration? Need to evaluate inner loop
- Inner loop uses  $O(N)$  time
- Total time is  $N * O(N) = O(N*N) = O(N^2)$

# Class Activity

```
void add( int A[ ], int B[ ], int n)
{
    for (i=0; i<n; i++)
    {
        for (j=0; j< n; j++)
        {
            c[i,j] =A[i][k] + B[k][j];

        }
    }
}
```

# Class Activity

```
void multiply( int A[ ], int B[ ], int n)
{
    for (i=0; i<n; i++)
    {
        for (j=0; j< n; j++)
        {
            c[i,j]= 0;
            for (k=0; k< n; ++k)
            {
                c[i,j]+=A[i][k]*b[k][j];
            }
        }
    }
}
```

# Analyzing Sequence of Statements

```
for (j=0; j < N; j++)  
    for (k =0; k < j; k++)  
        sum = sum + j*k;  
for (l=0; l < N; l++)  
    sum = sum -l;  
cout<<"Sum="<<sum;
```



## Analyzing Sequence of Statements

- For a sequence of statements, compute their complexity functions individually and add them up

for (j=0; j < N; j++)	}	$O(N^2)$
for (k =0; k < j; k++)		
sum = sum + j*k;		
for (l=0; l < N; l++)	}	$O(N)$
sum = sum -l;		
cout<<"Sum="<<sum;	}	$O(1)$

Total cost is  $O(N^2) + O(N) + O(1) = O(N^2)$

**SUM RULE**

# Analysing an Algorithm

- Loop index doesn't vary linearly

```
i = 1;
while ( i < n ) {
    S;
    i = 2 * i;
}
```

- $i$  takes values 1, 2, 4, ... until it exceeds  $n$

In Next Lecture

# Write Pseudocode and calculate complexity

---

- A code for finding the sum of the digits in the number.
- Example: 3554
- $3+5+5+4 = 17$

# Analysing an Algorithm

- Loop index doesn't vary linearly

```
i = 1;
while ( i < n )
{
    S;
    i = 2 * i;
}
```

- $i$  takes values 1, 2, 4, ... until it exceeds  $n$

$i$

- 1
- $1 \times 2 = 2$
- $2 \times 2 = 2^2$
- $2^2 \times 2 = 2^3$
- .
- .
- .
- $2^k$

# Analysing an Algorithm

$i$

- 1
- $1 \times 2 = 2$
- $2 \times 2 = 2^2$
- $2^2 \times 2 = 2^3$
- .
- .
- .
- $2^k$

