



# 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*



# 教材及考核方式

## 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

## 考核形式:

- (1) 课堂点名 (占10%)
- (2) Leetcode (占30%)
- (3) 期末考试 (占60%)

## 课程网站:

<https://bnu05pp.github.io/AlgorithmCourse/2023.html>



# Leetcode考核说明



- 每周周三8:00到24:00之间，将自己Leetcode截屏发给我学生邮箱2335346159@qq.com，邮件标题格式为"Leetcode截屏-[姓名]-[学号]-[日期]"
- 每周要求起码做3道题目，题目范围参照课程进度。每周3道题目难度全是简单，当次成绩是及格；有1个中等，当次成绩是良好；有1个困难，当次成绩是优秀。

 学习 题库 竞赛 讨论 求职 商店

Plus 会员 力扣面试



**彭鹏Hello\_world**  
peng-peng-hello\_world  
全站排名 100,000

关注了  
0

关注者  
0

编辑个人资料

定制你的理想工作机会

解题数量

2  
解决问题

|    |          |      |
|----|----------|------|
| 简单 | 2 / 744  | 暂无数据 |
| 中等 | 0 / 1558 | 暂无数据 |
| 困难 | 0 / 636  | 暂无数据 |

勋章成就

0

即将获得  
月度每日刷题全部达成

# 提纲



1.1 背景介绍

1.2 算法基本概念

1.3 算法与数据结构、程序的关系

1.4 重要问题类型

1.5 算法时间复杂度分析

1.6 一个算法例子：文本距离

## 1.1 背景介绍

## 1.2 算法基本概念

## 1.3 算法与数据结构、程序的关系

## 1.4 重要问题类型

## 1.5 算法时间复杂度分析

## 1.6 一个算法例子：文本距离

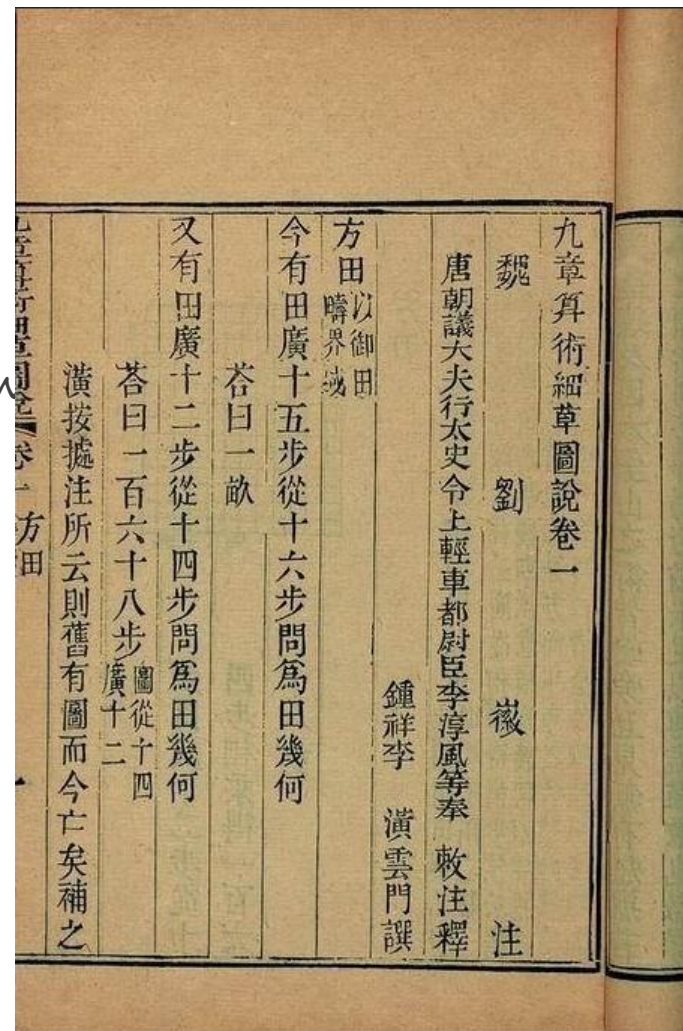
# 1.1 算法历史

## □ “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-Khw)

## ➤ 图灵机



# 1.1 算法历史

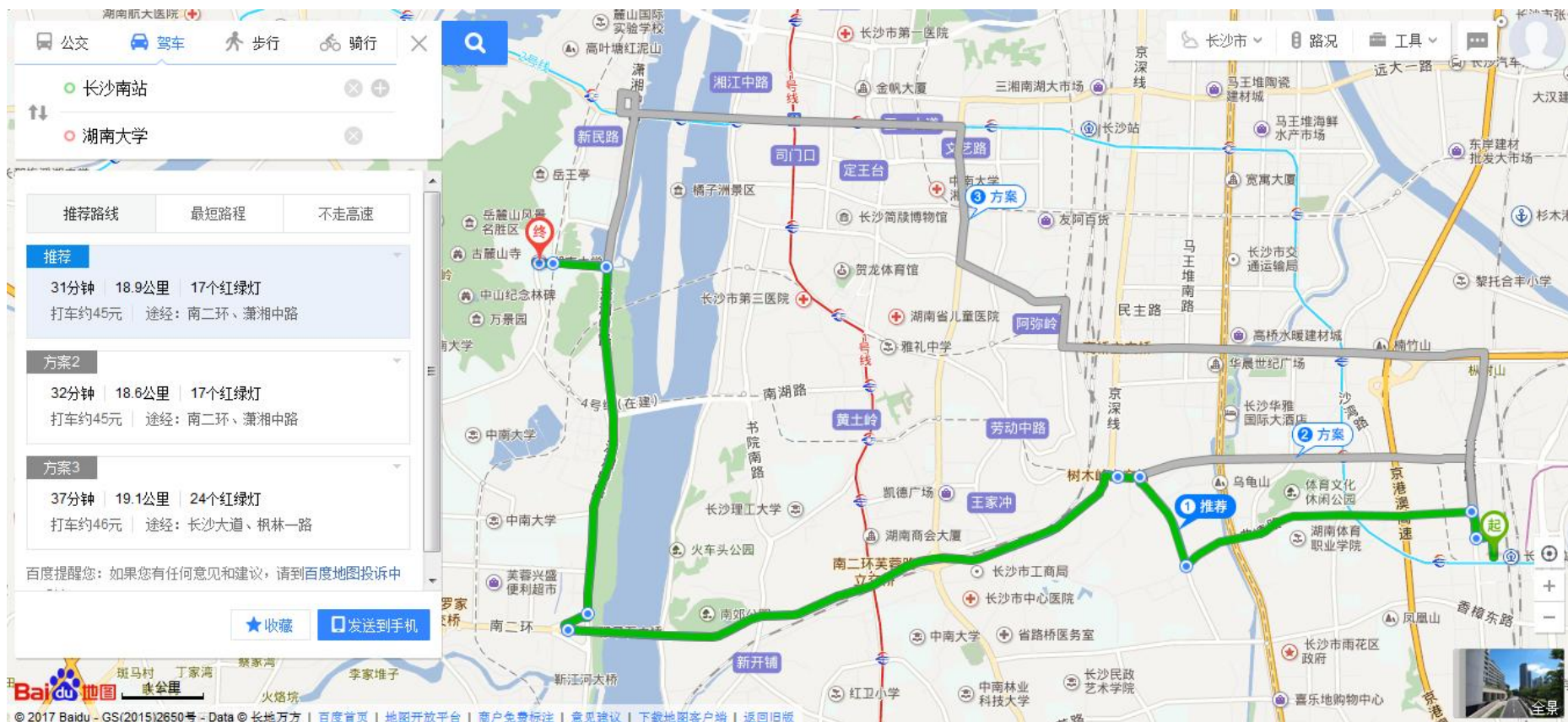


## □ 二十世纪十大著名算法

- 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



# 1.1 算法与生活-导航



# 1.1 算法与生活-导航



# 1.1 算法与生活-导航





1.1 背景介绍

**1.2 算法基本概念**

1.3 算法与数据结构、程序的关系

1.4 重要问题类型

1.5 算法时间复杂度分析

1.6 一个算法例子：文本距离

# 1.2 算法基本概念



## □ 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。

# 1.2 算法基本概念



## □ 算法的五个重要特性

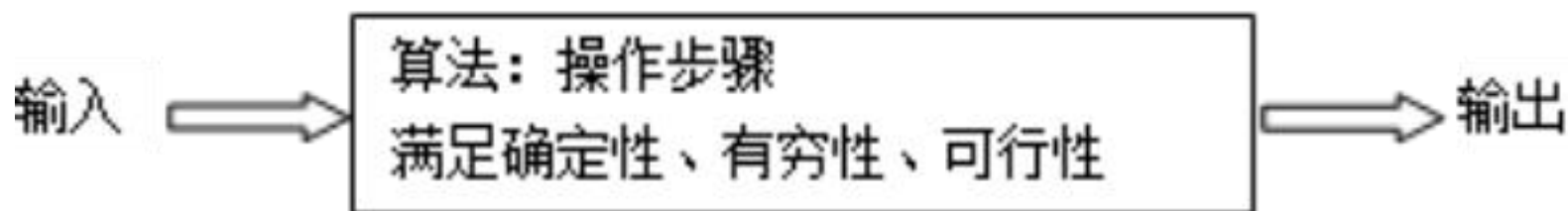


图 1.1 算法的概念

# 1.2 算法基本概念



## □ “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

# 1.2 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

$$sum = 1 + 2 + 3 + 4 + 5 + \dots + (n-1) + n$$

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。

武松打死老虎

武松/打死老虎

武松/打/死老虎



# 1.2 算法基本概念



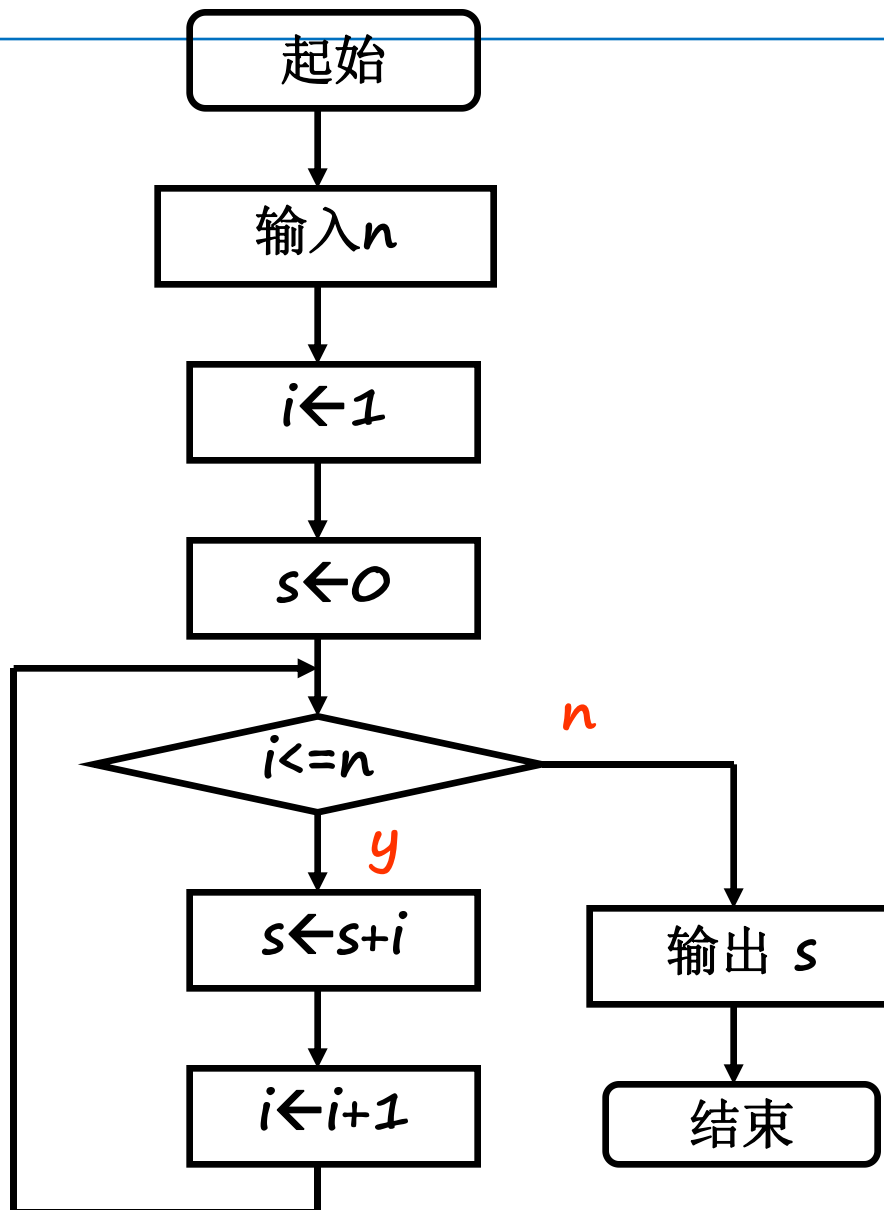
## □ 算法的描述方法

➤ 自然语言

➤ **流程图**

➤ 程序设计语言

➤ 伪代码



# 1.2 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

➤ 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。                                     |

