

# ***Analysis and Design of Algorithms***

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



*School of Software Engineering © Ye Luo*



# *Fundamentals of the Analysis of Algorithm Efficiency*

---

- *Algorithm analysis framework*
- *Asymptotic notations*
- *Analysis of non-recursive algorithms*
- *Analysis of recursive algorithms*

Since there are often many possible algorithms or programs that compute the same results, we would like to use the one that is fastest.

- *How do we decide how fast an algorithm is?* Since knowing how fast an algorithm runs for a certain input does not reveal anything about how fast it runs on other inputs, we need another measure that tells us how fast it is for any input. A formula that relates input size to the running time of the algorithm satisfies this requirement.

- *We also want to ignore machine dependent factors.* If an algorithm takes two seconds on one machine for a given input, a trivial way to get it to run in one second is to use a machine that is twice as fast. There is a constant multiplicative factor relating the speed of an algorithm on one machine and its speed on another, which we will ignore.

- *We are only interested in how fast an algorithm runs on large inputs,* since even slow algorithms finish quickly on small inputs.

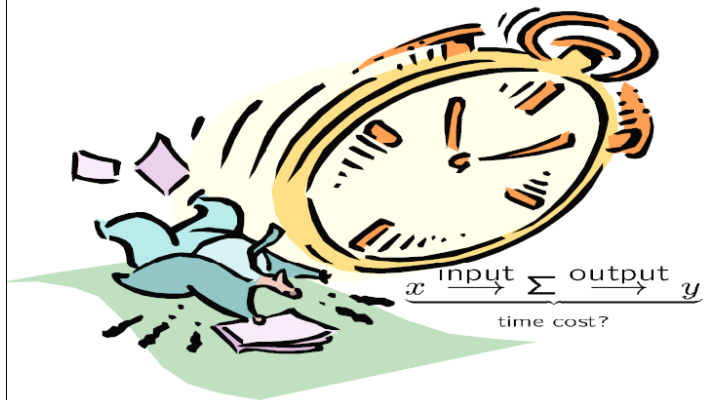
# Analysis of Algorithms

- **Analysis of algorithms means to investigate an algorithm's efficiency with respect to resources: running time and memory space.**

- ✦ with respect to *input size*, *input type*, and *algorithm function*

$$C = F(N, I, A)$$

- ✦ Time efficiency  $T(N, I)$ :  
how fast an algorithm runs.
- ✦ Space efficiency  $S(N, I)$ :  
the space an algorithm requires.



# *Analysis of Algorithms*

---

## ■ *Analysis Framework*

- ✦ *Measuring running time*
- ✦ *Measuring an input's size*
- ✦ *Orders of growth (of the algorithm's efficiency function)*
- ✦ *Worst-base, best-case and average-case efficiency*

# Analysis of Algorithms

---

## ■ Units for Measuring Running Time

- ✦ *Should we measure the running time using standard unit of time measurements, such as seconds, minutes?*
  - ➔ Depends on the speed of the computer
  - ➔ Depends on the quality of programing
- ✦ *Count the number of times each element operation is executed.*
  - ➔ Difficult and unnecessary
- ✦ **Solution:** *Count the number of times an algorithm's basic operation is executed.*
  - **Basic operation:** *the operation that contributes the most to the total running time.*
  - *For example, the basic operation is usually the **most time-consuming operation** in the algorithm's **innermost loop**.*

# *Analysis of Algorithms*

---

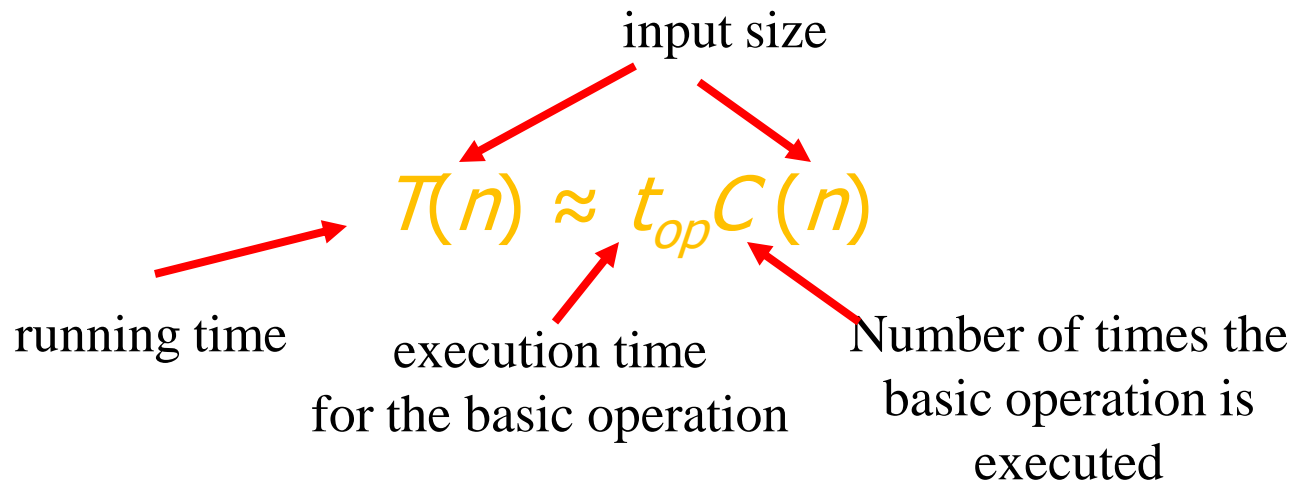
## ■ **Measuring Input Size**

- ✦ *Efficiency is defined as a function of input size*
- ✦ *Typically, algorithms run longer as the size of its input increases*
- ✦ *Input size depends on the problem.*
  - *Examples*
- ✦ *We are interested in **how efficiency scales wrt input size***

# Analysis of Algorithms

## ■ 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.



The efficiency analysis framework ignores the multiplicative constants and **focuses on the orders of growth of the  $C(n)$ .**

# Analysis of Algorithms

---

## ■ Order of growth

Exponential-  
growth functions

$n$	$\log_2 n$	$n$	$n \log_2 n$	$n^2$	$n^3$	$2^n$	$n!$
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	$10^6$	$2.0 \cdot 10^7$	$10^{12}$	$10^{18}$		



