Analysis and Design of Algorithms

Chapter 2: Fundamentals of the Analysis of Algorithm Efficiency

School of Software Engineering © Yanling Xu

Fundamentals of the Analysis of Algorithm Efficiency

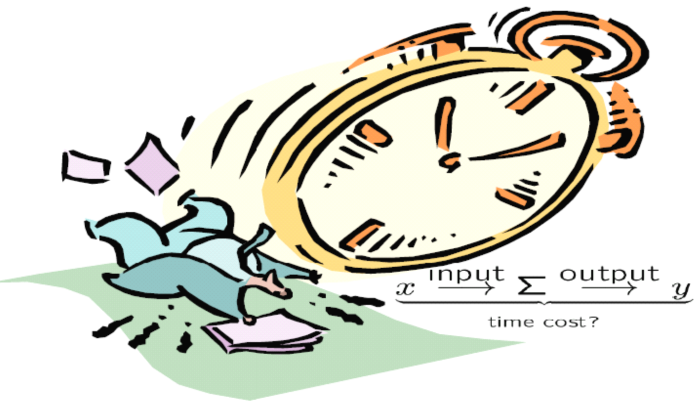
- ***Algorithm analysis framework***
- ***Asymptotic notations***
- ***Analysis of non-recursive algorithms***
- ***Analysis of recursive algorithms***

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

- ***Time efficiency T(N, I):***
  - *time consumed for an algorithm running on an computer*
  - *Element operations O_1, O_2, …, O_k,*
    execution time for the element operation t_1, t_2, …, t_k
    number of times element operation O_i is executed: e_i

$$T(N, I) = \sum_{i=1}^{k} t_i e_i(N, I)$$

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

Input size and basic operation examples

Problem	***Input size measure***	***Basic operation***
Searching for key in a list of n items	Number of list's items, i.e. n	Key comparison
Multiplication of two matrices	Matrix dimensions or total number of elements	Multiplication of two numbers
Evaluating $p(x) = a_n x^n + \ldots + a_0$	polynomial's degree n	Multiplication, addition
Typical graph problem	#vertices and/or edges	Visiting a vertex or traversing an edge

