

算法导论

主讲人：权丽君

Email: *ljquan@suda.edu.cn*

苏州大学 计算机学院



Do you know Algorithm 一词的由来



Algorithm(算法)一词的由来本身就十分有趣。

- 这个词一直到1957年之前在《韦氏新世界词典》(Webster's New World Dictionary)中还未出现，只能找到带有古代涵义的相近形式的一个词“**Algorism**”(算术)，指的是用阿拉伯数字进行算术运算的过程。
- 中世纪之后，对这个词的起源已经拿不准了，早期的语言学家试图推断它的来历，认为它是从把**algiros**(费力的)+**arithmos**(数字)组合起来派生而成的
- Algorism一词在历史中也经历了漫长的演变，据数学史学家发现研究，诞生于约公元前800多年。
- 但后来“algorism”的形式和意义就变得面目全非了。比如，如牛津英语字典所说明的，这个词是由于同**arithmetic**(算术)相混淆而形成的错拼词，由algorism变成algorithm。



Do you know Algorithm 一词的由来



- 一本早期的德文数学词典《数学大全辞典》(Vollständiges Mathematisches Lexicon) , 给出了Algorithmus (算法)一词的如下定义：“在这个名称之下，组合了四种类型的算术计算的概念，即加法、乘法、减法、除法”。
- 1950年左右, algorithm一词经常地同欧几里德算法(Euclid's algorithm)联系在一起。这个算法就是在欧几里德的《几何原本》(Euclid's Elements ,第VII卷) 中所阐述的求两个数的最大公约数的过程(即辗转相除法)。



为什么要学习算法



□ 算法是计算机科学的基石，是改造世界的有力工具！

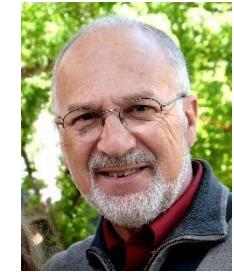
✓ 软件是计算机的灵魂，算法是计算机软件的灵魂

软件=数据结构+算法

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

✓ 算法是计算机科学的核心

the core of computer science (David Harel)



“微积分以及在微积分基础上建立起来的数学分析体系成就了现代科学，而算法则成就了现代世界”—— David Berlinski, 2000

互联网是20世纪最伟大的发明之一，改变了世界，改变了我们的生活！各种算法在支撑着整个互联网的正常运行，互联网的信息传输需要路由选择算法，互联网的信息安全需要加密算法，互联网的信息检索需要模式匹配算法，互联网的信息存储需要排序算法，……，没有算法也就没有互联网！



为什么要学习算法



- 算法是国家科技综合实力的体现
- 学习算法可以开发人们的分析能力

算法是解决问题的一类特殊方法，它是经过对问题的准确理解和定义获取答案的过程。

- 学习算法具有重要的现实意义



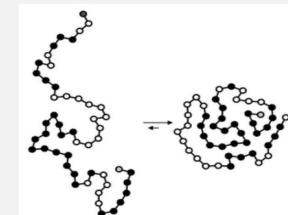


为什么要学习算法



- 当今计算机技术的进步，如大数据、AI等，算法的进步是关键。
- 算法的应用无所不在：
 - ✓ 互联网：网络搜索，数据包路由，分布式文件共享，.....
 - ✓ 生物学：人类基因组计划，蛋白质折叠，.....
 - ✓ 计算机：电路布局，文件系统，编译器，.....
 - ✓ 电脑图像：电影，电子游戏，虚拟现实.....
 - ✓ 安全：手机，电子商务，投票机，.....
 - ✓ 多媒体：MP3，JPG，DivX，高清电视，人脸识别，.....
 - ✓ 社交网络：推荐，新闻，广告，.....
 - ✓ 物理。N-body模拟，粒子碰撞模拟，.....

Google
YAHOO!
bing™





为什么要学习算法

□ 是你获得高薪职位的敲门砖！

For fun and profit.

Google™



Apple Computer

facebook

CISCO SYSTEMS



IBM

Nintendo®

JANE STREET

Morgan Stanley

NETFLIX

Adobe™

RSA SECURITY™

DE Shaw & Co

ORACLE

P
PANDORA®

Akamai

YAHOO!

amazon.com

Microsoft®

P X A R
ANIMATION STUDIOS



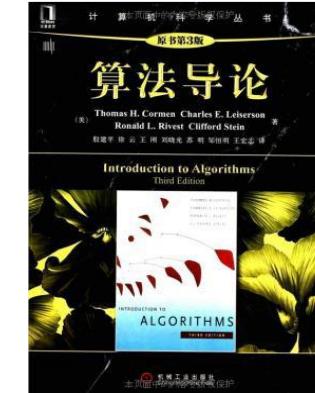


课程信息



- 课程名称：算法设计与分析
- 课程安排：34学时
- 授课时间：1-17周
- 教材信息：

- * 《算法导论》(第3版), Thomas 等著, 殷建平等译, 机械工业出版社, 2006.9
- * 《The Art of Computer Programming》, Donald E, Knuth, 1974, Turing Prize
- * 《算法图解》, Aditya Bhargava著, 袁国忠译, 人民邮电出版社, 2017
- * 《计算机算法设计与分析》(第4版), 王晓东, 电子工业出版社, 2007



读一本好书很重要



关于算法教学



□ 课程核心：

- ✓ 介绍算法设计与分析的基本理论、方法和技术，训练程序思维，奠定算法设计的基础。

□ 教学目的：

- ✓ 在理论学习上，掌握算法设计与分析的基本理论和方法，培养算法设计和算法分析的能力。
- ✓ 在实践教学上，掌握算法实现的技术、技巧，学习算法的正确性验证、效率分析、优化技术，以及算法在实际问题中的分析与应用。
- ✓ 培养独立思考和创新能力。



课程信息



□ 授课形式

- ✓ 课堂讲解（34学时）：基本理论讲解、基本方法的介绍分析；
- ✓ 课程作业：大约4~5次的作业；

□ 考核形式

- ✓ 期末考试(闭卷) 60%
- ✓ 期中考试(闭卷) 20%
- ✓ 作业+考勤 20%

□ 课程基础

- ✓ 学过数据结构
- ✓ 具备编程技能（C、C++、Python等）
- ✓ 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



教学内容



□ Part 1 基础知识

- ✓ 课程学习背景
- ✓ 算法分析基础

□ Part 2 排序和顺序统计学

- ✓ 归并排序、堆排序、快排排序
- ✓ 计数排序、基数排序、桶排序

□ Part 3 高级数据结构

- ✓ 基本数据结构、散列表、二叉搜索树
- ✓ 红黑树、数据结构的扩张

□ Part 4 算法设计策略

- ✓ 分治法、动态规划法、贪心法、回溯法、分枝限界法



第一章 算法在计算中的作用

本章尝试回答以下问题：

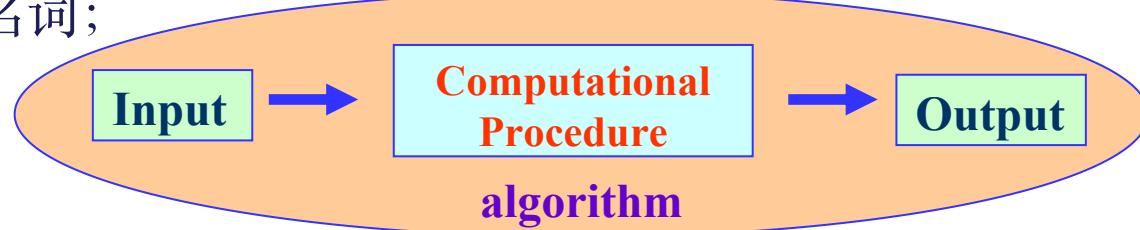
- 什么是算法？
- 为什么算法值得研究？
- 算法的作用是什么？



1.1 算法



- 简单说来，**算法**就是问题的程序化解决方案。
- **定义：**算法就是一个定义良好的可计算过程，它取一个或者一组值作为输入，并产生出一个或者一组值作为输出。即，算法就是一系列的计算步骤，用来将输入数据转换成输出结果。
- 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；



< 31,41,59,26,12,58 >



< 31,41,59,26,12,58 >

<31,41,26,12,58,59>

<31,26,12,41,58,59>

.....