Assume

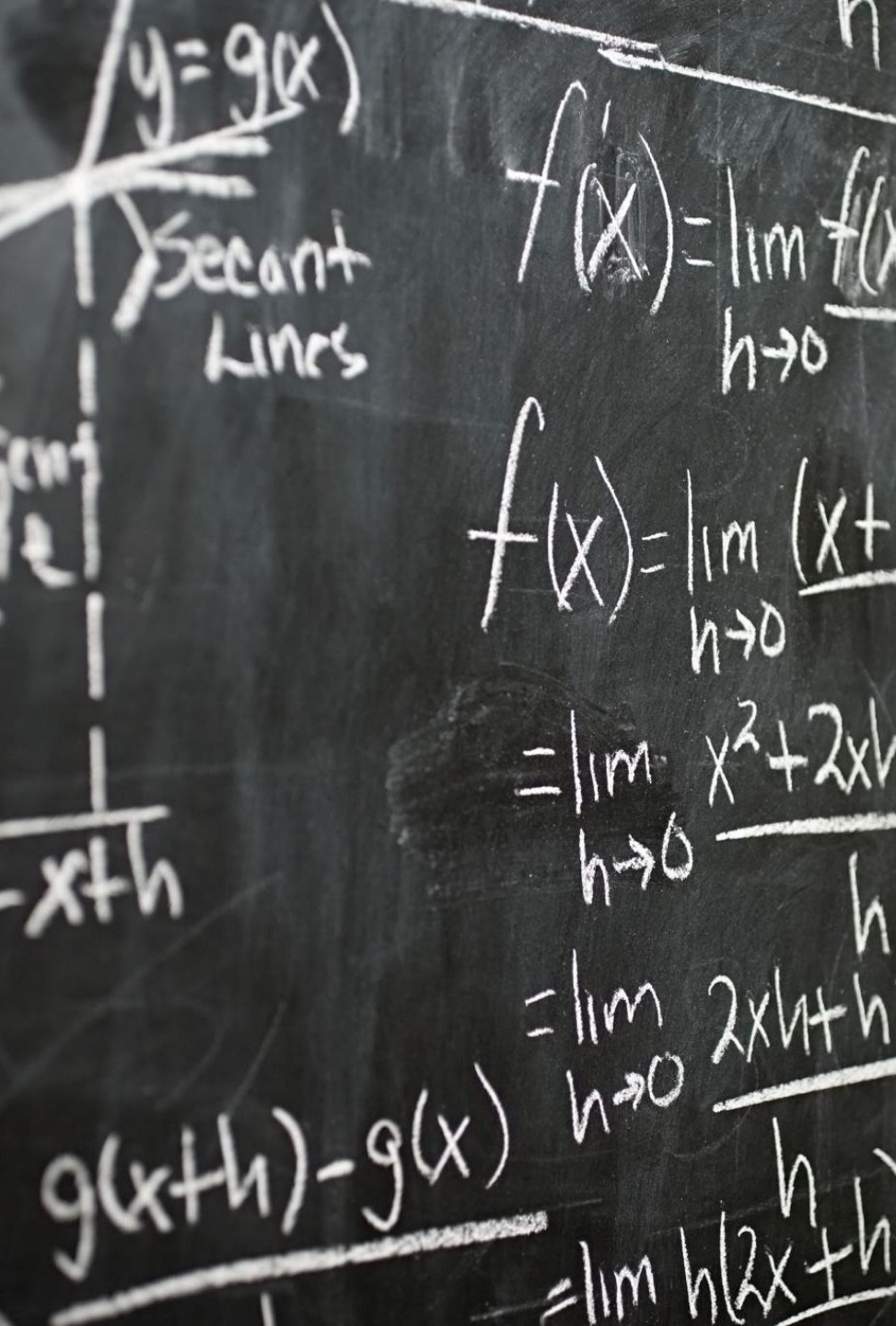
$i \geq n$

$$i = 2^k$$

$$2^k = n$$

$$K = \log_2(n)$$





# Analyzing Conditional Statements

---

What about conditional statements such as

```
if (condition)
    statement1;
else
    statement2;
```

where statement1 runs in  $O(N)$  time and statement2 runs in  $O(N^2)$  time?

We use "worst case" complexity: among all inputs of size  $N$ , that is the maximum running time?

The analysis for the example above is  $O(N^2)$


# Asymptotic Complexity

---

# Asymptotic Complexity

- 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  $\Theta(n^2)$ .
- Describes behavior of function in the limit.
- Written using ***Asymptotic Notation***.

# $10^6$ instructions/sec, runtimes



<u><math>N</math></u>	<u><math>O(\log N)</math></u>	<u><math>O(N)</math></u>	<u><math>O(N \log N)</math></u>	<u><math>O(N^2)</math></u>
10	0.000003	0.00001	0.000033	0.0001
100	0.000007	0.00010	0.000664	0.1000
1,000	0.000010	0.00100	0.010000	1.0
10,000	0.000013	0.01000	0.132900	1.7 min
100,000	0.000017	0.10000	1.661000	2.78 hr
1,000,000	0.000020	1.0	19.9	11.6 day
1,000,000,000	0.000030	16.7 min	18.3 hr	318 centuries

# Asymptotic Complexity

- The  $5N+3$  time bound is said to "grow asymptotically" like  $N$
- This gives us an approximation of the complexity of the algorithm
- Ignores lots of (machine dependent) details, concentrate on the bigger picture

# Asymptotic Analysis

- Asymptotic analysis is an analysis of algorithms that focuses on
  - Analyzing problems of large input size
  - Consider only the leading term of the formula
  - Ignore the coefficient of the leading term

# Comparing Functions: Asymptotic Notation

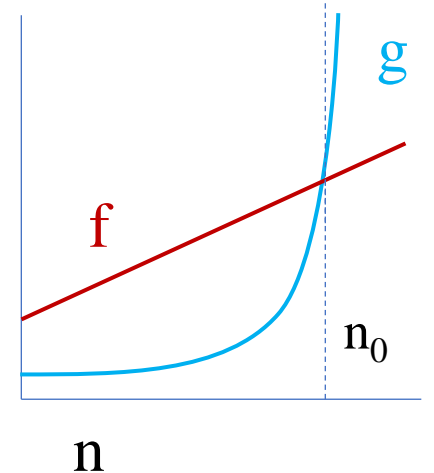
- Big Oh Notation: Upper bound
- Omega Notation: Lower bound
- Theta Notation: Tighter bound

# Big-Oh Notation

Given two functions  $f(n)$  &  $g(n)$  for input  $n$ , we say  $f(n)$  is in  $O(g(n))$  iff there exist positive constants  $c$  and  $n_0$  such that

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

Basically, we want to find a function  $g(n)$  that is eventually always bigger than  $f(n)$





# Big Oh Notation

If  $f(N)$  and  $g(N)$  are two complexity functions, we say

$$f(N) = O(g(N))$$

*(read " $f(N)$  as order  $g(N)$ ", or " $f(N)$  is big-O of  $g(N)$ ")*

if there are constants  $c$  and  $N_0$  such that for  $N > N_0$ ,

$$f(N) \leq c * g(N)$$

for all sufficiently large  $N$ .

