



华中科技大学
网络空间安全学院
School of Cyber Science and Engineering, HUST

算法设计与分析

文明

网络空间安全学院

华中科技大学

mwena@hust.edu.cn



算法设计与分析

个人简介

博士，香港科技大学（2014~2019）

本科，浙江大学（2010~2014）

访问学者，加州大学，戴维斯分校（2017~2018）

研究兴趣

- 软件安全（缺陷漏洞自动检测，模糊测试等）
- 程序分析（源代码分析，二进制分析等）
- 代码大数据分析

程序分析与安全

周三 5-6节/周五 3-4节



算法设计与分析

课程大纲

- 一、动态规划（递推、分治、倍增等重要算法思想）
- 二、二叉堆、并查集和树状数组
- 三、线段树、Splay及其他动态树
- 四、最小生成树、最短路径
- 五、图论、拓扑排序
- 六、KMP&Trie&AC自动机
- 七、网络流
- 八、线性规划
- 九、算法选讲



算法设计与分析

考核标准

简介：总共10周，每两周一次模拟考试共5次，课堂练习为在线作业。

考查成绩组成：两次模拟考试(要求到教室，10分) + 另外3次模拟考试至少参加一次(5分) + 完成课堂练习的50%(5分) + 12月份CSP考试折合分数

Do you know Algorithm 一词的由来

Algorithm(算法)一词的由来本身就十分有趣。

- ◆ 这个词一直到1957年之前在《韦氏新世界词典》(Webster's New World Dictionary)中还未出现，只能找到带有古代涵义的相近形式的一个词 “Algorism” (算术)，指的是用阿拉伯数字进行算术运算的过程。
- ◆ Algorism一词在历史中也经历了漫长的演变，据数学史学家发现研究，诞生于约公元前800多年，但后来 “algorism” 的形式和意义就变得面目全非了。
- ◆ 后来，如牛津英语字典所说明的，这个词是由于同arithmetic (算术)相混淆而形成的错拼词，由algorism变成algorithm。

◆ 软件是计算机的灵魂

◆ 算法是计算机软件的灵魂

➢ 软件=数据结构+算法

（ Niklaus Wirth，瑞士，计算机科学家，
图灵奖获得者，Pascal语言的发明者）

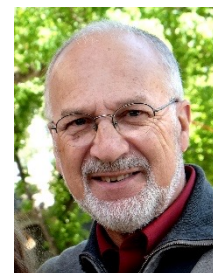
➢ 算法是计算机科学的核心

the core of computer science

（ David Harel ）

◆ 算法是国家科技综合实力的体现

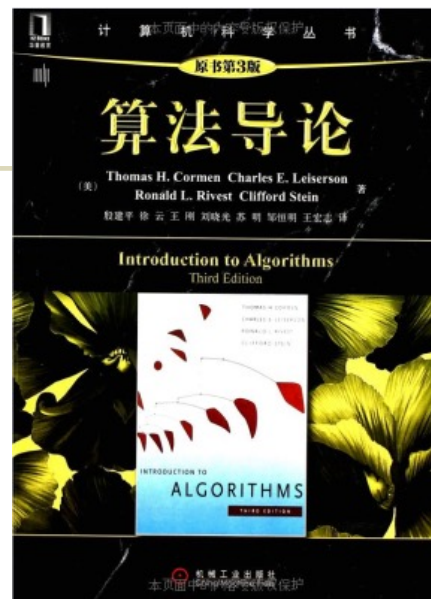
◆ 学习算法具有重要的现实意义



主要参考书

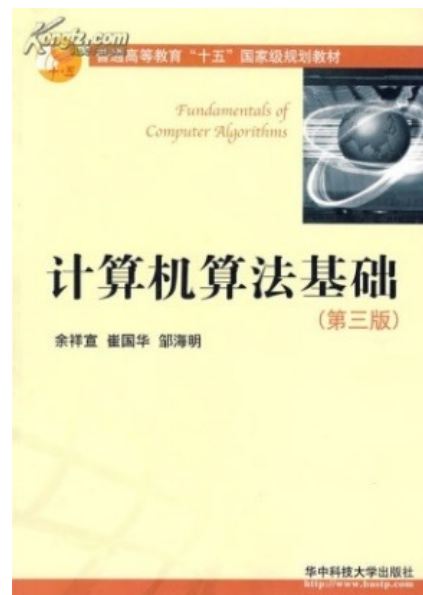
- **Introduction to algorithms**

Thomas H. Cormen, etc.,
third edition, The MIT Press.



- **计算机算法基础**

余祥宣等编著，华中科技大学出版社

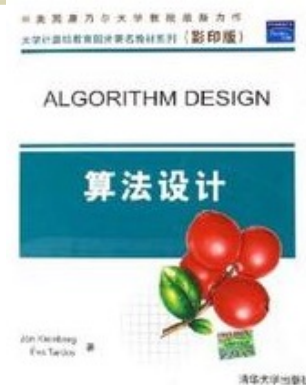


推荐阅读

- **Algorithm Design**

Jon Kleinberg, Eva Tardos etc.

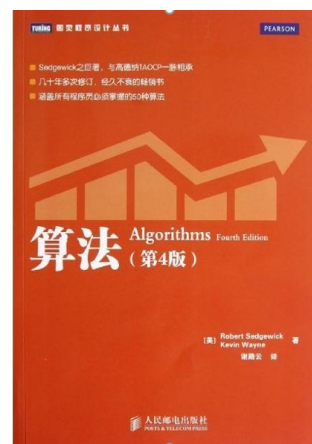
Cornell University



- **Algorithms**

Robert Sedgwick / Kevin

Princeton University



1.1 What are algorithms?

非形式地说，算法就是任何良定义（well-defined）的计算过程，该过程取某个值或值的集合作为输入（input），并产生某个值或者值的集合作为输出（output）。

—— 算法就是把输入转换成输出的计算步骤的一个序列。



—— 在计算机科学中，算法是使用计算机解一类问题的精确、有效方法的代名词；

◆ 算法 是一组有穷的规则，它规定了解决某一特定类型问题的一系列运算。

一般来说，**问题陈述**说明了期望的输入/输出关系，
而算法描述了一个特定的计算过程来实现这种输入到输出的转换。

Example：排序问题

‣ **问题陈述：**排序问题（*sorting problem*）的形式定义如下

Input: A sequence of n numbers $\langle a_1, a_2, \dots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \dots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \dots \leq a'_n$.