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.*

input size

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

running time

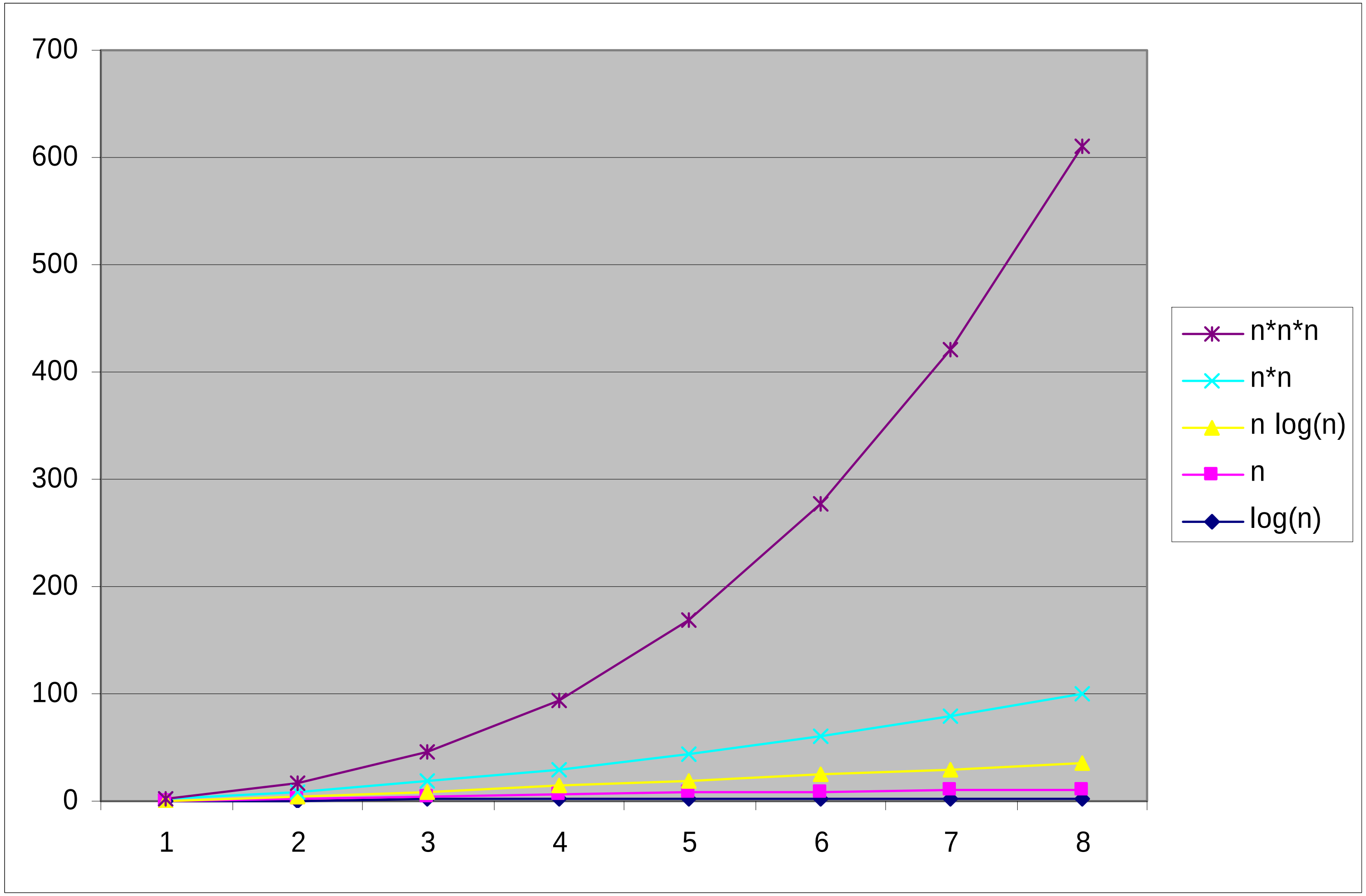
execution time
for the basic operation

Number of times the
basic operation is
executed

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

Analysis of Algorithms

Order of growth



Analysis of Algorithms

Order of growth

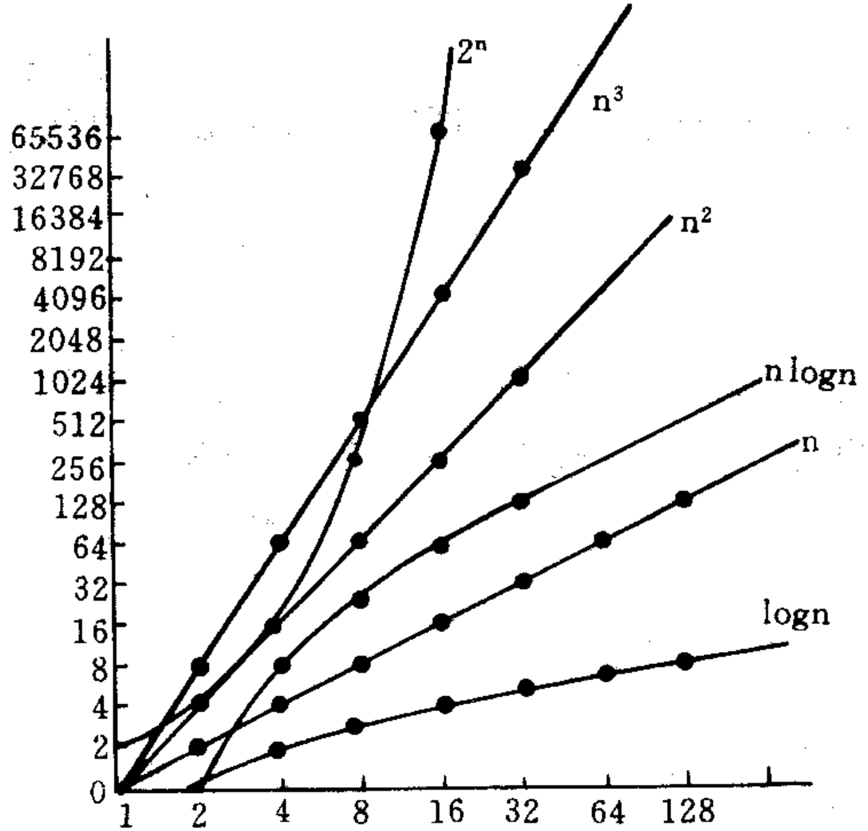


图 1.1 一般计算时间函数的曲线

Analysis of Algorithms

Basic Efficiency classes

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

fast — High time efficiency

1	constant
log n	logarithmic
n	linear
n log n	n log n
n^2	quadratic
n^3	cubic
2^n	exponential
$n!$	factorial

slow — 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 ← 0
    while i < n and A[i]≠K do
        i ← 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 ($0 \le p \le 1$);
  - 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 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?***

