Analysis and Design of Algorithms

Chapter 2: Fundamentals of the Analysis of Algorithm Efficiency

School of Software Engineering © Yaping Zhu

Fundamentals of the Analysis of Algorithm Efficiency

1. ***Algorithm analysis framework (分析框架)***
2. ***Asymptotic notations (渐近符号)***
3. ***Analysis of non-recursive algorithms***
4. ***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?
- We also want to ignore machine dependent factors.
- We are only interested in how fast an algorithm runs on large inputs.

Analysis of Algorithms

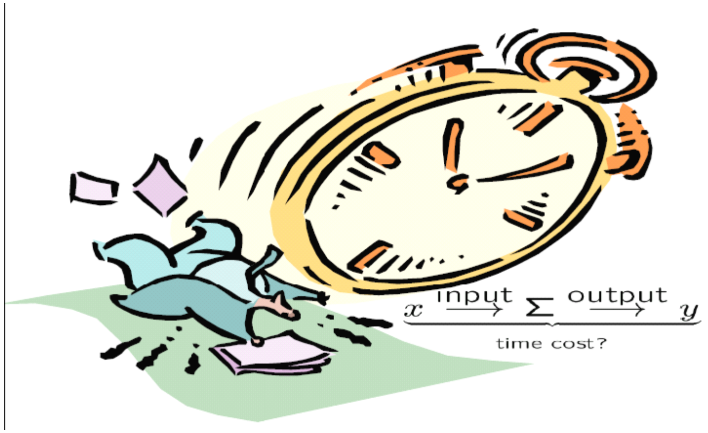
- ***Analysis of algorithms means to investigate an algorithm's efficiency with respect to resources: running time and memory space.***

- 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

① Measuring an input's size

② Units for measuring running time

③ Orders of growth (of the algorithm's efficiency function)

④ Worst-base, best-case and average-case efficiency

Analysis of Algorithms

Measuring an Input Size

- *Efficiency is defined as a function of input size. （Typically, algorithms run longer as the size of its input increases.）*
- *Choose which parameter:*
  - 排序、查找、寻找列表最小元素：列表的长度
  - n次多项式：多项式的次数（or 系数个数）
  - 矩阵：矩阵阶数、矩阵元素的个数
- *Input size depends on the problem.*
  - 拼音检查：独立字符、词
  - 数字：二进制表中的位数
- *We are interested in how efficiency scales with the input size.*

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 implementing the algorithm
  - ➔ Depends on the quality of the compiler used in generating the machine code.
- *Count the number of times each element operation is executed.*
  - ➔ Difficult and unnecessary

Analysis of Algorithms

Units for Measuring Running Time

- **Solution:** *Count the number of times an algorithm's basic operation (the most important) 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.*
    - 排序：键的比较
    - 四则运算：除法、乘法

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.*

n： input size

$$T(n) \approx t_{op} * C(n)$$

running time

execution time for the basic operation

number of times the basic operation is executed

- 如果这个算法运行在一台执行速度是当前机器10倍的机器上，运行速度多快?
- 如果输入规模翻倍，算法运行多长时间?

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

- 小规模输入在运行时间上差别不大，不足以将高效的算法和低效的算法区分开来

  例：欧几里得算法 and 连续整数检测算法

exponential-growth

棒棒哒！

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

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}		

- 对于一台每秒能做一万亿次（10^{12}）操作的计算机：

  完成2^{100}次操作需要$4*10^{10}$年

  （地球的估计年龄：45亿（$4.5*10^9$））

一个需要指数级操作次数的算法

只能用来解决规模非常小的问题。

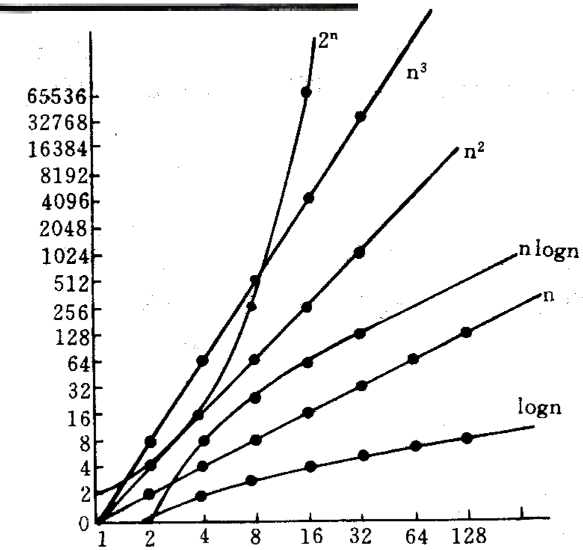


图 1.1 一般计算时间函数的曲线

Analysis of Algorithms

Order of growth

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}		

当参数n的值增长为原来的两倍时：

- $\log_2 n$: 次数增加1
- 线性函数：两倍
- $n \log n$: 略超过两倍
- n^2：4倍； n^3： 8倍
- 2^n： 平方
- $n!$ ： 多得多

Analysis of Algorithms

Worst-Case, Best-Case, and Average-Case Efficiency

- *For some algorithms efficiency depends on specifics 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*).

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

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

- *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.

