《计算复杂性理论》
第1讲 课程概述

山东师范大学信息科学与工程学院
段会川
2015年9月

目录

课程目标

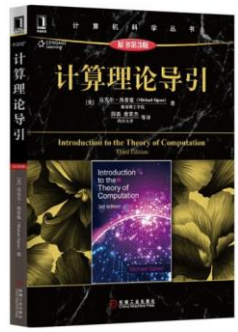
- 算法设计与分析
  - 算法分析基础
  - 典型的基础算法
  - 典型的算法设计方法、实例应用及分析
    - 穷举法、分治法、贪心法、动态规划法、回溯法、分支限界法
- 计算复杂性理论
  - 可计算函数
  - Turing Complete
  - 停机问题
  - NP完全问题

上课效果保证

- 幻灯片讲义
- 笔
- 教材

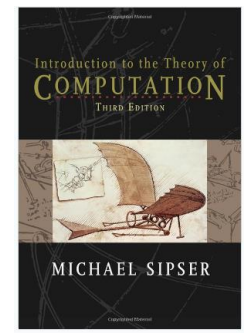
教学参考书

- 计算理论导引(第3版)
- (美) Michael Fredric Sipser.
- Course Technology, 3ed 2012
- 段磊、唐常杰译
  - 四川大学
- 机械工业出版社, 2015.8
- 296页， ¥69.00
- ISBN：9787111499718



教学参考书

- Introduction to the Theory of Computation, 3ed
- Michael Fredric Sipser
  - Dean of Science, MIT
- Course Technology, 2012
- 480pp，$248.95
- ISBN-13: 978-1133187790
- ISBN-10: 113318779X



参考书

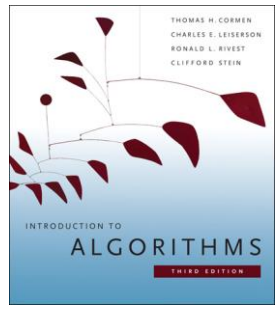
- 《可计算性与计算复杂性导引》
- 张立昂 编著
  - 北京大学软件与微电子学院
- 高等院校计算机专业及专业基础课系列教材
- 北京大学出版社
- 2011年8月
- ￥35.00, 256页

教学参考书

- 算法导论(第3版)
- (美) Thomas H.Cormen, Charles E.Leiserson, Ronald L.Rivest, Clifford Stein.
- MIT Press, 2009
- 殷建平、徐云、王刚、刘晓光、苏明等
  - 国防科技大学
- 机械工业出版社, 2013.1
- 793页， ¥128.00

教学参考书

- Introduction to Algorithms, 3rd Ed
- Thomas H.Cormen,
  - Dartmouth College;
- Charles E.Leiserson,
  - MIT;
- Ronald L.Rivest,
  - MIT;
- Clifford Stein,
  - Columbia University
- MIT Press, 2009
- 1292 pages, $92.00



Authors of Introduction to Algorithms

Algorithm engineering, parallel computing, speeding up computations with high latency

Thomas H. Cormen
Professor and Chair
Department of Computer Science ,
Dartmouth College

Specializing in the theory of parallel computing and distributed computing

Charles E.Leiserson,
Professor of Computer Science and Engineering,
Department of Electrical Engineering and Computer Science,
MIT

Authors of Introduction to Algorithms

Cryptographer, One of the inventors of the prominent RSA algorithm

Ronald L.Rivest,
Andrew and Erna Viterbi Professor of Computer Science, Department of Electrical Engineering and Computer Science,
MIT

design and analysis of algorithms, combinatorial optimization, operations research, network algorithms, scheduling, algorithm engineering and computational biology

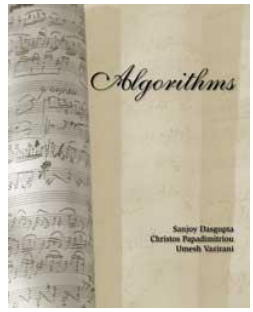
Clifford Stein,
Professor and Chair of the Industrial Engineering and Operations Research Department ,
Columbia University

教学参考书

- 算法概论
- Algorithms
- (美)Sanjoy Dasgupta, Christos H. Papadimitriou, Umesh Vazirani
- 王沛 唐扬斌 刘齐军(译)
  - 国防科大信管学院
- McGraw-Hill , 2006
- 清华大学出版社, 2008
- 345页， ¥39.99

