算法设计与分析

Computer Algorithm Design & Analysis

2025.10

王多强

QQ：1097412466

群名称：算法-2025秋

群　号：1067607721

加入QQ群：

群名称：算法-2025秋

群　号：1067607721

功　能：答疑、通知

加入微助教：

课堂名称：算法-2025秋

课堂编号：QJ864

功　　能：**签到**、课件资料

- 算法被公认为是计算机科学的基石。
- 在信息技术高速发展的今天，算法无处不在，小到加减乘除、大到大数据处理、AI等，都体现出了算法的强大力量。
- 学习算法具有重要现实意义。

关于《算法》教学

课程核心：

介绍算法的基本理论、方法和技术，奠定算法设计的基础，培养计算思维和工程实践能力。

教学目标：

- 掌握算法设计与分析的基本理论和方法
- 学习算法实现方面的技术、技巧，培养实践动手能力
- 培养独立思考和创新能力，培养分析、解决实际工程应用问题的能力。

本课程需要的基础

- 数据结构 √
- 程序设计语言（C/C++）：结构化设计 √
- 一定的数学基础：高数、离散、概率 √
- 一些背景知识：操作系统、编译

授课形式:

- 课堂教学：（√）
- 课堂讨论：
- 上机实践：实验，交实验报告

成绩评定：

考试：闭卷，70%

平时成绩：平时作业，30%

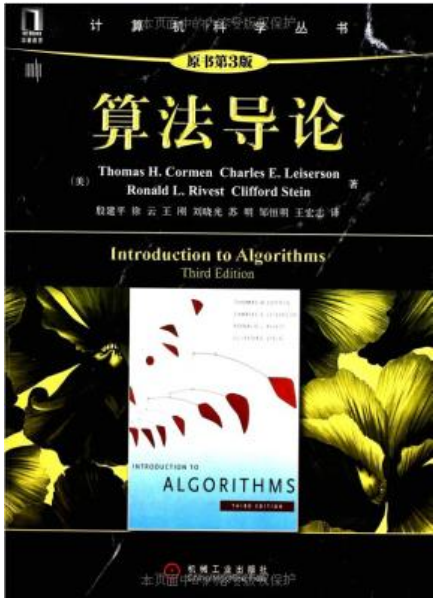
注：另有《算法设计与分析实践》，是理论课的配套实验课，单独记成绩，24学时/1.5学分。

主要参考书

- **Introduction to algorithms（3rd）**

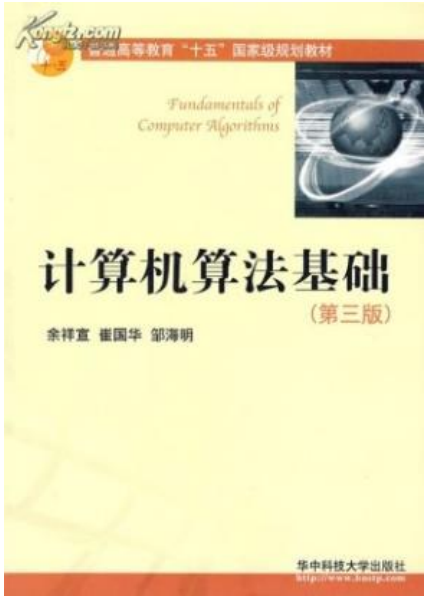
  Thomas H. Cormen,etc.,

  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

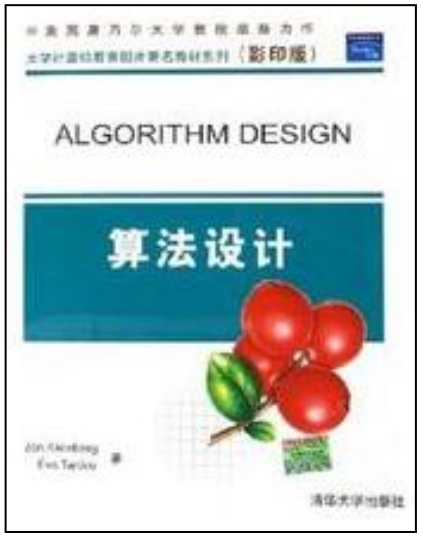


推荐阅读

- **Algorithm Design**

  Jon Kleinberg，Eva Tardos etc.

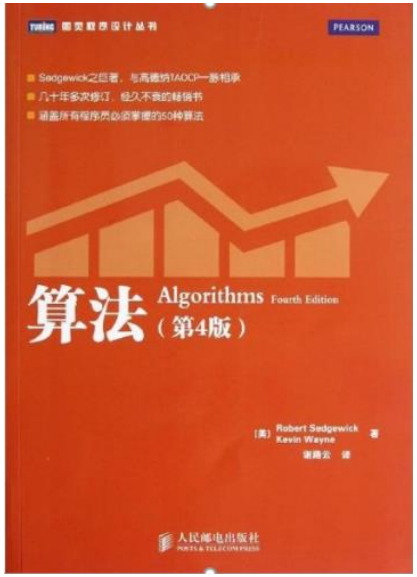
  Cornell University



- **Algorithms**

  Robert Sedgewick / Kevin

  Princeton University



第一讲：算法基础

- **Chapter 1** The Role of Algorithms in Computing
- **Chapter 2** Getting Started
- **Chapter 3** Growth of Functions

Chapter 1

The Role of Algorithms in Computing

算法在计算中的作用

1、算法的基本概念

非形式地说，算法就是经过精心设计的**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过程的描述。**

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 $< a_1, a_2, ..., a_n >$.