<12,26,31,41,58,59>



什么是算法？



- 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。

✓ 算法由运算组成：包括算术、逻辑、关系、赋值、过程调用等运算形式

- 一个算法通常具有如下特征：

- ① 输入：一个算法具有零个或者多个取自指定集合的输入值；
- ② 输出：对算法的每一次输入，算法具有一个或多个与输入值相联系的输出值；
- ③ 确定性：算法的每一个指令步骤都是明确的；
- ④ 有限性：对算法的每一次输入，算法都必须在有限步骤（即有限时间）内结束；
- ⑤ 正确性：对每一次输入，算法应产生出正确的输出值；
- ⑥ 通用性：算法的执行过程可应用于所有同类求解问题，而不是仅适用于特殊的输入。

特殊性：解决不同问题的算法是不相同的，没有万能的算法

- 计算过程：满足确定性、可行性、输入、输出，但不一定满足有限性的一组规则称为 **计算过程**。



相关概念：问题和问题实例



- 问题 (Problem) : 规定了输入与输出之间的关系，可以用通用语言来描述；
- 问题实例: 某一个问题的实例包含了求解该问题所需的输入；
- Example:
 - ✓ 排序问题——将一系列数按非降顺序进行排序

输入: 由n个数组成的一个序列 $\langle a'_1, a'_2, \dots, a'_n \rangle$

输出: 对输入系列的一个排列(重排) $\langle a_1, a_2, \dots, a_n \rangle$ ，使得 $a'_1 \leq a'_2 \leq \dots \leq a'_n$

- ✓ 排序问题的一个**实例**:

Input: $\langle 31, 41, 59, 26, 41, 58 \rangle$ — **Output:** $\langle 26, 31, 41, 41, 58, 59 \rangle$

- ✓ 算法: 冒泡排序、插入排序、归并排序等，将完成上述的排序过程。

目标一致，但方法各异



相关概念：输入实例与问题规模



□ 输入实例：问题的具体计算例子

如，排序问题的3个输入实例：

- ① 13,5,6,37,8,92,12
- ② 43,5,23,76,25
- ③ 53,67,32,42,22,33,4,39,56

□ 问题规模：算法的输入实例大小。

如，上面排序问题的3个输入实例的规模大小分别为7,5,9



相关概念：正确算法与不正确算法



□ 正确的算法

如果一个算法对问题每一个输入实例，都能输出正确的结果并停止，则称它为正确的。

□ 不正确的算法

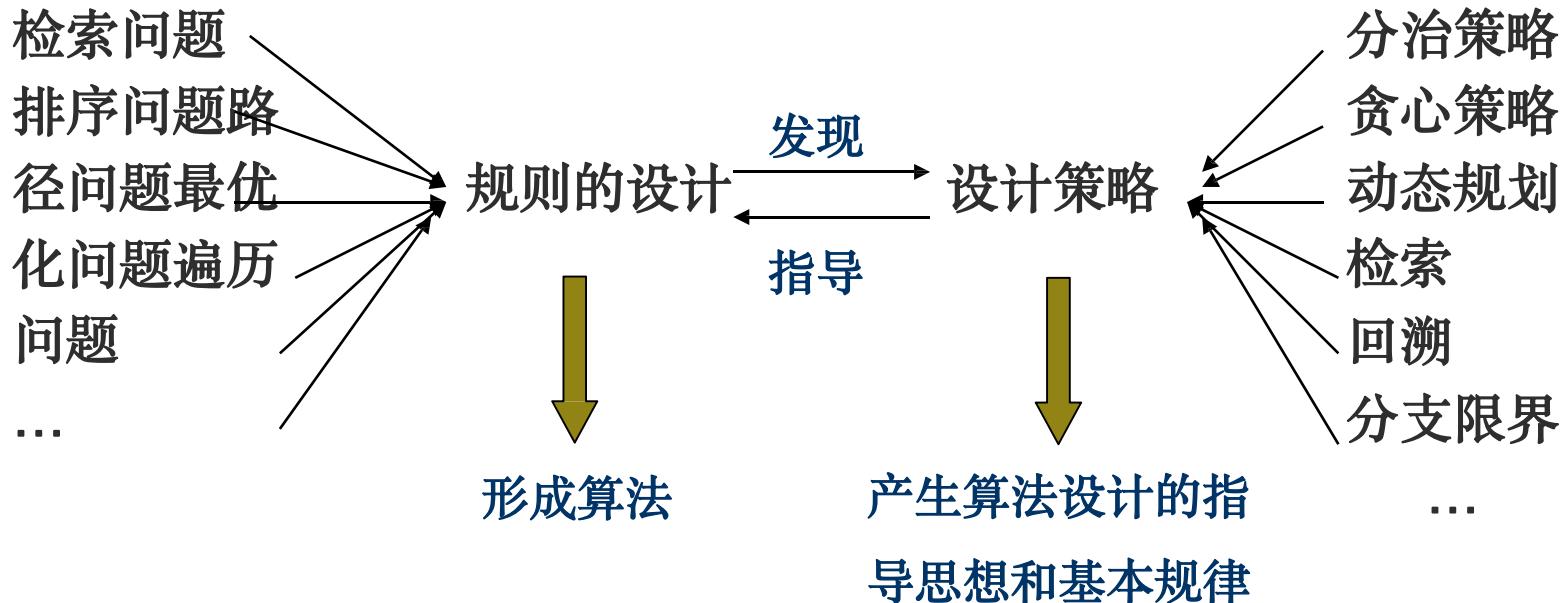
- ✓ 可能根本不会停止；
 - ✓ 停止时给出的不是预期的结果；
 - ✓ 如果算法的错误率可以控制，也是有用的。
-
- 但不正确的算法也并非绝对无用，在不正确的算法错误率可控时有可能也是有用和有效的（如NP问题和近似算法）。
 - 但通常，我们还只是关心正确的算法。



思考



- 在数据结构等课程里我们学会了解决一些具体问题的算法，如排序算法、路径算法。而在这些算法的背后有没有一些共性的东西？
- 能不能总结出来一些**规律**和**方法**，来指导新问题的求解、新算法的设计呢？





1.2作为一种技术的算法



□ 如果计算机无限快、存储器都是免费的,算法研究是否还需要?

- ① Yes ! 证明方案是正确的, 可以给出正确结果。
- ② Yes ! 希望自己的实现符合良好的软件工程实践要求, 采用最容易的实现方法。

□ 算法对于当代计算机非常重要 ! 类比硬件与软件的不同, 算法的迥异带来的意义可能更明显 ! 比如, 对100万个数字进行排序:

- ◆ 插入排序: $T(n) = c_1 n^2$
 - 计算机A: 10^9 指令/s
 - 世界最好的程序员
 - 机器语言

$$T(n) = 2n^2$$

$$t = \frac{2 \cdot (10^7)^2 \text{instruc}}{10^9 \text{instruc/s}} = 2 \times 10^5 \text{s} \approx 55.56 \text{h}$$

- ◆ 归并排序: $T(n) = c_2 n \lg n$
 - 计算机B: 10^7 指令/s
 - 普通程序员
 - 高级语言+低效编译器

$$T(n) = 50n \lg n$$

$$t = \frac{50 \cdot 10^7 \lg 10^7 \text{instruc}}{10^7 \text{instruc/s}} \approx 19.38 \text{m}$$



举例：3-SUM



- ❑ 3-SUM. Given N distinct integers, how many triples sum to exactly zero?
- ❑ 输入输出
 - ✓ For example, given array A = [30, -40, -20, -10, 40, 0, 10, 5], A solution set is:
 - ✓ (30, -40, 10) (30, -20, -10) (-40, 40, 0) (-10, 0, 10)
 - ✓ 4 triples sum to 0



```
# returns how many triples with  
# sum equal to 'sum' present in A[].  
def find3Numbers(A, N, sum):
```

```
    # Fix the first element as A[i]
```

```
    count = 0
```

```
    for i in range(0, N):
```

```
        # Fix the second element as A[j]
```

```
        for j in range(i + 1, N):
```

```
            # Now look for the third number
```

```
            for k in range(j + 1, N):
```

```
                if A[i] + A[j] + A[k] == sum:
```

```
                    count+=1
```

```
    return count
```