# Analysis of Algorithms

## ■ Order of growth

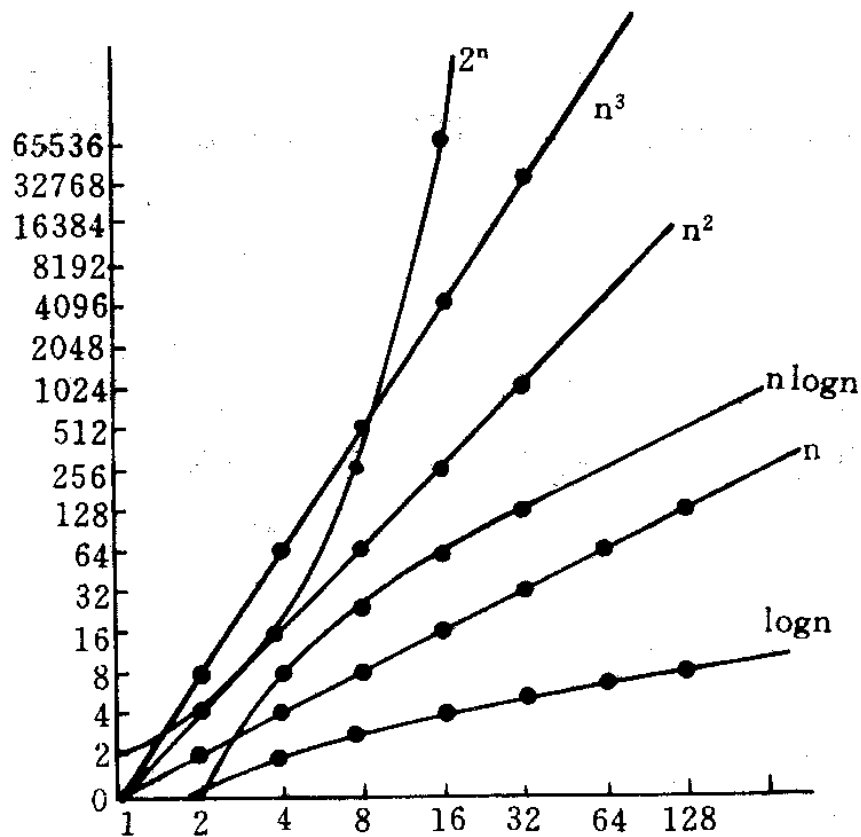


图 1.1 一般计算时间函数的曲线

# Analysis of Algorithms

---

## ■ Basic Efficiency classes

The time efficiencies of a large number of algorithms fall into only a few classes.

fast	1	constant	High time efficiency
	$\log n$	logarithmic	
	$n$	linear	
	$n \log n$	$n \log n$	
	$n^2$	quadratic	
	$n^3$	cubic	
	$2^n$	exponential	
slow	$n!$	factorial	low time efficiency

# Analysis of Algorithms

---

## ■ **Worst-Case, Best-Case, and Average-Case Efficiency**

✦ *For some algorithms efficiency depends on type of input.*

### ■ **Example: Sequential Search**

- *Problem:* Given a list of  $n$  elements and a search key  $K$ , find an element equal to  $K$ , if any.
- *Algorithm:* Scan the list and compare its successive elements with  $K$  until either a matching element is found (*successful search*) or the list is exhausted (*unsuccessful search*)

Given a sequential search problem of an input size of  $n$ ,  
what kind of input would make the running time the longest?  
How many key comparisons?

# Analysis of Algorithms

---

- *Example: Sequential Search*

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: Returns the index of the first element of  $A$  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  $i < n$                       // $A[i] = K$

    return  $i$

else

    return  $-1$

# Analysis of Algorithms

---

## ▪ Example: Sequential Search

- probability for successful search is  $p$  ( $0 \leq p \leq 1$ );
- probability for successful search on each position  $i$  ( $0 \leq i < n$ ) in an array is equal,  $p/n$ .

$$T_{avg}(n) = \sum_{size(I)=n} p(I)T(I)$$

$$= \left( 1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + 3 \cdot \frac{p}{n} + \dots + n \cdot \frac{p}{n} \right) + n \cdot (1 - p)$$

$$= \frac{p}{n} \sum_{i=1}^n i + n(1 - p) = \frac{p(n+1)}{2} + n(1 - p)$$

- ✓ dividing all instances of size  $n$  into several classes so that for each instance of the class the number of times the basic operation is executed is the same;
- ✓ a probability distribution of inputs is obtained or assumed

- $T_{worst}(n) = n$
- $T_{bset}(n) = 1$

# Analysis of Algorithms

---

- ✦ **Problem:** impossible to calculate  $e_i$  for every legal input  $I$
- ✦ **Solution:** to calculate  $e_i$  for some representative input
- ✦ **Worst case Efficiency**
  - *Efficiency (# of times the basic operation will be executed) for the worst case input of size  $n$*
  - *The algorithm runs the **longest** among all possible inputs of size  $n$*
  - *To see what kind of inputs yield the **largest** value of the basic operation's count  $C(n)$  among all possible inputs of size  $n$*
  - *Bounding an algorithm's efficiency from above*

# Analysis of Algorithms

---

## ✦ Best case

- *Efficiency (# of times the basic operation will be executed) for the best case input of size  $n$ .*
- *The algorithm runs the **fastest** among all possible inputs of size  $n$ .*
- *To see what kind of inputs yield the **smallest** value of the basic operation's count  $C(n)$  among all possible inputs of size  $n$*
- ***Bounding an algorithm's efficiency from above***
- *If the best-case efficiency of an algorithm is unsatisfactory, we can immediately discard it.*

# Analysis of Algorithms

---

## ✦ Average case:

- *Efficiency (#of times the basic operation will be executed) for a **typical/random** input of size  $n$ .*
- *NOT the average of worst and best case*
- *How to find the average case efficiency?*

$$T_{avg}(N) = \sum_{I \in D_N} P(I)T(N, I) = \sum_{I \in D_N} P(I) \sum_{i=1}^k t_i e_i(N, I)$$

$T_{avg}$  cannot be obtained by taking the average of  $T_{worst}$  and  $T_{best}$



# Analysis of Algorithms

---

## ■ Summary of the Analysis Framework

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

# Asymptotic notations

---

## ■ Asymptotic complexity

✦ If

$$T(n) \rightarrow \infty, \text{ as } n \rightarrow \infty;$$

$$(T(n) - t(n)) / T(n) \rightarrow 0, \text{ as } n \rightarrow \infty;$$

Then,  $t(n)$  is called **asymptotic state** of  $T(n)$ ,  $n \rightarrow \infty$

$t(n)$  is called **asymptotic complexity** of algorithm  $A$ ,  $n \rightarrow \infty$

### ■ Example:

$$\text{for } T(n) = 3n^2 + 4n \log n + 7, \quad t(n) = 3n^2$$

✦  $t(n)$  consider only the leading term of  $T(n)$

✦ ignore the constant coefficient

✦ only consider the **rank** of  $t(n)$

# Asymptotic notations

---

## ■ **Three notations used to compare orders of growth of an algorithm's basic operation count**

- ✦  $O(g(n))$ : class of functions  $t(n)$  that grow no faster than  $g(n)$   
Upper Bound
- ✦  $\Omega(g(n))$ : class of functions  $t(n)$  that grow at least as fast as  $g(n)$
- ✦  $\Theta(g(n))$ : class of functions  $t(n)$  that grow at same rate as  $g(n)$

# Asymptotic notations

## ■ O-notation

To prove formally that  $2^n + n^3$  is  $O(2^n)$ , let  $n_0 = 10$  and  $c = 2$ . We must show that for  $n \geq 10$ , we have

$$2^n + n^3 \leq 2 \times 2^n$$

✦ **Formal definition:** If we subtract  $2^n$  from both sides, we see it is sufficient to show that for  $n \geq 10$ , it is the case that  $n^3 \leq 2^n$ .