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

- **算　　法**：如冒泡排序、插入排序、归并排序等，用不同的规则具体实现一个计算过程完成无序到有序的转换。

解决同一个问题会有不同的算法，虽然最后的结果一般是一样的，但它们的计算规则和性能是不同的。

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。从内容上看，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**。

对算法的理解

- 算法由**运算**组成：包括算术、逻辑、关系、赋值、**过程调用**等各种形式的运算；
- **特殊性**：不同问题有不同的算法，没有能够求解所有问题的万能算法；
- **有穷性**：不仅书写上规则/运算/语句/步骤数量有穷（静态），算法的执行过程在时间上也是有限的（动态）。

2、什么叫做“**正确的算法**”？

若对每个**正确的输入**实例，算法都以**正确的输出**停机，则称该算法是**正确**的，并称一个正确的算法正确地解决了给定的问题。

与正确的算法相反，**不正确的算法**对某些输入实例可能根本不停机，或者可以给出一个回答然后停机，但结果与期望不符甚至相反。

- 需要注意的是，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，也有可能是有用和有效的（如求解NP问题的近似算法）。
- 但这里我们主要关心正确的算法的设计。

4、设计算法，关心两方面的内容

（1）**算法的正确性**：即能否得到正确答案，主要在于计算规则是否正确；

（2）**算法的性能**：用了多少时间和空间解决问题，是算法效率方面的问题。

通常，我们总希望用尽量少的时间和空间解决问题，但不同算法的时空性能不一样，所以我们要分析比较不同算法的性能。

如排序问题中，**插入排序 Vs. 归并排序**：

- 插入排序的时间复杂度是 $O(n^2)$，归并排序的时间复杂度是 $O(nlogn)$。如果n=1000，二者时间理论上相差近100 倍。
- 但同时，插入排序的空间复杂度是 $O(1)$，归并排序的空间复杂度是 $O(n)$。

算法效率的差异往往比运行程序的软硬件环境造成的差异更大（详见《算法导论》P6 对比插入排序和归并排序的分析）：

如插入排序的程序，即使用快**1000**倍的计算机运行，但由于算法本身性能低下，相比在“慢速”计算机上运行的归并排序程序，还要慢**20**倍。

随着计算机求解问题的规模越来越大，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。

多懂点算法知识，就可以更快地做事情，从而可以做更多的事情，也会做得更好一些。

Chapter 2

Getting Started

算法基础

- 本章的主要知识点：
  - 《算法导论》中描述算法的伪代码
  - 算法正确性证明
  - 算法分析基础

1、描述算法的方法

（1）自然语言

（2）实际的程序设计语言（如 C语言、Python等）

（3）流程图

（4）伪代码

2、《算法导论》中采用的一种伪代码描述形式

伪代码（*Pseudocode*），书写类似于 *C/C++*、*Java* 等实际语言，主要用于描述算法的设计思想和计算步骤，但**不是真实代码，**不能放在实际的计算机上运行。

伪代码没有固定的标准，《算法导论》的伪代码“规范”见 **P11**。

如：插入排序算法的伪代码描述。

(1) 排序问题的问题描述：

Input: A sequence of n numbers $< a_1, a_2, ..., a_n >$.

Output: A permutation (reordering) $< a'_1, a'_2, ..., a'_n >$ of the input sequence such that $a'_1 \le a'_2 \le ... \le a'_n$.

(2) 插入排序算法的基本思想（回顾数据结构的相关内容）

(3) 插入排序的伪代码描述：

```
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
```

- 关注格式，用缩进表示程序块
- 变量可以没有类型说明
- 语法结构类似实际语言

设 $A = < 5, 2, 4, 6, 1, 3 >$， **INSERTION-SORT** 在 A 上的排序过程如图所示：

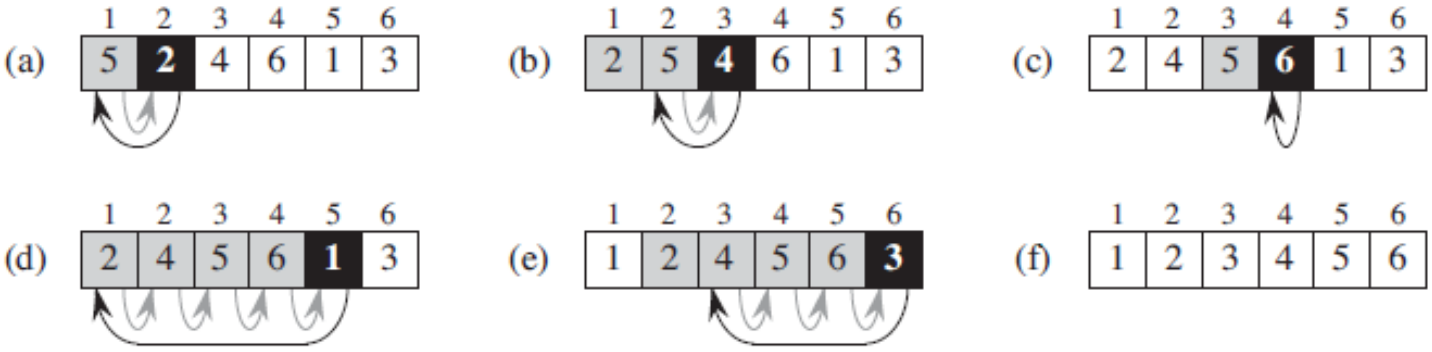


Figure 2.2 The operation of INSERTION-SORT on the array $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. **(a)–(e)** The iterations of the **for** loop of lines 1–8. In each iteration, the black rectangle 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.

自行阅读P9~P10相关内容。

2、使用循环不变式证明循环的正确性

（1）循环不变式（*Loop invariants*）

以INSERTION-SORT为例，在其 *for* 循环中，循环变量为 *j*。循环过程有这样一个性质：*j* 是当前正要被插入到序列中的元素所在的位置下标，*j* 之前的子数组 *A*[1~*j*-1] 是已经被排好序的子序列。这一性质，在 *j* 被赋予初值2、首次进入循环之前成立，以及其后每次循环之后（ *j* 加了1后）、进入下一次循环之前也成立。

◆ 把这种在第一次进入循环之前成立、并以后每次循环之后还成立的性质称为**循环不变性质**或**循环不变关系**、**循环不变式**。

- 插入排序 *for* 循环的循环不变式可再具体描述为：

在第1~8行的 *for* 循环的每次迭代开始时，**子数组 $A[1\sim j\text{-}1]$ 由原来在 $A[1\sim j\text{-}1]$ 中的元素组成，且已按序排列**。

```
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
```

- **可以利用循环不变关系证明循环的正确性。**

分三步走：

① **初始化**：证明初始状态时循环不变式成立，即证明循环不变式在循环开始之前为真；

② **保　持**：证明每次循环之后、下一次循环开始之前循环不变式仍为真；

③ **终　止**：证明循环可以在有限次迭代之后终止。

其中：

- 第①和②步类似于**数学归纳法**的证明策略；
- 第③步是保证算法可以**终止**；
- 如果① ~ ③都满足，则说明一个循环过程正确。

（为什么？有理论证明，自学：**程序的逻辑**）

利用循环不变关系证明插入排序的正确性。

（1）**初始化**：（即证明循环不变式在循环开始之前为真）

第一次循环之前，$j=2$，子数组 $A[1..j-1]$ 实际上只有一个元素，即 $A[1]$，且这个 $A[1]$ 是 A 中原来的元素。所以表明第一次循环之前循环不变式成立 —— **初始状态成立。**

```
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
```

```
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
```

