

# **CSC 411**

## **Design and Analysis of Algorithms**

---

### **Chapter 2 Fundamentals of the Analysis of Algorithm Efficiency**

#### **- Part 1**

Instructor: Minhee Jun

# Overview: Analysis of Algorithm Efficiency

---

- Issues:
  - correctness, time efficiency & space efficiency, simplicity, generality, optimality
- Efficiency Analysis
  - Efficiency can be studied in precise quantitative terms
  - Efficiency is primarily important in a practical point of view
- Topics to be covered
  - $O$  (big oh),  $\Omega$  (big omega),  $\Theta$  (big theta)
  - Analyze efficiency of non-recursive algorithms
  - Analysis efficiency of recursive algorithms
- Approaches
  - Empirical analysis
  - Theoretical analysis

# Empirical Analysis of Time Efficiency

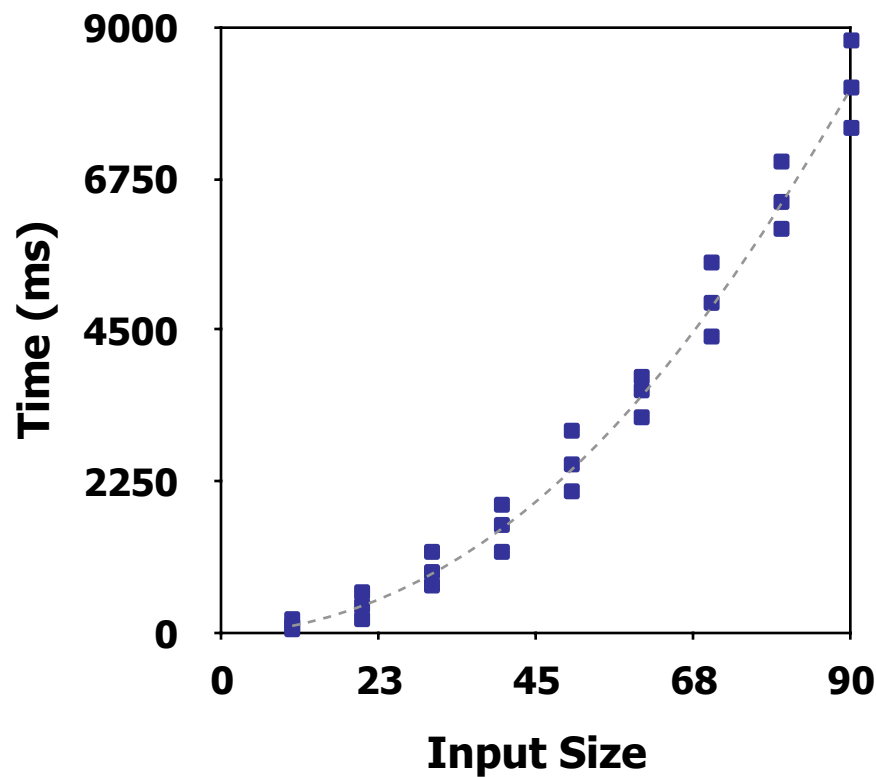
---

- Select a specific (typical) sample of inputs
- Use physical unit of time (e.g., milliseconds)  
or  
Count actual number of basic operation's executions
- Analyze the empirical data

# How to Calculate Running Time

---

- Almost all algorithms run longer on larger inputs
  - Write a program implementing the algorithm
  - Run the program with some inputs varying size and composition
  - Plot the results of measuring an input's size



# Limitations of experimental evaluation

---

- Experimental evaluation of running time is very useful but
  - It is necessary to implement the algorithm, which may be difficult (in terms of time) and can be expensive
  - 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

# Are the growth functions really matter?

---

Is the following statement true?

- With the advances in the speed of processors and the availability of large amounts of inexpensive memory, one can simply find a faster CPU to overcome the inefficiency of algorithm.
  - False

# Theoretical analysis of time efficiency

---

- Time efficiency is analyzed by determining the number of repetitions of the basic operation as a function of input size
  - let  $C(n)$  be the number of times this operation needs to be executed for the algorithm, and then  $T(n) \approx c \cdot C(n)$
  - $C(n)$  is computed approximately w.r.t. the order of growth
    - ex.  $C(n) = \frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \approx \frac{1}{2}n^2$

# How to theoretically calculate the running time?

---

- Most algorithms transform input objects into output objects



- The running time of an algorithm typically grows with the input size
  - idea: analyze running time  $T(n)$  as a function of input size



# Growth Functions

---

- Analysis is defined in general terms, based on:
  - the problem size (ex: number of items to sort)
  - key operation (ex: comparison of two values)
- A *growth function* shows the relationship between the size of the problem ( $n$ ) and the time it takes to solve the problem
- For example:

$$T(n) = 15n^2 + 45n$$

# Growth Functions

How much (unit) time is needed for a problem size of  $N$  if you have the growth function:  $t(n) = 15n^2 + 45n$