- A function  $t(n)$  is said to be in  $O(g(n))$ , denoted  $t(n) \in O(g(n))$ , if  $t(n)$  is **bounded above** by some constant multiple of  $g(n)$  for all large  $n$ , i.e., if there exist some positive constant  $c$  and some nonnegative integer  $n_0$  such that

$$t(n) \leq cg(n) \quad \text{for all } n \geq n_0$$

■ **E.g.:**

$$100n + 5 \leq 100n + n = 101n \leq 101n^2$$

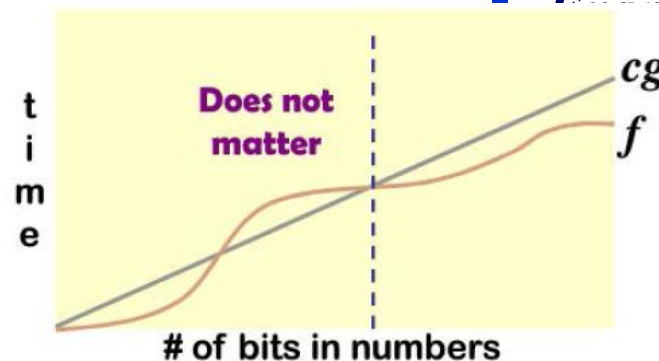
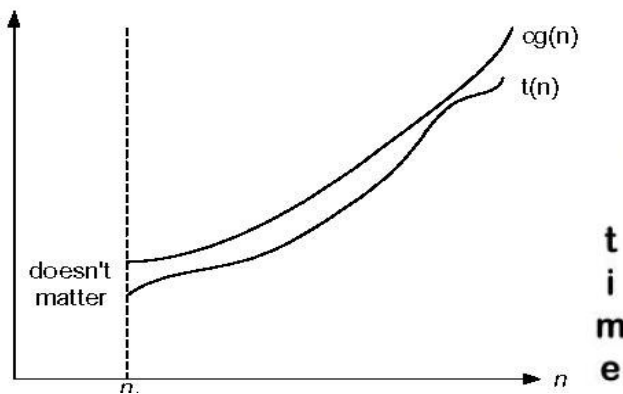
Example:

$$2 \in O(n^2)$$

$$2 + 2n \in O(n^2)$$

$$n + 5 \in O(n^2)$$

$$20 \in O(n)$$



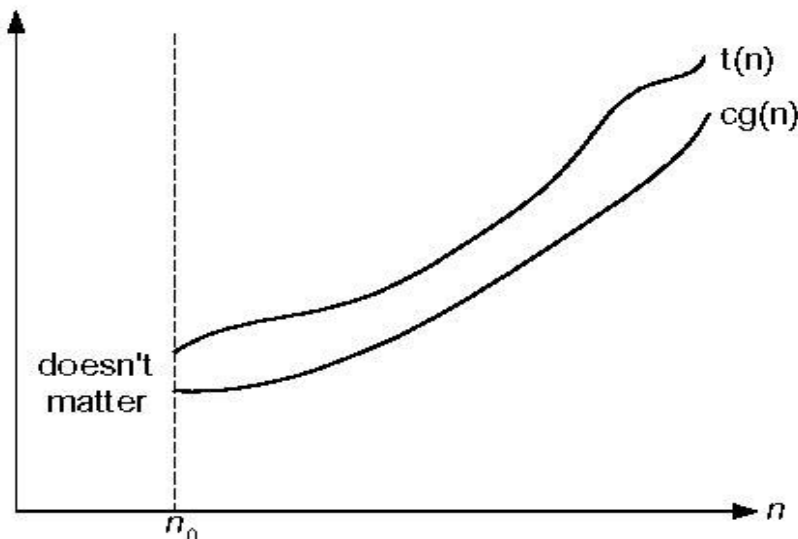
# Asymptotic notations

## ■ $\Omega$ -notation

### ✦ Formal definition:

- A function  $t(n)$  is said to be in  $\Omega(g(n))$ , denoted  $t(n) \in \Omega(g(n))$ , if  $t(n)$  is **bounded below** by some constant multiple of  $g(n)$  for all large  $n$ , i.e., if there exist some positive constant  $c$  and some nonnegative integer  $n_0$  such that

$$t(n) \geq cg(n) \quad \text{for all } n \geq n_0$$



### ■ Example:

- ♣  $10n^2 \in \Omega(n^2)$
- ♣  $10n^2 + 2n \in \Omega(n^2)$
- ♣  $10n^3 \in \Omega(n^2)$

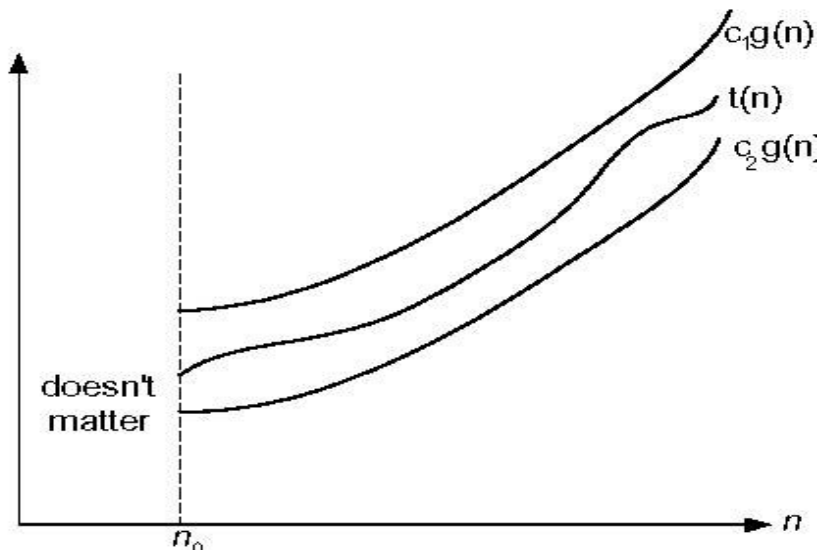
# Asymptotic notations

## ■ $\Theta$ -notation

### ✦ Formal definition:

- A function  $t(n)$  is said to be in  $\Theta(g(n))$ , denoted  $t(n) \in \Theta(g(n))$ , if  $t(n)$  is **bounded both above and below** by some positive constant multiples of  $g(n)$  for all large  $n$ , i.e., if there exist some positive constant  $c_1$  and  $c_2$  and some nonnegative integer  $n_0$  such that

