



湖南大学
HUNAN UNIVERSITY

第一章 算法分析与设计基础

—— 湖南大学信息科学与工程学院 ——

联系方式



任课教师：彭 鹏

办公室：软件大楼536

邮箱：hnu16pp@hnu.edu.cn

教材以及课程资源

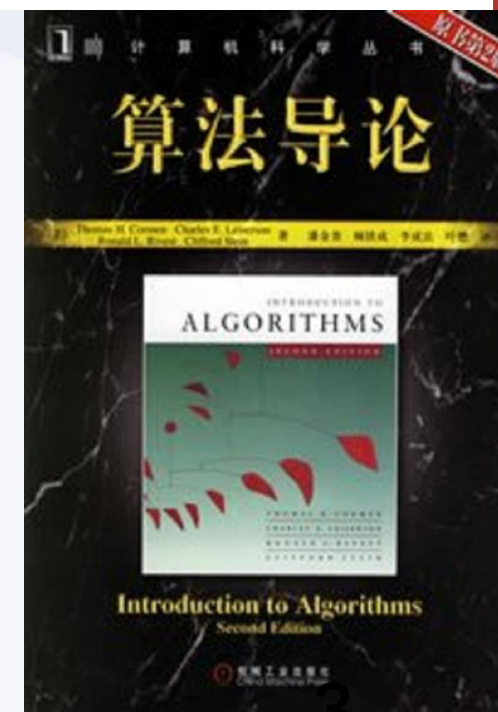


教材:

算法导论, [美] Thomas H.Cormen, [美] Charles E.Leiserson, [美] Ronald L.Rivest,
[美] Clifford Stein 著, 殷建平, 徐云, 王刚 等 译

课程网站:

<https://bnu05pp.github.io/AlgorithmCourse/index.html>



考核方式



考核形式:

- (1) 课堂点名 (占5%)
- (2) Leetcode (占45%)
- (3) 期末考试 (占50%)

Leetcode考核说明



- 每周周二8:00到24:00之间，将自己Leetcode截屏发给我学生邮箱 18711230730@163.com，邮件标题格式为"Leetcode截屏-[姓名]-[学号]-[日期]"
- 每周根据题目计分，题目范围参照课程进度。一个难度简单的题，计2分；一个难度中等的题，计3分；一个难度困难的题，计4分。单次分数上限10分。



目录

第一节

背景介绍

第二节

算法基本概念

第三节

算法与数据结
构、程序的关系

第四节

重要问题类型

第五节

算法时间复杂
度分析

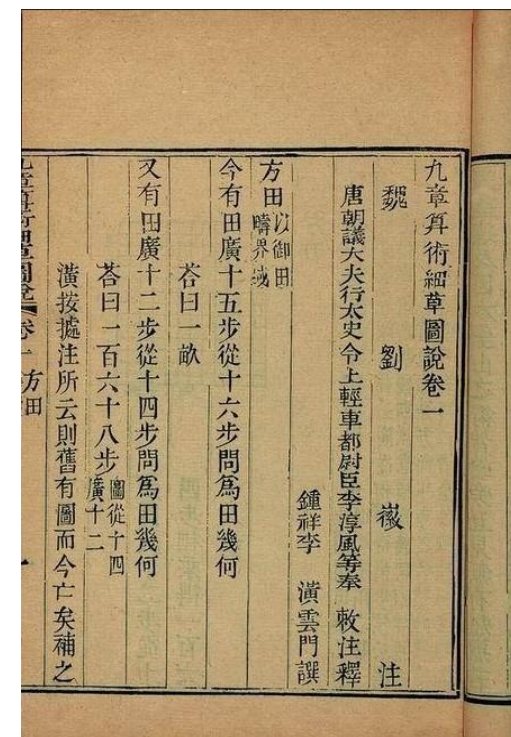
第六节

一个算法例子：
文本距离

算法历史



- “算法”一词的来历
 - 中文的“算法”：《九章算术》
 - 英文的“算法”（algorithm）：花拉子米（al-Khwārizmī）
 - 图灵机

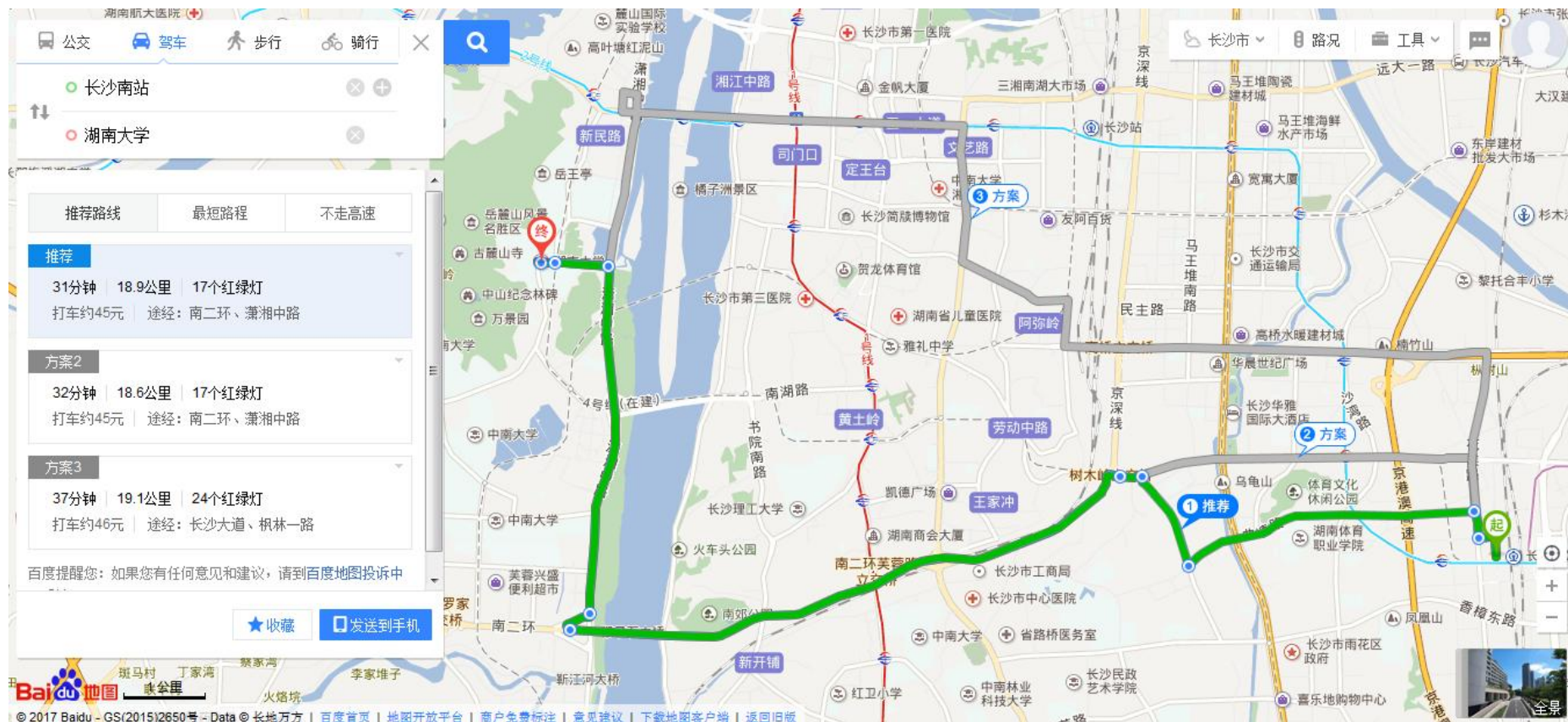




□ 二十世纪十大著名算法

- 蒙特卡洛方法：1946年
- 单纯形法：1947 年
- Krylov子空间迭代法：1950年
- 矩阵计算的分解方法：1951 年
- 优化的Fortran编译器：1957年
- 计算矩阵特征值的QR算法：1959-61 年
- 快速排序算法：1962年
- 快速傅立叶变换：1965年
- 整数关系探测算法：1977年
- 快速多极算法：1987年