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

Tavg cannot be obtained by taking the average of *Tworst* and *Tbest*

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*

  $T(n) \to \infty$, 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 algorthm A, $n \to \infty$

  - ***Example:***

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

- *$t(n)$ 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

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

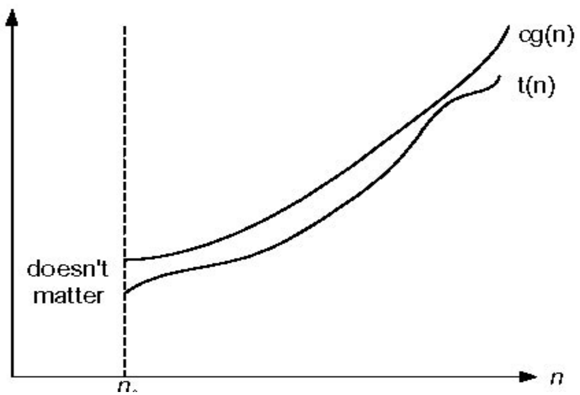
  - *Θ (g(n)): class of functions t(n) that grow at same rate as g(n)*

Asymptotic notations

O-notation

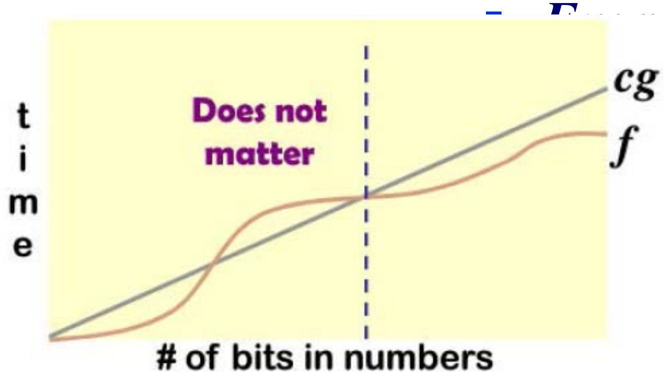
- *Formal definition:*
  - ***A function t(n) is said to be in O(g(n)), denoted t(n) $\in$ O(g(n)), if t(n) is bounded above by some constant multiple of g(n) for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that***

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



- ***E.g.:***

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



ple:

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

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

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

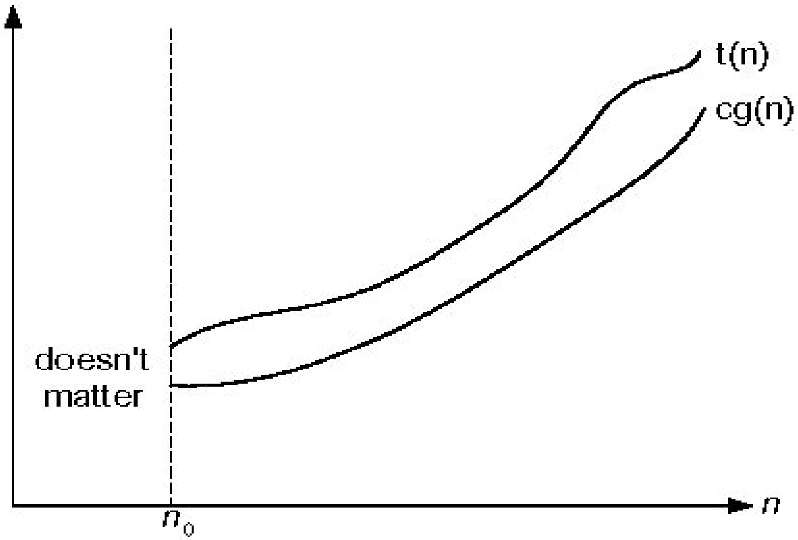
$20 \in O(n)$

Asymptotic notations

Ω-notation

- *Formal definition:*
  - ***A function t(n) is said to be in Ω(g(n)), denoted t(n) ∈ Ω(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***