# 1.2 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(代码)**
- 伪代码

# 1.2 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\*/

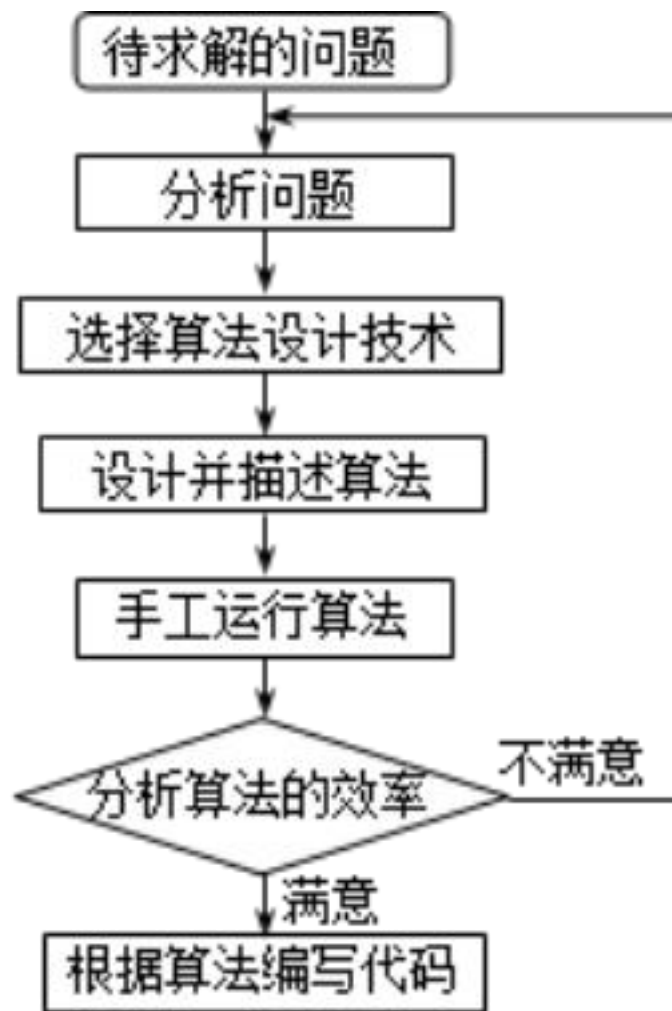
输出  $s$  的值;

算法结束

# 1.2 算法基本概念



## □ 算法设计的一般过程



# 提纲



1.1 背景介绍

1.2 算法基本概念

**1.3 算法与数据结构、程序的关系**

1.4 重要问题类型

1.5 算法时间复杂度分析

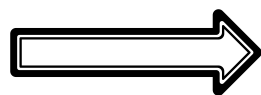
1.6 一个算法例子：文本距离

## 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

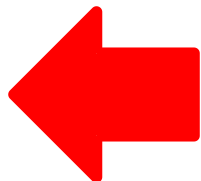


数据多维属性



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。



# 提纲



1.1 背景介绍

1.2 算法基本概念

1.3 算法与数据结构、程序的关系

**1.4 重要问题类型**

1.5 算法时间复杂度分析

1.6 一个算法例子：文本距离

# 1.4 重要问题类型



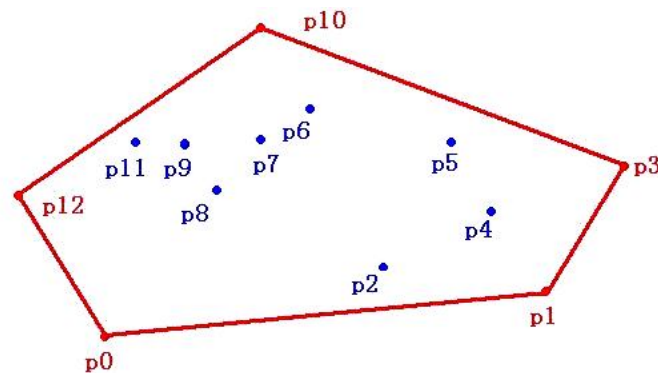
□ 查找问题

□ 排序问题

□ 图问题

□ 组合问题

□ 几何问题



# 提纲



1.1 背景介绍

1.2 算法基本概念

1.3 算法与数据结构、程序的关系

1.4 重要问题类型

**1.5 算法时间复杂度分析**

1.6 一个算法例子：文本距离

# 1.5 算法复杂性分析



□影响软件（算法）性能的因素？请同学们列举几项。

# 1.5 算法复杂性分析



## 10种排序算法的时间效率比较

| 算法输入<br>数据 | N=50<br>K=50 | N=200<br>K=100 | N=500<br>K=500 | N=2000<br>K=2000 | N=5000<br>K=8000 | N=10000<br>K=20000 | N=20000<br>K=20000 | N=20000<br>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               |

# 1.5 算法复杂性分析



## □ 算法的好与坏

### ➤ 高斯的故事

$$1+2+3+4+5+6+\dots+100$$

$$(1+100) + (2+99) \dots +$$

$$(50+51) = 50 \times 101 \quad (\text{高斯算})$$



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。”

