



算法分析与设计

Analysis and Design of Algorithm

任课教师：沈典

办公室：计算机楼368室

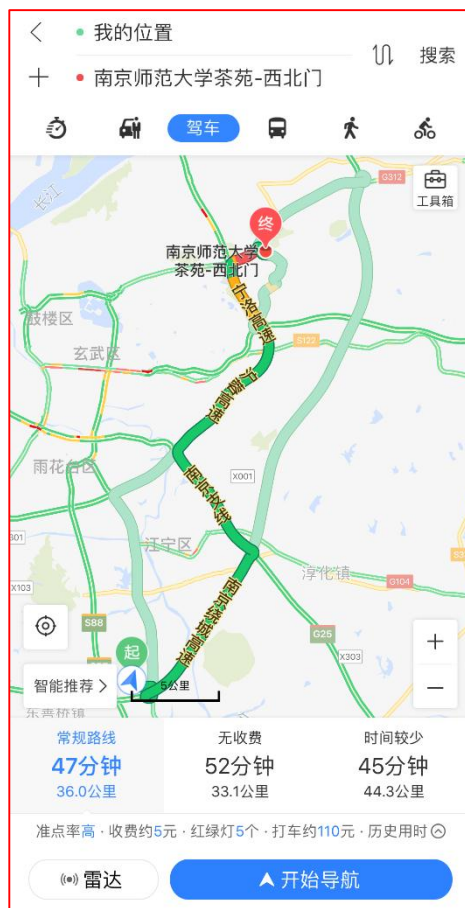
Email: dshen@seu.edu.cn

<http://dianshenseu.github.io>

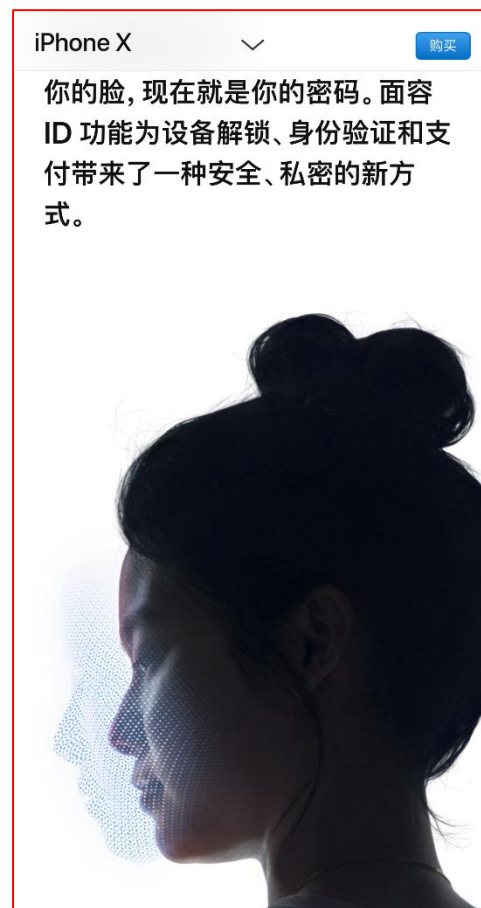
什么是算法 (Algorithm)



随机算法



寻路算法



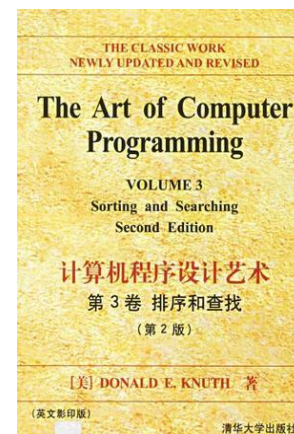
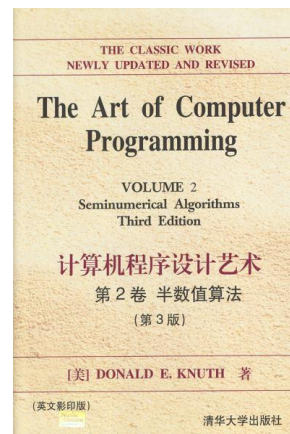
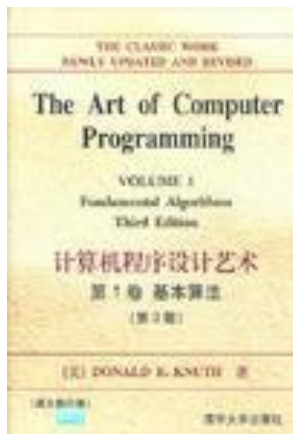
图像识别算法

什么是算法 (Algorithm)

百度百科：算法是指解题方案的准确而完整的描述，是一系列解决问题的清晰指令，算法代表着用系统的方法描述解决问题的策略机制。

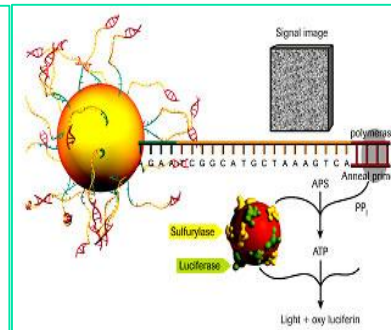
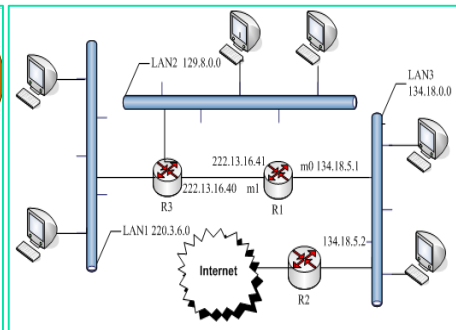
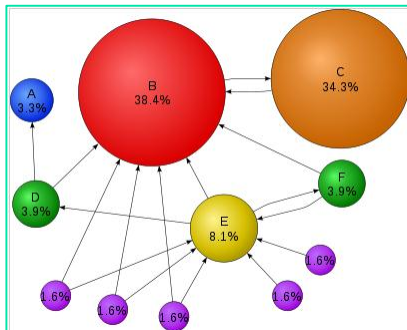


算法和程序设计技术的先驱者Donald Knuth：算法是带有输入输出的、有限的、确定的、有效的过程。



算法有哪些应用

- **互联网**：网页搜索、网络路由、BitTorrent...
- **生物信息**：人类基因组计划、蛋白质结构分析...
- **计算机图形**：电影、游戏、虚拟现实...
- **计算机安全**：手机、电子商务、投票系统...
- **多媒体**：MP3、JPG、HDTV...
- **人工智能**：人脸识别、AlphaGo、聊天机器人...
- **社会网络**：推荐系统、新闻推送、广告...
- **物理学**：粒子对撞机、反物质暗物质探测...



本科生学算法有什么用

求职面试： GFM、BAT、TMD...

读研深造： 考研面试、论文专利...

创新创业： 开发产品、性能优化...





本课程与其他课程的关系

编译
原理

操作
系统

计算机
网络

数据库
原理

算法分析与设计

数据结构与算法

程序设计基础及语言

面向对象程序设计



本课程的内容

NP完全性理论与近似算法

算法高级理论

随机化算法

线性规划与网络流

高级算法

递归
分治

动态
规划

贪心
算法

回溯与
分支限界

基础算法

算法分析与问题的计算复杂性

算法基础理论



参考书目

主要教材

- 王晓东. 算法设计与分析. 电子工业出版社

参考教材

- Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman. The Design and Analysis of Computer Algorithms. 1974年影印本, 铁道出版社
- Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman. 数据结构与算法. 1983年影印本, 清华大学出版社
- Thomas H. Cormen 等4人. 算法导论. MIT第2版影印本, 高教出版社



课程信息

■ 考核方式

- 平时（点名、课堂练习、课后作业）：**30%**
- 期末（考试）：**70%**

■ 作业要求

- 习题类型：算法分析题+算法实现题
- 截止日期：一周之内完成
- 提交方式：电子版（姓名、学号、题目、答案）
- 作业格式：建议用LaTeX编写作业（下载CTex并自学LaTeX）

■ 答疑方式

- 课程QQ群、计算机楼368室
- 课程助教：杨明璇（邮箱：220181709@seu.edu.cn）



第一章 算法概述

■ 学习要点:

- 理解算法的概念
- 理解什么是程序，程序与算法的区别和联系
- 掌握描述算法的方法
- 掌握算法的计算复杂性概念
- 掌握算法渐近复杂性的数学表述
- 了解NP类问题的基本概念

基础知识和算法设计



什么是算法

- 算法(Algorithm)

- 一个（由人或机器进行）关于某种运算规则的集合

输入 \rightarrow {规则} \rightarrow 输出

确定性 清晰、无歧义

有限性 指令执行次数、时间

- 特点：

- 执行时，不能包含任何主观的决定；
 - 不能有类似**直觉/创造力**等因素。

例子：

- 日常生活中做菜的过程，可否用算法描述？

✓ 如：“淡了”、“放点盐”、“再煮一会”。

✓ 可否用计算机完成？



- 算法必须规定明确的量与时间；
- 不能含糊字眼。

算法描述：

```
if (咸度 < 5) { 放1克盐; 煮半分钟; }
```

随机算法与近似算法

- 当然不是所有算法都有明确的选择，有些按照一定的概率进行选择。

- “随机”不等于“随意”



- 有些问题没有实用算法（算精确解需要多年）。

✓ 去寻找{规则集}



近似算法

可以预测误差，
且误差足够小

启发式算法

误差无法控制，
但可快速求解

在可接受的时间内可以算出足够好的**近似解**



算法和程序的关系

- 程序是算法用某种程序设计语言的具体实现
- 算法和程序的区别主要在于：
 - 在语言描述上，程序必须是用规定的程序设计语言来写，而算法很随意；
 - 在执行时间上，算法所描述的步骤一定是有限的，而程序可以无限地执行下去。
- 例如：操作系统是程序，但不是算法



算法和数据结构的关系

- 广义上讲，算法是一系列的运算步骤，它表达解决某一类计算问题的一般方法，对这类方法的任何一个输入，它可以按步骤一步一步计算，最终产生一个输出。
- 但是对于所有的计算问题，都离不开要计算的对象或者要处理的信息，而如何高效的把它们组织起来，就是数据结构关心的问题，所以算法是离不开数据结构的。



如何描述算法

- 算法的描述方法
 - 自然语言
 - 流程图
 - 程序设计语言：C、C++、Java、Python、...
 - 伪代码(Pseudocode)——算法语言
 - 类C/类Pascal语言
 - 结构化编程语言



如何描述算法

Algorithm 1 插入排序算法 Insert

Input: 待排序数组 T .