（2）**保持**：（即证明每次循环之后循环不变式仍为真）

观察 ***for*** 循环体，可以看到4~7行是将***A*[*j*-1]**、***A*[*j*-2]**、***A*[*j*-3]**…依次向右移动一个位置（***while***循环的正确性这里不另行证明），直到找到对当前 ***A*[*j*]** 而言适当的位置，之后在第8行将 ***A*[*j*]** 插入该位置。

这时子数组 ***A*[1..*j*]** 显然是由原来在 ***A*[1..*j*]** 中的元素组成，且已按序排列。再之后，执行 ***j* += 1**（下次循环开始之前的状态），此时原来的“***A*[1..*j*]**”变成了新的 ***A*[1..*j*-1]**。故循环不变式依然成立。

(3) **终止**：（即循环执行有限次迭代后终止）

可以看到，每次循环后 j 都加1，而循环的终止条件是 $j > n$（即$A.length$），所以必有在 n-1 次循环后 $j == n+1$，循环终止。

注：此时，循环不变式依然成立（即 $A[1..n]$ 是由原A 中的 n 个元素组成且已排序）。

```
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
```

由此，上述三点都成立，则根据利用**循环不变式**证明一个循环正确的证明策略可得插入排序算法的 *for* 循环过程是正确的。

进而插入排序算法也是正确的。准确地说，上述证明过程仅证明了算法中 *for* 循环的正确性，但 *for* 循环是插入排序过程的主体，所以也就说整个算法是正确的。

2.2 分析算法基础

1. 分析算法的目的

- **算法选择的需要**：对同一个问题可以设计不同的算法，不同的算法对时间和空间的需求是不同的。通过分析算法可以了解算法的时空性能，从而比较这些算法性能的好坏，**选择好的算法求解问题**，避免无益的人力和物力浪费。

- **算法优化的需要**：通过对算法分析，可以对算法作深入了解，发现其中可以改进的地方，从而可以**进一步优化算法**，让其更好地工作。

2. 重要的假设和约定

1）计算机模型的假设（形式理论模型，而非实际的计算机）

- ***RAM*模型（*random-access machine*，随机访问机）**

在 ***RAM*** 模型中，指令一条一条被取出和执行，不考虑并发操作，并且可以在固定的时间内存取任意数据单元。（关于 ***RAM*** 的讨论参考P13相关内容）

- **通常理解：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”**，并能以相同的时间和方式存取内存中的任意数据单元

区分计算机的理论模型与实际的计算机系统

2）计算的约定

一般不考虑计算机硬件、程序设计语言、机器指令等方面存在的现实差异和影响，**对算法中使用的运算进行抽象**，将**最基本的运算**，如算术运算、关系运算、过程调用等，都**抽象为可以在单位时间**内完成的运算。而对于复杂运算，如字符串比较，分解为基本的字符比较运算的序列。从而：

算法的计算时间等价于算法中基本运算的执行次数之和。

（将任何基本运算的一次执行视为可以在1个单位时间内完成）

如何分析算法的计算时间?

分析算法中使用了哪些运算，统计每种运算的执行次数，然后将所有运算的执行次数加起来，就得到算法计算时间的抽象表示：$T(n) = \sum f_i$

- n 是问题的规模，如输入的数据的个数。
- i 指示算法中的某个运算。
- f_i 是 i 运算的执行次数，称为i 运算的**频率计数**。

即，将算法的计算时间表示成算法**所用运算执行次数的和**。

3）测试数据集

- 算法的执行情况与输入数据的关系：

（1）**与输入数据的规模相关：**一般规模越大，执行时间越长。

（2）在不同的数据配置上，同一算法有不同的表现，可分为**最好情况**、**最坏情况**和**平均情况**等情况讨论。

如：插入排序的最好、最坏和平均计算时间是 $O(n)$、$O(n^2)$、$O(n^2)$。

编制不同的数据配置，分析算法的**最好、最坏、平均**情况下的工作情况是算法分析的一项重要工作。

3. 算法分析

- 算法分析的目标是求取算法时间及空间的**复杂度限界函数**。

  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 $\Theta(nlogn)$。

- **怎么获取算法复杂度的特征函数?**
  - 以下以时间分析为例进行说明。

注：首先要明确，这些特征函数是通过分析算法的控制逻辑得来的，是对算法所用**运算执行次数的统计结果**。与使用的程序设计语言和计算机硬件无关。

如何进行**时间**分析?

1）统计算法中各类运算的执行次数

（1）统计对象：运算

基本运算：具有**单位执行时间**的运算。

复合运算：具有固定执行时间的一条语句或一组语句组成的程序块，甚至是一个函数等。只要具有固定执行时间，其执行一次的时间都可视为单位时间t_0。

（2）统计内容：**频率计数**，即算法中基本运算或语句、程序块的**执行次数**。

- 顺序结构中的运算/语句执行次数计为1
- 循环结构中的运算/语句执行次数等于被循环执行的次数

例：简单结构中基本语句执行次数的统计

```
x←x+y
```

(a)

```
for i ←1 to n do
    x ← x + y
repeat
```

(b)

```
for i ←1 to n do
    for j ←1 to n do
        x ← x +y
    repeat
repeat
```

(c)

分析：

(a)： **x←x+y**执行了**1**次

(b)： **x←x+y**执行了**n**次

(c)： **x←x+y**执行了$\mathbf{n^2}$次

思考：

```
for i ←1 to n do
    for j ←i to n do
        x ← x +y
    repeat
repeat
```

(d)

插入排序时间分析：

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	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**：给出了每行的语句执行一次的时间。这里假定第 i 行语句的执行时间是 c_i，且 c_i 是一个常量（固定值）。
- ***times* 列**：给出了每行的语句被执行的总次数。其中，t_j 表示对当前的 j，第5行执行 *while* 循环测试的次数。

- **整个算法的执行时间是所有语句的执行时间之和。**
  - 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：

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

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	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	$n-1$

2）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，有最好、最坏、平均等执行情况。

如：**INSERTION-SORT**

- **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 *while* 循环的循环体都不会执行。

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	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	$n-1$

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	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	$n-1$

所以最好情况的运行时间为：

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

所以最好情况下， **INSERTION-SORT**的执行时间为 n 的**线性函数**，即具有 $an+b$ 的形式。

时间复杂度表示是$O(n)$

- **最坏情况**：初始数组是反向排序的。

此时对每一次 *for* 循环，内部的 *while* 循环都执行最多次数的测试（即 j -1 次）。

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	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	$n-1$