# 1.5 算法复杂性分析



## □ 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。

# 1.5 算法复杂性分析



## □ 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

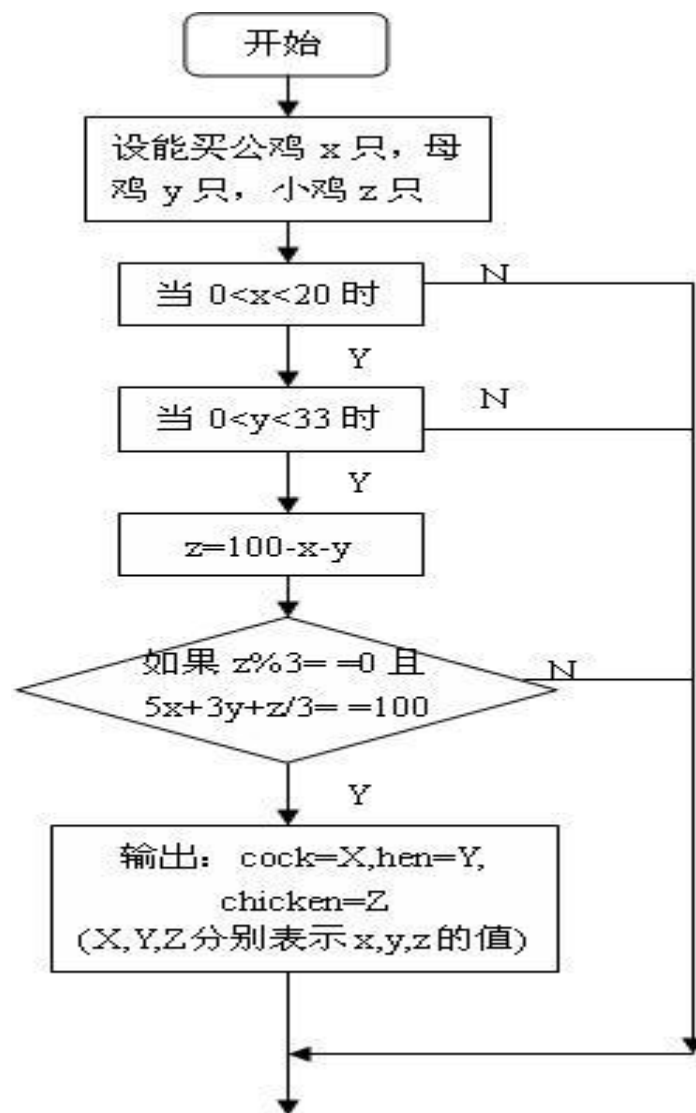
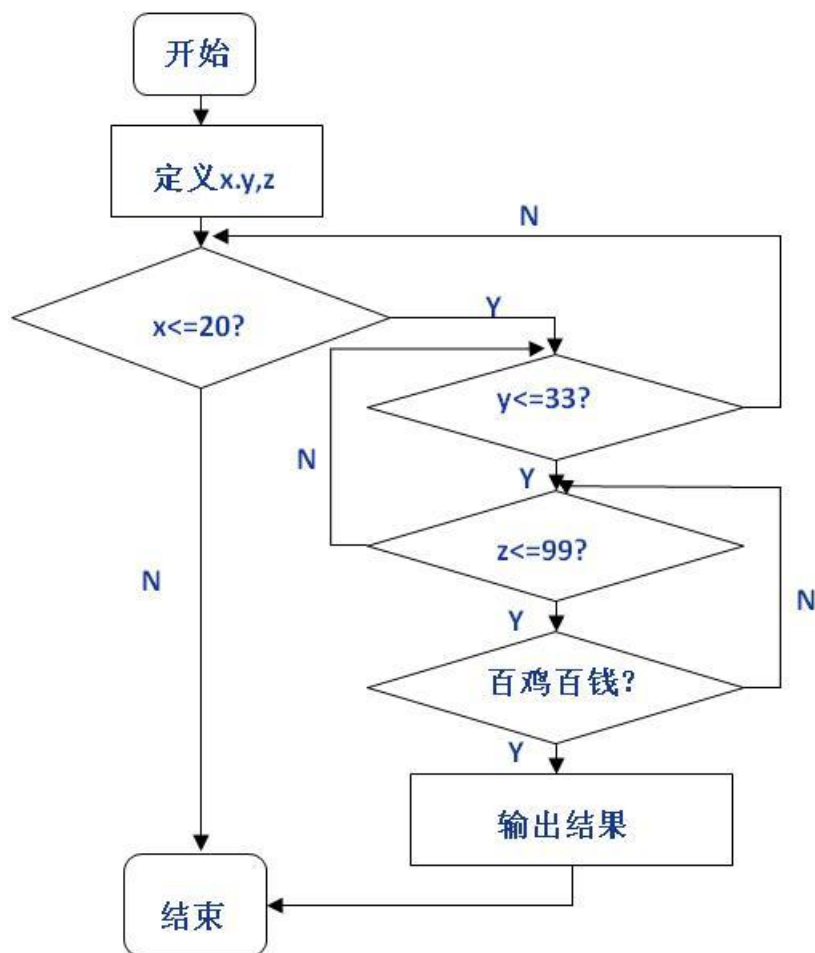
如何建立数学模型？