Output: 排序完成数组 T .

类C语言

```
template<class Type>
void Insert(Type T[], int n){
    //从第1个元素到第n个元素，n
    //把该元素插入到之前的数组相应位置上
    for(int i=2; i<n; i++){
        Type x = T[i]; int j = i-1;
        while(j > 0 && x < T[j]){
            T[j+1] = T[j]; j = j-1;
        }
        T[j+1] = x;
    }
}
```



如何描述算法

Algorithm 2 插入排序算法 Insert

Input: 待排序数组 T .

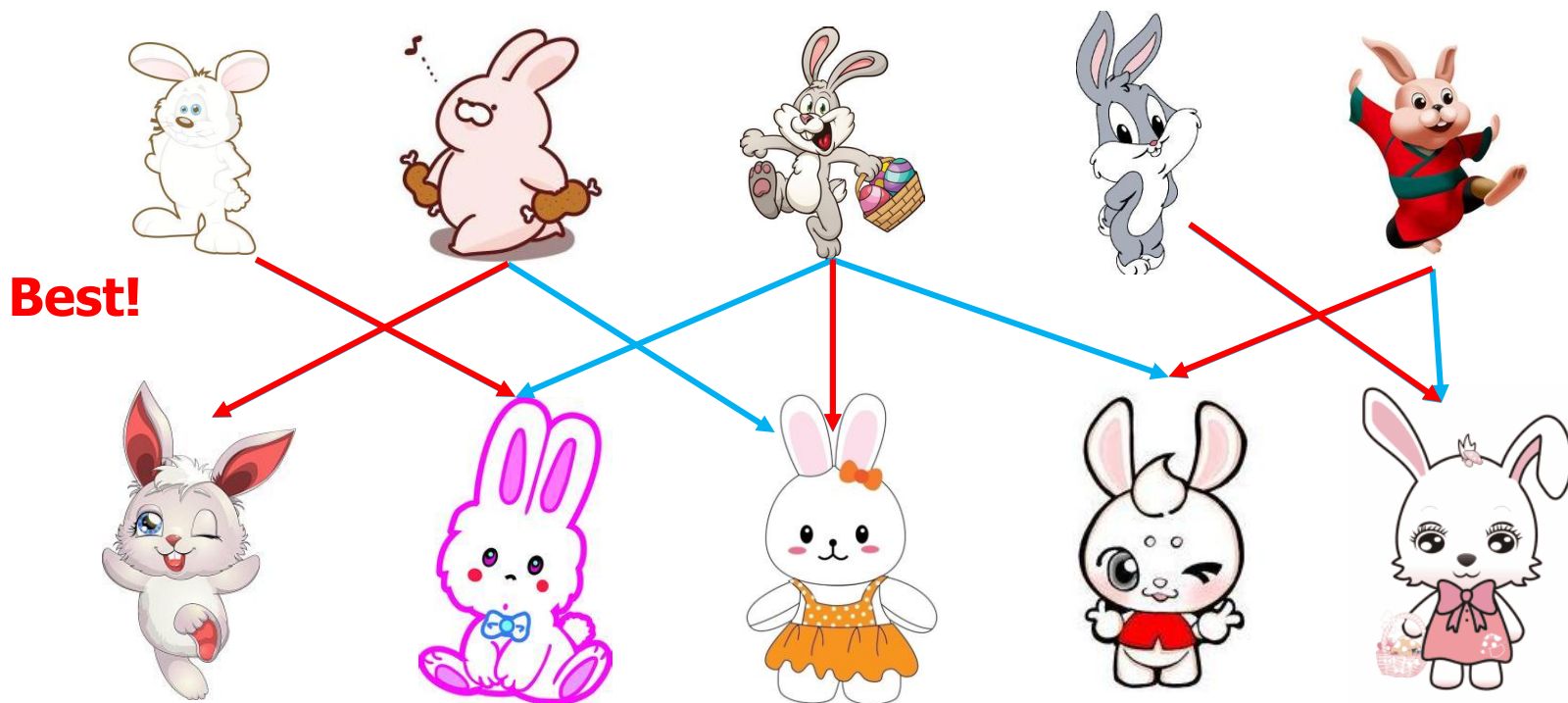
Output: 排序完成数组 T .

类Pascal语言

```
Procedure Insert( $T[1..n]$ )  
  //从第 2 列到第  $n$  个元素,  
  //把该元素插入到之前的数组相应位置上  
  for  $i \leftarrow 2$  to  $n$  do  
     $x \leftarrow T[i]; j \leftarrow i - 1;$   
    while  $j > 0$  and  $x < T[j]$  do  
       $T[j + 1] \leftarrow T[j]; j \leftarrow i - 1;$   
    end while  
     $T[j + 1] \leftarrow x;$   
  end for
```

如何设计一个算法（例1）

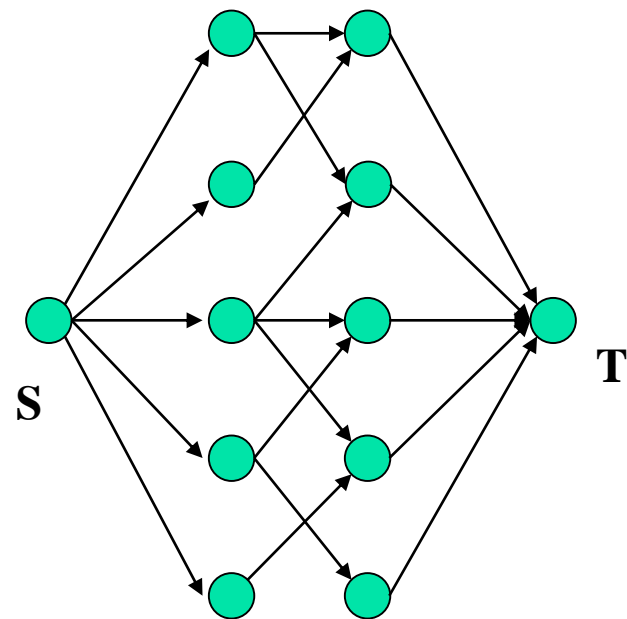
草原上生活着一群可爱的兔子，有5只公兔子和5只母兔子，每只公兔子都对若干母兔子有好感。如何帮助尽可能多的公兔子找到伴侣？



如何设计一个算法（例1）

- 步骤一：抽象为数学问题
 - 将公兔子和母兔子看作图中的点
 - 若公兔子a喜欢母兔子b，则在a和b之间连一条边
 - 增加一个源点S和终点T

最后，兔子匹配问题抽象成为
最大流问题（第8章）



- 步骤二：将求解过程用计算机语言描述
(较为复杂，此处省略，见第8章)



如何设计一个算法（例2）

每个月一对成熟的兔子会产生一对后代，而这对后代2个月后又会繁殖。即第1个月有1对小兔子；第2个月仍只有1对；第3个月有2对...假设兔子不死亡，**请问10个月后草原有几只兔子？**

	1月	2月	3月	4月	5月
刚出生	1	0	1	1	2
一个月	0	1	0	1	1
成熟	0	0	1	1	2
总数	1	1	2	3	5



如何设计一个算法（例2）

- 兔子数序列：0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55...
- 步骤一：抽象为数学问题
 - Fibonacci序列
 - 序列的表达式
$$\begin{cases} f_0 = 0; & f_1 = 1 \\ f_n = f_{n-1} + f_{n-2}, & n \geq 2 \end{cases}$$
- 步骤二：将求解过程用计算机语言描述

```
int Fibonacci(int n){  
    if(n<2) then return n  
    else return Fibonacci(n-1) + Fibonacci(n-2)  
}
```

算法分析



如何选择算法

- 当解决一个问题时，存在几种算法可供选择，如何决定哪个最好？
- 以排序算法为例：

算法	最坏情况	平均情况
插入排序	$O(n^2)$	$O(n^2)$
冒泡排序	$O(n^2)$	$O(n^2)$
快速排序	$O(n^2)$	$O(n\log n)$
归并排序	$O(n\log n)$	$O(n\log n)$