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	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	$n-1$

- **最坏情况的运行时间为：**

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

此时，执行时间的表达式为 n 的**二次函数**，即具有 an^2+bn+c 的形式。 时间复杂度表示是$O(n^2)$

- 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）。

如插入排序时间分析：

统计各条语句（复合运算）的执行次数，然后换算时间并求和：

```
INSERTION-SORT(A)                                   cost    times
1  for j = 2 to A.length                            c1      n
2      key = A[j]                                   c2      n − 1
3      // Insert A[j] into the sorted
           sequence A[1 . . j − 1].                 0       n − 1
4      i = j − 1                                    c4      n − 1
5      while i > 0 and A[i] > key                   c5      ∑_{j=2}^{n} t_j
6          A[i + 1] = A[i]                          c6      ∑_{j=2}^{n} (t_j − 1)
7          i = i − 1                                c7      ∑_{j=2}^{n} (t_j − 1)
8      A[i + 1] = key                               c8      n − 1
```

最好情况分析：

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) . \\ &= an+b \quad O(n) \end{aligned}$$

最坏情况分析：

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

$T(n)$：频率计数函数表达式，复杂表达式。

如何得出时间复杂度限界函数的表示 —— 简单表达式？

平均情况又该如何分析?

3）时间复杂度限界函数的获取

函数表达式中最高次项的次数称为函数表达式的**数量级**（或**函数的阶**）

在**频率计数函数表达式**中，这个**数量级**是衡量频率计数**大小**的一种测度，从本质上反映了算法复杂度的高低。

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——形式简单的函数表达式，记为：$g(n)$。

- $g(n)$ 通常是关于n的形式简单的**单项式函数**，如 n^2，$n\log n$等；
- $g(n)$ 通常是对算法中**最复杂的计算部分**分析而来的，代表着最高次项部分。 如，若频率计数函数表达式是 an^2+bn+c，则 $g(n)=n^2$。

- 对于足够大的 n，**低阶项**和**常数项**都不如**最高次项**重要，因此 $g(n)$ 忽略了函数表达式中次数较低的“次要”项，以体现算法复杂性最本质的特征，且忽略常系数。
- **但对于较小的 n 却未必，一般还要分情况讨论。**

如：$T(n) = n^4+10000$

当 $n<10$ 时候，算法时间的主要部分是10000。

2.3 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。

- 整数的算术运算是“能行”的，实数的算术运算可能是“不能行”的

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”。

- **时效性**：实际问题往往都有时间要求，只有在要求的时间内解决问题才是有意义的。

  例：国际象棋（启发）、数值天气预报

基于算法的时效性，只有把在相当有穷步内终止的算法投入到计算机上运行才是实际可行的。

——这就要通过“算法分析”，了解算法性质，给出算法计算时间的一个精确的描述，以衡量算法的执行速度，选择合适的算法有效地解决问题。

2.4 分析结论的证明

当我们有了分析的结论，怎么证明结论是正确的?

常用的方法:

- 直接推导（略）
- 数学归纳法
- 反证法
- 反例法

1）数学归纳法

数学归纳法是用来证明某些与自然数有关的数学命题的一种推理方法，有广泛的应用。

已知最早的使用数学归纳法的证明出现于 ***Francesco Maurolico*** 的 ***Arithmeticorum libri duo*** (1575年)。***Maurolico*** 证明了前 n 个奇数的总和是 n^2。

数学归纳法采用**递推策略**进行命题证明，分为标准的二部分完成：

第一步：基准情形（递归基础、初始情形、平凡情形）

该部分证明定理对于某个（某些）小的值是正确的，一般证明命题在 $n=1$（n_0）时命题成立。这是递推的基础。

第二步：归纳假设和推论的证明

该部分首先假设定理对于直到某个有限数 k 的所有情况都成立，然后使用这个假设证明定理对于下一个值（通常就是 $k+1$）也成立。如果证明了$k+1$正确，则完成整个证明。

完成了上述两步，就可下结论：对任何自然数 n（或 $n\geq n_0$）结论都正确，证毕。

以上两步密切相关，缺一不可。而且证明的关键是 $n=k+1$ 时命题成立的推证，这是无限递推下去的理论依据，它将判定命题的正确性由特殊推广到一般，使命题的正确性突破了有限而达到“无限”。

实例：证斐波那契数 $F_i < (5/3)^i$

这里：$F_0 = 1$，$F_1 = 1$；

$F_i = F_{i-1} + F_{i-2}$，$i \geq 2$

证明：

1. 初始情形

容易验证 $F_1 = 1 < 5/3$、$F_2 = 2 < (5/3)^2$，

所以，初始情形成立。

2. 归纳假设

设定理对于 $i = 1, 2, ..., k$ 都成立，现证明定理对于 $i=k+1$ 时也成立，即 $F_{k+1} < (5/3)^{k+1}$。

根据斐波那契数的定义有，

$$F_{k+1} = F_k + F_{k-1}$$

将归纳假设用于等号右边，有

$$F_{k+1} < (5/3)^k + (5/3)^{k-1}$$

$$< (3/5)(5/3)^{k+1} + (3/5)^2(5/3)^{k+1}$$

$$< (3/5+9/25)(5/3)^{k+1}$$

$$< (24/25)(5/3)^{k+1}$$

$$< (5/3)^{k+1}$$

$\therefore n=k+1$ 时结论成立。

由（1）、（2）知，定理得证。

2）反证法

反证法就是通过**假设定理不成立**，然后证明该假设将导致某个已知的性质不成立，从而证明假设是错误的而原始命题是正确的。

实例：证明存在无穷多个素数

反证法：假设定理不成立。则将存在最大的素数 P_k。

令 $P_1, P_2, \ldots, P_k$ 是依次排列的所有素数

令：$N = P_1 P_2 \ldots\ldots P_k + 1$

显然 N 是比 P_k 大的数。

思考：**N是素数吗?**

因为每个整数，或者是素数，或者是素数的乘积，而 N 均不能被$P_1, P_2, \ldots, P_k$ 整除（因为用$P_1, P_2, \ldots, P_k$ 中的任何一个除 N 总有余数1），所以 N 是素数。