穷举算法

Check each triple

```
# Driver program to test above function
```

```
A = [30, -40, -20, -10, 40, 0, 10, 5]
```

```
sum = 0
```

```
arr_size = len(A)
```

```
print(str(find3Numbers(A, arr_size, sum)))
```

```
+ " triples sum to "+str(sum)+"!")
```



经验性分析

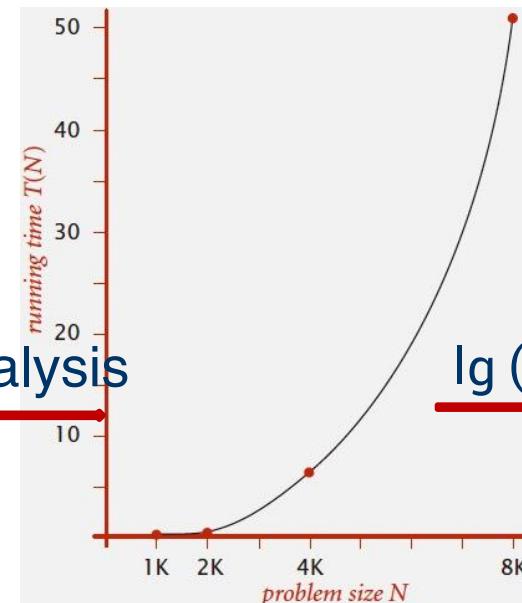
† Running OS X on Macbook

Pro 2.2GHz with 16GB RAM

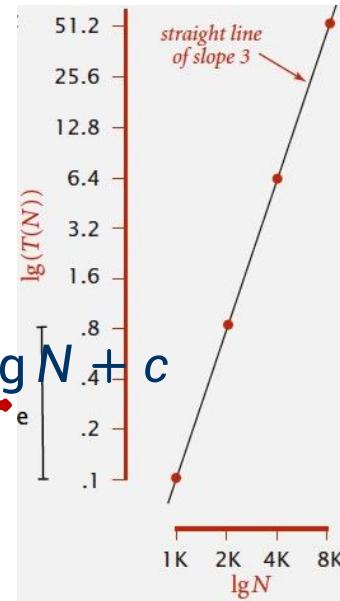
Run the program for various input sizes and measure running time.

N	Time (seconds)	Ratio	lg ration
250	0.61		
500	4.83	7.9	2.98
1,000	39.16	8.1	3.02
2,000	338.74	8.65	3.11
4,000	2934.28	8.66	3.11

Data analysis



$$\lg(T(n)) = b \lg N + c$$



seems to converge to a constant $b \approx 3$

$$\frac{T(2N)}{T(N)} = \frac{a(2N)^b}{aN^b} = 2^b$$

† plot. Plot running time $T(N)$ vs. input size N

Regression. Fit straight line through data points: $T(n) = aN^b$, where $a = 2^c$.



N^2 算法 for 3-Sum

Algorithm.

- Step 1: Traverse the array from start to end. (loop counter i).
- Step 2: Create a HashMap or set to store unique pairs.
- Step 3: Run another loop from $i+1$ to end of the array. (loop counter j)
- Step 4: If there is an element in the set which is equal to $x - arr[i] - arr[j]$, then count++ and break
- Step 5: Insert the jth element in the set.

```
def find3Numbers(A, N, sum):
```

```
    count=0
    for i in range(0, N):
        # Find pair in subarray A[i + 1..n-1]
        # with sum equal to sum - A[i]
        s = set()           ← Auxiliary Space: O(N)
        curr_sum = sum - A[i]
        for j in range(i + 1, N):
            if (curr_sum - A[j]) in s:
                count+=1
                s.add(A[j])   "inner loop"
    return count
```

$$T(N) \sim \binom{N}{2} \sim N^2$$

N	Time (seconds)	Ratio	Ig ration
250	0.01	4	2
500	0.04	3.5	1.81
1,000	0.14	4.14	2.05
2,000	0.58	4.03	2.01
4,000	2.34		

† Running OS X on Macbook Pro 2.
with 16GB RAM



3-SUM算法比较

Hypothesis. The HashMap-based N^2 algorithm for 3-SUM is significantly faster in practice than the brute-force N^3 algorithm.

N	Time1 (seconds)	Time2 (seconds)
250	0.61	0.01
500	4.83	0.04
1,000	39.16	0.14
2,000	338.74	0.58
4,000	2934.28	2.34
8,000		9.32
16,000		37.42

† Time1 for brute-force algorithm and Time2 for HashMap-based algorithm.
Running OS X on Macbook Pro 2.2GHz with 16GB RAM

Guiding principle. Typically, better order of growth \Rightarrow faster in practice.



第二章 算法基础

内容提要：

□ 本章将介绍一个贯穿整个课程、进行算法设计与分析的框架。

□ 算法分析基础

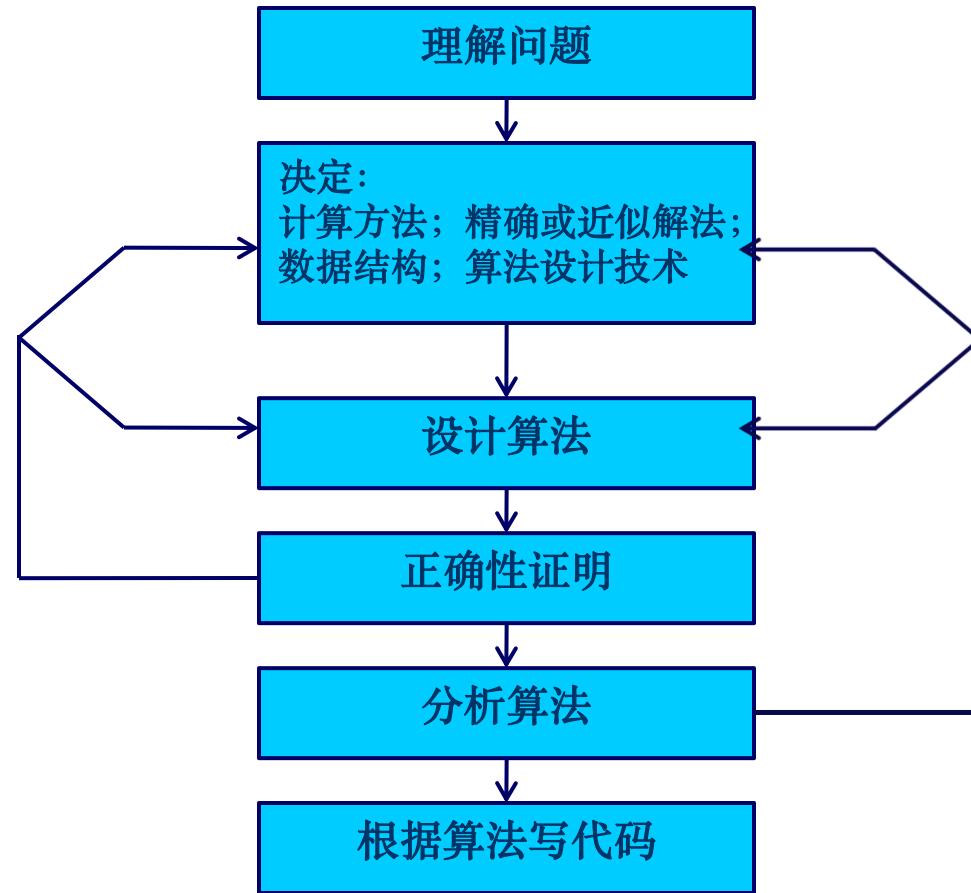
- ✓ 问题求解基础
- ✓ 算法描述方法
- ✓ 算法正确性证明
- ✓ 算法分析基本框架
- ✓ 举例：插入排序