插入排序算法的插入操作

输入

5	7	1	3	6	2	4
---	---	---	---	---	---	---

前面已经排好序，插入2

插入2

1	3	5	6	7	2	4
---	---	---	---	---	---	---

插入后

1	2	3	5	6	7	4
---	---	---	---	---	---	---

插入排序算法的运行实例

输入

5	7	1	3	6	2	4
---	---	---	---	---	---	---

初始

5	7	1	3	6	2	4
---	---	---	---	---	---	---

插入7

5	7	1	3	6	2	4
---	---	---	---	---	---	---

插入1

1	5	7	3	6	2	4
---	---	---	---	---	---	---

插入3

1	3	5	7	6	2	4
---	---	---	---	---	---	---

插入6

1	3	5	6	7	2	4
---	---	---	---	---	---	---

插入2

1	2	3	5	6	7	4
---	---	---	---	---	---	---

插入4

1	2	3	4	5	6	7
---	---	---	---	---	---	---


冒泡排序算法的一次巡回

输入

5	7	1	3	6	2	4
---	---	---	---	---	---	---

巡回

5	7	1	3	6	2	4
---	---	---	---	---	---	---



巡回后

5	1	3	6	2	4	7
---	---	---	---	---	---	---

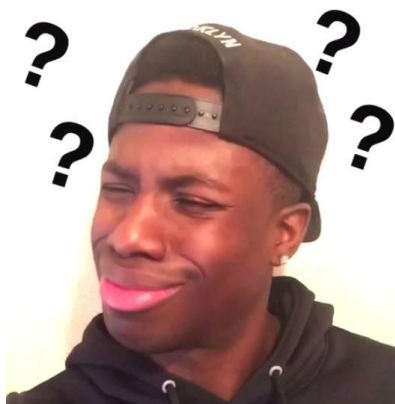
冒泡排序算法的运行实例

输入	5	7	1	3	6	2	4
巡回1	5	1	3	6	2	4	7
巡回2	1	3	5	2	4	6	7
巡回3	1	3	2	4	5	6	7
巡回4	1	3	2	4	5	6	7
巡回5	1	2	3	4	5	6	7
巡回6	1	2	3	4	5	6	7

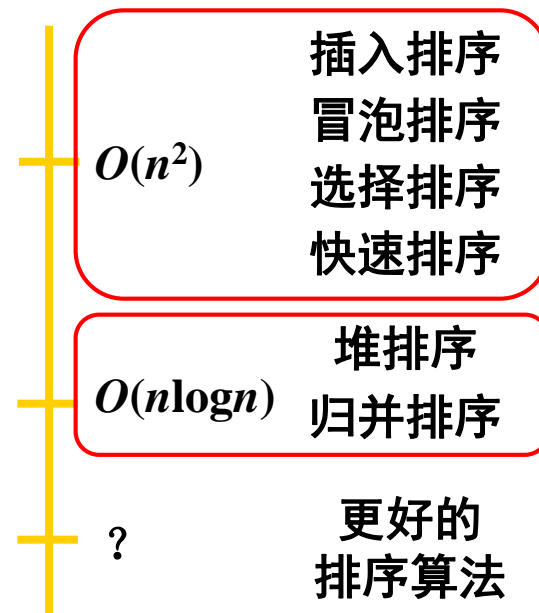
问题的计算复杂度分析

■ 问题

- 哪个排序算法效率最高？
- 是否可以找到更好的排序算法？
- 排序问题计算难度如何？
- 问题计算复杂度的估计方法



哪个排序算法效率最高？
如何分析排序问题计算难度？



评估算法的执行效率

■ 两种评估方法：

- **经验(Empirical)**：对各种算法编程，用不同实例进行实验；
- **理论(Theoretical)**：以数学化的方式确定算法所需要资源数与实例大小之间函数关系。

算法效率 → 算法的快慢

★ 时间/空间

某些方法实例中某些构件数的数量

- 排序：以参与排序的项数表示实例大小；
- 讨论图时，常用图的节点/边来表示

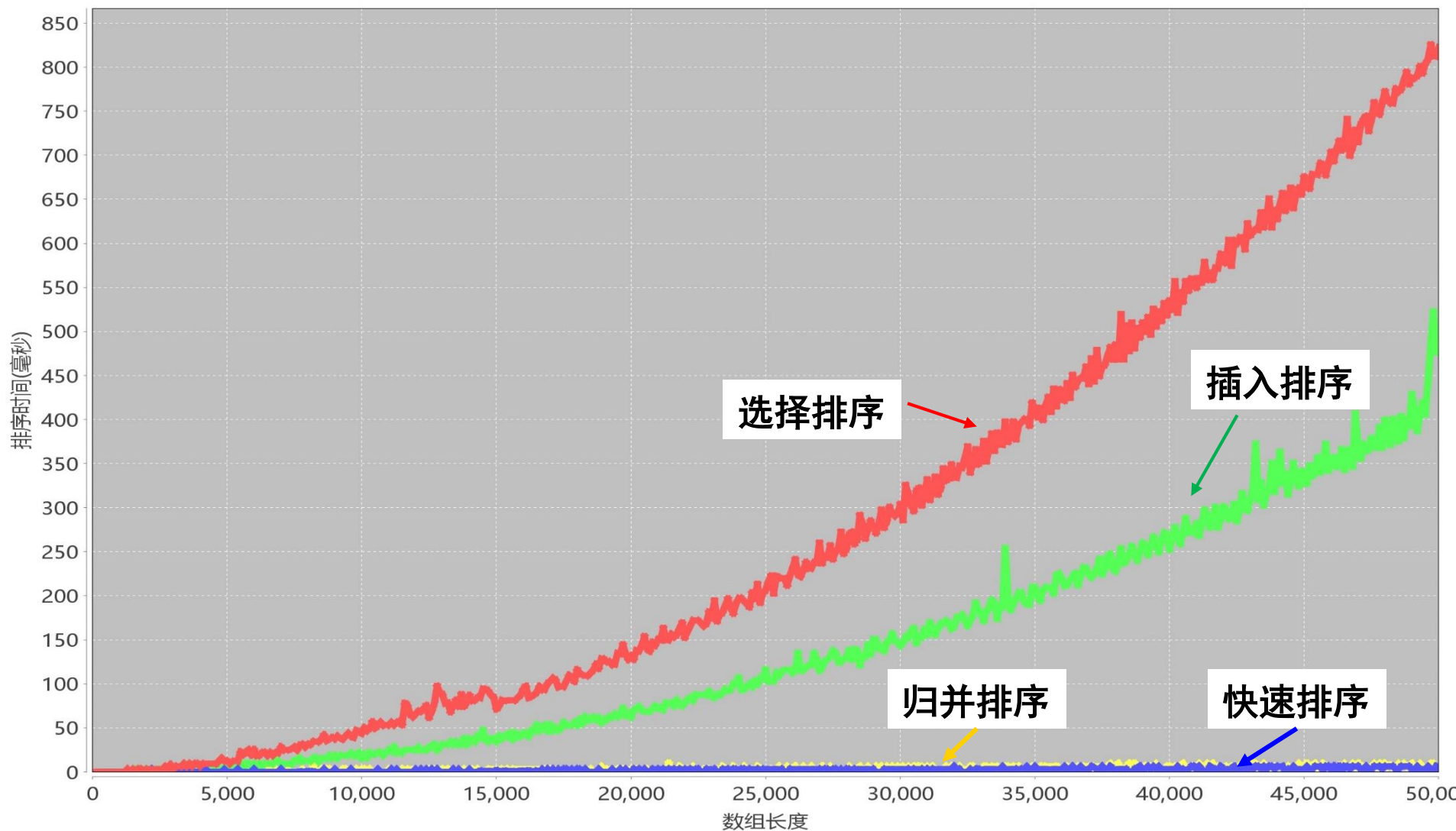


基于经验的评估方法

- 实验方案
 - 生成 n 个随机数并进行排序
($n=100, 200, \dots, 25000$)
 - 记录各个算法的排序时间
 - 用图的形式画出来

基于经验的评估方法

排序算法执行时间比较图



经验法存在的问题

- 经验法的问题
 - 依赖于计算机
 - 依赖于语言/编程技能
 - 需要一定的编程/调试时间
 - 只能评估部分实例的效率



理论法优点：既不依赖于计算机，也不依赖于语言/编程技能。节省了无谓编程时间；可研究任何在实例上算法效率

基于理论的评估方法

■ 执行时间的估计

- 定义评估函数 $T(n)$, n 为输入数据规模
- 如果存在一个正的常数 c , 而该算法对每个大小为 n 的实例的执行时间都不超过 $cT(n)$ 秒
- → 该算法的开销在 $T(n)$ 级内。

为什么定义常数 c ?



Apple I
CPU MOS 6502
@ 1 MHz



2017

iMac Pro
CPU Intel Xeon W
@ 3.2GHz 八核



平均和最坏情况分析

- 一个算法的时间耗费，对于不同的实例都会有所不同。可以定义两种时间复杂度：

- 最坏情况下的时间复杂度 $W(n)$

算法求解输入规模为 n 的实例所需要的最长时间

- 平均情况下的时间复杂度 $A(n)$

在给定同样规模为 n 的输入实例的概率分布下，算法求解这些实例所需要的平均时间



平均和最坏情况分析

■ $A(n)$ 的计算公式

- 设 S 是规模为 n 的实例集
- 实例 $I \in S$ 的概率是 P_I
- 算法对实例 I 执行的基本运算次数是 t_I

$$A(n) = \sum_{I \in S} P_I \times t_I$$

在某些情况下可以假定每个输入的实例概率相等



例子：检索/查找

检索问题

- 输入：
 - 非降顺序排列的无重复数值 L ，元素数 n ，数 x
- 输出： j
 - 若 x 在 L 中， j 是 x 首次出现的下标；
 - 否则 $j=0$
- 基本运算： x 与 L 中元素的比较



顺序检索算法

- $j=1$, 将 x 与 $L[j]$ 比较,
 - 如果 $x=L[j]$, 则算法停止, 输出 j ;
 - 如果不等, 则把 j 加1, 继续比较, 如果 $j>n$, 则停机并输出0.