教学参考书

- Sanjoy Dasgupta
  - Department of Computer Science and Engineering,
  - University of California, San Diego
- Christos Papadimitriou
  - Department of Electrical Engineering and Computer Science,
  - University of California at Berkeley
- Umesh Vazirani
  - Computer Science Division,
  - University of California at Berkeley



算法导论—算法的重要地位

- Before there were computers, there were algorithms. But now that there are computers, there are even more algorithms, and algorithms lie at the heart of computing.
- 在没有计算机的时候就已经有了算法。而今我们有了计算机，就有了更多的算法，而且算法处在计算的核心地位。

Cormen, Leiserson, Rivest and Stein
Introduction to Algorithms, 3rd ed., 2009

算法概论—算法的重要地位

- Look around you. Computers and networks are everywhere, enabling an intricate web of complex human activities: education, commerce, entertainment, research, manufacturing, health management, human communication, even war.
- 环视左右，您一定会发现电脑与网络无处不在。它们织就了一张复杂的人类活动图景，包括：教育、商业、娱乐、研究、制造、健康保障、人际交往，甚至战争。

Dasgupta et al.
Algorithms, 2006

算法概论—算法的重要地位

- Of the two main technological underpinnings of this amazing proliferation, one is obvious: the breathtaking pace with which advances in microelectronics and chip design have been bringing us faster and faster hardware.
- 有两项技术基石支撑着这些惊人的爆炸式进展，其中一项显而易见：即带给我们越来越快速硬件的微电子业和芯片设计制造业的飞速发展

Dasgupta et al.
Algorithms, 2006

算法概论—算法的重要地位

- This book tells the story of the other intellectual enterprise that is crucially fueling the computer revolution: efficient algorithms.
- 本书将阐述另一项更富智力性的基石，它是计算机业革命的关键助燃剂，这项技术就是：高效率的算法技术。

Dasgupta et al.
Algorithms, 2006

算法的来临—算法的重要地位

- Two ideas lie gleaming on the jeweler's velvet. The first is the calculus, the second, the algorithm. The calculus and the rich body of mathematical analysis to which it gave rise made modern science possible; but it has been the algorithm that has made possible the modern world.
- 有两种思想，像珠宝商放在天鹅绒上的宝石一样熠熠生辉。一个是微积分，另一个就是算法。微积分以及在微积分基础上建立起来的数学分析体系造就了现代科学，而算法则造就了现代世界。

David Berlinski
The Advent Of The Algorithm, 2000

目录

- 课程目标
- 教学参考书
- 算法的重要地位
- 算法的定义与特性
- 算法的伪码描述
- 递归算法
- 算法设计方法
- 算法分析的基本概念
- 算法复杂性的渐近表示
- Knuth的算法定义
- Knuth的算法5特性
- Wikipedia的算法定义
- 算法名称的由来
- 《算法导论》中的算法定义
- Euclid GCD算法
- 算法的伪代码描述标准
- 典型的算法设计方法

《算法导论》中关于算法的定义

非形式地说，**算法**(algorithm)就是任何良定义的计算过程，该过程取某个值或值的集合作为**输入**并产生某个值或值的集合作为**输出**。这样算法就是把输入转换成输出的计算步骤的一个序列。

我们也可以把算法看成是用于求解良说明的**计算问题**的工具。一般来说，问题陈述说明了期望的输入/输出关系。算法则描述一个特定的计算过程来实现该输入/输出关系。

例如，我们可能需要把一个数列排成非递减序。实际上，这个问题经常出现，并且为引入许多标准的设计技术和分析工具提供了足够的理由。下面是我们关于**排序问题**的形式定义。

输入：n 个数的一个序列$\langle a_1, a_2, \cdots, a_n\rangle$。

输出：输入序列的一个排列$\langle a_1', a_2', \cdots, a_n'\rangle$，满足 $a_1' \leqslant a_2' \leqslant \cdots \leqslant a_n'$。

例如，给定输入序列$\langle 31, 41, 59, 26, 41, 58\rangle$，排序算法将返回序列$\langle 26, 31, 41, 41, 58, 59\rangle$作为输出。这样的输入序列称为排序问题的一个**实例**(instance)。一般来说，**问题实例**由计算该问题解所必需的(满足问题陈述中强加的各种约束的)输入组成。

算法的定义—困难性

- **Algorithm does not have a generally accepted formal definition. Researchers are actively working on this problem.**
- **算法尚没有一致接受的形式化定义。对这样一个问题，研究人员正在加倍努力。**
- **Over the last 200 years the definition of algorithm has become more complicated and detailed as researchers have tried to pin down the term.**
- **在过去的200年里，研究人员对该术语不断深入地探索，使算法的定义得到了深化和细致。**