‣ **算 法：**冒泡排序、插入排序、归并排序等，将完成上述的
排序过程。

目标一致，但方法各异

什么叫做“正确的算法”？

- ▶ 若对每个输入实例，算法都以正确的输出停机，则称该算法是**正确**的，并称正确的算法解决了给定的问题。
- ▶ **不正确的算法**对某些输入实例可能根本不停机，也可能给出一个回答然后停机，但结果与期望不符甚至相反。
 - ▶ 但不正确的算法也并非绝对无用，在不正确的算法**错误率**可控时，有可能也是有用和有效的（如NP问题和近似算法）。
 - ▶ 但通常，我们还只是关心正确的算法。

1.2 作为一种技术的算法

- ◆ 对于算法，不仅强调其正确性，还有其性能的好坏。
 - 现实应用中，时间和空间都是有限的资源，我们应选择时间和空间方面有效的算法来求解问题。
 - 效率（Efficiency）对于算法的有效性有非常重要的影响。

为求解一个问题而设计的不同算法在效率方面常常具有显著的差别，这种差别可能比由于硬件和软件造成的差别还要重要。

For example：排序问题

insertion sort: time roughly equal to $c_1 n^2$

merge sort: time roughly equal to $c_2 n \lg n$

Let $c_1 = 2$, and $c_2 = 50$,
假设对1000万个数进行排序

令 $c_1 = 2$, and $c_2 = 50$, 对1000万 (10^7) 个数进行排序。并设插入排序在每秒执行百亿 (10^{10}) 条指令的计算机上运行, 归并排序在每秒仅执行1000万条指令的计算机上运行, 计算机速度相差1000倍。程序的实际执行时间有什么差别呢?

* *insertion sort:*

$$\frac{2 \cdot (10^7)^2 \text{ instructions}}{10^{10} \text{ instructions/second}} = 20,000 \text{ seconds (more than 5.5 hours) ,}$$

* *merge sort:*

$$\frac{50 \cdot 10^7 \lg 10^7 \text{ instructions}}{10^7 \text{ instructions/second}} \approx 1163 \text{ seconds (less than 20 minutes) .}$$

- * 可见使用一个运行时间“增长”较慢, 时间复杂度低、时间效率高的算法, 即使采用较差的编译器、运行速度较慢的计算机, 在数据规模足够大的时候, 也比时间效率低的算法快。

- ◆ 上述的例子表明，我们应该像对待计算机硬件一样把算法看成是一种技术，研究并选择有效的算法来解决现实问题。

- 思考：如果计算机速度无限快、内存无限大，那么我们还有研究算法的必要吗？

- ◆ 随着计算机能力的不断增强，我们使用计算机来求解的问题的规模会越来越大，算法之间效率的差异也变得越来越显著。因此，是否具有算法知识与技术的坚实基础是区分真正熟练的程序员与初学者的一个基本特征。

- 使用现代计算技术，如果你对算法懂得不多，尽管你也可以完成一些任务，但如果有一个好的算法背景，你可以做更多的事情，也会做得更好一些。

2.1 插入排序 (Insertion sort)

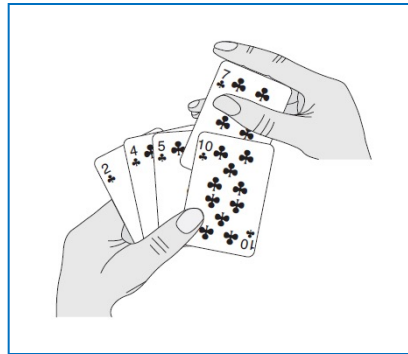
■ 排序问题的描述：

Input: A sequence of n numbers $\langle a_1, a_2, \dots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \dots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \dots \leq a'_n$.