Number of dishes (n)	$15n^2$	$45n$	$15n^2 + 45n$
1	15	45	60
2	60	90	150
5	375	225	600
10	1,500	450	1,950
100	150,000	4,500	154,500
1,000	15,000,000	45,000	15,045,000
10,000	1,500,000,000	450,000	1,500,450,000
100,000	150,000,000,000	4,500,000	150,004,500,000
1,000,000	15,000,000,000,000	45,000,000	15,000,045,000,000
10,000,000	1,500,000,000,000,000	450,000,000	1,500,000,450,000,000

**FIGURE 2.1** Comparison of terms in growth function

# Why emphasis on the order of growth for large input sizes?

- A difference in running times on small inputs is not what really distinguishes efficient algorithms from inefficient ones.
- For large values of  $n$ , it is the function's order of growth that counts

**TABLE 2.1** Values (some approximate) of several functions important for analysis of algorithms

$n$	$\log_2 n$	$n$	$n \log_2 n$	$n^2$	$n^3$	$2^n$	$n!$
10	3.3	$10^1$	$3.3 \cdot 10^1$	$10^2$	$10^3$	$10^3$	$3.6 \cdot 10^6$
$10^2$	6.6	$10^2$	$6.6 \cdot 10^2$	$10^4$	$10^6$	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
$10^3$	10	$10^3$	$1.0 \cdot 10^4$	$10^6$	$10^9$		
$10^4$	13	$10^4$	$1.3 \cdot 10^5$	$10^8$	$10^{12}$		
$10^5$	17	$10^5$	$1.7 \cdot 10^6$	$10^{10}$	$10^{15}$		
$10^6$	20	$10^6$	$2.0 \cdot 10^7$	$10^{12}$	$10^{18}$		

## Exercise 2.1

8. For each of the following functions, indicate how much the function's value will change if its argument is increased fourfold.

**a.**  $\log_2 n$     **b.**  $\sqrt{n}$     **c.**  $n$     **d.**  $n^2$     **e.**  $n^3$     **f.**  $2^n$

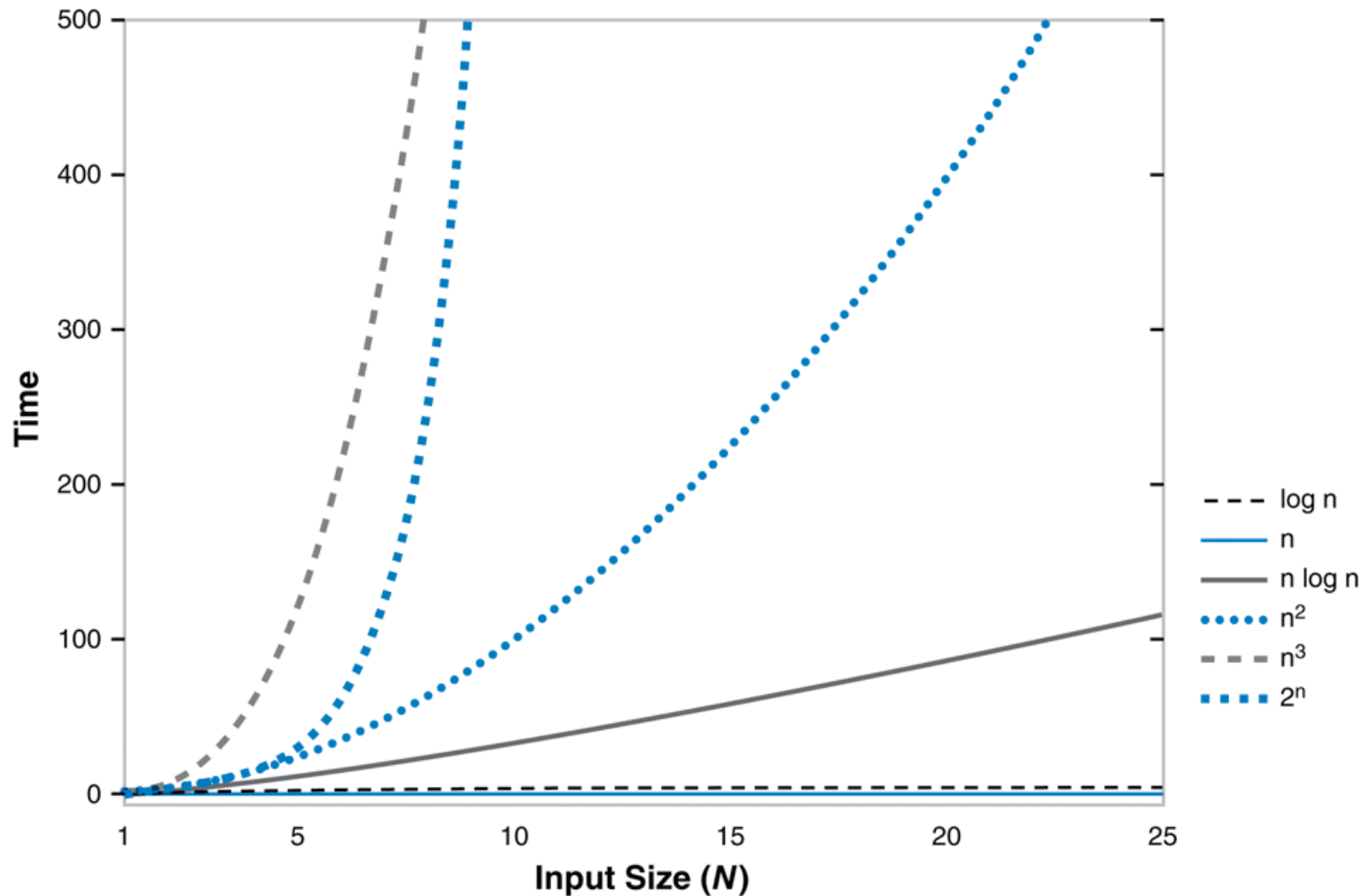
**Hint:** Use either the difference between or the ratio of  $f(4n)$  and  $f(n)$ , whichever is more convenient for getting a compact answer. If it is possible, try to get an answer that does not depend on  $n$ .