# 1.5 算法复杂性分析



## 百钱百鸡问题



# 1.5 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)
```

# 1.5 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？

# 1.5 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

# 1.5 算法复杂性分析



□ 什么样的算法是高效算法?

➤ *Correct*

➤ *Fast*

➤ *Small space*

➤ *General*

➤ *Simple*

➤ *Clever*

*Cool*

# 1.5 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



# 1.5 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

# 1.5 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1-事后统计

➤ 利用计算机内记时功能。

➤ 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。



# 1.5 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

➤ 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

# 1.5 算法复杂性分析-时间复杂度分析



## □ 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。

# 1.5 算法复杂性分析-时间复杂度分析

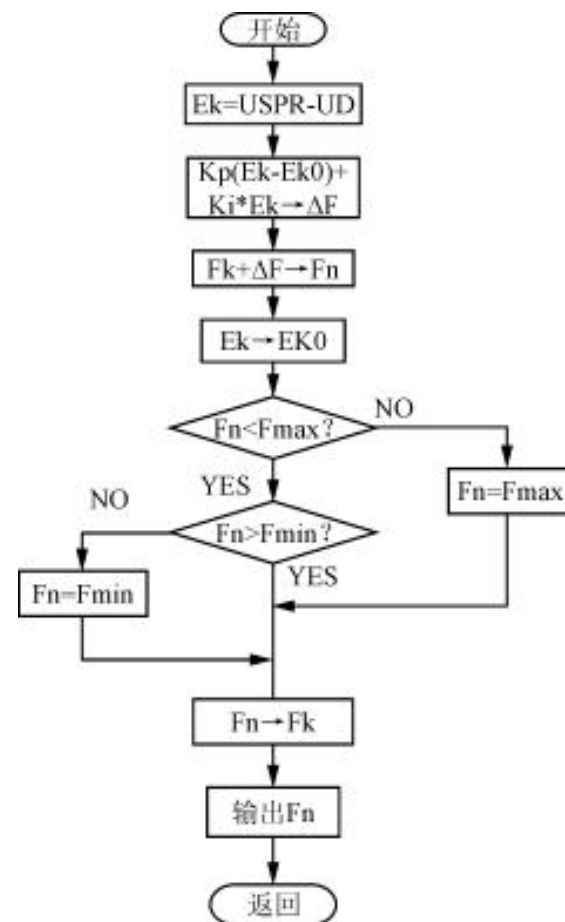


问题规模可控

## 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 $O(f(n))$ 。

# 1.5 算法复杂性分析-时间复杂度分析



## □ 渐进分析

| 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            |

# 1.5 算法复杂性分析-时间复杂度分析



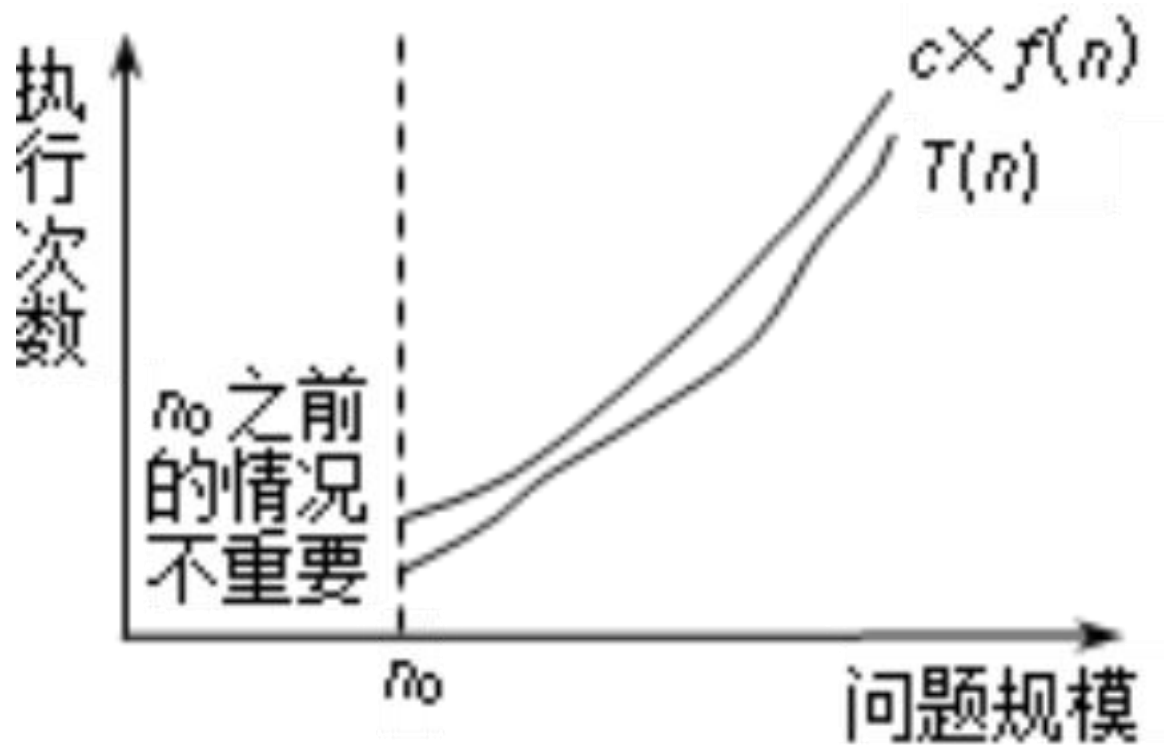
## □ 渐进分析

| 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       |

# 1.5 算法复杂性分析-时间复杂度分析



## 渐进符号——运行时间的上界



大  $O$  符号的含义



## 渐近符号——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  $T(n) = O(f(n)) = O(n)$





## 渐近符号——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 $\Omega$ 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

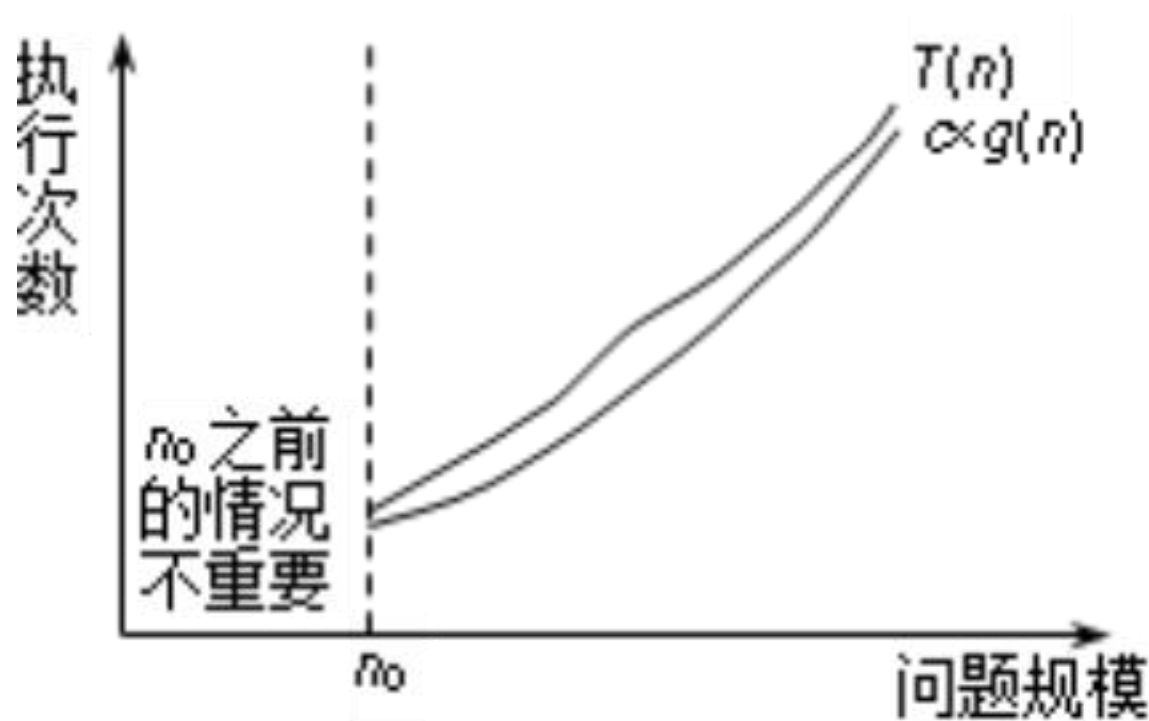


图 2.2 大  $\Omega$  符号的含义



## 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——运行时间的准确界

$\Theta$ 符号(运行时间的准确界)

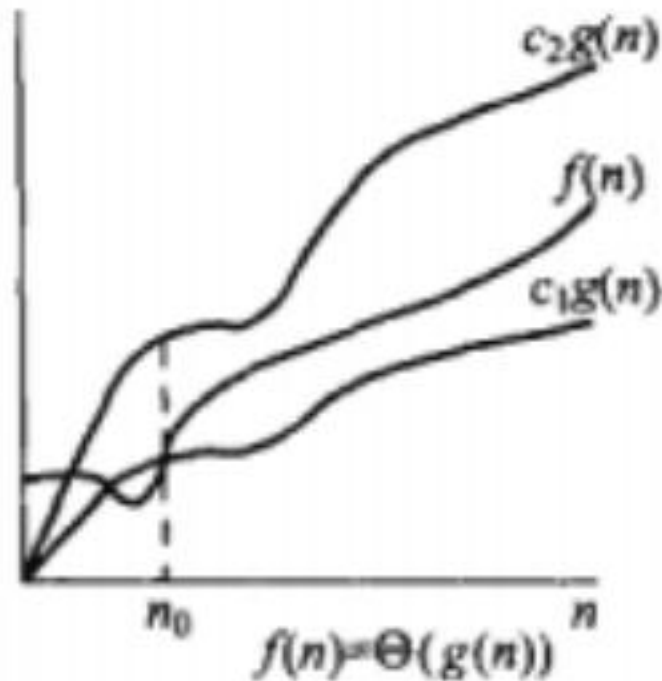
定义1.3 若存在三个正的常数 $c_1$ 、 $c_2$ 和 $n_0$ , 对于任意 $n \geq n_0$ , 都有 $c_1g(n) \geq T(n) \geq c_2 \times g(n)$ , 则称 $T(n) = \Theta(g(n))$ 。

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