\*big-O-Upper bound

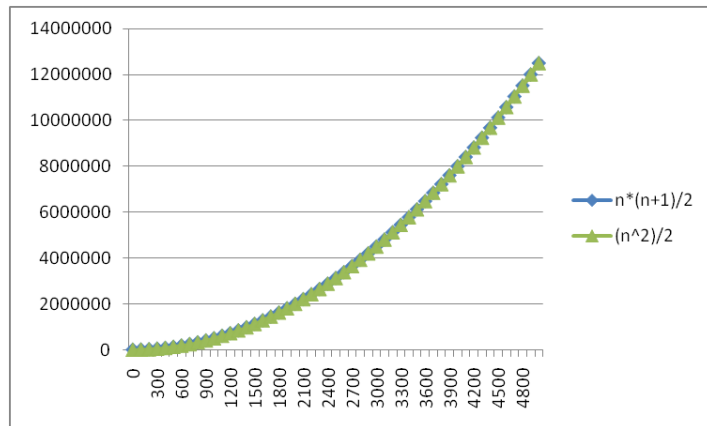


# Big-Oh Rules

- 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  $O(3n)$ ”

# No Need To Be So Exact

---



Constants do not matter

- Consider  $6N^2$  and  $20N^2$
- When  $N \gg 20$ , the  $N^2$  is what is driving the function's increase

Lower-order terms are also less important

- $N(N+1)/2$  vs. just  $N^2/2$
- The linear term is inconsequential

We need a better notation for performance that focuses on the dominant terms only

# What Dominates in Previous Example?

What about the +3 and 5 in  $5N+3$ ?

- As  $N$  gets large, the +3 becomes insignificant
- 5 is inaccurate, as different operations require varying amounts of time and also does not have any significant importance

What is fundamental is that the time is *linear* in  $N$ .

Asymptotic Complexity: As  $N$  gets large, concentrate on the highest order term:

- ✓ Drop lower order terms such as +3
- ✓ Drop the constant coefficient of the highest order term i.e.  $N$

Asymptotic Complexity: As  $N$  gets large, concentrate on the highest order term:

# Showing Order Briefly ...

- Show  $10N^2 + 15N$  is  $O(N^2)$



# Showing Order Briefly ...

- Show  $10N^2 + 15N$  is  $O(N^2)$
- Break into terms.
- $10N^2 \leq 10N^2$
- $15N \leq 15N^2$  for  $N \geq 1$  (Now add)
- $10N^2 + 15N \leq 10N^2 + 15N^2$  for  $N \geq 1$
- $10N^2 + 15N \leq 25N^2$  for  $N \geq 1$
- $c = 25, N_0 = 1$
- Note, the choices for  $c$  and  $N_0$  are not unique

# The Gist of Big-Oh

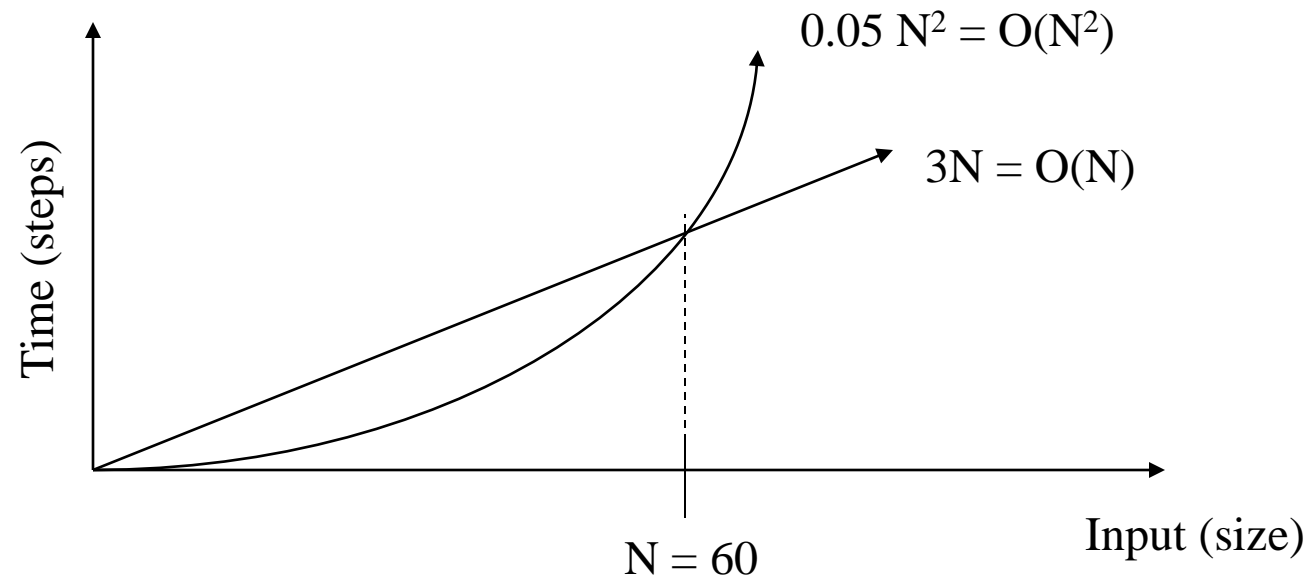
Take functions  $f(n)$  &  $g(n)$ , consider only the most significant term and remove constant multipliers:

- $5n+3 \rightarrow n$
- $7n+.5n^2+2000 \rightarrow n^2$
- $300n+12+n\log n \rightarrow n \log n$

Then compare the functions; if  $f(n) \leq g(n)$ , then  $f(n)$  is in  $O(g(n))$

# Comparing Functions

- As inputs get larger, any algorithm of a smaller order will be more efficient than an algorithm of a larger order





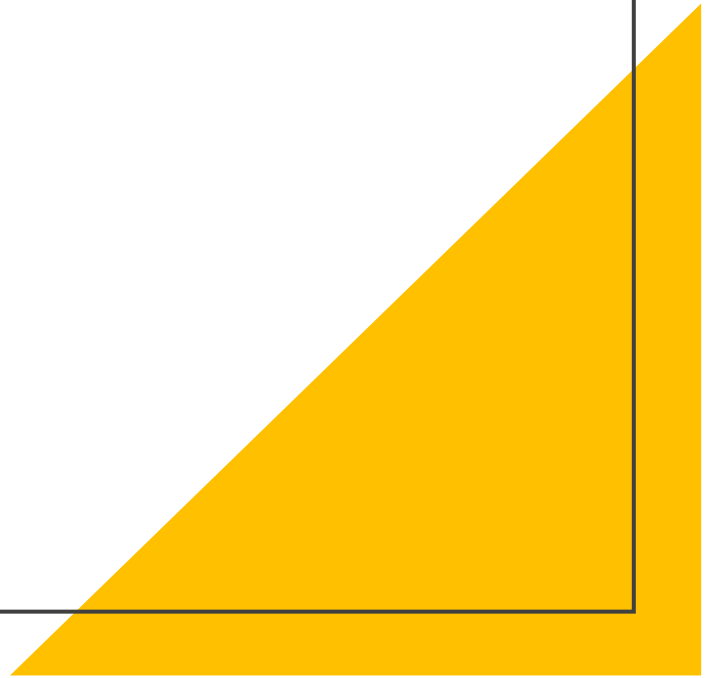
$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that}$   
 $0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\} .$