Attention：

最优情况并不是指规模最小的输入，而是使算法运行得最快的、规模为*n*的输入。

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?**

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

- ***Problem:*** *impossible to calculate e_i for every legal input I*
- ***Solution:*** *to calculate e_i for some representative input*

Analysis of Algorithms

- ***Example: Sequential Search***
  - probability for successful search is p ($0 \le p \le 1$);
  - probability for successful search on each position i ($0 \le i < n$) in an array is equal, p/n。

✓ 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

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

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

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

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

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***
  - ***Example:***

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

- *If* $T(n) \to \infty$, as $n \to \infty$;

  $(T(n) - t(n))/ T(n) \to 0$, as $n \to \infty$;

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

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

- *$t(n)$ 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 $g(n)$*

  $\leq$

- $\Omega(g(n))$: *class of functions $t(n)$ that grow at least as fast as $g(n)$*

  $\geq$

- $\Theta(g(n))$: *class of functions $t(n)$ that grow at same rate as $g(n)$*

  $=$

Asymptotic notations

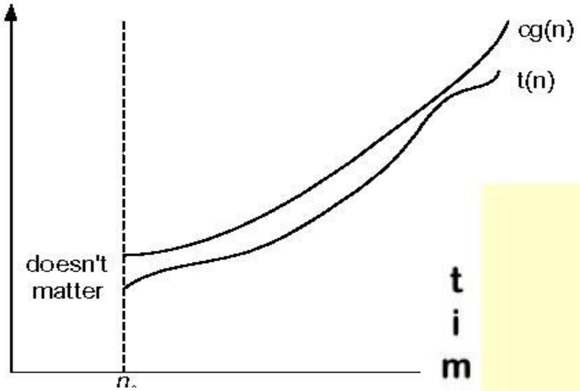
O-notation

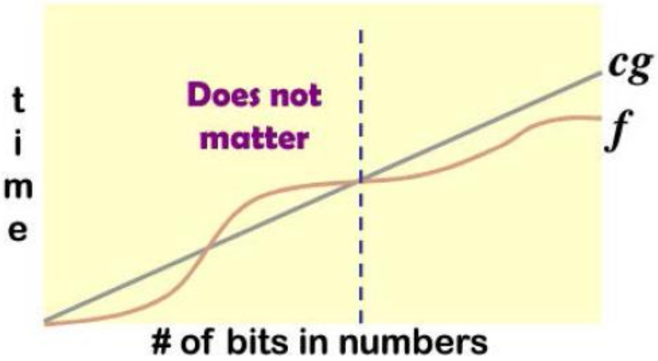
- *Formal definition:*
  - **A function $t(n)$ is said to be in $O(g(n))$, denoted $t(n) \in O(g(n))$, if $t(n)$ is bounded above by some constant multiple of $g(n)$ for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that**

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

- **e.g.:**

$100n + 5 <= 100n + n = 101n <= 101n^2$





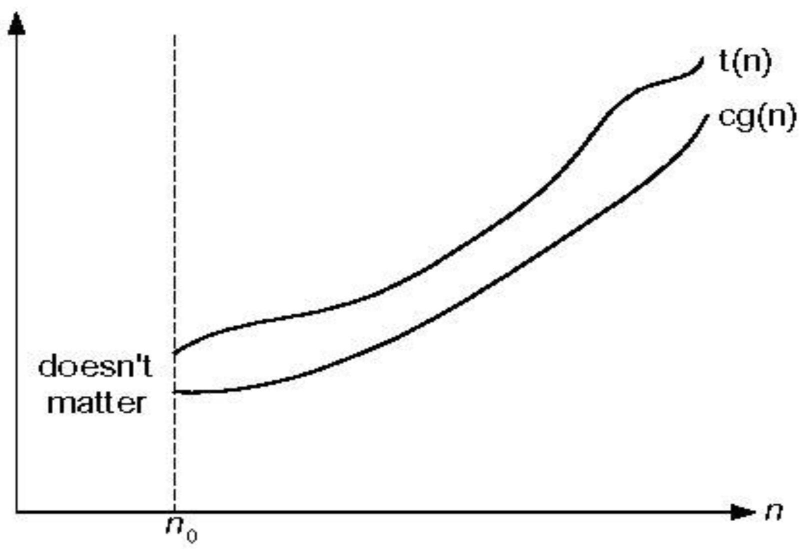
- ***Example:***
  - $10n^2 \in O(n^2)$
  - $10n^2 + 2n \in O(n^2)$
  - $100n + 5 \in O(n^2)$
  - $5n+20 \in O(n)$

Asymptotic notations

Ω-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**

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



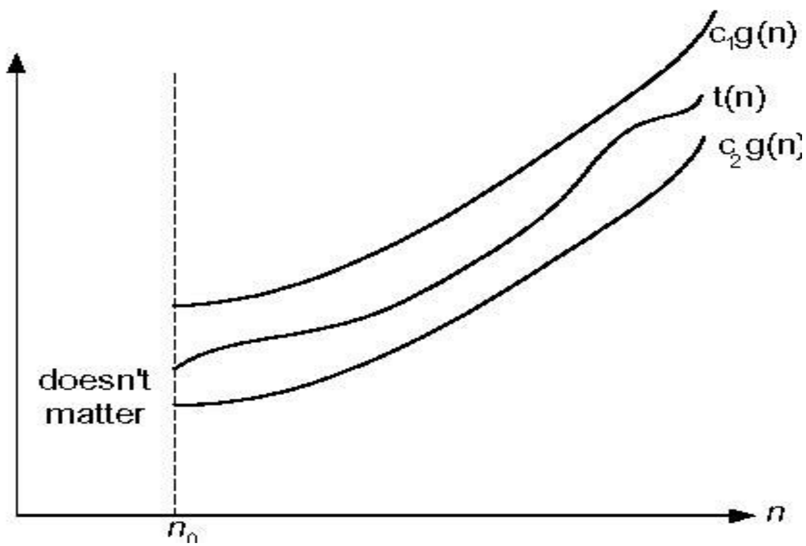
- ***Example:***
  - $10n^2 \in \Omega(n^2)$
  - $10n^2 + 2n \in \Omega(n^2)$
  - $10n^3 \in \Omega(n^2)$

Asymptotic notations

Θ-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**

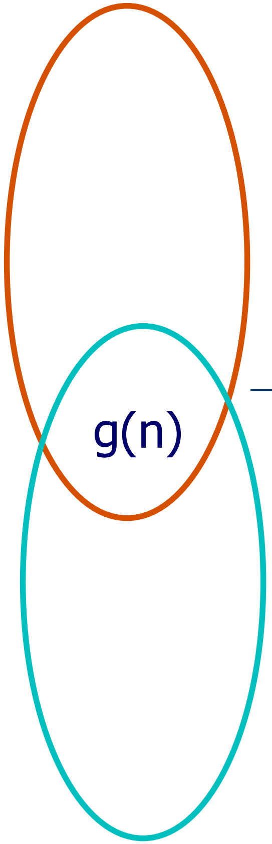
$$c_2 g(n) \le t(n) \le c_1 g(n) \quad \text{for all } n \ge n_0$$