而 $N>P_k$，故，与 P_k 是最大的素数的假设相矛盾。

∴ 假设不成立，原定理得证。

3）反例法

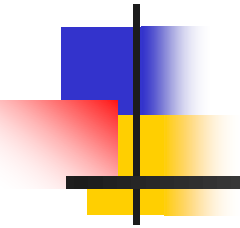
反例法就是举一组具体的数据（算例），带入定理给出的表达式，证明该数据计算出来的结果与预想结果不符，从而证明定理的结论有错。

特别的应用：证明定理不成立最直接的方法就是举出一个反例。

例：斐波那契数 $F_k \le k^2$ 是否成立？

举反例：令 $k=11$

则：$F_{11} = 144 > 11^2$，所以原命题不成立。



Chapter 3

Growth of Functions

函数的增长

本章研究算法的**渐近效率**，给出算法的运行时间 T 随问题规模 n 的变化关系，给出算法复杂性**限界函数**的定义及其**渐近记号**，讨论限界函数的相关性质。

渐近复杂性

设 $T(n)$ 是算法 A 的（时间或空间）复杂性函数，$T(n)$ 是渐近正的，当 n 单调递增趋于 ∞ 时，$T(n)$ 也单调递增趋于 ∞。

如果存在函数 $G(n)$，使得当 $n \to \infty$ 时有

$$(T(n)-G(n)) / T(n) \to 0,$$

则称 $G(n)$ 是 $T(n)$ 当 $n \to \infty$ 时的**渐近形态**，也称 $G(n)$ 是算法 A 当 $n \to \infty$ 的**渐近复杂性**。

这里，记 $G(n)$ 是 $T(n)$ 当 $n \to \infty$ 时的**渐近表达式**，并且这个表达式就是 $T(n)$ 中略去**低阶项**所留下的**主项**。

3.1 限界函数的定义

表示算法复杂度的**限界函数**常用的有三个：

上界函数、下界函数、紧确界函数

对应的渐近记号：　O　　Ω　　Θ

相关定义如下：

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：表示问题规模的某种测度，比如数据的多少。

$f(n)$：是与机器及语言有关的量，视为实际运行时间。

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

1. O 记号的定义（上界函数）

$O(g(n))$ 表示以下函数的集合：

$$O(g(n)) = \{ f(n): \text{存在正常数 } c \text{ 和 } n_0，\text{当 } n \geq n_0，\text{有 } 0 \leq f(n) \leq cg(n) \}.$$

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：如果一个算法用 n **值不变**的**同一类数据**（规模相等，性质相同）在某台机器上运行，该算法所用的时间**总不大于** $|g(n)|$ 的一个常数倍。

- ***O*** 记号给出的是**渐近上界**，称为**上界函数**（*upper bound*）
  - 上界函数代表了算法在**最坏情况下**的时间复杂度。
- **紧确的上界**

在确定上界函数时，总是试图找**数量级（函数的阶）最小**的 $g(n)$ 作为 $f(n)$ 的上界函数——**紧确上界**(*tight upper bound*)。

如：若：$3n+2 = O(n^2)$ ， $O(n^2)$ 是**松散**的上界；

而：$3n+2 = O(n)$ ， $O(n)$ 是**紧确**的上界。

2. Ω 记号的定义（下界函数）

$\Omega(g(n))$ 表示以下函数的集合：

$$\Omega(g(n)) = \{ f(n): \text{存在正常数 } c \text{ 和 } n_0，\text{当 } n \geq n_0，\text{有 } 0 \leq cg(n) \leq f(n) \}.$$

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：如果一个算法用 n 值不变的同一类数据在某台机器上运行，该算法所用的时间**总不小于** $|g(n)|$ 的一个常数倍。

- Ω 记号给出一个**渐近下界**，称为**下界函数**（*lower bound*）。
  - 下界函数一般代表了算法在**最好情况下**的时间复杂度。
- **紧确的下界**

在确定下界函数时，总是试图找**数量级（函数的阶）最大**的 $g(n)$ 作为 $f(n)$ 的下界函数——**紧确下界**(*tight lower bound*)。

如：若：$3n^2+2=\Omega(n)$，$\Omega(n)$ 是**松散**的下界；

而：$3n^2+2=\Omega(n^2)$，$\Omega(n^2)$ 是**紧确**的下界。

3. Θ 记号的定义（渐近紧确界）

$\Theta(g(n))$ 表示以下函数的集合：

$$\Theta(g(n)) = \{ f(n): \text{存在正常数 } c_1,\ c_2 \text{和 } n_0,\ \text{当 } n \geq n_0,\ \text{有}$$

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \}.$$

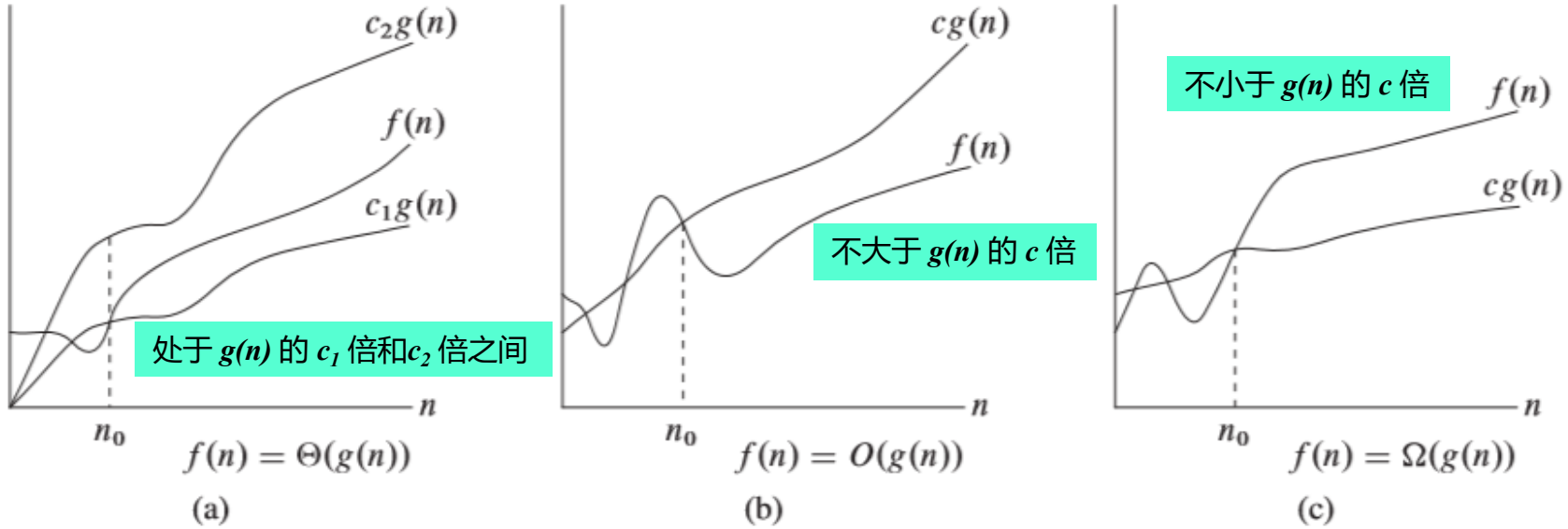
若 $f(n)$ 和 $g(n)$ 满足以上关系，记为：$f(n) \in \Theta(g(n))$，表示 $f(n)$ 是 $\Theta(g(n))$ 的一员。并通常记作：