# Big Omega Notation

# Example

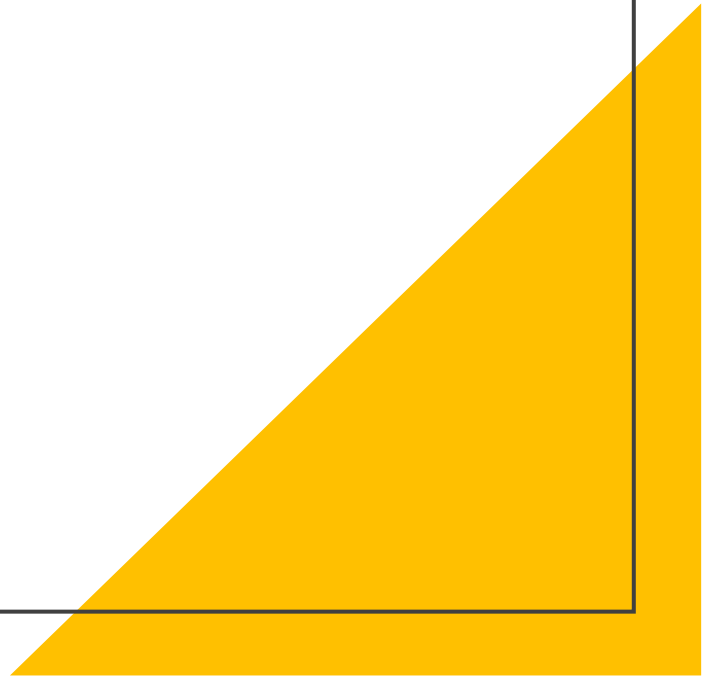
- Show  $2n + 3$  is  $\Omega(n)$

- 



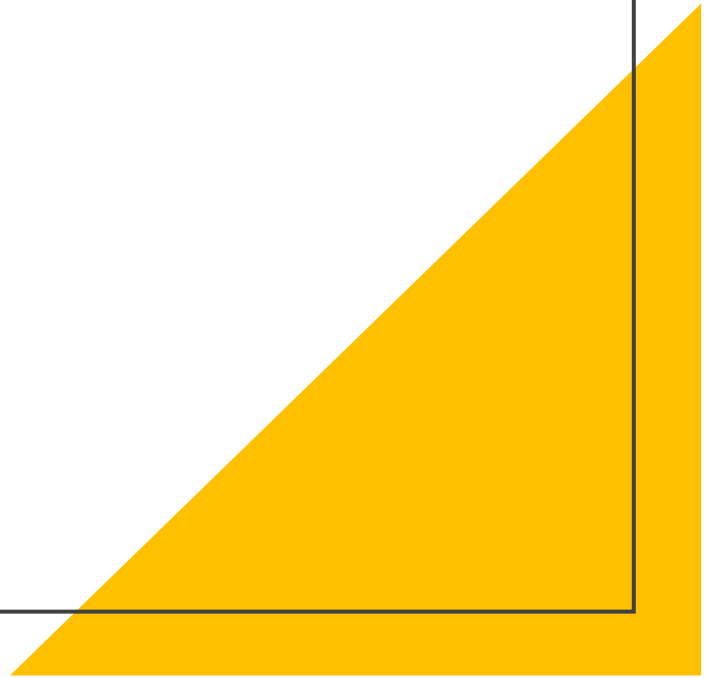
# Example

- Show  $2n^2 + n + 1$  is  $\Omega(n^2)$



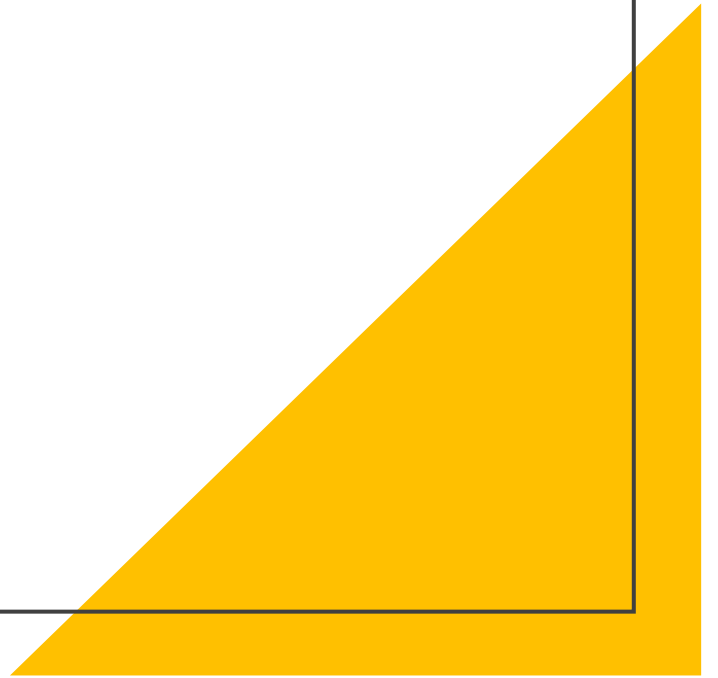
# Example

- Show  $2n^2 + n + 1$  is  $\Omega(n^2)$
- $2n^2 + n + 1 \geq cn^2$  ;for all  $n \geq 1$
- $c=1, n_0=1$



# Task

- Show  $8n^3 + 5n^2 + 7$  is  $\Omega(n^3)$

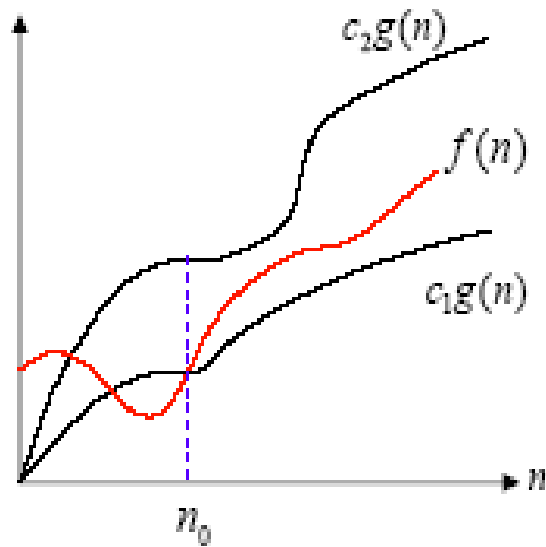


$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that}$   
 $0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \text{ for all } n \geq n_0\} .^1$

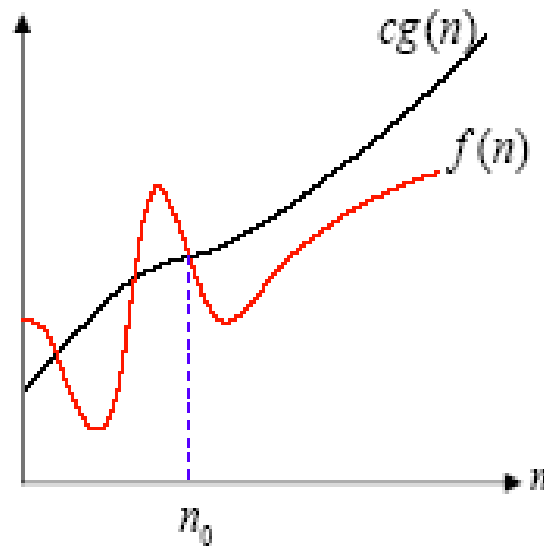
# Big Theta Notation

A large yellow triangle is positioned in the bottom right corner of the slide, pointing towards the top right.