Dasgupta et al.
Algorithms, 2006

高德纳·克努特(Donald Knuth)

Author of the seminal multi-volume work: *The Art of Computer Programming*,

The "father" of the analysis of algorithms

January 10, 1938

http://en.wikipedia.org/wiki/Algorithm_characterizations

The Art of Computer Programming

Knuth的算法定义

- **An algorithm is a finite set of rules that gives a sequence of operations for solving a specific type of problem.**
- **算法是一个有穷规则的集合，该集合给出求解某一特定问题的操作序列。**

Donald Knuth
The Art of Computer Programming, 1st ed., 1968

Knuth的算法5特性

Knuth (1968) has given a list of five properties that are widely accepted as requirements for an algorithm:

克努特(1968)给出了关于算法的5条性质，这5条性质被广泛地认为是算法的必要条件

1. Finiteness: "An algorithm must always terminate after a finite number of steps ... a very finite number, a reasonable number "
1. 有穷性：一个算法必须总是在有限数量的步骤后结束 —非常有限且合理数量的步骤

Knuth的算法5特性

2. Definiteness: "Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case“
2. 确定性：算法的每一个步骤必须被精确地定义；步骤中要执行的动作必须对每一种情况都精确、无二义地清楚说明

Knuth的算法5特性

3. Input: "...quantities which are given to it initially before the algorithm begins. These inputs are taken from specified sets of objects"
3. 输入：算法开始前给予的初始数据，这些输入从一个指定的对象集取得
4. Output: "...quantities which have a specified relation to the inputs“
4. 输出：与输入有指定关系的数据

Knuth的算法5特性

5. Effectiveness: "... all of the operations to be performed in the algorithm must be sufficiently basic that they can in principle be done exactly and in a finite length of time by a man using paper and pencil"
5. 有效性：算法中要执行的所有的操作必须足够地基础，以便理论上可以由人用铅笔和纸在有限的时间里准确地完成。

Knuth的算法5特性

- Knuth admits that, while his description of an algorithm may be intuitively clear, it lacks formal rigor, since it is not exactly clear what "precisely defined" means, or "rigorously and unambiguously specified" means, or "sufficiently basic", and so forth.
- 克努特承认，虽然他关于算法的描述可能在直觉上是清楚的，但是它缺乏形式化的严谨，因为什么是“精确定义的”(precisely defined)，什么是“严谨且无二义性地说明的”(rigorously and unambiguously specified), 什么是“足够的基础” (sufficiently basic)，等等都欠缺准确性。

Wikipedia的算法定义

- An algorithm is an effective method expressed as a finite list of well-defined instructions for calculating a function. Starting from an initial state and initial input (perhaps empty), the instructions describe a computation that, when executed, will proceed through a finite number of well-defined successive states, eventually producing "output" and terminating at a final ending state. The transition from one state to the next is not necessarily deterministic; some algorithms, known as randomized algorithms, incorporate random input.

Wikipedia

Wikipedia的算法定义

- 算法是计算一个函数的有效方法，该方法可由良好定义的指令构成的有限列表表示。
  该指令列表描述了一种计算，当执行时，它从一个初始状态和初始输入(可能为空)开始，依次经过有限个数的良好定义的状态，最终产生输出并终结于最后的结束状态。
  从一个状态到另一个状态的迁移不必须是确定性的，一些称为随机算法的算法允许接收随机的输入。

Wikipedia

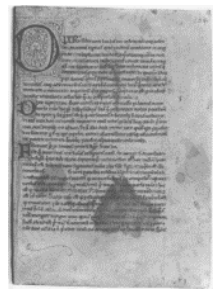
算法名称的由来

- **Muḥammad ibn Mūsā al-Khwārizmī, formerly Latinized as Algoritmi, was a Persian mathematician, astronomer and geographer during the Abbasid Caliphate (阿巴斯哈里发帝国), a scholar in the House of Wisdom in Baghdad.**

公元780-850

算术著作

- **In the twelfth century, Latin translations of his work on the Indian numerals introduced the decimal positional number system to the Western world.**



Arithmetic

算术便是算法

Arithmetic

代数著作

- **Al-Khwārizmī's The Compendious Book on Calculation by Completion and Balancing presented the first systematic solution of linear and quadratic equations in Arabic.**
- **In Renaissance Europe, he was considered the original inventor of algebra, although it is now known that his work is based on older Indian or Greek sources**