实例:

1	2	3	4	5
---	---	---	---	---

$x=4$, 需要比较4次

$x=2.5$, 需要比较5次



最坏情况的时间估计

- 不同的输入有 $2n+1$ 个，分别对应：
 - x 在 L 中，有 n 种情况；
 - 不在 L 中，在每个元素之间，有 $n+1$ 种情况。
- 最坏的输入：
 - $x=L[n]$ 或者不在 L 中，要做 n 次比较
- 最坏情况下时间： $W(n)=n$



平均情况的时间估计

- 输入实例的概率分布：
 - 假设 x 在 L 中的概率是 p ，且每个位置概率相等

$$A(n) = \sum_{i=1}^n i \times \frac{p}{n} + (1-p) \times n = \frac{p(n+1)}{2} + (1-p)n$$

$$\text{当 } p=1/2 \text{ 时} \quad A(n) = \frac{n+1}{4} + \frac{n}{2} \approx \frac{3n}{4}$$



思考

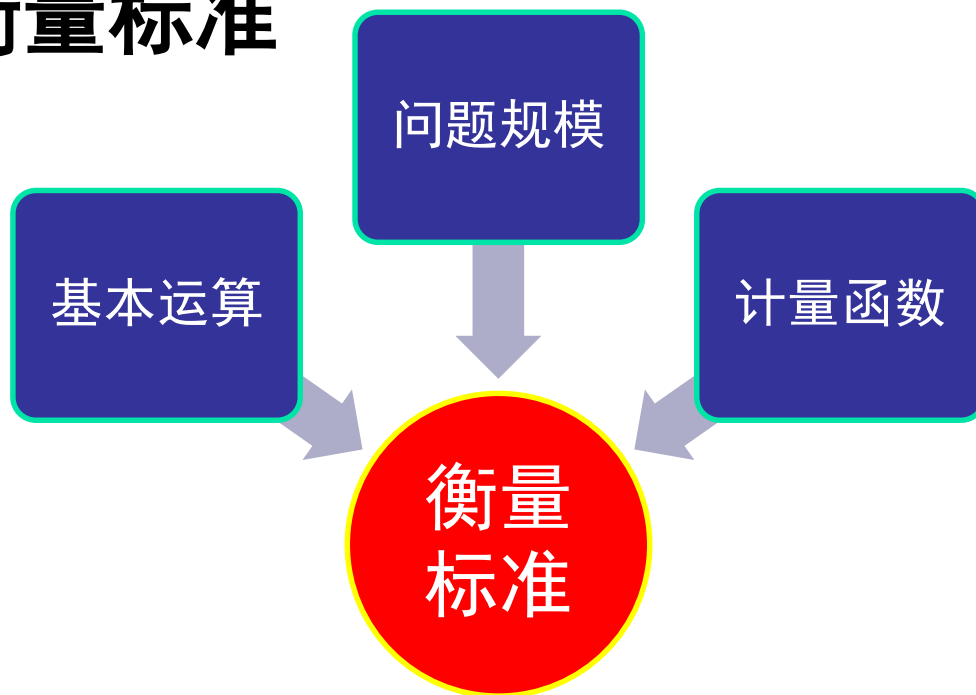
- 如何改进之前的顺序检索算法？
- 并分析其最坏情况和平均情况的时间复杂度。

顺序检索算法改进：

- $j=1$ ，将 x 与 $L[j]$ 比较，
 - 如果 $x=L[j]$ ，则算法停止，输出 j ；
 - 如果 $x>L[j]$ ，则把 j 加1，继续比较；如果 $x<L[j]$ ，则停机并输出0；
 - 如果 $j>n$ ，则停机并输出0.

算法好坏的衡量尺度

- 用所需的计算时间来衡量一个算法的好坏，不同的机器相互之间无法比较。
 - 能否有一个独立于具体计算机的客观衡量标准。
- 常见的衡量标准





问题的规模

- 问题的规模：一个或多个整数，作为输入数据量的测度。
 - 数表的长度(数据项的个数)，(问题：在一个长度为 n 的数组中寻找元素 x)；
 - 矩阵的最大维数(阶数) (问题：求两个实矩阵相乘的结果)
- 输入规模通常用 n 来表示，也可有两个以上的参数
 - 图中的顶点数和边数(图论中的问题)

基本运算

- 概念：

- 也称为“元运算”

- 指执行时间可以被一个常数限定，只与**环境**有关。

- 因此，分析时只需要关心执行的基本运算次数，而不是它们执行确切时间。

- 例子：

a 次加法，一个**加** $\leq t_a$;
 m 次乘法，一个**乘** $\leq t_m$;
 s 次赋值，一个**赋值** $\leq t_s$;

$$\left. \begin{array}{l} a \text{次加法, 一个加} \leq t_a; \\ m \text{次乘法, 一个乘} \leq t_m; \\ s \text{次赋值, 一个赋值} \leq t_s; \end{array} \right\} \begin{array}{l} t \leq at_a + mt_m + st_s \\ \leq \max(t_a, t_m, t_s) \times (a + m + s) \end{array}$$

只和**基本运算**相关

机器、语言编译





基本运算

- 一般可以认为加法和乘法都是一个单位开销的运算。
 - 理论上，这些运算都不是基本运算，因为操作数的**长度**影响了执行时间。
- 实际，只要实例中操作数长度相同，即可认为是基本运算。



基本运算

■ 例如

- 在一个表中寻找数据元素 x ： x 与表中的一个项进行比较；
- 两个实矩阵的乘法：实数的乘法（及加法） $C=AB$ ；
- 将一个数表进行排序，表中的两个数据项进行比较。

- 通常情况下，讨论一个算法优劣时，我们只讨论基本运算的执行次数。

因为它是占支配地位的，其他运算可以忽略。

怎样提高效率

- 硬件速度增加，算法效率还那么重要吗？
 - 大小为 n 的实例的执行，需要 $10^{-4} \times 2^n$ s。则：

$$n = 10 \quad 10^{-4} \times 2^{10}$$

$$n = 20 \quad 10^{-4} \times 2^{20}$$

$$n = 30 \quad 10^{-4} \times 2^{30}$$

扩大 2^{10} 倍

扩大 $2^{10} \times 2^{10}$ 倍

一年至多解

$n=38$

-
- **假设**计算机性能提高100倍，那么需要 $10^{-6} \times 2^n$ s。
一年时间还是求不出 $n=45$ 的实例
即，新的计算机最多解决 $n+\log_2 100$ 的实例， $n+7$ 。