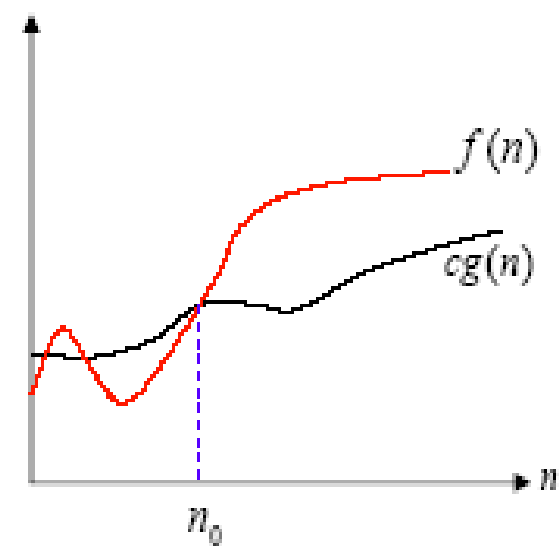
# Asymptotic notation



$$f(n) = \Theta(g(n))$$



$$f(n) = O(g(n))$$



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

- Example:
  - $f(n) = 3n^5 + n^4 = \Theta(n^5)$

# Polynomial and Intractable Algorithms

- Polynomial Time complexity

- An algorithm is said to be polynomial if it is  $O(n^d)$  for some integer  $d$
- Polynomial algorithms are said to be efficient
  - They solve problems in reasonable times!