算法与生活-导航



算法与生活-导航



算法与生活-导航



目录

第一节

背景介绍

第二节

算法基本概念

第三节

算法与数据结
构、程序的关系

第四节

重要问题类型

第五节

算法时间复杂
度分析

第六节

一个算法例子：
文本距离



□ 广义算法

➤ 算法是解决问题的方法，如一道菜谱、一个安装电脑的操作指南等。

□ 狭义算法：计算机算法

➤ 计算机算法是对特定问题求解步骤的一种描述，是指令的有限序列。



□ 算法的五个重要特性

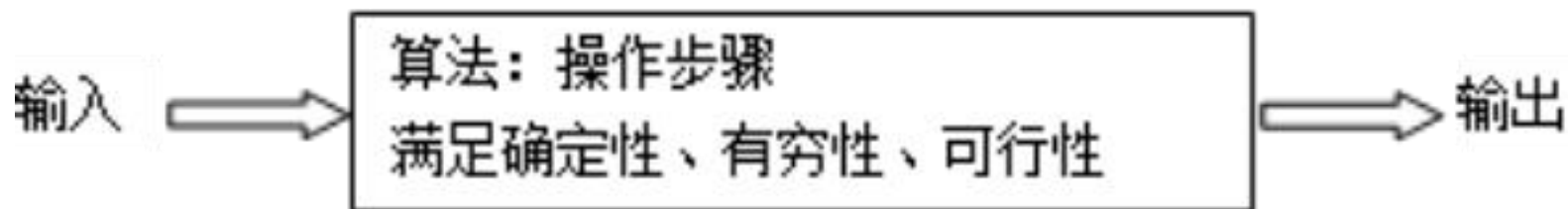


图 1.1 算法的概念



□ “好” 算法的五个其他特性

- 正确性
- 健壮性
- 可理解性
- 抽象分级
- 高效性



□ “好” 算法的五个其他特性

- 正确性
- 健壮性
- 可理解性
- 抽象分级
- 高效性



□ 算法的描述方法

➤ 自然语言

➤ 流程图

➤ 程序设计语言

➤ 伪代码

武松打死老虎

武松/打死老虎

武松/打/死老虎

sum=1+2+3+4+5……+ (n-1) +n为例

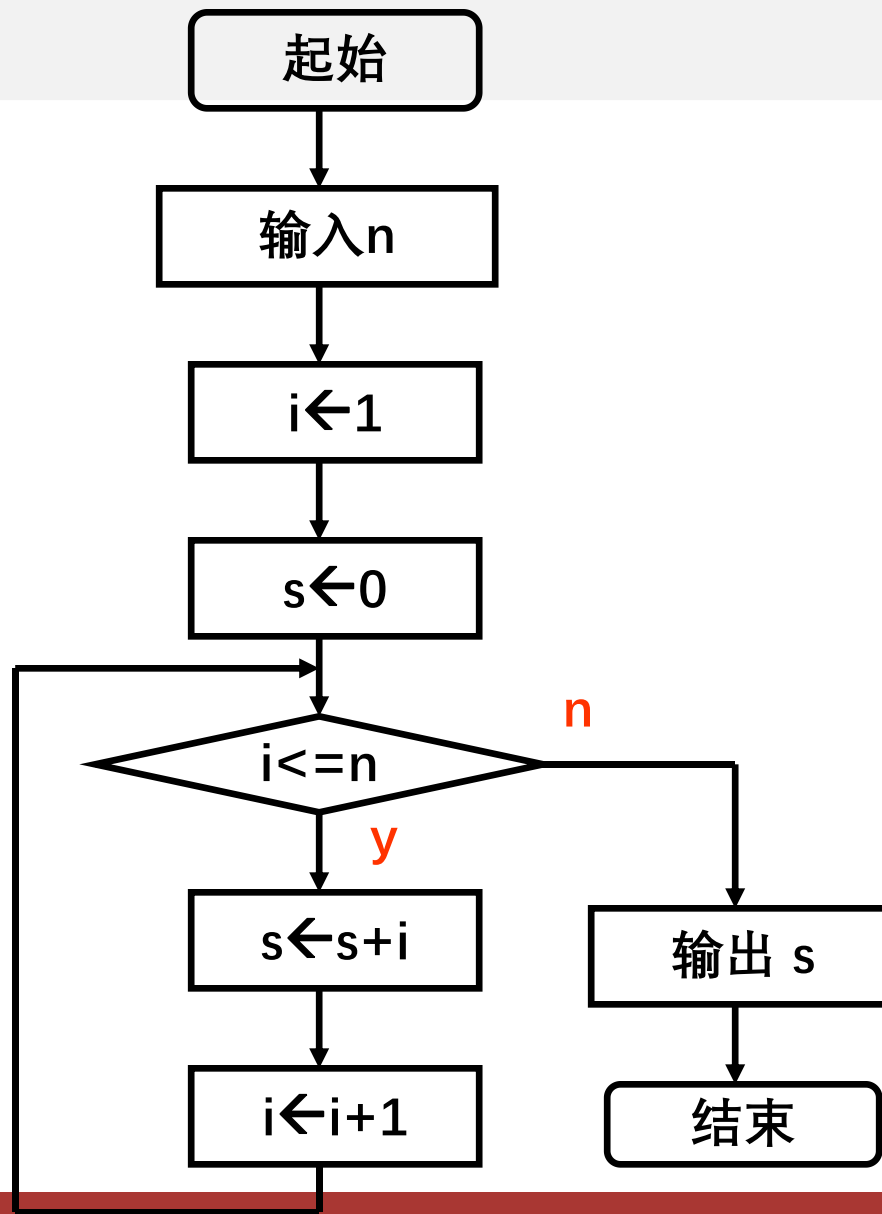
- ① 确定一个n的值;
- ② 假设等号右边的算式项中的初始值i为1;
- ③ 假设sum的初始值为0;
- ④ 如果 $i \leq n$ 时, 执行⑤, 否则转出执行⑧;
- ⑤ 计算sum加上i的值后, 重新赋值给sum;
- ⑥ 计算i加1, 然后将值重新赋值给i;
- ⑦ 转去执行④;
- ⑧ 输出sum 的值, 算法结束。

算法基本概念



□ 算法的描述方法






- 自然语言
- 流程图
- 程序设计语言
- 伪代码





□ 算法的描述方法

- 自然语言
- 流程图
- 程序设计语言
- 伪代码

| 符 号 | 名 称 | 作 用 |
|---|--------|--|
|  | 开始、结束符 | 表示算法的开始和结束符号。 |
|  | 输入、输出框 | 表示算法过程中，从外部获取的信息（输入），然后将处理过的信息输出。 |
|  | 处理框 | 表示算法过程中，需要处理的内容，只有一个入口和一个出口。 |
|  | 判断框 | 表示算法过程中的分支结构，菱形框的4个顶点中，通常用上面的顶点表示入口，根据需要其余的顶点表示出口。 |
|  | 流程线 | 算法过程中的指向流程的方向。 |



□ 算法的描述方法

- 自然语言
- 流程图
- 程序设计语言（代码）
- 伪代码

算法基本概念



□ 算法的描述方法

- 自然语言
- 流程图
- 程序设计语言
- 伪代码

算法开始

输入n的值;

可以加注解

$i \leftarrow 1;$

/*为 i 赋初值*/

$s \leftarrow 0;$

/*为s 赋初值*/

While($i \leq n$)

/*循环语句*/

{

/*循环开始*/

$s \leftarrow s + i;$

/*把 i 累加到 s*/

$i \leftarrow i + 1;$

/*记数*/

}

/*循环结束*/

输出 s 的值;

算法结束

算法基本概念



□ 算法设计的一般过程

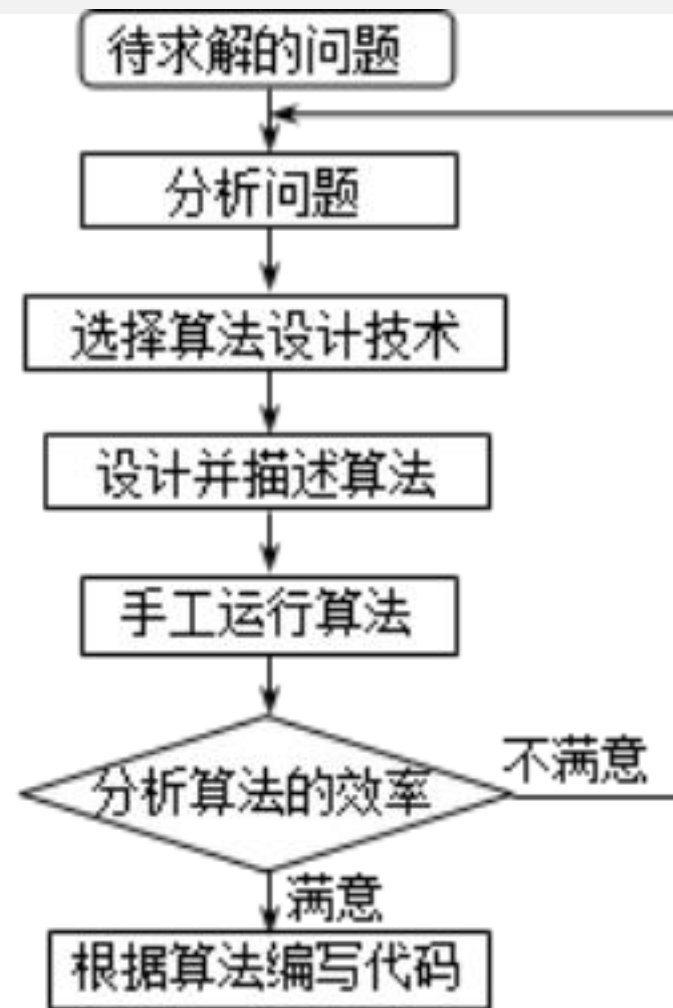


图 1.2 算法设计的一般过程

目录

第一节

背景介绍

第二节

算法基本概念

第三节

算法与数据结
构、程序的关系

第四节

重要问题类型

第五节

算法时间复杂
度分析

第六节

一个算法例子：
文本距离

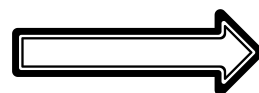


大数据时代

简洁表示，时间复杂度低



感兴趣信息
关键字



数据结构

内存



数据文件
数据多维属性



数据

查询结果





□ 算法与数据结构

- 数据结构是底层，算法高层。数据结构为算法提供服务。算法围绕数据结构操作。

□ 算法与程序

- 程序是某个算法或过程的在计算机上的一个具体的实现。
- 程序是依赖于程序设计语言的，甚至依赖于计算机结构的。
- 算法是脱离具体的计算机结构和程序设计语言的。