## Exercise 2.1

8. For each of the following functions, indicate how much the function's value will change if its argument is increased fourfold.

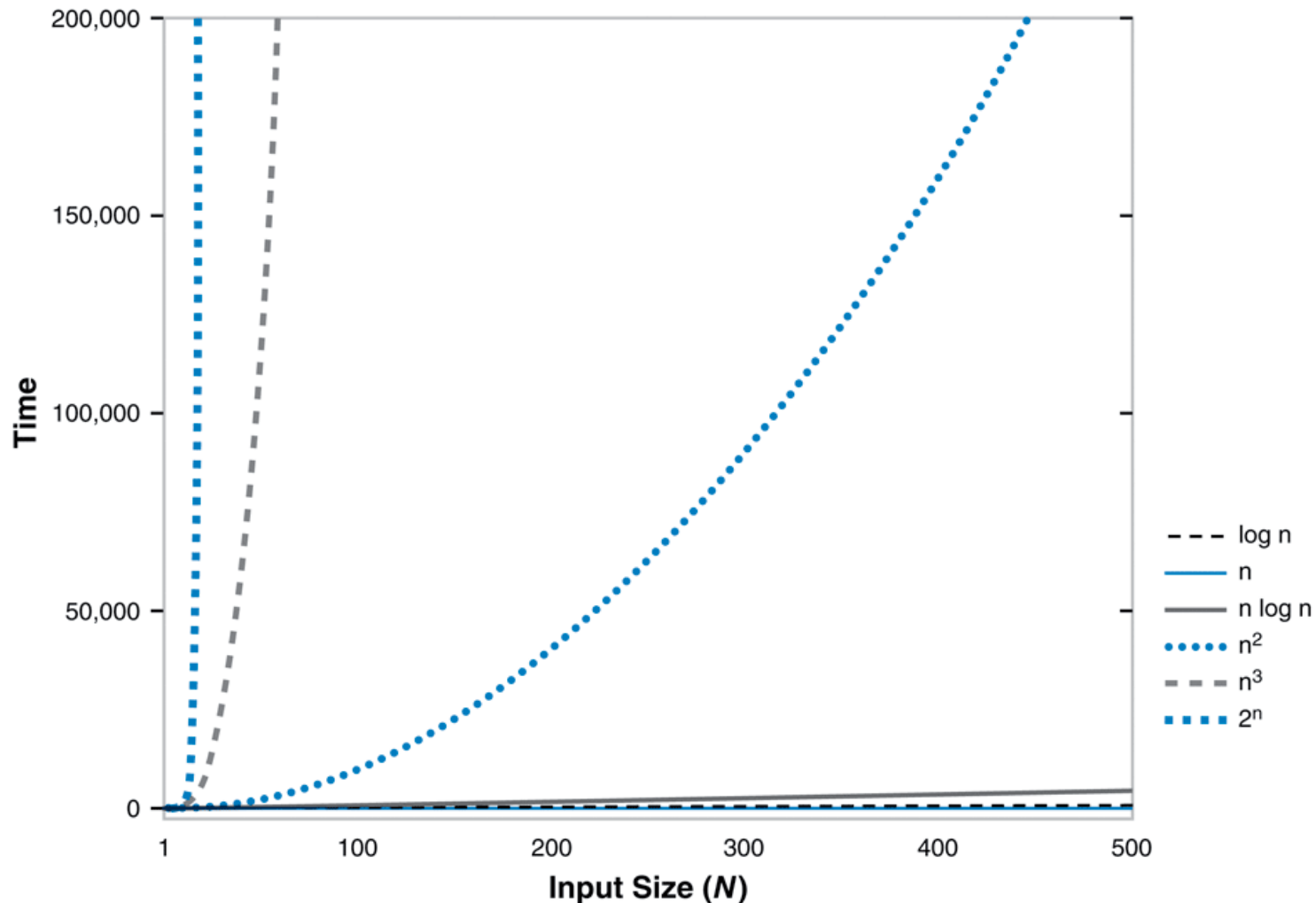
- a.**  $\log_2 n$     **b.**  $\sqrt{n}$     **c.**  $n$     **d.**  $n^2$     **e.**  $n^3$     **f.**  $2^n$