怎样提高效率

- **假设**改进算法，在原来计算机上，需 $10^{-2} \times n^3$ s。

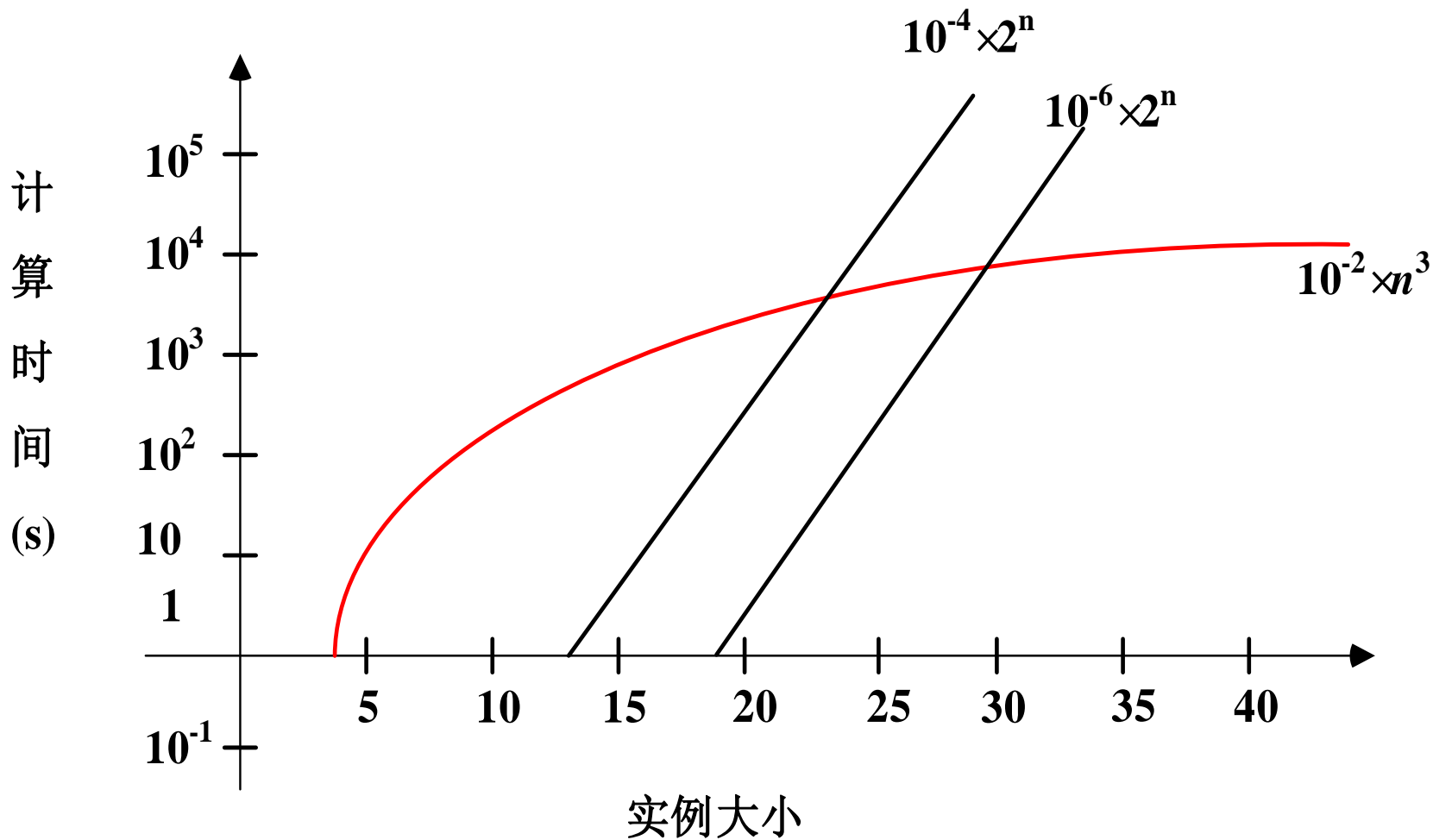
$$n = 10 \quad \approx 10s$$

$$n = 20 \quad 1 \sim 2 \text{ min}$$

$$n = 30 \quad \leq 4.5 \text{ min}$$

因此，用一天可以解决大小超过200的实例，一年处理 $n \approx 1500$ 实例。

怎样提高效率





怎样提高效率

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1,000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10,000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100,000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1,000,000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

数据来源: Princeton University 本科算法兴趣小组实验

深深地印在脑海里!



算法的计算量函数

- 用输入规模的某个函数来表示算法的基本运算量，称为**算法的时间复杂性(度)**。

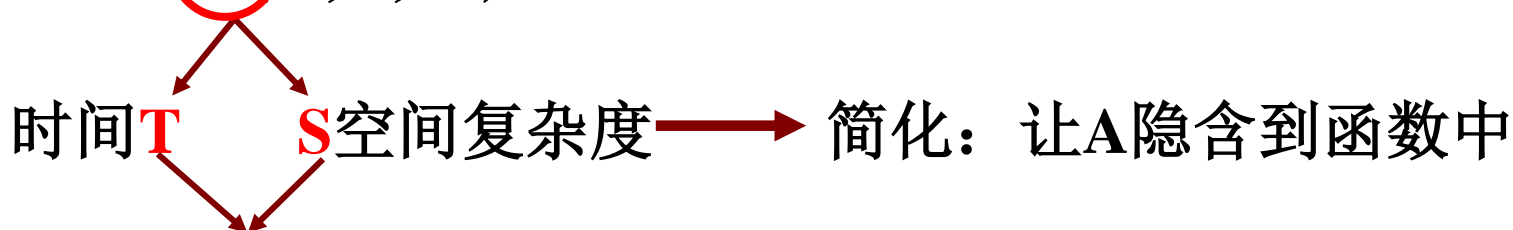
用 $T(N)$ 或 $T(N, M)$ 来表示，例如：

- $T(N)=5N$
- $T(N)=3N\log N$
- $T(N)=4N^3+N^2+1$
- $T(N)=2^N$
- $T(N, M)=2(N+M)$

算法的复杂性

- 算法的复杂性是算法运行所需计算机资源的量。
 - 需要时间的→时间复杂性 (Time Complexity)
 - 需要空间的→空间复杂性 (Space Complexity)
- 反映算法的效率，并与运算计算机独立。
- 依赖于问题的规模，输入和算法本身，用 C 表示。

$C = F(N, I, A)$ 是一个三元函数。

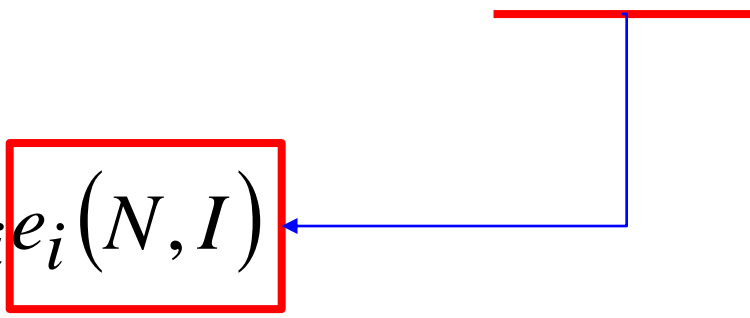


两者类似，但S简单

所以通常研究 $T(N, I)$ 在一台抽象计算机上运行所需时间。

T(N, I)的概念

- 设抽象计算机的元运算有 k 种，记为 O_1, \dots, O_k ，每执行一次所需时间为 t_1, \dots, t_k 。
- 对算法A，用到元运算 O_i 的次数为 e_i 与 N, I 相关。

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


T(N, I)的概念

- 不可能规模 N 的每种合法输入 I 都去统计 $e_i(N, I)$, 对于 I 分别考虑:

最坏情况、最好情况、平均情况

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

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

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

合法输入集



复杂性渐进性态

- 设 $T(N)$ 是前面定义的算法 A 复杂性函数。
 - N 递增到无限大, $T(N)$ 递增到无限大
 - 如存在 $\tilde{T}(N)$, 使 $N \rightarrow \infty$ 时, 有 $\frac{T(N) - \tilde{T}(N)}{T(N)} \rightarrow 0$
称 $\tilde{T}(N)$ 是 $T(N)$ 当 $N \rightarrow \infty$ 的渐进性态。
- 在数学上, $\tilde{T}(N)$ 是 $T(N)$ 当 $N \rightarrow \infty$ 的渐进表达式, 通常 $\tilde{T}(N)$ 是 $T(N)$ 中略去低阶项所留下的主项。
 - $\tilde{T}(N)$ 比 $T(N)$ 简单。



复杂性渐进性态

- 例如： $T(N) = 3N^2 + 4N \log N + 7$

$$\tilde{T}(N) = 3N^2$$

- 由 $\frac{T(N) - \tilde{T}(N)}{T(N)} = \frac{4N \log N + 7}{3N^2 + 4N \log N + 7} \rightarrow 0$

- 因为 $N \rightarrow \infty$, $T(N) \rightarrow \tilde{T}(N)$

- 所以有理由用 $\tilde{T}(N)$ 来替代 $T(N)$ 来度量A。



衡量算法的效率

- 当比较两个算法的渐近复杂性的阶不同时，只要确定各自的阶，即可判定哪个算法效率高。

等价于

- 只要关心 $\tilde{T}(N)$ 的阶即可，不必考虑其中常数因子。
- 简化算法复杂性分析的方法和步骤，只要考察问题的规模充分大时，算法复杂性在渐近意义下的阶。

回顾一下排序算法性能比较

类别	排序方法	时间复杂度			空间复杂度	稳定性
		平均时间	最好时间	最坏时间	辅助存储	
插入排序	直接插入	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	希尔排序	$O(n \log n)$	$O(n)$	$O(n^s)$	$O(1)$	不稳定
选择排序	简单选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
	堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	不稳定
交换排序	冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	不稳定
归并排序		$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	稳定
基数排序		$O(d(r + n))$	$O(d(r + n))$	$O(d(r + n))$	$O(rd + n)$	稳定
注：希尔排序 s 是所选分组($1 < s < 2$); 基数排序中, r 和 d 分别为关键字的基数和长度						



算法分析与设计

Analysis and Design of Algorithm

Lesson 02



要点回顾

- 基础知识

- 算法和程序的概念及其区别与联系
- 算法的作用和描述方法

- 算法设计

- 举了两个例子说明算法设计的一般过程
 - 抽象为数学模型
 - 写出具体求解过程

- 算法分析

- 算法效率
 - 基于经验的评估方法
 - 基于理论的评估方法



复杂性渐进性态

- 例如： $T(N) = 3N^2 + 4N \log N + 7$

$$\tilde{T}(N) = 3N^2$$

- 由 $\frac{T(N) - \tilde{T}(N)}{T(N)} = \frac{4N \log N + 7}{3N^2 + 4N \log N + 7} \rightarrow 0$

- 因为 $N \rightarrow \infty$, $T(N) \rightarrow \tilde{T}(N)$

- 所以有理由用 $\tilde{T}(N)$ 来替代 $T(N)$ 来度量A。

回顾一下排序算法性能比较

类别	排序方法	时间复杂度			空间复杂度	稳定性
		平均时间	最好时间	最坏时间	辅助存储	
插入排序	直接插入	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	希尔排序	$O(n \log n)$	$O(n)$	$O(n^s)$	$O(1)$	不稳定
选择排序	简单选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
	堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	不稳定
交换排序	冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	不稳定
归并排序		$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	稳定
基数排序		$O(d(r + n))$	$O(d(r + n))$	$O(d(r + n))$	$O(rd + n)$	稳定
注：希尔排序 s 是所选分组($1 < s < 2$);基数排序中, r 和 d 分别为关键字的基数和长度						