□ 算法设计策略之——分治法



问题求解基础

□ 算法设计和分析过程





问题求解基础



□ 求解过程说明：

- ✓ 理解问题：是否属于已知问题？确定算法输入范围；
- ✓ 了解计算设备的性能：清楚速度和计算资源的限制；
- ✓ 在精确解法和近似解法之间做选择：有时近似算法是唯一选择；
- ✓ 确定适当的数据结构
- ✓ 算法的设计技术：这是本课程学习重点和目标；
- ✓ 算法的描述：自然语言、流程图、伪代码；
- ✓ 算法的正确性证明：证明对于所有合法输入均能产生正确结果；
- ✓ 算法的分析：分析执行效率（**时间和空间**）、简单性、一般性；
- ✓ 为算法写代码：利用编程语言以计算机程序行使实现；



算法描述方法



- 伪代码拥有自然语言和类编程语言特性，经常被用于算法描述；
- 与真实代码(real code)的差异：
 - ✓ 对特定算法的描述更加的清晰与精确；
 - ✓ 不需要考虑太多技术细节（数据抽象、模块、错误处理等）；
 - ✓ 用伪代码可以体现算法本质；
 - ✓ 永远不会过时；



算法描述方法

□ 伪代码的一些约定：

- ✓ 书写上的“缩进”(缩排)表示程序中的分程序（程序块）结构；
- ✓ 循环结构(while, for, repeat) 和条件结构 (if, then, else) 与Pascal, C语言类似；
- ✓ “// ” or “ ” 来表示注释；
- ✓ 利用 $i \leftarrow j \leftarrow e$ 来表示多重赋值，等价于 $j \leftarrow e$ 和 $i \leftarrow j$.
- ✓ 变量是局部于给定过程的。
- ✓ 数组元素的访问方式: $A[i]$; $A[1 .. j] = < A[1], A[2], \dots, A[i] >$
- ✓ 符合数据一般组织成对象，由属性 (attribute) 或域(field)所组成；域的访问是由域名后跟方括号括住的对象名形式来表示, 如 $\text{length}[A]$;
- ✓ 参数采用按值传递方式；
- ✓ 布尔操作 “and” 和“or”具有短路能力: 如 “ x and (or) y ”: 无论 y 的值如何，必须首先计算 x 的值。



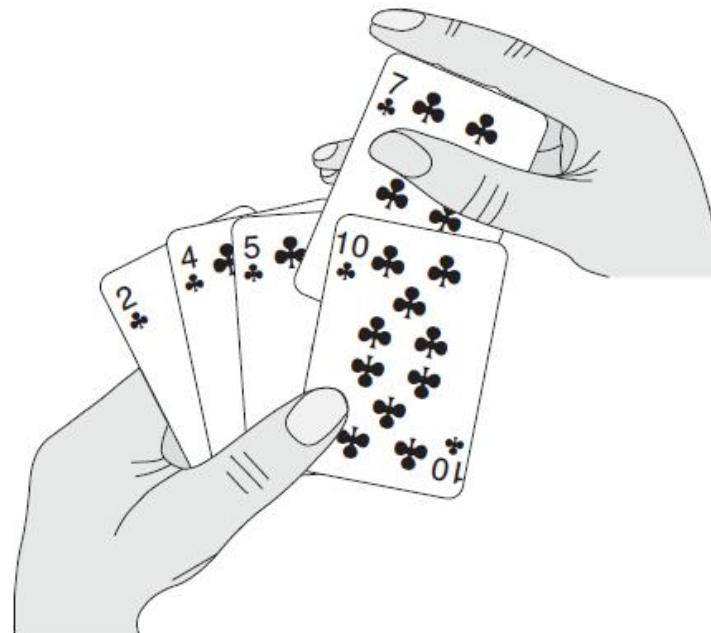
插入排序



问题: 把一系列数据按非递增的顺序排列

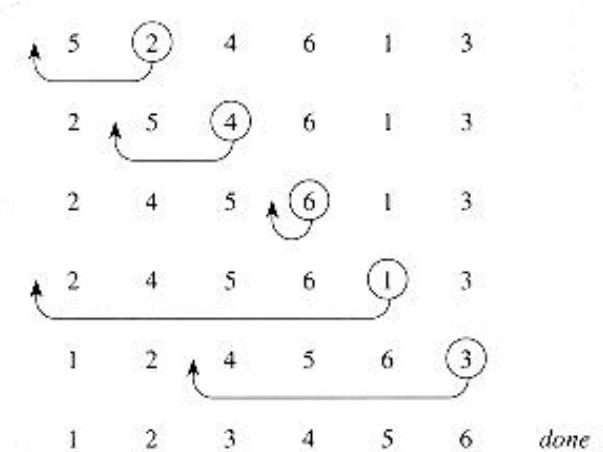
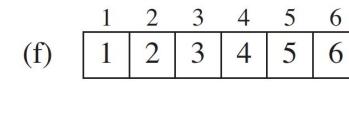
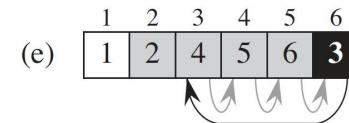
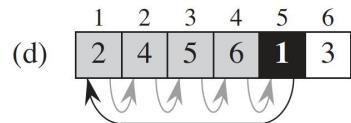
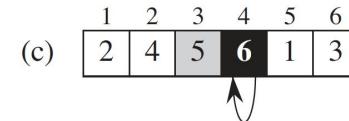
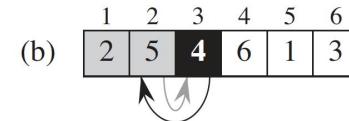
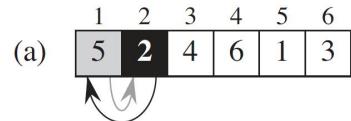
输入: n 个输入数 $\langle a_1, a_2, \dots, a_n \rangle$

输出: 输入系列的一个排序 $\langle a'_1, a'_2, \dots, a'_n \rangle$ ，使得 $a'_1 \leq a'_2 \leq \dots \leq a'_n$





插入排序



INSERTION-SORT(A)

```

1   for( $j = 2; j <= \text{length}[A]; j++$ ) // loop header
2   {
3       key =  $A[j]$ 
4       // Insert  $A[j]$  into the sorted sequence  $A[1 .. j-1]$ 
5       i ← j-1
6       while(  $i > 0 \&& A[i] > key$ )
7           {
8                $A[i+1] = A[i]$ 
9               i = i-1
10            }
11        $A[i+1] = key$ 
12   } // loop body below
    
```

j : 正要被插入的数
 $A[1..j - 1]$: 子数组构成了当前已排好
 $A[j + 1..n]$: 未排序的

例如:
 $A = \{5, 2, 4, 6, 1, 3\}$



使用循环不变式证明算法的正确性



□ 循环不变式 (Loop invariants)

- ✓ 以`INSERTION-SORT`为例，其`for`循环中，循环变量为`j`，循环过程具有以下性质 (`j`是当前正要被插入到序列中的元素所在的位置下标)：
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，而且每次循环之后 (`j`加了1)、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、以后每次循环之后还成立的关系称为“**循环不变关系**”或“**循环不变式**”、“**循环不变性质**”。



使用循环不变式证明算法的正确性



□ 循环不变式(loop Invariant)方法:

- ✓ 先确定算法的循环不变式；
- ✓ 然后检查此循环不变式在算法执行期间是否满足三个条件。

□ 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循环之后终止。

□ 证明循环正确性的思路：

- ✓ 第1) 和2) 步类似于数学归纳法的证明策略；
- ✓ 第3) 步保证算法可以终止；
- ✓ 如果1) ~3) 都满足，则说明循环过程正确。
(为什么？有理论保证，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。

0) 插入排序for循环的循环不变式可以描述为：

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

1) 初始话：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迭代之前循环不变式成立——初始状态成立。

2) 保持：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，这里不另行证明），直到找到对当前A[j]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]”变成新的A[1..j-1]。故循环不变式依然成立。