Algebra

算法名称的由来

- **In Renaissance Europe, he was considered the original inventor of algebra, although it is now known that his work is based on older Indian or Greek sources**
- **Some words reflect the importance of al-Khwārizmī's contributions to mathematics.**
- **"Algebra" is derived from al-jabr, one of the two operations he used to solve quadratic equations.**
- **Algorism and algorithm stem from Algoritmi, the Latin form of his name.**

公元780-850

算法名称的由来

- **Fibonacci's contribution**
  Al Khwarizmi's work could not have gained a foothold in the West were it not for the efforts of one man: the 13th century Italian mathematician Leonardo Fibonacci, who saw the potential of the positional system and worked hard to develop it further and propagandize it.
- **斐波那契的贡献**
  如果不是一个人的重要努力，Al Khwarizmi的工作将不会在西方扎根。这个人便是13世纪意大利数学家莱昂纳多 斐波那契，他洞察到了位置计数系统的巨大潜力，并且努力地开拓和传播它。

算法名称的由来

莱昂纳多·斐波那契(Leonardo Fibonacci)
公元1170 - 1250

典型算法—Euclid GCD算法

- **欧几里德最大公约数算法**
  - **Greatest Common Divisor (GCD)**
  - **gcd(a,0) = a**
  - **gcd(a,b) = gcd(b, a MOD b)**
  - **系少数几个古老而直到现代都很有用的算法**

- **欧几里德(Euclid, 325-265 BC)**
  - **古希腊数学家，几何学之父**
- **欧几里德算法的复杂度: $O(n^3)$**
  - **数字有关的算法输入规模是数字的二进制位数**
    - **n位二进制数的范围: $0 \sim 2^n-1$**

GCD算法—自然语言描述

自然语言也就是人们日常进行交流的语言，如汉语、英语等。其最大的优点是简单、通俗易懂，缺点是不够严谨、烦琐且不能被计算机直接执行。

欧几里得算法用自然语言描述如下：

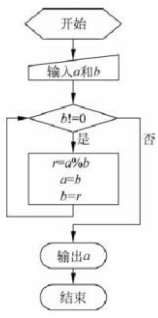
(1) 输入 a 和 b。

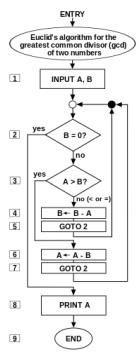
(2) 判断 b 是否为 0，如果不为 0，转步骤(3)；否则转步骤(4)。

(3) a 对 b 取余，其结果赋值给 r，b 赋值给 a，r 赋值给 b，转步骤(2)。

(4) 输出 a，算法结束。

GCD算法—流程图描述





GCD算法—程序设计语言描述

```
1  //Algorithm 1.1 (P1-2) (A01.01.cpp)
2  //欧几里德(Euclid)最大公约数(GCD)算法
3  //《几何原本》(300 B.C.)，辗转相除法
4  //《孙子算经》，公元 3-5 世纪
5  //《数书九章》，秦九韶，1247
6  #include "headers.h"
```

```
25 int euclidGCD(int m, int n, bool verbose)
26 {
27     if (verbose)
28         printf("%4d%4d\n", m, n);
29     int r;
30     do {
31         r = m % n;
32         m = n;
33         n = r;
34         if (verbose)
35             printf("%4d%4d\n", m, n);
36     } while(r);
37     return m;
38 }
```

GCD算法—伪代码描述

为了解决理解与执行这两者之间的矛盾，人们常常使用一种称为伪代码语言的描述方式来对算法进行描述。伪代码是介于自然语言与程序设计语言之间的一种用文字和符号结合的算法描述工具，它忽略了程序设计语言中一些严格的语法规则与描述细节，因此它比程序设计语言更容易描述和被人理解；它比自然语言更接近程序设计语言，因而较容易转换为能被计算机直接执行的程序。为此，对于计算机专业的初学者或非计算机人士来说，使用伪代码来描述算法倒是一个不错的选择。

欧几里得算法用伪代码描述如下：

```
Begin{算法开始}
   Step1: input a,b;
   Step2: if (b 不等于 0) 执行 Step3,否则执行 Step4;
   Step3: r = a % b,a = b,b = r,转 Step2;
   Step4: print a;
End{算法结束}
```