目录

第一节

背景介绍

第二节

算法基本概念

第三节

算法与数据结
构、程序的关系

第四节

重要问题类型

第五节

算法时间复杂
度分析

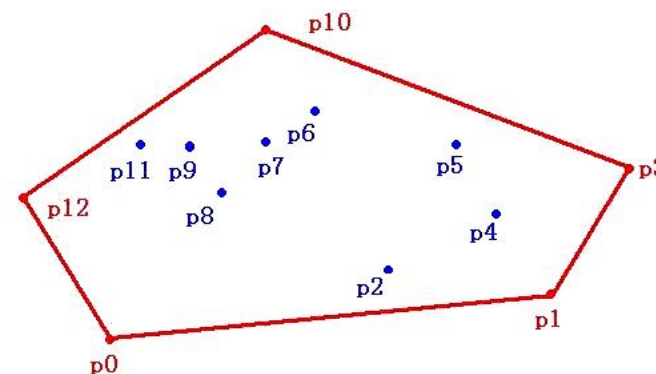
第六节

一个算法例子：
文本距离

重要问题类型



- 查找问题
- 排序问题
- 图问题
- 组合问题
- 几何问题



目录

第一节

背景介绍

第二节

算法基本概念

第三节

算法与数据结
构、程序的关系

第四节

重要问题类型

第五节

算法时间复杂
度分析

第六节

一个算法例子：
文本距离

算法复杂性分析



□ 影响软件（算法）性能的因素？

算法复杂性分析



10种排序算法的时间效率比较

| 算法\输入数据 | N=50 K=50 | N=200 K=100 | N=500 K=500 | N=2000 K=2000 | N=5000 K=8000 | N=10000 K=20000 | N=20000 K=20000 | N=20000 K=200000 |
|---------|--------------|----------------|----------------|------------------|------------------|--------------------|--------------------|---------------------|
| 冒泡排序 | 0ms | 15ms | 89ms | 1493ms | 9363ms | 36951ms | 147817ms | 143457ms |
| 插入排序 | 1ms | 13ms | 82ms | 1402ms | 8698ms | 34731ms | 134817ms | 134836ms |
| 希尔排序 | 0ms | 1ms | 6ms | 30ms | 110ms | 257ms | 599ms | 606ms |
| 选择排序 | 0ms | 5ms | 31ms | 461ms | 2888ms | 11736ms | 45308ms | 44838ms |
| 堆排序 | 0ms | 3ms | 9ms | 40ms | 124ms | 247ms | 525ms | 527ms |
| 归并排序 | 2ms | 6ms | 18ms | 75ms | 199ms | 392ms | 778ms | 793ms |
| 快速排序 | 0ms | 1ms | 2ms | 14ms | 36ms | 84ms | 196ms | 163ms |
| 计数排序 | 0ms | 1ms | 1ms | 5ms | 15ms | 32ms | 51ms | 62ms |
| 基数排序 | 0ms | 1ms | 4ms | 19ms | 47ms | 114ms | 237ms | 226ms |
| 桶排序 | 0ms | 2ms | 6ms | 25ms | 68ms | 126ms | 254ms | 251ms |

算法复杂性分析



□ 算法的好与坏

➤ 高斯的故事

$$1+2+3+4+5+6+\cdots+100 = ?$$

$$(1+100) + (2+99) + \cdots + (50+51) = 50 \times 101 \quad (\text{高斯算法})$$



高斯（数学王子）

高斯：“给我最大快乐的，不是已懂得知识，而是不断的学习；不是已有的东西，而是不断的获取；不是已达到的高度，而是继续不断的攀登。”



□ 百钱百鸡问题

中国古代数学家张丘建在他的《算经》中提出了他的著名的“百钱百鸡问题”：鸡翁一，值钱五；鸡母一，值钱三；鸡雏三，值钱一；百钱买百鸡，翁、母、雏各几何？

意思是公鸡每只5元、母鸡每只3元、小鸡3只1元，用100元钱买100只鸡，求公鸡、母鸡、小鸡的只数。



□ 百钱百鸡问题

回想算法设计过程

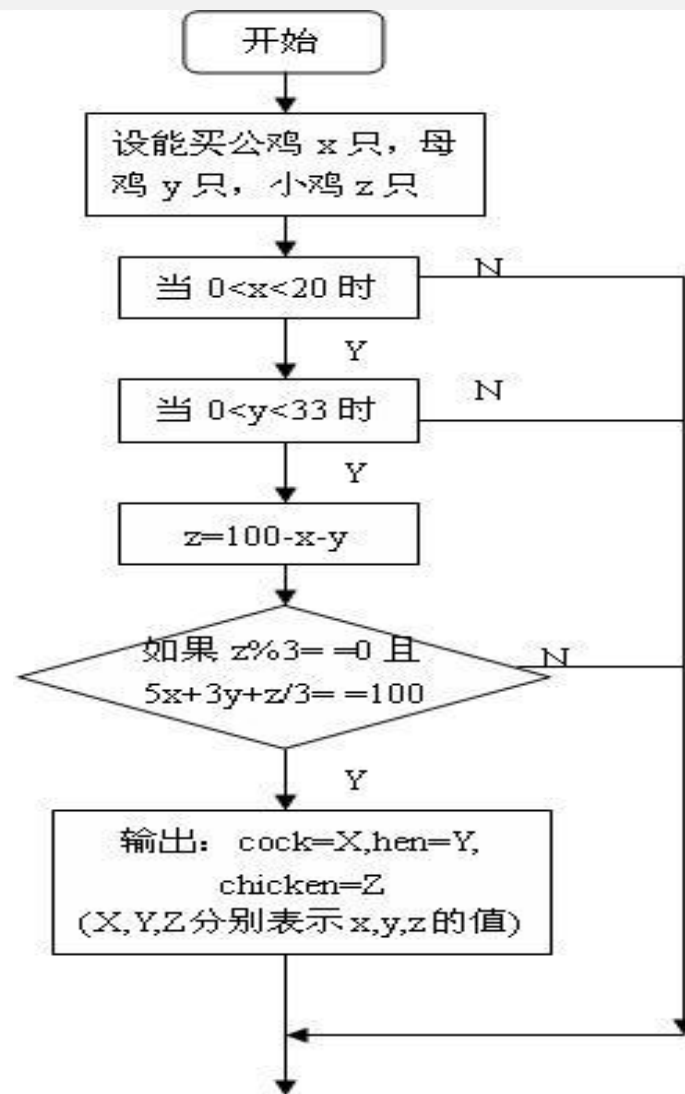
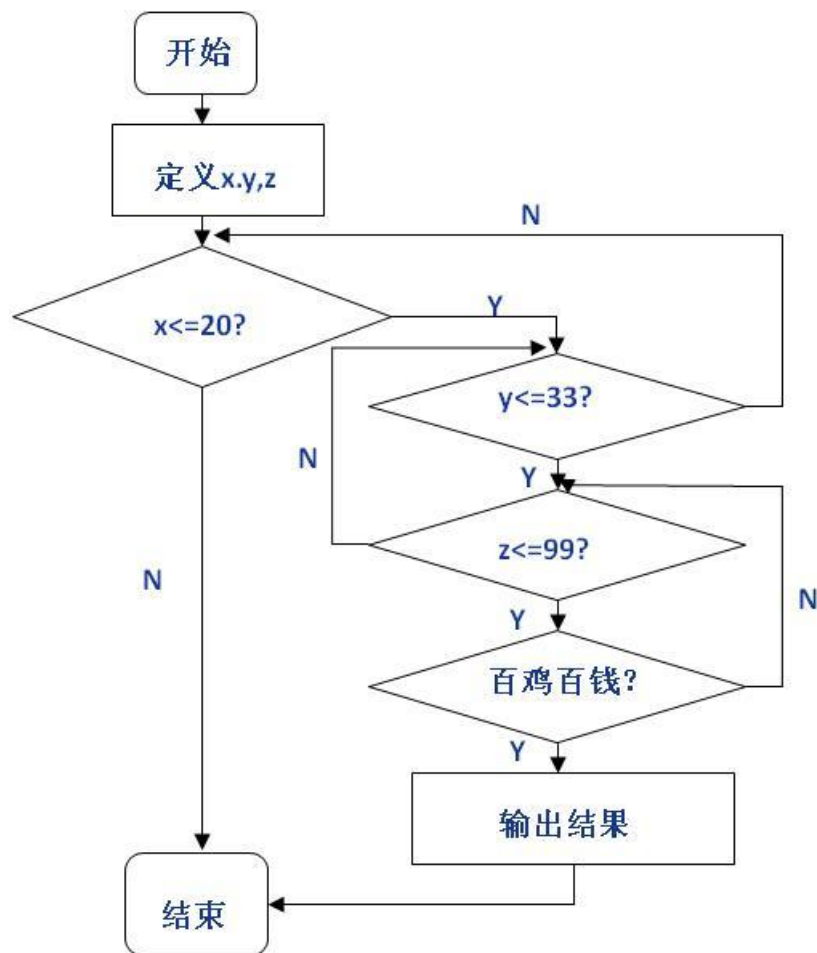
首先分析问题，并建立数学模型