- ***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



>= 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 $g(n)$

<= smaller or same order of growth

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

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**

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

- *ω(g(n)):*
  - **A function $t(n)$ is said to be in ω ($g(n)$), 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**

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

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 Ω-notation and Θ-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

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

- **Example:**
  - $5n^2 + 3n\log n \in O(n^2)$
  - $10n$ vs. $2n^2$
  - $n(n+1)/2$ vs. n^2
  - $\log_b n$ vs. $\log_c n$

case 1 & 2， $T(n) \in O(g(n))$;

case 2， $T(n) \in \Theta(g(n))$;

case 3 & 2， $T(n) \in \Omega(g(n))$;

Asymptotic notations

L'Hôpital's rule

- *If* $\lim_{n\to\infty} f(n) = \lim_{n\to\infty} g(n) = \infty$ **and the derivatives *f ´*, *g ´* exist**,
  Then,

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

- ***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. $n!$

$$\lim_{n\to\infty} \frac{\log_2 n}{\sqrt{n}} = \lim \frac{(\log_2 n)'}{(n^{\frac{1}{2}})'} = \lim \frac{\frac{1}{n \ln 2}}{\frac{1}{2} n^{-\frac{1}{2}}} = \frac{2}{\ln 2} \lim \frac{n^{\frac{1}{2}}}{n} = 0$$

$$\Rightarrow \log_2 n \in O(n^{\frac{1}{2}})$$

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: $a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0 \in \Theta(n^k)$.

- Exponential functions a^n have **different orders** of growth for different a's.

- order $\log n$ < order n^α ($\alpha > 0$) < order a^n < order $n!$ < order n^n

Asymptotic notations

- ***Summary of How to Establish Orders of Growth of an Algorithm's Basic Operation Count***

1. *Method 1: Using limits*
   - ***L'Hôpital's rule***

2. *Method 2: Using the properties*

3. *Method 3: Using the definitions of O, Ω, and Θ notation.*

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

Analysis of Algorithms

Basic Efficiency classes

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

fast

High time efficiency

1	constant (常数)
log *n*	logarithmic (对数)
n	linear (线性)
n log *n*	*n* log *n* (线性对数)
n^2	quadratic (平方)
n^3	cubic (立方)
2^n	exponential (指数)
n!	factorial (阶乘)

slow

low time efficiency

Analysis of Algorithms

表 2.2　基本的渐近效率类型

类　型	名　称	注　　释
1	常量	为数很少的效率最高的算法，很难举出几个合适的例子，因为典型情况下，当输入的规模变得无穷大时，算法的运行时间也会趋向于无穷大
$\log n$	对数	一般来说，算法的每一次循环都会消去问题规模的一个常数因子(参见 4.4 节)。注意，一个对数算法不可能关注它的输入的每一个部分(哪怕是输入的一个固定部分)：任何能做到这一点的算法最起码拥有线性运行时间
n	线性	扫描规模为 n 的列表(例如，顺序查找)的算法属于这个类型
$n \log n$	线性对数	许多分治算法(参见第 5 章)，包括合并排序和快速排序的平均效率，都属于这个类型
n^2	平方	一般来说，这是包含两重嵌套循环的算法的典型效率(参见下一节)。基本排序算法和 n 阶方阵的某些特定操作都是标准的例子
n^3	立方	一般来说，这是包含三重嵌套循环的算法的典型效率(参见下一节)。线性代数中的一些著名的算法属于这一类型
2^n	指数	求 n 个元素集合的所有子集的算法是这种类型的典型例子。“指数”这个术语常常被用在一个更广的层面上，不仅包括这种类型，还包括那些增长速度更快的类型
$n!$	阶乘	求 n 个元素集合的完全排列的算法是这种类型的典型例子

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 $maxval$

- **Input size:** ***array length n***

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:** 循环体中存在两种操作：比较运算：$A[i] > maxval$
  赋值运算：$maxval \leftarrow A[i]$
- 每做一次循环都会进行一次比较，而赋值运算不一定。

comparison (in the for loop, and executed on each repetition)

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 $maxval$

- **Check:** 对于所有大小为 ***n*** 的数组，比较次数都是相同的。
- 选用比较次数度量的时候，无需区分最差、平均和最优情况。

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 $maxval$

- **Time efficiency** :基本操作的执行次数求和

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

Time Efficiency of Non-recursive Algorithms

Steps in mathematical analysis of nonrecursive algorithms:

1. Decide on parameter(s) n indicating input size.
2. Identify algorithm's basic operation.
3. 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.
4. Set up **summation** for $C(n)$ reflecting the number of times the algorithm's basic operation is executed.
5. Simplify summation to find a closed-form formula or, at the very least, find its order of growth, using standard formulas.