# Comparison of typical growth functions for small values of N



**FIGURE 2.4** Comparison of typical growth functions for small values of  $n$

# Comparison of typical growth functions for large values of $N$



**FIGURE 2.5** Comparison of typical growth functions for large values of  $n$

# Orders of growth of some important functions

---

- All logarithmic functions  $\log_a n$  belong to the same class  $\Theta(\log n)$  no matter what the logarithm's base  $a > 1$  is
- All polynomials of the same degree  $k$  belong to the same class:  
 $a_k n^k + a_{k-1} n^{k-1} + \dots + a_0 \in \Theta(n^k)$
- Exponential functions  $a^n$  have different orders of growth for different  $a$ 's
- order  $\log n < \text{order } n^\alpha \ (\alpha > 0) < \text{order } a^n < \text{order } n! < \text{order } n^n$



## Exercise 2.1

9. For each of the following pairs of functions, indicate whether the first function of each of the following pairs has a lower, same, or higher order of growth (to within a constant multiple) than the second function.
- |                                    |   |
|------------------------------------|---|
| <b>a.</b> $n(n + 1)$ and $2000n^2$ | <b>b.</b> $100n^2$ and $0.01n^3$        |
| <b>c.</b> $\log_2 n$ and $\ln n$   | <b>d.</b> $\log_2^2 n$ and $\log_2 n^2$ |
| <b>e.</b> $2^{n-1}$ and $2^n$      | <b>f.</b> $(n - 1)!$ and $n!$           |

# How to Calculate Running Time

- Running time can be very different even on inputs of the same size
  - Searching problem example: finds the first prime number in an array by scanning it left to right

5	3	1	2	8	4	7	6
1	4	6	8	5	3	2	7

**ALGORITHM** *SequentialSearch*( $A[0..n - 1]$ ,  $K$ )

//Searches for a given value in a given array by sequential search

//Input: An array  $A[0..n - 1]$  and a search key  $K$

//Output: The index of the first element in  $A$  that matches  $K$

// or  $-1$  if there are no matching elements

$i \leftarrow 0$

**while**  $i < n$  **and**  $A[i] \neq K$  **do**

$i \leftarrow i + 1$

**if**  $i < n$  **return**  $i$

**else return**  $-1$

- Idea: analyze running time in the
  - best case
  - worst case
  - average case

# How to Calculate Running Time

---

5	3	1	2	8	4	7	6
1	4	6	8	5	3	2	7

**ALGORITHM** *SequentialSearch*( $A[0..n-1], K$ )

//Searches for a given value in a given array by sequential search

//Input: An array  $A[0..n-1]$  and a search key  $K$

//Output: The index of the first element in  $A$  that matches  $K$

//            or  $-1$  if there are no matching elements

$i \leftarrow 0$

**while**  $i < n$  **and**  $A[i] \neq K$  **do**

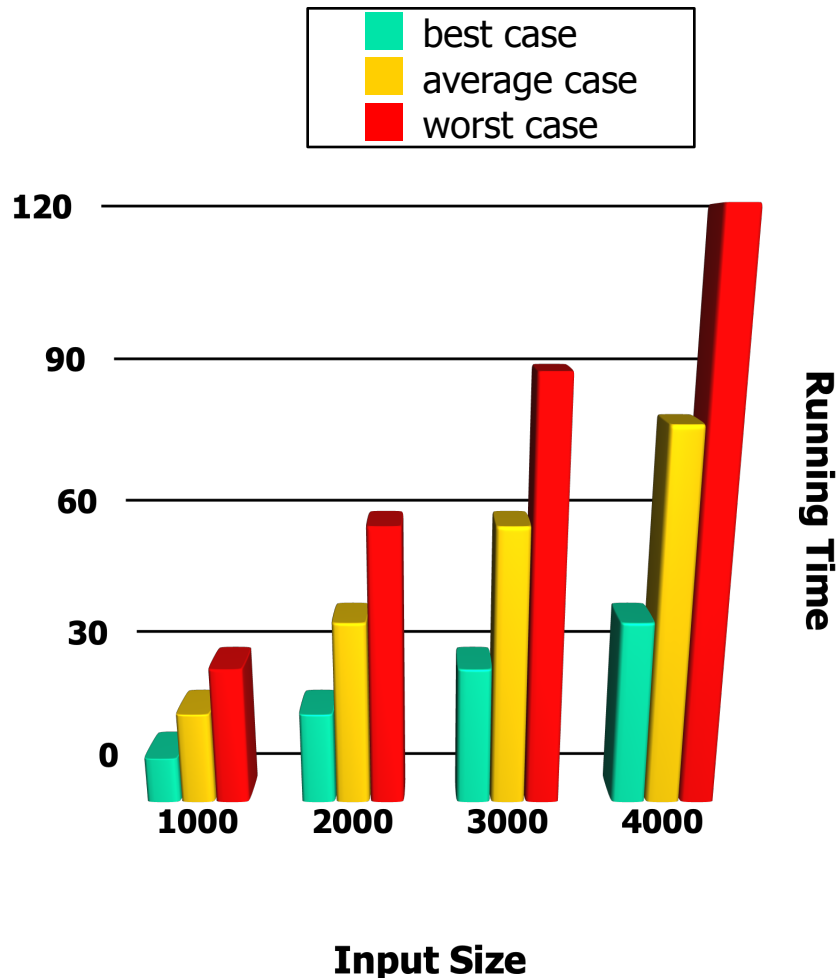
$i \leftarrow i + 1$

**if**  $i < n$  **return**  $i$

**else return**  $-1$

- Best case efficiency:
  - the first element equal to a search key
  - $C_{best}(n) = 1$
- Average case efficiency:
  - $C_{avg}(n) = [1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \dots + n \cdot \frac{p}{n}] + n * (1 - p)$
- Worst case efficiency:
  - The matching element happens to be the last one on the list
  - $C_{worst}(n) = n$