如何建立数学模型？

算法复杂性分析



百钱百鸡问题



算法复杂性分析



□ 百钱百鸡问题

```
def chicken_question1():
    #  $x + y + z = 100$ 
    #  $5x + 3y + z/3 = 100$ 
    iCount = 0
    for x in range(1, 20):
        for y in range(1, 33):
            for z in range(1, 99):
                iCount = iCount + 1
                if z % 3 == 0 and 5 * x + 3 * y +
z / 3 == 100 and x + y + z == 100:
                    print('公鸡: ', x, '只', '母鸡: ', y, '只', '小鸡: ', z, '只')
    print('iCount:', iCount)
```

```
def chicken_question2():
    #  $x + y + z = 100$ 
    #  $5x + 3y + z/3 = 100$ 
    iCount = 0
    for x in range(1, 20):
        for y in range(1, 33):
            iCount = iCount + 1
            z = 100 - y - x
            if z % 3 == 0 and 5 * x + 3 * y
+ z / 3 == 100:
                print('公鸡: ', x, '只', '母鸡: ', y, '只', '小鸡: ', z, '只')
    print('iCount:', iCount)
```

算法复杂性分析



□ 百钱百鸡问题

算法1和算法2，哪个运行快？为什么？

算法复杂性分析



□ 百钱百鸡问题

算法1:

公鸡: 4 只 母鸡: 18 只 小鸡: 78 只

公鸡: 8 只 母鸡: 11 只 小鸡: 81 只

公鸡: 12 只 母鸡: 4 只 小鸡: 84 只

iCount: 59584

771 function calls in 0.010 seconds

算法2:

公鸡: 4 只 母鸡: 18 只 小鸡: 78 只

公鸡: 8 只 母鸡: 11 只 小鸡: 81 只

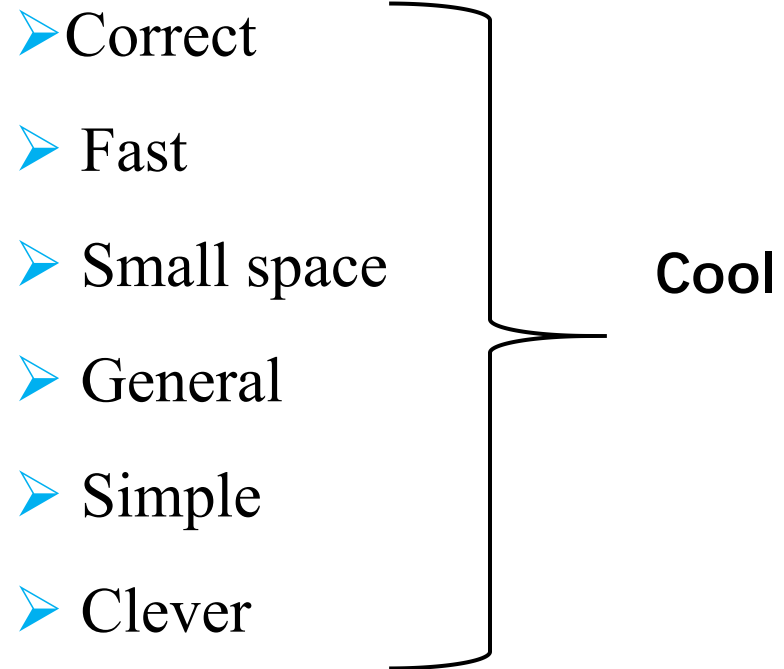
公鸡: 12 只 母鸡: 4 只 小鸡: 84 只

iCount: 608

771 function calls in 0.001 seconds



□ 什么样的算法是高效算法？



算法复杂性分析



□ 为什么需要高效的算法？

节省时间，存储需求，能源消耗，代价等

使用有限的资源解决大规模输入问题

（CPU，内存，硬盘灯）

优化旅行时间，调度冲突等



算法复杂性分析



- 是关于计算机程序性能和资源利用的研究
 - 时间复杂度分析
 - 空间复杂度分析



□ 算法运行时间的衡量方法1-事后统计

➤ 利用计算机内计时功能。

➤ 缺点：①必须先运行依据算法编制的程序；

②所得时间统计量依赖于硬件、软件等环境因素，掩盖算法本身的优劣；

③算法的测试数据设计困难。

算法复杂性分析--时间复杂度分析



□ 算法运行时间的衡量方法2-事前分析估计利用计算机内计时功能。

➤ 一个高级语言程序在计算机上运行所消耗的时间取决于：

① 依据的算法选用何种策略

② 问题的规模

③ 程序语言

④ 编译程序产生机器代码质量

⑤ 机器执行指令速度



□ 基本概念

- 问题规模：指输入量的多少。运行算法所需要的时间 T 是问题规模 n 的函数，记作 $T(n)$ 。
- 基本语句：执行次数与整个算法的执行次数成正比的语句。

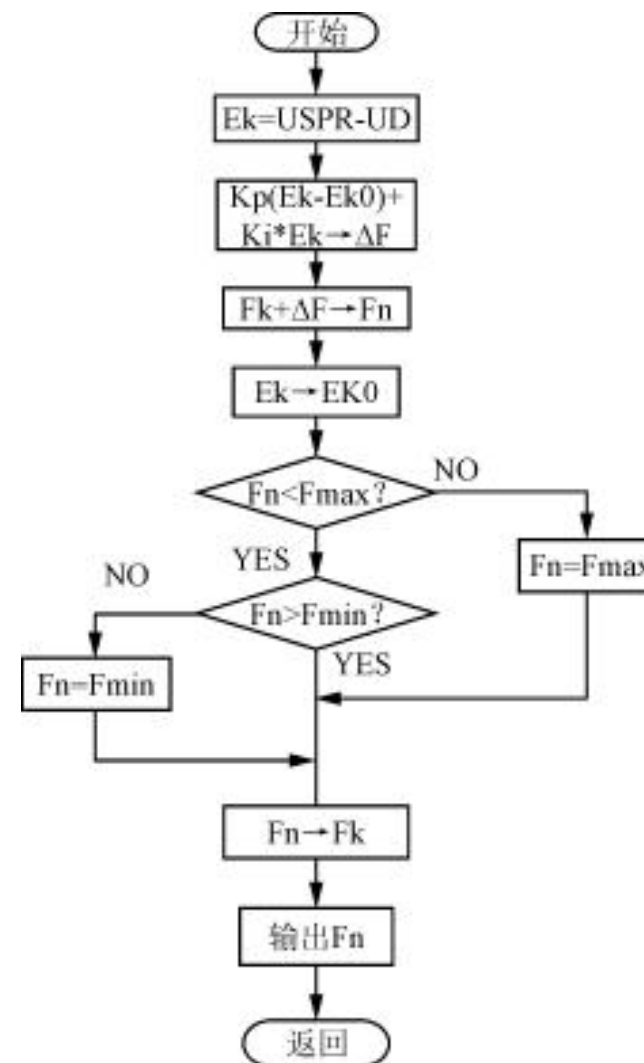


问题规模可控

估算算法运行时间的方法

迭代计数：计算循环的迭代次数

操作计数：找出关键操作，确定这些关键操作所需要的执行次数





渐进符号——运行时间的上界

定义1-1:

若存在两个正的常数 c 和 n_0 ,对于任意 $n \geq n_0$,都有 $T(n) \leq cf(n)$,则称 $T(n) = O(f(n))$ 。

大 O 符号描述增长率的上限,表示 $T(n)$ 的增长最多像 $f(n)$ 增长的那样快,换言之,当输入规模为 n 时,算法消耗时间的最大值,这个上限的阶越低,结果越有价值。