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



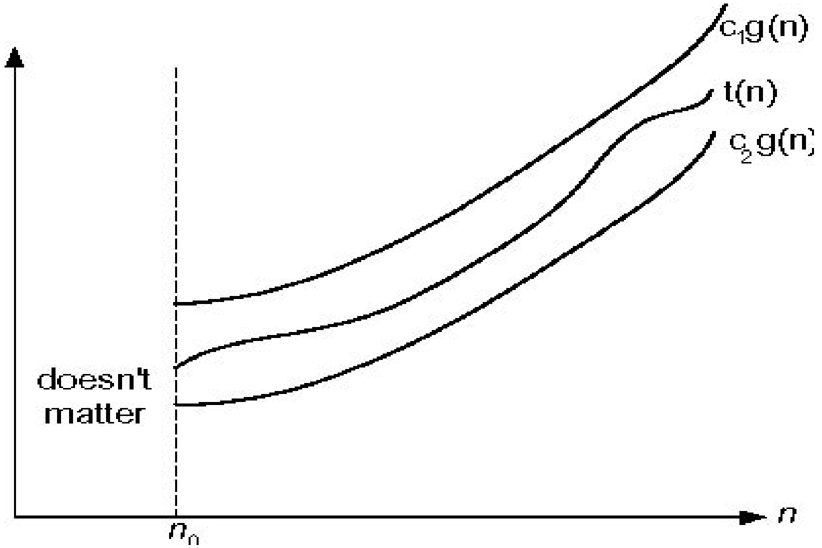
- ***Example:***
  - $10n^2 \in \Omega(n^2)$
  - $10n^2 + 2n \in \Omega(n^2)$
  - $10n^3 \in \Omega(n^2)$

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***

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



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

Asymptotic notations

Other notations

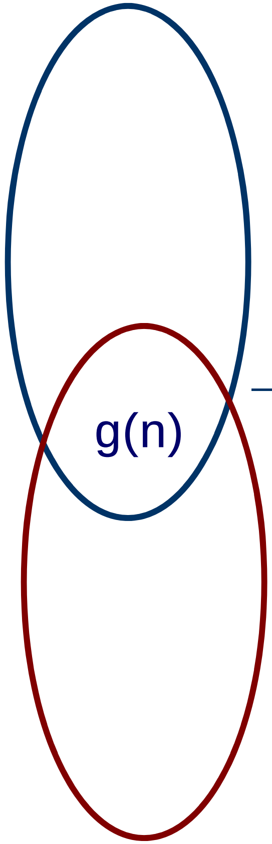
- $o(g(n))$:
  - ***A function t(n) is said to be in o(g(n)), denoted t(n) $\in$ o(g(n)), if t(n) is bounded above by some positive constant multiples of g(n) for all large n, i.e., if there exist some positive constant c and some nonnegative integer n0 such that***

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

- $\omega(g(n))$:
  - ***A function t(n) is said to be in ω (g(n)), denoted t(n) $\in$ ω(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 n0 such that***

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

Asymptotic notations



$>=$ higher or same order of growth

Ω(g(n)), functions that grow <u>at least as fast as</u> g(n)

$=$

Θ(g(n)), functions that grow <u>at the same rate as</u> g(n)

$<=$ smaller or same order of growth

O(g(n)), functions that grow <u>no faster than</u> 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 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

规则$O(g_1(n)) + O(g_2(n)) = O(\max\{ g_1(n), g_2(n) \})$ 的证明:

- $t_1(n) \in O(g_1(n))$， there exist some positive constant c_1 and nonnegative integer n_1 , such that, for all $n \geq n_1$， $t_1(n) \leq c_1 g_1(n)$.
- $t_2(n) \in O(g_2(n))$， there exist some positive constant c_2 and nonnegative integer n_2 , such that, for all $n \geq n_2$， $t_2(n) \leq c_2 g_2(n)$.
- denote $c_3 = \max\{c_1, c_2\}$， $n_3 = \max\{n_1, n_2\}$， $h(n) = \max\{g_1(n), g_2(n)\}$ 。
- for all $n \geq n_3$， we have
- $t_1(n) + t_2(n) \leq c_1 g_1(n) + c_2 g_2(n)$

$\leq c_3 g_1(n) + c_3 g_2(n) = c_3(g_1(n) + g_2(n))$

$\leq c_3 2 \max\{g_1(n), g_2(n)\}$

$= 2c_3 h(n) = O(\max\{g_1(n)\}, g_2(n)\}\})$.