算法的伪代码描述标准

- 说明中文和简化的英文名称，必要时标上序号
- 说明输入和输出
- 顺序标记行号
- 赋值语句：使用=或左箭头进行变量赋值
- 变量可以带有下标，如：x_1, y_i
- 使用方括号表示数组和元素的下标
- 变量可以使用数学符号，如：$\alpha, \emptyset$
- 可以有数学公式表示的运算，如：$x = \frac{b}{a^2}$
- 可用简化的自然语言描述操作，如：交换a和b的值
- 可以使用直观可理解的符号，如：$a \leftrightarrow b$
- 可以使用直观可理解的数学函数，如：$\sin\theta$

算法的伪代码描述标准

- 使用\\书写注释，如：swap(a,b) \\交换a和b的值
- 使用return结束算法，必要时返回结果
- 使用if语句表示分支，可以带有then, else和else if
- for循环：for i=1 to n step 2 \\step为1时可以省略
- while循环
- do … while循环
- repeat … until循环
- 复合语句
  - 分支或循环体有一个语句时，可以不用复合语句标识
  - 可以使用{ … }作为复合语句界定符
  - 可以使用begin … end作为复合语句界定符
  - 可以使用end if, end for, end while作为复合语句界定符
  - 使用缩格表示层次关系

欧几里得GCD算法

- 算法名称：欧几里得GCD算法(GCD)
- 输入：整数a, b
- 输出：a, b的GCD
- 1: while b≠0
- 2: c = a%b, a=b, b=c
- 3: end while
- 4: print a

典型的算法设计方法

- 穷举法(exhaustive search, brute-force, 蛮力法)
- 分治法(divide-and-conquer)
- 贪婪法(greedy search)
- 动态规划法(dynamic programming)
- 回溯法(backtracking)
- 分支限界法(branch-and-bound)

目录

算法分析的基本概念

- In computer science, the analysis of algorithms is the determination of the number of resources (such as time and storage) necessary to execute them. Most algorithms are designed to work with inputs of arbitrary length.
- 在计算机科学中，算法分析用来确定算法执行时必须的资源数量，包括时间资源和存储资源。
  大部分的算法被设计成可处理任意数量的输入。
  - 基本：编程对十个数33,78,29,5,98,22,38,4,34,70进行排序
  - 专业：编写一个排序算法

Wikipedia

算法分析的基本概念

- Usually the efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps (time complexity) or storage locations (space complexity).
- 通常，算法的效率和运行时间用一个运算步数相对于输入长度的函数(时间复杂性)或存储空间相对于输入长度的函数(空间复杂性)来刻画。
  - 二分查找算法的时间复杂度为log*n*
  - 归并排序的时间复杂度为*n*log*n*

Wikipedia

算法的空间复杂性

- 由于把数据写入到每一个存储单元，至少需要一个特定的时间间隔，因此，算法的空间复杂性不会超过其时间复杂性，即：
  - $S(n) = O(T(n))$
  - 如果不特别指明，说到算法的复杂性都是指算法的时间复杂性
  - 如果同一个问题的不同算法时间复杂性相同，这时空间复杂性就非常重要了

算法时间复杂性—事后统计法

- 对不同的算法进行程序实现，在相同的软硬件环境中以相同的若干组输入数据运行若干次，通过测量运行时间的平均值和运算结果优劣对算法进行比较的方法。

最坏、最好和平均复杂性

通常情况下，人们只考虑三种情况下的时间复杂性，即最坏情况、最好情况和平均情况，并分别记为 $T_{max}(n)$、$T_{min}(n)$和 $T_{avg}(n)$。设 D_n 是问题规模为 n 的算法的所有合法输入序列集合；I^0 是使算法的时间效率达到最差的合法输入序列集合；I^1 是使算法的时间效率达到最好的合法输入序列集合；$P(I)$是算法在应用中出现输入序列 I 的概率。在数学上有

$$T_{max}(n) = \max_{I \in D_n} T(n, I) = T(n, I^0)$$

$$T_{min}(n) = \min_{I \in D_n} T(n, I) = T(n, I^1)$$

$$T_{avg}(n) = \sum_{I \in D_n} T(n, I) P(I)$$

由此，针对特定的输入序列，算法的时间复杂性只与问题的规模 n 有关。一个不争的事实是：几乎所有的算法，规模越大所需的运行时间就越长。当 n 不断变化时，运行时间也会不断变化，故人们通常将算法的运行时间记为 $T(n)$。

最坏、最好和平均复杂性—线性搜索

- 算法名称：线性查找(linearsearch)
- 输入：n个元素的数组a和带查找的元素x
- 输出：x在a中的位置，0表示未找到
- 1: for i=1 to n
- 2: if x=a[i] then
- 3: return i
- 4: return 0