利用循环不变关系证明插入排序的正确性。

3) 终止: 循环可以有限次终止

可以看到, 每次循环后 j 都加1, 而循环终止条件是 $j > n$ (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) 。

4) 故此, 上述三条性质都成立, 根据证明策略, 插入排序算法是正确的一一确切地说是其中的for循环正确, 但for循环是插入排序过程的主体, 所以整个算法正确。



算法分析



□ 分析算法目的主要是预测算法需要资源的程度。

□ 资源包括：

- ✓ 时间
- ✓ 空间：内存
- ✓ 其他：通信带宽、硬件资源等

➤ 但从算法的角度，我们主要关心算法的时间复杂度和空间复杂度。

□ 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需求是不同的。
- 因此，通过对算法的分析可以了解算法的性能，从而在解决同一问题的不同算法之间比较性能的好坏，选择好的算法，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。



1 算法分析重要的假设



- 算法分析是指对一个算法所需要的资源进行预测，通常是对**计算时间和空间**的预测。算法分析的目的是为了从多个候选算法中选择一个最有效的算法，或去掉较差的算法。
- 进行算法分析之前，首先要确立有关实现技术的模型，通常采用**随机存取机 (random-access machine: RAM)** 计算模型。假设：
 - ✓ 指令时逐条执行的，没有并发操作；
 - ✓ 包含常用指令，每条指令执行时间为常量；
 - ✓ 数据类型有整数类型和浮点实数类型
 - ✓ 不对存储器层次进行建模
- 默认情况下，算法分析一般是指对算法时间效率的分析。



2 计算的约定



□ 思考1:

- ✓ 一般，我们所说的算法/程序的执行时间是什么单位？
- ✓ 日常生活中，时间是以小时、分钟、秒、毫秒等时间单位为计量单位的。

□ 但算法/程序的执行时间是什么？

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

□ 算法的执行时间是算法中所有运算执行时间的总和可以表示为：

$$\text{算法的执行时间} = \sum f_i t_i$$

其中： f_i ：是运算*i*的执行次数，称为该运算的频率计数

t_i ：是运算*i*在实际的计算机上每执行一次所用的时间

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

如何确定算法使用了哪些运算，每种运算的 f_i 和 t_i 又是多少？



3 运算的分类

依照运算的**时间特性**，将运算分为**时间固界于常数**的运算 和**时间非固界于常数**的运算。

□ 时间固界于常数的运算：

- 基本算术运算，如整数、浮点数的加、减、乘、除
- 字符运算、赋值运算、过程调用等

□ 特点：执行时间是固定量，与操作数无关。

例：
 $1+1 = 2$ vs $10000+10000 = 20000$
 $100*100 = 10000$ vs $10000*10000 = 100000000$
CALL INSERTIONSORT



□ 更一般的情况

- ✓ 设有n种运算 C_1, C_2, \dots, C_n , 它们实际执行时间分别是 t_1, t_2, \dots, t_n 。

令 $t_0 = \max(t_1, t_2, \dots, t_n)$, 则每种运算执行一次的时间都可以用 t_0 进行抽象, 即可用 t_0 进行限界(上界)。

- ✓ 进一步, 视 t_0 为一个单位时间, 则这些运算“理论”上都可视为仅花一个固定的单位时间 t_0 即可完成的运算

——称具有这种性质的运算为时间固界于常数的运算。



时间非固界于常数的运算

□ 特点：运算的执行时间与操作数相关，每次执行的时间是一个不定的量。

✓ 如：

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

□ 怎么分析时间非固界于常数的运算？

- ✓ 这类运算通常是由更基本的运算组成的。而这些基本运算是时间固界于常数的运算。所以分析时间非固界于常数的运算时，将其分解成若干时间固界于常数的运算即可。
- ✓ 如：字符串比较时间 t_{string}

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



思考：为什么要引入“时间圈界于常数的运算”这一概念？

□ 引入“单位时间”的概念，将“单位时间”视为“1”，从而将算法的理论执行时间转换成“**单位时间**”意义下的**执行次数**，从而实现了“执行时间”与时间复杂度概念的统一。

$$\sum f_i t_i$$



4 工作数据集的选择

- 算法的执行情况与输入的数据有什么样的关系呢?
 - 算法的执行时间与输入数据的规模相关，一般规模越大，执行时间越长
 - 如：插入算法，10个数的排序时间显然小于1000个数的排序时间
 - 在不同的数据配置上，同一算法有不同的执行情况，可分为最好、最坏和平均等情况讨论。
 - 思考：插入排序的最好、最坏和平均时间复杂度是多少？
- 编制不同的数据配置，分析算法的最好、最坏、平均工作情况是算法分析的一项重要工作



5 算法分析

- 算法分析的目的是求取算法时间/空间复杂度的**限界函数**。
- 限界函数通常是关于问题规模n的**特征函数**，被表示成O、Ω或Θ的形式。
如：归并排序的时间复杂度是 $\Theta(n \log n)$
- **怎么获取算法复杂度的特征函数？**
 - ✓ 注：首先明确，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。
 - ✓ 以下以时间分析为例进行说明。



如何进行时间分析?



□ 统计算法中各类运算的执行次数

✓ 统计对象：运算

- 基本运算：时间囿于常数的运算
- 复合运算：具有固定执行时间的程序块，如一条语句、复合语句，甚至一个函数，由基本运算组成。

✓ 统计内容：频率计数，即算法中语句或运算的执行次数。

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

$x \leftarrow x + y$

(a)

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

repeat

(b)

分析

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

(c)

(a)： $x \leftarrow x + y$ 执行了1次

(b)： $x \leftarrow x + y$ 执行了 n 次

(c)： $x \leftarrow x + y$ 执行了 n^2 次



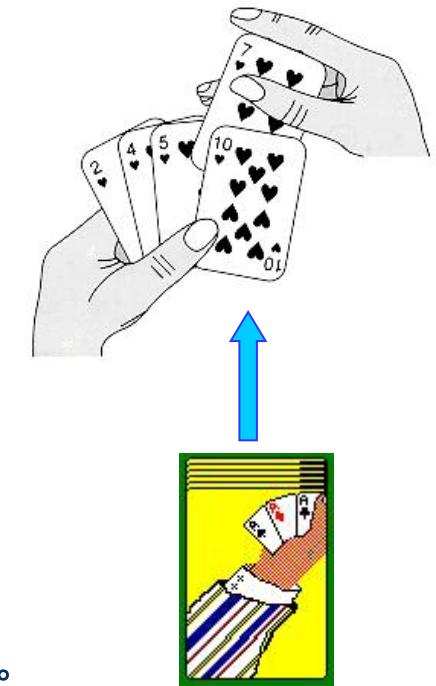
插入排序

口算法时间效率分析

INSERTION-SORT(A)

	cost	times
1 $\text{for}(j = 2; j <= \text{length}[A]; j++)$	c_1	n
2 { $\text{key} = A[j]$	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 $\text{while}(i > 0 \&\& A[i] > \text{key})$	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 }		
9 $A[i+1] = \text{key}$	c_8	$n-1$
10 }		

- 整个算法的执行时间是执行所有语句的时间之和。



总运行时间：

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



□ 即使规模相同，一个算法的执行时间也可能依赖于给定的输入。

✓ 如：**INSERTION-SORT**

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

INSERTION-SORT(A)	<i>cost</i>	<i>times</i>
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 $A[1 \dots j - 1]$.	0	$n - 1$
4 $i = j - 1$	c_4	$n - 1$
5 while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^n t_j$
6 $A[i + 1] = A[i]$	c_6	$\sum_{j=2}^n (t_j - 1)$
7 $i = i - 1$	c_7	$\sum_{j=2}^n (t_j - 1)$
8 $A[i + 1] = key$	c_8	$n - 1$



□ **最坏情况**: 初始数组是反向排序的。

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

□ **INSERTION-SORT(A)**

		<i>cost</i>	<i>times</i>
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	$n - 1$