1. 插入排序的基本思想

➤ 插入排序的基本思想：



➤ 插入排序是一个对少量元素排序比较有效的算法

2. 插入排序的伪代码描述：INSERTION-SORT

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1..j - 1].
4      i = j - 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i - 1
8      A[i + 1] = key
```

变量可以没有类型说明
语法结构类似现实语言

- INSERTION-SORT是一种“**原址排序**”：输入的原始数据在数组A中，算法在数组A空间中重排这些数，并且在任何时候，最多只有其中的常数个数字存储在数组之外。
- 过程结束时，输入数组A包含排序好的输出序列。

设 $A = \langle 5, 2, 4, 6, 1, 3 \rangle$ ，INSERTION-SORT在A上的排序过程

如图所示：

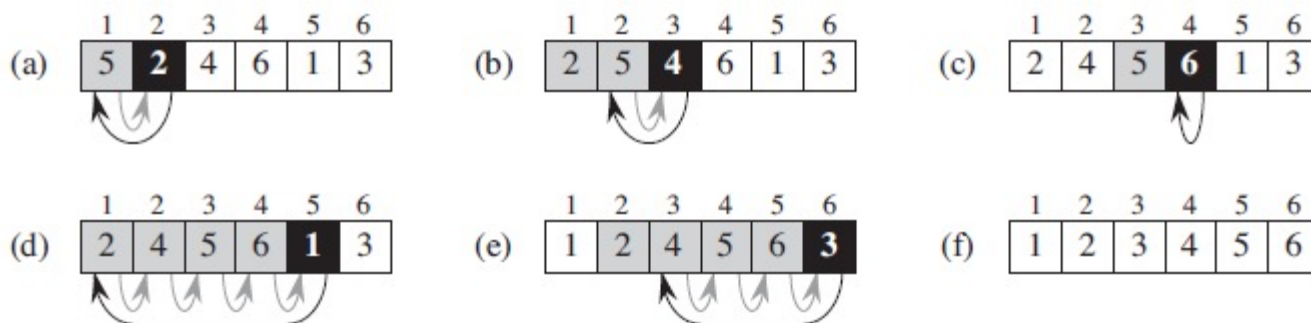


Figure 2.2 The operation of INSERTION-SORT on the array $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. (a)–(e) The iterations of the **for** loop of lines 1–8. In each iteration, the black rectangle holds the key taken from $A[j]$, which is compared with the values in shaded rectangles to its left in the test of line 5. Shaded arrows show array values moved one position to the right in line 6, and black arrows indicate where the key moves to in line 8. (f) The final sorted array.

算法的五个重要特性

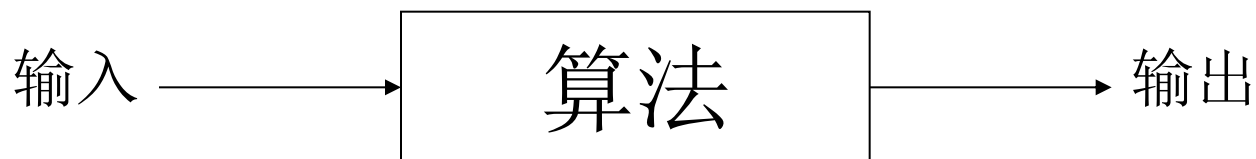
确定性、能行性、输入、输出、有穷性

- 1) **确定性**：算法使用的每种运算必须要有确切的定义，不能有二义性。
 - 不符合确定性的运算如：5/0，将6或7与x相加
- 2) **能行性**：算法中有待实现的运算都是基本的运算，原理上每种运算都能由人用纸和笔在“有限”的时间内完成。
 - 求和： $s = 1+2+3+4+\cdots+n+\cdots$

3) 输入：每个算法都有0个或多个输入。

- 这些输入是在算法开始之前给出的量，取自于特定的对象集合——定义域

4) 输出：一个算法产生一个或多个输出，这些输出是同输入有某种特定关系的量。



5) 有穷性

一个算法总是在执行了有穷步的运算之后终止。

□ **计算过程**：满足确定性、能行性、输入、输出，
但不一定满足有穷性的一组规则称为
计算过程。

- 操作系统是计算过程的典型代表：（不终止的运行过程）
- 算法是“可以终止的计算过程”。

2.2 分析算法

- 分析算法目的主要是预测算法需要资源的程度。
- 资源包括：
 - 时间
 - 空间：内存
 - 其他：通信带宽、硬件资源等

但从算法的角度，我们主要关心算法的时间复杂度和空间复杂度。

1. 分析算法的目的

- 算法选择的需要
- 算法优化的需要

2. 重要的假设和约定

1) 计算机模型的假设

- 计算机形式理论模型：有限自动机（FA）、Turing 机
- **RAM**模型（random-access machine，随机访问机）
- **通用的顺序计算机模型**：
 - ◆ 单CPU——串行算法（部分内容可能涉及多CPU（核）和并行处理的概念）
 - ◆ 有足够的“内存”，并能在固定的时间内存取数据单元

2) 计算的约定

算法/程序的执行时间是什么？

$O(n^2)$ 、 $O(n \log n)$...和执行时间有什么关系呢？

- 算法的执行时间是算法中所有运算执行时间的总和
可以表示为:

$$\text{算法的执行时间} = \sum f_i t_i$$

其中, f_i : 是运算i的执行次数, 称为该运算的**频率计数**

t_i : 是运算i在**实际的计算机**上每执行一次所用的时间

- ▣ f_i 仅与算法的控制流程有关, 与实际使用的计算机硬件和编制程序的语言无关。
- ▣ t_i 与程序设计语言和计算机硬件有关。

运算的分类

依照运算的**时间特性**，将运算分为**时间囿界于常数的运算**和**时间非囿界于常数的运算**。

- **时间囿界于常数的运算**：

特点：**执行时间是固定量**，与操作数无关。

例： $1+1 = 2$ vs $10000+10000 = 20000$

$100*100 = 10000$ vs $10000*10000 = 100000000$

CALL INSERTIONSORT

■ 时间非囿界于常数的运算

特点：运算的执行时间与操作数相关，**每次执行的时间是一个不定的量。**

如：

- 字符串操作：与字符串中字符的数量成正比，如字符串的比较运算strcmp。
- 记录操作：与记录的属性数、属性类型等有关

时间非囿界于常数的运算时，将其分解成若干时间囿界于常数的运算即可。

如：字符串比较时间 t_{string}

$$t_{\text{string}} = \text{Length}(\text{String}) * t_{\text{char}}$$

3) 工作数据集的选择

- 算法的执行情况与输入的数据有什么样的关系呢？
 - 算法的执行时间与输入数据的规模相关，一般规模越大，执行时间越长。
 - 编制不同的数据配置，分析算法的最好、最坏、平均工作情况是算法分析的一项重要工作

3. 算法分析

- ◆ 算法分析的目的是求取算法时间/空间复杂度的**限界函数**。
- ◆ **限界函数**通常是关于问题规模 n 的**特征函数**，被表示成 O 、 Ω 或 Θ 的形式。

如：归并排序的时间复杂度是 $\Theta(n \log n)$ 。

- ◆ **怎么获取算法复杂度的特征函数？**

如何进行时间分析？

- 统计算法中各类运算的执行次数

- **频率计数**，即算法中语句或运算的**执行次数**。

- 顺序结构中的运算/语句执行次数计为1

- 嵌套结构中的运算/语句执行次数等于被循环执行的次数

例：执行次数的统计

