Introduction to Algorithms

Ch.1 Introduction

Dr. He Huang
School of Computer Science and Technology
Soochow University

E-mail: huangh@suda.edu.cn

1

与MIT算法导论教材对应关系

教材 Chapter 1 ~ Chapter 3
Chapter 1. 算法在计算中的作用
Chapter 2. 算法基础
Chapter 3. 函数的渐进增长
Chapter 34. NP完全性理论(P/NP/NPC/NP-Hard)

2

Main Topics for this Chapter

- Some Basic Concepts 基本概念
- Asymptotic notations, and analysis 渐进时间表示及分析
- NP完全性理论 (区分并理解P/ NP/ NPC/ NP-Hard 几类问题)

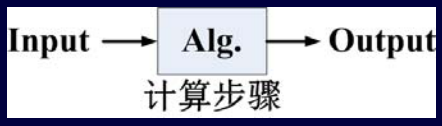
3

Chapter 1. Introduction

4

§ 1.1 算法(Algorithm)

- 非形式定义(Page 3 in the text book)：一个算法是任何一个良定义(well-defined)的计算过程，它接收某个值或值的集合作为输入，产生某个值或值的集合作为输出。因此，一个算法是一个计算步骤的序列，这些步骤将输入转化为输出。



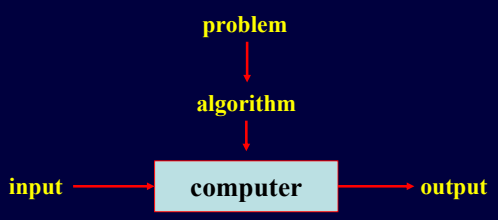
或者说，算法所描述的计算过程就是怎样达到所期望的I/O关系

5

§ 1.1 算法(Algorithm)

Another Def.

An *algorithm* is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.



6

例： 排序(Sorting)问题

- 问题描述
  - Input

    具有n个数的数列 $<a_1, a_2, \ldots, a_n>$
  - Output

    上述序列的一个排列$<a_1', a_2', \ldots, a_n'>$满足关系：$a_1' \le a_2' \le \ldots \le a_n'$
- 计算步骤
  - 如何达到上述关系

7

§ 1.1 算法(Algorithm)

- Instance (P3 text book)：一个问题的实例由计算该问题的一个解所需要的所有输入所组成。
- 正确性：若对每个输入实例，算法均终止于正确的输出，则称算法是正确的。

不正确的算法：
- 对某些输入实例不停机
- 虽然停机，但不是所期望的答案

Note：一个不正确的算法，若其错误的概率是可控制的，有时也是有用的 (Chapter 31 大素数算法，错误率可控算法)。

- 算法的描述：可以用英语说明，可以是程序语言，但是只要能精确描述计算过程即可。

8

Algorithms

Algorithm ?= Program

- Algorithm. (webster.com)

  -- A well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.

  --Broadly: a step-step procedure for solving a problem or accomplishing some end especially by a computer.

  --issues: correctness, efficiency (amount of work done and space used), storage (simplicity, clarity), optimality .etc.

9

§ 1.1 算法(Algorithm)——算法与程序的区别

- 算法

  算法是指解决问题的一种方法或一个过程，是若干*指令*的*有穷序列*，满足性质：

  (1)输入：有外部提供的量作为算法的输入

  (2)输出：算法产生至少一个量作为输出。

  (3)确定性：组成算法的每条指令是清晰，无歧义的。

  (4)有限性：算法中每条指令的执行次数是有限的，执行每条指令的时间也是有限的。

  正确性是前提

10

§ 1.1 算法(Algorithm)——算法与程序的区别

- 程序

  (1)程序是算法用某种程序设计语言的具体实现。

  (2)程序可以不满足算法的性质(4)。

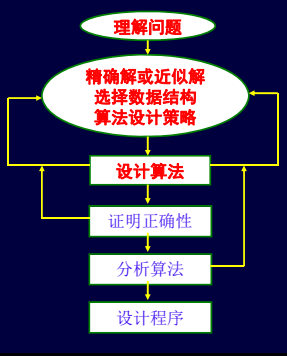
例如，操作系统是一个在无限循环中执行的程序，因而不是一个算法。

操作系统的各种任务可看成是单独的问题，每一个问题由操作系统中的一个子程序通过特定的算法来实现。该子程序得到输出结果后便终止。

11

§ 1.1 算法(Algorithm)

——问题求解(Problem Solving)



Some problems (P3-P4)

- Human Genome Project
  - 100,000 genes，sequences of the 3 billion chemical base pairs
- Internet
  - Finding good routes on which the data will travel
  - Search engine
- Electronic commerce
  - Public-key cryptography and digital signatures
- Manufacturing
  - Allocate scarce resources in the most beneficial way

13

Euclid's Algorithm

Problem: Find gcd(*m*,*n*), the greatest common divisor of two nonnegative, not both zero integers *m* and *n*

Examples: gcd(60,24) = 12, gcd(60,0) = 60, gcd(0,0) = ?

Euclid's algorithm is based on repeated application of equality

$$\gcd(m, n) = \gcd(n, m \bmod n)$$

until the second number becomes 0, which makes the problem trivial.

Example: gcd(60,24) = gcd(24,12) = gcd(12,0) = 12

14

Two descriptions of Euclid's algorithm

Step 1 If $n = 0$, return m and stop; otherwise go to Step 2

Step 2 Divide m by n and assign the value for the remainder to r

Step 3 Assign the value of n to m and the value of r to n. Go to Step 1.