该算法的运行时间至多是 $O(f(n))$ 。

算法复杂性分析--时间复杂度分析



□ 渐进分析

表 2-8-1

| 次数 | 算法 A ($2n + 3$) | 算法 A' ($2n$) | 算法 B ($3n + 1$) | 算法 B' ($3n$) |
|-----------|-------------------|----------------|-------------------|----------------|
| $n = 1$ | 5 | 2 | 4 | 3 |
| $n = 2$ | 7 | 4 | 7 | 6 |
| $n = 3$ | 9 | 6 | 10 | 9 |
| $n = 10$ | 23 | 20 | 31 | 30 |
| $n = 100$ | 203 | 200 | 301 | 300 |

算法复杂性分析--时间复杂度分析



□ 渐进分析

表 2-8-2

| 次数 | 算法 C ($4n+8$) | 算法 C' (n) | 算法 D ($2n^2+1$) | 算法 D' (n^2) |
|------------|-----------------|---------------|-------------------|-----------------|
| $n = 1$ | 12 | 1 | 3 | 1 |
| $n = 2$ | 16 | 2 | 9 | 4 |
| $n = 3$ | 20 | 3 | 19 | 9 |
| $n = 10$ | 48 | 10 | 201 | 100 |
| $n = 100$ | 408 | 100 | 20 001 | 10 000 |
| $n = 1000$ | 4 008 | 1 000 | 2 000 001 | 1 000 000 |



渐进符号——运行时间的上界

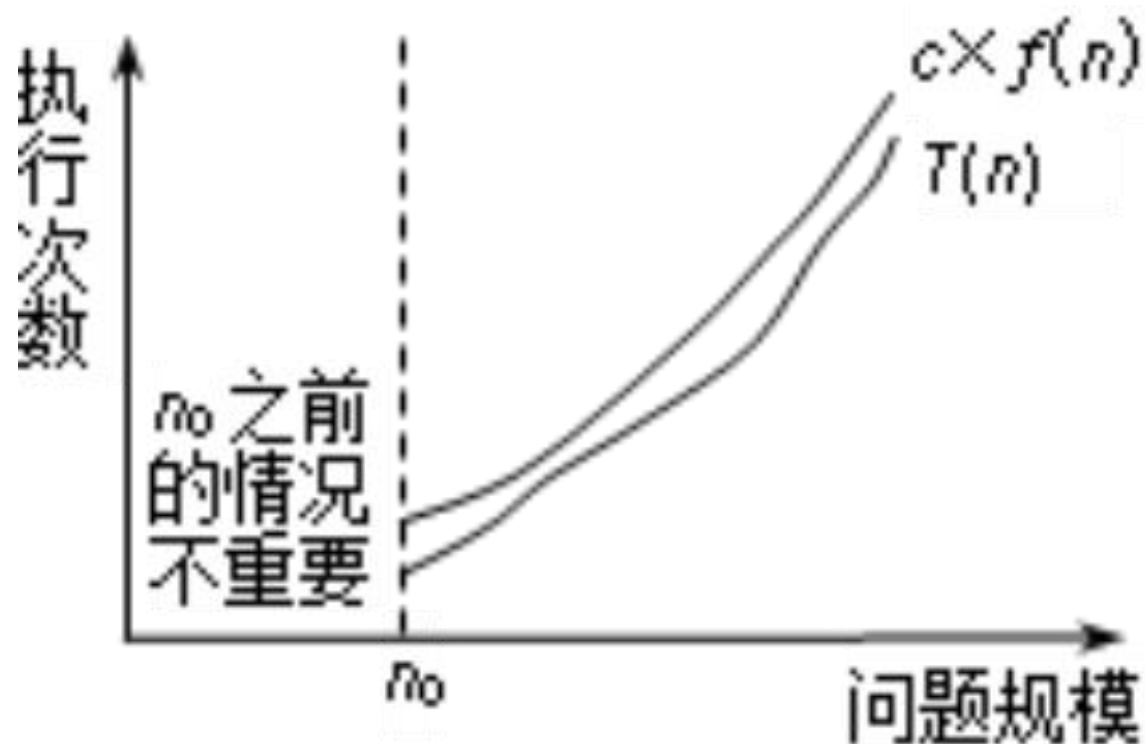


图 2.1 大 O 符号的含义



渐进符号——运行时间的上界

例如：

当有 $T(n) \leq 100n + n$

取 $n_0 = 5$ ，对任意 $n \geq n_0$ ，有：

$$T(n) \leq 100n + n = 101n$$

令 $c = 101$ ， $f(n) = n$ ，有：

$$T(n) \leq cn = cf(n)$$

所以 $T(n) = O(f(n)) = O(n)$



渐进符号——运行时间的下界

定义1-2

若存在两个正的常数 c 和 n_0 ,对于任意 $n \geq n_0$,都有 $T(n) \geq cg(n)$,则称 $T(n) = \Omega(g(n))$ 。

大 Ω 符号用来描述增长率的下限，也就是说，当输入规模为 n 时，算法消耗时间的最小值。与大 O 符号对称，这个下限的阶越高，结果就越有价值。

该算法的运行时间至少是 $\Omega(g(n))$ 。



渐进符号——运行时间的下界

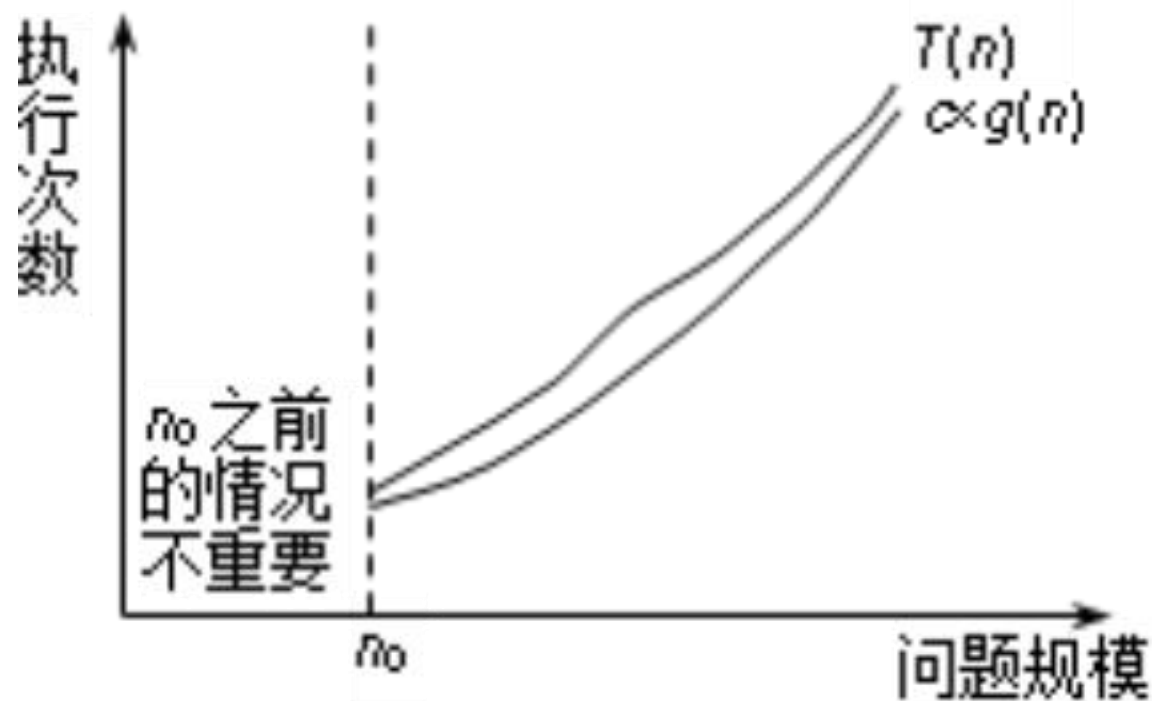


图 2.2 大 Ω 符号的含义



渐进符号——运行时间的下界

例如：

当有 $T(n) \geq n^2 + n \geq n^2$

取 $n_0=1$ ，任意 $n \geq n_0$ ，存在常数 $c=1$ ， $g(n)=n^2$ ，使得： $T(n) \geq n^2 = cg(n)$

所以， $T(n)=\Omega(g(n))$



渐进符号——运行时间的下界

Θ 符号(运行时间的准确界)