$$c_2 g(n) \leq t(n) \leq c_1 g(n) \quad \text{for all } n \geq n_0$$



### ■ Example:

- ♣  $10n^2 \in \Theta(n^2)$
- ♣  $an^2 + bn + c \in \Theta(n^2)$   
with  $a > 0$
- ♣  $(1/2)n(n-1) \in \Theta(n^2)$
- ♣  $n^2 + \log n \in \Theta(n^2)$

$$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$$

# Asymptotic notations

---

## ■ Other notations

### ✦ $o(g(n))$ :

- A function  $t(n)$  is said to be in  $o(g(n))$ , denoted  $t(n) \in o(g(n))$ , if  $t(n)$  is **bounded above** by some positive constant multiples of  $g(n)$  for all large  $n$ , i.e., if there exist some positive constant  $c$  and some nonnegative integer  $n_0$  such that

$$t(n) < cg(n) \quad \text{for all } n \geq n_0$$

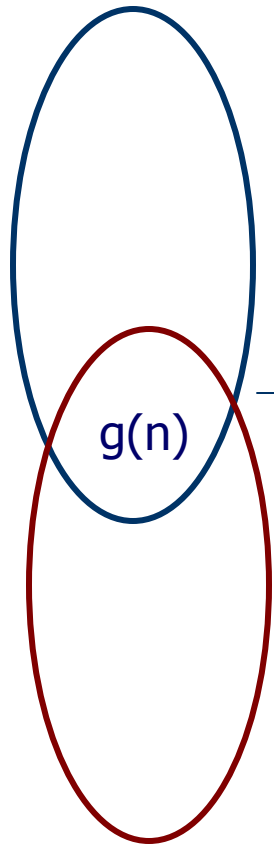
### ✦ $\omega(g(n))$ :

- A function  $t(n)$  is said to be in  $\omega(g(n))$ , denoted  $t(n) \in \omega(g(n))$ , if  $t(n)$  is **bounded below** by some positive constant multiples of  $g(n)$  for all large  $n$ , i.e., if there exist some positive constant  $c$  and some nonnegative integer  $n_0$  such that

$$t(n) > cg(n) \quad \text{for all } n \geq n_0$$

# Asymptotic notations

---



$\geq$  higher or same order of growth

——  $\Omega(g(n))$ , functions that grow at least as fast as  $g(n)$

$=$

——  $\Theta(g(n))$ , functions that grow at the same rate as  $g(n)$

$\leq$  smaller or same order of growth

——  $O(g(n))$ , functions that grow no faster than  $g(n)$



# Asymptotic notations

---

## ■ Some Properties of Asymptotic Order of Growth

- ✦  $f(n) \in O(f(n))$  反身性
- ✦  $f(n) \in O(g(n)), g(n) \in O(h(n)) \Rightarrow f(n) \in O(h(n))$ ; 传递性
- ✦  $f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$  互对称性  
 $f(n) \in \Theta(g(n)) \Leftrightarrow g(n) \in \Theta(f(n))$  对称性
- ✦  $O(g_1(n)) + O(g_2(n)) = O(g_1(n) + g_2(n))$  ; 数学计算
- ✦  $O(g_1(n)) * O(g_2(n)) = O(g_1(n) * g_2(n))$  ;
- ✦  $O(cf(n)) = O(f(n))$  ;
- ✦  $g(n) = O(f(n)) \Rightarrow O(f(n)) + O(g(n)) = O(f(n))$

*The analogous assertions are true for the  $\Omega$ -notation and  $\Theta$ -notation.*

# Asymptotic notations

---

## ■ **Some Properties of Asymptotic Order of Growth**

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

✦ *Implication:*

*for an algorithm that comprises two consecutively executed parts,  
The algorithm's overall efficiency will be determined by  
the part with a larger order of growth.*

### ■ **Example:**

✦  $5n^2 + 3n \log n \in O(n^2)$

✦ *check whether an array has identical elements:*

*first, sort the array by some sorting alg.,*

*——no more than  $(1/2)n(n-1)$  comparisons*

*then, scan the sorted array to check its consecutive  
elements for equality*

*——no more than  $(n-1)$  comparisons*

# Asymptotic notations

## ■ Using Limits for Comparing Orders of Growth

$$\lim_{n \rightarrow \infty} T(n)/g(n) = \begin{cases} 0 & \text{order of growth of } T(n) < \text{order of growth of } g(n) \\ c > 0 & \text{order of growth of } T(n) = \text{order of growth of } g(n) \\ \infty & \text{order of growth of } T(n) > \text{order of growth of } g(n) \end{cases}$$

case 1 & 2,  $T(n) \in O(g(n))$ ;      case 3 & 2,  $T(n) \in \Omega(g(n))$ ;  
case 2,  $T(n) \in \Theta(g(n))$ ;

### ■ Example:

$$\blacktriangleright 5n^2 + 3n \log n \in O(n^2)$$

$$\blacktriangleright 10n \quad \text{vs.} \quad 2n^2$$

$$\blacktriangleright n(n+1)/2 \quad \text{vs.} \quad n^2$$

$$\blacktriangleright \log_b n \quad \text{vs.} \quad \log_c n$$

# Asymptotic notations

## ■ L'Hôpital's rule

✦ If  $\lim_{n \rightarrow \infty} f(n) = \lim_{n \rightarrow \infty} g(n) = \infty$  and the derivatives  $f'$ ,  $g'$  exist, Then

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{f'(n)}{g'(n)}$$

$$\lim_{n \rightarrow \infty} \frac{\log_2 n}{\sqrt{n}} = \lim_{n \rightarrow \infty} \frac{(\log_2 n)'}{(n^{\frac{1}{2}})'} = \lim_{n \rightarrow \infty} \frac{\frac{1}{n \ln 2}}{\frac{1}{2} n^{-\frac{1}{2}}} = \frac{2}{\ln 2} \lim_{n \rightarrow \infty} \frac{n^{\frac{1}{2}}}{n} = 0$$
$$\Rightarrow \log_2 n \in O(n^{\frac{1}{2}})$$

### ■ Example:

- ✦  $(1/2)n(n-1) \in \Theta(n^2)$
- ✦  $\log_2 n \in O(n^{1/2})$
- ✦  $\log_2 n$  vs.  $n$
- ✦  $2^n$  vs.  $n!$

# Asymptotic notations

---

## ■ Orders of growth by 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.
- ✦  $\text{order } \log n < \text{order } n^\alpha \ (\alpha > 0) < \text{order } a^n < \text{order } n! < \text{order } n^n$

# Asymptotic notations

---

## ■ Summary of How to Establish Orders of Growth of an Algorithm's Basic Operation Count

- ✦ Method 1: Using limits

  - *L'Hôpital's rule*

- ✦ Method 2: Using the properties

- ✦ Method 3: Using the definitions of  $O$ -,  $\Omega$ -, and  $\Theta$ -notation.