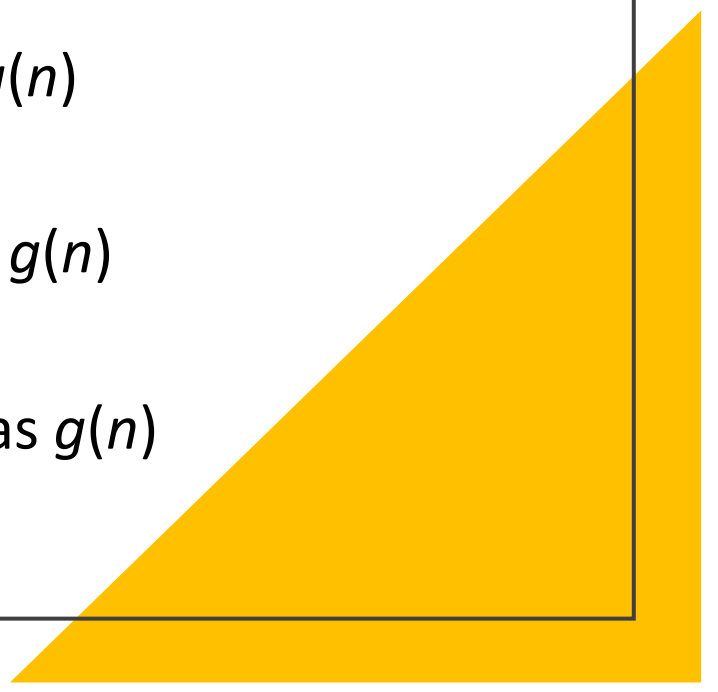
- Intractable algorithms

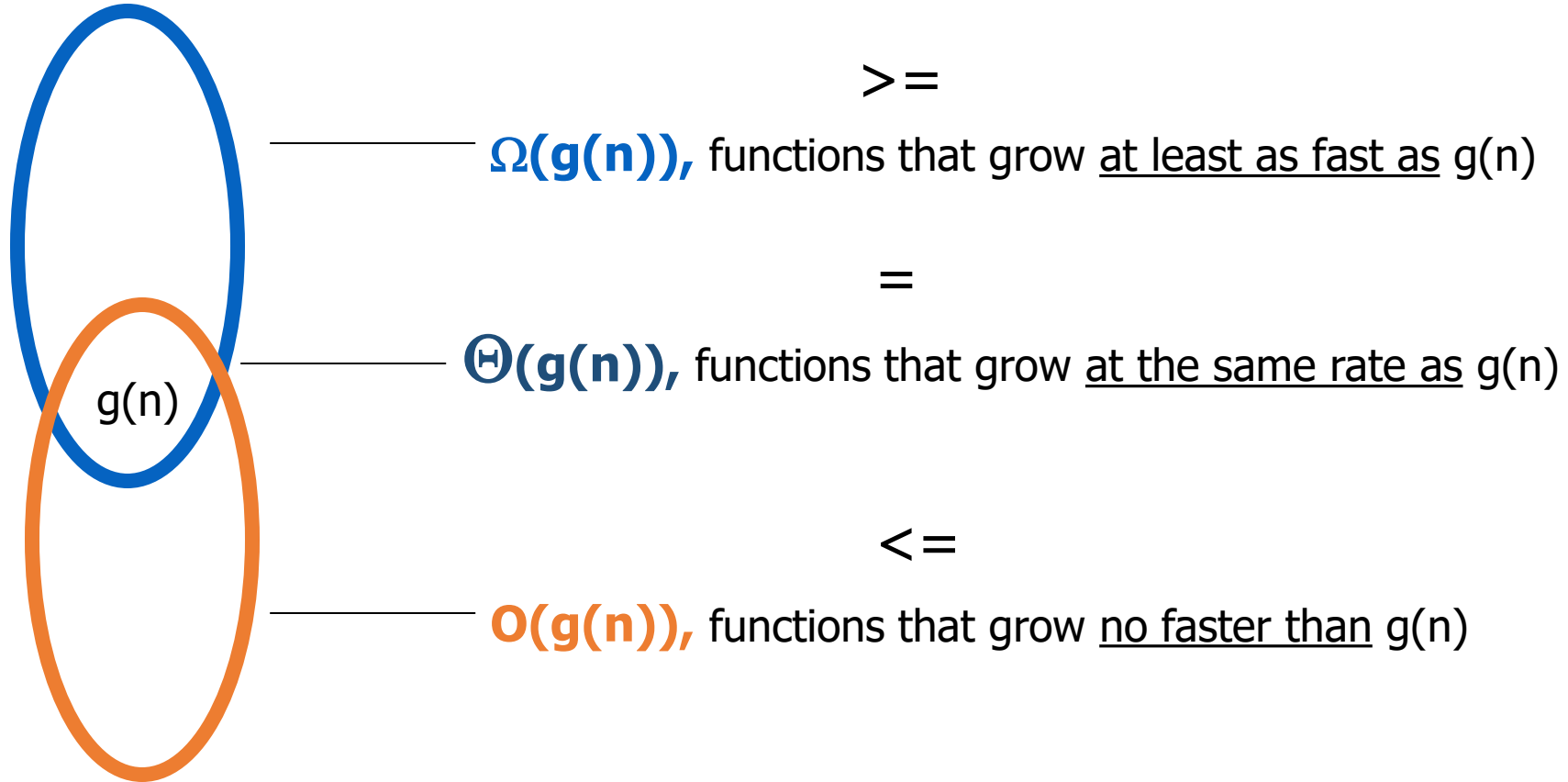
- Algorithms for which there is no *known* polynomial time algorithm
- *We will come back to this important class later*



# Asymptotic order of growth

A way of comparing functions that ignores constant factors and small input sizes

- $O(g(n))$ : class of functions  $f(n)$  that grow no faster than  $g(n)$
  - $\Theta(g(n))$ : class of functions  $f(n)$  that grow at same rate as  $g(n)$
  - $\Omega(g(n))$ : class of functions  $f(n)$  that grow at least as fast as  $g(n)$
- 
- A large yellow right-angled triangle is positioned in the bottom right corner of the slide, pointing towards the top right.



# Performance Classification

$f(n)$	Classification
1	<b>Constant:</b> run time is fixed, and does not depend upon $n$ . Most instructions are executed once, or only a few times, regardless of the amount of information being processed
$\log n$	<b>Logarithmic:</b> when $n$ increases, so does run time, but much slower. When $n$ doubles, $\log n$ increases by a constant, but does not double until $n$ increases to $n^2$ . Common in programs which solve large problems by transforming them into smaller problems.
$n$	<b>Linear:</b> run time varies directly with $n$ . Typically, a small amount of processing is done on each element.
$n \log n$	When $n$ doubles, run time slightly more than doubles. Common in programs which break a problem down into smaller sub-problems, solves them independently, then combines solutions
$n^2$	<b>Quadratic:</b> when $n$ doubles, runtime increases fourfold. Practical only for small problems; typically the program processes all pairs of input (e.g. in a double nested loop).
$n^3$	<b>Cubic:</b> when $n$ doubles, runtime increases eightfold
$2^n$	<b>Exponential:</b> when $n$ doubles, run time squares. This is often the result of a natural, “brute force” solution.

**TABLE 1** Commonly Used Terminology for the Complexity of Algorithms.

<i>Complexity</i>	<i>Terminology</i>
$\Theta(1)$	Constant complexity
$\Theta(\log n)$	Logarithmic complexity
$\Theta(n)$	Linear complexity
$\Theta(n \log n)$	Linearithmic complexity
$\Theta(n^b)$	Polynomial complexity
$\Theta(b^n)$ , where $b > 1$	Exponential complexity
$\Theta(n!)$	Factorial complexity


# Size does matter

What happens if we double the input size  $N$ ?