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 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 ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] = A[j] return false
return true
```

- **Input size:** *array length n*
- **Basic operation:** *comparison*
- **Check:** 是否需要考虑最差、平均和最优情况？

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

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1)-(i+1)+1] = \sum_{i=0}^{n-2} (n-i-1)$$

$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1)\sum_{i=0}^{n-2} 1 - \sum_{i=0}^{n-2} i = (n-1)^2 - \frac{(n-2)(n-1)}{2}$$

$$= \frac{n(n-1)}{2} \in \Theta(n^2)$$

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

- **Worst case:** 最内层循环每执行一次就会进行一次比较，对于循环变量 j 在 $i+1$ 和 $n-1$ 之间的每个值都会做一次循环；对于外层循环变量 i 在 0 和 $n-2$ 之间的每个值，上述过程都会再重复一遍。

Time Efficiency of Non-recursive Algorithms

- **Worst case：**

$$\sum_{i=0}^{n-2}(n-1-i) = (n-1) + (n-2) + \cdots + 1 = \frac{(n-1)n}{2}$$

Time efficiency:

$$C_{worst}(n) = \sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2}[(n-1)-(i+1)+1] = \sum_{i=0}^{n-2}(n-i-1)$$

$$= \sum_{i=0}^{n-2}(n-1) - \sum_{i=0}^{n-2} i = (n-1)\sum_{i=0}^{n-2} 1 - \sum_{i=0}^{n-2} i = (n-1)^2 - \frac{(n-2)(n-1)}{2}$$

$$= \frac{n(n-1)}{2} \in \Theta(n^2)$$

在最坏的情况下，对于 n 个元素的所有两两组合（共 $n(n-1)/2$ 对），都需要比较一次。

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 **Θ (*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 Θ (*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

- **Input size:** ***matrix order n*** （思考如果是一个**m*n**的矩阵与一个**n*k**的矩阵相乘，如何选取输入规模？）
- **Basic operation:** ***multiplication*** （最内层循环）
- **Check:** *？*

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

- **Time efficiency:**

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3.$$

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

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 ← 0 to n − 1 do
    for j ← 0 to n − 1 do
        C[i, j] ← 0.0
        for k ← 0 to n − 1 do
            C[i, j] ← C[i, j] + A[i, k] ∗ B[k, j]
return C
```

From another view:

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.

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

Time Efficiency of Non-recursive Algorithms

障碍：

- 循环变量的无规律变化
- 过于复杂而无法求解的求和表达式
- 分析平均情况时固有的难度

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.*

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 ← 1
    while n > 1 do
        count ← count + 1
        n ← ⌊n/2⌋
    return count
```

- 本算法中最频繁的操作不在**while**循环内部，而是决定是否继续执行循环体的比较运算 $n > 1$。
- 因为在循环的每次执行过程中，n 的值基本上都会减半，所以该次数大约是 $\log_2 n$.
- 比较运算 $n > 1$的精确计算公式实际上为 $\lfloor \log_2 n \rfloor + 1$ 。

Mathematical Analysis of Recursive Algorithms

Example 1: Recursive evaluation of n !

- Definition
  - **Recursive definition**

$$n! = \begin{cases} 1 & n = 0 \\ n(n-1)! & n > 0 \end{cases}$$

F(n) = 1 if n = 0
F(n) = n * F(n-1) if n > 0

- 伪代码

```
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)

```
Algorithm F(n)
 if n=0
     return 1              //base case
 else
     return F(n -1) * n           //general case
```

- **Input size:** ***n*** (简单起见，未使用二进制表示的位数)
- **Basic operation:** ***multiplication***
- **Check:** ***?***

Mathematical Analysis of Recursive Algorithms

Example 1: Recursive evaluation of n ! (‘cont)

Times of Basic operation for M(n)

$$M(n) = \underbrace{M(n-1)}_{\substack{\text{to compute}\\ F(n-1)}} + \underbrace{1}_{\substack{\text{to multiply}\\ F(n-1) \text{ by } n}} \quad \text{for } n > 0.$$

$\boxed{F(n) = n * F(n\text{-}1) \qquad \text{if } n > 0}$

Difference from non-recursive:

- 没有把 *M*(*n*) 直接定义成 *n* 的函数，而是定义成在另一点上（*n*-1）的值的函数.

递推关系/递推式

Mathematical Analysis of Recursive Algorithms

Example 1: Recursive evaluation of n ! (‘cont)

Solve $M(n) = M(n-1) + 1$

无数个序列，需要一个初始条件

To find the initial condition, see when the call stop in the pseudocode：

$$\textbf{if } n = 0 \textbf{ return } 1.$$

- 调用在 n=0 时结束，所以算法能够处理的 n 的最小值和 $M(n)$ 定义域上的最小值为0.
- 当 n=0 时，该算法不执行乘法操作.

$$M(0) = 0.$$

the calls stop when $n = 0$ | no multiplications when $n = 0$

Mathematical Analysis of Recursive Algorithms

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

算法中乘法次数 $M(n)$ 的递推关系和初始条件:

$$M(n) = M(n-1) + 1 \quad \text{for } n > 0, \qquad \text{recurrence relation}$$
$$M(0) = 0. \qquad \text{initial condition}$$

To solve recurrences, method of ***backward substitutions***（反向替换法）

$$\begin{aligned} M(n) &= M(n-1)+1 \\ &= [M(n-2)+1]+1 = M(n-2)+2 \\ &= [M(n-3)+1]+2 = M(n-3)+3 \\ &= \ldots\ldots \boxed{= M(n-i) + i} = \ldots\ldots \\ &= [M(n-n)+1]+n-1 = n \end{aligned}$$

数学归纳法证明

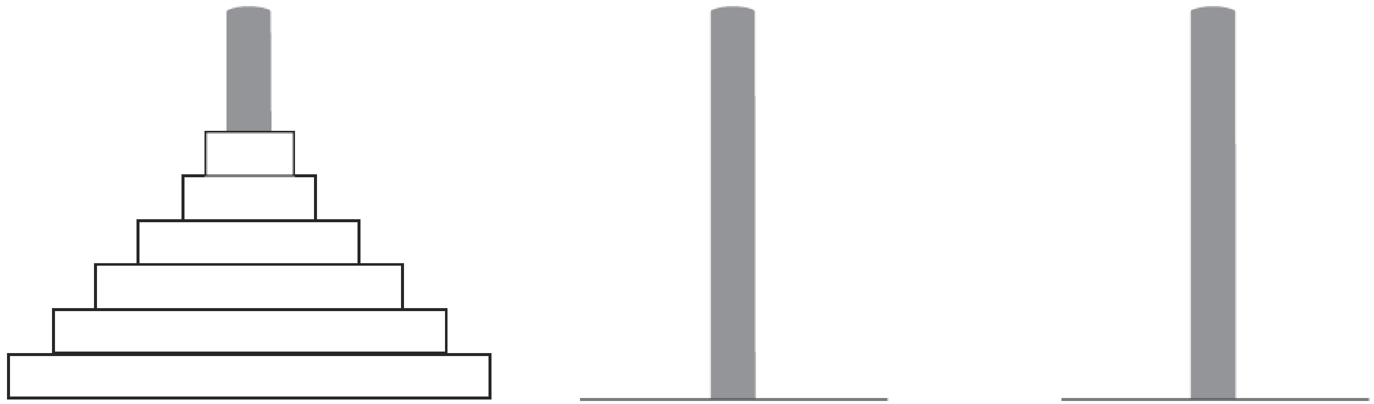
Mathematical Analysis of Recursive Algorithms