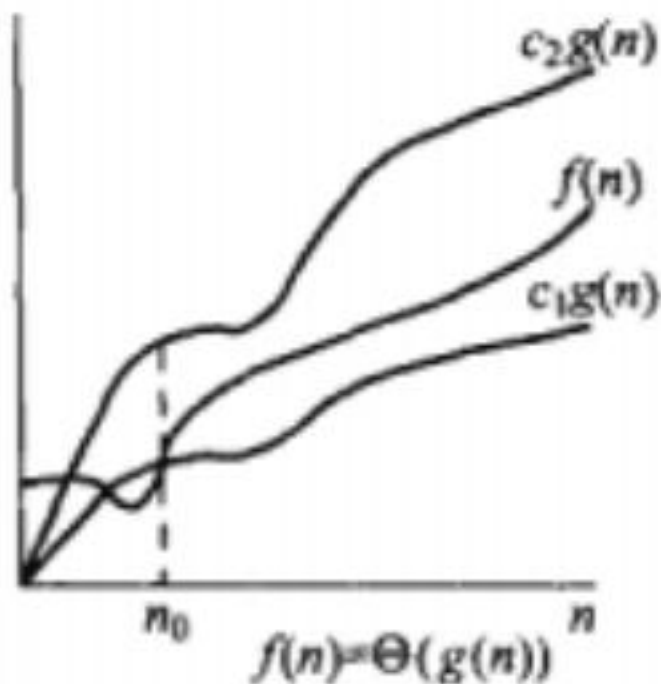
定义1.3 若存在三个正的常数 c_1 、 c_2 和 n_0 , 对于任意 $n \geq n_0$, 都有 $c_1g(n) \geq T(n) \geq c_2 \times g(n)$, 则称 $T(n) = \Theta(g(n))$ 。

Θ 符号意味着 $T(n)$ 与 $g(n)$ 同阶, 用来表示算法的精确阶。

算法复杂性分析—时间复杂度分析



渐进符号——运行时间的准确界





渐进符号——运行时间的准确界

例1 $T(n) = 3n - 1$

例2 $T(n) = 5n^2 + 8n + 1$



常用的级数公式及复杂度

- ❖ 算数级数： $T(n) = 1 + 2 + \dots + n = n(n+1)/2 = O(n^2)$
- ❖ 平方级数： $T(n) = 1^2 + 2^2 + \dots + n^2 = n(n+1)(2n+1)/6 = O(n^3)$
- ❖ 幂级数： $T_a(n) = a^0 + a^1 + a^2 + \dots + a^n, a > 0$
最常见： $1+2+4+\dots+2^n = 2^{n+1}-1 = O(2^{n+1}) = O(2^n)$ //总和与末项同阶
- ❖ 收敛： $1/1/2 + 1/2/3 + 1/3/4 + \dots + 1/(n-1)/n = 1 - 1/n = O(1)$
 $1 + 1/2^2 + \dots + 1/n^2 < 1 + 1/2^2 + \dots = \pi^2/6 = O(1)$
- ❖ 有必要讨论这类级数吗？难道，基本操作次数、存储单元数可能是分数？
- ❖ 调和级数： $h(n) = 1 + 1/2 + 1/3 + \dots + 1/n = \Theta(\log n)$
// $h(n) < 1 + \int_{1 \sim n} 1/x = 1 + \ln n = O(\log n)$
// $h(n) > \int_{1 \sim n+1} 1/x = \ln(n+1) = \Omega(\log n)$
- ❖ 对数级数： $\log 1 + \log 2 + \log 3 + \dots + \log n = \log(n!) = \Theta(n \log n)$



常用复杂度对比

$$\log_a n = \Theta(\log_b n)$$

$$\log n = O(n^k), \text{ 对于任意 } k > 0$$

$$n^k = O(n^m) (k \leq m)$$

$$n^k = O(2^n), \text{ 对于任意 } k$$



渐进符号——渐进比较

传递性

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

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

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



渐进符号——渐进比较

自反性

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

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

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



渐进符号——渐进比较

对称性

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

$$f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$$

$$f(n) = o(g(n)) \Leftrightarrow g(n) = \omega(f(n))$$

算法复杂性分析--时间复杂度分析



1. 最好情况时间复杂度 (best case time complexity) 。
2. 最坏情况时间复杂度 (worst case time complexity) 。
3. 平均情况时间复杂度 (average case time complexity) 。
4. 均摊时间复杂度 (amortized time complexity) 。



□ 三种情况下的时间复杂性

➤ 最坏情况

$$T_{\max}(n) = \max \{ T(I) \mid \text{size}(I)=n \}$$

➤ 最好情况

$$T_{\min}(n) = \min \{ T(I) \mid \text{size}(I)=n \}$$

➤ 平均情况

$$T_{\text{avg}}(n) = \sum_{\text{size}(I)=N} P(I)T(I)$$

其中， I 是问题的规模为 n 的实例， $p(I)$ 是实例 I 出现的概率。

算法复杂性分析--时间复杂度分析



表 2-2 输入规模为 100 万个数据元素时的最好、最坏及平均情况排序时间（单位：毫秒）

| 排 序 算 法 | 平 均 情 况 | 最坏情况（逆序） | 最好情况（正序） |
|---------------|---------|-----------|----------|
| 冒泡排序 | 549 432 | 1 534 035 | 366 936 |
| 选择排序 | 478 694 | 587 240 | 367 658 |
| 插入排序 | 253 115 | 515 621 | 0.897 |
| 希尔排序/增量 3 | 61 | 203 | 35 |
| 堆排序 | 79 | 126 | 74.8 |
| 归并排序 | 70 | 140 | 61 |
| 快速排序 | 39 | 93 | 30 |
| 基数排序/进制 100 | 117 | 118 | 116 |
| 基数排序/进制 1 000 | 89 | 90 | 88 |

算法复杂性分析--时间复杂度分析



□ 非递归算法分析的一般步骤

1. 决定用哪个（或哪些）参数作为算法问题规模的度量

可以从问题的描述中得到。

2. 找出算法中的基本语句

通常是最内层循环的循环体。

3. 检查基本语句的执行次数是否只依赖于问题规模

如果基本语句的执行次数还依赖于其他一些特性，则需要分别研究最好情况、最坏情况和平均情况的效率。

算法复杂性分析--时间复杂度分析



□ 非递归算法分析的一般步骤

4. 建立基本语句执行次数的求和表达式

计算基本语句执行的次数，建立一个代表算法运行时间的求和表达式。

5. 用渐进符号表示这个求和表达式

计算基本语句执行次数的数量级，用大O符号来描述算法增长率的上限。

算法复杂性分析--时间复杂度分析

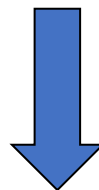


□ 小结:

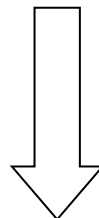
基本技术:

- ◆ 根据循环统计基本语句次数
- ◆ 用递归关系统计基本语句次数
- ◆ 用平摊方法统计基本语句次数

选取基本语句



统计基本语句频数



计算并简化统计结果

渐近复杂度:

O 、 Ω 、 Θ

标准:

- 平均时间复杂
- 最坏时间复杂
- 最好时间复杂度

基本技术:

- 常用求和公式
- 定积分近似求和
- 递归方程求解

目录

第一节

背景介绍

第二节

算法基本概念

第三节

算法与数据结
构、程序的关系

第四节

重要问题类型

第五节

算法时间复杂
度分析

第六节

一个算法例子：
文本距离

一个算法例子——Python 成本模型



Operators in Erik's notes, minus what was covered in lecture:

- 1. list operations:
 - (a) `L1.extend(L2)`
 - (b) `L2 = L1[i:j]`
 - (c) `b = (x in L) or L.index(x) or L.find(x)`
- 2. tuple and str
- 3. set
- 4. heapq

一个算法例子——Python 成本模型



Other points of interest:

- 1. equality checking (e.g. `list1 == list2`)
- 2. lists versus generators

Reference [http://6.006.scripts.mit.edu/~6.006/spring08/wiki/index.](http://6.006.scripts.mit.edu/~6.006/spring08/wiki/index.php?title=Python_Cost_Model)

[php?title=Python_Cost_Model](http://6.006.scripts.mit.edu/~6.006/spring08/wiki/index.php?title=Python_Cost_Model). **The Web site has runtime interpolation for various Python operations. The running times for various-sized inputs were measured, and then a least-square fit was used to find the coefficient for the highest order term in the running time.**

Difference between generators and lists. A good explanation is here: <http://wiki.python.org/moin/Generators>

一个算法例子：文档距离



docdist1.py is a straightforward solution to the document distance problem, and **docdist{2-8}.py** show algorithmic optimizations that improve the running time.

We start out by understanding the structure of the straightforward implementation, and then we'll look at the changes in each of the successive versions.

一个算法例子：文档距离



docdist1

We first look at main to get a high-level view of what's going on in the program.

```
1 def main():
2     if len(sys.argv) != 3:
3         print "Usage: docdist1.py filename_1 filename_2"
4     else:
5         filename_1 = sys.argv[1]
6         filename_2 = sys.argv[2]
7         sorted_word_list_1 = word_frequencies_for_file(filename_1)
8         sorted_word_list_2 = word_frequencies_for_file(filename_2)
9         distance = vector_angle(sorted_word_list_1, sorted_word_list_2)
10        print "The distance between the documents is: %0.6f (radians)"%distance
```

一个算法例子：文档距离



The method processes the command-line arguments, and calls word frequencies for file for each document, then calls vector angle on the resulting lists. How do these methods match up with the three operations outlined in lecture? It seems like word frequencies for file is responsible for operation 1 (split each document into words) and operation 2 (count word frequencies), and then vector angle is responsible for operation 3 (compute dot product).