$x \leftarrow x + y$ for $i \leftarrow 1$ to n do

$x \leftarrow x + y$

repeat

(a)

(b)

for $i \leftarrow 1$ to n do

for $j \leftarrow 1$ to n do

$x \leftarrow x + y$

repeat

repeat

(c)

分析：

(a)： $x \leftarrow x + y$ 执行了1次

(b)： $x \leftarrow x + y$ 执行了 n 次

(c)： $x \leftarrow x + y$ 执行了 n^2 次

思考：

for $i \leftarrow 1$ to n do

for $j \leftarrow i$ to n do

$x \leftarrow x + y$

repeat

repeat

(d)

插入排序时间分析：

INSERTION-SORT(<i>A</i>)	<i>cost</i>	<i>times</i>
1 for <i>j</i> = 2 to <i>A.length</i>	c_1	n
2 $key = A[j]$	c_2	$n - 1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j - 1]$.	0	$n - 1$
4 $i = j - 1$	c_4	$n - 1$
5 while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^n t_j$
6 $A[i + 1] = A[i]$	c_6	$\sum_{j=2}^n (t_j - 1)$
7 $i = i - 1$	c_7	$\sum_{j=2}^n (t_j - 1)$
8 $A[i + 1] = key$	c_8	$n - 1$

- **cost列**：给出了每一行语句执行一次的时间。
- **times列**：给出了每一行语句被执行的总次数。

- 整个算法的执行时间是执行所有语句的时间之和。
 - 如果语句*i*需要执行*n*次，每次需要 c_i 的时间，则该语句的总执行时间是： $c_i n$.
 - 令 $T(n)$ 是输入*n*个值时 INSERTION-SORT 的运行时间，则有：

$$\begin{aligned} T(n) = & c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) \\ & + c_7 \sum_{j=2}^n (t_j - 1) + c_8(n-1) . \end{aligned}$$

即使规模相同，一个算法的执行时间也可能依赖于给定的输入。

如：INSERTION-SORT

■ 最好情况：

INSERTION-SORT(<i>A</i>)	<i>cost</i>	<i>times</i>
1 for <i>j</i> = 2 to <i>A.length</i>	c_1	n
2 $key = A[j]$	c_2	$n - 1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j - 1]$.	0	$n - 1$
4 $i = j - 1$	c_4	$n - 1$
5 while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^n t_j$
6 $A[i + 1] = A[i]$	c_6	$\sum_{j=2}^n (t_j - 1)$
7 $i = i - 1$	c_7	$\sum_{j=2}^n (t_j - 1)$
8 $A[i + 1] = key$	c_8	$n - 1$

- **最好情况**：初始数组已经排序好了。此时对每一次for循环，内部的while循环体都不会执行。

所以最好情况的运行时间为：

$$\begin{aligned} T(n) &= c_1n + c_2(n-1) + c_4(n-1) + \underline{c_5(n-1)} + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8). \end{aligned}$$

- **最坏情况：** 初始数组是反向排序的。

INSERTION-SORT(<i>A</i>)	<i>cost</i>	<i>times</i>
1 for <i>j</i> = 2 to <i>A.length</i>	c_1	n
2 $key = A[j]$	c_2	$n - 1$
3 // Insert $A[j]$ into the sorted sequence $A[1 \dots j - 1]$.	0	$n - 1$
4 $i = j - 1$	c_4	$n - 1$
5 while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^n t_j$
6 $A[i + 1] = A[i]$	c_6	$\sum_{j=2}^n (t_j - 1)$
7 $i = i - 1$	c_7	$\sum_{j=2}^n (t_j - 1)$
8 $A[i + 1] = key$	c_8	$n - 1$

- 最坏情况的运行时间为：

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + \underline{c_5 \left(\frac{n(n+1)}{2} - 1 \right)} \\ &\quad + \underline{c_6 \left(\frac{n(n-1)}{2} \right)} + \underline{c_7 \left(\frac{n(n-1)}{2} \right)} + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} \right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8 \right) n \\ &\quad - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

- 除了最坏情况、最好情况分析，还有平均情况分析。

平均情况是规模为 n 的情况下，算法的平均执行时间。

- 通常情况下，平均运行时间是算法在各种情况下执行时间之和与输入情况数的比值（算术平均）。

但一般，我们更关心算法的最坏情况执行时间。

为什么？

- 一个算法的最坏情况执行时间给出了任何输入的运行时间的一个上界。知道了这个界，就能确保算法绝不需要更长的时间，我们就不需要再对算法做更坏的打算。
- 对某些算法，最坏情况经常出现。