渐近复杂性态

假设算法 A 的运行时间表达式 $T_1(n)$ 为：

$$T_1(n) = 30n^4 + 20n^3 + 40n^2 + 46n + 100 \quad (1\text{-}1)$$

算法 B 的运行时间表达式 $T_2(n)$ 为：

$$T_2(n) = 1000n^3 + 50n^2 + 78n + 10 \quad (1\text{-}2)$$

显然，当问题的规模足够大的时候，例如 $n=100$ 万，算法的运行时间将主要取决于时间表达式的第一项，其他项的执行时间只有它的几十万分之一，可以忽略不计。第一项的常数，随着 n 的增大，对算法的执行时间也变得不重要了。

于是，算法 A 的运行时间可以记为：$T_1^*(n) \approx n^4$，称 n^4 为 $T_1^*(n)$ 的阶。

同理，算法 B 的运行时间可以记为：$T_2^*(n) \approx n^3$，称 n^3 为 $T_2^*(n)$ 的阶。

渐近复杂性态

由上述分析可以得出一个结论：随着问题规模的增大，算法的时间复杂性主要取决于运行时间表达式的阶。如果要比较两个算法的效率，只需比较它们的阶就可以了。

定义 1 设算法的运行时间为 $T(n)$，如果存在 $T^*(n)$，使得

$$\lim_{n\to\infty} \frac{T(n) - T^*(n)}{T(n)} = 0$$

就称 $T^*(n)$ 为算法的渐进性态或渐进时间复杂性。

可见，问题规模充分大时，$T(n)$ 和 $T^*(n)$ 近似相等。因此，在算法分析中，对算法的时间复杂性和算法的渐进时间复杂性往往不加区分，并常用后者来对一个算法的时间复杂性进行衡量，从而简化了大规模问题的时间复杂性分析。

渐近上界(大O记法)

(1) 渐近上界记号：O(big-oh)。

定义 2 若存在两个正常数 c 和 n_0，使得当 $n \geqslant n_0$ 时，都有 $T(n) \leqslant cf(n)$，则称 $T(n) = O(f(n))$，即 $f(n)$ 是 $T(n)$ 的上界。换句话说，在 n 满足一定条件的范围内，函数 $T(n)$ 的阶不高于函数 $f(n)$ 的阶。

【例 1-1】 用 O 表示 $T(n)=10n+4$ 的阶。

这就是说当$n>n_0$时，f增长的速度不超过g的某个常数倍

存在 $c=11$，$n_0=4$，使得当 $n \geqslant n_0$ 都有：

$$T(n) = 10n + 4 \leqslant 10n + n = 11n$$

令 $f(n)=n$，可得

$$T(n) \leqslant cf(n)$$

即 $T(n)=O(f(n))=O(n)$。

应该指出，根据符号 O 的定义，用它评估算法的复杂性得到的只是问题规模充分大时的一个上界。这个上界的阶越低，则评估就越精确，结果就越有价值。如果有一个新的算法，其运行时间的上界低于以往解同一问题的所有其他算法的上界，就认为建立了一个解该问题所需时间的新上界。

常见的复杂度类

常见的几类时间复杂性有：

$O(1)$：常数阶时间复杂性。它的基本运算执行的次数是固定的，总的时间由一个常数来限界，此类时间复杂性的算法运行时间效率最高。

$O(n)$、$O(n^2)$、$O(n^3)$、…：多项式阶时间复杂性。大部分算法的时间复杂性是多项阶的，通常称这类算法为多项式时间算法。$O(n)$称为 1 阶时间复杂性，$O(n^2)$称为 2 阶时间复杂性，$O(n^3)$称为 3 阶时间复杂性…。

$O(2^n)$、$O(n!)$和 $O(n^n)$：指数阶时间复杂性。这类算法的运行效率最低，这种复杂性的算法根本不实用。如果一个算法的时间复杂性是指数阶的，通常称这个算法为指数时间算法。

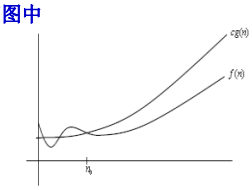
$O(n\log n)$和 $O(\log n)$：对数阶时间复杂性。除常数阶时间复杂性以外，它的效率最高。