渐近分析的记号

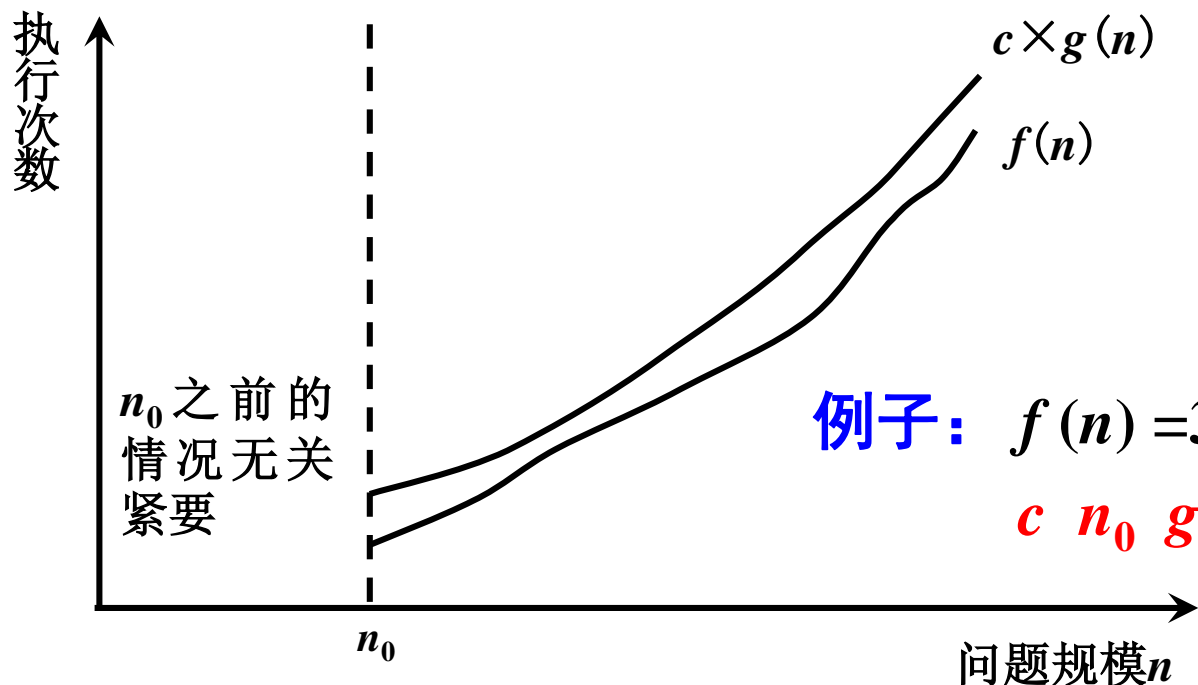
- 渐近上界记号 O
- 渐近下界记号 Ω
- 紧渐近界记号 Θ
- 非紧上界记号 o
- 非紧下界记号 ω

下面的讨论中，对所有 n ， $f(n) \geq 0$ ， $g(n) \geq 0$

渐近分析的记号

■ 渐近上界记号 O

- 若存在两个正的常数 c 和 n_0 ，使得对所有 $n \geq n_0$ ，都有： $f(n) \leq c \times g(n)$ ，则称 $f(n) = O(g(n))$



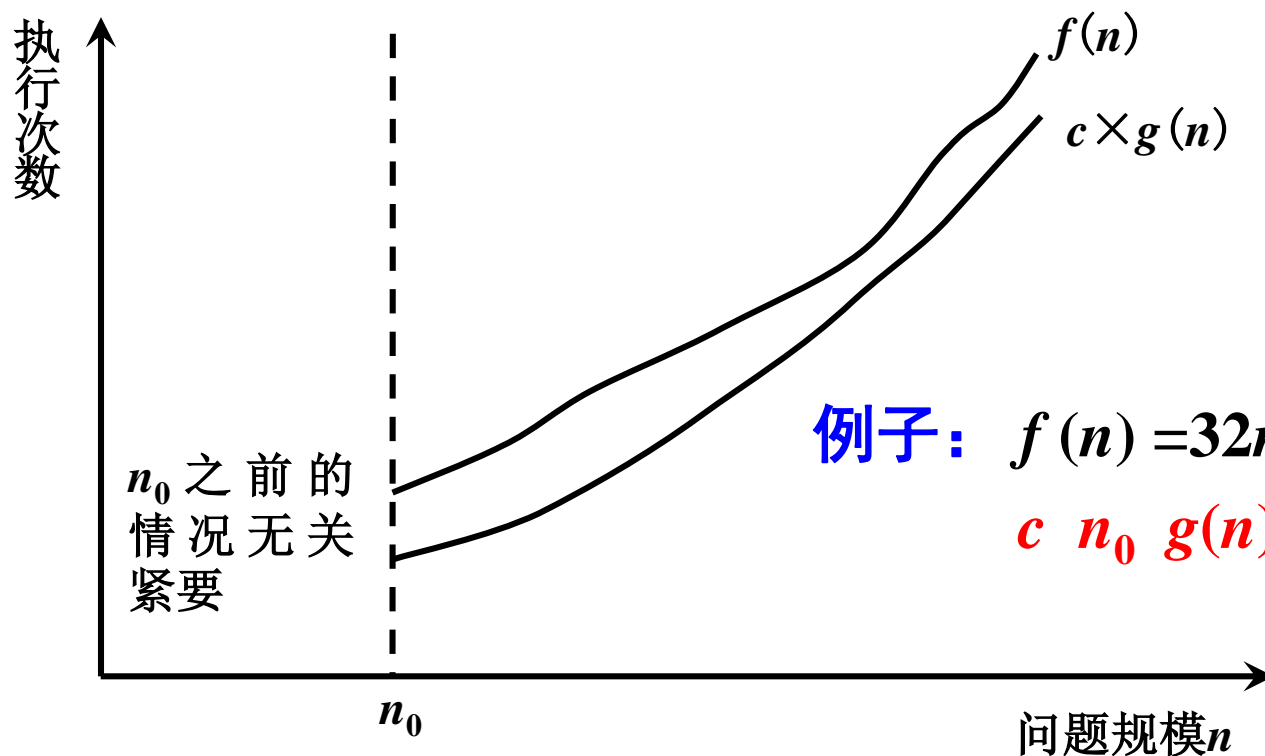
例子： $f(n) = 32n^2 + 17n + 1$

$c \quad n_0 \quad g(n) = ?$

渐近分析的记号

■ 渐近下界记号 Ω

- 若存在两个正的常数 c 和 n_0 ，使得对所有 $n \geq n_0$ ，都有： $f(n) \geq c \times g(n)$ ，则称 $f(n) = \Omega(g(n))$

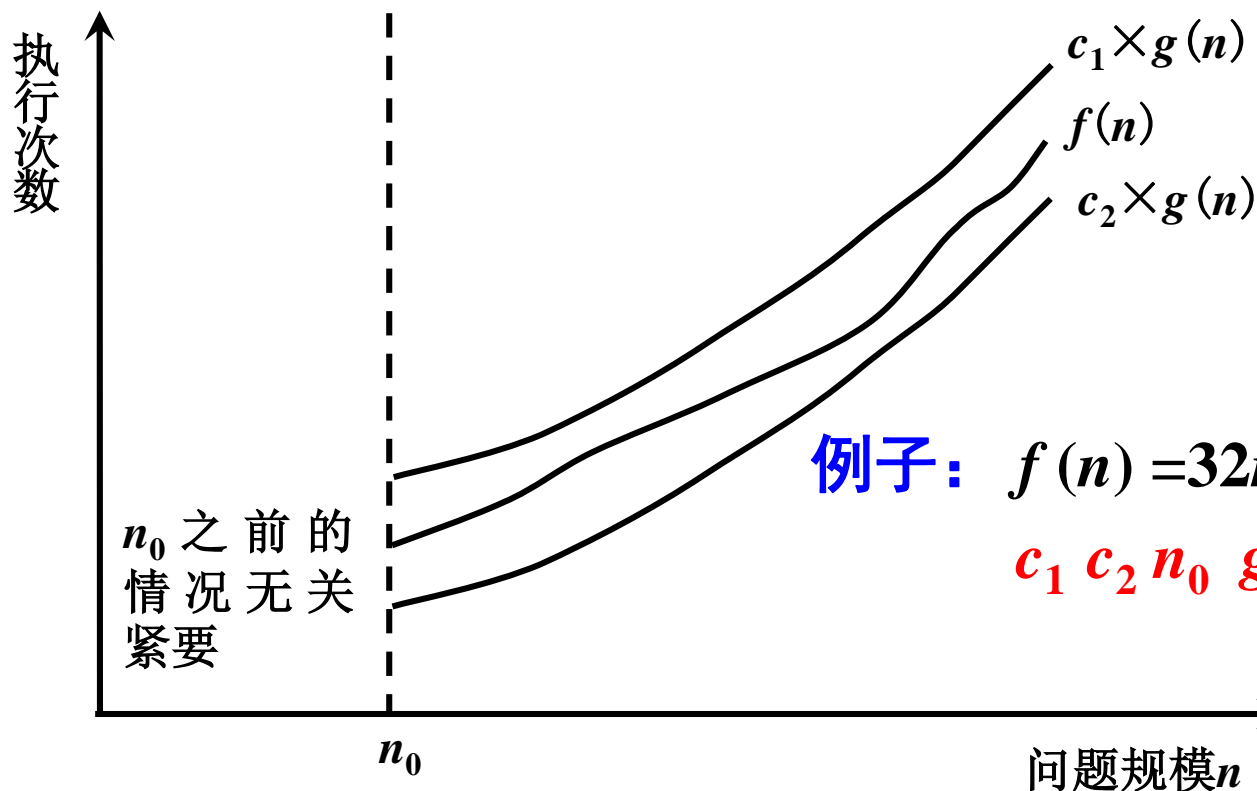


例子： $f(n) = 32n^2 + 17n + 1$
 $c \ n_0 \ g(n) = ?$

渐近分析的记号

■ 紧渐近界记号 Θ

- 若存在三个正的常数 c_1 、 c_2 和 n_0 ，使得对所有 $n \geq n_0$ ，都有： $c_1 \times g(n) \geq f(n) \geq c_2 \times g(n)$ ，则称 $f(n) = \Theta(g(n))$