- 对很多算法，平均情况往往与最坏情况大致一样。
 - ▣ 如插入排序，就一般情况而言，while循环中为确定当前 $A[j]$ 的插入位置，平均要检查一半的元素。那么导致的平均执行时间就数量级来说和最坏情况一样，依然是 n 的二次函数，只是常系数小了一些。

◆ **限界函数**：取自频率计数函数表达式中的**最高次项**，

并忽略常系数，记为： $g(n)$ 。

- $g(n)$ 通常是关于 n 的形式简单的单项式函数，如： $\log n$ ， $n \log n$ 等
- $g(n)$ 通常是对算法中**最复杂**的计算部分分析而来的。

Growth of Functions

函数的增长

3.1 限界函数的定义

算法时间复杂度的限界函数常用的有三个：

上界函数、下界函数、渐进紧确界函数。

对应的渐进记号： O Ω Θ

定义如下：

记：算法的实际执行时间为 $f(n)$ ，分析所得的限界函数为 $g(n)$

其中， n ：问题规模的某种测度。

$f(n)$ ：是与机器及语言有关的量。

$g(n)$ ：是事前分析的结果，一个形式简单的函数，与频率计数有关、而与机器及语言无关。

1. 上界，O记号

$O(g(n))$ 表示以下函数的集合：

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

- ◆ 若 $f(n)$ 和 $g(n)$ 满足以上关系，则记为 $f(n) \in O(g(n))$ ，表示 $f(n)$ 是集合 $O(g(n))$ 的成员。并通常记作

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

含义：

$f(n) = O(g(n))$ 表示如果算法用 n 值不变的同的一类数据（规模相等，性质相同）在某台机器上运行，所用的时间总小于 $|g(n)|$ 的一个常数倍。

- ◆ O记号给出的是渐进上界，称为上界函数（upper bound）
- ◆ 上界函数代表了算法最坏情况下的时间复杂度。
- ◆ 在确定上界函数时，应试图找阶最小的 $g(n)$ 作为 $f(n)$ 的上界函数——紧确上界(*tight upper bound*)。

如：若： $3n+2=O(n^2)$ 则是松散的界限；

而： $3n+2=O(n)$ 就是紧确的界限。

2. 下界， Ω 记号

$\Omega(g(n))$ 表示以下函数的集合：

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

- * 若 $f(n)$ 和 $g(n)$ 满足以上关系，则记为 $f(n) \in \Omega(g(n))$ ，表示 $f(n)$ 是集合 $\Omega(g(n))$ 的成员。并通常记作

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

含义：

$f(n) = \Omega(g(n))$ 表示如果算法用 n 值不变的同的一类数据在某台机器上运行，所用的时间总不小于 $|g(n)|$ 的一个常数倍。

-
- * Ω 记号给出一个渐进下界，称为下界函数（lower bound）。
 - * 在确定下界函数时，应试图找出数量级最大的 $g(n)$ 作为 $f(n)$ 的下界函数——紧确下界。

3. 渐进紧确界， Θ 记号：

$\Theta(g(n))$ 表示以下函数的集合：

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

- ◆ 若 $f(n)$ 和 $g(n)$ 满足以上关系，记为： $f(n) \in \Theta(g(n))$ ，表示 $f(n)$ 是 $\Theta(g(n))$ 的一员。通常记为：

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

含义：

$f(n)=\Theta(g(n))$ 表示如果算法用 n 值不变的同一类数据在某台机器上运行，所用的时间既不小于 $|g(n)|$ 的一个常数倍，也不大于 $|g(n)|$ 的一个常数倍，亦即 g 既是 f 的下界，也是 f 的上界。

- ◆ Θ 记号给出的是渐进紧确界(*asymptotically tight bound*)
- ◆ 从时间复杂度的角度看, $f(n) = \Theta(g(n))$ 表示是算法在最好和最坏情况下的计算时间就一个常数因子范围内而言是相同的, 可看作:

既有 $f(n) = \Omega(g(n))$, 又有 $f(n) = O(g(n))$

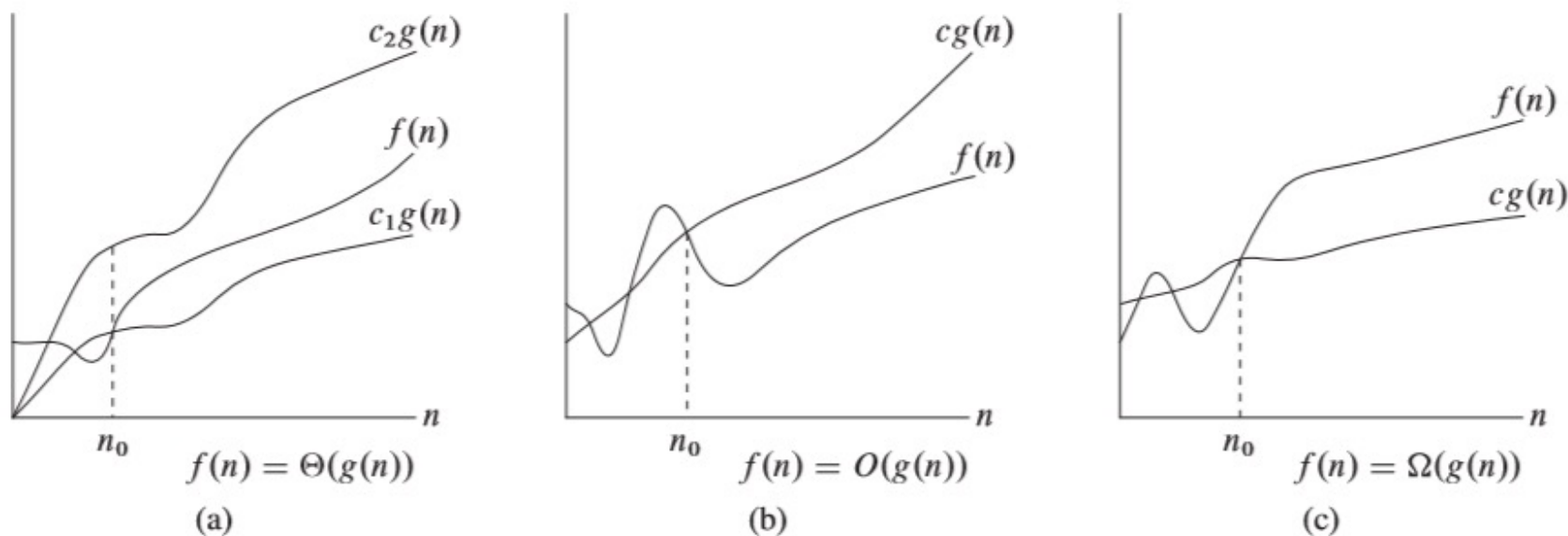


Figure 3.1 Graphic examples of the Θ , O , and Ω notations. In each part, the value of n_0 shown is the minimum possible value; any greater value would also work. **(a)** Θ -notation bounds a function to within constant factors. We write $f(n) = \Theta(g(n))$ if there exist positive constants n_0 , c_1 , and c_2 such that at and to the right of n_0 , the value of $f(n)$ always lies between $c_1g(n)$ and $c_2g(n)$ inclusive. **(b)** O -notation gives an upper bound for a function to within a constant factor. We write $f(n) = O(g(n))$ if there are positive constants n_0 and c such that at and to the right of n_0 , the value of $f(n)$ always lies on or below $cg(n)$. **(c)** Ω -notation gives a lower bound for a function to within a constant factor. We write $f(n) = \Omega(g(n))$ if there are positive constants n_0 and c such that at and to the right of n_0 , the value of $f(n)$ always lies on or above $cg(n)$.