Asymptotic notations

Using Limits for Comparing Orders of Growth

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

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

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

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

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

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

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

Asymptotic notations

some useful functions

取整函数

- $x-1 < \lfloor x \rfloor \le x \le \lceil x \rceil < x+1$；
- $\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$;
- 对于$n \ge 0$，$a,b>0$，有：
- $\lceil \lceil n/a \rceil /b \rceil = \lceil n/ab \rceil$;
- $\lfloor \lfloor n/a \rfloor /b \rfloor = \lfloor n/ab \rfloor$;
- $\lceil a/b \rceil \le (a+(b-1))/b$;
- $\lfloor a/b \rfloor \ge (a-(b-1))/b$;
- $f(x)=\lfloor x \rfloor$, $g(x)=\lceil x \rceil$为单调递增函数。

Asymptotic notations

some useful functions

多项式函数

- $p(n)= a_0+a_1n+a_2n^2+\ldots+a_dn^d$； $a_d>0$;
- $p(n) = \Theta(n^d)$;
- $f(n) = O(n^k) \Leftrightarrow f(n)$多项式有界；
- $f(n) = O(1) \Leftrightarrow f(n) \leq c$;
- $k \geq d \Rightarrow p(n) = O(n^k)$;
- $k \leq d \Rightarrow p(n) = \Omega(n^k)$;
- $k > d \Rightarrow p(n) = o(n^k)$;
- $k < d \Rightarrow p(n) = \omega(n^k)$.

Asymptotic notations

some useful functions

指数函数

- 对于正整数m,n和实数$a>0$:
- $a^0=1$;
- $a^1=a$;
- $a^{-1}=1/a$;
- $(a^m)^n = a^{mn}$;
- $(a^m)^n = (a^n)^m$;
- $a^m a^n = a^{m+n}$;
- $a>1 \Rightarrow a^n$为单调递增函数;
- $a>1 \Rightarrow \lim_{n\to\infty} \frac{n^b}{a^n} = 0 \Rightarrow n^b = o(a^n)$

Asymptotic notations

some useful functions

指数函数(2)

- $e^x \geq 1+x$;
- $|x| \leq 1 \Rightarrow 1+x \leq e^x \leq 1+x+x^2$;
- $e^x = 1+x+\Theta(x^2)$, as $x \to 0$;

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

$$\lim_{n\to\infty}\left(1+\frac{x}{n}\right)^n = e^x$$

Asymptotic notations

some useful functions

- 对数函数

  - $\log n = \log_2 n$; $\lg n = \log_{10} n$; $\ln n = \log_e n$;
  - $\log^k n = (\log n)^k$;
  - $\log \log n = \log(\log n)$;
  - for a>0,b>0,c>0

$$a = b^{\log_b a} \qquad \log_b(1/a) = -\log_b a \qquad \log_b a = \frac{1}{\log_a b}$$

$$\log_b a^n = n \log_b a \qquad \log_b a = \frac{\log_c a}{\log_c b}$$

$$\log_c(ab) = \log_c a + \log_c b \qquad a^{\log_b c} = c^{\log_b a}$$

Asymptotic notations

some useful functions

- 对数函数(2)
  - $|x| \le 1 \Rightarrow \quad \ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \cdots.$
  - for $x > -1$, $\quad \frac{x}{1+x} \le \ln(1+x) \le x$
  - for any $a > 0$, $\quad \lim_{n\to\infty} \frac{\log^b n}{(2^a)^{\log n}} = \lim_{n\to\infty} \frac{\log^b n}{n^a} = 0$, $\Rightarrow \log^b n = o(n^a)$