一个算法例子：文档距离



Next up, we'll take a look at word frequencies for file.

```
1 def word_frequencies_for_file(filename):  
2     line_list = read_file(filename)  
3     word_list = get_words_from_line_list(line_list)  
4     freq_mapping = count_frequency(word_list)  
5     return freq_mapping
```

一个算法例子：文档距离



The method first calls `read file`, which returns a list of lines in the input file. We'll omit the method code, because it is not particularly interesting, and we'll assume that `read file`'s running time is proportional to the size of the input file. The input from `read line` is given to `get words from line list`, which computes operation 1 (split each document into words). After that, `count frequency` turns the list of words into a document vector (operation 2).

一个算法例子：文档距离



```
1  def get_words_from_line_list(L):
2      word_list = []
3      for line in L:
4          words_in_line =
get_words_from_string(line)
5          word_list = word_list + words_in_line
6      return word_list
7
8  def get_words_from_string(line):
9      word_list = []
10     character_list = []
11     for c in line:
12         if c.isalnum():
13             character_list.append(c)
14     elif len(character_list)>0:
15         word = "".join(character_list)
16         word = word.lower()
17         word_list.append(word)
18         character_list = []
19     if len(character_list)>0:
20         word = "".join(character_list)
21         word = word.lower()
22         word_list.append(word)
23     return word_list
```

一个算法例子：文档距离



get words from string takes one line in the input file and breaks it up into a list of words. TODO: line-by-line analysis. The running time is $O(k)$, where k is the length of the line.

get words from line list calls get words from string for each line and combines the lists into one big list. Line 5 looks innocent but is a big

performance killer, because using $+$ to combine $\frac{W}{k}$ lists of length k is $O(\frac{W^2}{k})$

一个算法例子：文档距离



The output of get words from line list is a list of words, like ['a', 'cat', 'in', 'a', 'bag']. word frequencies from file passes this output to count frequency, which turns it into a document vector that counts the number of occurrences of each word, and looks like [['a', 2], ['cat', 1], ['in', 1], ['bag', 1]].

一个算法例子：文档距离



```
1 def count_frequency(word_list):
2     L = []
3     for new_word in word_list:
4         for entry in L:
5             if new_word == entry[0]:
6                 entry[1] = entry[1] + 1
7         break
8     else:
9         L.append([new_word,1])
10    return L
```

The implementation above builds the document vector by takes each word in the input list and looking it up in the list representing the under-construction document vector. In the worst case of a document with all different words, this takes $O(W^2 \times l)$ time, where W is the number of words in the document, and l is the average word length.

一个算法例子：文档距离



count frequency is the last function call in word frequencies for file. Next up, main calls vector angle, which performs operation 3, computing the distance metric.

```
1 def vector_angle(L1,L2):  
2     numerator = inner_product(L1,L2)  
3     denominator = math.sqrt(inner_product(L1,L1)*inner_product(L2,L2))  
4     return math.acos(numerator/denominator)
```

一个算法例子：文档距离



The method is a somewhat straightforward implementation of the distance metric

$$\arccos\left(\frac{L1 \cdot L2}{|L1||L2|}\right) = \arccos\left(\frac{L1 \cdot L2}{\sqrt{(L1 \cdot L1)L2 \cdot L2}}\right)$$

and delegates to inner product for the hard work of computing cross products.

一个算法例子：文档距离



```
1 def inner_product(L1,L2):
2     sum = 0.0
3     for word1, count1 in L1:
4         for word2, count2 in L2:
5             if word1 == word2:
6                 sum += count1 * count2
7     return sum
```

一个算法例子：文档距离



inner product is a straightforward inner-product implementation that checks each each word in the first list against the entire second list. The nested loops at lines 3 and 4 give the algorithm its running time of $O(L_1L_2)$, where L_1 and L_2 are the lengths of the documents' vectors (the number of unique words in each document).

docdist1 Performance Scorecard



| Method | Time |
|---|---|
| get words from line list count frequency word frequencies for file | $O(\frac{W^2}{k}) = O(W^2)$ $O(WL)$ $O(W^2)$ |
| inner product vector angle | $O(L_1L_2)$ $O(L_1L_2+L_1^2+L_2^2)=O(L_1^2+L_2^2)$ |
| main | $O(L_1L_2)$ $O(L_1L_2+L_1^2+L_2^2)=O(L_1^2+L_2^2)$ |

docdist1 Performance Scorecard



We assume that k (number of words per line) is a constant, because the documents will need to fit on screens or paper sheets with a finite length. W (the number of words in a document) is $\geq L$ (the number of unique words in a document).

$L_1L_2 + L_1^2 + L_2^2 = O(L_1^2 + L_2^2)$ because $L_1^2 + L_2^2 \geq L_1L_2$. Proof (assuming $L_1, L_2 \geq 0$):

$$(L_1 - L_2)^2 \geq 0$$

$$L_1^2 + L_2^2 - 2L_1L_2 \geq 0$$

$$L_1^2 + L_2^2 \geq 2L_1L_2$$

$$L_1^2 + L_2^2 \geq L_1L_2$$

一个算法例子：文档距离



docdist2 The document distance code invokes the Python profiler to identify the code that takes up the most CPU time. This ensures that we get the biggest returns on our optimization efforts.

```
1  if __name__ == "__main__":  
2      import cProfile  
3      cProfile.run("main()")
```

一个算法例子：文档距离



You can profile existing programs without changing them by adding `-m cProfile` to Python's command line. For example, the command below will run and profile `program.py`.

```
python -m cProfile -s time program.py
```

The profiler output for `docdist1` shows that the biggest time drain is `get words` from line list. The problem is that when the `+` operator is used to concatenate lists, it needs to create a new list and copy the elements of both its operands. Replacing `+` with `extend` yields a 30% runtime improvement.

一个算法例子：文档距离



Next up, we'll take a look at word frequencies for file.

```
1 def get_words_from_line_list(L):  
2     word_list = []  
3     for line in L:  
4         words_in_line = get_words_from_string(line)  
5         word_list.extend(words_in_line)  
6     return word_list
```

一个算法例子：文档距离



extend adds all the elements of an m -element list to an n -element list in $O(1+m)$, as opposed to $+$, which needs to create a new list, and therefore takes $O(1 + n + m)$ time. So concatenating $\frac{W}{k}$ lists of k elements takes $\sum \frac{W}{k} k = O(W)$ time.

docdist2 Performance Scorecard



| Method | Time |
|----------------------------------|--------------------|
| get words from line list | $O(W)$ |
| count frequency | $O(WL)$ |
| word frequencies for file | $O(WL)$ |
| inner product | $O(L_1L_2)$ |
| vector angle | $O(L_1^2+L_2^2)$ |
| main | $O(W_1L_1+W_2L_2)$ |

一个算法例子：文档距离



docdist3 Profiling docdist2 points to count frequency and inner product as good targets for `op_x0002_tinizations`. We'll speed up inner product by switching to a fast algorithm. However, the algorithm assumes that the words in the document vectors are sorted. For example, `[[‘a’, 2],[‘cat’, 1], [‘in’, 1], [‘bag’, 1]]` needs to become `[[‘a’, 2], [‘bag’, 1],[‘cat’, 1], [‘in’, 1]]`.

一个算法例子：文档距离



First off, we add a step to word frequencies for file that sorts the document vector produced by count frequency.

```
1 def word_frequencies_for_file(filename):
2     line_list = read_file(filename)
3     word_list = get_words_from_line_list(line_list)
4     freq_mapping = count_frequency(word_list)
5     insertion_sort(freq_mapping)
6     return freq_mapping
```

一个算法例子：文档距离



Then we implement insertion sort using the algorithm in the textbook.

```
1 def insertion_sort(A):  
2     for j in range(len(A)):  
3         key = A[j]  
4         i = j-1  
5         while i>-1 and A[i]>key:  
6             A[i+1] = A[i]  
7             i = i-1  
8         A[i+1] = key  
9     return A
```

一个算法例子：文档距离



Insertion sort runs in $O(L^2)$ time, where L is the length of the document vector to be sorted. Finally, inner product is re-implemented using a similar algorithm to the merging step in Merge Sort.