Euclid(m,n)

while $n \neq 0$ do

$r \leftarrow m \bmod n$

$m \leftarrow n$

$n \leftarrow r$

return m

15

Other methods for computing gcd(*m*,*n*)

Consecutive integer checking algorithm

Step 1 Assign the value of min{*m*,*n*} to *t*

Step 2 Divide *m* by *t*. If the remainder is 0, go to Step 3; otherwise, go to Step 4

Step 3 Divide *n* by *t*. If the remainder is 0, return *t* and stop; otherwise, go to Step 4

Step 4 Decrease *t* by 1 and go to Step 2

16

Other methods for gcd(*m*,*n*) [cont.]

Middle-school procedure

Step 1 Find the prime factorization of *m*

Step 2 Find the prime factorization of *n*

Step 3 Find all the common prime factors

Step 4 Compute the product of all the common prime factors and return it as gcd(*m*,*n*)

Is this an algorithm?

17

Sieve of Eratosthenes

Input: Integer $n \geq 2$

Output: List of primes less than or equal to n

for $p \leftarrow 2$ to n do $A[p] \leftarrow p$

for $p \leftarrow 2$ to $\lfloor \sqrt{n} \rfloor$ do

  if $A[p] \neq 0$ //p hasn't been previously eliminated from the list

    $j \leftarrow p * p$

    while $j \leq n$ do

      $A[j] \leftarrow 0$ //mark element as eliminated

      $j \leftarrow j + p$

Example: 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

18

Importance of algorithms

- **Problem: sorting 10,000,000 integers**
  - **Case 1: Computer A executes one billion instructions per second(1GHz), an algorithm taking time roughly equal to $2n^2$ to sort n integers.**
  - **Case 2: Computer B executes one hundred million instructions per second(100MHz), an algorithm taking time roughly equal to 50nlgn to sort n integers.**
    - Case 1:

$$\frac{2\times(10^7)^2 \text{ instructions}}{10^9 \text{ instructions/second}} = 200000 \text{seconds} \approx 55 \text{hours}$$

    - Case 2:

$$\frac{50\times 10^7 \times \log 10^7 \text{ instructions}}{10^8 \text{ instructions/second}} = 105 \text{seconds}$$

Importance of algorithms

Run time (nanoseconds)		1.3 N^3	10 N^2	47 N $\log_2 N$	48 N
Time to solve a problem of size	1000	1.3 seconds	10 msec	0.4 msec	0.048 msec
	10,000	22 minutes	1 second	6 msec	0.48 msec
	100,000	15 days	1.7 minutes	78 msec	4.8 msec
	million	41 years	2.8 hours	0.94 seconds	48 msec
	10 million	41 millennia	1.7 weeks	11 seconds	0.48 seconds
Max size problem solved in one	second	920	10,000	1 million	21 million
	minute	3,600	77,000	49 million	1.3 billion
	hour	14,000	600,000	2.4 billion	76 billion
	day	41,000	2.9 million	50 billion	1,800 billion
N multiplied by 10, time multiplied by		1,000	100	10+	10

§ 1.2 算法分析

- **伪代码(pseudo code)描述**

  我们课本采用伪代码描述算法 (P9 介绍了伪代码描述算法的优势，简洁地表述算法本质，忽略细节)

- **伪代码的一些约定**

  课本P11

The problem of sorting

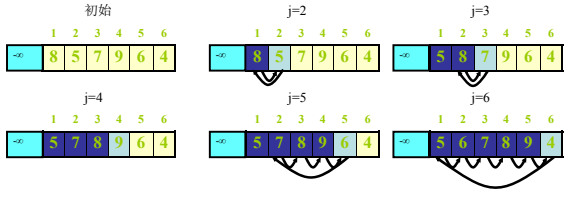
- Input: sequence $<a_1,a_2,\ldots,a_n>$ of *n* natural numbers
- Output: permutation $<a'_1, a'_2, \ldots, a'_n>$ such that $a'_1 \le a'_2 \le \ldots \le a'_n$
- Example
  --Input: <5,2,4,6,1,3>
  --Output: <1,2,3,4,5,6>

Insertion Sort

```
INSERT-SORT (A)                                  算法导论第二版的伪代码
1   for j ← 2 to length[A]
2       do 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              do A[i + 1] ← A[i]   ▷ move item back
7                 i ← i – 1
8          A[i + 1] ← key    ▷ find the insertion position
```

Insertion Sort

```
INSERT-SORT (A)
1   for j ← 2 to length[A]
2       do 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              do A[i + 1] ← A[i]   ▷ move item back
7                 i ← i – 1
8          A[i + 1] ← key    ▷ find the insertion position
```



Kinds of analysis

- **Worst-case: (usually)**
  --T(n) = maximum time of algorithm on any input of size n.
- **Average-case: (sometimes)**
  --T(n) = expected time of algorithm over all inputs of size n.
  --Need assumption of statistics distribution of inputs.
- **Best-case: (bogus假象)**
  --Cheat with a slow algorithm that works fast on some input.

Insertion sort analysis

- Worst case: Input reverse sorted

$$T(n) = \sum_{j=2..n} \Theta(j) = \Theta(n^2)$$

- Average case: All permutations equally likely

$$T(n) = \sum_{j=2..n} \Theta(j/2) = \Theta(n^2)$$

- Is insertion sort a fast sorting algorithm?
  --Moderately so, for small n.
  --Not at all, for large n.