以上几种复杂性的关系为：

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

渐近上界示例

- $f=O(g(n))$可以形象地表示在下图中
- $10n=O(n)$
  - 取n_0 =1，c=11，则$n>n_0$时，有$10n \leq c \cdot n$
- $0.1n=O(n)$
  - 取n_0 =1，c=1，则$n>n_0$时，有$0.1n \leq c \cdot n$
- $n+1000=O(n)$
  - 取n_0 =1000，c=2，则$n>n_0$时，有$n+1000 \leq c \cdot n$
- $3n^2+100n=O(n^2)$
  - 取n_0 =33，c=6，则$n>n_0$时，有$3n^2+100n \leq c \cdot n^2$



大O的极限判断准则

- 根据$f(n)=O(g(n))$的定义，可以推得如下的极限判断准则：
  - 当$n\to\infty$时，$f(n)=O(g(n))$，当且仅当$\lim\limits_{n\to\infty}\frac{f(n)}{g(n)}<\infty$
  - $\lim\limits_{n\to\infty}\frac{10n}{n}=10<\infty \Rightarrow 10n=O(n)$
  - $\lim\limits_{n\to\infty}\frac{0.1n}{n}=0.1<\infty \Rightarrow 0.1n=O(n)$
  - $\lim\limits_{n\to\infty}\frac{n+1000}{n}=1<\infty \Rightarrow n+1000=O(n)$
  - $\lim\limits_{n\to\infty}\frac{3n^2+100n}{n^2}=3<\infty \Rightarrow 3n^2+100n=O(n^2)$

大O的渐近紧界

- 显然
  - $3n = O(n)$、$3n = O(n^2)$、$3n = O(n^3)$
- 而$3n = O(n)$显然比后两者更有意义
  - 因此*f(n)=O(g(n))*中的*g(n)*通常取对*f(n)*最紧的界
  - 后两者的情况在数学上称为是平凡的(trivial)
- 非平凡(non-trivial)的g(n)一般满足如下极限
  - $0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$
  - 但有时找不到这样的$g(n)$
- 平凡的g(n)一般满足如下极限
  - $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

渐近复杂性是一种复杂度类

- $f(n) = O(g(n))$中的“=”号与一般数学中“=”号的意义是不同的
  - 此处“=”号可理解为“is”：*f(n)*是*O(g(n))*的
  - 因此，更易理解的表示法是：$f(n) \in O(g(n))$
  - 也就是说$g(n)$是一个函数类，更确切地说是一个复杂度类
  - 因而，我们可以说，对于任意的实数*a*、*b*且*a*>0，任何的$f(n) = an + b$都是属于$O(n)$(类)的，或者说都是复杂度为$O(n)$的
  - 更广泛地，我们有$\sum_{i=0}^{k} a_i n^k = O(n^k)$，当$a_k > 0$时

大O是上界或阶

- $f(n) = O(g(n))$还可做如下理解
  - 由于可能有$g(n) < f(n)$，直接将$g(n)$理解为$f(n)$的“上界”有违常理，较合理的理解为：$g(n)$是$f(n)$在某个常数倍意义上的“渐近上界”
  - 也可以将$g(n)$理解为“阶”的概念，即$f(n) = O(g(n))$表示复杂度$f(n)$是不超过$g(n)$阶的。
    非正式地，也将其说成是：$f(n)$是阶为$g(n)$的复杂度

渐近下界(大Ω记法)

(2) 渐近下界记号：Ω(big-omege)。

定义 3 若存在两个正常数 c 和 n_0，使得当 $n \geqslant n_0$ 时，都有 $T(n) \geqslant cf(n)$，则称 $T(n) = \Omega(f(n))$，即 $f(n)$ 是 $T(n)$ 的下界。换句话说，在 n 满足一定条件的范围内，函数 $T(n)$ 的阶不低于函数 $f(n)$ 的阶。它的概念与 O 的概念是相对的。

【例 1-2】 用 Ω 表示 $T(n) = 30n^4 + 20n^3 + 40n^2 + 46n + 100$ 的阶。

存在 $c = 30, n_0 = 1$，使得当 $n \geqslant n_0$ 都有

$$T(n) \geqslant 30n^4$$

令 $f(n) = n^4$，可得

$$T(n) \geqslant cf(n)$$

即 $T(n) = \Omega(f(n)) = \Omega(n^4)$。

渐近精确界(大Θ记法)