*the time efficiencies of a large number of algorithms fall into a few classes, as see in the list.*

# *Time Efficiency of Non-recursive Algorithms*

---

## ■ **Steps in mathematical analysis of nonrecursive algorithms:**

- ✦ *Decide on parameter  $n$  indicating **input size***
- ✦ *Identify algorithm's **basic operation***
- ✦ *Check whether the number of times the basic operation is executed depends only on the input size  $n$ . If it also depends on the type of input, investigate **worst, average, and best** case efficiency separately.*
- ✦ *Set up **summation** for  $C(n)$  reflecting the number of times the algorithm's basic operation is executed.*
- ✦ *Simplify summation to find a closed-form formula or, at the very least, find its order of growth, using standard formulas (see Appendix A)*

# *Time Efficiency of Non-recursive Algorithms*

---

- ✦ *useful basic rules and standard formulas for sum manipulation*

$$\sum_{i=l}^u (a^i \pm b^i) = \sum_{i=l}^u a^i \pm \sum_{i=l}^u b^i$$

$$\sum_{i=l}^u 1 = u - l + 1$$

$$\sum_{i=0}^n i = \sum_{i=1}^n i = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2)$$



# Time Efficiency of Non-recursive Algorithms

## ■ Example 1: Maximum element

**ALGORITHM** *MaxElement*( $A[0..n - 1]$ )

//Determines the value of the largest element in a given array

//Input: An array  $A[0..n - 1]$  of real numbers

//Output: The value of the largest element in  $A$

$maxval \leftarrow A[0]$

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

**if**  $A[i] > maxval$

$maxval \leftarrow A[i]$

**return**  $maxval$

- **Basic operation: comparison** (in the for loop, and executed on each repetition)
- **Input size: array length  $n$**
- **time efficiency :**

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 = \Theta(n)$$

# *Time Efficiency of Non-recursive Algorithms*

---

## ■ *Example 2: Element uniqueness problem*

**ALGORITHM** *UniqueElements*( $A[0..n - 1]$ )

//Determines whether all the elements in a given array are distinct

//Input: An array  $A[0..n - 1]$

//Output: Returns “true” if all the elements in  $A$  are distinct

//           and “false” otherwise

**for**  $i \leftarrow 0$  **to**  $n - 2$  **do**

**for**  $j \leftarrow i + 1$  **to**  $n - 1$  **do**

**if**  $A[i] = A[j]$  **return false**

**return true**

- *Basic operation: comparison*
- *Input size: array length  $n$*
- *time complexity*

# *Time Efficiency of Non-recursive Algorithms*

---

- The number of element comparison depends on
  - a) array size  $n$
  - b) whether there are equal elements in the array and, if there are, which array positions they occupy
- Worst case
  - a) arrays with no equal elements
  - b) arrays in which the last two elements are the only pair of equal ones

$$\begin{aligned}C_{\text{worst}}(n) &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-i-1) \\&= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \sum_{i=0}^{n-2} i = (n-1)^2 - \frac{(n-2)(n-1)}{2} \\&= \frac{n(n-1)}{2} \in \Theta(n^2)\end{aligned}$$

# *Time Efficiency of Non-recursive Algorithms*

---

## *Another algorithm for Element uniqueness problem*

- first, sort the array by some *sorting alg.*,  
—— time complexity for quick-sort alg. is  $\Theta(n \log n)$
- then, scan the sorted array to check its consecutive elements for equality  
—— no more than  $(n-1)$  comparisons
- so, the total *time complexity is  $\Theta(n \log n)$*

# Time Efficiency of Non-recursive Algorithms

## ■ Example 3: Matrix multiplication

**ALGORITHM** *MatrixMultiplication*( $A[0..n-1, 0..n-1]$ ,  $B[0..n-1, 0..n-1]$ )  
//Multiplies two  $n$ -by- $n$  matrices by the definition-based algorithm  
//Input: Two  $n$ -by- $n$  matrices  $A$  and  $B$   
//Output: Matrix  $C = AB$   
**for**  $i \leftarrow 0$  **to**  $n - 1$  **do**  
    **for**  $j \leftarrow 0$  **to**  $n - 1$  **do**  
         $C[i, j] \leftarrow 0.0$   
        **for**  $k \leftarrow 0$  **to**  $n - 1$  **do**  
             $C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$   
**return**  $C$

- *Basic operation: multiplication*
- *Input size: matrix order  $n$*

- *The complexity of square matrix multiplication, carried out by definition-based algorithm, is  $O(n^3)$ ,*

to compute  $n^2$  elements of the product matrix,

dot product of  $n$ -element row of matrix  $A$  and  $n$ -element column of matrix  $B$ <sup>37</sup>

$$C(n) = n * n^2$$

# Time Efficiency of Non-recursive Algorithms

---

## ■ Example 4: Counting binary digits

**ALGORITHM** *Binary*( $n$ )

//Input: A positive decimal integer  $n$

//Output: The number of binary digits in  $n$ 's binary representation

$count \leftarrow 1$

**while**  $n > 1$  **do**

$count \leftarrow count + 1$

$n \leftarrow \lfloor n/2 \rfloor$

**return**  $count$

The loop's variable changes in a different manner , it *cannot be investigated the way the previous examples are.*

*about  $\log_2 n$*

# *Mathematical Analysis of Recursive Algorithms*

---

## ■ *Steps in Mathematical Analysis of Recursive Algorithms*

- ✦ *Decide on parameter  $n$  indicating **input size***
- ✦ *Identify algorithm's **basic operation***
- ✦ *Check whether the number of times the basic operation is executed may vary on different inputs of the same size. (If it may, the **worst, average, and best cases** must be investigated separately.)*
- ✦ *Set up a **recurrence relation** and **initial condition(s)** for  $C(n)$ -the number of times the basic operation is executed for an input of size  $n$  (alternatively count recursive calls).*
- ✦ *Solve the recurrence or estimate the order of growth of the solution by backward substitutions or some other method*

# Mathematical Analysis of Recursive Algorithms

---

## ■ Example 1: Recursive evaluation of $n!$

### ✦ Definition

#### ■ Iterative Definition

$$\begin{aligned} F(n) &= 1 && \text{if } n = 0 \\ &= n * (n-1) * (n-2) \dots 3 * 2 * 1 && \text{if } n > 0 \end{aligned}$$

#### ■ Recursive definition

$$n! = \begin{cases} 1 & n = 0 \\ n(n-1)! & n > 0 \end{cases} \quad \begin{aligned} F(n) &= 1 && \text{if } n = 0 \\ F(n) &= n * F(n-1) && \text{if } n > 0 \end{aligned}$$

### ✦ Succinctness vs. efficiency

- ★ Be careful with recursive algorithms because their succinctness mask their inefficiency.