§ 1.2 算法分析

- **目的**

  分析算法就是估计算法所需资源，选取有效的算法。(时间，空间，通信带宽等)

- **计算模型**

  单处理机，RAM (Random Access Machine, 随机存取机)模型，其中指令是顺序执行的，无并发操作

- **涉及的知识基础**

  离散组合数学、概率论、代数等(分析)

  程序设计、数据结构(算法设计)

§ 1.2 算法分析(续)

- **时间分析**

  算法耗费的时间与{输入实例的大小 / 实例的构成}有关

  例如：插入排序就如同打牌时整理纸牌{纸牌数目 / 输入本身就有序}

  ❖**Input-Size**

  通常用整数表示，取决于被研究的问题

  例：排序一个数组，Size的自然度量是项数

  两数相乘，最好的度量是总的位数

  有时，需用两个或多个整数表示输入规模，如图的顶点和边

§ 1.2 算法分析(续)

- **时间分析**

  ❖运行时间

  ①用基本操作的数目(执行步数)来度量；(好处是算法分析独立于机器，即任何基本操作看作是单位时间)

  ②用更接近实际的计算机上实现的时间来度量；(如RAM模型，不同的指令具有不同的执行时间)

  但两者相差一个常数因子。

  ❖最坏时间

  最坏运行时间指Size为n时任何输入的最长运行时间。

§ 1.2 算法分析(续)

为何要分析算法的最坏运行时间(P15)？

①它是算法对于任何输入的运行时间的上界；

②对于某些算法，最坏情况常常发生，如在DB中搜索一个并不存在的记录；

③平均时间往往和最坏时间相当(常数因子不同，也有反例哦！)

❖平均运行时间

常常假定一个给定Size的所有输入是等概率的。实际上这种可能并不一定成立，但可以用随机化算法强迫它成立。有时平均时间和最坏时间不是同数量级，算法选择依据是：

最好、最坏的概率较小时，尽量选择平均时间较小的算法。

Analysis algorithms

- We shall assume a generic one-processor, random-access machine (RAM) model of computation.
  - Instructions are executed one after another, with no concurrent operations.
  - Each time, an instruction of a program is executed as an atom operation. An instruction includes arithmetic operations, logical operations, data movement and control operations.
  - Each such instruction takes a constant amount of time.
  - RAM capacity is large enough.
- Under RAM model: count fundamental operations

Analysis of insertion sort

		cost	times
	INSERT-SORT (A)		
1	for j ← 2 to length[A]	c_1	n
2	do 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	do A[i + 1] ← A[i] ▷ move item back	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 ▷ find the insertion position	c_8	n-1

t_j : the number of times the while loop test in line 5 is executed for the j value.

Analysis of insertion sort

• To compute *T(n)*, the running time of Insertion-sort, we sum the products of the cost and times columns, obtaining

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

$$+ c_7\sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

• The best-case if the array is already sorted,

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$
$$= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

--The running time is a linear function of n.

Analysis of insertion sort

• The worst-case results if the array is in reverse sorted order-that is, in decreasing order.

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n(n+1)/2 - 1) + c_6(n(n-1)/2)$$
$$c_7(n(n-1)/2) + c_8(n-1)$$
$$= (c_5/2 + c_6/2 + c_7/2)n^2$$
$$+ (c_1 + c_2 + c_4 + c_5/2 - c_6/2 - c_7/2 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

-- The running time is a quadratic function of n.

$$\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j = n(n+1)/2 - 1$$

$$\sum_{j=2}^{n}(t_j - 1) = \sum_{j=2}^{n}(j-1) = n(n-1)/2$$

§ 1.2 算法分析(续)

- 函数增长率

  an^2+bn+c

35

§ 1.3 算法设计

- 增量法：

  如插入排序中，A[1,…, j-1]已排序，将A[j]插入合适的地方使得A[1,…, j]有序。
- 分治法 (Divide-and-Conquer)
- 贪心法 (Greedy)
- 动态规划 (Dynamic Programming)
- 回溯法 (Backtracking)
- 分枝—限界法等 (Branch and Bound)

36

§ 1.4 为什么要研究算法

- 软件系统性能取决于算法的效率及快速的硬件，有时高效的算法更重要

37

Chapter 2. 函数增长率
Growth of functions

38

Topics:

- Growth of functions
- O/o/Θ/Ω/ω notations

39

Asymptotic Growth

- In the insertion-sort example we discussed that when analyzing algorithms we are
  - ✓ interested in worst-case running time as function of input size *n*.
  - ✓ not interested in exact constants in bound.
  - ✓ not interested in lower order terms (低阶项)
- A good reason for not caring about constants and lower order terms is that the RAM model is not completely realistic anyway.

Machine-independent time

- What is insertion sort's worst-case time?
  --It depends on the speed of our computer:
  >>relative speed (on the same machine)
  >>absolute speed (on different machine)
- BIG IDEA:
  --Ignore machine-dependent constants.
  --Look at growth of T(n) as n → ∞

"Asymptotic Analysis"

§ 2.1 渐近表示法

- 算法的渐近时间定义为一个函数，定义域为自然数集合N={0,1,2,...}(∵n表示Size)。但有时也将其扩展到实数或限制到自然数的某子集上。(P25)

a. θ记号（θ-notation）

- Def：给定一个函数g(n)，$\theta(g(n))$表示一个函数的集合：

$$\theta(g(n)) = \{f(n) | \exists \text{常数} c_1, c_2, n_0 > 0, \text{使得对所有的} n \geq n_0 \text{有} : 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\} // n\text{为size}$$

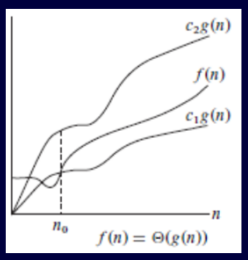
- 即 $f(n) \in \theta(g(n))$表示存在正常数c_1, c_2及足够大的n，使得f(n)夹在c_1g(n)和c_2g(n)之间
- 通常 $f(n) \in \theta(g(n))$ 表示为 $f(n) = \theta(g(n))$,其实f(n)是Θ(g(n))的成员 (可以理解为簇中的一员)