# 1.5 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



# 1.5 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

例1  $T(n) = 3n - 1$

例2  $T(n) = 5n^2 + 8n + 1$

# 1.5 算法复杂性分析-时间复杂度分析



## 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： $\log 1 + \log 2 + \log 3 + \dots + \log n = \log(n!) = \Theta(n \log n)$

# 1.5 算法复杂性分析-时间复杂度分析



常用的级数公式及复杂度

常用的求和公式(级数求和).docx





## 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

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

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

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

# 1.5 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*) 。

# 1.5 算法复杂性分析-时间复杂度分析



## □ 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。

# 1.5 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       |

# 1.5 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。
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。

# 1.5 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:

$O$ 、 $\Omega$ 、 $\Theta$

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



# 提纲



1.1 算法历史

1.2 算法与生活

1.3 算法基本概念

1.4 算法与数据结构、程序的关系

1.5 重要问题类型

1.6 算法时间复杂度分析

**1.7 一个算法例子：文本距离**

# 1.7 一个算法例子：文本相似度分析



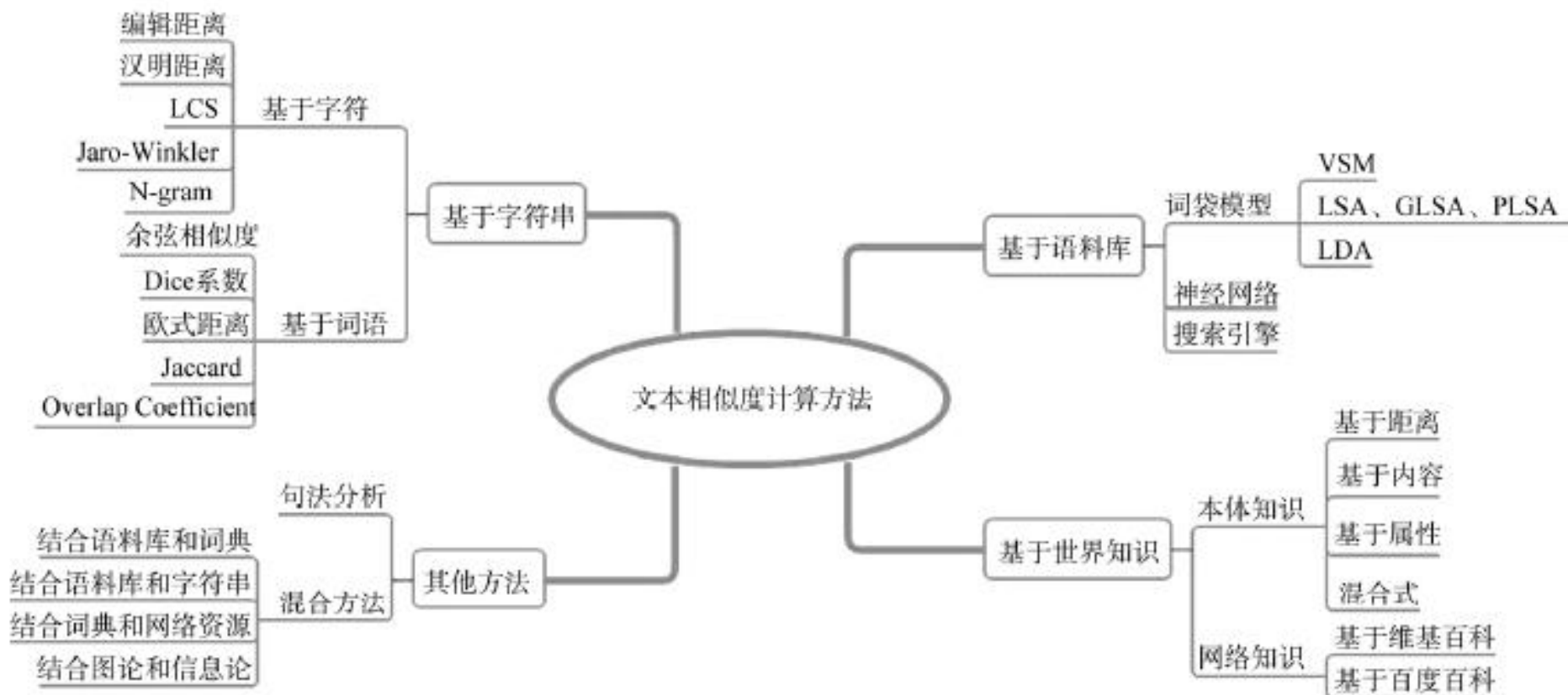
- Given two documents, how similar are they?

- Applications:

- Find similar documents
- Detect plagiarism / duplicates
- Web search
- (one “document” is query)
- 自动客服



# 1.7 一个算法例子：文本相似度分析

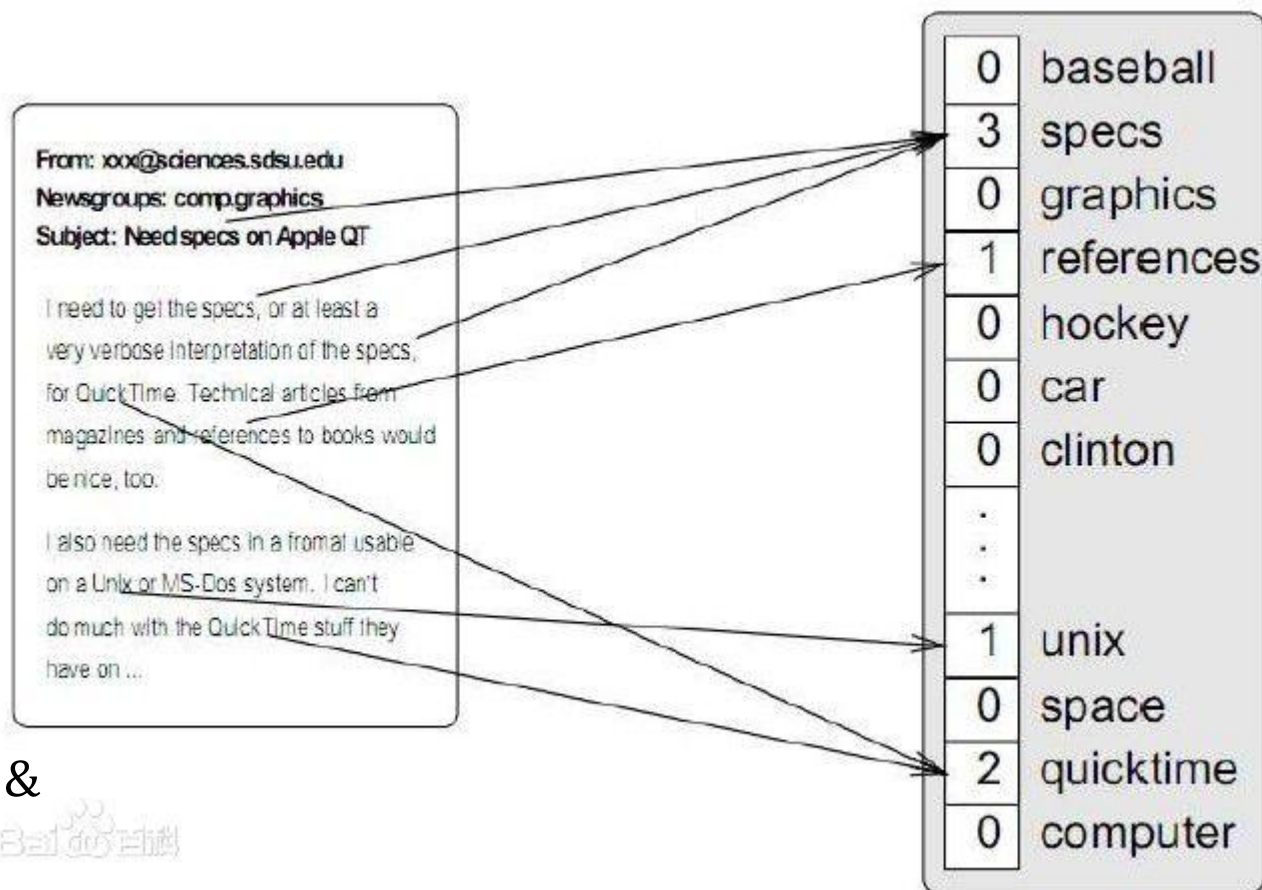


# 1.7 一个算法例子：文本相似度分析



## □ 向量空间模型(Vector Space Model, VSM)

- How to define “document”?
- **Word** = sequence of alphanumeric characters
- **Document** = sequence of words
  - Ignore punctuation & formatting



# 1.7 一个算法例子：文本相似度分析



## □ 向量空间模型(*Vector Space Model, VSM*)

- How to define “distance”?
- Idea: focus on shared words
- **Word frequencies:**
  - $D(w) = \#$   
occurrences of word  
 $w$  in document  $D$