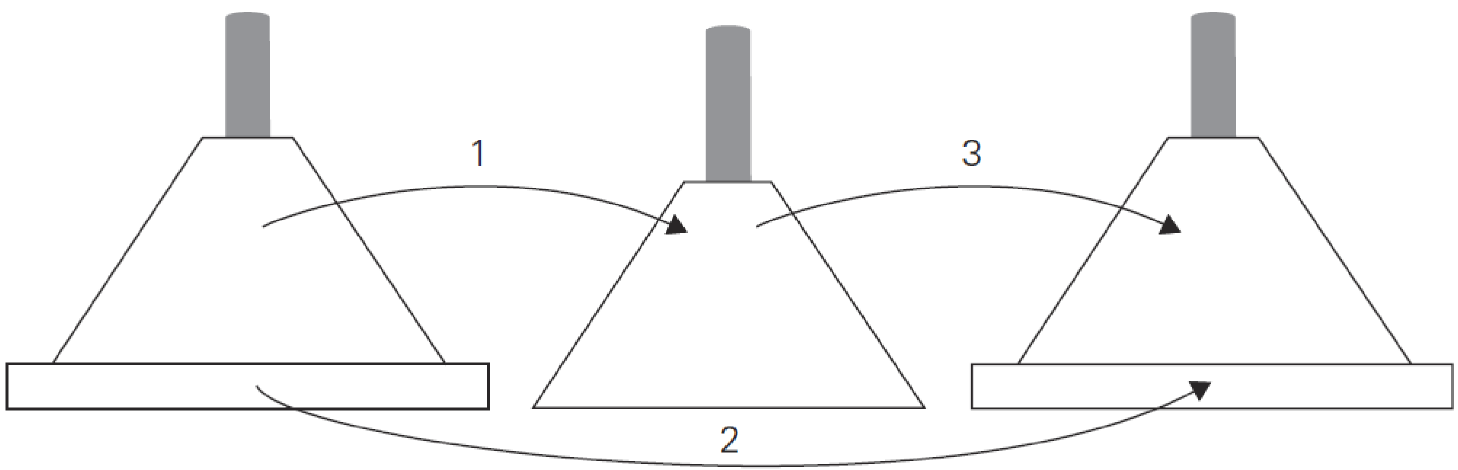
Steps in Mathematical Analysis of Recursive Algorithms

1. Decide on parameter *n* indicating input size.
2. Identify algorithm's basic operation.
3. 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.)
4. 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).
5. 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 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)  // n个盘子，从a移到b借助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);
      }
  }
```

- **Input size:** ***the number of disks, n***
- **Basic operation:** ***moving one disk***
- **Check:** ***?***

Mathematical Analysis of Recursive Algorithms

Example 2: The Tower of Hanoi Puzzle (‘cont)

- *Recurrence Relations*

Total number of moving : ***M*(*n*)**

$$M(n) = M(n-1) + 1 + M(n-1) \quad \text{for } n > 1.$$