算法时间复杂度的分类

根据**上界函数**的特性，可以将算法分为：**多项式时间算法**和**指数时间算法**。

➤ **多项式时间算法**：可用多项式函数对计算时间限界的算法

* 常见的多项式限界函数有：

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3)$$

复杂度越来越高

➤ **指数时间算法**：计算时间用指数函数限界的算法。

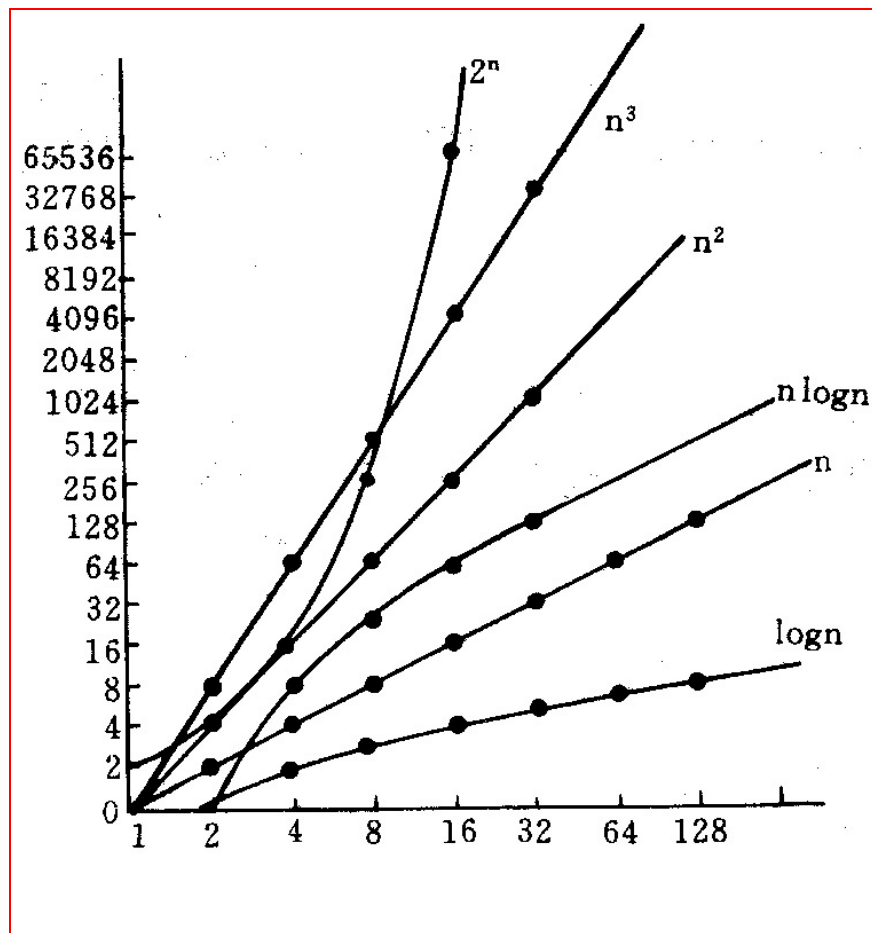
* 常见的指数时间限界函数：

$$O(2^n) < O(n!) < O(n^n)$$

复杂度越来越高

- * 当 n 取值较大时，指数时间算法和多项式时间算法在计算时间上非常悬殊。

计算时间的典型函数曲线：



对算法复杂性的一般认识

- 当数据集的规模很大时，要在现有的计算机系统上运行具有比 $O(n\log n)$ 复杂度还高的算法是比较困难的。
- 指数时间算法只有在 n 取值非常小时才实用。
- 要想在顺序处理机上扩大所处理问题的规模，有效的途径是降低算法的计算复杂度，而不是（仅仅依靠）提高计算机的速度。

例：证明 $n^2/2 - 3n = \Theta(n^2)$ （自学）

分析：根据 Θ 的定义，仅需确定正常数 c_1 , c_2 , and n_0 以使得

$$\text{对所有的 } n \geq n_0, \text{ 有: } c_1 n^2 \leq \frac{1}{2} n^2 - 3n \leq c_2 n^2$$

$$\text{解：两边同除 } n^2 \text{ 得: } c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2 .$$

只要 $c_1 \leq 1/14$, $c_2 \geq 1/2$, 且 $n_0 \geq 7$, 不等式即成立。

所以，这里取： $c_1 = 1/14$, $c_2 = 1/2$, and $n_0 = 7$, 即得证：

$$n^2/2 - 3n = \Theta(n^2) \quad \blacksquare$$

注：还有其它常量可选，但根据定义，只要存在一组选择（如上述的 c_1 , c_2 , 和 n_0 ）即得证。


再如：证明 $6n^3 \neq \Theta(n^2)$

采用反证法：假设 $6n^3 = \Theta(n^2)$

则存在 c_2 和 n_0 ，使得对所有的 $n \geq n_0$ ，有： $6n^3 \leq c_2 n^2$ 。

两边同除 n^2 得： $n \leq c_2/6$ ，

而 c_2 是常量，所以对任意大的 n ，该式不可能成立。

所以假设不成立。 

关于渐进记号的进一步说明：

(1) $f(n)=O(g(n))$ 不能写成 $g(n)=O(f(n))$ ， Ω 相同。

- $f(n)$ 与 $g(n)$ 并不等价，这里的等号不是通常相等的含义。

(2) 关于 $\Theta(1)$ ($O(1)$ 、 $\Omega(1)$ 有类似的含义)

- 因为任意常量都可看做是一个0阶多项式，所以可以把任意常量函数表示成 $\Theta(n^0)$ 或 $\Theta(1)$ 。
- 通常用 $\Theta(1)$ 表示具有常量计算时间的复杂度，即算法的执行时间为一个固定量，与问题的规模 n 没关系。

注： $\Theta(1)$ 有“轻微活用”的意思，因为该表达式没有指出是什么量趋于无穷
(见P27的说明)

(3) 等式和不等式中的渐进记号

类似以下的表达式：

$$T(n) = 2T(n/2) + \Theta(n)$$

当**渐进记号**出现在某个公式中，该如何理解？如上式的 $\Theta(n)$

- 将其解释为代表我们不关注名称的**匿名函数**，用以消除表达式中一些无关紧要的细节。
 - 这些细节存在但不被特别关注，分析时还是只对 $T(n)$ 的渐进行为感兴趣。**渐进记号仅代表低阶项部分**，在实际化简的过程中，要根据需要予以“具体化”，然后再进行化简处理。
 - 详见P28~29

4. o, ω 记号

O 、 Ω 给出的渐进上界或下界可能是也可能不是渐进紧确的。
这里引入 o 、 ω 记号专门用来表示一种非渐进紧确的上界或下界。

o 记号：对任意正常数 c ，存在常数 $n_0 > 0$ ，使对所有的 $n \geq n_0$ ，
有 $|f(n)| \leq c|g(n)|$ ，则记作： $f(n) = o(g(n))$ 。

含义：在 o 表示中，当 n 趋于无穷时， $f(n)$ 相对于 $g(n)$ 来说变得微不足道了，即 $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$

例： $2n = o(n^2)$ ，但 $2n \neq o(n)$ 、 $2n^2 \neq o(n^2)$

ω 记号：对任意正常数 c ，存在常数 $n_0 > 0$ ，使对所有的 $n \geq n_0$ ，
有 $c|g(n)| \leq |f(n)|$ ，则记作： $f(n) = \omega(g(n))$ 。

含义：在 ω 表示中，当 n 趋于无穷时， $f(n)$ 相对于 $g(n)$ 来说变得任意大了，即 $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$

例： $n^2/2 = \omega(n)$ ，但 $n/2 \neq \omega(n)$ 、 $n^2/2 \neq \omega(n^2)$

3.2 限界函数的性质

① 传递性 (Transitivity) :

$$\begin{aligned}f(n) = \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) &\implies f(n) = \Theta(h(n)), \\f(n) = O(g(n)) \text{ and } g(n) = O(h(n)) &\implies f(n) = O(h(n)), \\f(n) = \Omega(g(n)) \text{ and } g(n) = \Omega(h(n)) &\implies f(n) = \Omega(h(n)), \\f(n) = o(g(n)) \text{ and } g(n) = o(h(n)) &\implies f(n) = o(h(n)), \\f(n) = \omega(g(n)) \text{ and } g(n) = \omega(h(n)) &\implies f(n) = \omega(h(n)).\end{aligned}$$