Asymptotic notations

some useful functions

- 阶乘函数

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

$$n! = 1 \cdot 2 \cdot 3 \cdots n$$

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

Stirling's approximation

Asymptotic notations

- **some useful functions**
  - 阶乘函数（2）

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\alpha_n} \qquad \frac{1}{12n+1} < \alpha_n < \frac{1}{12n}$$

$$n! = o(n^n)$$

$$n! = \omega(2^n)$$

$$\log(n!) = \Theta(n \log n)$$

Asymptotic notations

- ***some useful functions***
  - 常用的整数求和公式

算法分析中，在统计语句的频率时，求和公式的一般形式为：

$$\sum_{g(n) \le i \le h(n)} f(i)$$

如：

$$\sum_{1 \le i \le n} i^k = \Theta(n^{k+1})$$

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-, Ω-, and Θ-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

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

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

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

Time Efficiency of Non-recursive Algorithms

Another algorithm for Element uniqueness problem

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

  —— time complexity for quick-sort alg. is $\boldsymbol{\Theta(n \log n)}$

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

  ——no more than (n-1) comparisons

- so, the total ***time complexity is*** $\boldsymbol{\Theta(n \log n)}$

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 ← 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
```

- ***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

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

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***

    F(n) = 1 if n = 0
    = n * (n-1) * (n-2)... 3 * 2 * 1 if n > 0

  - ***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

- 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,

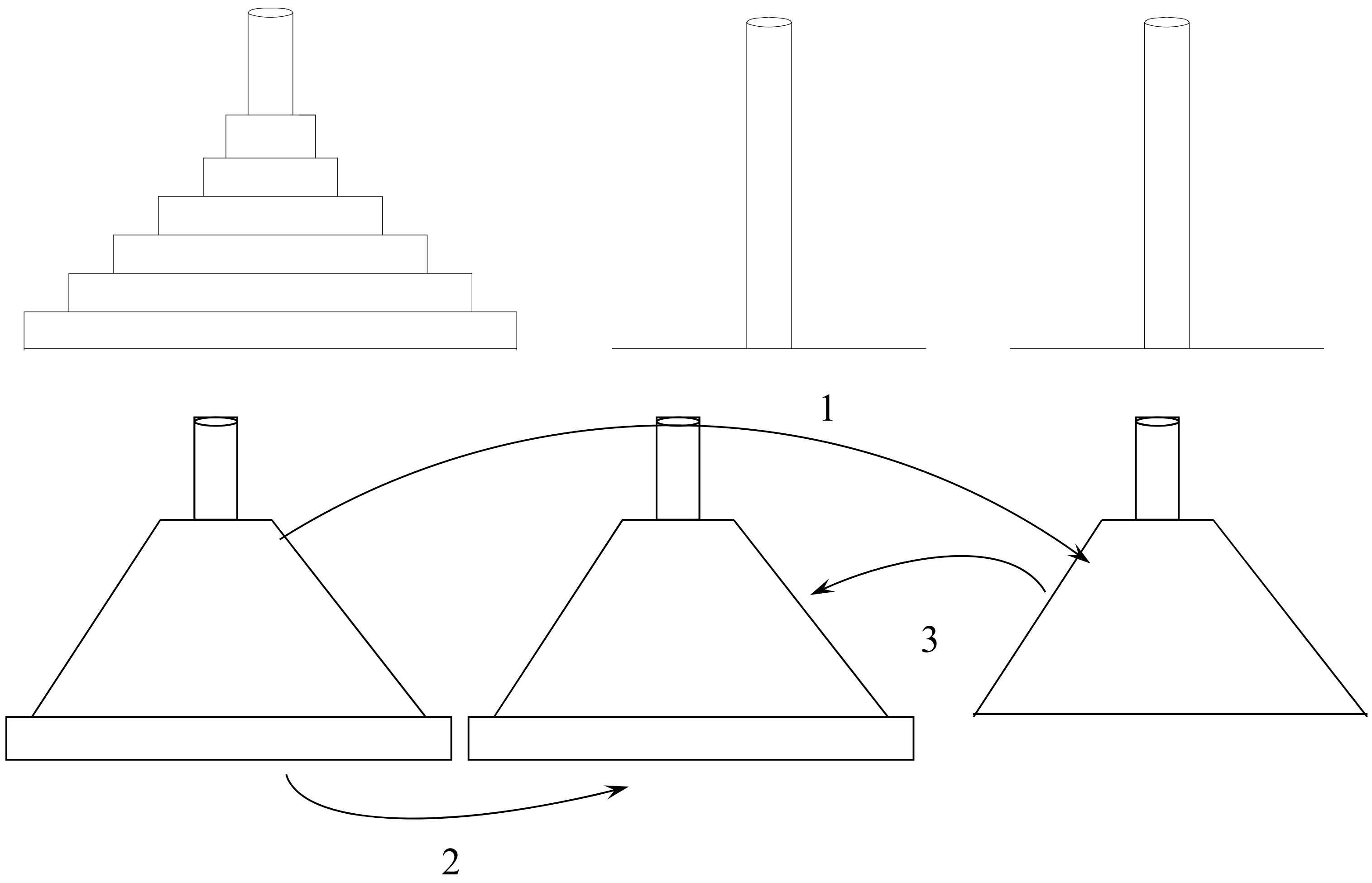
method of backward substitutions

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

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)

$C(1) = 1$

$C(n) = 2C(n-1) + 1 = 2^n - 1$ *for every n >1*

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

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

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

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

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

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

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

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 nondecreasing function and f(n) be a smooth function. If

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

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)

for $n = 2^k$,

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

$A(2^0) = 0$

backward substitutions:

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