$N$	$\log_2 N$	$5N$	$N \log_2 N$	$N^2$	$2^N$
8	3	40	24	64	256
16	4	80	64	256	65536
32	5	160	160	1024	$\sim 10^9$
64	6	320	384	4096	$\sim 10^{19}$
128	7	640	896	16384	$\sim 10^{38}$
256	8	1280	2048	65536	$\sim 10^{76}$

# Typical Big O Functions – "Grades"

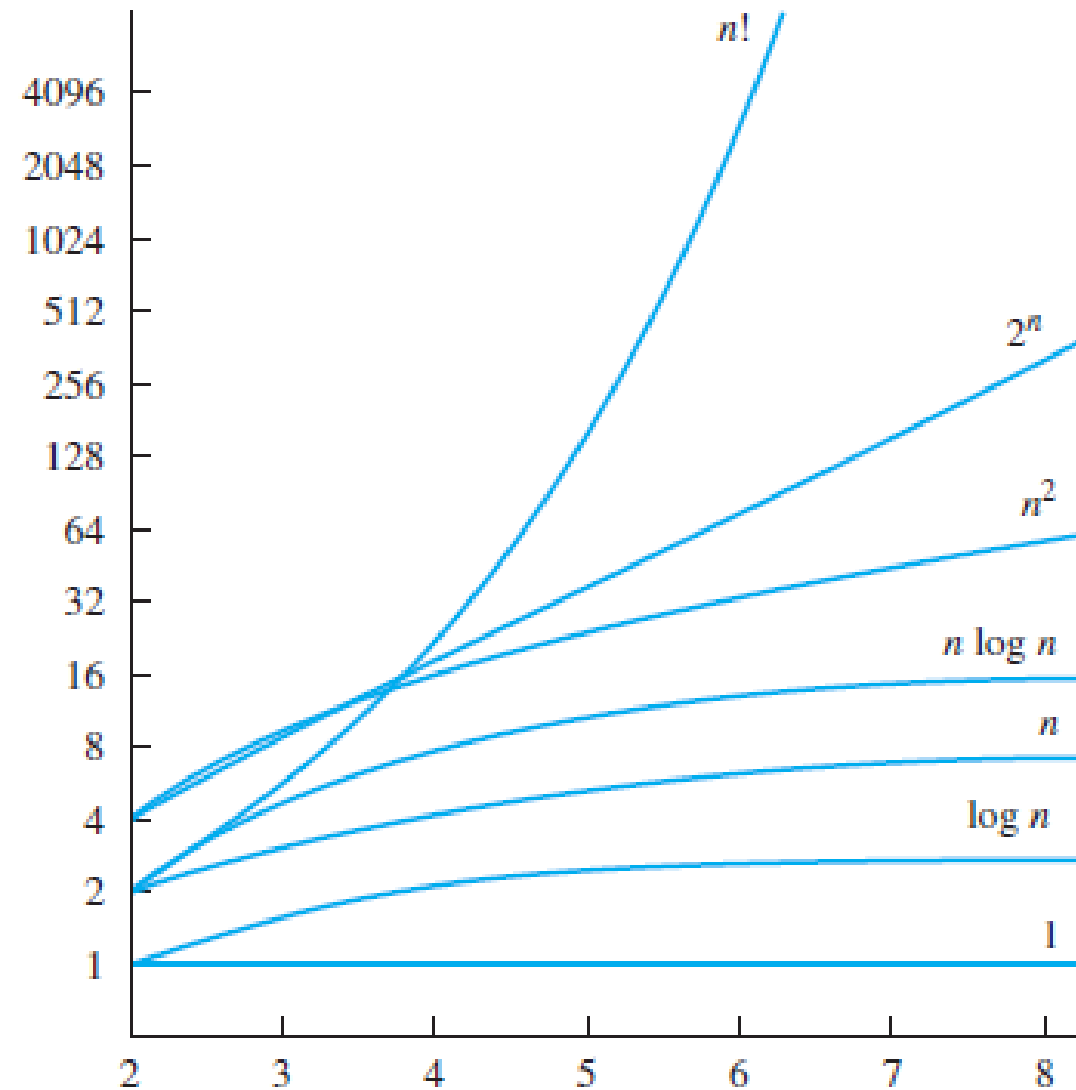
Function	Common Name
$N!$	factorial
$2^N$	Exponential
$N^d, d > 3$	Polynomial
$N^3$	Cubic
$N^2$	Quadratic
$N\sqrt{N}$	N Square root N
$N \log N$	$N \log N$
$N$	Linear
$\sqrt{N}$	Root - n
$\log N$	Logarithmic
1	Constant



Running time grows 'quickly' with more input.

Running time grows 'slowly' with more input.

A Display of the  
Growth of  
Functions  
Commonly Used  
in Big-O  
Estimates



# Review of Three Common Sets

$\mathbf{f(n) = O(g(n))}$  means  $c \times g(n)$  is an *Upper Bound* on  $f(n)$

$\mathbf{f(n) = \Omega(g(n))}$  means  $c \times g(n)$  is a *Lower Bound* on  $f(n)$

$\mathbf{f(n) = \Theta(g(n))}$  means  $c_1 \times g(n)$  is an *Upper Bound* on  $f(n)$   
*and*  $c_2 \times g(n)$  is a *Lower Bound* on  $f(n)$

These bounds hold for all inputs beyond some threshold  $n_0$ .



# Task

- Show  $2N^3 + 10N$  is  $O(N^3)$
- Show  $2N^3 + 10N$  is not  $O(N^2)$
- Show  $2N^3 + 10N$  is  $O(N^4)$

Thank you