# How to Calculate Running Time



- Best case efficiency is usually useless
- Average case efficiency is very useful but often difficult to determine
- Worst case efficiency is
  - Easier to analyze
  - Crucial to applications such as games, finance and robotics

## Exercise 2.1

4. a. *Glove selection* There are 22 gloves in a drawer: 5 pairs of red gloves, 4 pairs of yellow, and 2 pairs of green. You select the gloves in the dark and can check them only after a selection has been made. What is the smallest number of gloves you need to select to have at least one matching pair in the best case? In the worst case?

## Exercise 2.1

- b.** *Missing socks* Imagine that after washing 5 distinct pairs of socks, you discover that two socks are missing. Of course, you would like to have the largest number of complete pairs remaining. Thus, you are left with 4 complete pairs in the best-case scenario and with 3 complete pairs in the worst case. Assuming that the probability of disappearance for each of the 10 socks is the same, find the probability of the best-case scenario; the probability of the worst-case scenario; the number of pairs you should expect in the average case.

# Summary

---

- Both time and space efficiencies are measured as functions of the algorithm's input size.
- Time efficiency is measured by counting the number of times the algorithm's basic operation is executed. Space efficiency is measured by counting the number of extra memory units consumed by the algorithm.
- The efficiencies of some algorithms may differ significantly for inputs of the same size. For such algorithms, we need to distinguish between the worst-case, average-case, and best-case efficiencies.
- The framework's primary interest lies in the order of growth of the algorithm's running time (extra memory units consumed) as its input size goes to infinity

## 2.2 Asymptotic Notation for Growth Functions

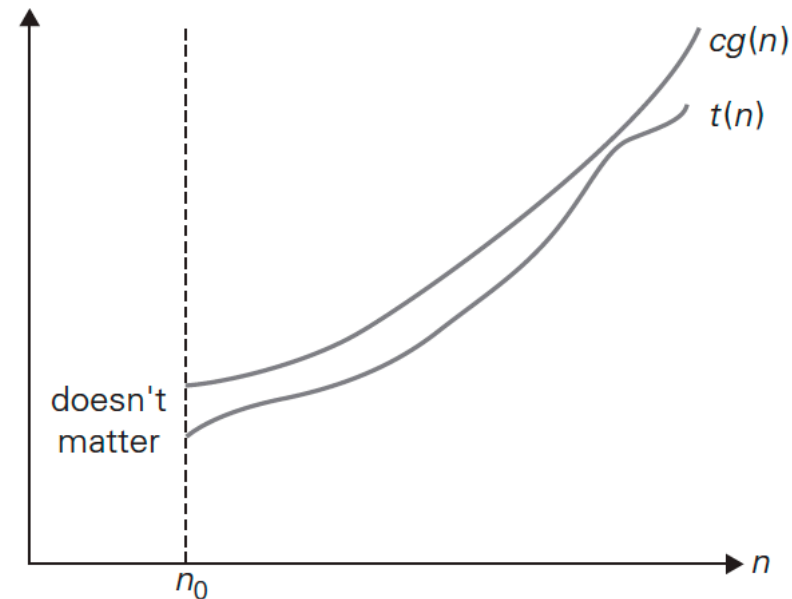
---

- It's not usually necessary to know the exact growth function
  - to specify the order, we use
    - $O$  (big oh):
    - $\Omega$  (big omega)
    - $\Theta$  (big theta)
- The key issue is the *asymptotic complexity* of the function
  - how it grows as  $n$  increases
- Determined by the dominant term in the growth function
  - this is referred to as *the order of the algorithm*
    - Example:  $O(n^2)$ ,  
 $n(n + 1) \in O(n^2)$ ,  
 $n^3 \notin O(n^2)$