(3) 渐近精确界记号：Θ(big-theta)。

定义 4 若存在三个正常数 c_1、c_2 和 n_0，使得当 $n \geqslant n_0$ 时，都有 $c_2 f(n) \leqslant T(n) \leqslant c_1 f(n)$，则称 $T(n) = \Theta f(n)$。Θ 意味着在 n 满足一定条件的范围内，函数 $T(n)$ 和 $f(n)$ 的阶相同。由此可见，Θ 用来表示算法的精确阶。

【例 1-3】 用 Θ 表示 $T(n) = 20n^2 + 8n + 10$ 的阶。

定理 1 若 $T(n) = a_m n^m + a_{m-1} n^{m-1} + \cdots + a_1 n + a_0 (a_i > 0, 0 \leqslant i \leqslant m)$ 是关于 n 的一个 m 次多项式，则 $T(n) = O(n^m)$，且 $T(n) = \Omega(n^m)$，因此有 $T(n) = \Theta(n^m)$。

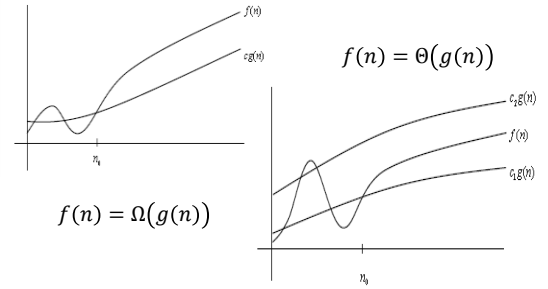
大Ω的极判断准则

- 大Ω的极判断准则：
  - $f(n) = \Omega(g(n))$，当且仅当 $\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq 0$
  - 如：$3n + 2 = \Omega(n)$
  - $3n^2 + 100n = \Omega(n^2)$
  - $3n^2 + 100n = \Omega(n)$

大Θ的极限判断准则

- 大Θ的极限判断准则
  - $f(n)=\Theta(g(n))$，当且仅当$0<\lim_{n\to\infty}\frac{f(n)}{g(n)}=C<\infty$
  - 如：$3n+2=\Theta(n)$
  - $3n^2+100n=\Theta(n^2)$
- 大Θ的O/Ω判断准则
  - 当同时满足$f(n)=O(g(n))$和$f(n)=\Omega(g(n))$时，$f(n)$与$g(n)$具有相同的阶，
  - 即$f(n)=\Theta(g(n))$.

大Ω与大Θ的图像解释



大O记法的由来

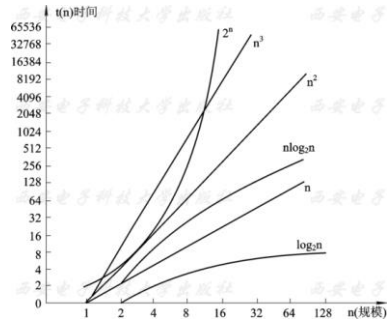
- 大O记法是由德国数论学家保罗 巴赫曼（Paul Bachmann，1837—1920）在其1892年的著作《解析数论》（Analytische Zahlentheorie）首先引入的。
- 另一位德国数论学家艾德蒙 朗道（Edmund Landau，1877—1938）在其著作中对大O记法进行了推广。
- O取自于“order of ...”（……阶），最初是一个大写的希腊字母'O'（Omicron）。
- 大O 、大Ω与大Θ是朗道—巴赫曼记号集中的主要成员

常见的渐近函数阶

符号	名称
$O(1)$	常数（阶，下同）
$O(\log^* n)$	迭代对数
$O(\log n)$	对数
$O[(\log n)^c]$	多对数
$O(n)$	线性，次线性
$O(n\log n)$	线性对数，或对数线性、拟线性、超线性
$O(n^2)$	平方
$O(n^c), \text{Integer}(c>1)$	多项式，有时叫作“代数”（阶）
$O(c^n)$	指数，有时叫作“几何”（阶）
$O(n!)$	阶乘，有时叫做“组合”（阶）

$\log^2 n=\log(\log n)=\log\log n$

常见渐近函数的曲线



常见函数的复杂度类

- 常数类：$C=\Theta(1)$
- 线性类：$an+b=\Theta(n), a>0$
- 二次类：$an^2+bn+c=\Theta(n^2), a>0$
- 多项式类：$\sum_{i=0}^{k}a_i n^k=O(n^k), a_k>0$
- 指数阶：$\Theta(2^n)$
- 对数阶：$\log n^k=\Theta(\log n)$
- 阶乘对数的阶：$\log n!=\Theta(n\log n)$
- 调和函数的阶：$H_n=\sum_{i=1}^{n}\frac{1}{i}=\Theta(\log n)$