例子： $f(n) = 32n^2 + 17n + 1$
 $c_1 \ c_2 \ n_0 \ g(n) = ?$



渐近分析的记号

■ 非紧上界记号 o

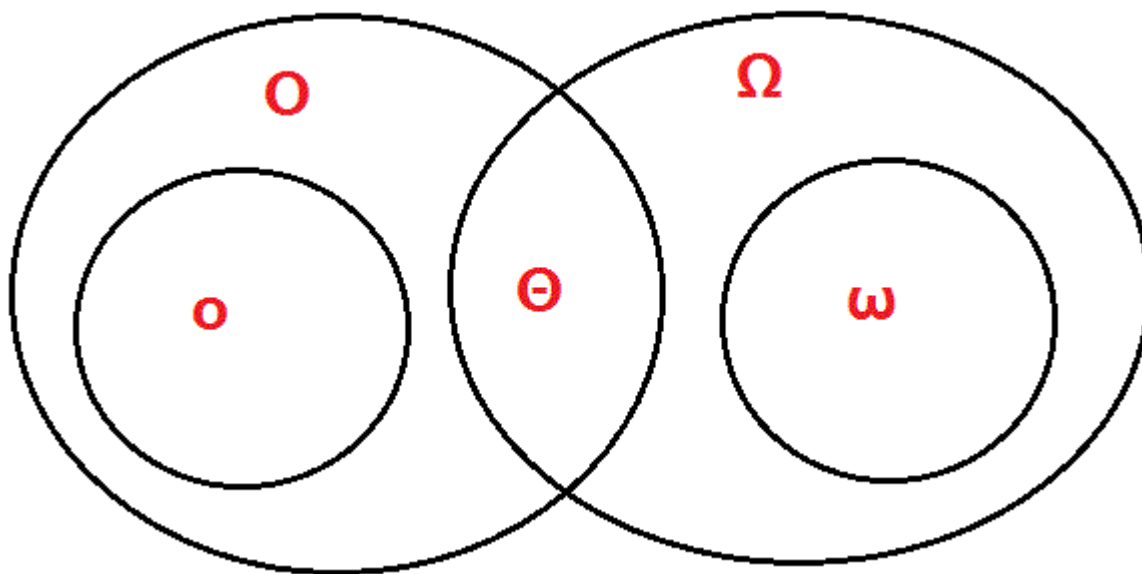
- $o(g(n)) = \{ f(n) \mid \text{对于任何正常数 } c > 0, \text{ 存在正数 } n_0 > 0 \text{ 使得对所有 } n \geq n_0 \text{ 有: } 0 \leq f(n) < cg(n) \}$
- 等价于 $f(n) / g(n) \rightarrow 0$, as $n \rightarrow \infty$ 。

■ 非紧下界记号 ω

- $\omega(g(n)) = \{ f(n) \mid \text{对于任何正常数 } c > 0, \text{ 存在正数 } n_0 > 0 \text{ 使得对所有 } n \geq n_0 \text{ 有: } 0 \leq cg(n) < f(n) \}$
- 等价于 $f(n) / g(n) \rightarrow \infty$, as $n \rightarrow \infty$ 。
- $f(n) \in \omega(g(n)) \Leftrightarrow g(n) \in o(f(n))$

渐近分析的记号

Θ (西塔)	紧确界。	相当于"="
O (大欧)	上界。	相当于" \leq "
o (小欧)	非紧的上界。	相当于"<"
Ω (大欧米伽)	下界。	相当于" \geq "
ω (小欧米伽)	非紧的下界。	相当于">"



例：渐近意义下的 O

- 如果存在正的常数 C 和自然数 N_0 ，使得当 $N \geq N_0$ 时，有 $f(N) \leq C \times g(N)$ ，则称函数 $f(N)$ 当 N 充分大时上有界，且 $g(N)$ 是它的一个上界，记为 $f(N) = O(g(N))$ 。
- 也即 $f(N)$ 的阶不高于 $g(N)$ 的阶。
 - $\forall N \geq 1, 3N \leq 4N \Rightarrow 3N = O(N)$ ；
 - $\forall N \geq 1, N + 1024 \leq 1025N \Rightarrow N + 1024 = O(N)$ ；
 - $\forall N \geq 10,$
 $2N^2 + 11N - 10 \leq 3N^2 \Rightarrow 2N^2 + 11N - 10 = O(N^2)$ ；
 - $\forall N \geq 1, N^2 \leq N^3 \Rightarrow N^2 = O(N^3)$ ；
 - $N^3 \neq O(N^2)$ ，无 $N \geq N_0$ 使得 $N^3 \leq CN^2 \quad N \leq C$ ；

反例



O 的运算性质

- $O(f) + O(g) = O(\max(f, g))$
- $O(f) + O(g) = O(f + g)$
- $O(f) \cdot O(g) = O(f \cdot g)$
- 如果 $g(N) = O(f(N)) \Rightarrow O(f) + O(g) = O(f)$
- $O(cf(N)) = O(f(N))$ 其中 c 是一个正的常数
- $f = O(f)$

下面考察性质的证明：



性质： $O(f) + O(g) = O(\max(f, g))$

证明： 设 $F(N) = O(f)$ 。根据符号 O 的定义，存在正常数 C_1 和自然数 N_1 ，使对所有 $N \geq N_1$ ，都有 $F(N) \leq C_1 f(N)$ 。

类似地， 设 $G(N) = O(g)$ ，则存在正的常数 C_2 和自然数 N_2 ， $N \geq N_2$ 有 $G(N) \leq C_2 g(N)$ 。



性质: $O(f) + O(g) = O(\max(f, g))$

令
$$\left. \begin{aligned} C_3 &= \max\{C_1, C_2\} \\ N_3 &= \max\{N_1, N_2\} \\ h(N) &= \max\{f, g\} \end{aligned} \right\} \quad \begin{aligned} &\forall N > N_3 \\ &F(N) \leq C_1 f(N) \leq C_1 h(N) \leq C_3 h(N) \\ &G(N) \leq C_2 g(N) \leq C_2 h(N) \leq C_3 h(N) \end{aligned}$$

同理 $G(N) \leq C_2 g(N) \leq C_2 h(N) \leq C_3 h(N)$

所以
$$\begin{aligned} O(f) + O(g) &= F(N) + G(N) \leq C_3 h(N) + C_3 h(N) \\ &= 2C_3 h(N) = O(h) = O(\max(f, g)) \end{aligned}$$

证毕!

NP完全性理论



重要的问题类型

- **查找/检索问题**

- 在给定的集合中寻找一个给定的值

- **排序问题**

- 按升序（降序）重新排列给定列表中的数据项

- **图问题**

- 图的遍历、最短路径以及有向图的拓扑排序

- **组合问题**

- 寻找一个组合对象（排列或子集）满足一些重要特性

- **几何问题**

- 处理类似于点、线、多面体这样的几何对象



问题的复杂度

当为某一问题设计算法时，我们总是追求最好的复杂度。但是，怎样才能知道已达到最佳呢？我们必需考虑问题的复杂度。

- 问题的复杂度就是**任一个**解决该问题的算法所必需的运算次数。

例如，任何一个采用比较大小的办法将 n 个数排序的算法需要至少 $\lg n!$ 次比较才行。那么 $\lg n!$ 就是（基于比较的）排序问题的复杂度。因为没有有一个算法可用少于 $\lg n!$ 次比较解决排序问题， $\lg n!$ 就成了算法复杂度的下界，即任一比较排序算法的复杂度必定为 $\Omega(\lg n!) = \Omega(n \lg n)$ 。

- 所以，如果能证明某个问题至少需要 $\Omega(g(n))$ 运算次数，那么 $\Omega(g(n))$ 就是所有解决该问题的算法的复杂度的下界。



问题的复杂度

- 反之，如果某一算法的复杂度是 $O(f(n))$ ，那么它解决的问题的复杂度不会超过 $O(f(n))$ 。
- 因此，任一算法的复杂度也是其所解决的问题的复杂度的上界。
- 通常在已知的某问题的复杂度下界和该问题最好算法的复杂度之间存在距离，算法研究的任务就是努力寻找更好的下界或更优的上界。

结论：找出问题的复杂度，即找出其算法的下界，是一重要的工作，因为它可以告诉我们是否还有改进当前算法的余地而省去很多徒劳无功的努力。

易解问题与难解问题

- 通常将存在多项式时间算法的问题看作是易解问题（Easy Problem）；
 - 排序问题、查找问题、欧拉回路
- 而将需要指数时间算法解决的问题看作是难解问题（Hard Problem）。
 - TSP问题、Hanio问题、Hamilton回路问题

问题规模

易解问题与难解问题

- 为什么把多项式时间复杂性作为易解问题和难解问题的分界线？

- 多项式函数与指数函数的增长率有本质的差别

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1,000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10,000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100,000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1,000,000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

不可解问题

- 不能用计算机求解(不论耗费多少时间)的问题称为**不可解问题**(Unsolvable Problem)
 - **图灵**停机问题(Turing Halting Problem)





P类问题和NP类问题

- 判定问题
- 确定性算法与P类问题
- 非确定性算法与NP类问题



判定问题

- 一个判定问题（Decision Problem）是仅仅要求回答“yes”或“no”的问题
- 判定问题的重要特性——证明比求解易
- 判定问题→语言的识别问题→计算模型