42

■ 大Θ的数学定义(渐进紧致界) 与上一页的$\theta(g(n))$无区别

对一个给定函数g(n)，用Θ(g(n))来表示以下的函数集合：

Θ(g(n))={f(n): 存在正常量c_1, c_2, n_0, 使得对所有n>=n_0, 有 $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$}



- 存在正常量c_1, c_2 使得对于足够大的n，函数f(n)能够被“夹在”c_1g(n)和c_2g(n)之间，则f(n)属于集合Θ(g(n)), f(n)=Θ(g(n))。
- 因为Θ(g(n))是一个集合，所以我们也可以记为f(n)∈Θ(g(n))
- 我们称g(n)是f(n)的一个渐进紧致(确)界(asymptotically tight bound)

43

§ 2.1 渐近表示法(续)

■ 意义：对所有的n≥n_0，函数f(n)在一个常数因子范围内等于g(n)

g(n)是f(n)的一个渐近紧致(确)界，即g(n)是f(n)的渐近上界和渐近下界

f(n)—算法的计算时间

g(n)—算法时间的数量级

不同的输入实例，一个算法的计算时间f(n)不一定相同，故算法的计算时间应该是一个函数集合。

Note： Θ定义中要求f(n)和g(n)是渐近非负的(n足够大时函数值非负)

否则 $\theta(g(n))$ 是空集

44

§ 2.1 渐近表示法(续)

■ 例 (Page 27 in textbook)：

❖ $T(n)=an^2+bn+c$ (a>0)

则 $T(n) = \theta(n^2)$

$\because$ 取$c_1 = \frac{a}{4}, c_2 = 7\frac{a}{4}, n_0 = 2\max(\frac{|b|}{a}, \sqrt{\frac{|c|}{a}})$

对所有$n > n_0$, 有$0 \le c_1 n^2 \le an^2 + bn + c \le c_2 n^2$成立

■ 记号 $\theta(1)$：

❖ 表示算法的运行时间与问题规模n无关

❖ 亦可理解为常值函数 $\theta(n^0)$ (任何常数是0次多项式)

45

Asymptotic Notation in Equations

■ Used to replace functions of lower-order terms to simplify equations/expressions.

■ For example,

$4n^3 + 3n^2 + 2n + 1 = 4n^3 + 3n^2 + \Theta(n)$
$= 4n^3 + \Theta(n^2) = \Theta(n^3)$

Or we can do the following: $4n^3 + 3n^2 + 2n + 1 = 4n^3 + f(n^2)$, where $f(n^2)$ simplifies the equation

课本P27，直觉上……

§ 2.1 渐近表示法(续)

b. $O-notation$(渐近上界)

❖ Def: 对给定函数g(n), $O(g(n))$ 是一个函数集合

$$O(g(n)) = \{f(n) | \exists 常数 c, n_0 > 0, such\ that$$
$$0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\}$$

❖ 即在一个常数因子范围内g(n)是f(n)的渐近上界

❖ Θ记号强于O 记号

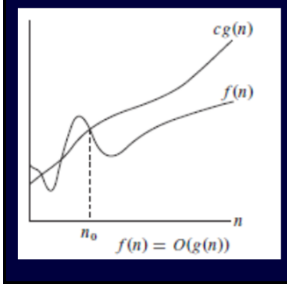
$\because \theta(g(n)) \subseteq O(g(n))$

$\therefore f(n) = \theta(g(n)) \Rightarrow f(n) = O(g(n))$

47

■ 大O的数学定义(渐进上界)

若g(n)和f(n)是定义在正整数集合上的两个函数，则f(n)=O(g(n))表示存在两个正的常数c和n_0，使得当n≥n_0时都满足0≤f(n)≤c·g(n)。



- 函数f(n)是集合O(g(n))的成员;
- f(n)=Θ(g(n))蕴含f(n)=O(g(n))，因为Θ是一个比O记号更强的概念;
- 我们称g(n)是f(n)的一个渐进上界(asymptotically upper bound)，限制算法最坏情况运行时间;

48

§ 2.1 渐近表示法(续)

- Note：
  - O描述上界，当用于界定一个最坏运行时间时，蕴含着该算法在任意输入上的运行时间都囿于此界
  - θ 则不然，一个算法的最坏运行时间是 $\theta(g(n))$ ，并非蕴含着该算法对每个输入实例的运行时间均囿于 $\theta(g(n))$
- 例：
  - 插入排序当输入初始有序时，它的运行时间是 $\theta(n)$，而不是 $\theta(n^2)$，但 $O(n^2)$ 对任何输入实例都成立。

    $\because\ n = O(n^2)$及$n^2 = O(n^2)$均成立，

    但是

    $n^2 = \theta(n^2)$而$n \neq \theta(n^2)$

49

Example

- $an^3+bn^2+cn+d = O(n^3)$,a>0

  Proof: 前面的例子已经表明$an^3+bn^2+cn+d = \Theta(n^3)$,又由于集合$\Theta(n^3)$是被$O(n^3)$包含的，所以得到证明

- $an^2+bn+d = O(n^3)$, a>0

Example

$1/3n^2 - 3n \in O(n^2)$ because $1/3n^2 - 3n \le cn^2$ if $c \ge 1/3 - 3/n$ which holds for $c = 1/3$ and $n > 1$

$k_1n^2+k_2n+k_3 \in O(n^2)$ because $k_1n^2+k_2n+k_3 \le (k_1+|k_2|+|k_3|)n^2$ and for $c > k_1 + |k_2| + |k_3|$ and $n \ge 1$, $k_1n^2 + k_2n + k_3 \le cn^2$

$k_1n^2+k_2n+k_3 \in O(n^3)$ as $k_1n^2+k_2n+k_3 \le (k_1+|k_2|+|k_3|)n^3$

O-notation

- Note:

  --When we say "the running time is $O(n^2)$" we mean that the worst-case running time is $O(n^2)$ – the best case might be better.