# O Notation

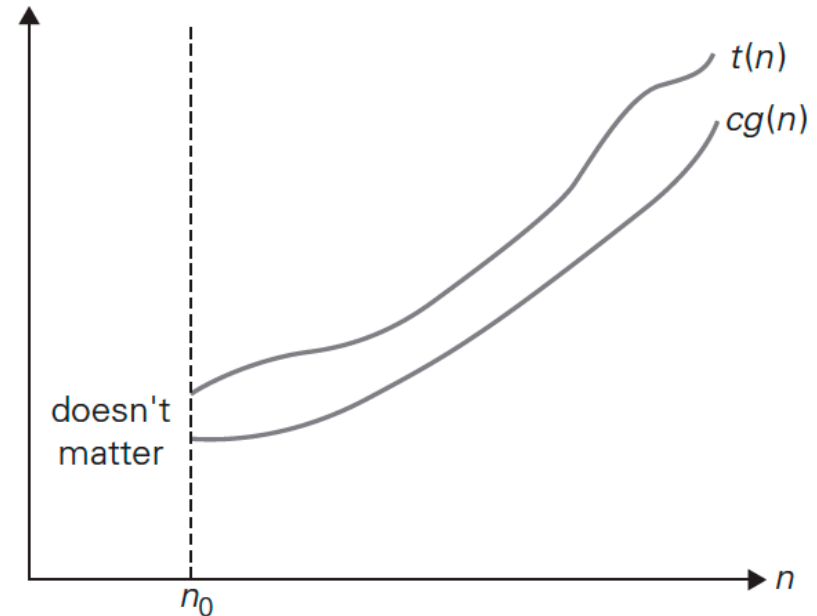
- O (big oh) notation
  - $t(n) \in O(g(n))$
  - If  $t(n)$  is bounded above by some constant multiple of  $g(n)$  for all large  $n$
  - i.e.  $t(n) \leq cg(n)$  for all  $n \geq n_0$
  - Example:  $t(n) = 100n + 5 \in O(n^2)$



**FIGURE 2.1** Big-oh notation:  $t(n) \in O(g(n))$ .

# $\Omega$ Notation

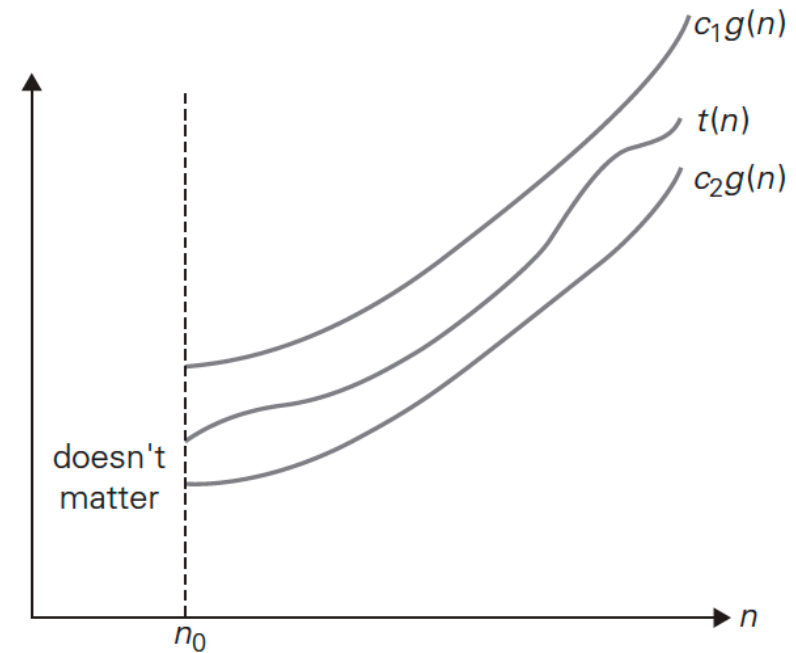
- $\Omega$  (big omega) notation
  - $t(n) \in \Omega(g(n))$
  - If  $t(n)$  is bounded below by some constant multiple of  $g(n)$  for all large  $n$
  - i.e.  $t(n) \geq cg(n)$  for all  $n \geq n_0$
  - Example:  $t(n) = n^3 \in \Omega(n^2)$



**FIGURE 2.2** Big-omega notation:  $t(n) \in \Omega(g(n))$ .

# Θ Notation

- Θ (big theta) notation
  - $t(n) \in \Theta(g(n))$
  - If  $t(n)$  is bounded both above and below by some constant multiple of  $g(n)$  for all large  $n$
  - i.e.  $c_2g(n) \leq t(n) \leq c_1g(n)$   
for all  $n \geq n_0$
  - Example:  $t(n) = n(n - 1) \in \Theta(n^2)$



**FIGURE 2.3** Big-theta notation:  $t(n) \in \Theta(g(n))$ .

# Establishing order of growth using the definition

---

Definition:  $f(n)$  is in  $O(g(n))$  if order of growth of  $f(n) \leq$  order of growth of  $g(n)$  (within constant multiple),  
i.e., there exist positive constant  $c$  and non-negative integer  $n_0$  such that

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

Examples:

- $10n$  is  $O(n^2)$
- $5n+20$  is  $O(n)$

# Some properties of asymptotic order of growth

---

- $f(n) \in O(f(n))$
- $f(n) \in O(g(n))$  iff  $g(n) \in \Omega(f(n))$
- If  $f(n) \in O(g(n))$  and  $g(n) \in O(h(n))$ , then  $f(n) \in O(h(n))$

Note similarity with  $a \leq b$

- If  $f_1(n) \in O(g_1(n))$  and  $f_2(n) \in O(g_2(n))$ , then
$$f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\})$$

## Exercise 2.2

2. Use the informal definitions of  $O$ ,  $\Theta$ , and  $\Omega$  to determine whether the following assertions are true or false.