# *Mathematical Analysis of Recursive Algorithms*

---

## ■ *Example 1: Recursive evaluation of $n!$ ('cont)*

**Algorithm**  $F(n)$

if  $n=0$

    return 1

    //base case

else

    return  $F(n-1) * n$

    //general case

# Mathematical Analysis of Recursive Algorithms

## ■ Example 1: Recursive evaluation of $n!$ ('cont)

input size:  $n$

basic operation: multiplication

Times of Basic operation for  $F(n)$

$$C(0) = 0$$

initial condition

$$C(n) = C(n-1) + 1$$

recurrence relation

to find the initial condition, to see  
when the call stop in the  
pseudocode

to solve recurrences,

method of backward substitutions

$$C(n) = C(n-1) + 1$$

$$= [C(n-2) + 1] + 1 = C(n-2) + 2$$

$$= [C(n-3) + 1] + 2 = C(n-3) + 3$$

$$= \dots = C(n-i) + i = \dots$$

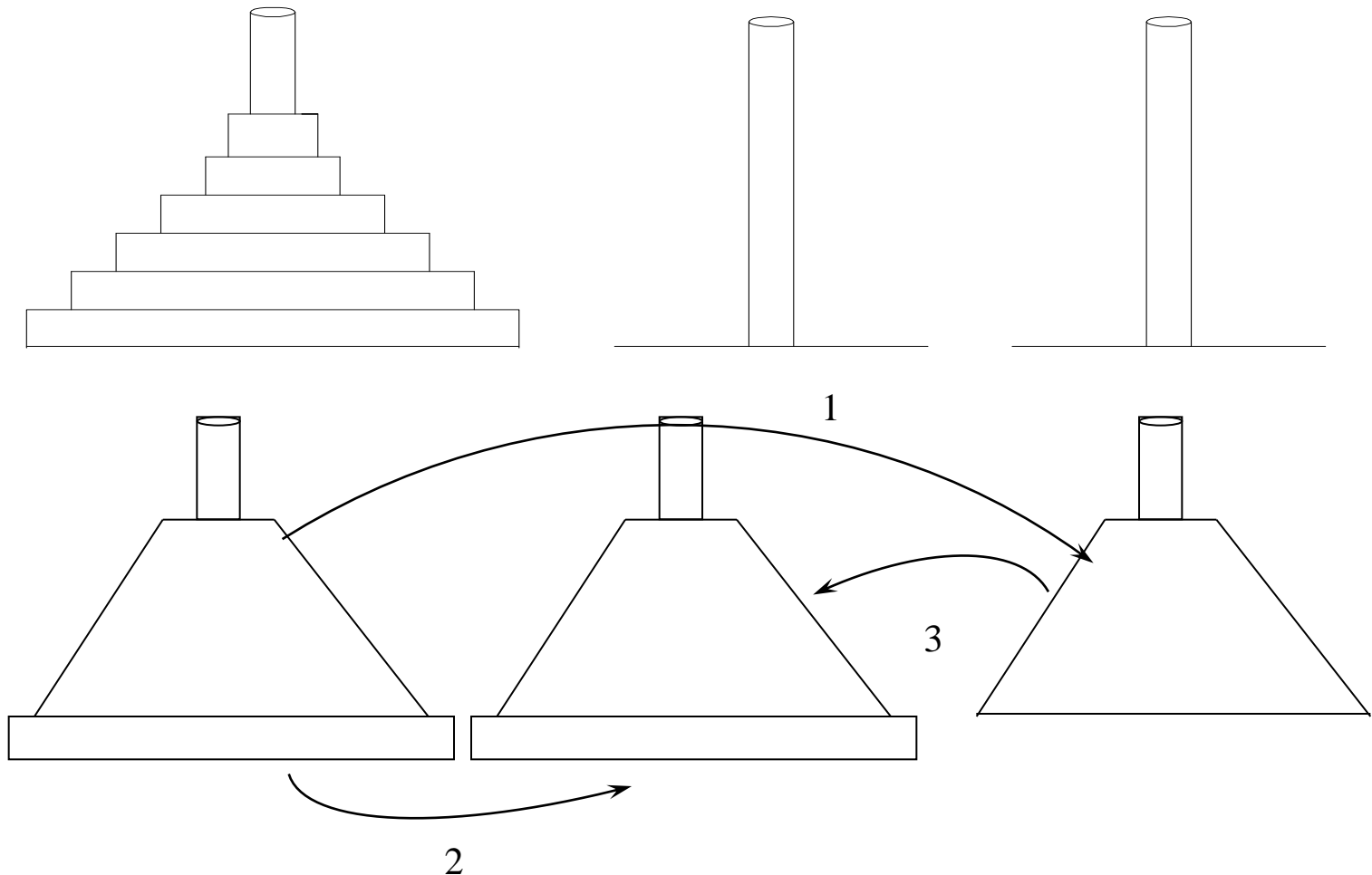
$$= [C(n-n) + 1] + n - 1 = n$$

can be proved by  
mathematical induction

# *Mathematical Analysis of Recursive Algorithms*

---

## ■ *Example 2: The Tower of Hanoi Puzzle*



# *Mathematical Analysis of Recursive Algorithms*

---

## ■ **Example 2: The Tower of Hanoi Puzzle ('cont)**

```
void hanoi(int n, int a, int b, int c)
{
    if (n > 0)
    {
        hanoi(n-1, a, c, b); // n-1个较小圆盘从塔座a移到c
        move(a,b);
        hanoi(n-1, c, b, a);
    }
}
```

# Mathematical Analysis of Recursive Algorithms

---

## ■ Example 2: The Tower of Hanoi Puzzle ('cont)

### ✦ Recurrence Relations

*input size: the number of disks,  $n$*

*basic operation: moving one disk*

*total number of moving :  $C(n)$*

$$C(1) = 1$$

$$C(n) = 2C(n-1) + 1 = 2^n - 1 \quad \text{for every } n > 1$$

$$C(n) \in \Theta(2^n)$$

$$C(n) = 2C(n-1) + 1$$

$$= 2(2C(n-2)+1)+1 = 2^2C(n-2)+2+1=...$$

$$= 2^iC(n-i)+2^{i-1}+2^{i-2}+...+2+1=...$$

$$= 2^{n-1}C(1)+2^{n-2}+2^{n-3}+...+2+1$$

$$= 2^{n-1}+2^{n-2}+2^{n-3}+...+2+1 \quad \text{等比数列}$$

$$= (1-q^n)/(1-q) = (2^n-1)/(2-1) = 2^n-1$$

# *Mathematical Analysis of Recursive Algorithms*

---

- **Example 3: Find the number of binary digits in the binary representation of a positive decimal integer**

**ALGORITHM** *BinRec*( $n$ )

//Input: A positive decimal integer  $n$

//Output: The number of binary digits in  $n$ 's binary representation

**if**  $n = 1$  **return** 1

**else return** *BinRec*( $\lfloor n/2 \rfloor$ ) + 1

*number of additions: in computing BinRec*

$$A(n) = A(\lfloor n/2 \rfloor) + 1 \text{ with } A(1) = 0$$

# Mathematical Analysis of Recursive Algorithms

---

## ■ ■ Example 3: ('cont)

*to solve recurrences,*

### Smoothness Rule

let  $T(n)$  be an eventually non-decreasing function and  $f(n)$  be a smooth function. If

$T(n) \in \Theta(f(n))$  for values of  $n$  that are powers of  $b$ ,

where  $b \geq 2$ , then

$T(n) \in \Theta(f(n))$  for any  $n$ .