  --Use of O-notation often makes it much easier to analyze algorithms; we can easily prove the $O(n^2)$ insertion-sort time bound.

  --We often abuse the notation a little:

  >> We often write $f(n) = O(g(n))$ instead of $f(n) \in O(g(n))$

  >> We often use $O(n)$ in equations: e.g. $2n^2 + 3n + 1 = 2n^2 + O(n)$ (meaning that $2n^2 + 3n + 1 = 2n^2 + f(n)$ where $f(n)$ is some function in $O(n)$)

  >>We use $O(1)$ to denote constant time.

§ 2.1 渐近表示法(续)

- 当说算法的运行时间上界是 $O(n^2)$ 往往是指其最坏运行时间，无须修饰语，对那些最好、最坏、平均时间数量级不同者均成立，而Θ则要分开表达、加修饰语。
- c. Ω记号 (渐近下界, lower bound)
  - Def：对给定函数g(n), $\Omega(g(n))$ 是一个函数集合：

$$\Omega(g(n)) = \{f(n) \mid \exists \text{常数} c,\ n_0 > 0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\}$$

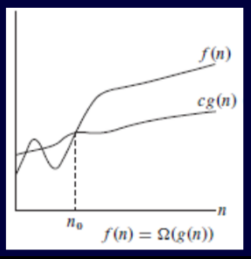
  - 意义：对所有n>n_0，f(n)在cg(n)上方。即在一个常数因子范围内g(n)是f(n)的渐近下界

53

- 大Ω的数学定义(渐进下界)

对一个给定函数g(n)，用Ω(g(n))来表示以下的函数集合：

Ω(g(n))={f(n): 存在正常量c, n_0, 使得对所有$n \ge n_0$ ≥0, 有 $0 \le cg(n) \le f(n)$}



- 存在正常量c使得对于n_0及其右边的所有n值，函数f(n)值总大于等于cg(n) ，则f(n)属于Ω(g(n))。
- 我们称g(n)是f(n)的渐进下界(asymptotically lower bound)

54

§ 2.1 渐近表示法(续)

- Th2.1 对任意函数f(n)和g(n)，$f(n)=\theta(g(n))$ 当且仅当 $f(n)=O(g(n))$和 $f(n)=\Omega(g(n))$ 。 (Page 28)

  即：g(n)是f(n)的渐紧界当且仅当g(n)是f(n)的渐近上界和渐近下界

- 当 Ω用来界定一个算法的最好情况下的运行时间时，蕴含着该算法在任意输入上的运行时间都囿于此界。
- 例：
  - 插入排序的下界是 $\Omega(n)$ ，对任何实例成立(即插入排序的最好运行时间是 $\Omega(n)$)

    $\because\ n=\Omega(n)\ \ n^2=\Omega(n)$

55

§ 2.1 渐近表示法(续)

d. 方程中的渐近记号 (P28~P29)

- 例：基于比较的排序时间下界是

$$\lg n!=n\lg n-1.44n+O(\lg n)$$

- 可消去不必要的细节，突出主项的常数因子等。

e. 小 o 记号(渐近非紧上界)

- 大 O 记号表示的渐近上界可以是渐近紧致的，也可以是渐近非紧界

$2n^2=O(n^2)\ \ \because\frac{2n^2}{n^2}\to$ 常数2，紧致界

$2n=O(n^2)\ \ \because\frac{2n}{n^2}\to 0$，非紧界

- 小 o 记号用来表示——函数的渐近非紧致上界

56

§ 2.1 渐近表示法(续)

e. 小 o 记号(渐近非紧上界)

- Def：　找出在形式化定义上与大O记号的两处差别

$$o(g(n))=\{f(n)\,|\,\forall 常数c>0,\exists 常数n_0>0\ such\ that\ 0\le f(n)<cg(n)\ \text{for all n}\ge n_0\}$$

只要n足够大，g(n)是f(n)的上界

例 $2n=o(n^2)$ 但 $2n^2\neq o(n^2)$ 。

直观上，当 $n\to\infty$ ，f(n)相对于g(n)是可忽略的

即： $f(n)=o(g(n))$蕴含着 $\lim_{n\to\infty}\frac{f(n)}{g(n)}=0$

或说f和g数量级不同，否则不可能对于任意常数c，都有cg(n)严格大于f(n)

57

§ 2.1 渐近表示法(续)

f. $\omega-$记号(渐近非紧下界)

- Def：　找出在形式化定义上与大Ω记号的两处差别

$$\omega(g(n))=\{f(n)\,|\,\forall 常数c>0,\exists 常数n_0>0\ such\ that\ 0\le cg(n)<f(n)\ \text{for all n}\ge n_0\}$$

即：对任意常数c>0，cg(n)对足够大的n要严格小于f(n)。 ∴g和f必定不是同数量级，(f量级>g量级)

- 例：

$$\frac{n^2}{2}=\omega(n)\ \ 但\frac{n^2}{2}\neq\omega(n^2)$$

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

58

§ 2.1 渐近表示法(续)

g. 函数间比较

- 许多实数的关系性质可用(引申)到渐近比较，下面假定f(n)和g(n)是渐近正的
- 传递性 (对于五种渐进记号均适用)