元素进行比较，

形式。



我们更关心算法的最坏情况执行时间。



□ 对于规模为 n 的任何输入，一般考察算法的最坏运行时间：

- ✓ 最坏情况运行时间是在任何输入情况下的一上界；
- ✓ 对于某些算法来说，最坏情况出现还是比较频繁的，如信息检索（信息经常不存在）；
- ✓ 大致上看，“平均情况”通常和最坏情况一样差。

□ 为什么？

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



时间复杂度限界函数的获取



□ 函数表达式中的最高次项的次数称为函数表达式的**数量级**（或阶）

- ✓ 限界函数：取自频率计数函数表达式中的**最高次项**，并忽略常系数，记为： $g(n)$ 。
- ✓ 增长量级(order of growth):一个算法的时间是由其增长速率所决定的，所以对于算法时间复杂度的研究主要侧重的是算法的渐进性能，即当问题的输入规模n很大时算法的运行效率。

例如：对于一个多项式时间的算法，如： $T(n) = An^2+Bn+C$, 当n很大时，
 $T(n)$ 的增长速率是由最高项 n^2 所决定。

□ 对于足够大的n，低阶项和常系数都不如最高次项中的因子重要，因此 $g(n)$ 忽略了函数表达式中次数较低的“次要”项，体现了算法复杂性的最本质特征，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： $T(n) = n^4+10000$



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 *Arithmetoricorum 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 （或 k 以内）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 $i = 1, 2, \dots, k$ 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

$$\begin{aligned} F_{k+1} &< (5/3)^k + (5/3)^{k-1} \\ &< (3/5)(5/3)^{k+1} + (3/5)^2(5/3)^{k+1} \\ &< (3/5 + 9/25)(5/3)^{k+1} \\ &< (24/25)(5/3)^{k+1} \\ &< (5/3)^{k+1} \end{aligned}$$

$\therefore n=k+1$ 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, \dots, P_k 是依次排列的所有素数



令： $N = P_1 P_2 \dots P_k + 1$

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, \dots, P_k 整除，因为用 P_1, P_2, \dots, 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 $F_k \leq k^2$ 是否成立? 举反例: 令 $k=11$
则: $F_{11}=144 > 11^2$, 所以原命题不成立。



第二章 算法基础

内容提要：

□ 本章将介绍一个贯穿整个课程、进行算法设计与分析的框架。

□ 算法分析基础

- ✓ 问题求解基础
- ✓ 算法描述方法
- ✓ 算法正确性证明
- ✓ 算法分析基本框架
- ✓ 举例：插入排序

□ 算法设计策略之——分治法



算法设计



□ 对于一个问题，可以有很多中解决方法。因此，算法设计策略也有很多。

□ 排序问题：

- ◆ Bubble Sort: Bubbling
- ◆ Insertion Sort: Incremental Approach (增量靠近)
- ◆ Merge Sort: Divide and Conquer (分而治之)
- ◆ Quick Sort: Location (元素定位)
-

□ 分治法最差的时间比插入排序法少得多



分治法



- **核心思想:** 分而治之，各个击破；
- **递归结构:** 为了解决一个给定的问题，算法要一次或多次地递归调用其自身来解决相关的子问题。
- **分治策略:** 将原问题划分为 n 个规模较小而结构与原问题相似的子问题；递归地解决这些子问题，然后再合并其结果，就得到原问题的解。
- **三个步骤:**
 - (1) **分解 (Divide)** : 将原问题分成一系列子问题；
 - (2) **解决 (Conquer)** : 递归地解各个子问题。若子问题足够小，则直接求解；
 - (3) **合并 (Combine)** : 将子问题的结果合并成原问题的解

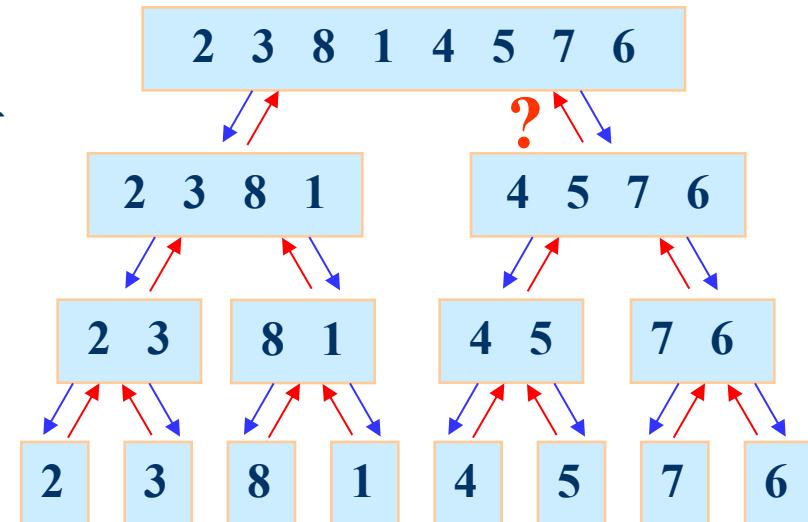


归并排序

□ 归并排序算法 (Merge Sort Algorithm)

- ① 分解：把 n 个元素分成各含 $n/2$ 个元素的子序列；
- ② 解决：用归并排序算法对两个子序列递归地排序；
- ③ 合并：合并两个已排序的子序列以得到排序结果。

在对子序列排序时，其长度为1时
递归结束。
单个元素被视为是已排好序的。



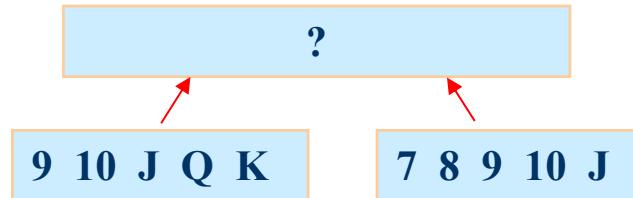


归并排序

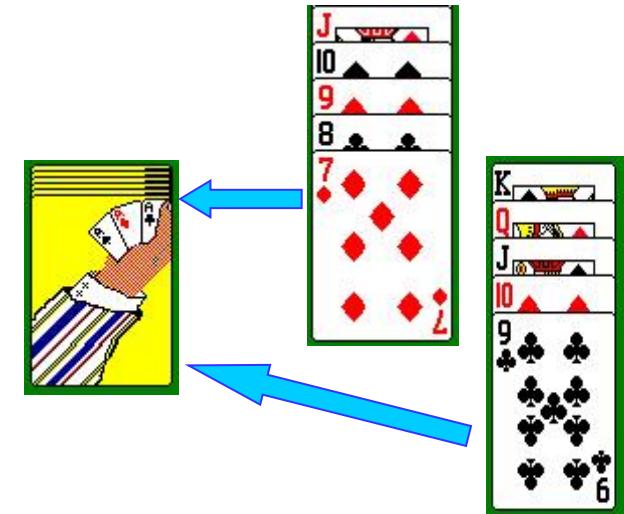


□ MERGE(A, p, q, r): 归并排序算法的关键步骤，合并步骤中合并两个已经排序好的子序列。

- ◆ A 是个数组， p, q, r 数组中元素的下标，且 $p \leq q < r$.
- ◆ 该过程假设子数组 $A[p .. q]$ 和 $A[q+1 .. r]$ 是有序的，并将它们合并成一个已排好序的子数组代替当前子数组 $A[p .. r]$ 。
- ◆ 合并过程:



The procedure takes time $\Theta(n)$, $n=r-p+1$,
其中 n 是待合并元素的总数





归并排序

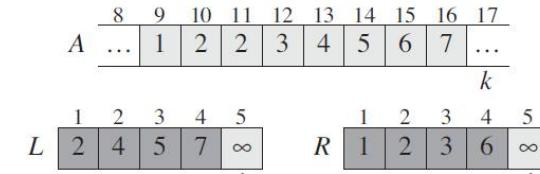
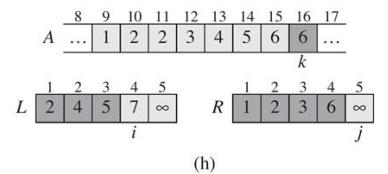
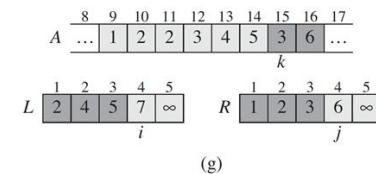
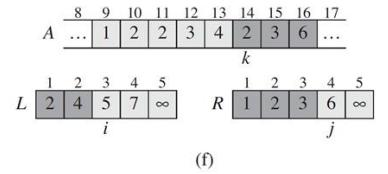
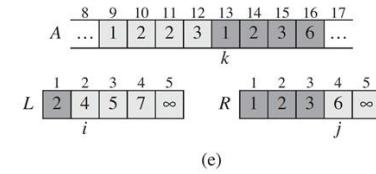
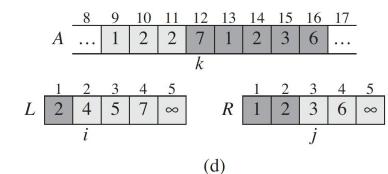
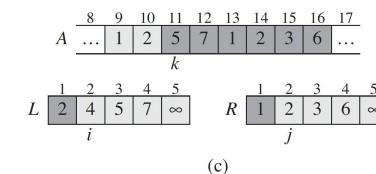
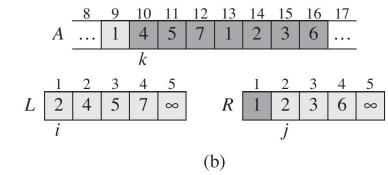
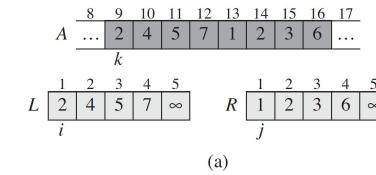
MERGE(A, p, q, r)

```

1    $n_1 \leftarrow q-p+1$ 
2    $n_2 \leftarrow r-q$ 
3   create arrays  $L[1 .. n_1+1]$  and  $R[1 .. n_2+1]$ 

4   for  $i \leftarrow 1$  to  $n_1$ 
5       do  $L[i] \leftarrow A[p+i-1]$ 
6   for  $j \leftarrow 1$  to  $n_2$ 
7       do  $R[j] \leftarrow A[q+j]$ 
8    $L[n_1+1] \leftarrow \infty$ 
9    $R[n_2+1] \leftarrow \infty$ 
10   $i \leftarrow 1$ 
11   $j \leftarrow 1$ 
12  for  $k \leftarrow p$  to  $r$ 
13      do if  $L[i] \leq R[j]$ 
14          then  $A[k] \leftarrow L[i]$ 
15           $i \leftarrow i+1$ 
16      else  $A[k] \leftarrow R[j]$ 
17           $j \leftarrow j+1$ 

```





□ 循环不变式为：子数组A[p..k-1]始终为排好序的状态,包含L[1..n₁+1]和R[1..n₂+1]中的k-p个最小元素，进而，L[i]和R[j]是各自所在数组中未被复制回数组A的最小元素。

① 初始状态(Initialization):

∴ 循环开始前k=p,所以子数组L[p..k-1]为空。

∴ L[i]和R[j]是各自所在数组中未被复制回数组A的最小元素。

② 保持状态(Maintance):

∴ 假设L[i] ≤ R[j]分析...

反之L[i] ≥ R[j],同样也保持循环不变式

③ 终止状态(Termination):退出循环时,

∴ 终止时k=r+1.子数组A[p..k-1]就是A[p..r]且按照从小到大的顺序包含L[1..n₁+1]和R[1..n₂+1]中的k-p=r-p+1个最小元素。L和R除了两个最大的元素外，其它所有元素都已经被复制回数组A

∴ 循环不变式为真
2022-2-21



归并排序

MERGE(A, p, q, r)	cost	times
1 $n_1 \leftarrow q-p+1$	c	1
2 $n_2 \leftarrow r-q$	c	1
3 create arrays $L[1 .. n_1+1]$ and $R[1 .. n_2+1]$	c	1
4 for $i \leftarrow 1$ to n_1	c	n_1+1
5 do $L[i] \leftarrow A[p+i-1]$	c	n_1
6 for $j \leftarrow 1$ to n_2	c	n_2+1
7 do $R[j] \leftarrow A[q+j]$	c	n_2
8 $L[n_1+1] \leftarrow \infty$	c	1
9 $R[n_2+1] \leftarrow \infty$	c	1
10 $i \leftarrow 1$	c	1
11 $j \leftarrow 1$	c	1
12 for $k \leftarrow p$ to r	c	$r-p+2$
13 do if $L[i] \leq R[j]$	c	$r-p+1$
14 then $A[k] \leftarrow L[i]$	c	x
15 $i \leftarrow i+1$	c	x
16 else $A[k] \leftarrow R[j]$	c	$r-p+1-x$
17 $j \leftarrow j+1$	c	$r-p+1-x$

$$\begin{aligned} & r-p+1 \\ & = n_1 + n_2 = n \end{aligned}$$

$$1 < x < n_1$$

$$\Theta(n_1+n_2)=\Theta(n)$$



归并排序



- 将MERGE过程作为合并排序中的一个子过程来使用。下面过程 MERGE-SORT(A, p, r) 对子数组 $A[p..r]$ 进行排序. 如果 $p \geq r$, 则该子数组中至多只有一个元素, 当然就是已排序的。否则, 分解步骤就计算出一个下标 q , 将 $A[p..r]$ 分成 $A[p..q]$ 和 $A[q+1..r]$, 前者包含 $\lceil n/2 \rceil$ 个元素, 后者包含 $\lfloor n/2 \rfloor$ 个元素。

MERGE-SORT(A, p, r)

- 1 if $p < r$
- 2 Then $q \leftarrow \lfloor p+r/2 \rfloor$
- 3 MERGE-SORT(A, p, q)
- 4 MERGE-SORT($A, q+1, r$)
- 5 MERGE(A, p, q, r)

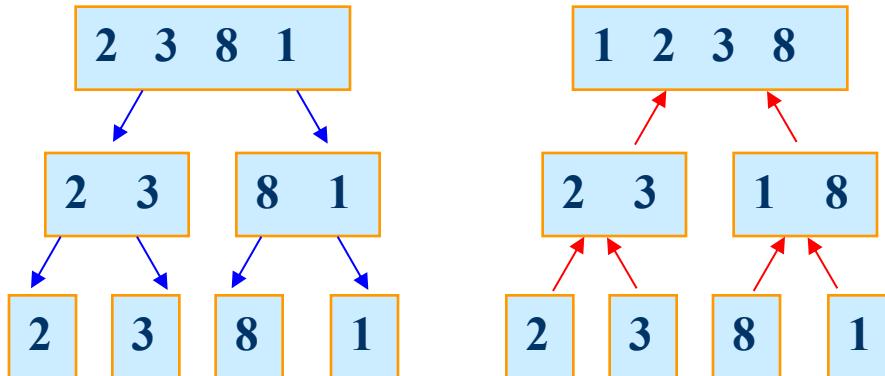


归并排序

MERGE-SORT(A, p, r)

1 if $p < r$
2 Then $q \leftarrow \lfloor (p+r)/2 \rfloor$

3 MERGE-SORT(A, p, q)
4 MERGE-SORT($A, q+1, r$)
5 MERGE(A, p, q, r)



MERGE-SORT($A, 1, 4$)

1 if $1 < 4$
2 Then $q \leftarrow \lfloor (1+4)/2 \rfloor = 2$
3 MERGE-SORT($A, 1, 2$)
4 if $1 < 2$
5 Then $q \leftarrow \lfloor (1+2)/2 \rfloor = 1$
6 MERGE-SORT($A, 1, 1$)
7 if $1 < 1$
8 MERGE-SORT($A, 2, 2$)
9 if $2 < 2$
10 MERGE($A, 1, 1, 2$)
11 MERGE-SORT($A, 3, 4$)
12 if $3 < 4$
13 Then $q \leftarrow \lfloor (3+4)/2 \rfloor = 3$
14 MERGE-SORT($A, 3, 3$)
15 if $3 < 3$
16 MERGE-SORT($A, 4, 4$)
17 if $4 < 4$
18 MERGE($A, 3, 3, 4$)
19 MERGE($A, 1, 2, 4$)
20 Soochow University