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

含义：如果一个算法用 n 值不变的同一类数据在某台机器上运行，该算法所用的时间**既不小于** $|g(n)|$ 的一个常数倍，**也不大于** $|g(n)|$ 的一个常数倍 —— $g(n)$ **既是** $f(n)$ **的下界，也是** $f(n)$ **的上界。**

- Θ 记号给出的是**渐近紧确界**(*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 $f(n) = O(g(n))$ 。



O、Θ、Ω 所反映的 $f(n)$ 和 $g(n)$之间函数关系的示意图

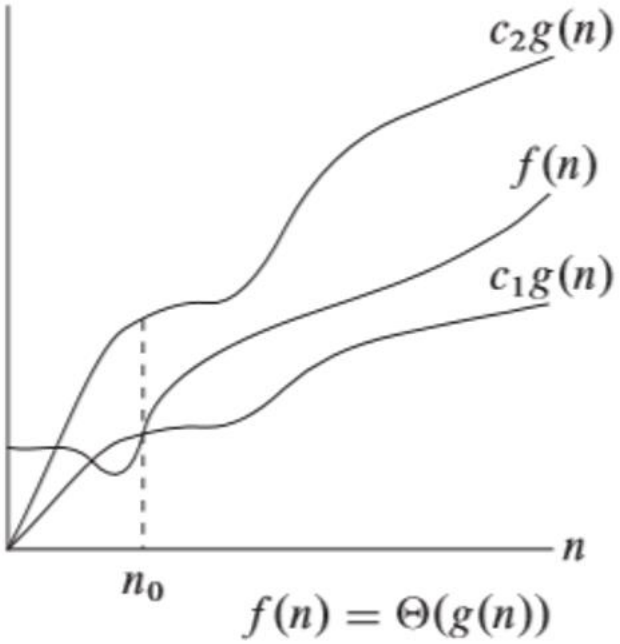
$f(n) = O(g(n))$：在 $n \geq n_0$ 时，$f(n)$ 的函数曲线处于 $cg(n)$ 的下方，表示 $f(n)$ 不大于 $g(n)$ 的 c 倍；

$f(n) = \Omega(g(n))$：在 $n \geq n_0$ 时，$f(n)$ 的函数曲线处于$cg(n)$ 的上方，表示 $f(n)$ 不小于 $g(n)$ 的 c 倍；

$f(n) = \Theta(g(n))$：在 $n \geq n_0$ 时，$f(n)$ 的函数曲线夹在 $c_1g(n)$ 和 $c_2g(n)$ 之间，表示 $f(n)$ 既不小于 $g(n)$ 的 c_1 倍，也不大于 $g(n)$ 的 c_2 倍。

并且，从 $f(n)$ 和 $g(n)$ 的函数曲线来看，$f(n) = \Theta(g(n))$ 的关系可视为一种**等价关系**：

当 $n \geq n_0$ 时，$f(n)$ 和 $g(n)$ 具有**相同的变化趋势**。



例：证明 $n^2/2-3n = \Theta(n^2)$

思路：从定义出发——根据 Θ 的定义，仅需确定正常数 c_1, c_2 和 n_0，

以使得对所有的 $n \geq n_0$，有：$c_1 n^2 \leq \left(\frac{1}{2}\right) n^2 - 3n \leq c_2 n^2$

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

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

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

注：一般情况下，c_1, c_2 和 n_0 可能还有**其它常量可选**，但根据 Θ 的定义，只要找到一组满足关系的取值即可，不用讨论所有可能的取值。

再如：证明 $6n^3 \neq \Theta(n^2)$

采用**反证法**：假设 $6n^3 = \Theta(n^2)$，下面证明该假设不成立。

若假设成立，则至少存在 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 就有： $n \leq c_2 / 6$，

由于 n **是变量**，c_2 **是常量**，所以对变量 n，不可能存在一个常量c_2，使得对 n 的任意取值小于等于 $c_2 / 6$，所以该不等式不可能成立，即假设不成立。 证毕 ■

关于渐近记号的进一步说明：

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

- 这里的等号不是“相等”含义，$f(n)$ 与 $g(n)$ 不是等价关系。

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

- 因为任意常量都可看做是 n 的一个0阶多项式，所以可以把一个**常量**表示成 n 的**0阶函数式**：$\Theta(n^0)$，即 $\Theta(1)$。
- 通常用 $\Theta(1)$ 表示一个算法具有**常量计算时间**，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：$\Theta(1)$ 本身有“轻微活用”的意思，因为该表达式本身没有指出是**什么量**趋于无穷（也就是没体现出n的存在，见P27的说明）

4. o, ω 记号

O、Ω 给出的渐近上界或下界可能是渐近紧确的，也可能不是。而 o、ω 记号专门用来表示一种**非渐近紧确**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 $n \geq n_0$，

若 $|f(n)| \leq c|g(n)|$，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 $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} = 0$

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 $n \geq n_0$，

若 $c|g(n)| \leq |f(n)|$，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 $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

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

O 和 o 的区别

$$O: f(n) = O(g(n)) \Leftrightarrow \exists c, n_0: (n \geq n_0 \Rightarrow f(n) \leq cg(n))$$

$$o: f(n) = o(g(n)) \Leftrightarrow \forall c: (\exists n_0: n \geq n_0 \Rightarrow f(n) \leq cg(n))$$

Ω 和 ω 的区别

$$\Omega: f(n) = \Omega(g(n)) \Leftrightarrow \exists c, n_0: (n \geq n_0 \Rightarrow cg(n) \leq f(n))$$

$$\omega: f(n) = \omega(g(n)) \Leftrightarrow \forall c: (\exists n_0: n \geq n_0 \Rightarrow cg(n) \leq f(n))$$