# 1.7 一个算法例子：文本相似度分析



## □ 向量空间模型 (Vector Space Model, VSM)

- Treat each document  $D$  as a vector of its words
  - One coordinate  $D(w)$  for every possible word  $w$

- Example:

- $D_1$  = “the cat”

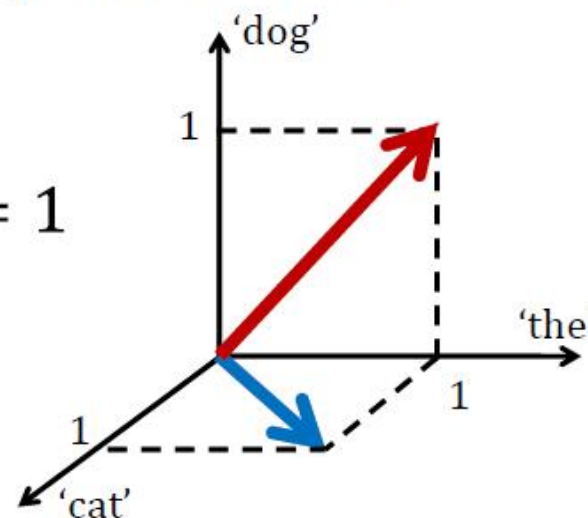
- $D_2$  = “the dog”

$$D_1 \cdot D_2 = 1$$

- Similarity between vectors?

- **Dot product:**

$$D_1 \cdot D_2 = \sum_w D_1(w) \cdot D_2(w)$$



<http://portal.acm.org/citation.cfm?id=361220>

# 1.7 一个算法例子：文本相似度分析



- Problem: Dot product not scale invariant

- Example 1:

–  $D_1$  = “the cat”

–  $D_2$  = “the dog”

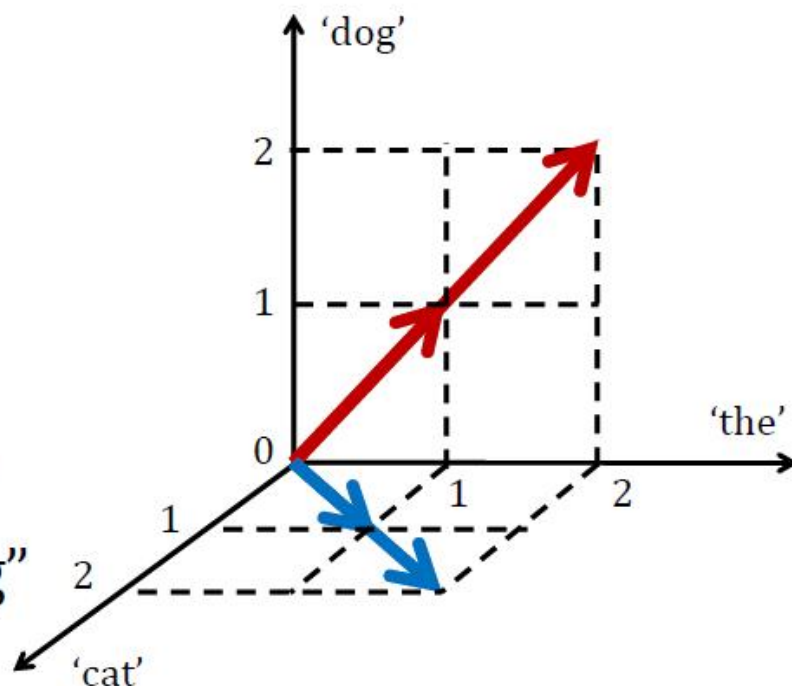
–  $D_1 \cdot D_2 = 1$

- Example 2:

–  $D_1$  = “the cat the cat”

–  $D_2$  = “the dog the dog”

–  $D_1 \cdot D_2 = 4$



<http://portal.acm.org/citation.cfm?id=361220>



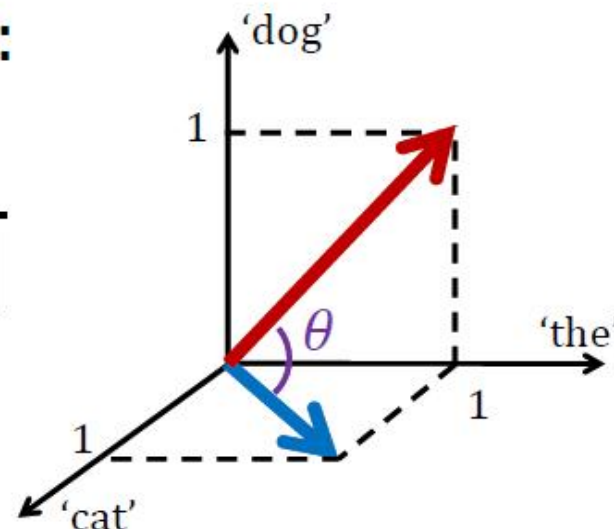
# 1.7 一个算法例子：文本相似度分析



- Idea: Normalize by # words:

$$\cos \theta = \frac{D_1 \cdot D_2}{||D_1|| \cdot ||D_2||}$$

- Geometric solution:  
angle between vectors



$$\theta(D_1, D_2) = \arccos \frac{D_1 \cdot D_2}{||D_1|| \cdot ||D_2||}$$

– 0 = “identical”, 90° = orthogonal (no shared words)

<http://portal.acm.org/citation.cfm?id=361220>



# 1.7 一个算法例子： 文本相似度分析



## □ Algorithm

1. Read documents
2. Split each document into words
3. Count word frequencies (document vectors)
4. Compute document distance

# 1.7 一个算法例子：文本相似度分析



## □ Algorithm

1. Read documents

**2. Split each document into words**

- `re.findall('\w+', doc)`
- But how does this actually work?

3. Count word frequencies (document vectors)

4. Compute dot product

# 1.7 一个算法例子：文本相似度分析



## □ Algorithm

1. Read documents

- 2. Split each document into words**

- For each line in document:

  - For each character in line:

    - If not alphanumeric:

      - Add previous word

        - (if any) to list

      - Start new word

3. Count word frequencies (document vectors)

4. Compute dot product

# 1.7 一个算法例子：文本相似度分析



## □ Algorithm

1. Read documents