② 自反性 (Reflexivity) :

$$\begin{aligned}f(n) &= \Theta(f(n)), \\f(n) &= O(f(n)), \\f(n) &= \Omega(f(n)).\end{aligned}$$

③ 对称性 (Symmetry) :

$$f(n) = \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n)).$$

④ 转置对称性 (Transpose Symmetry) :

$$\begin{aligned}f(n) = O(g(n)) &\text{ if and only if } g(n) = \Omega(f(n)), \\f(n) = o(g(n)) &\text{ if and only if } g(n) = \omega(f(n)).\end{aligned}$$

仅从函数的数学定义理解其含义

3.3 相关定理

定理1.1（多项式定理）若 $A(n)=a_m n^m+\cdots+a_1 n+a_0$ 是一个 n 的 m 次多项式，则有 $A(n) = O(n^m)$

即：变量 n 的固定阶数为 m 的多项式，与此多项式的最高阶 n^m 同阶。

证明：取 $n_0=1$ ，当 $n \geq n_0$ 时，有

$$\begin{aligned} |A(n)| &\leq |a_m| n^m + \cdots + |a_1| n + |a_0| \\ &= (|a_m| + |a_{m-1}|/n + \cdots + |a_0|/n^m) n^m \\ &\leq (|a_m| + |a_{m-1}| + \cdots + |a_0|) n^m \end{aligned}$$

令 $c = |a_m| + |a_{m-1}| + \cdots + |a_0|$ ，即有 $|A(n)| \leq cn^m$ 。证毕。

例：考虑二次函数 $f(n)=an^2+bn+c$ ，其中 a 、 b 、 c 为常量且 $a>0$ 。

根据上述思路，去掉低阶项并忽略常系数后即得：

$$f(n) = \Theta(n^2)$$

对比形式化证明：

取常量： $c_1=a/4$ ， $c_2=7a/4$ ， $n_0 = 2 \cdot \max(|b|/a, \sqrt{|c|/a})$

可以证明对所有的 $n \geq n_0$ ，有：

$$0 \leq c_1 n^2 \leq an^2 + bn + c \leq c_2 n^2$$

一般而言，对任意多项式 $p(n) = \sum_{i=0}^d a_i n^i$ ，其中 a_i 为常数且 $a_d > 0$ ，都有 $p(n) = \Theta(n^d)$

用于估算复杂性（函数阶的大小）的定理

[定理1.2] 对于任意正实数 x 和 ε ，有下面的不等式：

- 1) 存在某个 n_0 ，使得对于任何 $n \geq n_0$ ，有 $(\log n)^x < (\log n)^{x+\varepsilon}$
- 2) 存在某个 n_0 ，使得对于任何 $n \geq n_0$ ，有 $n^x < n^{x+\varepsilon}$ 。
- 3) 存在某个 n_0 ，使得对于任何 $n \geq n_0$ ，有 $(\log n)^x < n$ 。
- 4) 存在某个 n_0 ，使得对于任何 $n \geq n_0$ ，有 $n^x < 2^n$ 。
- 5) 对任意实数 y ，存在某个 n_0 ，使得对于任何 $n \geq n_0$ ，有

$$n^x (\log n)^y < n^{x+\varepsilon}$$

— 例:

根据定理1.2, 很容易得出:

$$n^3 + n^2 \log n = O(n^3) ;$$

$$n^4 + n^{2.5} \log^{20} n = O(n^4) ;$$

$$2^n n^4 \log^3 n + 2^n n^5 / \log^3 n = O(2^n n^5) .$$

定理1.3: 设 $d(n)$ 、 $e(n)$ 、 $f(n)$ 和 $g(n)$ 是将非负整数映射到非负实数的函数，则

(1) 如果 $d(n)$ 是 $O(f(n))$ ，那么对于任何常数 $a>0$ ， $ad(n)$ 是 $O(f(n))$ ；

(2) 如果 $d(n)$ 是 $O(f(n))$ ， $e(n)$ 是 $O(g(n))$ ，那么 $d(n)+e(n)$ 是 $O(f(n)+g(n))$ ；

(3) 如果 $d(n)$ 是 $O(f(n))$ ， $e(n)$ 是 $O(g(n))$ ，那么 $d(n)e(n)$ 是 $O(f(n)g(n))$ ；

(4) 对于任意固定的 $x>0$ 和 $a>1$ ， n^x 是 $O(a^n)$ ；

(5) 对于任意固定的 $x>0$ ， $\log n^x$ 是 $O(\log n)$ ；

(6) 对于任意固定的常数 $x>0$ 和 $y>0$ ， $\log^x n$ 是 $O(n^y)$ ；

例： $2n^3+4n^2\log n=O(n^3)$

证明： $\log n = O(n)$ 规则6

$4n^2\log n = O(4n^3)$ 规则3

$2n^3+4n^2\log n = O(2n^3+4n^3)$ 规则2

$2n^3+4n^3 = O(n^3)$ 规则1

所以， $2n^3+4n^2\log n = O(n^3)$

3.3 标准记号与常用函数（自学）

需要熟悉一些常用的数学函数和记号

1. Monotonicity（单调性）

- * A function $f(n)$ is *monotonically increasing*（单调递增） if $m \leq n$
implies $f(m) \leq f(n)$.
- * A function $f(n)$ is *monotonically decreasing*（单调递减） if $m \leq n$
implies $f(m) \geq f(n)$.
- * A function $f(n)$ is *strictly increasing*（严格递增）
if $m < n$ implies $f(m) < f(n)$
- * A function $f(n)$ is *strictly decreasing*（严格递减）
if $m < n$ implies $f(m) > f(n)$.

2. Floors and ceilings (向下取整和向上取整)

- * For all real x ,

$$x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$$

- * For any integer n ,