一个算法例子：文档距离



```
1 def inner_product(L1,L2):
2     sum = 0.0
3     i = 0
4     j = 0
5     while i<len(L1) and j<len(L2):
6         # L1[i:] and L2[j:] yet to be processed
7         if L1[i][0] == L2[j][0]:
8             # both vectors have this word
9             sum += L1[i][1] * L2[j][1]
10            i += 1
11            j += 1
12        elif L1[i][0] < L2[j][0]:
13            # word L1[i][0] is in L1 but not L2
14            i += 1
15        else:
16            # word L2[j][0] is in L2 but not L1
17            j += 1
18    return sum
```

一个算法例子：文档距离



The new implementation runs in $O(L_1 + L_2)$, where L_1 and L_2 are the lengths of the two document vectors. We observe that the running time for inner product (and therefore for vector angle) is asymptotically optimal, because any algorithm that computes the inner product will have to read the two vectors, and reading will take $\Omega(L_1 + L_2)$ time.

docdist3 Performance Scorecard



| Method | Time |
|----------------------------------|---------------------|
| get words from line list | $O(W)$ |
| count frequency | $O(WL)$ |
| insertion sort | $O(L^2)$ |
| word frequencies for file | $O(WL+L^2) = O(WL)$ |
| inner product | $O(L_1+L_2)$ |
| vector angle | $O(L_1+L_2)$ |
| main | $O(W_1L_1+W_2L_2)$ |

一个算法例子：文档距离



docdist4 The next iteration addresses count frequency, which is the biggest time consumer at the moment.

```
1 def count_frequency(word_list):
2     D = {}
3     for new_word in word_list:
4         if new_word in D:
5             D[new_word] = D[new_word]+1
6         else:
7             D[new_word] = 1
8     return D.items()
```

一个算法例子：文档距离



The new implementation uses Python dictionaries. The dictionaries are implemented using hash tables, which will be presented in a future lecture. The salient feature of hash tables is that inserting an element using `dictionary[key] = value` and looking up an element using `dictionary[key]` both run in $O(1)$ expected time.

Instead of storing the document vector under construction in a list, the new implementation uses a dictionary. The keys are the words in the document, and the value are the number of times each word appears in the document. Since both insertion (line 5) and lookup (line 7) take $O(1)$ time, building a document vector out of W words takes $O(W)$ time.

docdist4 Performance Scorecard



| Method | Time |
|----------------------------------|---------------------|
| get words from line list | $O(W)$ |
| count frequency | $O(W)$ |
| insertion sort | $O(L^2)$ |
| word frequencies for file | $O(W+L^2) = O(L^2)$ |
| inner product | $O(L_1+L_2)$ |
| vector angle | $O(L_1+L_2)$ |
| main | $O(L_1^2+L_2^2)$ |

一个算法例子：文档距离



docdist5 This iteration simplifies get words from string that breaks up lines into words. First off, the standard library function `string.translate` is used for converting uppercase characters to lowercase, and for converting punctuation to spaces. Second, the `split` method on strings is used to break up a line into words.

一个算法例子：文档距离



```
1 translation_table = string.maketrans(string.punctuation+string.uppercase,  
2                                     " "*len(string.punctuation)+string.lowercase)  
3  
4 def get_words_from_string(line):  
5     line = line.translate(translation_table)  
6     word_list = line.split()  
7     return word_list
```

一个算法例子：文档距离



The main benefit of this change is that 23 lines of code are replaced with 5 lines of code. This makes the implementation easier to analyze. A side benefit is that many functions in the Python standard library are implemented directly in C (a low-level programming language that is very close to machine code), which gives them better performance. Although the running time is asymptotically the same, the hidden constants are much better for the C code than for our Python code presented in `docdist1`.

docdist5 Performance Scorecard



Identical to the docdist4 scorecard.

一个算法例子：文档距离



docdist6 Now that all the distractions are out of the way, it's time to tackle insertion sort, which takes up the most CPU time, by far, in the profiler output for docdist5.

The advantages of insertion sort are that it sorts in place, and it is simple to implement. However, its worst-case running time is $O(N^2)$ for an N -element array. We'll replace insertion sort with a better algorithm, merge sort. Merge sort is not in place, so we'll need to modify word frequencies for file.

一个算法例子：文档距离



```
1 def word_frequencies_for_file(filename):  
2     line_list = read_file(filename)  
3     word_list = get_words_from_line_list(line_list)  
4     req_mapping = count_frequency(word_list)  
5     freq_mapping = merge_sort(freq_mapping)  
6     return freq_mapping
```

The merge sort implementation closely follows the pseudocode in the textbook.

一个算法例子：文档距离



```
1  def merge_sort(A):
2      n = len(A)
3      if n==1:
4          return A
5      mid = n//2
6      L = merge_sort(A[:mid])
7      R = merge_sort(A[mid:])
8      return merge(L,R)
9
10 def merge(L,R):
11     i = 0
12     j = 0
13     answer = []
14     while i<len(L) and j<len(R):
15         if L[i]<R[j]:
16             answer.append(L[i])
17             i += 1
18         else:
19             answer.append(R[j])
20             j += 1
21     if i<len(L):
22         answer.extend(L[i:])
23     if j<len(R):
24         answer.extend(R[j:])
25     return answer
```

一个算法例子：文档距离



docdist6 The textbook proves that merge sort runs in $O(n \log n)$ time. You should apply your knowledge of the Python cost model to convince yourself that the implementation above also runs in $O(n \log n)$ time.

docdist6 Performance Scorecard



| Method | Time |
|----------------------------------|----------------------------------|
| get words from line list | $O(W)$ |
| count frequency | $O(W)$ |
| insertion sort | $O(L \log L)$ |
| word frequencies for file | $O(W+L \log L) = O(L \log L)$ |
| inner product | $O(L_1+L_2)$ |
| vector angle | $O(L_1+L_2)$ |
| main | $O(L_1 \log L_1 + L_2 \log L_2)$ |

一个算法例子：文档距离



docdist6 TSwitching to merge sort improved the running time dramatically. However, if we look at docdist6's profiler output, we notice that merge is the function with the biggest runtime footprint. Merge sort's performance in practice is great, so it seems that the only way to make our code faster is to get rid of sorting altogether.

一个算法例子：文档距离



This iteration switches away from the sorted list representation of document vectors, and instead uses the Python dictionary representation that was introduced in docdist4. count frequency already used that representation internally, so we only need to remove the code that converted the Python dictionary to a list.

一个算法例子：文档距离



```
1 def count_frequency(word_list):
2     D = {}
3     for new_word in word_list:
4         if new_word in D:
5             D[new_word] = D[new_word]+1
6         else:
7             D[new_word] = 1
8     return D
```

一个算法例子：文档距离



This method still takes $O(W)$ time to process a W -word document.
word frequencies for file does not call merge sort anymore, and instead returns the dictionary built by count frequency.

```
1 def word_frequencies_for_file(filename):  
2     line_list = read_file(filename)  
3     word_list = get_words_from_line_list(line_list)  
4     freq_mapping = count_frequency(word_list)  
5     return freq_mapping
```


一个算法例子：文档距离



Next up, inner product makes uses dictionary lookups instead of merging sorted lists.

```
1 def inner_product(D1,D2):
2     sum = 0.0
3     for key in D1:
4         if key in D2:
5             sum += D1[key] * D2[key]
6     return sum
```

一个算法例子：文档距离



The logic is quite similar to the straightforward inner product in docdist1. Each word in the first document vector is looked up in the second document vector. However, because the document vectors are dictionaries, each takes $O(1)$ time, and inner product runs in $O(L_1)$ time, where L_1 is the length of the first document's vector.

docdist7 Performance Scorecard



| Method | Time |
|----------------------------------|--------------|
| get words from line list | $O(W)$ |
| count frequency | $O(W)$ |
| word frequencies for file | $O(W)$ |
| inner product | $O(L_1+L_2)$ |
| vector angle | $O(L_1+L_2)$ |
| main | $O(W_1+W_2)$ |

一个算法例子：文档距离



docdist8 At this point, all the algorithms in our solution are asymptotically optimal. We can easily prove this, by noting that each of the 3 main operations runs in time proportional to its input size, and each operation needs to read all its input to produce its output. However, there is still room for optimizing and simplifying the code.

一个算法例子：文档距离



There is no reason to read each document line by line, and then break up each line into words. The last iteration processes reads each document into one large string, and breaks up the entire document into words at once.

一个算法例子：文档距离



First off, read file is modified to return a single string, instead of an array of strings. Then, get words from line list is renamed to get words from file, and is simplified, because it doesn't need to iterate over a list of lines anymore. Last, word frequencies for file is updated to reflect the renaming.

一个算法例子：文档距离



```
1 def get_words_from_text(text):
2     text = text.translate(translation_table)
3     word_list = text.split()
4     return word_list
5
6 def word_frequencies_for_file(filename):
7     text = read_file(filename)
8     word_list = get_words_from_text(text)
9     freq_mapping = count_frequency(word_list)
10    return freq_mapping
```

docdist8 Performance Scorecard



| Method | Time |
|----------------------------------|--------------|
| get words from line list | $O(W)$ |
| count frequency | $O(W)$ |
| word frequencies for file | $O(W)$ |
| inner product | $O(L_1+L_2)$ |
| vector angle | $O(L_1+L_2)$ |
| main | $O(W_1+W_2)$ |



湖南大学
HUNAN UNIVERSITY

谢谢观赏

—— 实事求是 敢为人先 ——