*under very broad assumptions, the order of growth observed for  $n=2^k$ , gives a correct answer about the order of growth for all values of  $n$ .*

# *Mathematical Analysis of Recursive Algorithms*

---

## ■ ■ **Example 3: ('cont)**

for  $n = 2^k$ ,

$$A(2^k) = A(2^{k-1}) + 1 \quad \text{for } k > 0$$

$$A(2^0) = 0$$

backward substitutions:

$$\begin{aligned} A(2^k) &= A(2^{k-1}) + 1 \\ &= [A(2^{k-2}) + 1] + 1 = A(2^{k-2}) + 2 \\ &= [A(2^{k-3}) + 1] + 2 = A(2^{k-3}) + 3 \quad \dots\dots \\ &= A(2^{k-i}) + i \quad \dots\dots \\ &= A(2^{k-k}) + k \\ &= A(2^0) + k = A(1) + k = k \end{aligned}$$

then,  $A(n) = \log_2 n = \Theta(\log n)$



# Mathematical Analysis of Recursive Algorithms

---

For example 3 BinRec

$$A(n) = A(\lfloor n/2 \rfloor) + 1 \text{ with } A(1) = 0 \rightarrow A(n) \in \Theta(\log n)$$

In fact, we can prove  $A(n) = \lfloor \log n \rfloor$  is the solution to above recurrence.

Let  $n$  be even, i.e.,  $n = 2k$ .

The left-hand side is:

$$A(n) = \lfloor \log_2 n \rfloor = \lfloor \log_2 2k \rfloor = \lfloor \log_2 2 + \log_2 k \rfloor = (1 + \lfloor \log_2 k \rfloor) = \lfloor \log_2 k \rfloor + 1.$$

The right-hand side is:

$$A(\lfloor n/2 \rfloor) + 1 = A(\lfloor 2k/2 \rfloor) + 1 = A(k) + 1 = \lfloor \log_2 k \rfloor + 1.$$

Let  $n$  be odd, i.e.,  $n = 2k + 1$ .

The left-hand side is:

$$\begin{aligned} A(n) &= \lfloor \log_2 n \rfloor = \lfloor \log_2(2k + 1) \rfloor = \text{using } \lfloor \log_2 x \rfloor = \lceil \log_2(x + 1) \rceil - 1 \\ &= \lceil \log_2(2k + 2) \rceil - 1 = \lceil \log_2 2(k + 1) \rceil - 1 \\ &= \lceil \log_2 2 + \log_2(k + 1) \rceil - 1 = 1 + \lceil \log_2(k + 1) \rceil - 1 = \lfloor \log_2 k \rfloor + 1. \end{aligned}$$

The right-hand side is:

$$A(\lfloor n/2 \rfloor) + 1 = A(\lfloor (2k + 1)/2 \rfloor) + 1 = A(\lfloor k + 1/2 \rfloor) + 1 = A(k) + 1 = \lfloor \log_2 k \rfloor + 1.$$

The initial condition is verified immediately:  $A(1) = \lfloor \log_2 1 \rfloor = 0$ .

# *Mathematical Analysis of Recursive Algorithms*

---

## ■ *Fibonacci numbers*

✦ *The Fibonacci numbers:*

*0, 1, 1, 2, 3, 5, 8, 13, 21, ...*

✦ *The Fibonacci recurrence:*

*The  $n$ th Fibonacci number :*

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

$$F(0) = 0$$

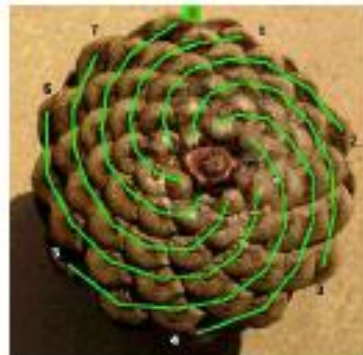
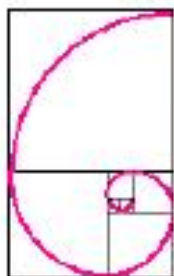
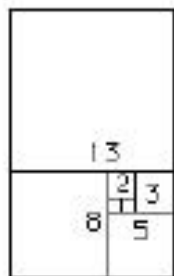
$$F(1) = 1$$



# Mathematical Analysis of Recursive Algorithms

## ■ applications

斐波那契螺旋：使所有种子具有差不多的大小却又疏密得当，不至于在圆心处挤了太多的种子而在圆周处却又稀稀拉拉。  
叶子的生长方式也是如此。



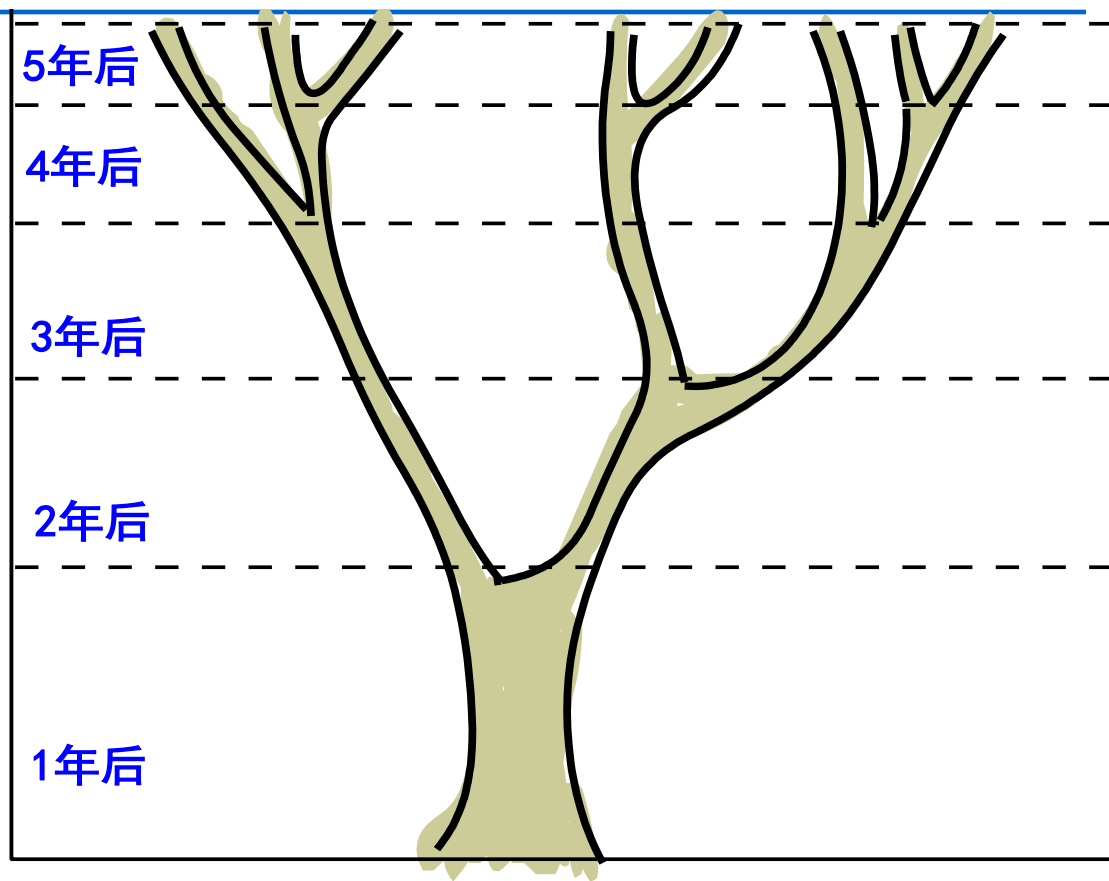
# Mathematical Analysis of Recursive Algorithms