$$\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$$

- * for any real number $x \geq 0$ and integers $a, b > 0$,

$$\begin{aligned} \left\lceil \frac{\lfloor x/a \rfloor}{b} \right\rceil &= \left\lceil \frac{x}{ab} \right\rceil, & \left\lceil \frac{a}{b} \right\rceil &\leq \frac{a + (b - 1)}{b}, \\ \left\lfloor \frac{\lceil x/a \rceil}{b} \right\rfloor &= \left\lfloor \frac{x}{ab} \right\rfloor, & \left\lfloor \frac{a}{b} \right\rfloor &\geq \frac{a - (b - 1)}{b}. \end{aligned}$$

3. Modular arithmetic (模运算)

- * If $(a \bmod n) = (b \bmod n)$, we write $a \equiv b \pmod{n}$ and say that a is *equivalent* to b , modulo n (模 n 时 a 等价于 b , a 、 b 同余).

4. Polynomials (多项式)

- * Given a nonnegative integer d , a *polynomial in n of degree d* is a function $p(n)$ of the form

$$p(n) = \sum_{i=0}^d a_i n^i$$

- * where the constants a_0, a_1, \dots, a_d are the *coefficients* of the polynomial and $a_d \neq 0$.

$$p(n) = \sum_{i=0}^d a_i n^i$$

- * A polynomial is asymptotically positive(渐进为正) if and only if $a_d > 0$.
- * For an asymptotically positive polynomial $p(n)$ of degree d , we have $p(n) = \Theta(n^d)$.
- * $f(n)$ is *polynomially bounded* (多项式有界) if $f(n) = O(n^k)$ for some constant k .

5. Exponentials (指数)

*For all real $a > 0$, m , and n , we have the following identities:

$$\begin{aligned}a^0 &= 1, \\a^1 &= a, \\a^{-1} &= 1/a, \\(a^m)^n &= a^{mn}, \\(a^m)^n &= (a^n)^m, \\a^m a^n &= a^{m+n}.\end{aligned}$$

*For all real constants a and b such that $a > 1$,

$$\lim_{n \rightarrow \infty} \frac{n^b}{a^n} = 0,$$

- from which we can conclude that $n^b = o(a^n)$.
- That is any exponential function with a base strictly greater than 1 grows faster than any polynomial function.

6. Logarithms (对数)

$$\begin{aligned}\lg n &= \log_2 n && \text{(binary logarithm) ,} \\ \ln n &= \log_e n && \text{(natural logarithm) ,} \\ \lg^k n &= (\lg n)^k && \text{(exponentiation) ,} \\ \lg \lg n &= \lg(\lg n) && \text{(composition) .}\end{aligned}$$

For all real $a > 0$, $b > 0$, $c > 0$, and n ,

$$\begin{aligned}a &= b^{\log_b a} , \\ \log_c(ab) &= \log_c a + \log_c b , \\ \log_b a^n &= n \log_b a , \\ \log_b a &= \frac{\log_c a}{\log_c b} , \\ \log_b(1/a) &= -\log_b a , \\ \log_b a &= \frac{1}{\log_a b} , \\ a^{\log_b c} &= c^{\log_b a} ,\end{aligned}$$

where, in each equation above, logarithm bases are not 1.

7. Factorials (阶乘)

The notation $n!$ (read “ n factorial”) is defined for integers $n \geq 0$ as

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ n \cdot (n-1)! & \text{if } n > 0. \end{cases}$$

Thus, $n! = 1 \cdot 2 \cdot 3 \cdots n$.

- * A weak upper bound on the factorial function is $n! \leq n^n$.
- * *Stirling's approximation*(斯特林近似公式)

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

- * Can prove more:
$$\begin{aligned} n! &= o(n^n), \\ n! &= \omega(2^n), \\ \lg(n!) &= \Theta(n \lg n) \end{aligned}$$

课外阅读：

算法导论（3rd）：3.1渐进记号

其他小节自行阅读

* 作业：

3.1-5 证明定理 3.1。