确定性算法与P类问题

- **定义1** 设A是求解问题 Π 的一个算法，如果在算法的整个执行过程中，每一步只有一个确定的选择，则称算法A是**确定性(Determinism)算法**
- **定义2** 如果对于某个判定问题 Π ，存在一个非负整数 k ，对于输入规模为 n 的实例，能够以 $O(n^k)$ 的时间运行一个确定性算法，得到yes或no的答案，则该判定问题 Π 是一个**P类(Polynomial)问题**

所有易解问题都是P类问题



非确定性算法与NP类问题

- **定义3** 设A是求解问题 Π 的一个算法，如果算法A以如下猜测并验证的方式工作，就称算法A是**非确定性(Nondeterminism)算法**
 - **猜测阶段**：在这个阶段，对问题的输入实例产生一个任意字符串y，在算法的每一次运行时，串y的值可能不同，因此，猜测以一种非确定的形式工作；
 - **验证阶段**：在这个阶段，用一个确定性算法验证：
 - 检查在猜测阶段产生的串y是否是合适的形式，如果不是，则算法停下来并得到no；
 - 如果串y是合适的形式，则验证它是否是问题的解，如果是，则算法停下来并得到yes，否则算法停下来并得到no。



非确定性算法与NP类问题

- **定义4** 如果对于某个判定问题 Π ，存在一个非负整数 k ，对于输入规模为 n 的实例，能够以 $O(n^k)$ 的时间运行一个非确定性算法，得到yes或no的答案，则该问题是一个**NP类(Nondeterministic Polynomial)问题**

关键：存在一个**确定性算法**，能够以**多项式时间**来检查和**验证**猜测阶段所产生的答案。

例如：NP类问题——Hamilton问题
Hanoi塔问题不是NP类问题



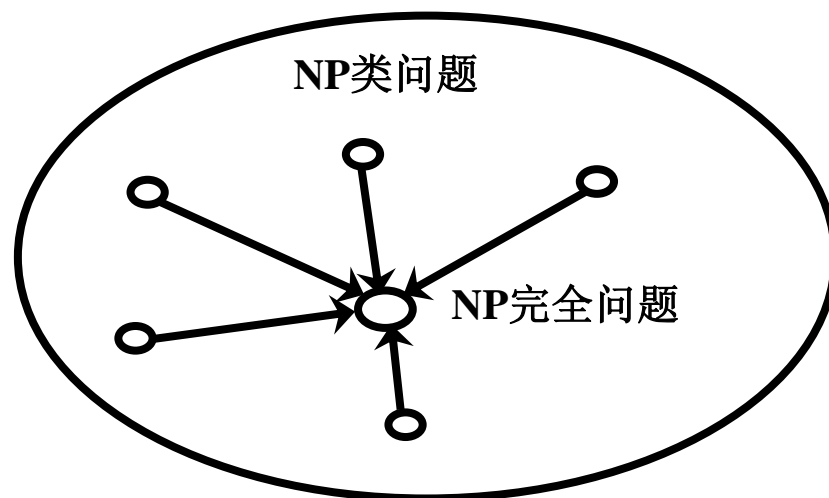
P类和NP类问题的主要差别

- P类问题可以用多项式时间的**确定性算法**来进行判定或求解；
- NP类问题可用多项式时间的**非确定性算法**来进行判定或求解。

$$P \subseteq NP$$

NP完全问题

- **定义5** 令 Π 是一个判定问题，如果问题 Π 属于NP类问题，并且对NP类问题中的每一个问题 Π' ，都有 $\Pi' \leq_p \Pi$ ，则称问题 Π 是一个**NP完全问题(NP Complete Problem)**，有时把NP完全问题记为**NPC**





一些基本的NP完全问题

- SAT问题(Boolean Satisfiability Problem)
- 最大团问题(Maximum Clique Problem)
- 图着色问题(Graph Coloring Problem)
- 哈密顿回路问题(Hamiltonian Cycle Problem)
- TSP问题(Traveling Salesman Problem)
- 顶点覆盖问题(Vertex Cover Problem)
- 最长路径问题(Longest Path Problem)
- 子集和问题(Sum of Subset Problem)



算法分析与设计

Analysis and Design of Algorithm

Lesson 03



要点回顾

■ 算法复杂度的概念

■ 时间复杂度

- 最坏情况时间复杂度 $W(n)$
- 平均情况时间复杂度 $A(n)$

■ 三大衡量标准

- 问题规模
- 基本运算
- 计量函数

■ 复杂度的渐近性态

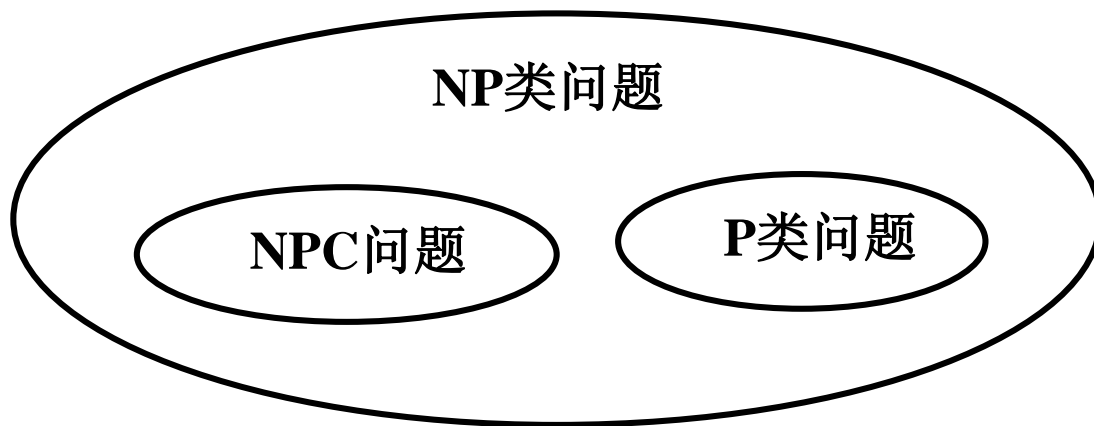
- 略去低阶项所留下的主项
- 五个渐近分析记号及其性质

1. 渐近上界记号 O
2. 渐近下界记号 Ω
3. 紧渐近界记号 Θ
4. 非紧上界记号 o
5. 非紧下界记号 ω

■ NP完全性理论介绍 什么是P? 什么是NP?

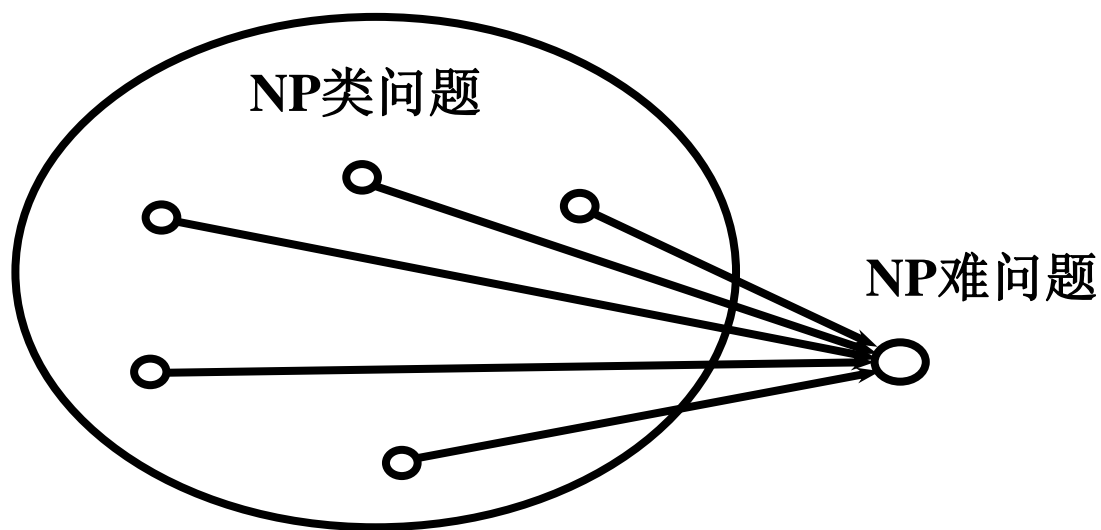
三者关系

- 目前人们猜测 $P \neq NP$ ，则P类问题、NP类问题、NP完全问题之间的关系如下：



NP难问题

- **定义6** 令 Π 是一个判定问题，如果对于NP类问题中的每一个问题 Π' ，都有 $\Pi' \leq_p \Pi$ ，则称判定问题 Π 是一个**NP难问题**。





NPC和NP难问题的区别

- 如果， Π 是NPC问题， Π' 是NP难问题，那么，他们之间的差别在于 Π 必定是NP类问题，而 Π' 不一定在NP类问题中。

一般而言，若判定问题属于NP完全问题，则相应的最优化问题属于NP难问题。

例如，判定图 $G = (V, E)$ 中是否存在哈密顿回路是NP完全问题，而求哈密顿回路中最短路径的TSP问题则是NP难问题



要点回顾

■ 第一章总结

- 理解算法的概念
- 理解算法、数据结构和程序的区别和联系
- 掌握描述算法的方法
- 掌握算法的计算复杂性概念
- 掌握算法渐近复杂性的数学表述
- 了解NP类问题的基本概念（P、NP、NPC、NPHard）

■ 算法的研究目标

- 建模—为数学问题并寻找算法（算法设计）
- 算法---算法评价，时间、空间复杂度（算法分析）
- 问题---一类算法的复杂度分析（计算复杂性理论）