## 兔子问题



# Mathematical Analysis of Recursive Algorithms



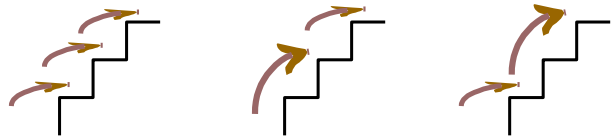
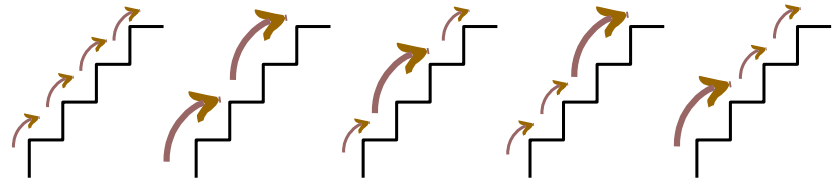
## 树枝生长问题



一株树苗在一段间隔，例如一年，以后长出一条新枝；第二年新枝“休息”，老枝依旧萌发；此后，老枝与“休息”过一年的枝同时萌发，当年生的新枝则次年“休息”。这样，一株树木各个年份的枝桠数，便构成斐波那契数列。这个规律，就是生物学上著名的“鲁德维格定律”。

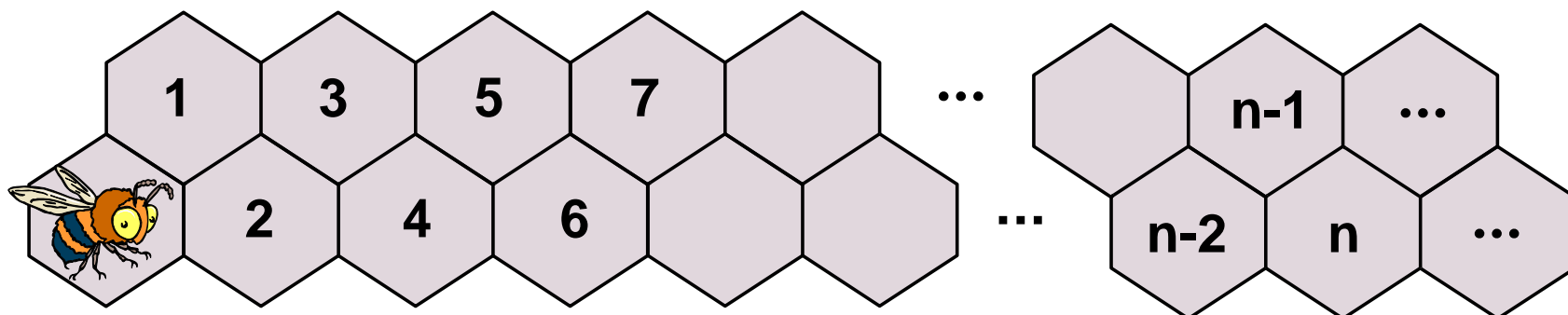
# Mathematical Analysis of Recursive Algorithms

上楼梯问题：楼梯时，若允许每次跨一级或两级，那么对于楼梯数为1, 2, 3, 4时上楼的方式数各是多少

楼梯级数	上楼方式	方式数
1		1
2		2
3		3
4		5
...	...    ...    ...    ...    ...	...

# Mathematical Analysis of Recursive Algorithms

蜜蜂进蜂房问题：一次蜜蜂从蜂房A出发，想爬到1、2、……、 $n$ 号蜂房，只允许它自左向右（不许反方向倒走）。则它爬到各号蜂房的路线多少？



蜜蜂爬进 $n$ 号蜂房有两种途径：

不经过 $n-1$ 号，直接从 $n-2$ 号进入 $n$ 号蜂房，这种路线有 $u_{n-2}$ 种

经过 $n-1$ 号，进入 $n$ 号蜂房，这种路线有 $u_{n-1}$ 种，

故： $u_n = u_{n-1} + u_{n-2}$

# Summary

---

- 算法的效率包括时间效率与空间效率
- 时间效率用输入规模的函数来度量，该函数的计算主要关注算法基本操作的执行次数
- 增长阶数 Order of growth
- Worst-Case, Best-Case, and Average-Case Efficiency
- $O$ -notation,  $\Theta$ -notation,  $\Omega$ -notation
- 递归与非递归算法的效率分析



# 思考题

---

2-1. Consider the definition-based algorithm for adding two  $n$ -by- $n$  matrices.

- a) What is its basic operation?
- b) How many times is it performed as a function of the matrix order  $n$ ?
- c) How many times is it performed as a function of the total number of elements in the input matrices?

2-2. for each of the following algorithms, indicate i) a natural size metric for its inputs; ii) its basic operation; iii) whether the basic operation count can be different for inputs of the same size.

- a) computing  $a!$
- b) computing the sum of  $n$  numbers

# 思考题

---

2-3. indicate whether the first function of each of the following pairs has a smaller, same, or large order of growth than the second function.

a)  $n(n+1)$  and  $2000n^2$

b)  $(n-1)!$  and  $n!$

c)  $2^{n-1}$  and  $2^n$

2-4. use the informal definitions of  $O$ ,  $\Omega$  and  $\Theta$  to determine whether the following assertions are true or false.

a)  $n(n+1)/2 \in O(n^3)$

b)  $n(n+1)/2 \in \Theta(n^2)$

# 思考题

---

2-5 习题2.3的4 5 6 10