$f(n)=\theta(g(n))$ and g(n)=θ(h(n)) $\Rightarrow f(n)=\theta(h(n))$ //渐紧界

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

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

$f(n)=o(g(n))$ and g(n)=o(h(n)) $\Rightarrow f(n)=o(h(n))$ //渐近非紧上界

$f(n)=\omega(g(n))$ and g(n)=ω(h(n)) $\Rightarrow f(n)=\omega(h(n))$ //渐近非紧下界

59

§ 2.1 渐近表示法(续)

g. 函数间比较

- 自反性

$f(n)=\theta(f(n)),\ f(n)=O(f(n)),f(n)=\Omega(f(n))$

渐近非紧界无自反性

- 对称性 (仅对渐进紧确界成立)

$f(n)=\theta(g(n))\ iff\ g(n)=\theta(f(n))$

紧致界有对称性

60

§ 2.1 渐近表示法(续)

g. 函数间比较

- 转置对称性

$f(n)=O(g(n))\ iff\ g(n)=\Omega(f(n))$
//大O与大Ω对调时，f、g对称
$f(n)=o(g(n))\ iff\ g(n)=\omega(f(n))$
//g是f的非紧上界等价于f是g的非紧下界

61

§ 2.1 渐近表示法(续)

g. 函数间比较

- 算数运算 (请同学课下证明)
  - $O(f(n))+O(g(n)) = O(\max\{f(n),g(n)\})$；
  - $O(f(n))+O(g(n)) = O(f(n)+g(n))$；
  - $O(f(n))*O(g(n)) = O(f(n)*g(n))$；
  - $O(cf(n)) = O(f(n))$；
  - $g(n)= O(f(n)) \Rightarrow O(f(n))+O(g(n)) = O(f(n))$

62

§ 2.1 渐近表示法(续)

由上述4个性质，可将两函数间的渐近比较类比于两个实数间的比较。

$f(n)=O(g(n))\approx a\le b$ //“$\approx$”相似于$f\sim a, g\sim b$
$f(n)=\Omega(g(n))\approx a\ge b$
$f(n)=\theta(g(n))\approx a=b$
$f(n)=o(g(n))\approx a<b$
$f(n)=\omega(g(n))\approx a>b$

但实数的三岐性(三分性质)不能类比到渐近表示中

三岐性 $\forall a,b\in\mathbb{R}$, 下述三种情况必有一个成立：
$a<b, a=b, or\ a>b$

即任意两实数间是可比较的

63

§ 2.1 渐近表示法(续)

并非所有函数都是渐近可比较的

即 $\exists f(n)$ 和 $g(n)$,
可能$f(n)=O(g(n))$不成立，
而$f(n)=\Omega(g(n))$也不成立
$\therefore$由$Th2.1$知，$f\neq\theta(g)$

例：函数 n 和 $n^{1+\sin n}$ 之间是无法渐近比较的

$1+\sin n \Rightarrow 0\sim 2$
即 $n^{1+\sin}$ 在 $O(1)\sim O(n^2)$之间波动

64

Asymptotic order of growth

- A way of comparing functions that ignores constant factors and small input sizes
  - $O(g(n))$: class of functions $f(n)$ that grow no faster than $g(n)$
  - $\Theta(g(n))$: class of functions $f(n)$ that grow at same rate as $g(n)$
  - $\Omega(g(n))$: class of functions $f(n)$ that grow at least as fast as $g(n)$

65

§ 2.2 标准记号和常用函数

司特林公式，重叠对数等

见教材P31.

66

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*

67

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

68

Example 3: Matrix multiplication

ALGORITHM *MatrixMultiplication*$(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$
//Multiplies two n-by-n matrices by the definition-based algorithm
//Input: Two n-by-n matrices A and B
//Output: Matrix $C = AB$
for $i \leftarrow 0$ **to** $n-1$ **do**
  for $j \leftarrow 0$ **to** $n-1$ **do**
    $C[i, j] \leftarrow 0.0$
    for $k \leftarrow 0$ **to** $n-1$ **do**
      $C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$
return C

69

Chapter 3. NP完全性理论

70

不可解问题

- 是否对于每个问题都有解决它的算法？
- 自然地，人们会想到另外一个问题：会不会所有的问题都可以找到渐进时间复杂度为多项式级的算法呢？
- 答案是否定的。有些问题甚至根本不可能找到一个正确的算法来，这称之为"不可解问题"(Undecidable Decision Problem)。
- 例：
  - Hamilton回路
  - 问题是这样的：给你一个图，问你能否找到一条经过每个顶点一次且恰好一次(不遗漏也不重复)最后又走回来的路(满足这个条件的路径叫做Hamilton回路)；
  - 这个问题现在还没有找到多项式级的算法。事实上，这个问题就是我们后面要说的NPC问题。

§3 NP完全性理论——计算机科学的局限性

- 可解性：问题及其可解性可用函数和可计算性来代替
- 可计算性理论：研究计算的一般性质的数学理论，它通过建立计算的数学模型(例如抽象计算机)，精确区分哪些是可计算的，哪些是不可计算的。
- 可计算函数：能够在抽象计算机上编出程序计算其值的函数。这样就可以讨论哪些函数是可计算的，哪些函数是不可计算的。
- Church-Turing论题：若一函数在某个合理的计算模型上可计算，则它在图灵机上也是可计算的。
- C-T论题结论：可计算性不依赖于计算模型
- 不可计算性：很多问题和函数是无法用具有有穷描述的过程完成计算的

72

不可计算问题：停机问题

- **停机问题**：能否写一个程序正确判定输入给它的任何一个程序是否会停机？

设程序halts(P,X)总是正确地判定程序P在其输入X上是否停机:若停机，则返回yes；否则死循环，返回no。设另有一程序:

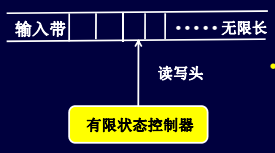
```
diagonal(Y){
    a: if halts(Y,Y) then
          goto a;
       else halt;
}
```

- **功能**：若halts断定当程序Y用其自身Y作为输入时Y停机,则diagonal(Y)死循环；否则它停机

diagonal(diagonal)是否停机？ 不可判定
它停机当且仅当halts(diagonal,diagonal)返回否，也就是：
diagonal停机当且仅当它自己不停机，矛盾！
即：halts(P,X)并不存在，停机问题是不可解的！

73

图灵机



- 有单带、多带等变种，计算能力等价

- **依据控制器的状态和读写头所读字符，决定执行以下3个操作之一、之二或全部：**
  1)改变有限状态控制器中的状态；
  2)读写头在相应的方格中写一符号；
  3)读写头左、右移一格或不动。
- **确定型图灵机DTM**：若对任一个状态和符号，要执行的动作都是唯一的
- **非确定型图灵机NTM**：执行的动作是有穷多个可供选择

74

§ 2.3 NP完全性理论 (阅读课本P617)

- **P类问题**：一类问题的集合，对其中的任一问题，都存在一个确定型图灵机M和一个多项式p，对于该问题的任何(编码)长度为n的实例，M都能在p(n)步内，给出对该实例的回答(Caution: Trick)。即：多项式时间内可解的问题

- **NP类问题(Nondeterministic Polynomial)**:一类问题的集合，对其中的任一问题，都存在一个非确定型图灵机M和一个多项式p，对于该问题的任何(编码)长度为n的实例，M都能在p(n)步内，给出对该实例的回答。

若NTM在每一步都恰有2步可供选择，则回答实例需考察$2^{p(n)}$种不同的可能性。

存在多项式时间的算法吗？

多项式时间内可验证问题 (指验证其解的正确性)

75

§ 2.3 NP完全性理论

- 之所以要定义NP问题，是因为通常只有NP问题才可能找到多项式的算法
- 我们不会指望一个连多项式地验证一个解都不行的问题存在一个解决它的多项式级的算法
- 很显然，所有的P类问题都是NP问题。也就是说，能多项式时间内解决一个问题，必然能在多项式时间验证一个问题的解——既然正解都出来了，验证任意给定的解也只需要比较一下就可以了

§ 2.3 NP完全性理论

- 关键是，人们想知道，是否所有的NP问题都是P类问题。
- 所有对NP问题的研究都集中在一个问题上，即究竟是否有P=NP？
- 目前为止这个问题还“啃不动”。但是，一个总的趋势、一个大方向是有的。人们普遍认为，P=NP不成立，也就是说，多数人相信，存在至少一个不可能有多项式级复杂度的算法的NP问题。

NPC问题

- 人们如此坚信P≠NP是有原因的，就是在研究NP问题的过程中找出了一类非常特殊的NP问题叫做NP-完全问题，也即所谓的NPC问题。C是英文单词“完全”的第一个字母。正是NPC问题的存在，使人们相信P≠NP。
- 为了说明NPC问题，我们先引入一个概念——归约(Reducibility)

归约

- 一个问题A可以归约为问题B的含义即是，可以用问题B的解法解决问题A，或者说，问题A可以“变成”问题B。
- “问题A可归约为问题B”有一个重要的直观意义：B的时间复杂度高于或者等于A的时间复杂度。也就是说，问题A不比问题B难。
  - 这很容易理解。既然问题A能用问题B来解决，倘若B的时间复杂度比A的时间复杂度还低了，那A的算法就可以改进为B的算法，两者的时间复杂度还是相同。

归约

- 归约具有一项重要的性质：归约具有传递性。如果问题A可归约为问题B，问题B可归约为问题C，则问题A一定可归约为问题C。
- 归约的标准概念：如果能找到这样一个变化法则，对任意一个程序A的输入，都能按这个法则变换成程序B的输入，使两程序的输出相同，那么我们说，问题A可归约为问题B。
- 我们所说的“可归约”是指的可“多项式地”归约(Polynomial-time Reducible)，即变换输入的方法是能在多项式的时间里完成的。归约的过程只有用多项式的时间完成才有意义。

P、NP及NPC类问题

- 多一归约
  - 假设L_1和L_2是两个判定问题，f将L_1的每个实例I变换成L_2的实例f(I)。若对L_1的每个实例I，I的答案为“是”当且仅当f(I)是L_2的答案为“是”的实例，则称f是从L_1到L_2的多一归约，记作：$L_1 \leq_m L_2$(传递关系)
  - 直观意义：将求解L_1的问题转换为求解L_2的问题，而问题L_1不会难于L_2
- 多项式时间多一归约：若f是多项式时间可计算，则上述归约称为多项式时间多一归约，也称多项式时间变换。记作：

$$L_1 \leq_m^p L_2$$

P、NP及NPC类问题

- NPC问题：对于一个(判定性)问题q，若
  (1) q∈NP
  (2) NP中任一问题均可多项式时间多一归约到q
  则称问题q为NP-完全的(NP-complete，NPC)
- NP-hard问题：若问题q仅满足条件(2)而不一定满足条件(1)，则问题q称为NP-难的(NP-hard)。显然：NPC⊆NP-hard (NP-hard是一个更大的集合)
- NPC和NP-hard关系
  NP-hard问题至少跟NPC问题一样难。
  NPC问题肯定是NP-hard的，但反之不一定