```
void hanoi(int n, int a, int b, int c)  // n个盘子，从a移到b借助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);
     }
  }
```

初始条件： $M(1) = 1,$

Mathematical Analysis of Recursive Algorithms

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

- *Recurrence Relations*

Total number of moving : *M*(*n*)

$$M(n) = 2M(n-1) + 1 \quad \text{for } n > 1,$$
$$M(1) = 1.$$

***backward substitutions*（反向替换法）：**

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

$= 2(2M(n-2)+1)+1 = 2^2M(n-2)+2+1=\ldots$

$= 2^iM(n-i)+2^{i-1}+2^{i-2} +\ldots+2+1 =\ldots$

$= 2^{n-1}M(1)+2^{n-2}+2^{n-3}+\ldots+2+1$

$= 2^{n-1}+2^{n-2}+2^{n-3}+\ldots+2+1$ 等比数列

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

Mathematical Analysis of Recursive Algorithms

- **Example 2: The Tower of Hanoi Puzzle (‘cont)**

  - *Succinctness (简洁性) vs. efficiency*
    - *Be careful with recursive algorithms because their succinctness mask their inefficiency.*

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

- **Input size:** ***n***
- **Basic operation:** ***addition***
- **Check:** *?*

计算 ***BinRec*** **(⌊*n*/2⌋)的加法次数为*A*(⌊*n*/2⌋)，**

Number of additions *in computing BinRec* (*n*):

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

Mathematical Analysis of Recursive Algorithms

Example 3: (‘cont)

Smoothness Rule:

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

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

where $b>=2$, then

$$T(n) \in \Theta\,(f(n)) \text{ for any } n.$$

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)***

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

$A(1) = 0$

for $n = 2^k$,

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

$A(2^0) = 0$

backward substitutions:

$A(2^k) = A(2^{k-1}) + 1$

$= [A(2^{k-2}) + 1] + 1 = A(2^{k-2}) + 2$

$= [A(2^{k-3}) + 1] + 2 = A(2^{k-3}) + 3$ ......

$= A(2^{k-i}) + i$ ......