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

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

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

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

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

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

Mathematical Analysis of Recursive Algorithms

*For example*3 *BinRec*

A(n) = A(⌊n/2⌋)+1 with A(1) = 0 → A(n) ∈ Θ (log n)

In fact, we can prove A(n) = ⌊ log n ⌋ *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 nth Fibonacci number :

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

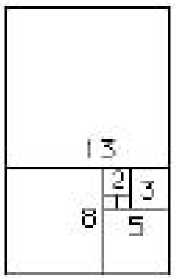
  $$F(0) = 0$$

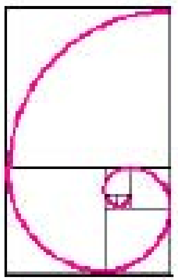
  $$F(1) = 1$$

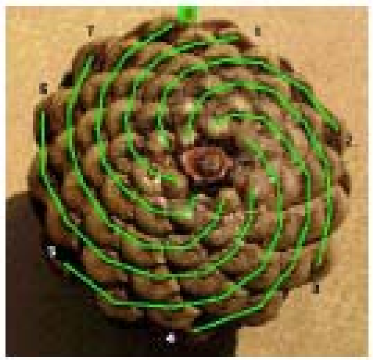
Mathematical Analysis of Recursive Algorithms

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。





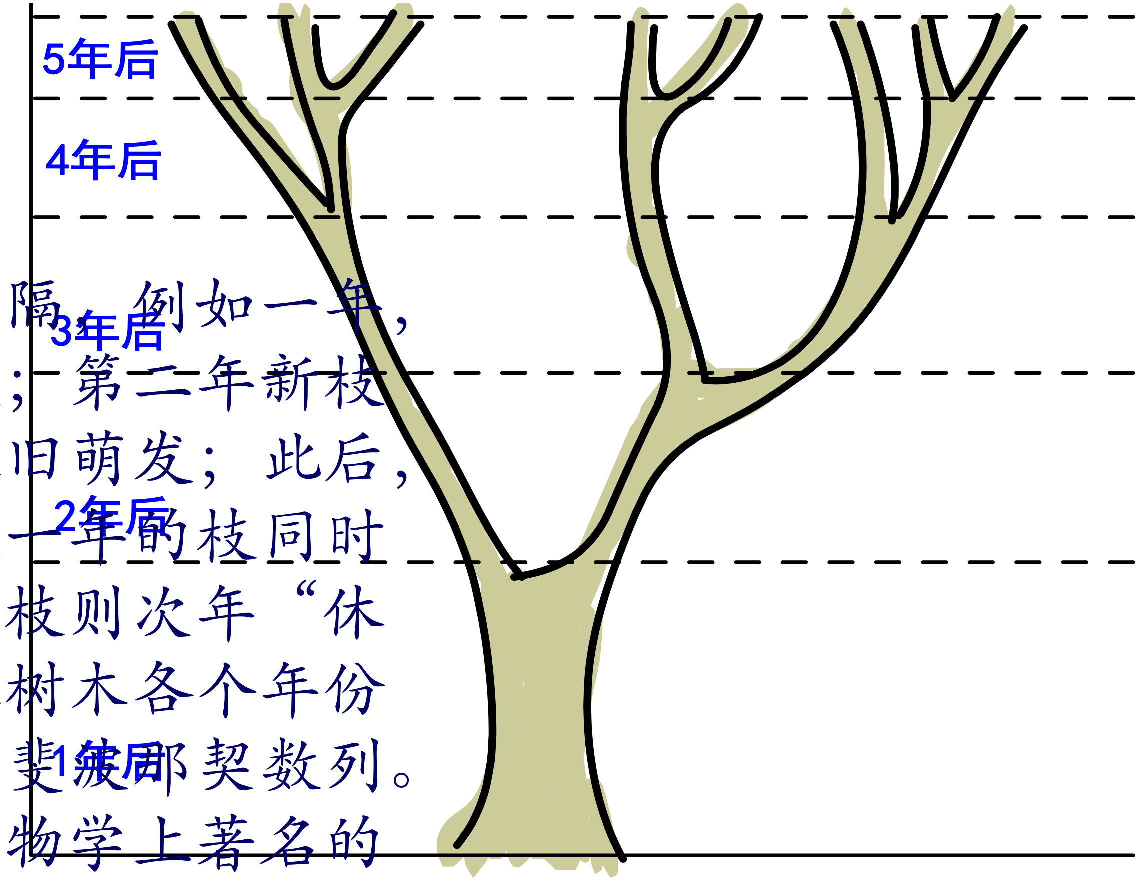



Mathematical Analysis of Recursive Algorithms

兔子问题

Mathematical Analysis of Recursive Algorithms

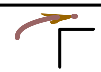
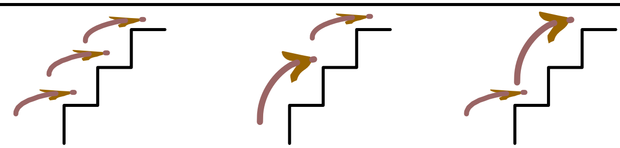
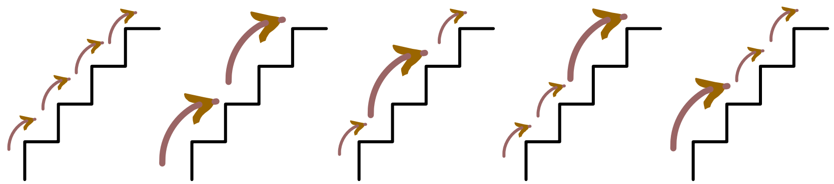
树枝生长问题



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"。

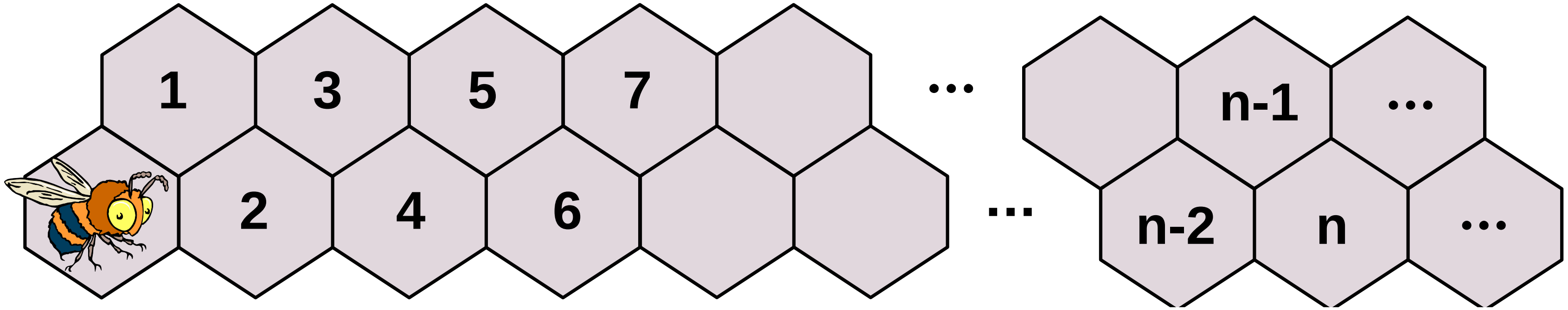
Mathematical Analysis of Recursive Algorithms

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： $u_n = u_{n-1} + u_{n-2}$