2. Split each document into words
3. **Count word frequencies (document vectors)**
  - a. Sort the word list
  - b. For each word in word list:
    - If same as last word:  
Increment counter
    - Else:  
Add last word and its counter to list  
Reset counter to 0
4. Compute dot product

# 1.7 一个算法例子：文本相似度分析



## □ Algorithm

1. Read documents
2. Split each document into words
3. Count word frequencies (document vectors)

### 4. **Compute dot product:**

For every possible word:  $\leftarrow \infty$  iterations...

Look up frequency in each document

Multiply

Add to total

# 1.7 一个算法例子：文本相似度分析



## □ Algorithm

1. Read documents
2. Split each document into words
3. Count word frequencies (document vectors)
4. **Compute dot product:**

For every word in first document:  $\leftarrow w_1$  iterations  
If it appears in second document:  $\leftarrow O(w_2)$   
Multiply word frequencies  $\leftarrow O(1)$   
Add to total  
 $O(w_1 w_2)$

# 1.7 一个算法例子：文本相似度分析



## □ Algorithm

1. Read documents
2. Split each document into words
3. Count word frequencies (document vectors)

### 4. **Compute dot product:**

- a. Start at first word of each document (in sorted order)
- b. If words are equal:  
    Multiply word frequencies  
    Add to total
- c. In whichever document has lexically lesser word, advance to next word
- d. Repeat until either document out of words

$\left. \begin{array}{l} \text{b. If words are equal:} \\ \text{    Multiply word frequencies} \\ \text{    Add to total} \end{array} \right\} O(|\text{word}|)$   
 $\left. \begin{array}{l} \text{c. In whichever document has lexically} \\ \text{    lesser word, advance to next word} \end{array} \right\} O(|\text{doc}|)$

# 1.7 一个算法例子：文本相似度分析



□ Python Implementations: 详见代码

➤ `docdist1.py`

➤ `docdist2.py`

➤ `docdist3.py`

➤ `docdist4.py`

➤ `docdist5.py`

➤ `docdist6.py`

➤ `docdist7.py`

➤ `docdist8.py`



# 1.7 一个算法例子：文本相似度分析



## Python Profiling