**a.**  $n(n + 1)/2 \in O(n^3)$       **b.**  $n(n + 1)/2 \in O(n^2)$   
**c.**  $n(n + 1)/2 \in \Theta(n^3)$       **d.**  $n(n + 1)/2 \in \Omega(n)$

# Useful Property for the Asymptotic Notations

---

- Entire algorithm's efficiency

If  $t_1(n) \in O(g_1(n))$  and  $t_2(n) \in O(g_2(n))$ , then

$$t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$$

- Example:  $t_1(n) = n(n - 1) \in O(n^2)$  and  $t_2(n) = n - 1 \in O(n)$   
entire algorithm's efficiency is  $O(\max\{g_1(n), g_2(n)\}) = n^2$

# Useful Property for the Asymptotic Notations

**PROOF** The proof extends to orders of growth the following simple fact about four arbitrary real numbers  $a_1, b_1, a_2, b_2$ : if  $a_1 \leq b_1$  and  $a_2 \leq b_2$ , then  $a_1 + a_2 \leq 2 \max\{b_1, b_2\}$ .

Since  $t_1(n) \in O(g_1(n))$ , there exist some positive constant  $c_1$  and some non-negative integer  $n_1$  such that

$$t_1(n) \leq c_1 g_1(n) \quad \text{for all } n \geq n_1.$$

Similarly, since  $t_2(n) \in O(g_2(n))$ ,

$$t_2(n) \leq c_2 g_2(n) \quad \text{for all } n \geq n_2.$$

Let us denote  $c_3 = \max\{c_1, c_2\}$  and consider  $n \geq \max\{n_1, n_2\}$  so that we can use both inequalities. Adding them yields the following:

$$\begin{aligned} t_1(n) + t_2(n) &\leq c_1 g_1(n) + c_2 g_2(n) \\ &\leq c_3 g_1(n) + c_3 g_2(n) = c_3 [g_1(n) + g_2(n)] \\ &\leq c_3 2 \max\{g_1(n), g_2(n)\}. \end{aligned}$$

Hence,  $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$ , with the constants  $c$  and  $n_0$  required by the  $O$  definition being  $2c_3 = 2 \max\{c_1, c_2\}$  and  $\max\{n_1, n_2\}$ , respectively. ■



# Useful Property for the Asymptotic Notations

---

- Comparing Orders of Growth

$$\lim_{n \rightarrow \infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n), \\ c & \text{implies that } t(n) \text{ has the same order of growth as } g(n), \\ \infty & \text{implies that } t(n) \text{ has a larger order of growth than } g(n).^3 \end{cases}$$

Examples:

•  $10n$                       vs.                       $n^2$

•  $n(n+1)/2$                       vs.                       $n^2$

# Example

---

**EXAMPLE 1** Compare the orders of growth of  $\frac{1}{2}n(n - 1)$  and  $n^2$ . (This is one of the examples we used at the beginning of this section to illustrate the definitions.)

$$\lim_{n \rightarrow \infty} \frac{\frac{1}{2}n(n - 1)}{n^2} = \frac{1}{2} \lim_{n \rightarrow \infty} \frac{n^2 - n}{n^2} = \frac{1}{2} \lim_{n \rightarrow \infty} \left(1 - \frac{1}{n}\right) = \frac{1}{2}.$$

Since the limit is equal to a positive constant, the functions have the same order of growth or, symbolically,  $\frac{1}{2}n(n - 1) \in \Theta(n^2)$ . ■

# Useful Property for the Asymptotic Notations

---

- Comparing Orders of Growth

The limit-based approach is often more convenient than the one based on the definitions because it can take advantage of the powerful calculus techniques developed for computing limits, such as L'Hôpital's rule

$$\lim_{n \rightarrow \infty} \frac{t(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{t'(n)}{g'(n)}$$

Example:  $\log n$  vs.  $n$

Example:  $\log n$  vs.  $\sqrt{n}$

# Example

---

**EXAMPLE 2** Compare the orders of growth of  $\log_2 n$  and  $\sqrt{n}$ . (Unlike Example 1, the answer here is not immediately obvious.)

$$\lim_{n \rightarrow \infty} \frac{\log_2 n}{\sqrt{n}} = \lim_{n \rightarrow \infty} \frac{(\log_2 n)'}{(\sqrt{n})'} = \lim_{n \rightarrow \infty} \frac{(\log_2 e) \frac{1}{n}}{\frac{1}{2\sqrt{n}}} = 2 \log_2 e \lim_{n \rightarrow \infty} \frac{1}{\sqrt{n}} = 0.$$

Since the limit is equal to zero,  $\log_2 n$  has a smaller order of growth than  $\sqrt{n}$ . (Since  $\lim_{n \rightarrow \infty} \frac{\log_2 n}{\sqrt{n}} = 0$ , we can use the so-called ***little-oh notation***:  $\log_2 n \in o(\sqrt{n})$ . Unlike the big-Oh, the little-oh notation is rarely used in analysis of algorithms.)