分治法分析



- 当一个算法含有对其自身的递归调用时，其运行时间总可以用一个**递归方程（或递归式）**来表示。该方程通过描述子问题与原问题的关系，来给出总的运行时间。我们可以利用数学工具来解递归式，并给出算法性能的界。
- 分治法给出的时间：

$$T(n) = \begin{cases} (1) & \text{if } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

- $D(n)$ 是把原问题分解为子问题所花的时间；
- $C(n)$ 是把子问题的解合并为原问题的解所花的时间；
- $T(n)$ 是一个规模为 n 的问题的运行时间。



分治法分析



- 为简化算法分析，通常假设 n 为2的幂次，使得每次分解产生的子序列长度恰为 $n/2$ 。这一假设并不影响递归式解的增长量级。
- 合并排序 n 个数最坏运行时间：
 - ① 当 $n=1$ 时，合并排序一个元素的时间是个常量；
 - ② 当 $n>1$ 时，运行时间分解如下：
 - 分解：仅仅是计算出子数组的中间位置，需要常量时间， $D(n)=\Theta(1)$ ；
 - 解决：递归地求解两个规模为 $n/2$ 的子问题，时间为 $2T(n/2)$ ；
 - 合并：MERGE过程的运行时间为 $C(n)=\Theta(n)$ 。

$$T(n) = \begin{cases} (1) & \text{if } n = 1 \\ 2T(n/2) + (n) & \text{if } n > 1 \end{cases}$$



分治法分析

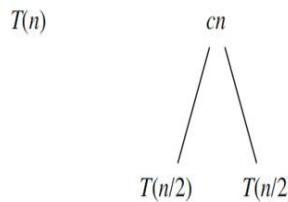
➤ 递归式重写为：

$$T(n) = \begin{cases} (1) & \text{if } n = 1 \\ 2T(n/2) + cn & \text{if } n > 1 \end{cases}$$

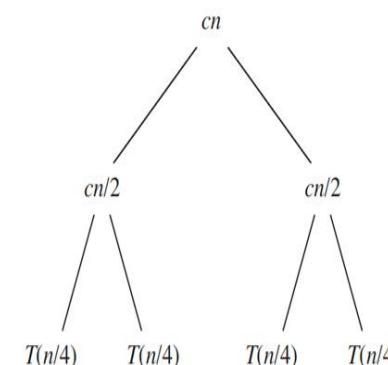


$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n/2) + cn & \text{if } n > 1 \end{cases}$$

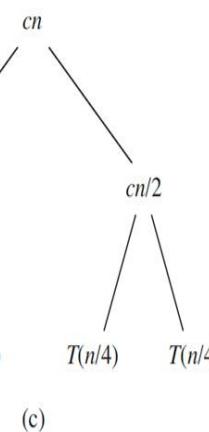
➤ 递归式求解如下：



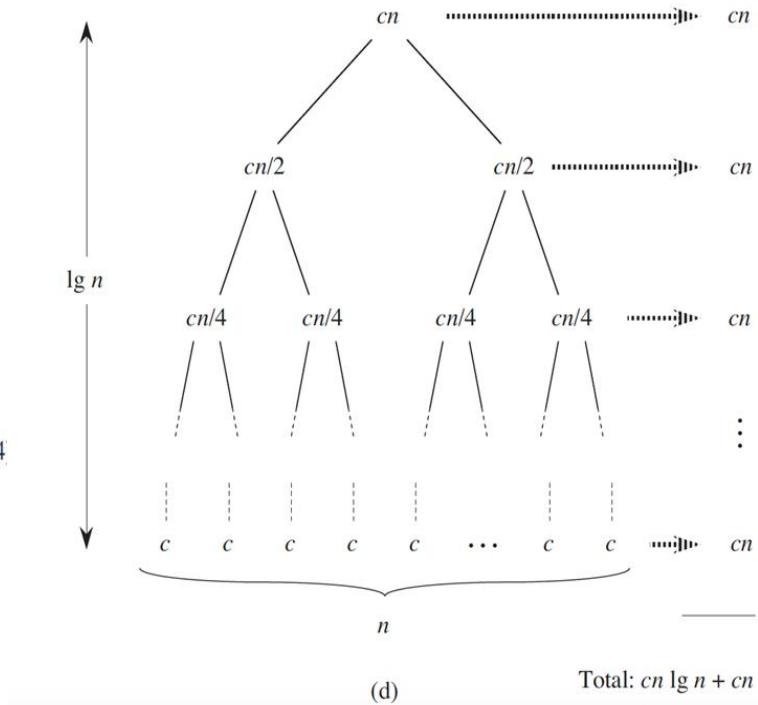
(a)



(b)



(c)

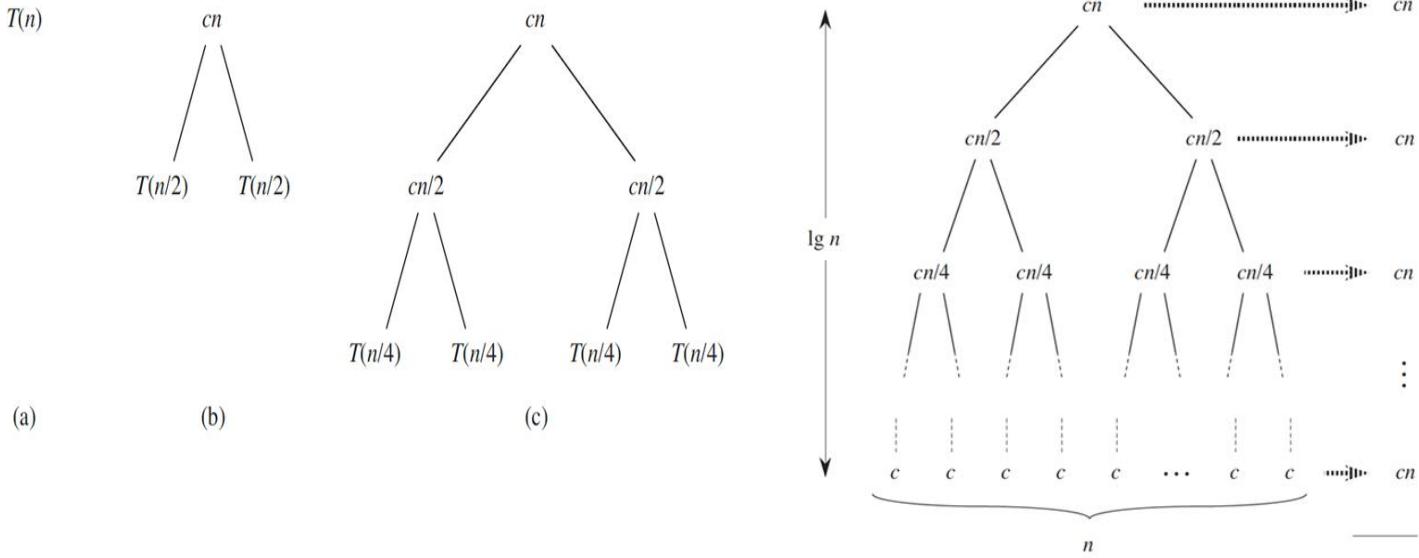


(d)

Total: $cn \lg n + cn$



分治法分析



➤ 给这棵树的每一层加上代价：

最底层之下的第 i 层有 2^i 个节点，每一个代价是 $c(n/2^i)$ 第 i 层总代价是 $2^i c(n/2^i) = cn$ 。

最底层有 n 个节点，每一个节点的代价是 c ，该层总代价为 cn ；

➤ 树总共层数是 $\lg n + 1$ 层，每一层代价都是 cn ，所以总代价为：

$$cn(\lg n + 1) = cn \lg n + cn = (n \lg n)$$

➤ 最坏情况明显比插入排序 $\Theta(n^2)$ 要好



谢谢 !

作业： 2.1-2 2.1-4

2.2-2: 写出选择排序的算法

2.3-6