```
import cProfile
cProfile.run("main()")
```

File t2.bobsey.txt : 6667 lines, 49785 words, 3354 distinct words

File t3.lewis.txt : 15996 lines, 182355 words, 8530 distinct words

The distance between the documents is: 0.574160 (radians)

3309360 function calls in 33.453 seconds

Ordered by: standard name

| ncalls | totttime | percall | cumtime | percall | filename:lineno(function)                  |
|--------|----------|---------|---------|---------|--|
| 1      | 0.001    | 0.001   | 33.453  | 33.453  | <string>:1(<module>)                       |
| 2      | 0.000    | 0.000   | 0.000   | 0.000   | _bootlocale.py:11(getpreferredencoding)    |
| 2      | 0.000    | 0.000   | 28.484  | 14.242  | docdist1.py:100(word_frequencies_for_file) |
| 3      | 4.964    | 1.655   | 4.964   | 1.655   | docdist1.py:116(inner_product)             |
| 1      | 0.000    | 0.000   | 4.964   | 4.964   | docdist1.py:131(vector_angle)              |
| 1      | 0.004    | 0.004   | 33.452  | 33.452  | docdist1.py:141(main)                      |
| 2      | 0.000    | 0.000   | 0.012   | 0.006   | docdist1.py:37(read_file)                  |
| 2      | 17.998   | 8.999   | 18.803  | 9.401   | docdist1.py:49(get_words_from_line_list)   |
| 22663  | 0.529    | 0.000   | 0.805   | 0.000   | docdist1.py:61(get_words_from_string)      |
| 2      | 9.666    | 4.833   | 9.668   | 4.834   | docdist1.py:86(count_frequency)            |
| 41     | 0.000    | 0.000   | 0.001   | 0.000   | iostream.py:195(schedule)                  |
| 38     | 0.000    | 0.000   | 0.000   | 0.000   | iostream.py:307(_is_master_process)        |
| 38     | 0.000    | 0.000   | 0.000   | 0.000   | iostream.py:320(_schedule_flush)           |
| 38     | 0.000    | 0.000   | 0.001   | 0.000   | iostream.py:382(write)                     |
| 41     | 0.000    | 0.000   | 0.000   | 0.000   | iostream.py:93(_event_pipe)                |
| 41     | 0.000    | 0.000   | 0.000   | 0.000   | socket.py:334(send)                        |
| 41     | 0.000    | 0.000   | 0.000   | 0.000   | threading.py:1038(_wait_for_tstate_lock)   |
| 41     | 0.000    | 0.000   | 0.000   | 0.000   | threading.py:1080(is_alive)                |



# 算法是什么？总结

- 从哲学角度看：算法是解决一个问题的抽象行为序列。
- 从技术层面上看：算法是一个计算过程，它接受一些输入，并产生某些输出。
- 从抽象层次上看：算法是一个将输入转化为输出的计算步骤序列。
- 从宏观层面上看：算法是解决一个精确定义的计算问题的工具。