# Useful Property for the Asymptotic Notations

---

- Sterling's formula


$$n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \text{ for large values of } n.$$

# Example

---

**EXAMPLE 3** Compare the orders of growth of  $n!$  and  $2^n$ . (We discussed this informally in Section 2.1.) Taking advantage of Stirling's formula, we get

$$\lim_{n \rightarrow \infty} \frac{n!}{2^n} = \lim_{n \rightarrow \infty} \frac{\sqrt{2\pi n} \left(\frac{n}{e}\right)^n}{2^n} = \lim_{n \rightarrow \infty} \sqrt{2\pi n} \frac{n^n}{2^n e^n} = \lim_{n \rightarrow \infty} \sqrt{2\pi n} \left(\frac{n}{2e}\right)^n = \infty.$$

Thus, though  $2^n$  grows very fast,  $n!$  grows still faster. We can write symbolically that  $n! \in \Omega(2^n)$ ; note, however, that while the big-Omega notation does not preclude the possibility that  $n!$  and  $2^n$  have the same order of growth, the limit computed here certainly does. 

## Exercise 2.2

3. For each of the following functions, indicate the class  $\Theta(g(n))$  the function belongs to. (Use the simplest  $g(n)$  possible in your answers.) Prove your assertions.

a.  $(n^2 + 1)^{10}$

b.  $\sqrt{10n^2 + 7n + 3}$

c.  $2n \lg(n + 2)^2 + (n + 2)^2 \lg \frac{n}{2}$

d.  $2^{n+1} + 3^{n-1}$

e.  $\lfloor \log_2 n \rfloor$

# Basic asymptotic efficiency classes

---

Class	Name
1	constant
$\log n$	logarithmic
$n$	linear
$n \log n$	$n$ -log- $n$ or linearithmic
$n^2$	quadratic
$n^3$	cubic
$2^n$	exponential
$n!$	factorial



# Analyzing Loop Execution

- A loop executes a certain number of times (say  $n$ )
- Thus the complexity of a loop is  $n$  times the complexity of the body of the loop
- When loops are nested, the body of the outer loop includes the complexity of the inner loop

# Analyzing Loop Execution

- A loop executes a certain number of times (say  $n$ )
- Thus the complexity of a loop is  $n$  times the complexity of the body of the loop
- When loops are nested, the body of the outer loop includes the complexity of the inner loop
  - Example:

```
x=0;
for (int i=0; i<n; i++) {
    x = x + 1;
}
```
  - The time complexity of the loop is  $O(n)$  because the loop executes  $n$  times and the body of the loop is  $O(1)$

# Analyzing Loop Execution

- What is the time complexity of the following loop?

```
for (int i=0; i<n; i++)  
{  
    x = x + 1;  
    for (int j=0; j<n; j++)  
    {  
        y = y - 1;  
    }  
}
```

$O(n^2)$

$O(n)$

- The time complexity of the loop is  $O(n^2)$  because the loop executes  $n$  times and the body of the loop, including a nested loop, is  $O(n)$

# Examples

- Find the sum of 1 to n.

```
int sum=0;
for (int i=1; i<=n; i++) {
    sum = sum + i;
}
```

Does there exist a  
better algorithm?

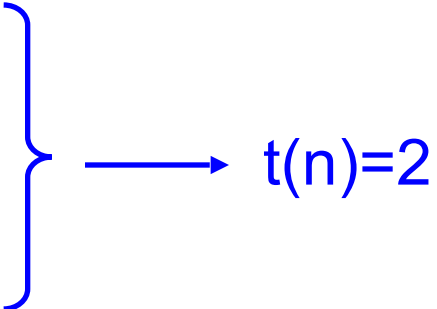
```
int sum = n*(n+1)/2;
```

- The time complexity of the for loop is  $O(n)$
- The time complexity of the formula is  $O(1)$

# Analyzing Method Calls

- To analyze method calls, we simply replace the method call with the order of the body of the method
- A call to the following method is  $O(1)$

```
public void printsum(int count)
{
    sum = count*(count+1)/2;
    System.out.println(sum);
}
```



$t(n)=2$

# More examples

- What is the time complexity of the following while loop?

```
while (count < n) {  
    x = x + 1;  
    count++;  
}
```

```
while (count < 2n) {  
    x = x + 2;  
    count++;  
}
```

- The time complexity of the either while loop is  $O(n)$

## More examples:

---

```
for (int count=0; count<n; count++)  
{  
    print_sum(count);  
}  
.....
```

} The time complexity  
is  $O(n^2)$

```
public void print_sum(int count)  
{  
    int sum=0;  
    for (int i=0; i<count; i++)  
    {  
        sum = sum + i;  
    }  
    System.out.println(sum);  
}
```

# Big-Oh Examples

---

- $7n-2$ 
  - $7n-2$  is  $O(n)$
- $3n^3 + 20n^2 + 5$ 
  - $3n^3 + 20n^2 + 5$  is  $O(n^3)$
- $3 \log n + 5$ 
  - $3 \log n + 5$  is  $O(\log n)$

## Rules of Big-Oh

1. Drop lower-order terms
2. Drop constant factors

## Relatives of Big-Oh?

Big-Omega and Big-Theta