$= A(2^{k-k}) + k$

$= A(2^0) + k = A(1) + k = k$

$A(2^n) = n$, then, $A(n) = \log_2 n = \Theta(\log n)$

Mathematical Analysis of Recursive Algorithms

For example 3 BinRec

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

In fact, we can prove $A(n) = \lfloor \log_2 n \rfloor$ 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:
$A(n) = \lfloor \log_2 n \rfloor = \lfloor \log_2 (2k+1) \rfloor =$ using $\lfloor \log_2 x \rfloor = \lceil \log_2 (x+1) \rceil - 1$
$\lceil \log_2 (2k+2) \rceil - 1 = \lceil \log_2 2(k+1) \rceil - 1$
$= \lceil \log_2 2 + \log_2 (k+1) \rceil - 1 = 1 + \lceil \log_2 (k+1) \rceil - 1 = \lfloor \log_2 k \rfloor + 1$.
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 :

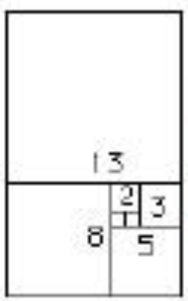
    $$F(n) = F(n-1) + F(n-2)$$

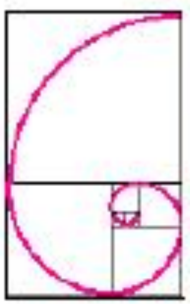
    $F(0) = 0, \quad F(1) = 1$ (两个初始条件)

Mathematical Analysis of Recursive Algorithms

applications

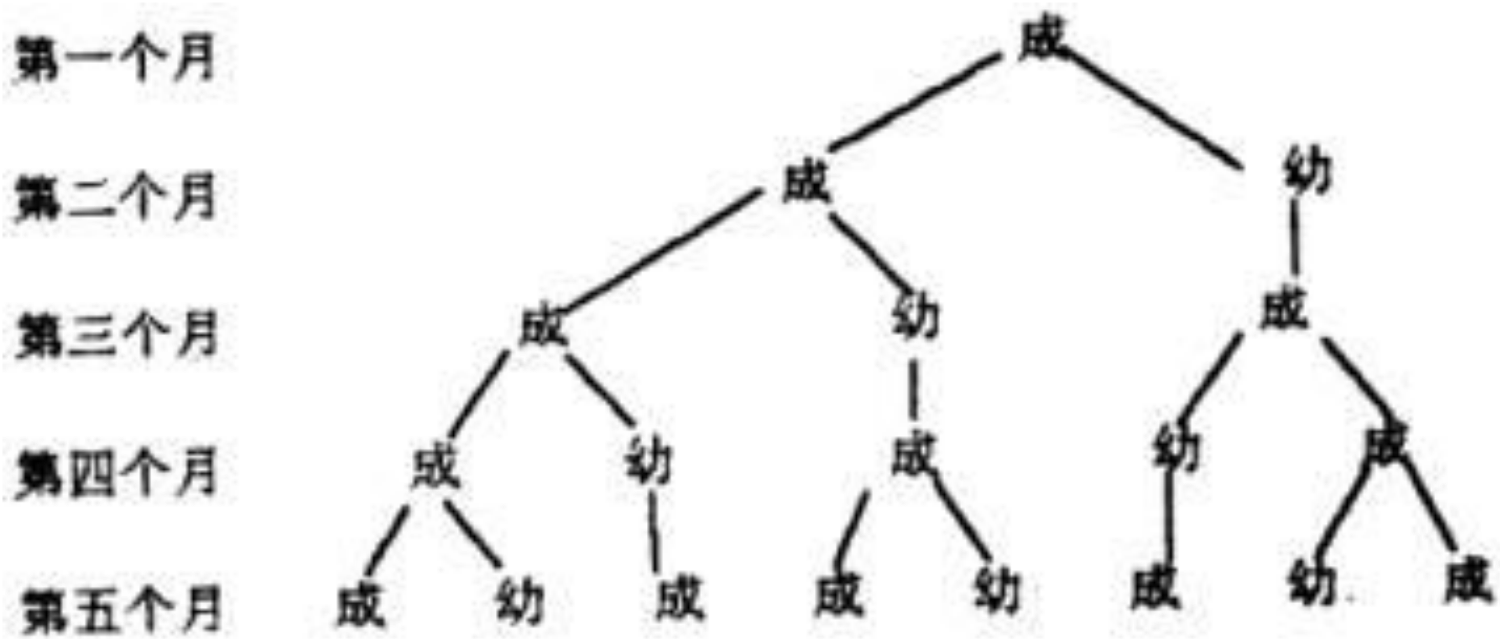
斐波那契螺旋：使所有种子具有差不多的大小却又疏密得当，不至于在圆心处挤了太多的种子而在圆周处却又稀稀拉拉。
叶子的生长方式也是如此。





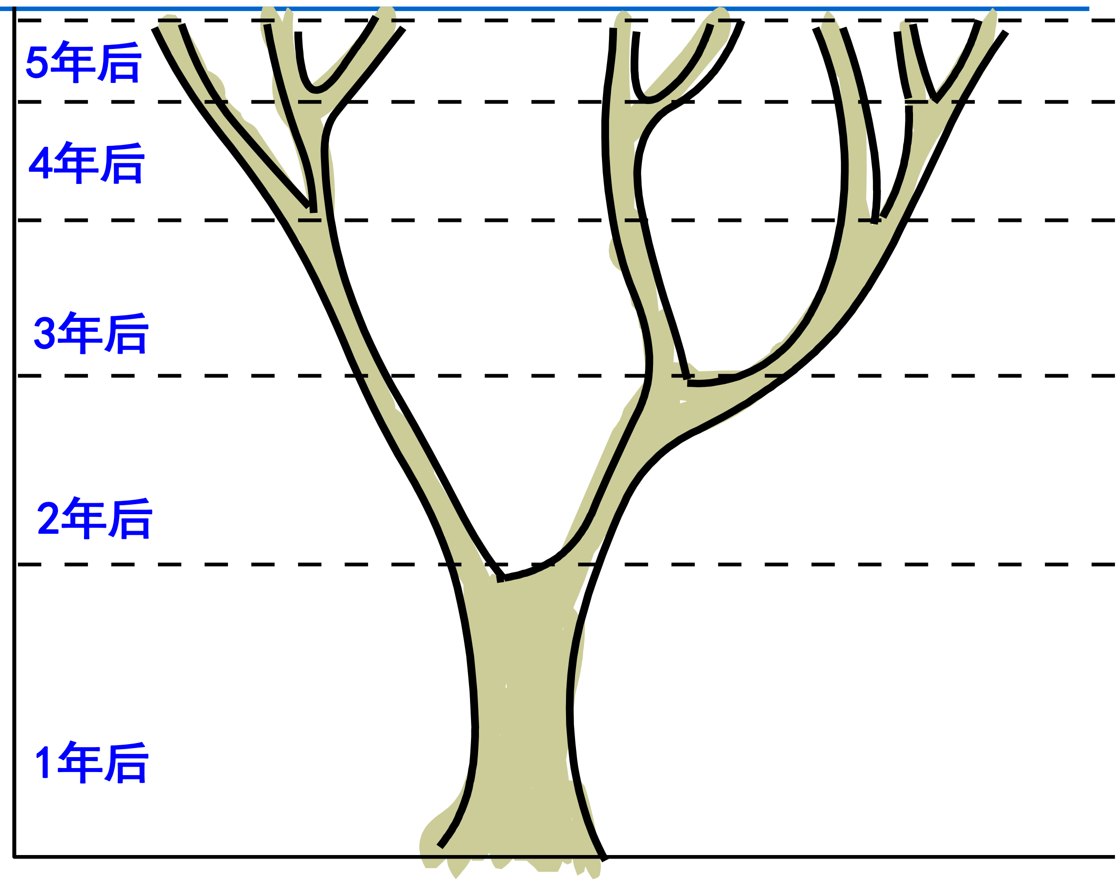
Mathematical Analysis of Recursive Algorithms

兔子问题



Mathematical Analysis of Recursive Algorithms

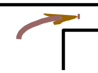
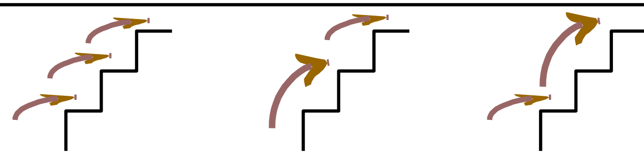
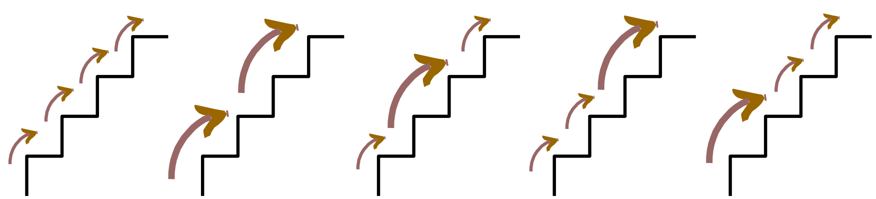
树枝生长问题



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”。

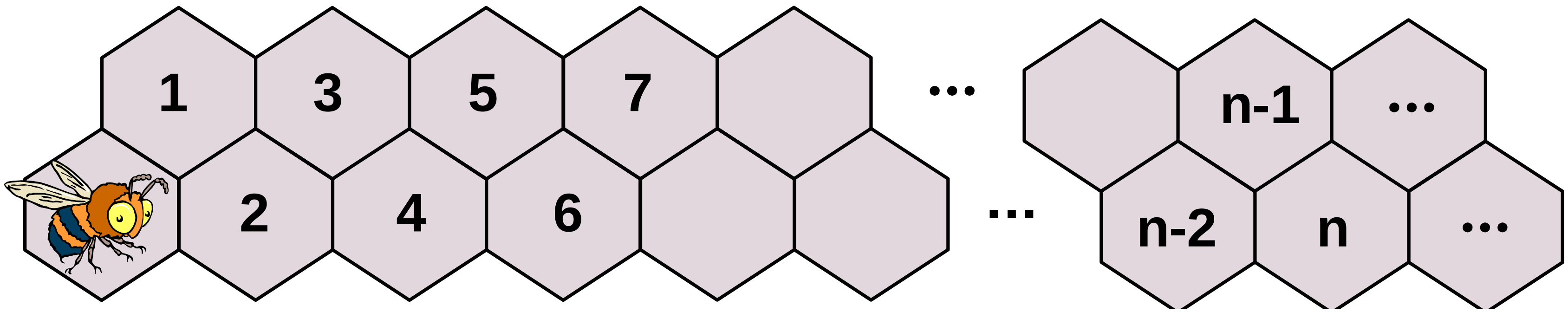
Mathematical Analysis of Recursive Algorithms

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： $u_n = u_{n-1} + u_{n-2}$。

Mathematical Analysis of Recursive Algorithms

Fibonacci numbers

- *The Fibonacci recurrence:*

  The nth Fibonacci number :

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

  $$F(0) = 0, \quad F(1) = 1$$ **(两个初始条件)**

ALGORITHM $F(n)$

//Computes the nth Fibonacci number recursively by using its definition