3.2 限界函数的性质

① **传递性**（*Transitivity*）：

$$f(n)=\Theta(g(n))\ and\ g(n)=\Theta(h(n))\Rightarrow f(n)=\Theta(h(n))$$

$$f(n)=O(g(n))\ and\ g(n)=O(h(n))\Rightarrow f(n)=O(h(n))$$

$$f(n)=\Omega(g(n))\ and\ g(n)=\Omega(h(n))\Rightarrow f(n)=\Omega(h(n))$$

$$f(n)=o(g(n))\ and\ g(n)=o(h(n))\Rightarrow f(n)=o(h(n))$$

$$f(n)=\omega(g(n))\ and\ g(n)=\omega(h(n))\Rightarrow f(n)=\omega(h(n))$$

证明： 如证：$f(n)=O(g(n))\ and\ g(n)=O(h(n))\Rightarrow f(n)=O(h(n))$

从**定义**出发进行证明：

对 $f(n)=O(g(n))$：存在正常数 c_1 和 n_{01}，当 $n\geq n_{01}$，有 $0\leq f(n)\leq c_1g(n)$；

同理对 $g(n)=O(h(n))$：存在正常数 c_2 和 n_{02}，当 $n\geq n_{02}$，有 $0\leq g(n)\leq c_2h(n)$;

根据 $\leq$ **关系的传递性**，就有，当 $n\geq \max(n_{01},n_{02})$ 时有：

$$0\leq f(n)\leq c_1g(n)\leq c_1c_2h(n),$$

令 $c=c_1c_2$，则有 $0\leq f(n)\leq ch(n)$。根据 O 的定义，就有 $f(n)=O(h(n))$. 证毕。

② **自反性**（*Reflexivity*）：

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

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

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

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*）：

$$f(n) = O(g(n)) \text{ if and only if } g(n) = \Omega(f(n))$$

$$f(n) = o(g(n)) \text{ if and only if } g(n) = \omega(f(n))$$

以上性质均可从定义出发进行证明。

3.3 相关定理

定理3.1（多项式定理）若 $A(n)=a_m n^m+\ldots+a_1 n+a_0$ 是一个 n 的 m 次多项式，则有 $A(n)=O(n^m)$ 。

即：变量 n 的 m 阶 的多项式与此多项式的最高次项 n^m 同阶。

证明：取 $n_0=1$，当 $n\geq n_0$ 时，有

$$|A(n)| \leq |a_m|n^m+\ldots+|a_1|n+|a_0|$$

$$=(\,|a_m|+|a_{m-1}|/n+\ldots+|a_0|/n^m\,)\,n^m$$

$$\leq(\,|a_m|+|a_{m-1}|+\ldots+|a_0|\,)\,n^m$$

令 $c=|a_m|+|a_{m-1}|+\ldots+|a_0|$，即有 $|A(n)|\leq cn^m$ ，根据 O 的定义就有 $A(n)=O(n^m)$ 。证毕。

定理3.1的应用：

如果一个算法的时间复杂度函数是多项式形式，则其**阶函数**（**复杂度函数**）就可取该多项式的最高次项，且省略系数。

$$A(n) = a_m n^m + \ldots + a_1 n + a_0 \Rightarrow A(n) = O(n^m)$$

理解：根据渐近关系，对于足够大的 n，**低阶项（包括常数项）是无足轻重的。**即当 n 较大时，即使最高阶项的一个很小部分都足以“**支配**”**所有的低阶项。**所以用阶函数表示限界函数时，低阶项和常数项均被忽略。

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：

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

可对比下面形式化证明：

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

可以证明对所有的 $n\geq n_0$，有：

$$0\leq c_1n^2\leq an^2+bn+c\leq c_2n^2$$

一般而言，对任意多项式 $p(n)=\sum_{i=0}^{d}a_in^i$，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3.2**] 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3.2，很容易得出：

$n^3 + n^2 \log n = O(n^3)$；

$n^4 + n^{2.5}\log^{20} n < n^4 + n^3 = O(n^4)$；

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

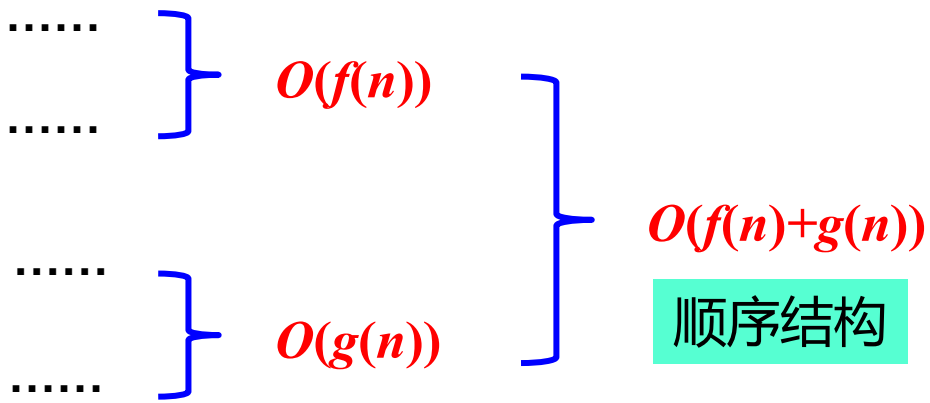
定理3.3：设 $d(n)$、$e(n)$、$f(n)$ 和 $g(n)$ 是将非负整数映射到非负实数的函数，则