例：停机问题是NP-hard而非NPC的！

∵该问题不可判定，即无任何算法(无论何复杂度)求解该问题
∴该问题∉NP。但是

 可满足问题SAT$\leq_p$停机问题

P、NP及NPC类问题

- NP=?P
  ∵确定型图灵机是非确定型图灵机的特例，∴P⊆NP
  是否有NP⊆P？即是否NP=P？

美国麻省的Clay数学研究所于2000年5月24日在巴黎法兰西学院宣布：对七个“千年数学难题”中的每一个均悬赏100万美元，
而问题NP=?P位列其首:

1. P问题对NP问题
2. 霍奇猜想
3. 庞加莱猜想(2002.11-2003.7，俄罗斯数学家佩雷尔曼在3篇论文预印本中证明了几何化猜想，2006被授予菲尔兹奖)
4. 黎曼假设
5. 杨 - 米尔斯存在性和质量缺口
6. 纳维叶 - 斯托克斯方程的存在性与光滑性
7. 贝赫和斯维讷通 - 戴尔猜想

P、NP及NPC类问题

- P、NP、NPC和NP-hard之关系
  NPC是NP中最难的问题，但是NP-hard至少与NPC一样难

- 如何证明问题q是NP-hard或是NPC的？
  若已知q'∈NPC或q'∈NPH，且q' $\leq_p$ q，则q∈NPH；若进一步有q∈NP，则q∈NPC。
  即：要证q是NPH的，只要找到1个已知的NPC或NPH问题q'，然后将q'多项式归约到q即可。若还能验证q ∈NP，则q是NPC的。
  ∵ NP中任意问题均可多项式归约到q'，由于$\leq_p$有传递性
  ∴他们也都能多项式归约到q，由定义可知q是NPH的

NP-完全性理论

- Cook的贡献：第一个NPC问题

史提芬·库克(Stephen Arthur Cook，1939 -)NP完全性理论的奠基人，他在1971年论文"The Complexity of Theorem Proving Procedures"中，给出了第一个NP完备的证明，即Cook定理： 可满足性问题(Satisfiablity problem) 是NP完全问题，亦即 SAT∈NPC。且证明了：

SAT∈P当且仅当P=NP

Cook于1961年获Michigan大学学士学位，1962和1966年分获哈佛大学硕士与博士学位。1966-1970，他在UC Berkeley担任助教授；1970年加盟多伦多大学，现为该校CS和数学系教授，他的论文开启了NP完备性的研究，令该领域于之后的十年成为计算机科学中最活跃和重要的研究。因其在计算复杂性理论方面的贡献，尤其是在奠定NP完全性理论基础上的突出贡献而荣获1982年度的图灵奖。

85

NP-完全性理论

- NP-完全性理论的局限性

易解问题：可多项式时间内求解的问题

难解问题：需超多项式时间求解的问题

NP-完全性理论既没有找到第二类问题的多项式时间的算法，也没有证明这样的算法就不存在，而是证明了这类问题计算难度之等价性(彼此间困难程度相当)。因此，NPC具有如下性质：若其中1个问题多项式可解当且仅当其他所有NPC问题亦多项式可解

- 难解问题与易解问题之相似性

1) 最短/最长简单路径

单源最短路径问题：对有向图G，时间O(VE)，P问题

两点间最长路径：NPC问题，即使所有边上权为1

2) 欧拉环/哈密尔顿圈(G为无向图或有向图)

欧拉环：G中有通过每条边恰好一次的环？P，多项式时间可解

哈氏圈：G中有通过每个顶点恰好1次的圈？NPC

86

NP类问题的求解

- 减少搜索量

简单算法是穷举搜索，时间为指数

减少搜索量：分枝限界法，隐枚举法、动态规划等

可以提高效率，但时间复杂度不变

- 优化问题

降低优化要求，求近似解，以得到一个多项式时间的算法。即：找寻在容许的时间内得到容许精度的近似最优解的算法

- 近似比：近似算法所能得到的可行解(次优解)与最优解之间的比值

87

关于归约总结

- 减将一个问题归约为另一个问题不存在一劳永逸的方法，一些归约过程极其简单(e.g. 哈密顿回路归约为TSP)，一些归约极其复杂。

注意事项及一些技巧策略：

1. 必须满足的形式：将问题X的任意输入转换为关于问题Y的某些输入；
2. 利用归约源的限制优势：从哈密顿环问题进行归约比从TSP问题进行归约更为直接。因为TSP问题中，边的权重可以是任意正整数，而不是只可以取1或者0
3. (非常有用！！！)寻找特例：某些NPC恰恰是其他NPC的特例，比如partition prob是knapsack prob的特例，如果你知道问题X是NPC，并且X是Y的特例，那么Y必定也是NPC，为什么？

88

关于归约总结(续)

- 注意事项及一些技巧策略：

3. (非常有用！！！)寻找特例：某些NPC恰恰是其他NPC的特例，比如partition prob是knapsack prob的特例，如果你知道问题X是NPC，并且X是Y的特例，那么Y必定也是NPC，为什么？

因为Y比X更具一般性，问题Y至少与X一样难！！！

4. 寻找合适的规约源：

有的时候，我们会选择跨域归约策略，3-CNF可满足性问题是一个进行跨域归约合适的归约源。既可以规约到团问题(Graph)也可以归约到子集和(knapsack branch)；

在图问题中，如果需要选择部分图，且无需考虑顶点顺序，那么顶点覆盖问题通常是一个合适的归约源

89

关于归约总结(续)

- 注意事项及一些技巧策略：

5. 获取最大收益和最小补偿：

哈密顿环问题的输入图G转化为TSP问题加权图G'时，我们当然可以使用G中出现的边作为TSP问题相应的边。我们对这些边赋予非常小的权重：0，我们利用这些边可以获得巨大的收益。

90