//Input: A nonnegative integer n
//Output: The nth Fibonacci number
if $n \leq 1$ **return** n
else return $F(n-1) + F(n-2)$

Mathematical Analysis of Recursive Algorithms

Fibonacci numbers

ALGORITHM $F(n)$

//Computes the nth Fibonacci number recursively by using its definition
//Input: A nonnegative integer n
//Output: The nth Fibonacci number
if $n \le 1$ **return** n
else return $F(n-1) + F(n-2)$

- **Input size:** ***n***
- **Basic operation:** ***addition***
- **Check:** *?*
- **Recurrence Relations:** 设计算F(*n*)的加法次数为A(*n*)

$$A(n) = A(n-1) + A(n-2) + 1 \quad \text{for } n > 1,$$
$$A(0) = 0, \qquad A(1) = 0.$$

Mathematical Analysis of Recursive Algorithms

Fibonacci numbers

Solve

$$A(n) = A(n-1) + A(n-2) + 1 \quad \text{for } n > 1,$$
$$A(0) = 0, \qquad A(1) = 0.$$

将非齐次递推式转化成齐次递推式：

$$[A(n)+1] - [A(n-1)+1] - [A(n-2)+1] = 0$$

令 $B(n) = A(n) + 1$**，则：**

$$B(n) - B(n-1) - B(n-2) = 0,$$
$$B(0) = 1, \qquad B(1) = 1.$$

解带常数系数的齐次二阶线性递推式（附录B）：

$$A(n) = B(n) - 1 = F(n+1) - 1 = \frac{1}{\sqrt{5}}(\phi^{n+1} - \hat{\phi}^{n+1}) - 1.$$

$A(n) \in \Theta(\phi^n)$ ⇦ 指数级 相同的函数值被一遍一遍重复计算

Mathematical Analysis of Recursive Algorithms

Fibonacci numbers

ALGORITHM $Fib(n)$

//Computes the nth Fibonacci number iteratively by using its definition
//Input: A nonnegative integer n
//Output: The nth Fibonacci number
$F[0] \leftarrow 0;\ F[1] \leftarrow 1$
for $i \leftarrow 2$ **to** n **do**
$\quad F[i] \leftarrow F[i-1] + F[i-2]$
return $F[n]$

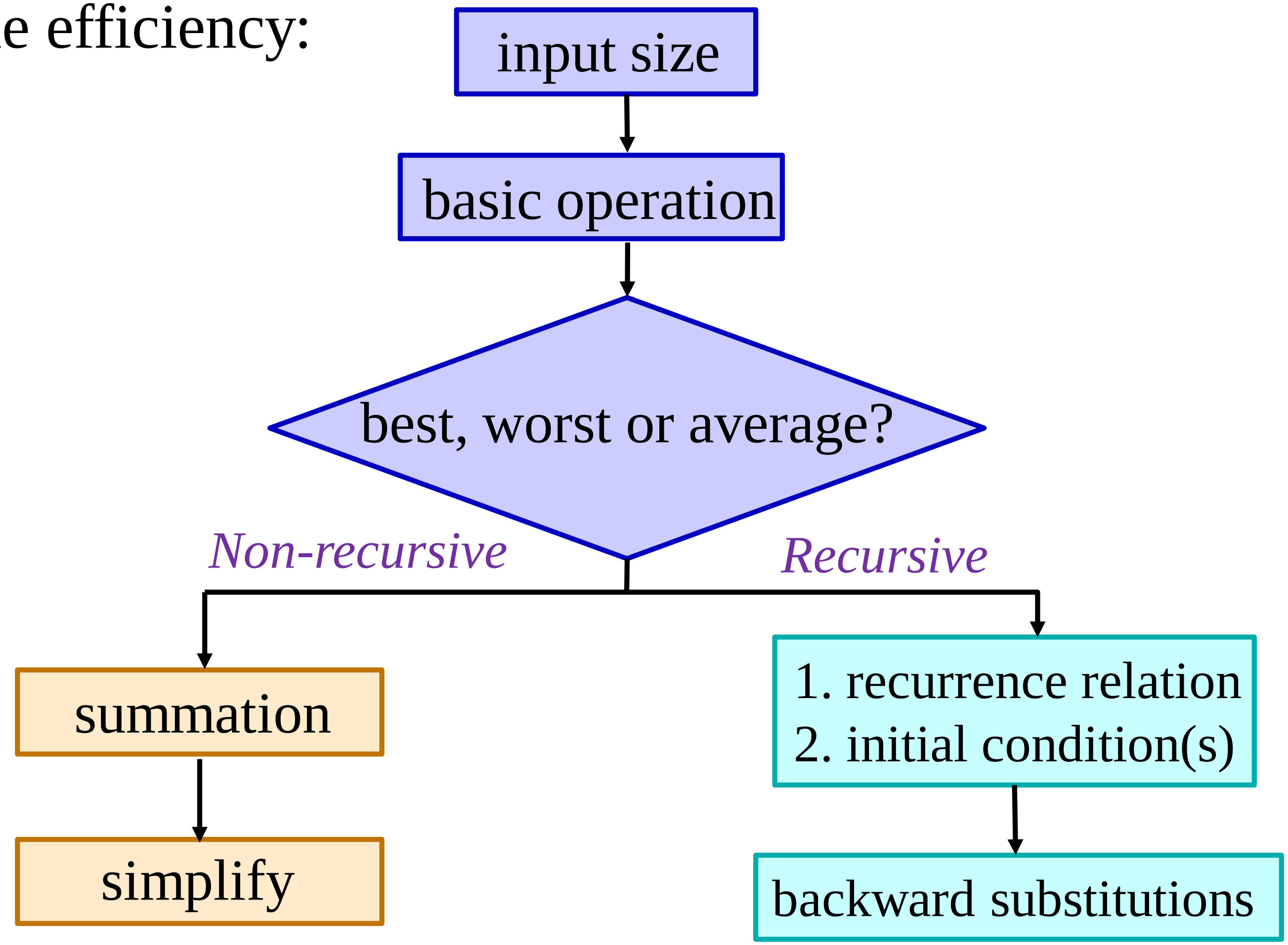
- **Input size:** ***n***
- **Basic operation:** ***addition***
- **Check:** ***?***
- 很明显，这个算法要做 *n*-1 次加法运算：$\Theta(n)$

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, Θ-notation, Ω-notation
- 递归与非递归算法的效率分析

Summary

求解Time efficiency:



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 definitons of O, Ω and Θ 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.