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

证明：　$d(n) = O(f(n)) \Rightarrow d(n) \le cf(n) \Rightarrow ad(n) \le acf(n)$

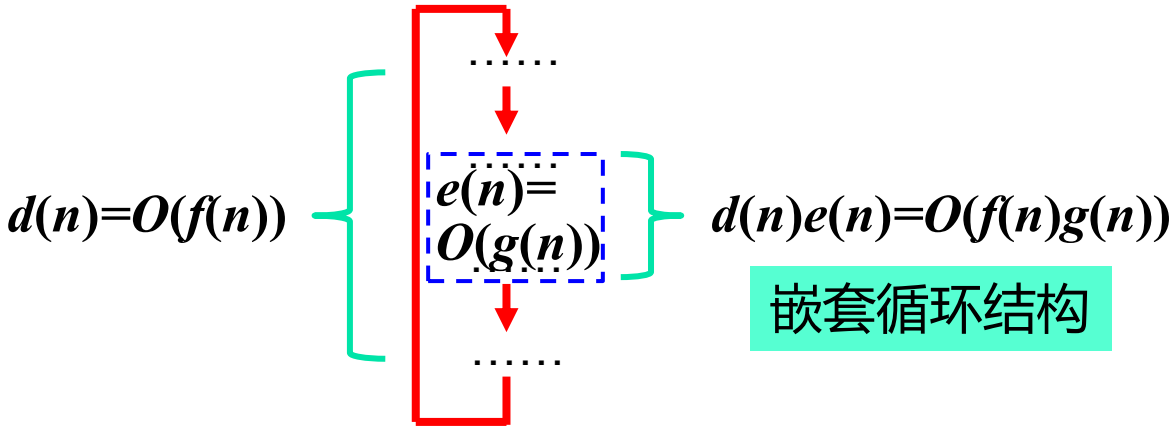
$\Rightarrow ad(n) = O(f(n))$

(2) 如果 $d(n)$ 是 $O(f(n))$，　$e(n)$ 是 $O(g(n))$，那么 $d(n) + e(n)$ 是 $O(f(n) + g(n))$ —— **加法法则**；



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

$O(f(n)g(n))$ —— **乘法法则**；



(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 $O(\log n)$；

$$\log n^x \Rightarrow x\log n \Rightarrow O(\log n)$$

(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，**乘法法则**

$2n^3+4n^2\log n = O(2n^3+4n^3)$ //规则2，**加法法则**

$2n^3+4n^3 = O(n^3)$ //规则1，**去掉常系数**

所以，$2n^3+4n^2\log n = O(n^3)$

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

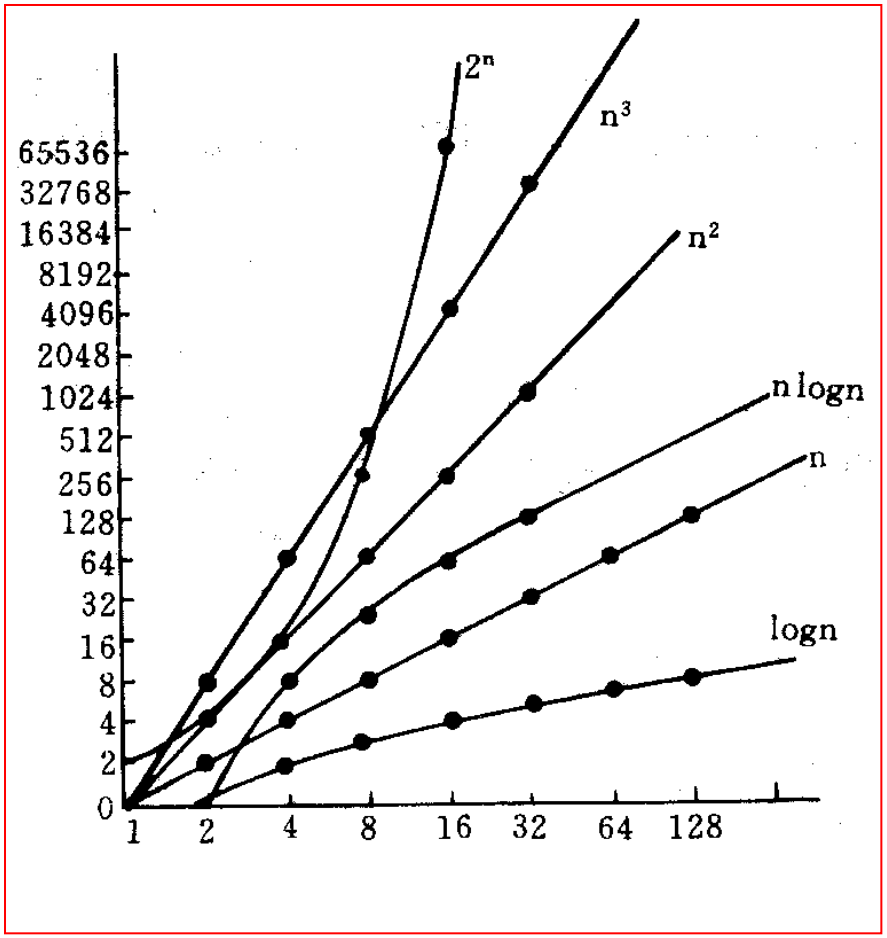
- **指数时间算法**：计算时间用**指数函数**限界的算法。
  - 常见的指数限界函数：

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

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：



复杂度函数的值的比较

表1.1 典型函数的值

$\log n$	n	$n\log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294967296

对算法复杂性的一些认识

- 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 $O(nlogn)$ 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。

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 $m \le n$ implies $f(m) \le f(n)$.

- A function f(n) is ***monotonically decreasing*** ***（单调递减）***

  if $m \le n$ implies $f(m) \ge f(n)$.

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

- for any real number x≥0 and integers a, b > 0,

$$\left\lceil \frac{\lceil x/a \rceil}{b} \right\rceil = \left\lceil \frac{x}{ab} \right\rceil, \qquad \left\lceil \frac{a}{b} \right\rceil \leq \frac{a + (b-1)}{b},$$

$$\left\lfloor \frac{\lfloor x/a \rfloor}{b} \right\rfloor = \left\lfloor \frac{x}{ab} \right\rfloor, \qquad \left\lfloor \frac{a}{b} \right\rfloor \geq \frac{a - (b-1)}{b}.$$

3. Modular arithmetic（模运算）

- If (a mod n) = (b mod 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 $a_0, a_1, \ldots, a_d$ are the ***coefficients*** of the polynomial and $a_d \neq 0$.

- A polynomial is asymptotically positive (渐近为正) 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,

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

  - 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 $n! \leq n^n$.
- ***Stirlings 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：

$$\begin{aligned} n! &= o(n^n)\,, \\ n! &= \omega(2^n)\,, \\ \lg(n!) &= \Theta(n \lg n) \end{aligned}$$

阅读算法导论（3rd）：第1~3章相关内容

作业1：抄写　通过抄写，熟悉伪代码的书写方法

P10，INSERTION-SORT

P19，MERGE-SORT

P17，MERGE

作业2：

1.2-2

Suppose we are comparing implementations of insertion sort and merge sort on the same machine. For inputs of size n, insertion sort runs in $8n^2$ steps, while merge sort runs in $64n \lg n$ steps. For which values of n does insertion sort beat merge sort?

1.2-3

What is the smallest value of n such that an algorithm whose running time is $100n^2$ runs faster than an algorithm whose running time is 2^n on the same machine?

思考题：2.2-2 选择算法